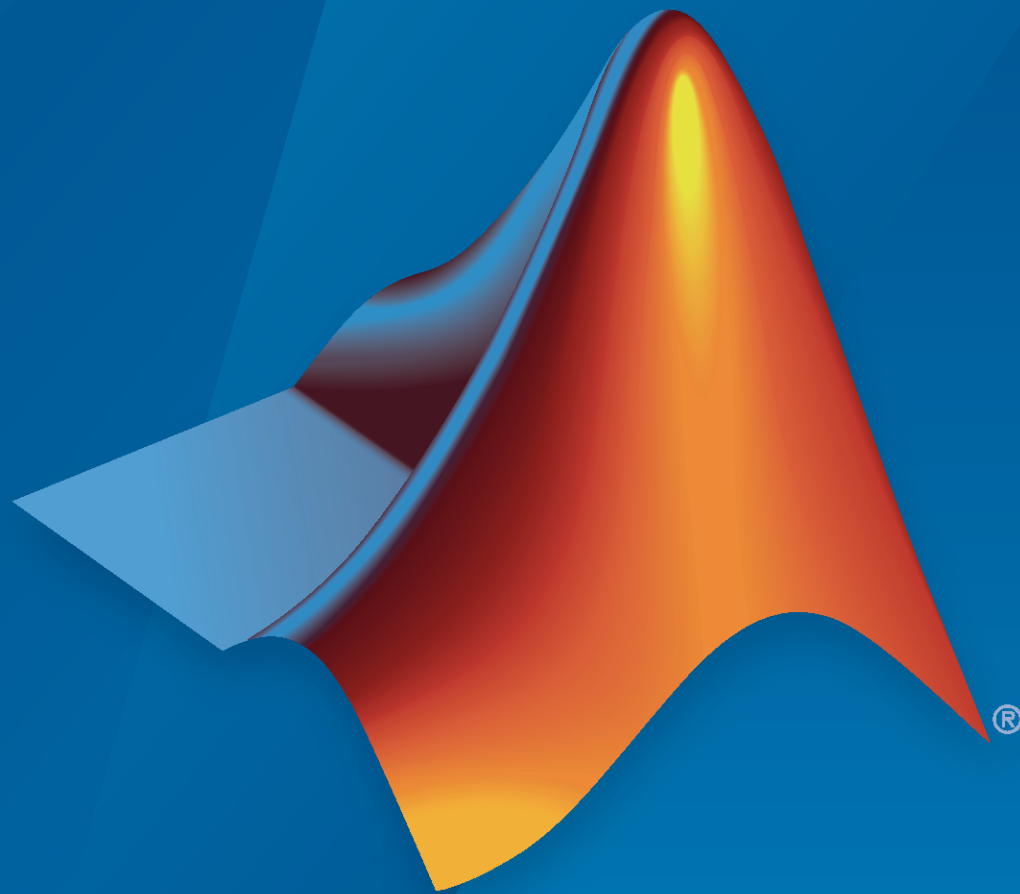


Simulink[®] Test[™]

User's Guide



MATLAB[®]&SIMULINK[®]

R2023a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Simulink® Test™ User's Guide

© COPYRIGHT 2015–2023 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 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 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)
March 2019	Online Only	Revised for Version 3.0 (Release 2019a)
September 2019	Online Only	Revised for Version 3.1 (Release 2019b)
March 2020	Online Only	Revised for Version 3.2 (Release 2020a)
September 2020	Online Only	Revised for Version 3.3 (Release 2020b)
March 2021	Online Only	Revised for Version 3.4 (Release 2021a)
September 2021	Online Only	Revised for Version 3.5 (Release 2021b)
March 2022	Online Only	Revised for Version 3.6 (Release 2022a)
September 2022	Online Only	Revised for Version 3.7 (Release 2022b)
March 2023	Online Only	Revised for Version 3.8 (Release 2023a)

1	Test Strategies
	<hr/>
	Link to Requirements 1-2
	Requirements Traceability Considerations 1-2
	Establish Requirements Traceability for Testing 1-3
	Requirements-Based Testing for Model Development 1-7

2	Test Harness
	<hr/>
	Test Harness and Model Relationship 2-2
	Harness-Model Relationship for a Model Component 2-3
	Harness-Model Relationship for a Top-Level Model 2-4
	Resolving Parameters 2-5
	Test Harness Considerations 2-5
	Test Harness Construction for Specific Model Elements 2-8
	Signal Conversion 2-8
	Function Calls 2-8
	Physical Signal Connections 2-9
	Bus Signals 2-9
	String Signals 2-9
	Non-Graphical Connections 2-10
	Export Function Models 2-10
	Execution Semantics 2-11
	Sample Time Specification 2-11
	Create or Import Test Harnesses and Select Properties 2-13
	Create a Test Harness For a Top Level Model 2-13
	Create Test Harnesses for Model Components 2-13
	Import a Test Harness 2-14
	Preview and Open Test Harnesses 2-14
	Change Test Harness Properties 2-14
	Considerations for Selecting Test Harness Properties 2-15
	Create Test Harness Dialog Box Properties 2-15
	Import Test Harness Dialog Box Properties 2-21
	Customize Test Harness Creation Default Property Values 2-22
	Refine, Test, and Debug a Subsystem 2-24
	Manage Test Harnesses 2-31
	Internal and External Test Harnesses 2-31

Manage External Test Harnesses	2-31
Convert Between Internal and External Test Harnesses	2-32
Preview and Open Test Harnesses	2-33
Model and Test Harness Locking	2-34
Find Test Cases Associated with a Test Harness	2-34
Export Test Harnesses to Standalone Models	2-35
Move and Clone Test Harnesses	2-35
Clone and Export a Test Harness to a Separate Model	2-37
Delete Test Harnesses Programmatically	2-39
Export Test Harness to Previous Version	2-40
Customize Test Harnesses	2-42
Callback Function Definition and Harness Information	2-42
Display Harness Information struct Contents	2-44
Share Data Between Callbacks	2-44
Customize a Test Harness to Create Mixed Source Types	2-45
Test Harness Callback Example	2-46
Create Test Harnesses from Standalone Models	2-49
Test Harness Import Workflow	2-49
Component Compatibility for Test Harness Import	2-50
Import a Standalone Model as a Test Harness	2-51
Synchronize Changes Between Test Harness and Model	2-54
Set Synchronization for a New Test Harness	2-54
Change Synchronization of an Existing Test Harness	2-57
Synchronize Configuration Set and Model Workspace Data	2-57
Check for Unsynchronized Component Differences	2-57
Rebuild a Test Harness	2-58
Push Changes from Test Harness to Model	2-58
Check Component and Push Parameter to Main Model	2-58
Test Library Blocks	2-62
Library Testing Workflow	2-62
Library and Linked Subsystem Test Harnesses	2-62
Edit Library Block from a Test Harness	2-63
Testing a Library and a Linked Block	2-63
SIL Testing a Reusable Library Subsystem	2-68

Test Sequences and Assessments

3

Test Sequence Basics	3-2
Test Sequence Hierarchy	3-2
Test Sequence Scenarios	3-2
Transition Types	3-2
Create a Basic Test Sequence	3-4
Create Basic Test Assessments	3-5
Use Stateflow Chart for Test Harness Inputs and Scheduling	3-8
Use a Stateflow Chart for Test Harness Scheduling	3-8
Use a Stateflow Chart as a Test Harness Source	3-9

Stateflow Chart as Test Harness Scheduler and Source	3-10
Assess Simulation and Compare Output Data	3-14
Overview	3-14
Compare Simulation Data to Baseline Data or Another Simulation	3-15
Post-Process Results With a Custom Script	3-15
Run-Time Assessments	3-15
Logical and Temporal Assessments	3-17
Assess Model Simulation Using verify Statements	3-18
Activate verify Statements in the Test Assessment Block	3-18
Author verify Statements	3-21
Verify Multiple Conditions at a Time	3-23
Assess a Model by Using When Decomposition	3-25
Test Sequence Editor	3-30
Define Test Sequences	3-30
Manage Test Steps	3-30
Manage Input, Output, and Data Objects	3-32
Find and Replace	3-33
Automatic Syntax Correction	3-34
Output and View Active Step Data	3-34
Transitions, Temporal Operators, and Messages in Test Sequence Blocks	
.....	3-37
Transition Between Steps Using Temporal or Signal Conditions	3-37
Temporal Operators	3-37
Transition Operators	3-38
Use Messages in Test Sequences	3-39
Generate Test Signals	3-44
Signal Generation Functions	3-44
Sinusoidal and Random Number Functions in Test Sequences	3-46
Using an External Function in a Test Sequence Block	3-49
Programmatically Create a Test Sequence	3-52
Programmatically Create and Run Test Sequence Scenarios	3-56
Use Test Sequence Scenarios in the Test Sequence Editor and Test	
Manager	3-59
Scenario Parameter Section	3-67
Test Sequence and Assessment Syntax	3-68
Assessment Statements	3-68
Temporal Operators	3-69
Transition Operators	3-70
Signal Generation Functions	3-71
Logical Operators	3-73
Relational Operators	3-74

Debug a Test Sequence	3-75
View Test Step Execution During Simulation	3-75
Set Breakpoints to Enable Debugging	3-75
View Data Values During Simulation	3-76
Step Through Simulation	3-76
Test Downshift Points of a Transmission Controller	3-78
Examine Model Verification Results by Using Simulation Data Inspector	3-83
Fix Requirements-Based Testing Issues	3-87
Assess Temporal Logic by Using Temporal Assessments	3-93
Create a Temporal Assessment	3-93
Define Temporal Assessment Conditions	3-94
Evaluate the SUT	3-96
Link Temporal Assessments to Requirements	3-97
Test Traffic Light Control by Using Logical and Temporal Assessments	3-99
Logical and Temporal Assessment Syntax	3-107
Bounds Check Assessments	3-107
Trigger-Response Assessments	3-107
Custom Assessments	3-109
Logical and Temporal Assessment Conditions	3-109
Define Variables in the Assessment Callback Section	3-110

Observers

4

Access Model Data Wirelessly by Using Observers	4-2
Observer Reference Block	4-3
Connect Signals or Other Model Data Using an Observer Port Block	4-4
Trace Observed Items to Model Signals and Objects	4-6
Simulate a System Model with an Observer Reference Block	4-6
Verify Heat Pump Temperature by Using Observers	4-7
Convert Verification Subsystem to an Observer Reference	4-9
Observer Considerations and Limitations	4-12
Observe Messages	4-13
Message Bus Elements	4-13
Add a Message Observer	4-13
Observe a Message Signal	4-14
Observe Conditional Subsystem Signals	4-17
Add an Observer for a Conditional Subsystem	4-17
Observe a Signal in a Conditional Subsystem	4-18
Limitations	4-20

Observe Internal Variables of an FMU	4-21
Observe Internal Variables to Tune PID Controller Inside an FMU	4-21

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

5

SIL Verification for a Subsystem	5-2
Create a SIL Verification Harness for a Controller	5-2
Configure and Simulate a SIL Verification Harness	5-4
Compare the SIL Block and Model Controller Outputs	5-4
Use SIL/PIL to Verify Generated Code from an Earlier Release	5-6
Reuse Generated Code	5-6
SIL Verification of a Subsystem using Code Generated from an Earlier Release	5-6
Code Generation Verification Workflow with Simulink Test	5-14
Import Test Cases for Equivalence Testing	5-19
Settings for Test Case Simulations	5-19
Top-Level Model	5-19
Model Block in SIL/PIL Mode	5-20
Model Block or Reusable Library Subsystem in a Test Harness	5-21
Back-to-Back Testing a Model Using the SIL/PIL Manager App	5-22
Test Integrated Code	5-27
Test Integrated C Code	5-27
Testing Code in an S-Function Block	5-27

Test Manager Test Cases

6

Manage Test File Dependencies	6-3
Package a Test File Using Projects	6-3
Find Test File Dependencies and Impact	6-4
Share a Test File with Dependencies	6-6
Compare Model Output to Baseline Data	6-7
Create the Test Case	6-7
Run the Test Case and View Results	6-7
Creating Baseline Tests	6-10
Batch Equivalence Testing of Multiple Components	6-13
Test a Simulation for Run-Time Errors	6-16
Configure the Model	6-16
Create the Test Case	6-16
Run the Test Case	6-17

View the Error	6-17
Automatically Create a Set of Test Cases	6-19
Creating Test Cases from Model Elements	6-19
Generating Test Cases from a Model	6-19
Generate Tests and Test Harnesses for a Model or Components	6-24
Open the Create Test for Component Wizard	6-24
Select Model or Component to Test	6-25
Set Up Test Inputs	6-26
Test Method	6-27
Save Test Data	6-28
Generate the Test Harness and Test Case	6-29
Override Model Parameters in a Test Case	6-31
Test Two Simulations for Equivalence	6-35
Create and Run a Back-to-Back Test	6-41
Run the Back-to-Back Test	6-45
View the Back-to-Back Test Results	6-45
Perform Back-to-Back (MIL/SIL) Equivalence Test for an Atomic Subsystem	6-47
Testing AUTOSAR Compositions	6-55
Automate Testing for Highway Lane Following	6-60
Synchronize Tests	6-71
Use External Excel or MAT-File Data in Test Cases	6-72
Data Mapping	6-72
Create a Test Case from an Excel Spreadsheet	6-73
Import an Excel Spreadsheet into an Existing Test Case	6-74
Add Multiple Microsoft Excel Spreadsheets as Input to a Test Case	6-75
Include Microsoft Excel Test Data in Test Results	6-75
Importing Test Data from Microsoft Excel	6-75
Add a MAT-File as an External Input	6-78
Create Data Files for Test Case Input	6-80
Generate an Excel Template	6-80
Format Test Case Data in Excel	6-83
Create a MAT-File for Input Data	6-83
Capture Simulation Data in a Test Case	6-85
Add Logged Signals When Creating a Test Harness	6-85
Add Logged Signals in the Test Manager	6-85
Capture Data from Local and Global Data Stores	6-87
Log Leaf Signals of a Bus	6-88
Use Triggers to Start and Stop Signal Logging	6-89
Run Tests in Multiple Releases of MATLAB	6-94
Considerations for Testing in Multiple Releases	6-94
Add Releases Using Test Manager Preferences	6-95

Run Baseline Tests in Multiple Releases	6-95
Run Equivalence Tests in Multiple Releases	6-96
Run Simulation Tests in Multiple Releases	6-97
Assess Temporal Logic in Multiple Releases	6-98
Collect Coverage in Multiple-Release Tests	6-99
Examine Test Failures and Modify Baselines	6-102
Examine Test Failure Signals and Update Baseline Test	6-102
Manually Update Signal Data in a Baseline	6-104
Create and Run Test Cases with Scripts	6-107
Create and Run a Baseline Test Case	6-107
Create and Run an Equivalence Test Case	6-108
Run a Test Case and Collect Coverage	6-109
Create and Run Test Case Iterations	6-109
Test Models Using MATLAB-Based Simulink Tests	6-111
Classes and Methods	6-111
Creating a Baseline MATLAB-Based Simulink Test	6-112
Linking to Requirements from a MATLAB-Based Simulink Test File	6-114
Limitations of MATLAB- Based Tests	6-115
Using MATLAB-Based Simulink Tests in the Test Manager	6-116
Collect Coverage Using MATLAB-Based Simulink Tests	6-120
Test Iterations	6-125
Create Table Iterations	6-125
Create Scripted Iterations	6-128
Sweep Through a Set of Parameters	6-131
Capture Baseline Data from Iterations	6-133
Collect Coverage in Tests	6-135
Set Up Coverage Collection Using the Test Manager	6-135
View Coverage Results in the Test Manager	6-137
Add Tests for Missing Coverage	6-139
Coverage Filtering Using the Test Manager	6-140
Test Coverage for Requirements-Based Testing	6-142
Increase Test Coverage for a Model	6-147
Run Tests Using Parallel Execution	6-151
When Do Tests Benefit from Using Parallel Execution?	6-151
Use Parallel Execution	6-151
Set Signal Tolerances	6-153
Modify Criteria Tolerances	6-153
Change Leading Tolerance in a Baseline Comparison Test	6-153
Specify Test Properties in the Test Manager	6-158
Test Case, Test Suite, and Test File Sections Summary	6-158
Create Test Case from External File	6-160
Tags	6-160

Description	6-160
Requirements	6-160
System Under Test	6-160
Simulation 1 and Simulation 2	6-162
Parameter Overrides	6-162
Callbacks	6-163
Inputs	6-164
Simulation Outputs	6-165
Configuration Settings Overrides	6-166
Baseline Criteria	6-167
Equivalence Criteria	6-168
Iterations	6-168
Logical and Temporal Assessments	6-169
Custom Criteria	6-170
Coverage Settings	6-170
Test File Options	6-172
Test File Content	6-172
Preferences	6-173
Increase Coverage by Generating Test Inputs	6-174
Overall Workflow	6-174
Generate Test Cases Using Simulink Design Verifier	6-175
Process Test Results with Custom Scripts	6-179
MATLAB Testing Framework	6-179
Define a Custom Criteria Script	6-179
Reuse Custom Criteria and Debug Using Breakpoints	6-180
Custom Criteria Programmatic Interface Example	6-182
Assess the Damping Ratio of a Flutter Suppression System	6-185
Create, Store, and Open MATLAB Figures	6-188
Create a Custom Figure for a Test Case	6-188
Include Figures in a Report	6-189
Test Models Using MATLAB Unit Test	6-191
Overall Workflow	6-191
Considerations	6-191
Comparison of Test Nomenclature	6-191
Basic Workflow Using MATLAB® Unit Test	6-192
Output Results for Continuous Integration Systems	6-194
Test a Model for Continuous Integration Systems	6-194
Model Coverage Results for Continuous Integration	6-196
Parametric Sweep for a Simscape Thermal Model	6-198
Projector Controller Testing Using verify and Real-Time Tests	6-204
Test Execution Order	6-209
Single Test Case on a Single Model	6-209
Multiple Test Cases on Multiple Models	6-209
Multiple Test Cases in a Single Test Suite on a Single Model	6-210
Multiple Test Cases in Multiple Test Suites on a Single Model	6-211

Test Case with Parameter Overrides	6-211
Filter Test Execution, Results, and Coverage	6-213
Add Tags	6-213
Filter Tests and Results	6-213
Run Filtered Tests	6-213
Filter Coverage	6-213

Test Manager Results and Reports

7

View Test Case Results	7-2
View Results Summary	7-2
Visualize Test Case Simulation Output and Criteria	7-3
Debugging Test Failures Using Model Slicer	7-7
Export Test Results	7-16
Generate Test Results Reports	7-17
Create a Test Results Report	7-17
Save Reporting Options with a Test File	7-17
Generate Reports Using Templates	7-17
Generating a Test Results Report	7-20
Customize Test Results Reports	7-21
Inherit the Report Class	7-21
Method Hierarchy	7-21
Modify the Class	7-22
Generate a Report Using the Custom Class	7-24
Append Code to a Test Report	7-25
Results Sections	7-27
Summary	7-28
Test Requirements	7-28
Iteration Settings	7-28
Errors	7-28
Logs	7-28
Description	7-28
Parameter Overrides	7-28
Coverage Results	7-29
Aggregated Coverage Results	7-29
Scope coverage results to linked requirements	7-29
Add Tests for Missing Coverage	7-29
Applied Coverage Filters	7-29
Generate Test Specification Reports	7-30
Customize Test Specification Reports	7-34
Remove Content or Change Report Formatting and Section Ordering ...	7-34

Add Content to a Test Specification Report	7-37
Debugging Equivalence Test Failures Using Model Slicer	7-41

Real-Time Testing

8

Test Models in Real Time	8-2
Overall Workflow	8-2
Real-Time Testing Considerations	8-3
Complete Basic Model Testing	8-3
Set up the Target Computer	8-3
Configure the Model or Test Harness	8-3
Add Test Cases for Real-Time Testing	8-4
Assess Real-Time Execution Using verify Statements	8-8
Reuse Desktop Test Cases for Real-Time Testing	8-9
Convert Desktop Test Cases to Real-Time	8-9
Use External Data for Real-Time Tests	8-9
Reuse Desktop Test Case for Real-Time Testing	8-10
Install and Set Up the Simulink Test Support Package for ASAM XIL	
Standard	8-13
Install the Support Package	8-13
Set Up the Test Bench	8-13
Configure and Build Your Model	8-13
Real-Time Testing with the Simulink Test Support Package for ASAM XIL	
Standard	8-15
Simulink Test ASAM XIL Standard Workflow	8-15
Limitations	8-18
Troubleshooting	8-18
Create Tests Using the Simulink Test Support Package for ASAM XIL	
Standard	8-19
ASAM XIL Test With Data Acquisition Triggering	8-19
ASAM XIL Test Without Data Acquisition Triggering	8-21

Testing Custom C/C++ Code

9

Importing and Testing Custom C/C++ Code	9-2
Import Code Using the Wizard or the API	9-2
Code Importer Generated Artifacts	9-2
Limitations and Workarounds	9-3
Import Custom Code for Unit Testing Using API Commands	9-5
Conduct Unit Testing on Imported Custom Code by Using the Wizard .	9-11

- Test Model Against Requirements and Report Results** **10-2**
 - Requirements - Test Traceability Overview **10-2**
 - Display the Requirements **10-2**
 - Link Requirements to Tests **10-3**
 - Run the Test **10-4**
 - Report the Results **10-5**

- Analyze Models for Standards Compliance and Design Errors** **10-7**
 - Standards and Analysis Overview **10-7**
 - Check Model for Style Guideline Violations and Design Errors **10-7**

- Perform Functional Testing and Analyze Test Coverage** **10-9**
 - Incrementally Increase Test Coverage Using Test Case Generation **10-9**

- Analyze Code and Test Software-in-the-Loop** **10-12**
 - Code Analysis and Testing Software-in-the-Loop Overview **10-12**
 - Analyze Code for Defects, Metrics, and MISRA C:2012 **10-12**
 - Test Code Against Model Using Software-in-the-Loop Testing **10-17**

Test Strategies

- “Link to Requirements” on page 1-2
- “Requirements-Based Testing for Model Development” on page 1-7

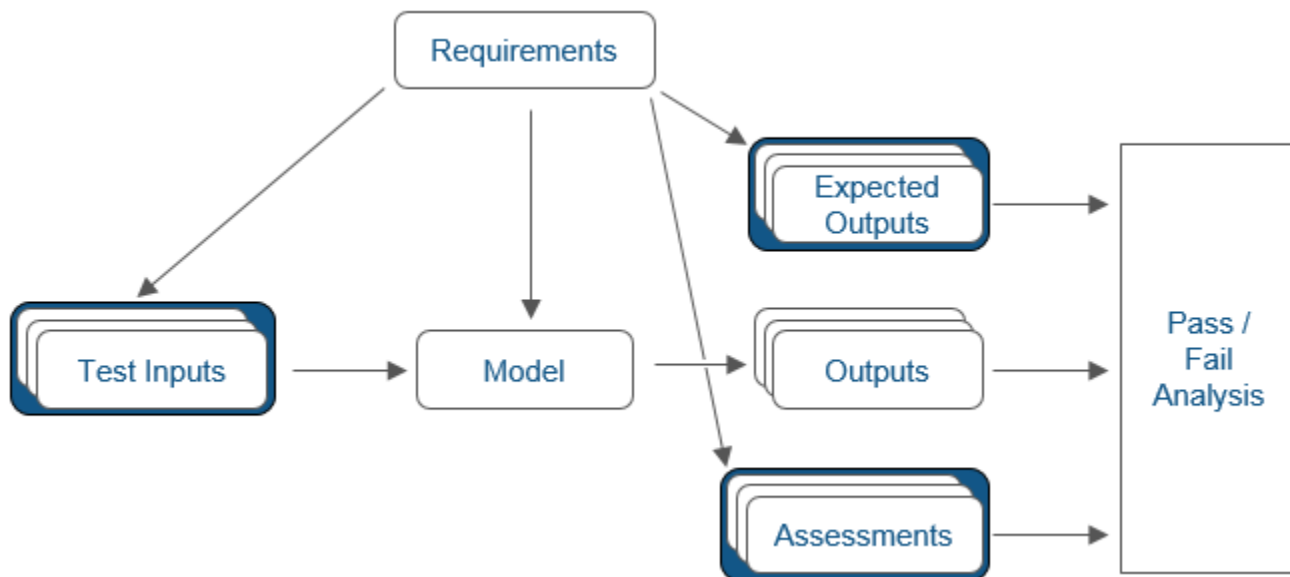
Link to Requirements

In this section...

“Requirements Traceability Considerations” on page 1-2

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

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



Requirements Traceability Considerations

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


- Some blocks and subsystems are recreated during test harness rebuild operations. Requirements linking is not supported for these blocks and subsystems in a test harness:
 - Conversion subsystems between the component under test and the sources or sinks
 - Test Sequence blocks that schedule function calls
 - Blocks that drive control input signals to the component under test
 - Blocks that drive Goto or From blocks that pass component under test signals
 - Data Store Read and Data Store Write blocks
- If you use external requirements storage, performing the following operations requires reestablishing requirements links to model objects inside test harnesses:
 - Cut/paste or copy/paste a subsystem with a test harness
 - Clone a test harness
 - Move a test harness from a linked block to the library block

Establish Requirements Traceability for Testing

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


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, on the **Apps** tab, under Model Verification, Validation, and Test, click **Requirements Manager**. In the **Requirements** tab, click **Highlight Links**  Highlight Links.

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 remove the highlighting from 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 In the left pane, in the **Test Browser** tab, select the test case.
- 3 In Simulink in the **Apps** tab, click **Requirements Manager**.
- 4 To link a test case to a:
 - Simulink block, right-click the block and select **Requirements > Link to Current Test Case** from the context menu.
 - Test step, double-click the test sequence block in the test harness to open the Test Sequence Editor. Right-click the test step and select **Requirements > Link to Current Test Case** from the context menu.


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 Requirements Toolbox.

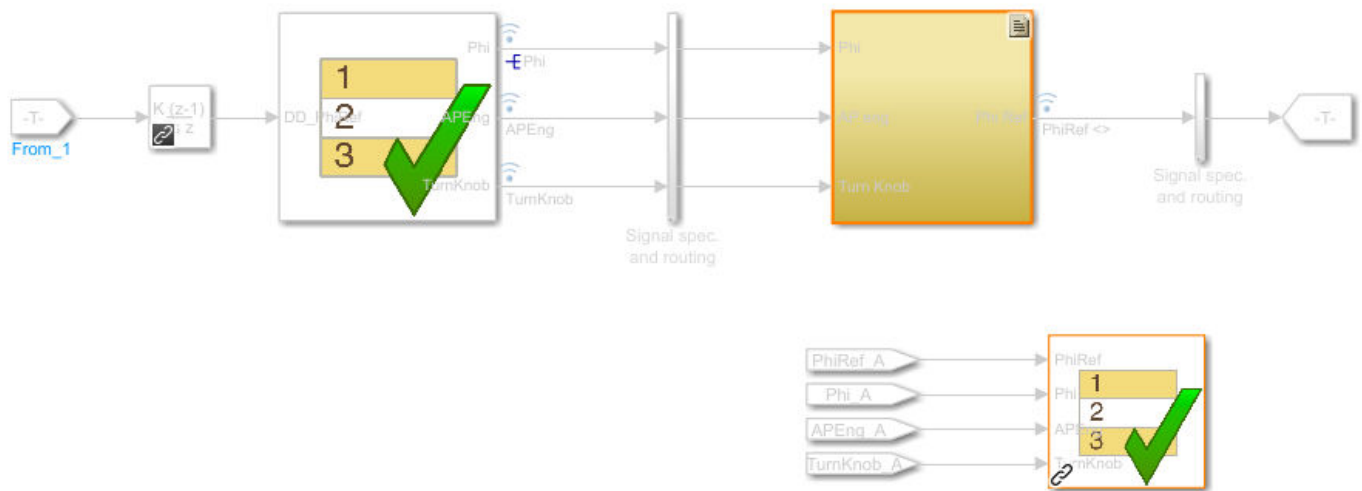
- 1 Open the model, the test file, and the harness.

```
openExample("simulinktest/ModelCoverageMATLABUnitExample", ...
    supportingFile="RollAutopilotMdlRef.slx")
```

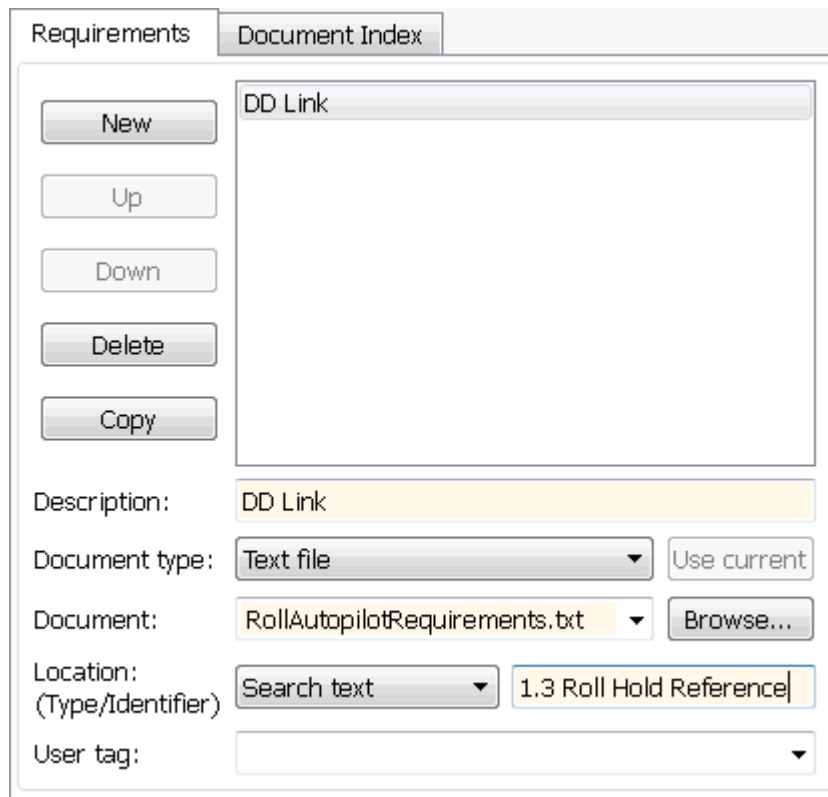
```
openExample("simulinktest/ModelCoverageMATLABUnitExample", ...
    supportingFile="AutopilotTestFile.mldatx")
sltest.harness.open("RollAutopilotMdlRef/Roll Reference",...
    "RollReference_Requirement1_3")
```

- 2 In the test harness, on the **Apps** tab, under Model Verification, Validation, and Test, click **Requirements Manager**. In the **Requirements** tab, click **Highlight Links**  Highlight Links .

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 (legacy)
 - 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 In the test harness, open the Test Sequence block. Add a requirements link that links the InitializeTest step to the test case.
- a In the Test Manager, in the left pane, in the **Test Browser** tab, select Requirement 1.3 Test.

- b In the test harness, double-click the test sequence block to open the Test Sequence Editor. Right-click the `InitializeTest` step and select **Requirements > Link to Current Test Case** from the context menu.

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

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

Requirements-Based Testing for Model Development

Test an autopilot subsystem against a requirement.

This example demonstrates testing a subsystem against a requirement, using the test manager, test harness, Test Sequence block, and Test Assessment block. The requirements document links to the test case and test harness, and `verify` statements assess the component under test.

As you build your model, you can add test cases to verify the model against requirements. Subsequent users can run the same test cases, then add test cases to accomplish further verification goals such as achieving 100% coverage or verifying generated code.

This example tests the Roll Reference subsystem against a requirement using three scenarios. A Test Sequence block provides inputs, and a Test Assessment block evaluates the component. The Roll Reference subsystem is one component of an autopilot control system. Roll Reference controls the reference angle of the aircraft's roll control system. The subsystem fails one assessment, prompting a design change that limits the subsystem output at high input angles.

Paths and Example Files

Enter the following to store paths and file names for the example:

```
topModel = 'TestAndVerificationAutopilotExample';  
rollModel = 'RollAutopilotMdlRef';  
testHarness = 'RollReference_Requirement1_3';  
testFile = 'AutopilotTestFile.mldatx';  
reqDoc = 'RollAutopilotRequirements.txt';
```

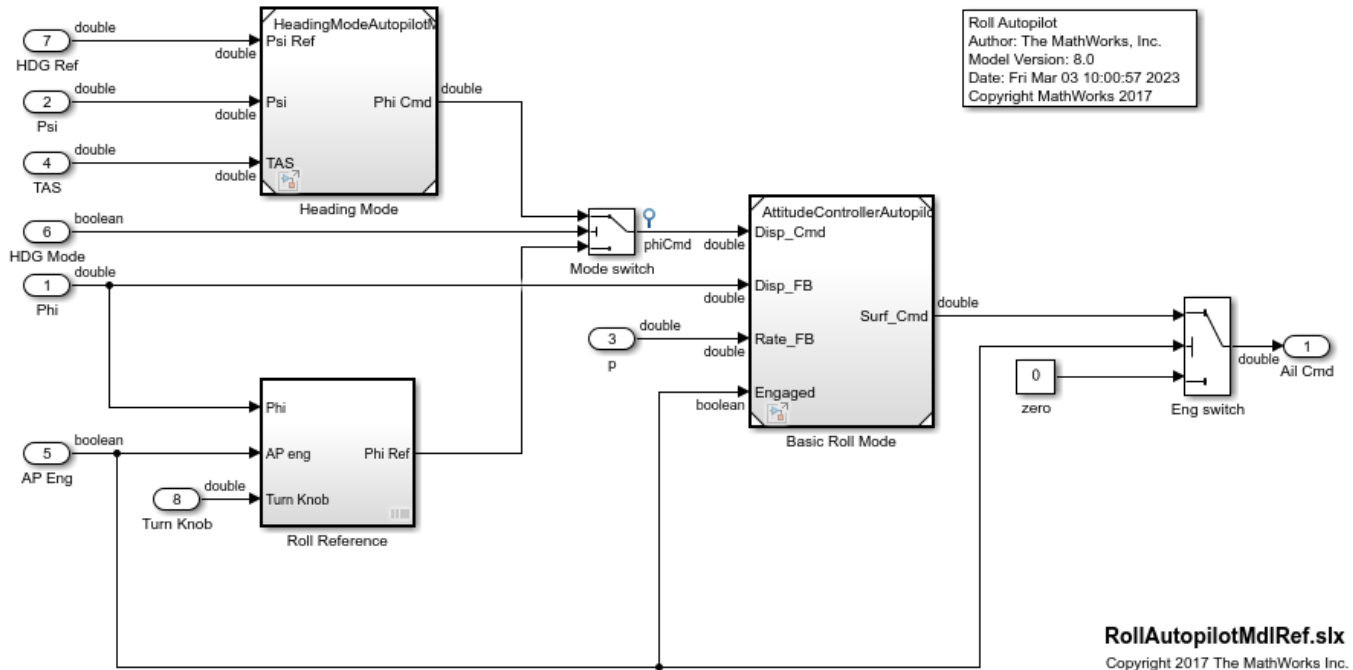
Open the Test File and Model

Open the RollAutopilotMdlRef model. The full control system TestAndVerificationAutopilotExample references this model.

```
open_system(rollModel)
```

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).



Open the test file in the Test Manager.

```
tf = sltest.testmanager.load(testFile);
sltest.testmanager.view;
```

Open the requirements document. In the test browser, expand **AutopilotTestFile** and **Basic Design Test Cases** in the tree, and click **Requirement 1.3 test**. In the Requirement 1.3 test tab, expand **Requirements**. Double-click on any of the requirements links to open the Requirements Editor, where you can review the requirements.

Requirement 1.3 Test

[AutopilotTestFile](#) » [Basic Design Test Cases](#) » [Requirement 1.3 Test](#)

Simulation Test

Create Test Case from External File

▶ TAGS

▶ DESCRIPTION

▼ REQUIREMENTS*

[1.3.1.1 phiref = 0 if phi < 6](#)

[1.3.1.2 phiref = 30 if phi > 30](#)

[1.3.1.3 phiref = tk if tk >= 3](#)

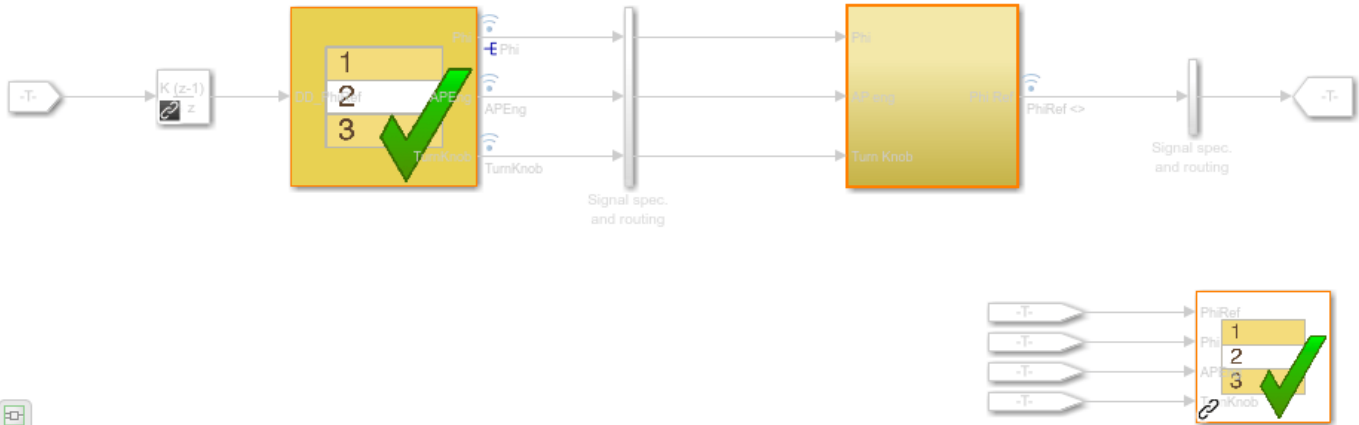
In the document, requirement 1.3.1 states: When roll hold mode becomes the active mode, the roll hold reference shall be set to the actual roll angle of the aircraft, except under the conditions described in the child requirements.

- Child requirement 1.3.1.1 states: The roll hold reference shall be set to zero if the actual roll angle is less than 6 degrees, in either direction, at the time of roll hold engagement.
- Child requirement 1.3.1.2 states: The roll hold reference shall be set to 30 degrees in the same direction as the actual roll angle if the actual roll angle is greater than 30 degrees at the time of roll hold engagement.
- Child requirement 1.3.1.3 states: The roll reference shall be set to the cockpit turn knob command, up to a 30 degree limit, if the turn knob is commanding 3 degrees or more in either direction.

The test case creates three scenarios to test the normal conditions and exceptions in the requirement.

The requirements document traces to the test harness using URLs that map to the Test Sequence block and test steps. Open the test harness and highlight the component associated with reference requirement 1.3.

```
sltest.harness.open([rollModel '/Roll Reference'], testHarness)
rmi('highlightModel', 'RollReference_Requirement1_3')
```



The Test Sequence block, Test Assessment block, and component under test link to the requirements document. Highlight requirements links by selecting **Apps > Requirements Manager** and then, clicking Highlight Links in the test harness model. You can also highlight links in the Test Sequence Editor by clicking **Toggle requirements links highlighting** in the toolbar.

Test Sequence

Open the Test Sequence block.

```
open_system('RollReference_Requirement1_3/Test Sequence')
```

Step	Transition	Next Step
InitializeTest Phi = 0; APEng = false; TurnKnob = 0; % Initializes test sequence outputs	1. true	AttitudeLevels ▼
<input checked="" type="checkbox"/> AttitudeLevels TurnKnob = 0; EndTest = 0; % Tests correct PhiRef for several attitude levels Add step after - Add sub-step	1. EndTest == 1	TurnKnobLevels ▼
<input checked="" type="checkbox"/> APEngage_LowRoll % Tests low attitude	1. duration(DD_PhiRef == 0,sec) >= DurationLimit % transitions when the discrete derivative of PhiRef % is equal to 0 for a certain time limit. This means the % signal is not changing.	APEngage_MedRoll ▼
SetLowPhi Phi = 4; APEng = false;	1. true	EngageAP_Low ▼

The Test Sequence block creates test inputs for three scenarios:

In each test, the test sequence sets a signal level, then engages the autopilot. The test sequence checks that PhiRef is stable for a minimum time DurationLimit before it transitions to the next signal level. For the first two scenarios, the test sequence sets the EndTest local variable to 1, triggering the transition to the next scenario.

These scenarios check basic component function, but do not necessarily achieve objectives such as 100% coverage.

Test Assessments

Open the Test Assessment block.

```
open_system('RollReference_Requirement1_3/Test Assessment')
```

Step

GlobalAssessment

NormalRange when `(abs(Phi) >= 6 && Phi <= 30) && APEng == true && abs(TurnKnob) <= 0.001`
`verify(abs(PhiRef - Phi) < 0.001, 'Simulink:verify_normal', 'PhiRef must equal Phi for normal operation')`

BelowLowLimit when `abs(Phi) < 6 && APEng == true && abs(TurnKnob) <= 0.001`
`verify(abs(PhiRef - 0) < 0.001, 'Simulink:verify_low', 'PhiRef must equal 0 for low angle operation')`

ExceedPosLimit when `Phi > 30 && APEng == true && abs(TurnKnob) <= 0.001`
`verify(abs(PhiRef - 30) < 0.001, 'Simulink:verify_high_pos', 'PhiRef must equal 30 for high pos angle operation')`

ExceedNegLimit when `Phi < -30 && APEng == true && abs(TurnKnob) <= 0.001`
`verify(abs(PhiRef - (-30)) < 0.001, 'Simulink:verify_high_neg', 'PhiRef must equal -30 for high neg angle operation')`

The Test Assessment block evaluates Roll Reference. The assessment block is a library linked subsystem, which facilitates test assessment reuse between multiple test harnesses. The block contains `verify` statements covering:

- The requirement that $\text{PhiRef} = \text{Phi}$ when Phi operates inside the low and high limits.
- The requirement that $\text{PhiRef} = 0$ when $\text{Phi} < 6$ degrees.
- The requirement that $\text{PhiRef} = 30$ when $\text{Phi} > 30$ degrees.
- The requirement that when `TurnKnob` is engaged, $\text{PhiRef} = \text{TurnKnob}$ if $\text{TurnKnob} \geq 3$ degrees.

Verify the Subsystem

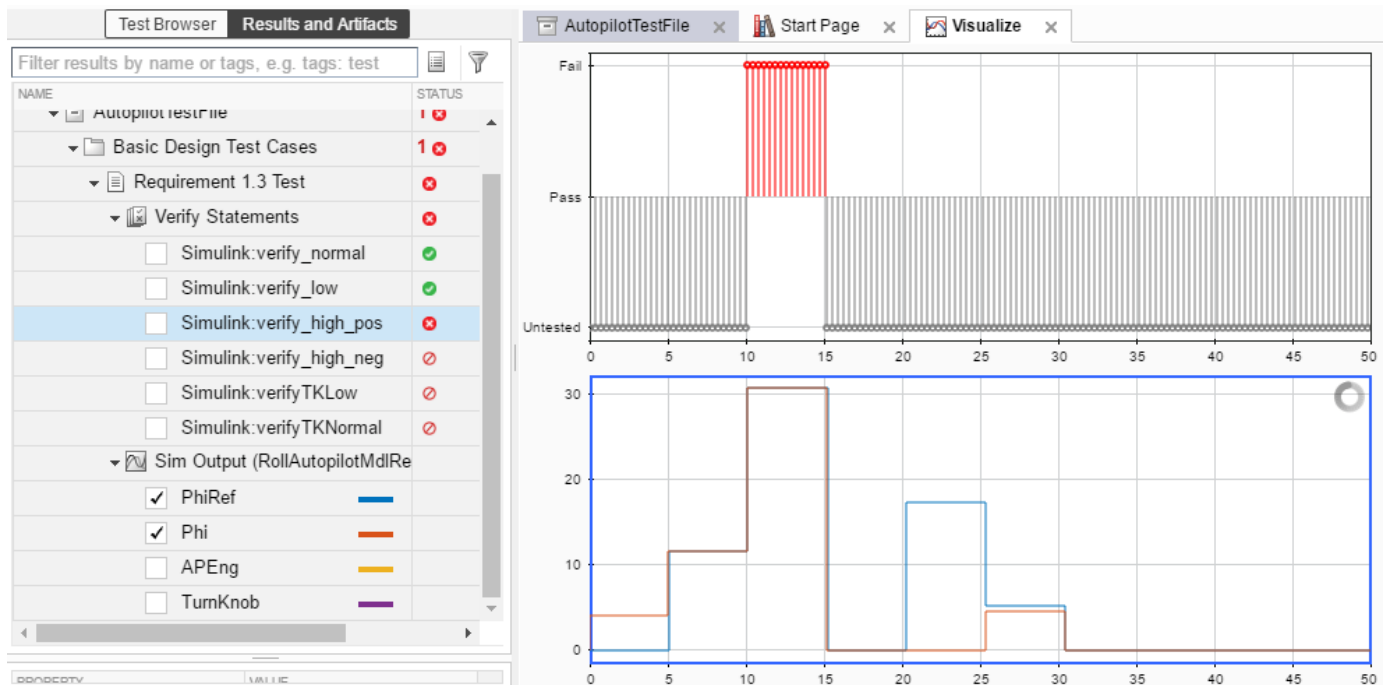
To run the test, in the Test Manager, right-click **Requirement 1.3 Test** in the Test Browser pane, and click **Run**.

The simulation returns `verify` statement results and simulation output in the Test Manager. The `verify_high_pos` statement fails.

- 1 Click **Results and Artifacts** in the test manager.
- 2 In the results tree, expand **Verify Statements**. Click **Simulink: verify_high_pos**. The trace shows when the statement fails.



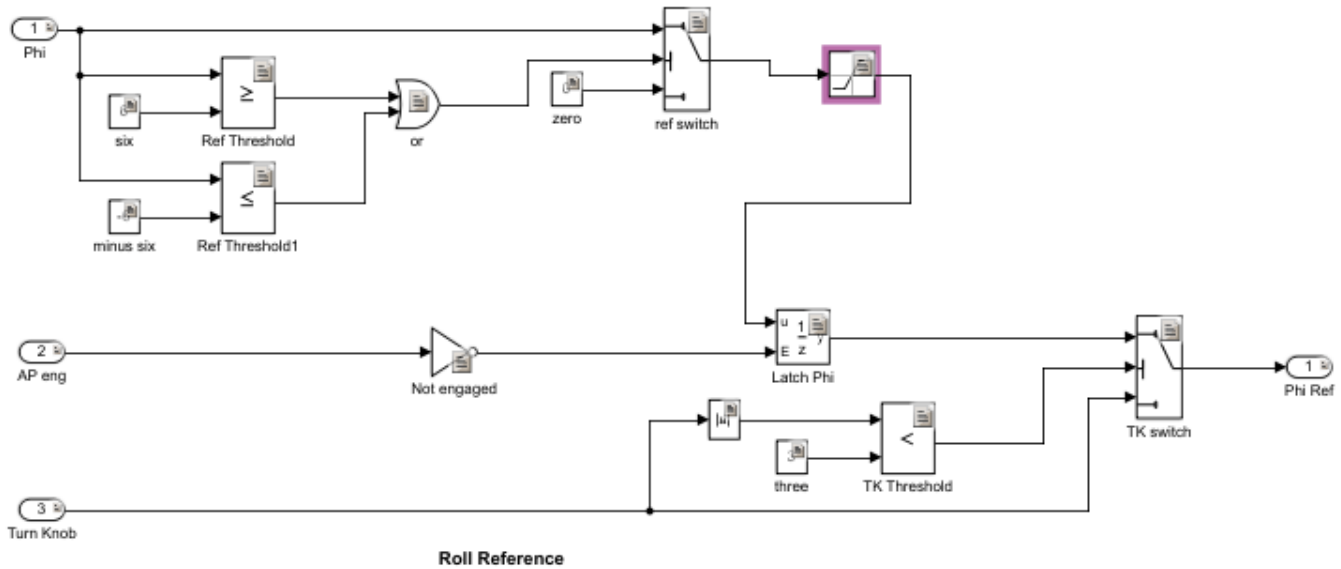
- 1 Click **Subplots** in the toolstrip and select two plots arranged vertically. Select the lower plot in the **Visualize** pane.
- 2 In the results tree, expand **Results, Requirement 1.3 Test**, and **Sim Output**.
- 3 Select **PhiRef** and **Phi**. The output traces align with the **verify** results in the above plot. Observe that **PhiRef** exceeds 30 degrees when **Phi** exceeds 30 degrees.



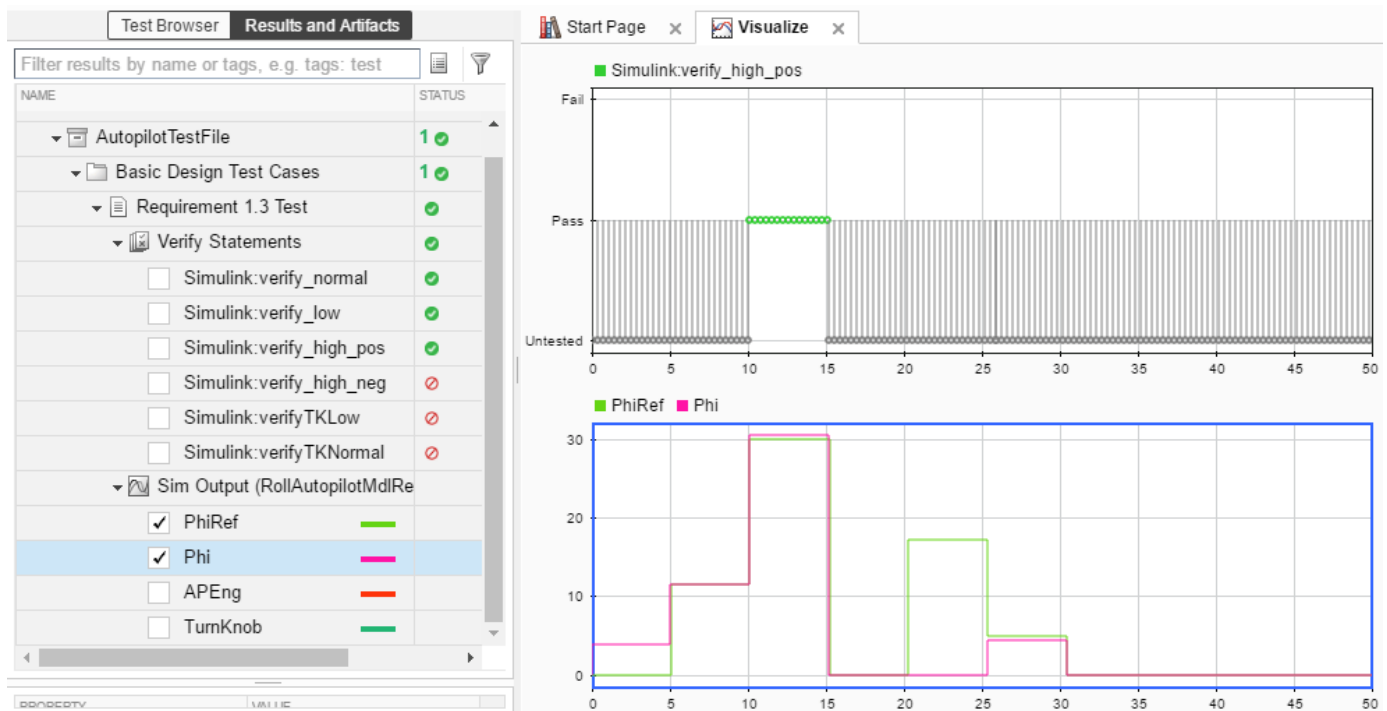
Update RollReference to limit the PhiRef signal.

- 1 Close the test harness.
- 2 Add a Saturation block to the model as shown.

- 3 Set the lower limit to -30 and the upper limit to 30.
- 4 Link the block to its requirement. From the Requirements browser, drag requirement 1.1.2 to the Saturation block. An icon appears on the block, and the requirement is highlighted.



Run the test again. The verify statement passes, and the output in the test manager shows that PhiRef does not exceed 30 degrees.



```
close_system(rollModel,0);
close_system(topModel,0);
```

```
close_system('RollRefAssessLib',0);
sltest.testmanager.clear;
sltest.testmanager.clearResults;
sltest.testmanager.close;
clear topModel reqDoc rollModel testHarness testFile harnessLink
```

Test Harness

- “Test Harness and Model Relationship” on page 2-2
- “Test Harness Construction for Specific Model Elements” on page 2-8
- “Create or Import Test Harnesses and Select Properties” on page 2-13
- “Refine, Test, and Debug a Subsystem” on page 2-24
- “Manage Test Harnesses” on page 2-31
- “Customize Test Harnesses” on page 2-42
- “Create Test Harnesses from Standalone Models” on page 2-49
- “Synchronize Changes Between Test Harness and Model” on page 2-54
- “Test Library Blocks” on page 2-62

Test Harness and Model Relationship

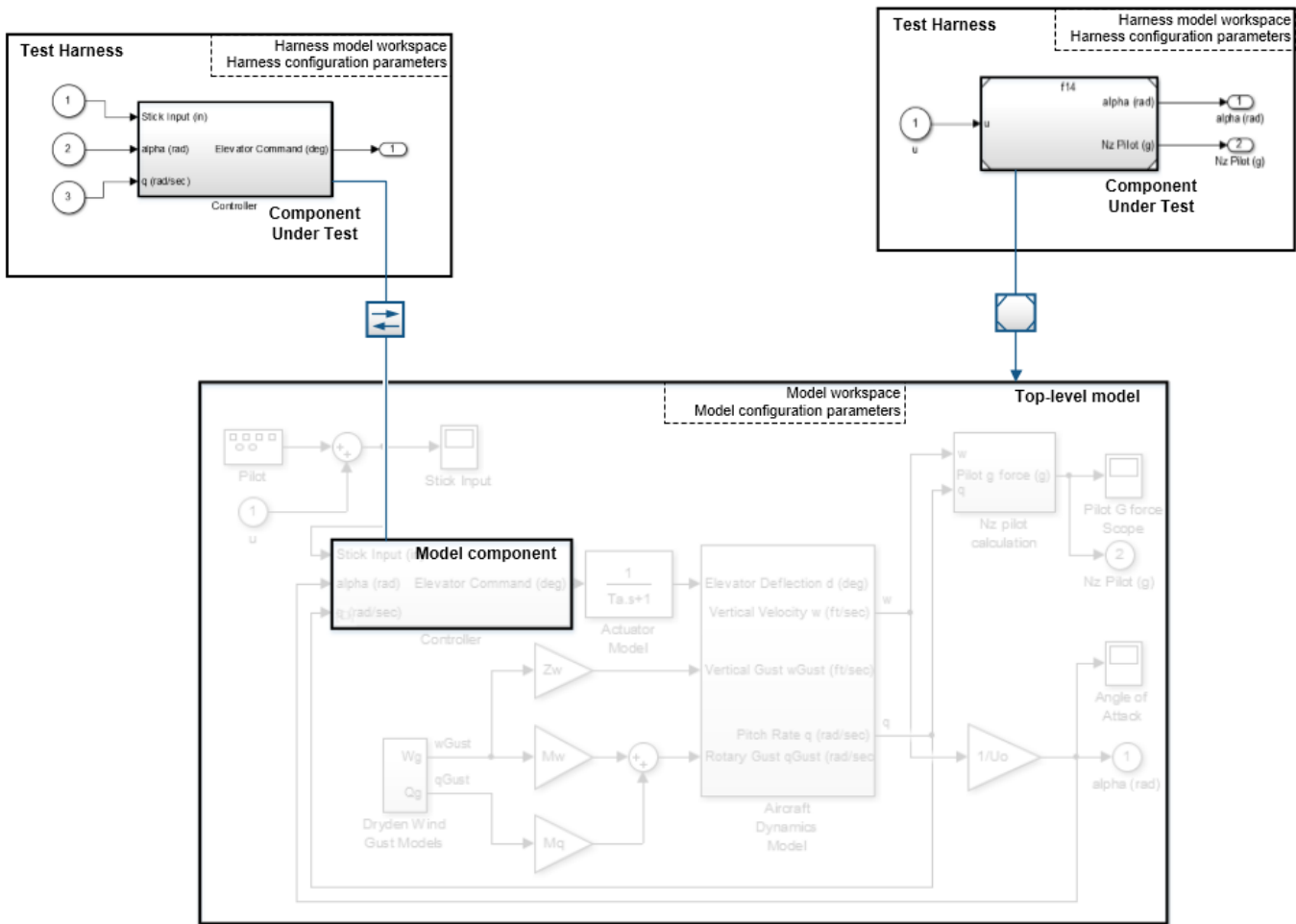
In this section...
“Harness-Model Relationship for a Model Component” on page 2-3
“Harness-Model Relationship for a Top-Level Model” on page 2-4
“Resolving Parameters” on page 2-5
“Test Harness Considerations” on page 2-5

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

When you create an external harness, a metadata XML file is also created. The XML file contains the unique ID of the design model, which maintains the association between the model and its harness. The metadata file does not need to be in the same folder as the model, as long as they are both on the MATLAB® path.

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

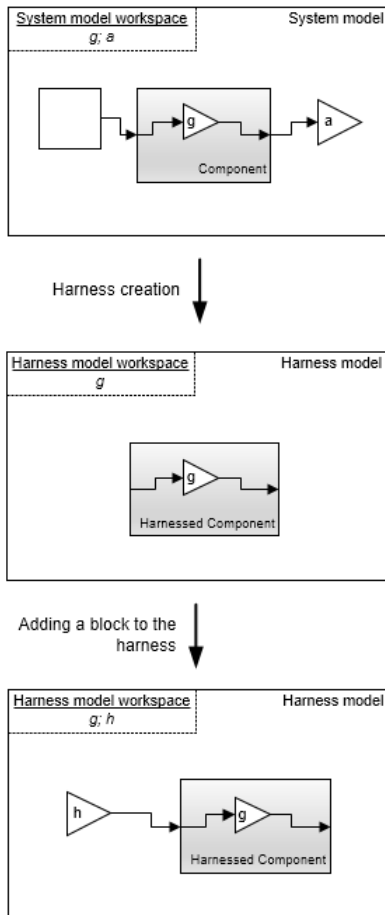
- A model component, such as a subsystem, library block, Subsystem Reference block, Model block, or System Composer™ component. The test harness isolates the component in a separate simulation environment. If you convert a Subsystem or Subsystem Reference block to a Model block, the test harnesses are transferred to the model reference (see “Test Harness Considerations” on page 2-5).
- A top-level model. The component under test is a Model block referencing the main model. You can also build a test harness in a subsystem 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 the parameters associated with the component. For example, suppose that you create a test harness for a component that contains a Gain block and then add a second Gain block to the harness.

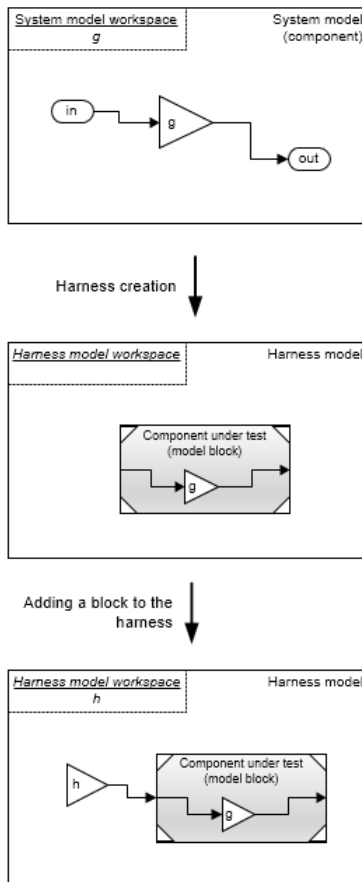
- The parameter g defines part of the component under test, so the harness model workspace contains a copy of g .
- The parameter a defines part of the main model outside of the component under test, so the harness model workspace does not contain a copy of a .
- The parameter h is the gain of the Gain block that you added to the harness. Because this block is outside the component under test, 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 that references the main model, so the parameters remain in the main model workspace. For example, suppose that you create a test harness for a top-level model that contains a Gain block and then add a second Gain block to the harness.

- The component under test references the main model, and the parameter g exists in the main model workspace. The harness model workspace does not contain a copy of g .
- The parameter h is the gain of the Gain block that you added to the harness. Because this block is outside the component under test, h exists only in the harness model workspace.



Resolving Parameters

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

Test Harness Considerations

- You can build a test harness for these types of model components:
 - Model blocks
 - Subsystem Reference blocks
 - Subsystem blocks
 - Reusable library subsystems that have function interfaces and are at the top-level of the library
 - Stateflow® blocks, including Stateflow charts, Truth Table blocks, State Transition Table blocks, and Test Sequence blocks
 - System Composer components
 - C Caller blocks
 - MATLAB Function blocks
 - User-defined function blocks

- Test harnesses are not supported for these types of Stateflow objects:
 - Any component within a Subsystem Reference block
 - Atomic subcharts
 - Simulink based states
 - Simulink functions
 - MATLAB functions
- Open only one test harness at a time for each Simulink model.
- Do not comment out the component under test in the test harness. Commenting out the component under test might cause unexpected behavior.
- Model and test harness locking is specific to each type of synchronization. For information on synchronization, see “Synchronization Mode” on page 2-19.
- The signal names used in the component under test propagate from the model to the test harness. For subsystem harnesses, some propagated signal names might be visible only after you compile the harness. For block diagram harnesses, signal names are propagated even if you do not select **Show propagated signals** in the Signal Properties dialog box.
- Subsystem and Subsystem Reference blocks
 - Test harnesses attached to Subsystem models:
 - Always synchronize with the underlying model
 - Are created without compiling
 - Do not support post-build callbacks
 - Do not auto shape inputs
 - If a subsystem has a test harness, you cannot expand the subsystem contents into the model that contains the subsystem. Delete the test harness before expanding the subsystem. For more information, see “Expand Subsystem Contents”.
 - Subsystem Reference blocks sync their block parameters, but not their block contents.
 - When you convert a Subsystem or Subsystem Reference block to a Model block, the test harnesses are transferred to the model reference. Harnesses on the Subsystem block are converted to block diagram harnesses. Nested harnesses within the subsystem are copied to identical blocks in the model reference. All transferred harnesses are internal harnesses in the model reference. Test harnesses might be renamed when they are transferred. You can see feedback about the harness transfer in the **Complete Conversion** pane of the Conversion Advisor or at the MATLAB Command line.

These limitations apply to converting a Subsystem or Subsystem Reference block to a Model block:

- SIL and PIL harnesses are not transferred.
- Requirements in a test harness for a subsystem are not transferred. You must copy them manually.
- If your test harness contains a To Workspace block, the block variable is not saved in the base workspace after the test finishes running. Upon test completion, the base workspace is restored to its original state.
- The Upgrade Advisor and XML differencing are not supported for test harness models.
- A test harness with a Signal Editor block source does not support:

- Frame-based signals
- Variable-dimension signals
- For a test harness with a Test Sequence block or Stateflow chart as the source, all inputs to the component under test must operate with the same sample time.
- These considerations apply to collecting coverage in a test harness:
 - 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, 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 2-54.
 - 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` (Simulink Coverage).

For information on coverage, see “Collect Coverage in Tests” on page 6-135

See Also

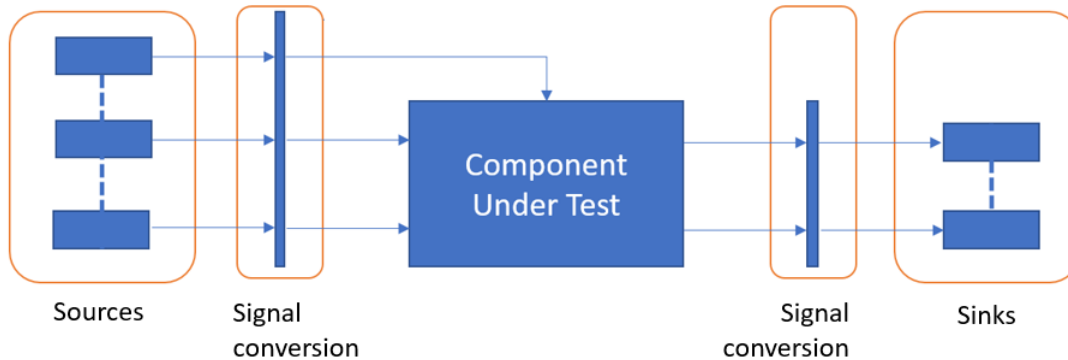
“Create a Test Harness” | Model Reference Conversion Advisor

More About

- “Compare Capabilities of Model Components”

Test Harness Construction for Specific Model Elements

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



Signal Conversion

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

- Data type
- Dimensions
- Complexity

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

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

By default, signal conversion subsystems are locked from editing.

Function Calls

Function Call Drivers

If the component under test has function call inputs, a Test Sequence block, MATLAB Function block, or Stateflow chart 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 Editor	Ground
From Workspace	Ground
From File	Ground
Test Sequence	Ground
Chart	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, MATLAB Function block, or Stateflow chart 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, MATLAB Function block, or Stateflow chart.

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 if the harness owner is not a Subsystem Reference block, or if the harness owner is a Subsystem Reference block that does not have an attached data dictionary.
- 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 or Import Test Harnesses and Select Properties” on page 2-13

Create or Import Test Harnesses and Select Properties

In this section...

“Create a Test Harness For a Top Level Model” on page 2-13

“Create Test Harnesses for Model Components” on page 2-13

“Import a Test Harness” on page 2-14

“Preview and Open Test Harnesses” on page 2-14

“Change Test Harness Properties” on page 2-14

“Considerations for Selecting Test Harness Properties” on page 2-15

“Create Test Harness Dialog Box Properties” on page 2-15

“Import Test Harness Dialog Box Properties” on page 2-21

“Customize Test Harness Creation Default Property Values” on page 2-22

Test harnesses are separate workspaces you can use to isolate tests from your design model. You can create test harnesses for your whole model or for one or more components in your model. You can also import a test harness from a standalone test model.

Create a Test Harness For a Top Level Model

To create a test harness for a top-level model (including subsystem and model reference models):

- 1 Right-click in the Simulink model and click **Test Harness > Create for Model** to open the Create Test Harness dialog box.
- 2 After selecting the desired options, click **OK** to create the test harness.

For information on the Create Test Harness dialog box properties, see “Create Test Harness Dialog Box Properties” on page 2-15.

Create Test Harnesses for Model Components

To create a test harness for one or multiple model components:

- 1 On the **Apps** tab, under Model Verification, Validation, and Test, click **Simulink Test**.
- 2 On the **Tests** tab, click **Simulink Test Manager** to open the Test Manager.
- 3 Create a new test file in the Test Manager.
- 4 Click **New > Test for Model Component**, which opens the Create Test for Model Component workflow wizard.
- 5 On the first page of the wizard, specify the model and select the component or components. If you select multiple components, the wizard creates a test harness for each component.
- 6 Create the test harness or harnesses by completing the wizard pages.

Note The Create Test for Model Component workflow wizard exposes a subset of test harness options. If your test harness does not need to use non-default options, use the wizard to create a harness quickly. If you need to change other options, use the Test Manager for the test harness you

created with the wizard. For information on using the wizard and the properties it sets, see “Generate Tests and Test Harnesses for a Model or Components” on page 6-24.

Import a Test Harness

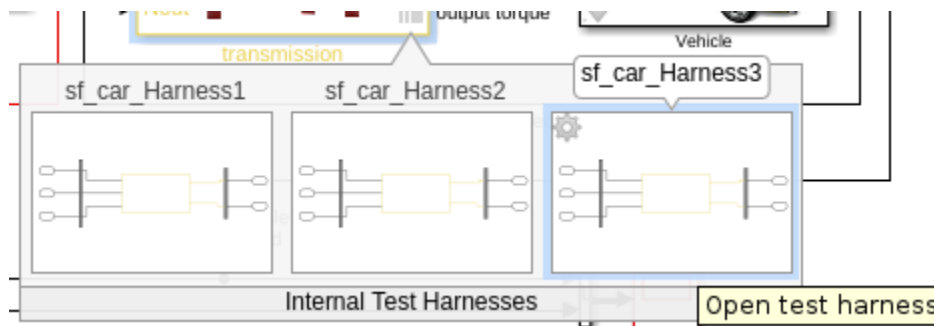
To import a test harness from a standalone test model:

- 1 Right-click in the Simulink model or on a model component and click **Test Harness > Import for Model** or **Test Harness > Import for Component**, respectively, to open the Import Test Harness dialog box.
- 2 After selecting the desired options, click **OK** to import the test harness.

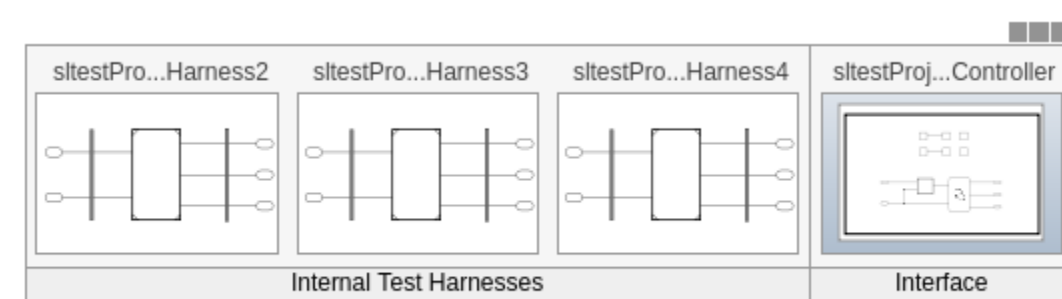
For information on the Import Test Harness dialog box properties, see “Import Test Harness Dialog Box Properties” on page 2-21.

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.



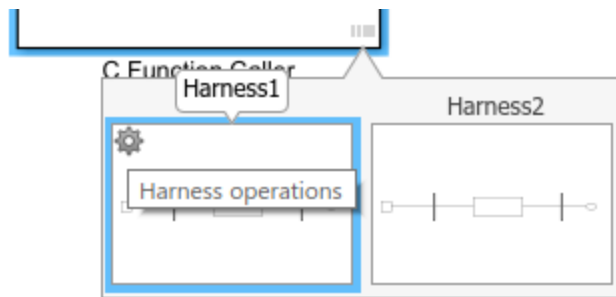
To view test harnesses for a model block diagram, click the pullout icon in the model canvas. To open a test harness, click a tile.



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.

Note The following sections describe the test harness properties in the Create Test Harness dialog box. For information on the test harness properties in the Create Test for Model Component wizard, see “Generate Tests and Test Harnesses for a Model or Components” on page 6-24.

Create Test Harness Dialog Box Properties

Harness Name

Test harnesses must use valid MATLAB file names.

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 sets 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 in the current working folder, and adds a harness information XML file to the model folder. The harness information file can be in any location that is on the MATLAB path.

See “Manage Test Harnesses” on page 2-31.

Select Function Interface

Select the function interface to associate with the reusable library subsystem test harness. This option appears only if the component under test is a reusable library subsystem with a function interface.

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 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.

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 Editor, From Workspace, or From File source blocks. This option does not apply to Subsystem models.

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. Use a scheduler to control the number of times and the order in which blocks or subsystems are executed. To include a scheduler block in your test harness, select a block from the drop-down list. You can use a Test Sequence block, a MATLAB Function block, or a Stateflow chart as the scheduler.

- **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.

For information on schedulers, see “Rate Transitions” (Embedded Coder), “Rate Transitions and Asynchronous Blocks” (Embedded Coder), or “Schedule Simulink Functions by Using Stateflow” (Stateflow)

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.

When running the test harness, if you encounter an error about executing a function call at an initialize, reset, or terminate port, use these commands to hide and disconnect the ports.

Subsystem_name is the system under test in the test harness.

```
set_param(<Subsystem_name>, 'ShowModelInitializePort', 'off');
set_param(<Subsystem_name>, 'ShowModelResetPorts', 'off');
set_param(<Subsystem_name>, 'ShowModelTerminatePort', 'off');
```

Add Separate Assessment Block

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

A Test Assessment block is a separate Test Sequence block configured with properties commonly used for verifying the component under test. For more information, see “Assess Simulation and Compare Output Data” on page 3-14 and “Assess Model Simulation Using verify Statements” on page 3-18.

Log Output Signals

Select **Log output signals** to log all output signals of the component under test. You can use this option only when creating a new harness. Signals are logged during test case execution and return test results. If an output signal does not have a name or a propagated name, it is assigned one in the harness using the format <component under test name>:<output port number>. To remove a signal from being logged, open the harness, right-click the signal, and select **Stop Logging Selected Signals**.

Open Harness After Creation

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

Create without compiling the model

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

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

You may need to add blocks such as signal conversion blocks to the test harness. You can rebuild the harness when you are ready to compile the main model. For more information, see “Synchronize Changes Between Test Harness and Model” on page 2-54.

Test harnesses for Subsystem models are created without compiling the 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.

Subsystem model test harnesses do not support SIL or PIL verification.

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. To regenerate code for the SIL or PIL block in the test harness, select **Rebuild Harness > Update Harness Configuration Settings and Model Workspace**.

Use generated code to create SIL/PIL block

If generated code for the SIL/PIL block already exists, select this property to use that existing code instead of regenerating the code. This option is available only for subsystem harnesses. It does not apply to Subsystem model test harnesses.

Build folder

Specify the folder that contains the generated code for the SIL/PIL block. This option is available only if you selected **Use generated code to create SIL/PIL block**.

Post-create callback method

You can customize your test harness using one or more post-create callbacks. 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. If you specify more than one callback, separate them using commas. The callbacks run in the order that they are listed. For more information, see “Customize Test Harnesses” on page 2-42. This option does not apply to Subsystem model test harnesses.

Rebuild harness on open

When you select this property, the test harness rebuilds every time you open it. If you specified to use existing generated code for a SIL/PIL subsystem using `sltest.harness.create` or `sltest.harness.set`, the harness rebuild uses that code instead of regenerating it. For details on the rebuild process, see “Synchronize Changes Between Test Harness and Model” on page 2-54. This option does not apply to Subsystem model test harnesses.

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 2-54. This option does not apply to Subsystem model test harnesses.

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 2-42. This option does not apply to Subsystem model test harnesses.

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. The synchronization mode also affects model and harness locking.

For additional information and limitations, see “Synchronize Changes Between Test Harness and Model” on page 2-54, “Set Synchronization for a New Test Harness” on page 2-54, and “Model and Test Harness Locking” on page 2-34.

Synchronization Type	Description	Availability	Model, CUT, and Harness Locking When Harness Is Open
Synchronize on harness open and close	When the test harness opens, the test harness components and parameters synchronize from the model to the test harness. When the test harness closes, the same elements synchronize from the harness to the model.	Available for: <ul style="list-style-type: none"> • Subsystems, including Stateflow charts and MATLAB Function blocks • For Subsystem Reference blocks, only the block parameters are synchronized • Model blocks • S-function blocks Not available for: <ul style="list-style-type: none"> • Block diagrams • SIL/PIL harnesses • Subsystem model harnesses 	The main model and harness are unlocked for all types of CUTs. Subsystem CUTs in the model are locked. Subsystem CUTs in the harness are unlocked,

Synchronization Type	Description	Availability	Model, CUT, and Harness Locking When Harness Is Open
<p>Synchronize on harness open</p>	<p>When the harness opens, the harness components and parameters synchronize from the model to the test harness.</p>	<p>Available for:</p> <ul style="list-style-type: none"> • Block diagrams • Subsystems, including Stateflow charts and MATLAB Function blocks • For Subsystem Reference blocks, only the block parameters are synchronized • Model reference blocks • S-function blocks <p>Not available for:</p> <ul style="list-style-type: none"> • SIL/PIL harnesses • Subsystem model harnesses 	<p>The main model and harness are unlocked for all types of CUTs.</p> <p>Subsystem CUTs in the model and the harness are locked.</p>
<p>Synchronize only during push and rebuild</p>	<p>Synchronizes when you click Push Changes or Rebuild Harness. Push synchronizes changes from the test harness to the model. Rebuild synchronizes changes from the model to the test harness.</p>	<p>Available for:</p> <ul style="list-style-type: none"> • Subsystems, including Stateflow charts and MATLAB Function blocks • Model reference blocks • S-function blocks • Subsystem models, which always synchronize on the push and rebuild only. <p>Not available for:</p> <ul style="list-style-type: none"> • Block diagrams • SIL/PIL harnesses • Components in libraries 	<p>The main model, harness, and all types of CUTs in the model and harness, including subsystems, are unlocked.</p>

Synchronization Type	Description	Availability	Model, CUT, and Harness Locking When Harness Is Open
Synchronize only during rebuild	Synchronizes only when you click Rebuild Harness . Changes synchronize from the model to the test harness.	Available for: <ul style="list-style-type: none"> • Block diagrams • Model reference blocks • SIL/PIL verification mode components Not available for: <ul style="list-style-type: none"> • Subsystems, including Stateflow charts and MATLAB Function blocks • S-function blocks • Components in libraries 	The main model, harness, and all types of CUTs in the model are unlocked. All types of CUTs in the harness are unlocked, except SIL/PIL verification mode components, which are locked and masked.

Import Test Harness Dialog Box Properties

Name

Name to use for the imported test harness. Test harnesses must use valid MATLAB file names.

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 sets 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 in the current working folder, and adds a harness information XML file to the model folder. The harness information file can be in any location that is on the MATLAB path.

See “Manage Test Harnesses” on page 2-31.

Simulink model to import

Name of or full path to the Simulink standalone test model to import.

Component under Test in imported model

Path to the component under test in the imported Simulink standalone test model.

Rebuild harness on open

When you select this property, the test harness rebuilds every time you open it. If you specified to use existing generated code for a SIL/PIL subsystem using `sltest.harness.create` or `sltest.harness.set`, the harness rebuild uses that code instead of regenerating it. For details on the rebuild process, see “Synchronize Changes Between Test Harness and Model” on page 2-54. This option does not apply to Subsystem model test harnesses.

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 2-54. This option does not apply to Subsystem model test harnesses.

Customize Test Harness Creation Default Property Values

To set default property values for the creation of new test harnesses, use an `sl_customization` file or the `setHarnessCreateDefaults` function. All newly created test harnesses use the new default values.

To see the current default test harness property values, use `sltest.harness.getHarnessCreateDefaults`.

For an individual test harness, you can change the property values from the default values by using the Create Test Harness dialog box or the `sltest.harness.create` function. Using either of these options does not change the default values used when creating a new test harness. See also “Create Test Harness Dialog Box Properties” on page 2-15 and “Change Test Harness Properties” on page 2-14.

Set Defaults by Using an `sl_customization.m` File

To change the default property values used when creating new test harnesses, you can create an `sl_customization.m` file.

- 1 Create an `sl_customization.m` file and specify the new default property values.

You can set all name-value arguments of `sltest.harness.create`, except where noted.

This sample `sl_customization.m` file sets the harness name to `myTestHarness`, sets a post-create callback to use the `addHarnessAnnotation` function, saves the harness internally, and logs outputs:

```
function sl_customization(cm)
    % Create the struct with the harness options
    myStruct.Name="myTestHarness"
    myStruct.PostCreateCallback = "addHarnessAnnotation";
    myStruct.SaveExternally = false;
    myStruct.LogOutputs = true;

    % Invoke harness customization
    cObj = cm.SimulinkTestCustomizer;
    cObj.setHarnessCreateDefaults(myStruct);
end
```

- 2 Save the `sl_customization.m` file.

- 3 Add the file to the MATLAB path.
- 4 Register the new customizations by reloading Simulink or by using `sl_refresh_customizations`. For more information, see “Register Customizations with Simulink”.

Note When you register a file, its values become the default property values. All new test harnesses use the default property values and all previously registered values are cleared.

To view the customized default values, use `sltest.harness.getHarnessCreateDefaults`.

Set Defaults by Using the `setHarnessCreateDefaults` Function

You can also use `sltest.harness.setHarnessCreateDefaults` to set the default property values. You can set any of the name-value pair properties described in `sltest.harness.create`, except where noted. Using `sltest.harness.setHarnessCreateDefaults` saves and registers the default property values. However, if you already set and registered values using an `sl_customization.m` file, using `sltest.harness.setHarnessCreateDefaults` overwrites the values specified in the file.

See Also

`sltest.harness.setHarnessCreateDefaults` |
`sltest.harness.getHarnessCreateDefaults` | Test Sequence

Related Examples

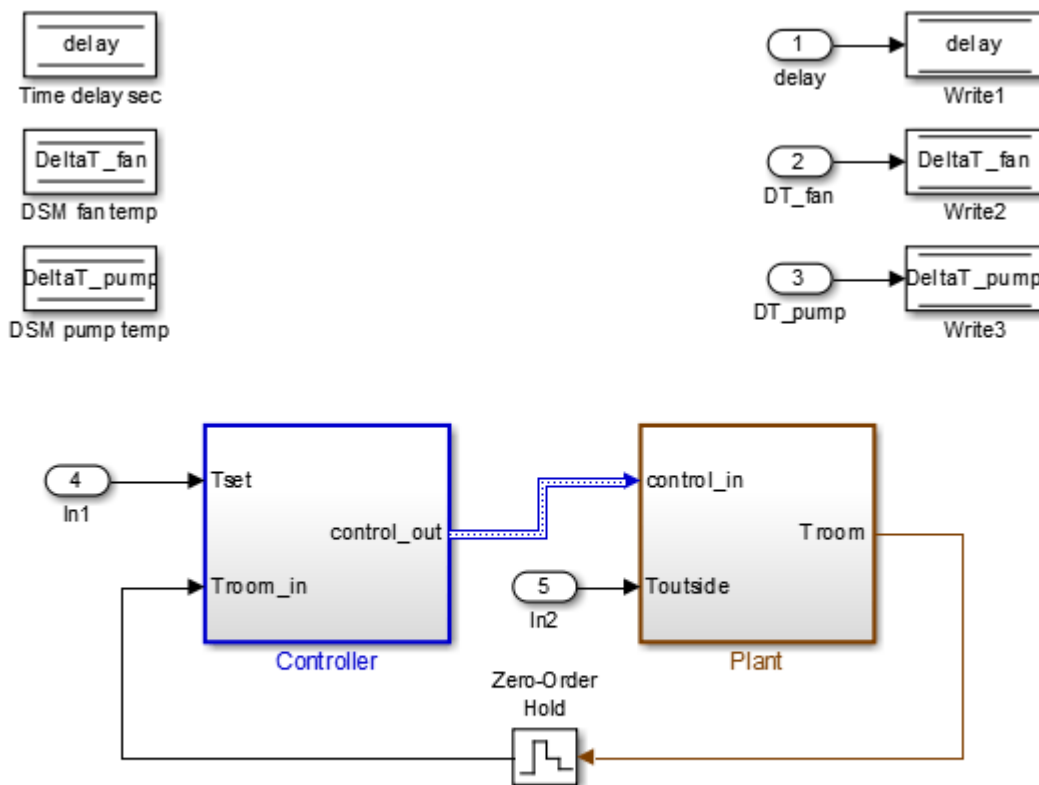
- “Test Harness and Model Relationship” on page 2-2
- “Synchronize Changes Between Test Harness and Model” on page 2-54
- “Register Customizations with Simulink”
- “Manage Test Harnesses” on page 2-31
- “Customize Test Harnesses” on page 2-42

Refine, Test, and Debug a Subsystem

This example shows how to refine and test a controller subsystem using a test harness. 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. In the example, the main model is a controller-plant model of an air conditioning/heat pump unit. The controller must operate according to several requirements.

Open the Model

sltestHeatpumpExample



Copyright 1990-2014 The MathWorks, Inc.

In the example model:

- The controller accepts the 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.

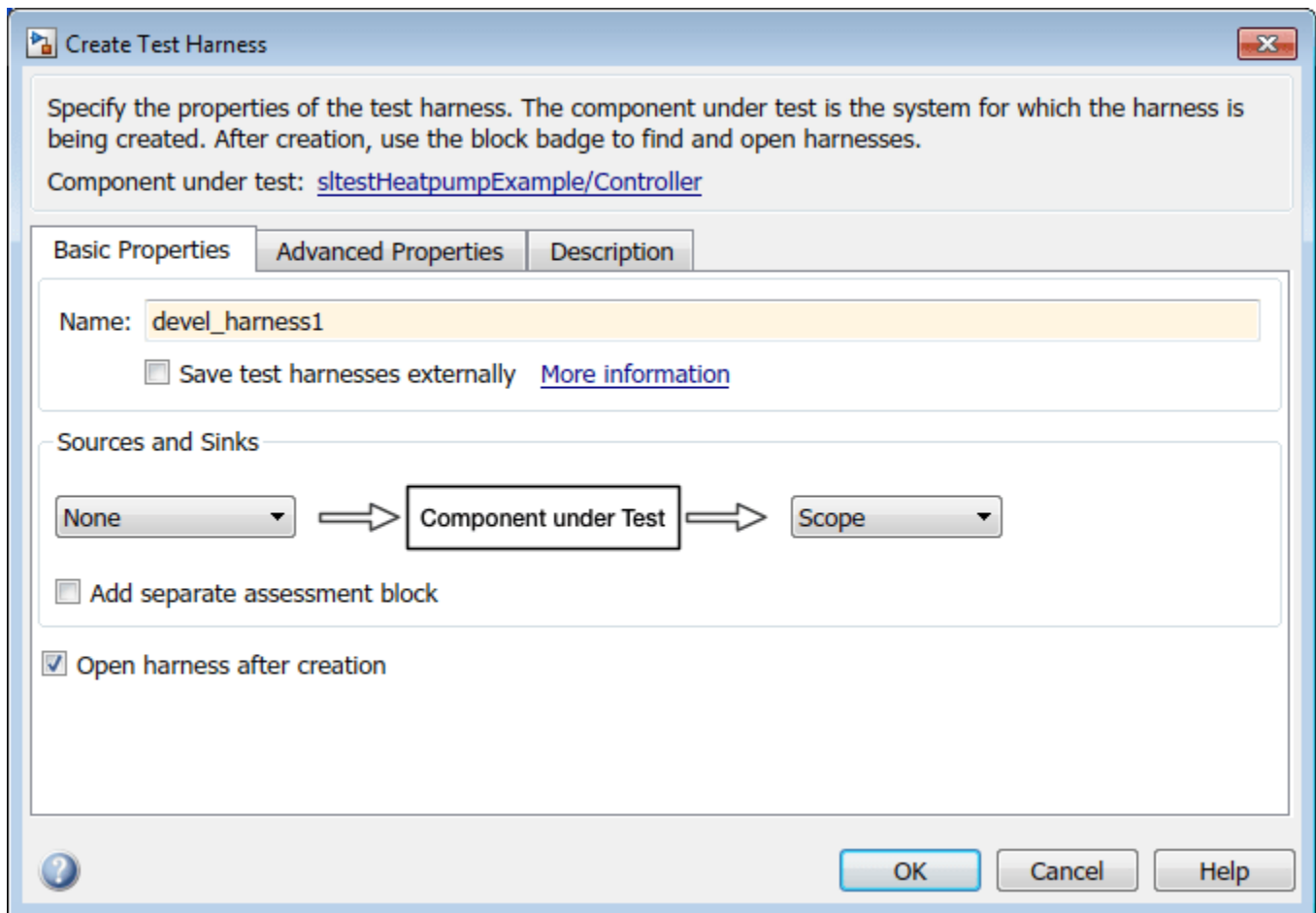
Temperature Condition States

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

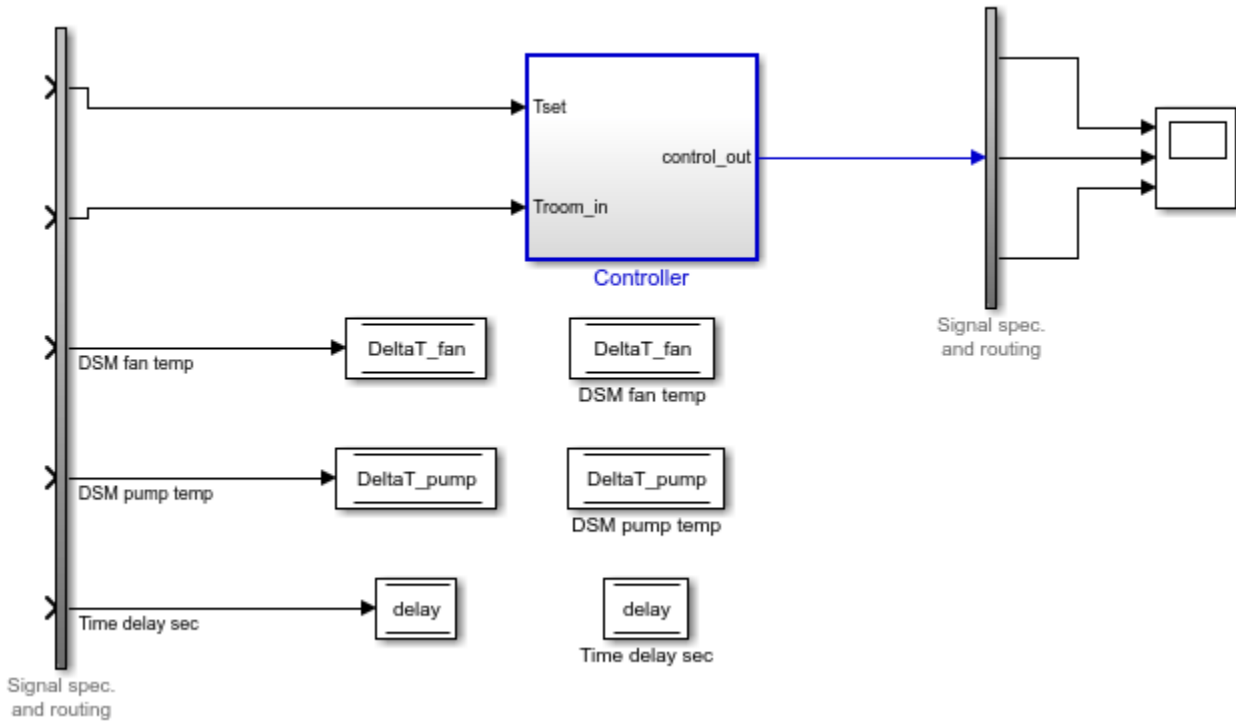
Temperature Condition	System State	Fan Command	Pump Command	Pump Direction
$ T_{room_in} - T_{set} < \Delta T_{fan}$	idle	0	0	0
$\Delta T_{fan} \leq T_{room_in} - T_{set} < \Delta T_{pump}$	fan only	1	0	0
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} < T_{room_in}$	cooling	1	1	-1
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} > T_{room_in}$	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_harness1
 - Clear **Save test harness externally**
 - **Sources and Sinks:** None and Scope
 - Clear **Add separate assessment block**
 - Select **Open harness after creation**



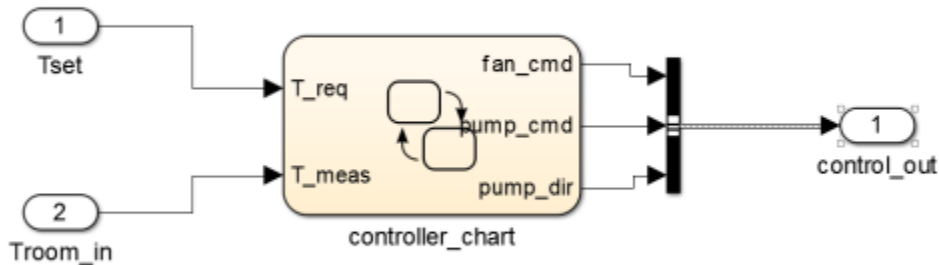
3. Click **OK** to create the harness.



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.

devel_harness_1 ▶ Controller ▶



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

Add Test Inputs and Test the Controller

1. Navigate to the top level of devel_harness1.

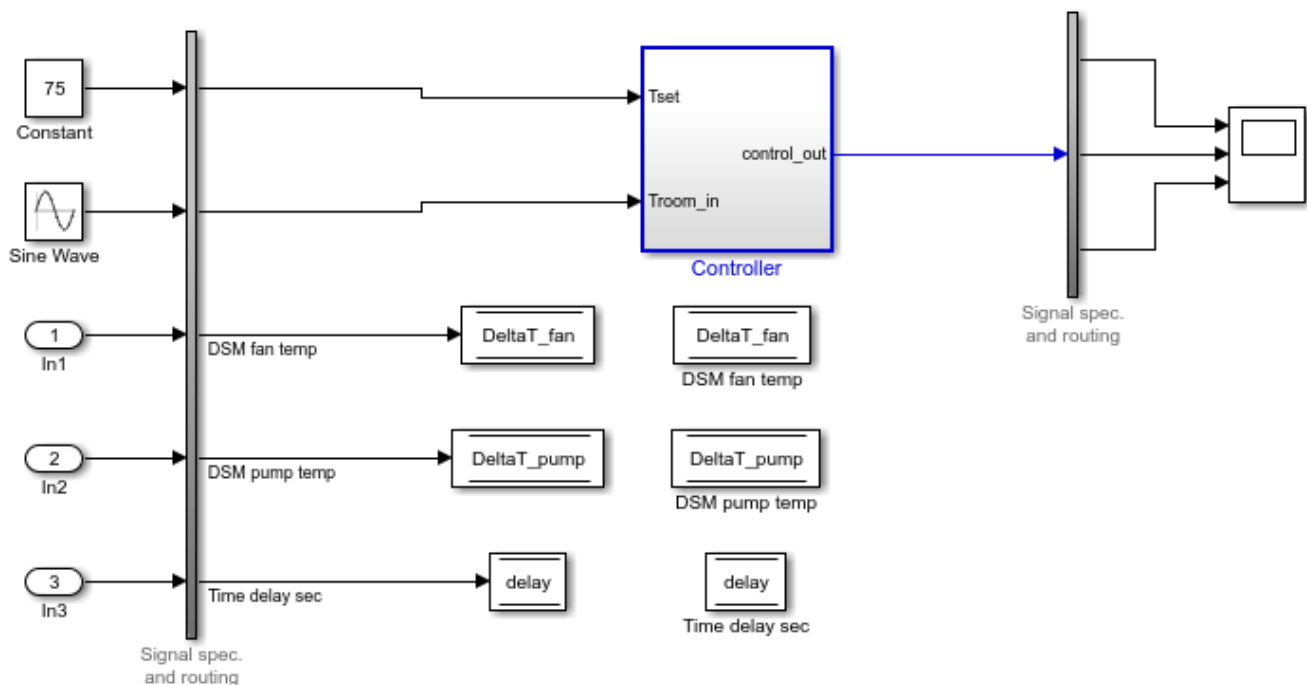
2. Create a test input for the harness with a constant T_{set} and a time-varying T_{room} . Connect a Constant block to the T_{set} input and set the value to 75.

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 T_{room_in} .

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

- **Amplitude:** 15
- **Bias:** 75
- **Frequency:** $2\pi/3600$
- **Phase (rad):** 0
- **Sample time:** 1
- Select **Interpret vector parameters as 1-D**

5. Connect Inport blocks to the Data Store Write inputs.



6. Click **Model Settings** to open 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. Then, click **Apply**.

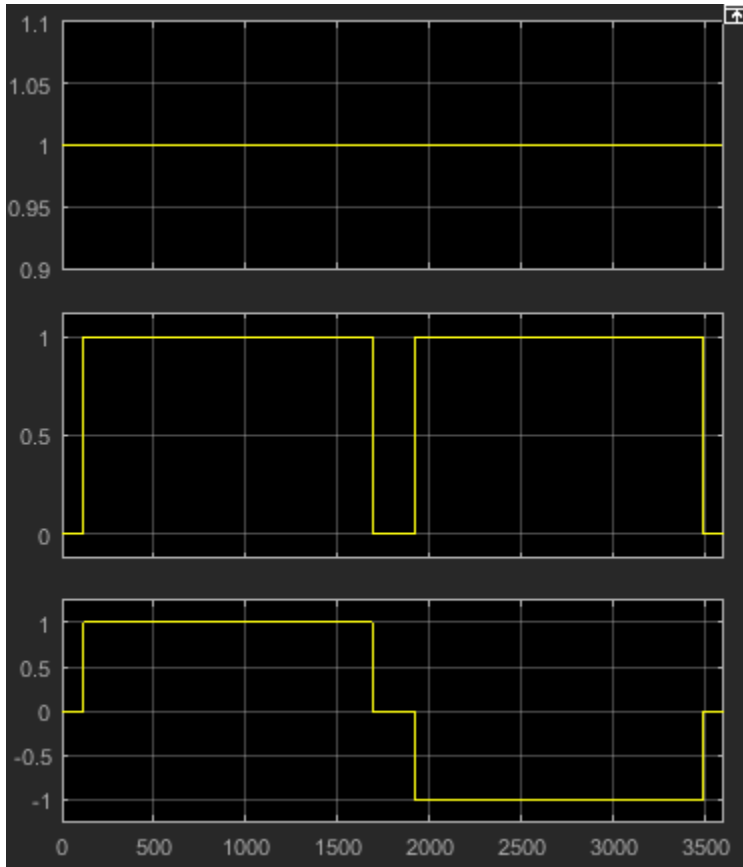
7. In the **Solver** pane, set **Stop time** to 3600. Click **OK**.

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 that the controller output, `fan_cmd`, is 1 during the IDLE condition where $|T_{room} - T_{set}| < \Delta T_{fan}$. This is a bug because `fan_cmd` should be 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

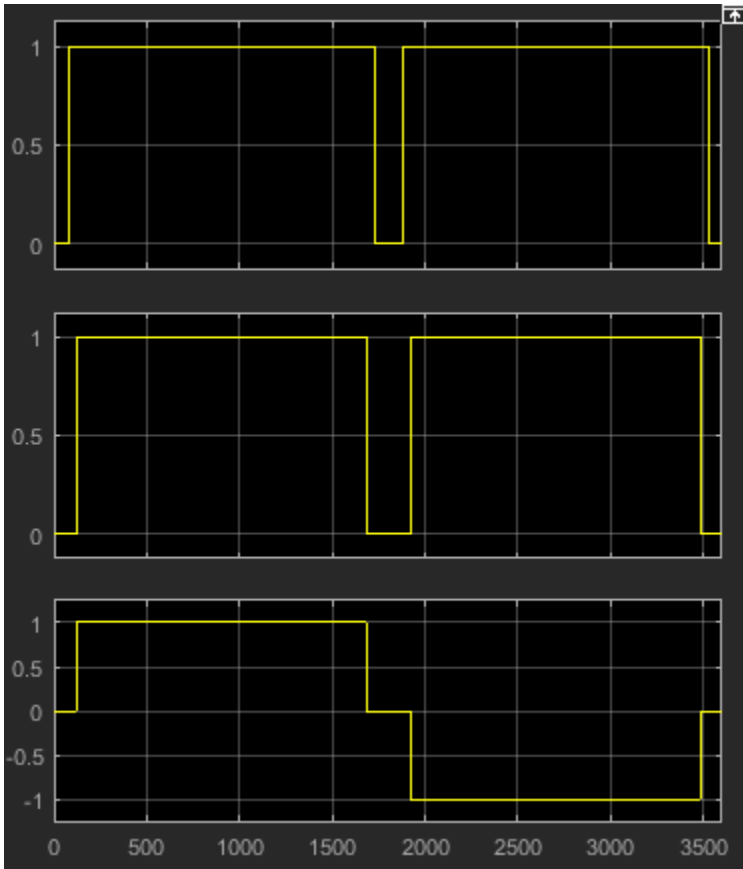
entry:

```
fan_cmd = 0;
```

```
pump_cmd = 0;
```

```
pump_dir = 0;
```

5. Simulate the harness model again and observe the outputs.



fan_cmd now meets the requirement to equal 0 at IDLE.

Manage Test Harnesses

In this section...

“Internal and External Test Harnesses” on page 2-31
“Manage External Test Harnesses” on page 2-31
“Convert Between Internal and External Test Harnesses” on page 2-32
“Preview and Open Test Harnesses” on page 2-33
“Model and Test Harness Locking” on page 2-34
“Find Test Cases Associated with a Test Harness” on page 2-34
“Export Test Harnesses to Standalone Models” on page 2-35
“Move and Clone Test Harnesses” on page 2-35
“Clone and Export a Test Harness to a Separate Model” on page 2-37
“Delete Test Harnesses Programmatically” on page 2-39
“Export Test Harness to Previous Version” on page 2-40

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 configuration management 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 or Import Test Harnesses and Select Properties” on page 2-13.

Manage External Test Harnesses

Harnesses stored externally use a separate SLX file for each harness, and a `<modelName>_harnessInfo.xml` file that contains metadata linking the model and the harnesses. Changing test harnesses can change the `harnessInfo.xml` file. The metadata and the model are linked using a unique ID for the model. The test harness metadata XML file is created and stored by default in the same folder as the model. You can move the metadata XML file to a different folder on the MATLAB path, if desired. The link between the model and its harnesses persists as long as the harness metadata file, model, and harnesses are all on the MATLAB path.

Follow these guidelines for external test harnesses:

Warning Do not delete or make any manual changes to the `harnessInfo.xml` file. Deleting the `harnessInfo.xml` file might sever 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.
- Folders containing test harness SLX files must be on the MATLAB path.
- If the test harness `harnessInfo.xml` file is not in the same folder as the model, the XML file or its folder 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 **Save > Save as**. Using **Save as** copies external test harnesses to the destination folder of the new model, renames the harnesses, 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 **Save > 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:

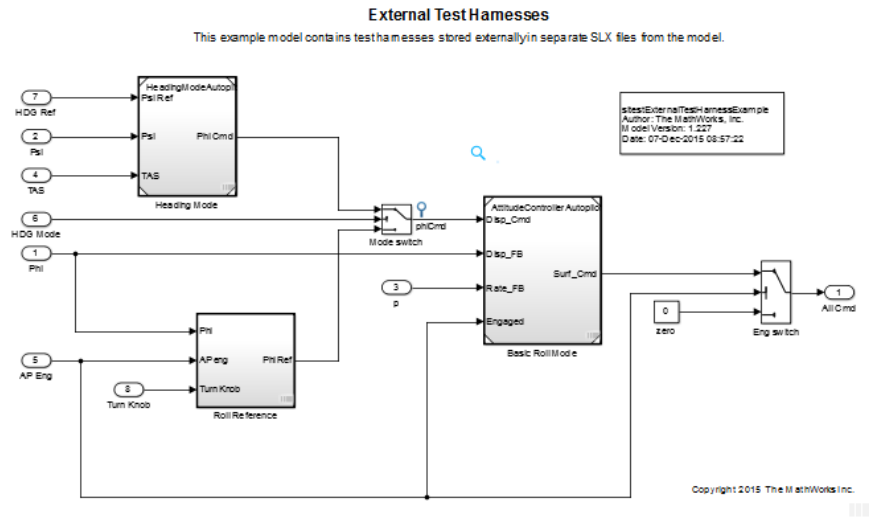
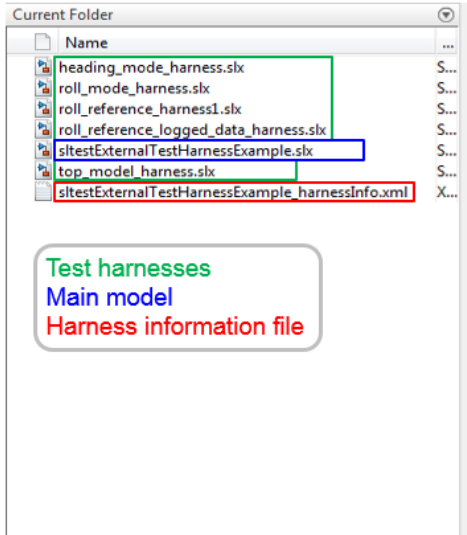
- Develop your model using internal test harnesses so that you can more easily share the model for review. When you complete your design and place the model under change control, convert to external harnesses.
- Use the configuration management model as the starting point for a new design. Test the existing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

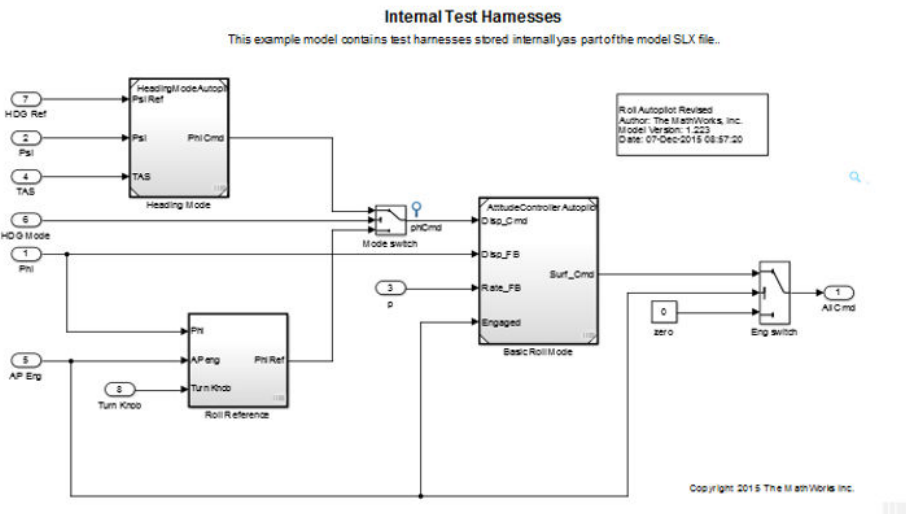
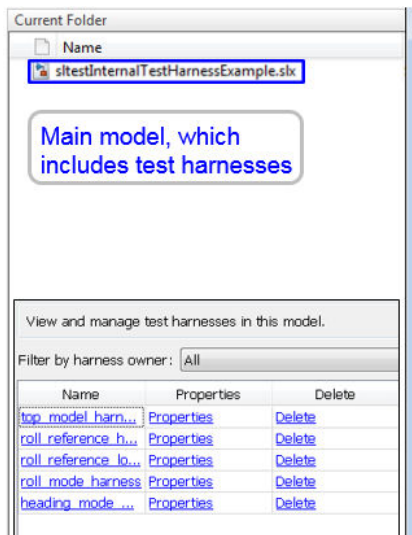
- 1 Navigate to the top of the main model.
- 2 On the **Apps** tab, under Model Verification, Validation, and Test, click **Simulink Test**. Then, on the **Tests** tab, click **Manage Test Harnesses > Convert to External Harnesses** or **Convert to Internal Harnesses**.
- 3 A dialog box provides information on the conversion procedure and the affected test harnesses. Click **Yes** to continue.

The harnesses are converted.

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

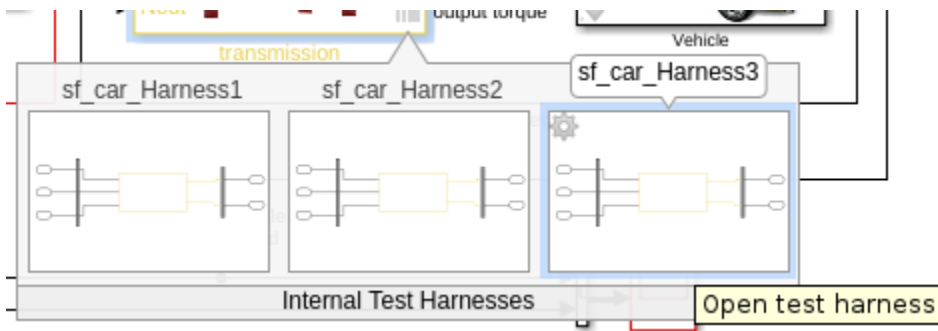


Inversely, conversion to internal test harnesses moves the test harness SLX files and the harnessInfo.xml file.

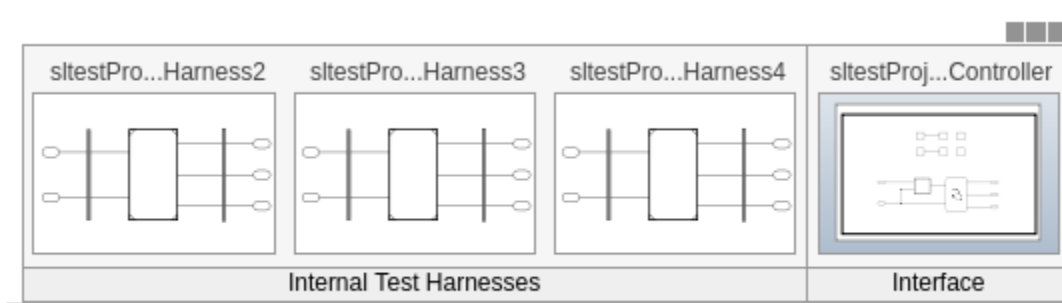


Preview and Open Test Harnesses

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.



Model and Test Harness Locking

Model and test harness locking is specific to each type of synchronization. For information, see “Synchronization Mode” on page 2-19.

Find Test Cases Associated with a Test Harness

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

Test harness properties ...

Open test cases:

- [slitestProjectorFanSpeedTestSuite : Fan Speed = 2300](#)
- [slitestProjectorFanSpeedTestSuite : Fan Speed = 1800](#)
- [slitestProjectorFanSpeedTestSuite : Fan Speed = 1300](#)
- [slitestProjectorFanSpeedTestSuite : Fan Speed = 800](#)



Export Test Harnesses to Standalone Models

You can export test harnesses to standalone 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 individual harness model, on the **Apps** tab, under **Model Verification, Validation, and Test**, click **Simulink Test**.
 - 2 In the **Harness** tab, click **Detach and Export**.
 - 3 In the Export Test Harness to Independent Model dialog box, click **OK**.
 - 4 In the Save As dialog box, enter a filename for the standalone harness model and click **OK**.
 - 5 The harness converts to a standalone model.

Converting removes the harness from the main model and breaks the relationship to the main model. If a model has only one harness, its `harnessInfo.xml` file is deleted. If a model has more than one harness and you delete one of them, the `harnessInfo.xml` file is updated.

- To export all harnesses in a model:
 - 1 Navigate to the top level of the model. Do not select any blocks.
 - 2 On the **Apps** tab, under **Model Verification, Validation, and Test**, click **Simulink Test**.
 - 3 In the **Harness** tab, click **Detach and Export**.
 - 4 In the Export Test Harness to Independent Model dialog box, click **OK**.
 - 5 In the Save As dialog box, enter a filename for the separate model and click **OK**.
 - 6 All test harnesses are exported and converted into standalone models.

Exporting removes the harnesses from the main model, deletes the `harnessInfo.xml` file, and breaks the relationships to the main model.

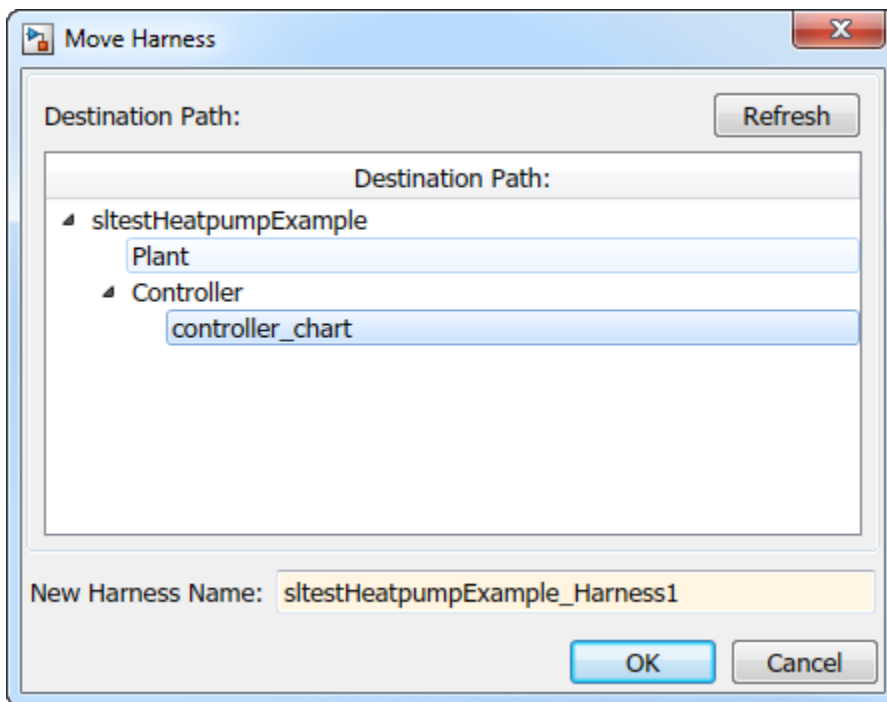
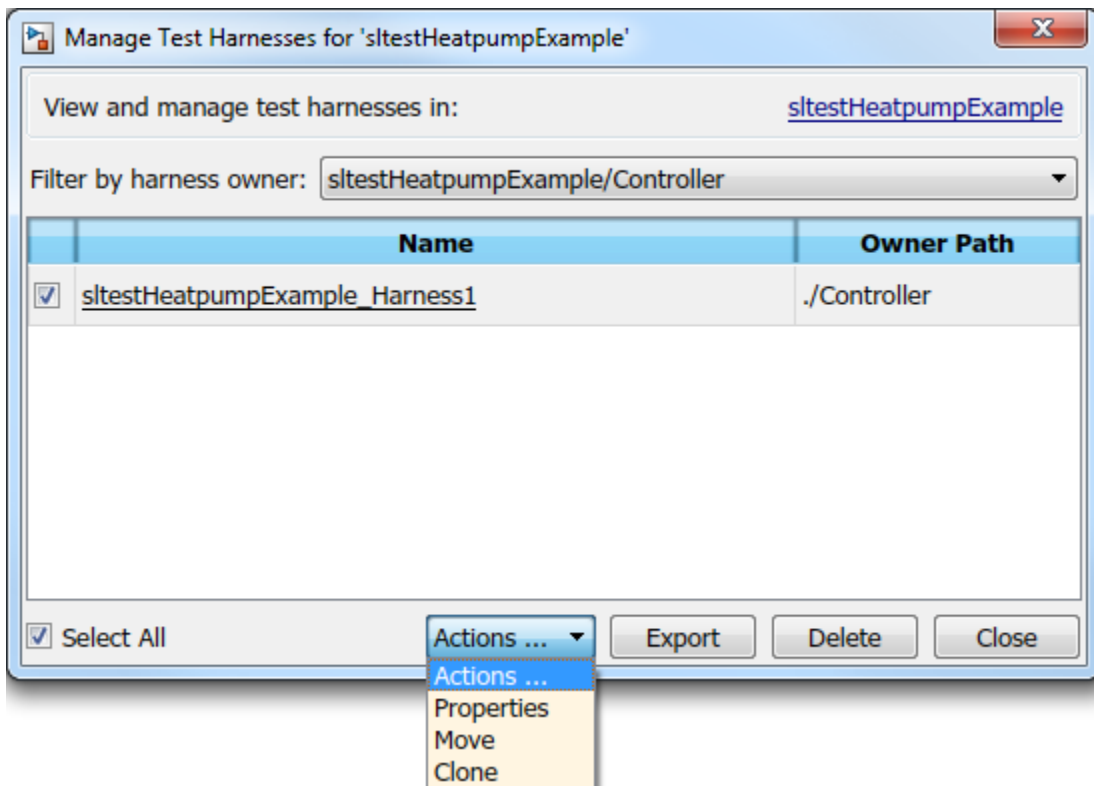
See `sltest.harness.export`.

Move and Clone Test Harnesses

Simulink Test gives you the ability to move or 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.
- Subsystem Reference block harnesses to other Subsystem Reference block harnesses.
- Subsystem Reference block harnesses to and from Subsystem model harnesses.

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 or 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.

Test Harness Transfer When Converting Subsystems to Model References

When you convert a Subsystem or Subsystem Reference block to a Model block, the test harnesses are transferred to the model being referenced. Harnesses on the Subsystem block are converted to block diagram harnesses. Nested harnesses in subsystems are copied to identical blocks in the Model block. All transferred harnesses are internal harnesses in the model reference. Test harnesses might be renamed when they are transferred. You can see feedback about the harness transfer in the **Complete Conversion** pane of the Conversion Advisor or at the MATLAB command line.

These limitations apply to converting a Subsystem or Subsystem Reference block to a Model block:

- SIL and PIL harnesses are not transferred.
- Requirements in a test harness for a subsystem are not transferred. You must copy them manually.

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

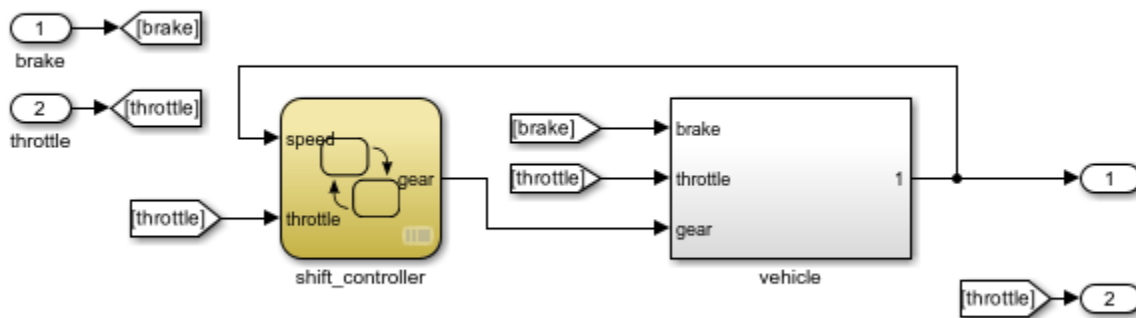
- 1 If you don't know the exact properties of the test harness you want to clone, get them using `sltest.harness.find`. You need the harness owner ID and the harness name.
- 2 Clone the test harness using `sltest.harness.clone`.
- 3 Export the test harness to a separate model using `sltest.harness.export`. 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: 10.0118
  ownerFullPath: 'sltestTestSequenceExample/shift_controller'
        ownerType: 'Simulink.SubSystem'
        isOpen: 0
        canBeOpened: 1
  verificationMode: 0
        saveExternally: 0
        rebuildOnOpen: 0
  rebuildModelData: 0
  postRebuildCallback: ''
        graphical: 0
        origSrc: 'Test Sequence'
        origSink: 'Test Assessment'
  synchronizationMode: 0
  existingBuildFolder: ''
  functionInterfaceName: ''
```

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')
```

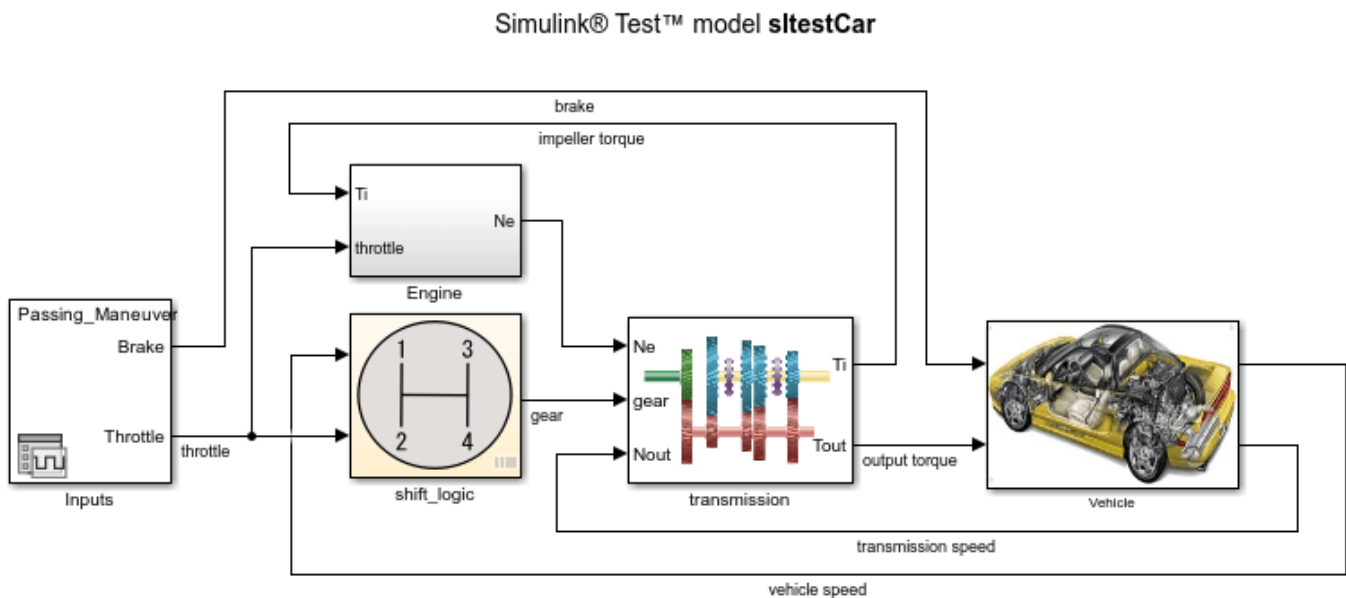
```
clear('model')
clear('properties')
close_system('sltestTestSequenceExample',0)
```

Delete Test Harnesses Programmatically

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

1. Open the model

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



Copyright 1997-2019 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
verificationMode
saveExternally
rebuildOnOpen
rebuildModelData
postRebuildCallback
graphical
origSrc
origSink
synchronizationMode
existingBuildFolder
functionInterfaceName
```

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);
```

Export Test Harness to Previous Version

When you export a Simulink model that has one or more test harnesses to a previous MATLAB version, the impact on the harnesses depends on whether they are internal or external.

- For models with internal harnesses, the harnesses stay internal and are exported with the model.
- For models with external harnesses, the harnesses in the exported model are converted to internal harnesses. To change the internal harnesses back to external, see “Convert Between Internal and External Test Harnesses” on page 2-32.

You cannot export internal or external harnesses without also exporting their parent model to a previous version. To work around this restriction, export the harness to a standalone model, export the harness model to the previous version, and then import it into the main model. After importing the harness, check the harness settings. You might need to reset some properties, such as the synchronization mode and rebuild. See “Export Test Harnesses to Standalone Models” on page 2-35 and “Change Test Harness Properties” on page 2-14.

See Also

```
sltest.harness.create | sltest.harness.clone | sltest.harness.delete |
sltest.harness.export | sltest.harness.find | sltest.harness.load |
sltest.harness.open | sltest.harness.move | Model Reference Conversion Advisor
```

More About

- “Test Harness and Model Relationship” on page 2-2

Customize Test Harnesses

In this section...

“Callback Function Definition and Harness Information” on page 2-42

“Display Harness Information struct Contents” on page 2-44

“Share Data Between Callbacks” on page 2-44

“Customize a Test Harness to Create Mixed Source Types” on page 2-45

“Test Harness Callback Example” on page 2-46

You can customize a test harness by using one or more functions that run as callbacks after creating the test harness. You can use one function to run as a callback for rebuilding the test harness. In the function, write the commands to customize your test harness. For example, you can create functions to:

- 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.

To customize a test harness using callback functions:

- 1 Create the callback function.
- 2 In the function, use the Simulink programmatic interface to script the commands to customize the test harness. For more information, see the functions listed in “Programmatic Model Editing”.
- 3 Specify the function or functions as the post-create or a single function as a post-rebuild callback:
 - For a new test harness,
 - If you are using the UI, enter the function names in the **Post-create callback method**, separated by commas, or a single function name in the **Post-rebuild callback method** in the Create Test Harness dialog box.
 - If you are using `sltest.harness.create`, specify the function as the `PostCreateCallback` or `PostRebuildCallback` value. For the `PostCreateCallback`, you can specify more than one function.
 - 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.

Another way to customize test harnesses is by setting your own defaults for creating harnesses. For more information, see “Create or Import Test Harnesses and Select Properties” on page 2-13.

Callback Function Definition and Harness Information

The callback function declaration is

```
function myfun(x)
```

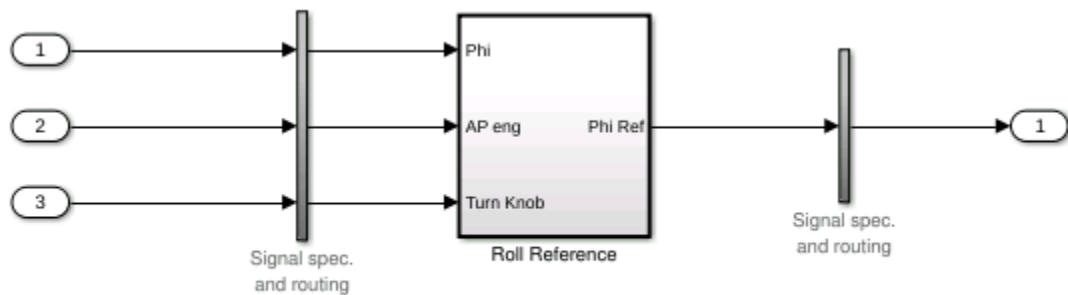
where `myfun` is the function name and `myfun` accepts input `x`. The `x` input is a struct of information about the test harness automatically created when the test harness uses the callback. You can choose the function and argument names.

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');
```

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.

Share Data Between Callbacks

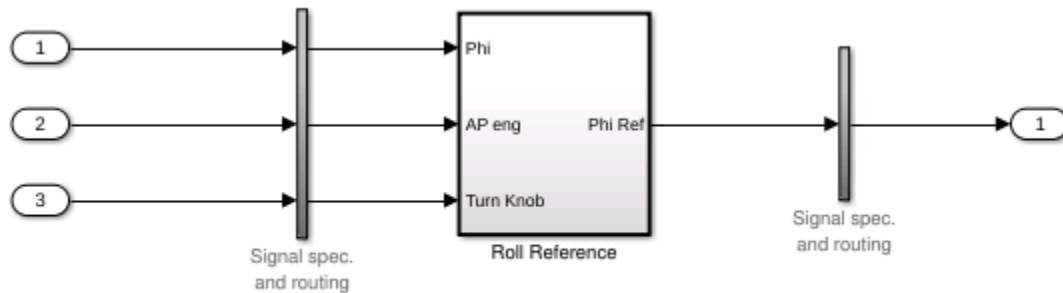
Callback scripts are evaluated in the MATLAB base workspace. To share data between callback scripts, use `assignin` and `evalin` to store and retrieve data from the base workspace. For example,

```
assignin('base', 'a', 2);
a = evalin('base', 'a');
```

For parallel execution, post-load and cleanup callbacks are evaluated on the parallel MATLAB worker, where the callbacks have their own base workspaces. Variables created in a test case pre-load callback and test file and test suite setup callbacks are also available in the parallel MATLAB worker base workspace. Prior to execution, the base workspace variables are transferred from the client MATLAB to the parallel MATLAB workers.

Customize a Test Harness to Create Mixed Source Types

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



The function follows the procedure:

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

```
function harnessCustomization(harnessInfo)

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

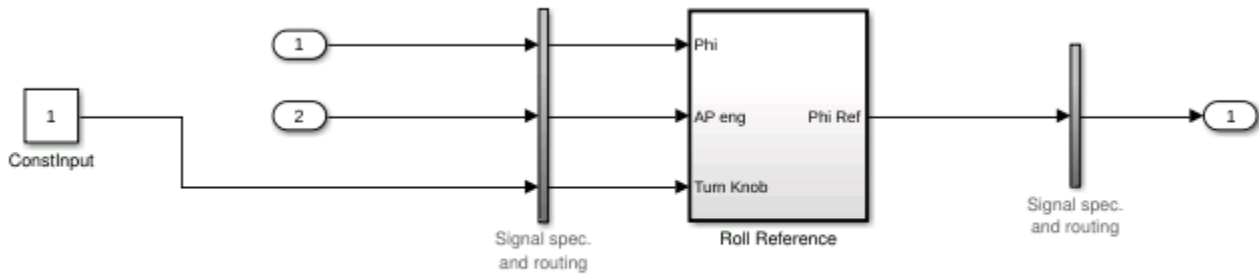
% Add Constant block:
constBlock = add_block('simulink/Sources/Constant', ...
    [harnessName '/ConstInput']);

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

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

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

end
```



Test Harness Callback Example

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

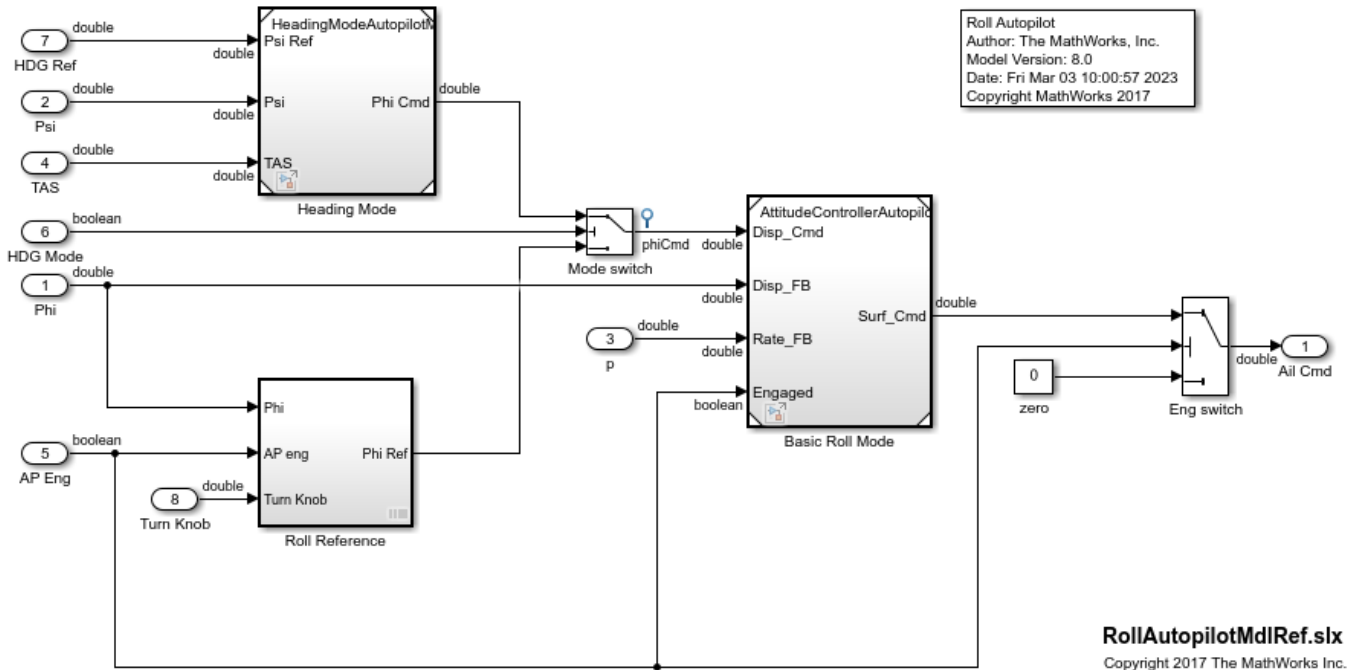
The Model

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

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

Requirements-based Testing for Controller Development

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



Get Path to the Harness Customization Function

```
cbFile = '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)
```

As an alternative to including output logging code in the callback, you can use **Log Output Signals** in the Create New Harness dialog box, or use `'LogHarnessOutputs', true` as an input to `sltest.harness.create`. These options log all component under test output signals in the test harness and return test results for those signals.

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

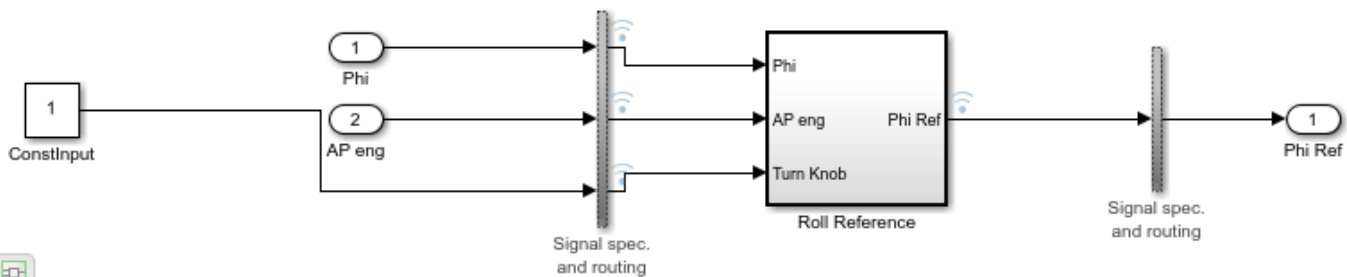
2. In the harness creation dialog box, for **Post-create callback method**, enter `harnessSourceLogCustomization`.

3. 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 or Import Test Harnesses and Select Properties” on page 2-13

Create Test Harnesses from Standalone Models

In this section...
“Test Harness Import Workflow” on page 2-49
“Component Compatibility for Test Harness Import” on page 2-50
“Import a Standalone Model as a Test Harness” on page 2-51

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 interfaces
- 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™.

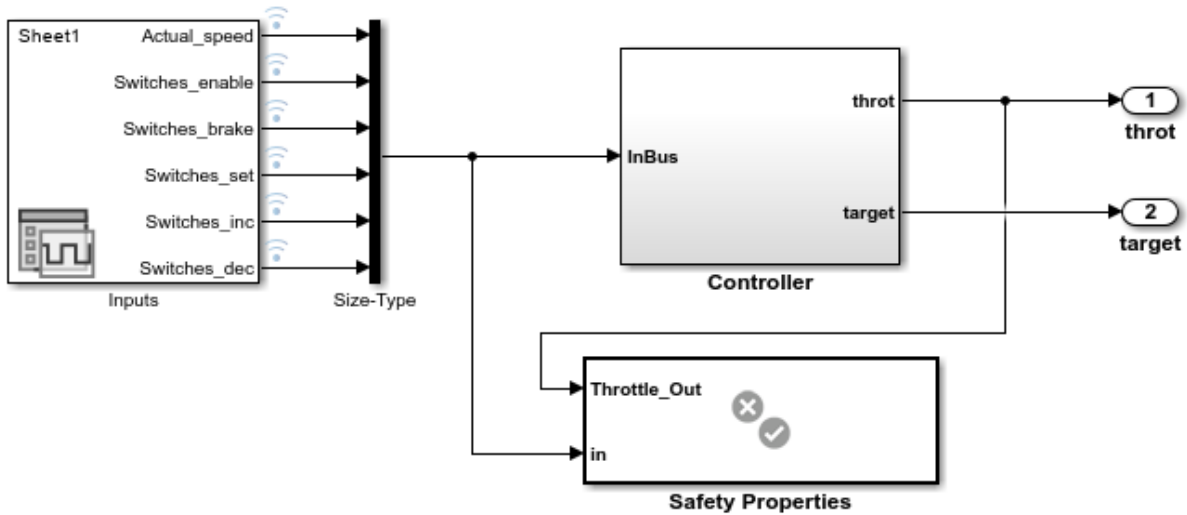
Test Harness Import Workflow

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.

Simulink Test Basic Cruise Control Verification



Copyright 2006-2022 The MathWorks, Inc.

Component Compatibility for Test Harness Import

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

Component Compatibility for Test Harness Import

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

You cannot create a test harness by importing:

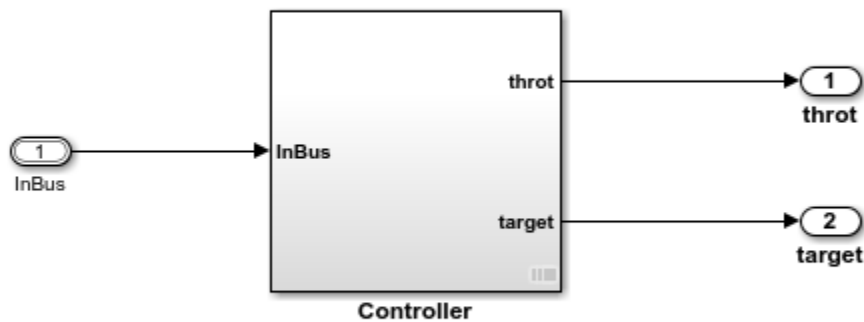
- Libraries
- Models that have existing test harnesses
- Models with unsaved changes. Save open models before importing

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.

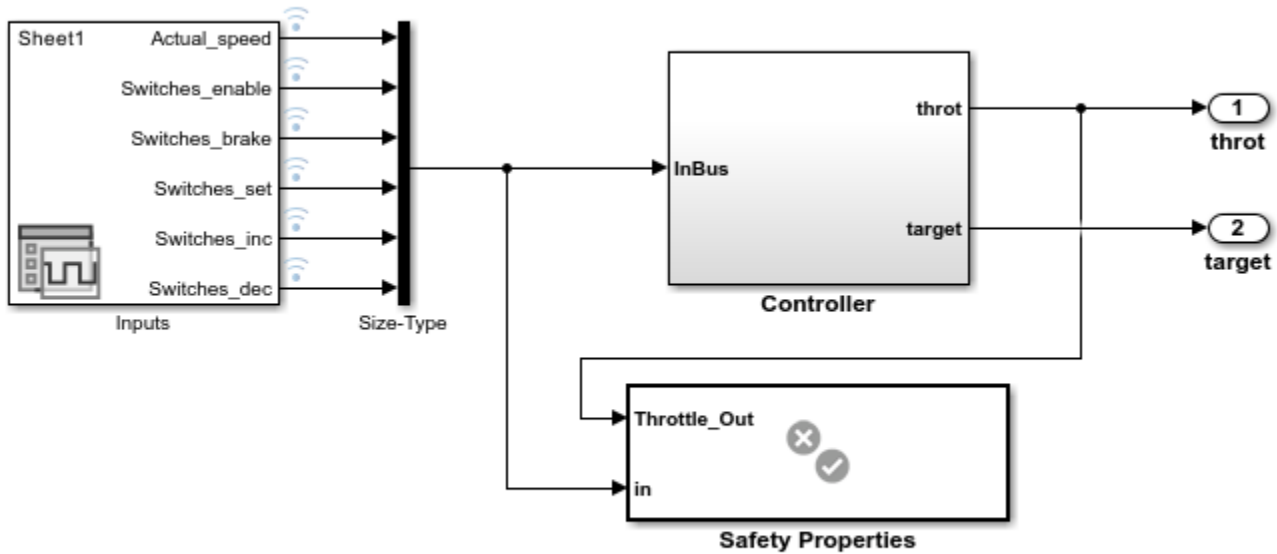
Simulink Test: Basic Cruise Control



Copyright 2006-2022 The MathWorks, Inc.

The test model contains a Signal Editor 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

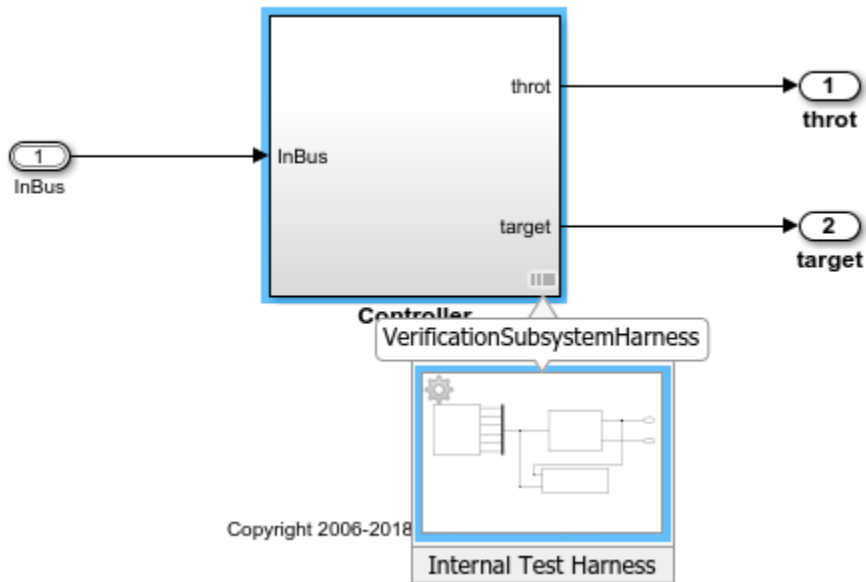


Copyright 2006-2022 The MathWorks, Inc.

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 2-54

Synchronize Changes Between Test Harness and Model

In this section...

- “Set Synchronization for a New Test Harness” on page 2-54
- “Change Synchronization of an Existing Test Harness” on page 2-57
- “Synchronize Configuration Set and Model Workspace Data” on page 2-57
- “Check for Unsynchronized Component Differences” on page 2-57
- “Rebuild a Test Harness” on page 2-58
- “Push Changes from Test Harness to Model” on page 2-58
- “Check Component and Push Parameter to Main Model” on page 2-58

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:

- Component under test (CUT) — CUT block and blocks, signals, and other entities in the CUT.
- Block parameters — Values of parameters associated with the block. To see the parameters of a block, right-click on the block and select **Block Parameters** from the context menu.
- Optionally, the active configuration set of the model or test harness. See “Manage Configuration Sets for a Model” for information on configuration sets

To synchronize the active configuration set and model workspace parameters between the test harness and main model, select **Update Configuration Parameters and Model Workspace data on rebuild** in the **Advanced Properties** tab of the Create Test Harness dialog box.

You do not need to synchronize base workspace data because it is available to both the test harness and main model. Subsystem model test harnesses always sync with their underlying models.

Set Synchronization for a New Test Harness

When you create 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 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 (CUT) in your harness, you can select from different synchronization types. These options are available in the Create Test Harness dialog box or by using the `SynchronizationMode` property of `sltest.harness.create`.

For all synchronization types, you can simulate the main model even if a test harness is open. You can also create harnesses for model components other than the current component under test and its nested subsystems. You cannot, however, have more than one harness open at time, so a newly created harness does not open automatically.

The locking information in the table indicates whether you can change the model, harness, or CUT in the model or harness when the test harness is open.

Synchronization Type	Description	Availability	Model, CUT, and Harness Locking When Harness Is Open
Synchronize on harness open and close	When the test harness opens, the test harness components and parameters synchronize from the model to the test harness. When the test harness closes, the same elements synchronize from the harness to the model.	Available for: <ul style="list-style-type: none"> • Subsystems, including Stateflow charts and MATLAB Function blocks • For Subsystem Reference blocks, only the block parameters are synchronized • Model blocks • S-function blocks Not available for: <ul style="list-style-type: none"> • Block diagrams • SIL/PIL harnesses • Subsystem model harnesses 	The main model and harness are unlocked for all types of CUTs. Subsystem CUTs in the model are locked. Subsystem CUTs in the harness are unlocked,
Synchronize on harness open	When the harness opens, the harness components and parameters synchronize from the model to the test harness.	Available for: <ul style="list-style-type: none"> • Block diagrams • Subsystems, including Stateflow charts and MATLAB Function blocks • For Subsystem Reference blocks, only the block parameters are synchronized • Model reference blocks • S-function blocks Not available for: <ul style="list-style-type: none"> • SIL/PIL harnesses • Subsystem model harnesses 	The main model and harness are unlocked for all types of CUTs. Subsystem CUTs in the model and the harness are locked.

Synchronization Type	Description	Availability	Model, CUT, and Harness Locking When Harness Is Open
<p>Synchronize only during push and rebuild</p>	<p>Synchronizes when you click Push Changes or Rebuild Harness. Push synchronizes changes from the test harness to the model. Rebuild synchronizes changes from the model to the test harness.</p>	<p>Available for:</p> <ul style="list-style-type: none"> • Subsystems, including Stateflow charts and MATLAB Function blocks • Model reference blocks • S-function blocks • Subsystem models, which always synchronize on the push and rebuild only. <p>Not available for:</p> <ul style="list-style-type: none"> • Block diagrams • SIL/PIL harnesses • Components in libraries 	<p>The main model, harness, and all types of CUTs in the model and harness, including subsystems, are unlocked.</p>
<p>Synchronize only during rebuild</p>	<p>Synchronizes only when you click Rebuild Harness. Changes synchronize from the model to the test harness.</p>	<p>Available for:</p> <ul style="list-style-type: none"> • Block diagrams • Model reference blocks • SIL/PIL verification mode components <p>Not available for:</p> <ul style="list-style-type: none"> • Subsystems, including Stateflow charts and MATLAB Function blocks • S-function blocks • Components in libraries 	<p>The main model, harness, and all types of CUTs in the model are unlocked. All types of CUTs in the harness are unlocked, except SIL/PIL verification mode components, which are locked and masked.</p>

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 keeps the generated code referenced by the SIL/PIL block as a reflection of 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 active 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 **Check Harness** 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 block 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 **Rebuild Harness**. 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. If you specified to use existing generated code for a SIL/PIL subsystem using `sltest.harness.create` or `sltest.harness.set`, the harness rebuild uses that code instead of regenerating it.

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

Push Changes from Test Harness to Model

After changing your system in the test harness, you can push changes to the main model. In the test harness, select **Push Changes**. This process overwrites the component in the main model.

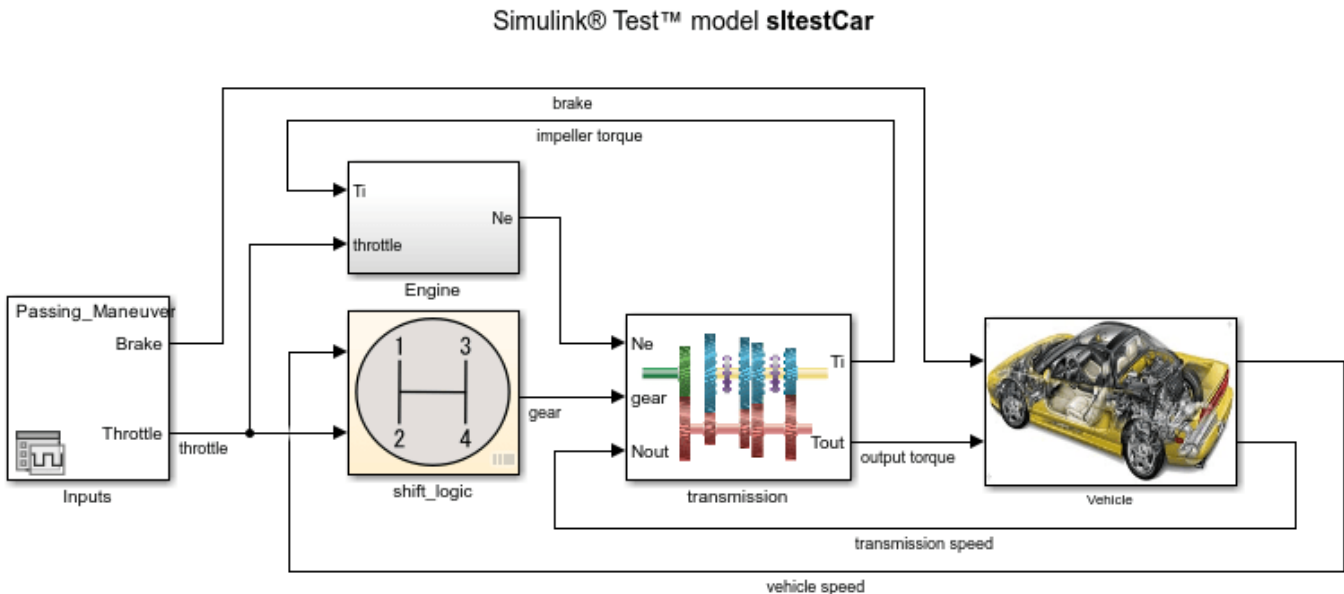
Check Component and Push Parameter to Main Model

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

This example also includes programmatic steps.

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

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

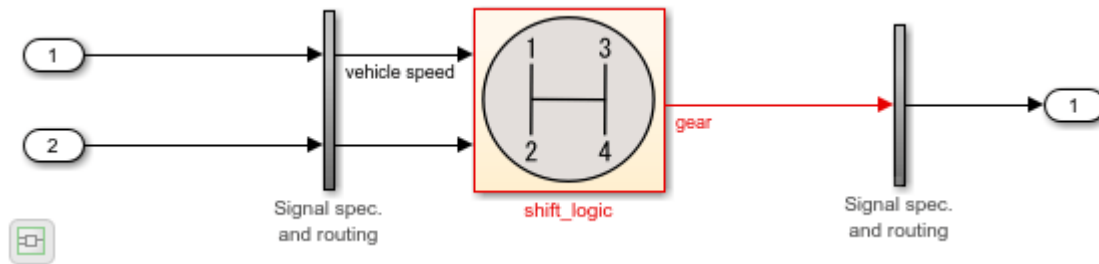


Update the Mask Parameter in the Test Harness

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

To open the test harness from the command line, use:

```
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**.

To change the value from the command line, use:

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

Update the Parameter to the Main Model

1. In the test harness, on the **Harness** tab, click **Push Changes**.
2. In the main model, double-click the shift_logic subsystem. The parameter value is updated.

To push the change using the command line, use:

```
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 model are s

close_system('sltestCar',0);

See Also

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

Related Examples

- “Test Harness and Model Relationship” on page 2-2
- “SIL Verification for a Subsystem” on page 5-2

Test Library Blocks

In this section...

“Library Testing Workflow” on page 2-62

“Library and Linked Subsystem Test Harnesses” on page 2-62

“Edit Library Block from a Test Harness” on page 2-63

“Testing a Library and a Linked Block” on page 2-63

“SIL Testing a Reusable Library Subsystem” on page 2-68

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. You can also test software-in-the-loop (SIL) code generated for a reusable library subsystem. First, create test harnesses for a library block to test your design. Once the library block meets your requirements, create test harnesses for linked blocks and test the subsystem instances. You can move test harnesses from the library to an instance and an instance to the library.

Library Testing Workflow

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

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

Library and Linked Subsystem Test Harnesses


A test harness for a library subsystem has specific properties:

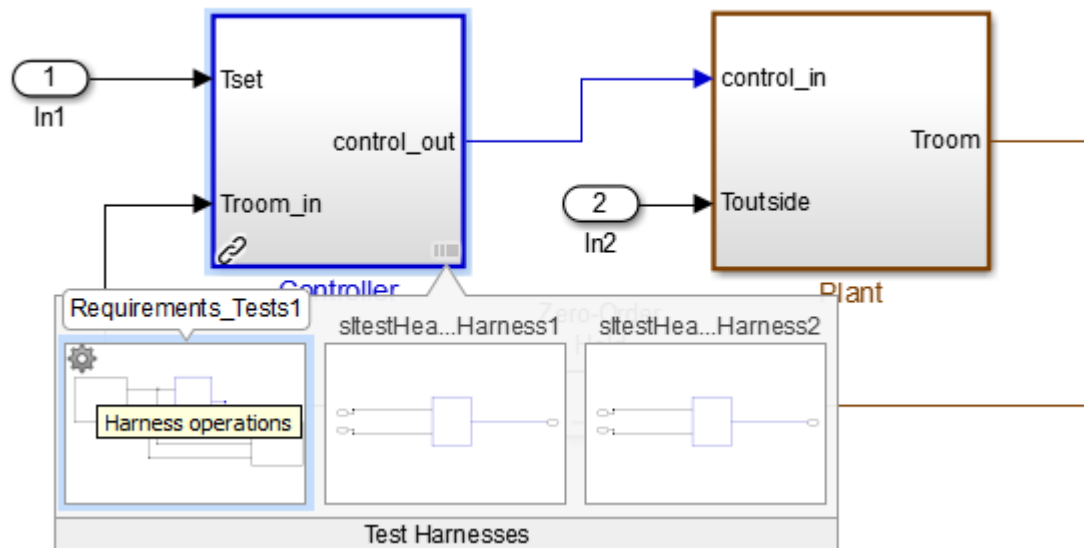
- Libraries do not compile, so a test harness for a library subsystem does not use compiled attributes such as data type or sample rate.
- A test harness for a library subsystem does not generate conversion subsystems for the block inputs and outputs.
- A library subsystem test harness does not use push or rebuild operations, because libraries do not use configuration parameters.

When you create a linked subsystem from a library subsystem, test harnesses copy to the linked instance. If you do not need the test harnesses, you can delete them. For instructions on deleting all test harnesses from a model, see “Manage Test Harnesses” on page 2-31.

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 or Import Test Harnesses and Select Properties” on page 2-13.

Testing a Library and a Linked Block

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

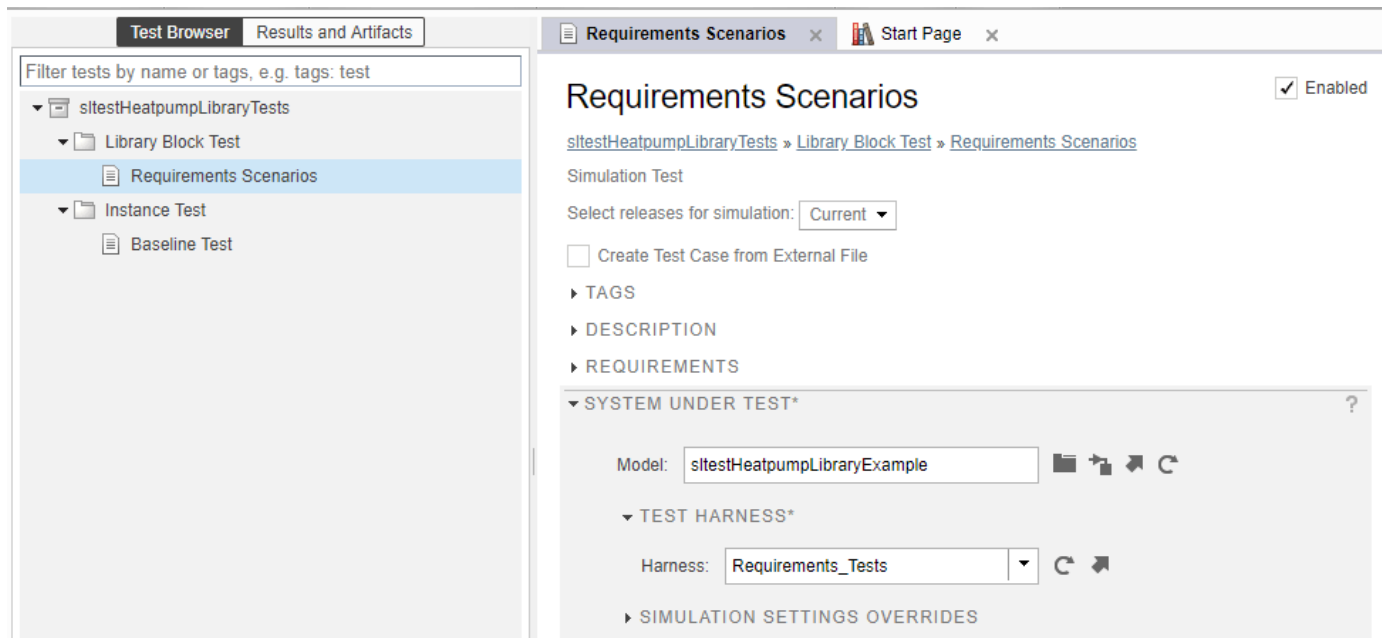
This example demonstrates a test case that confirms a library block meets a short set of requirements. After testing the library block, you execute a baseline test of a linked block and capture the baseline results. You then promote the baseline test harness to the library.

The library block controls a simple heat pump system by supplying on/off signals to a fan and compressor, and specifying the heat pump 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.

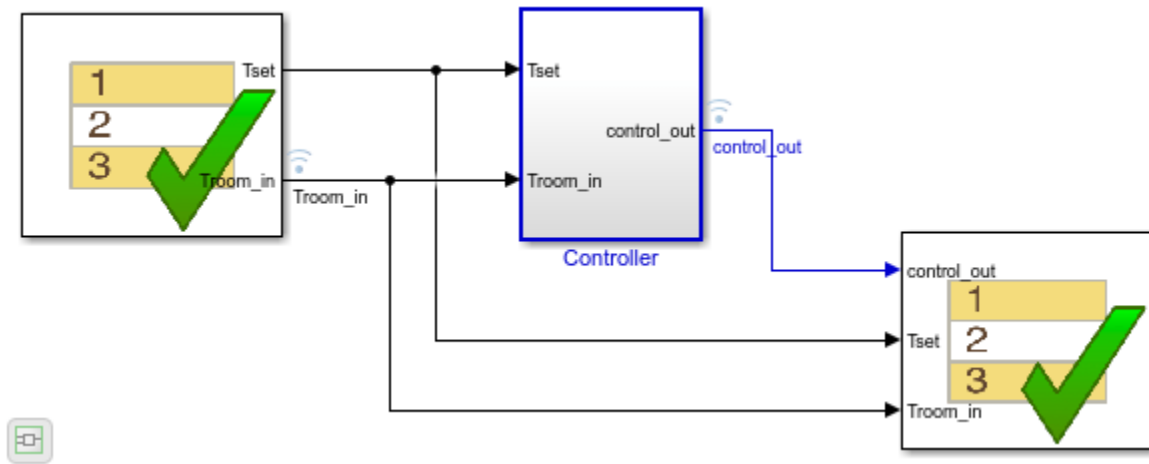
```
testFile = 'sltestHeatpumpLibraryTests.mldatx';
library = 'sltestHeatpumpLibraryExample';
system = 'sltestHeatpumpLibraryLinkExample';
sltest.testmanager.load(testFile);
sltest.testmanager.view;
```



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

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





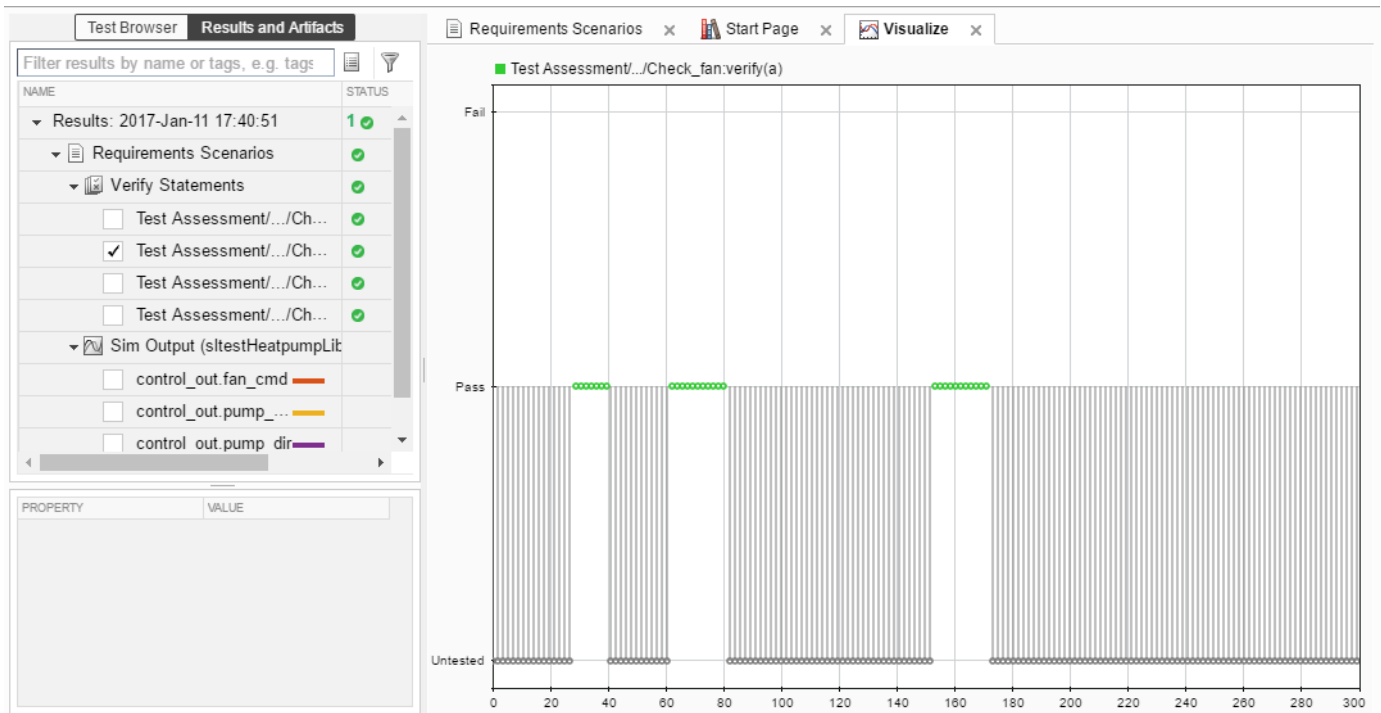
The Test Sequence block sets three scenarios for the controller:

- The controller at idle
- The controller activating the fan only
- The controller activating the heating 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

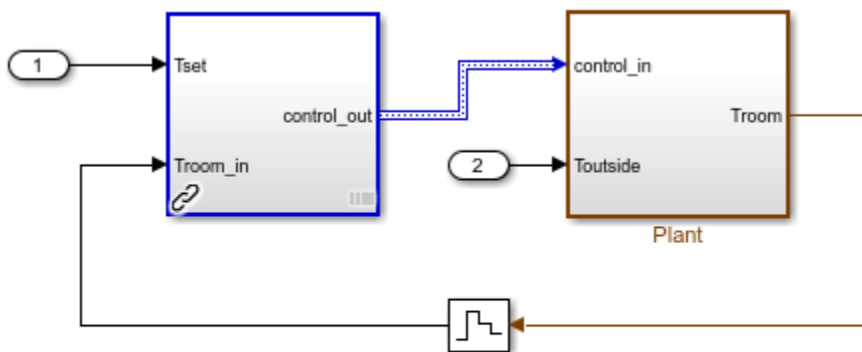
In the test manager, run the Requirements Scenarios test case. The verify statement results show that the control_out signals pass.



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.

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



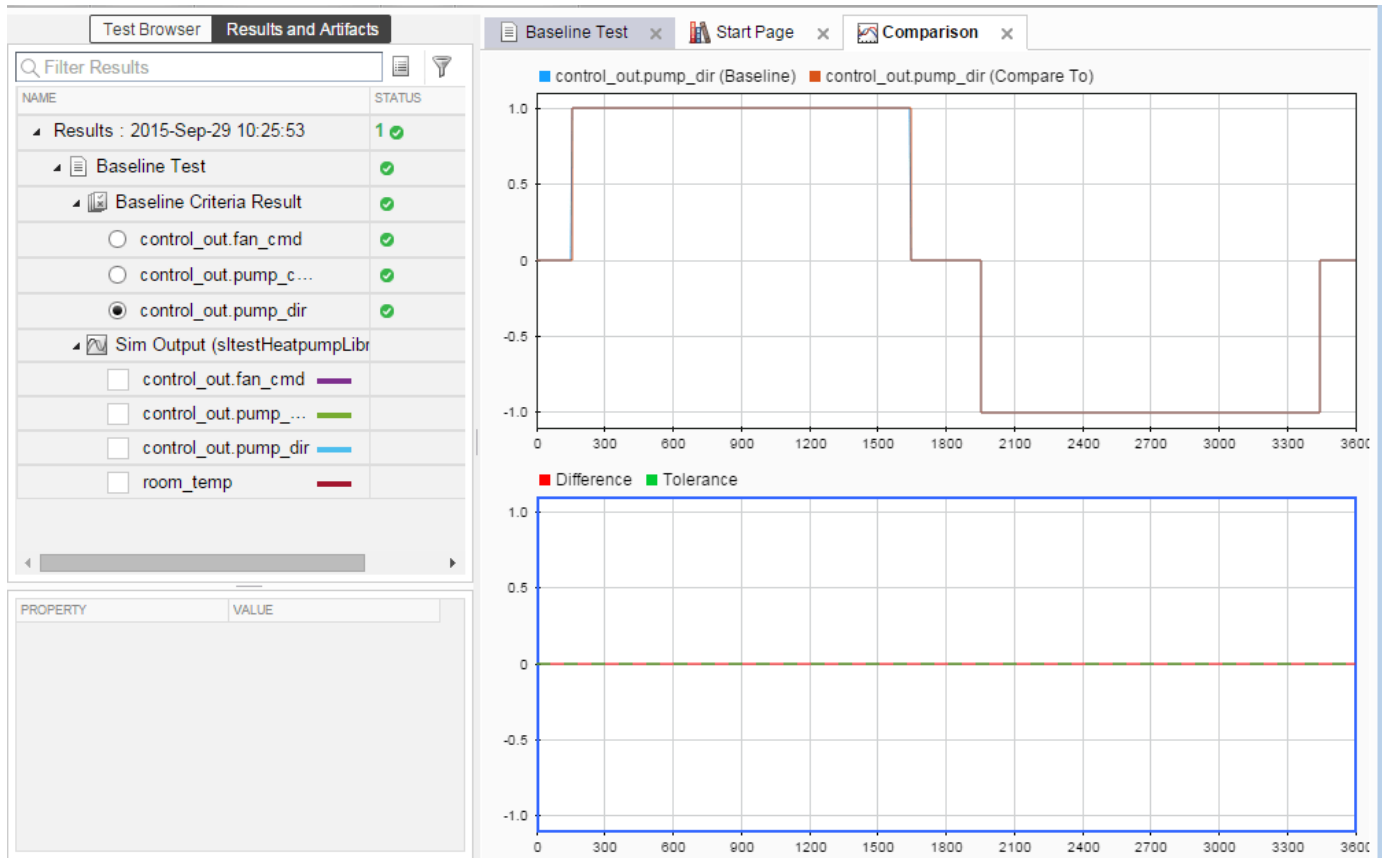
Copyright 1990-2014 The MathWorks, Inc.

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:

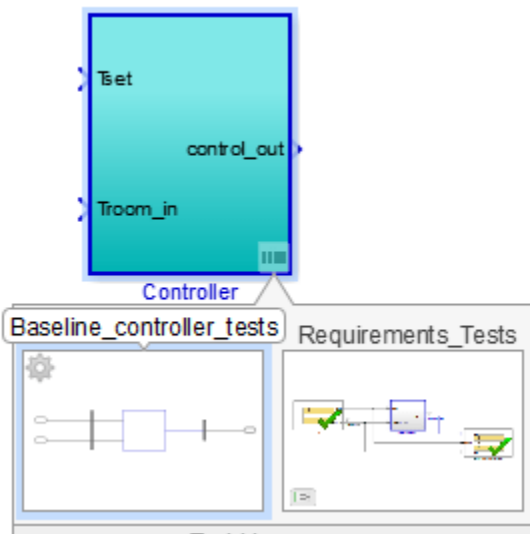
1. In the `sltestHeatpumpLibraryLinkExample` model, click the harness badge and hover over the **Baseline_controller_tests** test harness.

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

SIL Testing a Reusable Library Subsystem

This example shows how to unit test a reusable component in a library. It tests software-in-the-loop (SIL) code generated for a subsystem by using an equivalence test.

The reusable library subsystem must be at the top level of the library and must have function interfaces to lock down the subsystem interface. For information on reusable subsystem libraries, function interfaces, and workflow limitations, see “Library-Based Code Generation for Reusable Library Subsystems” (Embedded Coder).

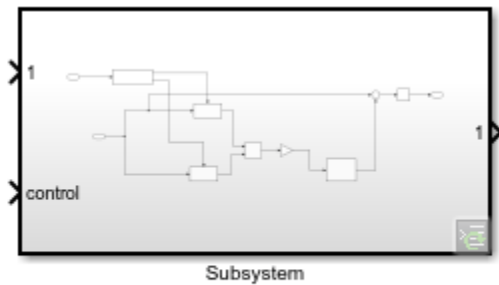
Set Up the Code Generation Environment

```
orig = Simulink.fileGenControl('get','CodeGenFolderStructure');
Simulink.fileGenControl('set','CodeGenFolderStructure',...
    Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);
```

Open the Reusable Library

This library contains a subsystem block at the top level.

ReuseLibSubsysExample



Copyright 2019 The MathWorks, Inc.

Build the Library

Build the reusable library to generate code and create the function interfaces. After the code generation completes, you can view the function interfaces by clicking the lower right corner the library to open the Manage Function Interfaces dialog box.

```
slbuild('ReuseLibSubsystemExample');

### Starting build procedure for: Double
### Generating code and artifacts to 'Target environment subfolder' folder structure
### Generating code into build folder: C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest-ex98874249
### Invoking Target Language Compiler on Double.rtw
### Using System Target File: B:\matlab\rtw\c\ert\ert.tlc
### Loading TLC function libraries
.....
### Initial pass through model to cache user defined code
.
### Caching model source code
.....
### Writing header file Subsystem_KFja4Edz.c
.
### Writing header file Double_types.h
### Writing header file Double.h
### Writing header file rtwtypes.h
### Writing header file Subsystem_KFja4Edz.h
### Writing source file Double.c
.
### Writing header file Double_private.h
### Writing source file ert_main.c
### TLC code generation complete (took 4.052s).
### Saving binary information cache.
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest-ex98874249\IntelWin64\_shared'
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest-ex98874249\IntelWin64\_shared'
### Successful completion of code generation for: Double
```

The following files will be copied from IntelWin64_shared to C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest-ex98874249

```
Subsystem_KFja4Edz.c
Subsystem_KFja4Edz.h
shared_file.dmr
```

```

Files copied from IntelWin64\_shared to C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest
### Starting build procedure for: Single
### Generating code and artifacts to 'Target environment subfolder' folder structure
### Generating code into build folder: C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest
### Invoking Target Language Compiler on Single.rtw
### Using System Target File: B:\matlab\rtw\c\vert\ert.tlc
### Loading TLC function libraries
.....
### Initial pass through model to cache user defined code
.
### Caching model source code
.....
### Writing header file Subsystem_qEbBEFCF.c
### Writing header file Single_types.h
.
### Writing header file Single.h
### Writing header file Subsystem_qEbBEFCF.h
### Writing source file Single.c
### Writing header file Single_private.h
.
### Writing source file ert_main.c
### TLC code generation complete (took 2.979s).
### Saving binary information cache.
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest-ex98874249\IntelWin64\_shared'
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest-ex98874249\IntelWin64\_shared'
### Successful completion of code generation for: Single

```

The following files will be copied from IntelWin64_shared to C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest

```

Subsystem_qEbBEFCF.c
Subsystem_qEbBEFCF.h
shared_file.dmr

```

Files copied from IntelWin64_shared to C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp3bfa7070\simulinktest

Select the Subsystem Component and Open the Test Manager

Click on the Subsystem in the library model to select it. Then, open the Test Manager.

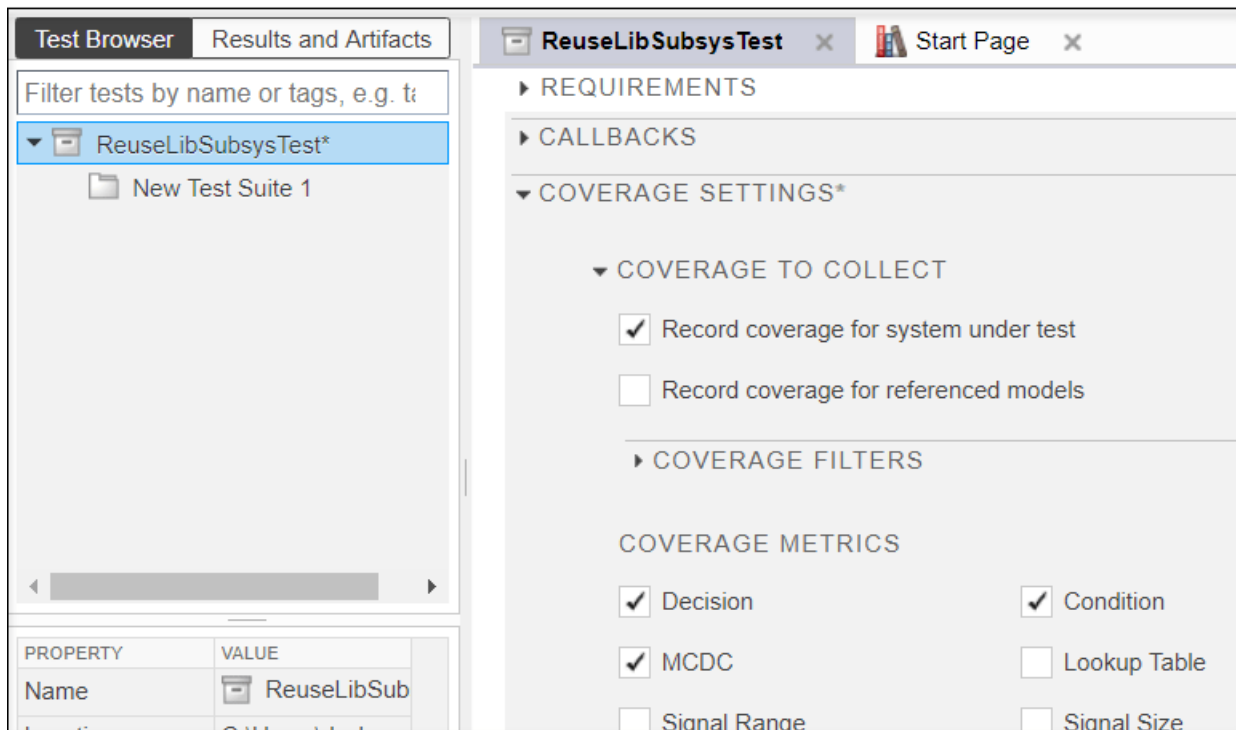
```
sltestmgr
```

Open the Test File

- 1 In the Test Manager, click **Open** and select the `ReuseLibSubsysTest.mldatx` file.

Enable Coverage Collection

- 1 Select **ReuseLibSubsysTest** in the Test Browser pane.
- 2 Then, expand the Coverage Settings section in the main pane.
- 3 In Coverage to Collect, check **Record coverage for system under test**.
- 4 In Coverage Metrics, check that **Decision**, **MCDC**, and **Condition** are selected.






Open the Create Test for Model Component wizard


- 1 Click the arrow under **New** and select **Test for Model Component** to open the Create Test for Model Component wizard.
- 2 On the first page of the wizard, click the Use currently selected model component icon next to the **Component** field. Both the **Component** and **Top Model** fields fill in.
- 3 Click the refresh icon next to the Select function interface field and select Double from the dropdown.
- 4 Click Next.

Create Test for Model Component



System › Test Inputs › Verification Strategy › Generated Test

What is your Component Under Test (CUT)?

Top Model:   

Component: 

Function Interface Settings:

Select a function interface:  

[Click here to know more about function interfaces](#)

Select the Test Inputs

- 1 Select **Use Design Verifier to generate test input scenarios**.
- 2 Click Next.

System › Test Inputs › Verification Strategy › Generated Test

How do you want to setup the inputs?

Use Design Verifier to generate test input scenarios
Create inputs using Simulink Design Verifier. [Design Verifier Settings](#)

Simulate top model and use the recorded component inputs in the analysis

Specify inputs in the created harness
Create a new test harness for component. Inputs should be added to the harness

Select How to Test the Component

- 1 Select **Perform back-to-back testing**.
- 2 Set **Simulation1** to Normal mode.

- 3 Set **Simulation2** to Software-in-the-Loop (SIL) mode.
- 4 Click Next.

[System](#) › [Test Inputs](#) › [Verification Strategy](#) › [Generated Test](#)

How do you want to test the component?

Use component under test output as baseline
Simulate the top model and record the outputs of the component to be used as baseline

Perform back-to-back testing
Set up a test to compare the component under test outputs in different simulation modes

Select simulation modes:

Simulation1: ▼

Simulation2: ▼

Set Model coverage objectives as Enhanced MCDC

Define the verification logic in the created harness
No verification logic will be automatically added to the test

Specify the Input Source, Data File Format, and Test File

- 1 Select Inports as the test harness input source.
- 2 Select MAT as the file format.
- 3 Select **Add tests to currently selected test file**.

[System](#) › [Test Inputs](#) › [Verification Strategy](#) › [Generated Test](#)

How do you want to save the test data?

Select test harness input source: Specify the file format:

Specify location to save test data:

Where do you want to save the generated test(s)?

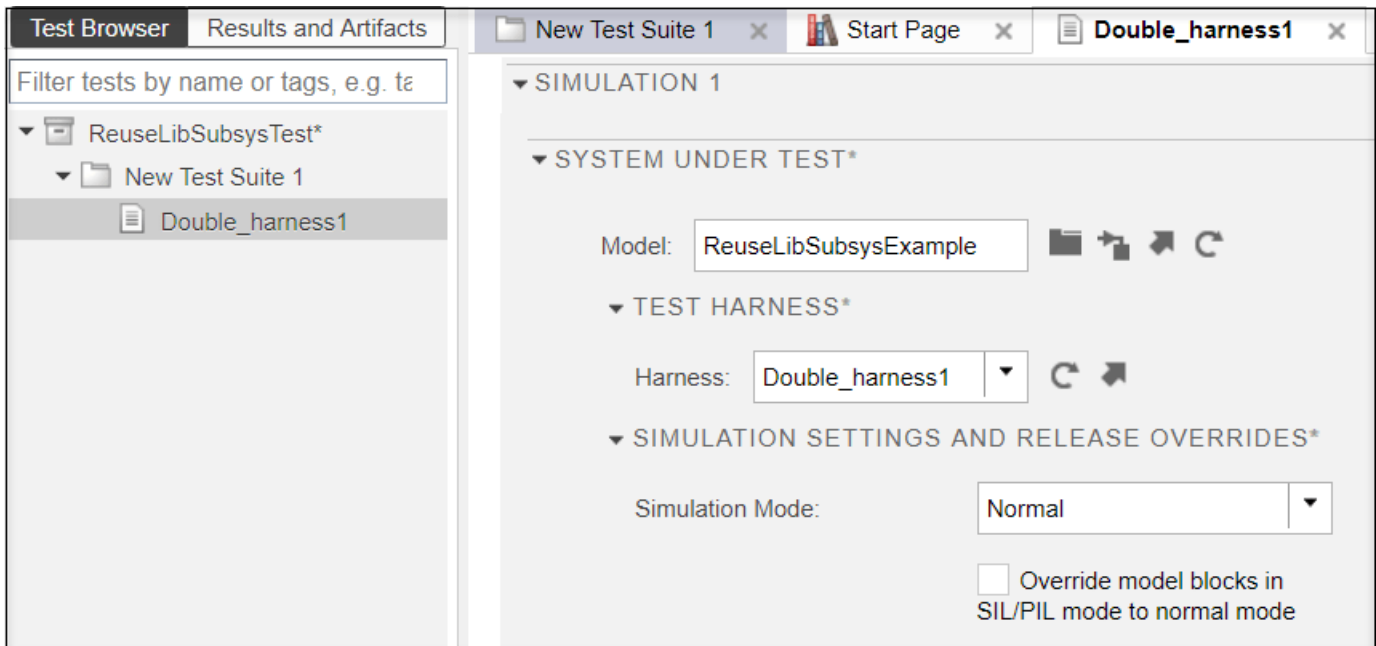
Add tests to the currently selected test file.
 Create a new test file containing the test(s).

Test File Location:

Generate the Test Case and Return to the Test Manager

Click Done to generate the Double_harness1 test case for the Double function interface.

After test case generation completes, the Double_harness1 test case appears in the Test Manager. Notice that the **Simulation Mode** for Simulation1 is set to Normal and Simulation2 is set to Software-in-the-Loop (SIL) mode.

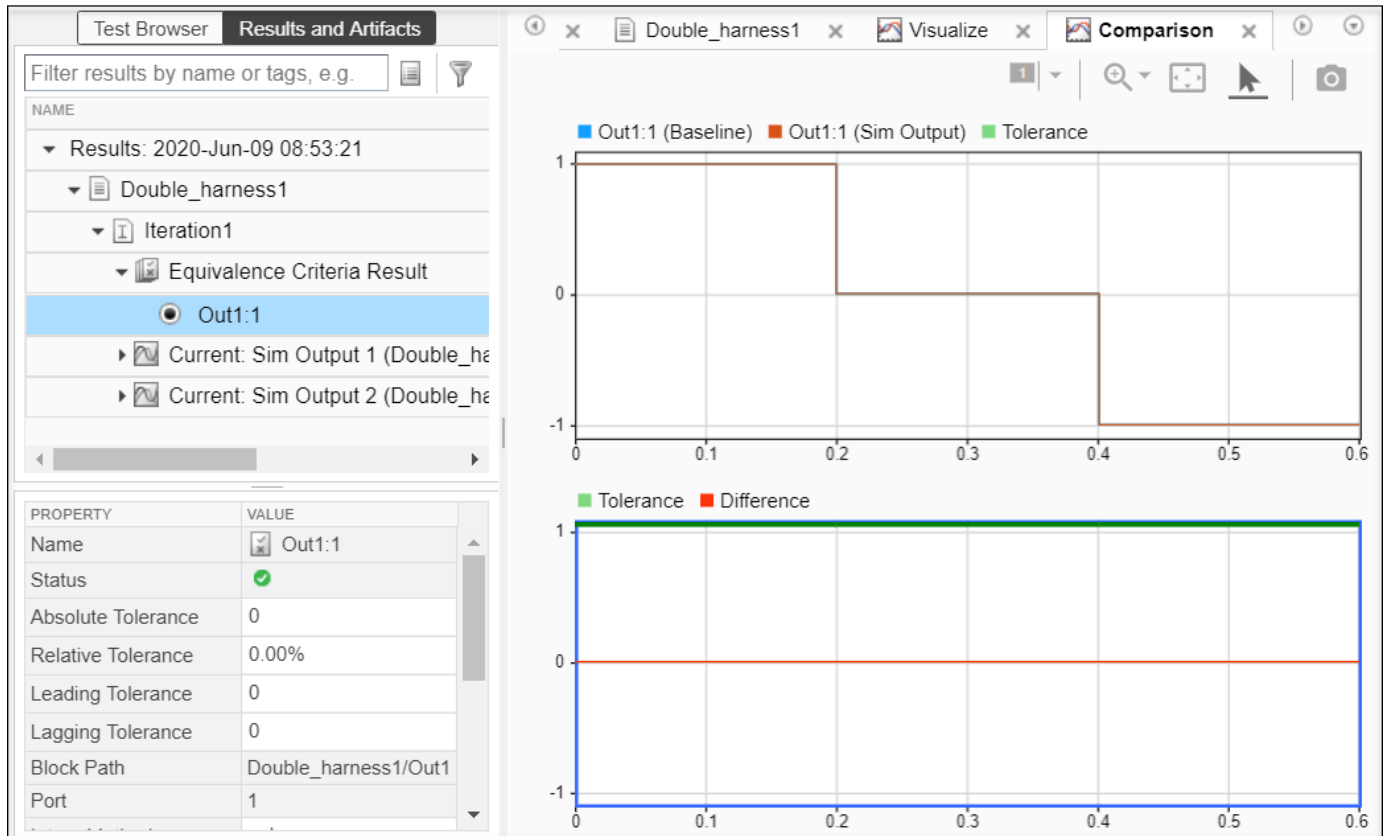


Run the Test

Click Run.

View Equivalence Results

Expand the Results to view the Equivalence Criteria Result. Notice that the Differences plot shows no differences between the two signals, which indicates that the Normal and SIL simulations are producing the same results. The SIL code associated with the reusable library subsystem can be reused for other SIL tests.



View Coverage Results

Expand the Aggregated Coverage section to view the coverage results. The two coverage results that show 100% are for the equivalence test run for the Double function interface. The other two results show 0% or no coverage because the Single function interface was not tested.

▼ AGGREGATED COVERAGE RESULTS ?

Create a coverage report from coverage results to justify or exclude missing coverage. The filters and updated coverage values will be displayed with this result.

ANALYZED MODEL	REPORT	SIM MO...	COMPLEXI...	DECISION	EXECUTION	FUNCTION
ReuseLibSubsysExample/Subsystem	🔍	Normal	4	100% █	100% █	--
ReuseLibSubsysExample/Subsystem	🔍	Normal	0	--	--	--
Subsystem_7AqDCZ2Q.c	🔍	SIL	6	0% █	0% █	0% █
Subsystem_FrFkRb4O.c	🔍	SIL	6	100% █	100% █	100% █

+ Add Tests for Missing Coverage - Export

Clean up

```
Simulink.fileGenControl('set','CodeGenFolderStructure',orig);
```

See Also**More About**

- “Library-Based Code Generation for Reusable Library Subsystems” (Embedded Coder)
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Embedded Coder)
- “Create or Import Test Harnesses and Select Properties” on page 2-13
- “Import Test Cases for Equivalence Testing” on page 5-19
- “Generate Tests and Test Harnesses for a Model or Components” on page 6-24

Test Sequences and Assessments

- “Test Sequence Basics” on page 3-2
- “Use Stateflow Chart for Test Harness Inputs and Scheduling” on page 3-8
- “Assess Simulation and Compare Output Data” on page 3-14
- “Assess Model Simulation Using verify Statements” on page 3-18
- “Verify Multiple Conditions at a Time” on page 3-23
- “Assess a Model by Using When Decomposition” on page 3-25
- “Test Sequence Editor” on page 3-30
- “Transitions, Temporal Operators, and Messages in Test Sequence Blocks” on page 3-37
- “Generate Test Signals” on page 3-44
- “Using an External Function in a Test Sequence Block” on page 3-49
- “Programmatically Create a Test Sequence” on page 3-52
- “Programmatically Create and Run Test Sequence Scenarios” on page 3-56
- “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59
- “Scenario Parameter Section” on page 3-67
- “Test Sequence and Assessment Syntax” on page 3-68
- “Debug a Test Sequence” on page 3-75
- “Test Downshift Points of a Transmission Controller” on page 3-78
- “Examine Model Verification Results by Using Simulation Data Inspector” on page 3-83
- “Fix Requirements-Based Testing Issues” on page 3-87
- “Assess Temporal Logic by Using Temporal Assessments” on page 3-93
- “Test Traffic Light Control by Using Logical and Temporal Assessments” on page 3-99
- “Logical and Temporal Assessment Syntax” on page 3-107

Test Sequence Basics

In this section...

“Test Sequence Hierarchy” on page 3-2

“Test Sequence Scenarios” 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

A test sequence consists of test steps arranged in a hierarchy. You can use a test sequence to define test inputs and to define how a test will progress in response to the simulation. A test step contains actions that execute at the beginning of the step. A test step can contain transitions that define when the step stops executing, and which test step executes next. Actions and transitions use MATLAB as the action language. You create test sequences by using the Test Sequence block and the Test Sequence Editor on page 3-30. See “Use Stateflow Chart for Test Harness Inputs and Scheduling” on page 3-8.

Test Sequence Hierarchy

Test sequences defined in Test Sequence blocks 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.

Test Sequence Scenarios

In a Test Sequence block, you can define multiple test sequences, which are called test sequence scenarios. By using scenarios, you can define distinct test sequences without having multiple Test Sequence blocks in your test harness. You can run test sequence scenarios in these ways:

- Activate a single scenario from the Test Sequence Editor and run the model
- Activate a single scenario using API commands and run the model
- Control the active scenario with a workspace variable and run the model
- Use a custom test script to loop through scenarios when running the model
- Define iterations in the Test Manager to run more than one scenario in a single test case

For more information and examples of using test sequence scenarios, see “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59 and “Programmatically Create and Run Test Sequence Scenarios” on page 3-56.

Transition Types

Test sequences defined in Test Sequence blocks 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. For a list of transitions, see “Test Sequence and Assessment Syntax” on page 3-68.

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

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

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

This When decomposition contains three verify statements. Each verify statement is active when the signal gear is equal to a different value. For more information, see “Assess a Model by Using When Decomposition” on page 3-25.

Symbols	Step	Transition	Next Step
Input 1. speed 2. throttle 3. gear Output Local Constant Parameter Data Store Memory	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 use a Test Sequence block to create a simple test sequence for a transmission shift logic controller.

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


```
openExample('TransmissionDownshiftTestSequence')
```
- 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**:
 - Select **Test Sequence** from the source drop-down menu.
 - Select **Add separate assessment block**.
 - Select **Open harness after creation**.
- 4 Click **OK**. The test harness for the `shift_controller` subsystem opens.
- 5 Double-click the Test Sequence block. The Test Sequence Editor opens.

Symbols	Step	Transition	Next Step	Description
Input 1. gear	Run <pre>%% Initialize data outputs. speed = 0; throttle = 0;</pre>			
Output 1. speed 2. throttle				

- 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
Input 1. gear	Accelerate speed = 10*ramp(et); throttle = 100;	1. duration(gear == 4) >= Limit	Stop ▼
Output 1. speed 2. throttle	Stop throttle = 0; speed = 0;		
Local			
Constant Limit			
Parameter			
Data Store Memory			

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

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

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

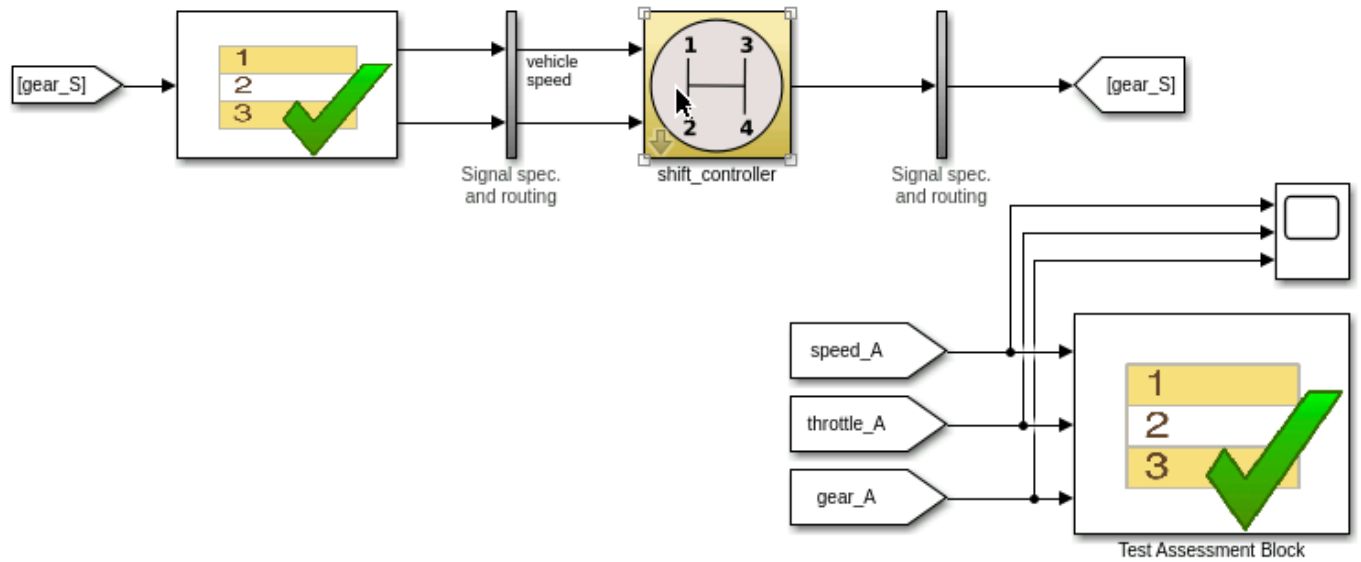
Check3rd when gear == 3
 verify(speed < 105)

Else

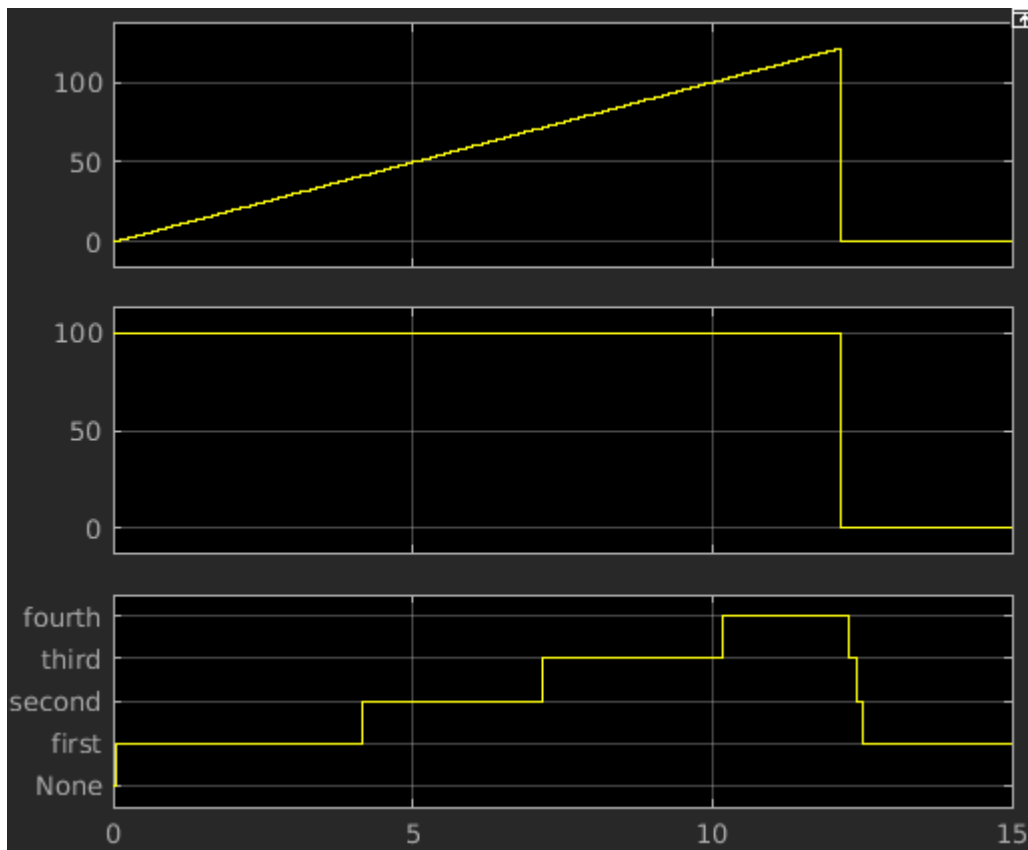
Symbols	Step	Transition	Next Step
Input 1. speed 2. throttle 3. gear	Assessments		
Output	Check1st when gear == 1 verify(speed < 45)		
Local	Check2nd when gear == 2 verify(speed < 75)		
Constant	Check3rd when gear == 3 verify(speed < 105)		
Parameter	Else		
Data Store Memory			

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

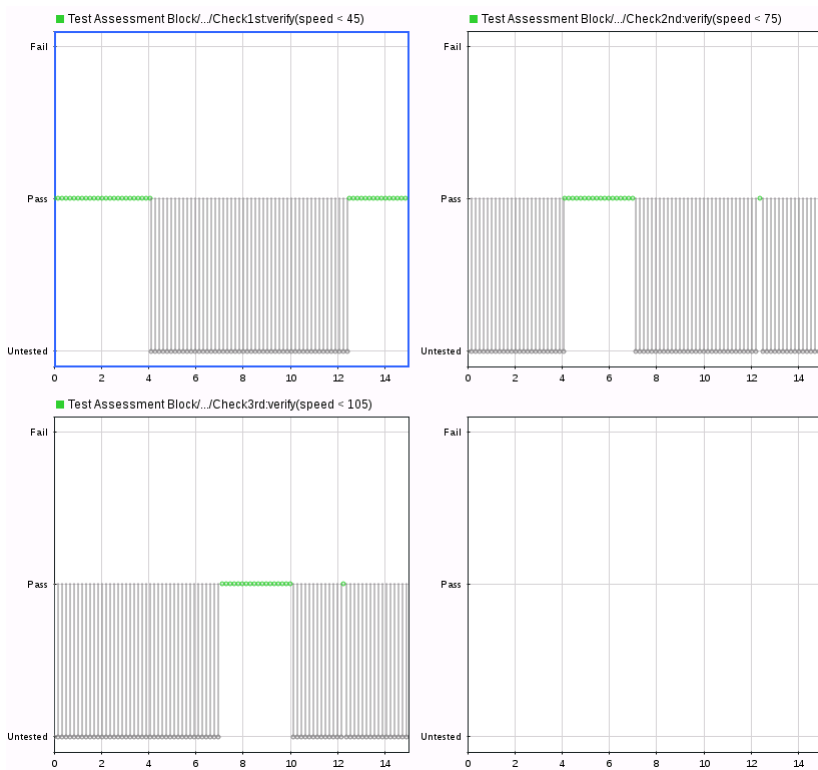
- Add a scope to the harness and connect the speed, throttle, and gear signals to the scope.



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



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



See Also

Test Sequence

More About

- "Test Sequence Editor" on page 3-30
- "Test Sequence and Assessment Syntax" on page 3-68
- "Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager" on page 3-59
- "Programmatically Create and Run Test Sequence Scenarios" on page 3-56
- "Assess a Model by Using When Decomposition" on page 3-25
- "Use Stateflow Chart for Test Harness Inputs and Scheduling" on page 3-8

Use Stateflow Chart for Test Harness Inputs and Scheduling

In this section...

“Use a Stateflow Chart for Test Harness Scheduling” on page 3-8

“Use a Stateflow Chart as a Test Harness Source” on page 3-9
--

“Stateflow Chart as Test Harness Scheduler and Source” on page 3-10

Use a Stateflow Chart for Test Harness Scheduling

You can define test harness scheduling using a Test Sequence block, a MATLAB Function block, or a Stateflow chart. If you use a Stateflow chart as a scheduler, you can use Stateflow features that are not available with either the Test Sequence block or MATLAB Function block. You can define more complicated scheduling by using Stateflow variants, graphical functions, super transitions, and super steps. For example, with Stateflow variants, you can specify multiple test scenarios in a single test harness. If you do not need to test multiple test scenarios or use complicated sequence logic, use the Test Sequence block, which has simpler syntax for test scheduling.

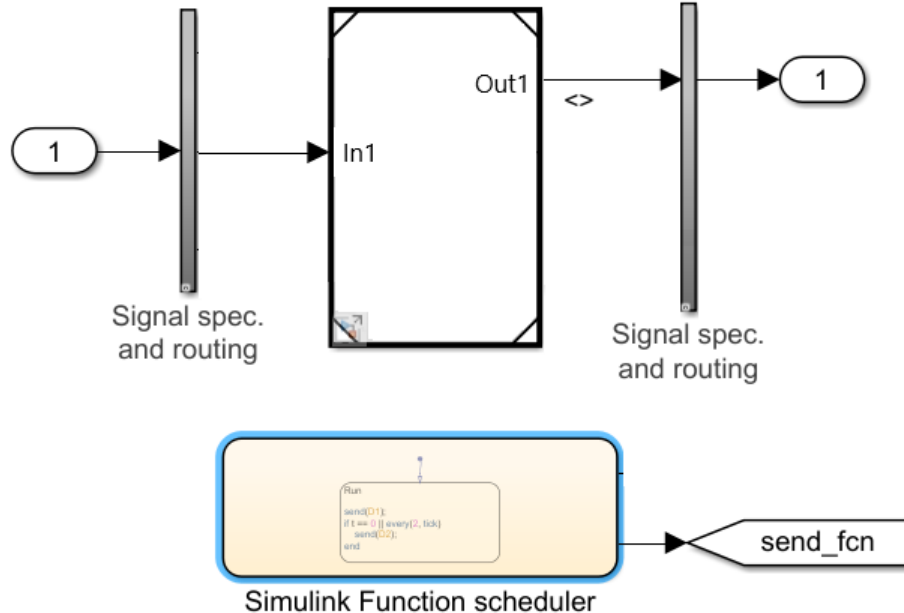
Note You must have a Stateflow license to use a chart for test harness inputs or scheduling.

To use a Stateflow chart as a test harness test scheduler, the model or subsystem under test must have at least one function call signal.

When setting up a test harness from a model, the steps for using a chart as the scheduler are:

- 1 In a model or subsystem, right-click and select **Test Harness > Create for Model** or **Create for <subsystem>**, respectively.
 - For a model, in the Create a Test Harness dialog box, set **Add scheduler for function-calls and rates** to Chart.
 - For a subsystem, in the Create a Test Harness dialog box, set **Generate function call signals** to Chart.

A chart named Simulink Function scheduler is added to the test harness.



- 2 Open the Stateflow chart and define the test sequence using Stateflow states, transitions and other objects. The Stateflow states serve the same purpose as the sequence steps in a Test Sequence block. The transitions define the criteria for moving from one state to another.

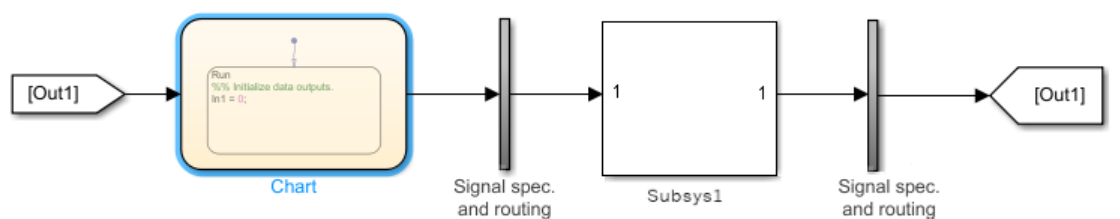
To programmatically specify a Stateflow chart as a scheduler, set the SchedulerBlock property of `sltest.harness.create` to Chart.

Use a Stateflow Chart as a Test Harness Source

When creating a test harness from a model, the steps for using a chart as the test harness source are:

- 1 In a model or subsystem, right-click and select **Test Harness > Create for Model** or **Create for <subsystem>**, respectively.
- 2 In the Create Test Harness dialog box, in the **Sources and Sinks** section, select Chart instead of Inport.

A chart is added to the test harness. For example,



- 3 Open the Stateflow chart and define the test harness sources using Stateflow logic.

To programmatically specify a Stateflow chart as a source, set the Source property of `sltest.harness.create` to Chart.

Stateflow Chart as Test Harness Scheduler and Source

This example shows how to use a single Stateflow chart as both a test scheduler and source in a test harness. The test harness for the `sltest_autosar_chart.slx` model in this example has already been created.

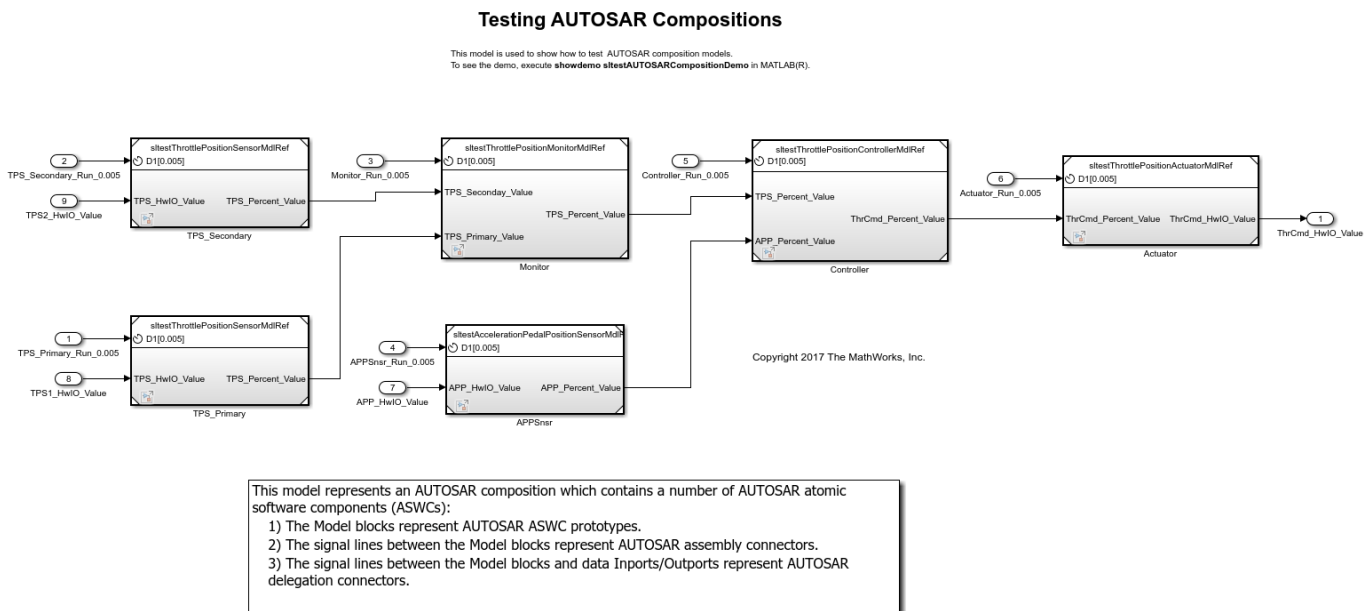
`sltest_autosar_chart` is an AUTOSAR composition model of a throttle position controller for an automobile. AUTOSAR composition models contain a network of interconnected Model blocks, each of which represents an atomic AUTOSAR software component (ASWC). The Simulink inports and outports represent AUTOSAR ports. The signal lines represent AUTOSAR component connectors.

The inputs that capture the primary and secondary throttle position are modeled using an external time series input and are directly fed through the Chart without modification. This modeling style is useful when some stimulus inputs can be modeled and others are only available as externally captured data.

Navigate to a directory with write permissions before running this example.

Open the Model

```
open_system('sltest_autosar_chart')
```

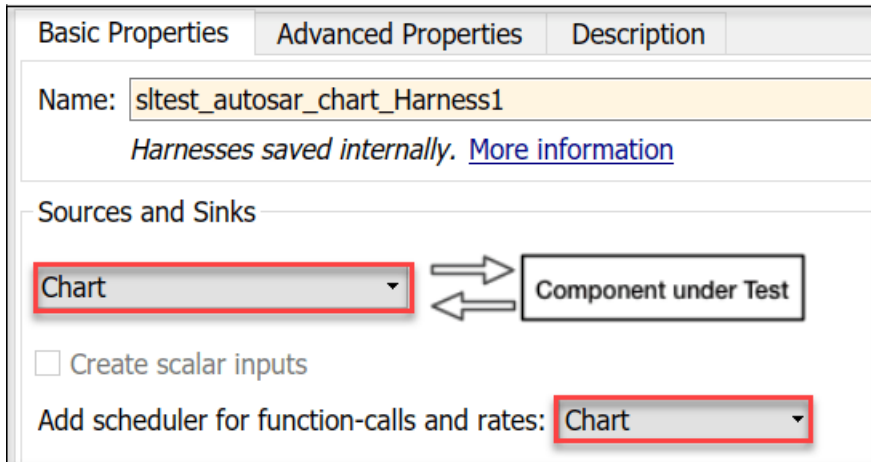


Copyright 2017 The MathWorks, Inc.

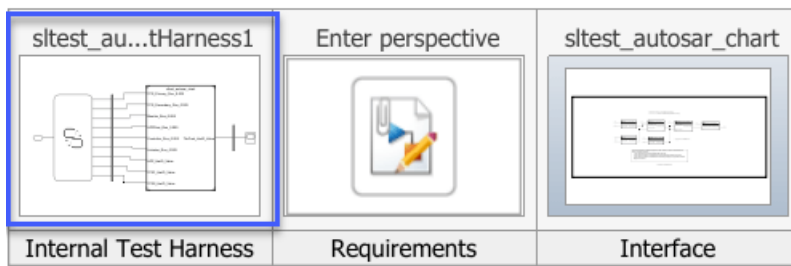
Open the Test Harness

The test harness has already been created for this example.

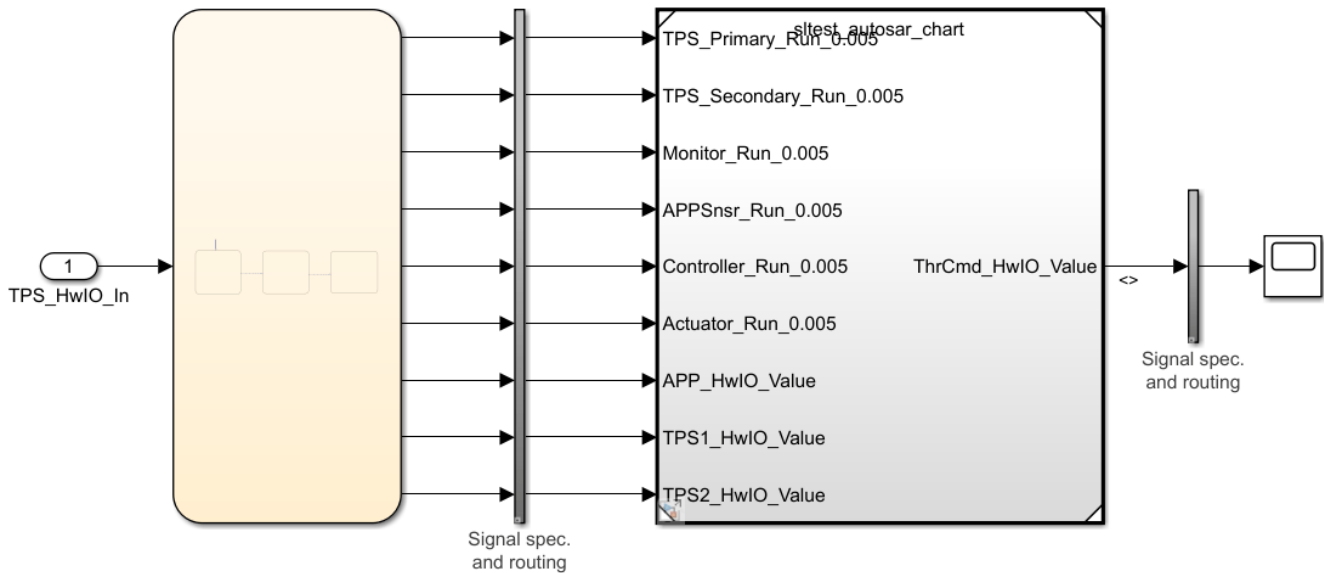
This image shows the portion of the Create Test Harness dialog where Chart was selected as both the source and scheduler. You do not need to recreate the test harness.



To open the harness, use the perspective control in the lower-right corner of the editor canvas and click **Internal Test Harness**.

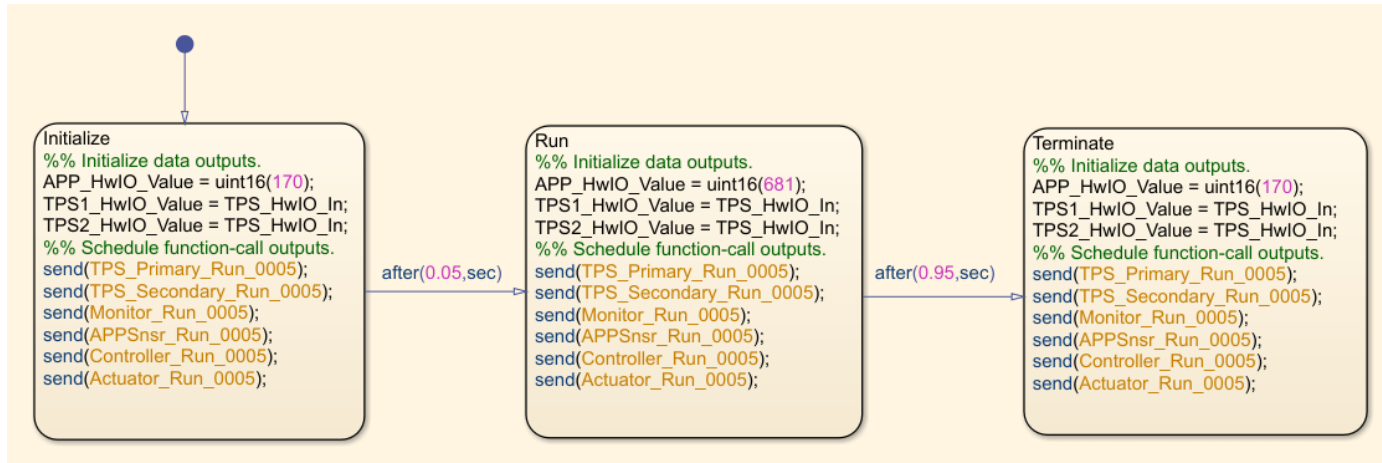


The test harness opens.



Open the Stateflow Chart

Double-click the chart in the test harness to view the scheduling logic.



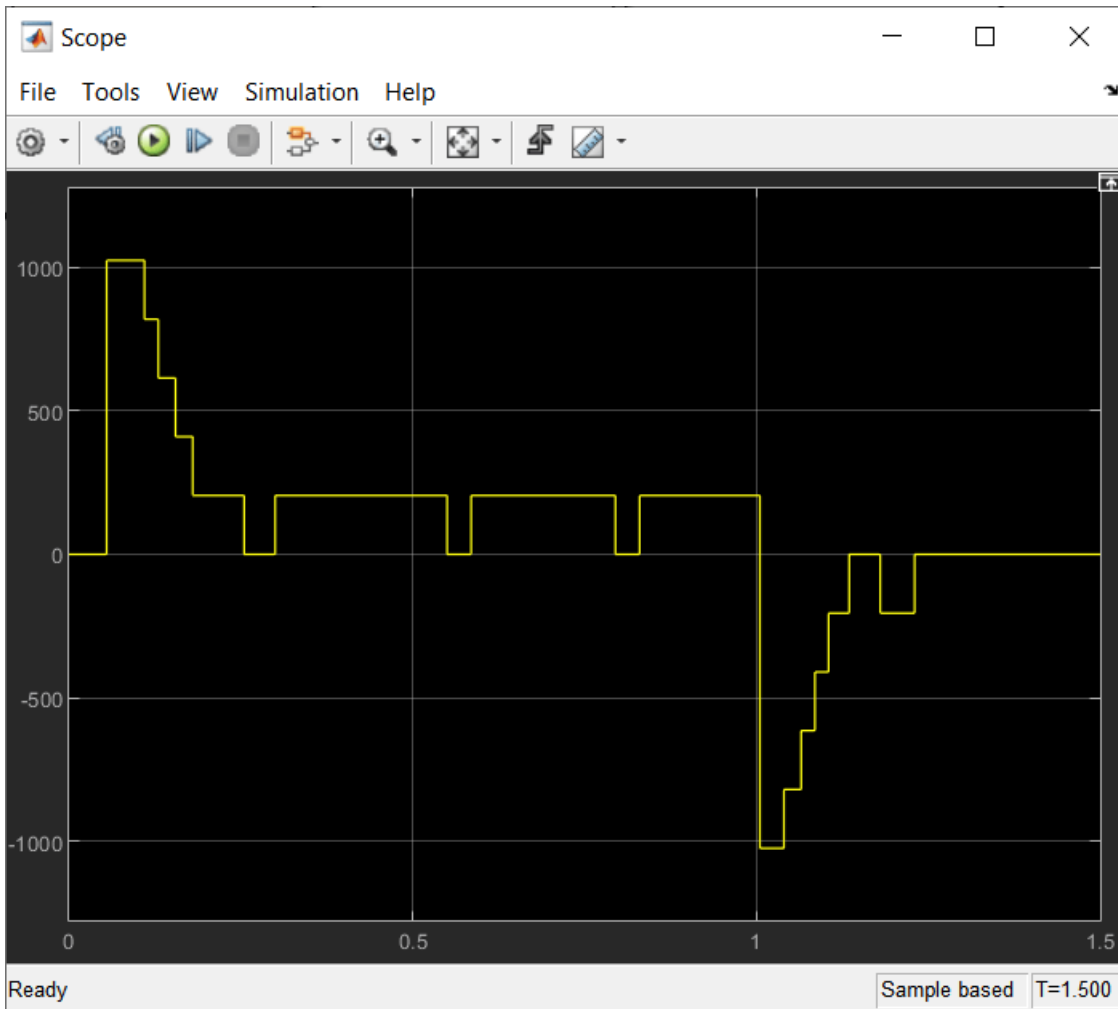
The component under test (the AUTOSAR model) requires the accelerator pedal position sensor input `APP_HwIO_Value`, which is modeled in the chart by three states.

The `Initialize` state sets the input to a nominal value (170) and the `Run` state models a steady acceleration command for 950 ms. The acceleration command is reset to the nominal value in the `Terminate` state.

The component under test uses the export-function modeling style. (See “Export-Function Models Overview”.) When the test harness was created, its Stateflow chart was configured to call each root-level Simulink Function block and send a trigger event to each function-call subsystem in the model. In this example, the code to send the trigger events is in each state after the stimulus waveforms have been generated.

Run the Model

Run the model from the test harness. To see the throttle command output, open the Scope in the test harness.



See Also

`sltest.harness.create` | Test Sequence | Function Caller

More About

- “Chart Programming” (Stateflow)
- “Transitions, Temporal Operators, and Messages in Test Sequence Blocks” on page 3-37
- “Create or Import Test Harnesses and Select Properties” on page 2-13
- “Test Sequence Basics” on page 3-2
- “Test Sequence Editor” on page 3-30

Assess Simulation and Compare Output Data

In this section...
“Overview” on page 3-14
“Compare Simulation Data to Baseline Data or Another Simulation” on page 3-15
“Post-Process Results With a Custom Script” on page 3-15
“Run-Time Assessments” on page 3-15
“Logical and Temporal Assessments” on page 3-17

Overview

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

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

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

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

- In a test case, you can:
 - Compare simulation output to baseline data.
 - Compare the output of two simulations.
 - Post-process simulation output using a custom script.
 - Assess temporal properties using logical and temporal assessments. If you have one or more defined assessments and their associated symbols in a test case, you can use the API or the Test Manager to obtain a list and information about them, copy them to another test case, and remove them from a test case. For information on using the API, see `sltest.testmanager.Assessment`, `sltest.testmanager.AssessmentSymbol`, and `sltest.testmanager.TestCase`. For the Test Manager, see “Assess Temporal Logic by Using Temporal Assessments” on page 3-93.
- In a test harness or model, you can:
 - Verify logical conditions in run-time using a `verify` statement, which returns a `pass`, `fail`, or `untested` result for each time step.
 - Use `assert` statements to stop simulation on a failure.
 - Use blocks from the Model Verification or Simulink Design Verifier library.

Compare Simulation Data to Baseline Data or Another Simulation

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

Set a numeric tolerance using absolute or relative tolerances. Set time tolerances using leading and lagging tolerances. For numeric data, you can specify absolute tolerance, relative tolerance, leading tolerance, or lagging tolerance. For enumerated or logical data, you can specify leading or lagging tolerance. Results outside the tolerances fail. For more information, see “Set Signal Tolerances” on page 6-153.

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-Process Results With a Custom Script

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

Write a custom criteria script in the Test Manager **Custom Criteria** section of the test case. Custom criteria results appear in the **Results and Artifacts** pane. Results are shown for individual MATLAB Unit Test qualifications. For more information, see “Process Test Results with Custom Scripts” on page 6-179.

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

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

Run-Time Assessments

verify Statements

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

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

case, results appear in the Simulation Data Inspector. With a test case, results appear in the Test Manager.

For information on using `verify` statements in your model, see “Assess Model Simulation Using `verify` Statements” on page 3-18.

assert Statements

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

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

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

Assessments for Real-Time Testing

If you are using a real-time test case, or if you want to reuse a desktop simulation test case on a real-time target, use `verify` statements. `verify` statements are built into the real-time application, and run on the real-time target. See “Assess Model Simulation Using `verify` Statements” on page 3-18.

Model Verification Blocks

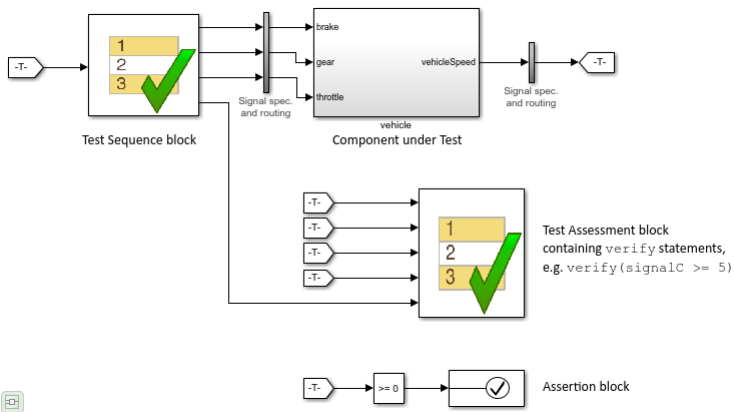
Use blocks from the Simulink “Model Verification” library or the Simulink Design Verifier library to assess signals in your model or test harness. `pass`, `fail`, or `untested` results from each block appear in the Test Manager. For more information, see “Examine Model Verification Results by Using Simulation Data Inspector” on page 3-83.

Note All Model Verification library blocks, including the Assertion block, do not produce verification results when used in For Each subsystems. Use a Test Sequence block with `verify` statements instead.

Examples of Run-Time Assessments

This example test harness includes:

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



Logical and Temporal Assessments

Logical and temporal assessments evaluate temporal properties such as model timing and event ordering over logged data. Use temporal assessments for additional system verification after the simulation is complete. Temporal assessments are associated with test cases in the Test Manager. Author temporal assessments by using the **Logical and Temporal Assessments Editor**. See “Assess Temporal Logic by Using Temporal Assessments” on page 3-93 for more information.

Temporal assessment evaluation results appear in the **Results and Artifacts** pane. Use the Expression Tree to investigate results in detail. If you have a Requirements Toolbox license, you can establish traceability between requirements and temporal assessments by creating requirement links. See “Link to Requirements” on page 1-2 for more information.

See Also

Related Examples

- “Compare Model Output to Baseline Data” on page 6-7
- “Test Two Simulations for Equivalence” on page 6-35

Assess Model Simulation Using verify Statements

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

Activate verify Statements in the Test Assessment Block

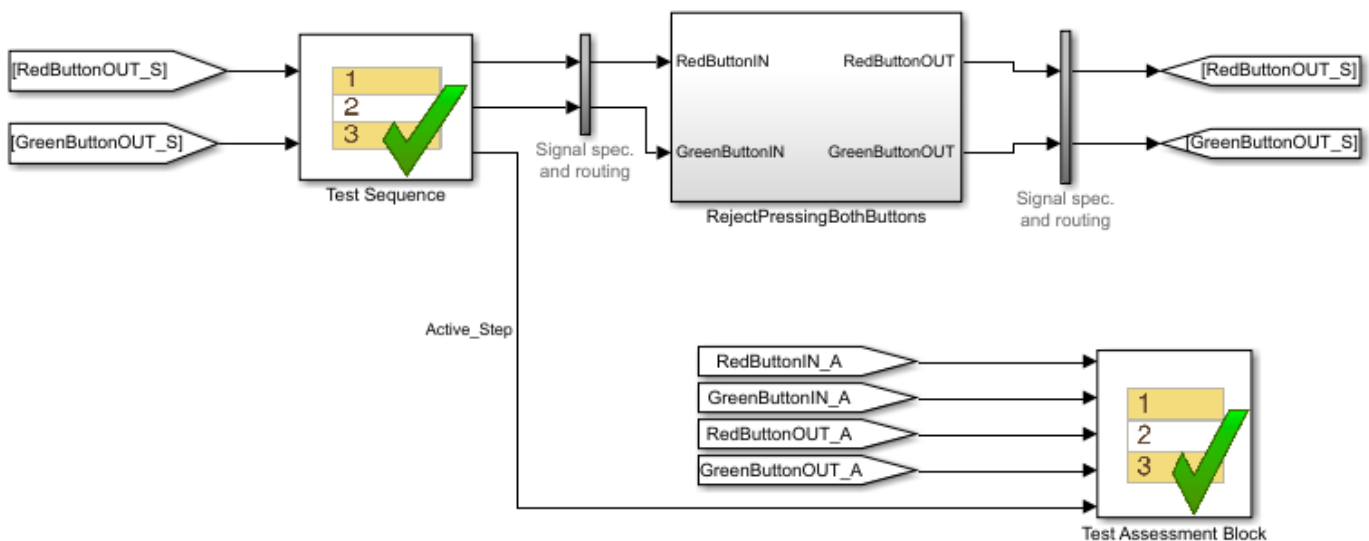
The Test Assessment contains a `When` decomposition sequence. The `When` decomposition sequence helps you clearly define the simulation condition that activates each `verify` statement:

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

Activate verify Statements with Test Sequence Steps

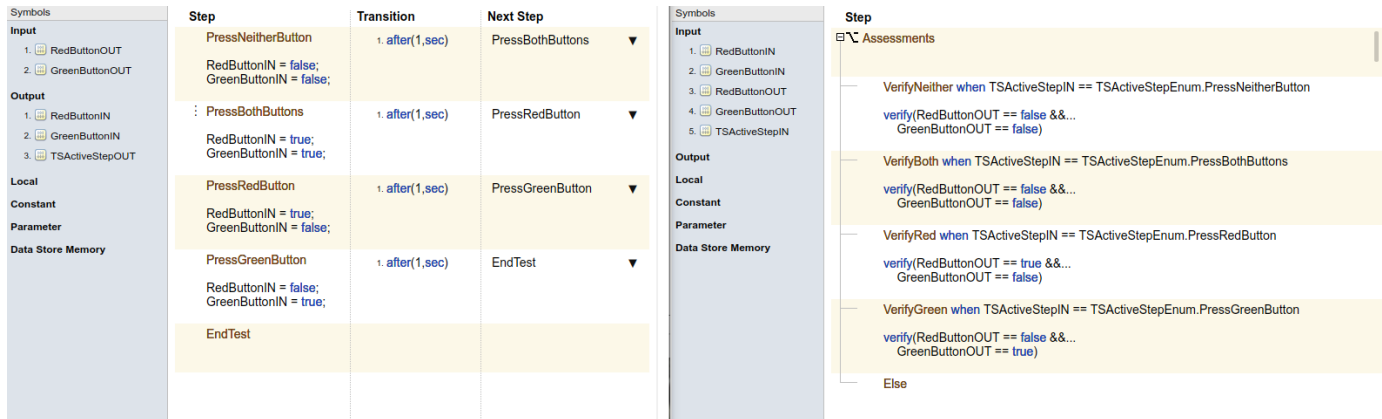
Connect the Test Sequence and Test Assessment block with the active step signal from the Test Sequence block. Activate each `verify` statement with the active step.

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



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

The `Else` step in this example has no actions and handles simulation conditions that do not match any of the preceding `when` conditions. The name of such a step can be `Else` or any other desired name. This step cannot contain a `when` condition.



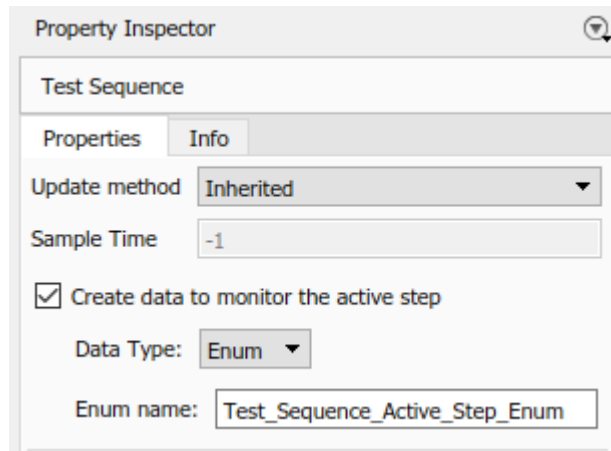
To activate verify statements in a Test Assessment with active steps in a Test Sequence block:

- 1 Create active step data output for the Test Sequence block.

- a Select the Test Sequence block.


Create a new enumerated data output. In the Property Inspector, select **Create data to monitor the active step** and set the **Data Type** to Enum.

- b Enter a name in **Enum name**.



- 2 Create a data input for the Test Assessment block:

- a Open the Test Assessment block.

- b In the **Symbols** pane, hover next to **Input**, then click **Add data** .

- c Name the input.

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

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

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

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

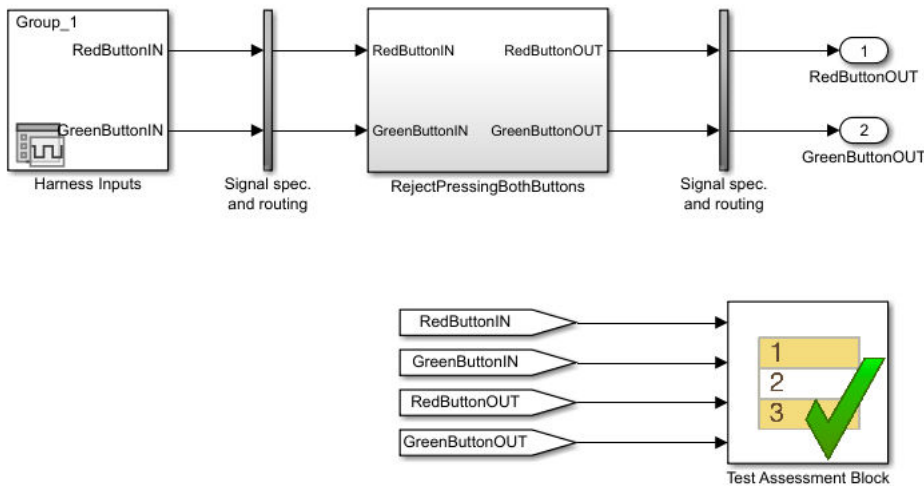
```
VerifyBoth when TSetActiveStepIN == ...
    Test_Sequence_Active_Step_Enum.PressBothButtons
```

- c Add verify statements to each assessment step.

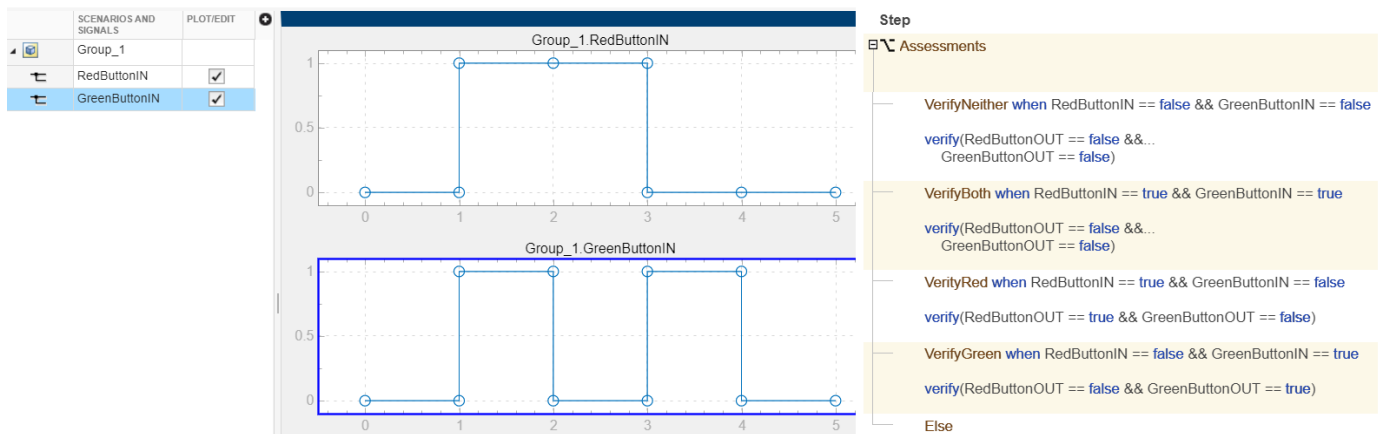
Activate verify Statements with Signal Conditions

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

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



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



Author verify Statements

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

A verify statement returns a pass, fail, or untested result for each time step and for the overall simulation. A fail at any time step results in an overall fail. If there are no failing results, a pass at any time step results in an overall pass. Otherwise, the overall result is untested. Results appear in the **Verify Statements** section of the test results. For details on verify syntax and considerations for using it, see the verify reference page.

Example

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

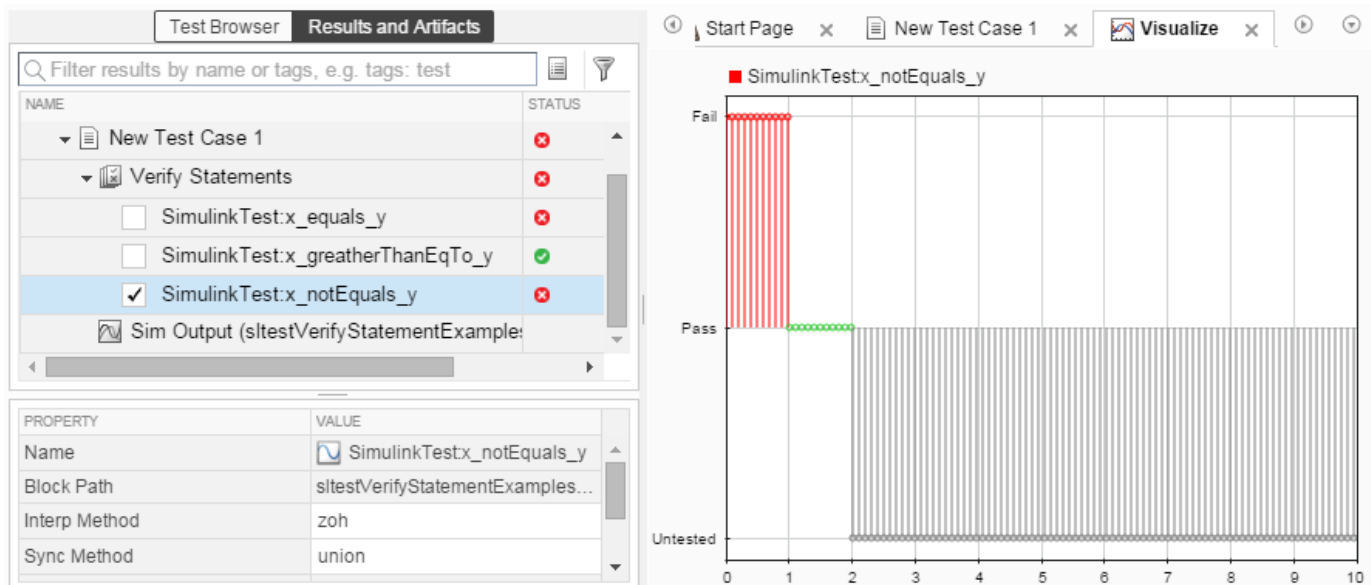
- `verify(x >= y)` passes overall because it is true for the entire test sequence.
- `verify(x == y)` and `verify(x ~= y)` fail because they fail in `step_1_2` and `step_1_1`, respectively.

Step

```

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

The Test Manager displays the results:



See Also

“Test Sequence Editor” on page 3-30 | Test Sequence | Test Assessment | verify |
`sltest.testmanager.Assessment` | `sltest.testmanager.AssessmentSymbol` |
`sltest.testmanager.TestCase`

Related Examples

- “Verify Multiple Conditions at a Time” on page 3-23
- “Requirements-Based Testing for Model Development” on page 1-7

Verify Multiple Conditions at a Time

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

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

Parallel Input Conditions and Expected Outputs

Condition	Expected Output
<code>RedButtonIN == false && GreenButtonIN == false</code>	<code>RedButtonOUT == false && GreenButtonOUT == false</code>
<code>GreenButtonIN == false</code>	<code>GreenButtonOUT ~= true</code>
<code>RedButtonIN == false</code>	<code>RedButtonOUT ~= true</code>
<code>RedButtonIN == true && GreenButtonIN == true</code>	<code>RedButtonOUT == false && GreenButtonOUT == false</code>
<code>RedButtonIN == true && GreenButtonIN == false</code>	<code>RedButtonOUT == true && GreenButtonOUT == false</code>
<code>RedButtonIN == false && GreenButtonIN == true</code>	<code>RedButtonOUT == false && GreenButtonOUT == true</code>

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

Symbols

Input

1. RedButtonIN
2. GreenButtonIN
3. RedButtonOUT
4. GreenButtonOUT

Output

Local

Constant

Parameter

Data Store Memory

Step	Transition	Next Step
<p>Assessments</p> <pre> if RedButtonIN == false && GreenButtonIN == false verify(RedButtonOUT == false && GreenButtonOUT == false) end if GreenButtonIN == false verify(GreenButtonOUT ~= true) end if RedButtonIN == false verify(RedButtonOUT ~= true) end if RedButtonIN == true && GreenButtonIN == true verify(RedButtonOUT == false && GreenButtonOUT == false) end if RedButtonIN == true && GreenButtonIN == false verify(RedButtonOUT == true && GreenButtonOUT == false) end if RedButtonIN == false && GreenButtonIN == true verify(RedButtonOUT == false && GreenButtonOUT == true) end </pre>		

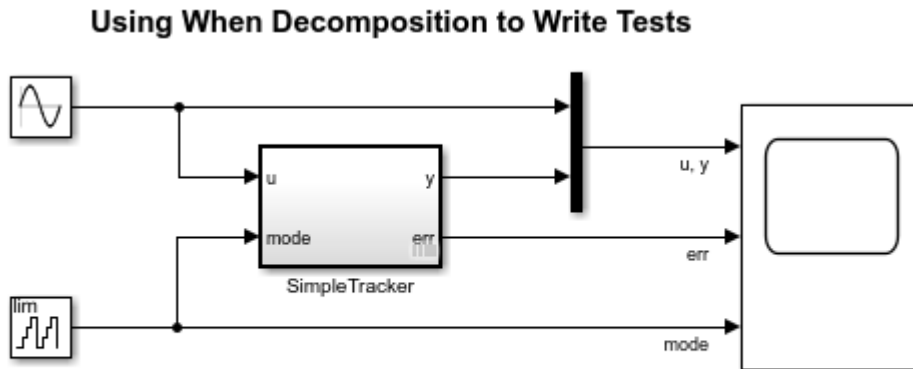
See Also

Test Assessment

Assess a Model by Using When Decomposition

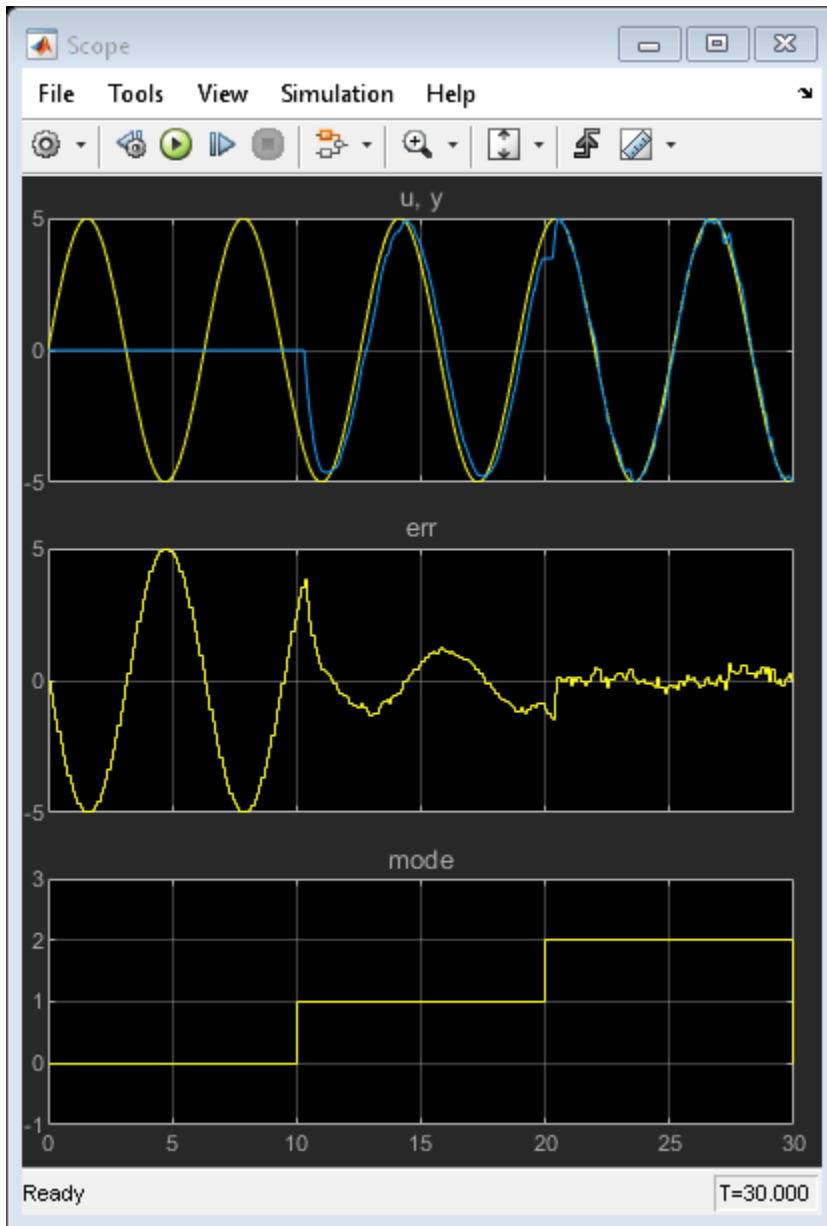
This example shows how to use *When* decomposition in a Test Sequence block to author assessments in a test harness.

This model implements a simple signal tracker that operates in three modes: 0 (Off), 1 (Slow), and 2 (Quick).



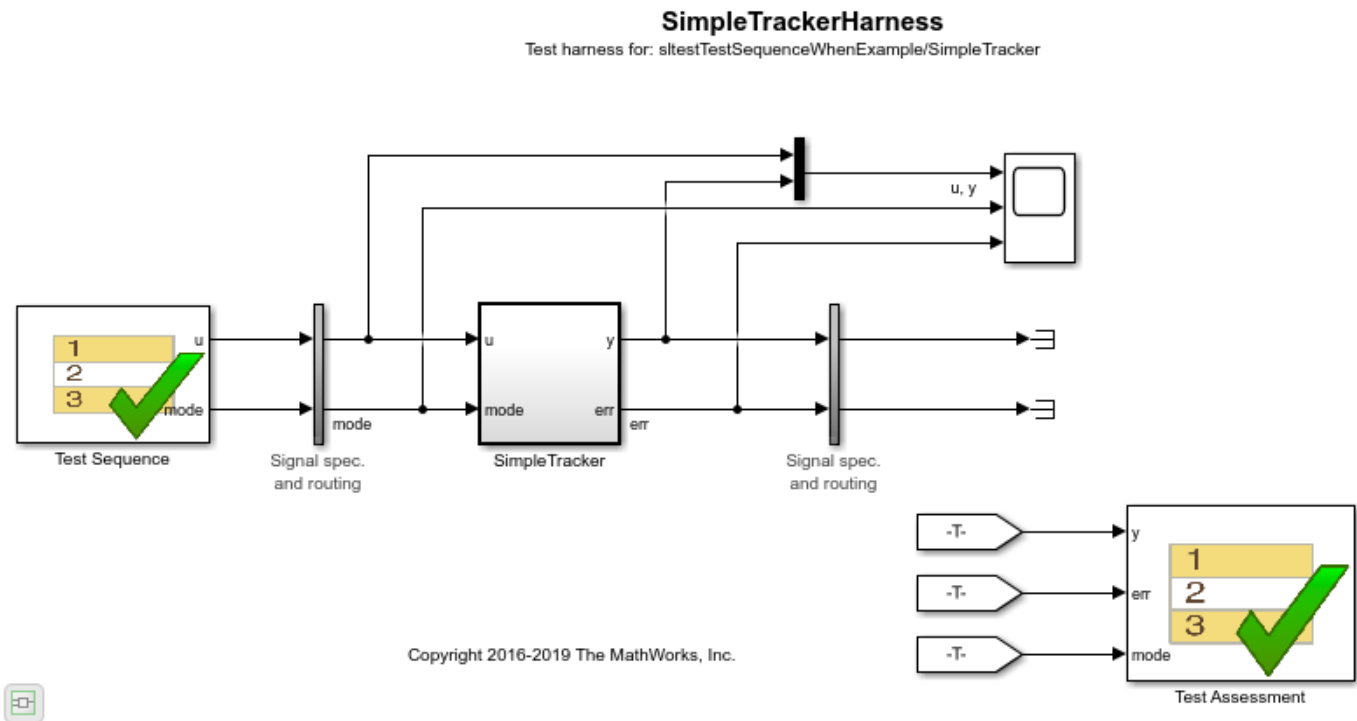
Copyright 2015-2019 The MathWorks, Inc.

To observe the output and error of the signal tracker, simulate the model.



Open the Test Harness

The SimpleTracker subsystem has a test harness that contains a Test Assessment block.



The Test Assessment block assesses the behavior of the SimpleTracker subsystem by using a When decomposition test sequence.

Step

☐ CheckError

OffMode when mode == uint8(0)

`verify(elapsed < 0.5 || y == 0, 'After 0.5 sec in Off, y must remain 0');`

SlowMode when mode == uint8(1)

`verify(elapsed < 0.5 || err < 2, 'After 0.5 sec in Slow, err must remain < 2');`

QuickMode

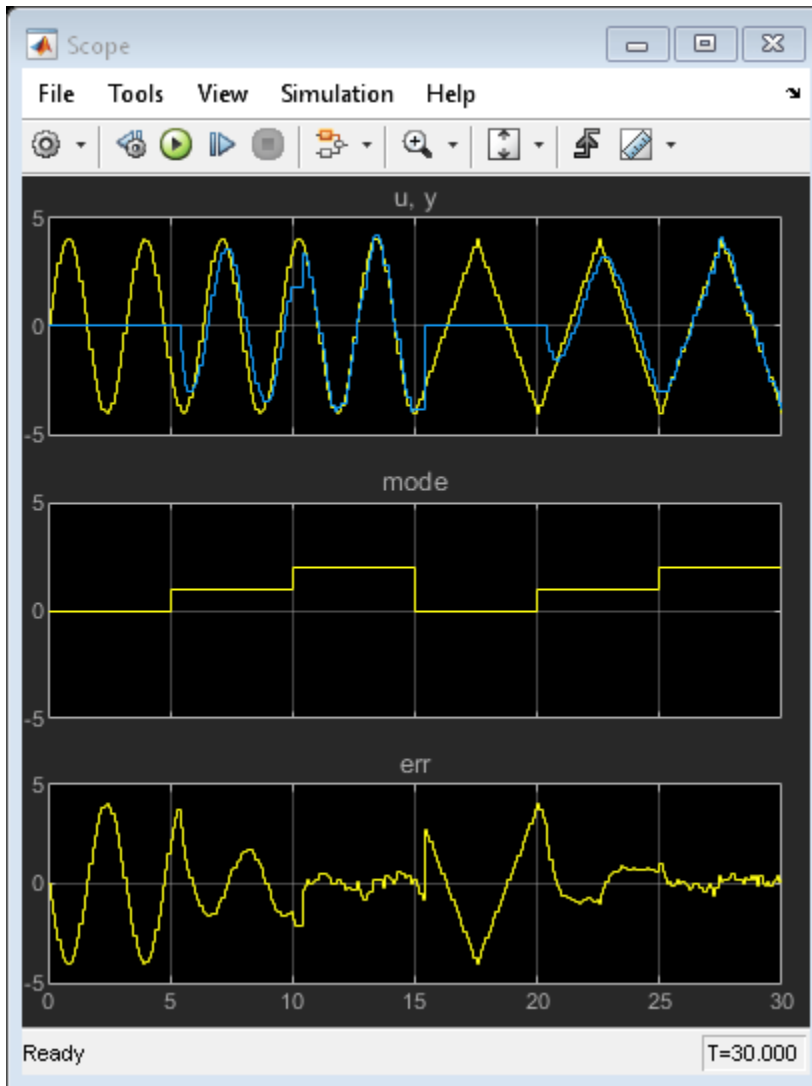
`verify(elapsed < 0.5 || err < 1, 'After 0.5 sec in Quick, err must remain < 1');`

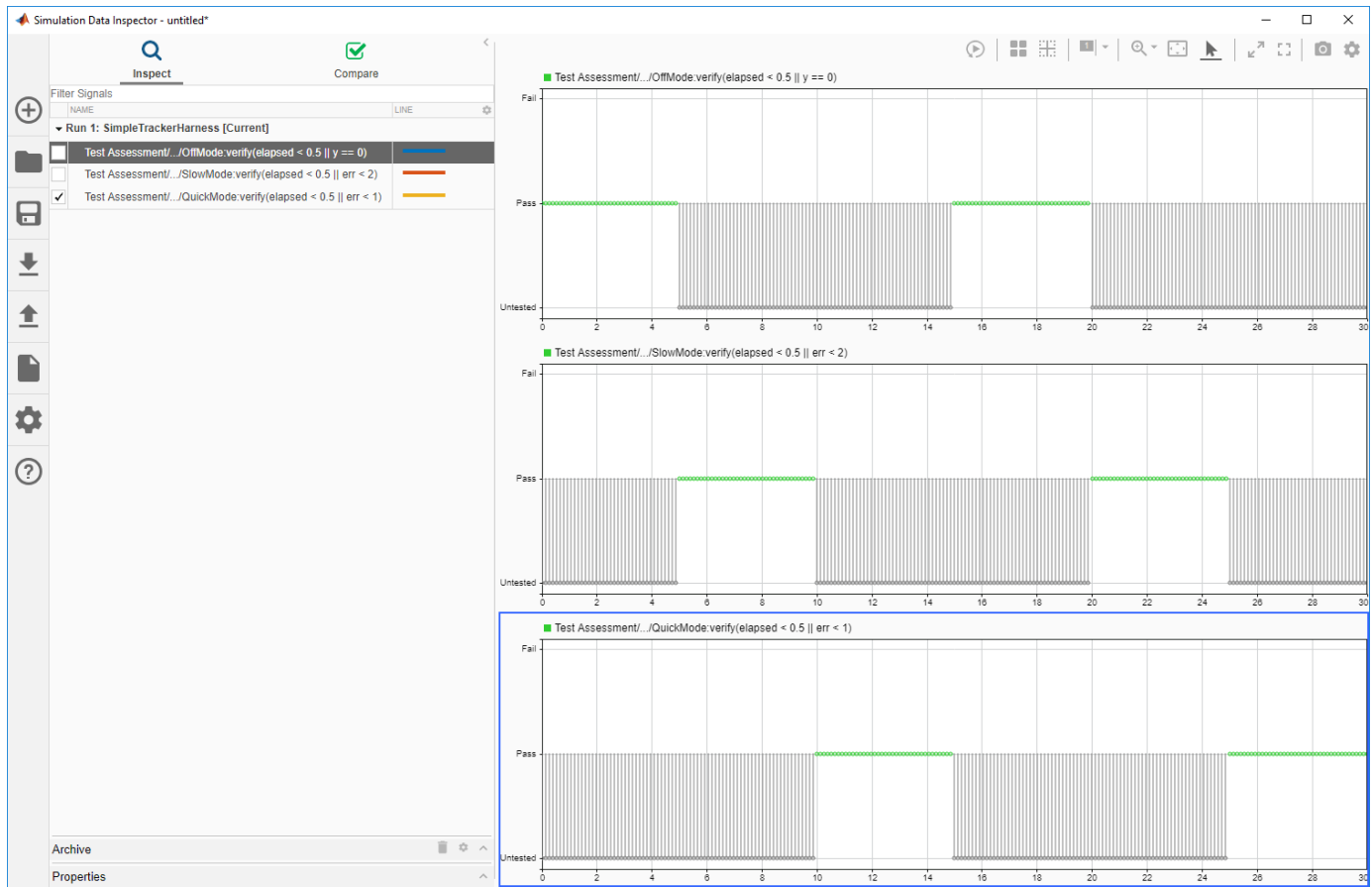
The test sequence determines the appropriate verify() statements to run based on the value of mode. The CheckError step has a When decomposition with three substeps:

- OffMode is active when the value of mode is 0 (Off).
- SlowMode is active when the value of mode is 1 (Slow).
- QuickMode is active for all other values of mode.

Run the Model Assessments

To run the assessments, simulate the test harness. Open the Simulation Data Inspector to inspect the result of the assessments.





Close the test harness and main model.

See Also

Test Assessment | Test Sequence | `sltest.testmanager.Assessment` | `sltest.testmanager.AssessmentSymbol` | `sltest.testmanager.TestCase`

Related Examples

- “Test Sequence Basics” on page 3-2
- “Test Sequence and Assessment Syntax” on page 3-68

Test Sequence Editor

The Test Sequence Editor enables you to define and modify test sequences for Test Sequence and Test Assessment blocks. To open the Test Sequence Editor, double-click a Test Sequence or Test Assessment block.


Define Test Sequences

A test sequence consists of test steps arranged in a hierarchy. Test steps can contain transitions that define how a test progresses in response to the simulation. Test steps can also have a **When** decomposition that uses logic similar to an `if-elseif-else` statement. By default:

- New Test Sequence blocks contain two standard transition test steps.
- New Test Assessment blocks contain a **When** decomposition test step with two sub-steps.

For more information, see “Transition Types” on page 3-2.

To define a test sequence:

- 1** Add test steps, as described in “Manage Test Steps” on page 3-30.
- 2** In the **Step** cell, define outputs and assessments.
- 3** To add a transition from a test step:
 - a** Point to the **Transition** cell and click **Add transition**.
 - b** In the **Transition** cell, define the conditions for exiting the step.
 - c** In the **Next Step** cell, select the next test step from the drop-down list.
- 4** To define a step with a **When** decomposition:
 - a** Right-click a test step and select **When decomposition**. The step displays the icon .
 - b** Add sub-steps, as described in “Manage Test Steps” on page 3-30.
 - c** In the **Step** cell of each sub-step, enter the when operator, followed by a condition. Do not add a condition to the last sub-step.

Test Sequence Scenarios

To define multiple test sequences in a single Test Sequence block, use scenarios. In the left pane of the Test Sequence Editor, click **Scenarios**, then click **Use Scenarios**. The existing test steps and transitions are moved into a scenario tab named `Scenario_1`. Add more scenarios to define more test sequences. For more information on Test Sequence scenarios, see “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59.

Manage Test Steps

In the Test Sequence Editor, you can add and delete test steps to your test sequence. You can also reorder the test steps and change their position in the hierarchy.

Add and Delete Test Steps

To add a test step, right-click an existing step and select **Add step before** or **Add step after**.

To add a test step in a lower hierarchy level, right-click the parent step and select **Add sub-step**.

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

Copy and Paste Test Steps

To copy a test step, right-click the area to the left of the step name and select **Copy step**. Alternatively, select the test step and use the shortcut **Ctrl+C**.

To cut a test step, right-click the area to the left of the step name and select **Cut step**. Alternatively, select the test step and use the shortcut **Ctrl+X**.


To paste a test step, right-click the area to the left of a step name and select **Paste step**, then:

- **Paste before step**
- **Paste after step**
- **Paste sub-step**

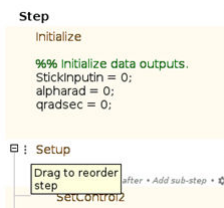
Alternatively, select the test step and use the shortcut **Ctrl+V**.

Reorder Test Steps and Transitions

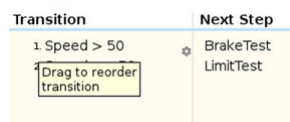
To reorder the test steps in a test sequence:

- 1 Point to a test step. The icon  appears to the left of the step name.
- 2 Click and drag the icon to reorder the test step.

You can reorder test steps within the same hierarchy level. When you move a test step, sub-steps move with the test step.



To reorder step transitions within the same test step, click and drag a transition number to reorder the transition. The corresponding next step moves with the transition.




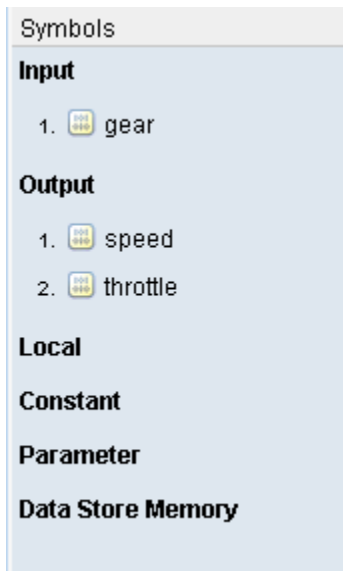
Change Test Step Hierarchy

To move a test step to a lower level in the hierarchy, right-click the step and select **Indent step**. You can only indent a test step when the preceding step is at the same hierarchy level. You cannot indent the first test step in a sequence or the first step in a hierarchy group.







To move a test step to a higher level in the hierarchy, right-click the step and select **Outdent step**. You can only move the last step in a hierarchy group to a higher level in the hierarchy.






Manage Input, Output, and Data Objects


In the **Symbols** sidebar of the Test Sequence Editor, you add, edit, or delete symbols in the Test Sequence block. You can access these symbols from test steps at any hierarchy level. To show or hide the **Symbols** sidebar, click the **Symbols Sidebar** button  on the Test Sequence Editor toolbar.




To add a data symbol, point to the node for a symbol type and click an add symbol button. Available options and additional setup steps depend on the symbol type.

Symbol Type	Description	Procedure for Adding Symbol
Input	Options for input entries include: <ul style="list-style-type: none"> Data Messages 	<ol style="list-style-type: none"> In the Symbols sidebar, point to the Input node and click either: <ul style="list-style-type: none">  Add data  Add message Enter the name of the input and press Enter.
Output	Options for output entries include: <ul style="list-style-type: none"> Data Messages Function Calls Triggers 	<ol style="list-style-type: none"> In the Symbols sidebar, point to the Output node and click: <ul style="list-style-type: none">  Add data  Add message  Add function call  Add trigger Enter the name of the output and press Enter.


Symbol Type	Description	Procedure for Adding Symbol
Local	Local data entries are available only inside the Test Sequence block in which they are defined.	<ol style="list-style-type: none"> 1 In the Symbols sidebar, point to the Local node and click  Add data. 2 Enter the name of the local variable and press Enter. Initialize the local variable in the first test step.
Constant	Constants are read-only data entries available only inside the Test Sequence block in which they are defined.	<ol style="list-style-type: none"> 1 In the Symbols sidebar, point to the Constant node and click  Add data. 2 Enter the name of the constant and press Enter. 3 Point to the name of the constant and click  Edit. 4 In the dialog box, in the Constant Value field, enter the value of the constant.
Parameter	Parameters are available inside and outside the Test Sequence block.	<ol style="list-style-type: none"> 1 Using the Model Explorer, add a parameter in the workspace of the model that contains the Test Sequence block. 2 In the Symbols sidebar, point to the Parameter node and click  Add data. 3 Enter the name of the parameter and press Enter.
Data Store Memory	Data Store Memory entries are available inside and outside the Test Sequence block.	<ol style="list-style-type: none"> 1 Using the Model Explorer, add a Simulink.Signal object in the workspace of the model that contains the Test Sequence block. Alternatively, add a Data Store Memory block to the model. 2 In the Symbols sidebar, point to the Data Store Memory node and click  Add data. 3 Enter the name of the data store and press Enter.

To edit a data symbol, point to the name of the symbol and click  **Edit**.

To delete a data symbol, point to the name of the symbol and click  **Delete**.

Find and Replace

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

- 1 To open the **Find & Replace** tool, click the  icon in the toolbar.
- 2 In the **Find what** field, enter the text you want to locate.

- 3 In the **Replace with** field, enter the updated text.
- 4 To locate the text, click **Find Next** or **Find Previous**.
- 5 To replace the old text with the updated text, click **Replace**.

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

Automatic Syntax Correction

The Test Sequence Editor changes the syntax automatically for:

- **Increment and decrement operations**, such as `a++` and `a--`. For example, `a++` is changed to `a=a+1`.
- **Assignment operations**, such as `a+=expr`, `a-=expr`, `a*=expr`, and `a/=expr`. For example, `a+=b` is changed to `a=a+b`.
- **Evaluation operations**, such as `a!=expr` and `!a`. For example, `a!=b` is changed to `a~=b`.
- **Explicit casts for literal constant assignments**. For example, if `y` is defined as type `single`, then `y=1` is changed to `y=single(1)`.

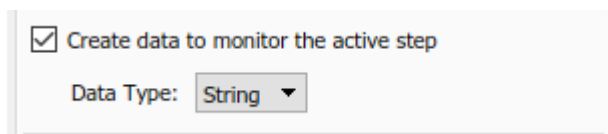
Output and View Active Step Data

When you run a test, the current step in the test sequence is the active step. When you enable creating active step data, a new output port is added to the Test Sequence block for the active step signal. You can analyze the active step data or use the output signal as an input to other blocks in your test harness. For example, a Test Assessment block can use the active step input as a trigger signal. You can also plot the active step data in the Simulation Data Inspector to see how the active step changes over time.

Enable Active Step Output

To create the active step data:

- 1 From the Test Sequence Editor, open the Model Explorer. Alternatively, open the Property Inspector from the test harness or model that contains the Test Sequence or Test Assessment block.
- 2 Enable **Create data to monitor the active step**.



- 3 Set **Data Type** to `String` or `Enum`. The default value is `String`.
 - `String` — Output the active step data as a string. Use this option if you use duplicate step names across scenarios or if you use duplicate substep names in different steps of the same scenario. When you select `String`, the output includes the step name and indicates which scenario is active. The step name string is `<scenario>.<step>.<substep>`. If the block does not have an active step at a time step, the output is an empty string. This situation might occur if a step uses an enabled subsystem and that subsystem is not enabled during that time step.

- Enum — Output the active step as an enumerated value. Selecting Enum enables the **Enum name** parameter. The data is output in the format <Enum name>.<step name>. The scenario of that step is not included in the active step name. You cannot use Enum for the active step output if you have duplicate step or substep names in your test sequence.

The screenshot shows a configuration panel with the following elements:

- A checked checkbox labeled "Create data to monitor the active step".
- A "Data Type:" label followed by a dropdown menu set to "Enum".
- An "Enum name:" label followed by a text input field containing "Test_Sequence_Active_Step_Enum".

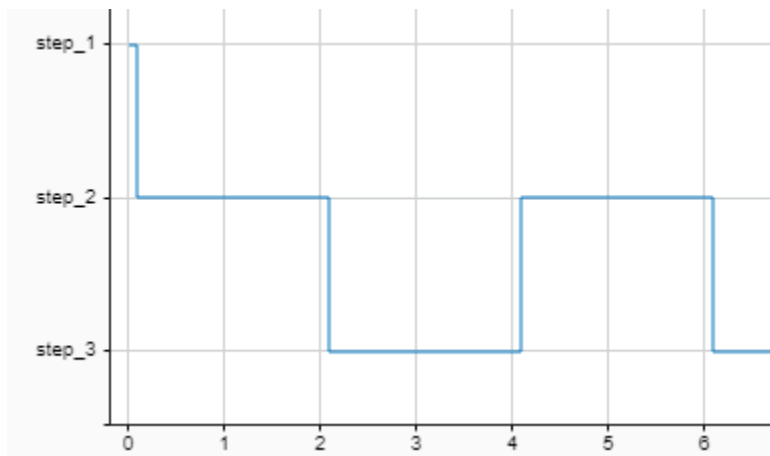
- 4 Click **Apply**.

View Active Step Data

You can view the active step output data in the Simulation Data Inspector, or if you run your test in the Test Manager, in the Test Manager **Results** pane. Before running the test, in the harness, right-click the active data output signal of the Test Sequence block and click **Log Selected Signal**. If you want to plot the active step output, you do not need to connect the active step output signal to any component.

After running a test, open the Simulation Data Inspector or the Test Manager **Results**. The format of the active step plot differs depending on whether the output is an enumerated or string type.

- Enumerated type — The x-axis is time and the y-axis is the step.



- String type — The x-axis is time. Steps appear as blocks of time during the period they are active. If there is not enough space to show the full string names, the Simulation Data Inspector truncates the beginning of the names so the step name displays.



If there are too many step blocks to display in the plot, only the steps that fit are shown.

You can add the active step data plot to a Test Results report in the same way you include other test results.

Use Active Step Output as Input to Another Block

To use the active step output signal as input to another block, such as a Test Assessment block, connect the signal to the input port of the block. You can then use the signal to trigger actions and assessments based on the active step.

See Also

Test Assessment | Test Sequence

Related Examples

- “Test Sequence Basics” on page 3-2
- “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59
- “Test Sequence and Assessment Syntax” on page 3-68
- “Use Stateflow Chart for Test Harness Inputs and Scheduling” on page 3-8

Transitions, Temporal Operators, and Messages in Test Sequence Blocks

In this section...

“Transition Between Steps Using Temporal or Signal Conditions” on page 3-37

“Temporal Operators” on page 3-37

“Transition Operators” on page 3-38

“Use Messages in Test Sequences” on page 3-39

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 block steps through several actions based on changes in the signal switch. See `hasChanged`.

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

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: <code>et(msec)</code>

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

Syntax in the table uses these arguments:

TimeUnits

The units of time

Value: sec | msec | usec

Examples:

msec

Condition

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

Examples:

u > 0

x <= 1.56

Transition Operators

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

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

Use Messages in Test Sequences

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

Receive Messages and Access Message Data

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

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

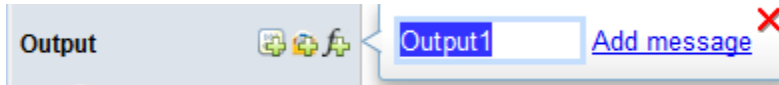


`receive(M)` determines whether a message is present in the input queue M, and removes the message from the queue. `receive(M)` returns `true` if a message is in the queue, and `false` if not. Once the message is received, you can access the message data using the dot notation, `M.data`, or forward the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

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

Send Messages

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



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

Forward Messages

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

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

Compare Test Sequences Using Data and Messages

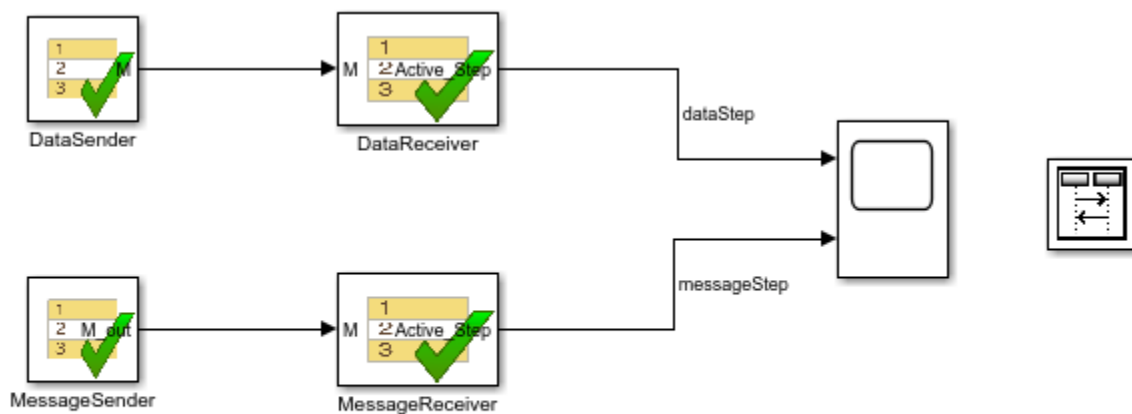
This example demonstrates message inputs and outputs, sending, and receiving a message. The model compares two pairs of test sequences. Each pair is comprised of a sending and receiving test sequence block. The first pair sends and receives data, and the second sends and receives a message.

Set the model name variable.

```
model = 'sltest_testsequence_data_vs_message';
```

Open the model.

```
open_system(model)
```



Test Sequences Using Data

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

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

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

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

Test Sequences Using Messages

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

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

The MessageReceiver block waits 3 seconds, then transitions to step S2. Step S2's transition evaluates the queue M with `receive(M)`, removing the message from the queue. `receive(M)`

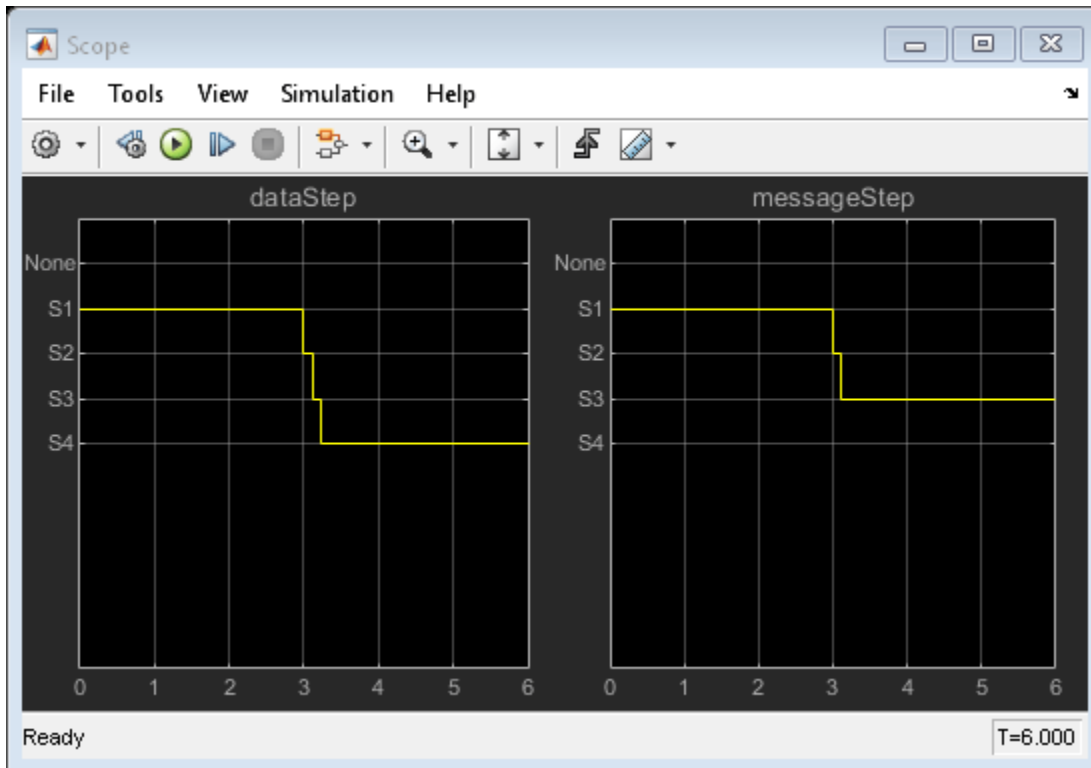
returns `true` since the message is present. `M.data == 3.5` compares the message data to the expected value. The statement is true, and the sequence transitions to step S3.

Step	Transition	Next Step	Description
S1	1. <code>after(3,sec)</code>	S2	Waits.
S2	1. <code>receive(M) && 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)
```

See Also

Test Sequence | “Test Sequence and Assessment Syntax” on page 3-68

Related Examples

- “Assess Model Simulation Using verify Statements” on page 3-18
- “Generate Test Signals” on page 3-44
- “Use Stateflow Chart for Test Harness Inputs and Scheduling” on page 3-8

Generate Test Signals

In this section...

“Signal Generation Functions” on page 3-44

“Sinusoidal and Random Number Functions in Test Sequences” on page 3-46

In the Test Sequence block, you can generate signals to use for testing. First, define an output data symbol using the **Data Symbols** pane, and then use that output name with a signal generation function in a test step. For information on adding symbols, see “Manage Input, Output, and Data Objects” on page 3-32. For an example that shows how to implement signal functions in a Test Sequence block, see “Sinusoidal and Random Number Functions in Test Sequences” on page 3-46

Signal Generation Functions

The following table lists common functions you can use in the Test Sequence block to create test signals, random number values, and natural exponents. It also describes the `latch` function, which saves and returns a specific value evaluated within a test sequence step. For more information about each function, click its name in the first column.

Some signal generation functions use the temporal operator `et`, which is the elapsed time of the test step in seconds. For additional operators related to `et` that you can use in test sequence steps, see “Temporal Operators” on page 3-37.

Note Scaling, rounding, and other approximations of argument values can affect function outputs.

Function	Syntax	Description	Example
<code>sin</code>	<code>sin(x)</code>	Returns the sine of x , where x is in radians.	A sine wave with a period of 10 sec: <code>sin(et*2*pi/10)</code>
<code>cos</code>	<code>cos(x)</code>	Returns the cosine of x , where x is in radians.	A cosine wave with a period of 10 sec: <code>cos(et*2*pi/10)</code>
<code>square</code>	<code>square(x)</code>	Square wave output with a period of 1 and range -1 to 1 . Within the interval $0 \leq x < 1$, <code>square(x)</code> returns the value 1 for $0 \leq x < 0.5$ and -1 for $0.5 \leq x < 1$. <code>square</code> is not supported in Stateflow charts.	Output a square wave with a period of 10 sec: <code>square(et/10)</code>

Function	Syntax	Description	Example
sawtooth	sawtooth(x)	Sawtooth wave output with a period of 1 and range -1 to 1. Within the interval $0 \leq x < 1$, sawtooth(x) increases. sawtooth is not supported in Stateflow charts.	Output a sawtooth wave with a period of 10 sec: sawtooth(et/10)
triangle	triangle(x)	Triangle wave output with a period of 1 and range -1 to 1. Within the interval $0 \leq x < 0.5$, triangle(x) increases. triangle is not supported in Stateflow charts.	Output a triangle wave with a period of 10 sec: triangle(et/10)
ramp	ramp(x)	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 is not supported in Stateflow charts.	Ramp one unit for every 5 seconds of test step elapsed time: ramp(et/5)
heaviside	heaviside(x)	Heaviside step signal, returning 0 for $x < 0$ and 1 for $x \geq 0$. heaviside is not supported in Stateflow charts.	Output a heaviside signal after 5 seconds: heaviside(et-5)
exp	exp(x)	Returns the natural exponential function, e^x .	An exponential signal progressing at one tenth of the test step elapsed time: exp(et/10)
rand	rand	Uniformly distributed pseudorandom values	Generate new random values for each simulation by declaring rand extrinsic with coder.extrinsic. Assign the random number to a local variable. For example: coder.extrinsic('rand') nr = rand sg = a + (b-a)*nr


Function	Syntax	Description	Example
randn	randn	Normally distributed pseudorandom values	Generate new random values for each simulation by declaring randn extrinsic with coder.extrinsic. Assign the random number to a local variable. For example: <pre>coder.extrinsic('randn') nr = randn sg = nr*2</pre>
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 is not supported in Stateflow charts.	Latch b to the value of torque: <pre>b = latch(torque)</pre>

Sinusoidal and Random Number Functions in Test Sequences

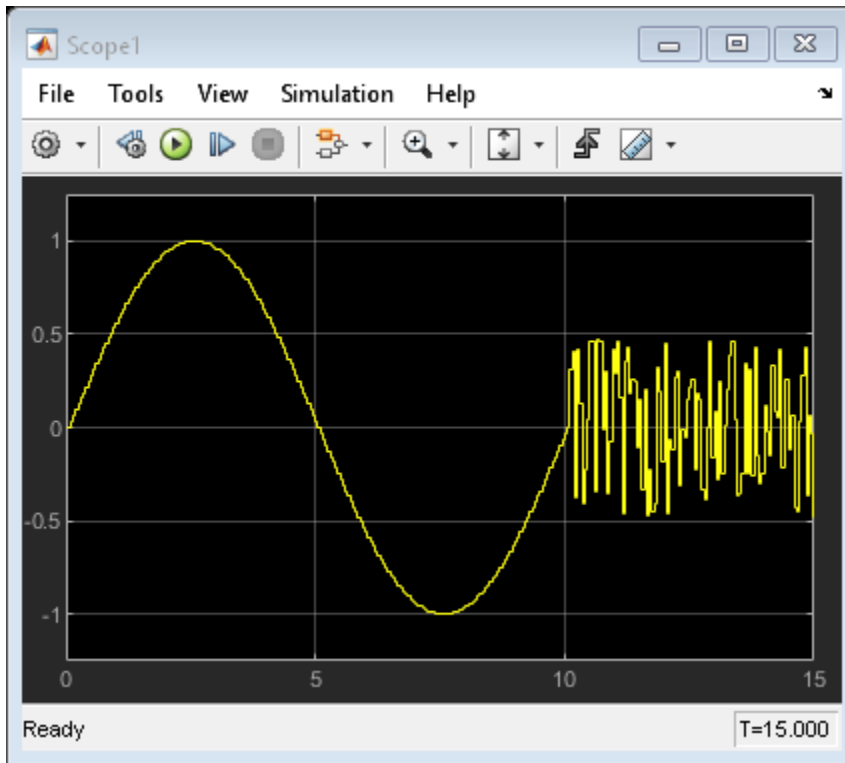
This example shows how to produce a sine and a random number test signal in a Test Sequence block.

If you recreate this test sequence, before running it, set the nr symbol size and type. Hover over the nr symbol and click its Edit icon to open the Data Inspector. Set the Size to 1 and the Type to double.

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

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

The test sequence produces signal sg.



See Also

Test Sequence | "Test Sequence and Assessment Syntax" on page 3-68

Related Examples

- "Assess Model Simulation Using verify Statements" on page 3-18
- "Transitions, Temporal Operators, and Messages in Test Sequence Blocks" on page 3-37
- "Using an External Function in a Test Sequence Block" on page 3-49

Using an External Function in a Test Sequence Block

This example shows how to call an externally-defined function from the Test Sequence block. The provided function, `Attenuate.m`, is defined in a script on the MATLAB® path and is called from the test sequence. The `Attenuate.m` function is:

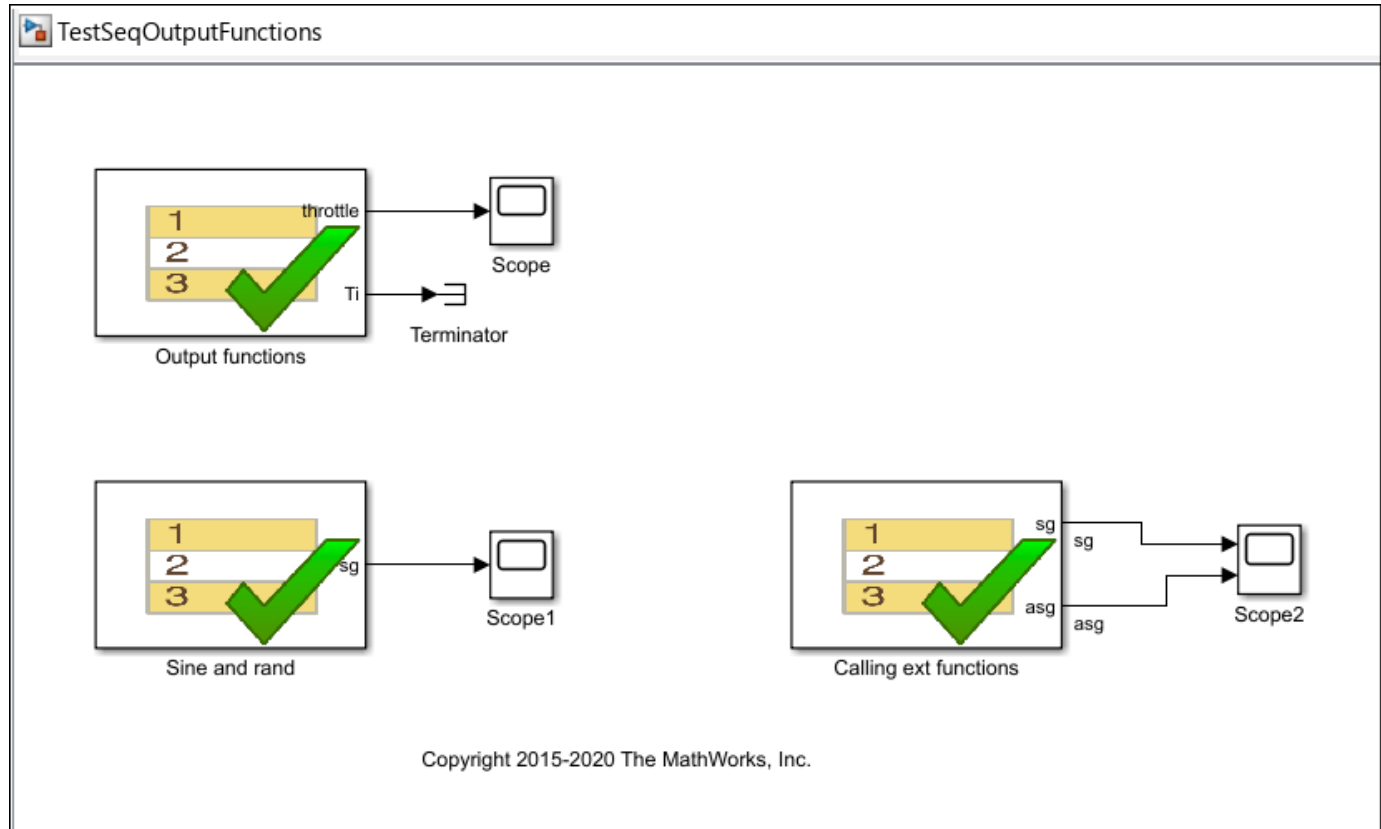
```
function[y] = Attenuate(x)

y = 0.65*x;

end
```

Open the TestSeqOutputFunctions model

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



Open the Test Sequence block

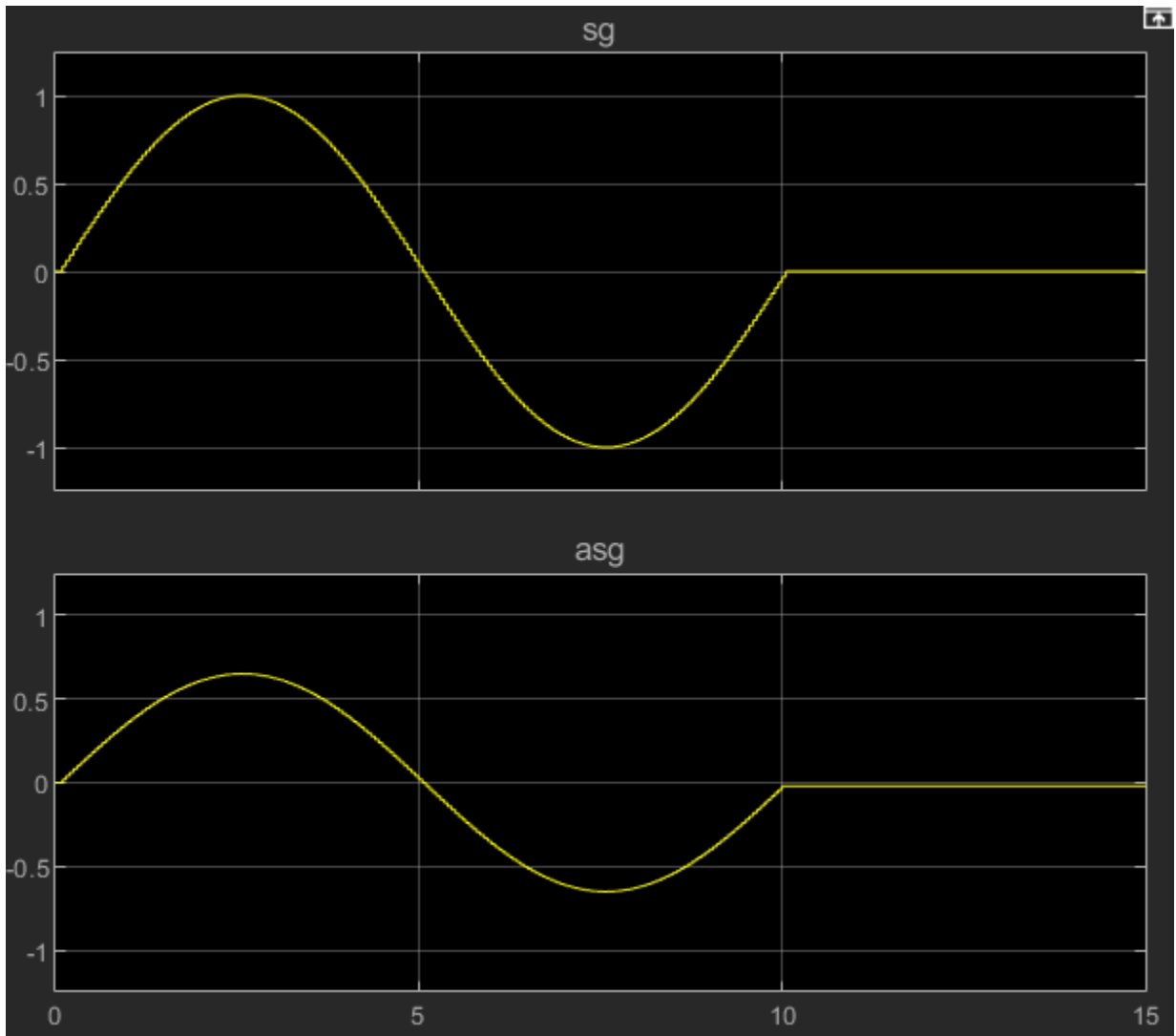
In the model, double-click the Test Sequence block, Calling ext functions. The **ReducedSine** step in the Test Sequence block uses the external function, `Attenuate`, to reduce the amplitude of the signal `sg`.

Symbols	Step	Transition	Next Step
Input	Initialize sg = 0;	1. true	ReducedSine ▼
Output 1. sg 2. asg	ReducedSine sg = sin(et*2*pi/10); asg = Attenuate(sg);	1. after(10,sec)	Stop ▼
Local			
Constant			
Parameter			
Data Store Memory	Stop sg = 0;		

Simulate the model

Simulate the model and view the output signal sg and attenuated signal asg in the Scope2 block.

```
sim(model)
open_system(['model '/Scope2'])
```



```
close_system(model,0)
```

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(model)
```

2. Create the test harness.

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

ans =

```
struct with fields:
    model: 'sltestCruiseChart'
    name: 'Harness1'
    description: ''
    ownerHandle: 0.1782
    ownerFullPath: 'sltestCruiseChart'
    ownerType: 'Simulink.BlockDiagram'
    verificationMode: 'Normal'
    saveExternally: 0
    rebuildOnOpen: 0
    rebuildModelData: 0
    postRebuildCallback: ''
    graphical: 0
    origSrc: 'Test Sequence'
    origSink: 'Test Assessment'
    synchronizationMode: 'SyncOnOpen'
    existingBuildFolder: ''
    functionInterfaceName: ''
```

Author the Test Sequence

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

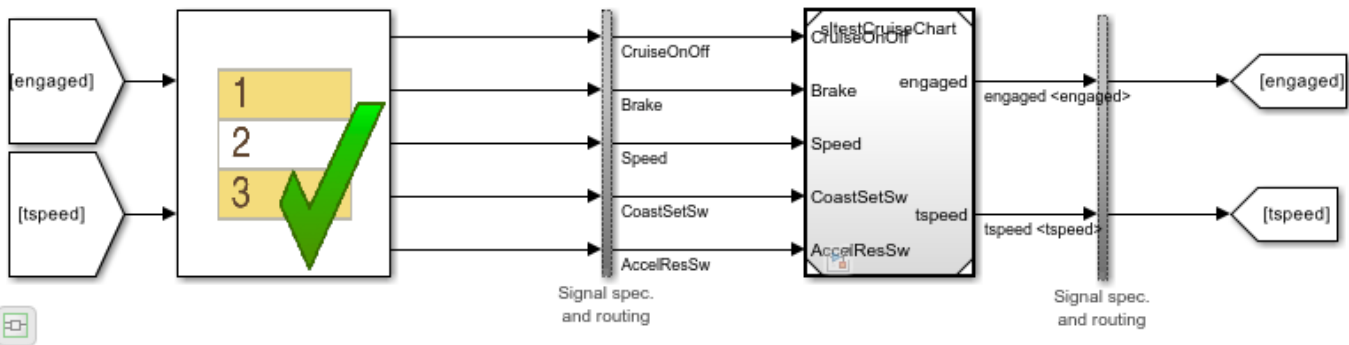
% Add a step to verify that the cruise control is off.
highLimVerifyActions = sprintf('verify(engaged == false)');
sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
    'LimitTest.VerifyHigh','LimitTest.RampUp','Action',highLimVerifyActions)

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

```

Open the test harness to view the test sequence.

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



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

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

Close the Test Harness and Model

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

See Also

“Test Sequence and Assessment Syntax” on page 3-68

Programmatically Create and Run Test Sequence Scenarios

This example shows how to create and define multiple test scenarios in a single Test Sequence block. Being able to define more than one test sequence in a block lets you reduce the number of separate Test Sequence blocks in your test harness.

This example uses the `HeatPumpScenario1` model, which already has a test harness that contains a Test Sequence block. In this example, you convert the block to use scenarios, add a new scenario to the block, edit a scenario step, and activate the new scenario so that it runs when the model simulates.

This example also shows how to use scenarios in iterations to run multiple iterations in a single test case.

Open the Model and Test Harness

Open the Controller subsystem of the `HeatPumpScenario1` model and its test harness, `ScenarioTest`.

```
open_system('HeatPumpScenario1')
sltest.harness.open('HeatPumpScenario1/Controller','ScenarioTest');
```

Set Up the Scenarios

Enable Scenarios

Set the Test Sequence block to use scenarios. The existing steps and transitions are moved into a scenario that, in this example, is named `FirstScenario`. Note that once you change a Test Sequence block to use scenarios, you cannot revert that block to non-scenario mode.

```
sltest.testsequence.useScenario('ScenarioTest/Test Sequence',...
    'FirstScenario');
```

Add Another Scenario

Add a second scenario to the Test Sequence block. Name the scenario `NewScenario`.

```
sltest.testsequence.addScenario('ScenarioTest/Test Sequence','NewScenario');
```

Edit the Scenario Contents

Edit the first step of the new scenario to change the values of the `Troom_in` and `Tset` variables. Preface the name of the step with the scenario name that contains the step. Similarly, when adding or changing transitions, you must also preface the transition with the scenario name.

```
action = sprintf('Troom_in = 75;\nTset = 75;\n');
sltest.testsequence.editStep('ScenarioTest/Test Sequence',...
    'NewScenario.step_1','Action',action);
```

To view the scenario contents, use `sltest.testsequence.findStep(blockPath)`, which returns an array containing the step names for all scenarios. Then, use

```
sltest.testsequence.readStep(stepName) or
sltest.testsequence.readTransition(stepName)
```

to see the contents of the specified step or transition, respectively. You can also view the scenario contents by double-clicking the Test Sequence block in the harness to open the block editor.

Specify Which Scenario to Run

Specify the new scenario to run during model simulation. This scenario is the *active* scenario, which is the only scenario that runs during the simulation. For an alternative way to activate and run a scenario, or to run scenarios using iterations, see below.

```
sltest.testsequence.activateScenario('ScenarioTest/Test Sequence',...
    'NewScenario');
```

Run the Model

Run the Model Using the New Scenario

You can run only one active scenario a time, unless you use a loop at the command line or in a script, or run iterations (see below). Note that fast restart is supported when switching active scenarios and running the model.

```
sim('ScenarioTest')
```

In the test harness, view the Scope blocks to see the simulation results for the new scenario.

Run a Different Scenario

Activate the first scenario.

```
sltest.testsequence.activateScenario('ScenarioTest/Test Sequence',...
    'FirstScenario');
```

Rerun the model.

```
sim('ScenarioTest')
```

In the test harness, view the Scope blocks to see the simulation results for the first scenario.

Use a Workspace Variable to Activate and Run a Scenario

In some cases, such as for looping through scenarios, you might want to use a workspace variable to control which scenario to activate, instead of using `activateScenario`. The steps for using a workspace variable are:

- 1 Set the scenario control source to the workspace by using `sltest.testsequence.setScenarioControlSource('ScenarioTest/Test Sequence',sltest.testsequence.ScenarioControlSource.Workspace);`
- 2 Create a variable in the base workspace, model workspace, or data dictionary to specify the active scenario using its index value. For example, `Active_Scenario_Index = 1;`
- 3 Run the model, which uses the steps and transitions in the active scenario.

To run a different scenario, change the `Active_Scenario_Index` to the desired scenario, for example, `Active_Scenario_Index = 2`, and then rerun the model.

To change the name of the active scenario parameter from `Active_Scenario_Index` to, for example, `ScenarioIndex`, use `sltest.testsequence.editSymbol('ScenarioTest/Test Sequence',... 'Active_Scenario_Index','Name','ScenarioIndex');`

and then create the `ScenarioIndex` variable in the base workspace. Use `Scenario_Index = 2` to set the variable to run the scenario identified by index 2, and then run the model.

Run Scenarios Using Iterations

You can use iterations to run multiple scenarios in a single test case. The following steps use the same model, Test Sequence block, and scenarios defined above.

1. Set up the test file, test suite, and test case.

```
tf = sltest.testmanager.TestFile('Scenario Iterations Test');  
ts = getTestSuites(tf);  
tc = createTestCase(ts, 'simulation', 'Sim Iterations');
```

2. Set the model and test harness for the test case.

```
setProperty(tc, 'Model', 'HeatPumpScenario1');  
setProperty(tc, 'HarnessOwner', 'HeatPumpScenario1/Controller', 'HarnessName', 'ScenarioTest');
```

3. Get the names of the scenarios in the Test Sequence block.

```
tseq_block = 'ScenarioTest/Test Sequence';  
scenarioNames = sltest.testsequence.getAllScenarios(tseq_block);
```

4. Set the Test Sequence block and default scenario for the test case.

```
setProperty(tc, 'TestSequenceBlock', tseq_block);  
setProperty(tc, 'TestSequenceScenario', 'FirstScenario');
```

5. Use a loop to create the iterations, assign a scenario to each iteration, and add the iterations to the test case.

```
for i = 1:numel(scenarioNames)  
    testItr = sltestiteration;  
    setTestParam(testItr, 'TestSequenceScenario', scenarioNames{i});  
    addIteration(tc, testItr);  
end
```

6. Run the test case.

```
run(tc);
```

Close the Test Harness and Model

```
sltest.harness.close('HeatPumpScenario1/Controller', 'ScenarioTest');  
close_system('HeatPumpScenario1', 0);
```

See Also

`sltest.testsequence.useScenario` | `sltest.testsequence.setScenarioControlSource` |
`sltest.testsequence.getActiveScenario` | `sltest.testsequence.editScenario` |
`sltest.testsequence.deleteScenario` | `sltest.testsequence.addScenario` |
`sltest.testsequence.activateScenario`

More About

- “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59

Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager

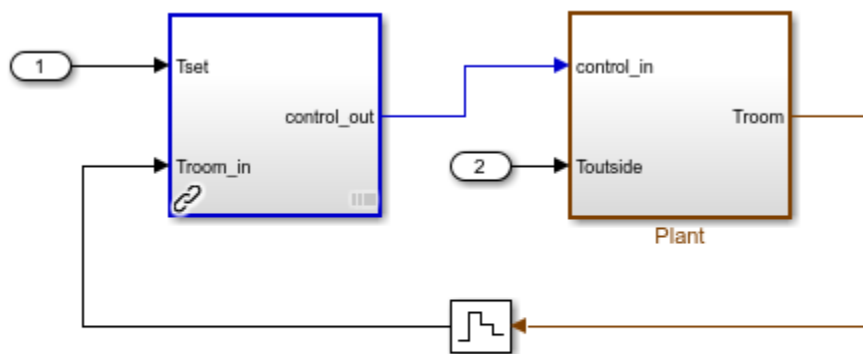
This example shows how to create and use scenarios in the Test Sequence Editor. Scenarios let you include multiple test sequences in a Test Sequence block. If your test harness includes more than one Test Sequence block, you can move each test sequence to a scenario in a single Test Sequence block.

The example also shows how to use the Test Manager to set up and use iterations to run multiple scenarios in a single test case.

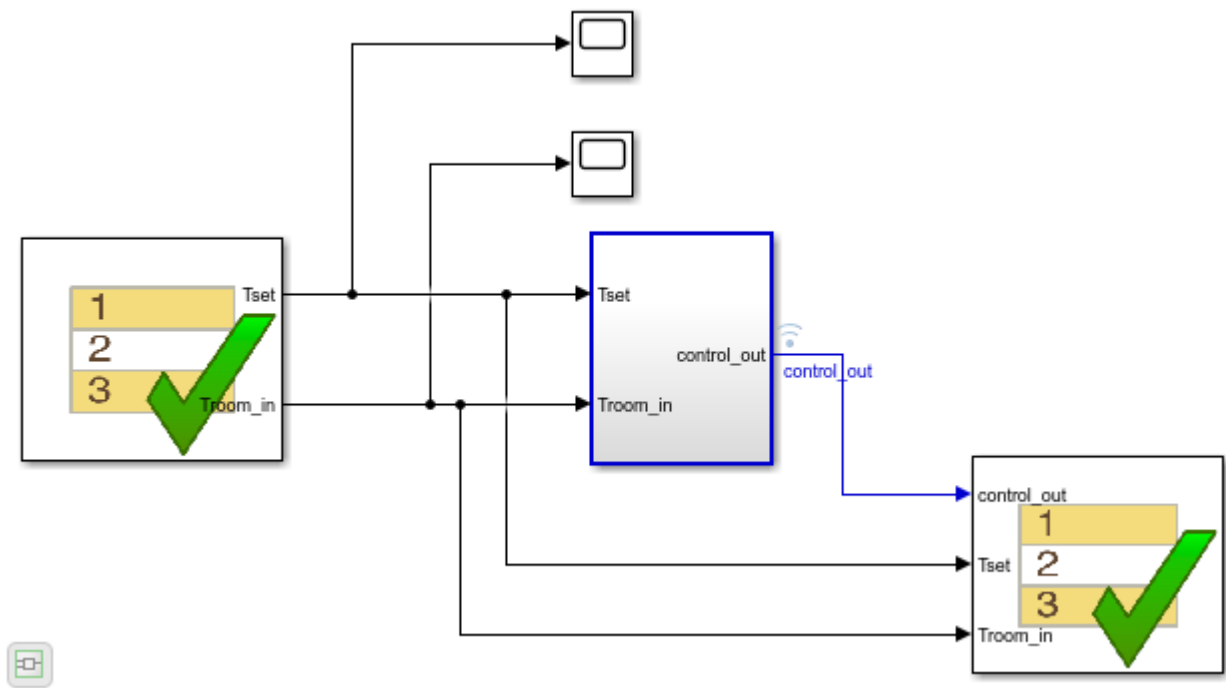
Open the Model and Test Harness

Open the HeatPumpScenario model, ScenarioTest harness, and Test Sequence Editor.

```
open_system('HeatPumpScenario')
sltest.harness.open('HeatPumpScenario/Controller','ScenarioTest');
open_system('ScenarioTest/Test Sequence')
```



Copyright 1990-2014 The MathWorks, Inc.



Enable Scenarios

In the panel on the left side of the Test Sequence Editor, switch to the **Scenarios** tab and click **Use Scenarios**.

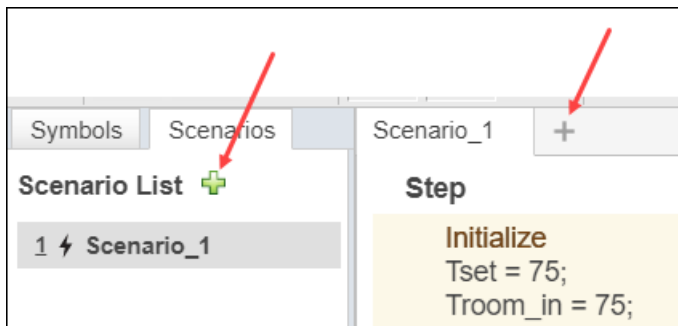
Step	Transition	Next Step
Initialize Tset = 75; Troom_in = 75;	1. true	Test_signals ▼
Test_signals Troom_in = 75+ramp(et*0.1);	1. Troom_in >= 75.5	Stop ▼
Stop		

In the **Start Using Scenarios dialog box**, click **OK**, which confirms that when you switch to scenario mode you cannot revert the Test Sequence block to non-scenario mode. The existing steps and transitions are moved into a tab named **Scenario_1**.

Scenario_1		+
Step	Transition	Next Step
Initialize Tset = 75; Troom_in = 75;	1. true	Test_signals
Test_signals Troom_in = 75+ramp(et*0.1);	1. Troom_in >= 75.5	Stop
Stop		

Add a New Scenario

To add a new scenario, in the **Scenarios** tab, click the plus sign next to **Scenario List**. Alternatively, click on the plus sign next to the header of the **Scenario_1** tab. The new scenario is named **Scenario_2**.

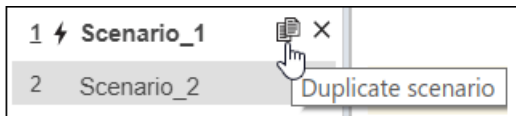
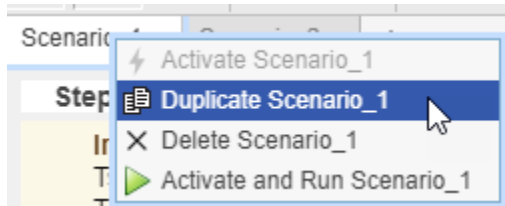


Scenario_1		Scenario_2	+
Step	Transition	Next Step	
step_1	1. true	step_2	
step_2			

Duplicate a Scenario

To start a new scenario from an existing one, you can duplicate it.

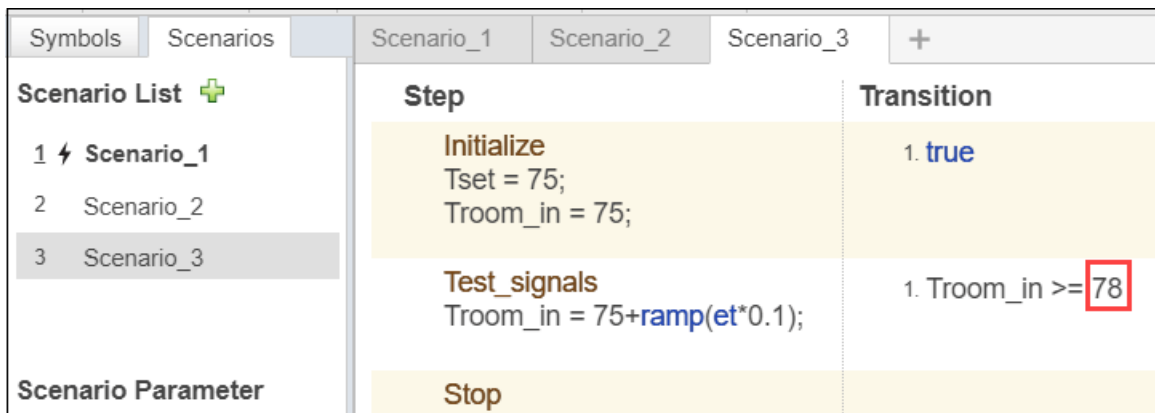
Right-click the **Scenario_1** tab label and select **Duplicate Scenario_1** from the context menu. Alternatively, go to the **Scenarios** side panel, point to **Scenario_1** in the **Scenario List** to display the **Duplicate scenario** button, and click it.



The **Scenario List** section updates and lists the new scenario, **Scenario_3**, which has the same content as **Scenario_1**.

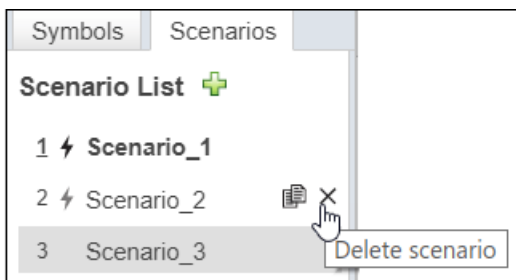
Edit the Steps and Transitions

Modify Scenario_3 to change the transition of the **Test_signals** step to `Troom_in >= 78`.

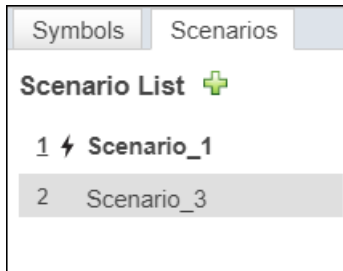


Delete a Scenario

Right-click the **Scenario_2** tab label and select **Delete Scenario_2** from the context menu. Alternatively, in the left pane, in the **Scenarios** tab, point to **Scenario_2** to display the **Delete Scenario** icon. Click the Delete Scenario icon and then click **OK** in the dialog box to delete **Scenario_2**.



The name of **Scenario_3** does not change, but the scenario index shown to the left of the scenario name in the **Scenario List** changes to 2 because it is now the second scenario.

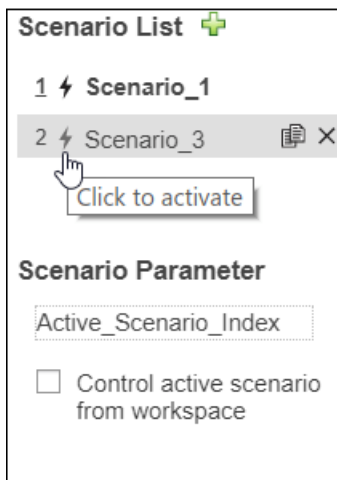


Activate Scenario

In the left pane, in the **Scenario** tab, a black lightning bolt icon and a bold Scenario name indicate that **Scenario_1** is the currently active scenario. If you run the model, only the active scenario runs.

To change the active scenario to **Scenario_3**, right-click the **Scenario_3** tab and select **Activate Scenario_3** from the context menu. Alternatively, in the left pane, in the **Scenarios** tab, point to **Scenario_3** to display the **Click to Activate** icon, which is a gray lightning bolt. Click the lightning bolt to make **Scenario_3** the active scenario.

You can also control the active scenario from the command line. See the **Programmatically Control the Active Scenario** section below.



Run the Active Scenario

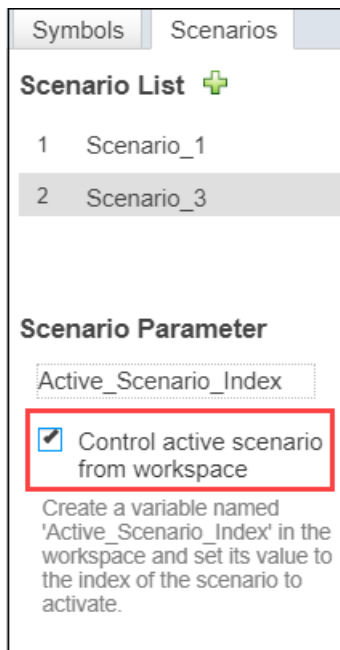
Right-click the **Scenario_3** tab label and select **Run Scenario_3** from the context menu. If you did not activate the scenario, the menu option is **Activate and Run Scenario_3**. Alternatively, run the model normally to run the active scenario. The Test Sequence Editor displays the active scenario during model simulation.

Programmatically Control the Active Scenario

You can alternatively use a variable in the base workspace, model workspace, or a data dictionary to programmatically control the active scenario. To activate a scenario, set the variable value to the index of the scenario.

1. In the Test Sequence Editor **Scenarios** tab, go to the **Scenario Parameter** section.
2. Enable **Control active scenario from workspace**. The previously active scenario deactivates. When you control the active scenario using a workspace variable, the Test Sequence Editor does not know which scenario is active until you click **Run**.

If you select **Control active scenario from workspace**, you cannot activate a scenario from the Test Sequence Editor using the right-click context menu or the **Scenario List**.



3. For this example, in the base workspace, create a variable named `Active_Scenario_Index` and set it to activate the first scenario, `Scenario_1`, by entering: `Active_Scenario_Index = Simulink.Parameter(1);`
4. Return to the Test Sequence Editor and click **Run**. `Scenario_1` runs.

Instead of using `Active_Scenario_Index` as the name of the variable, you can specify a different name.

1. For this example, in the Test Sequence Editor, in the **Scenario Parameter** section, click on `Active_Scenario_Index` and enter a new name, such as `ChangeScenario`.
2. In the base workspace, create a variable named `ChangeScenario`. Set it to the desired scenario index, such as 1, by entering: `ChangeScenario = Simulink.Parameter(1);`
3. Return to the Test Sequence Editor and click **Run**. `Scenario_1` runs.

Use Iterations to Run Multiple Scenarios in a Test Case

You can run multiple scenarios in a test case by using iterations. This section describes scenarios in iterations using the Test Manager.

1. In the Test Manager, create a test file, test suite, and simulation test case.

2. In the **System Under Test** section, set the **Model** to HeatPumpScenario and the **Harness** to ScenarioTest.
3. In the **Inputs** section, click the Refresh icon next to the **Test Sequence Block** field to populate it with paths to Test Sequence blocks in the harness.
4. Set **Test Sequence Block** to the ScenarioTest/Test Sequence block, which has the scenarios to use in the iterations.
5. Click the Refresh icon next to the **Override with Scenario** field to populate it with the scenarios in the selected block.
6. Set **Override with Scenario** to Scenario_1, which set that scenario as the default for all iterations. This scenario overrides the active scenario in the Test Sequence block. In the **Iterations** section, you can change this default scenario to another scenario for each iteration.

▼ INPUTS

Include input data in test result

Stop simulation at last time point

EXTERNAL INPUTS

NAME	FILE
No inputs available. Click Add to add new inputs.	

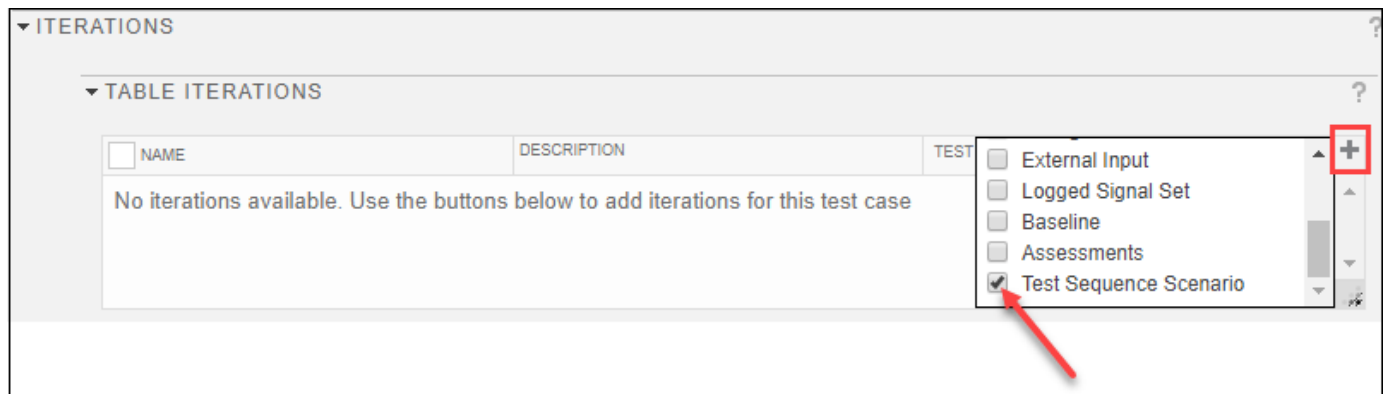
Signal Editor scenario [Model Settings] ▼ ↻ ↗

Test Sequence Block: ScenarioTest/Test Sequence ▼ ↻

Override with Scenario: Scenario_1 ▼ ↻

If you do not select a scenario, the active scenario in the Test Sequence block is used as the default.

7. In the **Iterations** section, expand **Table Iterations**, click the plus sign in the upper right of the table and select **Test Sequence Scenario** to add that column to the table.



8. Click **Add** at the bottom of the **Table Iterations** to add the individual iterations. The **Test Sequence Scenario** column for each iteration shows the default scenario. In this case, it shows [Default] Scenario_1.

Alternatively, click **Auto Generate** and select **Test Sequence Scenario** to generate an iteration for each scenario in the selected Test Sequence block. In the **Test Sequence Scenario** column, each iteration is assigned a separate iteration.

9. In the **Test Sequence Scenario** column of an iteration, click the scenario name to view a list of available scenarios. Select a different scenario from the default for one of the iterations.

10. Run the test.

See Also

sltest.harness.open

More About

- “Programmatically Create and Run Test Sequence Scenarios” on page 3-56
- “Test Sequence Editor” on page 3-30

Scenario Parameter Section

Use the options in the Scenario Parameter section to control the active scenario with a variable in the base workspace, model workspace, or a data dictionary.

- 1** Enable **Control active scenario from workspace**. The previously active scenario is deactivated and no active scenario is shown in the Scenario List. When you control the active scenario using a workspace variable, the Test Sequence Editor does not know which scenario is active until you click **Run**.
- 2** If desired, edit `Active_Scenario_Index` to change the variable name.
- 3** Create a variable with the same name in the base workspace, model workspace, or a data dictionary. For example, in the base workspace, create a variable named `Active_Scenario_Index`.
- 4** Set the value of the variable to the index of the scenario to activate. For example, in the base workspace, use `Active_Scenario_Index = Simulink.Parameter(2)` to activate the second scenario.
- 5** Click **Run** in the Test Sequence Editor to run the scenario.

Test Sequence and Assessment Syntax

In this section...

“Assessment Statements” on page 3-68
 “Temporal Operators” on page 3-69
 “Transition Operators” on page 3-70
 “Signal Generation Functions” on page 3-71
 “Logical Operators” on page 3-73
 “Relational Operators” on page 3-74

This topic describes syntax used within Test Sequence and Test Assessment blocks, and Stateflow charts. In the blocks, you use this syntax for test step actions, transitions, and assessments. In charts, you use this syntax in states and transitions.

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

Test Sequence and Test Assessment blocks use MATLAB as the action language. You can also use strings, including string comparisons, in test sequence steps and transitions. You define actions, transitions, assessments with assessment operators, temporal operators, transition operators, signal generation functions, logical operators, and relational operators. Except for `verify`, Stateflow charts can use all operators in MATLAB or C as the action language. `verify` can be used only with MATLAB language and you cannot use strings in `verify` statements. For example:

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

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

- To transition when h changes to 0:

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

- To verify that x is greater than y:

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

Assessment Statements

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

Keyword	Statement Syntax	Description	Example
verify	<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 > y,... 'SimulinkTest:greaterThan',... 'x and y values are %d, %d',... x,y)</pre>

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

Syntax in the table uses these arguments:

expression

Logical statement assessed

Examples:

`h > 0 && k == 0`

identifier

Label applied to results in the Test Manager

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

Examples:

`'SimulinkTest:greaterThan'`

errorMessage

Label applied to messages in the diagnostic viewer

Value: String

Examples:

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

Temporal Operators

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

Operator	Syntax	Description	Example
et	et(TimeUnits)	The elapsed time of the test step in TimeUnits. Omitting TimeUnits returns the value in seconds.	The elapsed time of the test sequence step in milliseconds: <code>et(msec)</code>

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

Syntax in the table uses these arguments:

TimeUnits

The units of time

Value: sec | msec | usec

Examples:

msec

Condition

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

Examples:

u > 0

x <= 1.56

Transition Operators

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

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

The following table lists common functions you can use in the Test Sequence block to create test signals, random number values, and natural exponents. It also describes the latch function, which saves and returns a specific value evaluated within a test sequence step. For more information about each function, click its name in the first column.

Some signal generation functions use the temporal operator `et`, which is the elapsed time of the test step in seconds. For additional operators related to `et` that you can use in test sequence steps, see “Temporal Operators” on page 3-37.

Note Scaling, rounding, and other approximations of argument values can affect function outputs.

Function	Syntax	Description	Example
sin	sin(x)	Returns the sine of x, where x is in radians.	A sine wave with a period of 10 sec: sin(et*2*pi/10)
cos	cos(x)	Returns the cosine of x, where x is in radians.	A cosine wave with a period of 10 sec: cos(et*2*pi/10)

Function	Syntax	Description	Example
square	square(x)	<p>Square wave output with a period of 1 and range -1 to 1.</p> <p>Within the interval $0 \leq x < 1$, square(x) returns the value 1 for $0 \leq x < 0.5$ and -1 for $0.5 \leq x < 1$.</p> <p>square is not supported in Stateflow charts.</p>	<p>Output a square wave with a period of 10 sec:</p> <p>square(et/10)</p>
sawtooth	sawtooth(x)	<p>Sawtooth wave output with a period of 1 and range -1 to 1.</p> <p>Within the interval $0 \leq x < 1$, sawtooth(x) increases.</p> <p>sawtooth is not supported in Stateflow charts.</p>	<p>Output a sawtooth wave with a period of 10 sec:</p> <p>sawtooth(et/10)</p>
triangle	triangle(x)	<p>Triangle wave output with a period of 1 and range -1 to 1.</p> <p>Within the interval $0 \leq x < 0.5$, triangle(x) increases.</p> <p>triangle is not supported in Stateflow charts.</p>	<p>Output a triangle wave with a period of 10 sec:</p> <p>triangle(et/10)</p>
ramp	ramp(x)	<p>Ramp signal of slope 1, returning the value of the ramp at time x.</p> <p>ramp(et) effectively returns the elapsed time of the test step.</p> <p>ramp is not supported in Stateflow charts.</p>	<p>Ramp one unit for every 5 seconds of test step elapsed time:</p> <p>ramp(et/5)</p>
heaviside	heaviside(x)	<p>Heaviside step signal, returning 0 for $x < 0$ and 1 for $x \geq 0$.</p> <p>heaviside is not supported in Stateflow charts.</p>	<p>Output a heaviside signal after 5 seconds:</p> <p>heaviside(et-5)</p>
exp	exp(x)	<p>Returns the natural exponential function, e^x.</p>	<p>An exponential signal progressing at one tenth of the test step elapsed time:</p> <p>exp(et/10)</p>

Function	Syntax	Description	Example
rand	rand	Uniformly distributed pseudorandom values	Generate new random values for each simulation by declaring rand extrinsic with <code>coder.extrinsic</code> . Assign the random number to a local variable. For example: <pre>coder.extrinsic('rand') nr = rand sg = a + (b-a)*nr</pre>
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>
latch	latch(x)	Saves the value of x at the first time <code>latch(x)</code> evaluates in a test step, and subsequently returns the saved value of x. Resets the saved value of x when the step exits. Reevaluates <code>latch(x)</code> when the step is next active. <p><code>latch</code> is not supported in Stateflow charts.</p>	Latch b to the value of torque: <pre>b = latch(torque)</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	<code>~p</code>	not p	<code>verify(~p)</code>
Conjunction	<code>p && q</code>	p and q	<code>verify(p && q)</code>
Disjunction	<code>p q</code>	p or q	<code>verify(p q)</code>
Implication	<code>~p q</code>	if p, q. Logically equivalent to implication $p \rightarrow q$.	<code>verify(~p q)</code>

Operation	Syntax	Description	Example
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 && q) (~p && ~q))</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”. 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(x > y)</code>
$x < y$	Less than	<code>verify(x < y)</code>
$x \geq y$	Greater than or equal to	<code>verify(x >= y)</code>
$x \leq y$	Less than or equal to	<code>verify(x <= y)</code>
$x == y$	Equal to	<code>verify(x == y)</code>
$x ~= y$	Not equal to	<code>verify(x ~= y)</code>

See Also

Related Examples

- “Assess Model Simulation Using `verify` Statements” on page 3-18
- “Transitions, Temporal Operators, and Messages in Test Sequence Blocks” on page 3-37
- “Generate Test Signals” on page 3-44
- “Programmatically Create a Test Sequence” on page 3-52

Debug a Test Sequence

In this section...

“View Test Step Execution During Simulation” on page 3-75

“Set Breakpoints to Enable Debugging” on page 3-75

“View Data Values During Simulation” on page 3-76

“Step Through Simulation” on page 3-76


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`, which is in the `matlab/help//toolbox/sltest/examples` folder. To open the model, enter

```
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 regular or conditional breakpoints. Regular 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. Conditional breakpoints halt simulation only when the specified condition is met. When simulation halts, you can view the data used in the test sequence to investigate the sequence simulation behavior.

You can add regular and conditional breakpoints to test step actions and transitions.

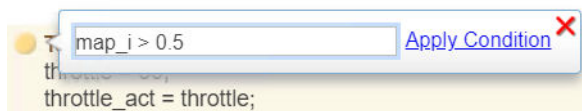
- To add a regular breakpoint to a test step, right-click the step or action and select **Break while executing step**. For a transition, point to the transition, click the gear icon , and select **Break when transition taken**. A red icon indicates a regular breakpoint.

Step	Transition
<p> Stabilize_Engine</p> <pre>throttle = 20; % 0 ==> sensor fai speed = 300; % 0 ==> sensor fai ego = ego_i; % 12 ==> sensor fa map = map_j; % 0 ==> sensor fai</pre>	<p>1. after(10, sec)</p>

Step	Transition
Stabilize_Engine throttle = 20; % 0 ==> sensor f speed = 300; % 0 ==> sensor f ego = ego_i; % 12 ==> sensor f map = map_i; % 0 ==> sensor f	1. after(10, sec)

- To add a conditional breakpoint, first add a regular breakpoint. Then, right-click the breakpoint icon and select **Set or Modify Condition**.

In the text field of the dialog box, specify the condition to apply to the step or transition and click **Apply Condition**. To indicate that it is a conditional breakpoint, the icon color changes to yellow.

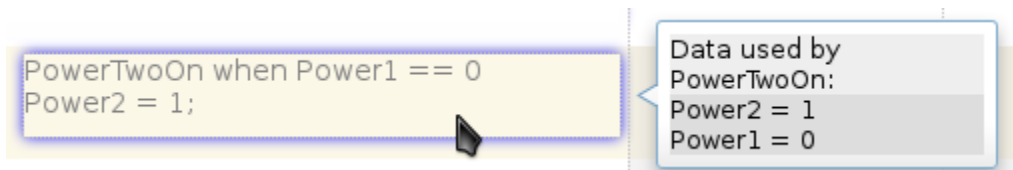


- To change a conditional breakpoint back to a regular breakpoint, right-click the breakpoint icon and select **Set or Modify Condition**. Delete the conditional text and click **Apply Condition**. The breakpoint icon color changes to red.
- You can remove a breakpoint using these methods:
 - Click the breakpoint icon.
 - Right-click the breakpoint icon and select **Clear Breakpoint**.
 - For a step or action, right-click the step breakpoint icon and deselect **Break while executing step**. For a transition, point to the transition, click the gear, and deselect **Break when transition taken**.

After adding breakpoints, simulate the test sequence by clicking **Run**.

View Data Values During Simulation






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



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

Step Through Simulation

When simulation halts, you can step through the test sequence using the toolbar buttons.

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

See Also

Test Sequence | “Test Sequence Editor” on page 3-30

Test Downshift Points of a Transmission Controller

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

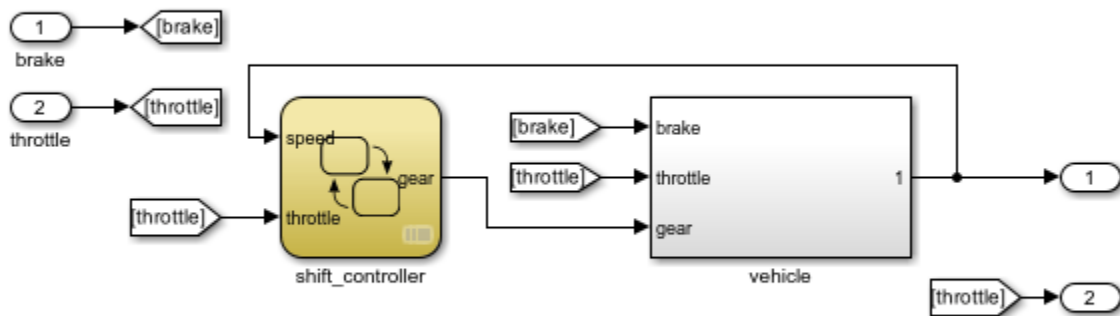
The Model and Controller

This example uses the `TransmissionDownshiftTestSequence` model, which is a simplified drivetrain system arranged in a controller-plant configuration. The objective is to unit test the downshift behavior of the transmission controller.

The Test

The controller should downshift between gear ratios in response to a increasing 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.

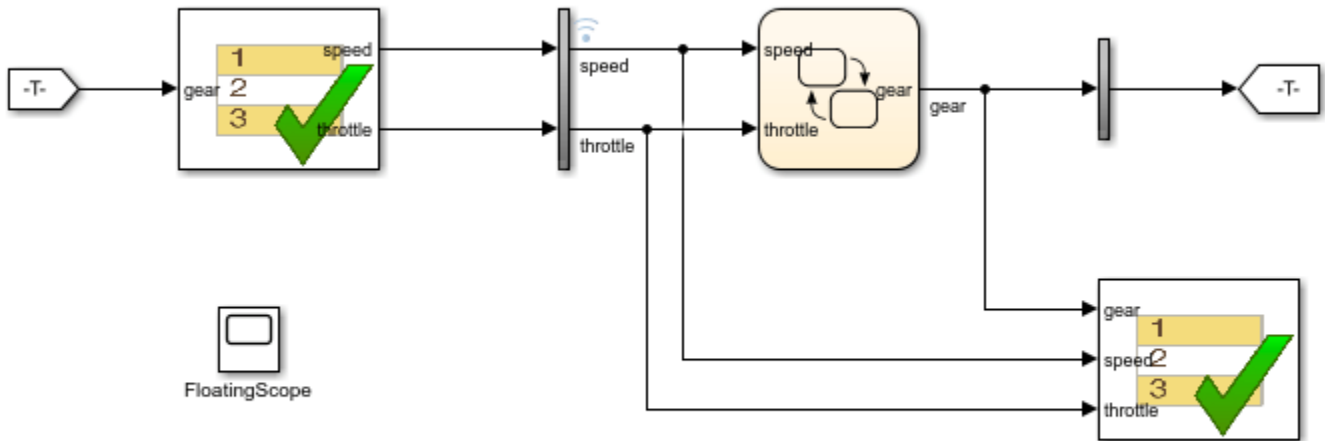
Testing Downshift Points of a Transmission Controller



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`. The test harness contains a Test Sequence block and a Test Assessment block connected to the controller subsystem.



Copyright 2014-2017 The MathWorks, Inc.

The Test Sequence

Double-click the Test Sequence block to open the Test Sequence Editor.

The test sequence ramps speed to 75 to initialize the controller in 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 value shall be greater than or equal to 0.
- Gear value shall be greater than 0.
- Throttle value 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 an assertion, 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 `verify` statements corresponding to the maximum speeds in gears 3, 2, and 1:

- Vehicle speed shall not exceed 90 in gear 3.
- Vehicle speed shall not exceed 50 in gear 2.
- Vehicle speed shall not exceed 30 in gear 1.

A When decomposition sequence contains the `verify` statements. In the When decomposition sequence, signal conditions determine the active step. A step includes a condition preceded by the `when` operator. The last step `Else` covers undefined conditions and does not use a `when` statement. For more information on When decomposition, see "Transition Types" in "Test Sequence Basics" on page 3-2.

```
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')
```

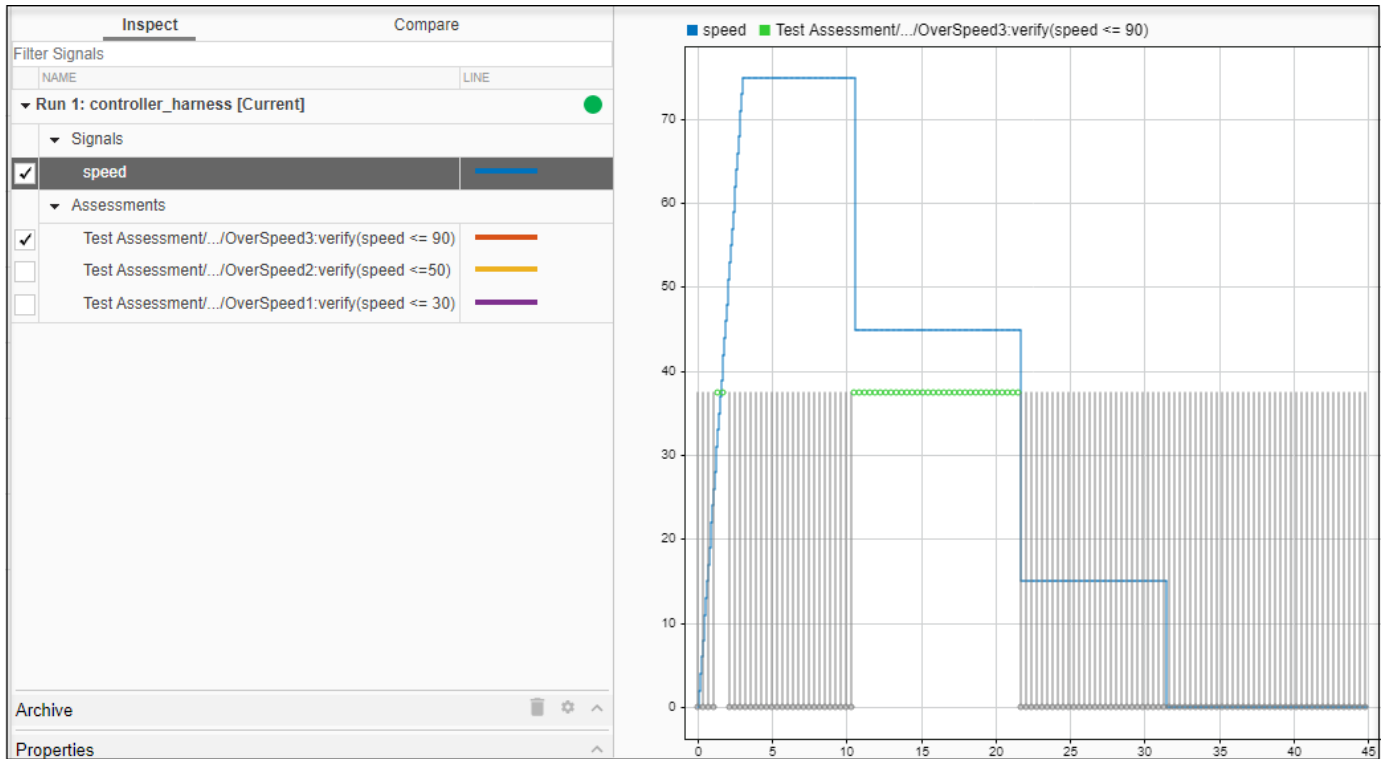
Step	Transition	Next Step
<pre>AssertConditions % These conditions ensure simulation validity. assert(speed >= 0, 'speed must be >= 0'); assert(throttle >= 0, 'throttle must be >= 0 and <= 100'); assert(throttle <= 100, 'throttle must be >= 0 and <= 100'); assert(gear > 0, 'gear must be > 0');</pre>		
<pre>OverSpeed3 when gear==3 % Verify speed within specified range for 3rd gear verify(speed <= 90, 'Engine overspeed in gear 3')</pre>		
<pre>OverSpeed2 when gear==2 % Verify speed within specified range for 2nd gear verify(speed <= 50, 'Engine overspeed in gear 2')</pre>		
<pre>OverSpeed1 when gear==1 % Verify speed within specified range for 1st gear verify(speed <= 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.



Examine Model Verification Results by Using Simulation Data Inspector

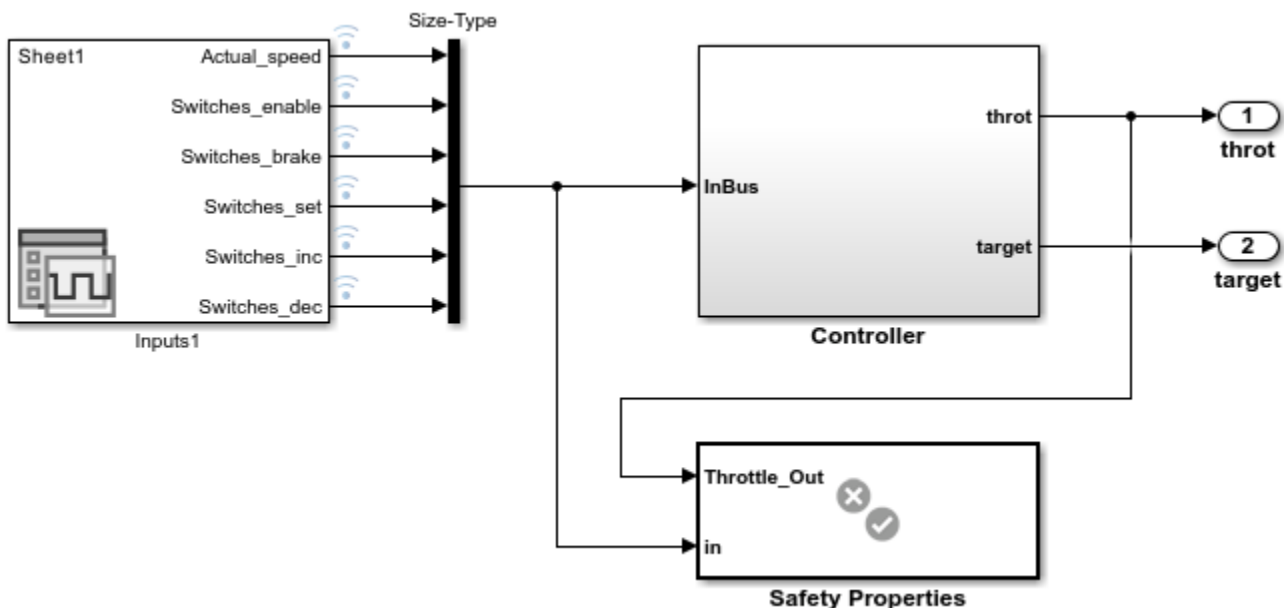
This example shows how to use the Simulation Data Inspector to view the output from a model verification block in a system under test. If you have Simulink® Test™, model verification blocks return Pass, Fail, or Untested results at each time step. By examining the results of a model verification block, you can:

- Determine the simulation time when a failure occurs.
- Compare the verification results with other relevant signals.
- Trace failures from the Simulation Data Inspector back to the model.

For more information, see “Model Verification Blocks” on page 3-16.

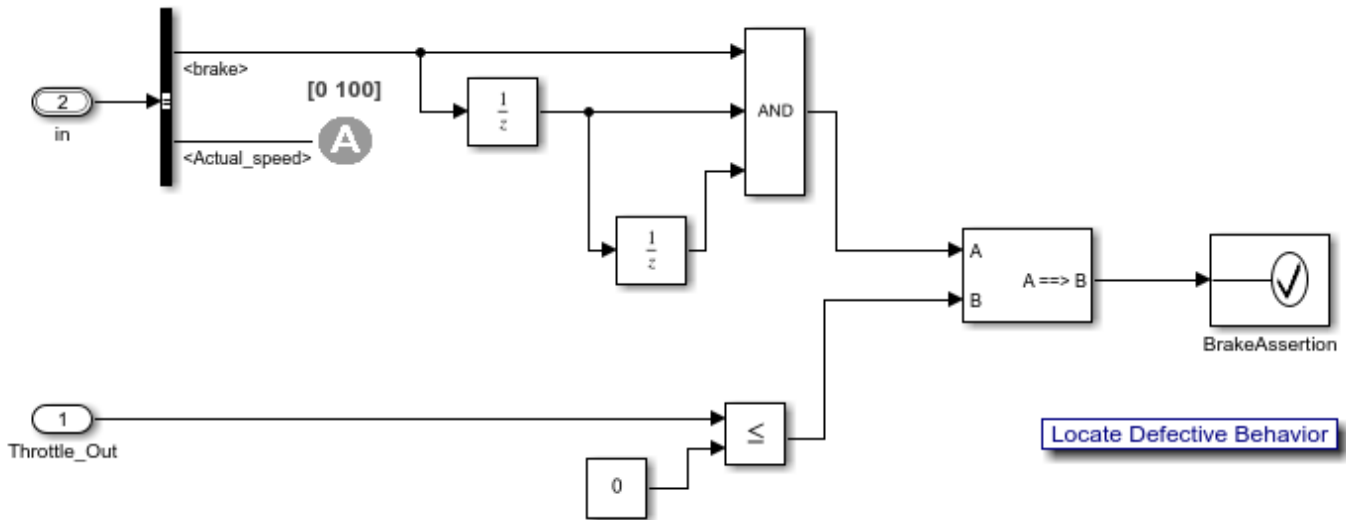
Verify Model Behavior With Assertion Block

In this example, the subsystem block Controller models the cruise control system in a car. This subsystem outputs the throttle value based on the difference between the actual and target speeds.



The verification subsystem Safety Properties uses an Assertion block to check that the system disengages when the brake is applied for three consecutive time steps.

Property: When the brake is applied for three consecutive steps, the throttle goes to zero.

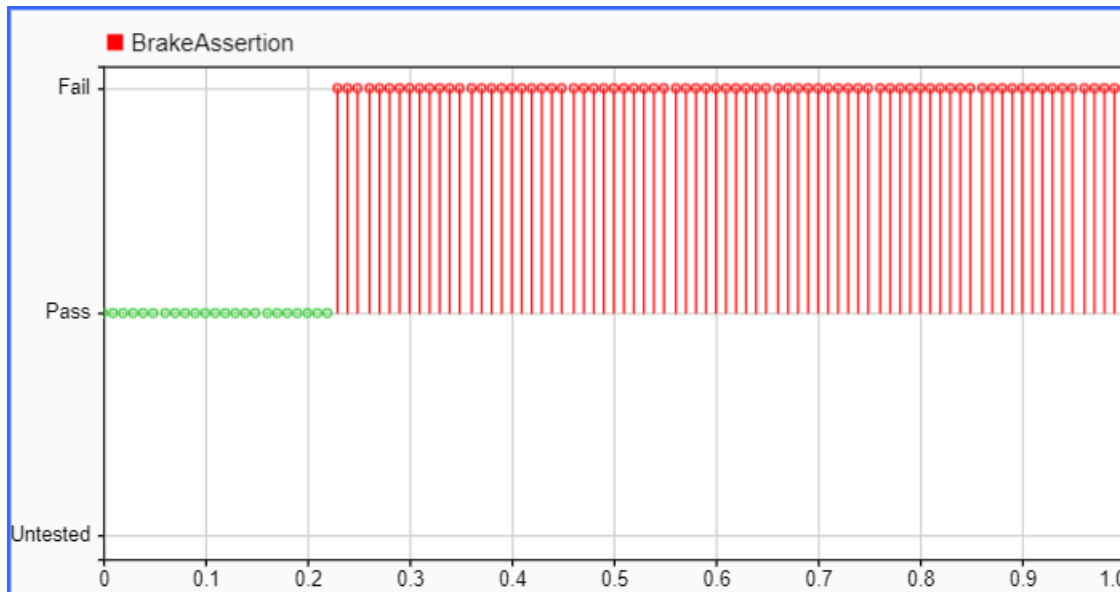


Determine Simulation Time of Failure

Simulate the model and view the output of the Assertion block in the Simulation Data Inspector.

- 1 In the **Simulation** tab, click **Run**.
- 2 In the **Simulation** tab, under **Review Results**, select **Data Inspector**.
- 3 In the Simulation Data Inspector navigation pane, select BrakeAssertion.

The results show that the assertion fails at 0.23 seconds.

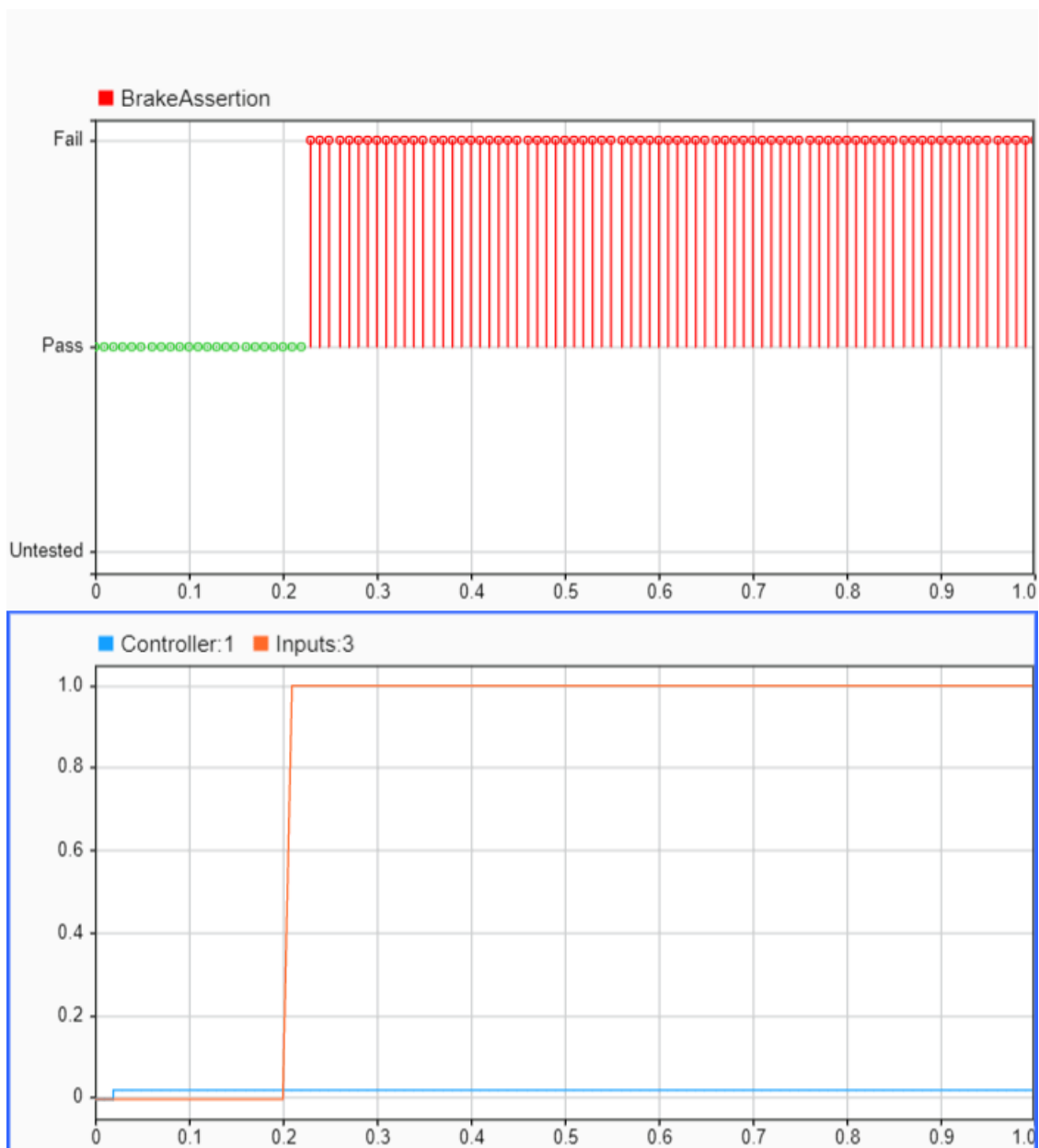


Compare Verification Results with Other Signals

Examine the cause of the failure by plotting the values of the brake and throttle signals.

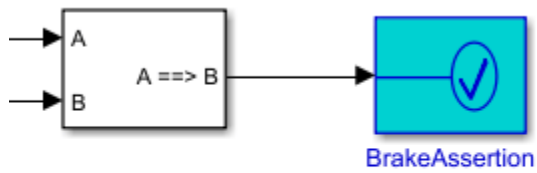
- 1 Right-click the `throt` signal and select **Log Selected Signals**.
- 2 Simulate the model.
- 3 Configure the Simulation Data Inspector with two subplots.
- 4 In the Simulation Data Inspector navigation pane, select the signals to plot. For the first subplot, select `BrakeAssertion`. For the second subplot, select `Controller:1` (throttle) and `Inputs:3` (brake).

The results show that pressing the brake at 0.2 seconds does not disengage the throttle.



Trace Failure Back to the Model

Find the block that produces a verification result by tracing the result from the Simulation Data Inspector back to the model. In the Simulation Data Inspector navigation pane, right-click `BrakeAssertion` and select **Highlight in Model**. The editor opens the verification subsystem and highlights the Assertion block.



Locate Defective Behavior

See Also

`sltest.getAssessments` | Assertion | Implies (Simulink Design Verifier) | Proof Assumption

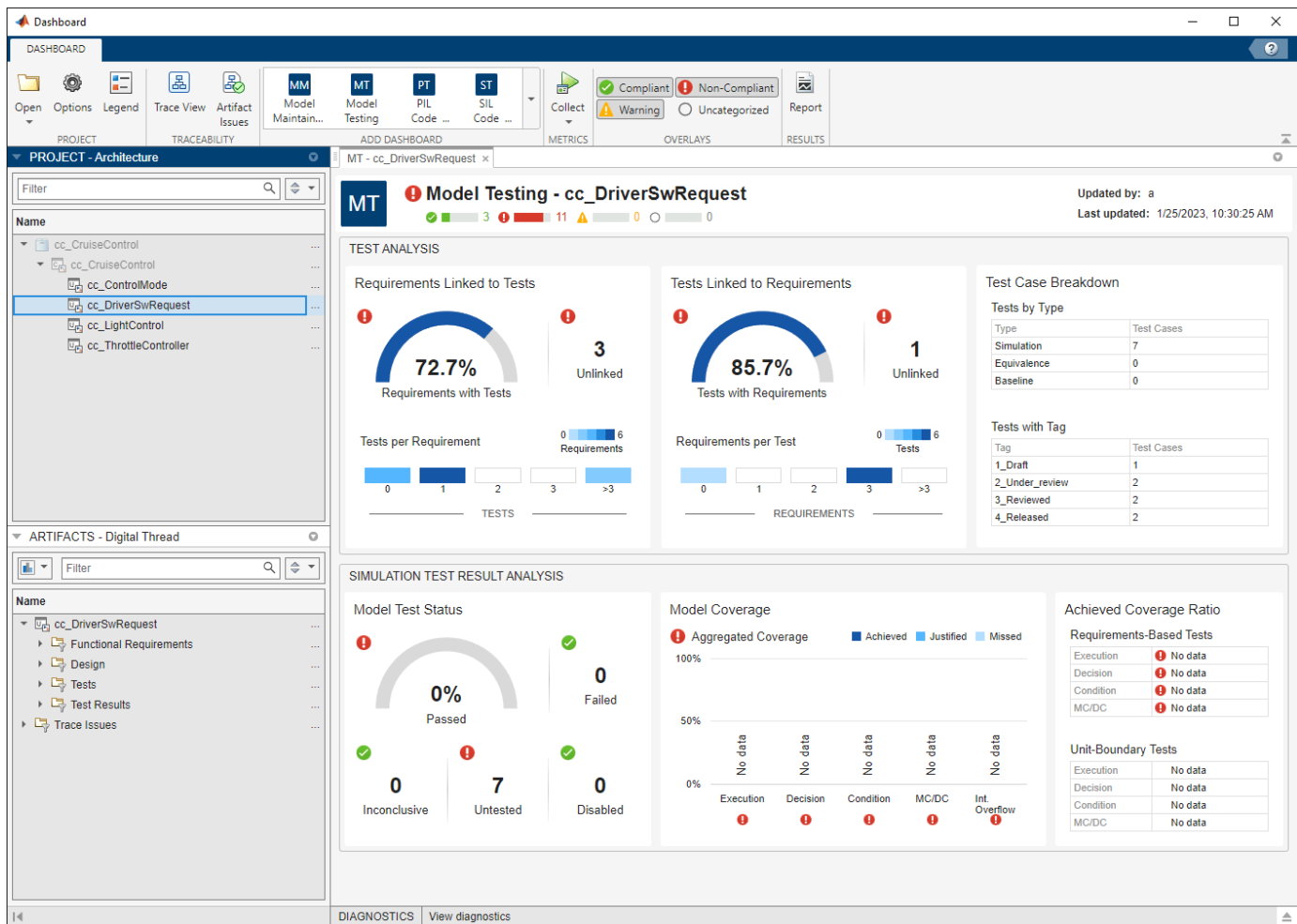
Fix Requirements-Based Testing Issues

This example shows how to address common traceability issues in model requirements and tests by using the Model Testing Dashboard. The dashboard analyzes the testing artifacts in a project and reports metric data on quality and completeness measurements such as traceability and coverage, which reflect guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. The dashboard widgets summarize the data so that you can track your requirements-based testing progress and fix the gaps that the dashboard highlights. You can click the widgets to open tables with detailed information, where you can find and fix the testing artifacts that do not meet the corresponding standards.

Collect Metrics for the Testing Artifacts in a Project

The dashboard displays testing data for a model and the artifacts that the unit traces to within a project. For this example, open the project and collect metric data for the artifacts.

- 1 Open the project that contains the models and testing artifacts. For this example, in the MATLAB® Command Window, enter `dashboardCCProjectStart("incomplete")`.
- 2 Open the Dashboard window. To open the Model Testing Dashboard: on the **Project** tab, click **Model Testing Dashboard** or enter `modelTestingDashboard` at the command line.
- 3 In the **Project** panel, the dashboard organizes unit models under the component models that contain them in the model hierarchy. View the metric results for the unit `cc_DriverSwRequest`. In the **Project** panel, click the name of the unit, **cc_DriverSwRequest**. When you initially select **cc_DriverSwRequest**, the dashboard collects the metric results for uncollected metrics and populates the widgets with the data for the unit.



Link a Requirement to its Implementation in a Model

The **Artifacts** panel shows artifacts such as requirements, tests, and test results that trace to the unit selected in the **Project** panel.

In the **Artifacts** panel, the **Trace Issues** folder shows artifacts that do not trace to unit models in the project. The **Trace Issues** folder contains subfolders for:

- **Unexpected Implementation Links** — Requirement links of **Type Implements** for a requirement of **Type Container** or **Type Informational**. The dashboard does not expect these links to be of **Type Implements** because container requirements and informational requirements do not contribute to the Implementation and Verification status of the requirement set that they are in. If a requirement is not meant to be implemented, you can change the link type. For example, you can change a requirement of **Type Informational** to have a link of **Type Related to**.
- **Unresolved and Unsupported Links** — Requirement links which are broken or not supported by the dashboard. For example, if a model block implements a requirement, but you delete the model block, the requirement link is now unresolved. The Model Testing Dashboard does not support traceability analysis for some artifacts and some links. If you expect a link to trace to a unit and it does not, see the troubleshooting solutions in “Resolve Missing Artifacts, Links, and Results” (Simulink Check).

- **Untraced Tests** — Tests that execute on models or subsystems that are not on the project path.
- **Untraced Results** — Results that the dashboard can no longer trace to a test. For example, if a test produces results, but you delete the test, the results can no longer be traced to the test.


Address Testing Traceability Issues

The widgets in the **Test Analysis** section of the Model Testing Dashboard show data about the unit requirements, tests for the unit, and links between them. The widgets indicate if there are gaps in testing and traceability for the implemented requirements.

Link Requirements and Tests

For the unit `cc_DriverSwRequest`, the **Tests Linked to Requirements** section shows that some of the tests are missing links to requirements in the model.

To see detailed information about the missing links, in the **Tests Linked to Requirements** section, click the widget **Unlinked**. The dashboard opens the **Metric Details** for the widget with a table of metric values and hyperlinks to each related artifact. The table shows the tests that are implemented in the unit, but do not have links to requirements. The table is filtered to show only tests that are missing links to requirements.



Metric Details - Tests linked to requirements

Metric that determines if each test case or test iteration for the model is linked to at least one requirement in the project.

Artifact	Source	Requirement Link Status
Detect long decrement	cc_DriverSwRequest_Tests.mldatx	Missing linked requirements

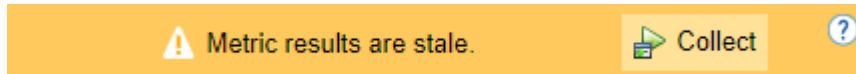
The test `Detect long decrement` is missing linked requirements.

- 1 In the **Artifact** column of the table, point to **Detect long decrement**. The tooltip shows that the test **Detect long decrement** is in the test suite **Unit test for DriverSwRequest**, in the test file **cc_DriverSwRequest_Tests**.
- 2 Click **Detect long decrement** to open the test in the Test Manager. For this example, the test needs to link to three requirements that already exist in the project. If there were not already requirements, you could add a requirement by using the Requirements Editor.
- 3 Open the software requirements in the Requirements Editor. In the **Artifacts** panel of the Dashboard window, expand the folder **Functional Requirements > Implemented** and double-click the requirement file **cc_SoftwareReqs.slreqx**.
- 4 View the software requirements in the container with the summary **Driver Switch Request Handling**. Expand **cc_SoftwareReqs > Driver Switch Request Handling**.
- 5 Select multiple software requirements. Hold down the **Ctrl** key as you click **Output request mode**, **Avoid repeating commands**, and **Long Increment/Decrement Switch recognition**. Keep these requirements selected in the Requirements Editor.
- 6 In the Test Manager, expand the **Requirements** section for the test `Detect long decrement`. Click the arrow next to the **Add** button and select **Link to Selected Requirement**. The traceability link indicates that the test `Detect long decrement` verifies the three requirements **Output request mode**, **Avoid repeating commands**, and **Long Increment/Decrement Switch recognition**.

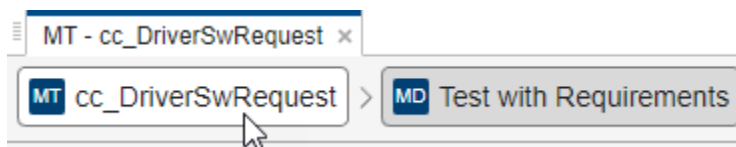
- 7 The metric results in the dashboard reflect only the saved artifact files. To save the test suite `cc_DriverSwRequest_Tests.mldatx`, in the **Test Browser**, right-click `cc_DriverSwRequest_Tests` and click **Save**.

Refresh Metric Results in the Dashboard

The dashboard detects that the metric results are now stale and shows a warning banner at the top of the dashboard.



- 1 Click the **Collect** button on the warning banner to re-collect the metric data so that the dashboard reflects the traceability link between the test and requirements.
- 2 View the updated dashboard widgets by returning to the **Model Testing** results. At the top of the dashboard, there is a breadcrumb trail from the **Metric Details** back to the **Model Testing** results. Click the breadcrumb button for `cc_DriverSwRequest` to return to the **Model Testing** results for the unit.



The **Tests Linked to Requirements** section shows that there are no unlinked tests. The **Requirements Linked to Tests** section shows that there are 3 unlinked requirements. Typically, before running the tests, you investigate and address these testing traceability issues by adding tests and linking them to the requirements. For this example, leave the unlinked artifacts and continue to the next step of running the tests.

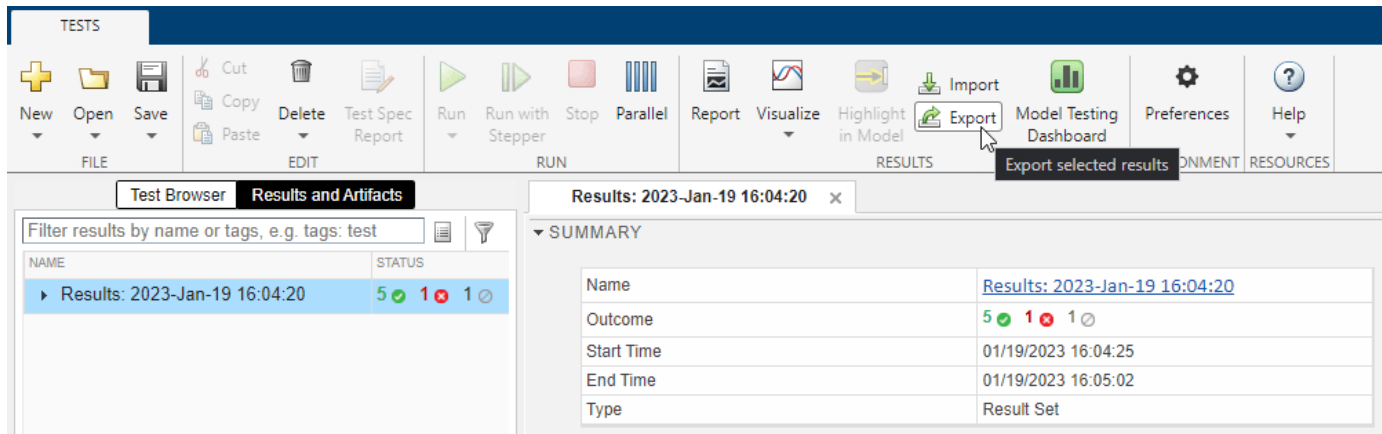
Test the Model and Analyze Failures and Gaps

After you create and link unit tests that verify the requirements, run the tests to check that the functionality of the model meets the requirements. To see a summary of the test results and coverage measurements, use the widgets in the **Simulation Test Result Analysis** section of the dashboard. The widgets help show testing failures and gaps. Use the metric results to analyze the underlying artifacts and to address the issues.

Perform Unit Testing

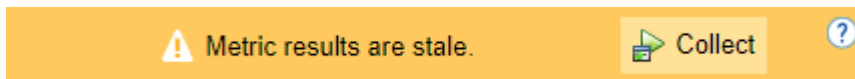
Run the tests for the model by using the Test Manager. Save the test results in your project and review them in the Model Testing Dashboard.

- 1 Open the unit tests for the model in the Test Manager. In the Model Testing Dashboard, in the **Artifacts** panel, expand the folder **Tests > Unit Tests** and double-click the test file `cc_DriverSwRequest_Tests.mldatx`.
- 2 In the Test Manager, click **Run**.
- 3 Select the results in the **Results and Artifacts** pane.
- 4 Save the test results as a file in the project. On the **Tests** tab, in the **Results** section, click **Export**. Name the results file `Results1.mldatx` and save the file under the project root folder.



The Model Testing Dashboard detects the results and automatically updates the **Artifacts** panel to include the new test results for the unit in the subfolder **Test Results > Model**.

The dashboard also detects that the metric results are now stale and shows a warning banner at the top of the dashboard.



The **Stale** icon **STALE** appears on the widgets in the **Simulation Test Result Analysis** section to indicate that they are showing stale data that does not include the changes.

Click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current artifacts.

Address Testing Failures and Gaps

For the unit `cc_DriverSwRequest`, the **Model Test Status** section of the dashboard indicates that one test failed and one test was disabled during the latest test run.

- 1 To view the disabled test, in the dashboard, click the **Disabled** widget. The table shows the disabled tests for the model.
- 2 Open the disabled test in the Test Manager. In the table, click the test artifact **Detect long decrement**.
- 3 Enable the test. In the **Test Browser**, right-click the test and click **Enabled**.
- 4 Re-run the test. In the **Test Browser**, right-click the test and click **Run** and save the test suite file.
- 5 View the updated number of disabled tests. In the dashboard, click the **Collect** button on the warning banner. Note that there are now zero disabled tests reported in the **Model Test Status** section of the dashboard.
- 6 View the failed test in the dashboard. Click the breadcrumb button for `cc_DriverSwRequest` to return to the **Model Testing** results and click the **Failed** widget.
- 7 Open the failed test in the Test Manager. In the table, click the test artifact **Detect set**.
- 8 Examine the test failure in the Test Manager. You can determine if you need to update the test or the model by using the test results and links to the model. For this example, instead of fixing the

failure, use the breadcrumbs in the dashboard to return to the **Model Testing** results and continue on to examine test coverage.

Check if the tests that you ran fully exercised the model design by using the coverage metrics. For this example, the **Model Coverage** section of the dashboard indicates that some conditions in the model were not covered. Place your cursor over the **Decision** bar in the widget to see what percent of condition coverage was achieved.

- 1 View details about the decision coverage by clicking one of the **Decision** bars. For this example, click the **Decision** bar for **Achieved** coverage.
- 2 In the table, expand the model artifact. The table shows the test results for the model and the results files that contains them. For this example, click on the hyperlink to the source file **Results1.mldatx** to open the results file in the Test Manager.
- 3 To see detailed coverage results, use the Test Manager to open the model in the Coverage perspective. In the Test Manager, in the **Aggregated Coverage Results** section, in the **Analyzed Model** column, click **cc_DriverSwRequest**.
- 4 Coverage highlighting on the model shows the points that were not covered by the tests. For this example, do not fix the missing coverage. For a point that is not covered in your project, you can add a test to cover it. You can find the requirement that is implemented by the model element or, if there is none, add a requirement for it. Then you can link the new test to the requirement. If the point should not be covered, you can justify the missing coverage by using a filter.

Once you have updated the unit tests to address failures and gaps in your project, run the tests and save the results. Then examine the results by collecting the metrics in the dashboard.

Iterative Requirements-Based Testing with the Model Testing Dashboard

In a project with many artifacts and traceability connections, you can monitor the status of the design and testing artifacts whenever there is a change to a file in the project. After you change an artifact, use the dashboard to check if there are downstream testing impacts by updating the tracing data and metric results. Use the **Metric Details** tables to find and fix the affected artifacts. Track your progress by updating the dashboard widgets until they show that the model testing quality meets the standards for the project.

Assess Temporal Logic by Using Temporal Assessments

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

To work with temporal assessments in the Test Manager:

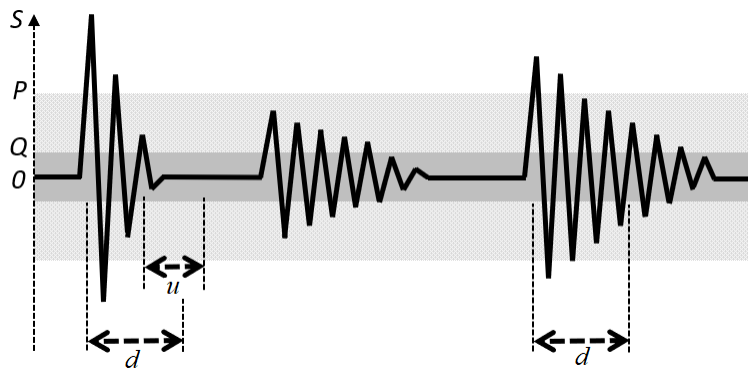
- 1 Select an assessment template.
- 2 Enter the assessment conditions.
 - Map symbols to model elements, such as signals, time series, or constants.
 - View the assessment summary.

You can copy, paste, and delete assessments by selecting the summary and right-clicking to display the context menu or by using keyboard shortcuts. When you copy and paste an assessment, it is added to the end of the assessments. If you delete an assessment, you cannot paste it even if you copied it before deleting it.

- 3 Run the test case.
- 4 Use the results to assess the system under test (SUT) against your requirements.

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

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

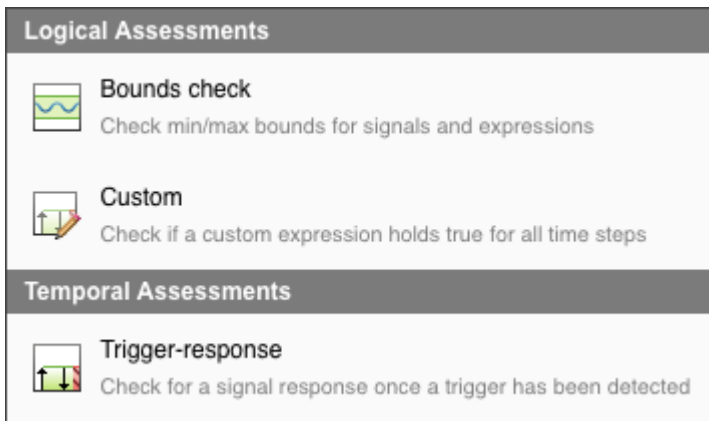


Create a Temporal Assessment

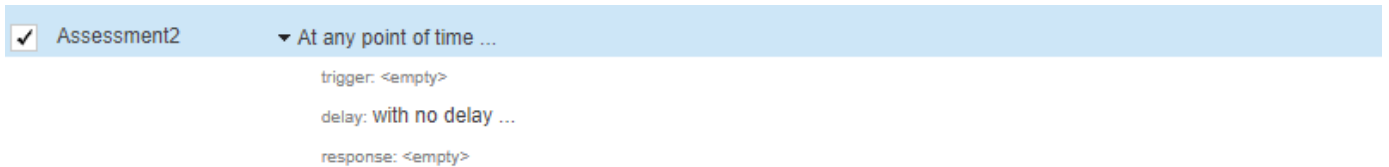
To create a temporal assessment:

- 1 Create or open a test case in the Test Manager.
- 2 Navigate to the **Logical and Temporal Assessments** Editor.
- 3 Click **Add Assessment**. These assessment templates are available:
 - **Logical Assessment Templates**
 - **Bounds Check** — Check maximum and minimum bounds for signals and expressions.

- **Custom** — Check if a logical expression holds true for all time steps.
- **Temporal Assessment Template**
 - **Trigger-Response** — Check for a signal response when a trigger is detected.



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



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

Define Temporal Assessment Conditions

A Trigger-response assessment requires a:

- Trigger parameter
- Response parameter
- Optional Delay parameter

For the forced oscillation damping problem:

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

All time units are seconds.

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

Resolve Assessment Parameter Symbols

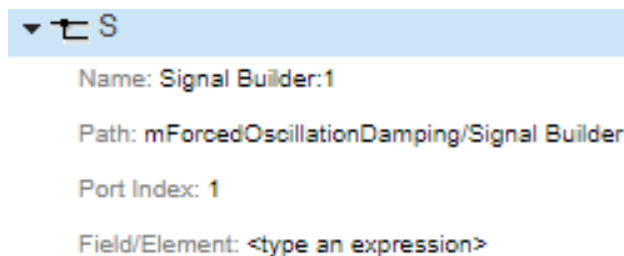
To resolve a symbol, right-click the symbol. Two options are available:

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



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

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



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

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

- To map a symbol to a bus signal containing a bus element `fieldA`, enter `.fieldA`.
- To map a symbol to the signal element that corresponds to index (5,5) in a signal array, enter `(5,5)`.
- To combine both expressions, enter `.fieldA(5,5)`.

- 2 **Map to expression** - Assign a scalar constant value or variable to a symbol.

When you select **Map to expression**, you must enter an expression in the **Expression** field. The expression must be in MATLAB code and evaluate to a scalar literal or a `timeseries` object. If the expression is long or complicated, you can use the **Assessment Callback** section to write MATLAB code that retrieves the model, test, and simulation data and assigns the data to variables, then assign the variable to the symbol expression. See “Define Variables in the Assessment Callback Section” on page 3-110 for more information on defining variables.

To retrieve data stored in workspace variables, use the `evalin` function. For example, to assign the workspace variable `var` to a symbol, enter `evalin("base", "var")` in the symbol

expression directly, or enter `v = evalin("base", "var")` in the **Assessment Callback** section and enter `v` in the symbol expression.

Because the `t` symbol is automatically bound to the simulation time, you do not need to map it to an expression. `t` is not visible in the **Symbols** pane.

Tip Entering `sig = sltest_simout.logout.get('mySignal')` in the **Assessment Callback** section and using **Map to expression** to map a symbol to the `sig` variable is equivalent to using **Map to model element** to map a symbol to the `mySignal` signal.


If you map a symbol to a discrete data signal that is linearly interpolated, the interpolation is automatically changed to zero-order hold during the assessment evaluation. Additionally, an information icon (i) appears next to the symbol name in the **Symbols** pane. Point to the icon and a tooltip appears which indicates that the linear interpolation was overridden.

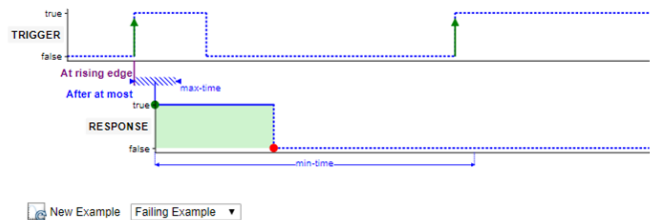
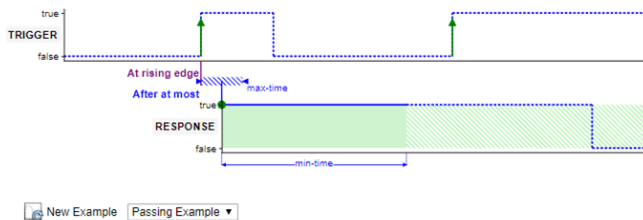
Review the Temporal Assessment Summary

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

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

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

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



Evaluate the SUT

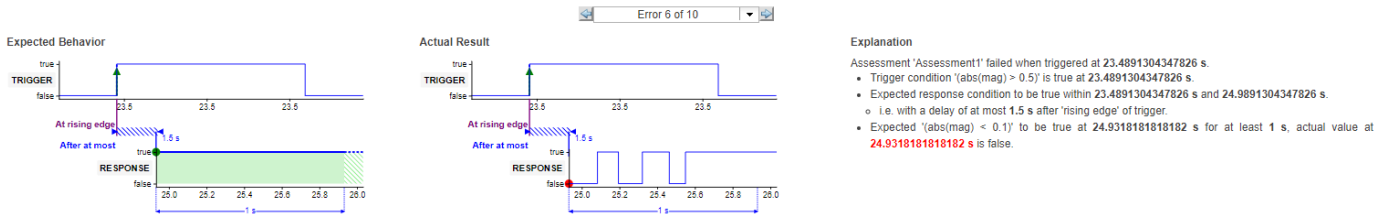
Run the test case to evaluate the SUT. Temporal assessments are evaluated after simulation by using logged signal data. Use the test case results to review the SUT against your requirements.

You can run test cases that contain logical or temporal assessments in multiple releases. For more information, see “Assess Temporal Logic in Multiple Releases” on page 6-98.

View Assessment Results

View the results of the assessment evaluation from the **Results and Artifacts** pane of the Test Manager. Select the test case and click the assessment in the **Results** tree to open a new

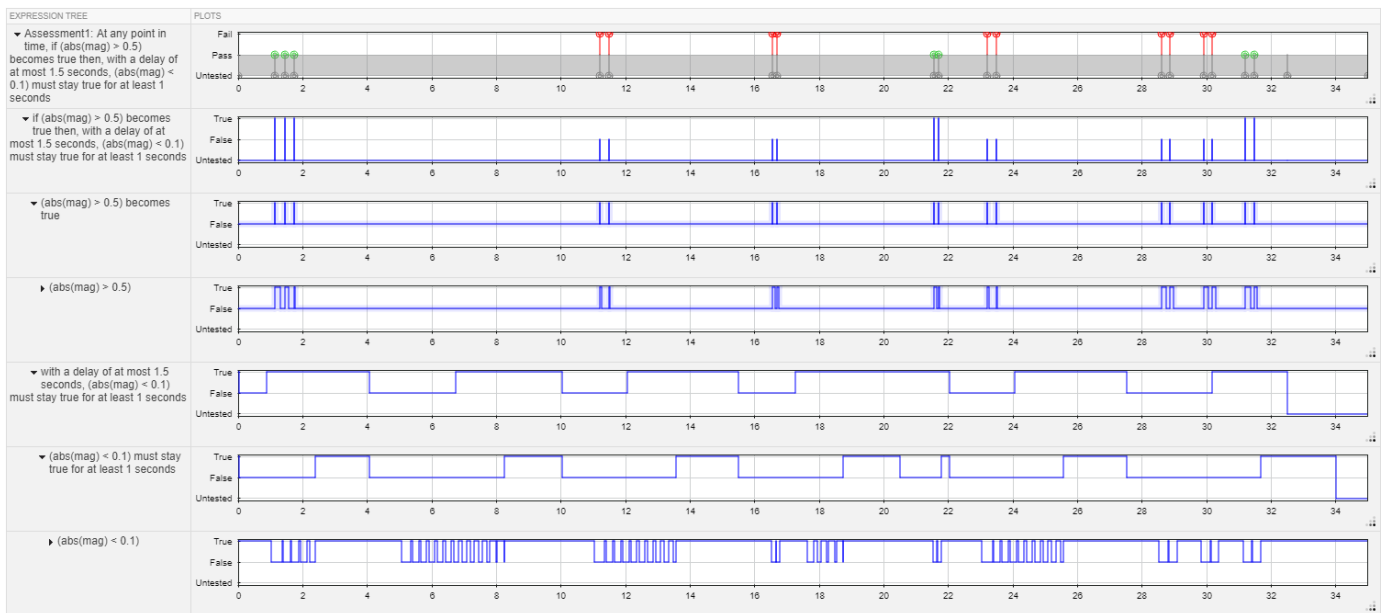
Assessment Result tab. Simulink Test evaluates the assessment and displays the expected behavior and the actual result of the assessment execution with a description of the assessment failures at different time steps.



Note The assessment result figures cannot be exported to a Simulink Test report.

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

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



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

Link Temporal Assessments to Requirements

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

See Also

`sltest.testmanager.Assessment` | `sltest.testmanager.AssessmentSymbol` |
`sltest.testmanager.TestCase`

More About

- “Logical and Temporal Assessment Syntax” on page 3-107
- “Assess Temporal Logic in Multiple Releases” on page 6-98

Test Traffic Light Control by Using Logical and Temporal Assessments

This example shows how to use logical and temporal assessments to test the signal logic for a two-light traffic intersection. It also shows how to minimize untested results.

The model used in this example represents a controller in a two-light traffic intersection. The changes between the traffic light states depend on the traffic lights and the timing delay parameters defined in a Stateflow® chart. For more information about the Stateflow logic used in the model, see “Monitor Chart Activity by Using Active State Data” (Stateflow).

Open and Run the Model

Open and simulate the model.

```
model = 'sltestTrafficLight';
open_system(model)
sim(model)
```

Plot the states of the two lights in the Simulation Data Inspector.

```
runData = Simulink.sdi.Run.getLatest;
LightState1 = getSignalsByName(runData, 'Light1');
LightState2 = getSignalsByName(runData, 'Light2');
Simulink.sdi.setSubPlotLayout(2,1);
plotOnSubPlot(LightState1,1,1,true);
plotOnSubPlot(LightState2,1,1,false);

plotOnSubPlot(LightState2,2,1,true);
plotOnSubPlot(LightState1,2,1,false);

Simulink.sdi.view
```

The traffic lights are approximately on opposite schedules. When one light is in the Green state, the other light is in the Red state, and vice versa. Additionally, the lights must pass through the Yellow state when transitioning from Green to Red.

To explore the Stateflow logic, consider:

- Adding a breakpoint in one of the two atomic subcharts to step through the logic. For more information on debugging state charts, see “Set Breakpoints to Debug Charts” (Stateflow).
- Inspecting the logic using the Sequence Viewer.
- Visualizing different signals using the Simulation Data Inspector.

Requirements and Enumeration Types of the Model

To test the logic of the traffic controller, the Stateflow chart outputs an enumerated type that corresponds to the Red, Yellow, or Green state of the light. By default, Stateflow automatically generates the enumerated type definition. To create a custom enumeration definition, see “Define State Activity Enumeration Type” (Stateflow).

Simulating the model creates the built-in enumeration definition. To confirm that the enumerated type definition `LightModeType` exists, use `which LightModeType`.

Design Requirements

The test file in this example tests the traffic light model against several requirements::

- The number of cars waiting at a light is always greater than or equal to zero.
- At any point in time, at least one of the lights is red.
- Every time the light becomes yellow, it stays yellow for a fixed amount of time within a tolerance value of 0.5 seconds before changing to red.
- Every time the light becomes green, it stays green between a minimum and maximum time within a tolerance of 0.5 seconds before changing to yellow.

The test file tests the requirements by using a bounds check assessment, a custom assessment, and two trigger-response assessments. For more information about linking to requirements, see “Link Temporal Assessments to Requirements” on page 3-97.

Run the Logical and Temporal Assessments

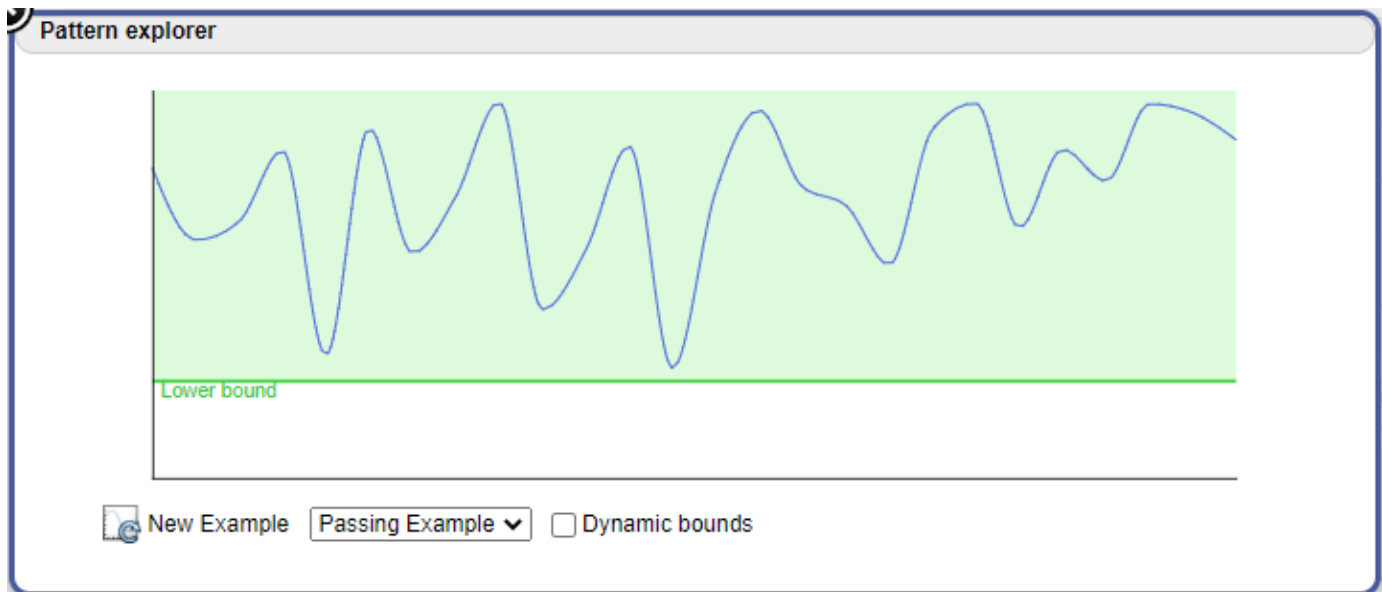
Load the test file and open the test assessments in the Test Manager.

```
sltest.testmanager.load('test_traffic.mldatx');  
sltest.testmanager.view;
```

Run the Bounds Check Test

Check whether the number of cars waiting at the light is always greater than or equal to zero by using a bounds check logical assessment. The symbol `NumCars` is mapped to the output of the `Car_Monitor1` subsystem. The `Car_Monitor1` subsystem outputs an `int32` type, so the lower-bound expression is cast as `int32(0)`. For more information about data type requirements, see “Data Types in Assessment Conditions” on page 3-110.

- 1 In the Test Manager, click **New Test Case 1** in the **Test Browser** pane.
- 2 Expand the **Logical and Temporal Assessments** section of the Test Manager
- 3 Select **Waiting Cars** in the table and verify that its **Assessment** logic is correct.
- 4 In the upper right of the **Visual Representation** pane, click the Explore Pattern icon to open the Pattern Explorer.
- 5 View the Passing and Failing Examples. This image shows a passing example:



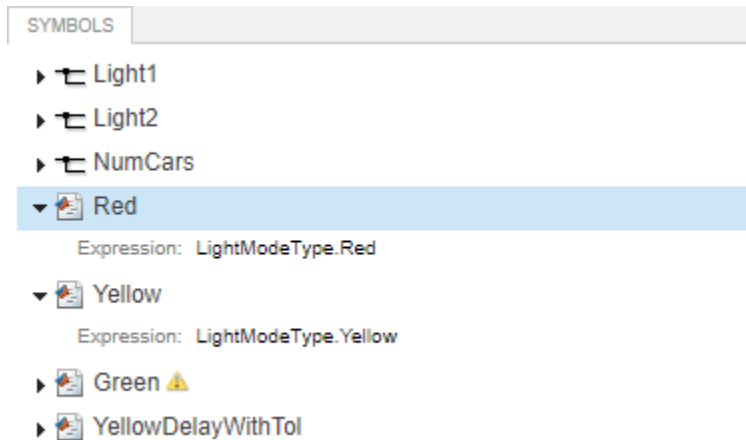
6. With the **Waiting cars** assessment selected, run the test case.
7. To view the results, expand the Results in the **Results and Artifacts** panel.
8. Select **New Test Case 1 > Logical and Temporal Assessments > Waiting cars**.

Run the Custom Logical Assessment

Compare the states of both lights at all time steps by using a custom logical assessment.

For safety reasons, at no point in time should both lights be green. In addition, other configurations are undesirable, such as one green light and one yellow light or two yellow lights. The assessment checks that one of the lights is always red.

In the **Logical and Temporal Assessments** section, this assessment uses the symbols, Red and Yellow, which each correspond to their respective color in the enumerated type definition. For instance, the Expression field for the Red symbol references the Red enumeration member of the LightModeType enumeration - LightModeType.Red. The Green symbol appears as an unused symbol because it is not used until you implement the Green to Yellow assessment. See Create a Trigger-Response Assessment to Evaluate Green to Yellow Transitions on page 3-104.



Note that the **Visual Representation** preview is blank because it is only available for bounds check and trigger-response assessments. Also, the custom-expression field of the custom check must follow the syntax rules described in “Logical and Temporal Assessment Conditions” on page 3-109.

- 1 Return to the **Test Browser** pane and expand the **Logical and Temporal Assessments** section.
- 2 Select **Both lights safety check**.
- 3 Rerun the test case.
- 4 To view the results, expand the Results in the **Results and Artifacts** pane and select **New Test Case 1 > Logical Temporal Assessments > Both lights safety check**.

Run the Trigger Response Logical Assessment

Use a trigger response logical assessment to assess the logic when `Light1` transitions from the `Yellow` state to the `Red` state. The assessment triggers when `Light1` enters the `Yellow` state. As shown in the Stateflow chart, for the `after(YELLOWDELAY, sec)` transitions in each atomic subchart, the state switches from `Yellow` to `Red` after a fixed delay of `YELLOWDELAY` seconds. To meet the requirements, the `YELLOWDELAY` value is adjusted by a tolerance value, `tol` for the assessment in the assessment callback.

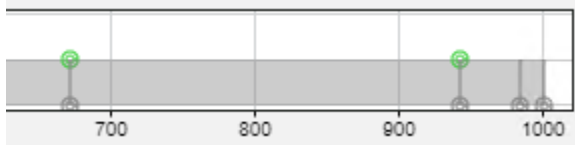


- 1 Under **Logical and Temporal Assessments**, enable **Light1 YellowToRed Transition** assessment.
- 2 Verify its **Assessment** logic summary.
- 3 Expand the summary and the trigger section and set **time-reference** to rising edge of trigger.
- 4 Run the test case.
- 5 In the **Results and Artifacts** pane, click **New Test Case 1 > Logical and Temporal Assessments > Light1 YellowToRed Transition**.
- 6 Observe that the assessment fails.

The Expected Behavior and Actual Result graphs show the assessment failures and the **Explanation** section describes the failures. The failures occur at the four points when the assessment triggers. In **Error 1 of 4**, the trigger condition becomes true at $t = 132.1$ and the **Explanation** section explains that the test expected the response condition to be true at 132.1 seconds. This result contradicts the requirement that the light stay yellow for a fixed amount of time before changing to red. The assessment fails because the evaluation of the response is at the rising edge of the trigger. The `Light1 == Red` response should not be evaluated until the trigger is false.

In the **Test Browser** pane, adjust the **time-reference** to falling edge of trigger. This setting ensures that the `Light1 == Red` response evaluates only when `Light1` is no longer yellow.

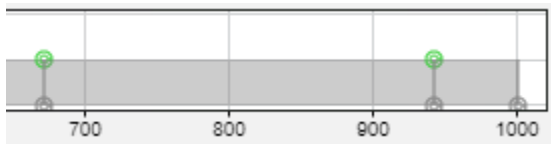
Rerun the test case. Now the assessment passes at the four points of the simulation where `Light1` turns yellow. Additionally, there is a point marked at $t = 984.5$, which corresponds to the point after which the assessment can no longer be evaluated. The logic specifies that the trigger condition must stay true for most `YellowDelayWithTol` seconds, or 15.5 seconds. After $t = 984.5$, there is not enough time left in the simulation to accurately assess the logic.



You might notice similar behavior for other assessments based on the timing parameters supplied to the trigger.

You can reevaluate the assessments to minimize untested results.

- 1 Click **New Test Case 1** in the current results.
- 2 At the bottom of the **Property-Value** pane in the lower left, enable **Extended Assessment Result**. The assessments are reevaluated and the results are updated.
- 3 The plot now shows that there are no untested results.



Create a Trigger-Response Assessment to Evaluate Green to Yellow Transitions

This assessment evaluates the `Light1` transitions from Green to Yellow during the first half of simulation. The assessment triggers when the light changes from Red to Green. As described in the **Timing of Traffic Lights** section of “Monitor Chart Activity by Using Active State Data” (Stateflow), the transition from Green to Yellow occurs within a fixed window of time based on the `greenLightRequested` parameter. To set up the parameters and use the built-in symbol `t` to restrict the assessment to the first half of the simulation, create an assessment callback.

Set Up the Parameters

- 1 In the Test Manager, expand the **Logical and Temporal Assessments** section.
- 2 Click **Add Assessment** and choose **Trigger-response**.
- 3 Double click the **Name** field. Rename the assessment `Light1 GreenToYellow`.
- 4 Set the trigger condition. Click the drop-down next to the **trigger** field and select **becomes true and stays true for between**. This logic is required because the light must stay green before switching to yellow. After you select the trigger type, the **condition**, **min-time (sec)**, **max-time (sec)**, and **time-reference** fields become visible.
- 5 For **condition**, enter `t < 500 & Light1 == Green`. The assessment uses the built-in symbol `t` to trigger a check when `Light1` becomes Green within the first 500 seconds of the simulation.
- 6 For **min-time (sec)**, enter `GreenMin`.
- 7 For **max-time (sec)**, enter `GreenMax`. Note that `GreenMin` and `GreenMax` are not yet defined and appear as Unresolved symbols in the **Symbols** pane.

- 8 Set the **time-reference** to falling edge of trigger, which ensures that the response when Light1 becomes yellow is evaluated only when Light1 is no longer green.
- 9 Leave the **delay** as with no delay.
- 10 Set the **response** to must be true. This option evaluates a single instance of time and captures whether the transition to Yellow occurs. After selecting the response type, the **condition** field becomes visible.
- 11 For **condition**, enter `Light1 == Yellow`.
- 12 Resolve the GreenMin and GreenMax symbols, by adding this assessment callback code to the existing code in the **Assessment Callback**. The callback extracts the Stateflow parameters that correspond to the Green transition, then adjusts them by the tolerance value specified in the requirements to prepare them for the min-time and max-time trigger fields.

```
greenMin = maskObj.getParameter('MINGREENDELAY');
greenMin = str2double(greenMin.Value);
greenMinAdj = greenMin - tol;
```

```
greenMax = maskObj.getParameter('GREENDELAY');
greenMax = str2double(greenMax.Value);
greenMaxAdj = greenMax + tol;
```

13. In **Symbols**, right-click the symbol name GreenMin and choose Map to expression.

14. In the **Expression** field, enter the variable name greenMinAdj.

15. Resolve the symbol GreenMax by repeating steps 13 and 14 by using greenMaxAdj for the expression.

View the Light1 GreenToYellow Logic

Collapse the assessment to read a summary of its logic:

The screenshot shows the assessment logic for 'Light1 GreenToYellow'. The main pane displays the following text: 'At any point of time, if t<500 & Light1 == Green becomes true and stays true for between GreenMin seconds and GreenMax seconds then, starting from falling edge of trigger, with no delay, Light1 == Yellow must be true'. The 'None' option is selected for the response type. On the right, the 'SYMBOLS' pane lists the following symbols: Light1, Light2, NumCars, Red, Yellow, Green, YellowDelayWithTol, GreenMin, and GreenMax.

You can use the **Visual Representation** to preview the logic of the assessment. After Light1 turns green, it must stay green for a period of time within the minimum and maximum times. When the Light1 trigger is false and the light is no longer green, the Light1 trigger is true and the light changes to yellow .

Run the Assessment and View the Results

Run the assessment.

In the **Results and Artifacts** pane, expand **New Test Case 1 > Logical and Temporal Assessments**. Select **Light1 GreenToYellow** and observe that the assessment triggers twice in the

first half of the simulation. The trigger times align with the initial Simulation Data Inspector results on page 3-99 when **Light1** enters the Green state at $t = 12.1$ and $t = 282.2$.

✔ **Light1 GreenToYellow**

The screenshot displays the Simulation Data Inspector interface for the assessment "Light1 GreenToYellow".

- ASSESSMENT:** At any point of time, if $t < 500$ & **Light1 == Green** becomes true and stays true for between **GreenMin seconds** and **GreenMax seconds** then, starting from falling edge of trigger, with no delay, **Light1 == Yellow** must be true.
- SYMBOLS:** Light1, Yellow, Green, GreenMin, GreenMax.
- EXPRESSION TREE:** Light1 GreenToYellow: At any point in time, if $((t < 500) \& (Light1 == Green))$ becomes true and stays true for between 119.9 seconds and 180.1 seconds then, starting from falling edge of trigger, with no delay, $(Light1 == Yellow)$ must be true.
- PLOTS:** A timeline plot showing the assessment status over 1000 seconds. The status is "Untested" until $t = 0$, then "Pass" (green circle) at $t = 12.1$ and $t = 282.2$. There are also "Fail" (red circles) at $t = 0$, $t = 1000$, and a "Fail" (grey circle) at approximately $t = 820$.

See Also

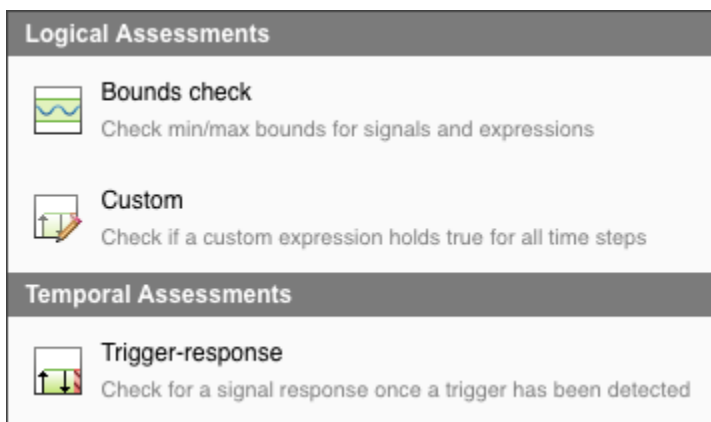
Related Examples

- "Logical and Temporal Assessment Syntax" on page 3-107
- "Test Sequence and Assessment Syntax" on page 3-68

Logical and Temporal Assessment Syntax

Simulink Test provides three logical and temporal assessment templates:

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



Bounds Check Assessments

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






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

Trigger-Response Assessments





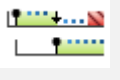
Create trigger-response assessments to verify a signal response when a trigger is detected. A trigger-response assessment requires:

- Trigger parameter
- Response parameter
- Optional Delay parameter

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

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

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

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

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

Custom Assessments

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

Logical and Temporal Assessment Conditions

You can enter MATLAB expressions that include these operators as the assessment conditions:

- **Logical operators:** &, |, and ~
- **Relational operators:** <, <=, ==, ~=, >=, and >
- **Arithmetic operators:** +, -, and * (multiplication by scalar constants only)
- **Cast operators:**
 - Floating-point number: single and double
 - Unsigned integer: uint8, uint16, and uint32

- Signed integer: `int8`, `int16`, and `int32`
- Logical: `logical`

The functional forms of the logical, relational, and arithmetic operators are not supported. In addition to operators, you can use the `abs` function to construct assessment conditions. You can also use the `t` symbol to construct assessment conditions, which is automatically bound to simulation time. Use of the `t` symbol as a `min-time` or `max-time` parameter in assessment conditions is not supported. Event-based signals are not supported in logical or temporal assessments.

Data Types in Assessment Conditions

Logical and temporal assessment conditions support the built-in data types listed on “Data Types Supported by Simulink”, with the exception of `string`. You can also use `Simulink.defineIntEnumType`. Fixed-point data types are not supported in assessments.

All operands in an assessment condition must be of the same data type. You can use cast operators to change the data type of an operand or change an operand to a symbol and map the symbol to an expression. Read about mapping symbols to expression on “Resolve Assessment Parameter Symbols” on page 3-95. Read about defining variables for use in an expression on “Define Variables in the Assessment Callback Section” on page 3-110. When you map a symbol to an expression, the expression must be the same data type as other operands in the assessment condition.

When mapping a symbol to a bus signal or a multidimensional signal, you must map the symbol to only one element from the bus or multidimensional signal. Read about mapping to model elements on “Resolve Assessment Parameter Symbols” on page 3-95. The data type of the selected element from the bus or multidimensional signal should always be a supported type, and must be the same data type as other operands in the assessment condition.

Define Variables in the Assessment Callback Section

The **Assessment Callback** section allows you to define variables that you can use in logical and temporal assessment conditions and expressions. You can define variables the same way you do in the MATLAB workspace. This callback also has access to predefined variables that contain data from your test, model, and simulation, such as a signal from a Simulink block. You can define a variable as a function of this data. These objects are available:

Object Name	Description
<code>TestResult</code>	The test case result (<code>sltest.testmanager.TestCaseResult</code>) or test iteration result (<code>sltest.testmanager.TestIterationResult</code>) created from the simulation.
<code>sltest_simout</code>	An array of simulation outputs (<code>Simulink.SimulationOutput</code>).
<code>sltest_testCase</code>	Current test case object (<code>sltest.testmanager.TestCase</code>).
<code>sltest_bdroot</code>	Cell array of models simulated by the test case. Can be a harness model.
<code>sltest_sut</code>	Cell array of systems under test. For a harness, this array contains the component under test.

Object Name	Description
sltest_isharness	Cell array that returns true if sltest_bdroot is a harness model.
sltest_iterationName	Name of current test iteration.

After defining the variables in the callback, you can map the symbols to variables for use in assessment conditions and expressions. Read about mapping a symbol to an expression on “Resolve Assessment Parameter Symbols” on page 3-95 for information on how to map symbols to variables.

The variables created in the **Assessment Callback** section can only be used in conditions and expressions in the **Logical and Temporal Assessments** pane. These variables cannot be used in other areas of the Test Manager. The **Assessment Callback** is saved as part of the test file.

See Also

More About

- “Assess Temporal Logic by Using Temporal Assessments” on page 3-93
- “Assess Temporal Logic in Multiple Releases” on page 6-98

Observers

- “Access Model Data Wirelessly by Using Observers” on page 4-2
- “Observe Messages” on page 4-13
- “Observe Conditional Subsystem Signals” on page 4-17
- “Observe Internal Variables of an FMU” on page 4-21

Access Model Data Wirelessly by Using Observers

In this section...

“Observer Reference Block” on page 4-3

“Connect Signals or Other Model Data Using an Observer Port Block” on page 4-4

“Trace Observed Items to Model Signals and Objects” on page 4-6

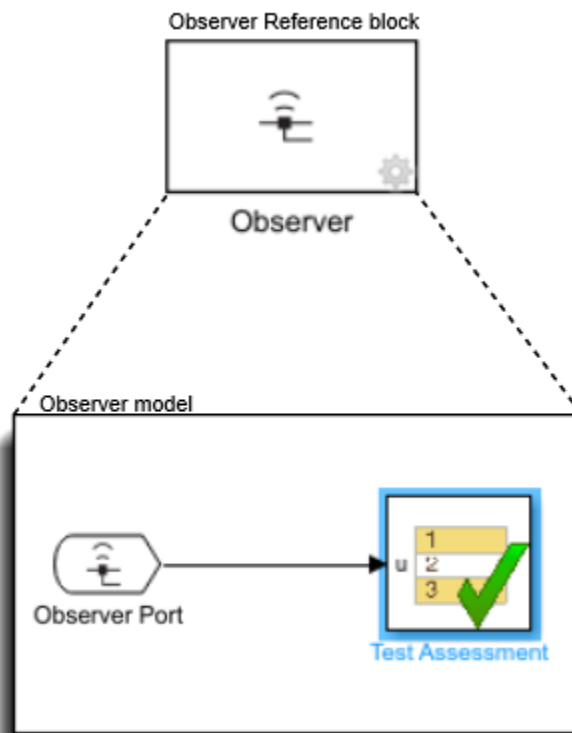
“Simulate a System Model with an Observer Reference Block” on page 4-6

“Verify Heat Pump Temperature by Using Observers” on page 4-7

“Convert Verification Subsystem to an Observer Reference” on page 4-9

“Observer Considerations and Limitations” on page 4-12

Observers allow you to monitor the dynamic response of your system model while preserving the system model design and system result integrity. Observers use two types of blocks, Observer Reference blocks and Observer Port blocks. The Observer Reference block wirelessly links a system model to an Observer model, which contains verification logic. Inside an Observer model, you use Observer Port blocks to access data from the system model to drive the verification logic.



The types of Simulink signals and model data you can observe are:

- Continuous-time and Discrete-time signals
- Zero-order hold signals

- Scalar signals
- Wide signals
- Nonvirtual bus signals
- Messages
- Conditional subsystem signals
- Stateflow local data parameters, except locals, parameters, signals, and other data defined in a Simulink subsystem inside a Stateflow state.
- Stateflow state self activity, except if that activity is in a Simulink subsystem inside a Stateflow state.

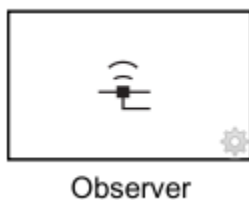
Observer Reference Block

Observer Reference blocks wirelessly link a system model to an Observer model. Observer Reference blocks can only be at the top level of a system model and do not have input or output ports. You map your Simulink signals or other model data to the Observer Port blocks that are contained within the Observer model. Once you map the Observer Port blocks to a signal or data, you can connect the ports to the verification subsystem within the Observer model. Running your system model also runs the linked Observer model.

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

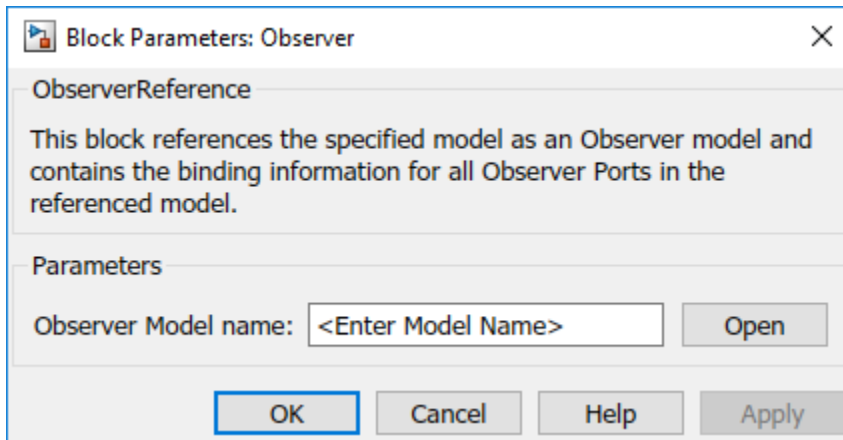
Add an Observer Reference Block

The Observer Reference block references a separate verification model that you use to verify your system model. To add an Observer Reference block to your system model, first, in the Simulink toolstrip, open Apps and click Simulink Test in the Model Verification, Validation, and Test section. Click **Add Observer Reference** in the **Tests** tab. Alternately, right-click the top level of your Simulink canvas and select **Observers > Add Observer Reference here** from the context menu. An Observer Reference block is added to your system model, and an Observer model is created and opened. You must save the Observer model in a writable folder on the MATLAB path.



Connect an Existing Observer Model

To connect an Observer Reference block to an Observer model that you have already created, first save your Observer model in a writable folder on the MATLAB path. Next, right-click on the Observer Reference block and select **Block Parameters (ObserverReference)**.



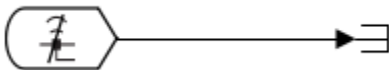
Enter the name of the Observer model that you want to connect to your system and select **Apply**. When you double-click your Observer Reference block, your Observer model opens in a new window.

Create an Observer Model from Signals or Other Model Data

To create an Observer model that is mapped to a signal line or observable data in your model, select one or more signals or the data that you want to observe. Then, click **Add Observer Reference** in the **Tests** tab. Alternately, right-click on the signal or data and select **Observers**, the item type to observe, and **New Observer**. Simulink creates an Observer model and adds an Observer Reference block to your system model.

Connect Signals or Other Model Data Using an Observer Port Block

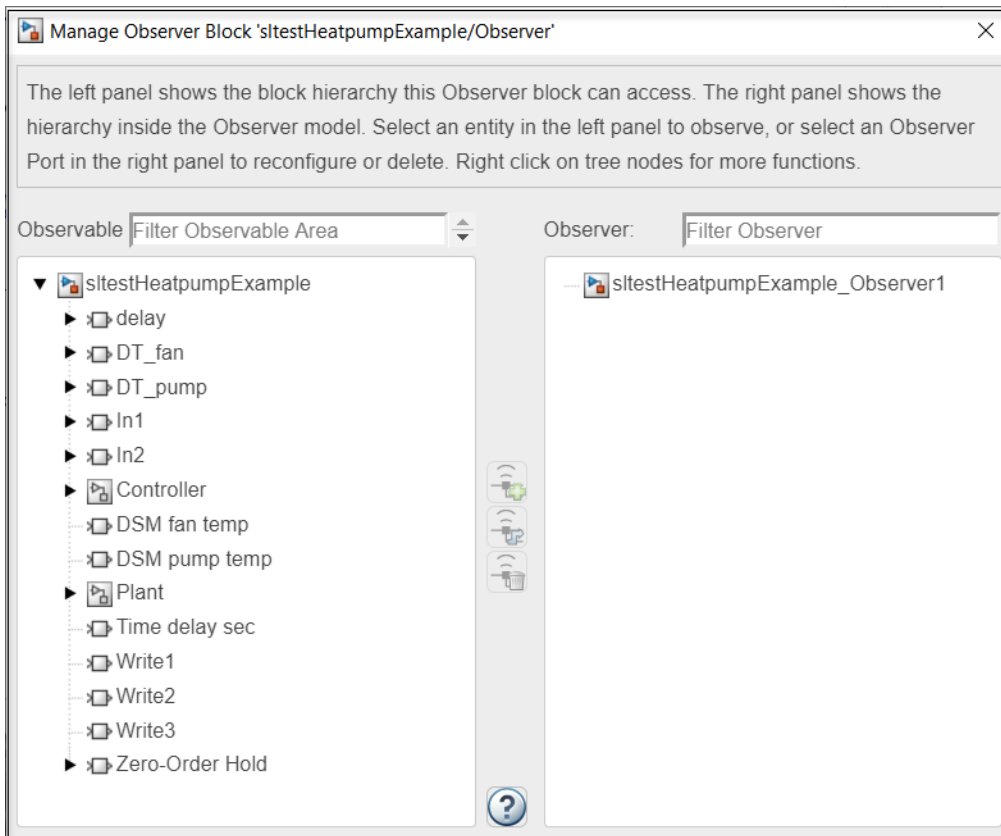
Each Observer model contains one or more Observer Port blocks. After mapping an Observer Port block to a model object or signal within a system model, the Observer Port block outputs the same output as its mapped object or signal. A new Observer Port block shows a line through the signal symbol, signifying that the block is not mapped to any signal or object.



Access the Manage Observer Dialog Box

To map an Observer Port block to a signal or object in your system model, open the Manage Observer dialog box using one of these methods:

- In the **Tests** tab, click **Manage Observer**.
- Click the gear in the lower-right corner of the Observer Reference block.
- Right-click the Observer Reference block and select **Observers > Manage Observer**.
- In the Observer model, double-click an Observer Port block.

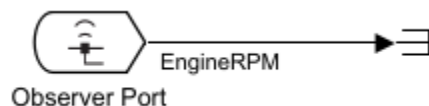


Using the Manage Observer dialog box you can:

- Filter and select signals and objects for observation
- Add, remove, or configure Observer Port blocks
- Trace signals and objects between observer ports and models

On the left side of the Manage Observer dialog box is the Observable Area panel. The Observable Area panel displays the block hierarchy and observable outputs of your model. Observed signals or objects appear bold in the hierarchy.

The right side of the Manage Observer dialog box shows the Observer panel. The Observer panel displays the block hierarchy, including Observer ports in the Observer Reference block. An Observer Port block that is mapped to a signal or object appears bold and displays the signal to which it is attached. Once the Observer Port is mapped to a signal or object, its block icon updates to show that the Observer Port is attached to a signal or object.




To view the full path of an observed object, point to an Observer Port block.

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

Map an Observer Port Block to a Signal or Object

To map a signal or object to an Observer Port block, open the Manage Observer dialog box. In the Observable Area panel, select the signal or object that you want to observe. To map the signal or object to a new Observer Port block, double-click the selected item or click the Add New Observer

Port icon . To map the signal or object to an existing Observer Port block, select the Observer

Port in the Observer panel and click the Reconfigure Observer Port icon . In the Observer model, you can then connect the output from the Observer Port to a verification subsystem to test your results.

Trace Observed Items to Model Signals and Objects

You can trace observed items and their observer ports within the Manage Observer dialog box. You can also trace items between the Manage Observer dialog box and the system model, and between the system model and the Observer model.

To trace an observed item to its observer port within the Manage Observer dialog box, use one of these methods:

- Double-click on the ObserverPort item in the Observer panel. The observed item is highlighted in the Observable Area panel.
- Right-click on the ObserverPort item in the Observer panel and select **Show in left panel**. The observed item is highlighted in the Observable Area panel.

To trace an observed item or observer port between the Manage Observer dialog box and the system model, use one of these methods:

- Right-click on the ObserverPort item in the Observer panel or in the Observable Area panel and select **Show in model**. The observed item is highlighted in the model.
- Right-click on the observed signal or object in the system model and select **Go to associated Observer Ports**. The associated Observer Ports are highlighted in the Observer model.

To trace an observer port and observed item between the system model and the Observer model, use one of these methods:

- Right-click on the Observer Port in the Observer model and select **Observers > Go to observed <item type>**. The observed signal or object is highlighted in the system model.
- Right-click on the observed signal or object in the system model and select **Observers > Go to associated Observer Ports**. The associated Observer Ports are highlighted in the Observer model.

Simulate a System Model with an Observer Reference Block

The Observer model is used to monitor signals in your system model and check that your system model is running within specified parameters. With or without an Observer Reference block, your system model simulation results are the same. The Observer Reference block does not affect the compilation of your system model.

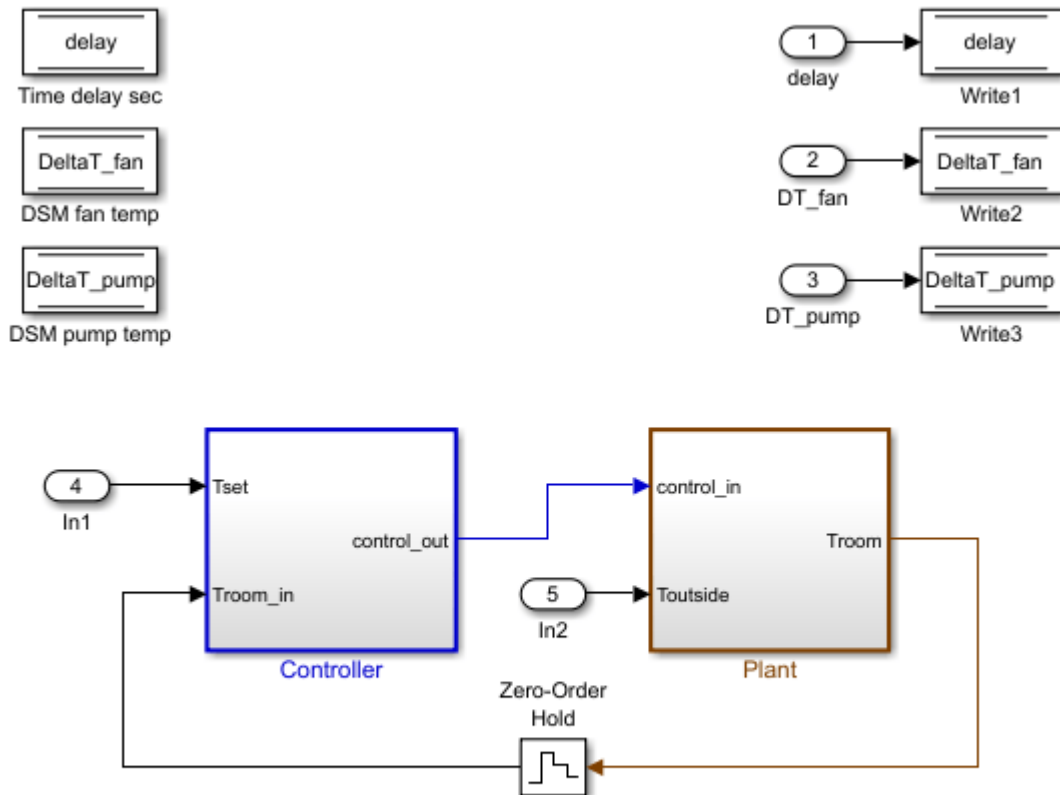
Note Both the system model and Observer model must run in normal simulation mode. Both models can run at fixed-step or variable-step rate, or one model can run at fixed rate and the other at variable rate. The two models can also use the same or different solvers. See “Choose a Solver”.

Verify Heat Pump Temperature by Using Observers

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

Open the Model

`sltestHeatpumpExample`



Copyright 1990-2023 The MathWorks, Inc.

Create the Observer Model and Observer Reference Block

1. In the **Apps** tab of the model, click **Simulink Test** in the Model Verification, Validation, and Test section. The **Tests** tab opens

2. In the **Tests** tab, click **Add Observer Reference**. Simulink adds an Observer Reference block to your system model and creates an Observer model called `sltestHeatpumpExample_Observer1`.

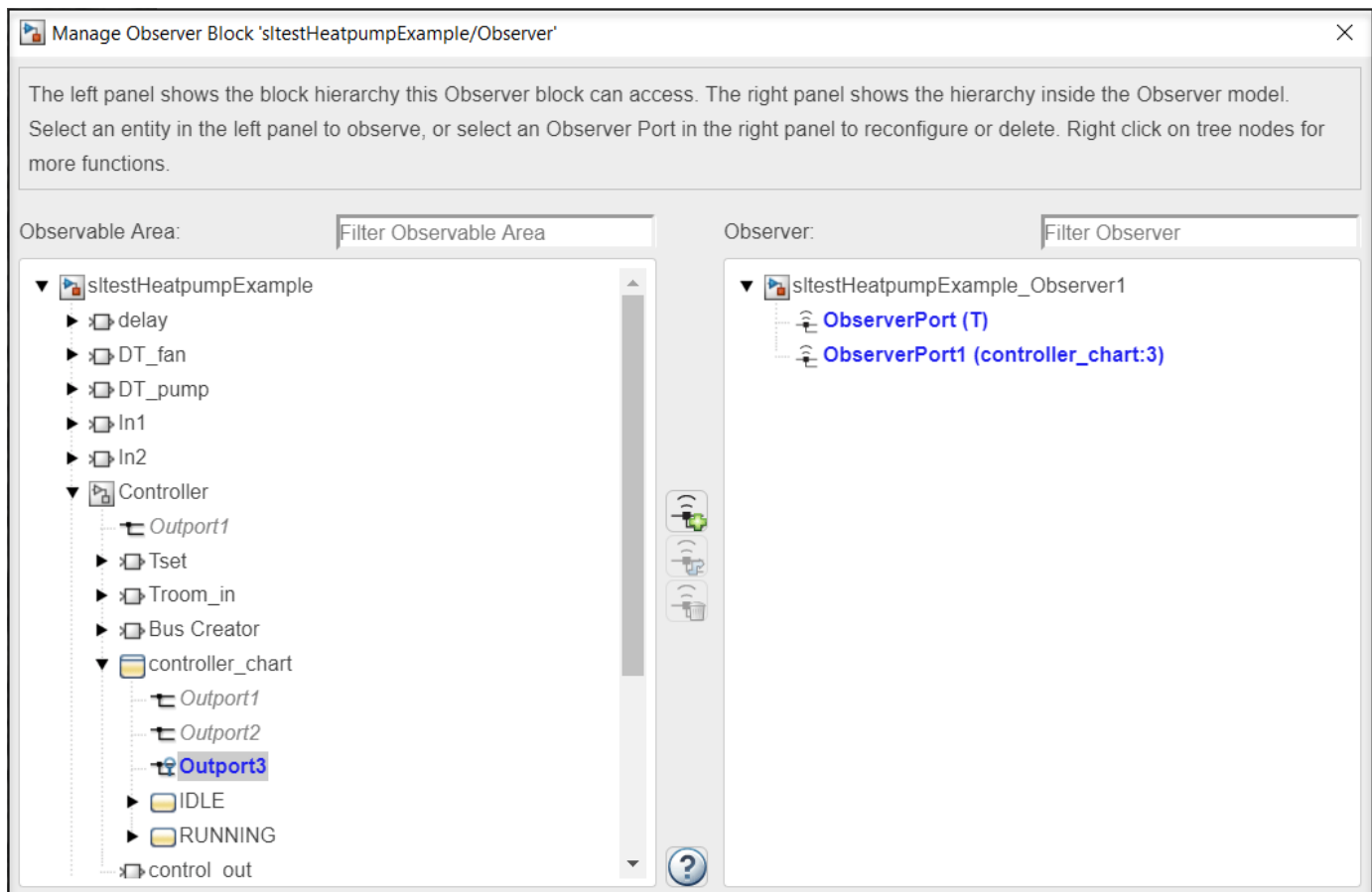
Create and Map Observer Port Blocks

1. In the main model, open the Plant subsystem and right-click the signal `T`. Select **Observers > Observe selected signals > sltestHeatpumpExample/Observer (sltestHeatpumpExample_Observer1)**. The Observer model adds an Observer Port block that is mapped to signal `T`.

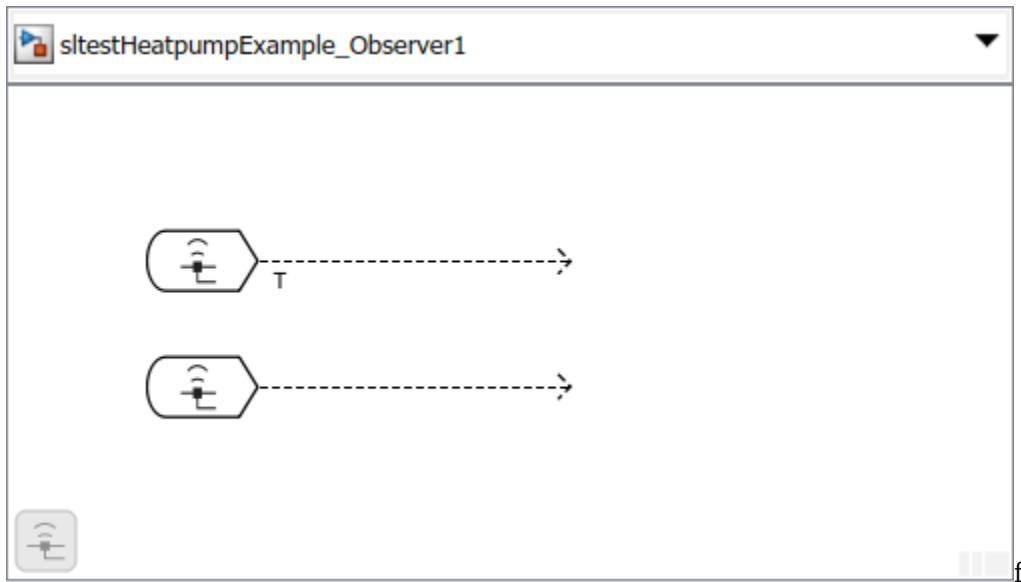
2. Save the new Observer model.

3. In the **Tests** tab of the Observer model, click **Add Observer Port** to add another Observer Port block. Double-click the new `ObserverPort1` block to open the Manage Observer dialog box. In the Observer panel, the second Observer Port, `ObserverPort1`, is listed below the first port.

4. To map `ObserverPort1` to the Simulink data `pump_dir`, click `ObserverPort1`. In the **Observable Area** panel, expand `Controller` and `controller_chart`, and select `Outputport3`. Click the Reconfigure icon between the two panels. The `ObserverPort1` name updates to `ObserverPort1 (controller_chart:3)`.



5. The Observer Port blocks are in the Observer model and are now mapped and ready to be connected to scopes or a verification subsystem.

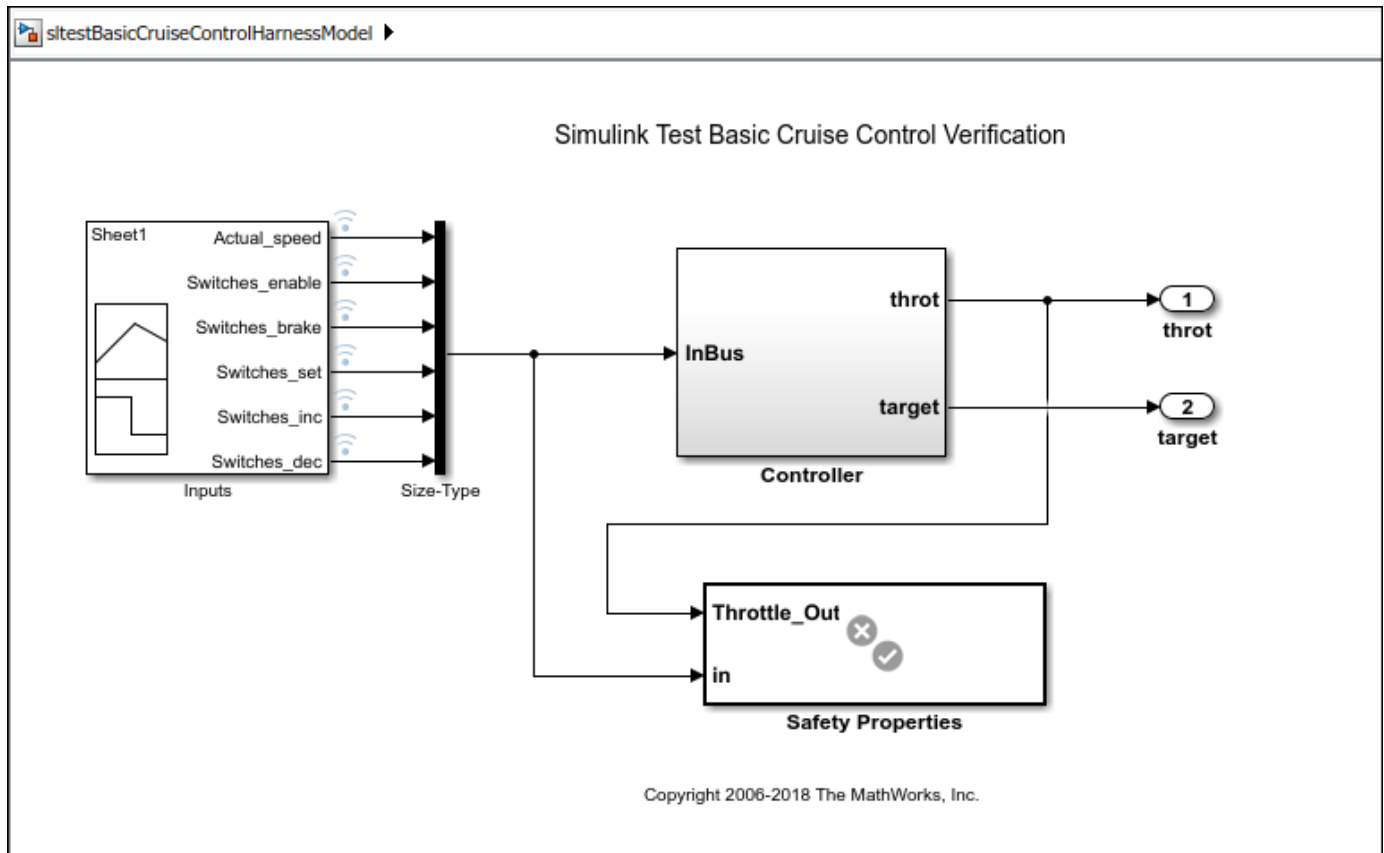


Copyright 2022 The MathWorks, Inc.

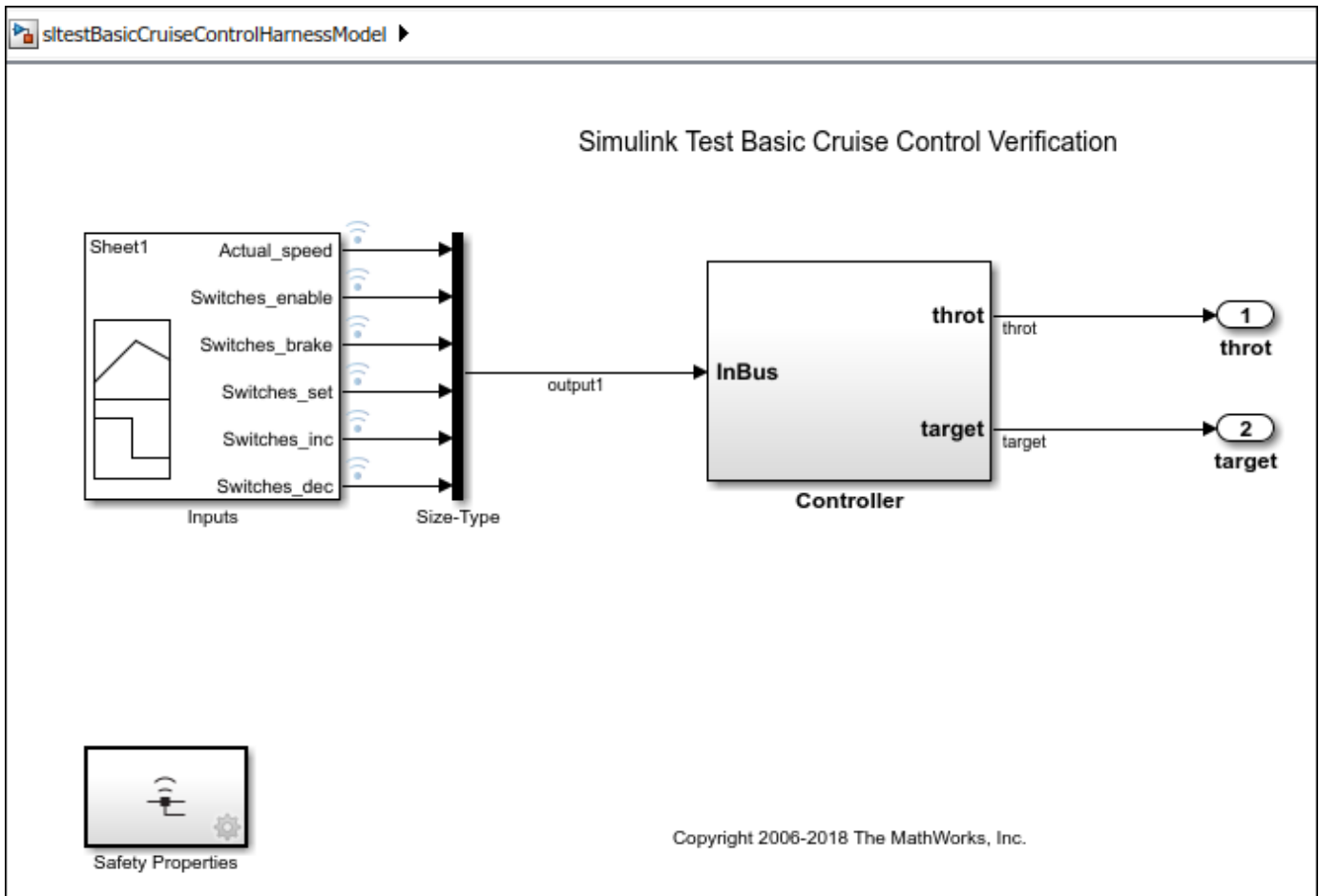
Convert Verification Subsystem to an Observer Reference

Converting a Verification Subsystem to an Observer Reference block is a way to declutter a system model. Select the subsystem to convert and, in the **Tests** tab, click **Send to Observer**. Alternately, right-click the verification subsystem and select **Observers > Move selected block to Observer > New Observer**. This operation cannot be undone.

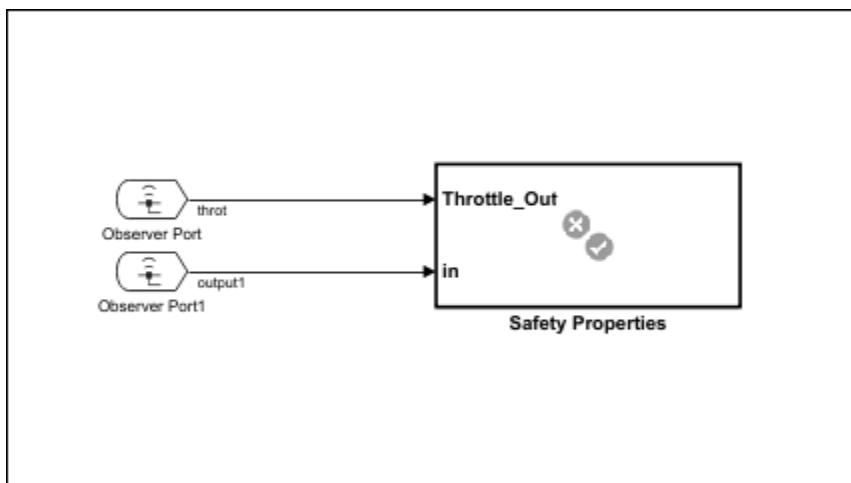
This model contains the Verification Subsystem, Safety Properties.



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



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



Observer Considerations and Limitations

Model Simulation

An Observer model does not simulate if:

- The Observer model contains root-level Inport or Outport blocks.
- The Observer model is a library or subsystem reference model.

Observer Reference Blocks

Observer Reference blocks are ignored during simulation if:

- You use any simulation mode other than normal mode (for example, accelerator, SIL/PIL).
- You are generating code.
- The Observer Reference block is in a model reference hierarchy. Observer Reference blocks are supported only at the root of the top model.
- The Observer Reference block is in an Observer model. Recursion of Observer models is not supported.

Data Export and Output

- Logging signals or data store memory and saving final operating points are supported for Observers. All other data export options, such as time, state, output, final state, and save to file, are not supported.
- To Workspace and Dashboard blocks in Observers are not supported and do not produce output.

Mismatched Settings Between the Observer and Design Model

When these settings in the Observer model differ from the settings in the design model, the design model settings are used and the Observer model settings are ignored.

- Data import or export settings
- Coverage settings
- Solver stop time

See Also

Observer Port | Observer Reference

More About

- “Observe Messages” on page 4-13
- “Observe Conditional Subsystem Signals” on page 4-17

Observe Messages

Simulink uses messages to communicate between model components. When your model includes one or more message signals, you can create an Observer model to observe the message data. When you create the Observer model, it creates metadata associated with the message. You can test the message semantics and verify the message properties by using both the message data and metadata.

Message Bus Elements

When you create an Observer for a message signal, the Observer model automatically includes an Observer Port block and a Bus Selector block. The bus has two elements:

- `OrigPayload` — The message data being observed.
- Metadata with these elements:
 - `sltestEventMetadata.Message.id` — ID of message being observed, returned as an `int32` integer.
 - `sltestEventMetadata.Message.order` — Order of message action in the simulation, returned as an `int32` integer.
 - `sltestEventMetadata.Message.eventType` — The event type, returned as `slTestEventType` object. Valid values are `MessageArrival`, `MessageDeparture`, `MessageDrop`, and `Invalid`.
 - `sltestEventMetadata.Message.time` — Simulation time when the message was sent, received, or dropped, returned as a `double`.

Add a Message Observer

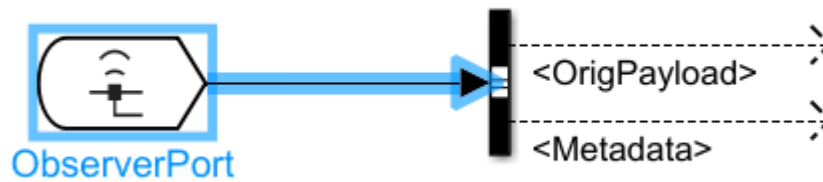
To add an observer for messages:

- 1 In the model that has one or more blocks that output messages, right-click the output message signal to observe.

Note Messages from blocks with asynchronous sample times are supported only for SimEvents® blocks that generate their own sample times.

- 2 Select **Observers > Observe selected signals > New Observer** to create an Observer model for the message signal. An Observer Reference block is added to the main model and an Observer model is created.

The new Observer model contains an Observer Port block and Bus Selector block. The Bus Selector block has two outputs, `OrigPayload` and `Metadata`. You can use the default settings in the Observer Port block to observe the outputs from the bus.



- 3 Connect the outputs from the bus to a block, such as a Test Assessment block or a Stateflow chart, to specify the logic to analyze or verify the message data or metadata.

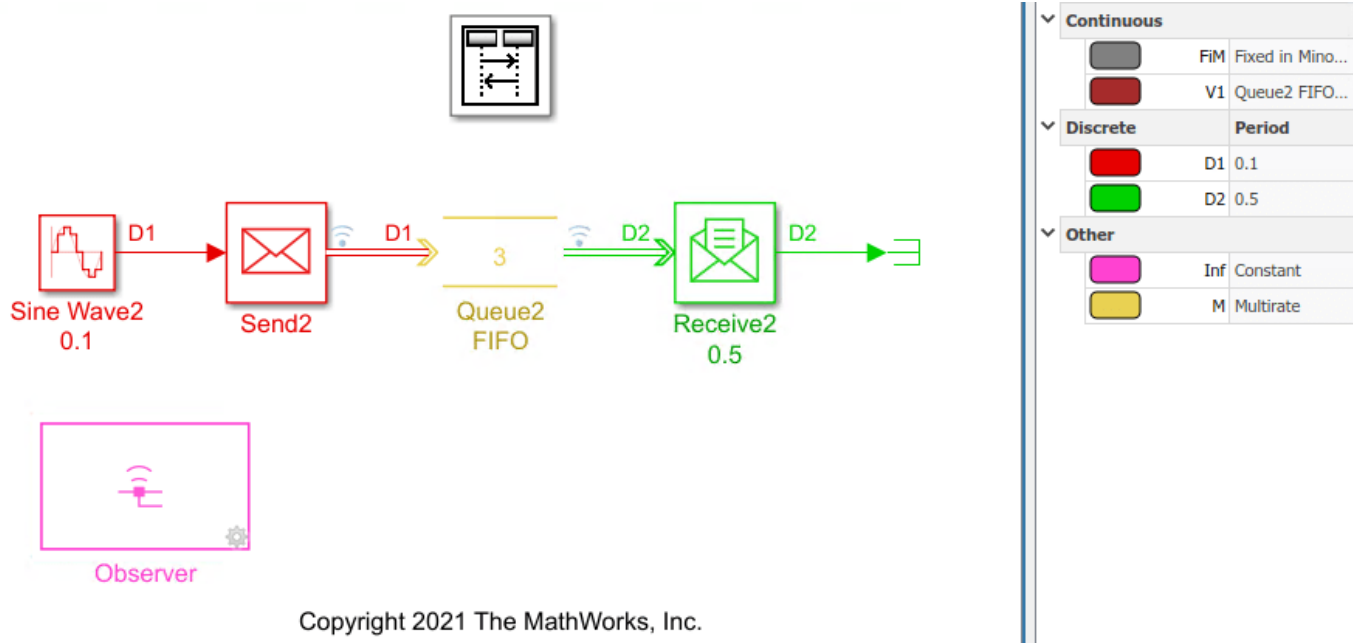
Observe a Message Signal

Open a model that has a block with message output. For this example, open the Overflow model, which contains an Observer model.

```
open_system('Overflow')
```

In this model, the Sine Wave2 block sends data every 0.1 seconds, but the Receive2 block receives data only every 0.5 seconds. Because the Queue2 FIFO block holds a maximum of three messages, an overflow occurs and some messages are dropped from the queue.

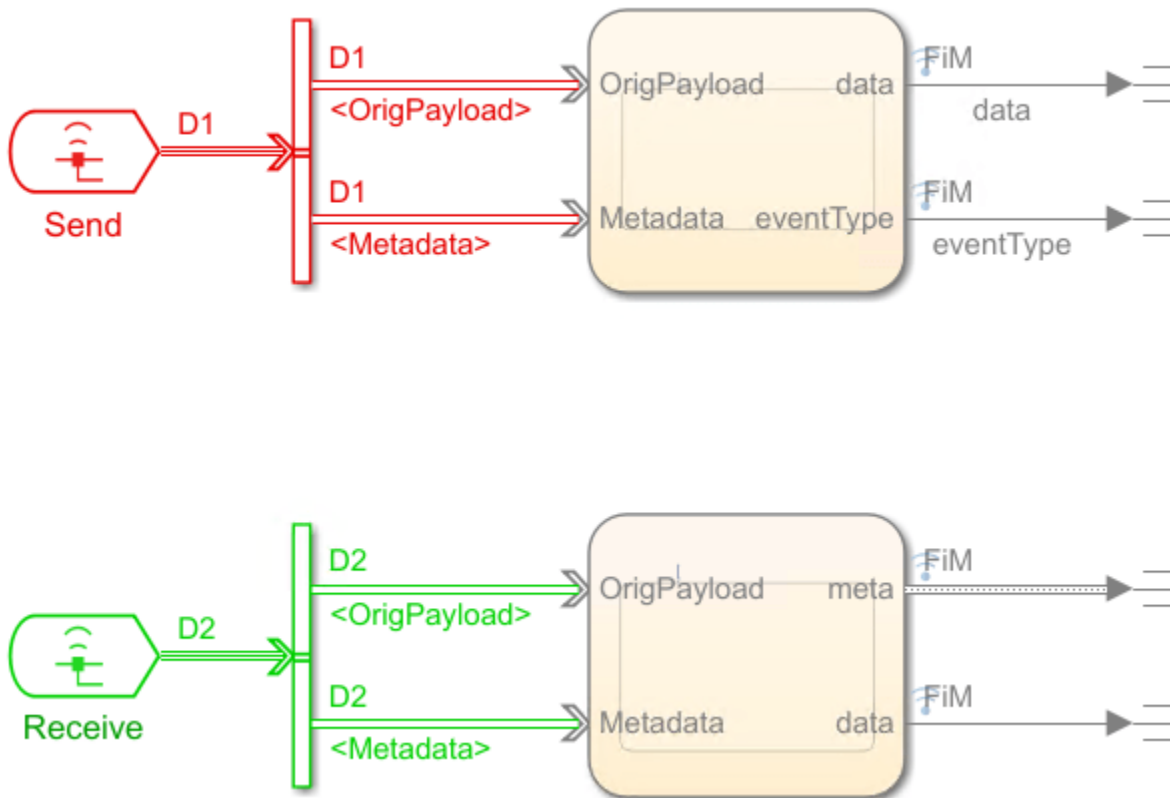
The Observer model was set up by selecting the message signals, right-clicking on one of them, and selecting **Observers > Observe selected signals > New Observer**. The Observer block, which is an Observer Reference block, was automatically added to the main model.



Open the Observer model by double-clicking the Observer block in the main model, or by using this command:


```
open_system('Overflow_Observer_1')
```

The Observer model contains two Observer Port blocks, **Send** and **Receive**, which correspond to the signals selected in the main model. Each of these blocks is connected to a Bus Selector block with two outputs. The **OrigPayload** output is the message data being observed and the **Metadata** output contains the message ID, order of the message in the simulation, the message event type, and the simulation time of the message.

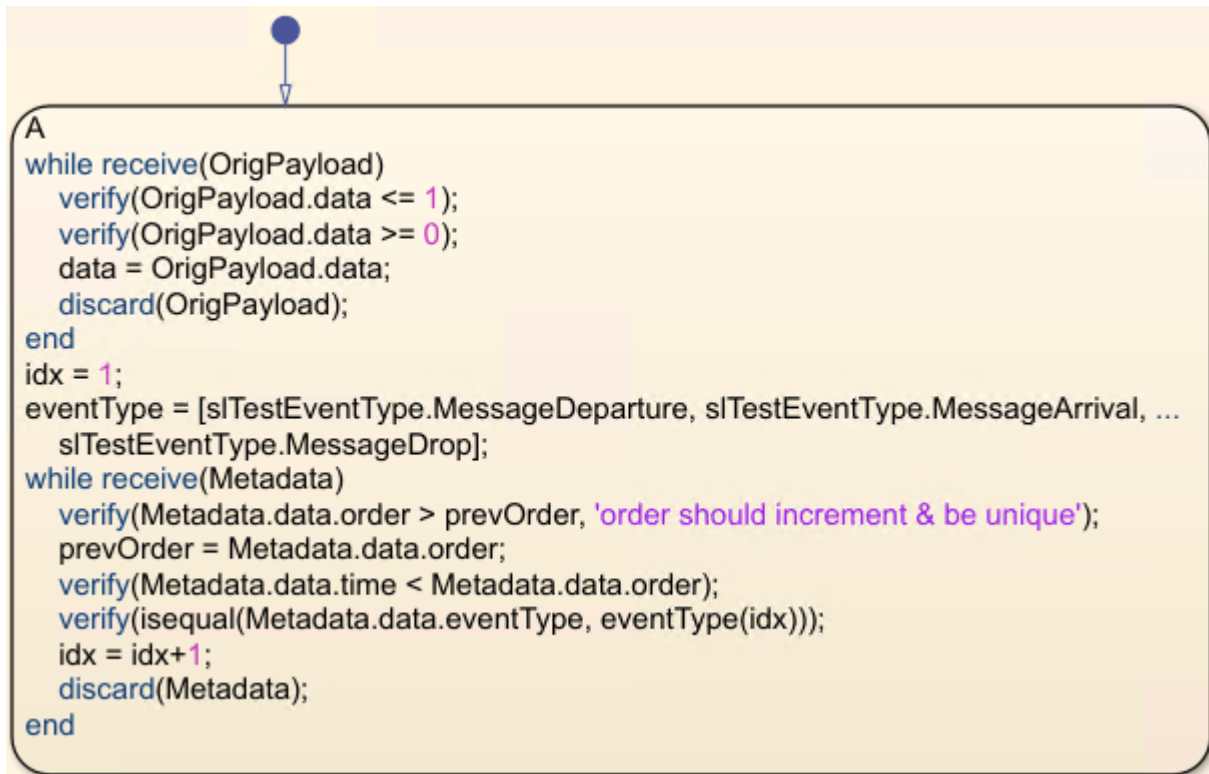


The bus signal outputs connect to two Stateflow charts, which analyze the message data.

The chart for the messages from the **Receive** Observer port block contains logic that verifies that the message data and metadata are not corrupted due to messages being dropped because of Queue block overflows.

The chart logic verifies that:

- Sine wave data is in the expected range, which is between 0 and 1
- The order of the messages and simulation time increments and that each order value is unique
- Message event type is a valid type



See Also

Observer Port | Observer Reference

More About

- “Messages”
- “Simulink Messages Overview”
- “Access Model Data Wirelessly by Using Observers” on page 4-2

Observe Conditional Subsystem Signals

Conditional subsystems are controlled by an external signal that enables or triggers the subsystem. By observing a conditional subsystem, you can check that the subsystem runs only when the controlling signal activates it, and check other functionality in the conditional subsystem. The blocks for which you can add observers are:

- Enabled Subsystem
- Triggered Subsystem
- Enabled and Triggered Subsystem
- If Action Subsystem
- Switch Case Action Subsystem

For more information on conditional subsystems, see “Conditionally Executed Subsystems Overview”.

When you add an observer to a signal in a conditional subsystem, Simulink adds an Observer Reference block at the top level of the main model and creates an Observer model, which contains an Observer Port block in an aperiodic partition. This partition controls the scheduling of the conditional data. The testing logic in the partition also runs conditionally. Because the partition is created automatically when you create the observer, you do not need to change the partition or scheduling settings. You can change the name of the partition, but the name must be unique within the model.

Add an Observer for a Conditional Subsystem

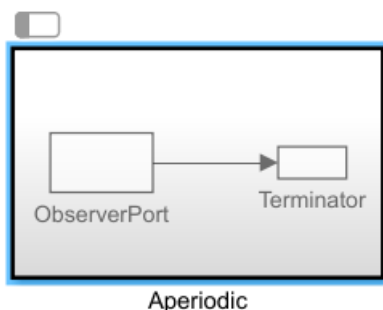
The steps to add an observer to a conditional subsystem are:

- 1 In the conditional subsystem, right-click the signal to observe.
- 2 Select **Observers > Observe selected signals > New Observer** to create an observer for the signal.

Alternatively, you can select the signal to observe, pause on the ellipsis to open the action bar,

then click **Observe in New Observer** .

Simulink adds an Observer Reference block to the main model and creates a new Observer model. The new Observer model contains an aperiodic partition that contains an Observer Port block and a Terminator block. For information about partitions, see the "Partitioning a Model" section of “Create Partitions”



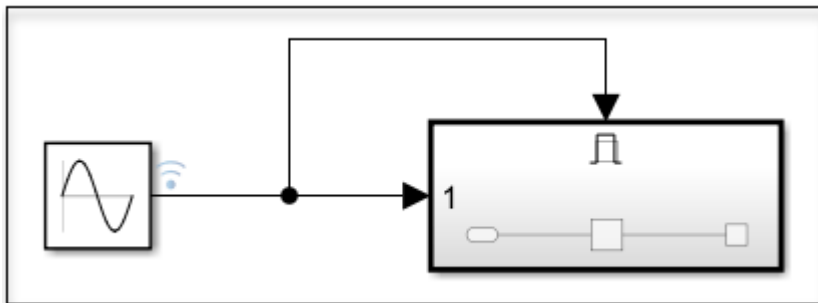
- 3 Replace the Terminator block with a verification block, such as a Test Assessment or Verification Subsystem block and connect the verification block to the Observer Port output. Add logic to verification block to analyze or verify the conditional subsystem signal.
- 4 Run the model from the main model, not from the Observer model.

Observe a Signal in a Conditional Subsystem

This example shows how to add an observer for a signal in a conditional subsystem. The `ObserveCondSubsys` model used in this example contains a sine wave control signal as the input to an Enabled Subsystem block. The Enabled Subsystem block is active only when the sine input is positive.

1. Open the `ObserveCondSubsys` model.

```
open_system('ObserveCondSubsys')
```

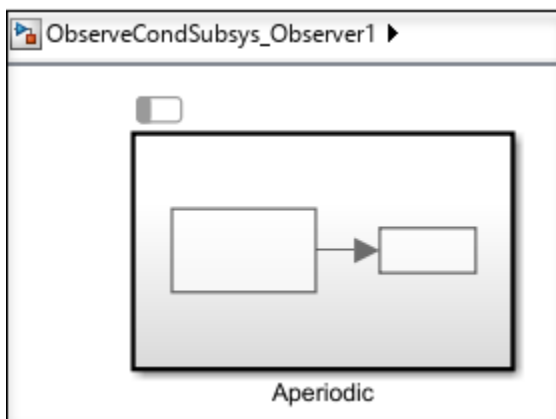
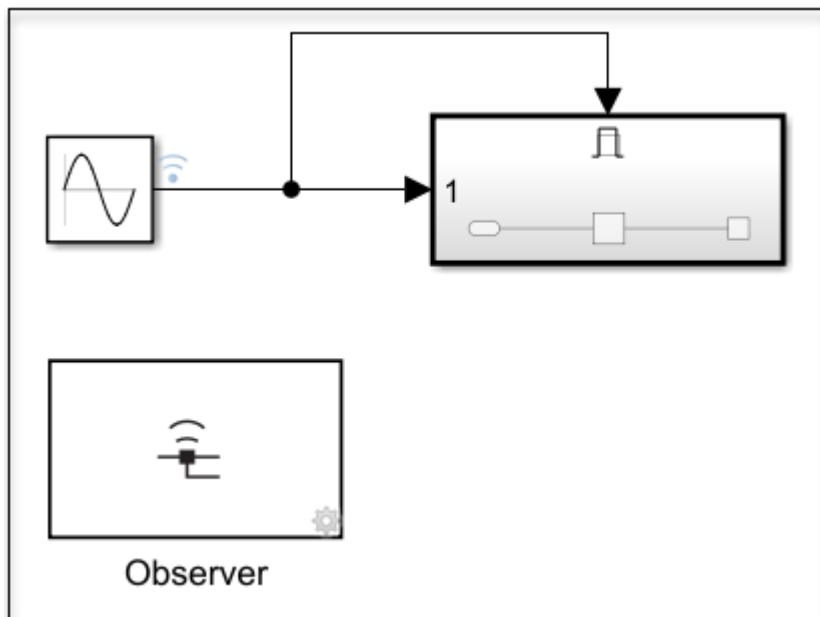


2. Open the Enabled Subsystem block.

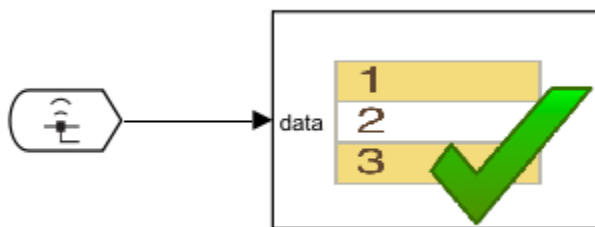
3. Select the Integrator block output signal. Pause on the ellipses to open the action bar, then click **Observe in New Observer**.



Simulink adds an Observer Reference block to the top level of the model and opens a new Observer model. This model has an aperiodic partition that contains an Observer Port block with its output connected to a Terminator block.



4. Open the Aperiodic partition by double-clicking it.
5. Replace the Terminator block with a Test Assessment block.



6. Open the Test Assessment block.
7. Delete **step_1_1** and **step_1_2**.

8. Right-click **step_1** and clear the **When decomposition** check box.

9. In **step_1**, add `verify(data >= 0);`

Step

```
step_1  
verify(data >= 0);
```

10. Close the Test Assessment block.

11. Go to the main model and click **Run**.

12. After the model runs, at the MATLAB command line, use these commands to view the result.

```
run = Simulink.sdi.Run.getLatest;  
dataset = Simulink.sdi.exportRun(run.Id);  
assessmentSignalIndices = find(arrayfun(@(idx)...  
    isequal(class(dataset{idx}), 'sltest.Assessment'),...  
    1:dataset.numElements));  
result = arrayfun(@(idx) dataset{idx}.Result,assessmentSignalIndices)
```

The `verify` statement produces a passing result, which indicates that the Enabled Subsystem block and its associated observer are active only when the sine input is positive.

Copyright 2022 The MathWorks, Inc.

Limitations

- Only one Observer Port can be in an aperiodic partition.
- Aperiodic partition names must be unique within a model.
- You cannot edit the scheduling trigger in the aperiodic partition.
- You cannot drag and drop or reorder items in the Schedule Editor or by using `simulink.schedule.OrderedSchedule`.
- You cannot add events to the aperiodic partitions of the Observer model.
- Partition subsystems and Function-Call Subsystem and For Each Subsystem blocks do not support observers.

See Also

Observer Port | Observer Reference

More About

- “Access Model Data Wirelessly by Using Observers” on page 4-2
- “Conditionally Executed Subsystems Overview”
- “Events in Schedule Editor”

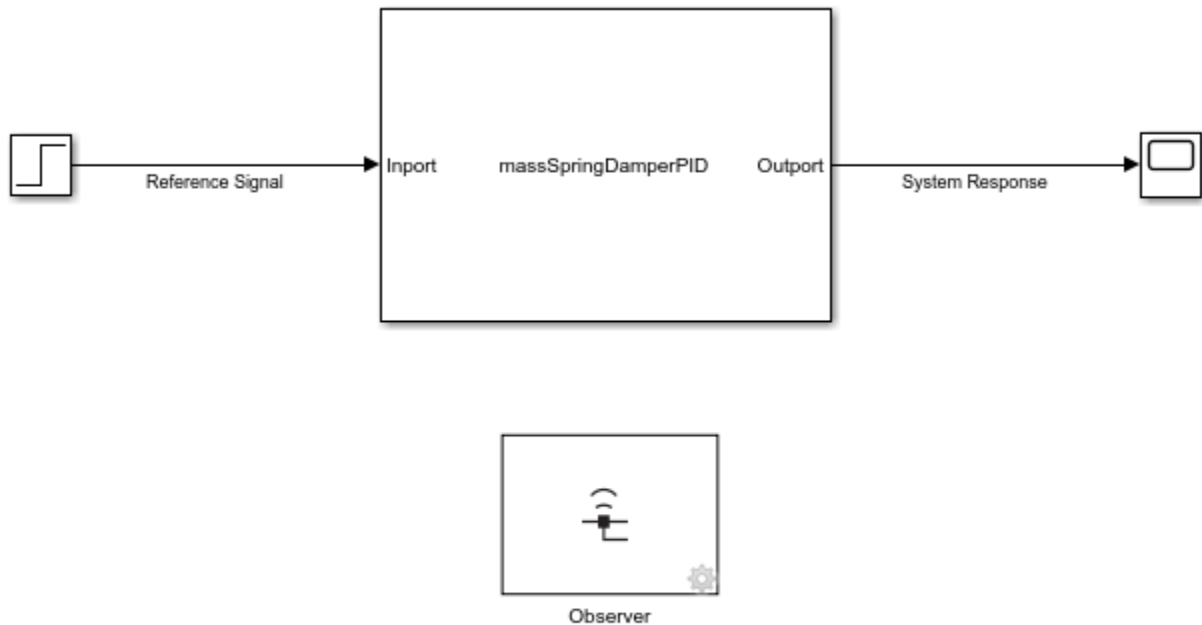
Observe Internal Variables of an FMU

Observers enable wireless access to internal variables of a Functional Mockup Unit (FMU) in Simulink. Use the Observer Reference and Observer Port blocks to access data and signals, configured as an internal variable, inside an FMU. You can observe scalar and bus signals configured as internal variables of the FMU.

Observe Internal Variables to Tune PID Controller Inside an FMU

In this system, the PID controller calculates the force that needs to be exerted on the mass to control its position. The Observer blocks allows you to tune the PID controller by providing wireless access to the control input and tracking errors that are configured as internal variables.

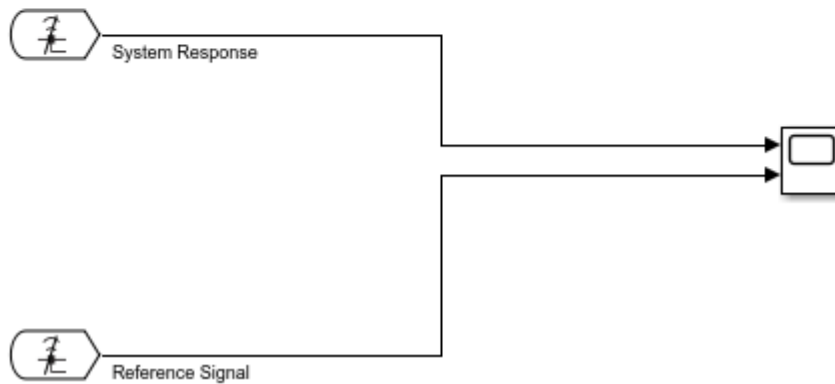
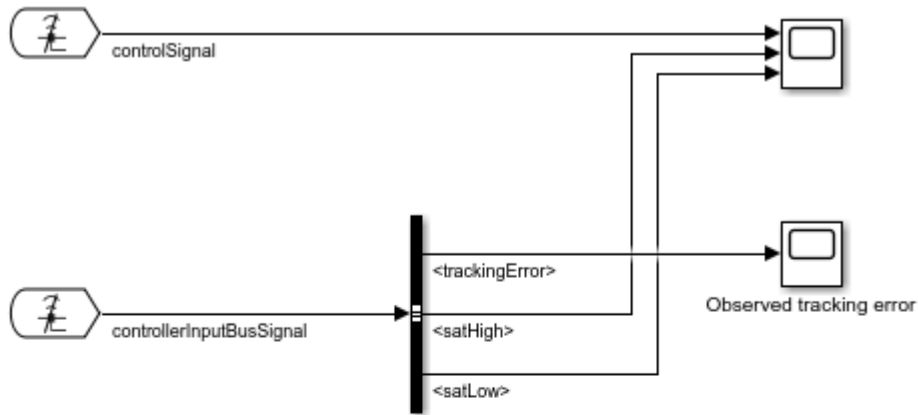
Import FMU With Internal Variables Into Simulink and Add Observer



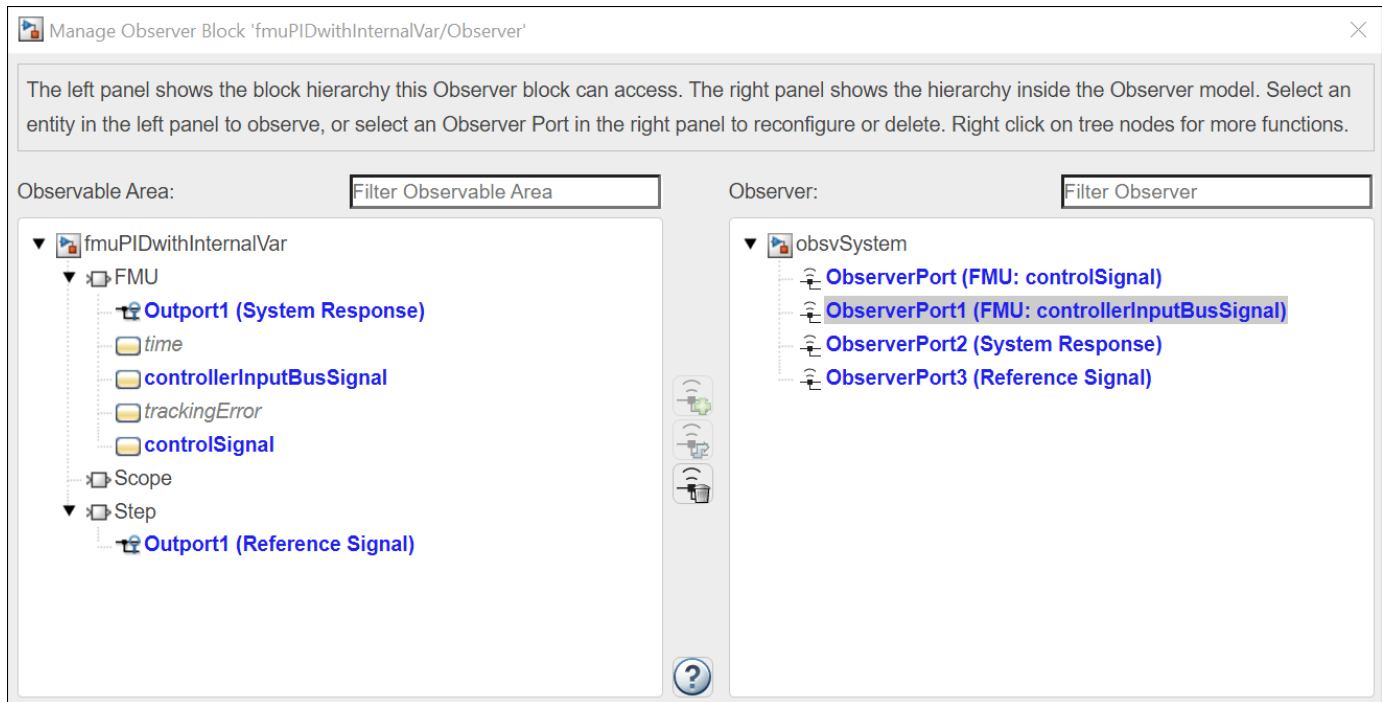
Copyright 2023 The MathWorks, Inc.

Use the FMU Import block to import the FMU into Simulink. This example uses an FMU generated using Simulink. However, the observer block is compatible with FMUs generated using external tools. Input to the FMU is the reference signal specifying the desired position of the mass. Output from the FMU is the actual position of the mass. The PID coefficients and input saturations are tunable parameters of the FMU. The system inside the FMU consists of a cascaded PID and a mass-spring-damper system. The reference signal specifies the desired position of the mass. The PID controller provides the force as input to the mass-spring-damper system to control the position of the mass. Tracking error, control inputs and controller saturations are configured as internal variables of the system.

Add an Observer block to the main model that contains the FMU. The Observer block references an Observer model that contains the observer ports that access signals in the main model. Create the Observer model and specify its name in the **Observer Model name** of the Observer Block parameter dialog box. You can open the observer model by double-clicking the Observer block.



Add Observer Port blocks in the Observer model. Double-click on the observer ports to open the Manage Observer Block dialog box. The internal variables of the FMU are listed under the FMU block.

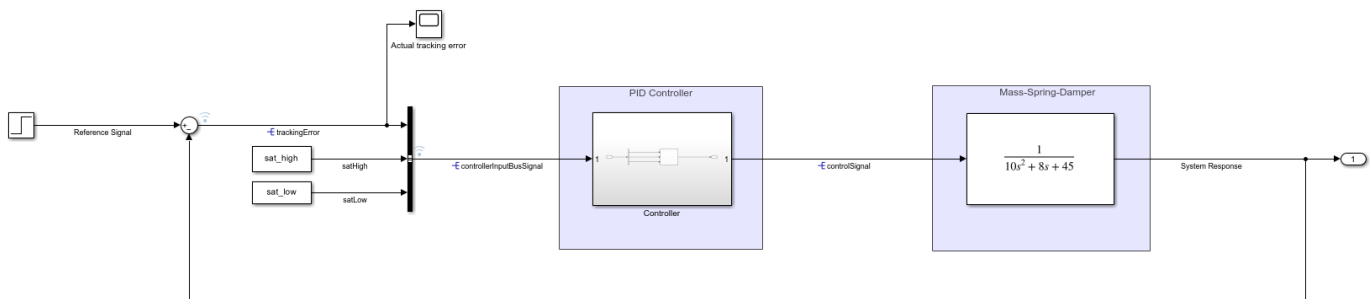


Connect the observer ports to internal variables of the FMU. For more information on connecting observer ports to observable entities, see “Access Model Data Wirelessly by Using Observers” on page 4-2.

The outputs of the observer ports can be used to wirelessly access the internal variables of the FMU along with other signals in the main model.

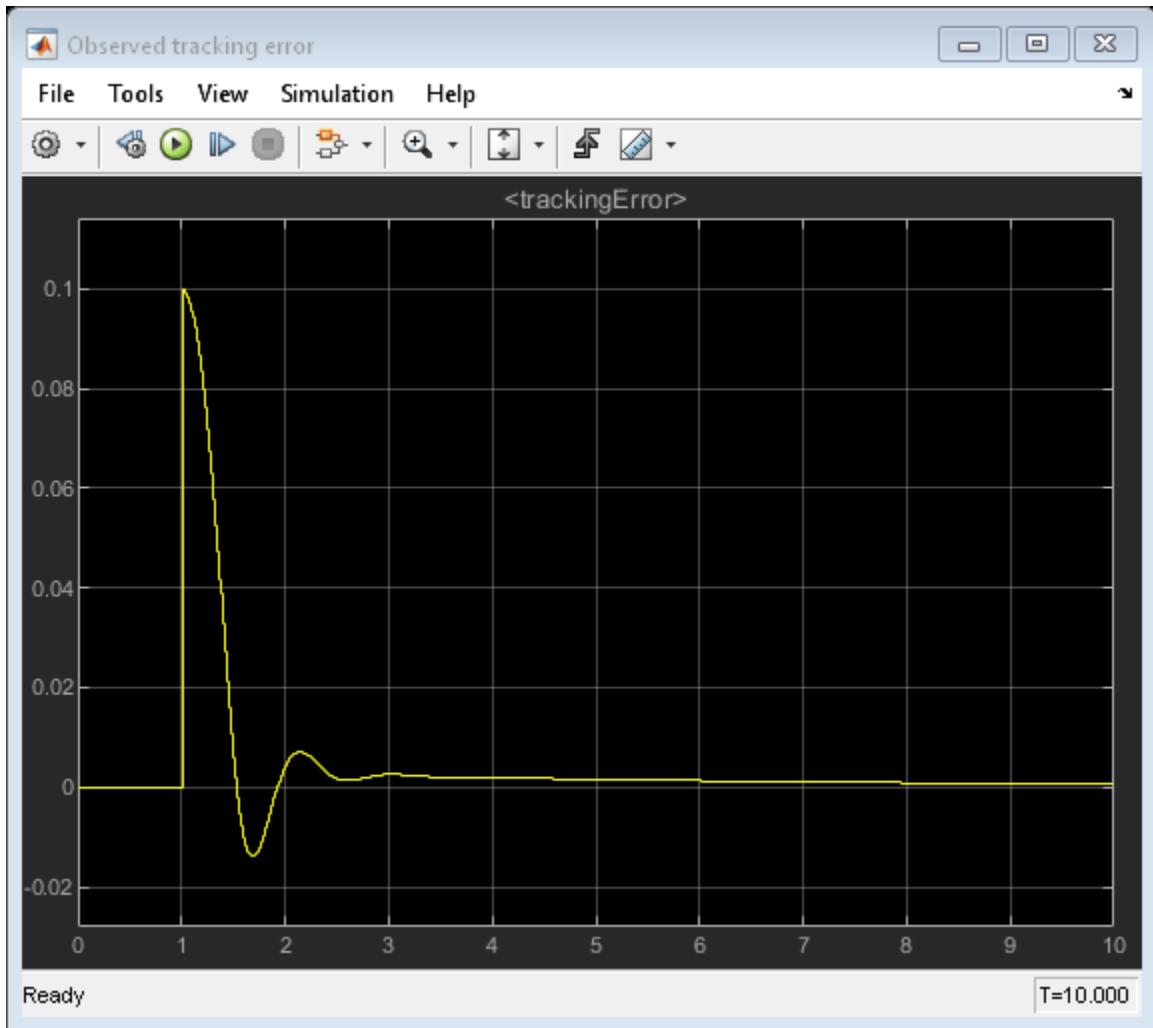
Compare Observer Output and Actual Signals

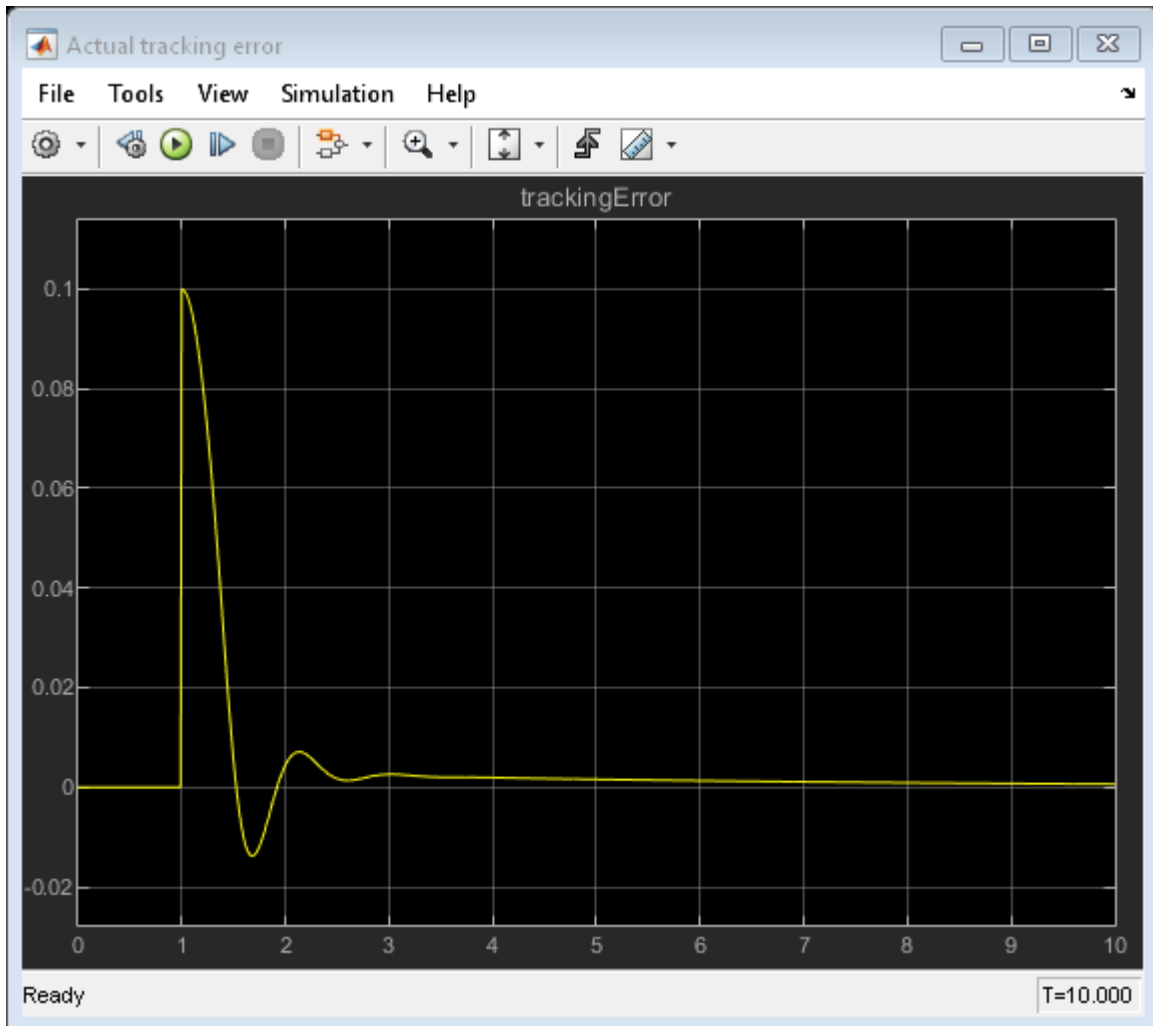
Open the original model that was used to create the FMU. Compare the tracking error and control inputs with their corresponding observed internal variables for identical controller parameters and reference signals.



Copyright 2023 The MathWorks, Inc.

Open the scope blocks to check if the tracking errors are identical for the actual and observed signals.





You can now use the observed signals to tune the PID controller inside the FMU in the main model.

See Also

Observer Port | Observer Reference

More About

- “Access Model Data Wirelessly by Using Observers” on page 4-2

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

- “SIL Verification for a Subsystem” on page 5-2
- “Use SIL/PIL to Verify Generated Code from an Earlier Release” on page 5-6
- “Code Generation Verification Workflow with Simulink Test” on page 5-14
- “Import Test Cases for Equivalence Testing” on page 5-19
- “Test Integrated Code” on page 5-27

SIL Verification for a Subsystem

In this section...

“Create a SIL Verification Harness for a Controller” on page 5-2

“Configure and Simulate a SIL Verification Harness” on page 5-4

“Compare the SIL Block and Model Controller Outputs” on page 5-4

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. 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 model code coverage. You cannot collect coverage on the SIL or PIL blocks. 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.

When you create an equivalence test that compares normal and SIL or PIL simulation modes, a separate test harness is used to test each mode. However, if you are equivalence testing an atomic subsystem or Model block, a single test harness can be used for both the normal and SIL or PIL simulations. For information about when the a single harness is used for atomic subsystem equivalence tests, see “Generate Tests and Test Harnesses for a Model or Components” on page 6-24.

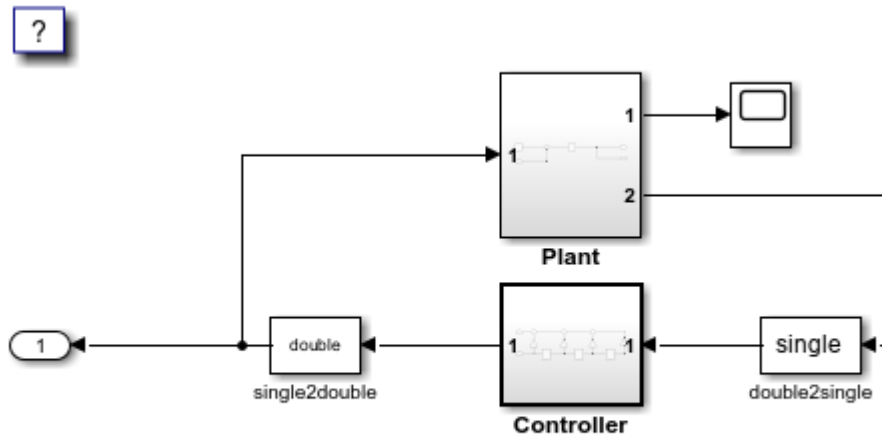
This example models a closed-loop controller-plant system. The controller regulates the plant output.

Create a SIL Verification Harness for a Controller

Create a SIL verification harness using data that you log from a closed-loop controller-plant system. The controller subsystem regulates the plant output. You need an Embedded Coder license for this example. Another way to create a SIL harness is with the Create Test for Model Component Wizard (see “Generate Tests and Test Harnesses for a Model or Components” on page 6-24 and “Create and Run a Back-to-Back Test” on page 6-41).

- 1 Open the example model by entering this command in the MATLAB Command Window.

```
openExample('ecoder/SILPILVerificationExample', ...
            supportingFile='SILBlock.slx')
```



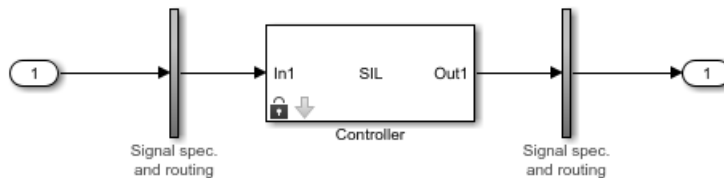
Copyright 2004-2020 The MathWorks, Inc.

- 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 for '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

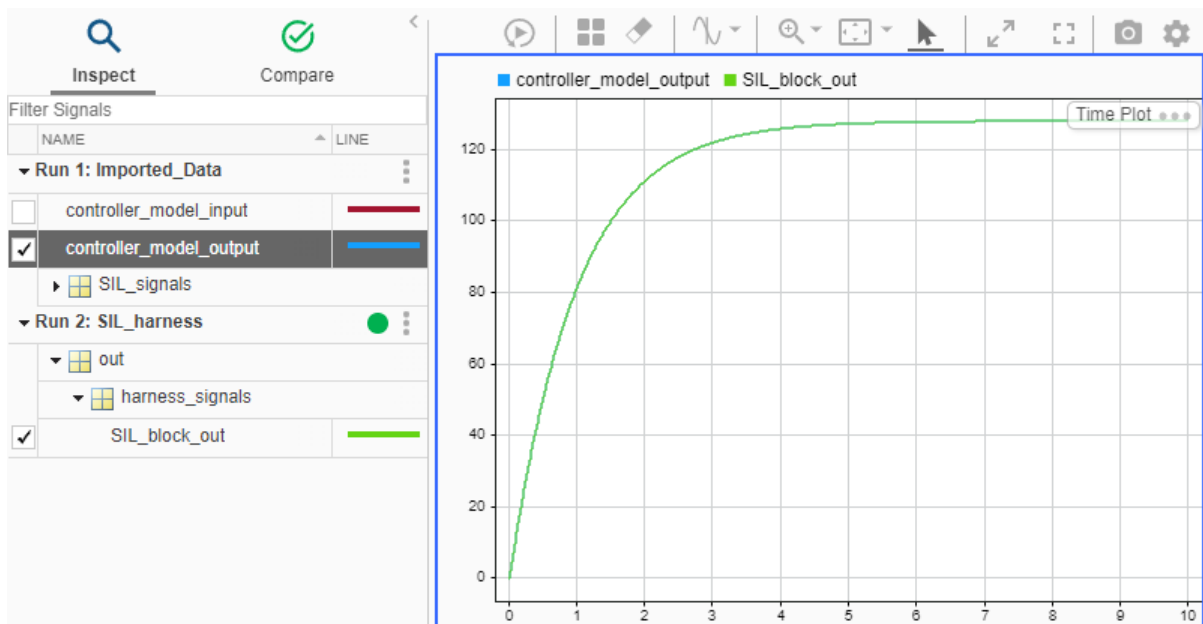

```
set_param("SIL_harness",SignalLogging="on",SignalLoggingName=...
" harness_signals",SignalLoggingSaveFormat="Dataset");
```
- 3 Right-click the output signal of the SIL block and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `SIL_block_out`. Select **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, in the **Simulation** tab, in the **Review Results** section, click **Data Inspector**  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 **Name**, select all of the check boxes 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”.



- 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

- “Create and Run a Back-to-Back Test” on page 6-41
- “Generate Tests and Test Harnesses for a Model or Components” on page 6-24
- “Configure and Run SIL Simulation” (Embedded Coder)
- “Use SIL/PIL to Verify Generated Code from an Earlier Release” on page 5-6
- “Generate Subsystem Code as Separate Function and Files” (Simulink Coder)

Use SIL/PIL to Verify Generated Code from an Earlier Release

For an atomic subsystem, you can use SIL/PIL simulation in the current release to verify code that was generated for that subsystem in a previous release. You do not have to regenerate the code, which saves test harness generation time. You cannot reuse generated code for test harnesses for whole models or Model blocks.

Note You must have an Embedded Coder license to reuse generated code from an earlier release.

Reuse Generated Code

In an earlier release, if you created a test harness that generated code and verified it using SIL/PIL, you can reuse that code, rather than regenerating it, in the current release. To reuse generated code, you must know the location of the folder that contains the code. The steps for reusing generated code and verifying it using SIL/PIL are:

- 1 Right-click an atomic subsystem in your model and select **Test Harness > Create for '<subsystem_name>'**.
- 2 In the Advanced Properties tab of the Create Test Harness dialog box:
 - Set Select **Verification Mode** to Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL).
 - Select **Use generated code to create SIL/PIL block**.
 - In the **Build folder** text box, enter the full path to the folder that contains the previously generated code.
- 3 Click OK to create the test harness using the generated code.
- 4 Create another normal or SIL/PIL mode test harness for the model that does not use generated code.
- 5 Create a test case and run the test.
- 6 Analyze the test results and verify that the results match the results produced by the same code in the earlier release.

To use previously generated code verified using a SIL/PIL subsystem programmatically, use the `ExistingBuildFolder` property of `sltest.harness.create` or `sltest.harness.set` to specify the location of the generated code.

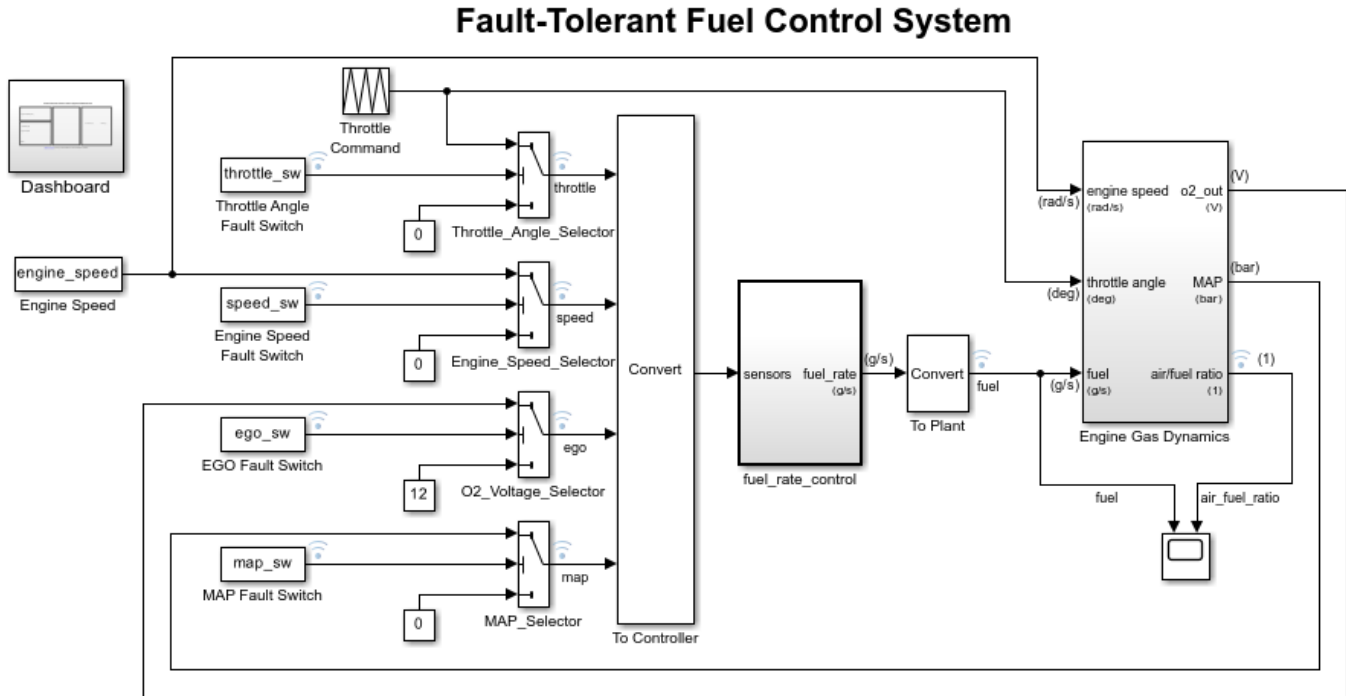
SIL Verification of a Subsystem using Code Generated from an Earlier Release

This example shows how to use code that was generated in a previous release to verify that the model in the current release continues to work as expected. In the current release you can create a test harness using the previously generated code, rather than having to regenerate it.

The model in this example is `sldemo_fuelsys_ex`, which represents a fuel control system for a gasoline engine. The system under test is the `fuel_rate_control` subsystem. A normal mode simulation in the current release is compared to a SIL mode simulation from an earlier release.

Open the Fuel Control System Model

sldemo_fuelsys_ex

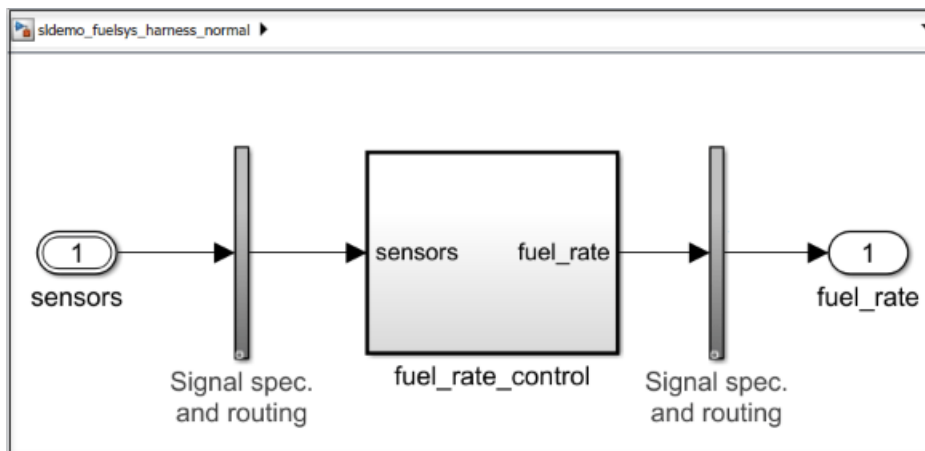


[Open the Dashboard](#) subsystem to simulate any combination of sensor failures.

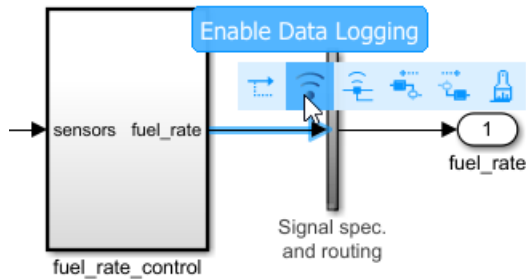
Copyright 1990-2023 The MathWorks, Inc.

Create the Normal Test Harness and Select the Signal to Log

1. Right-click the `fuel_rate_control` subsystem and select **Test Harness > Create for 'fuel_rate_control'**. The Create Test Harness dialog box opens.
2. Change the **Name** of the harness to `sldemo_fuelsys_harness_normal` and click **OK** to create the normal mode harness.



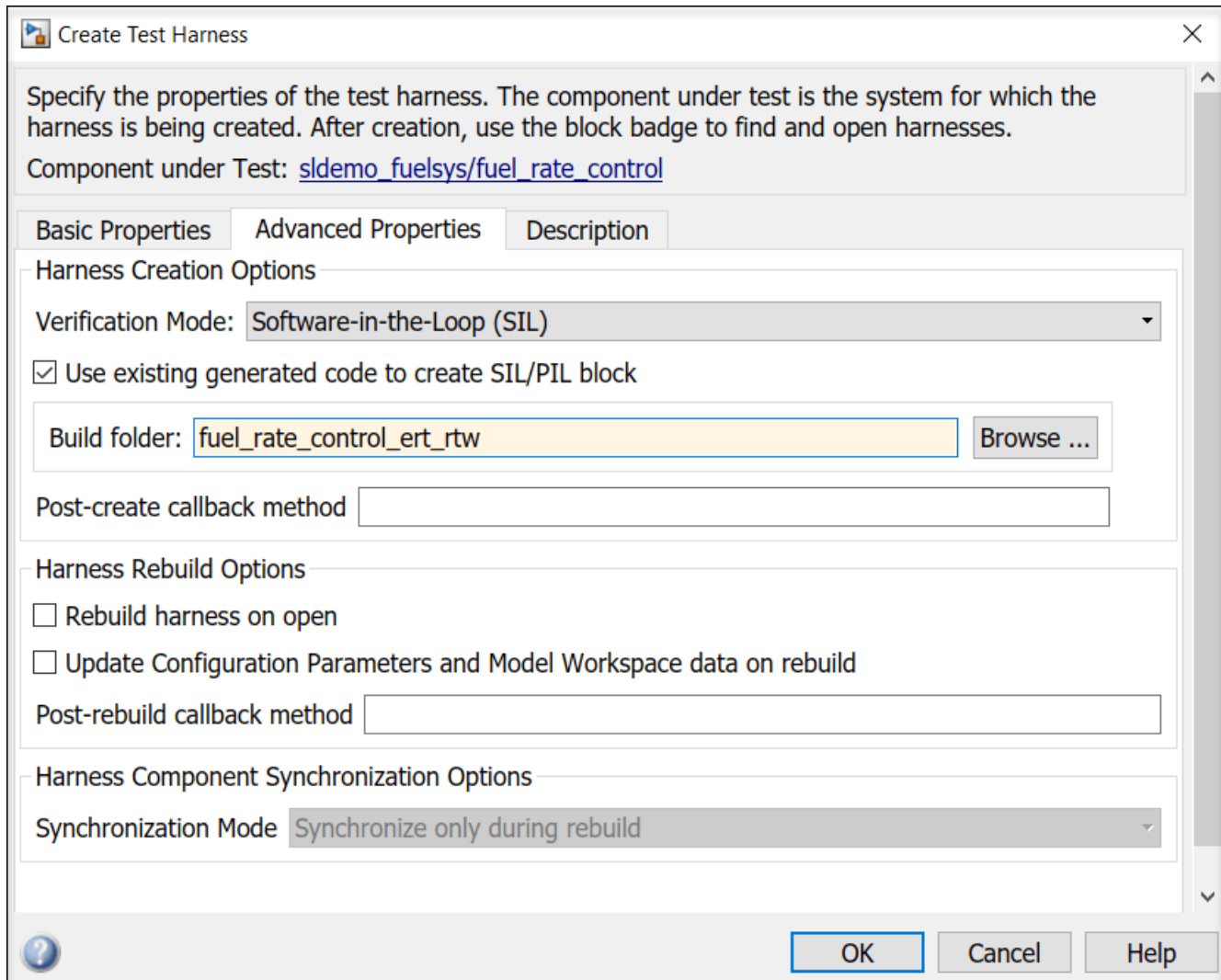
3. Select the signal exiting the subsystem in the test harness. Pause on the ellipsis to open the action bar and select **Enable Data Logging**.



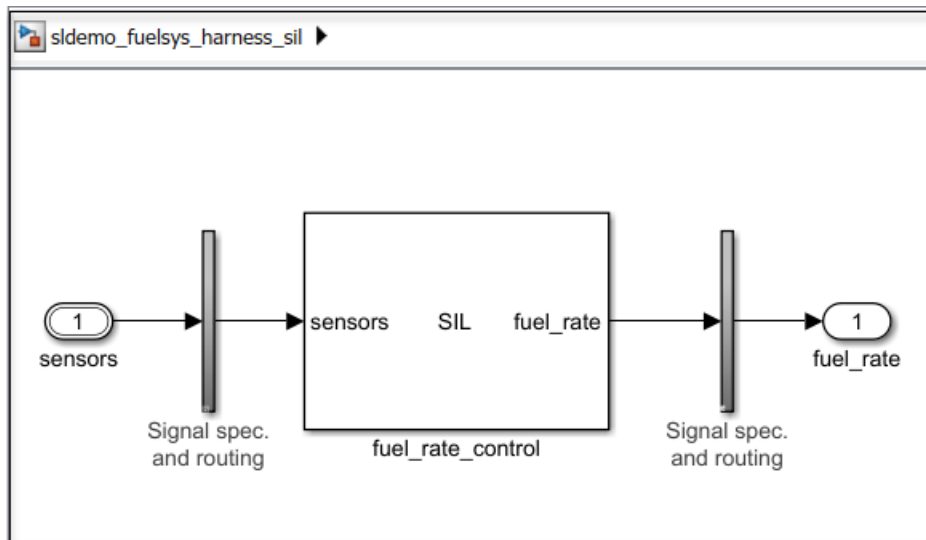
4. Close the `sldemo_fuelsys_harness_normal` harness. You do not need to explicitly save the harness.

Create the SIL Test Harness and Select the Signal to Log

1. Right-click the `fuel_rate_control` subsystem again and select **Test Harness > Create for 'fuel_rate_control'** to open the Create Test Harness dialog box.
2. Change the **Name** of the harness to `sldemo_fuelsys_harness_sil`.
3. On the **Advanced Properties** tab, set the harness as a SIL harness that verified code generated in an earlier release.
 - 1 Change the **Verification Mode** to Software-in-the-Loop (SIL).
 - 2 Select **Use existing generated code to create SIL/PIL block**.
 - 3 In **Build folder**, enter `fuel_rate_control_ert_rtw`, which is the name of the folder that contains the code verified using the SIL subsystem in the earlier release.



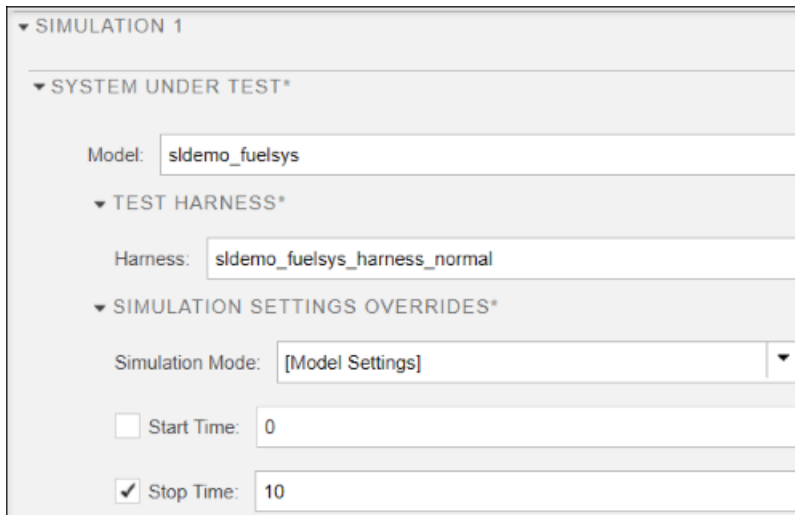
4. Click OK to create the SIL harness.



5. Select the signal exiting the subsystem in the test harness. Pause on the ellipsis to open the action bar and select **Enable Data Logging**.

Create an Equivalence Test Case

1. Use `sltestmgr` to open the Test Manager.
2. Click **New > Test File**. Right-click on the test file and change its name to `SIL_reuse`.
3. Delete New Test Case 1.
4. Highlight New Test Suite 1 and click **New > Equivalence Test**.
5. Change the name of New Test Case 1 to `SIL equivalence test case`.
6. In the **System Under Test** section for **Simulation 1**,
 - 1 Set the **Model** to `sldemo_fuelsys`.
 - 2 Under **Test Harness > Harness**, select `sldemo_fuelsys_harness_normal`.
 - 3 Under **Simulation Settings Overrides**, select **Stop Time** and set it to 10.



▼ SIMULATION 1

▼ SYSTEM UNDER TEST*

Model: sldemo_fuelsys

▼ TEST HARNESS*

Harness: sldemo_fuelsys_harness_normal

▼ SIMULATION SETTINGS OVERRIDES*

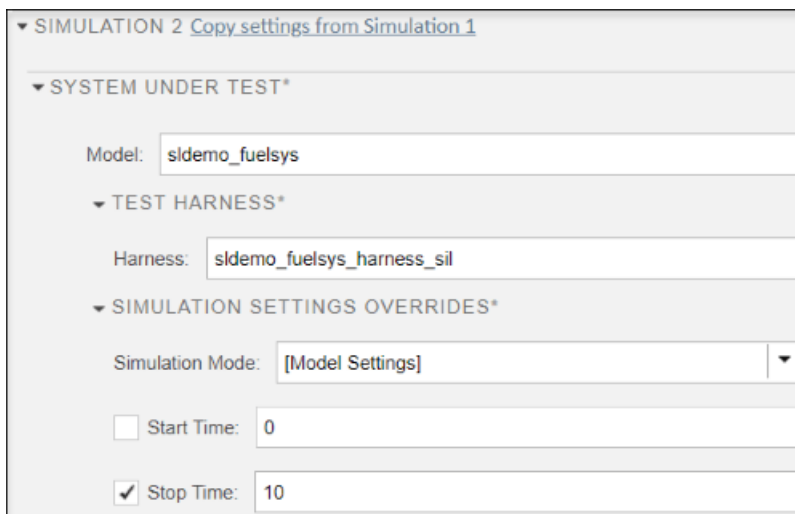
Simulation Mode: [Model Settings]

Start Time: 0

Stop Time: 10

7. For **Simulation 2**,

- 1 Set the **Model** to sldemo_fuelsys.
- 2 Under **Test Harness > Harness**, select sldemo_fuelsys_harness_sil.
- 3 Under **Simulation Settings Overrides**, leave **Release** as Current. Set **Stop Time** and set it to 10.



▼ SIMULATION 2 [Copy settings from Simulation 1](#)

▼ SYSTEM UNDER TEST*

Model: sldemo_fuelsys

▼ TEST HARNESS*

Harness: sldemo_fuelsys_harness_sil

▼ SIMULATION SETTINGS OVERRIDES*

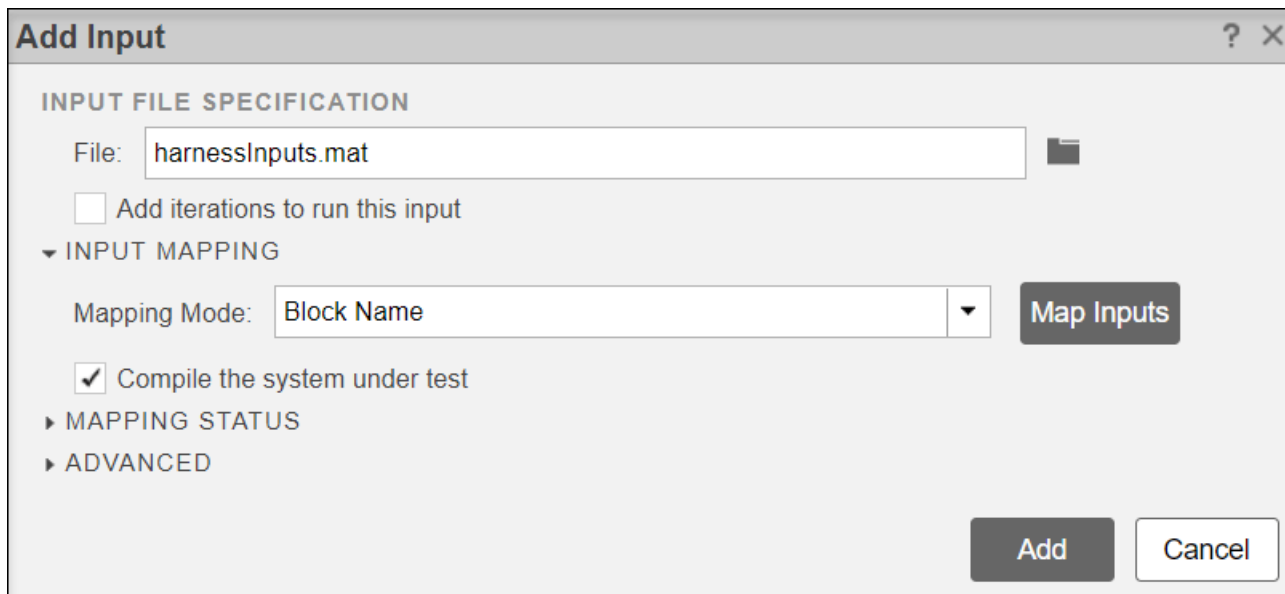
Simulation Mode: [Model Settings]

Start Time: 0

Stop Time: 10

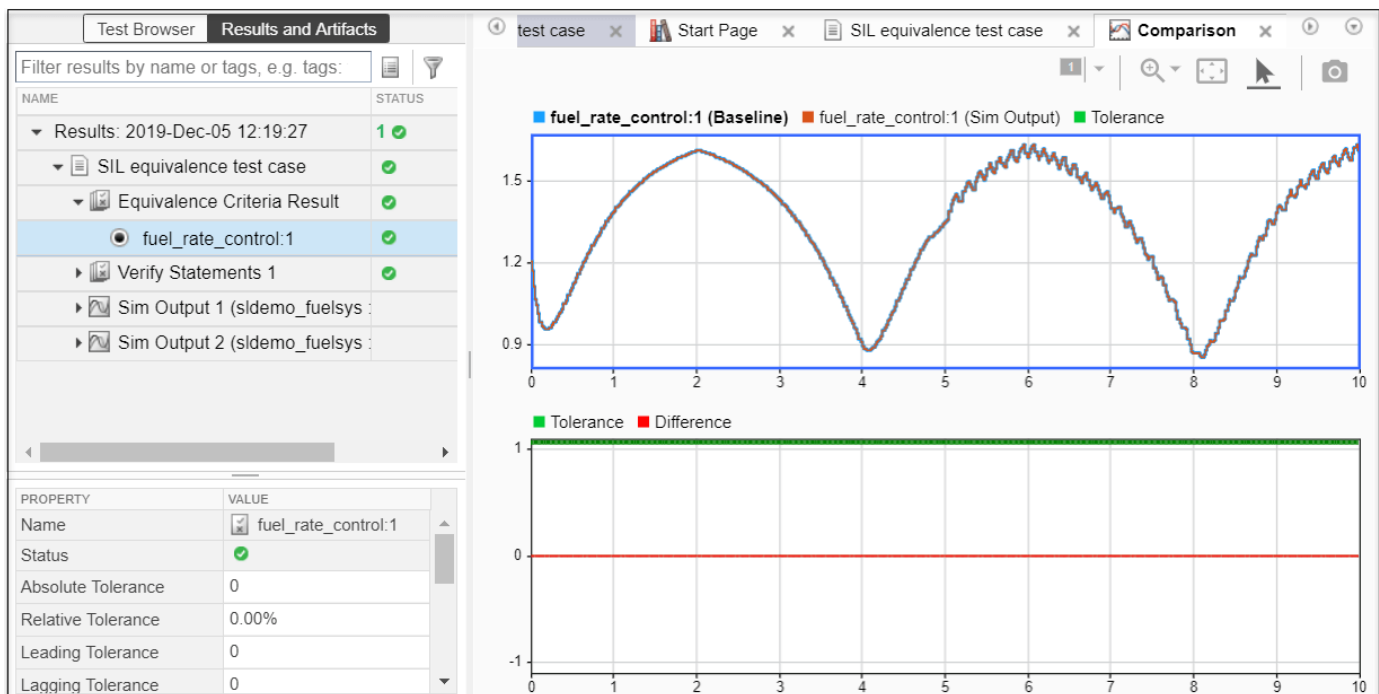
Specify the Harness Inputs

For both simulations, in the **Inputs** sections, click **Add** and, in the Add Input dialog box, in the **File** field, enter harnessInputs.mat. Click **Map Inputs** and then click **Add** to set up the inputs.



Run the Test and View the Output and Results

Click **Run** to run the equivalence test. In the **Results and Artifacts** pane, expand Equivalence Criteria Result to view the output.



The upper plot shows the output of both test harnesses. The lower plot shows that the difference between **fuel_rate_control:1 (Baseline)** and **fuel_rate_control:1 (Sim Output)** is zero. This difference means that the two results plots match exactly. This matching indicates that the code

verified using SIL from the earlier release and the code generated in the current release produce the same results.

See Also

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

More About

- “Create or Import Test Harnesses and Select Properties” on page 2-13
- “SIL Verification for a Subsystem” on page 5-2

Code Generation Verification Workflow with Simulink Test

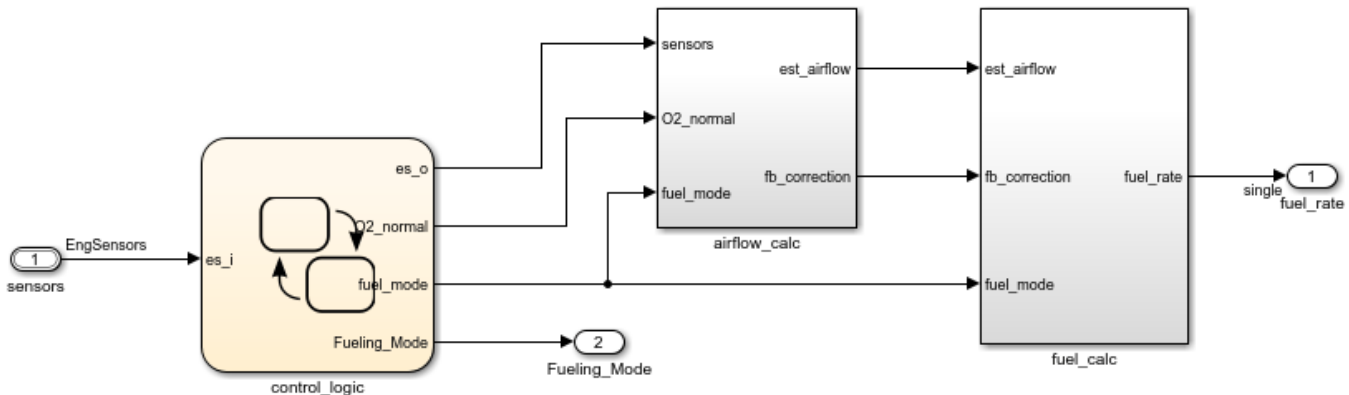
Perform code generation verification for a model. Copyright 2015 The MathWorks, Inc.

This example shows how to perform code generation verification (CGV) for a model using test harnesses, Test Sequence blocks and the test manager. Switch to a directory with write permissions.

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

Code Generation Verification Workflow with Simulink Test

This model is used to show how to perform code generation verification (CGV) using test harnesses, Test Sequence blocks and the test manager. To see the demo, execute `showdemo sltestFuelRateControlCGVDemo` in MATLAB(R).



Copyright 2015 The MathWorks, Inc.

Description of the Model

This example uses a model of a fuel-rate controller for a gasoline engine. The controller uses four sensors from the system to determine the proper fuel rate. The four sensors used from the system are throttle angle, speed, EGO and manifold absolute pressure [MAP].

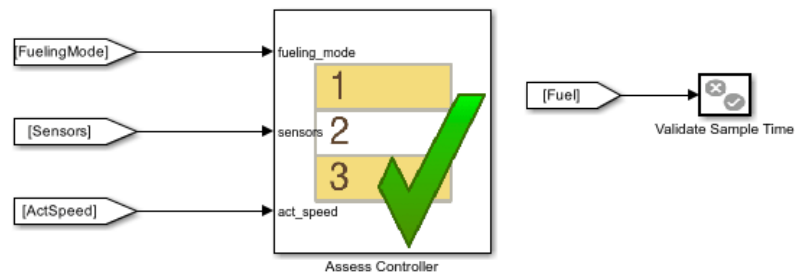
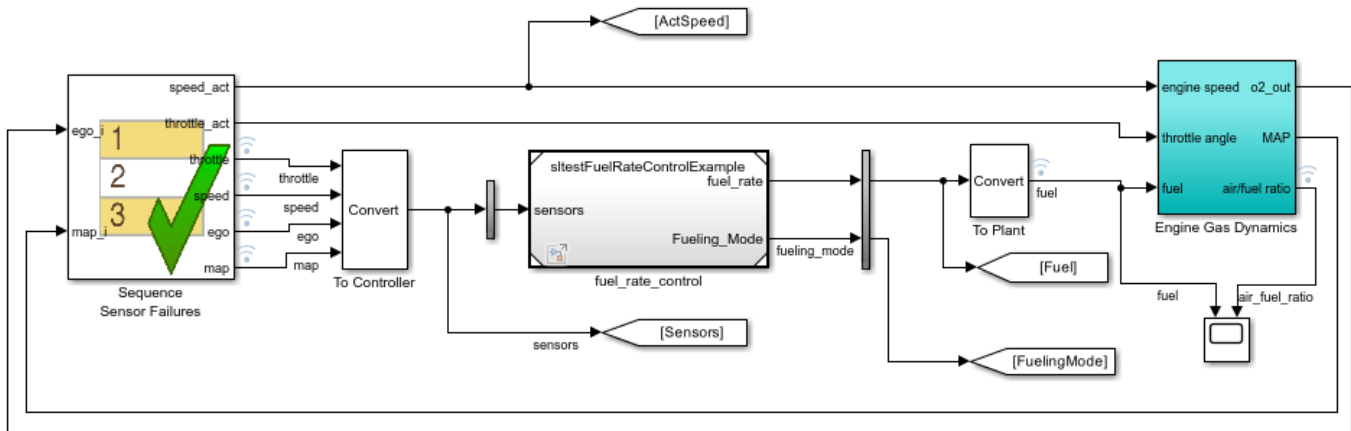
The model uses three subsystems to calculate the fuel rate using the sensor inputs: `control_logic`, `airflow_calc`, and `fuel_calc`. The core control logic is implemented in the Stateflow® chart named `control_logic`. The control logic handles single sensor failures and engine overspeed protection. If a single sensor fails, operation continues but the air/fuel mixture is richer to allow smoother running at the cost of higher emissions. If more than one sensor has failed, the engine shuts down as a safety measure, since the air/fuel ratio cannot be controlled reliably.

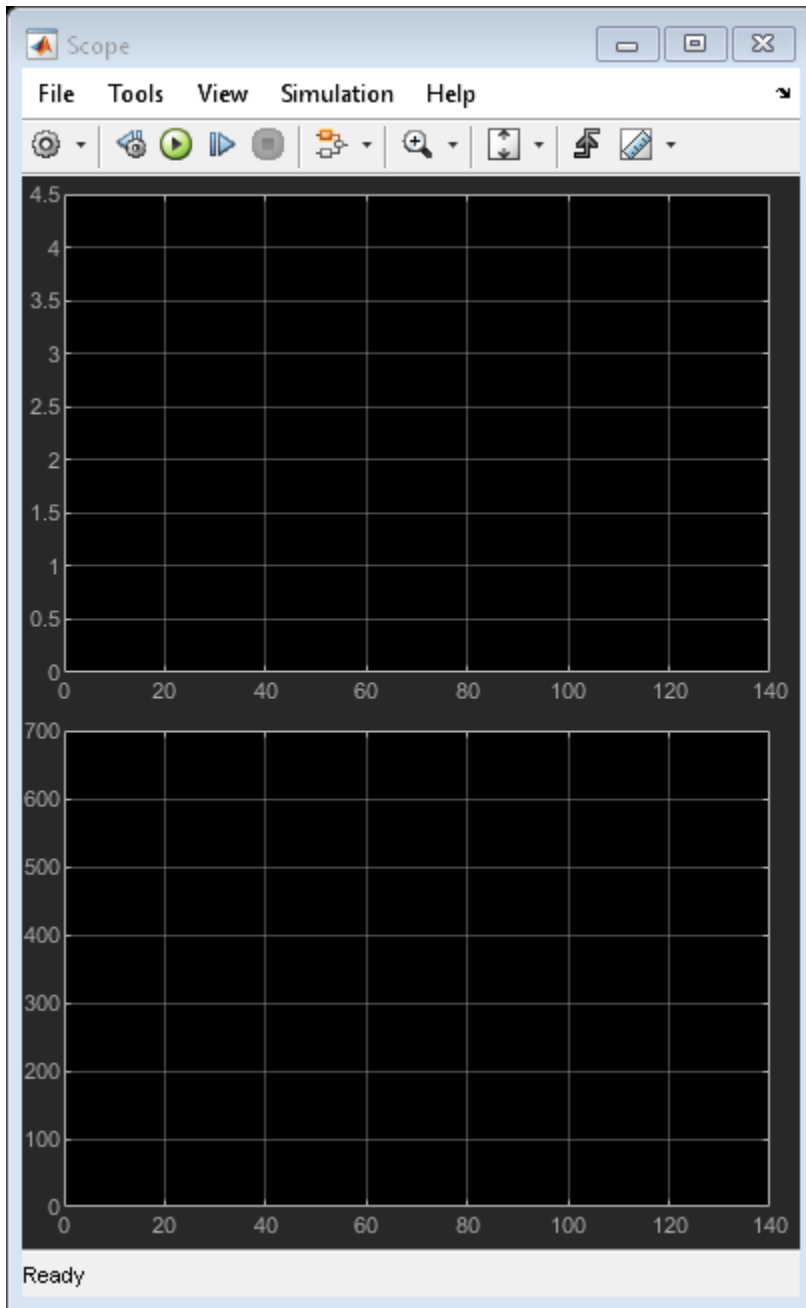
The model estimates the airflow rate and multiplies the estimate by the reciprocal of the desired ratio to give the fuel rate.

Opening the Test Harness

A Test Harness named `fuel_rate_control_cgv` has been created for the entire model. The harness can be opened by clicking on the perspectives pullout icon in the bottom-right corner of the model canvas and choosing the `fuel_rate_control_cgv` thumbnail. Ensure that the top level of the model is in view before you click on the icon. Alternately, the harness can be opened using the following API:

```
sltest.harness.open mdl, 'fuel_rate_control_cgv';
```





Modeling the Plant

The test harness has been modeled as a closed-loop test with a Test Sequence block to drive fuel-rate controller. The computed `fuel_rate` from the output of the controller is used to drive a model of the gasoline engine. The fuel rate combines with the actual air flow in the Engine Gas Dynamics subsystem to determine the resulting mixture ratio as sensed at the exhaust. Feedback from the oxygen sensor to the Test Sequence block provides a closed-loop adjustment of the rate estimation in order to maintain the ideal mixture ratio.

Notice that the plant has been modeled in the test harness instead of the main model. The main model is free of extraneous clutter so that code can be easily built for an ECU with minimal changes to the model.

Modeling Sensor Failures

The Test Sequence block named `Sequence Sensor Failures` models various sensor failure and engine overspeed scenarios. It accepts feedback from the plant and drives the controller with sensor data. This modeling pattern allows the Test Sequence block to control the feedback signals received by the Controller block and function as a canvas for authoring test cases. Open the Test Sequence block to see the modeled test scenarios.

```
open_system('fuel_rate_control_cgv/Sequence Sensor Failures');
```

Test Scenarios

For the first 10 seconds of simulation, the test is in stabilization mode, where the closed loop inputs from the plant is passed through to the controller. The throttle and speed inputs are set to nominal values that are within the normal operating envelope of the controller. The `Stabilize_Engine` step models this state.

The Test then steps through the following modes:

- 1 `Test_Overspeed`: The throttle is ramped from 30 to 700
- 2 `Reset_To_Normal_Speed`: The throttle is ramped down to 400
- 3 `Test_EGO_Fault`: Simulate failure for 3 sec, then return to normal state
- 4 `Test_Throttle_Fault`: Simulate failure for 3 sec, then return to normal state
- 5 `Test_Speed_Fault`: Simulate failure for 3 sec, then return to normal state
- 6 `Test_Map_Fault`: Simulate failure for 3 sec, then return to normal state
- 7 `Test_Multi_Fault`: Simulate MAP and EGO failure for 3 sec
- 8 `Reset_MAP`: Normalize MAP sensor, and simulate just EGO failure for 3 sec
- 9 `Reset_To_Normal`: Terminate the test

Test Assessments

The Test Sequence block `Assess Controller` verifies the controller output for the various test cases modeled by the `Sequence Sensor Failures` block. The following assessments are modeled:

- 1 Assert that the fueling mode is in `Warmup` mode for the first 4.8 seconds
- 2 Assert that fueling mode switches to `Overspeed` mode when the actual speed exceeds 628
- 3 Assert that the fueling mode is not in `Single_Failure` mode when multiple sensors have failed.

```
open_system('fuel_rate_control_cgv/Assess Controller');
```

Running a Simulation

Simulate the test harness by clicking `Play` in the toolstrip and observe the `fuel_rate` and air-fuel ratio signals in the scope. Alternately, run the following command: `sim('fuel_rate_control_cgv')`

Notice that no assertions trigger during the simulation, which indicate that all assessments modeled in `Assess Controller` pass.

Configuring a back-to-back Test in the Test Manager

As part of code generation verification (CGV) for the controller system, it is important to assert that the functional behavior of the controller is same during normal and software-in-the-loop (SIL) simulation modes. The test manager is used to perform this verification.

Use the function `sltestmgr` to open the test manager and load the example test file using the function:

```
sltest.testmanager.TestFile('sltestFuelRateControlComparisonTestSuite.mldatx')
)
```

Modeling the Test Case

The equivalence test has been configured in the test manager so that the controller is simulated in normal and SIL mode and the numerical results are compared between these two runs. Explore the structure of the test case by clicking on different nodes of the test hierarchy in the **Test Browser**.

Running the Test Case

Run the test in the test manager using the function `sltest.testmanager.run`.

Alternately, In the test manager, select the CGV Test1 node in the **Test Browser** pane and click **Run** in the toolstrip. The pass/fail result is available in the **Results and Artifacts** pane.

Creating the Report

A report can then be generated to view the result of the equivalence test. Use the following commands to generate the report. You can also launch the report after creation using the API with the `LaunchReport` option set to `true`.

```
sltest.testmanager.report(cgvresult, 'cgvresult.zip', 'IncludeTestResults', int32(0));
close_system mdl, 0);
clear mdl;
```

Import Test Cases for Equivalence Testing

You can use the SIL/PIL Manager app in Embedded Coder to export test cases to the Test Manager. By using the app to export software-in-the-loop (SIL) or processor-in-the-loop (PIL) test cases, you do not have to write complicated test scripts for back-to-back testing.

Note You need both Simulink Test and Embedded Coder to use this feature.

Using **Export to Test Manager** in the SIL/PIL Manager app in **Automated Verification** mode exports a test case with two simulations, each in a different simulation mode. For back-to-back testing, you usually use Normal mode and SIL mode or Normal mode and PIL mode. When you export from the app, the Test Manager opens with the new equivalence test case in the Test Browser pane. If you export to a new test file, the Test Browser opens with a new test file and a new test suite for the test case. The test case includes a panel for each simulation (**SIMULATION 1** and **SIMULATION 2**). See SIL/PIL Manager (Embedded Coder) and “SIL/PIL Manager Verification Workflow” (Embedded Coder) for information on how to use the app to export a test case.

Settings for Test Case Simulations

The **System Under Test** in the SIL/PIL Manager app determines the settings for the test case simulations in the Test Manager. These settings for each type of system under test are described for exporting a test case that includes a SIL mode simulation. For a test that includes a PIL mode simulation, the settings are the same for each type of system under test.

- “Top-Level Model” on page 5-19
- “Model Block in SIL/PIL Mode” on page 5-20
- “Model Block or Reusable Library Subsystem in a Test Harness” on page 5-21

Top-Level Model

When the system under test is a Top model, the exported test case tests the entire model. The Test Harness field in Test Manager is blank.

Before exporting the test case, these settings are in the SIL/PIL Manager app.

System Under Test	Top model
Simulation Mode	Normal
SIL/PIL Mode	Software-in-the-Loop (SIL)

After exporting the test case, these settings are in the Test Manager for **SIMULATION 1**.

Property	Setting	Location in Test Manager
Model	Top model	SIMULATION 1 > SYSTEM UNDER TEST
Simulation mode	Normal	SIMULATION 1 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES

Property	Setting	Location in Test Manager
Override model blocks in SIL/PIL to normal mode	Selected To run the simulation in Normal mode, model blocks set to SIL/PIL mode are overridden.	SIMULATION 1 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES

After exporting the test case, these settings are in the Test Manager for **SIMULATION 2**.

Property	Setting	Location in Test Manager
Model	Top model	SIMULATION 2 > SYSTEM UNDER TEST
Simulation mode	Software-in-the-Loop (SIL)	SIMULATION 2 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES
Override model blocks in SIL/PIL to normal mode	Not selected The model blocks set to SIL or PIL mode run in SIL or PIL mode, respectively.	SIMULATION 2 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES

Model Block in SIL/PIL Mode

When the system under test is **Model blocks in SIL/PIL mode**, the exported test case is a Model block in SIL or PIL simulation mode. The Test Harness field in Test Manager is blank.

Before exporting the test case, these settings are in the SIL/PIL Manager app.

System Under Test	Model blocks in SIL/PIL mode
Top Model Mode	Normal

After exporting the test case, these settings are in the Test Manager for **SIMULATION 1**.

Property	Setting	Location in Test Manager
Model	Top model	SIMULATION 1 > SYSTEM UNDER TEST
Simulation mode	Normal	SIMULATION 1 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES
Override model blocks in SIL/PIL to normal mode	Selected To run the simulation in Normal mode, model blocks set to SIL/PIL mode are overridden.	SIMULATION 1 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES

After exporting the test case, these settings are in the Test Manager for **SIMULATION 2**.

Property	Setting	Location in Test Manager
Model	Top model	SIMULATION 2 > SYSTEM UNDER TEST
Simulation mode	Normal The system under test runs in SIL or PIL mode as set in the SIL/PIL Manager app. Other blocks run in Normal mode.	SIMULATION 2 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES
Override model blocks in SIL/PIL to normal mode	Not selected The model blocks set to SIL or PIL mode run in SIL or PIL mode, respectively.	SIMULATION 2 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES

Model Block or Reusable Library Subsystem in a Test Harness

When the system under test is a Model block or a reusable library subsystem in a test harness, the exported test case is that block or subsystem in SIL or PIL simulation mode. Use the SIL/PIL Manager app from within the test harness.

Before exporting the test case, these settings are in the SIL/PIL Manager app.

System Under Test	Name of Model block or reusable library subsystem in the test harness. This field is not editable because you cannot change an entire harness to SIL/PIL mode.
Simulation Mode	Normal
SIL/PIL Mode	Software-in-the-Loop (SIL)

After exporting the test case, these settings are in the Test Manager for **SIMULATION 1**.

Property	Setting	Location in Test Manager
Model	Model block name	SIMULATION 1 > SYSTEM UNDER TEST
Harness	Harness name	SIMULATION 1 > SYSTEM UNDER TEST > TEST HARNESS
Simulation mode	Normal	SIMULATION 1 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES
Override model blocks in SIL/PIL to normal mode	Selected To run the simulation in Normal mode, Model blocks set to SIL/PIL mode are overridden.	SIMULATION 1 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES

After exporting the test case, these settings are in the Test Manager for **SIMULATION 2**.

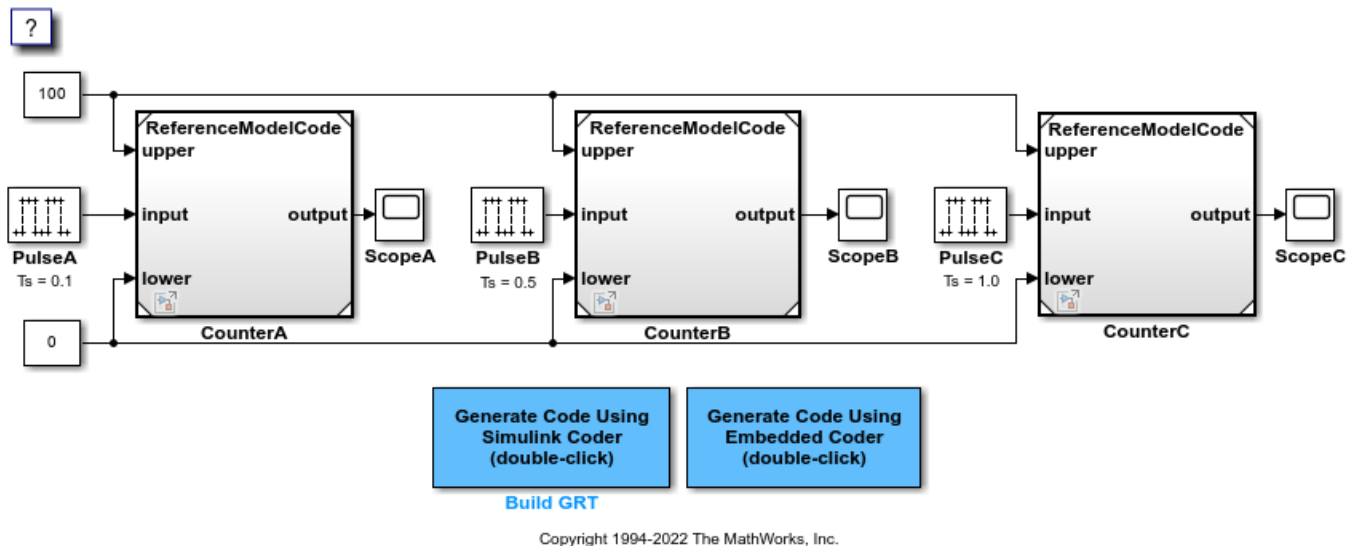
Property	Setting	Location in Test Manager
Model	Model block name	SIMULATION 2 > SYSTEM UNDER TEST
Harness	Harness name	SIMULATION 2 > SYSTEM UNDER TEST > TEST HARNESS
Simulation mode	Software-in-the-Loop (SIL)	SIMULATION 2 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES
Override model blocks in SIL/PIL to normal mode	Not selected The Model blocks set to SIL or PIL mode run in SIL or PIL mode, respectively.	SIMULATION 2 > SYSTEM UNDER TEST > SIMULATION SETTING OVERRIDES

Back-to-Back Testing a Model Using the SIL/PIL Manager App

This example shows how to perform back-to-back testing with a test case exported from the Embedded Coder SIL/PIL Manager app. The test case compares a model simulated in Normal mode and in Software-in-the-Loop (SIL) mode.

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

```
openExample('simulinkcoder/FilePackagingModelsCodeAndDataExample', ...
    'supportingFile', 'TopModelCode');
```



Note Steps 2 through 4 apply specifically to this TopModelCode model. These steps might not be needed for other models.

- 2 For this model, click **Signal Table** in the Simulation tab. Select **Test Point** and **Log data** for the CounterA, CounterB, and CounterC signals.

Model Data Editor					
Inports/Outports		Signals	Data Stores	States	Parameters
Instrumentation					
Source	Name	Test Point	Log Data		
CounterA		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
CounterB		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
CounterC		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
Pulse (Ts=0.1)		<input type="checkbox"/>	<input type="checkbox"/>		

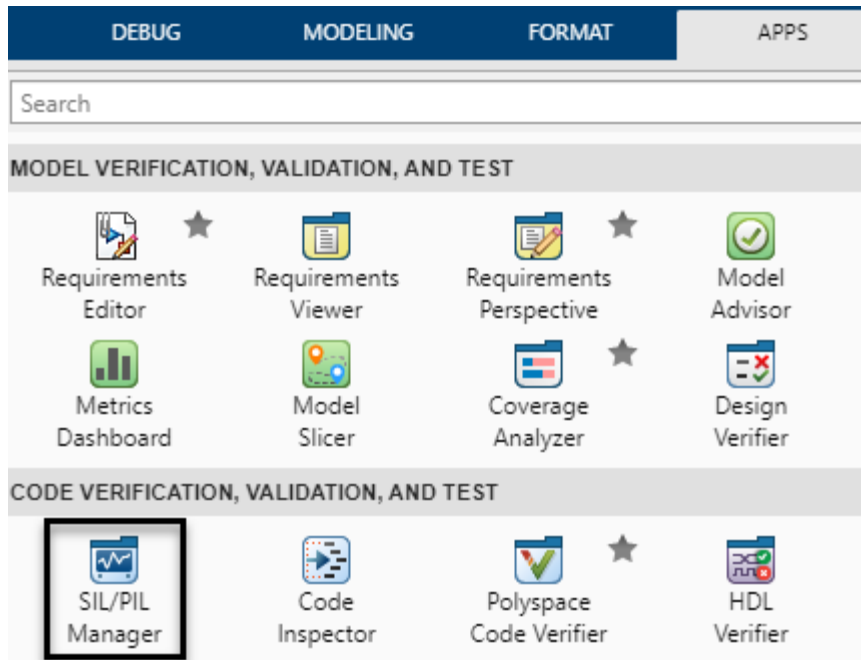
3 Right-click in the model and select **Model Configuration Parameters**. In the Configuration Parameters dialog box,

- In **Data Import/Export**, set the **Format** to Dataset.
- In **Code Generation > Interface**, select **signals** in the **Generate C API for** section.

Click **OK**.

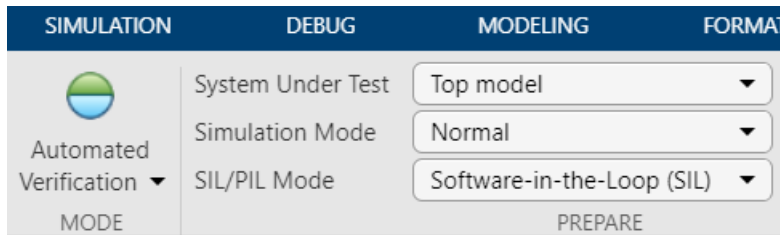
4 Right-click on the Model blocks and select **Open as Top Model**. In the Configuration Parameters dialog box, set the same items as in Step 3.

5 Expand the Apps tab in the model window and click **SIL/PIL Manager** under Code Verification, Validation, and Test.

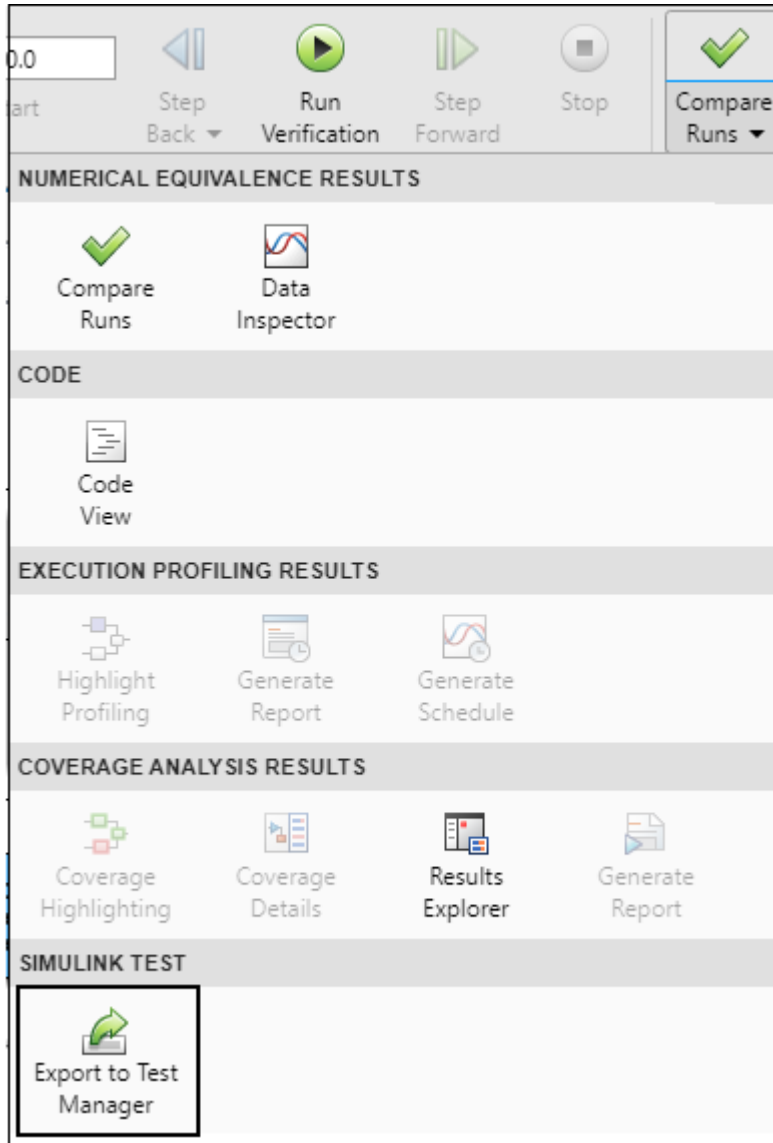


6 In the SIL/PIL Manager toolstrip, if they are not already selected, select

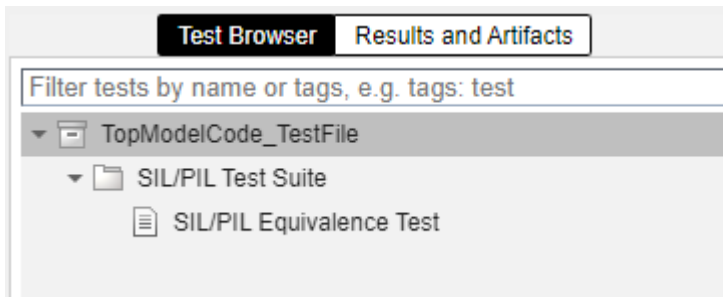
- **Automated Verification**
- **System Under Test** — Top Model
- **Simulation Mode** — Normal
- **SIL/PIL Mode** — Software-in-the-Loop (SIL)



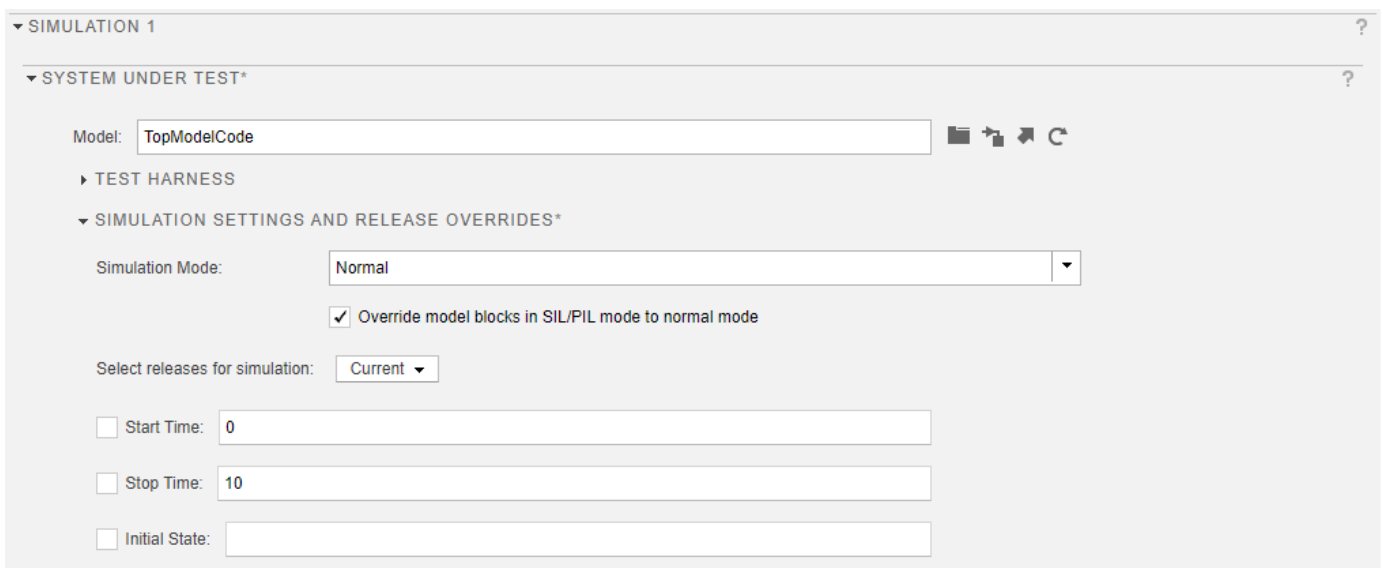
- 7 To export the test case, expand **Compare Runs** and click **Export to Test Manager**.



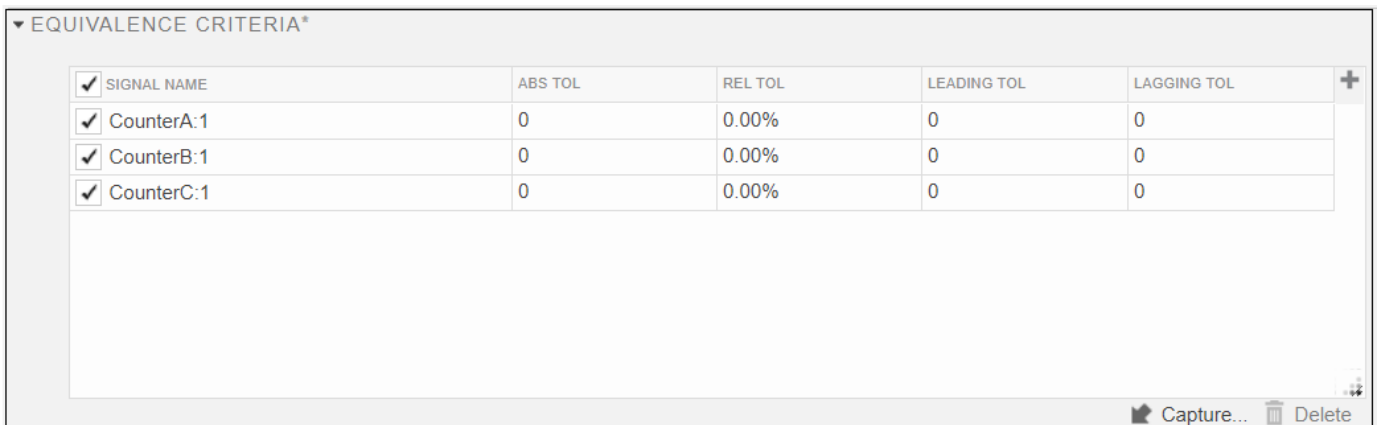
- 8 In the Export SIL/PIL Test Cases dialog box, use the default values and click OK. The Test Manager opens.
- 9 In the Test Manager, to see the imported test case and settings, expand TopModelCode_TestFile and SIL/PIL Test Suite in the Test Browser.



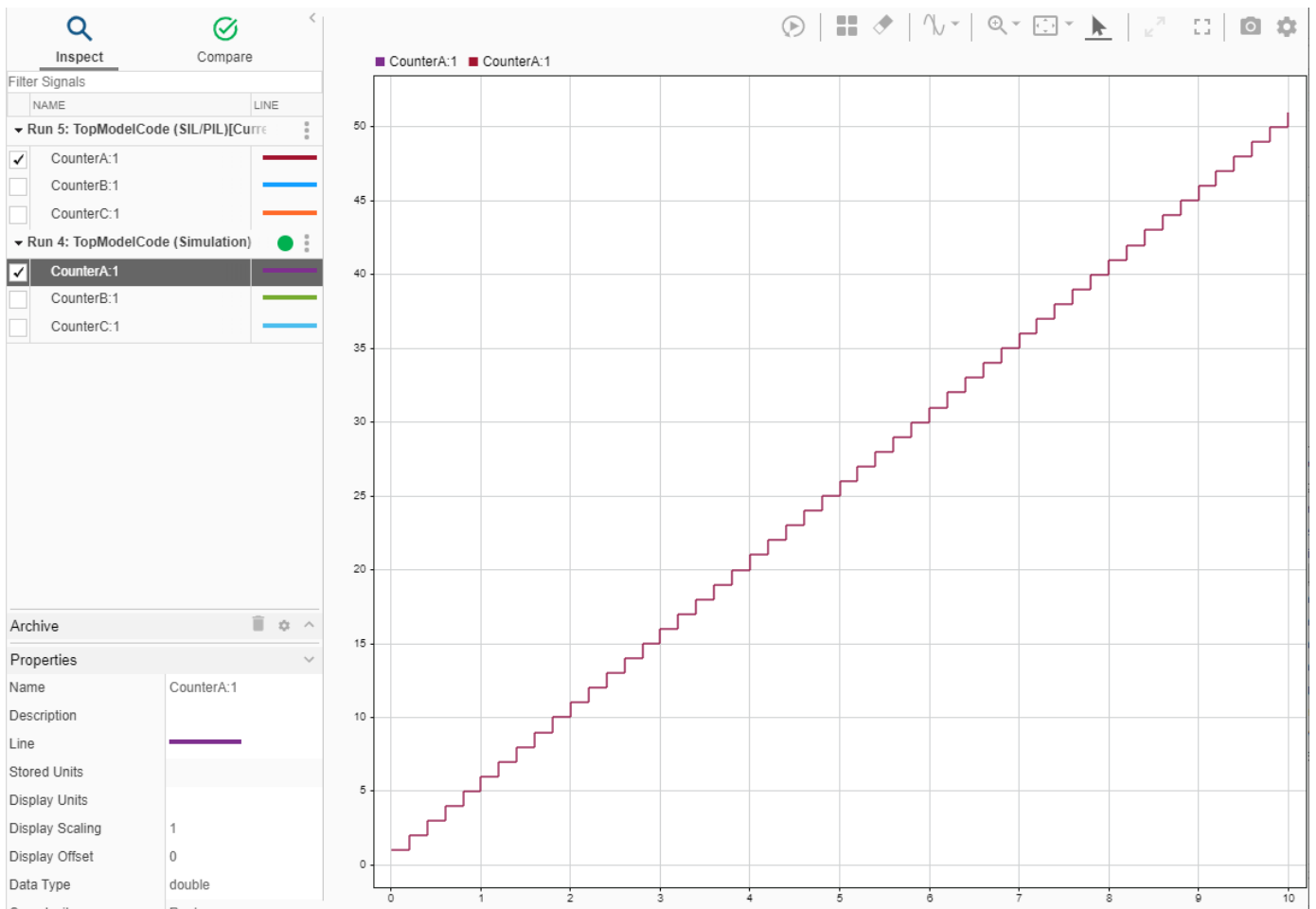
- 10 Select the SIL/PIL Equivalence test case. To see the settings for the simulation modes, expand the **SIMULATION 1** and **SIMULATION 2** sections. The expanded **SIMULATION 1** section is



- 11 Open the **Equivalence Criteria** section and click **Capture**. The model simulates and the section lists the signals to compare in the test case.



- 12 Click **Run** to run the test case.
- 13 In the Test Manager, the Results and Artifacts panel shows the pass/fail results. A Code Generation Report opens in a separate window.
- 14 Select one or more signals to plot the results.



The plot shows that the outputs from the two simulations are the same.

See Also

More About

- SIL/PIL Manager (Embedded Coder)
- “SIL/PIL Manager Verification Workflow” (Embedded Coder)
- “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” (Embedded Coder)

Test Integrated Code

In this section...

“Test Integrated C Code” on page 5-27

“Testing Code in an S-Function Block” on page 5-27

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.

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 2-54 and “Create or Import Test Harnesses and Select Properties” on page 2-13.

- If you change the test harness configuration parameters, you can push the configuration set to the main model. Click **Push Changes**, 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, select **Update Configuration Parameters and Model Workspace data on rebuild**. You can also select **Rebuild Harness on Open**, which rebuilds every time the harness opens.
 - For existing test harnesses, in the harness preview, select one or more of **Rebuild Harness > Rebuild on Open**, **Rebuild Harness without Compiling Model**, and **Update Harness Configuration Settings and Model Workspace**. The **Update Harness Configuration Settings and Model Workspace** option updates the settings every time a rebuild occurs.

Testing Code in an S-Function Block

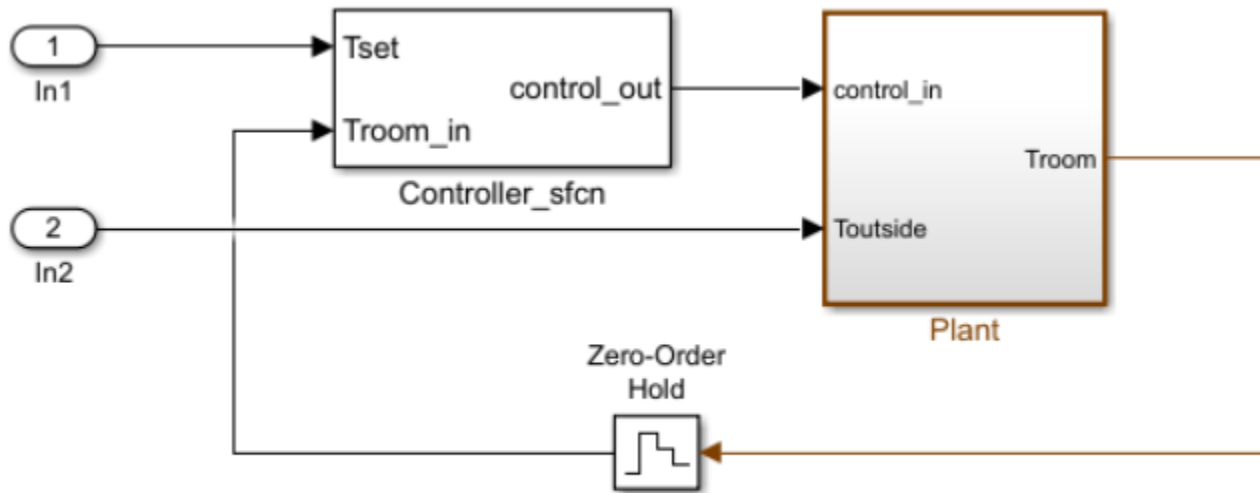
This example shows how to use a test harness to test code in an S-Function block. The main model for this example is a controller-plant model of an air conditioning/heat pump unit.

Note that this example works only on a 64-bit Windows platform.

S-Functions are computer language descriptions of Simulink blocks written in MATLAB, C, C++ or Fortran. You can test code wrapped in S-Functions by using 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.

Open the Model

```
sltestHeatpumpSfunExample
```



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.


Temperature Condition States

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
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} < T_{room_in}$	cooling	1	1	-1
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} > T_{room_in}$	heating	1	1	1

Create Test Case

1. On the **Apps** tab, under **Model Verification, Validation, and Test**, click **Simulink Test**. Then, on the **Tests** tab, click **Simulink Test Manager**.
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 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 the test harness properties. Set **Name** to test_harness_1 and Set **Sources** and **Sinks** to **None** and **Scope**.
3. Click **OK** to create the test harness.
4. In the Test Manager highlight **New Test Case 1**. In **System Under Test**, expand **Test Harness**. Refresh the test harness list and select test_harness_1 for the **Harness**.

Add Inputs and Set Simulation Parameters

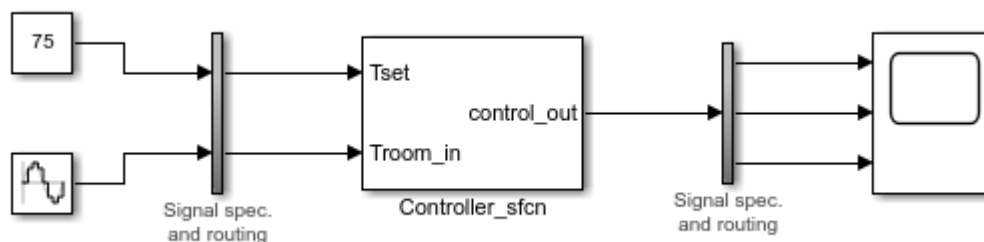
Create inputs in the test harness, with a constant Tset and a time-varying Troom_in.

1. In the test harness model, 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**. Then, click **OK**.

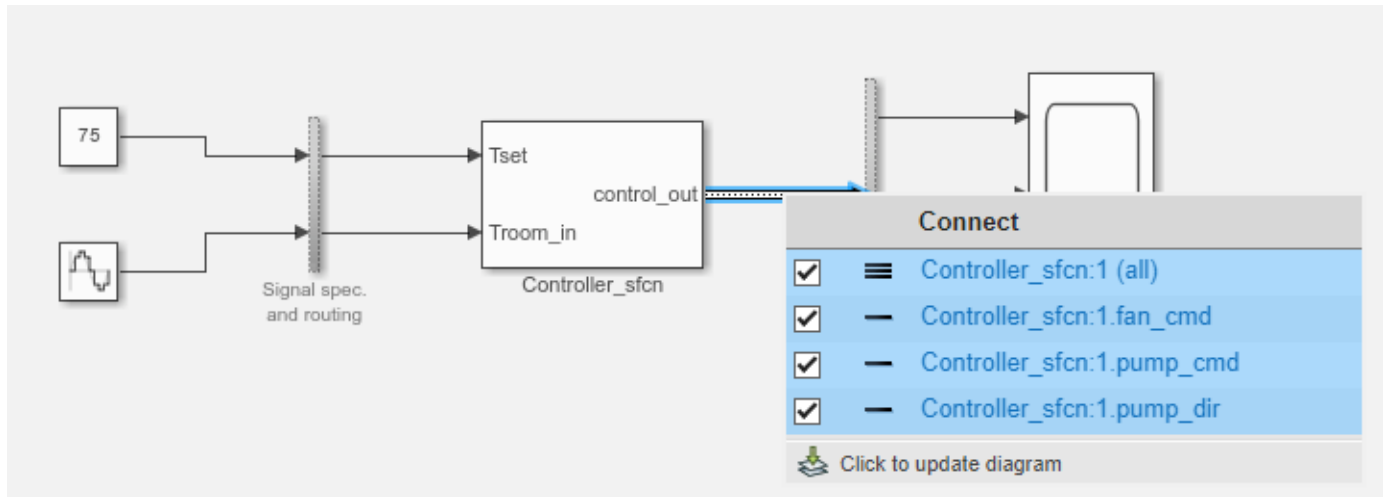
The resulting harness is:



4. In the Simulation toolstrip, set the **Stop Time** to 3600.

Obtain Baseline Data

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



2. In the Connect popup, select **Controller_sfcn:1(all)** and click **Click to update diagram**. Then, return to the Test Manager and click **Done** in the dialog box.

3. Under **Baseline Criteria**, click **Capture** to record a baseline data set from simulating the test harness. In the Capture Baseline dialog box, enter `sfcn_baseline` as the MAT format **File** name and click **Capture**. 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"/> Controller_sfcn:1	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* ?

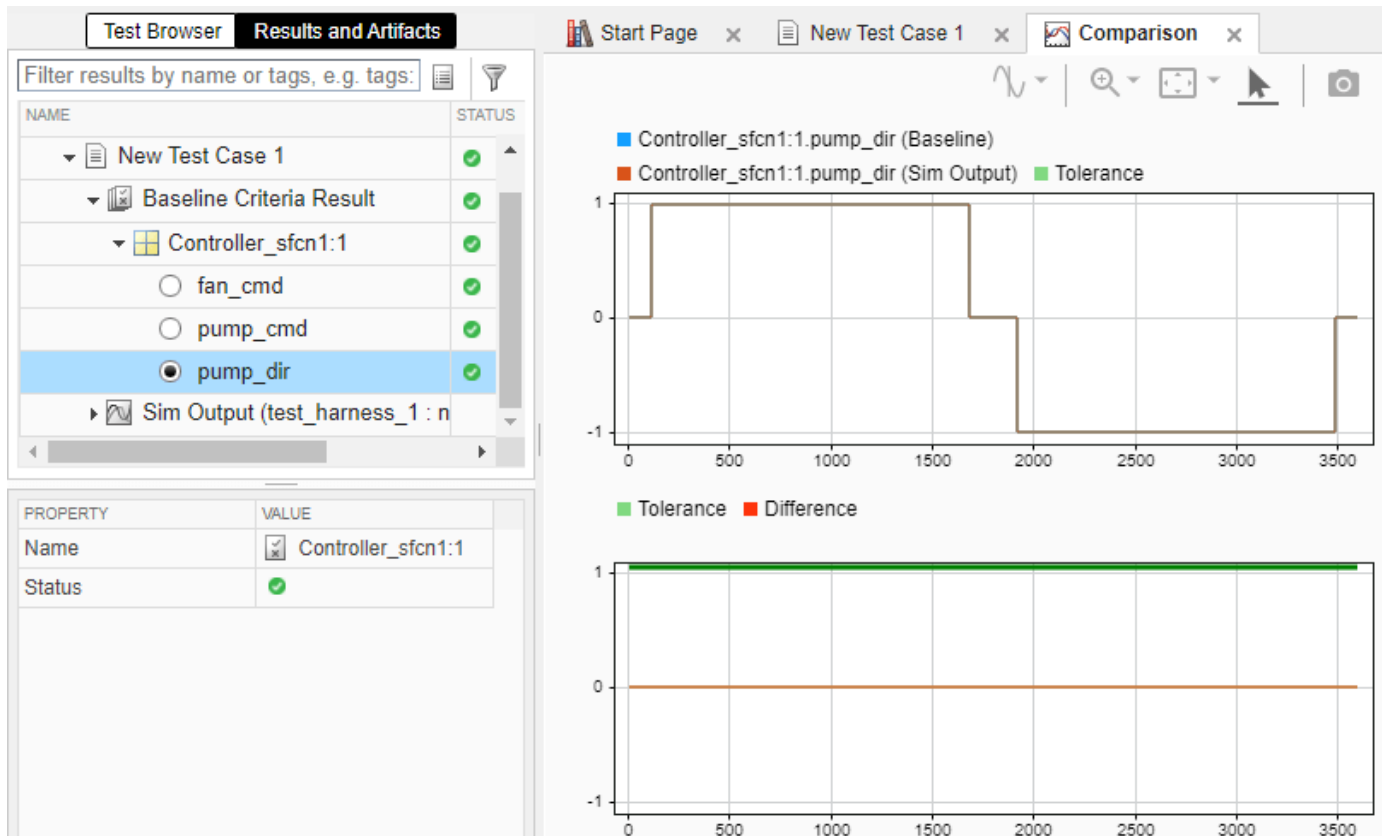
Include baseline data in test result

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 Test Case and View Results

1. Run the test case. The test results appear in the **Results and Artifacts** pane.
2. Expand the results to view the baseline criteria result. The baseline test passes because the simulation output is identical to the baseline data.



See Also

`sltest.CodeImporter`

Related Examples

- “Conduct Unit Testing on Imported Custom Code by Using the Wizard” on page 9-11
- “Import Custom Code for Unit Testing Using API Commands” on page 9-5

Test Manager Test Cases

- “Manage Test File Dependencies” on page 6-3
- “Compare Model Output to Baseline Data” on page 6-7
- “Creating Baseline Tests” on page 6-10
- “Batch Equivalence Testing of Multiple Components” on page 6-13
- “Test a Simulation for Run-Time Errors” on page 6-16
- “Automatically Create a Set of Test Cases” on page 6-19
- “Generate Tests and Test Harnesses for a Model or Components” on page 6-24
- “Override Model Parameters in a Test Case” on page 6-31
- “Test Two Simulations for Equivalence” on page 6-35
- “Create and Run a Back-to-Back Test” on page 6-41
- “Perform Back-to-Back (MIL/SIL) Equivalence Test for an Atomic Subsystem” on page 6-47
- “Testing AUTOSAR Compositions” on page 6-55
- “Automate Testing for Highway Lane Following” on page 6-60
- “Synchronize Tests” on page 6-71
- “Use External Excel or MAT-File Data in Test Cases” on page 6-72
- “Create Data Files for Test Case Input” on page 6-80
- “Capture Simulation Data in a Test Case” on page 6-85
- “Run Tests in Multiple Releases of MATLAB” on page 6-94
- “Examine Test Failures and Modify Baselines” on page 6-102
- “Create and Run Test Cases with Scripts” on page 6-107
- “Test Models Using MATLAB-Based Simulink Tests” on page 6-111
- “Using MATLAB-Based Simulink Tests in the Test Manager” on page 6-116
- “Collect Coverage Using MATLAB-Based Simulink Tests” on page 6-120
- “Test Iterations” on page 6-125
- “Capture Baseline Data from Iterations” on page 6-133
- “Collect Coverage in Tests” on page 6-135
- “Test Coverage for Requirements-Based Testing” on page 6-142
- “Increase Test Coverage for a Model” on page 6-147
- “Run Tests Using Parallel Execution” on page 6-151
- “Set Signal Tolerances” on page 6-153
- “Specify Test Properties in the Test Manager” on page 6-158
- “Preferences” on page 6-173
- “Increase Coverage by Generating Test Inputs” on page 6-174
- “Process Test Results with Custom Scripts” on page 6-179
- “Assess the Damping Ratio of a Flutter Suppression System” on page 6-185

- “Create, Store, and Open MATLAB Figures” on page 6-188
- “Test Models Using MATLAB Unit Test” on page 6-191
- “Output Results for Continuous Integration Systems” on page 6-194
- “Parametric Sweep for a Simscape Thermal Model” on page 6-198
- “Projector Controller Testing Using verify and Real-Time Tests” on page 6-204
- “Test Execution Order” on page 6-209
- “Filter Test Execution, Results, and Coverage” on page 6-213

Manage Test File Dependencies

In this section...

“Package a Test File Using Projects” on page 6-3

“Find Test File Dependencies and Impact” on page 6-4

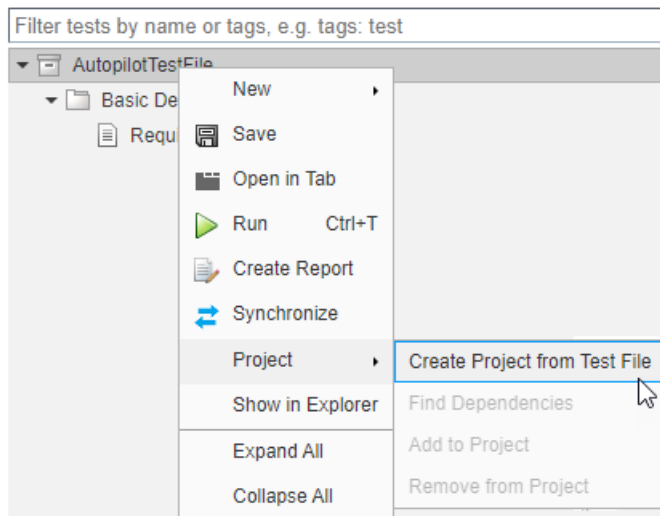
“Share a Test File with Dependencies” on page 6-6

You can help track and manage your test file dependencies by creating a 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 project also helps you share tests with other users.

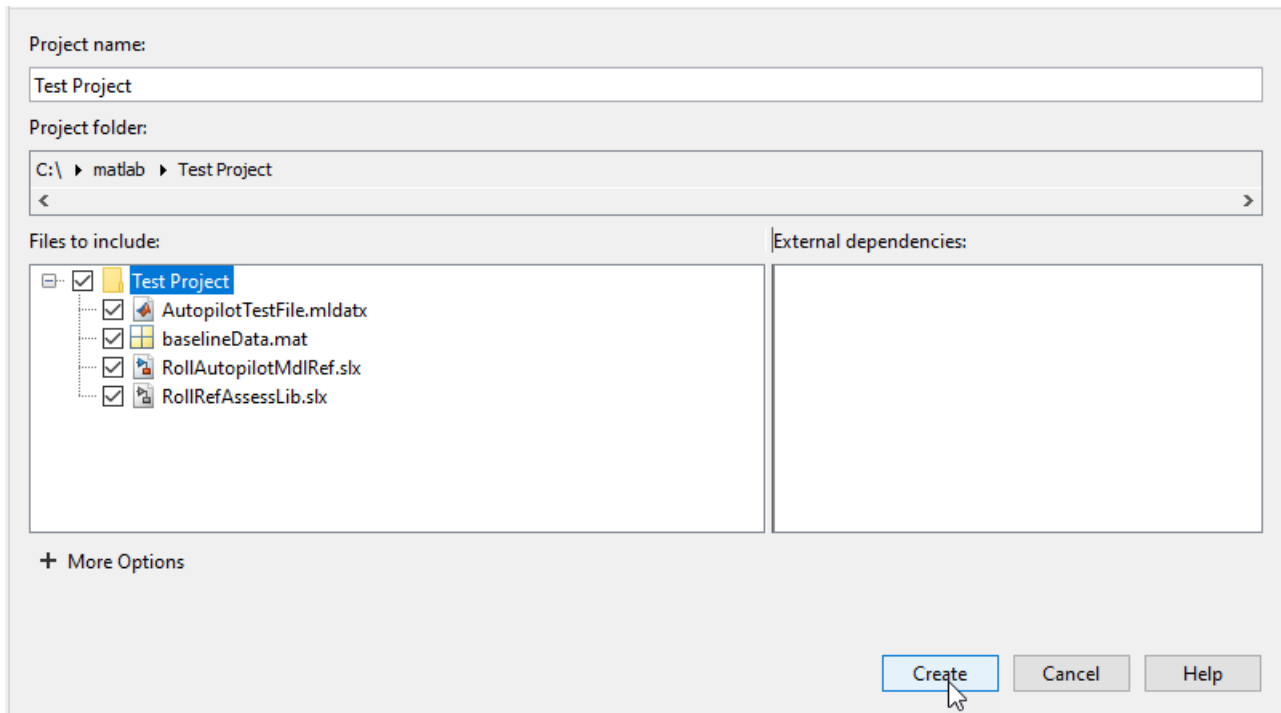
Package a Test File Using Projects

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

Project opens and identifies the file dependencies of the test file. In this example, the test file contains a baseline test case that uses a baseline data file.



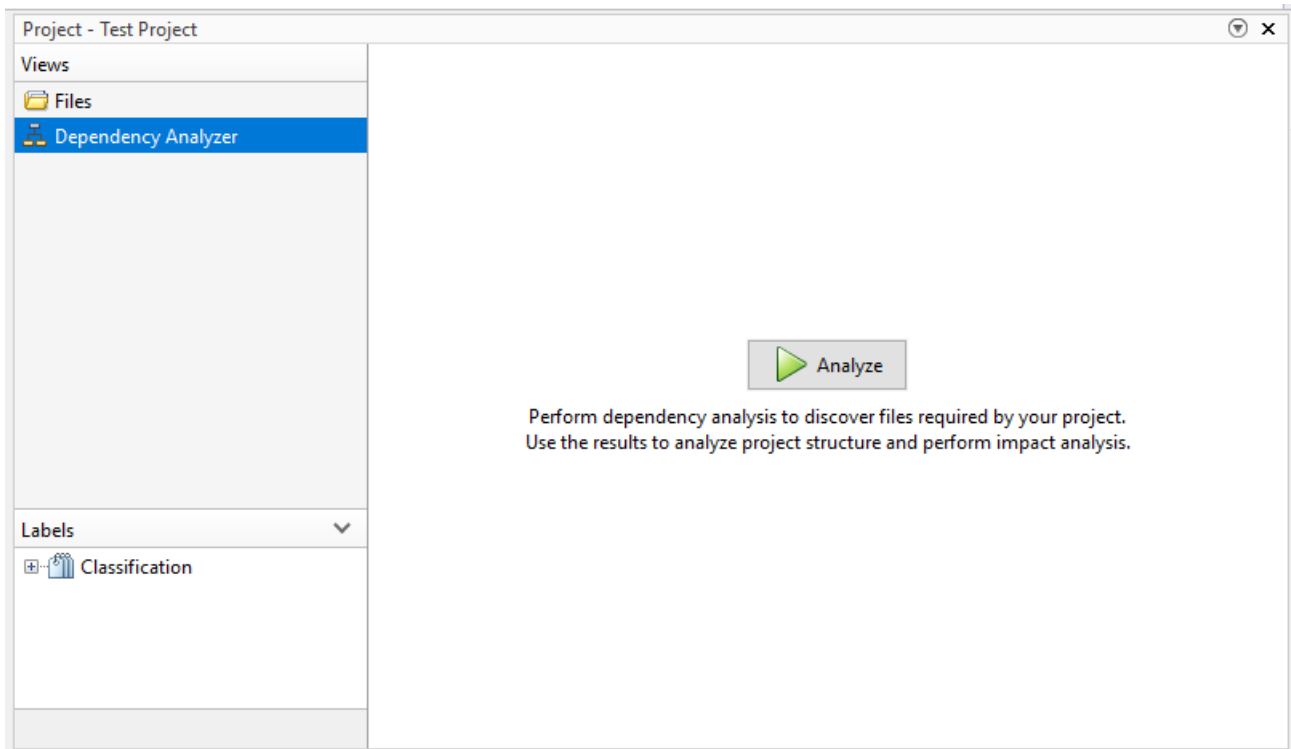
- 3 Specify project name, and verify the list of selected file dependencies.
- 4 Click **Create**.



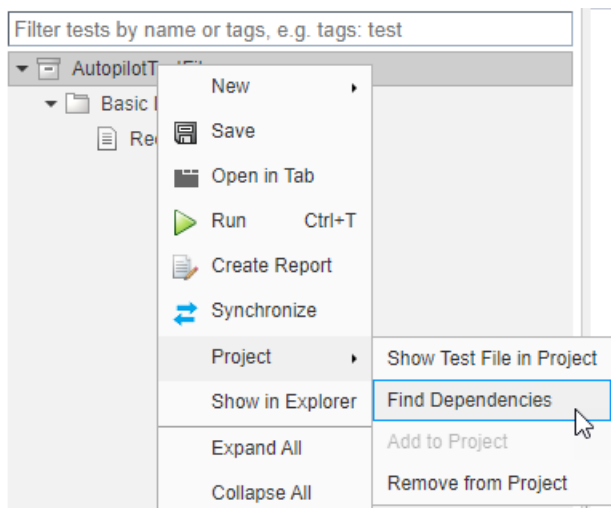
Find Test File Dependencies and Impact

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

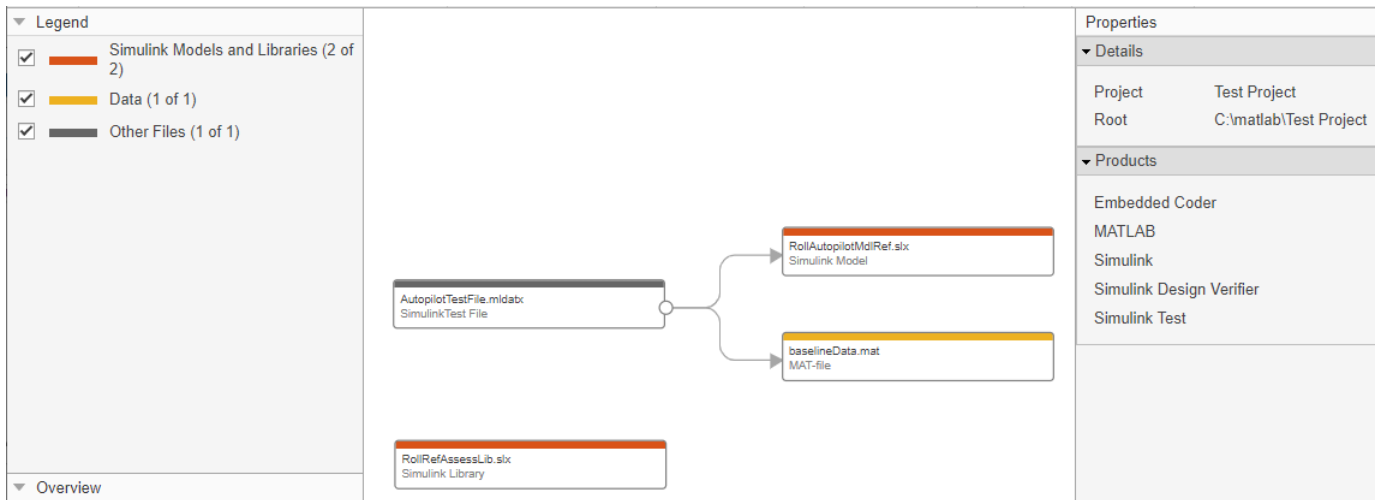
- 1 From the project, click **Analyze**



or from the Test Manager, right-click the test file. Select **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 Dependency Analyzer toolstrip, in the **Impact Analysis** section, click **Impacted**.

If you want to run a test file again, double-click the test file in the graph to open the Test Manager. In the Test Manager, click **Run**.

Share a Test File with Dependencies

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

See Also

Related Examples

- “What Are Projects?”


Compare Model Output to Baseline Data

To test the simulation output of a model against a defined baseline, use a baseline test case. This example uses 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 model using `openExample('sldemo_absbrake')`.
- 2 To open the Test Manager from the model, on the **Apps** tab, under Model Verification, Validation, and Test, click **Simulink Test**. Then, on the **Tests** tab, click **Simulink 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>	baseline1	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 “Set Signal Tolerances” on page 6-153.

Run the Test Case and View Results

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

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

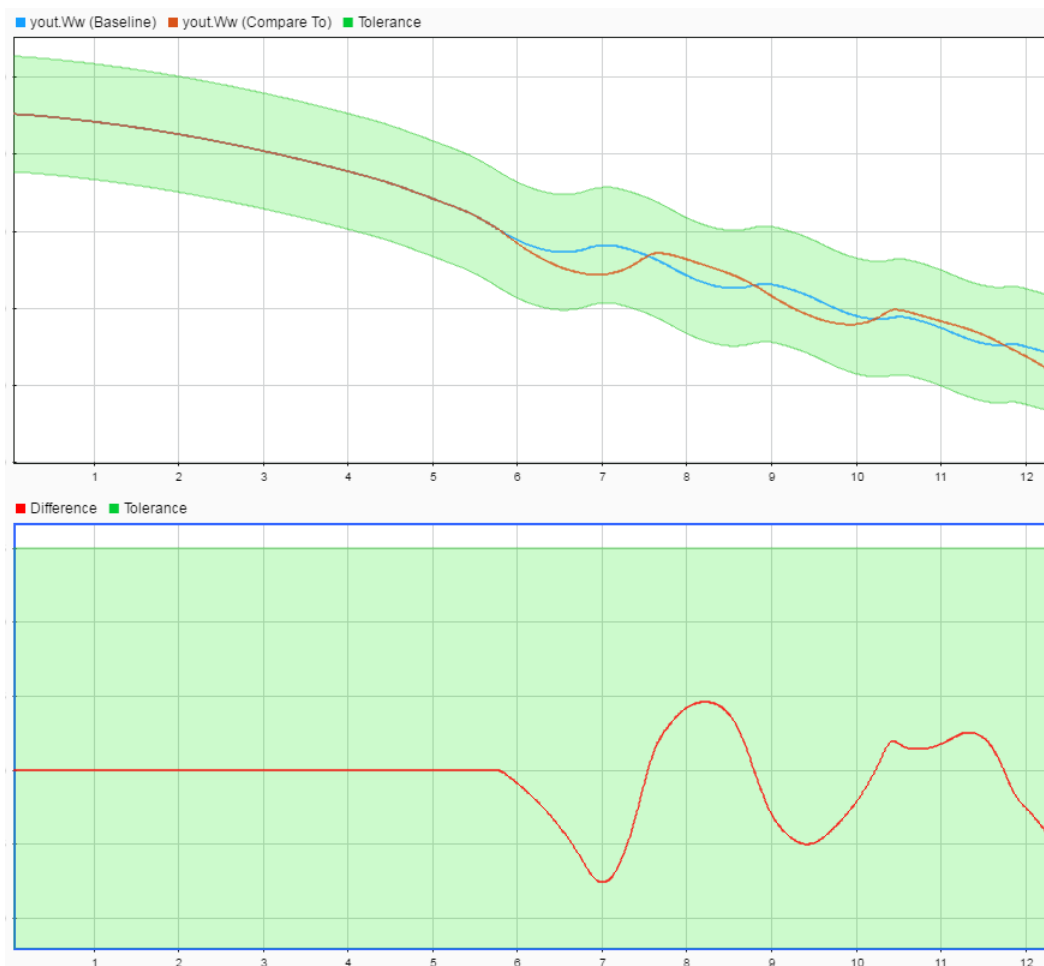
- 4 Expand the results until you see the baseline criteria result. Right-click the result and select **Expand All Under**.

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

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

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

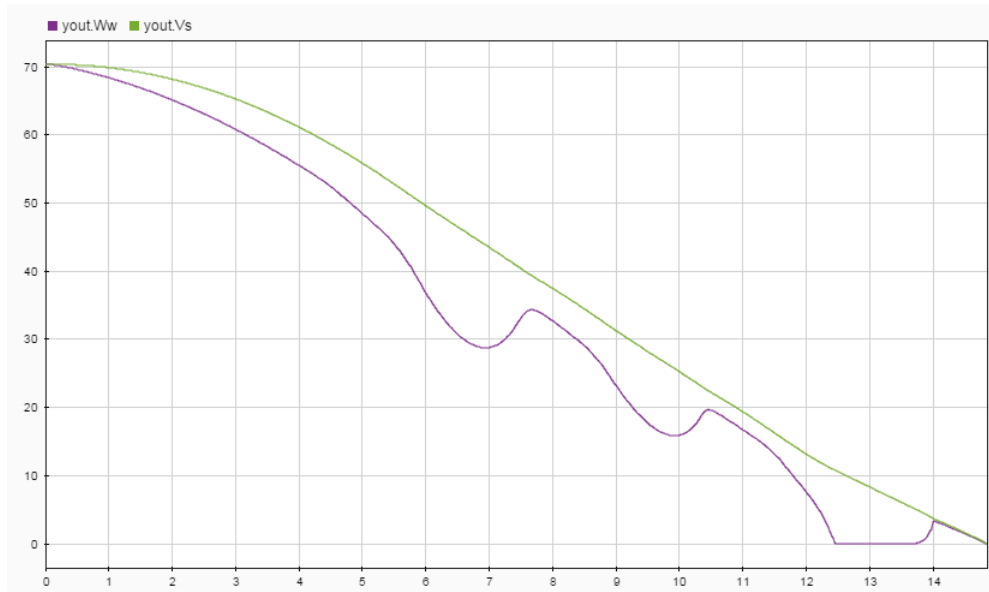
The **Comparison** tab opens and shows the criteria comparisons for the `yout.Ww` signal and the tolerance.



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

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” on page 7-16.

See Also

Test Manager

Related Examples

- “Set Signal Tolerances” on page 6-153
- “Capture Baseline Criteria” on page 6-167
- “Run Tests in Multiple Releases of MATLAB” on page 6-94

Creating Baseline Tests

Verify simulation result against a baseline dataset created from a model.

This example shows you how to create baselines tests for a model. The example uses the model `sltestBaselineBasicExample` to generate a baseline dataset of expected results by simulating the model. The baseline test case checks that the simulation results produce the same output as the baseline dataset, which determines the pass/fail criteria of the test case.

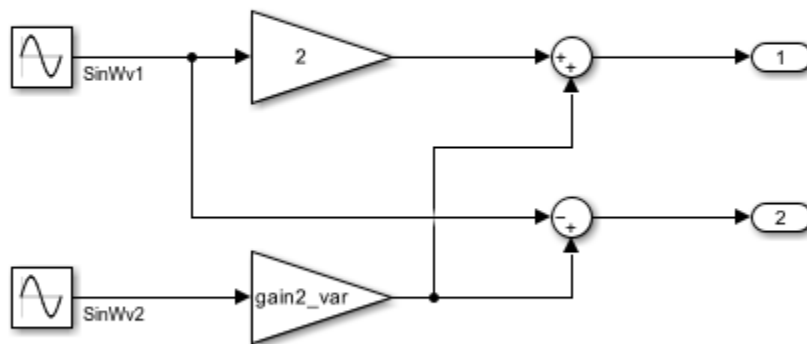
Open the Model and Test Manager

1. Open the model.

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

Creating Baseline Tests

This model is used to show how baseline tests can be created and executed in the Test Manager. To see the demo, execute `showdemo sltestTestManagerBaselineDemo` in MATLAB(R).



Copyright 2015 The MathWorks, Inc.

2. From the model, in the **Apps** tab, click **Simulink Test** from the Model Verification, Validation, and Test section. Then click **Test Manager** in the **Tests** tab.

3. Create a new test file using the Test Manager toolstrip.

4. Name the test file, and save it in a writable folder.

Capture Baseline

1. Under **System Under Test**, for **Model**, enter `sltestBaselineBasicExample`. Capture a baseline for the test case by expanding the **Baseline Criteria** section and clicking **Capture**. Save the file `BaselineData` in a writable folder.

The test case runs, and baseline data is captured for the root outports.

▼ BASELINE CRITERIA*

Include baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL	
▼ <input checked="" type="checkbox"/> BaselineData.mat	0	0.00%	0	0	+
<input checked="" type="checkbox"/> Out1:1	0	0.00%	0	0	
<input checked="" type="checkbox"/> Out2:1	0	0.00%	0	0	

2. Click **Run** from the toolbar to execute the test.

Requirements Scenarios x Start Page x New Test Case 1 x

New Test Case 1 Enabled

[baselinecomparison](#) » [New Test Suite 1](#) » [New Test Case 1](#)





Baseline Test

Select releases for simulation:

Create Test Case from External File

- ▶ TAGS
- ▶ DESCRIPTION
- ▶ REQUIREMENTS

▼ SYSTEM UNDER TEST* ?

Model:    

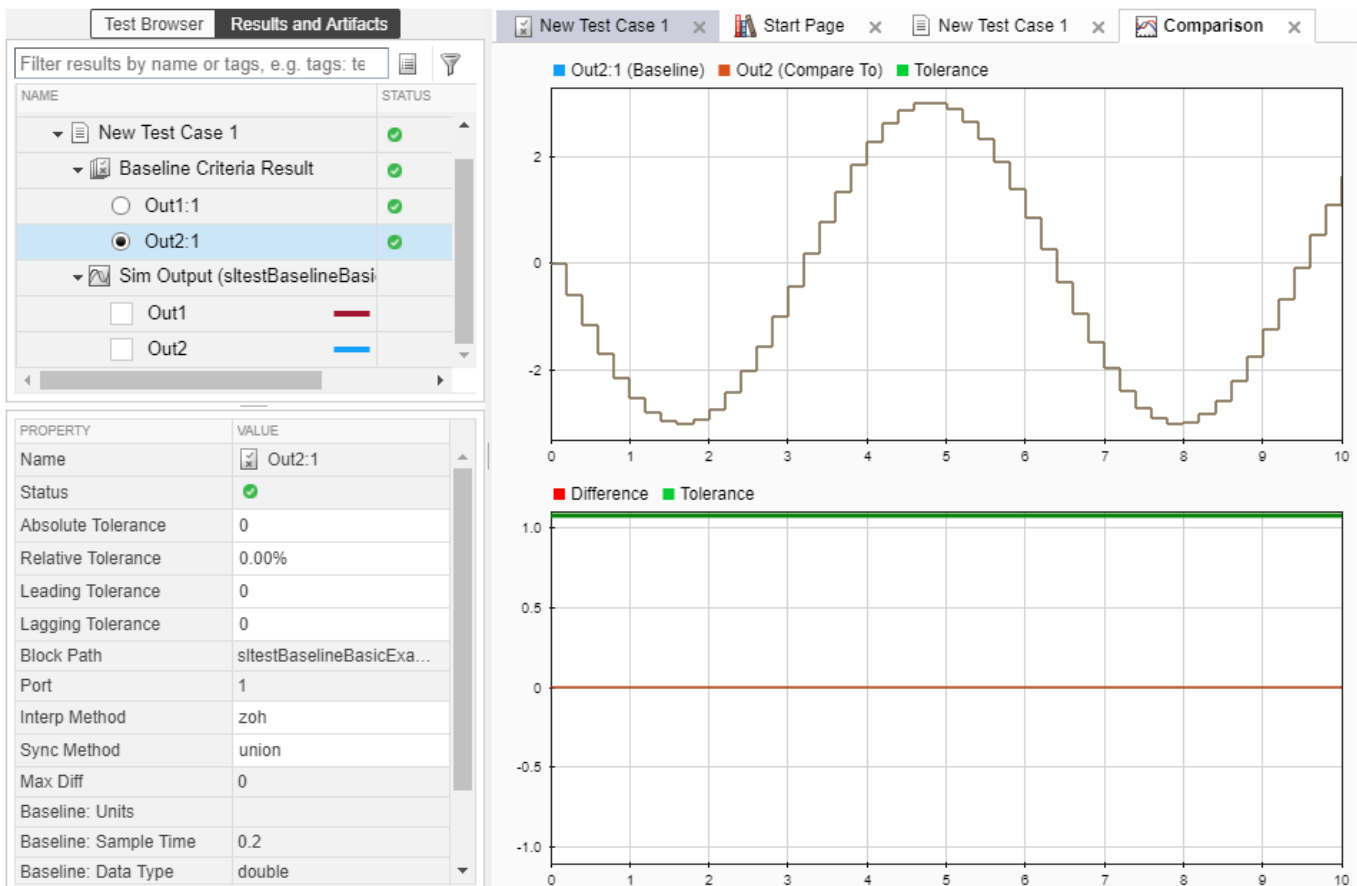
- ▶ TEST HARNESS
- ▶ SIMULATION SETTINGS OVERRIDES

Visualize Baseline Test Results

1. After the test completes, expand all rows in the **Results and Artifacts** pane. The test case passes because the simulation results match the baseline results.

ExampleTestFile	1	✓
New Test Suite 1	1	✓
New Test Case 1		✓
Baseline Criteria Result		✓
Out1		✓
Out2		✓
Sim Output (sltestBaseline		
Out1		—
Out2		—

2. Select the option button for Out2 under **Baseline Criteria Result** to visualize the data comparison.



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


Batch Equivalence Testing of Multiple Components

This example shows how to create equivalence test cases and test harnesses for multiple components in batch mode using `sltest.testmanager.createTestForComponent`. Coverage collection and report generation are also included in the example.

The `sltestCruiseControl` model used in this example has Atomic Subsystem blocks, Virtual Subsystems blocks & Model Reference Blocks .

Note that this example is configured only for Windows machines.

Open the Model

```
topModel = "sltestCruiseControl";
load_system(topModel);
```

Generate the Code

The equivalence tests in this example compare normal mode code and generated code. Embedded Coder is required to generate the code for the model. The model has been configured with appropriate C code generation and coder mapping settings for Windows 64-bit systems. You can change these settings, if desired, before generating the code.

```
slbuild(topModel);

### Starting serial model reference code generation build.
### Checking status of model reference code generation target for model 'sltestCruiseControlMode'
### Model reference code generation target (sltestCruiseControlMode.c) for model sltestCruiseControlMode is up to date.
### Starting build procedure for: sltestCruiseControlMode
### Generating code and artifacts to 'Model specific' folder structure
### Code for the model reference code generation target for model sltestCruiseControlMode is up to date.
### Saving binary information cache.
### Skipping makefile generation and compilation because C:\Users\dschwartz\OneDrive - MathWorks\Documents\MATLAB\Examples\sltestCruiseControlMode\sltestCruiseControlMode.c is up to date.
### Successful completion of code generation for: sltestCruiseControlMode
### Model reference code generation target for sltestCruiseControlMode is up to date.
### Checking status of model reference code generation target for model 'sltestDriverSwRequest'
### Model reference code generation target (sltestDriverSwRequest.c) for model sltestDriverSwRequest is up to date.
### Starting build procedure for: sltestDriverSwRequest
### Generating code and artifacts to 'Model specific' folder structure
### Code for the model reference code generation target for model sltestDriverSwRequest is up to date.
### Saving binary information cache.
### Skipping makefile generation and compilation because C:\Users\dschwartz\OneDrive - MathWorks\Documents\MATLAB\Examples\sltestDriverSwRequest\sltestDriverSwRequest.c is up to date.
### Successful completion of code generation for: sltestDriverSwRequest
### Model reference code generation target for sltestDriverSwRequest is up to date.
### Simulink cache artifacts for 'sltestCruiseControlMode' were created in 'C:\Users\dschwartz\OneDrive - MathWorks\Documents\MATLAB\Examples\sltestCruiseControlMode\sltestCruiseControlMode.c'
### Simulink cache artifacts for 'sltestDriverSwRequest' were created in 'C:\Users\dschwartz\OneDrive - MathWorks\Documents\MATLAB\Examples\sltestDriverSwRequest\sltestDriverSwRequest.c'
### Starting build procedure for: sltestCruiseControl
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\Users\dschwartz\OneDrive - MathWorks\Documents\MATLAB\Examples\sltestCruiseControl\sltestCruiseControl.c
### Generated code for 'sltestCruiseControl' is up to date because no structural, parameter or coder mapping changes were detected.
### Saving binary information cache.
### Skipping makefile generation and compilation because C:\Users\dschwartz\OneDrive - MathWorks\Documents\MATLAB\Examples\sltestCruiseControl\sltestCruiseControl.c is up to date.
### Successful completion of code generation for: sltestCruiseControl
```

Build Summary

```
0 of 3 models built (3 models already up to date)
Build duration: 0h 0m 11.953s
```

Specify the Components to Test

The example tests all of the atomic subsystems that have nonreusable function packaging in the generated code.

```
componentsToTest = find_system(topModel,...
    "BlockType","SubSystem",...
    "TreatAsAtomicUnit","on",...
    "RTWSystemCode","Nonreusable function");
```

To improve traceability of testing artifacts, such as test case and test harness names, customize the default names of the created harnesses. The names will use the component name instead of the owning model name.

```
sltest.harness.setHarnessCreateDefaults("Name","$Component$_Harness");
```

Create the Test Cases and Test Harnesses in Batch Mode

Use the `sltest.testmanager.createTestForComponent` API to create multiple test cases and test harnesses at the same time. Using `createTestForComponent`, you specify to create a test file and the test filename. You also specify the top model, components to test, and the test type and simulations settings. For the test inputs, you specify to use Simulink Design Verifier to automatically generate the inputs in Microsoft Excel format. If you have other existing inputs, you can add them to the test cases and test both those inputs and the generated inputs.

```
[tc, status] = ...
    sltest.testmanager.createTestForComponent(...
        "CreateTestFile",true,...
        "TestFile","myB2BTestsDemoEx.mldatx",...
        "TopModel",topModel,...
        "Component",componentsToTest,...
        "TestType","equivalence",...
        "Simulation1Mode","Normal",...
        "Simulation2Mode","Software-in-the-Loop (SIL)",...
        "SLDVTTestGeneration","on",...
        "CreateExcelFile",true);
```

Test Execution

The `createTestForComponent` API generated the test file, test harnesses, and test input signals in the current working directory. Now, save the harnesses attached to the model, and save the test file.

```
tf = sltest.testmanager.TestFile("myB2BTestsDemoEx.mldatx");
tf.saveToFile;
```

Enable coverage collection for test file and run the tests.

```
cov = getCoverageSettings(tf);
cov.RecordCoverage = true;
cov.MetricSettings = "dcmr";
tf.saveToFile;

resultSet = tf.run;
sltest.testmanager.exportResults(resultSet,"myB2BResults.mldatx");
sltestmgr;
```

Generate a report with coverage and equivalence test results.

```
sltest.testmanager.report(...
    resultSet, "myB2BResultsReport.pdf", ...
    "IncludeCoverageResult", true, ...
    "IncludeSimulationSignalPlots", true, ...
    "IncludeComparisonSignalPlots", true, ...
    "IncludeTestResults", 0, ...
    "IncludeSimulationMetadata", true);
```

Results: 2021-Dec-20 15:17:24

Result Type: Result Set
 Parent: None
 Start Time: 20-Dec-2021 15:17:30
 End Time: 20-Dec-2021 15:19:38
 Outcome: Total: 4, Passed: 4

Aggregated Coverage Results

Analyzed Model	Sim Mod e	Compl exity	Decisio n	Conditio n	MCDC	Func tion	Func tion call	Execu tion
sltestCruiseControl/TargetSpeedThrottle	Normal	21	100%	100%	100%	--	--	100%
sltestCruiseControl/TargetSpeedThrottle/enabled	Normal	2	100%	--	--	--	--	--
sltestCruiseControl/TargetSpeedThrottle	SIL	29	98%	100%	100%	100%	100%	98%
sltestCruiseControl/TargetSpeedThrottle/enabled	SIL	29	100%	--	--	100%	--	100%

[Back to Report Summary](#)

myB2BTestsDemoEx

Test Result Information

Result Type: Test File Result
 Parent: [Results: 2021-Dec-20 15:17:24](#)
 Start Time: 20-Dec-2021 15:17:30
 End Time: 20-Dec-2021 15:19:38
 Outcome: Total: 4, Passed: 4

Clean Up

```
bdclose({topModel, 'sltestDriverSwRequest', 'sltestCruiseControlMode'})
sltest.testmanager.clear
sltest.testmanager.clearResults
sltest.testmanager.close
```

See Also

```
sltest.testmanager.createTestForComponent
```

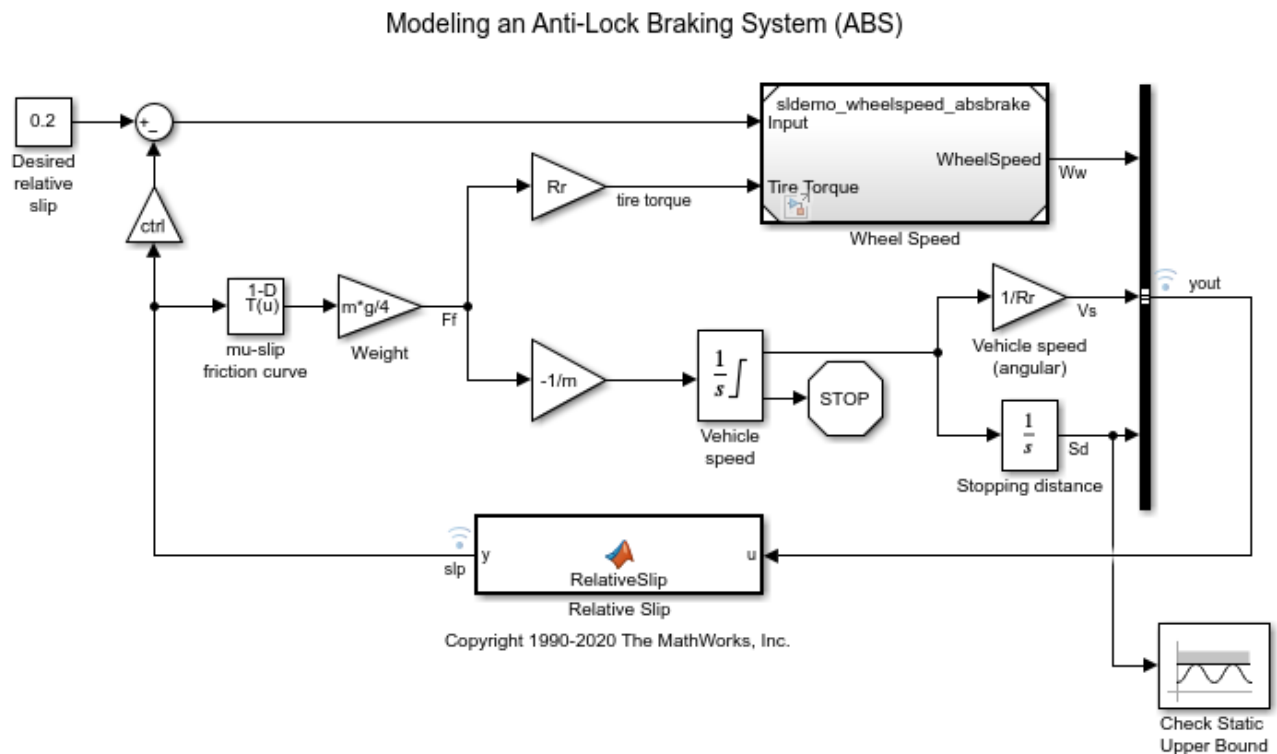
Test a Simulation for Run-Time Errors

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 To open the model, type `openExample('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, on the **Apps** tab, under Model Verification, Validation, and Test, click **Simulink Test**. Then, on the **Tests** tab, click **Simulink 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 the Error

Click on the Upper Bound Test to view the run-time error.

The screenshot shows the Test Manager interface with the 'Results and Artifacts' tab selected. On the left, a tree view shows the test results for '2021-Jan-15 16:08:15'. The 'Upper Bound Test' is highlighted in blue and shows a failure status (1 error). Below it, 'Verify Statements' is also failed, and 'Check Static Upper Bound' is unchecked and failed. Under 'Sim Output (sldemo_absbrake)', several sub-items like 'yout.Ww', 'yout.Vs', 'yout.Sd', and 'slp' are listed with progress bars.

The main panel shows the 'SUMMARY' for the 'Upper Bound Test'. It includes a table with the following data:

Name	Upper Bound Test
Outcome	1 ✖
Start Time	01/15/2021 16:09:00
End Time	01/15/2021 16:10:41
Type	Simulation Test
Test File Location	C:\Users\ldscharw\demo_br_ex.mldatx
Test Case Definition	
Rerun Test Case	
Tags	
Cause of Failure	Errors running test case
▶ Simulation Metadata	

Below the summary, the 'ERRORS' section contains two messages:

- An error occurred ('Simulink:blocks:AssertionAssert') when calling 'sim':
- Assertion detected in 'sldemo_absbrake/Check Static Upper Bound' at time 12.1928

See Also

More About

- “Run Tests in Multiple Releases of MATLAB” on page 6-94

Automatically Create a Set of Test Cases

In this section...

“Creating Test Cases from Model Elements” on page 6-19

“Generating Test Cases from a Model” on page 6-19

Creating Test Cases from Model Elements

You can automatically create a set of test cases and iterations that correspond to blocks and test harnesses in your model. You specify whether the test cases are baseline, equivalence, or simulation test cases. To automatically create test cases, your model must contain either or both of the following:

- One Signal Editor block at the top level of the model. If the block has only one scenario, a test case is created. If the block has more than one scenario, an iteration is created for each scenario.
- Test harnesses. If a test harness contains one (and only one) Signal Editor block at the top level, a test case is created for the scenario in the block. If the block has more than one scenario, an iteration is created for each scenario.

To automatically create test cases or iterations for your model:

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

Generating Test Cases from a Model

Generate test cases based on model hierarchy.

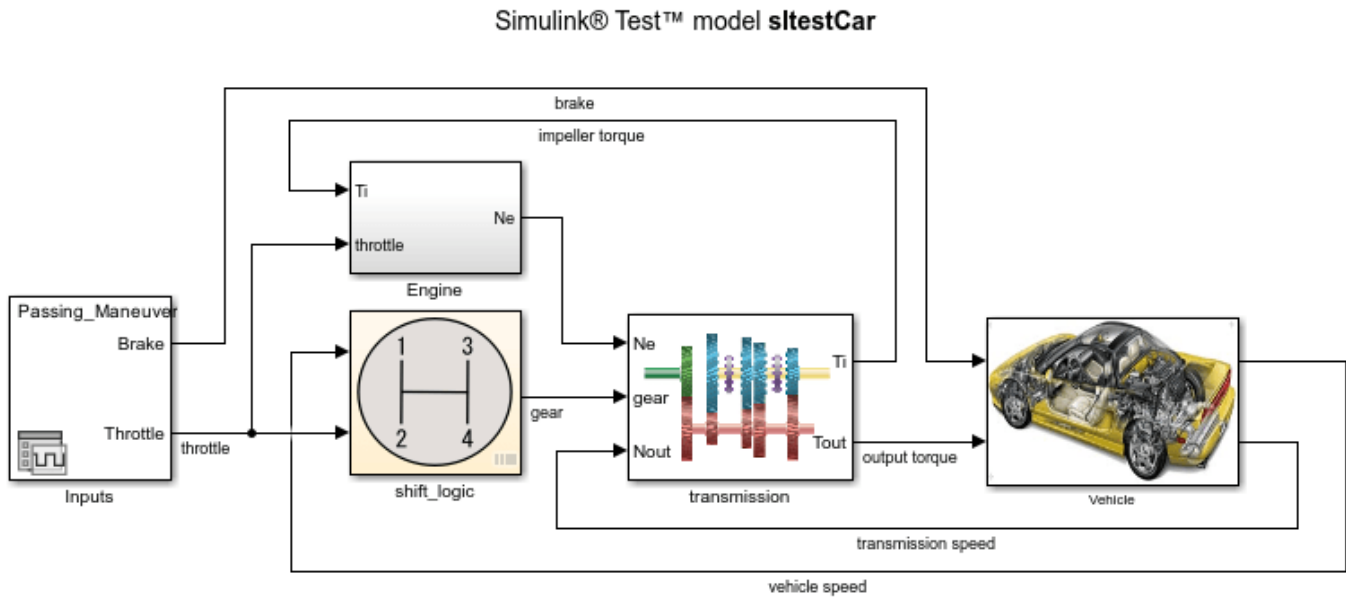
This example shows how to generate test cases based on the components in your model. This example uses the model `sltestCar`, which has been pre-configured with the following:

- Signal Editor block at the top level of the model
- Test harnesses at the top level of the model
- Signal Editor block at the top level of the test harness

Open the Model and Test Manager

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

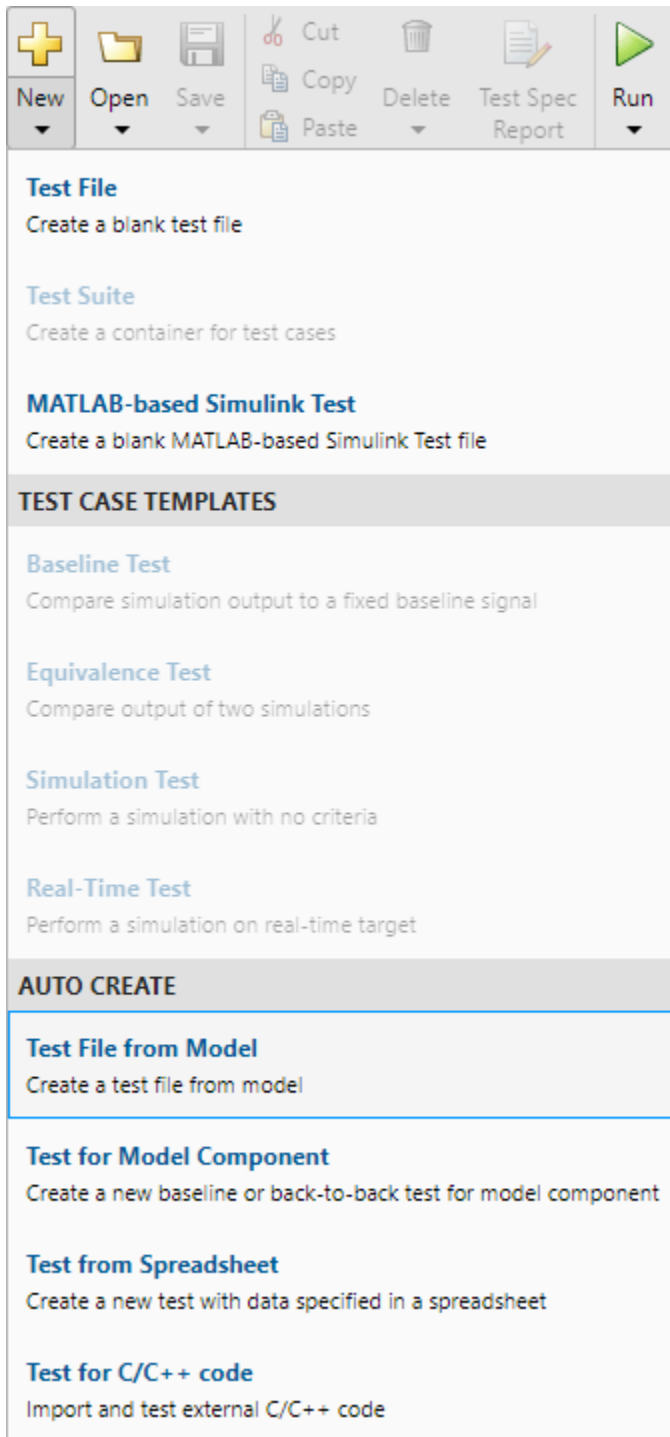
```
Model = 'sltestCar';  
open_system(Model);
```



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

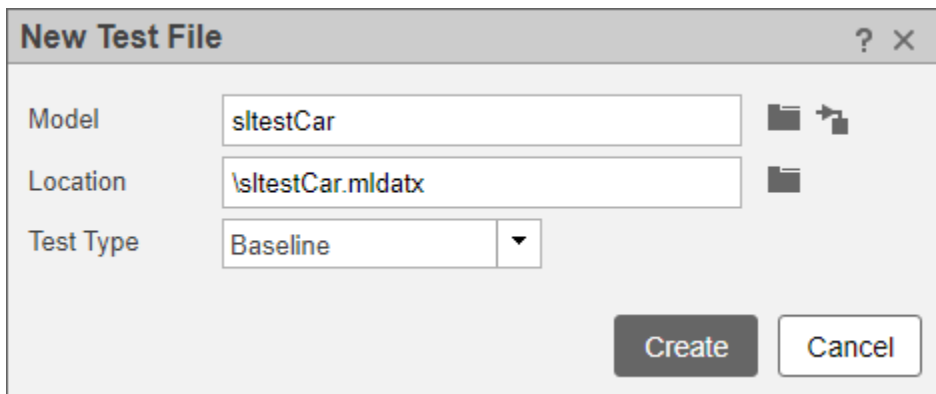
Generate Test Cases From the Model

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



- 1 In the **New Test File** dialog box, click the **Use current model** button to specify `sltestCar` 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.

- Click **Create**.



The sltestCar/Inputs test case uses table iterations.

NAME	DESCRIPTION	SIGNAL EDITOR SCENARIO
<input checked="" type="checkbox"/> Coasting	None	Coasting
<input checked="" type="checkbox"/> Gradual_Acceleration	None	Gradual_Acceleration
<input checked="" type="checkbox"/> Hard_braking	None	Hard_braking
<input checked="" type="checkbox"/> Passing_Maneuver	None	Passing_Maneuver

Before you run the test, you must specify the baseline criteria for each generated test case.

```
close_system(Model, 0);
clear Model;
```

See Also

More About

- “Synchronize Tests” on page 6-71
- “Specify Test Properties in the Test Manager” on page 6-158
- “Compare Model Output to Baseline Data” on page 6-7

- “Test a Simulation for Run-Time Errors” on page 6-16
- “Test Two Simulations for Equivalence” on page 6-35
- “Import Test Cases for Equivalence Testing” on page 5-19
- “Generate Tests and Test Harnesses for a Model or Components” on page 6-24

Generate Tests and Test Harnesses for a Model or Components

In the Test Manager, the Create Test for Component wizard creates a test harness and test case for a model or component in the model. You can also use the wizard to create test harnesses and test cases for multiple components in the model. Components for which you can create test harnesses include subsystems, Stateflow charts, and Model blocks. For a full list of components supported by test harnesses, see “Test Harness and Model Relationship” on page 2-2. If you select a model component that is not compatible with the wizard, that component does not appear in the wizard.

In the wizard, you specify:

- The model to test.
- The component or components to test, if you are not testing a whole model.
- The test inputs.
- The type of test to run on the component.
- Whether to save test data in a MAT-file or Excel®. For more information on using Excel files in the Test Manager, see “Format Test Case Data in Excel” on page 6-83.

For an example that uses the wizard, see “Create and Run a Back-to-Back Test” on page 6-41.

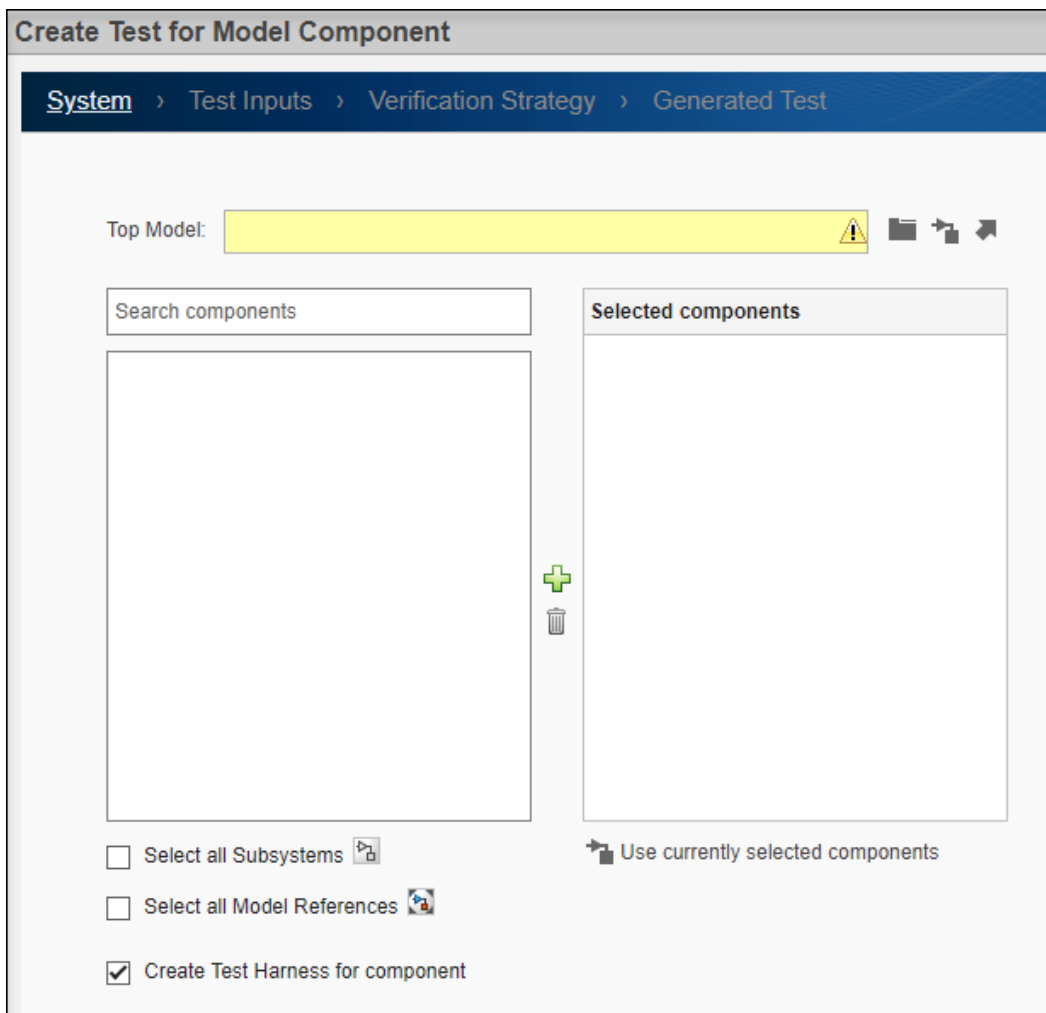
Open the Create Test for Component Wizard


Before you run the wizard, check your model.

- If you are testing the code for an atomic subsystem by using an equivalence test that uses software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode, verify that the subsystem already has generated code.
- If you are testing the code generated for a reusable library subsystem, before opening the Create Test for Component wizard, verify that the subsystem has defined function interfaces and that the library already has generated code. For information on reusable library subsystems, function interfaces, and generated code, see “Library-Based Code Generation for Reusable Library Subsystems” (Embedded Coder). You must have an Embedded Coder license to verify generated code.
- If you are testing one or more components, select the components in the model before opening the wizard. When the wizard opens, click **Use currently selected components** to add the selected components and fill in the **Top Model** field.

To open the Create Test for Component wizard, in the Test Manager select **New > Test for Model Component**.

Select Model or Component to Test





If you have not selected any components in the model, on the first page of the wizard, click the **Use current model** button  to fill in the **Top Model** field.

Then, if you are testing:

- A whole model, do not select add components to the **Selected components** pane.



To test the whole model without creating a test harness, clear **Create Test Harness for component**. If you are testing a specific component or components in the model, the wizard creates the test harnesses automatically, and the **Create Test Harness for component** option does not display.

- One or more components and you did not select the components in the model before opening the wizard, select the component or press **Ctrl** and click the components to test. Then click the plus button  to add the components to the **Selected components** pane. To remove one or more components, press **Ctrl** and click to select them in the **Selected components** pane, and click the remove button .

- Component in a Model block, you do not need to specify the Model block as the top model. Use the name of the model that contains the Model block as the **Top Model**.
- Reusable library subsystem that has a function interface, a **Function Interface Settings** option displays. Select the function interface for which you want to create a test. The **Function Interface Settings** option is not displayed if you select multiple components.



Function Interface Settings:

Select a function interface:  

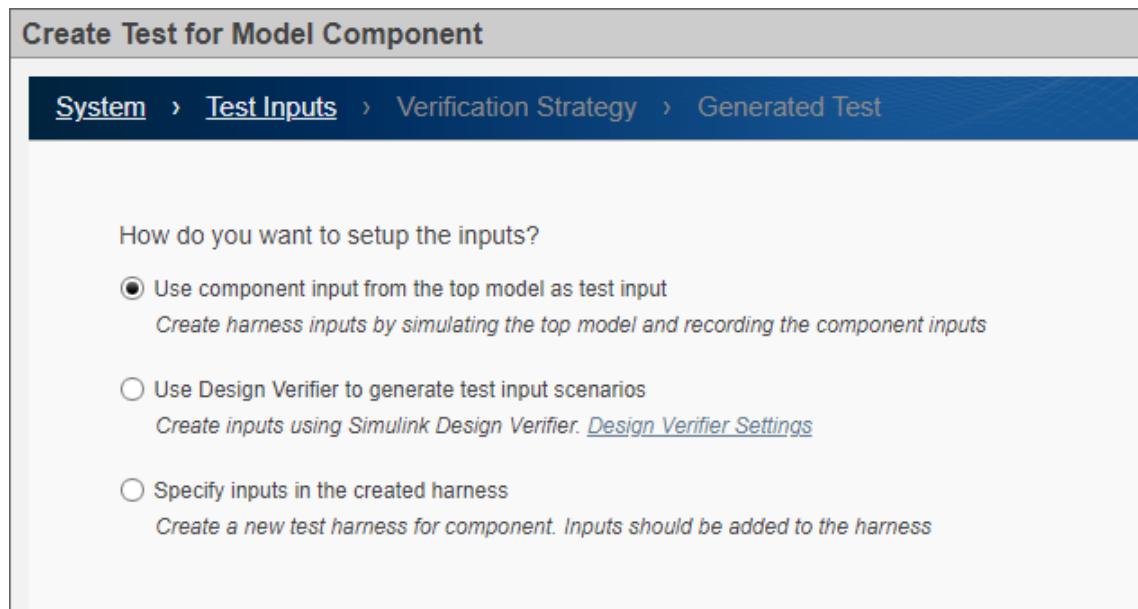
Reusable libraries contain components and subsystems that can be shared with multiple models. You can share the code generated by the subsystems if those subsystems are at the top level of the reusable library and if they have function interfaces. Function interfaces specify the subsystem input and output block parameter settings.

Note For an export-function model, the test harness creates a Test Sequence block automatically.

If you have selected one or more components in the model, on the first page of the wizard, click **Use currently selected components** to fill in the **Top Model** field and add the selected components automatically.

Click **Next** to go to the next page of the wizard.

Set Up Test Inputs



Create Test for Model Component

[System](#) > [Test Inputs](#) > [Verification Strategy](#) > [Generated Test](#)

How do you want to setup the inputs?

- Use component input from the top model as test input
Create harness inputs by simulating the top model and recording the component inputs
- Use Design Verifier to generate test input scenarios
Create inputs using Simulink Design Verifier. [Design Verifier Settings](#)
- Specify inputs in the created harness
Create a new test harness for component. Inputs should be added to the harness

On the **Test Inputs** page, select how to obtain the test inputs.

- **Use component input from the top model as test input** — Simulate the model and record the inputs to the component. Then, use those inputs as the inputs to the created test harness. Use this option for debugging.

Note If you are testing a subsystem that has function calls, you cannot obtain inputs by simulating the model because function calls cannot be logged. Use either of the other two options to obtain the inputs.

- **Use Design Verifier to generate test input scenarios** — Create test harness inputs to meet test coverage requirements using Simulink Design Verifier. This option appears only if Simulink Design Verifier is installed.

Simulate top model and use the recorded component inputs in the analysis — When coverage in test cases generated from Design Verifier is lower than expected, select this option to include top model simulation for the Design Verifier analysis.

- **Specify inputs in the created harness** — After the wizard creates the harness, open the harness in the Test Manager and manually specify the harness inputs. This option does not appear if you chose not to create a test harness.

Test Method

Create Test for Model Component

System > Test Inputs > Verification Strategy > Generated Test

How do you want to test the component?

Use component under test output as baseline
Simulate the top model and record the outputs of the component to be used as baseline

Perform back-to-back testing
Set up a test to compare the component under test outputs in different simulation modes

Select simulation modes:

Simulation1:

Simulation2:

Define the verification logic in the created harness
No verification logic will be automatically added to the test

On the **Verification Strategy** page, select how to test the component.

- **Use component under test output as baseline** — Simulate the model and record the outputs from the components, which are used as the baseline.
- **Perform back-to-back testing** — Compare the results of running the component in two different simulation modes. For each simulation, select the mode from the drop-down menu. To conduct SIL testing on an atomic subsystem or a reusable library subsystem, the subsystem or library that contains the subsystem must already have generated code.

If you selected **Use Design Verifier to generate test input scenarios** on the **Test Inputs** tab, and you select **Simulation2** to Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL), the wizard displays the **Set Model Coverage Objective as Enhanced MCDC** option. Enhanced MCDC extends MCDC coverage by generating test cases that avoid masking effects from downstream blocks. See “Enhanced MCDC Coverage in Simulink Design Verifier” (Simulink

Design Verifier) and “Create Back-to-Back Tests Using Enhanced MCDC” (Simulink Design Verifier) .

- **Define the verification logic in the created harness** — After the wizard creates the harness, open the harness. Manually specify the verification logic using a Test Sequence or Test Assessments block in the generated harness. Alternatively, use logical and temporal assessments or custom criteria in the generated test case. This option does not appear if you are testing a top-level model and choose not to create a test harness.

Save Test Data

Create Test for Model Component

System > Test Inputs > Verification Strategy > Generated Test

How do you want to save the test data?

Select test harness input source: Inports

Specify the file format: EXCEL

Specify location to save test data:

Where do you want to save the generated test(s)?

Test File Location:

On the **Generated Test** page, select the format in which to save the test data and specify the filename for the generated tests.

- **Select test harness input source** — Select how the inputs generated by the Design Verifier are applied to the test harness. This option appears only if you select **Use Design Verifier to generate test input scenarios** on the Test Inputs tab.
 - **Inports** — Create a test harness with Inport blocks as the source.
 - **Signal Editor** — Create a test harness with the Signal Editor as the source that contains the input scenarios generated by the Design Verifier.
- **Specify the file format** — Specify the type of file in which to save data. This option appears only if you select **Inports** as the input source.
 - **Excel** — Saves the test inputs, outputs, and parameters to one sheet in an Excel spreadsheet file. For tests with multiple iterations, each iteration is in a separate sheet. For more information on using Excel files in the Test Manager, see “Format Test Case Data in Excel” on page 6-83.

- **MAT** — Saves inputs and outputs in separate MAT files. For tests that use Simulink Design Verifier, the wizard saves the inputs and parameters in one file and the outputs in a baseline file.
- **Specify location to save test data** — Specify the full path of the file. Alternatively, you can use the default filename and location, which saves `sltest_<model name>` in the current working folder. This option is available only for Excel format files. MAT files are saved to the default location specified in the model configuration settings.
- **Test File Location** — Specify the full path where you want to save the generated test files. Alternatively, you can use the default filename, which is `sltest_<model name>_tests`. The file is saved in the current working folder. This field appears only if you did not have a test file open in the Test Manager before you opened the wizard.


If you had a test file open in the Test Manager before you opened the wizard, these options are displayed instead of **Test File Location**:

- **Add tests to the currently selected test file** — The generated tests are added to the test file that was selected in the **Test Browser** panel of the Test Manager when you opened the wizard.
- **Create a new test file containing the test(s)** — A new test file is created for the tests. It appears in the **Test Browser** panel of the Test Manager.

Where do you want to save the generated test(s)?

Add tests to the currently selected test file.

Create a new test file containing the test(s).

Test File Location: 

Generate the Test Harness and Test Case

Click **Done** to generate a test harness and test case. A test harness is not created if you are testing a whole model and deselected **Create a Test Harness** on the first tab of the wizard.

The Test Manager then opens with the test case in the **Test Browser** pane and, if a test harness was created, the test harness name in the **Harness** field of the **System Under Test** section. The test case is named `<model name>_Harness<#>`.

Note If the model has an existing external harness, the wizard creates an additional external test harness for the component under test. If no harness exists or if an internal harness exists, the wizard creates an internal test harness.

If you are testing the code for an atomic subsystem or model block using an equivalence test and, on the **Verification Strategy** tab, you set **Simulation2** to Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL), the wizard creates only one test harness for both the normal and SIL or PIL simulation modes. For other equivalence tests, the wizard creates two harnesses, one for each simulation mode. For the following types of subsystems and model configurations, the wizard creates two test harnesses, even if the subsystem is atomic.

- Virtual subsystems
- Function-call, For Each, If Action, S-Function, Initialize Function, Terminate Function, and Reset Function subsystems

- Stateflow charts
 - Subsystems where the `ERTFilePackagingFormat` property is set to `Compact` if the subsystem code has the `PreserveStaticInFcnDecls` set to `on`.
 - Subsystems that generate inline code, such as subsystems where the `RTWSystemCode` property is not either `Nonreusable function` or `Reusable function`).
 - Subsystems that contain referenced models, S-Function, Data Store Read, or Data Store Write blocks
 - Subsystems that have virtual buses at their interface
 - Subsystems that include LDRA or BullsEye code coverage
 - Subsystems that include signal logging
-

See Also

`sltest.testmanager.createTestForComponent`

More About

- “Create and Run a Back-to-Back Test” on page 6-41
- “Test Two Simulations for Equivalence” on page 6-35
- “Compare Model Output to Baseline Data” on page 6-7
- “Test a Simulation for Run-Time Errors” on page 6-16
- “Automatically Create a Set of Test Cases” on page 6-19

Override Model Parameters in a Test Case

Compare simulation to baseline data using a parameter override and the Test Manager.

This example shows how to override a parameter defined in a model workspace using the Test Manager, and view its effect on model output compared to a baseline.

Open the Test File

Open the Test Manager.

```
sltest.testmanager.view
```

Open the test file.

```
tf = sltest.testmanager.load('sltestParameterOverridesTest.mldatx');
```

Overriding a Model Parameter

1. Expand the test suite in the **Test Browser** pane and select the **Test Override** test case.
2. Scroll down to the **Baseline Criteria** section and click **Capture**.
3. Save the baseline file to writable folder.

▼ BASELINE CRITERIA*

Include baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL	+
▼ <input checked="" type="checkbox"/> DemoBaseline.mat	0	0.00%	0	0	
<input checked="" type="checkbox"/> Mux:1	0	0.00%	0	0	

4. Expand the **Parameter Overrides** section in the test case and click **Add**.
5. In the dialog box, click the Refresh button to display available parameters. Select **a**.
6. Click **OK**.

<input checked="" type="checkbox"/> WORKSPACE VARIABLE	CURRENT VALUE	SOURCE	MODEL ELEMENT
<input checked="" type="checkbox"/> a	1	model workspace	sltestParameterOverridesExample/Mu

7. The test case displays **a** in the overrides table. Double-click the **Override Value** and enter **1.1**.

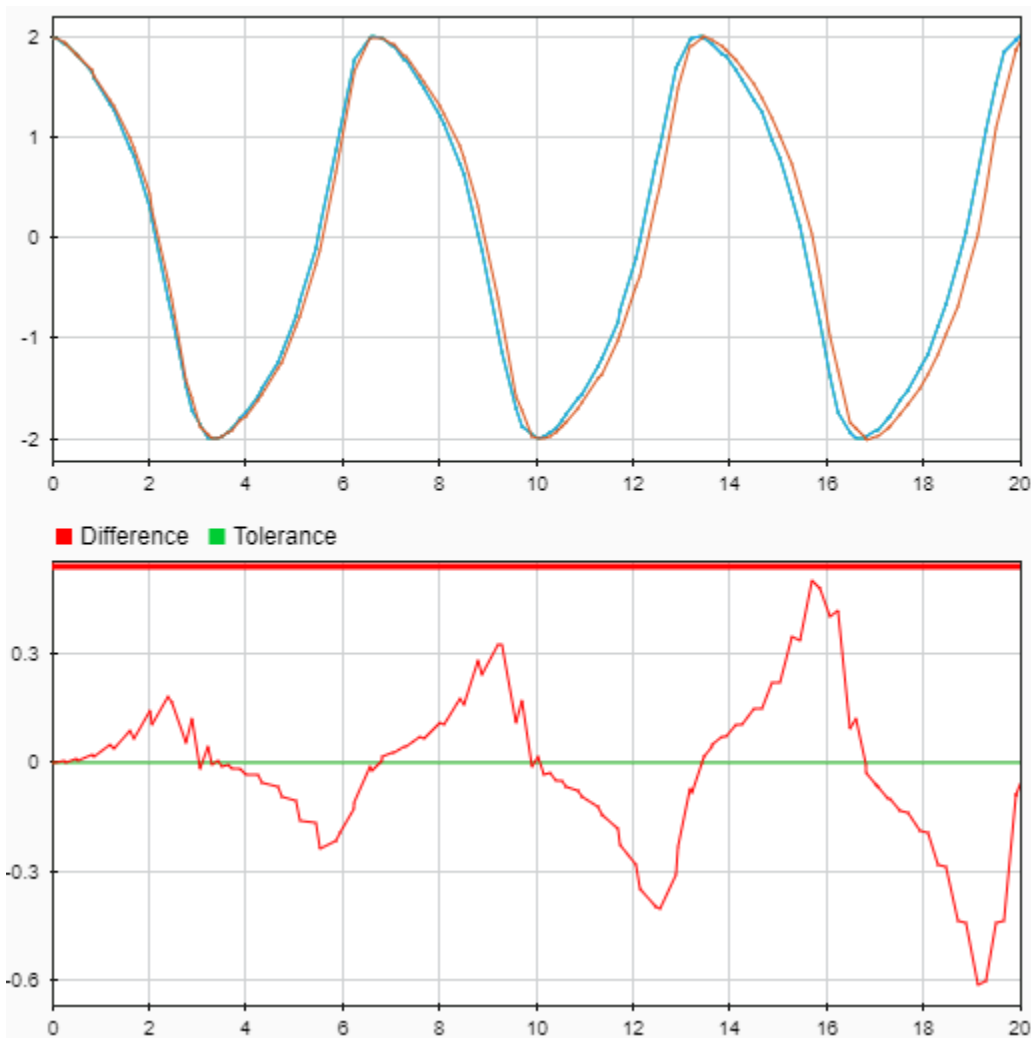
PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
▼ <input checked="" type="checkbox"/> Parameter Set 1			
<input checked="" type="checkbox"/> a	1.1	model workspace	sltestParameterOverridesExample/Mu

Run the Test and View Results

Select the test file in the **Test Browser** pane and click **Run**. In the **Results and Artifacts** pane, expand the results to see the **Baseline Criteria Result** and **Sim Output**.

Test Override	✘
Baseline Criteria Result	✘
<input type="radio"/> Mux:1[1]	✘
<input type="radio"/> Mux:1[2]	✘
Sim Output (sltestParameterOve	
<input type="checkbox"/> Mux:1[1]	—
<input type="checkbox"/> Mux:1[2]	—

Select Mux: 1[1] inside **Baseline Criteria Result** to see how overriding the parameter affected the mux signal when compared to the captured baseline. The comparison output shows a maximum difference of approximately **0.6**.



Overriding Parameters using Data Files

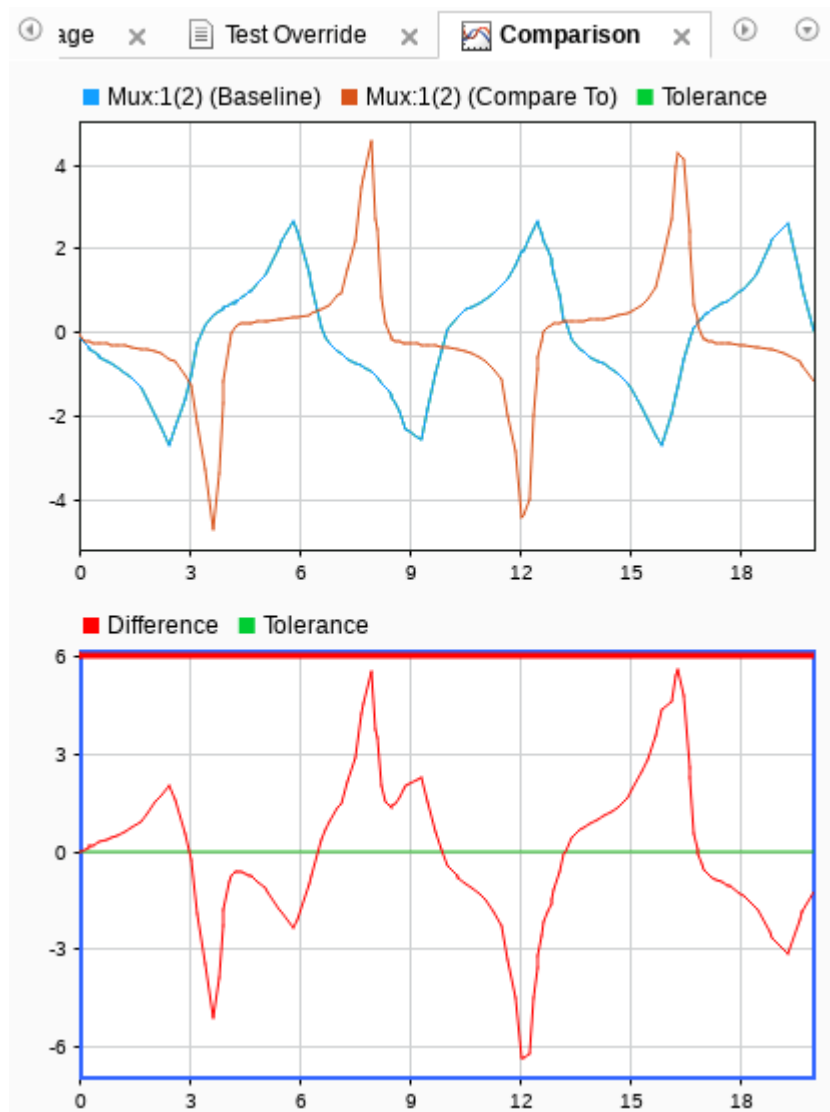
1. Return to the test case and scroll to the **Parameter Overrides** section.
2. Click the **Add** arrow and select **Add File** from drop down.

Select the `sltestOverrideExampleData.mat` file from the `matlab\examples\simulinktest` folder. This file contains data that can be used by the test case to override the parameters.

PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
<input type="checkbox"/> Parameter Set 1			
<input checked="" type="checkbox"/> a	1.1	model workspace	sltestParameterOverridesExample/Mu
<input checked="" type="checkbox"/> sltestParametersOverrideD...			
<input checked="" type="checkbox"/> a	2.718281828...		

Select a row, right-click, and select **Export**. Exports the variable to the MATLAB® base workspace with a variable name **a**.

Run the test again and review the results.



```
sltest.testmanager.clearResults;  
close(tf);  
sltest.testmanager.close
```

Test Two Simulations for Equivalence

Verify model equivalence in normal and SIL simulation mode.

This example shows how to test for equivalence between two models using test harnesses and the test manager. One model runs in normal mode, and a test harness model created from a subsystem runs in software-in-the-loop (SIL) mode.

The equivalence test case in the test manager compares signal output between two simulations to determine equivalence. Signals from the main model and the test harness are set up for logging in this example. The logged signals are used as the equivalence criteria between normal and SIL mode.

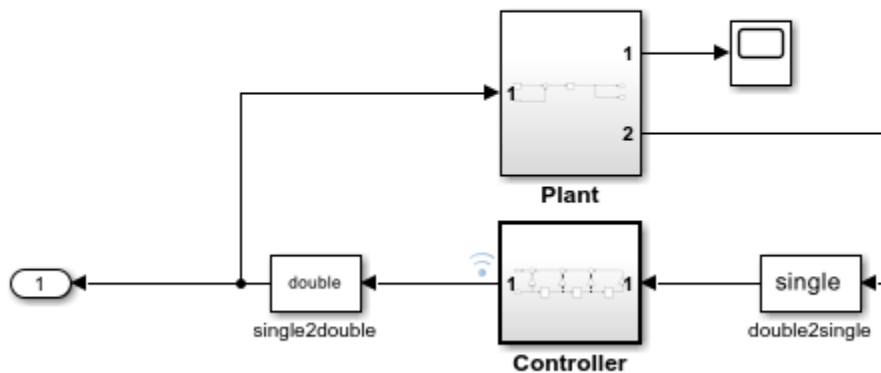
Configure the Model

Open the `sltestNormalSILEquivalenceExample` model.

```
mdl = 'sltestNormalSILEquivalenceExample';
harnessOwner = 'sltestNormalSILEquivalenceExample/Controller';
open_system(mdl);
```

Test Two Simulations for Equivalence

This example shows how to test for equivalence between two models using Simulink Test. To run the demo, enter `showdemo sltestSimulationEquivalence` in MATLAB(R).



Copyright 2015 The MathWorks, Inc.

Turn on signal logging in the model.

```
set_param(mdl, 'SignalLogging', 'on', 'SignalLoggingName', 'SIL_signals');
```

Mark the Controller subsystem output and input signals for logging.

```
ph_controller_in = get_param('sltestNormalSILEquivalenceExample/Controller/In1', 'PortHandles');
ph_controller_out = get_param('sltestNormalSILEquivalenceExample/Controller', 'PortHandles');
```

```
set_param(ph_controller_in.Outport(1), 'DataLogging', 'on');
set_param(ph_controller_out.Outport(1), 'DataLogging', 'on');
clear ph_controller_in ph_controller_out;
```

Simulate the model and output the logged signals. The signal data is used as input for the test harness.

```
out = sim mdl;
```

Get the logged signal data.

```
out_data = out.get('SIL_signals');
control_in1 = out_data.get(2);
```

Create a Test Harness for SIL Verification.

The command to create the harness will generate code. Switch to a directory with write permissions.

```
origDir = pwd;
dirName = tempname;
mkdir(dirName);
cd(dirName);
cleanup = onCleanup(@( )cd(origDir));
sltest.harness.create(harnessOwner, 'Name', 'SIL_Harness', 'VerificationMode', 'SIL');
```

```
### Starting build procedure for: Controller
### Successful completion of build procedure for: Controller
### Creating SIL block ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
```

Build Summary

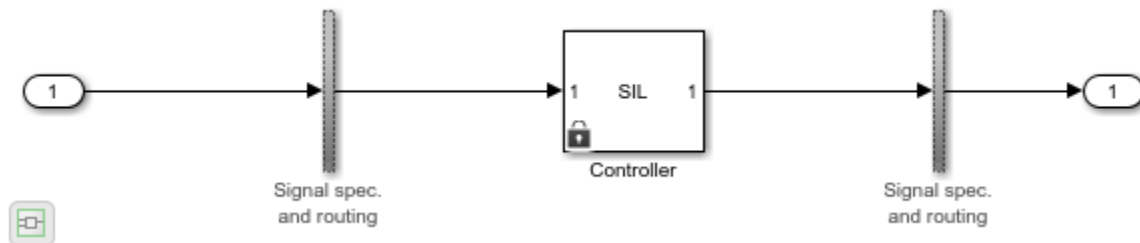
Top model targets built:

Model	Action	Rebuild Reason
Controller	Code generated and compiled.	Code generation information file does not exist.

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 49.004s
```

Open the test harness.

```
sltest.harness.open(harnessOwner, 'SIL_Harness');
```



Set Up Logging in the Test Harness

Turn on signal logging in the test harness.

```
set_param('SIL_Harness', 'SignalLogging', 'on', 'SignalLoggingName', 'SIL_signals');
```

Mark the test harness output for signal logging to use in the equivalence test case.


```
ph_harness_out = get_param('SIL_Harness/Controller','PortHandles');
set_param(ph_harness_out.Outport(1),'DataLogging','on');
clear ph_harness_out;
```

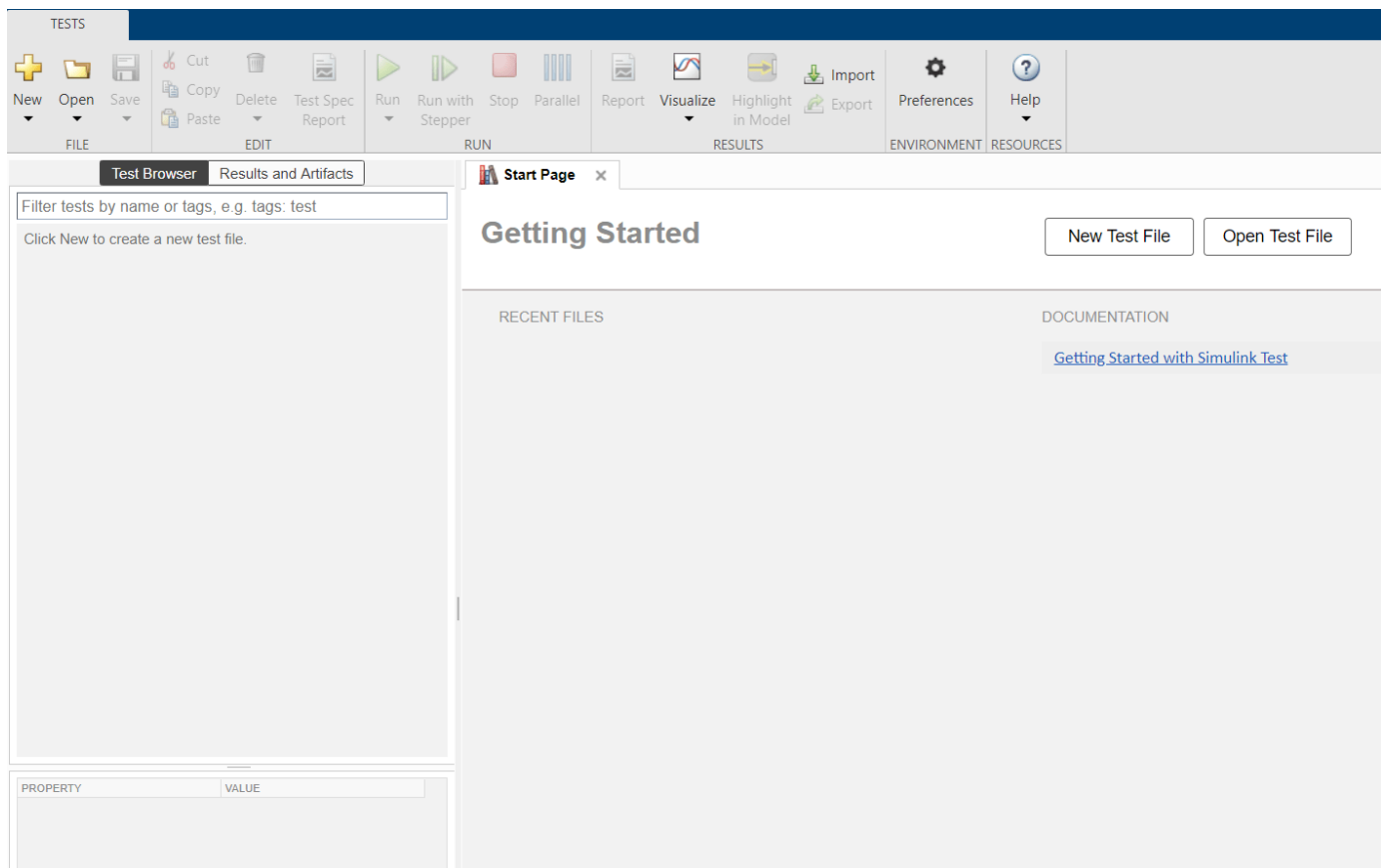
Assign the input data from the simulation to the test harness.

```
set_param('SIL_Harness','LoadExternalInput','on',...
    'ExternalInput','control_in1.Values');
```

Create an Equivalence Test Case in the Test Manager

Open the test manager by opening the **Apps** tab, clicking **Simulink Test** in the Model Verification, Validation, and Test section, and then clicking "Simulink Test Manager" in the "Tests" tab. Alternately, use the command

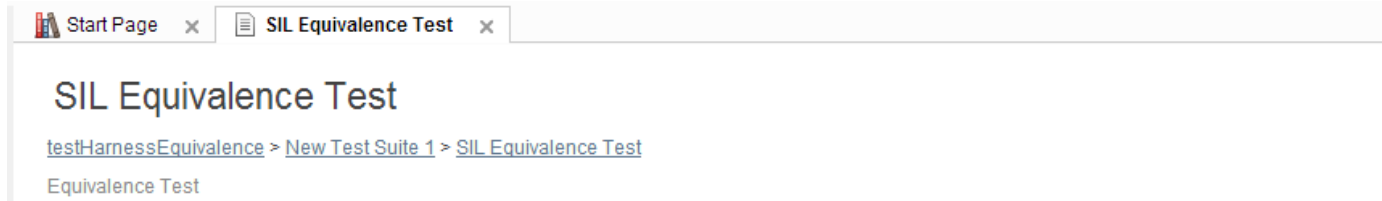
```
sltestmgr
```



Create an equivalence test case.

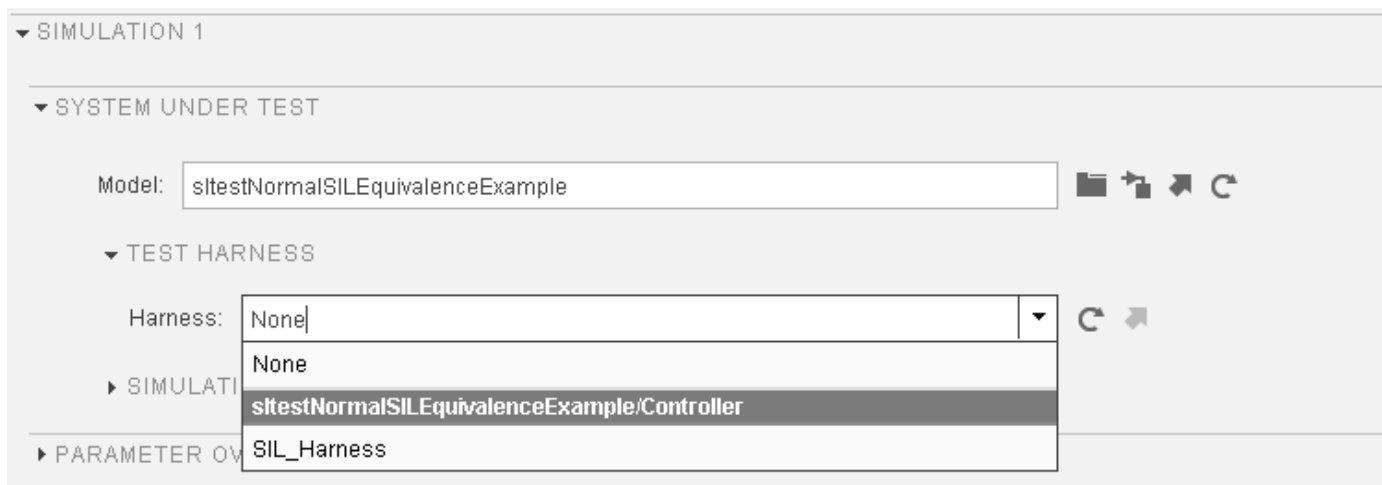
- 1 From the test manager toolbar, click the **New** arrow and select **Test File > Blank Test File**.
- 2 Specify the test file as `testHarnessEquivalence.mldatx`. The test manager creates the test file with a new test suite and baseline test case by default.
- 3 In the **Test Browser** pane, select the baseline test case, `New Test Case 1`, and click **Delete**.
- 4 Select `New Test Suite 1`.

- 5 From the toolstrip, click the **New** arrow and select **Equivalence Test**.
- 6 In the **Test Browser** pane, right-click the new equivalence test case and select **Rename**. Name the new equivalence test case **SIL Equivalence Test**.



Assign the test harness to the equivalence test case **Simulation 1**.

- 1 Expand **Simulation 1** and **System Under Test**.
- 2 Click the **Use current model** button to assign `sltestNormalSILEquivalenceExample` to **Model**.
- 3 Expand **Test Harness**.
- 4 Click the **Refresh** button to get an up-to-date list of available test harnesses.
- 5 Select `SIL_Harness` from the **Harness** menu to use as the **System Under Test**.



Assign the `sltestNormalSILEquivalenceExample` model as **Simulation 2**.

- 1 Collapse **Simulation 1**.
- 2 Expand **Simulation 2** and **System Under Test**.
- 3 Click the **Use current model** button to assign `sltestNormalSILEquivalenceExample` to **Model**.
- 4 Collapse **Simulation 2**.

Capture the equivalence criteria. Under **Equivalence Criteria**, click **Capture** to run the test harness in **Simulation 1** and identify the equivalence signal.

▼ EQUIVALENCE CRITERIA

<input checked="" type="checkbox"/> SIGNAL NAME	ABS TOL	REL TOL
<input checked="" type="checkbox"/> Controller:1	0	0.00%

Capture... Delete

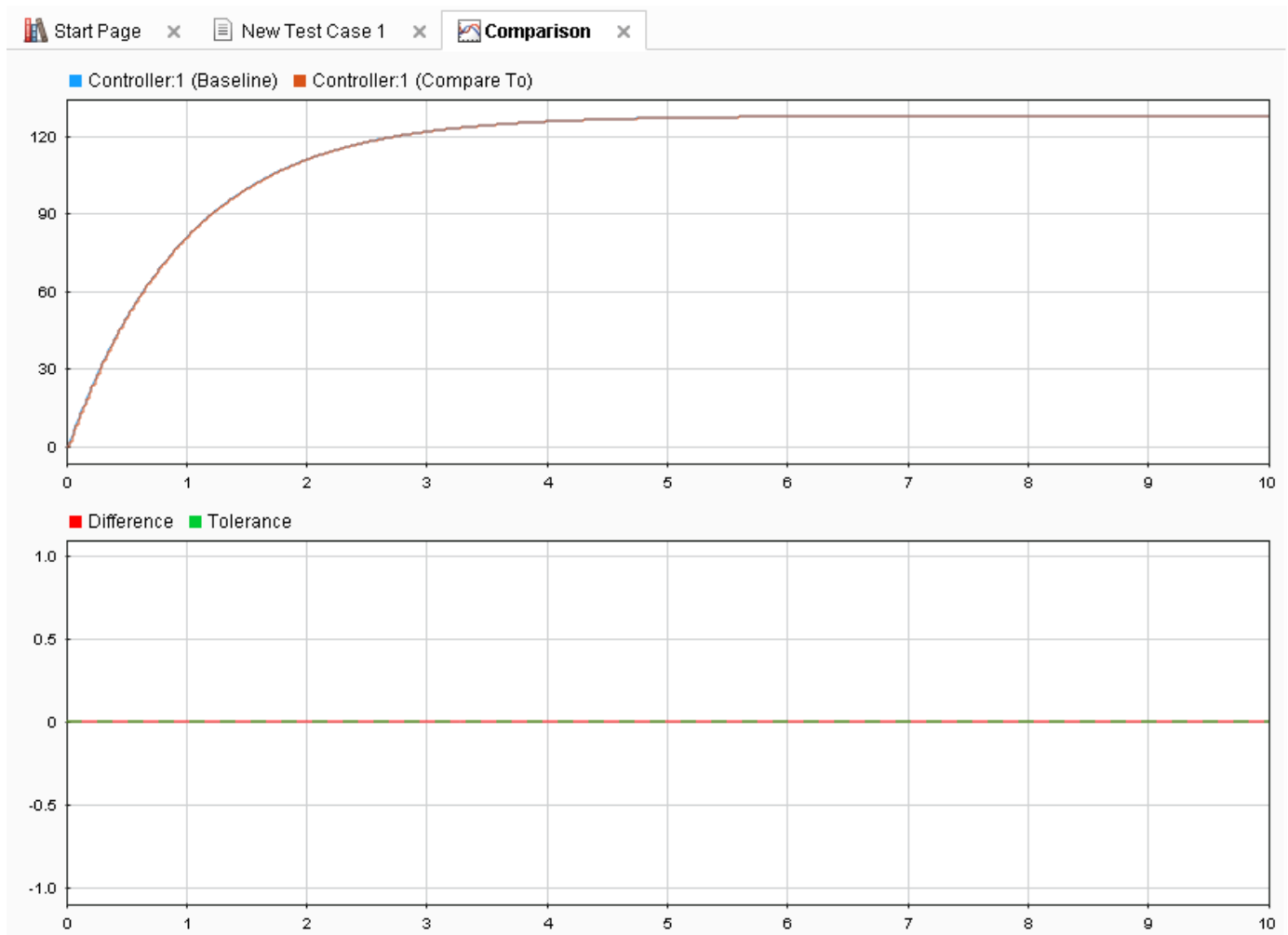
Run the Test Case and View the Results

Select SIL Equivalence Test in the **Test Browser** pane and click **Run** in the toolbar. The test manager switches to the **Results and Artifacts** pane and runs the equivalence test case. The test case passes because the signal comparison between the model and the test harness matches. Expand the results set and select the Controller:1 option button to plot the signal comparison.

Test Browser Results and Artifacts

Filter Results

NAME	STATUS
▼ Results: 2015-Dec-04 21:47:03	1 ✓
▼ New Test Case 1	✓
▼ Equivalence Criteria Result	✓
● Controller:1	✓
▶ Sim Output 1 (sltestNormalSILEquivalenc	
▶ Sim Output 2 (sltestNormalSILEquivalenc	



```
close_system mdl, 0);  
clear mdl harnessOwner cleanup control_in1 origDir out out_data;
```

See Also

Test Manager

More About

- “Create and Run a Back-to-Back Test” on page 6-41
- “SIL Verification for a Subsystem” on page 5-2

Create and Run a Back-to-Back Test

In this section...

“Run the Back-to-Back Test” on page 6-45

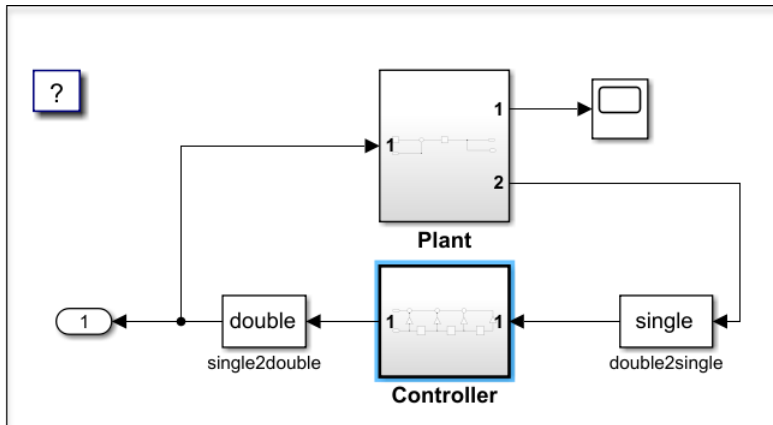
“View the Back-to-Back Test Results” on page 6-45

This example shows how to create and run a back-to-back test, which is also known as an equivalence test. Back-to-back tests compare the results of normal simulations with the generated code results from software-in-the-loop, processor-in-the-loop, or hardware-in-the-loop simulations.

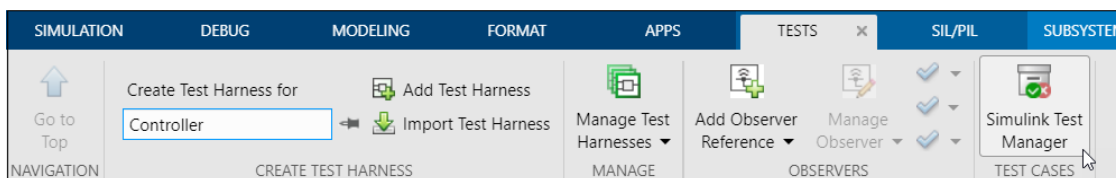
- 1 Open the SILBlock model.

```
openExample('SILBlock.slx')
```

- 2 Click the Controller subsystem to select it.



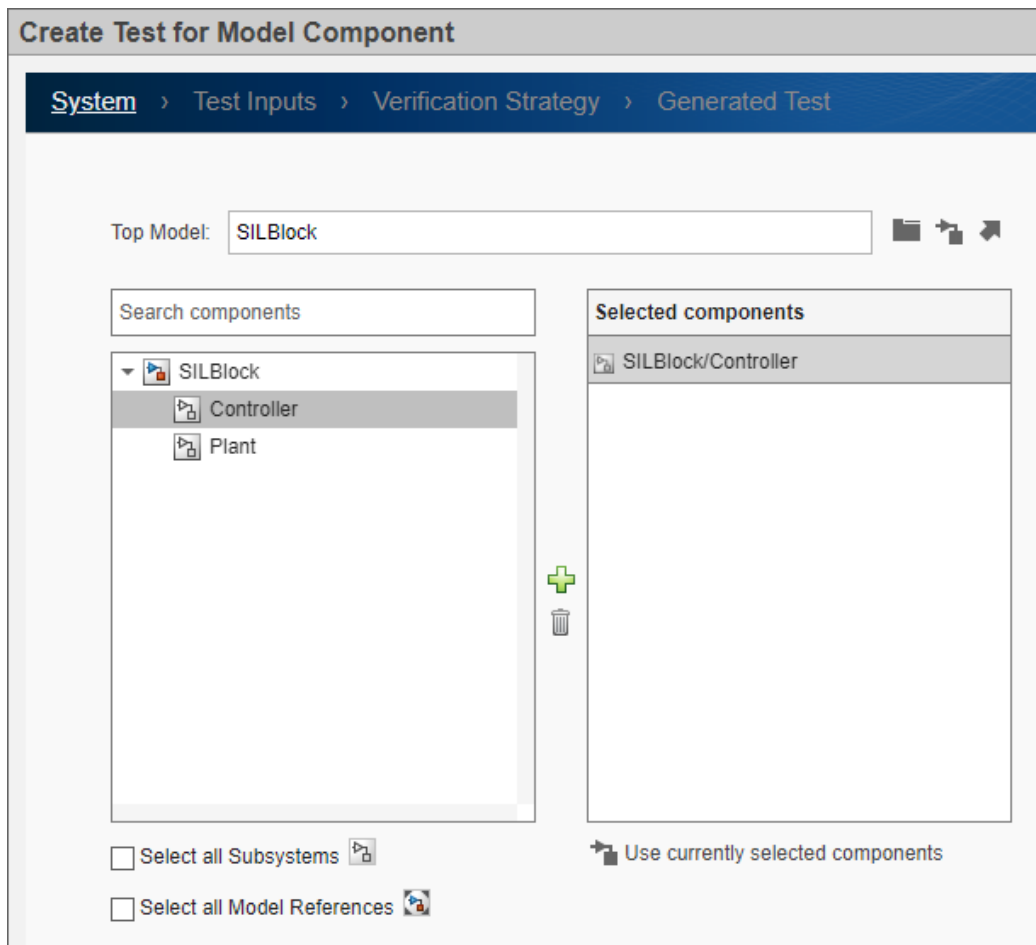
- 3 To open the **Simulink Test** tab, in the **Apps** tab, in the Model Verification, Validation, and Test section, click **Simulink Test**.
- 4 To open the Test Manager, in the **Tests** tab, click **Simulink Test Manager**.



- 5 Click **New > Test for Model Component**. The Create Test for Model Component wizard opens.

- 6 To specify the **Top model**, click the Use current model button  next to the **Top Model** field.

To add the Controller subsystem you selected in the model, click **Use currently selected components**.



- 7 Click **Next** to specify how to obtain the test harness inputs. Select **Use component input from the top model as test input**. This option runs the model and creates the test harness inputs using the inputs to the selected model component.

Create Test for Model Component

[System](#) > [Test Inputs](#) > [Verification Strategy](#) > [Generated Test](#)

How do you want to setup the inputs?

- Use component input from the top model as test input
Create harness inputs by simulating the top model and recording the component inputs
- Use Design Verifier to generate test input scenarios
Create inputs using Simulink Design Verifier. [Design Verifier Settings](#)
- Specify inputs in the created harness
Create a new test harness for component. Inputs should be added to the harness

- 8 Click **Next** to select the testing method. Click **Perform back-to-back testing**. For **Simulation1**, use Normal. For **Simulation2**, use Software-in-the-Loop (SIL).

Create Test for Model Component

[System](#) > [Test Inputs](#) > [Verification Strategy](#) > [Generated Test](#)

How do you want to test the component?

- Use component under test output as baseline
Simulate the top model and record the outputs of the component to be used as baseline
- Perform back-to-back testing
Set up a test to compare the component under test outputs in different simulation modes

Select simulation modes:

Simulation1: ▼

Simulation2: ▼

- Define the verification logic in the created harness
No verification logic will be automatically added to the test

- 9 Click **Next** to specify the test harness input source, format, and where to save the test data and generated tests. For **Specify the file format** in which to save the test data, select EXCEL. For **Specify the location to save test data**, use the default location name. Enter B2BtestFile for the **Test File Location**.

Create Test for Model Component

[System](#) > [Test Inputs](#) > [Verification Strategy](#) > [Generated Test](#)

How do you want to save the test data?

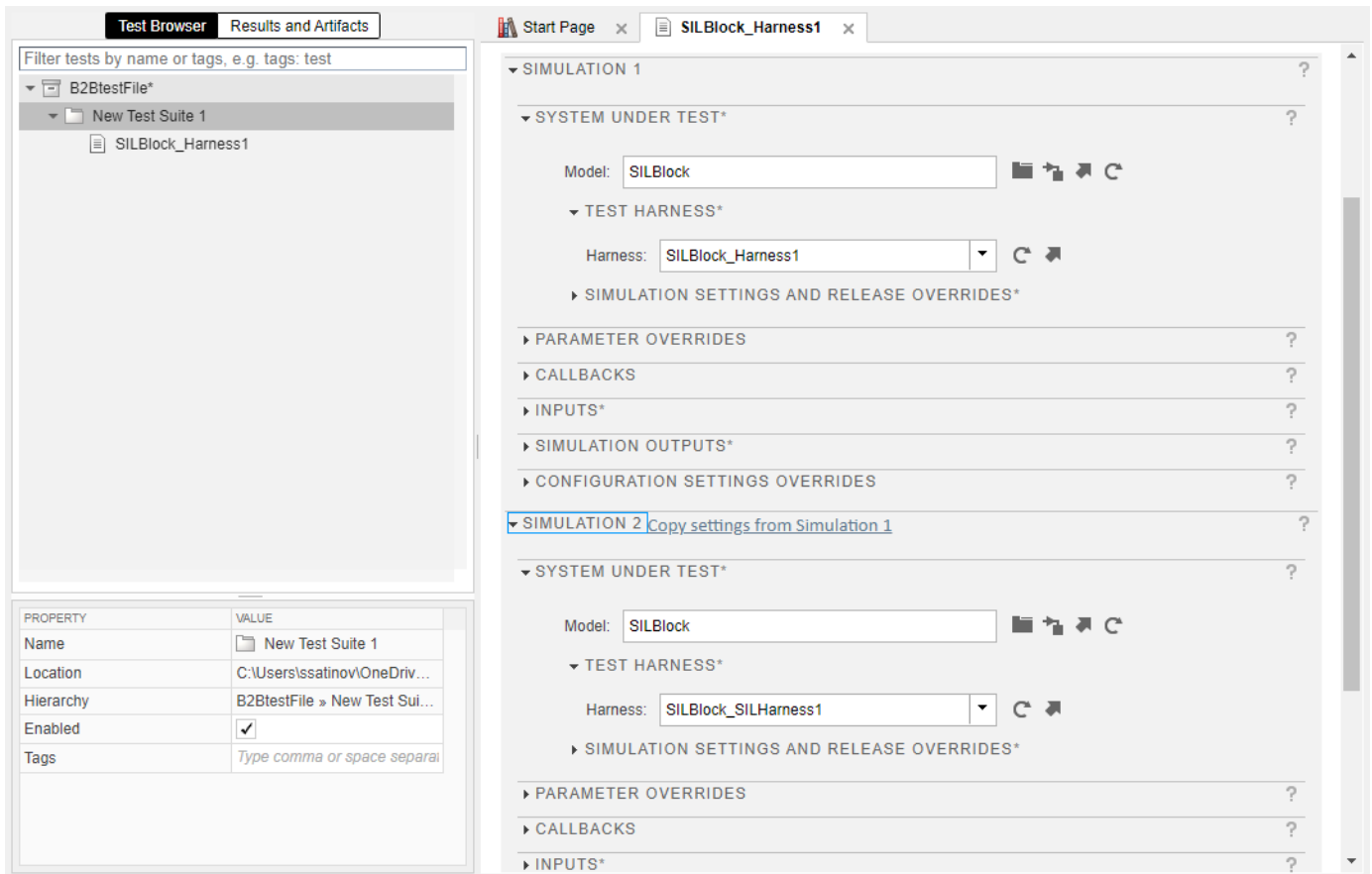
Select test harness input source: Specify the file format:

Specify location to save test data:

Where do you want to save the generated test(s)?

Test File Location:

10 Click **Done**. The test harness and test case are created and the wizard closes.

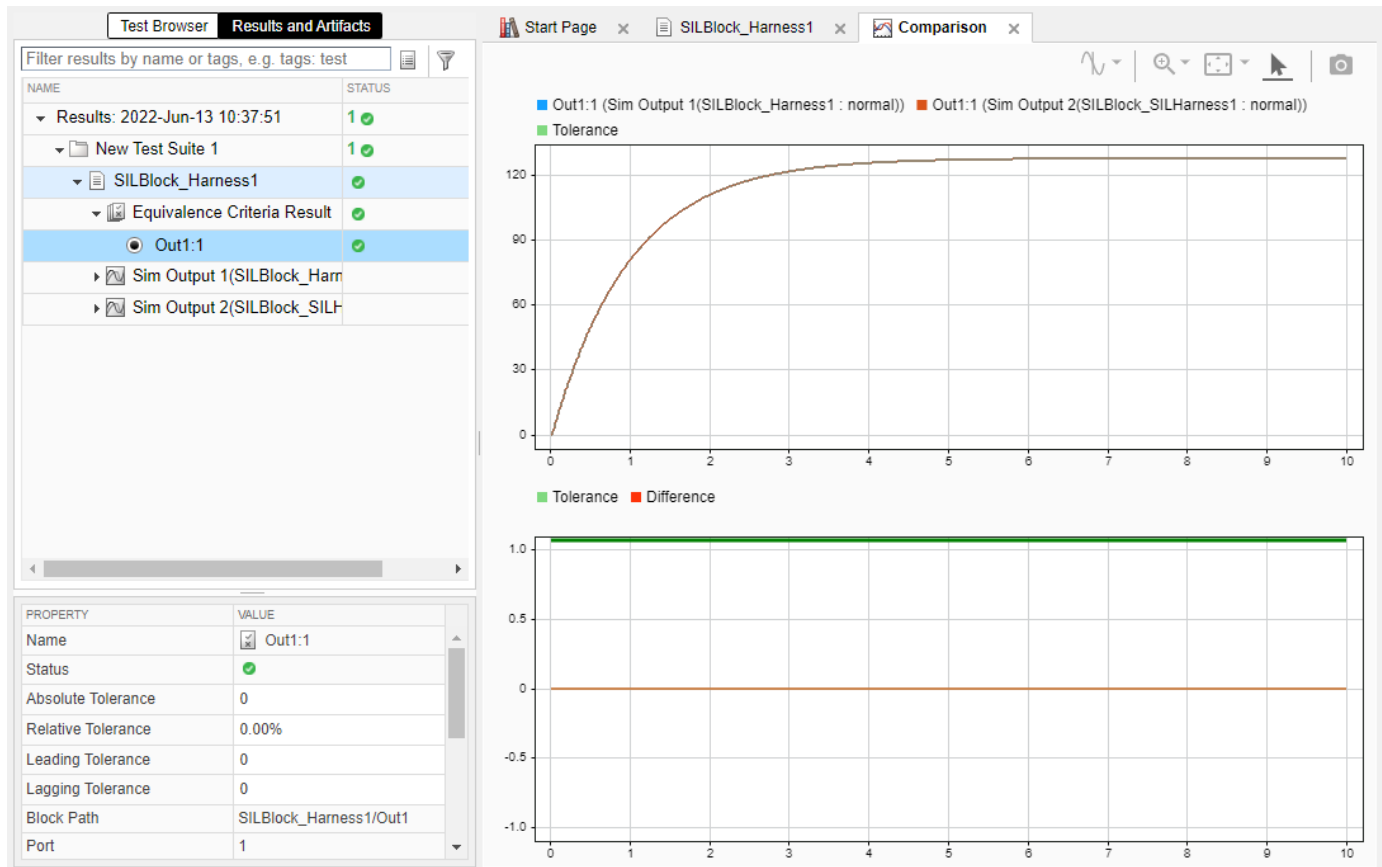


Run the Back-to-Back Test

To run the back-to-back test, click **Run**.

View the Back-to-Back Test Results

Expand the Results hierarchy in the **Results and Artifacts** panel. Select **Out1:1** under Equivalence Criteria Result. The upper plot shows that the output signals align and the lower plot shows that there is zero difference between the output signals.



See Also

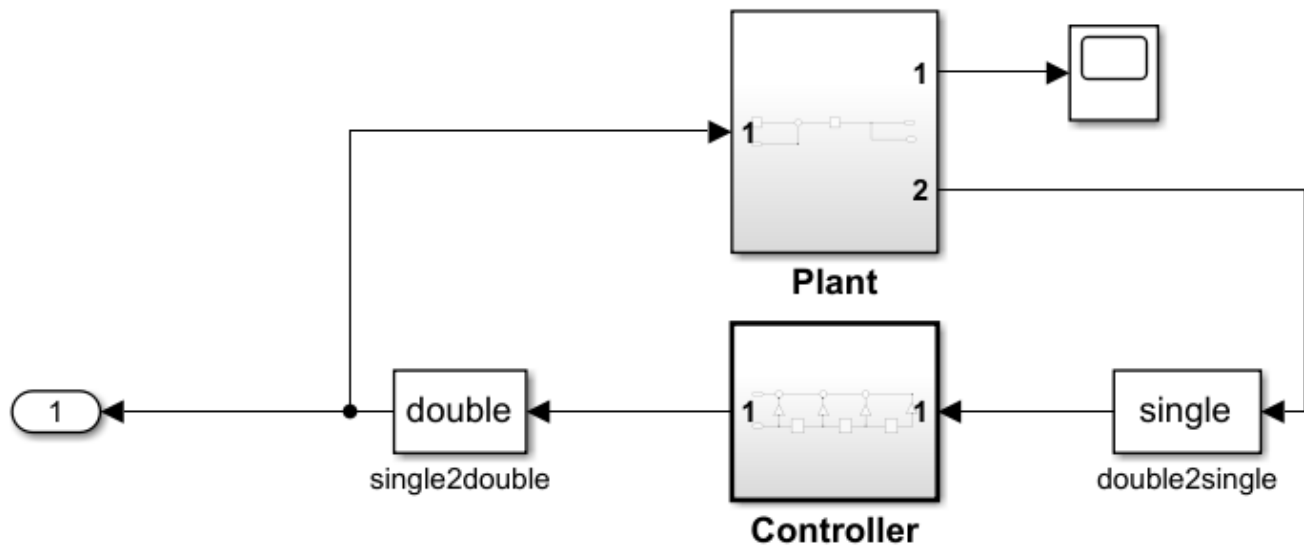
[sltest.testmanager.TestSuite](#) | [sltest.testmanager.TestFile](#) | **Test Manager**

More About

- “Generate Tests and Test Harnesses for a Model or Components” on page 6-24
- “Perform Back-to-Back (MIL/SIL) Equivalence Test for an Atomic Subsystem” on page 6-47
- “Create and Run Back-to-Back Tests Using Enhanced MCDC” (Simulink Design Verifier)

Perform Back-to-Back (MIL/SIL) Equivalence Test for an Atomic Subsystem

This example shows how to create and run a back-to-back test, which is also known as an equivalence test, for an atomic subsystem.



Generate Code for the Model

Generate code for the entire model.

```
model = 'sltestMILSILEquivalence';
open_system(model);
% Configure for code generation
if ismac
    lProdHWDeviceType = 'Intel->x86-64 (Mac OS X)';
elseif isunix
    lProdHWDeviceType = 'Intel->x86-64 (Linux 64)';
else
    lProdHWDeviceType = 'Intel->x86-64 (Windows64)';
end
set_param(model, 'ProdHWDeviceType', lProdHWDeviceType);
slbuild(model);

### Starting build procedure for: sltestMILSILEquivalence
### Successful completion of build procedure for: sltestMILSILEquivalence
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
sltestMILSILEquivalence	Code generated and compiled.	Code generation information file does not

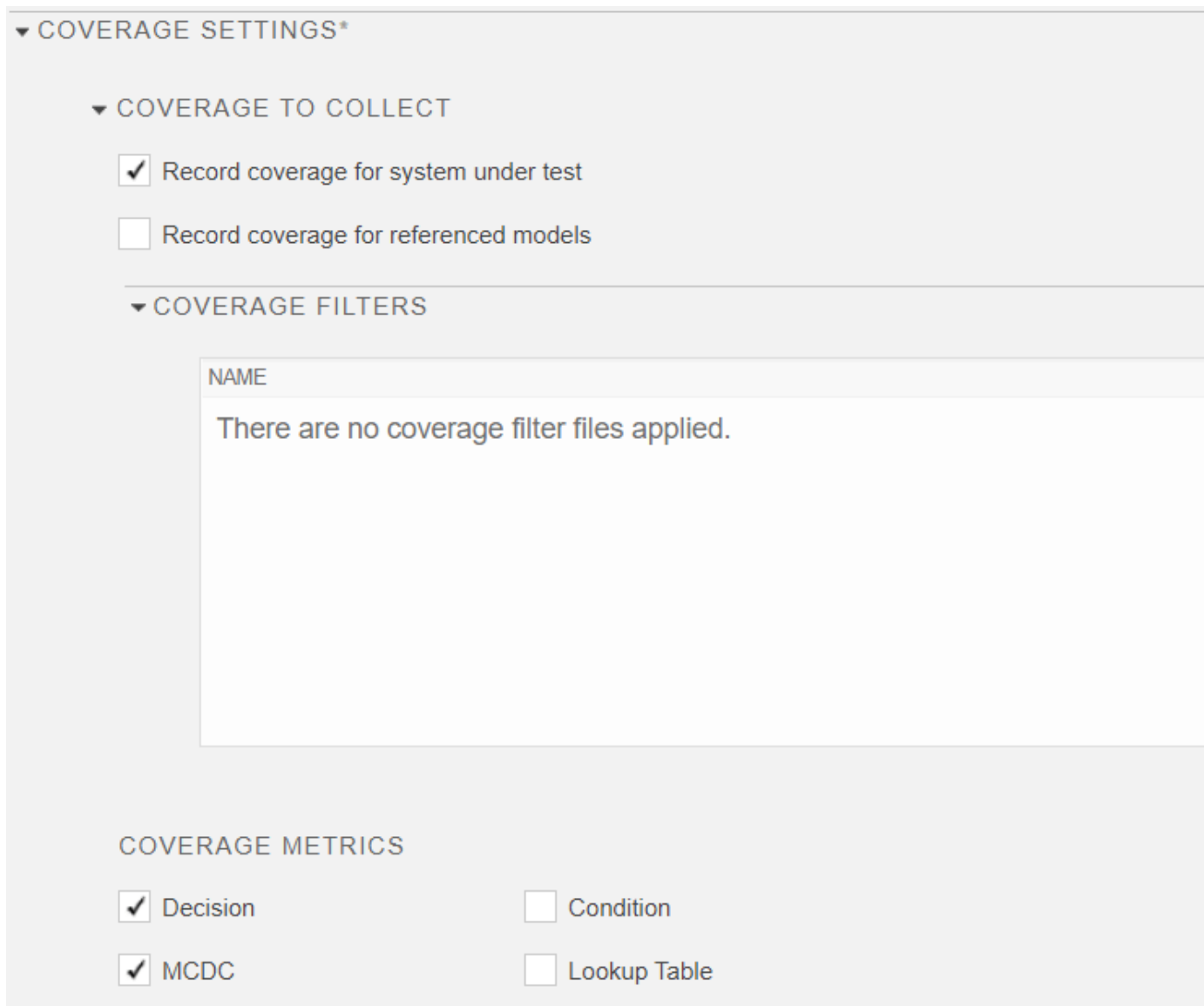
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 19.23s

Create a Back-to-Back Test with the Test For Model Component Wizard

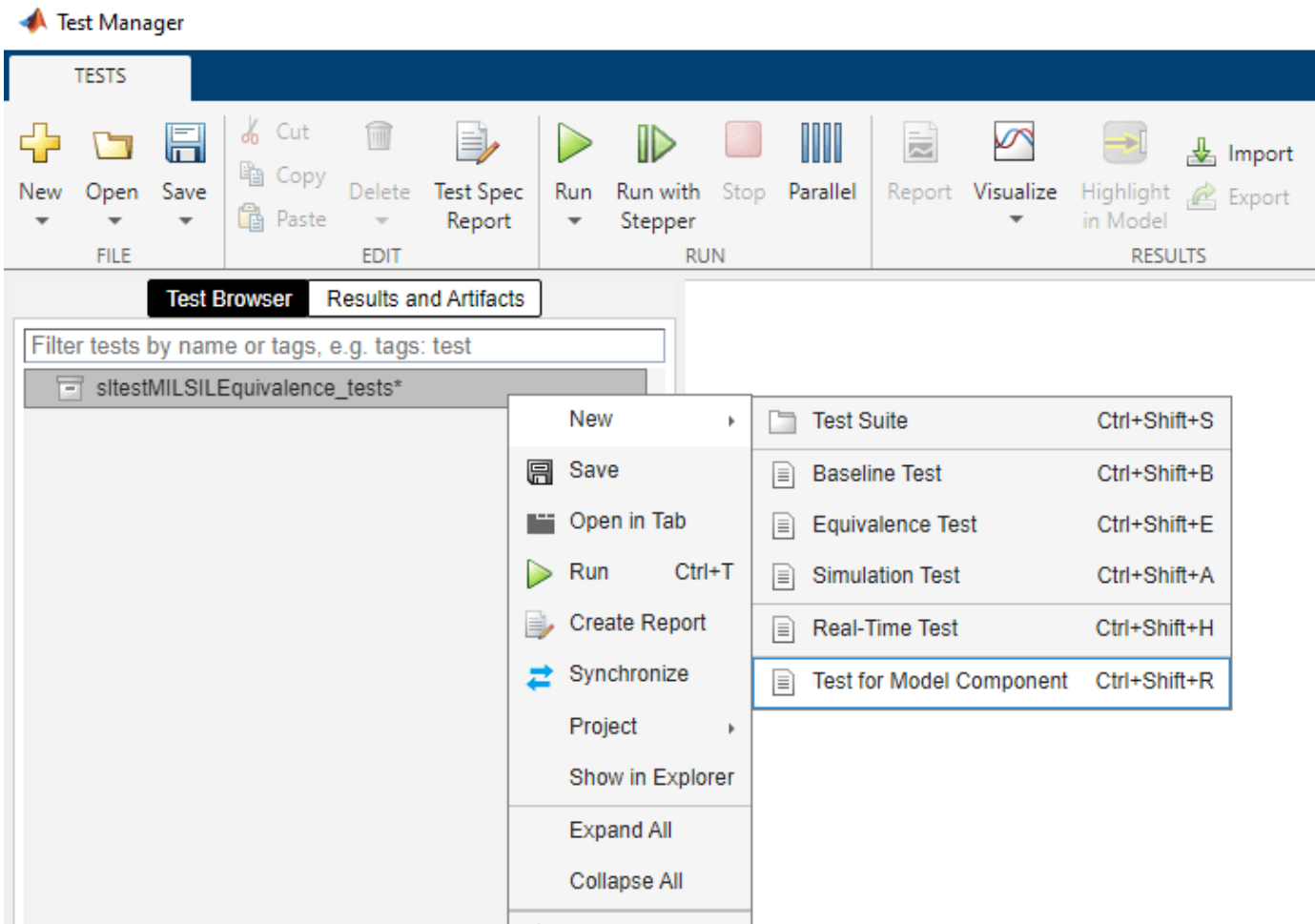
1. Create a new test file and open the Test Manager.

```
tFile = sltest.testmanager.TestFile('sltestMILSILEquivalence_tests.mldatx');  
sltest.testmanager.view();
```

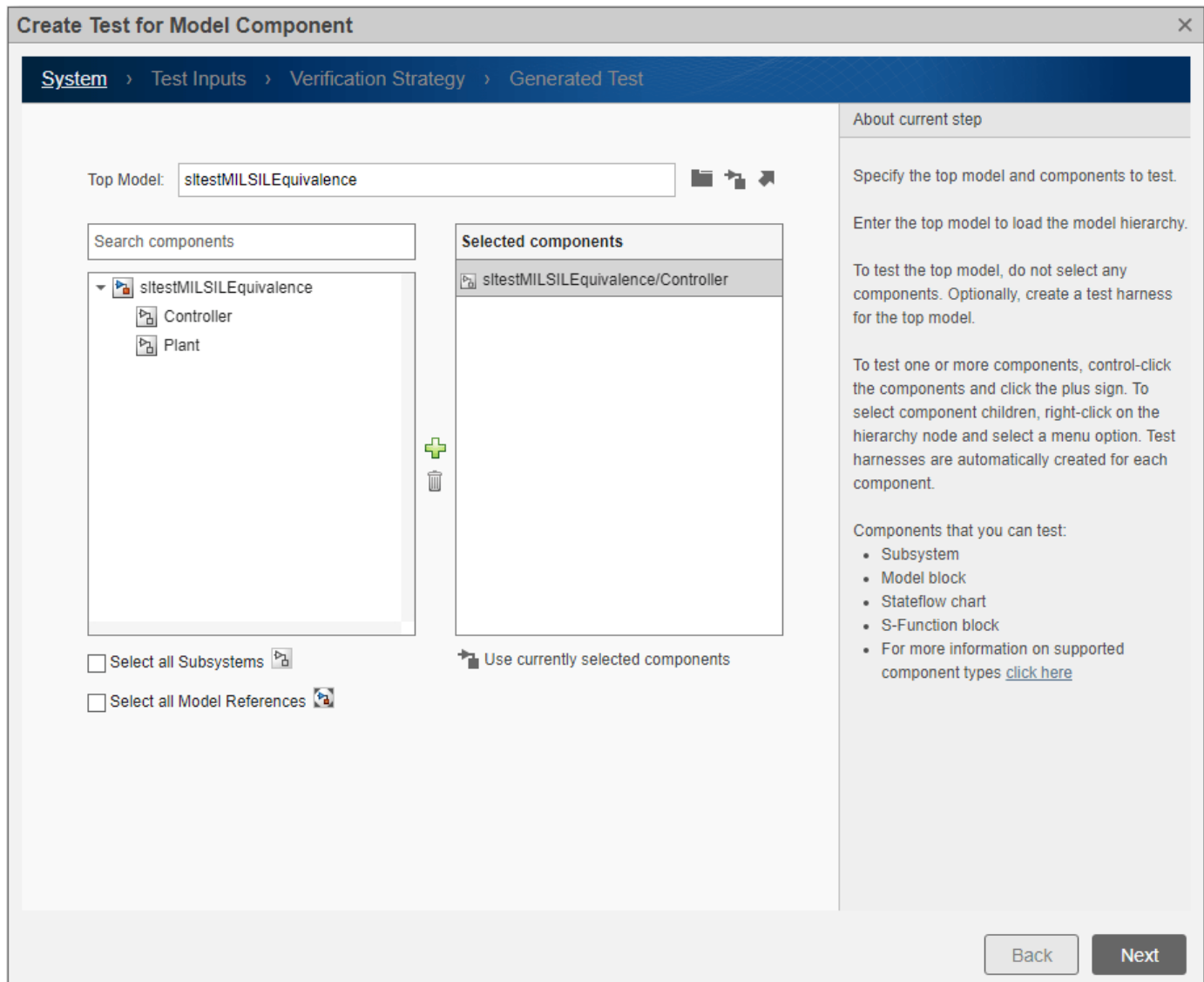
2. Enable coverage collection for the test file. in the **Coverage Settings** section of the Test Manager, enable **Record coverage for system under test** under Coverage to Collect and select **Decision** and **MCDC** under Coverage Metrics.



3. Right-click the test file and select **New > Test For Model Component**.



4. Click the Use current model icon to add the **Top Model**. Select Controller and click the plus sign. Then click **Next**.



5. Select **Use Design Verifier to generate test input scenarios**. Then click **Next**.

Create Test for Model Component [X]

System > Test Inputs > Verification Strategy > Generated Test

How do you want to setup the inputs?

- Use component input from the top model as test input
Create harness inputs by simulating the top model and recording the component inputs
- Use Design Verifier to generate test input scenarios
Create inputs using Simulink Design Verifier. [Design Verifier Settings](#)
 - Simulate top model and use the recorded component inputs in the analysis
- Specify inputs in the created harness
Create a new test harness for component. Inputs should be added to the harness

About current step

Select how to generate input vectors to test the component.

About selected option

Runs Design Verifier Analysis on the specified component to generate inputs that satisfy coverage objectives.

To change Design Verifier settings for the parent model click the link.

A test harness with the selected input source shall be created.

Back Next

6. Select **Perform back-to-back testing**. Check that **Simulation1** is set to Normal and **Simulation2** is set to Software-in-the-Loop (SIL). Then click **Next**.

Create Test for Model Component

System > Test Inputs > Verification Strategy > Generated Test

How do you want to test the component?

Use component under test output as baseline
Simulate the top model and record the outputs of the component to be used as baseline

Perform back-to-back testing
Set up a test to compare the component under test outputs in different simulation modes

Select simulation modes:

Simulation1: Normal

Simulation2: Software-in-the-Loop (SIL)

Set Model coverage objectives as Enhanced MCDC

Define the verification logic in the created harness
No verification logic will be automatically added to the test

About current step

Select how to verify the test outputs.

You can perform baseline testing, back to back (equivalency) testing, or manually set up the verification logic in the auto created test harness.

About selected option

Auto creates the required test harnesses for the component under test.

Sets up a test case containing two simulations to compare the harness outputs in the selected simulation modes.

You can set 'Enhanced MCDC' as coverage objectives for Design Verifier test generation. [Click here](#) for more details.

Back Next

7. Click **Done** on the Generated Test page. The back-to-back (equivalence) test is created.

▼ SIMULATION 1

▼ SYSTEM UNDER TEST*

Model:

▼ TEST HARNESS*

Harness:

▼ SIMULATION SETTINGS AND RELEASE OVERRIDES*

Simulation Mode:

▼ SIMULATION 2 [Copy settings from Simulation 1](#)

▼ SYSTEM UNDER TEST*

Model:

▼ TEST HARNESS*

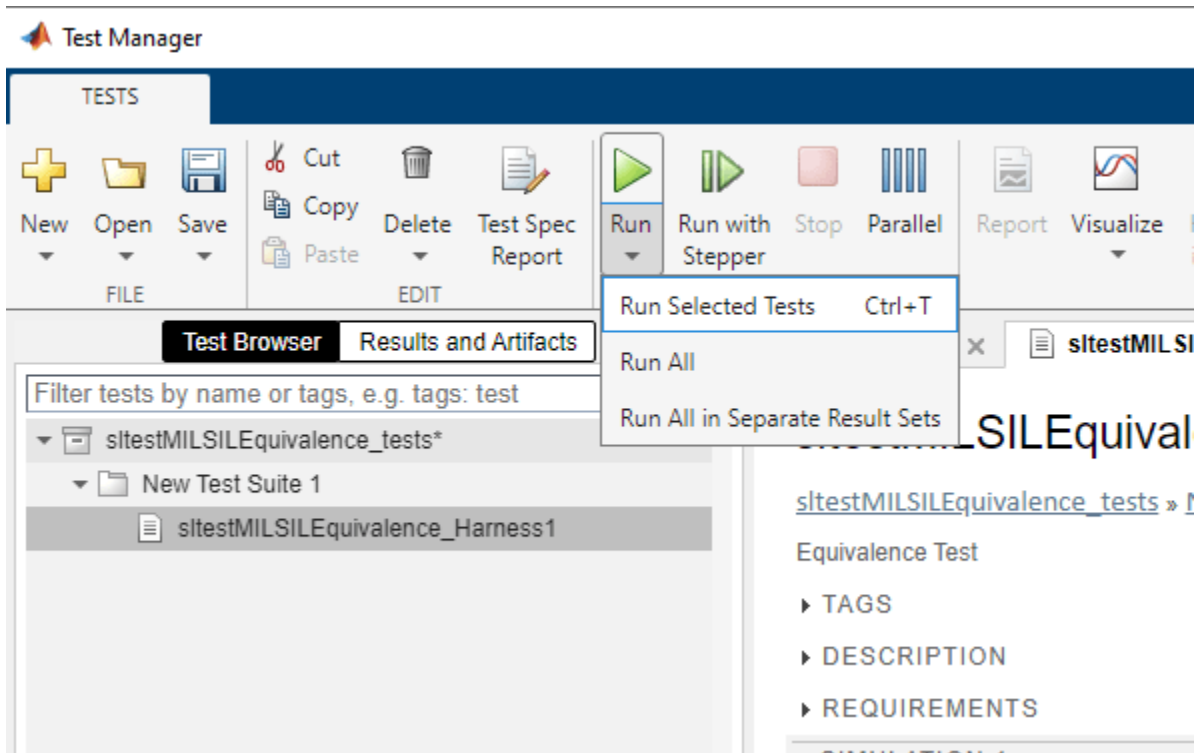
Harness:

▼ SIMULATION SETTINGS AND RELEASE OVERRIDES*

Simulation Mode:

Run the Test and View the Results

Select **sltestMILSILEquivalence_Harness1**. Then click the arrow below **Run** and select Run Selected Tests.



When the test completes, select **Results** and view the **Summary** and **Aggregated Coverage Results** sections.

▼ SUMMARY

Name	Results: 2022-Jun-28 12:16:30
Outcome	1
Start Time	06/28/2022 12:16:36
End Time	06/28/2022 12:18:07
Type	Result Set

▼ AGGREGATED COVERAGE RESULTS

Create a coverage report from coverage results to justify or exclude missing coverage. The filters and updated coverage values will be displayed with this result.

ANALYZED MODEL	REPORT	SIM MODE	COMPLEXITY	EXECUTION	FUNCTION
sltestMILSILEquivalence/Controller		Normal	1	100%	--
sltestMILSILEquivalence/Controller		SIL	14	100%	100%

Testing AUTOSAR Compositions

Run back-to-back tests on an AUTOSAR composition model.

This example demonstrates test harness features and back-to-back testing workflows for an AUTOSAR composition model. Switch to a directory with write permissions.

The example uses a model of a throttle position controller for an automobile. It is based closely on a shipping AUTOSAR Blockset example. For details, see “Import AUTOSAR Composition to Simulink” (AUTOSAR Blockset).

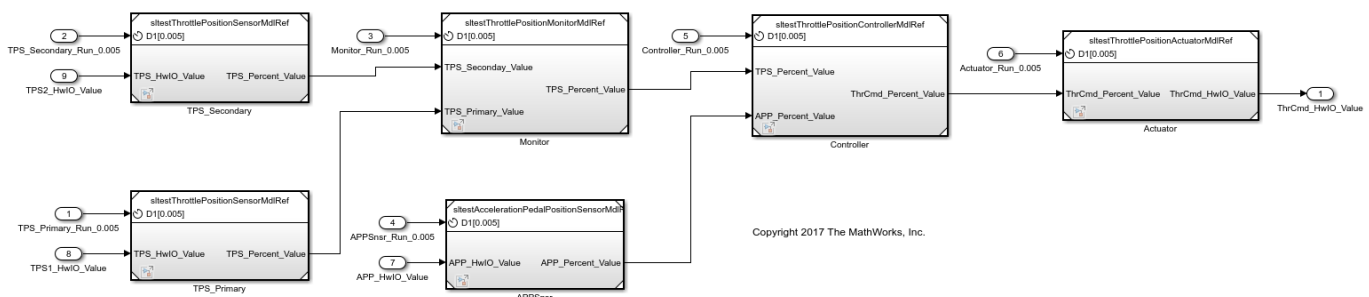
Open the AUTOSAR Composition Model

AUTOSAR composition models contain a network of interconnected Model blocks, each of which represents an atomic AUTOSAR software component (ASWC). The throttle position controller composition was created by an AUTOSAR authoring tool (AAT) and imported into Simulink using an ARXML file that describes the composition.

The composition model contains six component models, one for each atomic software component in the composition. Simulink inports and outports represent AUTOSAR ports and signal lines represent AUTOSAR component connectors.

```
mdl = 'sltestThrottlePositionControlCompositionExample.slx';
open_system(mdl);
```

Testing AUTOSAR Compositions



This model represents an AUTOSAR composition which contains a number of AUTOSAR atomic software components (ASWCs):

- 1) The Model blocks represent AUTOSAR ASWC prototypes.
- 2) The signal lines between the Model blocks represent AUTOSAR assembly connectors.
- 3) The signal lines between the Model blocks and data Inports/Outports represent AUTOSAR delegation connectors.

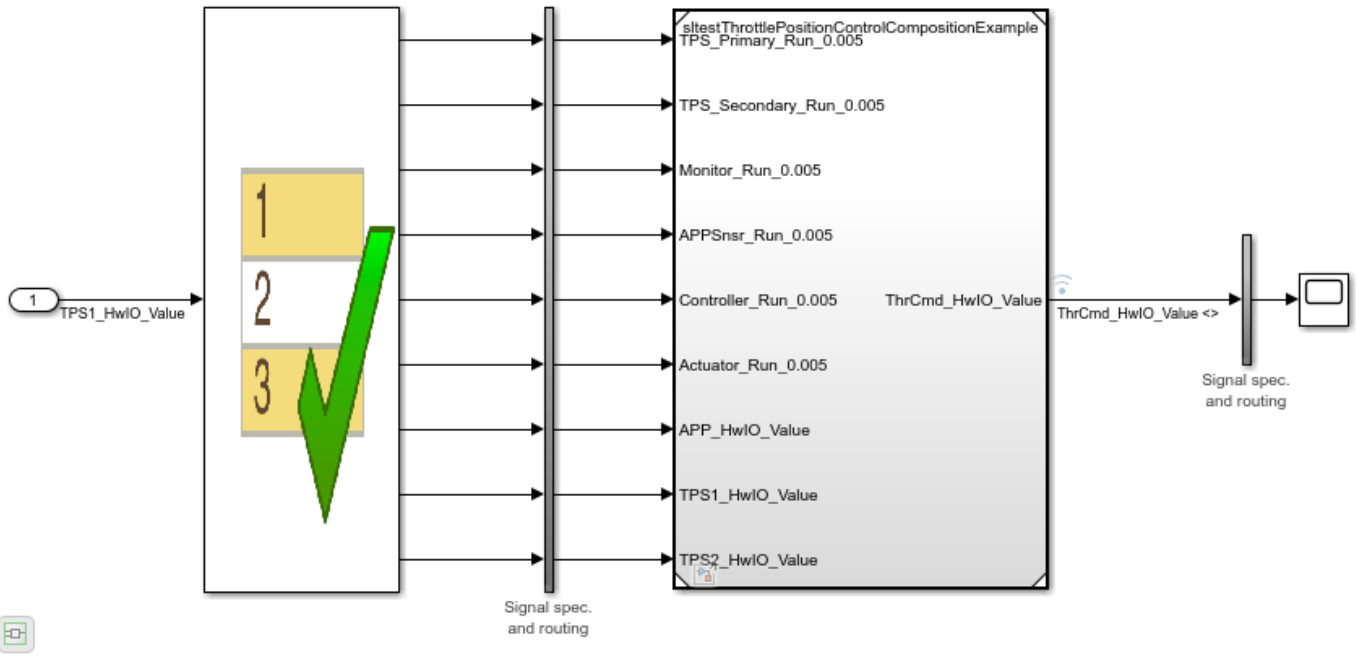


Copyright 2017 The MathWorks, Inc.

Open Test Harness

A test harness for the model has been generated and can be opened using the perspective control in the lower right corner of the editor canvas. Alternately, use:

```
sltest.harness.open('sltestThrottlePositionControlCompositionExample', ...
    'BasicSchedulerTest');
```



A Test Sequence block is used as the source. The component under test requires the accelerator pedal position sensor input APP_HwIO_Value, which is modeled in the Test Sequence block using a simple three step sequence:

Step	Transition	Next Step	Description
<pre>Initialize %% Initialize data outputs. APP_HwIO_Value = uint16(170); TPS1_HwIO_Value = TPS_HwIO_In; TPS2_HwIO_Value = TPS_HwIO_In; %% Schedule function-call outputs. send(TPS_Primary_Run_0005) send(TPS_Secondary_Run_0005) send(Monitor_Run_0005) send(APPsnsr_Run_0005) send(Controller_Run_0005) send(Actuator_Run_0005)</pre>	1. after(0.05,sec)	Run	Initialize the acceleration pedal position command to a nominal value and hold for 50 ms.
<pre>Run %% Initialize data outputs. APP_HwIO_Value = uint16(681); TPS1_HwIO_Value = TPS_HwIO_In; TPS2_HwIO_Value = TPS_HwIO_In; %% Schedule function-call outputs. send(TPS_Primary_Run_0005) send(TPS_Secondary_Run_0005) send(Monitor_Run_0005) send(APPsnsr_Run_0005) send(Controller_Run_0005) send(Actuator_Run_0005)</pre>	1. after(0.95, sec)	Terminate	Start the test by applying a steady acceleration command for 950 ms.
<pre>: Terminate %% Initialize data outputs. APP_HwIO_Value = uint16(170); TPS1_HwIO_Value = TPS_HwIO_In; TPS2_HwIO_Value = TPS_HwIO_In; %% Schedule function-call outputs. send(TPS_Primary_Run_0005) send(TPS_Secondary_Run_0005) send(Monitor_Run_0005) send(APPsnsr_Run_0005) send(Controller_Run_0005) send(Actuator_Run_0005)</pre>			Reset the acceleration command to the nominal value and terminate the test.

The **Initialize** step sets the input to a nominal value and the **Run** step models a steady acceleration command for 950 ms. The acceleration command is reset to the nominal value in the **Terminate** step. The component under test requires two additional inputs that capture the primary and secondary throttle position sensor readouts. These inputs are modeled using an external time series input and are directly fed through the Test Sequence block without modification. This modeling style is useful when some stimulus inputs can be modeled and others are only available as externally captured data.

Test Harnesses for Export Functions

The component under test is the AUTOSAR composition model, which uses the export-function modeling style. When you create a test harness for an export-function model, the harness will contain a Test Sequence block configured to call each root-level Simulink Function block and send a trigger event to each function-call subsystem in the model. The generated Test Sequence block can be used as a convenient starting point for modeling a scheduler.

In this example, since the input signal data is also being generated by a Test Sequence source block, the code to send the trigger events has been consolidated into a single Test Sequence block and

embedded in each step after the stimulus waveforms have been generated. The call order of the trigger events are computed using compiled information from the composition model.

- 1 send(TPS_Primary_Run_0005)
- 2 send(TPS_Secondary_Run_0005)
- 3 send(Monitor_Run_0005)
- 4 send(APPSnsr_Run_0005)
- 5 send(Controller_Run_0005)
- 6 send(Actuator_Run_0005)

Simulate the model to see the throttle command output from the component under test.

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



Back-to-back Testing

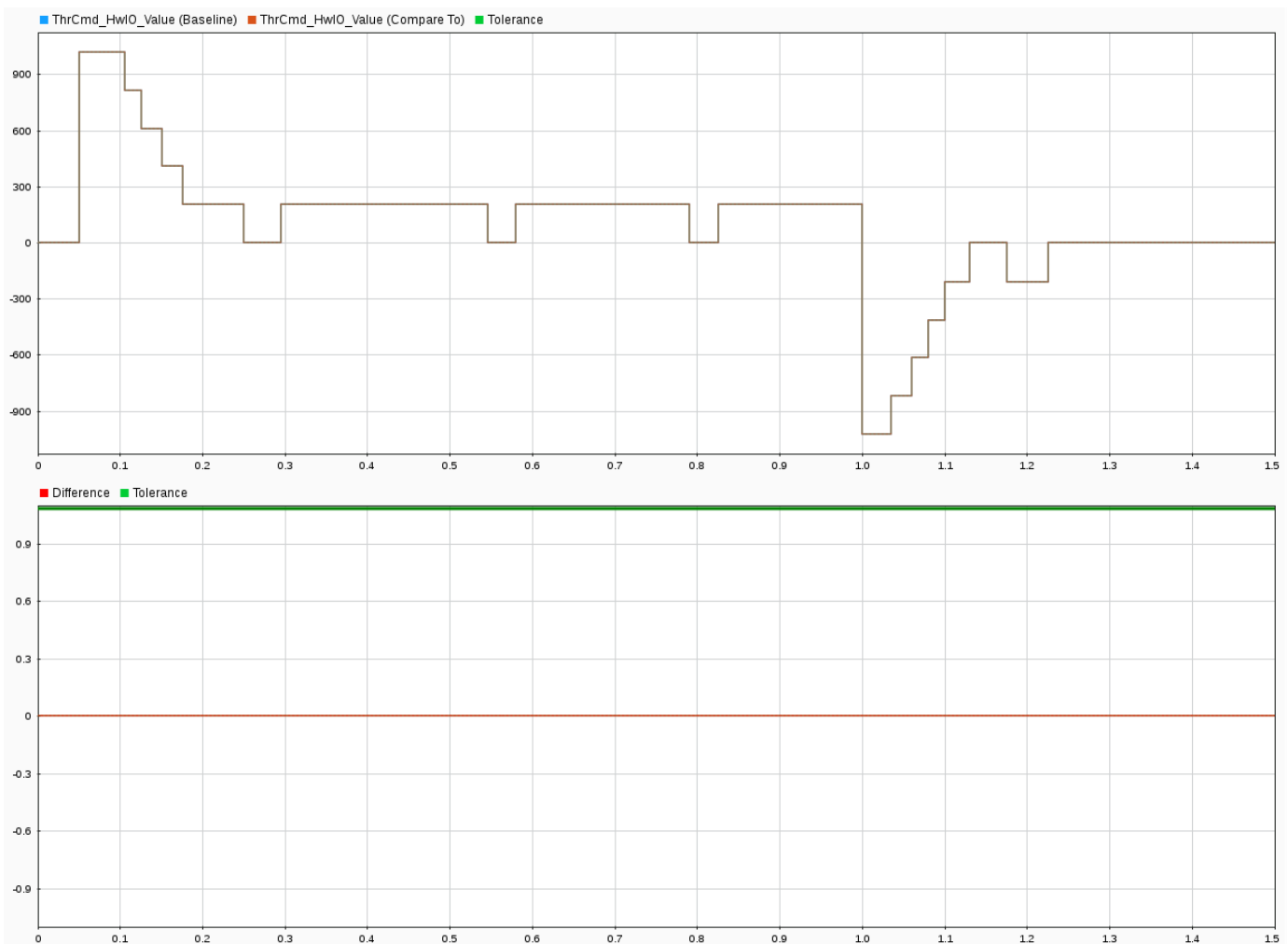
The test manager can be used to lock down simulation behavior and verify equivalence in software-in-the-loop (SIL) mode. Open the test file and run the equivalence test.

```

close_system mdl,0;
file_mldatx = 'sltestThrottlePositionControlTests.mldatx';
open(file_mldatx);
sltest.testmanager.run;

```

The test case verifies the open-loop behavior of the Throttle Position Controller ASWC within the % composition model. The first part of the equivalence test case runs the test harness containing the composition in normal simulation mode. The second part of the test uses the Post-Load callback to switch the Throttle Position Controller ASWC to software-in-the-loop (SIL) mode with Top model code interface. The results of both simulations show that the behavior is equivalent.



Cleanup

```

clear sltestThrottlePositionControlData HBridgeCmd_LkupTbl ...
    SensorSelection SetpointPercent_LkupTbl TPSPPrimaryPercent_LkupTbl...
    TPSSSecondaryPercent_LkupTbl TPSPercent_LkupTbl tout logout mdl file_mldatx;
sltest.testmanager.clear;
sltest.testmanager.clearResults;
sltest.testmanager.close;

```

Automate Testing for Highway Lane Following

This example shows how to assess the functionality of a lane-following application by defining scenarios based on requirements, automating testing of components and the generated code for those components. The components include lane-detection, sensor fusion, decision logic, and controls. This example builds on the “Highway Lane Following” (Automated Driving Toolbox) example.

Introduction

A highway lane-following system steers a vehicle to travel within a marked lane. It also maintains a set velocity or safe distance from a preceding vehicle in the same lane. The system typically includes lane detection, sensor fusion, decision logic, and controls components. System-level simulation is a common technique for assessing functionality of the integrated components. Simulations are configured to test scenarios based on system requirements. Automatically running these simulations enables regression testing to verify system-level functionality.

The “Highway Lane Following” (Automated Driving Toolbox) example showed how to simulate a system-level model for lane-following. This example shows how to automate testing that model against multiple scenarios using Simulink Test™. The scenarios are based on system-level requirements. In this example, you will:

- 1 Review requirements:** The requirements describe system-level test conditions. Simulation test scenarios are created to represent these conditions.
- 2 Review the test bench model:** Review the system-level lane-following test bench model that contains metric assessments. These metric assessments integrate the test bench model with Simulink Test for the automated testing.
- 3 Disable runtime visualizations:** Runtime visualizations are disabled to reduce execution time for the automated testing.
- 4 Automate testing:** A test manager is configured to simulate each test scenario, assess success criteria, and report results. The results are explored dynamically in the test manager and exported to a PDF for external reviewers.
- 5 Automate testing with generated code:** The lane detection, sensor fusion, decision logic, and controls components are configured to generate C++ code. The automated testing is run on the generated code to verify expected behavior.
- 6 Automate testing in parallel:** Overall execution time for running the tests is reduced using parallel computing on a multi-core computer.

Testing the system-level model requires a photorealistic simulation environment. In this example, you enable system-level simulation through integration with the Unreal Engine from Epic Games®. The 3D simulation environment requires a Windows® 64-bit platform.

```
if ~ispc
    error("The 3D simulation environment requires a Windows 64-bit platform");
end
```

To ensure reproducibility of the simulation results, set the random seed.

```
rng(0);
```


Review Requirements

Requirements Toolbox™ lets you author, analyze, and manage requirements within Simulink. This example contains ten test scenarios, with high-level testing requirements defined for each scenario. Open the requirement set.


To explore the test requirements and test bench model, open a working copy of the project example files. MATLAB copies the files to an example folder so that you can edit them. The TestAutomation folder contains the files that enables the automate testing.

```
addpath(fullfile(matlabroot, 'toolbox', 'driving', 'drivingdemos'));
helperDrivingProjectSetup('HighwayLaneFollowing.zip', 'workDir', pwd);
```

```
open('HighwayLaneFollowingTestRequirements.slsreq')
```

Alternatively, you can also open the file from the **Requirements** tab of the Requirements Manager app in Simulink.

The screenshot shows the Requirements Editor interface. On the left, a tree view shows a requirement set named 'HighwayLaneFollowingTestRequirements' containing 10 requirements. Requirement 3, 'scenario_LFACC_03_Curve_StopnGo', is selected. The main pane displays the details for this requirement, including its properties (Type: Functional, Index: 3, Custom ID: 3, Summary: scenario_LFACC_03_Curve_StopnGo) and a table with three columns: Test Description, Target vehicles, and Requirements on Ego vehicle. The Test Description includes a small image of a car on a road and text describing a 'Stop and go test in curved road'. The Target vehicles section lists 'Lead vehicle' and 'Slow moving vehicle - 1' and '2'. The Requirements on Ego vehicle section lists 'Initial velocity: 14m/s', 'Lateral deviation < 0.45m', and 'Time gap > 0.8s'. Below the table, there are fields for Keywords, Revision information, and Links, with a link to 'scenario_LFACC_03_Curve_StopnGo' under the Verified by section.

Test Description	Target vehicles	Requirements on Ego vehicle
<p>Stop and go test in curved road</p> 	<p>Lead vehicle: Initial velocity: 13.6m/s Headway: 50m Event: Lead vehicle travels with initial velocity for 4s, then slows down to 8m/s and after 10s increase its velocity to 13m/s in ego lane.</p> <p>Slow moving vehicle - 1: Set velocity: 8m/s Headway: 1100m Lane: Adjacent to ego lane</p> <p>Slow moving vehicle - 2:</p>	<p>Initial velocity: 14m/s</p> <p>Lateral deviation < 0.45m</p> <p>Time gap > 0.8s</p> <p>Expected behavior: Ego vehicle should first decelerate when lead vehicle slows down and then accelerate when lead vehicle increase its velocity such that it maintains safety time gap from lead vehicle while following the lanes.</p>

Each row in this file specifies the requirements in textual and graphical formats for testing the lane-following system for a test scenario. The scenarios with the **scenario_LF_** prefix enable you to test lane-detection and lane-following algorithms without obstruction by other vehicles. The scenarios with the **scenario_LFACC_** prefix enable you to test lane-detection, lane-following, and ACC behavior with other vehicles on the road.

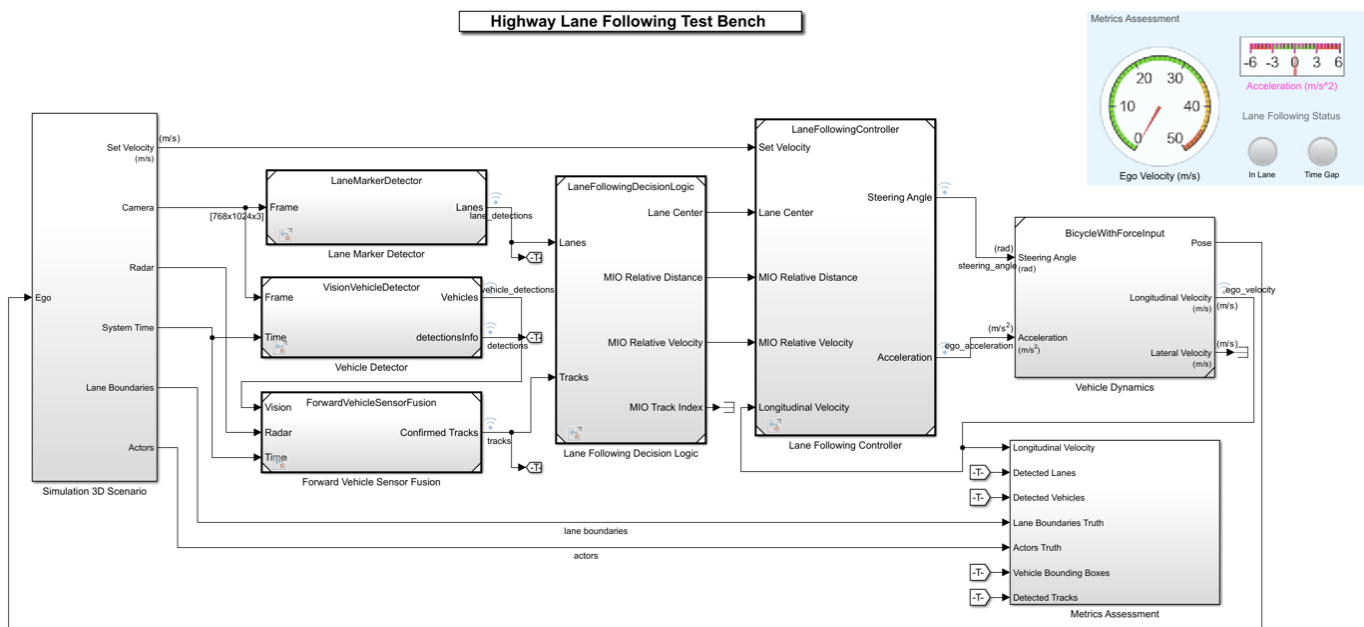
- 1 scenario_LF_01_Straight_RightLane — Straight road scenario with ego vehicle in right lane.
- 2 scenario_LF_02_Straight_LeftLane — Straight road scenario with ego vehicle in left lane.
- 3 scenario_LF_03_Curve_LeftLane — Curved road scenario with ego vehicle in left lane.
- 4 scenario_LF_04_Curve_RightLane — Curved road scenario with ego vehicle in right lane.
- 5 scenario_LFACC_01_Curve_DecelTarget — Curved road scenario with a decelerating lead vehicle in ego lane.
- 6 scenario_LFACC_02_Curve_AutoRetarget — Curved road scenario with changing lead vehicles in ego lane. This scenario tests the ability of the ego vehicle to retarget to a new lead vehicle while driving along a curve.
- 7 scenario_LFACC_03_Curve_StopnGo — Curved road scenario with a lead vehicle slowing down in ego lane.
- 8 scenario_LFACC_04_Curve_CutInOut — Curved road scenario with a fast moving car in the adjacent lane cuts into the ego lane and cuts out from ego lane.
- 9 scenario_LFACC_05_Curve_CutInOut_TooClose — Curved road scenario with a fast moving car in the adjacent lane cuts into the ego lane and cuts out from ego lane aggressively.
- 10 scenario_LFACC_06_Straight_StopandGoLeadCar — Straight road scenario with a lead vehicle that breaks down in ego lane.

These requirements are implemented as test scenarios with the same names as the scenarios used in the HighwayLaneFollowingTestBench model.

Review Test Bench Model

This example reuses the HighwayLaneFollowingTestBench model from the “Highway Lane Following” (Automated Driving Toolbox) example. Open the test bench model.

```
open_system("HighwayLaneFollowingTestBench");
```



Copyright 2019-2023 The MathWorks, Inc.

This test bench model has **Simulation 3D Scenario**, **Lane Marker Detector**, **Vehicle Detector**, **Forward Vehicle Sensor Fusion**, **Lane Following Decision Logic** and **Lane Following Controller** and **Vehicle Dynamics** components.

This test bench model is configured using the `helperSLHighwayLaneFollowingSetup` script. This setup script takes `scenarioName` as input. `scenarioName` can be any one of the previously described test scenarios. To run the setup script, use code:

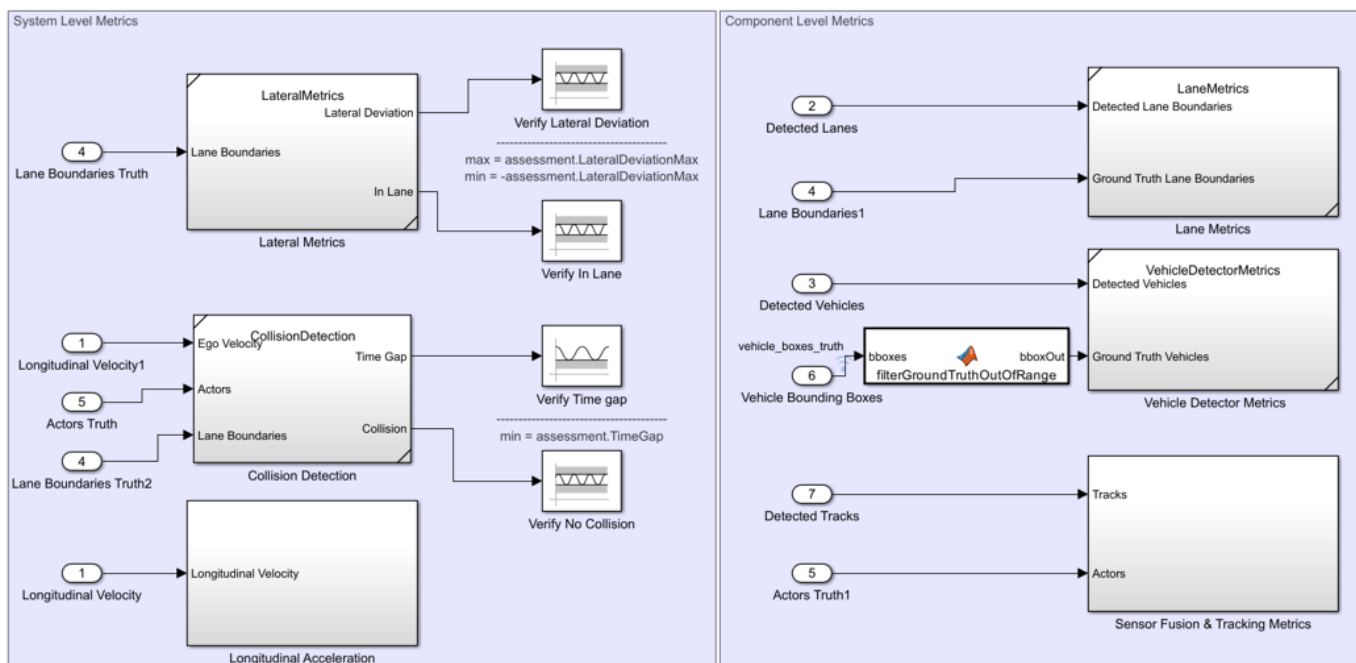
```
scenarioName = "scenario_LFACC_03_Curve_StopnGo";
helperSLHighwayLaneFollowingSetup("scenarioFcnName", scenarioName);
```

You can now simulate the model and visualize the results. For more details on the analysis of the simulation results and the design of individual components in the test bench model, see the “Highway Lane Following” (Automated Driving Toolbox) example.

In this example, the focus is more on automating the simulation runs for this test bench model using Simulink Test for the different test scenarios. The **Metrics Assessment** subsystem enables integration of system-level metric evaluations with Simulink Test. Open the **Metrics Assessment** subsystem.

```
open_system("HighwayLaneFollowingTestBench/Metrics Assessment");
```

Metrics Assessment



Using this example, you can evaluate the system-level behavior using four system-level metrics. Additionally, you can also compute component-level metrics to analyze individual components and their impact on the overall system performance.

System-Level Metrics

- **Verify Lateral Deviation** — This block verifies that the lateral deviation from the center line of the lane is within prescribed thresholds for the corresponding scenario. Define the thresholds when you author the test scenario.
- **Verify In Lane** — This block verifies that the ego vehicle is following one of the lanes on the road throughout the simulation.
- **Verify Time gap** — This block verifies that the time gap between the ego vehicle and the lead vehicle is more than 0.8 seconds. The time gap between the two vehicles is defined as the ratio of the calculated headway distance to the ego vehicle velocity.
- **Verify No Collision** — This block verifies that the ego vehicle does not collide with the lead vehicle at any point during the simulation.

Component-Level Metrics

- **Lane Metrics** — This block verifies that distances between the detected lane boundaries and the ground truth data are within the thresholds specified in a test scenario.
- **Vehicle Detector Metrics** — This blocks computes and logs true positives, false negatives, and false positives for the detections.
- **Sensor Fusion & Tracking Metrics** — This subsystem computes generalized optimal subpattern assignment (GOSPA) metric, localization error, missed target error, and false track error. For more information on these metrics, see “Forward Vehicle Sensor Fusion” (Automated Driving Toolbox) example.

Disable Runtime Visualizations

The system-level test bench model visualizes intermediate outputs during the simulation for the analysis of different components in the model. These visualizations are not required when the tests are automated. You can reduce execution time for the automated testing by disabling them.

Disable runtime visualizations for the **Lane Marker Detector** subsystem.

```
load_system('LaneMarkerDetector');
blk = 'LaneMarkerDetector/Lane Marker Detector';
set_param(blk, 'EnableDisplays', 'off');
```

Disable runtime visualizations for the **Vehicle Detector** subsystem.

```
load_system('VisionVehicleDetector');
blk = 'VisionVehicleDetector/Pack Detections/Pack Vehicle Detections';
set_param(blk, 'EnableDisplay', 'off');
```

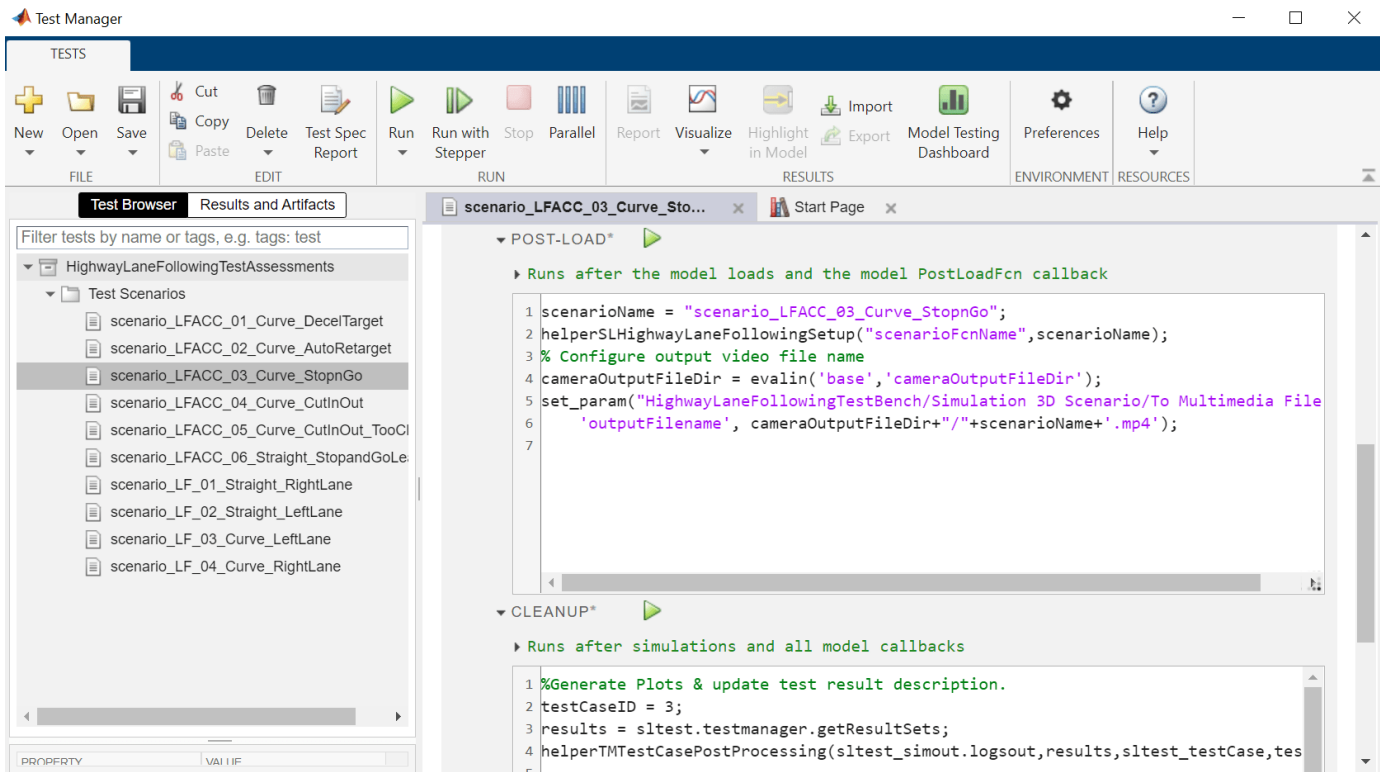
Configure the Simulation 3D Scene Configuration (Automated Driving Toolbox) block to run the Unreal Engine in headless mode, where the 3D simulation window is disabled.

```
blk = ['HighwayLaneFollowingTestBench/Simulation 3D Scenario/', ...
      'Simulation 3D Scene Configuration'];
set_param(blk, 'EnableWindow', 'off');
```

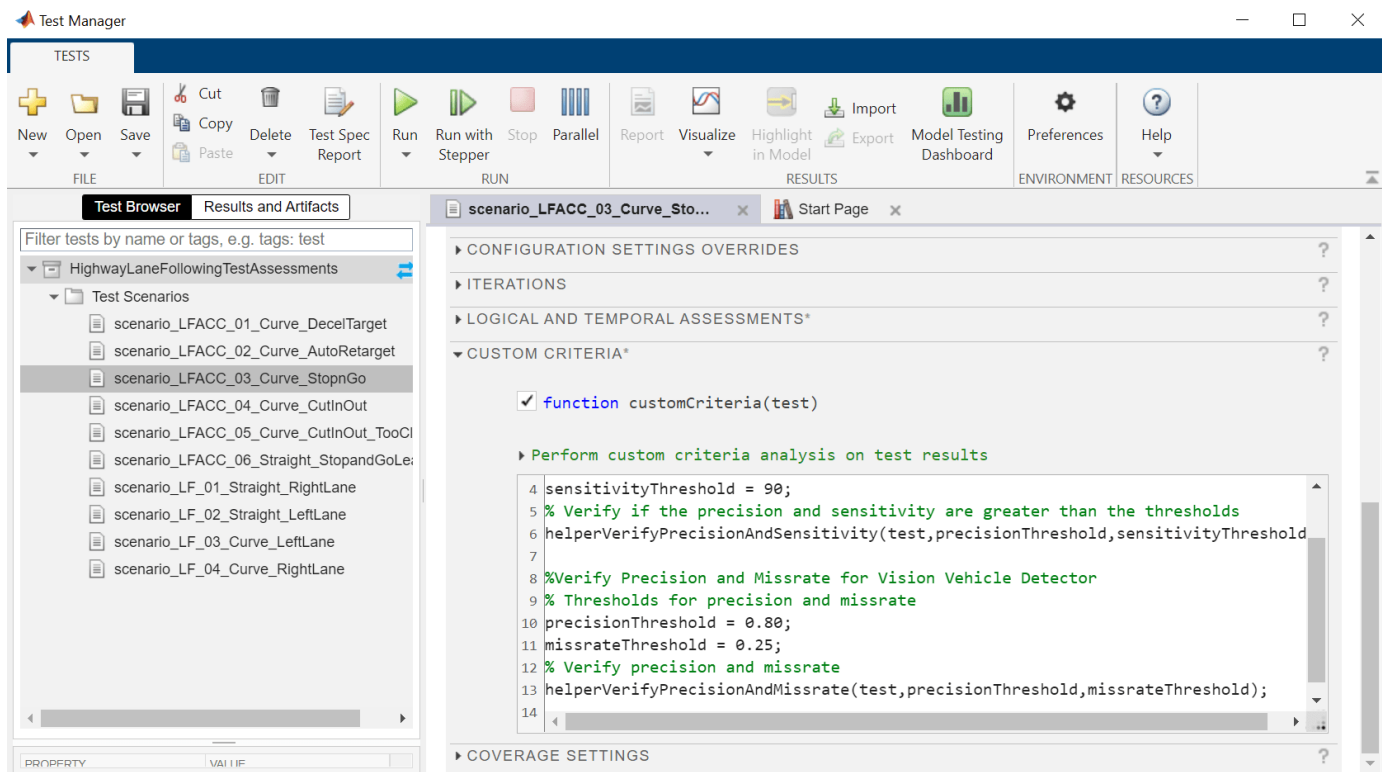
Automate Testing

The Test Manager is configured to automate the testing of the lane-following application. Open the HighwayLaneFollowingTestAssessments.mldatx test file in the Test Manager.

```
sltestmgr;
testFile = sltest.testmanager.load('HighwayLaneFollowingTestAssessments.mldatx');
```



Observe the populated test cases that were authored previously in this file. Each test case is linked to the corresponding requirement in the Requirements Editor for traceability. Each test case uses the POST-LOAD callback to run the setup script with appropriate inputs and to configure the output video filename. After the simulation of the test case, it invokes helperTMTTestCasePostProcessing from the CLEAN-UP callback to assess performances of the overall system and individual components by generating the plots explained in the “Highway Lane Following” (Automated Driving Toolbox) example.



After simulation of the test case, Simulink Test also invokes these functions from the CUSTOM CRITERIA callback to get additional metrics for lane marker detector and vehicle detector components:

- `helperVerifyPrecisionAndSensitivity` — Verifies that the precision and sensitivity metrics of the lane marker detector component are within the predefined threshold limit.
- `helperVerifyPrecisionAndMissrate` — Verifies that the precision and miss rate metrics of the vehicle detector component are within the predefined threshold limit.

Run and explore results for a single test scenario:

To reduce command-window output, turn off the MPC update messages.

```
mpcverbosity('off');
```

To test the system-level model with the `scenario_LFACC_03_Curve_StopnGo` test scenario from Simulink Test, use this code:

```
testSuite = getTestSuiteByName(testFile, 'Test Scenarios');
testCase = getTestCaseByName(testSuite, 'scenario_LFACC_03_Curve_StopnGo');
resultObj = run(testCase);
```

To generate a report after the simulation, use this code:

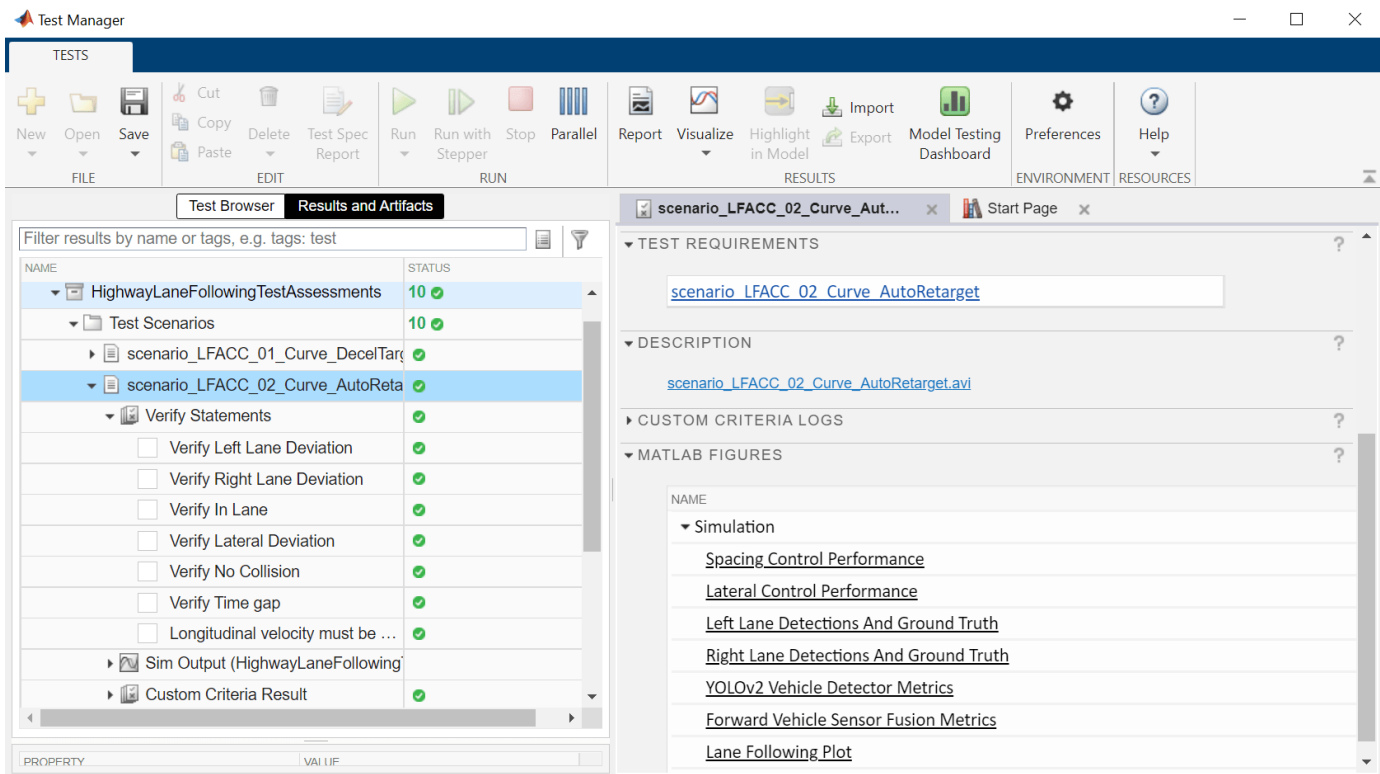
```
sltest.testmanager.report(resultObj, 'Report.pdf', ...,
    'Title', 'Highway Lane Following', ...,
    'IncludeMATLABFigures', true, ...,
    'IncludeErrorMessage', true, ...,
    'IncludeTestResults', 0, 'LaunchReport', true);
```

Examine the report `Report.pdf`. Observe that the **Test environment** section shows the platform on which the test is run and the MATLAB® version used for testing. The **Summary** section shows the outcome of the test and duration of the simulation in seconds. The **Results** section shows pass/fail results based on the assessment criteria. This section also shows the plots logged from the `helperGenerateFilesForLaneFollowingReport` function.

Run and explore results for all test scenarios:




















You can simulate the system for all the tests by using `sltest.testmanager.run`. Alternatively, you can simulate the system by clicking **Play** in the Test Manager app.

After completion of the test simulations, the results for all the tests can be viewed in the **Results and Artifacts** tab of the Test Manager. For each test case, the Check Static Range blocks in the model are associated with the Test Manager to visualize overall pass/fail results.



You can find the generated report in current working directory. This report contains a detailed summary of pass/fail statuses and plots for each test case.


Summary

Name	Outcome	Duration (Seconds)
 HighwayLaneFollowingTestAssessments	10 	2149.277
 Test Scenarios	10 	2149.278
 scenario LFACC 01 Curve DecelTarget		448.803
 scenario LFACC 02 Curve AutoRetarget		218.906
 scenario LFACC 03 Curve StopnGo		268.746
 scenario LFACC 04 Curve CutInOut		200.885
 scenario LFACC 05 Curve CutInOut TooClose		237.335
 scenario LFACC 06 Straight StopandGoLeadCar		128.585
 scenario LF 01 Straight RightLane		157.953
 scenario LF 02 Straight LeftLane		145.923
 scenario LF 03 Curve LeftLane		177.465
 scenario LF 04 Curve RightLane		164.469

Verify test status in Requirements Editor:

Open the Requirements Editor and select **Display**. Then, select **Verification Status** to see a verification status summary for each requirement. Green and red bars indicate the pass/fail status of simulation results for each test.

The screenshot shows the Requirements Editor interface. The left pane lists 10 requirements under the 'HighwayLaneFollow...' category. Requirement 3, 'scenario_LFACC_03_Curve_StopnGo', is selected. The right pane shows the details for this requirement, including its type (Functional), index (3), and summary. A table titled 'Test Description' provides details for a 'Stop and go test in curved road' scenario, including target vehicle parameters and requirements on the ego vehicle.

Test Description	Target vehicles	Requirements on Ego vehicle
<p>Stop and go test in curved road</p> 	<p>Lead vehicle: Initial velocity: 13.6m/s Headway: 50m Event: Lead vehicle travels with initial velocity for 4s, then slows down to 8m/s and after 10s increase its velocity to 13m/s in ego lane.</p> <p>Slow moving vehicle - 1: Set velocity: 8m/s Headway: 1100m Lane: Adjacent to ego lane</p> <p>Slow moving vehicle - 2:</p>	<p>Initial velocity: 14m/s</p> <p>Lateral deviation < 0.45m</p> <p>Time gap > 0.8s</p> <p>Expected behavior: Ego vehicle should first decelerate when lead vehicle slows down and then accelerate when lead vehicle increase its velocity such that it maintains safety time gap from lead vehicle while following the lanes.</p>

Automate Testing with Generated Code

The HighwayLaneFollowingTestBench model enables integrated testing of **Lane Marker Detector**, **Vehicle Detector**, **Forward Vehicle Sensor Fusion**, **Lane Following Decision Logic**, and **Lane Following Controller** components. It is often helpful to perform regression testing of these components through software-in-the-loop (SIL) verification. If you have Embedded Coder™ Simulink Coder™ license, then you can generate code for these components. This workflow lets you verify that the generated code produces expected results that match the system-level requirements throughout simulation.

Set **Lane Marker Detector** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Lane Marker Detector';
set_param(model, 'SimulationMode', 'Software-in-the-loop');
```

Set **Vehicle Detector** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Vehicle Detector';
set_param(model, 'SimulationMode', 'Software-in-the-loop');
```

Set **Forward Vehicle Sensor Fusion** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Forward Vehicle Sensor Fusion';  
set_param(model, 'SimulationMode', 'Software-in-the-loop');
```

Set **Lane Following Decision Logic** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Lane Following Decision Logic';  
set_param(model, 'SimulationMode', 'Software-in-the-loop');
```

Set **Lane Following Controller** to run in Software-in-the-loop mode.

```
model = 'HighwayLaneFollowingTestBench/Lane Following Controller';  
set_param(model, 'SimulationMode', 'Software-in-the-loop');
```

Now, run `sltest.testmanager.run` to simulate the system for all the test scenarios. After the completion of tests, review the plots and results in the generated report.

Enable the MPC update messages again.

```
mpcverbosity('on');
```

Automate Testing in Parallel

If you have a Parallel Computing Toolbox™ license, then you can configure Test Manager to execute tests in parallel using a parallel pool. To run tests in parallel, save the models after disabling the runtime visualizations using `save_system('LaneMarkerDetector')`, `save_system('VisionVehicleDetector')` and `save_system('HighwayLaneFollowingTestBench')`. Test Manager uses the default Parallel Computing Toolbox cluster and executes tests only on the local machine. Running tests in parallel can speed up execution and decrease the amount of time it takes to get test results. For more information on how to configure tests in parallel from the Test Manager, see “Run Tests Using Parallel Execution” on page 6-151.


Synchronize Tests

If you change the system under test, you can synchronize the test cases to reflect the model changes. Also, if you remove model components, you can disable or delete test cases in the Test Manager when you synchronize.

Synchronizing your test file automatically creates a new test case for:

- Each new scenario in the Signal Editor block at the top level of your model and the top level of each test harness. The model must have only one Signal Editor block at those levels to create a test case.
- Each new test harness in the model.

To synchronize your test file:

- 1** In the Test Manager **Test Browser** pane, hover over the test filename that you want to update.
- 2** Click the synchronization button  next to the test filename.
- 3** Follow the prompts to specify:
 - The type of test file to create for the new components
 - Whether to disable or delete out-of-date components

Disabled tests appear in the list in italic.

See Also

More About

- “Automatically Create a Set of Test Cases” on page 6-19

Use External Excel or MAT-File Data in Test Cases

In this section...

“Data Mapping” on page 6-72

“Create a Test Case from an Excel Spreadsheet” on page 6-73

“Import an Excel Spreadsheet into an Existing Test Case” on page 6-74

“Add Multiple Microsoft Excel Spreadsheets as Input to a Test Case” on page 6-75

“Include Microsoft Excel Test Data in Test Results” on page 6-75

“Importing Test Data from Microsoft Excel” on page 6-75

“Add a MAT-File as an External Input” on page 6-78

Test cases can use data defined in external MAT-files or Microsoft® Excel files. For information about the Excel file format, see “Format Test Case Data in Excel” on page 6-83.

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.

Data Mapping

Mapping Modes

To use external data, you map the data to your model (system under test [SUT]) using these mapping modes:

- The names of the inport block the signal data corresponds to
- The full block pathname, in the form `system/block`
- The name of the signal associated with the inport block
- The port number, that is, sequential port numbers of the inport blocks, starting at 1

For more information about how Simulink handles inport mapping, see “Map Root Inport Signal Data”.

Mapping Status

When you map external inputs to model elements, the mapping creates these possible results. These results appear under **Inputs** in the Test Manager interface in the **Status** column:

- Mapped — The mapping succeeded and no further action is needed.
- Failed — The mapping failed. Click the **Failed** link for more information.
- Warning — The mapping occurred with warnings. Click the **Warning** link to see whether you need to address them
- Stale — This status can occur when you update your external inputs in Test Manager. A stale state occurs if you did not map the new inputs. To address this status, click the **Status** link, which opens the Add Input dialog box. Click **Map Inputs** to map the new input data and then click **Add**.

Create a Test Case from an Excel Spreadsheet

This example shows how to import an Excel Spreadsheet to define a test case.

1. Open Simulink Test Manager.
2. Select **New > Test from Spreadsheet**. This opens the **Create Test from Spreadsheet** Wizard.
3. Select **Use existing test data from a spreadsheet**. Then, click on the folder icon and select `coordinate_text.xlsx` as the spreadsheet. Click **Next**.
4. Enter `coordinate_transform_test` as the model. Click **Next**.
5. On the Attributes page, check that the attribute categories in the spreadsheet are displayed.
6. Click **Validate** to map each input to the model by block name. When the attributes are validated, the icons change to green. If necessary, change the spreadsheet and/or system under test, click **Refresh**, and validate again. Click **Next**.
7. Specify a location to save the test file and click **Done**.

Simulink Test creates a new test case and imports the spreadsheet. The fields defined in the spreadsheet are locked to the spreadsheet, and cannot be edited in the Test Manager. To change the locked fields, edit the spreadsheet outside of MATLAB.

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			

If you cannot see all the data in a column, click + in the upper right corner to hide other columns and resize the desired column.

For multi-dimensional signals, each dimension is represented in a separate column in the spreadsheet. By default, only the dimensions with non-zero values are included. If all dimensions have zero value, then only the last dimension is included in the spreadsheet.

Import an Excel Spreadsheet into an Existing Test Case

If you have a test case and want to add test data to it from an Excel spreadsheet, you must associate the test case 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. Selecting this option again causes values in the spreadsheet to overwrite the values in the test.

Add Multiple Microsoft Excel Spreadsheets as Input to a Test Case

You can import multiple Microsoft Excel spreadsheets 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.

- 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”.

- 7 Click **Add**.

Include Microsoft Excel Test Data in Test Results

- 1 In the test case, expand the **Inputs** section and click **Include input data in test result**.
- 2 Under the **External Inputs** table, click **Add**.
- 3 In the Add Input dialog box, specify the Excel filename and the mapping mode, which specifies how to map the Excel data to root-level Inport blocks in the model.
- 4 Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.
- 5 Click **Add**.

See “Importing Test Data from Microsoft Excel” on page 6-75 for a complete example.

Importing Test Data from Microsoft Excel

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®.

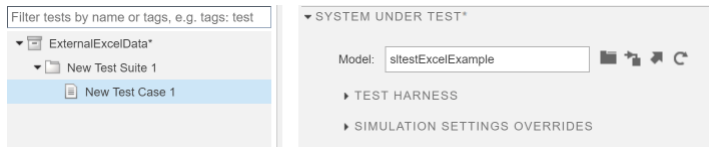
Create a Test File

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 input 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 current directory. This file contains two tabs, named **Acceleration** and **Braking**. Each tab represents a complete set of inputs for a single simulation.
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/OR RANGE SPECIFICATION

▼ INPUT MAPPING

Mapping Mode: ▼

Compile the system under test

▼ MAPPING STATUS

Successfully mapped inputs.

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	sltestExcelExample/Throttle	Throttle	✓
2	sltestExcelExample/Brake	BrakeTorque	✓

▶ ADVANCED

The **Mapping Mode** controls the method used to map data from the Microsoft Excel sheet to root-level Inport blocks in the model. For more information, see “Use External Excel or MAT-File Data in Test Cases” on page 6-72.

The test case shows the inputs mapped.

▼ INPUTS*

Include input data in test result

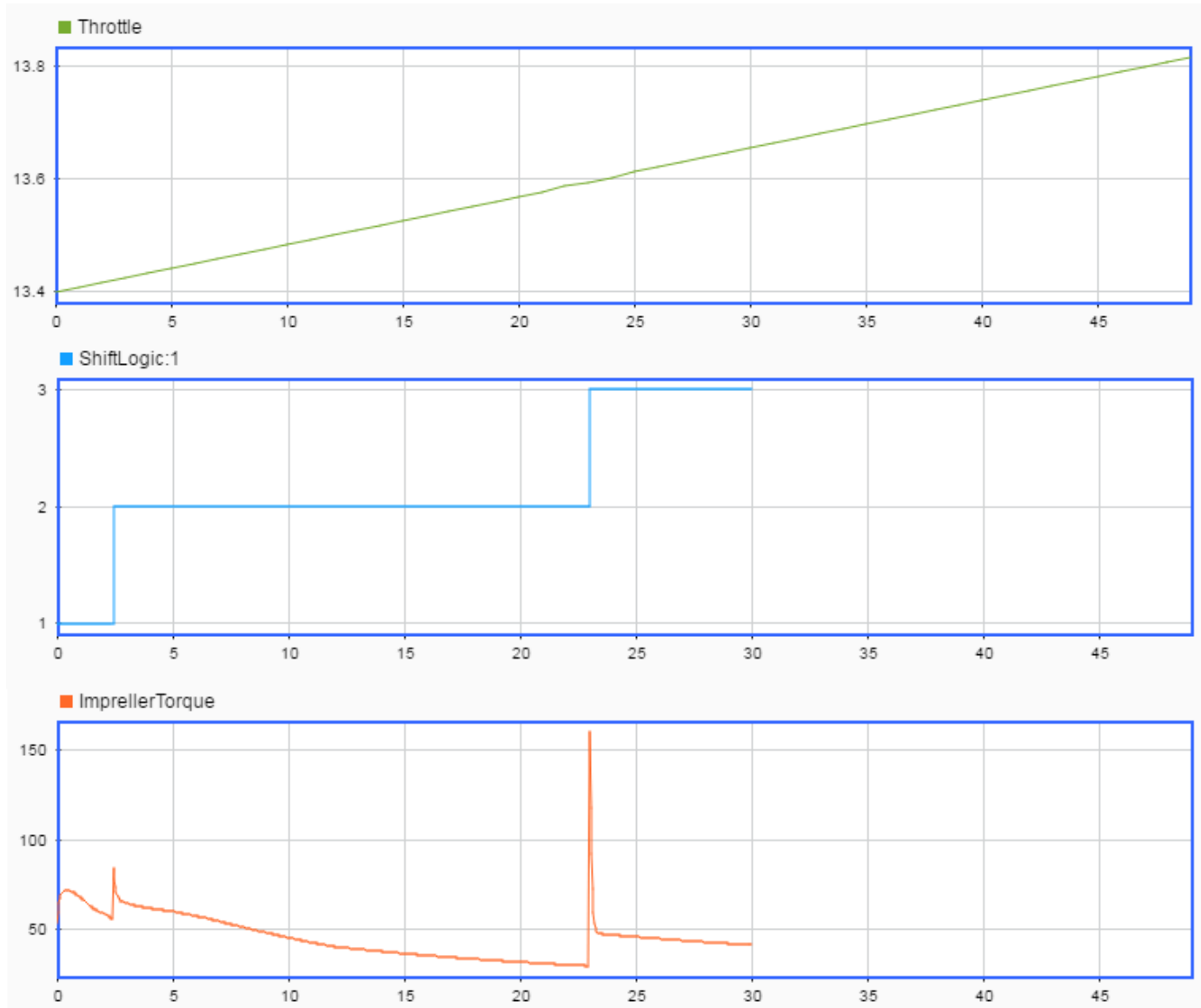
Stop simulation at last time point

EXTERNAL INPUTS

NAME	FILE	SHEET	STATUS
<input checked="" type="checkbox"/> Acceleration	M:	Acceleration	Mapped

Run the Test

1. In the toolbar, click **Run**.
2. In the **Results and Artifacts** pane, you can plot signals from the external inputs or the simulation output.



Add a MAT-File as an External Input

- 1 In the test case, expand the **Inputs** section and click **Add**.
- 2 Browse to the MAT-file and click **Add**.
- 3 Under **Input Mapping**, choose a mapping mode.
- 4 Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.

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

5 Click **Add**.

See Also

`sltest.testmanager.TestInput` | `sltest.io.SimulinkTestSpreadsheet`

More About

- “Map Signal Data to Root Input Ports”
- “Map Root Inport Signal Data”
- “Create Data Files for Test Case Input” on page 6-80

Create Data Files for Test Case Input

You can use Test Manager to create MAT-file and Microsoft Excel data files to use as inputs to test cases. You generate a template that contains the signal names and the times, and then enter the data.

Creating a data file also adds the file to the list of available input files for the test case. After you add input data, you can then select the file to use in your test case.

You can create files for input data only for tests that run in the current release. To select the release, in the test case, use the **Select releases for simulation** list.

You can edit input files. After you create the template, select the file from the list of input files and click **Edit**. MAT-files open in the Signal Editor. Excel files open in Excel.

Selecting the **Add an iteration that runs this input** check box adds an iteration to the test case under **Table Iterations** and assigns the input file to it. After you create the input file, continue to specify the iteration. For more information on iterations, see “Test Iterations” on page 6-125.

Generate an Excel Template

You can generate a template test spreadsheet from a model or harness (system under test [SUT]). You can then complete the spreadsheet with external data and import it into Simulink Test as a test case.

The Create Test from Spreadsheet wizard parses the SUT for test attributes and automatically generates a template spreadsheet and a test case:

- Inputs — Inputs are characterized by root input ports
- Parameters — Named parameters in the model
- Comparison signals — Logged signals and output ports

The wizard allows you to filter and edit the attributes needed for testing. The resulting spreadsheet has separate column sets for inputs, parameters, and comparison signals. If multiple iterations are required, a separate sheet in the same file is generated for each scenario. You can expand the spreadsheet to add time-based signal data, tolerances, and parameter overrides. See “Format Test Case Data in Excel” on page 6-83 for the full description of the format readable by Simulink Test.

You can use the `coordinate_transform_test` model as an example for the process. Its associated Excel file is `coordinate_test.xlsx`.

- 1 Open the test manager. On the **Apps** tab, under Model Verification, Validation, and Test, click **Simulink Test**. Then, on the **Tests** tab, click **Simulink Test Manager**.
- 2 Open the wizard. From Simulink Test Manager, select **New > Test from Spreadsheet**. Select **Create a test template file for specifying data** and follow the prompts.
- 3 In the **Attributes** page, select which attribute categories are to be included in the spreadsheet. For example, if parameter overrides are not necessary for the tests, clear Parameters. The attribute categories shown on the page are derived from the SUT. Comparison signals are always shown.
- 4 If the test requires all attributes in a category as is, select **Yes, include all attributes in the spreadsheet** and click **Next**. If not, select **No, I want to filter and edit the attributes**. This shows a page with a tab for each attribute category.

- 5 If you are filtering the attributes, in the **Parameters** and **Comparison** tabs, clear the attributes that are not needed. For example, you can remove a logged signal from this list if it is not to be used for comparison in the tests.
- 6 Optionally change tolerances in the **Comparison** page. The tolerance settings apply to all signals in the list. To specify different tolerances for each signal, edit the spreadsheet after it is generated.

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:

If you change the SUT during the selection process, click **Refresh** to synchronize the attribute lists with the SUT. Once selection is complete, click **Next** and keep following the prompts.

- 7 In the **Scenarios** page, specify the number of test scenarios and a base name for the sheets in the spreadsheet.

If comparison signals are selected, the wizard runs the model to capture the baseline. Make sure that the model does not run indefinitely by setting a finite stop time. The wizard creates two files:

- Excel spreadsheet — The spreadsheet includes columns for inputs, parameters, and comparison signals. Inputs and comparisons have different time bases. An identical sheet for each test scenario is generated. Complete the spreadsheet outside MATLAB to uniquely define each scenario.

time	Magnitude	Angle	Parameter:	Value:	time	point.pixels_x	point.pixels_y
						AbsTol: 0.001	AbsTol: 0.001
						RelTol: 0.1	RelTol: 0.1
						BlockPath: coordinate_transform/Screen coordinates	BlockPath: coordinate_transform/Screen coordinates
			Source: Input			Source: Output	
0	0	0	Xscale	1	0	0	0
10	0	0	Yscale	1	0.2	0	0
					0.4	0	0
					0.6	0	0
					0.8	0	0

- Test file — The test case imports the Excel spreadsheet. The fields defined in the spreadsheet are locked to the spreadsheet, and cannot be edited in the Test Manager.

New Test Case 1

Enabled

[coordinate_test_signals](#) » [New Test Suite 1](#) » [New Test Case 1](#)

Baseline Test

Select releases for simulation:

Create Test Case from External File

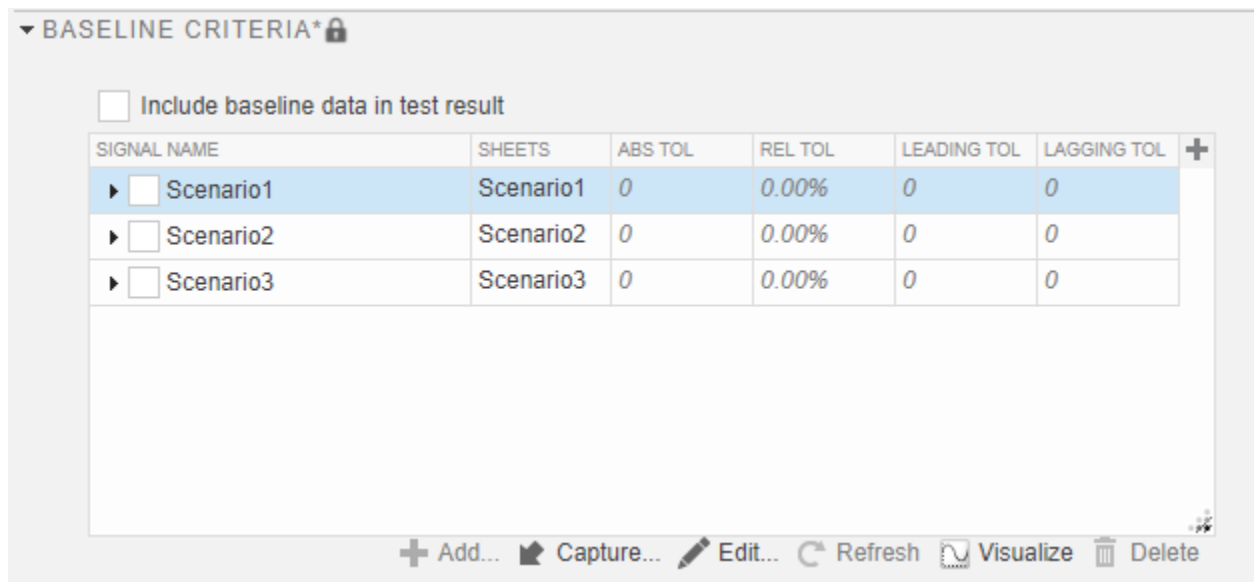
File:  

▶ 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 MATLAB. If you change a parameter, you must capture the baseline again by clicking the **Capture** button.



Format Test Case Data in Excel

You can specify signal data in a Microsoft Excel file to use as input to your test case or as baseline criteria (outputs). The Excel file includes time and signal data. To support a range of models and configurations, you can specify signal data of most data types. 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.

Note For information on how to format data in Excel files, see “Microsoft Excel Import, Export, and Logging Format”.

Additional information specific to test cases includes:

- Importing the file as input data — Use the **Inputs** section of the test case, described in “Use External Excel or MAT-File Data in Test Cases” on page 6-72.
- Using the Excel file as expected outputs — Select the file to add it as baseline data, in the **Baseline Criteria** section of the test case, described in “Baseline Criteria” on page 6-167. For more information, see “Multiple Runs”.
- Capturing inputs and expected outputs in Test Manager — 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 6-167.
- Specifying tolerances — See “Compare Model Output to Baseline Data” on page 6-7.

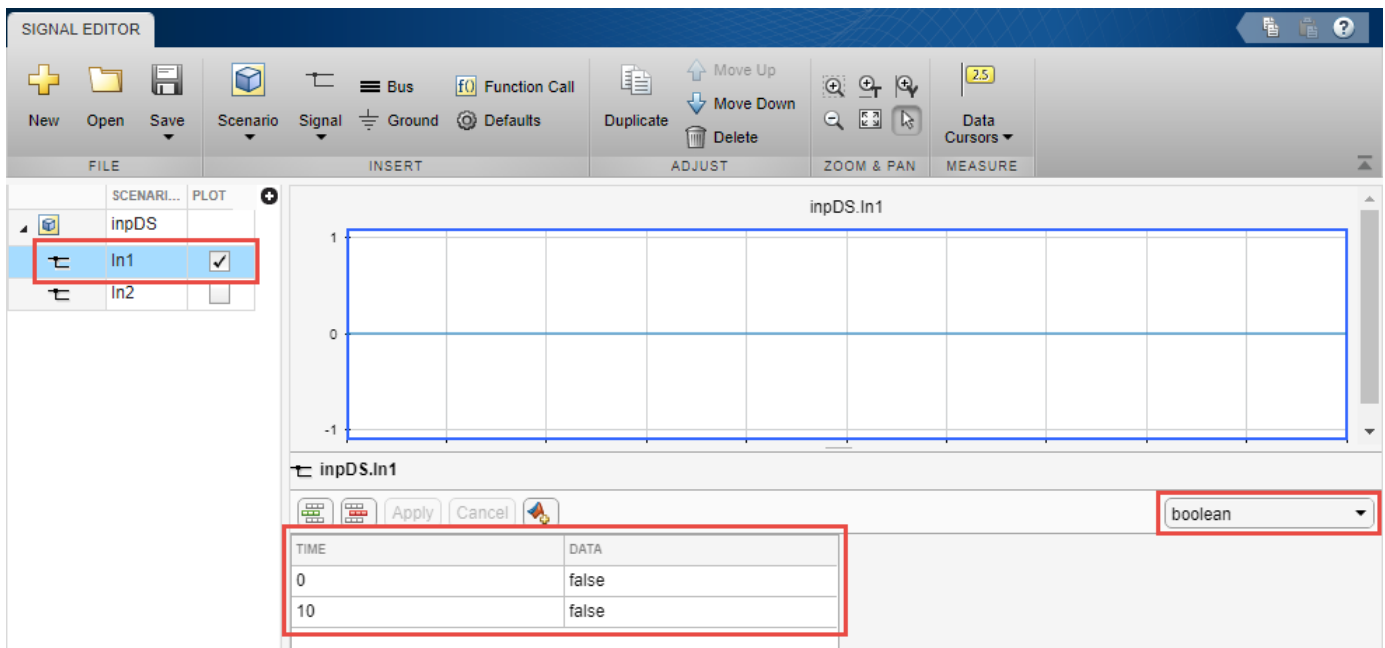
Create a MAT-File for Input Data

- 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

[sltest.testmanager.BaselineCriteria](#) | [sltest.testmanager.TestInput](#)

More About

- “Use External Excel or MAT-File Data in Test Cases” on page 6-72
- “Simulation Settings and Release Overrides” on page 6-161
- “Baseline Criteria” on page 6-167

Capture Simulation Data in a Test Case

Capture signal data in your test results by adding signals to the **Simulation Outputs** section of the test case. Each output is called a logged signal. Signals listed in **Simulation Outputs** appear in the test results along with signals that are already selected for logging in Simulink. You can also log a portion of a signal by using conditional or duration triggers to start and stop logging.

You can use logged signals for data comparison in baseline criteria, equivalence criteria, custom criteria, and for data visualization in the Simulation Data Inspector. Logged signals enable you to further test your Simulink model without changing the model. In addition to signals from the top model, you can also log signals from subsystems and model references. You can select signals associated with local and global data store memory, and from data store memory that uses a `Simulink.Signal` object.

Add Logged Signals When Creating a Test Harness

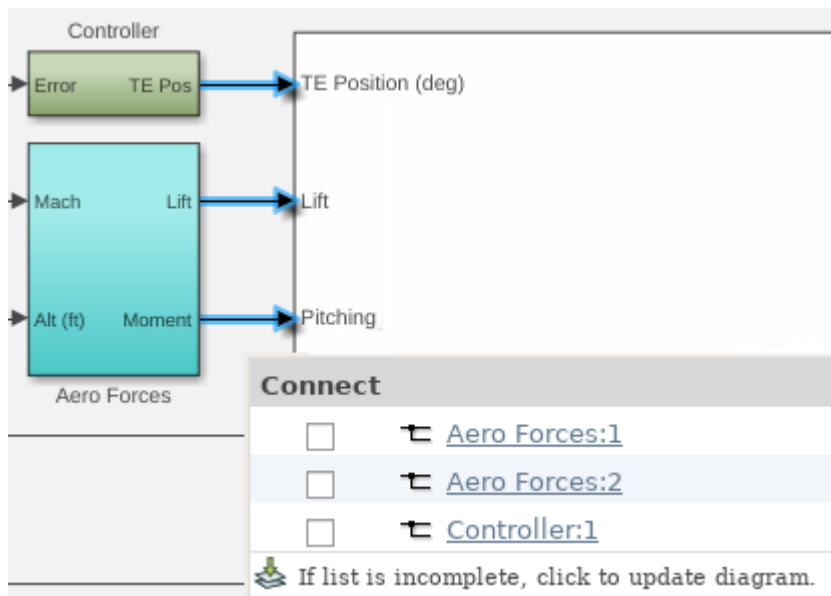
When you right-click a model or model component and click **Test Harness > Create for <Model or Model Element>**, the Create Test Harness dialog box opens. To log all output signals of the component under test, select **Log Output Signals**, which is on the **Basic Properties** tab of the Create New Harness dialog for Subsystem block diagram and library harnesses. For all other types of harnesses, the option is on the **Advanced Properties** tab. Signals that are not compatible with logging do not have a logging badge attached to them. The compatibility is determined at compile time. After the harness is created, you can turn off logging for a signal by opening the harness, right-clicking the signal, and selecting **Stop Logging Selected Signals**. If a signal does not have a name, it is assigned one for test execution using the format `<component under test name>:<output port number>`. Unnamed propagated signals use the name of the signal from which they propagate. To log all test harness signals programmatically, include `'LogHarnessOutputs'`, `true` as an input to `sltest.harness.create`.

Add Logged Signals in the Test Manager

To add signals:

- 1 Open the model `sltestFlutterSuppressionSystemExample`.

```
openExample('sltestFlutterSuppressionSystemExample')
```
- 2 Use `sltest.testmanager.view` to open the Test Manager.
- 3 Click **Open** and open the `sltestFlutterCriteriaTest.mldatx` test file.
- 4 In the Test Manager, under **Simulation Outputs**, click **Add**.
- 5 In the system under test, highlight blocks or signals that you want to log. To select multiple items, click and drag a selection box over multiple items.
- 6 A dialog box appears. Select signals in the dialog box.



- 7 Continue adding signals to the test case. Each time you select a signal, the dialog box also shows previously logged signals. You can remove a signal from logging by clearing the selection.
- 8 When you finish adding signals, return to the Test Manager and click **Done**.
- 9 The signals appear in the **Logged Signals** table in the test case.

LOGGED SIGNALS			
NAME	SOURCE	PORT INDEX	PLOT INDEX
<input checked="" type="checkbox"/> Signal Set 1			
<input checked="" type="checkbox"/> Aero Forces:1	sltestFlutterSuppression...	1	
<input checked="" type="checkbox"/> Aero Forces:2	sltestFlutterSuppression...	2	
<input checked="" type="checkbox"/> Controller:1	sltestFlutterSuppression...	1	

Plot signals on the specified plots after simulation

To add a signal set, click the **Add** arrow and select **Signal Set**.

To specify a specific plot for a signal, enter a number in the **Plot Index** column. By default, the signals appear on one plot.

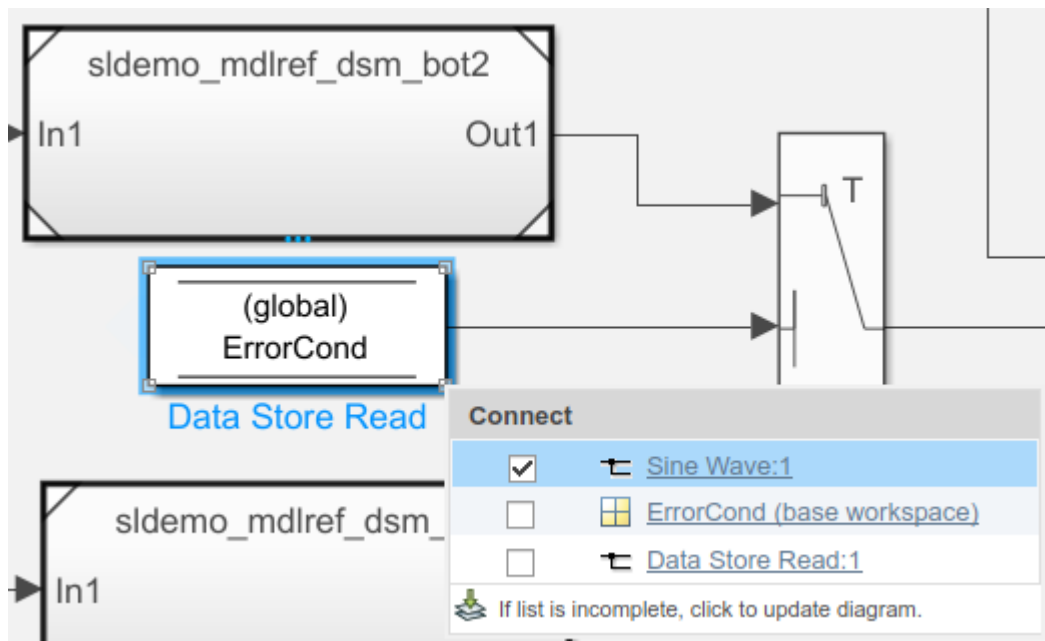
You can specify to display the plot immediately after running the test by selecting the **Plot signals on the specified plots after simulation** check box.

After you run the test, the logged signals appear in the test case results under **Sim Output**. Select each signal to display on the plot. If you specify a plot index, the signal appears in the plot number you specified.

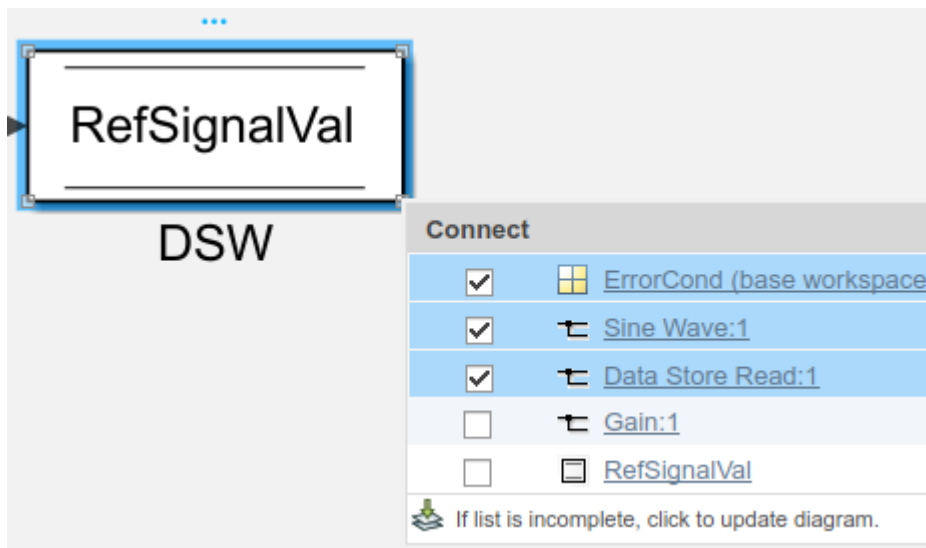
Capture Data from Local and Global Data Stores

Perform similar steps to add simulation output associated with data store memory:

- 1 Use `openExample('sldemo_mdref_dsm')` to open the model, which contains local and global data store memory.
- 2 Use `sltest.testmanager.view` to open the Test Manager.
- 3 Click **New > Create File from ModelNew** to create a new test file.
- 4 Go to the top model, select the Sine Wave block signal and click **Log Selected Signal**. Mark the Data Store Read (ErrorCond) block signal for logging, too.
- 5 Select the Data Store Read block in the top model. In the **Modeling** tab, click **Update Model**. The model displays the signal storage class for the block, (`global`).
- 6 In the Test Manager, select New Test Case 1 and expand the **Simulation Outputs** section. Click **Add** in the lower right of **Logged Signals**.
- 7 In the model, select the Sine Wave block and in the Connect dialog box, select Sine Wave:1. Then, select the Data Store Read block and in the dialog box, select Data Store Read: 1 and ErrorCond (base workspace).



- 8 In the model, double-click `sldemo_mdref_dsm_bot` to open it. Then, open the subsystem `PositiveSS`. Select the Data Store Write block. The dialog box displays the input signal from the Gain block and the data store memory, `RefSignalVal`.



- 9 Select the RefSignalVal data store memory for logging.
- 10 Finish selecting signals by clicking **Done** in the Test Manager window. In the Test manager, the signals appear under **Logged Signals**. The **Source** column displays the full path information for each signal.

LOGGED SIGNALS			
NAME	SOURCE	PORT INDEX	
<input checked="" type="checkbox"/> Signal Set 1			
<input checked="" type="checkbox"/> Data Store Read:1	sldemo_mdref_dsm/Data Store Read	1	
<input checked="" type="checkbox"/> ErrorCond	base workspace		
<input checked="" type="checkbox"/> RefSignalVal	sldemo_mdref_dsm/A/DSM		
<input checked="" type="checkbox"/> Sine Wave:1	sldemo_mdref_dsm/Sine Wave	1	

Plot signals on the specified plots after simulation

Log Leaf Signals of a Bus

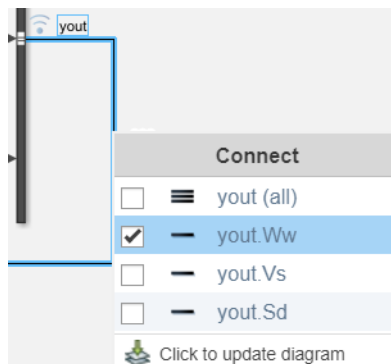
In addition to logging an entire Bus block, you can select one or more individual signals within a bus and add them to the **Logged Signals** section of Test Manager. For large buses, adding only the needed signals might reduce the amount of time it takes a test case to run.

- 1 Open the model using `openExample('sldemo_absbrake')`.
- 2 Use `sltest.testmanager.view` to open the Test Manager.
- 3 Click **New > Test File** to create a new test file. Name and save the file.

Under **System Under Test**, click the **Use current model** icon.

- 4 Under **Simulation Outputs**, click **Add**.
- 5 In the system under test, select the signal exiting the Bus, `yout`.

- 6 A dialog box appears. Select the desired bus or bus leaf signals in the dialog box. Three horizontal lines indicate the whole bus. A single horizontal line next to a signal name indicates that it is a bus leaf signal. If the leaf signals do not appear in the dialog, click **Click to update diagram**.



- 7 Continue adding signals to the test case, such as the non-bus Sd and tire torque signals
 8 When you finish adding signals, return to the Test Manager and click **Done**.
 9 The signals appear in the **Logged Signals** table in the test case.

LOGGED SIGNALS			
NAME	SOURCE	PORT INDEX	PLOT INDEX
▼ ✓ Signal Set 1			
✓ Sd	sldemo_absbrake/Stopp...	1	
✓ tire torque	sldemo_absbrake/Rr	1	
✓ yout.Ww	sldemo_absbrake/Bus ...	1	

Plot signals on the specified plots after simulation

+ Add Delete

To add a signal set, click the **Add** arrow and select **Signal Set**.

To specify a specific plot for a signal, enter a number in the **Plot Index** column. By default, the signals appear on one plot.

You can specify to display the plot immediately after running the test by selecting the **Plot signals on the specified plots after simulation** check box.

Use Triggers to Start and Stop Signal Logging

By default, output signals are logged for the entire simulation. To specify when signal logging starts and stops, use output triggers. An output trigger can be a condition expression or a duration. When a condition expression evaluates to true, logging starts or stops. The start duration is the time in seconds when logging starts after simulation starts. The stop duration is when logging stops after signal logging starts. To specify the triggers and the symbols to use in the conditional triggers, In the Test Manager, use the **Output Triggers** subsection under **Simulation Outputs** for a test case. To obtain the trigger results programmatically, use `sltest.testmanager.OutputTrigger` to specify triggers, and `sltest.testmanager.OutputTriggerResult` to obtain the trigger results. You must, however, use the Test Manager to define and map symbols that are used in conditional triggers.

Add Triggers to a Test Case

This example shows how to add triggers to a test case by using the Test Manager. The triggers start logging output data when the Ww output signal is greater than 40 and stop logging output data after 4 seconds.

1. Open the model.

```
sldemo_absbrake
```

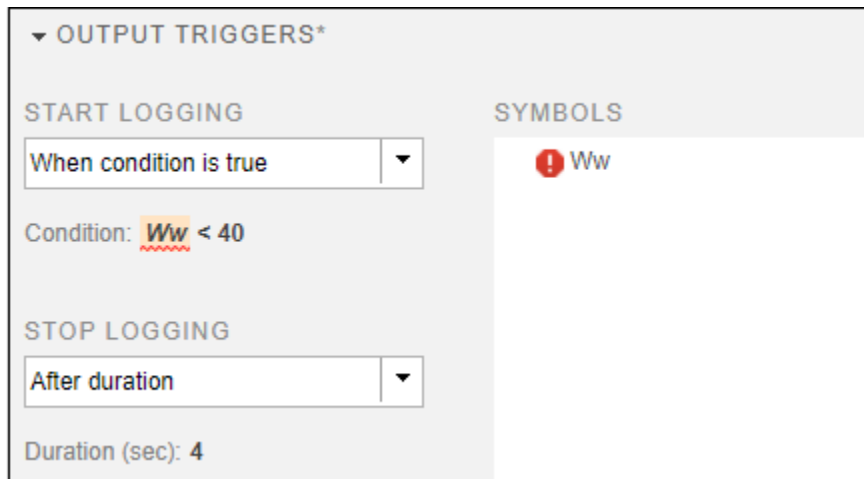
2. Open the test file in the Test Manager. This file contains a predefined test case, Trigger Test Case.

```
sltest.testmanager.load('AddTriggersToTest.mldatx');
sltest.testmanager.view;
```

3. Select **Trigger Test Case** and expand the **Simulation Outputs** section.

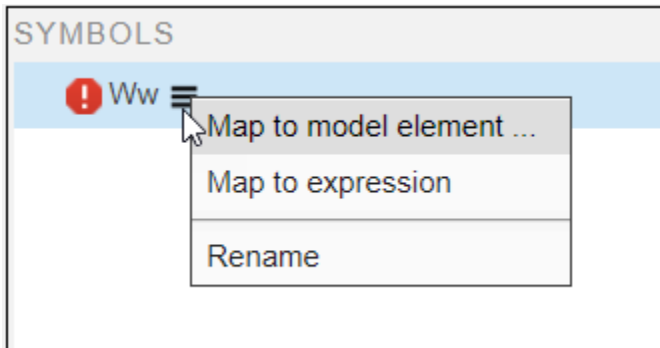
4. Under **Output Triggers**, set **Start Logging** to When condition is true. Then, click the edit icon next to **Condition** and enter $Ww < 40$.

5. Set **Stop Logging** to After Duration. Click next to **Duration (sec)** and enter 4. Leave **Shift time to zero** selected. This option shifts the logged signal so the triggered output shows as starting at time 0.

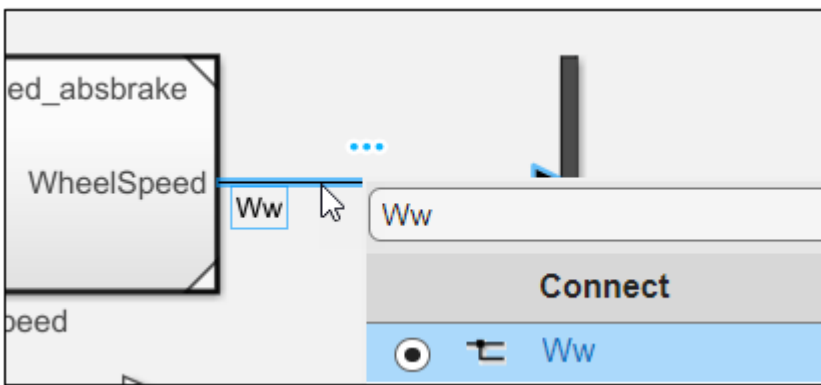


Ww appears in **Symbols** with a red icon, which indicates it is an unresolved symbol.

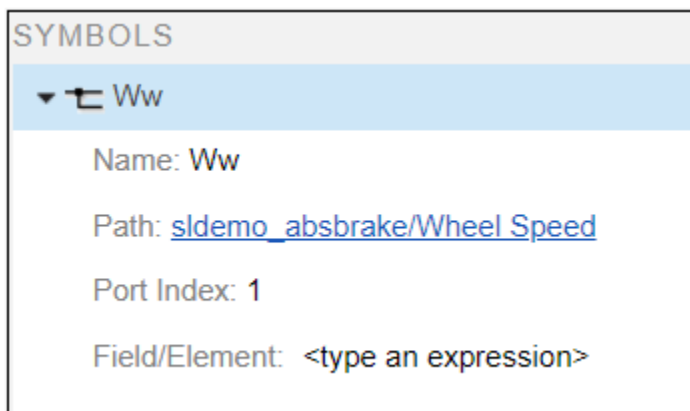
6. Point to Ww, click on the menu icon and and select **Map to model element**.



In the model, select the Ww signal. In the Connect dialog box, select Ww.



7. Return to the Test Manager and click **Done**. The Ww symbol updates and displays its details.



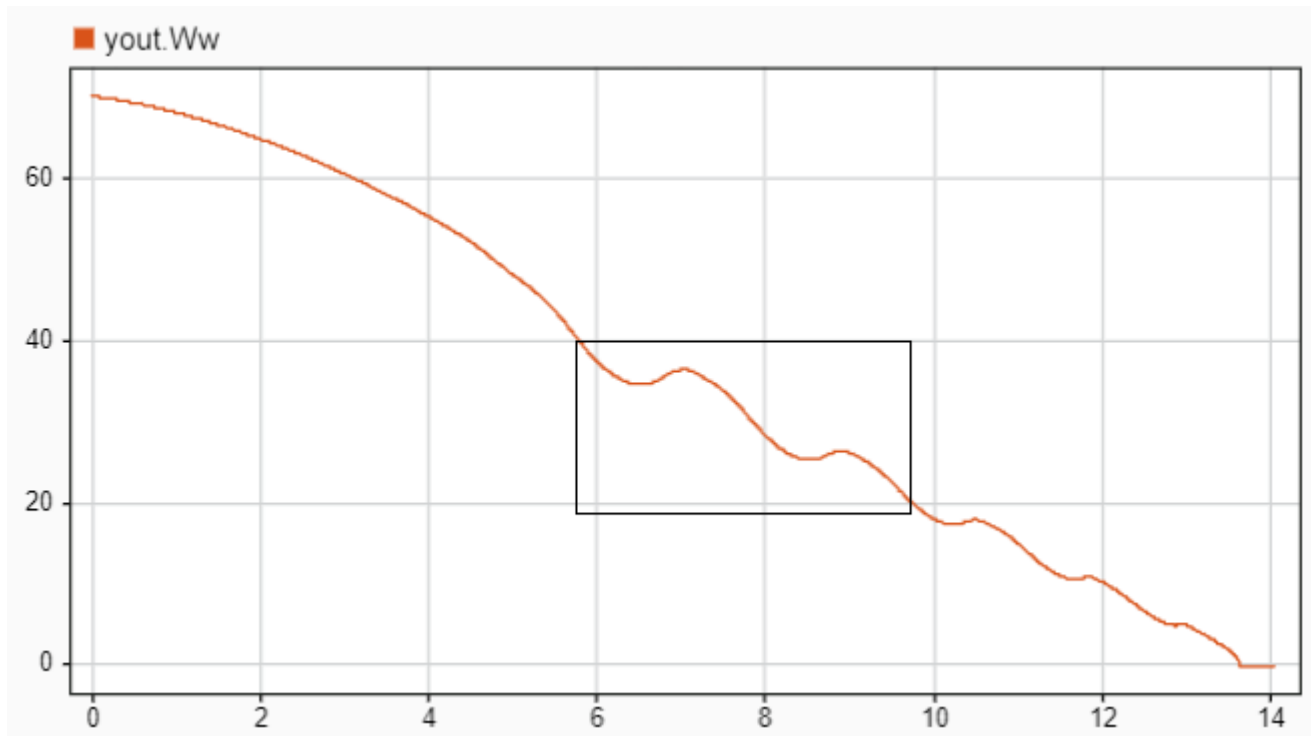
8. Run the test.

9. In the **Results and Artifacts** pane, display the triggered Ww output.



Compare the triggered output plot to the plot below, which shows the output when the entire simulation is logged. The triggered portion is highlighted. The triggered plot above is shifted to start at time 0. Logging starts when the signal equals 40 and stops 4 seconds later.

To create the plot of the entire simulation, set **Start Logging** to *On simulation start* and **Stop Logging** to *When simulation stops*.



11. To see a summary of the simulation, including the trigger information, select **Trigger Test Case** and expand **Simulation Metadata**.

NAME	STATUS
Results: 2022-Dec-21 12:59:58	1 ✔
Trigger Test Case	✔
Sim Output (sldemo_absbrake)	
yout	
Ww	—
Vs	—
Sd	—
slp	—
Ww	—

Simulation Metadata	
Model	sldemo_absbrake
Simulation Mode	normal
Override SIL or PIL Mode	false
Configuration Set	Configuration
Start logging criteria	When condition is true 'Ww < 40'
Stop logging criteria	After duration '4'
Shift time to zero	true
Logging Start Time in Simulation	5.809845948115707
Logging Stop Time in Simulation	9.8098459481157079
Logged Run Start Time	0
Logged Run Stop Time	4.0000000000000009
Start Time	0
Stop Time	14.007553608887029
Checksum	4074074133 400020064 2644322550

12. Clean up and close the Test Manager and close the model.

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

Trigger Limitations

These limitations apply to test cases that have conditional or duration triggers enabled:

- You cannot collect coverage.
- If a test case has iterations, the start and stop triggers defined for the test case apply to all iterations.

See Also

```
sltest.testmanager.TestCase | sltest.testmanager.LoggedSignal |
sltest.testmanager.LoggedSignalSet | Simulink.Signal |
sltest.testmanager.OutputTrigger | sltest.testmanager.OutputTriggerResult |
sltest.testmanager.TriggerMode | Test Manager
```

More About

- “Assess Simulation and Compare Output Data” on page 3-14
- “Compare Model Output to Baseline Data” on page 6-7

Run Tests in Multiple Releases of MATLAB

If you have more than one release of MATLAB installed, you can run tests in multiple releases. Starting with R2011b, you can also run tests in releases that do not have Simulink Test. Running tests in multiple releases enables you to use test functionality from later releases while running the tests in your preferred release of Simulink. You can also compare test results across multiple releases to better understand Simulink changes before upgrading to a new version of MATLAB and Simulink.

Although you can run test cases on models in previous releases, the release you run the test in must support the features of the test. For example, if 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 the releases to your list of available releases using Test Manager preferences. See “Add Releases Using Test Manager Preferences” on page 6-95.

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 Models”.
- 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”.

Test Case Compatibility with Previous Releases

When collecting coverage in multiple-release tests, you can run tests cases up to three years (six releases) prior to the current release. Tests that contain logical or temporal assessments are supported in R2016b and later releases.

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
- Models with observers
- Input data defined in an external Excel document
- Including custom figures from test case callbacks

Add Releases Using Test Manager Preferences

Before you can create tests for multiple releases, use Test Manager preferences to include the MATLAB release you want to test in. You can also delete a release that you added to the available releases list. However, you cannot delete the release from which you are running Test Manager.

- 1 In the Test Manager, 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/Remove releases** to open the Release Manager.
- 4 In the Release Manager, click **Add**.
- 5 Browse to the location of the MATLAB release you want to add and click **OK**.
- 6 To change the release name that will appear in the Test Manager, edit the **Name** field.
- 7 Close the Release Manager. The Preferences dialog box shows the selected releases. Deselect releases you do not want to make available for running tests.

Run Baseline Tests in Multiple Releases

When you run a baseline test with the 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.

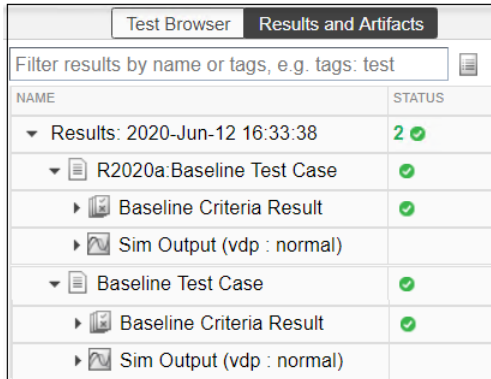
- 1 Make sure that the release has been added to your Test Manager preferences.
- 2 Create a test file, if necessary, and add a baseline test case to it.
- 3 Select the test case.
- 4 Under **System Under Test**, enter the name of the model you want to test.
- 5 Set up the rest of the test.
- 6 Capture the baseline. Under **Baseline Criteria**, click **Capture**. Specify the format and file in which to save the baseline and select the release in which to capture the baseline. Then, click **Capture** to simulate the model.

For more information about capturing baselines, see “Capture Baseline Criteria” on page 6-167.

After you create the baseline, run the test in the selected releases. Each release you selected generates a set of results.

- 1 In the test case, expand **Simulation Setting and Release Overrides** and, in the **Select releases for simulation** drop-down menu, select the releases you want to use to compare against your baseline.
- 2 Specify the test options.
- 3 From the toolstrip, click **Run**.

For each release that you select when you run the test case, the 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: 2020-Jun-12 16:33:38	2
R2020a:Baseline Test Case	
Baseline Criteria Result	
Sim Output (vdp : normal)	
Baseline Test Case	
Baseline Criteria Result	
Sim Output (vdp : normal)	

Run Equivalence Tests in Multiple Releases

When you run an equivalence test, you compare two simulations. Each simulation runs in a single release, which can be the same or different. Examples of equivalence tests include comparing models run in different model simulation modes, such as normal and software-in-the-Loop (SIL), or comparing different tolerance settings.

- 1 Make sure that the releases have been added to your Test Manager preferences.
- 2 Create a test file, if necessary, and add an equivalence test case to it.
- 3 Select the test case.
- 4 Under **Simulation 1, System Under Test**, enter the name of the model you want to test.
- 5 Expand **Simulation Setting and Release Overrides** and, in the **Select releases for simulation** drop-down menu, select the release for Simulation 1 of the equivalence test. For an equivalence test, only one release can be selected for each simulation.
- 6 Set up the rest of the test.
- 7 Repeat steps 4 through 6 for **Simulation 2**.
- 8 In the toolstrip, click **Run**.

The test runs each simulation in the release you selected and compares the results for equivalence. For each release that you selected when you ran the test case, the 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: 2020-Jun-21 14:51:59	1 ✓
▼ SIL/PIL Equivalence test	✓
▼ Equivalence Criteria Result	✓
○ CounterA:1	✓
○ CounterB:1	✓
○ CounterC:1	✓
▼ Sim Output 1 (rtwdemo_mdref)	
☑ CounterA:1	—
☐ CounterB:1	—
☐ CounterC:1	—
▼ R2020a: Sim Output 2 (rtwden)	
☑ CounterA:1	—
☐ CounterB:1	—
☐ CounterC: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 releases have been added to your Test Manager preferences.
- 2 Create a test file, if necessary, and add a simulation test case template to it.
- 3 Select the test case.
- 4 Under **System Under Test**, enter the model you want to test.
- 5 Expand **Simulation Setting and Release Overrides** and, in the **Select releases for simulation** drop-down menu, select the release options for the simulation.
- 6 Under **Simulation Outputs**, select the signals to log.
- 7 In the toolbar, click **Run**.

The test runs, simulating for each release you selected. For each release, the 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: 2020-Jun-10 12:39:00	1 ✓
▶ R2020a: Test Case 1	✓
▶ R2019b: Test Case 1	✓
▶ Test Case 1	✓

Assess Temporal Logic in Multiple Releases

You can run tests that contain logical and temporal assessments in multiple releases to test signal logic for models created in an earlier release. You can also compare assessment results across releases when you run the tests in multiple releases. For more information, see “Assess Temporal Logic by Using Temporal Assessments” on page 3-93.

You can run these test case types with logical and temporal assessments:

- Baseline tests
- Equivalence tests
- Simulation tests

Run Tests with Logical and Temporal Assessments

To run tests logic with logical and temporal assessments in multiple releases:

- 1 Start MATLAB R2021b or later.
- 2 Open the Test Manager. For more information, see “Open the Test Manager”.
- 3 In the Test Manager, add the releases to your Test Manager preferences. For more information, see “Add Releases Using Test Manager Preferences” on page 6-95.
- 4 Create a new test file with a baseline, equivalence, or simulation test case, or open an existing one. For more information, see:
 - “Create a Simple Baseline Test”
 - “Create and Run a Back-to-Back Test” on page 6-41
 - “Test a Simulation for Run-Time Errors” on page 6-16
- 5 In the Test Manager, specify your test case properties, including the system under test and other properties that you want to apply. For more information, see “Specify Test Properties in the Test Manager” on page 6-158.
- 6 Add a logical or temporal assessment to your test case. For more information, see “Assess Temporal Logic by Using Temporal Assessments” on page 3-93 and “Logical and Temporal Assessment Syntax” on page 3-107.
- 7 Select the releases to run the test in. In the Test Manager, select your test case. In **System Under Test**, under **Simulation Settings and Release Overrides**, next to **Select releases for simulation**, select the releases to run the test case in from the list.

If you are using a baseline or simulation test case, you can run the test in multiple releases in a single run by selecting multiple releases from the list. If you are using an equivalence test case, you can select one release under **Simulation 1** and another release under **Simulation 2**. For more information, see:

- “Run Baseline Tests in Multiple Releases” on page 6-95
 - “Run Equivalence Tests in Multiple Releases” on page 6-96
 - “Run Simulation Tests in Multiple Releases” on page 6-97
- 8 Run the test. In the Test Manager, click **Run**.

Evaluate Assessment Results

The **Results and Artifacts** pane displays the test results for each release you selected. The test release appears in the name of each test result from a release other than the version you ran Test Manager from.

The screenshot shows the 'Test Browser' interface with the 'Results and Artifacts' tab selected. At the top, there is a search bar with the text 'Filter results by name or tags, e.g. tags: tes'. Below the search bar is a table with two columns: 'NAME' and 'STATUS'. The table contains a hierarchical list of test results:

NAME	STATUS
▼ Results: 2021-Jul-07 08:21:18	2 ✓
▼ R2020a: TC_Simulation	✓
▶ Sim Output (sltestBaselineBasi	
▼ Logical and Temporal Assessm	✓
○ Assessment1	✓
▼ TC_Simulation	✓
▶ Sim Output (sltestBaselineBasi	
▼ Logical and Temporal Assessm	✓
● Assessment1	✓

You can evaluate the assessment results independently from other pass-fail criteria. For example, while a baseline test case might fail due to a failing baseline criteria, a logical or temporal assessment in the test case might pass.

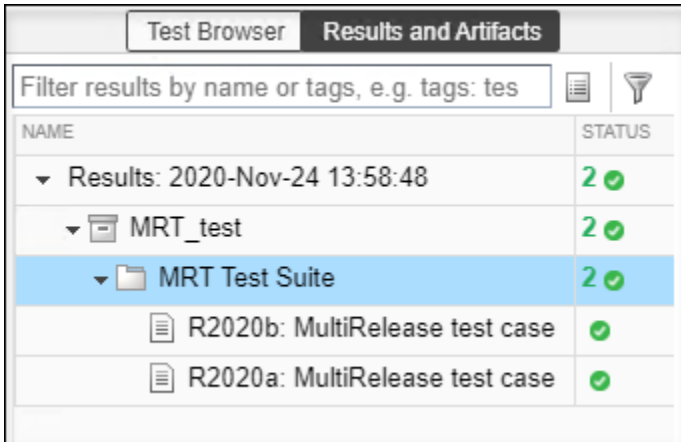
You can also examine detailed assessment signal behavior. For more information, see “View Assessment Results” on page 3-96.

Collect Coverage in Multiple-Release Tests

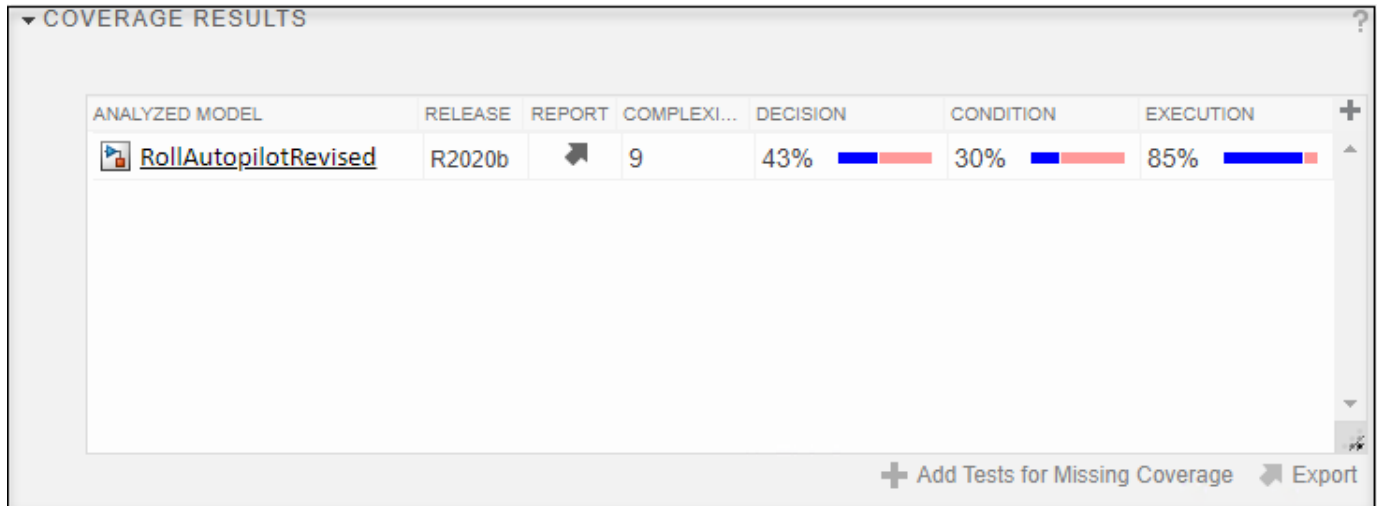
To add coverage collection for multiple releases, you must have a Simulink Coverage license. Set up your test as described in “Run Baseline Tests in Multiple Releases” on page 6-95, “Run Equivalence Tests in Multiple Releases” on page 6-96, or “Run Simulation Tests in Multiple Releases” on page 6-97. You can use external test harnesses to increase coverage for multiple-release tests. Before you capture the baseline or run the equivalence or simulation test, enable coverage collection.

- 1 Click the test file that contains your test case. To collect coverage for test suites or test cases, you must enable coverage at the test file level.
- 2 In the **Coverage Settings** section, select **Record coverage for system under test**, **Record coverage for referenced models**, or both.
- 3 Select the types of coverage to collect under **Coverage Metrics** to collect.

After you run the test, the **Results and Artifacts** pane shows the pass-fail results for each release in the test suite.









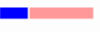



To view the coverage results for a release, select its test case and expand the **Coverage Results** section. The table lists the model, release, and the coverage percentages for the metrics you selected.



To view aggregated coverage results for the releases in your test, select the test suite that contains the releases and expand the **Aggregated Coverage Results** section.

▼ AGGREGATED COVERAGE RESULTS ?

ANALYZED MODEL	RELEASE	REPORT	COMPLEXI...	DECISION	CONDITION	EXECUTION	
 RollAutopilotRevised	R2020a		9	43% 	30% 	85% 	+
 RollAutopilotRevised	R2020b		9	43% 	30% 	85% 	+

+ Add Tests for Missing Coverage ↗ Export

To use the current release to add tests for missing coverage to an older release, click the row and click **Add Tests for Missing Coverage**. You can also use coverage filters, generate reports, merge results, import and export results, and scope coverage to linked requirements. For more information, see “Collect Coverage in Tests” on page 6-135 and “Increase Test Coverage for a Model” on page 6-147.

See Also

`sltest.testmanager.getpref` | `sltest.testmanager.setpref`

More About

- “Simulation Settings and Release Overrides” on page 6-161
- “Create a Simple Baseline Test”
- “Create and Run a Back-to-Back Test” on page 6-41
- “Test a Simulation for Run-Time Errors” on page 6-16
- “Assess Temporal Logic by Using Temporal Assessments” on page 3-93
- “Collect Coverage in Tests” on page 6-135

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 6-102.
- 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 6-104.

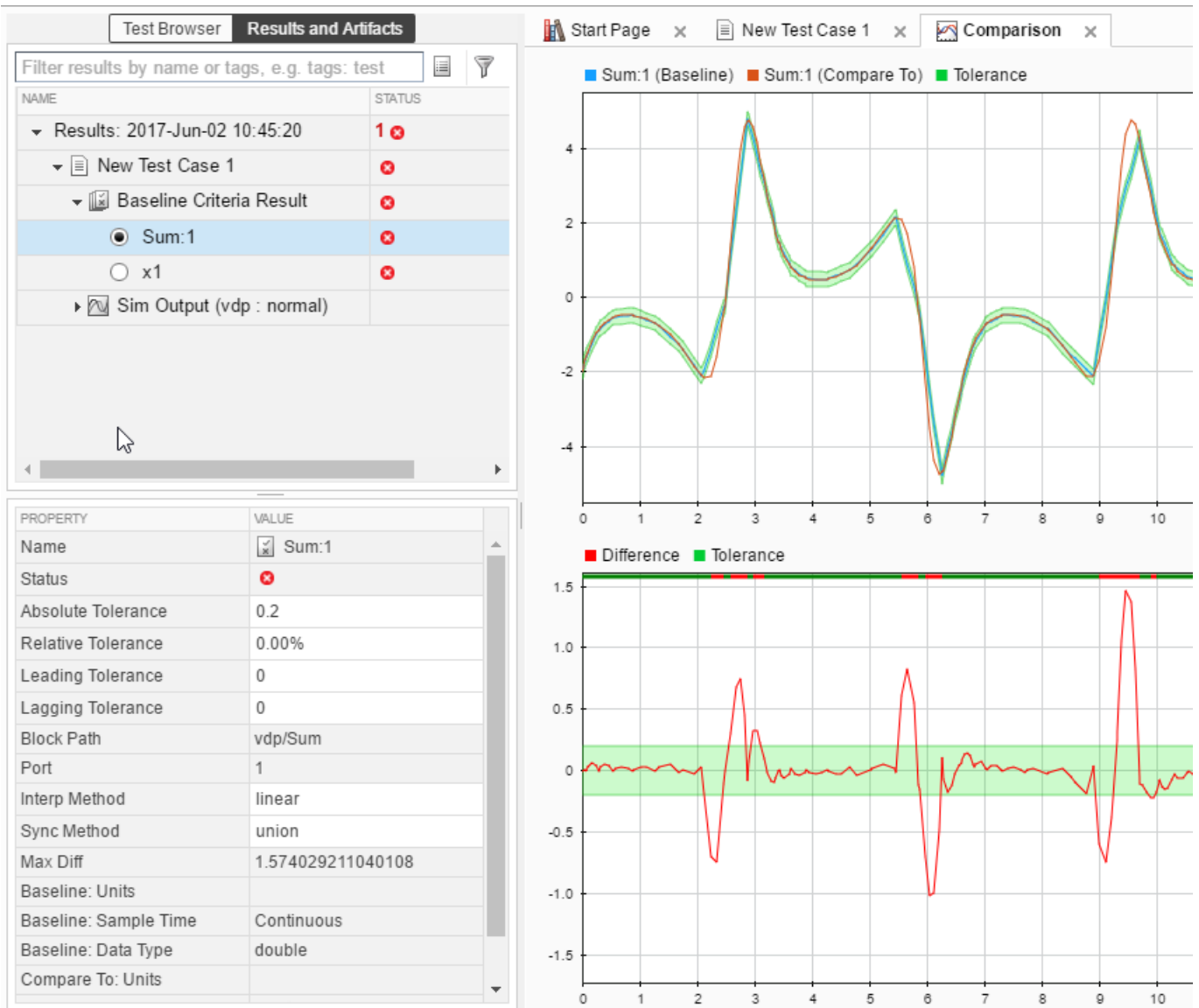
Examine Test Failure Signals and Update Baseline Test

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.

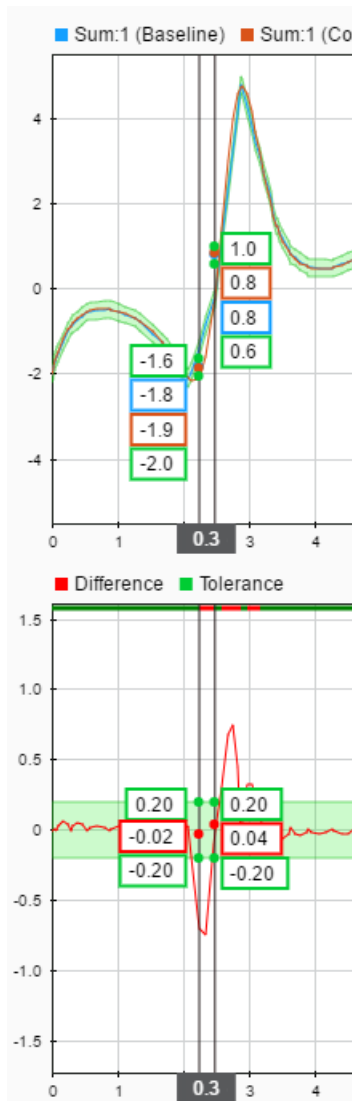
Suppose that your model uses a new solver. When you run the test case, the results do not match, causing the test to fail.

- 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 toolbar, 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**



button.

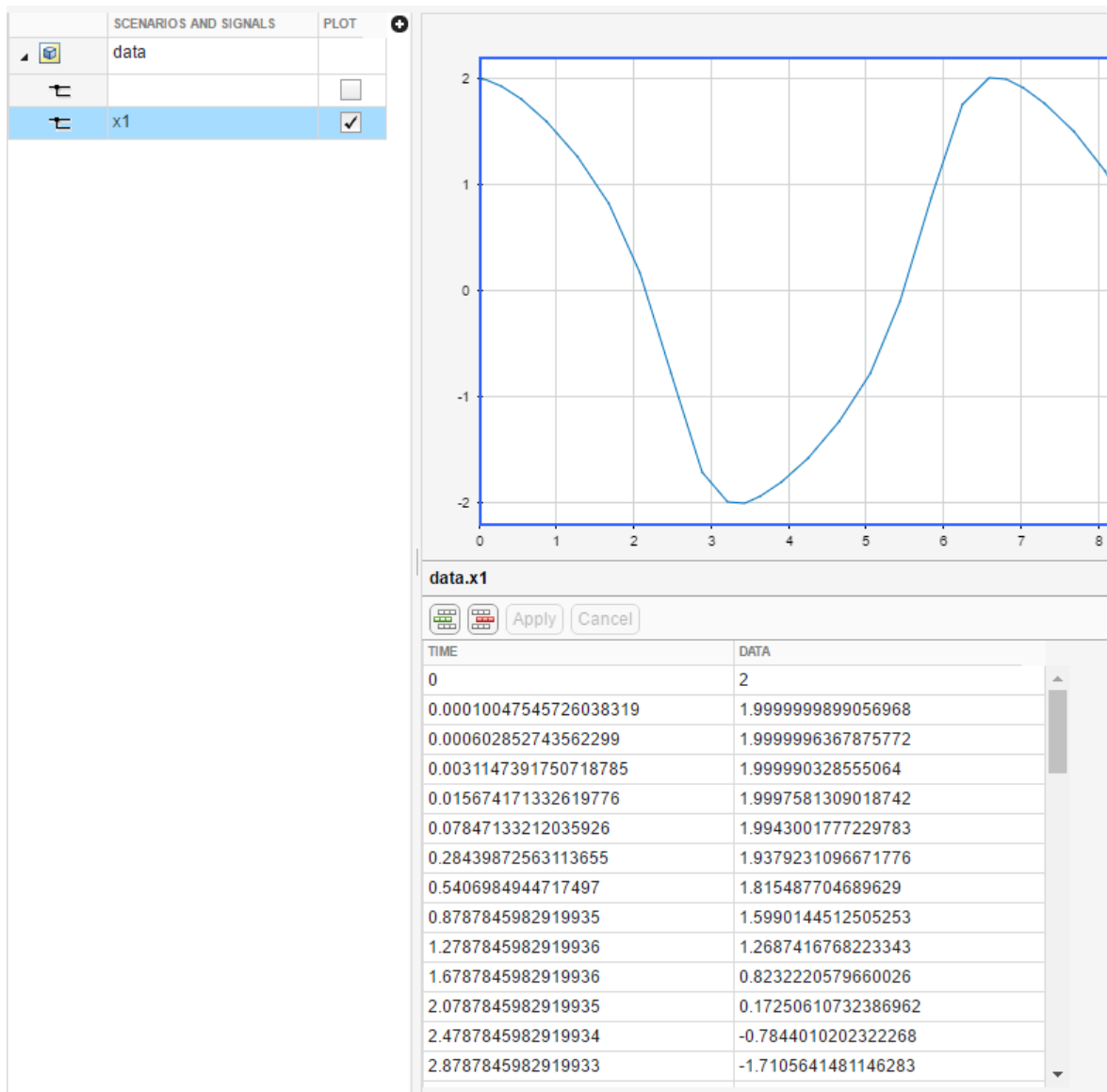
- To update the entire signal, select **Replace One Signal in Baseline File** from the dropdown.
- To update only the data in the failure region, select **Replace Signal Segment in Baseline File** from the dropdown.
- To replace all signal data in the baseline with the new data, select **Use All Sim Output Signals as Baseline** from the dropdown.

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. If the baseline is a Microsoft Excel file, you can edit the data 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**.

To update baseline signal data in an Excel file:

- 1 Open the Excel file that contains the baseline you want to edit.
- 2 Go to the sheet that contains the baseline. The sheet name corresponds to the baseline source name in the Test Manager **Baseline Criteria** section.
- 3 Edit the signal data in the sheet, and then save the Excel file.

See Also

More About

- “Create and Edit Signal Data”
- “Inspect Simulation Data”
- “Compare Model Output to Baseline Data” on page 6-7

Create and Run Test Cases with Scripts

In this section...

“Create and Run a Baseline Test Case” on page 6-107

“Create and Run an Equivalence Test Case” on page 6-108

“Run a Test Case and Collect Coverage” on page 6-109

“Create and Run Test Case Iterations” on page 6-109

For a list of functions and objects in the Simulink Test programmatic interface, see “Test Scripts”.

Create and Run a Baseline Test Case

This example shows how to use `sltest.testmanager` functions, classes, and methods to automate tests and generate reports. You can create a test case, edit the test case criteria, run the test case, export simulation output, and generate results reports programmatically. The example compares the simulation output of the model to a baseline.

```
% Open the model for this example
openExample('sldemo_absbrake');

% 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);

% Get the test case result and the Sim Output run dataset
tcr = getTestCaseResults(ResultsObj);
runDataset = getOutputRuns(tcr);

% 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);

% Export the Sim Output run dataset
dataset = export(runDataset);

```

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.

```

% Open the model for this example
openExample('sldemo_absbrake');

% 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.

```
% Open the model for this example
openExample('sldemo_autotrans');

% 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
rs = run(tf);

% Get the coverage results
cr = getCoverageResults(rs);

% Open the Test Manager to view results
sltest.testmanager.view;
```

In the **Results and Artifacts** pane of the Test Manager, click on Results. You can view the aggregated coverage results.

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 Editor scenario for each iteration.

```
% Open the model for this example
openExample('sldemo_autotrans');

% 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, 'SignalEditorScenario', '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, 'SignalEditorScenario', '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

More About

- “Import Test Cases for Equivalence Testing” on page 5-19

Test Models Using MATLAB-Based Simulink Tests

A MATLAB-based Simulink test is defined in a MATLAB code (.m) file that you create in MATLAB, and then open, run, and view results in the Test Manager. The test file is a class definition file that inherits from `sltest.TestCase`. The inheritance enables you to open the test file in the Test Manager. When you open a MATLAB test file in the Test Manager, it appears and behaves the same as a test created in the Test Manager, although with some limited functionality (see “Limitations of MATLAB- Based Tests” on page 6-115). In addition to using a MATLAB test in the Test Manager, you can use the MATLAB test at the command line like any other unit test file.

Because these test files are text (.m) files, you can edit, compare to and merge with other .m test files, and link from the file to requirements. In contrast, test files created in the Test Manager or by using the Simulink Test API are saved as binary MLDATX files.

Classes and Methods

TestCase Class and Methods

The `sltest.TestCase` class and its methods work specifically with MATLAB tests. You can use these methods in test files and at the command line, except for `sltest.TestCase.forInteractiveUse`, which can only be used at the command line. In addition to these methods, you can use the `matlab.unittest.TestCase` methods with MATLAB tests.

<code>sltest.TestCase</code>	Class from which to inherit
<code>loadSystem</code>	Loads model
<code>simulate</code>	Simulate model
<code>assumeSignalsMatch</code>	Assume two sets of data are equivalent
<code>assertSignalsMatch</code>	Assert two sets of data are equivalent
<code>fatalAssertSignalsMatch</code>	Fatal assert two sets of data are equivalent
<code>verifySignalsMatch</code>	Verify two sets of data are equivalent
<code>sltest.TestCase.forInteractiveUse</code>	Create test case for use at command line
<code>createTemporaryFolder</code>	Create temporary folder that is deleted when test case goes out of scope
<code>createSimulationInput</code>	Creates <code>Simulink.SimulationInput</code> or <code>sltest.harness.SimulationInput</code> object.

Test Harness Class

`sltest.harness.SimulationInput` creates an object that you can use to specify changes applied to a test harness during simulation. In addition to using this class for MATLAB-based Simulink tests, you can use it in other MATLAB code.

Test Runner Methods

These methods of `matlab.unittest.TestRunner` apply specifically to MATLAB-based Simulink tests.

- `addModelCoverage` — enables model coverage collection using the test runner. Instead of using this method, if you open your MATLAB-based Simulink test file in the Test Manager, you can enable coverage collection there.

- `addSimulinkTestResults` — pushes test results to the Simulink Test Manager.

Plugin Classes

These `sltest.plugins` classes enable functionality for MATLAB-based tests. In addition to these methods, you can use other `sltest.plugins` classes with these tests. The plugins can be attached to a `matlab.unittest.TestRunner` to enable functionality while running an `sltest.TestCase` test.

<code>sltest.plugins.MATLABTestCaseIntegrationPlugin</code>	Enable integrating MATLAB test simulation and test results with the Test Manager
<code>sltest.plugins.ToTestManagerLog</code>	Enable writing text output to Test Manager results Logged Signals pane of the Test Manager
<code>sltest.plugins.ModelCoveragePlugin</code>	Enable collecting model coverage

Creating a Baseline MATLAB-Based Simulink Test

To create a baseline MATLAB test:

- 1 Create a MATLAB code (.m) file that defines the test cases. You can launch the MATLAB Editor from the command line, or from the Test Manager by using **New > MATLAB-Based Simulink Test (.m)**.

See “Author Class-Based Unit Tests in MATLAB”. The only difference for MATLAB tests is that the class must inherit from `sltest.TestCase`, instead of from `matlab.unittest.TestCase`.

This sample MATLAB test file includes one test, which is defined in the `testOne` function. When you run the test in the Test Manager, the test loads the model named `sltestMATLABBasedTestExample`. It then sets the value of the `gain2_var` variable, and simulates the model. Finally, the test compares model simulation output to the baseline data MAT file.

```
classdef myTest < sltest.TestCase
    methods (Test)
        function testOne(testCase)
            testCase.loadSystem...
                ('sltestMATLABBasedTestExample');
            evalin('base','gain2_var = 2.01;');
            simOut = testCase.simulate...
                ('sltestMATLABBasedTestExample');
            testCase.verifySignalsMatch(simOut,'baselineOne.mat',...
                'AbsTol',0.015);
        end
    end
end
```

- 2 If you are using a test harness, replace the `simOut` line in the above MATLAB test file with

```
simOut = testCase.simulate('sltestMATLABBasedTestExample',...
    'WithHarness','sltestMATLABBasedTestExample_harness1');
```

or, to specify the stop the simulation time, replace `simOut` with

```
in = testCase.createSimulationInput('sltestMATLABBasedTestExample',...
    'WithHarness','sltestMATLABBasedTestExample_harness1');
```

```
in = in.setModelParameter("StopTime","10")
simOut = testCase.simulate(in);
```

- 3 If a baseline data MAT-file does not already exist or if you need to update it, at the MATLAB command line, use:

```
runtests(<test>, 'GenerateBaselines', true)
```


For the sample file, <test> is 'myTest/testOne'.

When you generate baselines, the test begins running. It pauses to open a Simulation Data Inspector report, and you are prompted at the MATLAB command line to review the baseline data. When you approve the data, it saves the baseline data to a new MAT file or updates the existing MAT-file. Then, the test continues to run, but fails because the new or updated baseline data is not included in the current run. Rerun the test using the `runtests` command to use the new or updated baseline and produce a passing result. You can also rerun the test by using the yellow rerun hyperlink at the command line.

- 4 Optionally, if you have a Simulink Coverage license, you can include coverage collection in your test by using the `sltest.plugins.ModelCoveragePlugin`. See “Collect Coverage Using MATLAB-Based Simulink Tests” on page 6-120 for an example and information on coverage collection.

Alternatively, you can collect coverage by using the Test Manager. After you open the test file in the Test Manager, select the test file, select a **Coverage Settings** option, and select the coverage metrics. The test suites and test cases in the file inherit the coverage settings from the test file. If you set the coverage collection and metrics in the Test Manager, you do not need to import or add the coverage plugins to the runner.

If you set the coverage metrics in the Test Manager and then close the Test Manager, the coverage settings are not saved to the MATLAB-based Simulink test code (.m) file.

- 5 Optionally, if you have a Requirements Toolbox license, you can add requirements. Open the Test Manager and update the test file.
 - a Click **Open > Open MATLAB-Based Simulink Test (.m)** and select the test file. The test file loads and its test hierarchy displays in the **Test Browser** pane. If you select the test file, the **Requirements** and **Test File Content** panes appear in the Test Manager.
 - b Add requirements by expanding the **Requirements** section by clicking **Add** to open the Outgoing Links Editor. See “Link to Requirements” on page 1-2 for information on adding requirements.
- 6 To update the MATLAB code (.m) test file from the Test Manager, click the **Open test in the MATLAB Editor** link.
- 7 After you edit the code file and save your changes, or after adding coverage or requirements, return to the Test Manager and click the synchronization button  next to the test file in the **Test Browser** pane.
- 8 Run the test, view the results, and create a test results report.
 - a Click **Run** to run the test.
 - b To view the results, expand the rows in the **Results and Artifacts** pane.
 - c To view coverage results, in the **Results and Artifacts** pane, select the **Results** item and expand the **Aggregated Coverage** section. See “Collect Coverage in Tests” on page 6-135 for information.

- d Optionally, create a test results report. See “Generate Test Results Reports” on page 7-17.

Alternatively, instead of adding coverage (Step 4) and running the test (Step 8) in the Test Manager, you can use these commands at the MATLAB command line to add coverage, run the test, and push the results to the Test Manager. Then, when you open the Test Manager, the test results are displayed.

```
suite = testsuite('myTests');
runner = testrunner('textoutput');
runner.addModelCoverage(...
    "CollectMetrics",["MDC", "Condition"]);
runner.addSimulinkTestResults("ExportToFile",...
    "testmgr_results.mldatx");
runner.run(suite);
```

Linking to Requirements from a MATLAB-Based Simulink Test File

Note You must have Requirements Toolbox to include requirements links.

To add links to requirements from a file being edited in the MATLAB Editor, see “Requirements Traceability for MATLAB Code” (Requirements Toolbox). For MATLAB test files, you add links using the same process. However, the text you select in the MATLAB code (.m) file determines the type of link and the test to which it is added. If you select:

- Class definition line (e.g., `classdef myClass < sltest.TestCase`) — Adds a **Verified By** link for the whole test file
- Text inside a test function — Adds a **Verified By** link for that function
- Text across multiple test functions — Adds a **Verified By** link for the first function in the selection
- Any other text selected — Adds a **Related To** link for the selection

After you add requirements links, you can view the verification status in the Requirements Editor by clicking **Display > Verification Status**. To update the status of a **Verified By** requirement, right-click on the requirement and select **Run Tests**. See “Review Requirements Verification Status” (Requirements Toolbox).

Links that you create in the MATLAB code (.m) file appear in the **Requirements** section of the Test Manager. Linking to requirements from in the Test Manager works the same as described in “Link to Requirements” on page 1-2.

When you have the Requirements Editor open and you click on an incoming link that is for a MATLAB test, if you have a Simulink Test license, the Test Manager opens and goes to the associated test. If a license is not available, the MATLAB Editor opens and goes to the line of code associated with that requirement.

For a parameterized test, Requirements Toolbox does not support linking to individual parameterized versions of the test. In your .m file, if you create a link from parameterized test to a requirement, the link is associated with all versions of that test. In the Test Manager, if you create a link from a version of a parameterized test to a requirement, the link is associated with all versions of that test.

Limitations of MATLAB- Based Tests

MATLAB-based Simulink tests do not support:

- Test types other than baseline tests.
- Running tests in parallel.
- Running tests in multiple releases.
- Test tags and descriptions.
- Callbacks. (However, while callbacks are not supported in the Test Manager for MATLAB-based tests, you can use `TestClassSetup` and `TestMethodSetup`, or fixtures in your `.m` file to achieve similar functionality. See “Write Setup and Teardown Code Using Classes”.)
- Logical and temporal assessments.
- Enabling coverage collection or changing coverage metrics at the test suite or test file level in the Test Manager.
- Saving coverage settings to the MATLAB-based Simulink test MATLAB code (`.m`) file from the Test Manager.

See Also

`addSimulinkTestResults` | `addModelCoverage` | `sltest.TestCase` | `matlab.unittest.TestCase` | `matlab.unittest.TestRunner`

Related Examples

- “Using MATLAB-Based Simulink Tests in the Test Manager” on page 6-116
- “Collect Coverage Using MATLAB-Based Simulink Tests” on page 6-120
- “Test Models Using MATLAB Unit Test” on page 6-191
- “Link to Requirements” on page 1-2
- “Requirements Traceability for MATLAB Code” (Requirements Toolbox)
- “Link Test Cases to Requirements” (Requirements Toolbox)

Using MATLAB-Based Simulink Tests in the Test Manager

This example shows how to create a MATLAB®-based Simulink® test, generate a baseline, and load, run, and view test results in the Test Manager. When you load a MATLAB-based Simulink test case code .m file into the Test Manager, the test case appears and behaves the same as a test case created directly in the Test Manager. The only difference is that for a MATLAB-based Simulink test, you can select or change coverage collection and metrics only at the test file level, not at either the test suite or test case level.

This example using internal test harness `sltestMATLABBasedTestExample_harnrss` verifies the `sltestMATLABBasedTestExample` model against a generated baseline.

Baseline Test Class Definition File

The class definition file, `BaselineTest.m`, has already been created and is provided with this example.

The test case file, `BaselineTest.m`, is derived from `sltest.TestCase`, which in turn is derived from `matlab.unittest.TestCase`. All of the `matlab.unittest.TestCase` methods are also available as a part of `sltest.TestCase`.

Baseline Test File Contents

The class definition file, `BaselineTest.m`, contains:

```
classdef BaselineTest < sltest.TestCase

    methods (Test)
        function testOne(testCase)
            testCase.loadSystem('sltestMATLABBasedTestExample');
            evalin('base','gain2_var = 2.01;');
            simOut = testCase.simulate('sltestMATLABBasedTestExample',...
                'WithHarness','sltestMATLABBasedTestExample_harness');
            testCase.verifySignalsMatch(simOut,'baseline1.mat','AbsTol',0.015);
        end
    end
end
```

end

The file includes:

- Inheritance from `sltest.TestCase`.
- A test function named `testOne`, which is in a `methods` block that has the `Test` attribute.

The `testOne` function:

- Uses the `testCase.loadSystem` method to load the `sltestMATLABBasedTestExample` model.
- Changes the value of the `gain2_var` in the model to 2.01.
- Uses the `testCase.simulate` method to simulate the model with the harness.
- Uses the `testCase.verifySignalsMatch` method to compare the output of `simulate`, `simOut`, to the baseline data MAT-file named `baseline1.mat`. It also sets an absolute tolerance. If you remove the tolerance setting from the file before running the test, the test fails because the value of `gain2_var` was changed from its original value in the model.

Baseline Data File

The baseline data file, `baseline1.mat`, has already been generated and is provided with this example. The baseline data file was created using this process:

1. Use `runtests('BaselineTest/testOne','GenerateBaselines',true)`.
2. After the baseline test runs, a Simulation Data Inspection report shows the output from the signals. View the Actual Results in the report and approve the baseline data. The data is saved in a MAT-file, which for this example is named `baseline1.mat`.

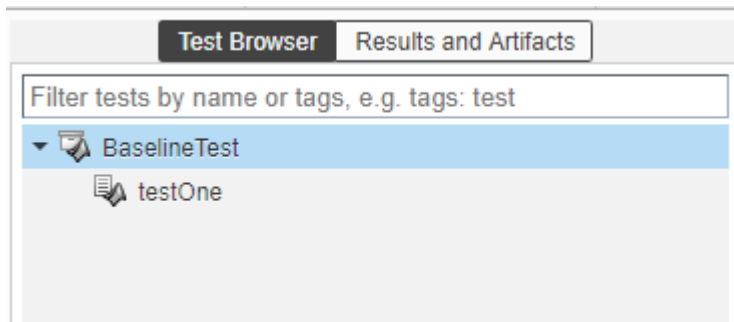
Open the MATLAB-based Simulink Test in the Test Manager

1. Open the Test Manager.

```
sltest.testmanager.view
```

2. In the Test Manager, click **Open** and select **Open MATLAB-based Simulink Test (.m)**.
3. In the Open File dialog box, select `BaselineTest.m`.

The Test Manager populates the **Test Browser** with **testOne** from the `BaselineTest.m` file.



Set Up Coverage Collection

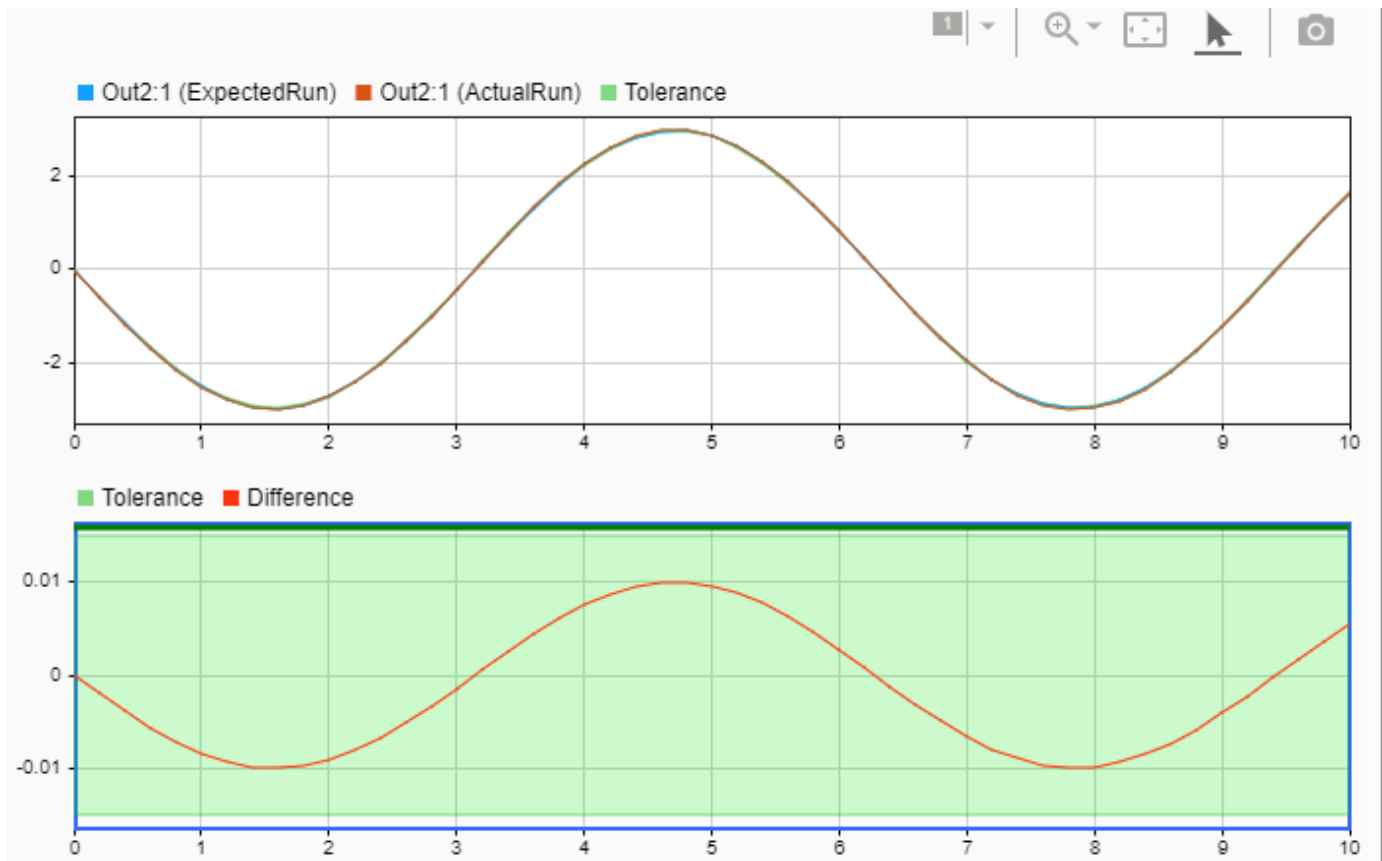
If you want to collect coverage for the test, select the `BaselineTest` file. In the Coverage Settings section, select the Coverage to Collect and the Coverage Metrics. If you want to collect coverage for the test, select the test file, `BaselineTest`, in the **Test Browser** pane. In the Coverage Settings section, select a **Coverage to Collect** option and **Coverage Metrics** options.

Run and Visualize the Results in the Test Manager

1. Click **Run** to execute the test.
2. After the test completes, expand all rows in the **Results and Artifacts** pane. Notice that `testOne` passes.

Test Browser Results and Artifacts	
Filter results by name or tags, e.g. tags: test	
NAME	STATUS
▼ Results: 2020-Jun-11 09:47:14	2 ●
▼ BaselineTest	2 ●
▼ testOne	●
▼ Simulation Output Comparison Result	●
○ Out1:1	●
○ Out2:1	●
▼ Sim Output (sltestMATLABBasedTestE	
□ Out1:1	—
□ Out2:1	—

3. To view the data comparison, select **Out2:1** under **testOne > Simulation Output Comparison Results**. The simulation and baseline signals match within the specified tolerance.



4. If you collected coverage, select the Baseline Test test file and view the Aggregated Coverage Results section.

4. If you collected coverage, select the BaselineTest test file and view the **Aggregated Coverage Results** section.

Clear and close the Test Manager

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

See Also

Related Examples

- “Test Models Using MATLAB-Based Simulink Tests” on page 6-111

Collect Coverage Using MATLAB-Based Simulink Tests

This example shows how to use a MATLAB®-based Simulink® test to collect coverage on a model with a test harness, and use the MATLAB Test Framework to populate the results in the Test Manager. MATLAB-based Simulink tests are MATLAB code (.m) file test case class definitions that inherit from `sltest.TestCase`.

MATLAB-Based Simulink Test File

The MATLAB-based Simulink test file, `TestHarnessWithModelCoverage.m`, has been created and is provided with this example. The test file contains two test functions. Each one has a harness model to drive input data to test the subsystem `TestHarnessWithModelCoverage/Subsystem1` and compare with corresponding baseline. This test uses a `Simulink.SimulationOutput` object when simulating the model.

```
classdef TestHarnessWithModelCoverage < sltest.TestCase

    methods (Test)
        function testOne(testCase)
            in = testCase.createSimulationInput('simpleSwitchWithSubsystemIn',...
                'WithHarness','simpleSwitchWithSubsystemIn_Harness1');
            simOut = testCase.simulate(in);
            testCase.verifySignalsMatch(simOut,'baselineOne.mat');
        end
        function testTwo(testCase)
            in = testCase.createSimulationInput('simpleSwitchWithSubsystemIn',...
                'WithHarness','simpleSwitchWithSubsystemIn_Harness2');
            simOut = testCase.simulate(in);
            testCase.verifySignalsMatch(simOut,'baselineTwo.mat');
        end
    end
end

end
```

Create a TestRunner and Test Suite

Create a `TestRunner` to run the `sltest_ratelim` model.

```
import matlab.unittest.TestRunner;
runner = TestRunner.withTextOutput;
```

Create a `TestSuite` to use with the `TestRunner`.

```
suite = testsuite('TestHarnessWithModelCoverage');
```

Configure the Test Runner

Use plugin methods to configure the `TestRunner` to add test results from an `sltest.TestCase` to the Test Manager. Add the `TestRunnerPlugin` to the `TestRunner`.

```
import sltest.plugins.MATLABTestCaseIntegrationPlugin;
runner.addPlugin(MATLABTestCaseIntegrationPlugin);
```

The `DiagnosticsOutputPlugin` and the `ToTestManagerLog` stream the diagnostics from an `sltest.TestCase` run to the logs of `TestCaseResults` in the Test Manager. The diagnostics

include passing diagnostics for tests that pass. Add the `DiagnosticsOutputPlugin` and `ToTestManagerLog` to the `TestRunner`.

```
import sltest.plugins.ToTestManagerLog;
import matlab.unittest.plugins.DiagnosticsOutputPlugin;
streamOutput = ToTestManagerLog();
diagnosticsOutputPlugin = DiagnosticsOutputPlugin...
    (streamOutput, 'IncludingPassingDiagnostics', true);
runner.addPlugin(diagnosticsOutputPlugin);
```

Configure Coverage Collection for a Simulink Model

Models in an `sltest.TestCase` that are simulated using the `simulate` method can collect coverage. Use the `ModelCoveragePlugin` to configure coverage metrics collection. This example collects MCDC coverage. Add the `ModelCoveragePlugin` to the `TestRunner`.

```
import sltest.plugins.coverage.CoverageMetrics;
import sltest.plugins.ModelCoveragePlugin;
mcdcMetrics = CoverageMetrics('MDC', true);
runner.addPlugin(ModelCoveragePlugin('Collecting', mcdcMetrics));
```

Alternatively, you can turn on coverage collection and set coverage metrics in the Test Manager. If you use this alternative, you do not need to import or add the coverage plugins to the runner.

Collect and Add Coverage and Test Results to the Test Manager

Now that the `TestRunner` is fully configured, use the `run` function to collect coverage and add the coverage and test results to the Test Manager.

```
run(runner, suite);
```

```
Setting up ResultSetFixture
Done setting up ResultSetFixture
```

```
Running TestHarnessWithModelCoverage
..
Done TestHarnessWithModelCoverage
```

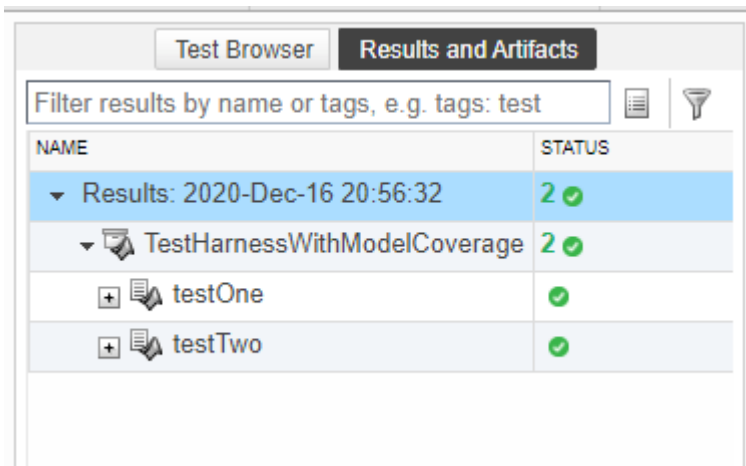
```
Coverage Report for simpleSwitchWithSubsystemIn/Subsystem1
  C:\TEMP\Bdoc23a_2213998_3568\ib570499\29\tp379082cd_eda0_4fd9_ae11_0f395fa79b20.html
Tearing down ResultSetFixture
Done tearing down ResultSetFixture
```

`run` also generates a report that includes cumulative coverage for the test suite that was run. Use the [Coverage Report for sltest_ratelim](#) link to view the report.

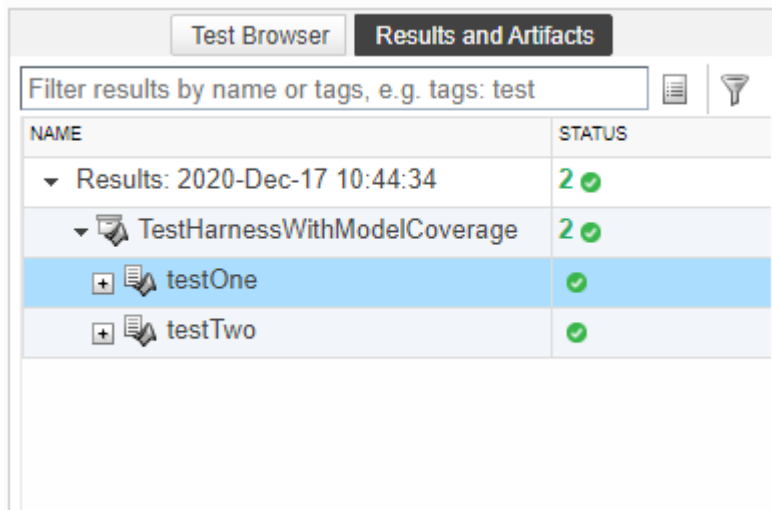
Open the Test Manager

```
sltest.testmanager.view
```

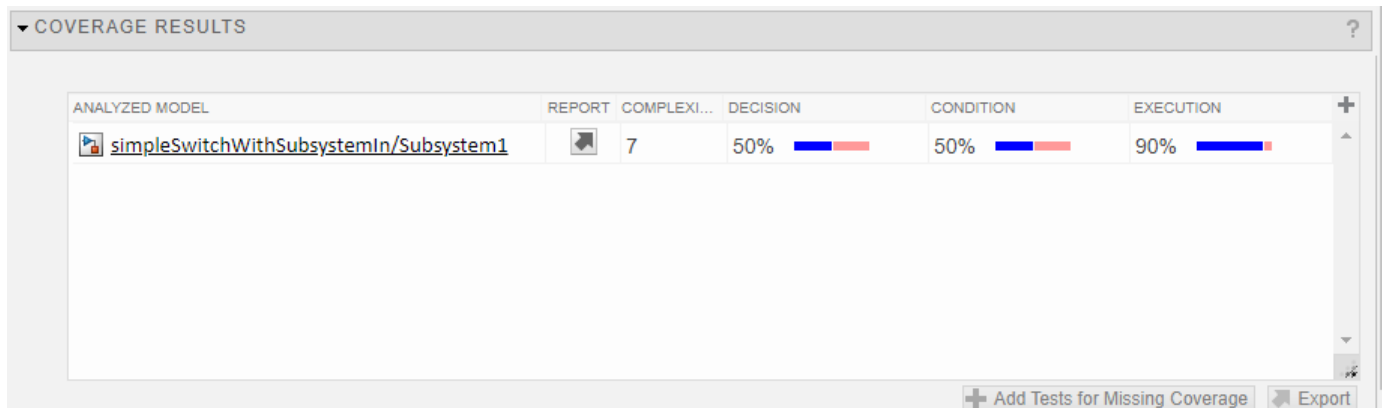
Select the **Results and Artifacts** pane and expand the **Results** and **BaselineTestWithCoverage** rows.



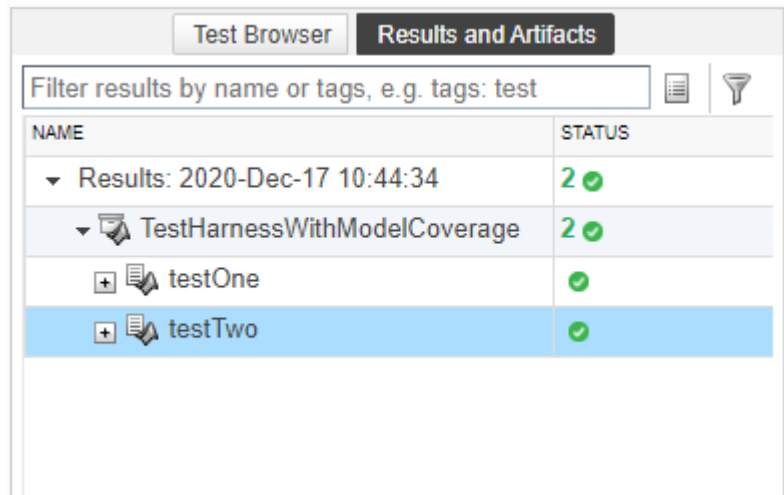
Select the testOne row.



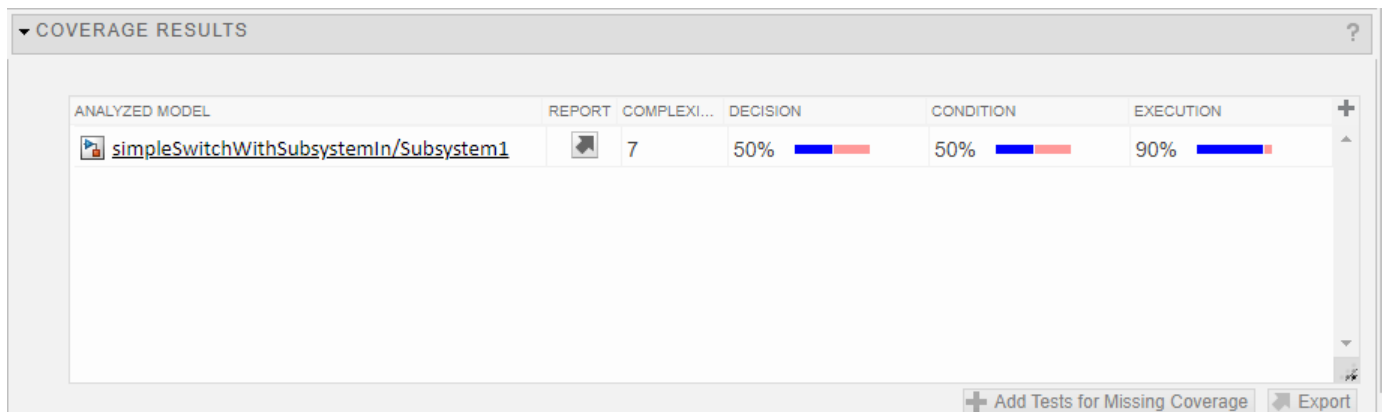
The **Coverage Results** section shows the coverage collected for sltest_ratelim from testOne.



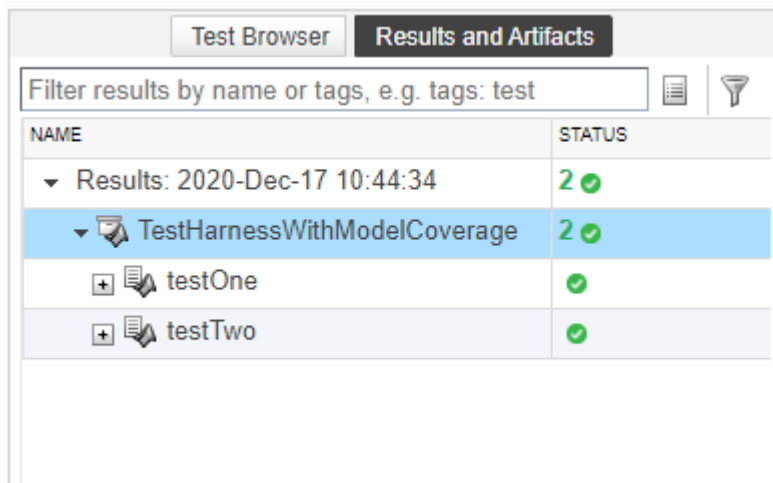
Select the testTwo row.



The **Coverage Results** section shows the coverage collected for `sltest_ratelim` from `testTwo`.





Select the `BaselineTestWithCoverage` row.



The **Aggregated Coverage Results** section shows the aggregation of the coverage collected for `sltest_ratelim` from `testOne` and `testTwo`. The aggregated results show full coverage for the specified coverage metrics.



The screenshot shows a window titled "AGGREGATED COVERAGE RESULTS" with a table of coverage data. The table has five columns: "ANALYZED MODEL", "REPORT", "COMPLEXI...", "DECISION", and "CONDITION", and a sixth column for "EXECUTION". The data row shows 100% coverage for all metrics. At the bottom right, there are buttons for "Add Tests for Missing Coverage" and "Export".

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	CONDITION	EXECUTION
 simpleSwitchWithSubsystemIn/Subsystem1		7	100% 	100% 	100% 

Test Iterations

In this section...

“Create Table Iterations” on page 6-125

“Create Scripted Iterations” on page 6-128

“Sweep Through a Set of Parameters” on page 6-131

You can run the same test case with different data or configuration sets by using test case iterations. Iterations can use different:

- Parameters
- External inputs
- Configuration sets
- Signal Editor scenarios
- Test Sequence scenarios
- Baseline data
- Simulation modes

Set up iterations in the **Iterations** section of a test case. You can create iterations using the table in the Iterations section of the Test Manager or by using a script.

To use Test Sequence scenarios in iterations, first, in the **Inputs** section, set **Test Sequence Block** to the block that contains the scenarios. Then, select a scenario from **Override with Scenario** to use that scenario as the default for each iteration. If you don't select a scenario, the active scenario in the Test Sequence block is used as the default. Use the **Test Sequence Scenario** column in the table to change the scenario for an iteration. For more information, see “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59.

To use different simulation modes, such as normal and software-in-the-loop (SIL), for a baseline or simulation test, first, set up the test case. Then, in the **Iterations** table, click **Auto Generate**. In the Auto Generate Iterations dialog box, select **Simulation Modes** and one or more other options. For each option, the number of iterations created is doubled, one for the mode of the model and one for SIL mode.

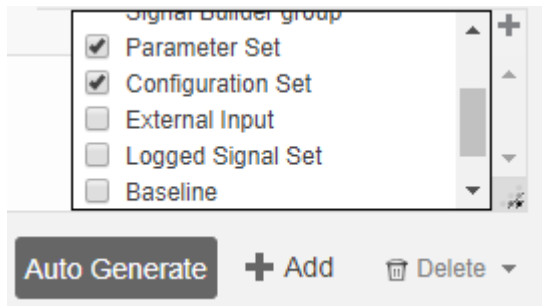
If the test collects coverage using Simulink Coverage, the same coverage settings apply to all iterations in the test case.

Whether you use table or scripted iterations, you can see the iterations in the test case by clicking the **Show Iterations** button.

Create Table Iterations

Table Iterations provide a quick way to add iterations based items in your model or test case. To create iterations with the table, first make the appropriate columns visible:

- 1 Expand the **Iterations > Table Iterations** section.
- 2 In the table, add or remove columns by clicking the **+** button and selecting items in the list. For example, to select display parameter and configuration sets, select the **Parameter Set** and **Configuration Set** items.



Add Iterations Manually

- 1 To manually add iterations, click **Add**. The table displays a new iteration row.
- 2 Assign an iteration name and select items for the iteration. For example, this test case has four iterations. Each iteration uses a different combination of external input and baseline data.

TABLE ITERATIONS*		
<input checked="" type="checkbox"/> NAME	EXTERNAL INPUT	BASELINE
<input checked="" type="checkbox"/> Passing	BrakeThrottle_InputData.xlsx	BrakeThrottleBaseline1.mat
<input checked="" type="checkbox"/> GradualAccel	BrakeThrottle_InputData.xlsx (1)	BrakeThrottleBaseline2.mat
<input checked="" type="checkbox"/> HardBrake	BrakeThrottle_InputData.xlsx (2)	BrakeThrottleBaseline3.mat
<input checked="" type="checkbox"/> Coast	BrakeThrottle_InputData.xlsx (3)	BrakeThrottleBaseline4.mat

Generate Table Iterations

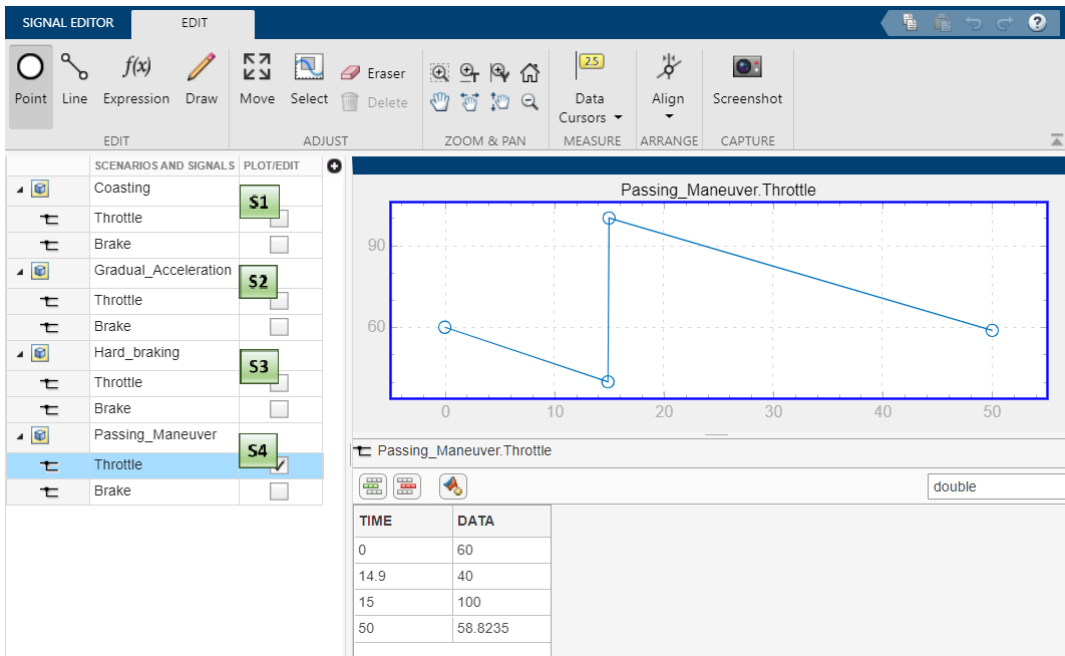
You can also automatically generate iterations from data in your test case and model:

- 1 Click the **Auto Generate** button.
- 2 Select items for which to generate iterations.

For Test Sequence scenarios, an iteration is generated for each scenario for the block you selected in the **Test Sequence Block** in the **Inputs** section.

If you select multiple items, iterations are created in sequential pairings. For example:

- The model `sldemo_autotrans` has a Signal Editor block with four signal scenarios, labeled Coasting, Gradual_Acceleration, Hard_braking, and Passing_Maneuver, each of which has Throttle and Brake signals. To open this model, type `openExample('sldemo_autotrans')` at the command line. To view the Signal Editor, double-click the ManeuversGUI block to open the Block Parameters dialog. Then, click the Signal Editor launch button under Signal Properties.
- The test case has three parameter sets, labeled P1, P2, and P3.
- Automatically generating iterations from Signal Editor scenarios and parameter sets results in three iterations. The iterations are limited by the three parameter sets. Each iteration contains one Signal Editor scenario and one parameter set. The Signal Editor scenario and parameter set are matched in the order that they are listed in the Signal Editor block and parameter set section.



PARAMETER OVERRIDES

PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
<input checked="" type="checkbox"/> Parameter Set 1	P1		
<input type="checkbox"/> Parameter Set 2	P2		
<input type="checkbox"/> Parameter Set 3	P3		

ITERATIONS

TABLE ITERATIONS

NAME	DESCRIPTION	REQUIREMENTS	SIGNAL BUILDER GROUP	PARAMETER SET
<input checked="" type="checkbox"/> Iteration1	None	None	Passing Maneuver	Parameter Set 1
<input checked="" type="checkbox"/> Iteration2	None	None	Gradual Acceleration	Parameter Set 2
<input checked="" type="checkbox"/> Iteration3	None	None	Hard braking	Parameter Set 3

Auto Generate + Add Refresh Delete

3 Specify an optional naming rule for the iterations. In the **Iteration naming rule** box, enter the rule using:

- The name of each setting you want to use in the name, with spaces removed
- An underscore or space to separate each setting

For example, if you want to include the name of the parameter set, configuration set, and baseline filename, enter `ParameterSet_ConfigurationSet_Baseline`.

Section Option	Purpose
Signal Editor scenario	Applies to the Inputs section of a simulation, baseline, or equivalence test case, for the specified Signal Editor Scenario . Each Signal Editor scenario is used to generate an iteration.

Section Option	Purpose
Parameter Set	Applies to the Parameter Overrides section of a simulation, baseline, or equivalence test case. Each parameter override set is used to generate an iteration.
External Input	Applies to the Inputs section of a simulation, baseline, or equivalence test case, for the specified External Inputs sets. Each external input set is used to generate an iteration.
Configuration Set	Applies to the Configuration Setting Overrides section of a simulation, baseline, or equivalence test case. Each iteration uses the configuration setting specified.
Logged Signal Set	Applies to the Logging section of a simulation, baseline, or equivalence test case. Each logged signal set is used to generate an iteration.
Baseline	Applies only to baseline test case types, specifically to the Baseline Criteria section of a baseline test case. Each baseline criteria set is used to generate an iteration.
Test Sequence scenario	Applies to the Inputs section of a simulation, baseline, or equivalence test case, for the specified Test Sequence Block . Each Test Sequence scenario is used to generate an iteration.
Simulation Modes	Applies to the Iterations table section of a simulation or baseline test case. A iteration for the current simulation mode of the model and a SIL iteration are created for each other test setting selected in the Auto Generate Iterations dialog box.
Simulation 1 or 2	Applies only to equivalence test case types. At the top of the Auto Generate Reports dialog box, there is a menu for Simulation 1 or Simulation 2 . These sections correspond to the two simulation sections within the equivalence test case.

Create Scripted Iterations

You can run a custom set of iterations using a script in the **Scripted Iterations** section. For example, you can define parameter sets or customize iteration order by using a custom iteration. Scripted iterations are generated at run time when a test executes and run before loading the model.

▼ SCRIPTED ITERATIONS*

▶ Help on creating test iterations:

```

1
2 %% Iterate over all External Inputs.
3
4 % Determine the number of possible iterations
5 numSteps = length(sltest_externalInputs);
6
7 % Create each iteration
8 for k = 1 : numSteps
9     % Set up a new iteration object
10    testItr = sltestiteration();
11
12    % Set iteration settings
13    setTestParam(testItr, 'ExternalInput', sltest_externalInputs{k});
14
15    % Register the iteration to run in this test case
16    addIteration(sltest_testCase, testItr); % You can pass in an optional iteration name
17 end
18

```

Iteration Templates

Generate an iteration script using templates

Iteration Script Components

An iteration script must contain certain components. The most basic iteration script contains three elements:

- 1 An iteration object, created using `sltestiteration`.
- 2 An iteration setting, set using `setTestParam`.
- 3 The iteration registration, added using `addIteration`.

For example, this script creates an iteration that runs one signal scenario from a Signal Editor block.

```

%% Iterate Using a Signal Editor Scenario

% Set up a new iteration object
testItr = sltestiteration;

% Set iteration setting using Signal Editor scenario
setTestParam(testItr, 'SignalEditorScenario', ...
    sltest_signalEditorScenarios{1});

% Add the iteration to run in this test case
% The predefined sltest_testCase variable is used here
addIteration(sltest_testCase, testItr);

```

For more information about the test iteration class, see `sltest.testmanager.TestIteration`. You can iterate over multiple items, such as Signal Editor or Test Sequence scenarios. You can iterate over all Signal Editor or Test Sequence scenarios in the block by putting the basic iteration script in a loop:

```
%% Iterate Over All Signal Editor Scenarios
```

```

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

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

    % Set iteration settings
    setTestParam(testItr, 'SignalEditorScenario', ...
        sltest_signalEditorScenarios{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_signalEditorScenarios` — Name of Signal Editor scenarios, defined as a 2-D cell array of character vectors.
- `sltest_loggedSignalSets` — Name of logged signal sets, defined as a 2-D cell array of character vectors.
- `sltest_testSequenceScenarios` — — Name of test sequence scenarios, defined as a 2-D cell array of character vectors.
- `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 Editor scenarios, 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 6-125.

For example, if you want to run all Signal Editor scenarios 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**.

Sweep Through a Set of Parameters

Scripted iterations can be used to test a model by sweeping through a set of parameters. You can use this script to try different values for the model workspace parameter `Iei` and model parameter `UpperSaturationLimit` in the model `sltestCar`. Add the script under **Iterations > Scripted Iterations**.

```
%% Iterate over Iei parameter

% Set up the parameter values to sweep over
IeiValues = [0.021,0.022,0.022,0.023];
UprSatValues = [2000,3000,4000,5000];
numSteps = length(IeiValues);

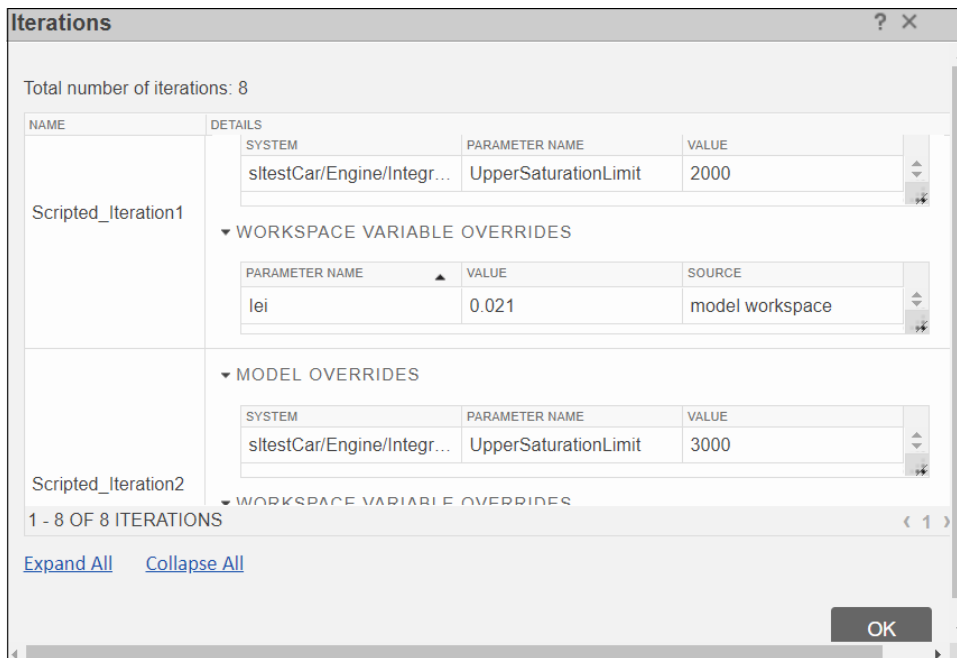
% Create each iteration
for k = 1 : numSteps
    % Set up a new iteration object
    testItr = sltestiteration;

    % Set value of Iei (parameter in model workspace)
    setVariable(testItr,'Name','Iei','Source','model workspace',...
        'Value',IeiValues(k));

    % Set value of UpperSaturationLimit model parameter
    testItr.setModelParam('sltestCar/Engine/Integrator',...
        'UpperSaturationLimit',UprSatValues(k));

    % Add the iteration to run in this test case
    addIteration(sltest_testCase,testItr);
end
```

After you add the script, click **Show Iterations**. You can see the iterations that the script created.



Running the test generates a result for each iteration.

Test Browser		Results and Artifacts
Filter results by name or tags, e.g. tags: test		
NAME	STATUS	
▼ Results: 2018-Aug-10 16:04:14	4	
▼ New Test Case 3	4	
▶ Scripted_Iteration1		
▶ Scripted_Iteration2		
▶ Scripted_Iteration3		
▶ Scripted_Iteration4		

See Also

`sltest.testmanager.TestIteration | setModelParam`

Related Examples

- “Capture Baseline Data from Iterations” on page 6-133
- “Create and Run Test Cases with Scripts” on page 6-107
- “Programmatically Create and Run Test Sequence Scenarios” on page 3-56
- “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59

Capture Baseline Data from Iterations

This example shows how to create a baseline test by capturing data from a test case with table iterations. The iterations are create from the Signal Editor scenarios in the `sltestcar` model.

1. Open the model.

```
Model = 'sltestCar';
open_system(Model);
```

2. Open the Test Manager.

```
sltestmgr
```

3. In the Test Manager, click **Test File from Model** from the **New** dropdown.

4. Specify the test file.

- 1 Enter `sltestCar` as the **Model**.
- 2 Enter a test filename or full path in **Location**.
- 3 Select **Baseline** as the **Test Type**.

5. Select the test case by expanding the test file and selecting the **sltestCar/Inputs** test case.

6. Select the signals for the baseline data:

- 1 In the **Simulation Outputs** section, click **Add**.
- 2 In the model canvas, select the output `torque` signal and in the Connect dialog, check the box for that signal. Select the `vehicle_speed` signal and check its box in the dialog.
- 3 In the Test Manager message dialog box, click **Done**.
- 4 The signals appear in the **Logged Signals** table.

7. To view the iterations for the test case, expand the **Iterations** and **Table Iterations** sections. The iterations for the selected test case automatically appear. The iterations correspond to the four Signal Editor scenarios.

8. Capture baseline data for the iterations.

- 1 In the **Baseline Criteria** section, click the arrow next to **Capture**.
- 2 Select **MAT** as the **File format**.
- 3 Specify the location to save the baseline data files in the **File** field.
- 4 Select **Capture Baselines for Iterations**.
- 5 Click **Capture**.

The model simulates for all Signal Editor scenarios. 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	LEADING TOL	LAGGING TOL	+
▶ <input type="checkbox"/> Coasting.mat	0	0.00%	0	0	
▶ <input type="checkbox"/> Gradual_Acceleration.mat	0	0.00%	0	0	
▶ <input type="checkbox"/> Hard_braking.mat	0	0.00%	0	0	
▶ <input type="checkbox"/> Passing_Manuever.mat	0	0.00%	0	0	

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

▼ ITERATIONS* ?

▼ TABLE ITERATIONS* ?

<input checked="" type="checkbox"/> NAME	DESCRIPTION	SIGNAL EDITOR SCENA... ▲	BASELINE	+
<input checked="" type="checkbox"/> Coasting	None	Coasting	Coasting.mat	▲
<input checked="" type="checkbox"/> Gradual_Acceleration	None	Gradual_Acceleration	Gradual_Acceleration.mat	
<input checked="" type="checkbox"/> Hard_braking	None	Hard_braking	Hard_braking.mat	
<input checked="" type="checkbox"/> Passing_Manuever	None	Passing_Manuever	Passing_Manuever.mat	▼

Auto Generate + Add 🗑 Delete ▼

```
sltest.testmanager.close
close_system(Model,0)
```

See Also

Related Examples

- “Test Iterations” on page 6-125
- “Creating Baseline Tests” on page 6-10
- “Create a Simple Baseline Test”

Collect Coverage in Tests

In this section...

“Set Up Coverage Collection Using the Test Manager” on page 6-135

“View Coverage Results in the Test Manager” on page 6-137

“Add Tests for Missing Coverage” on page 6-139

“Coverage Filtering Using the Test Manager” on page 6-140

Coverage refers to determining the testing completeness of models and generated code by analyzing how much of the model has been exercised. To collect coverage using the Simulink Test Test Manager or `sltest.testmanager.CoverageSettings`, you must have Simulink Coverage installed. Although you can set up and run test cases using only Simulink Coverage, Simulink Test provides additional test creation and test management features. For tests with coverage collection turned on, the Test Manager includes the coverage of each metric you choose to collect in the results. If you have Requirements Toolbox installed, you can also use the Test Manager to verify that coverage results are traced to specific requirements.

Note Coverage is supported for Model Reference blocks, atomic Subsystem blocks, and top-level models configured for Software-in-the-Loop (SIL) or Processor-in-the-Loop (PIL). Coverage is not supported for SIL or PIL models in subsystems.

For information on considerations when collecting coverage in a test harness, see Test Harness Considerations in “Test Harness and Model Relationship” on page 2-2.

Set Up Coverage Collection Using the Test Manager

In the Test Manager, you can enable coverage and select the coverage metrics at the test file level. Test suites and test cases inherit the coverage settings from the test file. You can turn off coverage collection for individual test suites and test cases. However, you cannot turn off coverage at the test suite or test case level for MATLAB-based Simulink tests. For information about MATLAB-based Simulink tests, see “Test Models Using MATLAB-Based Simulink Tests” on page 6-111.

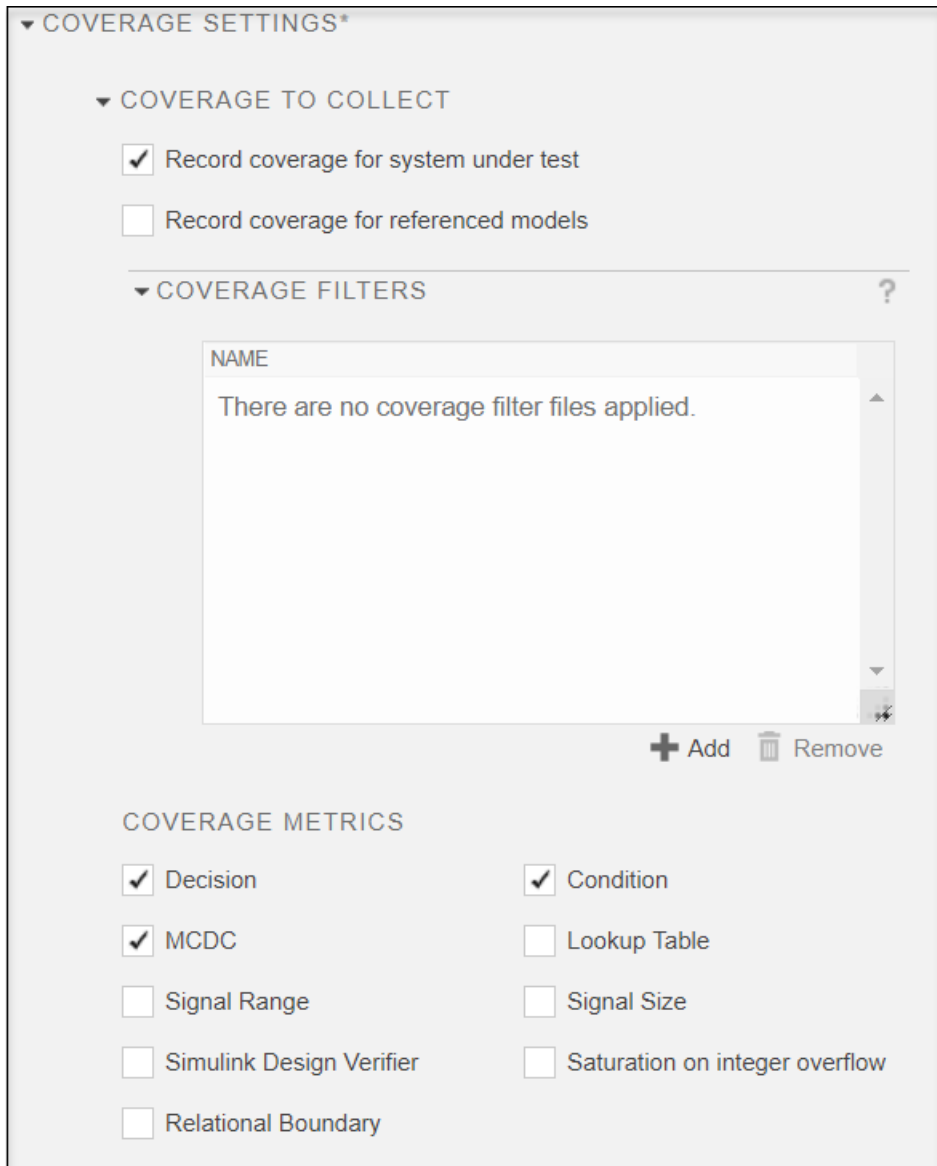
Note The Test Manager uses its coverage settings and not the coverage settings from the model Configuration Parameters.

To set up the Test Manager to include coverage collection:

- 1 Create a test file and set up a test case for your model.
- 2 Select the test file and expand the **Coverage Settings** section. Under **Coverage to Collect**, select the coverage:
 - **Record coverage for system under test** — Collects coverage for the model or, when included, the component specified in the “System Under Test” on page 6-160 section for each test case. If you are using a test harness, the system under test is the component for which the harness is created. The test harness is not the system under test.
 - For a block diagram, the system under test is the whole block diagram.

- For a Model block, the system under test is the referenced model.
- For a subsystem, the system under test is that subsystem.
- **Record coverage for referenced models** — Collects coverage for models that are referenced from within the specified system under test. If the test harness references another model, the coverage results are included for that model, too.

The selected coverage settings propagate from the test file to the test suites and test cases in the test file.



▼ COVERAGE SETTINGS*

▼ COVERAGE TO COLLECT

Record coverage for system under test

Record coverage for referenced models

▼ COVERAGE FILTERS ?

NAME

There are no coverage filter files applied.

+ Add Remove

COVERAGE METRICS

Decision Condition

MDCD Lookup Table

Signal Range Signal Size

Simulink Design Verifier Saturation on integer overflow

Relational Boundary

- 3 Optionally, to add or remove existing coverage filter files, click **Add** or **Remove**, respectively, in the Coverage Filters section and select the filter file. More than one filter file can be applied at the same time.
- 4 Select the coverage metrics to collect. For information on metrics, see “Types of Model Coverage” (Simulink Coverage) and “Model Objects That Receive Coverage” (Simulink Coverage).

- Run the test. Coverage is collected for the test suites and test cases in the test file.

To turn off coverage collection for a test suite or test case, select the test suite or test case and then, deselect the **Coverage to Collect** option. To turn off one or more types of coverage collection, deselect them from the **Coverage Metrics** section. For MATLAB-Based Simulink tests, you can change the coverage collection and coverage metrics only at the test file level.

View Coverage Results in the Test Manager

View Aggregated Coverage Results and Metrics

After you collect coverage, use the **Results and Artifacts** pane in the Test Manager to view the results. Coverage results aggregated for all test cases are reported in results sets. Filtered coverage results are shown only at the Results level, and not at the test file, test suite, or test case level. When analyzing filtered coverage results, viewing them at the Results level provides overall coverage information. For example, if coverage is less than 100% for a specific test case, the missing coverage might be included in a different test case.

Select a **Results** item in the pane and expand the **Aggregated Coverage Results** section. The coverage percentage is shown for each metric and the colors summarize the coverage results.

- Dark blue — Satisfied coverage
- Red — Unsatisfied coverage
- Light blue — Justified coverage

The screenshot shows the Test Manager interface with the 'Results and Artifacts' pane active. The left sidebar shows a tree view with 'Results: 2020-Jul-16 10:42:39' selected, which has a status of '1' with a green checkmark. Below it is 'covtest' with a green checkmark. The main pane displays the 'SUMMARY' section for the selected results set, showing a 'Name' of 'Results: 2020-Jul-16 10:42:39', an 'Outcome' of '1' with a green checkmark, and 'Start Time' and 'End Time' of '07/16/2020 10:42:46'. Below the summary is the 'AGGREGATED COVERAGE RESULTS' section, which includes a table with the following data:

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	CONDITION	MCDC	EXECUTION
slvndemo_covfilt		29	48%	60%	50%	74%

At the bottom of the pane, there are checkboxes for 'Scope coverage results to linked requirements' and 'Add Tests for Missing Coverage', and an 'Export' button.

To aggregate results from different test files into a single result set, select the separate results in the **Results and Artifacts** list. Then, from the context menu, select **Merge Coverage Results**. A results set that contains the combined coverage results appears in the list.

Scoping Coverage for Requirements-Based Tests

For requirements-based design and testing, such as for compliance to DO-178B, enable **Scope coverage results to linked requirements** to check that your model design is executing the

requirements and that the tests are verifying those requirements. Both Simulink Coverage and Requirements Toolbox licenses are required. This option is available only if the results set contains more than one simulation, such as multiple test cases or iterations.

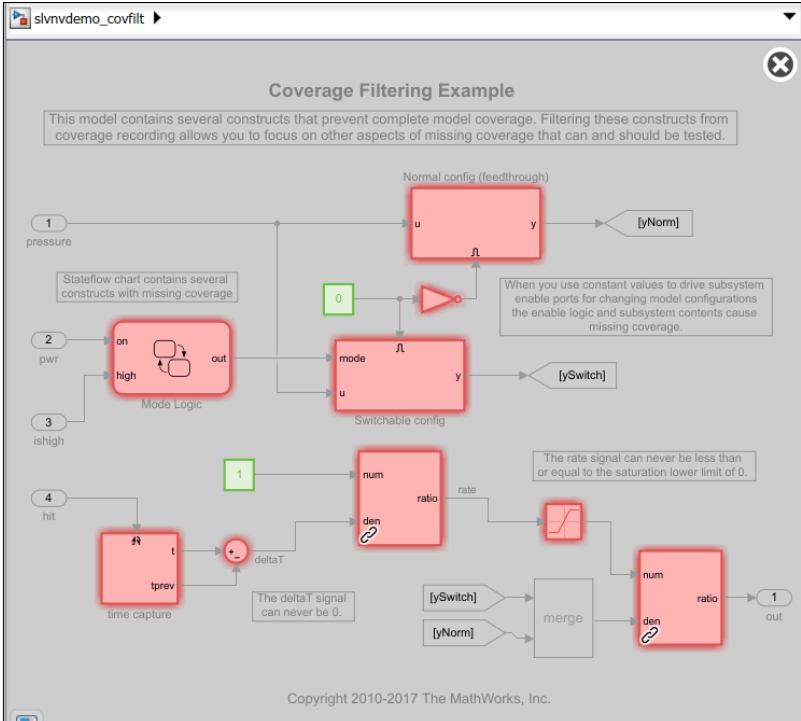
When the **Scope coverage results to linked requirements** check box is selected, coverage results include only tests that are directly linked to requirements and are explicitly tested. The aggregated results update automatically without having to resimulate the model. If you have tests that touch a model component but are not directly linked to a requirement, your aggregated coverage results percentages might decrease when you enable scoping. To obtain 100% coverage to your requirements, you might need to update your tests, add requirements links, or justify or exclude some items from coverage.

Trace Coverage Results to the Model

To navigate from the test coverage results in the Test Manager to the model, click the model name in the Aggregated Coverage Results table.

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	CONDI...
 slnvdemo_covfilt		29	48%	 60%

The model opens, and its Coverage Report opens in the Coverage Details pane of the model window. In this sample model, the model elements are red because they have less than 100% coverage.



Copyright 2010-2017 The MathWorks, Inc.

Coverage Report for slnvdemo_covfilt

Table of Contents

1. [Analysis Information](#)
2. [Tests](#)
3. [Summary](#)
4. [Details](#)

Analysis Information

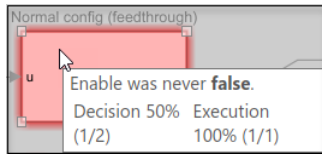
Model Information

Model version	1.25
Author	The MathWorks, Inc.
Last saved	Mon Dec 16 10:38:29 2019

Harness information

Harness model(s)	slnvdemo_covfilt_Harness1
Harness model owner	slnvdemo_covfilt

Point to a model element to see a summary of its metrics and block execution.



Click a model element to scroll to its detailed coverage results information in the **Coverage Details** pane.

7. SubSystem block "Switchable config"

[Justify or Exclude](#)

Parent: [/slvnydemo_covfilt](#)

Uncovered Links:

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	2	5
Decision	50% (1/2) decision outcomes	17% (1/6) decision outcomes
Execution	NA	20% (1/5) objective outcomes

Decisions analyzed

enable logical value	50%
false	101/101
true	0/101

Create a Coverage Report

To create a report of the coverage for a model, click the arrow in the **Report** column of the **Aggregated Coverage Results** table.

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION
slvnydemo_covfilt		29	36%

Add Tests for Missing Coverage

If you have a Simulink Design Verifier license, you can generate additional test cases to increase the coverage in your models.

In the Test Manager,

- 1 In the Test Manager, select the **Results and Artifacts** pane.
- 2 Select the **Results** item for which you want to collect more coverage.
- 3 In the right pane, in the **Aggregated Coverage Results** section, select the line with incomplete coverage in the table.

- 4 At the bottom of the **Aggregated Coverage Results** section, click **Add Tests for Missing Coverage**.
- 5 In the Add Test for Missing Coverage dialog box,
 - **Harness** — Select whether to use the existing harness or create a new harness
 - **Source** — Select the source of the inputs to the harness. If you use an existing harness, the **Source** field is read only.
 - **Test Case** — Select whether to use the existing test case or create a new test case. If you create a new harness, the only option is to use a new test case.
 - **Test Type** — Select the type of test to use for the new test case. This field is displayed if you select to create either a new harness or a new test case.
 - **Test File** — Select whether to use the existing test file or create a new test file. This field is displayed if you select to create either a new harness or a new test case.
 - **Location** — If you select to create a new test file, specify the path and name of the test file.
- 6 Click **OK** to generate test cases that add the missing coverage.
- 7 If you created a new test case or new harness, in the **Test Browser** pane, drag and drop the test case into the test suite that contains the original test case.
- 8 Rerun the test suite.

For a complete example of how to increase test coverage in the Test Manager, see “Increase Test Coverage for a Model” on page 6-147.


Alternatively, you can create and use tests to increase coverage programmatically by using `sltest.testmanager.addTestsForMissingCoverage` and `sltest.testmanager.TestOptions`.

Coverage Filtering Using the Test Manager

Coverage filter rules specify one or more model objects or lines of generated code to exclude from coverage collection or for which you want to justify the coverage results. A set of coverage filter rules is contained in a filter file, which can be applied to the model or code being tested. You can apply more than one filter file to a test and also, reuse filter files for different models. When you apply a new or updated filter, the aggregated coverage results, which are shown for a result set, update automatically. You do not have to resimulate your model. For more information, see “Coverage Filtering” (Simulink Coverage).

To view the filtered coverage results, select a result set (that is, a **Results** item) in the **Results and Artifacts** pane.

From the Test Manager, you can:

- Add or remove an existing coverage filter file — In the **Test Browser** pane, select the test file and expand the **Coverage Settings** section. Click **Add** or **Remove** at the bottom of the **Coverage Filters** or **Applied Coverage Filters** table and select the coverage filter file to add or remove, respectively. More than one coverage filter file can be applied to the coverage results.
- Edit or create a filter file, define a filter rule, and justify or exclude coverage — From a **Coverage Report** or the **Coverage Details** pane of a model, open the Simulink Coverage Filter Editor by clicking on a justify icon  or a **Justify** or **Exclude** link. When the Filter Editor is open, the Test Manager is locked. When you close the Filter Editor, the Test Manager is enabled and the

results and applied filters list are updated with your changes. For information on using the Filter Editor, see “Creating and Using Coverage Filters” (Simulink Coverage) and “Create, Edit, and View Coverage Filter Rules” (Simulink Coverage).

- Append currently applied coverage filters to the test file — Click **Update Test File**.
- View coverage results — Select a **Results** item in the **Results and Artifacts** pane and expand the **Aggregated Coverage Results** section.

For more information on coverage filters, rules, and files, see the Coverage Filtering topics in “Analyze Coverage and View Results” (Simulink Coverage).

See Also

`sltest.testmanager.CoverageSettings` |
`sltest.testmanager.addTestsForMissingCoverage` | `sltest.testmanager.TestOptions`

More About

- “Analyze Coverage and View Results” (Simulink Coverage)
- “Model Objects That Receive Coverage” (Simulink Coverage)
- “Perform Functional Testing and Analyze Test Coverage” on page 10-9
- “Test Coverage for Requirements-Based Testing” on page 6-142
- “Assess Coverage Results from Requirements-Based Tests” (Simulink Coverage)
- “Trace Coverage Results to Requirements” (Simulink Coverage)
- “Requirements-Based Testing” (Requirements Toolbox)
- “Add Tests for Missing Coverage” on page 7-29
- “Collect Coverage Using MATLAB-Based Simulink Tests” on page 6-120

Test Coverage for Requirements-Based Testing

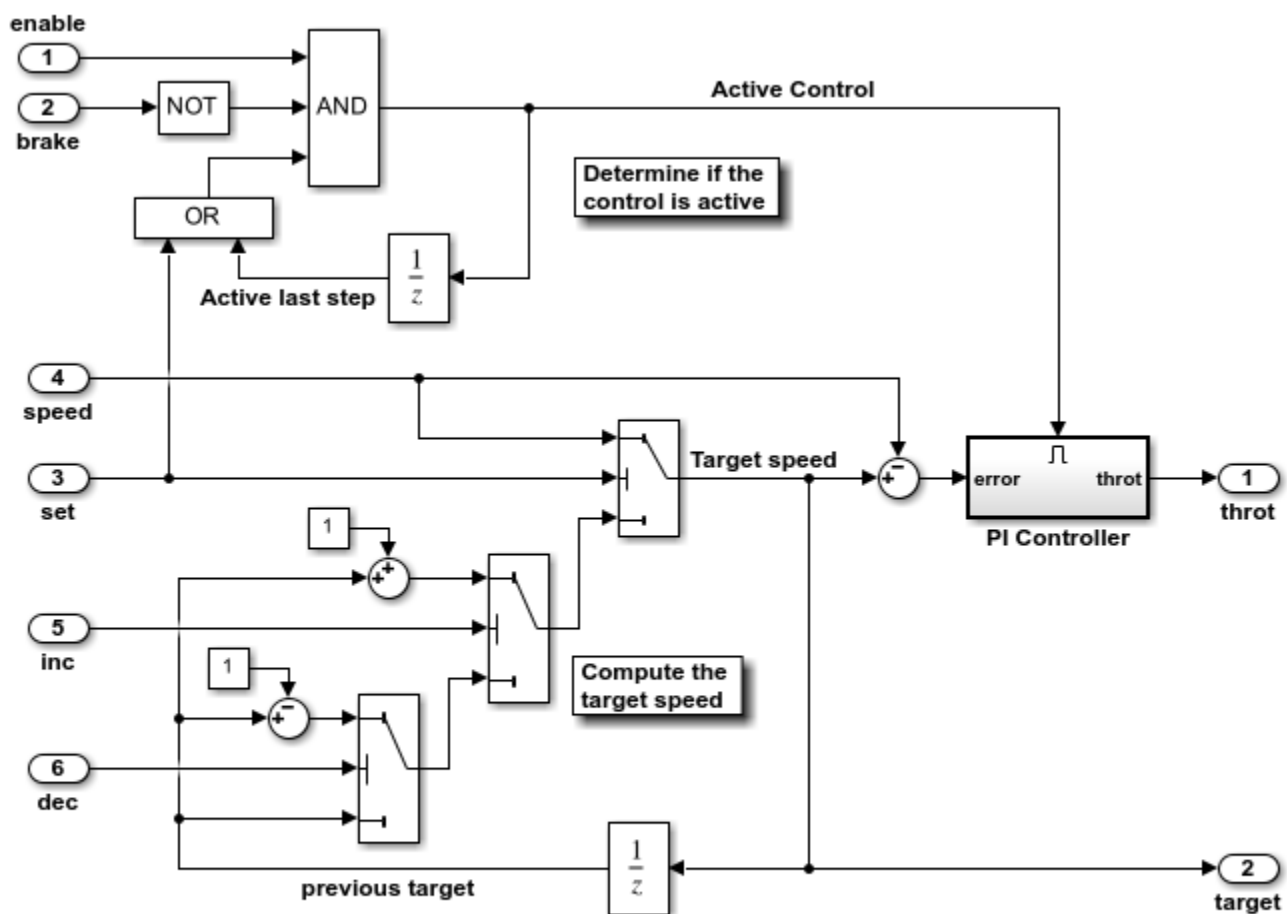
This example shows how to collect test coverage for a model that implements requirements. Coverage refers to determining the testing completeness by analyzing how much of the model logic is exercised. For requirements-based testing, coverage results can be scoped to linked requirements. With this scoping you can assess if each model element is covered by the intended test case.

The example shows how scoping coverage results to linked requirements can reveal both inadequate requirement linking and testing gaps. It also shows how to increase the coverage.

The model in this example is `cruiseControlRBTCovExample`, which represents a cruise control system. This model implements and is linked to requirements. A test file has already been created for this example.

Open the Cruise Control Model

`cruiseControlRBTCovExample`



Copyright 2006-2020 The MathWorks, Inc.

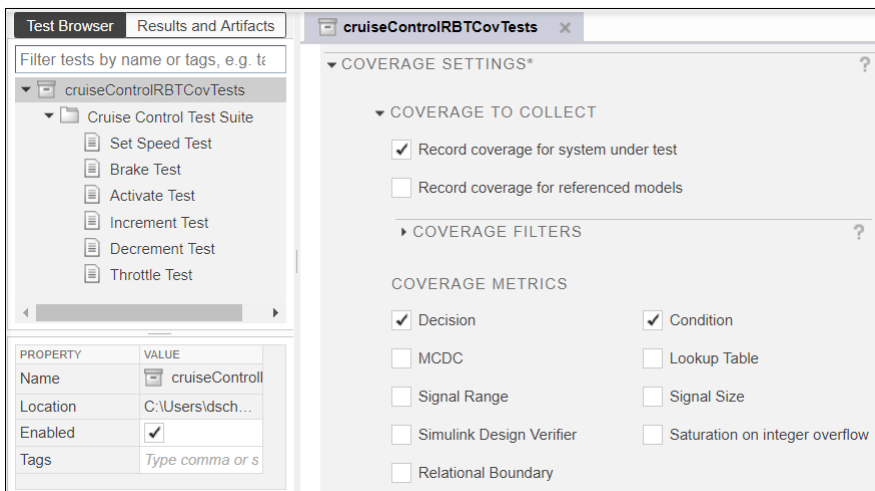
View the Linked Requirements

The requirements for this cruise control system have been captured in the Requirements Editor. To view the requirements, use `slreq.open('cruiseControlRBTCovReqs.slreqx')`.

Open the Test Manager and Test File

Use `sltestmgr` to open the Test Manager.

Click **Open** and select `cruiseControlRBTCovTests.mldatx`. The tests have been written to verify that the model behavior meets the specified requirements. They have also been set up to record Decision and Condition coverage. Expand Coverage Settings to see the selected metrics.



Each test case verifies and is linked to a requirement. For example, the Throttle Test verifies the THROTTLE requirement. This requirement specifies that the throttle is applied smoothly if the speed differs from the target. The test verifies this behavior using a logical assessment, which checks that the throttle rate of change is between -1 and 1 radians per second, as defined in the requirement description.

Run the Test and View Coverage Results

Run the test.

Click on Results in the Results and Artifacts pane when the test finishes running. Note that the tests pass and that 100% aggregated coverage is reported.

The screenshot shows the 'AGGREGATED COVERAGE RESULTS' table with the following data:

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	CONDITION	EXECUTION
cruiseControlRBTCovExample		8	100%	100%	100%

Below the table, there is a note: "Create a coverage report from coverage results to justify or exclude missing coverage. The filters and updated coverage values will be displayed with this result."

Turn on Scoping the Test Results to Linked Requirements

Click the top-level Results in the Results and Artifacts pane. Then, in the Aggregated Coverage Results pane, click the Scope coverage results to linked requirements check box. Scoping the results means that each test only contributes coverage for the corresponding model elements that implement the requirement verified by that test. Scoping checks that model elements are covered by the intended test cases. The coverage results, which update automatically, now show aggregated coverage for Decision and Execution at 92% and 76%, respectively.

▼ AGGREGATED COVERAGE RESULTS ?

Create a coverage report from coverage results to justify or exclude missing coverage. The filters and updated coverage values will be displayed with this result.

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	CONDITION	EXECUTION	+
cruiseControlRBTcovExample		8	92% <div style="width: 92%; height: 10px; background-color: blue; border: 1px solid red;"></div>	100% <div style="width: 100%; height: 10px; background-color: blue; border: 1px solid red;"></div>	76% <div style="width: 76%; height: 10px; background-color: blue; border: 1px solid red;"></div>	^

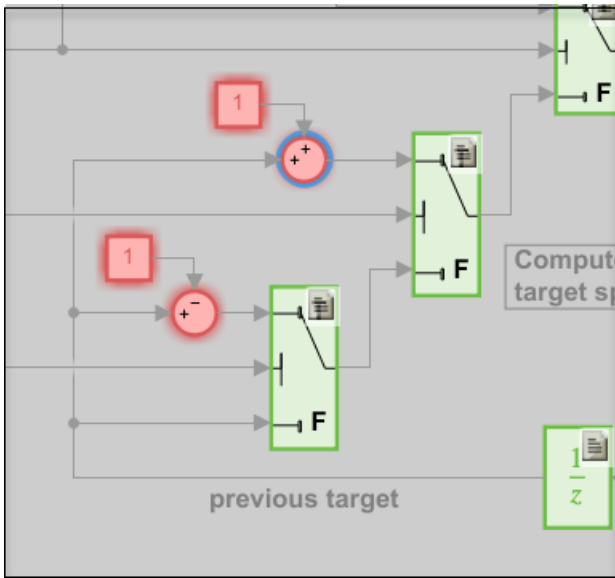
Scope coverage results to linked requirements
 + Add Tests for Missing Coverage
Export

View the Coverage Results in the Model

Click on the model name in the Analyzed Model column to highlight the coverage results in the model and display the Coverage Report details.

In the model, if the Requirements table is not shown below the model, open it by clicking the Perspectives views in the lower right corner of the model canvas and then, clicking Requirements.

Open the Controller subsystem. Blocks that do not have 100% coverage appear in red. Two sets of Constant and Sum blocks are not linked to requirements and were never executed.



Link Blocks to Requirements

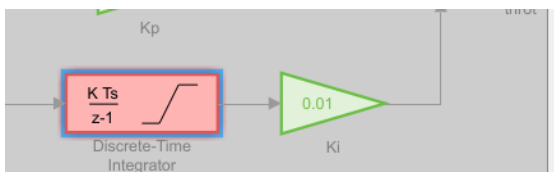
In this case, the missing coverage indicates insufficient requirements linking. These Constant and Sum blocks are necessary for implementing the INCREMENT and DECREMENT requirements and should be linked to the appropriate requirements.

In the table in the Requirements pane, expand `cruiseControlRbtCovReqs`. Right-click on the upper Constant block and select **Requirements > Link to Selection in Requirements Browser**. Then, click on the INCREMENT requirement in the Requirements table. Repeat this for the upper Sum block.

For the lower Constant and Sum blocks, repeat the linking steps, but link to the DECREMENT requirement.

Increase Coverage from a Specific Test

Open the PI Controller and click on the Discrete-Time Integrator block. The Coverage Details show that the `true` decision for the upper limit was executed by the Increment Test (T4), rather than the Throttle Test (T6). Since the block is part of the implementation of the THROTTLE requirement, it should have been tested by the Throttle Test, which verifies the THROTTLE requirement. The Increment Test does not verify this requirement and does not contribute coverage for this block when the `Scope model coverage to linked requirements` setting is enabled.



true	0/801 T4
------	-------------

To resolve the missing coverage for this block, the Throttle Test needs to be updated to exercise the Discrete-Time Integrator block more.

In the Test Browser pane of the Test Manager, select Throttle Test. Under Inputs, select `td_throttle_updated.mat` as the External Inputs file. This updated input throttle data file has

some additional seconds of test data, which increase the target speed more aggressively while maintaining the actual speed.

Select `cruiseControlRBTCovTests` in the Test Browser pane and rerun the test. Click the **Scope coverage results to linked requirements** check box. The coverage results show 100% coverage, which indicates that the tests adequately execute the model.

Revised Test Reveals an Issue in the Design

The revised Throttle Test now fails verification. The failure occurs because the throttle increases too aggressively and is outside the required boundaries specified in the test. This indicates an issue with the model design. The PI Controller block implementation would need to be updated to apply the throttle within the required limits, including when the target and actual speeds differ significantly.

Conclusion

In summary, scoping coverage results to linked requirements can help reveal gaps in testing. Scoping accomplishes this by assessing that each model element is exercised by the test that verifies the corresponding requirement.

See Also

Related Examples

- “Collect Coverage in Tests” on page 6-135
- “Increase Test Coverage for a Model” on page 6-147

Increase Test Coverage for a Model

Increase test coverage by generating test inputs.

If your tests achieve incomplete model coverage, you can increase coverage by generating test inputs using Simulink® Design Verifier™. This example shows how to increase test coverage beyond an initial test case. You measure initial coverage of a test case. Then, you generate new test cases, add them to the test suite, run the tests, and review aggregate coverage.

Workflow

This example tests a component of an autopilot system using a test harness. Time series data from the base workspace is mapped to root inports in the test harness. The test file is configured to collect coverage.

The example workflow is:

- 1 Measure model coverage of the initial test case.
- 2 Generate additional tests to achieve greater coverage.
- 3 Add the new test cases to the test file.
- 4 Run all test cases and review aggregate coverage.

Paths and Example Files

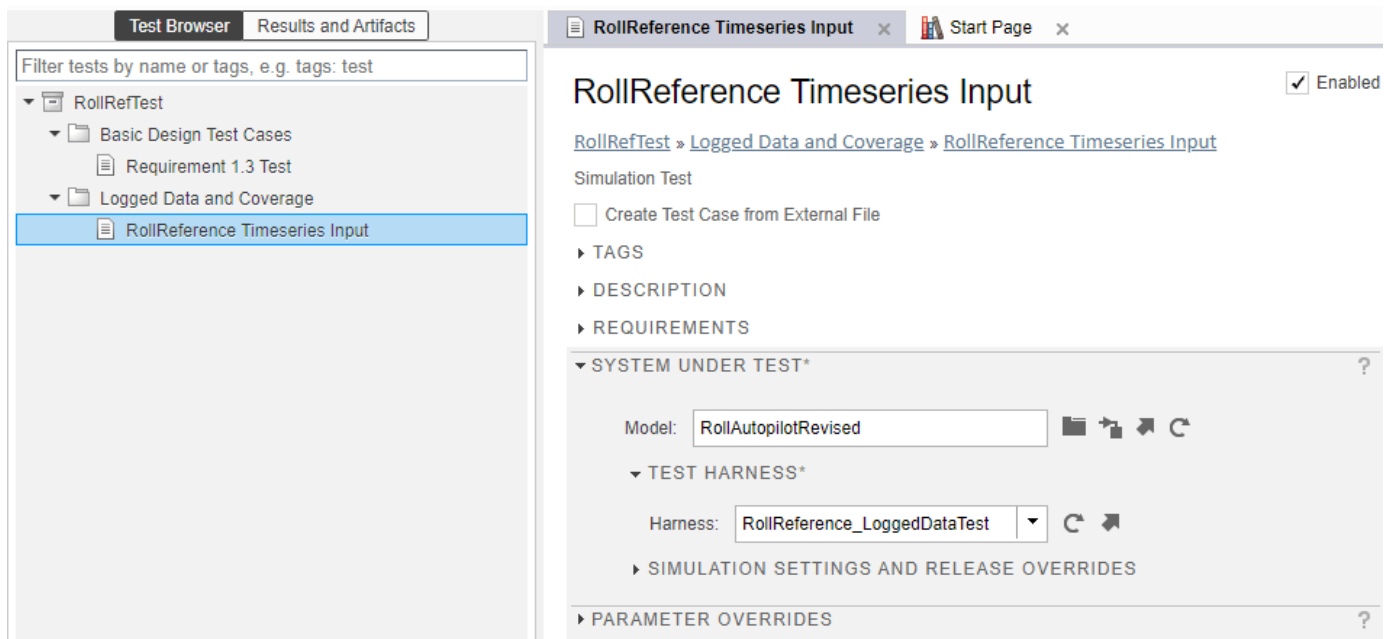
Set paths and filenames for the example.

```
rollModel = 'RollAutopilotRevised';  
testHarness = 'RollReference_LoggedDataTest';  
testFile = 'RollRefTest.mldatx';
```

Run the Initial Test and Review Coverage

1. Open the test file.

```
sltest.testmanager.load(testFile);  
sltest.testmanager.view;
```



2. Run the test. In the **Test Browser**, highlight the **Logged Data and Coverage** test suite. Click **Run**.

3. After the test completes, in the test results, expand the **Coverage Results** section. The test achieves partial coverage for the Roll Reference subsystem.

- Decision coverage: 80%
- Condition coverage: 70%
- MCDC 25%

AGGREGATED COVERAGE RESULTS

Create a coverage report from coverage results to justify or exclude missing coverage. The filters and updated coverage values will be displayed with this result.

ANALYZED MODEL	REPORT	COMPLEX...	DECISION	CONDITION	MCDC	EXECUTION
RollAutopilotRevised/roll Reference		6	80%	70%	25%	100%

Generate Tests to Increase Model Coverage

Generate additional tests for missing coverage.

1. Below the coverage result, click **Add Tests for Missing Coverage**.

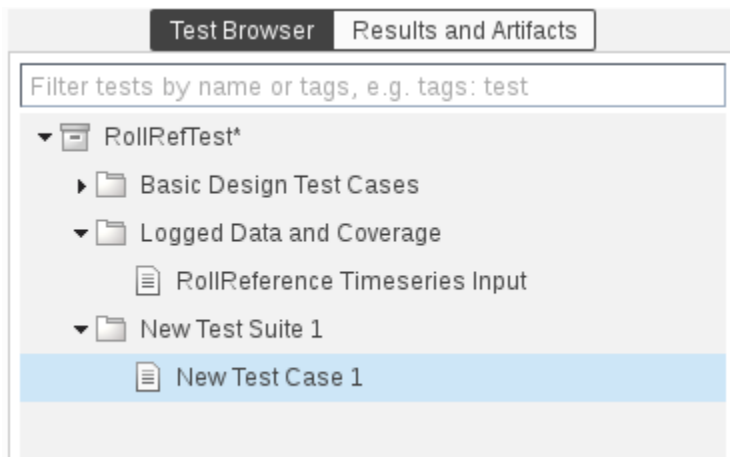
2. In the **Add Tests for Missing Coverage** dialog box, set these options:

- **Harness:** RollReference_LoggedDataTest. This maps the new test inputs to the existing test harness.
- **Test Case:** Create a new test case. This creates a new test case with the generated test inputs.
- **Test Type:** Baseline Test. This gives the option to capture baseline data output from the model for the generated tests.

- **Test File::** RollRefTest. This re-uses the existing test file.

3. Click **OK**. A dialog box shows progress of the test case generation. When test case generation is complete, a new test case appears in the Test Manager.

Alternatively, you can add tests programmatically by using the `sltest.testmanager.addTestsForMissingCoverage` function.



Run the New Test Case

1. Drag and Drop the new test case into the **Logged Data and Coverage** test suite.
2. Run the **Logged Data and Coverage** test suite again.
3. When simulation completes, in the **Results and Artifacts** section, select the result set and expand the **Aggregated Coverage Results**. The test suite achieves complete coverage:
 - Decision: 100%
 - Condition: 100%
 - MCDC: 100%

AGGREGATED COVERAGE RESULTS							
<i>Create a coverage report from coverage results to justify or exclude missing coverage. The filters and updated coverage values will be displayed with this result.</i>							
ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	CONDITION	MCDC	EXECUTION	
RollAutopilotRevised/Roll Reference		6	100%	100%	100%	100%	

Cleanup

Clear variables and test results, and close the model.

```
clear reqDoc rollModel testFile testHarness topModel;
sltest.testmanager.clearResults;
```

```
sltest.testmanager.close;  
close_system('RollAutopilotRevised',0);
```

See Also

Related Examples

- “Collect Coverage in Tests” on page 6-135
- “Test Coverage for Requirements-Based Testing” on page 6-142

Run Tests Using Parallel Execution

In this section...

“When Do Tests Benefit from Using Parallel Execution?” on page 6-151

“Use Parallel Execution” on page 6-151

Running tests in parallel can speed up execution and decrease the amount of time it takes to get test results. If you have a Parallel Computing Toolbox license, you can execute tests in parallel on your local machine or cluster. If you have a MATLAB Parallel Server license, you can execute tests in parallel on a remote cluster, such as in the cloud.

When Do Tests Benefit from Using Parallel Execution?

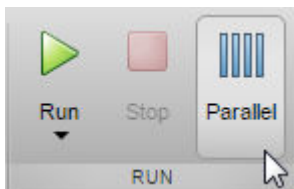
In general, parallel execution can help reduce test execution time if you have

- A complex Simulink model that takes a long time to simulate
- Numerous long-running tests, such as iterations

Use Parallel Execution

To run tests in parallel:

- 1 Set up and open a parallel pool on the desired cluster, or set the desired cluster as the default. If you have a Parallel Computing Toolbox license, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox). If you have a MATLAB Parallel Server license, see “Running Code on Clusters and Clouds” (MATLAB Parallel Server). If you do not set your default cluster or have a parallel pool open, the Test Manager uses its default cluster, which is on the local machine.
- 2 Open the Test Manager.
- 3 On the Test Manager toolstrip, click the **Parallel** button.



- 4 Run a test file. The test file executes using parallel pool.

Note Baseline, equivalence, custom and assessments criteria evaluation occurs on the host MATLAB and not on the parallel MATLAB workers. Before parallel test execution begins, the base workspace variables from the host MATLAB are transferred to the base workspaces of the parallel MATLAB workers. However, after parallel test execution completes, the base workspace variables are not transferred back to the host MATLAB.

- 5 To turn off parallel execution, click the **Parallel** button to toggle it off.

Starting a parallel pool can take time, which would slow down test execution. To reduce time:

- Make sure that the parallel pool is already running before you run a test. By default, the parallel pool shuts down after being idle for a specified number of minutes. To change the setting, see “Specify Your Parallel Preferences” (Parallel Computing Toolbox).
- Load Simulink on all the parallel pool workers.

See Also

`sltest.testmanager.run`

Related Examples

- “Clusters and Clouds” (Parallel Computing Toolbox)

Set Signal Tolerances

In this section...

“Modify Criteria Tolerances” on page 6-153

“Change Leading Tolerance in a Baseline Comparison Test” on page 6-153

You can specify tolerances in the **Baseline Criteria** or **Equivalence Criteria** sections of baseline and equivalence test cases. You can specify relative, absolute, leading, and lagging tolerances for a signal comparison. Leading and lagging tolerances allow you to compensate for differences in time between signals. The units for tolerances are seconds.

To learn about how tolerances are calculated, see “How the Simulation Data Inspector Compares Data”.

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 pass/fail results.

Change Leading Tolerance in a Baseline Comparison Test

Specify a tolerance when the difference between results falls in a range you consider acceptable. Suppose that your model under test uses a particular solver. Solvers are sometimes updated from one release to the next, and new solvers also become available. If you use an updated solver or change solvers, you can specify an acceptable tolerance for differences between your baseline and later tests. Leading and lagging tolerances allow you to re-evaluate criteria if there are differences in time, for example, due to solver the data is off by .04 seconds you can shift it left or right to account for this.

Generate the Baseline

Generate the baseline for the `sf_car` model, which uses the `ode-5` solver.

- 1 Open the `sf_car` model by using `openExample('sf_car')`.
- 2 Open the Test Manager and create a test file named `Solver Compare`. In the test case, set the system under test to `sf_car`.
- 3 Select the signal to log. Under **Simulation Outputs**, click **Add**. In the model, select the `shift_logic` output signal. In the Signal Selection dialog box, select the check box next to `shift_logic` and click **Add**.
- 4 Save the baseline. Under **Baseline Criteria**, click **Capture**. Set the file format to `MAT`. Name the baseline `solver_baseline` and click **Capture**.

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

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
<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 Solvers and Run the Test Case

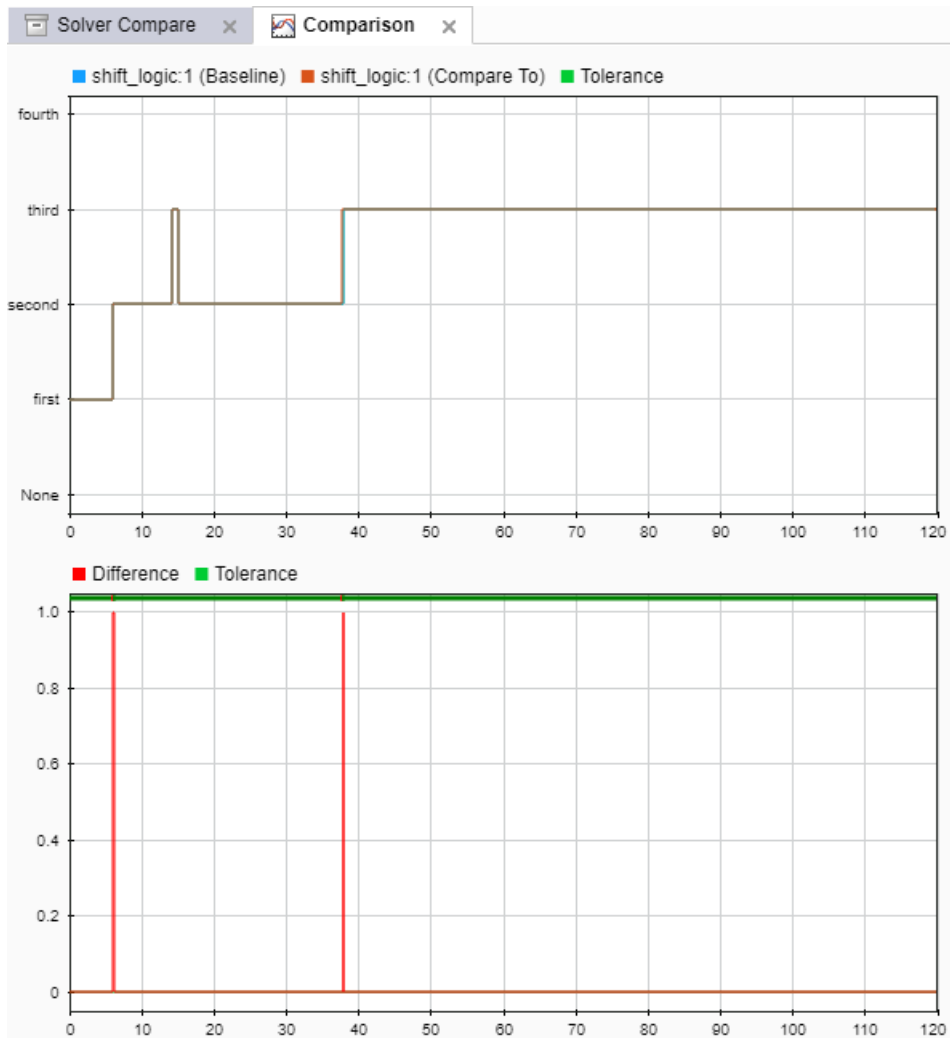
Suppose that you want to use a different solver with your model. You run a test to compare results using the new solver with the baseline.

- 1 In the model, change the solver to ode1.
- 2 In the Test Manager, with the Solver Compare test file selected, click **Run**.

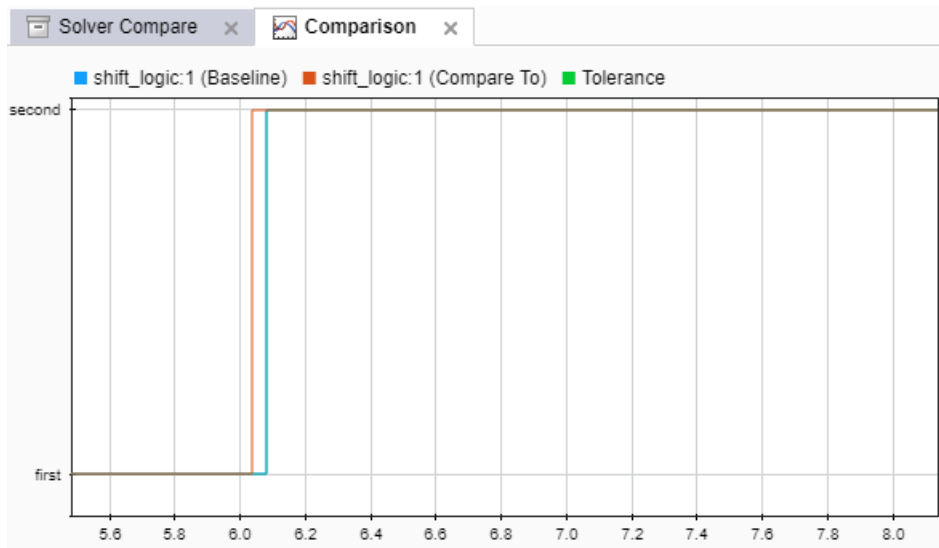
In the **Results and Artifacts** pane, notice that the test failed.

- 3 Expand the results of the failed test. Under **Baseline Criteria Result**, select the shift_logic signal.

The **Comparison** tab shows where the difference occurred.



- 4 Zoom the comparison chart where the results diverged. The comparison signal changes ahead of the baseline, that is, it leads the baseline signal.



Preview and Set a Leading Tolerance Value

You can use leading and lagging tolerances to allow for slight offsets in time between the simulation and baseline data. Suppose that your team determines that a tolerance the size of the simulation step size (.04 seconds in this case) is acceptable. In the Test Manager, set a leading tolerance value. Use a leading tolerance for the signal whose change occurs ahead of your baseline. Use a lagging tolerance for a signal whose change occurs after your baseline.

You can preview how the tolerance value affects the test to see if the test passes with the specified tolerance. Then set the tolerance on the baseline criteria and rerun the test.

- 1 Preview whether the tolerance you want to use causes the test to pass. With the result signal selected, in the property box, set **Leading Tolerance** to .04.

PROPERTY	VALUE
Name	<input checked="" type="checkbox"/> shift_logic:1
Status	✔
Absolute Tolerance	0
Relative Tolerance	0.00%
Leading Tolerance	<u>0.04</u>
Lagging Tolerance	0
Block Path	sf_car/shift_logic

When you change this value, the status changes to show that the failed tests pass.

- 2 When you are satisfied with the tolerance value, enter it in the baseline criteria so you can rerun the test and save the new pass-fail result. In the **Test Browser** pane, select the test case in the Solver Compare test.
- 3 Under **Baseline Criteria**, change the **Leading Tol** value for the solver_baseline.mat file to .04.

By default, each signal inherits this value from the baseline file. You can override the value for each signal.

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
▼ <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

4 Run the test again. The test passes.

5 To store the tolerance value and the passed test with the test file, save the test file.

See Also

`sltest.testmanager.BaselineCriteria` | `sltest.testmanager.SignalCriteria`

Related Examples

- “Compare Model Output to Baseline Data” on page 6-7

Specify Test Properties in the Test Manager

In this section...
"Test Case, Test Suite, and Test File Sections Summary" on page 6-158
"Create Test Case from External File" on page 6-160
"Tags" on page 6-160
"Description" on page 6-160
"Requirements" on page 6-160
"System Under Test" on page 6-160
"Simulation 1 and Simulation 2" on page 6-162
"Parameter Overrides" on page 6-162
"Callbacks" on page 6-163
"Inputs" on page 6-164
"Simulation Outputs" on page 6-165
"Configuration Settings Overrides" on page 6-166
"Baseline Criteria" on page 6-167
"Equivalence Criteria" on page 6-168
"Iterations" on page 6-168
"Logical and Temporal Assessments" on page 6-169
"Custom Criteria" on page 6-170
"Coverage Settings" on page 6-170
"Test File Options" on page 6-172
"Test File Content" on page 6-172

The Test Manager has property settings that specify how test cases, test suites, and test files run. To open the Test Manager, use `sltest.testmanager.view`. For information about the Test Manager, see **Test Manager**

Test Case, Test Suite, and Test File Sections Summary

When you open a test case, test suite, or test file in the Test Manager, the test settings are grouped into sections. Test cases, test suites, and test files have different sections and settings. Click a test case, test suite, or test file in the **Test Browser** pane to see its settings.

Test Section	Test Case	Test Suite	Test File
"Tags" on page 6-160	✓	✓	✓
"Description" on page 6-160	✓	✓	✓
"Requirements" on page 6-160	✓	✓	✓
"System Under Test" on page 6-160	✓		

Test Section	Test Case	Test Suite	Test File
"Simulation 1 and Simulation 2" on page 6-162	✓		
"Parameter Overrides" on page 6-162	✓		
"Callbacks" on page 6-163	✓	✓	✓
"Inputs" on page 6-164	✓		
"Simulation Outputs" on page 6-165	✓		
"Configuration Settings Overrides" on page 6-166	✓		
"Baseline Criteria" on page 6-167	✓		
"Equivalence Criteria" on page 6-168	✓		
"Iterations" on page 6-168	✓		
"Logical and Temporal Assessments" on page 6-169	✓		
"Custom Criteria" on page 6-170	✓		
"Coverage Settings" on page 6-170	✓	✓	✓
"Test File Options" on page 6-172			✓
"Test File Content" on page 6-172			✓

If you do not want to see all of the available test sections, you can use the Test Manager preferences to hide sections:

- 1 In the Test Manager toolstrip, click **Preferences**.
- 2 Select the **Test File**, **Test Suite**, or **Test Case** tab.
- 3 Select the sections to show, or clear the sections to hide. To show only the sections in which you have already set or changed settings, clear all selections in the **Preferences** dialog box.
- 4 Click **OK**.

Sections that you already modified appear in the Test Manager, regardless of the preference setting.

To set these properties programmatically, see `sltest.testmanager.getpref` and `sltest.testmanager.setpref`.

Create Test Case from External File

To use an existing Excel file that is in the supported Simulink Test format to create a test case, select **Create test case from external file**. Then, enter the path to the file. The supported Excel format is described in “Microsoft Excel Import, Export, and Logging Format”.

To use an Excel or MAT file that is not in the supported format, write an adapter function so you can use that file in the Test Manager. Then, register the file using `sltest.testmanager.registerTestAdapter` function. If you have registered an adapter, when you select **Create test case from external file**, two fields appear, one for the path to the Excel or MAT file and one for the adapter function name. See `sltest.testmanager.registerTestAdapter` for information and an example.

Tags

Tag your test file, test suite, or test case with categorizations, such as `safety`, `logged-data`, or `burn-in`. Filter tests using these tags when executing tests or viewing results. See “Filter Test Execution, Results, and Coverage” on page 6-213.

For the corresponding API, see the `Tags` property of `sltest.testmanager.TestFile`, `sltest.testmanager.TestSuite`, or `sltest.testmanager.TestCase`, respectively.

Description

Add descriptive text to your test case, test suite, or test file.

For the corresponding API, see the `Description` property of `sltest.testmanager.TestFile`, `sltest.testmanager.TestSuite`, or `sltest.testmanager.TestCase`, respectively.

Requirements


If you have Requirements Toolbox installed, you can establish traceability by linking your test file, test suite, or test case to requirements. For more information, see “Link Test Cases to Requirements” (Requirements Toolbox).

To link a test case, test suite, or test file to a requirement:


- 1 Open the Requirements Editor. In the Simulink Toolstrip, on the **Apps** tab, under Model Verification, Validation, and Test, click **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.

For the corresponding API, see the `Requirements` property of `sltest.testmanager.TestFile`, `sltest.testmanager.TestSuite`, or `sltest.testmanager.TestCase`, respectively.

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 folder that contains the model to the path using the preload callback. See “Callbacks” on page 6-163.

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 Editor scenarios, and available configuration sets, click the **Refresh** button .

For the corresponding API, see the `Model` name-argument pair of `setProperty`.

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.

For the corresponding API, see the `HarnessName` name-argument pair of `setProperty`.

Simulation Settings and Release Overrides

To override the **Simulation Mode** of the model settings, select a new mode from the list. If the model contains SIL/PIL blocks and you need to run in Normal mode, enable **Override model blocks in SIL/PIL mode to normal mode**. For the corresponding API, see the `OverrideSILPILMode` name-argument pair of `setProperty`.

You can simulate the model and run tests in more than one MATLAB release that is installed on your system. Use **Select releases for simulation** to select available releases. You can use releases from R2011b forward.

To add one or more releases so they are available in the Test Manager, click **Add releases in Select releases for simulation** to open the **Release** pane in the Test Manager Preferences dialog box. Navigate to the location of the MATLAB installation you want to add, and click **OK**.

You can add releases to the list and delete them. You cannot delete the release in which you started the MATLAB session.

For more information, see “Run Tests in Multiple Releases of MATLAB” on page 6-94. For the corresponding API, see the `Release` name-argument pair of `setProperty`.

System Under Test Considerations

- The **System Under Test** cannot be in fast restart or external mode.
- To stop a test running in **Rapid Accelerator** mode, press **Ctrl+C** at the MATLAB command prompt.

- When running parallel execution in rapid accelerator mode, streamed signals do not show up in the Test Manager.
- The **System Under Test** cannot be a protected model.

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” on page 6-35 .

For the corresponding API, see the `SimulationIndex` name-argument pair of `setProperty`.

Parameter Overrides

Specify parameter values in the test case to override the parameter values in the model workspace, data dictionary, base workspace, or in a model reference hierarchy. 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 copy an individual parameter and paste it into another parameter set, highlight the parameter and right-click to use **Copy** and **Paste** from the context menu. You can also copy and paste parameter sets.

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, including MAT-files generated by Simulink Design Verifier. Click the **Add** arrow and select **Add File** to create a parameter set from a MAT-file.

For an example that uses parameter overrides, see “Override Model Parameters in a Test Case” on page 6-31.

For the corresponding APIs, see the `sltest.testmanager.ParameterOverride` class, and the `OverrideStartTime`, `OverrideStopTime`, `OverrideInitialState`,

OverrideModelOutputSettings, and ConfigSetOverrideSetting name-argument pairs of the setProperty method.

Parameter Overrides Considerations

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

Callbacks

Test-File Level Callbacks

Two callback scripts are available in each test file that execute at different times during a test:

- Setup runs before test file executes.
- Cleanup runs after test file executes.

For the corresponding test case APIs, see the PreloadCallback, PostloadCallback, CleanupCallback, and PreStartRealTimeApplicationCallback name-argument pairs of the TestCase setProperty method.

For the corresponding test file APIs, see the SetupCallback and CleanupCallback name-argument pairs of the test file TestFile setProperty method.

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.

If a test suite does not have any test cases, the test suite callbacks do not execute.

For the corresponding APIs, see the SetupCallback and CleanupCallback name-argument pairs of the TestSuite setProperty method.

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.

See “Test Execution Order” on page 6-209 for information about the order in which callbacks occur and models load and simulate.

To run a single callback script, click the **Run** button  above the corresponding script.

You can use predefined variables in the test case callbacks:

- sltest_bdroot available in **Post-Load**: The model simulated by the test case. The model can be a harness model.

- `sltest_sut` available in **Post-Load**: The system under test. For a harness, it is the component under test.
- `sltest_isharness` available in **Post-Load**: Returns true if `sltest_bdroot` is a harness model.
- `sltest_simout` available in **Cleanup**: Simulation output produced by simulation.
- `sltest_iterationName` available in **Pre-Load**, **Post-Load**, and **Cleanup**: Name of the currently executing test iteration.

`disp` and `fprintf` do not work in callbacks. To verify that the callbacks are executed, use a MATLAB script that includes breakpoints in the callbacks.

The test case callback scripts are not stored with the model and do not override Simulink model callbacks. Consider the following when using callbacks:

- To stop execution of an infinite loop from a callback script, press **Ctrl+C** at the MATLAB command prompt.
- `sltest.testmanager` functions are not supported.

For the corresponding APIs, see the `PreloadCallback`, `PostloadCallback`, `CleanupCallback`, and `PreStartRealTimeApplicationCallback` name-argument pairs of the `TestCase.setProperty` method.

Assessment Callback

You can enter a callback to define variables and conditions used only in logical and temporal assessments by using the **Assessment Callback** section. See “Assessment Callback” on page 6-169 in the Logical and Temporal Assessments section for more information.


For the corresponding API, see `setAssessmentsCallback`.

Inputs

A test case can use input data from:

- A Signal Editor block in the system under test. Select **Signal Editor scenario** and select the scenario. The system under test can have only one Signal Editor block at the top level.
- An external data file. In the **External Inputs** table, click **Add**. Select a MAT-file or Microsoft Excel file.

For more information on using external files as inputs, see “Use External Excel or MAT-File Data in Test Cases” on page 6-72. For information about the file format for Microsoft Excel files in Test Manager, see “Format Test Case Data in Excel” on page 6-83.

- Scenarios in a Test Sequence block. First, click the refresh arrow  next to the **Test Sequence Block** field, then select the Test Sequence block in the model that contains the scenarios. If you do not also select a scenario from **Override with Scenario** and do not use iterations, then the test runs the active scenario in the selected Test Sequence block. If you do not also select a scenario, but do use iterations, then the active scenario in the Test Sequence block is the default for all the iterations.

Use **Override with Scenario** to override the active scenario in the selected Test Sequence block. Click the refresh arrow next to the **Override with Scenario** field. Then, select the scenario to use

instead of the active scenario or as the default for the iterations. In the **Iterations** section, you can change the scenario assigned to each iteration. For more information, see “Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager” on page 3-59.

- An input file template that you create and populate with data. See “Create Data Files for Test Case Input” on page 6-80.

To include the input data in your test results set, select **Include input data in test result**.

If the time interval of your input data is shorter than the model simulation time, you can limit the simulation to the time specified by your input data by selecting **Stop simulation at last time point**.

For more information on test inputs, see the Test Authoring: Inputs page.

Edit Input Data Files in Test Manager

From the Test Manager, you can edit your input data files.

To edit a file, select the file and click **Edit**. You can then edit the data in the Signal Editor for MAT-files or Microsoft Excel for Excel files.

To learn about the syntax for Excel files, see “Format Test Case Data in Excel” on page 6-83.

For the corresponding API, see `sltest.testmanager.TestInput`.

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.

Logged Signals — In the **Logged Signals** subsection, click **Add**. Follow the user interface.

To copy a logged signal set to another test case in the same or a different test file, select the signal set in **Logged Signals**, right-click to display the context menu, and click **Copy**. Then, in the destination test case, select the signal set in **Logged Signals**, right-click the signal, and click **Paste**. You can copy and paste more than one logged signal set at a time.

For a test case, you can use the **SDI View File** setting to specify the path to a Simulation Data Inspector (SDI) view file. You can assign a different view file to each test case. The view file configures which signals to plot and their layout in the test case results. The Test Manager does not support some configurations in the SDI view file, such as axes plot layouts other than time plots and axes layouts other than *N*-by-*M* grids. However, the Test Manager applies a similar configuration, if possible. You cannot save an SDI view file from the Test Manager, although when you save the test and results in an MLDATX test file, the file saves the current layout for that test. Use `Simulink.sdi.saveView` to create and save an SDI view file. For more information, see “Save and Share Simulation Data Inspector Data and Views”.

Other Outputs — Use the options in the **Other Outputs** subsection to add states, final states, model output values, data store variables, and signal logging values to your test results. To enable selecting one or more of these options, click **Override model settings**.

- **States** — Include state values between blocks during simulation. You must have a Sequence Viewer block in your model to include state values.

- **Final states** — Include final state values. You must have a Sequence Viewer block in your model to include final state values.
- **Output** — Include model output values.
- **Data stores** — Include logged data store variables in Data Store Memory blocks in the model. This option is selected by default.
- **Signal logging** — Include logged signals specified in the model. This option is selected by default. If you selected **Log Signal Outputs** when you created the harness, all of the output signals for the component under test are logged and returned in test results, even though they are not listed in the **Simulation Outputs** section. To turn off logging for one of the signals, in the test harness, right-click a signal and select **Stop Logging Selected Signals**.

For more information, see “Capture Simulation Data in a Test Case” on page 6-85. For the corresponding API, see the `OverrideModelOutputSettings` name-argument pair of `setProperty`.

Output Triggers —Use the **Output Triggers** subsection to specify when to start and stop signal logging based on a condition or duration. **** test passes if pass while triggered even if test fails outside of triggered time/condition true****.

The **Start Logging** options are:

- **On simulation start** — Start logging data when simulation starts.
- **When condition is true** — Start logging when the specified condition expression is true. Click the edit symbol next to **Condition** to display an edit box, where you enter the condition.
- **After duration** — Start logging after the specified number of seconds have passed since the start of simulation. Click on the value next to **Duration(sec)** to display an edit box where you enter the duration in seconds.

The **Stop Logging** options are:

- **When simulation stops** — Stop logging data when simulation ends.
- **When condition is true** — Stop logging when the specified condition expression is true. Click the edit symbol next to **Condition** to display an edit box, where you enter the condition. Variables in the condition appear in the **Symbols** editor, where you can map them to a model element or expression, or rename them.
- **After duration** — Stop logging after the specified number of seconds have passed since logging started. Click on the value next to **Duration(sec)** to display an edit box where you enter the duration in seconds.

Shift time to zero — Shifts the logging start time to zero. For example, if logging starts at time 2, then selecting this option shifts all times back by 2 seconds.

Symbols — Click **Add** to map a signal from the model to a symbol name. You can use that symbol in a trigger condition. For information on using and mapping symbols, see “Assess Temporal Logic by Using Temporal Assessments” on page 3-93

Configuration Settings Overrides

For 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 for a test case without modifying the model. The configuration settings overrides options are:

- **Do not override model settings** — Use the current model configuration settings
- **Name** — Name of active configuration set. A model can have only one active configuration set. Refresh the list to see all available configuration sets and select the desired one to be active. If you leave the default [Model Settings] as the name, the simulation uses the default, active configuration set of the model.
- **Attach configuration set in a file** — Path to the external file (**File Location**) that contains a configuration set variable. The variable you specify in **Variable Name** references the name of a configuration set in the file. For information on creating a configuration set, see `Simulink.ConfigSet` and “Save a Configuration Set”. For information on configuration set references, see “Share a Configuration with Multiple Models”.

For the corresponding API, see the `ConfigSetOverrideSetting`, `ConfigSetName`, `ConfigSetVarName`, `ConfigSetFileLocation`, and `ConfigSetOverrideSetting` name-argument pairs of `setProperty`.

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.

Include Baseline Data in Results and Reports

Click **Include baseline data in test result** to include baseline data in test result plots and test reports.

Capture Baseline Criteria

To capture logged signal data from the system under test to use as the baseline criteria, click **Capture**. Then follow the prompts in the Capture Baseline dialog box. Capturing the data compiles and simulates the system under test and stores the output from the logged signals to the baseline. For a baseline test example, see “Compare Model Output to Baseline Data” on page 6-7.

For the corresponding API, see the `captureBaselineCriteria` method.

You can save the signal data to a MAT-file or a Microsoft Excel file. To understand the format of the Excel file, see “Format Test Case Data in Excel” on page 6-83.

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 of MATLAB” on page 6-94.

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 Baselines for Iterations**. This check box appears only if your test case already contains iterations. For more information on iterations, see “Test Iterations” on page 6-125.

Specify Tolerances

You can specify tolerances to determine the pass-fail criteria of the test case. You can specify absolute, relative, leading, and lagging tolerances for individual signals or the entire baseline criteria set.

After you capture the baseline, the baseline file and its signals appear in the table. In the table, you can set the tolerances for the signals. To see tolerances used in an example for baseline testing, see “Compare Model Output to Baseline Data” on page 6-7.

For the corresponding API, see the `Abstol`, `RelTol`, `LeadingTol`, and `LaggingTol` properties of `sltest.testmanager.BaselineCriteria`.

Add File as Baseline

By clicking **Add**, you can select an existing file as a baseline. You can add MAT-files and Microsoft Excel files as the baseline. Format Microsoft Excel files as described in “Format Test Case Data in Excel” on page 6-83.

For the corresponding API, see the `addInput` method.

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 6-104.

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 6-102.

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” on page 6-35.

For the corresponding API, see the `captureEquivalenceCriteria` method.

Iterations

Use iterations to repeat a test with different parameter values, configuration sets, or input data.

- You can run multiple simulations with the same inputs, outputs, and criteria by sweeping through different parameter values in a test case.
- Models, external data files, and Test Sequence blocks can contain multiple test input scenarios. To simplify your test file architecture, you can run different input scenarios as iterations rather than as different test cases. You can apply different baseline data to each iteration, or capture new baseline data from an iteration set.
- You can iterate over different configuration sets, for example to compare results between solvers or data types. You can also iterate over different scenarios in a Test Sequence block.

To create iterations from defined parameter sets, Signal Editor scenarios, Test Sequence scenarios, external data files, or configuration sets, use table iterations. To create a custom set of iterations from the available test case elements, write a MATLAB iteration script in the test case.

To run the iterations without recompiling the model for each iteration, enable **Run test iterations in fast restart**. When selected, this option reduces simulation time.

For more information about test iterations, see “Test Iterations” on page 6-125. For more information about fast restart, see “How Fast Restart Improves Iterative Simulations”.

For the corresponding API, see `sltest.testmanager.TestIteration`.

Logical and Temporal Assessments

Create temporal assessments using the form-based editor that prompts you for conditions, events, signal values, delays, and responses. When you collapse the individual elements, the editor displays a readable statement summarizing the assessment. See “Assess Temporal Logic by Using Temporal Assessments” on page 3-93 and “Logical and Temporal Assessment Syntax” on page 3-107 for more information.

To copy and paste an assessment or symbol, select the assessment or symbol and right-click to display the context menu. You can select a single assessment or symbol or select multiple assessments or symbols. Alternatively, to copy or paste selected assessments or symbols, use **Ctrl+C** or **Ctrl+V**. Pasting the assessment adds it to the end of the assessment list in the current test case. You can also paste to a different test case. The assessments and their symbol names change to the default names in the pasted assessment. You can also use the context menu to delete assessments. To delete symbols, use the **Delete** button. If you delete an assessment or symbol, you cannot paste it even if you copied it before deleting it.

Assessment Callback

You can define variables and use them in logical and temporal assessment conditions and expressions in the **Assessment Callback** section.

Define variables by writing a script in the **Assessment Callback** section. You can map these variables to symbols in the **Symbols** pane by right-clicking the symbol, selecting **Map to expression**, and entering the variable name in the **Expression** field. For information on how to map variables to symbols, see **Map to expression** under “Resolve Assessment Parameter Symbols” on page 3-95.

The **Assessment Callback** section has access to the predefined variables that contain test, simulation, and model data. You can define a variable as a function of this data. For more information,

see “Define Variables in the Assessment Callback Section” on page 3-110. For the corresponding API methods, see `setAssessmentsCallback` and `getAssessmentsCallback`.

If your assessments use `at least`, `at most`, `between`, or `until` syntax, select **Extend Results** to produce the minimum possible untested results. In some cases, none or not all untested results can be tested, so the results will still show some untested results. When you extend the test results, previously passing tests might fail. Leave **Extend Results** checked unless you need to avoid an incompatibility with earlier test results.

Symbol `t` (time)

The symbol `t` is automatically bound to simulation time and can be used in logical and temporal assessment conditions. This symbol does not need to be mapped to a variable and is not visible in the **Symbols** pane. For example, to limit an assessment to a time between 5 and 7 seconds, create a **Trigger-response** assessment and, in the trigger condition, enter `t < 5 & t > 7`. To avoid unexpected behavior, do not define a new symbol `t` in the **Symbols** pane.

Symbol Data Type

If you map a symbol to a discrete data signal that is linearly interpolated, the interpolation is automatically changed to zero-order hold during the assessment evaluation.

Custom Criteria

This section includes an embedded MATLAB editor to define custom pass/fail criteria for your test. Select **function customCriteria(test)** to enable the criteria script in the editor. Custom criteria operate outside of model run time; the script evaluates after model simulation.

Common uses of custom criteria include verifying signal characteristics or verifying test conditions. MATLAB Unit Test qualifications provide a framework for verification criteria. For example, this custom criteria script gets the last value of the signal `PhiRef` and verifies that it equals 0:

```
% Get the last value of PhiRef from the dataset Signals_Req1_3
lastValue = test.sltest_simout.get('Signals_Req1_3').get('PhiRef').Values.Data(end);

% Verify that the last value equals 0
test.verifyEqual(lastValue,0);
```

See “Process Test Results with Custom Scripts” on page 6-179. For a list of MATLAB Unit Test qualifications, see “Table of Verifications, Assertions, and Other Qualifications”.

You can also define plots in the **Custom Criteria** section. See “Create, Store, and Open MATLAB Figures” on page 6-188.

For the corresponding API, see `sltest.testmanager.CustomCriteria`.

Coverage Settings

Use this section to configure coverage collection for a test file. The settings propagate from the test file to the test suites and test cases in the test file. You can turn off coverage collection or one or more coverage metrics for a test suite or test case, unless your test is a MATLAB-based Simulink test.

For MATLAB-based Simulink tests, you can change the coverage settings only at the test file level. If you change the coverage settings in the Test Manager, the changes are not saved to the MATLAB-

based Simulink test script file. If you also set the coverage using the `sltest.plugins.ModelCoveragePlugin` in a MATLAB-based Simulink test script (.m) file or at the command line, the Test Manager uses the coverage settings from the test script instead of the Test Manager coverage settings.

Coverage is not supported for SIL or PIL blocks.

The coverage collection options are:

- **Record coverage for system under test** — Collects coverage for the model or, when included, the component specified in the “System Under Test” on page 6-160 section for each test case. If you are using a test harness, the system under test is the component for which the harness is created. The test harness is not the system under test.
 - For a block diagram, the system under test is the whole block diagram.
 - For a Model block, the system under test is the referenced model.
 - For a subsystem, the system under test is the subsystem.
- **Record coverage for referenced models** — Collects coverage for models that are referenced from within the specified system under test. If the test harness references another model, the coverage results are included for that model, too.
- **Exclude inactive variants** — Excludes from coverage results these variant blocks that are not active at any time while the test runs:
 - Variant blocks in Simulink with **Variant activation time** set to startup
 - Variant configurations in Stateflow charts

When displaying the test results, if you select or clear this option in the **Aggregated Coverage Results** section, the coverage results update automatically. For information, see “Model Coverage for Variant Blocks” (Simulink Coverage).

Note Coverage settings, including coverage filter files, in the Test Manager override all coverage settings in the model configuration. In the Test Manager, **Do not override model settings** in the Configuration Settings section and **Override model settings** in the Simulation Outputs section do not apply to coverage.

By default the Test Manager includes external MATLAB functions and files in the coverage results. You can exclude external MATLAB functions and files by using `set_param(model, 'CovExternalEMLEnable', 'off', 'CovSFcnEnable', 'off');` at the command line. Alternatively, you can exclude MATLAB functions and files by using the **Include in analysis** setting in the Coverage Analyzer app from within the Simulink model.

For more information about collecting coverage, see “Collect Coverage in Tests” on page 6-135. For the corresponding API, see `sltest.testmanager.CoverageSettings`.

For information on the **Coverage Metrics** options, see “Types of Model Coverage” (Simulink Coverage).

For information about MATLAB-based Simulink tests, see “Using MATLAB-Based Simulink Tests in the Test Manager” on page 6-116.

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.

For the corresponding API, see the `CloseFigures` property of `sltest.testmanager.Options`.

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 6-188.

For the corresponding API, see the `SaveFigures` property of `sltest.testmanager.Options`.

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.

Note To enable the options to specify the number of plots per page, select **Plots for simulation output and baseline**.

By default, the model name, simulation start and stop times, and trigger information are included in the report.

For the corresponding API, see the `GenerateReport` property of `sltest.testmanager.Options`.

For detailed reporting information, see “Export Test Results” on page 7-16 and “Customize Test Results Reports” on page 7-21.

Test File Content

For a MATLAB-based Simulink test, displays the contents of the M file that defines the test. This section appears only if you opened or created a new MATLAB-based Simulink test. See “Using MATLAB-Based Simulink Tests in the Test Manager” on page 6-116.

See Also

Test Manager | `sltest.testmanager.getpref` | `sltest.testmanager.setpref`

Preferences

Set test sections to display and releases to use for testing.

Display Tab - Select test sections to show when you select a test file, test suite, or test case in the Test Manager. Sections that you have changed from their default settings are always shown, even if you deselect them in this tab. To show only the sections in which you have already set or changed settings, clear all selections.

Release Tab - Select releases to use with tests. Only installed releases are available to add. Releases you select are available in **Selected releases for simulation list** in the System Under Test section.

Increase Coverage by Generating Test Inputs

In this section...

“Overall Workflow” on page 6-174

“Generate Test Cases Using Simulink Design Verifier” on page 6-175

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, on the **Apps** tab, under Model Verification, Validation, and Test, click **Design Verifier**. On the **Tests** tab, click **Simulink Test Manager**. In the **Review Results** section, click **Load Earlier Results**. 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**. Test cases and iterations for Requirements Table blocks are linked automatically to the corresponding requirements.
- 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. Both MAT and Excel files are supported when the source is Inport.
 - **Signal Editor**: The inputs are in scenarios in a Signal Editor block inside the test harness. The Signal Editor block supports MAT files that contain these inputs. You can edit these scenarios in the Signal 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. Simulink Design Verifier saves a MAT or Excel data file that also includes parameter settings. You can view or override these settings in the Parameter Overrides section of the Test Manager.

Note Another way to import test cases from Simulink Design Verifier is with the Create Test for Component wizard. For information, see “Generate Tests and Test Harnesses for a Model or Components” on page 6-24.

Generate Test Cases Using Simulink Design Verifier

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 used in this example is a closed-loop heat pump system. The controller inputs are the measured room temperature and the set temperature. The controller outputs a bus of three signals that control the fan, heat pump, and direction of the heat pump (heat or cool). The model contains a harness that tests the heating and cooling scenarios.

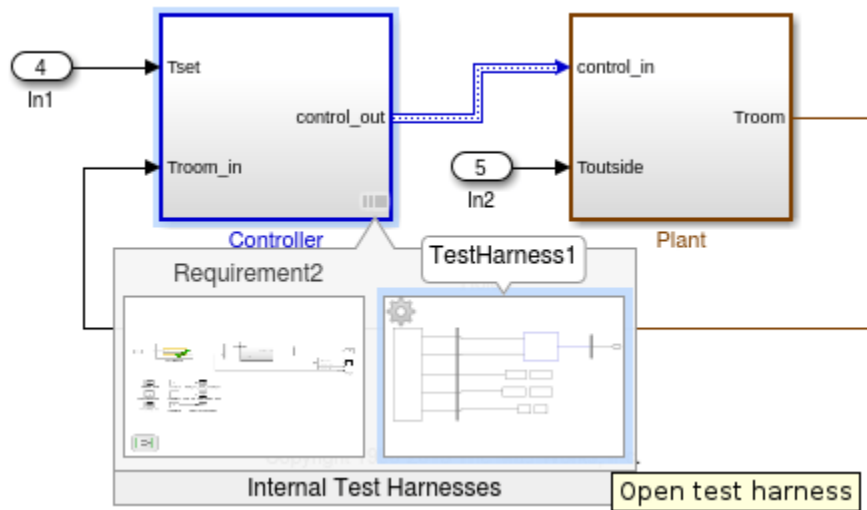
Open the Model

```
sltestTestCaseFromDVExample
```

Generate Tests and Export to Simulink Test

1. Right-click the Controller block and select **Design Verifier > Generate Tests for Subsystem**. Simulink Design Verifier generates the tests for the component.
2. In the results summary window, click **Export test cases to Simulink Test**.
3. In the Export Design Verifier Test Cases dialog box, enter:
 - **Test Harness:** TestHarness1
 - **Harness Source:** Signal Editor
 - Select **Use a new test file**
 - **Test File:** TestFile_GeneratedTests.mldatx
 - **Test Case:** <Create a new test case>
4. Click **OK**.

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

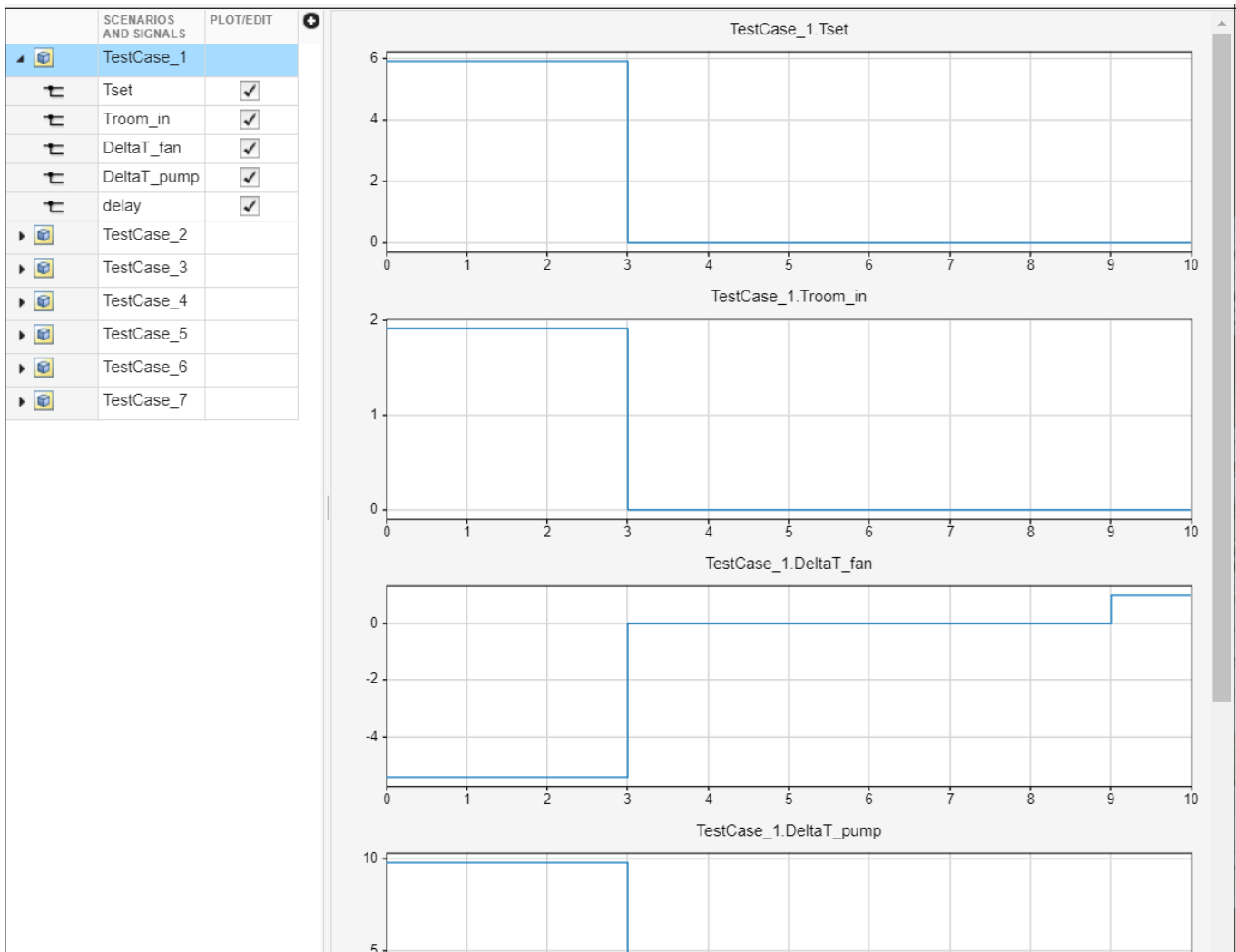


5. Click the TestHarness1 thumbnail to open the harness. Then double-click the Harness Inputs Signal Editor block source.

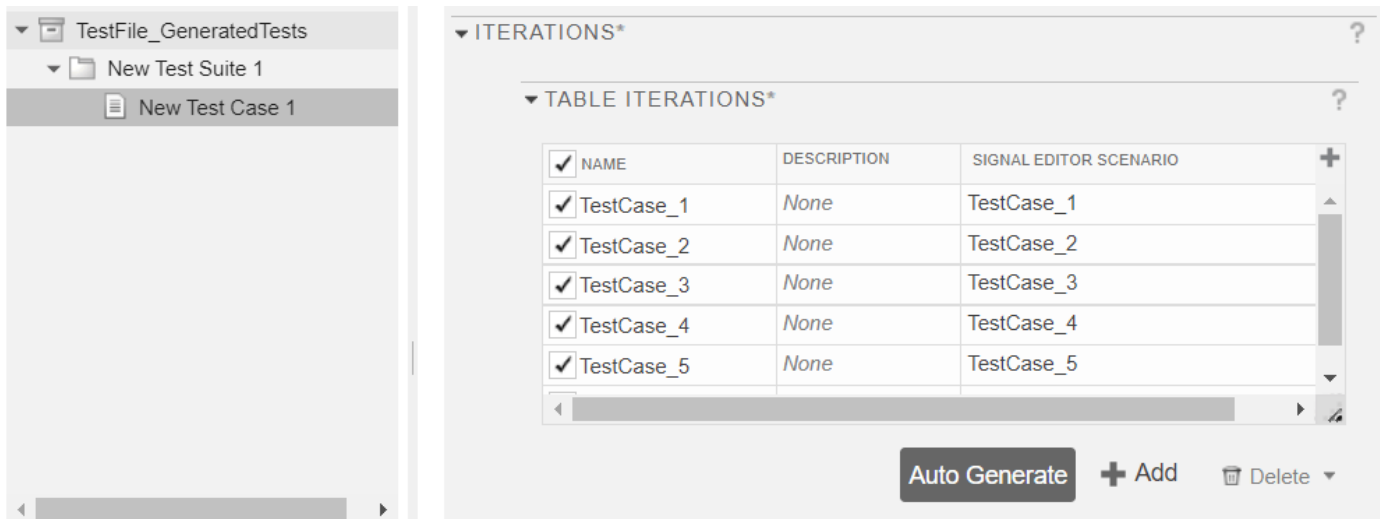
6. In the Block Parameters: Harness Inputs dialog box, click **Launch Signal Editor**



7. To see the test inputs in the Signal Editor, expand a test case and select the 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 Editor source. Expand the Iterations section to see the iterations corresponding to the signal scenarios.



See Also

sltest.import.sldvData

More About

- “Generate Tests and Test Harnesses for a Model or Components” on page 6-24

Process Test Results with Custom Scripts

In this section...

“MATLAB Testing Framework” on page 6-179

“Define a Custom Criteria Script” on page 6-179

“Reuse Custom Criteria and Debug Using Breakpoints” on page 6-180

“Custom Criteria Programmatic Interface Example” on page 6-182

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. See `Simulink.SimulationOutput`. If you require run-time verifications, use a `verify()` statement in a Test Assessment or Test Sequence block. See “Assess Model Simulation Using verify Statements” on page 3-18.

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 “Table of Verifications, Assertions, and Other Qualifications”. 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**.

3. In the test case, expand the **Custom Criteria** section.

4. Enable the custom criteria script by selecting function `customCriteria(test)`.

5. In the embedded MATLAB editor, add the following script below the `customCriteria.out = true` line. This script gets and verifies the final values of the signals `Phi` and `APEng`. `Signals_Req1_3` is one of the test requirements and `sltest_simout` is the simulation output.

```
% 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),'.']);
```

6. Run the test case.

7. 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	✔
☑ Final Phi value: 0. { verifyEqual	✔
☑ Final APEng value: 0. { verifyEq	✔

Reuse Custom Criteria and Debug Using Breakpoints

This example shows how to author custom criteria in a standalone function and add a breakpoint to that function. The custom criteria function `sltestCheckFinalRollRefValues.m` is provided as the starting point for this example.

Using a standalone function lets you:

- Reuse the custom criteria in multiple test cases.
- Set breakpoints in the criteria script for debugging.
- Investigate the simulation output using the command line.

1. Open the function file in the MATLAB Live Editor, and open the test file in the Test Manager

```
open('sltestCheckFinalRollRefValues.m')
```

```
sltest.testmanager.load('AutopilotTestFile.mldatx');
sltest.testmanager.view
```

2. In the **Test Browser** pane of the Test Manager, select **AutopilotTestFile > Basic Design Test Cases > Requirement 1.3 Test**.

3. Expand the **Custom Criteria** section, and enable the custom criteria script by selecting function `customCriteria(test)`. Then, in the embedded MATLAB editor, replace the content with the function call to the custom criteria:

```
sltestCheckFinalRollRefValues(test)
```

4. In `sltestCheckFinalRollRefValues.m` file in the MATLAB Live Editor, set a breakpoint by clicking on the 8 of line number 8.

5. In the Test Manager, run the test case. When the breakpoint is encountered, the test case pauses running, the command window displays the line where the breakpoint occurred, and the command line prompt changes to `K>>`, which indicates debugging mode.

6. 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: ''
```

7. The property `sltest_simout` contains the simulation data. To view the data `PhiRef`, at the command line enter

```
test.sltest_simout.get('Signals_Req1_3').get('PhiRef')
```

```
ans =

Simulink.SimulationData.Signal
Package: Simulink.SimulationData

Properties:
    Name: 'PhiRef'
    PropagatedName: ''
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
    PortType: 'outport'
    PortIndex: 1
    Values: [1x1 timeseries]
```

8. In the `slttestCheckFinalRollRefValues.m` file in the MATLAB Live Editor, click **Continue** to continue running the custom criteria script.

9. In the **Results and Artifacts** pane, expand the **Custom Criteria Result**. Both criteria pass.

▼ Requirement 1.3 Test	✖
▶ Verify Statements	✖
▶ Sim Output (RollAutopilotMdlRef :	
▼ Custom Criteria Result	✔
☑ Final Phi value: 0. { verifyEqual	✔
☑ Final APEng value: 0. { verifyEq	✔

To reuse the script in another test case, call the same function from the **Custom Criteria** of that test case.

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 = slttest.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 6-191
- “Create, Store, and Open MATLAB Figures” on page 6-188

Assess the Damping Ratio of a Flutter Suppression System

This example shows how to use a custom criteria script to verify that wing oscillations are damped in multiple altitude and airspeed conditions.

The Test 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.

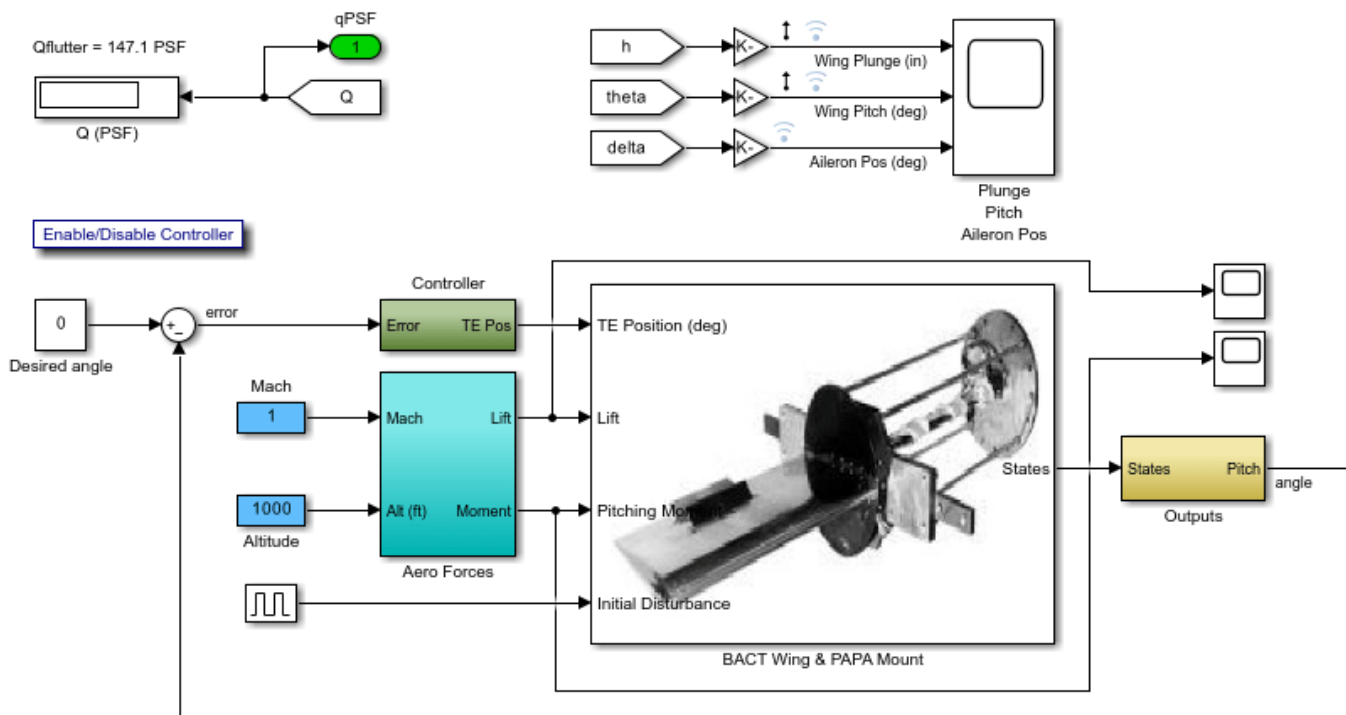
The test iterates over 16 combinations of Mach and Altitude. The test case uses custom criteria with Curve Fitting Toolbox™ to find the peaks of the wing pitch, and determine the damping ratio. If the damping ratio is not greater than zero, the assessment fails.

Click **Open Script** to open the test file.

```
open('sltestFlutterCriteriaTest.mldatx')
```

In the **Test Browser**, select **Altitude and mach iterations**. Open the model by clicking the arrow next to **Model** in the **System Under Test** section.

```
open_system('sltestFlutterSuppressionSystemExample.slx')
```



Copyright 2003-2021 The MathWorks, Inc.

Custom Criteria Script

The test case custom criteria uses this script to verify that the damping ratio is greater than zero.

```
% Get time and data for pitch
Time = test.sltest_simout.get('sigsOut').get('pitch').Values.Time(1:15000);
Data = test.sltest_simout.get('sigsOut').get('pitch').Values.Data(1:15000);

% Find peaks
[~, peakIds] = findpeaks(Data, 'minpeakheight', 0.002, 'minpeakdistance', 50);
peakTime= Time(peakIds);
peakPos = Data(peakIds);
rn = peakPos(1)./peakPos(2:end);
L = 1:length(rn);

% Do curve fitting
fittedModel = exponentialFitAndPlot(L, rn);
delta = fittedModel.d;

% Find damping ratio
dRatio = delta/sqrt((2*pi)^2+delta^2);

% Make sure damping ratio is greater than 0
test.verifyGreaterThan(dRatio,0,'Damping ratio must be greater than 0');
```

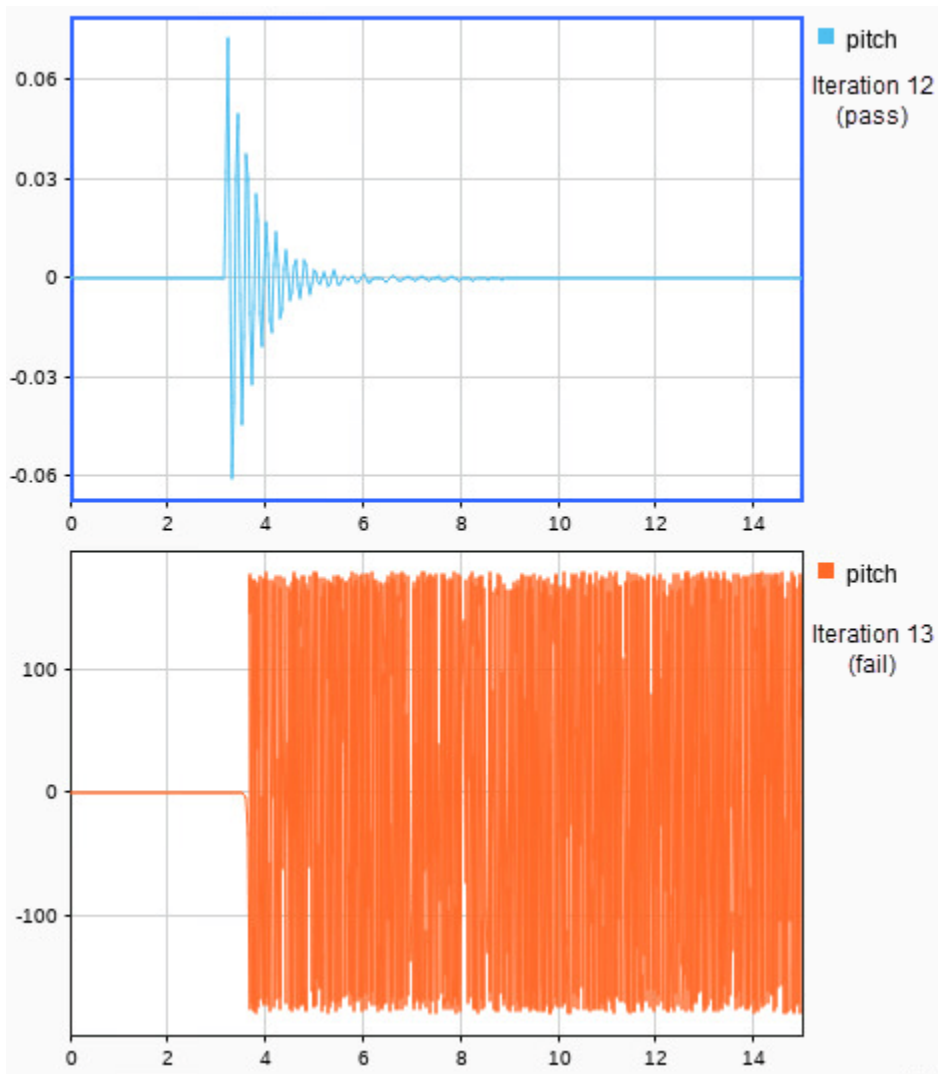
Test Results

Running the test case returns two conditions in which the damping ratio is greater than zero.

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

▼ I Scripted_Iteration13	✖
▶ Sim Output (sltestFlutterSuppressionSystemExa	
▶ Custom Criteria Result	✖
▼ I Scripted_Iteration14	✖
▶ Sim Output (sltestFlutterSuppressionSystemExa	
▶ Custom Criteria Result	✖
▶ I Scripted_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)
```

Create, Store, and Open MATLAB Figures

In this section...

“Create a Custom Figure for a Test Case” on page 6-188

“Include Figures in a Report” on page 6-189

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 to test using `openExample('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 object and store in sldemo_absbrake_output variable
out = test.sltest_simout.get('sldemo_absbrake_output');

% Plot wheel speed and car speed
```

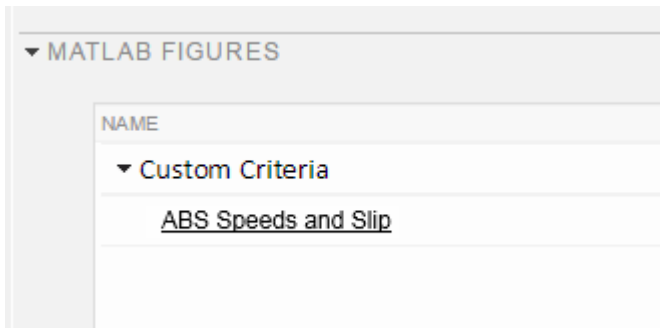


```

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 | getOptions (TestSuite) | getOptions (TestCase) |
getOptions (TestFile)`

Related Examples

- “Export Test Results” on page 7-16

Test Models Using MATLAB Unit Test

In this section...

“Overall Workflow” on page 6-191

“Considerations” on page 6-191

“Comparison of Test Nomenclature” on page 6-191

“Basic Workflow Using MATLAB® Unit Test” on page 6-192

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 lets you return coverage results at to the command line. If you set up coverage in the Test Manager, you do not need to use this plugin.
- 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. Create the test runner.

```
import matlab.unittest.TestRunner
apranner = TestRunner.withNoPlugins;
```

4. Add the plugin to produce Test Manager results.

```
import sltest.plugins.TestManagerResultsPlugin
tmr = TestManagerResultsPlugin;
addPlugin(aprunner, tmr)
```

5. Run the test.

```
apresults = run(aprunner,apsuite);
```

6. View the summary of the test, which shows that the test failed because of a verification failure.

```
apresults.Details.SimulinkTestManagerResults
```

```
ans =
```

```
  TestResultContainer with properties:
```

```
    TestResult: [1x1 sltest.testmanager.TestCaseResult]
```

Enable warnings.

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

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

See Also

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

Related Examples

- “Output Results for Continuous Integration Systems” on page 6-194
- “Run Tests for Various Workflows”

Output Results for Continuous Integration Systems

In this section...

“Test a Model for Continuous Integration Systems ” on page 6-194

“Model Coverage Results for Continuous Integration” on page 6-196

You can create model tests that are compatible with continuous integration (CI) systems such as Jenkins. To create CI-compatible results, run your Simulink Test files using MATLAB Unit Test.

To run CI-compatible tests, follow this general procedure:

- 1 Create a test suite from the MLDATX test file.
- 2 Create a test runner.
- 3 Create plugins for the test output or coverage results.
 - For test outputs, use the TAPPlugin or XMLPlugin.
 - For model coverage, use the ModelCoveragePlugin and CoberturaFormat. When collecting model coverage in Cobertura format:
 - Only top model coverage is reflected in the Cobertura XML.
 - Only model Decision coverage is reflected, and it is mapped to Condition elements in Cobertura XML.
- 4 Create plugins for CI-compatible output.
- 5 Add the plugins to the test output or coverage results.
- 6 Add the test output plugins or coverage result plugins to the test runner.
- 7 Run the test.

Test a Model for Continuous Integration Systems

This example shows how to test a model, publish Test Manager results, and output results in TAP format with a single execution.

You use MATLAB® Unit Test to create a test suite and a test runner, and customize the runner with these plugins:

- `matlab.unittest.plugins.TestReportPlugin` produces a MATLAB Test Report.
- `sltest.plugins.TestManagerResultsPlugin` adds Test Manager results to the MATLAB Test Report.
- `matlab.unittest.plugins.TAPPlugin` outputs results to a TAP file.

The test case creates a square wave input to a controller subsystem and sweeps through 25 iterations of parameters *a* and *b*. The test compares the *alpha* output to a baseline with a tolerance of 0.0046. The test fails on those iterations in which the output exceeds this tolerance.

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 7-17.

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.automation.streams.ToFile
tapFile = 'F14Output.tap';
tap = TAPPlugin.producingVersion13(ToFile(tapFile));
addPlugin(f14runner,tap)
```

8. Run the test.

Several iterations fail, in which the signal-baseline difference exceeds the tolerance criteria.

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

```
Generating test report. Please wait.
```

```
    Preparing content for the test report.
```

```
        Adding content to the test report.
```

```
        Writing test report to file.
```

```
Test report has been saved to:
```

```
    C:\TEMP\Bdoc23a_2213998_3568\ib570499\16\tp8135f7b4\simulinktest-ex40056435\F14Report.pdf
```

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('AutopilotTestFile.mldatx');
apsuite = testsuite(aptest.FilePath);
```

2. Create a test runner.

```
trun = TestRunner.withNoPlugins;
```

3. Set the coverage metrics to collect. This example uses decision coverage. In the Cobertura output, decision coverage is listed as condition elements.

```
import sltest.plugins.coverage.CoverageMetrics

cmet = CoverageMetrics('Decision',true);
```

4. Set the coverage report properties. This example produces a file `R13Coverage.xml` in the current working folder. Ensure your working folder has write permissions.

```
import sltest.plugins.coverage.ModelCoverageReport
import matlab.unittest.plugins.codecoverage.CoberturaFormat

rptfile = 'R13Coverage.xml';
rpt = CoberturaFormat(rptfile)

rpt =
    CoberturaFormat with no properties.
```

5. Create a model coverage plugin. The plugin collects the coverage metrics and produces the Cobertura format report.

```
import sltest.plugins.ModelCoveragePlugin

mcp = ModelCoveragePlugin('Collecting',cmet,'Producing',rpt)

mcp =
    ModelCoveragePlugin with properties:
```



```
RecordModelReferenceCoverage: '<default>'
      MetricsSettings: [1x1 sltest.plugins.coverage.CoverageMetrics]
```

6. Add the coverage plugin to the test runner.

```
addPlugin(trun,mcp)
```

```
% Turn off command line warnings:
warning off Stateflow:cdr:VerifyDangerousComparison
warning off Stateflow:Runtime:TestVerificationFailed
```

7. Run the test.

```
APResult = run(trun,apsuite)
```

```
APResult =
  TestResult with properties:
      Name: 'AutopilotTestFile > Basic Design Test Cases/Requirement 1.3 Test'
      Passed: 0
      Failed: 1
      Incomplete: 0
      Duration: 0.4877
      Details: [1x1 struct]
```

```
Totals:
  0 Passed, 1 Failed, 0 Incomplete.
  0.48767 seconds testing time.
```

8. Reenable warnings.

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

See Also

TestRunner | TestSuite | sltest.plugins.ModelCoveragePlugin |
 sltest.plugins.TestManagerResultsPlugin |
 matlab.unittest.plugins.TestReportPlugin | matlab.unittest.plugins.TAPPlugin

More About

- “Test Models Using MATLAB Unit Test” on page 6-191

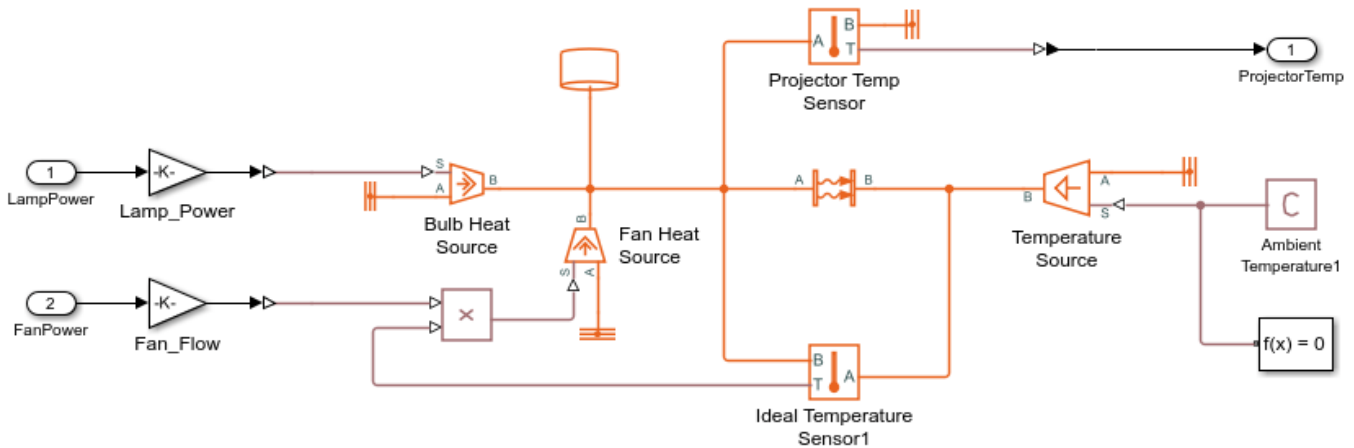
Parametric Sweep for a Simscape Thermal Model

This example shows how to test a physical system, and how to optimize a parameter using a test harness, test sequence, and the test manager. The example uses a system-level thermal model of a projector which includes Simscape® thermal blocks.

Set Up Variables

Set the required variables for the example.

```
Model = 'sltestProjectorFanSpeedExample';
Harness = 'FanSpeedTestHarness';
TestSuite = 'sltestProjectorFanSpeedTestSuite.mldatx';
open_system(Model);
```



Copyright 2015-2020 The MathWorks, Inc.

Test Plan and System Requirements

This test demonstrates sweeping through several fan speeds to determine the optimal value. In short, the optimal fan speed results in the fastest response without damaging the system. In detail, the optimal fan speed:

- Prevents the system from exceeding the specified maximum temperature.
- Minimizes the time for the system to reach the temperature at which the lamp emits visible light.

The document `sltestProjectorFanSpeedExampleRequirements.slreqx` captures these detailed requirements and the test procedure.

Test-specific model items reside in the test harness, keeping main models free of unnecessary blocks, suitable for code generation, and suitable for integration with other models.

Open the Test File

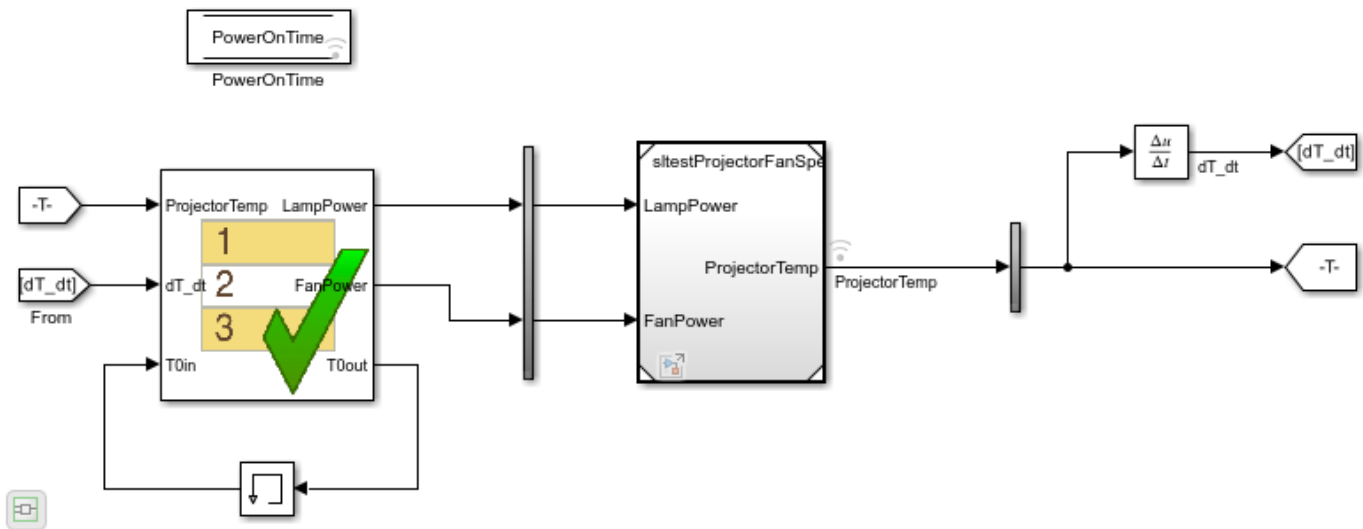
Open the Test Manager to view the test suite controlling the parameter sweep. From the model, open the Simulink Test app and click on Simulink Test Manager. Open the file referenced by TestSuite. You can also enter

```
open(TestSuite)
```

Description of the Test

The test investigates the transient and steady-state thermal characteristics of the system. The test sequence initializes the system to ambient temperature, then powers the projector lamp. When the system reaches a steady-state condition, the lamp switches off. This test is modeled in a test harness using a Test Sequence block. Run the following to open the test harness:

```
sltest.harness.open(Model,Harness);
```



Requirements Linking

The test suite contains links to the requirements document. You can view the requirements link by opening the test suite in the Test Browser, and clicking the links in the **Requirements** section.

The Test Sequence

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

Step	Transition	Next Step
InitializeSysToAmbient LampPower = 0; FanPower = 1; T0out = latch(ProjectorTemp); % latch captures the temp at the % start of the step	1. duration(abs(dT_dt/(ProjectorTemp-T0in)) < Threshold) >= DurationLimit % Transitions when the ratio of the temperature change to the initial temp % difference is less than a threshold (set to 0.1%) for a certain time. % Essentially seeks to transition when system reaches steady-state.	SystemHeatingTest ▼
SystemHeatingTest FanPower = latch(FanPower); LampPower = 1; T0out = latch(ProjectorTemp); PowerOnTime = latch(t);	1. duration(abs(dT_dt/(ProjectorTemp-T0in)) < Threshold) >= DurationLimit	SystemCoolingTest ▼
SystemCoolingTest FanPower = latch(FanPower); LampPower = 0; T0out = latch(ProjectorTemp);	1. duration(abs(dT_dt/(ProjectorTemp-T0in)) < Threshold) >= DurationLimit	Stop ▼
Stop LampPower = 0; FanPower = 0;		

The T0out and T0in signals store the initial projector temperature at each test step.

PowerOnTime stores simulation time when the lamp signal activates. This facilitates subsequent data analysis.

The transition condition detects the steady-state condition. At steady-state, the system temperature change is a small fraction (Threshold) of the difference between the current projector temperature and the initial projector temperature at each step. This condition must hold for a minimum time DurationLimit, in this case 10 seconds.

You may link the steps in the test sequence blocks against prepopulated requirements in the requirements document sltestProjectorFanSpeedExampleRequirements.slreqx.

Description of the Parameter Sweep

The pre-load callbacks contain the command to set the fan speed for each test case under the Fan Speed Parametric Study test suite. The parameter overrides contain the command to recalculate fan airflow from fan speed, and then override the test harness parameter. You can view these commands in the **Callbacks** and **Parameter Overrides** section of each test case.

Fan Speed = 1300 x

▶ SIMULATION SETTINGS OVERRIDES

▼ PARAMETER OVERRIDES ?

PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
<input checked="" type="checkbox"/> Parameter Set 1 <input checked="" type="checkbox"/> Fan_Flow	Fan_Speed*(Fan_Air_Displa...	base workspace	FanSpeedTestHarness/sitestProj...

+ Add Refresh Export Delete

▼ CALLBACKS ?

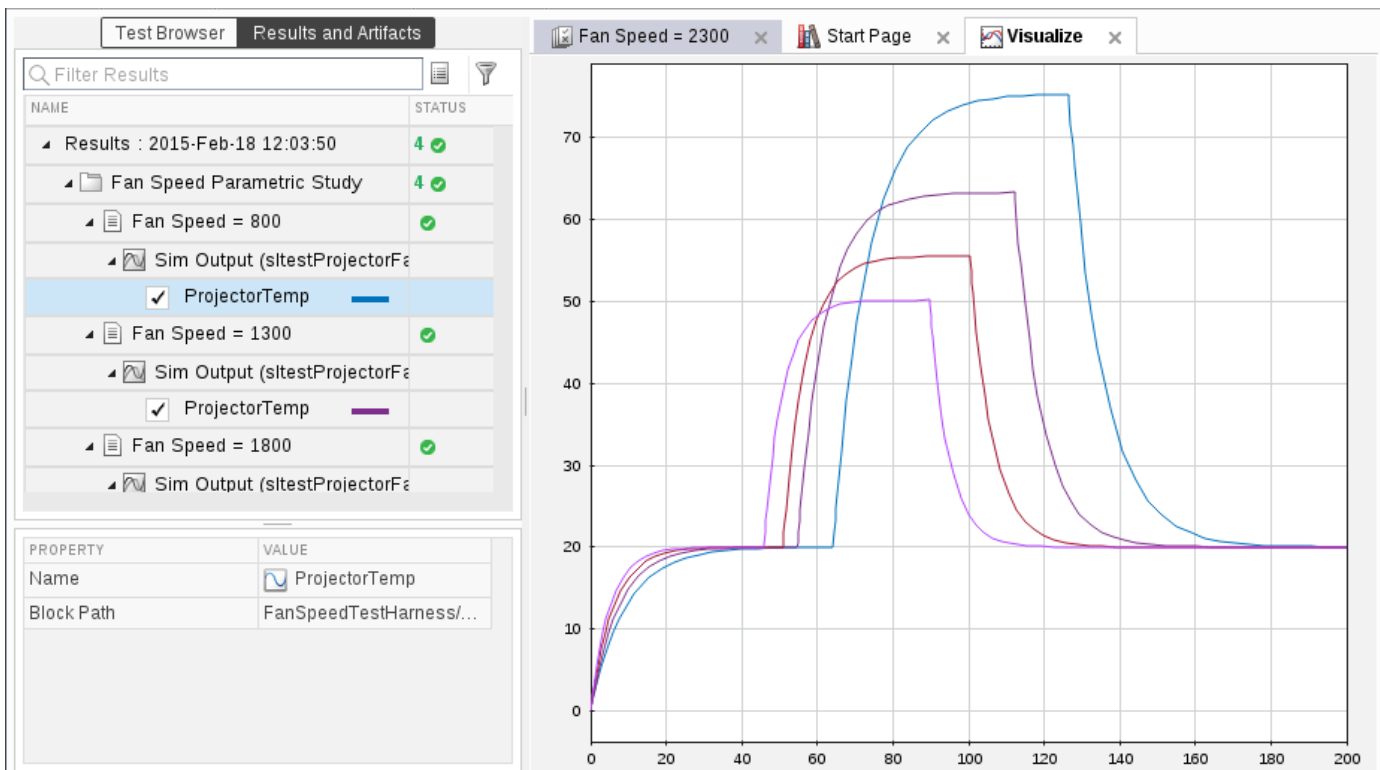
▼ PRE-LOAD ▶

```

1 % Runs before the model loads and any model callbacks
2 Fan_Speed = 1300;
```

Run the Test

In the **Test Browser**, highlight **Fan Speed Parametric Study** and click **Run**. When the test suite simulation completes, open the results for each test case and select **ProjectorTemp**. View the results in the Test Manager.



Export the Data

With the Test Manager you can export data for post-processing. In the **Results and Artifacts** pane of the Test Manager, right-click **Sim Output** for each test case and select **Export**.

This example includes the exported data in four MAT files, located in the example folder:

```
ProjectorTempFanSpeed800.mat  
ProjectorTempFanSpeed1300.mat  
ProjectorTempFanSpeed1800.mat  
ProjectorTempFanSpeed2300.mat
```

Investigate Response Time and Maximum Projector Temperature

Since the test sequence transitions execute when the system reaches steady-state, and the fan speed changes the system response, the lamp activates at different simulation times for each of the four test cases. Simplify the graphical results analysis by plotting each response with the lamp activation at the same time.

Extract the lamp activation response data, and plot the system response for the four fan speeds. Evaluate the results against these criteria:

- The temperature shall not exceed 65 deg C.
- The lamp emits visible light above 45 deg C. Minimize the time to reach this temperature.

Load the results. At the command line, enter

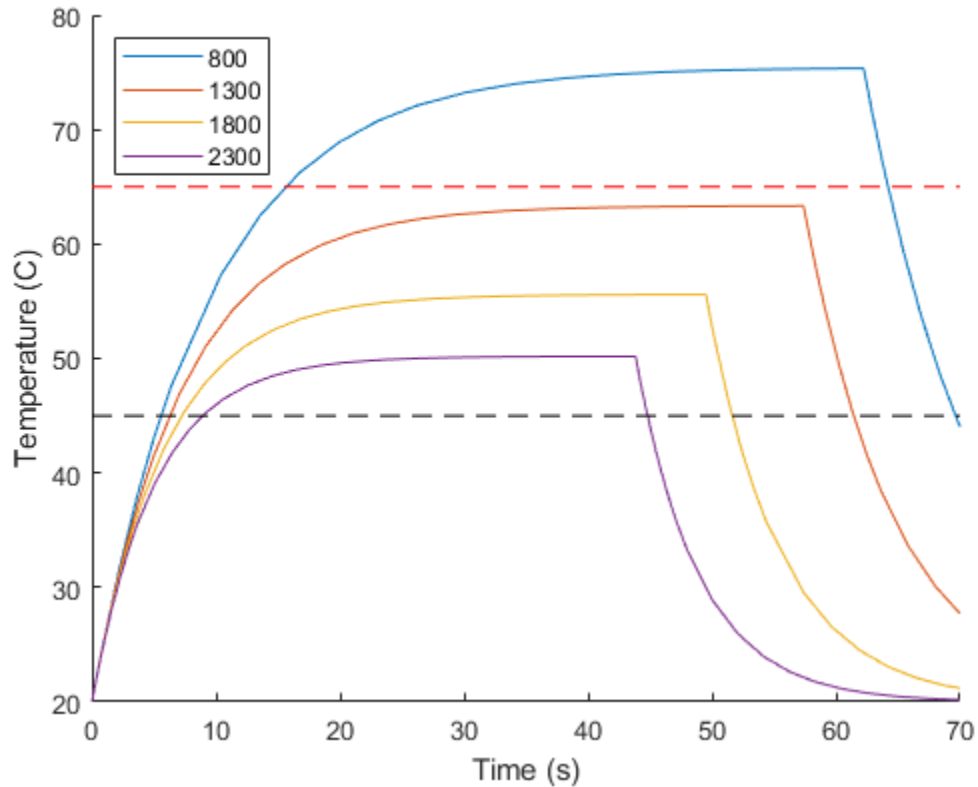
```
DataAt800 = load('ProjectorTempFanSpeed800.mat');  
DataAt1300 = load('ProjectorTempFanSpeed1300.mat');  
DataAt1800 = load('ProjectorTempFanSpeed1800.mat');  
DataAt2300 = load('ProjectorTempFanSpeed2300.mat');
```

The script `ArrangeProjectorData.m` arranges the temperature and power on data from the output for each run.

```
ArrangeProjectorData
```

The script `PlotProjectorThermalResponse.m` plots the thermal response of the projector after the lamp activates, for each of the fan speeds.

```
PlotProjectorThermalResponse
```



Results Interpretation

The results show that while the highest fan speed results in the lowest maximum temperature, it also takes the longest time to reach the lamp activation temperature. The lowest fan speed results in the fastest lamp activation, but the system exceeds the maximum specified temperature by a significant margin.

Fan speed = 1300 keeps the system under the maximum temperature spec, and the system also reaches lamp activation temperature approximately 3 seconds faster than with the highest fan speed.

```
close_system(Model,0);
```

```
clear Model;
clear Harness;
clear TestSuite;
close(figure(1));
```

See Also

[Test Manager](#) | [Test Sequence](#)

More About

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

Projector Controller Testing Using verify and Real-Time Tests

Perform real-time testing on a target computer and verify simulation and real-time results.

This example demonstrates testing a projector control system using model simulation and real-time execution on a target computer. The tests verify the controller by using test sequence scenarios that exercise the top-level controller model. The controller uses a push button input and a temperature sensor input, and outputs signals controlling the fan, fan speed, and projector lamp.

This example uses Simulink® Real-Time™. Before beginning, review the Simulink Real-Time system requirements.

Set the test file, model, and internal harness names for the example.

```
testFile = 'sltestProjectorCtrlTests.mldatx';
model = 'sltestProjectorController';
testharness = 'Test_Scenarios';
```

Open the model.

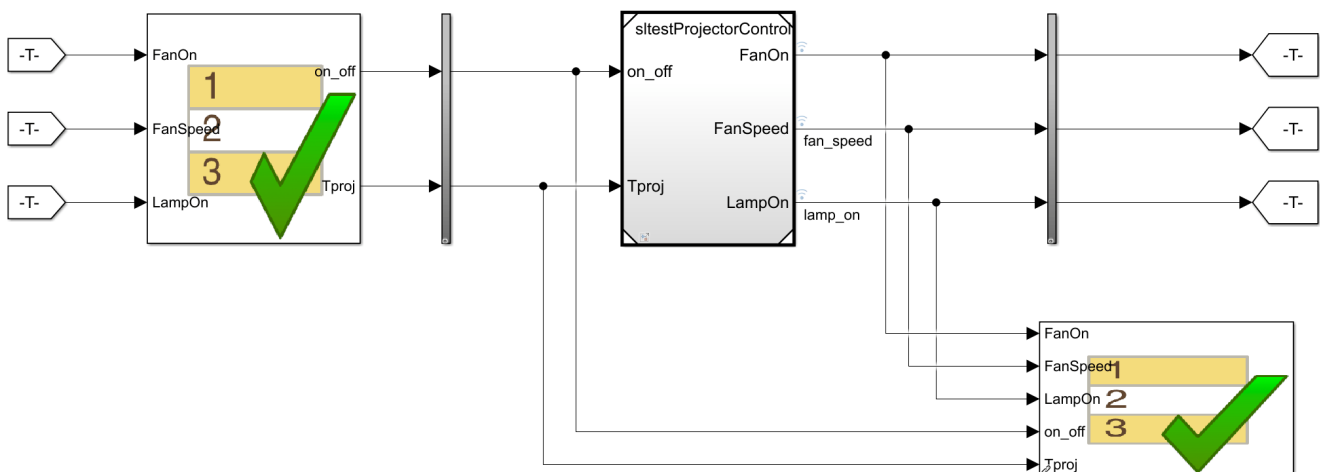
```
open_system(model)
```

View the Test Harness

Open the Test_Scenarios internal test harness.

```
sltest.harness.open(model, testharness);
```

The test harness uses a Test Sequence block to define the test scenarios and a Test Assessment block to verify the results.



In the test harness, open the Test Sequence block to view the scenarios, which are defined in tabs.

Nominal_Operation	On_Off_Button	Basic_Overheat	Overheat_Shutoff	+
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;				

Open the Test Assessment block to view the verify statements.

Step	Transition	Next Step	Description
GlobalAssess verify(~LampOn == true FanOn == true,... 'Simulink:verify_lamp_implies_fan',... 'Fan must be on if lamp is on');			The logical statement inside this verify statement is equivalent to LampOn -> FanOn. See requirement 1.4.
OverheatCondition when Tproj > 65 verify(LampOn == false &&... FanOn == true && FanSpeed == High,... 'Simulink:verify_overheat',... 'In overheat condition lamp must be off and fan high');			
HighTempCondition when Tproj > 58 verify(FanOn == true && FanSpeed == High,... 'Simulink:verify_high_temp',... 'In high temp condition fan must be high');			
NormalCondition when (Tproj < 58 && LampOn == true) verify(duration(FanSpeed == High) <= 10,... 'Simulink:verify_fan_normal',... 'In normal condition fan must return to normal within 10 s');			This verifies that the fan is never at high for longer than 10 sec if the projector is at a normal temp. See requirement 1.8.
Else			

Open the Test File and Configure the Real-Time Target Computer

Open the test file in the Test Manager by entering:

```
open(testFile)
```

The test file contains a test suite with two test cases, each of which tests the four test scenarios. The **Simulation_Tests** test case simulates the model, and the **HIL_Tests** test case runs the tests on a real-time target computer.

Before running the example:

- 1 Configure your target computer using the Simulink Real-Time Explorer.
- 2 Connect to your target computer.
- 3 If your target computer is not the default target, update **Target Computer** in the **HIL_Tests** test case's **System Under Test** section.

For more information on real-time configuration see “System Configuration” (Simulink Real-Time).

Run the Model Simulation Tests

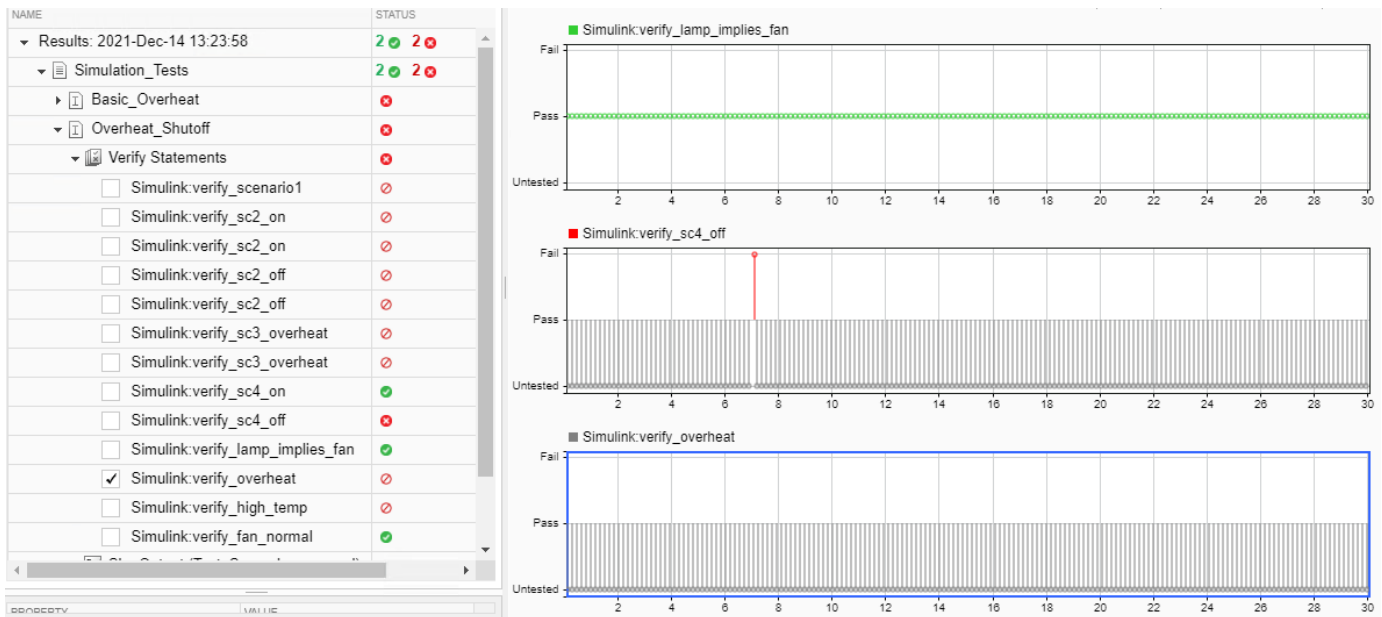
Run the **Simulation_Tests** test case. After simulation completes, click the **Results and Artifacts** pane in the test manager.

Expand the **Simulation_Tests** results and each scenario to see the **Verify Statements** results. The **verify** statements demonstrate fail, pass, and untested results:

- In all scenarios except **Basic_Overheat**, the controller does not operate in high-temperature or overheat mode, so the **verify_overheat** and **verify_high_temp** statements of the associated **verify** statements are untested.
- In all scenarios, the controller passes the test that if the lamp is on, the fan is also on: **verify_lamp_implies_fan**.
- In the **Overheat_Shutoff** scenario, the controller passes the test that the system stays off if the **on_off** button is pressed when the temperature is above a limit: **verify_sc4_on**. For other scenarios, **verify_sc4_on** is untested.
- In only the **Overheat_Shutoff** scenario, the controller fails the test that the system shuts off if the **on_off** button is pressed when the temperature is above a limit: **verify_sc4_off**. Resolving this failure requires modifying the **OnOff Check** subsystem in the main model.

For more information, see “Assess Model Simulation Using **verify** Statements” on page 3-18.

In the **Overheat_Shutoff** scenario, select the **verify_sc4_off**, **verify_lamp_implies_fan**, and **verify_overheat** results to visualize the **verify** statement results.



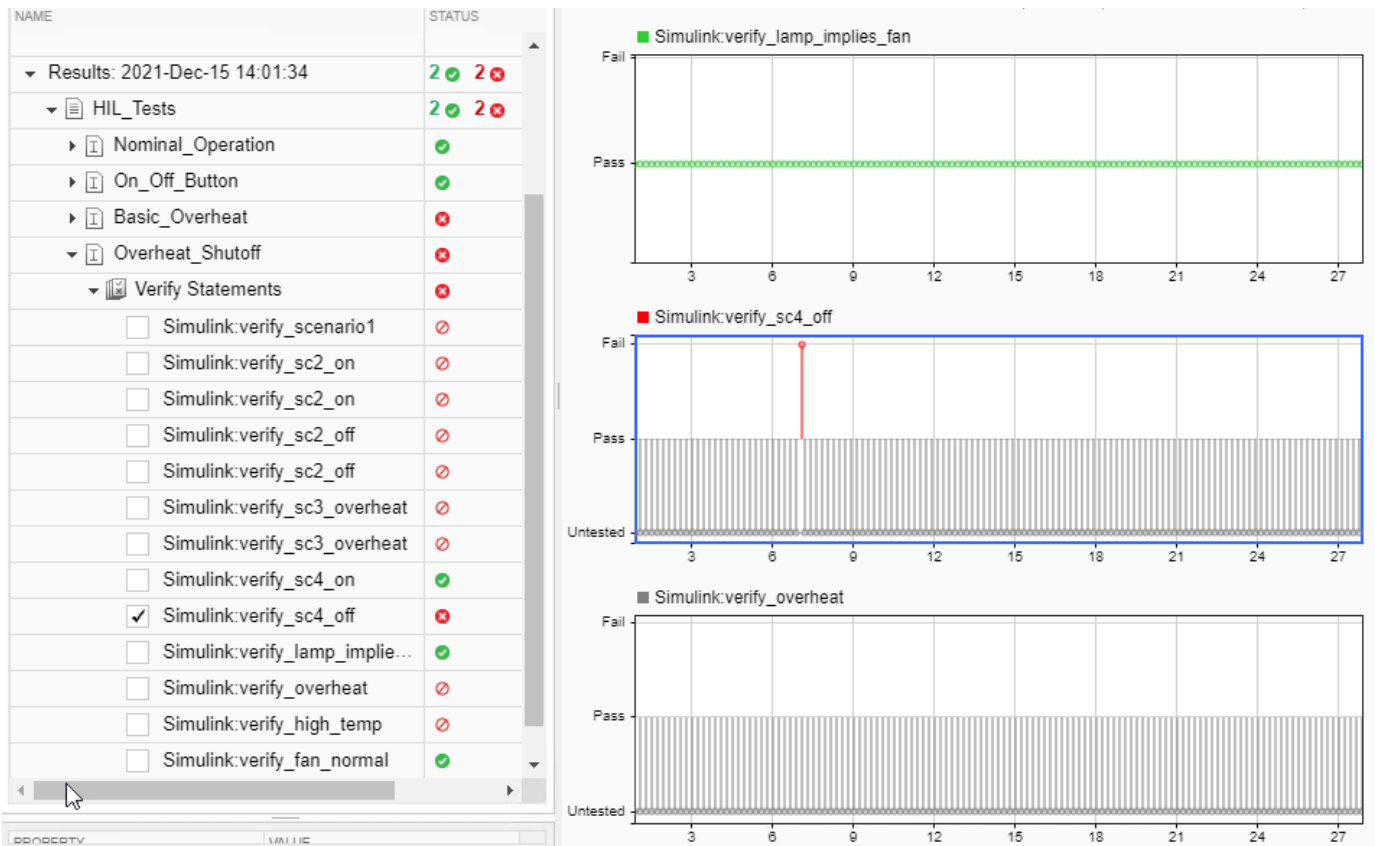
Execute the Real-Time Tests and Review the Results

The real-time test case (**HIL_Tests**) verifies that real-time execution results match model simulation results, and that the verify statements pass.

In the Test Manager, run the real-time test case (**HIL_Tests**).

The Results of the Simulation_Tests and HIL_Tests show matching pass, fail, and untested statuses.

In the **Overheat_Shutoff** scenario, select the `verify_sc4_off`, `verify_lamp_implies_fan`, and `verify_overheat` results to visualize the verify statement results. The **Verify Statements** section shows similar results to the model simulation.



```

close_system(testharness,0)
close_system(model,0)
sltest.testmanager.clear;
sltest.testmanager.clearResults;
sltest.testmanager.close
clear testFile testHarness model;

```

Test Execution Order

When you execute a test, Simulink Test opens the model to be tested, runs callback functions, closes the models, and cleans up. If you have scripted iterations, they run before the model is loaded. The order in which tests execute depends on:

- Whether you run a single test case or run a test suite containing one or more test cases
- The number of models tested
- The number of test cases

For serial simulations, the test cases run in the order they are listed in the Test Manager. To change the order in which test cases run in the Test Manager, drag and drop test cases into the desired order in the **Test Browser** pane. You cannot drag and drop test suites. If you run your simulations in parallel mode, the test cases might not run in the order displayed in the Test Manager.

If your test cases include callbacks, note that `disp` and `fprintf` do not work in callbacks. Instead, use the `ShowSimulationLogs` property of `sltest.testmanager.setpref`. To verify that the callbacks are executed, use a MATLAB script that includes breakpoints in the callbacks.

Single Test Case on a Single Model

If you select a specific test case to run on a single model and the model is not open before the test runs, the execution order is:

- 1 Run test case Pre-Load callback.
- 2 Run model PreLoadFcn callback.
- 3 Load model.
- 4 Run model PostLoadFcn callback.
- 5 Run test case Post-Load callback.
- 6 Simulate model.
- 7 Run test case Cleanup callback.
- 8 Run model CloseFcn callback.

If you run a test suite that contains a test case, the test suite Setup callback runs before the first step and the test suite Cleanup callback runs after the last step.

Multiple Test Cases on Multiple Models

If you run multiple test cases that run on separate models and the models are not open before the test runs, the execution order, which is shown for two test cases run on two models, is:

- 1 Run test case 1 Pre-Load callback.
- 2 Run model 1 PreLoadFcn callback.
- 3 Load model 1.
- 4 Run model 1 PostLoadFcn callback.
- 5 Run test case 1 Post-Load callback.
- 6 Simulate model 1 for test case 1.

- 7 Run test case 1 Cleanup callback.
- 8 Run test case 2 Pre-Load callback.
- 9 Run model 2 PreLoadFcn callback.
- 10 Load model 2.
- 11 Run model 2 PostLoadFcn callback.
- 12 Run test case 2 Post-Load callback.
- 13 Simulate model 2 for test case 2.
- 14 Run test case 2 Cleanup callback.
- 15 Run model 1 CloseFcn callback.
- 16 Run model 2 CloseFcn callback.

If you run a test suite that includes test cases, the test suite Setup callback runs before the first step and the test suite Cleanup callback runs after the last step.

The order in which models are closed using the CloseFcn might be different than the order in which they were opened or run. In the above example, steps 15 and 16 might be switched.

Multiple Test Cases in a Single Test Suite on a Single Model

If you run multiple test cases in a test suite on a single model and the model is not open before the test runs, the execution order, which is shown for two test cases, is:

- 1 Run test case 1 Pre-Load callback.
- 2 Run model PreLoadFcn callback.
- 3 Load model.
- 4 Run model PostLoadFcn callback.
- 5 Run test case 1 Post-Load callback.
- 6 Simulate model.
- 7 Run test case 1 Cleanup callback.
- 8 Run test case 2 Pre-Load callback.
- 9 Run test case 2 Post-Load callback.
- 10 Simulate model.
- 11 Run test case 2 Cleanup callback.
- 12 Run model CloseFcn callback.

If the model is open before the test runs, the execution order, which is shown for two test cases, is:

- 1 Run test case 1 Pre-Load callback
- 2 Run test case 1 Post-Load callback
- 3 Simulate model
- 4 Run test case 1 Cleanup callback
- 5 Run test case 2 Pre-Load callback
- 6 Run test case 2 Post-Load callback
- 7 Simulate model

8 Run test case 2 Cleanup callback

Notice that the model `PreLoadFcn` and `PostLoadFcn` callbacks do not execute because the model is already loaded before the test runs. The model `CloseFcn` callback does not execute either because the model is left open after test completion.

Multiple Test Cases in Multiple Test Suites on a Single Model

Suppose you have two test suites that each contain two test cases, such as:

- Test suite 1
 - Test case 1-1
 - Test case 1-2
- Test suite 2
 - Test case 2-1
 - Test case 2-2

The execution order of the callbacks is:

- 1 Run test suite 1 Setup callback.
- 2 Run test suite 2 Setup callback.
- 3 Run test case 1-1 Pre-Load callback.
- 4 Run test case 1-2 Pre-Load callback.
- 5 Run test case 2-1 Pre-Load callback.
- 6 Run test case 2-2 Pre-Load callback.
- 7 Run test case 1-1 PostLoad callback.
- 8 Simulate model.
- 9 Run test case 1-1 Cleanup callback.
- 10 Run test case 1-2 Post-Load callback.
- 11 Simulate model.
- 12 Run test case 1-2 Cleanup callback.
- 13 Run test suite 1 Cleanup callback.
- 14 Run test case 2-1 Post-Load callback.
- 15 Simulate model.
- 16 Run test case 2-1 Cleanup callback.
- 17 Run test case 2-2 Post-Load callback.
- 18 Simulate model.
- 19 Run test case 2-2 Cleanup callback.
- 20 Run test suite 2 Cleanup callback.

Test Case with Parameter Overrides

For a test case with parameter overrides, the execution order is:

- 1 Run test case Pre-Load callback.
- 2 Load model.
- 3 Read parameter overrides, which triggers a model update.
- 4 Run test case Post-Load callback.
- 5 Simulate model.
- 6 Run test case Cleanup callback.
- 7 Run model CloseFcn callback.

Filter Test Execution, Results, and Coverage

In this section...

“Add Tags” on page 6-213
 “Filter Tests and Results” on page 6-213
 “Run Filtered Tests” on page 6-213
 “Filter Coverage” on page 6-213

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.

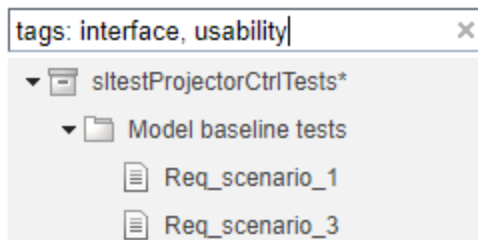
Add Tags

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 tags: id1, id2, ... where id1 and id2 are example test tags. Enter multiple tags separated by commas to return tests containing any tag in the list.



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**.

Filter Coverage

For information on filtering test coverage, see “Coverage Filtering Using the Test Manager” on page 6-140

Test Manager Results and Reports

- “View Test Case Results” on page 7-2
- “Debugging Test Failures Using Model Slicer” on page 7-7
- “Export Test Results” on page 7-16
- “Generate Test Results Reports” on page 7-17
- “Generating a Test Results Report” on page 7-20
- “Customize Test Results Reports” on page 7-21
- “Append Code to a Test Report” on page 7-25
- “Results Sections” on page 7-27
- “Generate Test Specification Reports” on page 7-30
- “Customize Test Specification Reports” on page 7-34
- “Debugging Equivalence Test Failures Using Model Slicer” on page 7-41

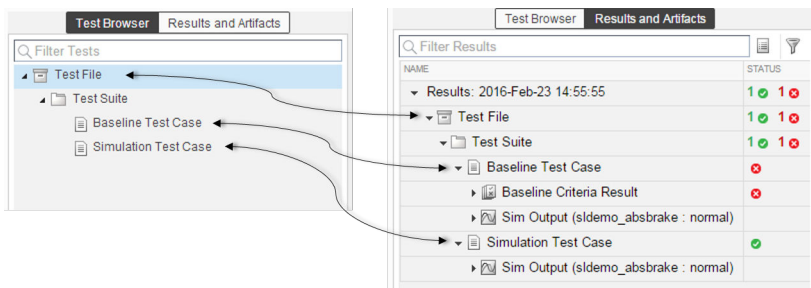
View Test Case Results

In this section...

“View Results Summary” on page 7-2

“Visualize Test Case Simulation Output and Criteria” on page 7-3

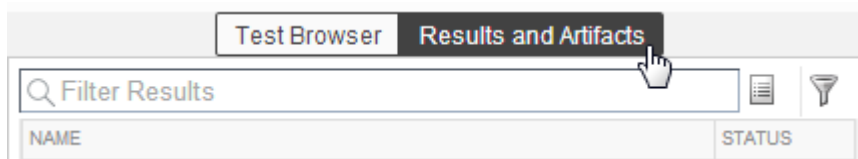
After a test case has finished running in the Test Manager, the test case result becomes available in the **Results and Artifacts** pane. Test results are organized in the same hierarchy as the test file, test suite, and test cases that were run from the **Test Browser** pane. In addition, the **Results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

▼ SUMMARY

Name	Baseline Test Case
Outcome	1 ✕
Start Time	02/23/2016 14:55:55
End Time	02/23/2016 14:55:56
Type	Baseline Test
Test File Location	C:\MATLAB\Test File.mldatx
Test Case Definition	↩
Rerun Test Case	▶
Baseline File	C:\MATLAB\test_capture.mat
Cause of Failure	Criteria evaluation resulted in failure.
▶ Simulation Metadata	

▼ 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.



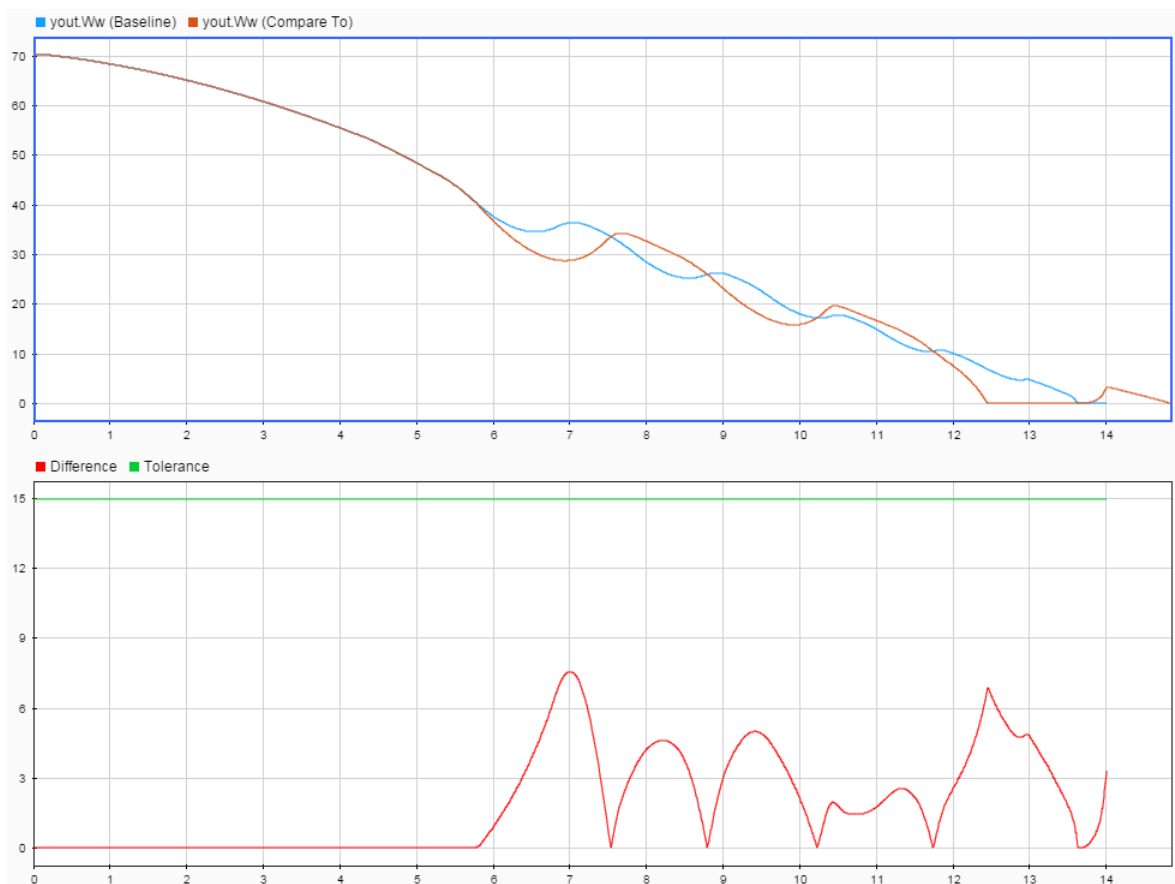
To view equivalence or baseline criteria comparisons:

- 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 “Compare Model Output to Baseline Data” on page 6-7.

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**.

See Also

Test Manager

More About

- “Collect Coverage in Tests” on page 6-135

Debugging Test Failures Using Model Slicer

This example shows how to debug Simulink Test baseline and verification failures by using the Model Slicer.

A predefined baseline test case is provided for use with the `sltestDemo_fuelsys` model. The baseline is captured from an earlier state of the model. After capturing % the baseline, a design error is introduced in the model, which causes the baseline test to fail. Then, the Model Slicer is used to debug the failure and localize the design error.

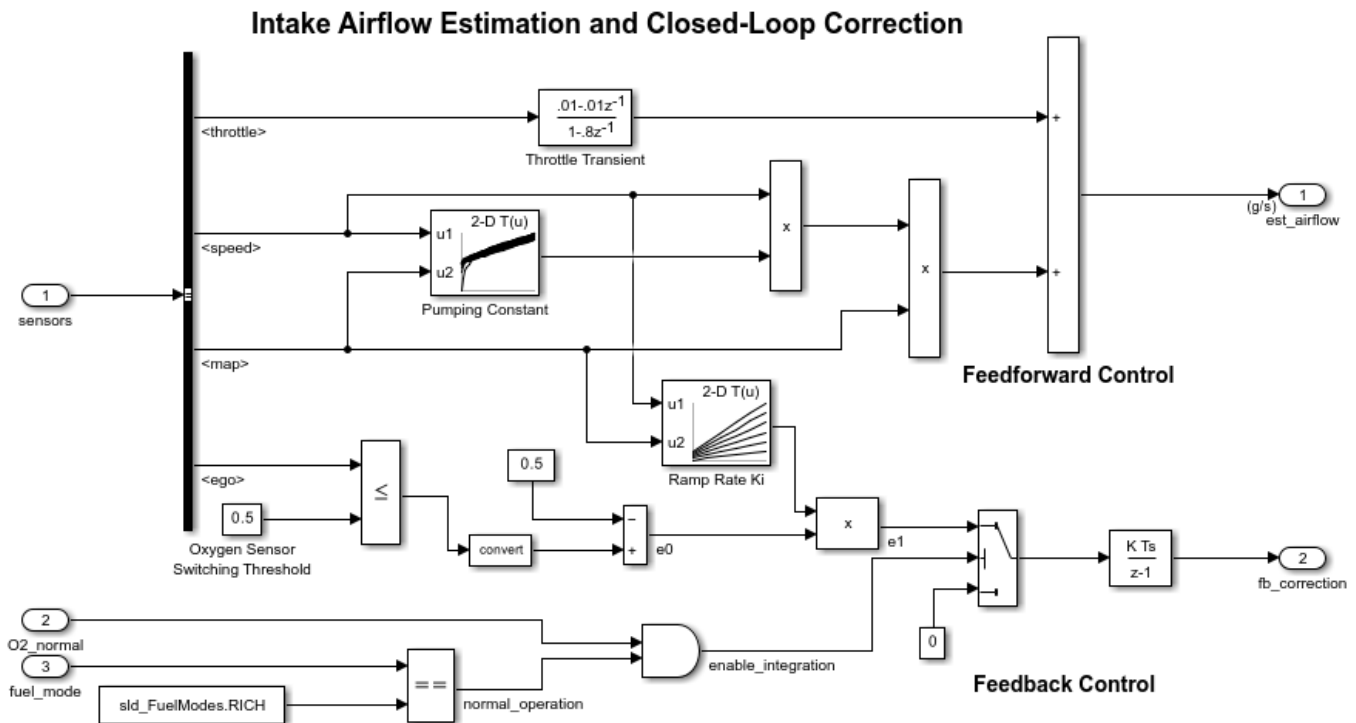
For information about the model slicer, see the Simulink Check documentation.

Step 1: Setting Up the Artifacts

This section describes how to run the test case and view the results.

1. Open the `sltestDemo_fuelsys` model.

```
open_system('sltestDemo_fuelsys');
```

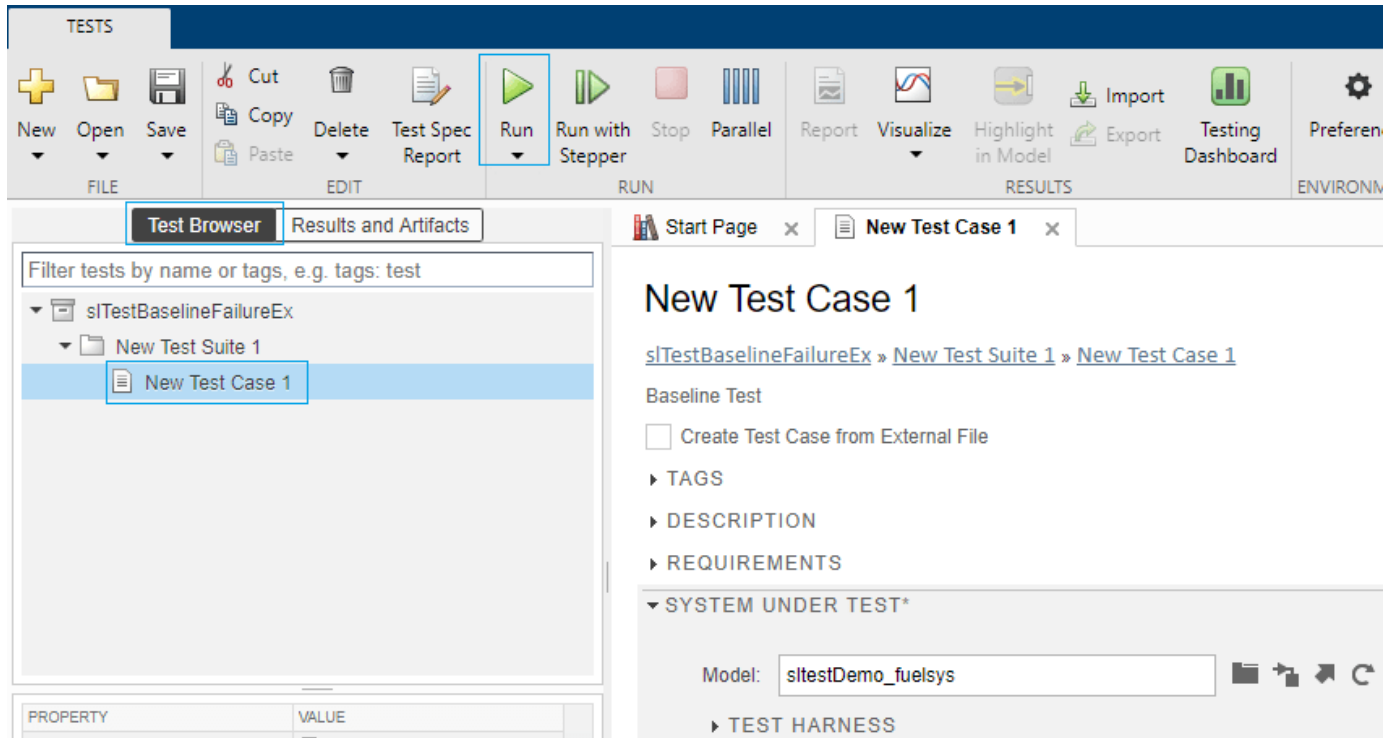


Copyright 1998-2020 The MathWorks, Inc.

2. Click **APPS** > **Model Verification, Validation, and Test** > **Simulink Test** to open the Simulink Test toolstrip.

3. Click **Tests** > **Simulink Test Manager** to open the Test Manager.

4. To open the existing test file, from the Test Manager toolbar, click **Open** and select `sITestBaselineFailureEx`.
5. After the test file loads, select **New Test Case1** in the **Test Browser** pane.
6. Click **Run**.



7. The new test results appear at the top of the **Results and Artifacts** pane. Right-click the result and select **Expand All Under**, so that you see the **Baseline Criteria Result** and the **Verify Statements**.

The screenshot shows the 'Test Browser' interface with the 'Results and Artifacts' tab active. A search bar at the top allows filtering results by name or tags. Below the search bar is a table with two columns: 'NAME' and 'STATUS'. The table lists various test results, including a summary for 'Results: 2020-Dec-08 18:49:01' showing 1 failure. Underneath, there are several test cases and criteria, some of which are failed. The failed items are highlighted with blue boxes: 'air_fuel_ratio', 'ego', 'fuel', and 'FuelModeAssertion'. The 'Sim Output (sltestDemo_fuelsy)' is expanded and highlighted in blue. At the bottom, there is a table with 'PROPERTY' and 'VALUE' columns, showing the 'Name' as 'Sim Output (sltestDemo...)' and the 'Model' as 'sltestDemo_fuelsys'.

NAME	STATUS
Results: 2020-Dec-08 18:49:01	1 ✖
New Test Case 1	✖
Baseline Criteria Result	✖
<input checked="" type="radio"/> air_fuel_ratio	✖
<input type="radio"/> speed	✔
<input type="radio"/> map	✔
<input checked="" type="radio"/> ego	✖
<input type="radio"/> throttle	✔
<input checked="" type="radio"/> fuel	✖
Verify Statements	✖
<input type="checkbox"/> CheckRange	✔
<input checked="" type="checkbox"/> FuelModeAssertion	✖
Sim Output (sltestDemo_fuelsy)	

PROPERTY	VALUE
Name	Sim Output (sltestDem...
Model	sltestDemo_fuelsys

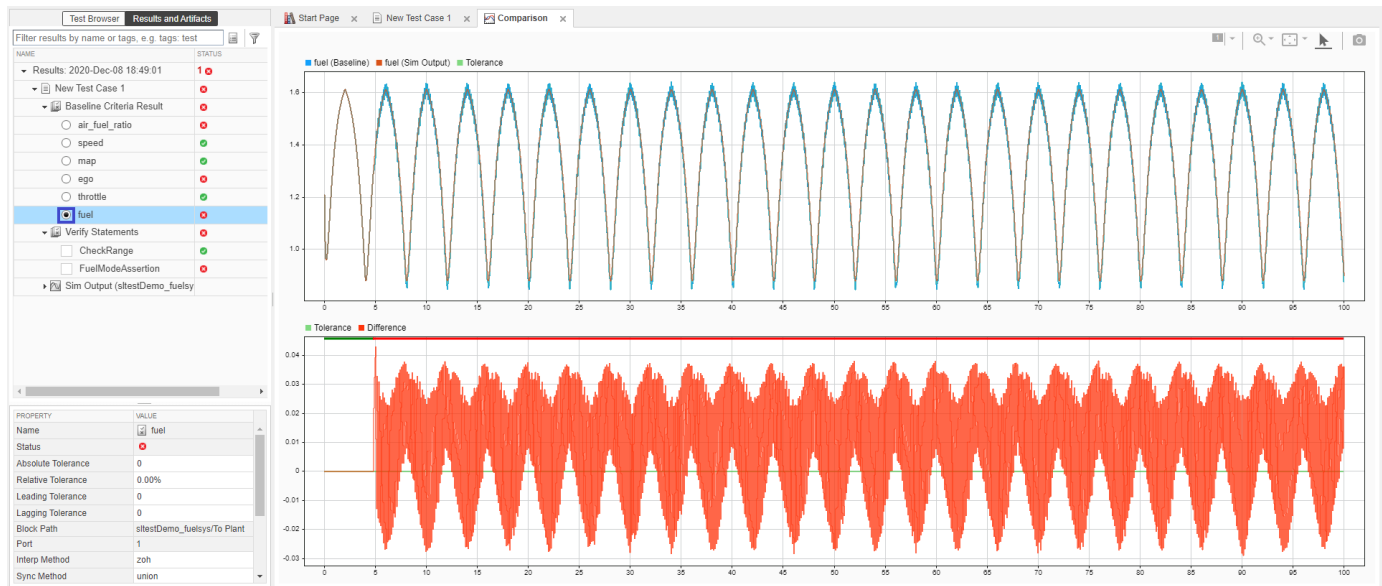
Observe that four signals have failed: `air_fuel_ratio`, `ego`, `fuel`, and `FuelModeAssertion`. This example uses the failed `fuel` signal to illustrate the debugging workflow.

Step 2: Entering Debug Session

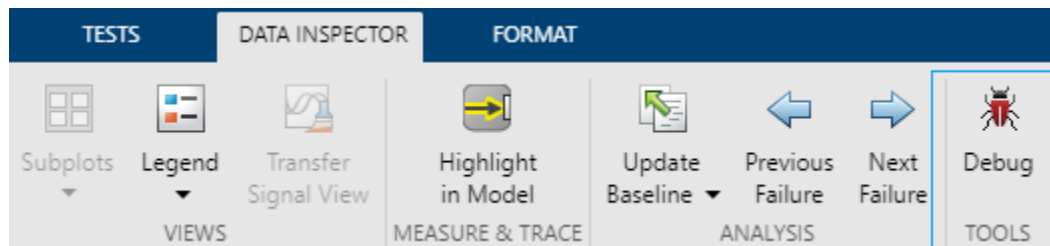
This section describes how to setup the Model Slicer for debugging the failed `fuel` signal.

1. To compare the `fuel` signals between the model and the baseline, expand the **Baseline Criteria Result** and select the radio button next to the `fuel` signal. Likewise, to debug a verify signal, expand the **Verify Statements** and select the failed verify signal. Another way to select a failed signal is from the **Signal to Debug** dropdown list in the toolbar.

In the plot area, compare the model output to the baseline data.

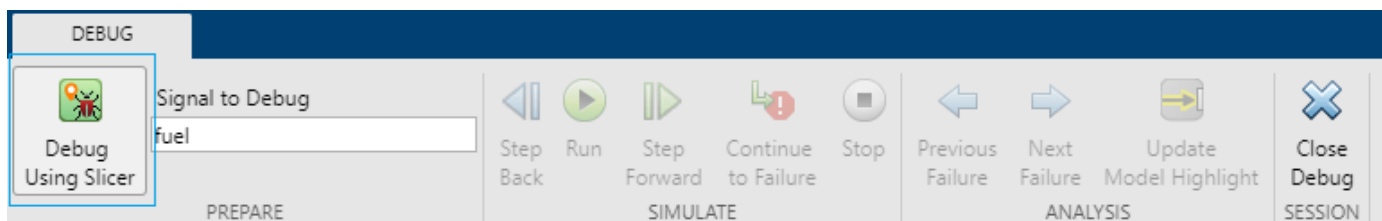


2. Click **Debug** in the TOOLS section of the toolstrip. Note that the **Debug** option is enabled only when a failed baseline or verify signal is plotted.



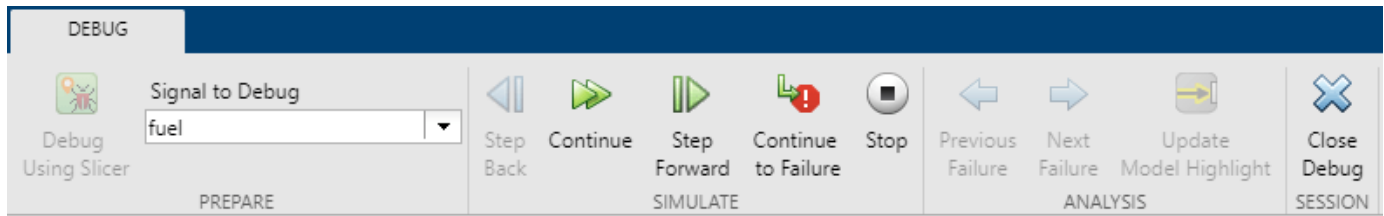
The **DEBUG** tab replaces all existing toolstrip tabs. Multiple Test Manager options are hidden or disabled to create the debug environment.

3. To set up the Model Slicer, click **Debug Using Slicer**.



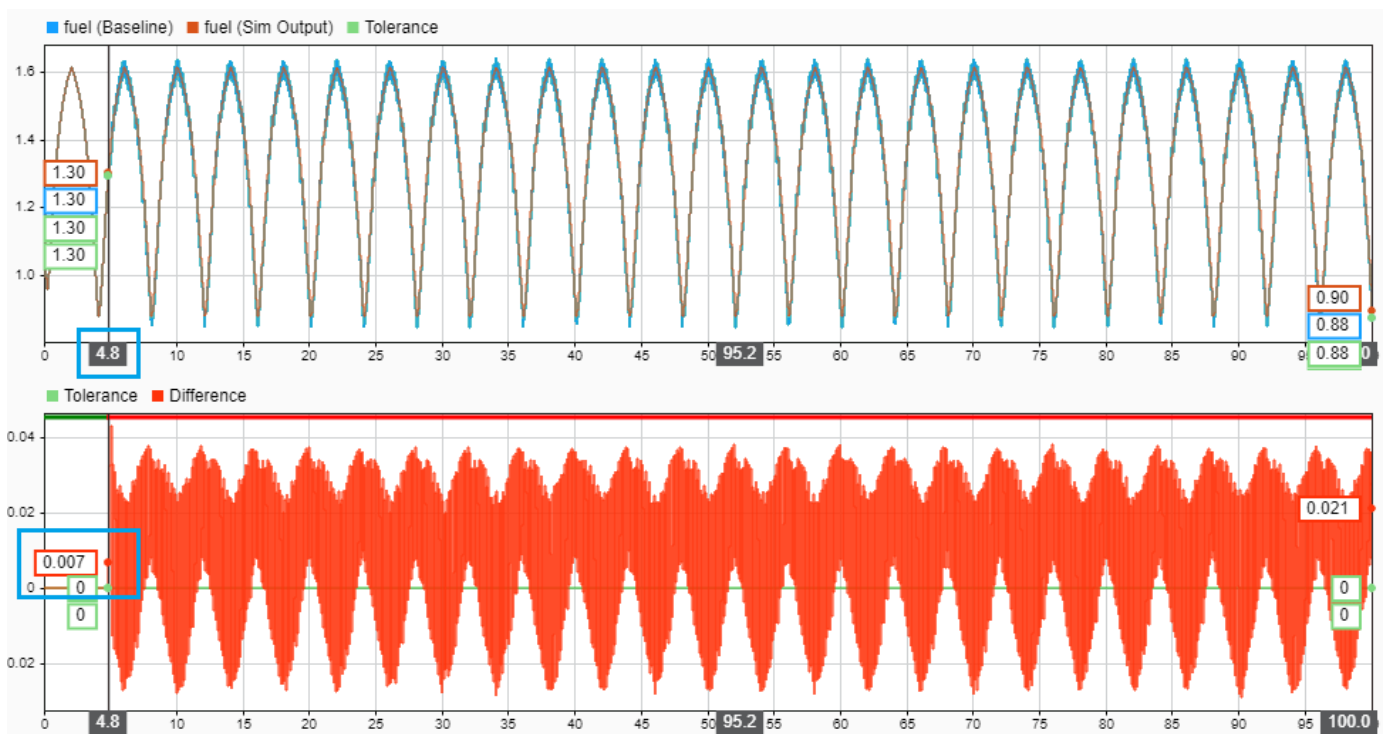
Debug Using Slicer prepares the debugging session by:

- 1 Rerunning the test case and creating new debugging results. This makes sure that the failure still exists in the current state of the test model.
- 2 Launching the Model Slicer on the test model.
- 3 Automatically plotting the selected failed signal in the debugging results, and setting the failed signal as the starting time point.

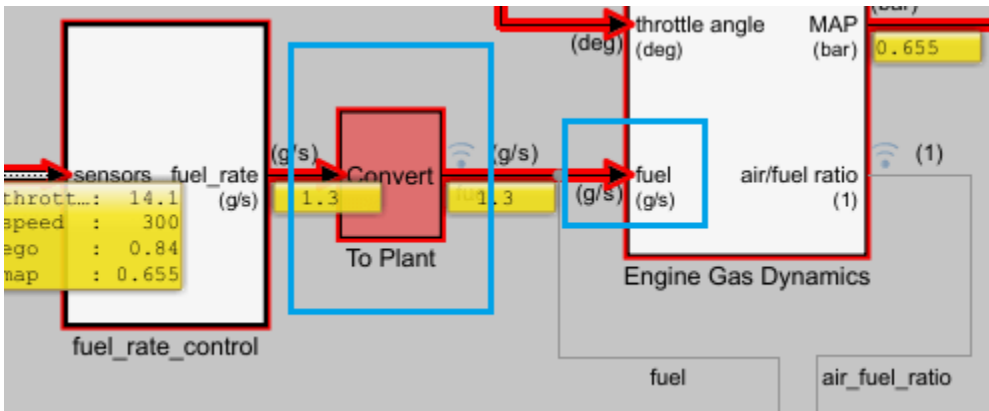


Observe these changes at the failure:

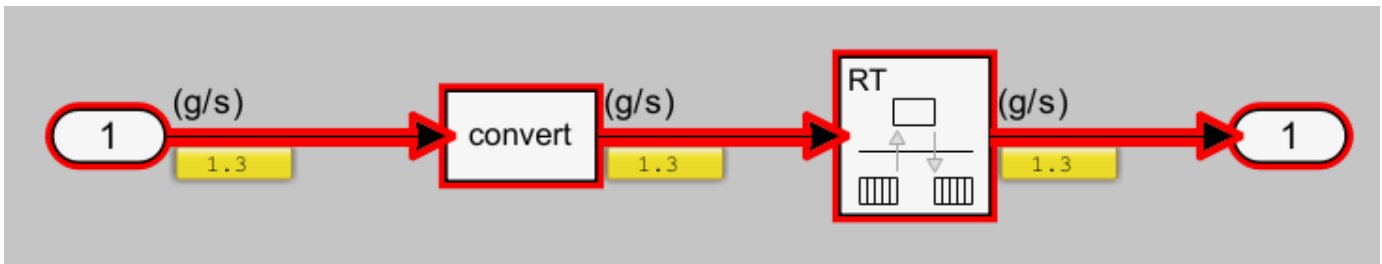
- Simulation pauses at $T = 4.81$.
- Data cursors update accordingly.
- Difference between the Baseline and Sim Output is 0.007.



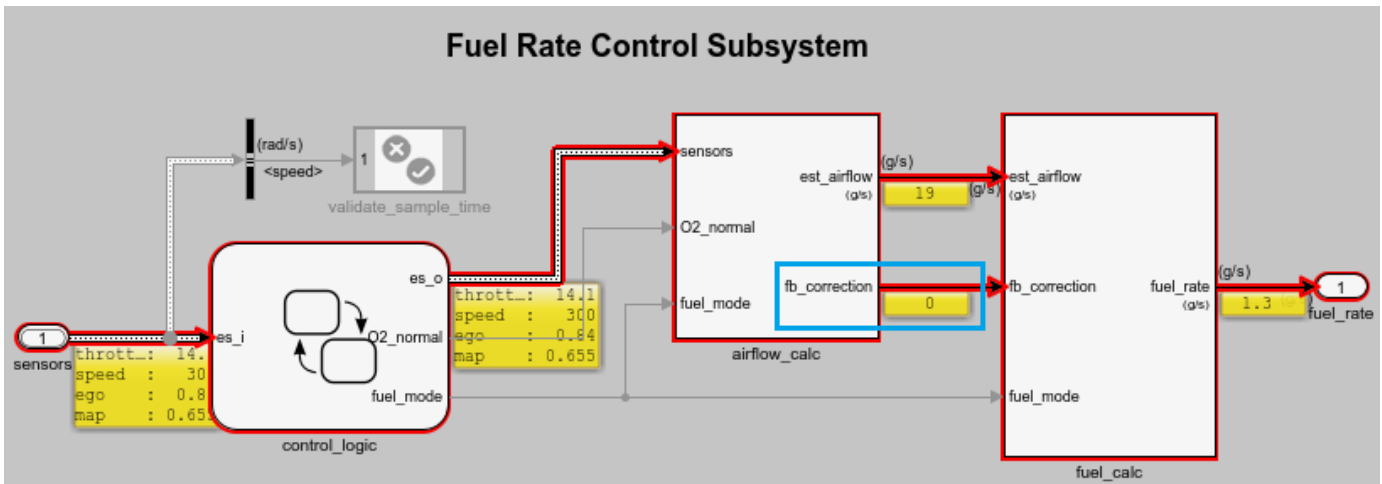
From the Model Slicer highlighting, you can find the cause of this difference, and see that the `sltestDemo_fuelsys/To Plant/fuel` value depends on `sltestDemo_fuelsys/To Plant`.



3. Open `sltestDemo_fuelsys/To Plant`. Notice that there is no change in the value being propagated.

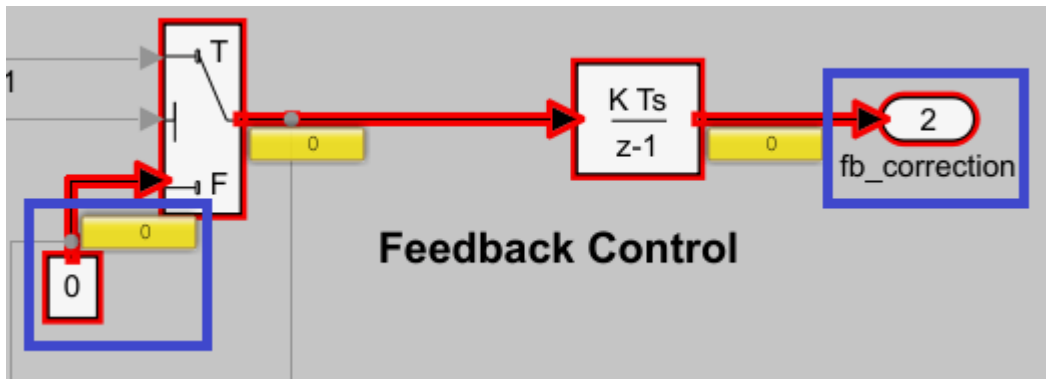


4. Open `sltestDemo_fuelsys/fuel_rate_control`.



Observe that the `fb_correction` value is 0. The difference between the Baseline and the Sim Output is 0.007, which is a small value. It might be that `fb_correction` is not calculated correctly.

5. Open `sltestDemo_fuelsys/fuel_rate_control/airflow_calc`, which computes the `fb_correction`, and observe the data dependencies.

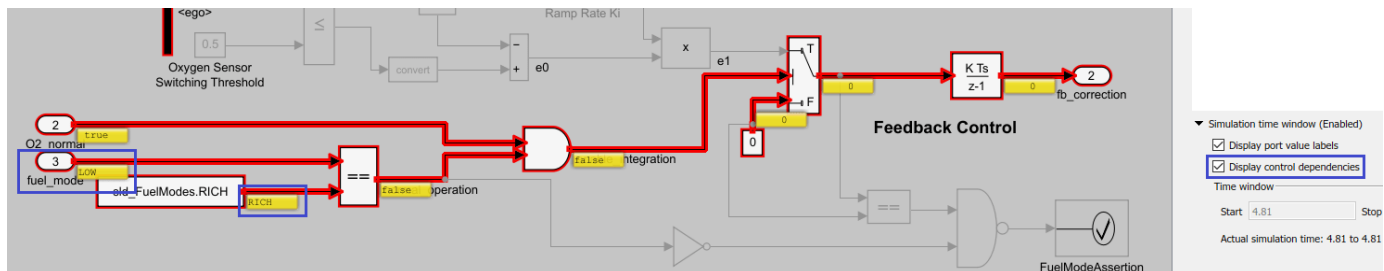


Notice that the constant, 0, is being passed through the `sltestDemo_fuelsys/fuel_rate_control/airflow_calc/hold` integrator switch block. To determine why the control port evaluates to false for the switch block, the control dependencies need to be highlighted on the model.

6. Enable **Display Control Dependencies** from the **Simulation Time Window** section in the Model Slicer Dialog docked on the model.

Observe that:

- `sltestDemo_fuelsys/fuel_rate_control/airflow_calc/fuel_mode` value is LOW, but `fb_correction` is still **zero**.
- `fuel_mode` is compared to `sltestDemo_fuelsys/fuel_rate_control/airflow_calc/Enumerated Constant`, which evaluates to false.



The Enumerated Constant value is set to `sld_FuelModes.RICH`. It should be checking against the `fuel_mode` value `sld_FuelModes.LOW`.

Step 4: Incorporating the fix

1. Exit the debugging session by clicking **SESSION > Close Debug**.
2. Open the model and update the `sltestDemo_fuelsys/fuel_rate_control/airflow_calc/Enumerated Constant` value to `sld_FuelModes.LOW`.
3. Save the model.
4. Run the test case and view the results.

Now, observe that the test results show the test as having passed.

Capabilities and Limitations

- If you use the Test Manager to set a simulation mode to one other than normal mode, such as SIL or PIL, you cannot use the Model Slicer for debugging.
- If the simulation mode is set in the model configuration, the Model Slicer changes the mode of the model and all referenced models to run in normal mode and then, you can use the Model Slicer for debugging.
- For models that do not support Fast Restart mode, the SIMULATION section of the toolstrip is disabled. Use the ANALYSIS section to debug the failure.
- The ANALYSIS section is available only when the model is not simulating, such as when you click **Continue** or **Stop** in the SIMULATION section. It highlights a time region instead of a time step. To define a time region, move the data cursors manually, or use **Next Failure** or **Previous Failure**. Then, you can use **Update Slicer Highlight** to update the model highlighting for the defined time slice.
- The Results must be generated from the current release.

Export Test Results

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.

Test results are saved separately from the test file. To save results, select the result in the Test Manager, 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 :	

See Also

Related Examples

- “Generating a Test Results Report” on page 7-20

More About

- “Generate Test Results Reports” on page 7-17

Generate Test Results Reports

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 Test Manager, 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 In the toolstrip, click **Report**.
- 3 Enter the title page information and select the information you want to include in the report. To enable the option to specify the number of plots per page, select **Plots for simulation output and baseline**.
- 4 Select the **File Format** to use for the generated file and, if desired, change the default **File Name** and path.
- 5 If you created a customized report template, enter the path and name of the **Template File**. See “Generate Reports Using Templates” on page 7-17.
- 6 If you created a customized report class to change fonts, add tables, add model images, etc., enter the **Report Class** name. See “Generate a Report Using the Custom Class” on page 7-24.
- 7 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 6-188.
- 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

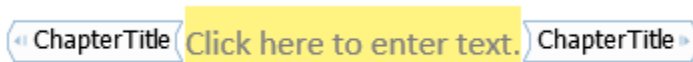
If you have a MATLAB Report Generator™ license, you can create reports from a Microsoft Word template. The resulting report is a Microsoft Word document.

The report generator in Simulink Test fills information into rich text content controls in your Microsoft Word template .dotx 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 Microsoft Word Templates” (MATLAB Report Generator).

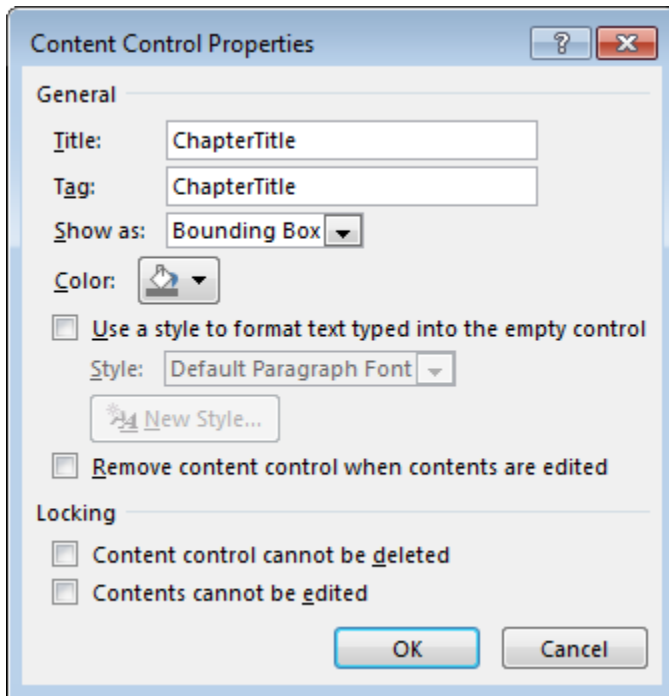
In a Microsoft Word template, you can add rich text content controls. Each Simulink Test report section can be inserted into the rich text content controls. The control names are:

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

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



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



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.
- 3 In the toolbar, click **Report**.
- 4 Enter the title page information and specify the information you want to include in the report.

- 5 Select DOCX for the **File Format**.
- 6 Specify the full path and filename of your Microsoft Word template in the **Template File** field.
- 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 by 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.
- 3 In the toolbar, click **Report**.
- 4 Enter the title page information and specify the information you want to include in the report.
- 5 Select ZIP or PDF for the **File Format**. Selecting ZIP generates an HTML report.
- 6 Specify the full path and filename of your template in the **Template File** field. 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**.

See Also

Related Examples

- “Generating a Test Results Report” on page 7-20
- “Templates” (MATLAB Report Generator)
- “Create, Store, and Open MATLAB Figures” on page 6-188

More About

- “Export Test Results” on page 7-16

Generating a Test Results Report

Report test results for a baseline test.

This example shows how to generate a test results report from the Test Manager using a baseline test case. The model used for this example is `sltestTestManagerReportsExample`.

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.

```
exampleFile = '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 shows all the test results. The report opens when it is completed.

```
sltest.testmanager.report(baselineObj, 'baselineReport.zip', ...  
    'IncludeTestResults', 0, 'IncludeComparisonSignalPlots', true, ...  
    'IncludeSimulationSignalPlots', true, 'NumPlotRowsPerPage', 3);
```

View the report when it is finished generating. In the `baselineReport.zip` folder, open the `report.html` file.

```
sltest.testmanager.clear;  
sltest.testmanager.clearResults;
```

See Also

More About

- “Generate Test Results Reports” on page 7-17

Customize Test Results Reports

In this section...

“Inherit the Report Class” on page 7-21

“Method Hierarchy” on page 7-21

“Modify the Class” on page 7-22

“Generate a Report Using the Custom Class” on page 7-24

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.

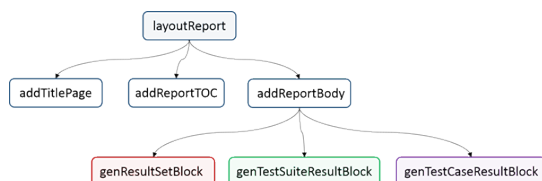
Inherit the Report Class

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”. Then, add code to the inherited class or methods to create your customizations.

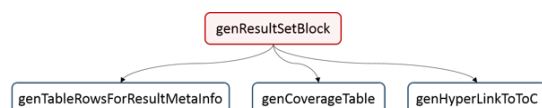
```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    %
    % Report customization code here
    %
end
```

Method Hierarchy

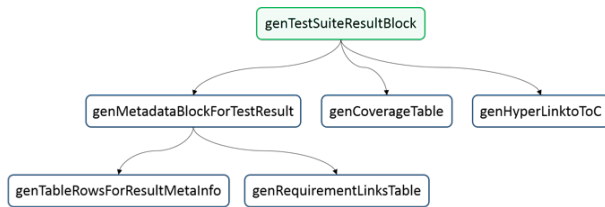
When you create the subclass, the derived class inherits methods from the `sltest.testmanager.TestResultReport` class. The body of the report is separated into three main groups: the result set block, the test suite result block, and the test case result block.



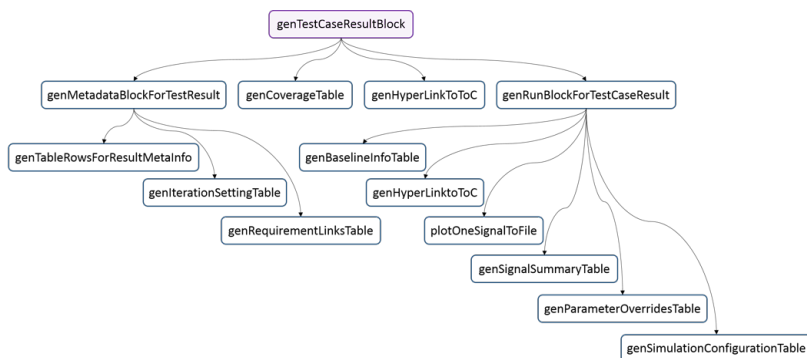
The result set block contains the result set table, the coverage table, and links to the table of contents.



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”.

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
  
```



```

end
end

```

A more complex modification of the generated report is to include a snapshot of the model that was tested.

```

% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    methods
        function this = CustomReport(resultObjects,reportFilePath)
            this@sltest.testmanager.TestResultReport(resultObjects,reportFilePath);
        end
    end

    methods(Access=protected)
        % Method to customize test case/iteration result section in the report
        function docPart = genTestCaseResultBlock(obj,result)
            % result: A structure containing test case or iteration result
            import mlreportgen.dom.*;

            % Call the superclass method to get the default behavior
            docPart = genTestCaseResultBlock@sltest.testmanager.TestResultReport(...
                obj,result);

            % Get the test case result data for putting in the report
            tcrObj = result.Data;

            % Insert model screenshot at the test case result level
            if isa(tcrObj, 'sltest.testmanager.TestCaseResult')

                % Initialize model name
                modelName = '';

                % Check in the test case result if it has model information. If
                % not, it means there were iterations in the test case or a
                % model was not used.
                testSimMetaData = tcrObj.SimulationMetaData;

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

                % Get iteration results
                iterResults = getIterationResults(tcrObj);

                % Get the model name in case test case had iterations
                if (~isempty(iterResults))
                    modelName = iterResults(1).SimulationMetaData.modelName;
                end

                % Insert model snapshot. This will not work for harnesses. With
                % minimal changes we can also open the harness used for
                % testing.
                if (~isempty(modelName))
                    try
                        open_system(modelName);
                        outputFileName = [tempdir, modelName, '.png'];
                        if exist(outputFileName,'file')
                            delete(outputFileName);
                        end
                        print(outputFileName, '-s', '-dpng');
                        para = sltest.testmanager.ReportUtility.genImageParagraph(...
                            outputFileName,...
                            '5.2in', '3.7in');
                        append(docPart,para);
                    catch
                    end
                end
            end
        end
    end
end
end
end
end

```

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 filename 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”

Append Code to a Test Report

This example shows how to use a customization class to print integrated code in a test results report. If you test models that include handwritten code, you can print the code to a report to be reviewed with the test results.

The cruise control model integrates handwritten C code using an S-Function builder block. The C code is a utility function that disregards simultaneous pressing of two buttons: `AcceL/Res` and `Coast/Set`.

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

Example Files

Before running this example, set the filenames.

```
rptCustom = 'textAppendReport.m';  
resultsFile = 'DoublePressSfcnSimTestResults';  
filePath = fullfile(tempdir, 'textAppendedReport.zip');
```

Report Customization Class

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

```
open(rptCustom)
```

Load the Results and Create the Report

1. Load the test results file.

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

2. Create the test report using the customization.

```
sltest.testmanager.report(result, filePath, 'CustomReportClass', 'textAppendReport', ...  
    'IncludeTestResults', 0)
```

3. The report appends the S-Function wrapper code:

S-Function Wrapper

```

/*
 * Include Files
 *
 */
#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 Test Results Reports” on page 7-21.

```

sltest.testmanager.clearResults;
sltest.testmanager.close;

```

Results Sections

In this section...

“Summary” on page 7-28
 “Test Requirements” on page 7-28
 “Iteration Settings” on page 7-28
 “Errors” on page 7-28
 “Logs” on page 7-28
 “Description” on page 7-28
 “Parameter Overrides” on page 7-28
 “Coverage Results” on page 7-29
 “Aggregated Coverage Results” on page 7-29
 “Scope coverage results to linked requirements” on page 7-29
 “Add Tests for Missing Coverage” on page 7-29
 “Applied Coverage Filters” on page 7-29

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 x

▼ SUMMARY

Name	Baseline Test Case
Outcome	1
Start Time	01/05/2016 21:38:14
End Time	01/05/2016 21:38:18
Type	Baseline Test
Test File Location	C:\MATLAB\Test File.mldatx
Test Case Definition	
Rerun Test Case	
▶ Simulation Metadata	

▶ TEST REQUIREMENTS

▶ ITERATION SETTINGS

▶ ERRORS

▶ LOGS

▶ DESCRIPTION

▶ COVERAGE RESULTS

Summary

For a selected test case, 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.

For a selected Results item, the **Summary** section includes information for the Result Set, which applies to all of its child test suites and test cases.

Test Requirements

A list of test requirements linked to the test case. See “Requirements” on page 6-160 for more information on linking requirements to test cases.

Iteration Settings

If you are using iterations to run test cases, then this section appears in the results. For more information about test iterations, see “Test Iterations” on page 6-125.

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. To view simulation logs in the command window, enter this command at the command line before running the test:

```
sltest.testmanager.setpref('ShowSimulationLogs',...  
    'IncludeOnCommandPrompt',true)
```

Logs

This section displays simulation warnings captured from the Simulink Diagnostic Viewer. To view simulation logs in the command window, enter this command at the command line before running the test:

```
sltest.testmanager.setpref('ShowSimulationLogs',...  
    'IncludeOnCommandPrompt',true)
```

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 for the selected test case results appear in this section. Coverage results are aggregated at the test file level. For more information about coverage, see “Collect Coverage in Tests” on page 6-135.

Aggregated Coverage Results

At the Results level, lists the model analyzed for test coverage and includes a link to generate a coverage report. This section also reports the complexity level and the decision and execution percentages.

Scope coverage results to linked requirements

Controls whether coverage results include all executed items or only executed items that are explicitly linked to requirements. If not selected, coverage results include all executed items. If selected, displays coverage results only for tests explicitly linked to requirements.

Add Tests for Missing Coverage

Generate tests for missing coverage using Simulink Design Verifier. To add an iteration to an existing test case, select the test case name in Test Case. To create a new test case, select <Create a new test case> and specify the Test Type and Test Filename. See “Increase Test Coverage for a Model” on page 6-147 and “Increase Coverage by Generating Test Inputs” on page 6-174.

Applied Coverage Filters

At the Results level, lists the filter files applied to the coverage results shown in the Aggregated Coverage Results section.

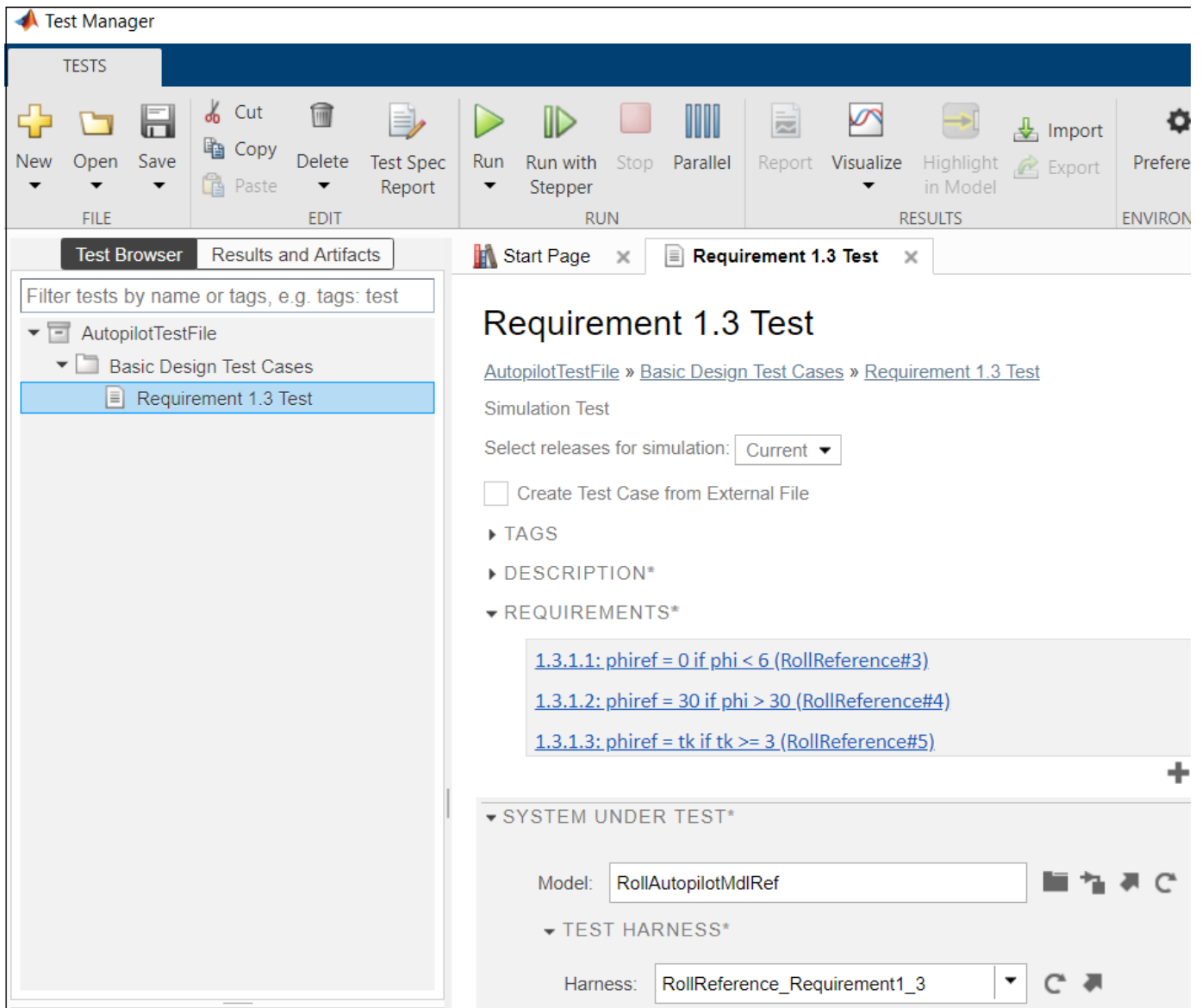
Generate Test Specification Reports

Test specification reports are reports of the test settings and parameters used for test cases, test suites, or test files. Common uses for these reports are capturing information for test procedure design reviews and archiving test information. You can create the report before or after running a test. In addition to using the Test Manager to create the report, you can create the report programmatically. See the `sltest.testmanager.TestSpecReport` reference page for examples.

For a test specification report, all of the items you select must be of the same type, either test files, test suites, or test cases. If you select a mixture of test files, suites, and cases, the **Test Spec Report** button and the context menu **Create Report** option are dimmed. If you select a test file, the report includes all of its test suites and test cases. If you select a test suite, the report includes all of its test cases.

This example uses an existing test file (`AutopilotTestFile.mldatx`), which was created for the `RollAutopilotMdlRef.slx` model and its `RollReference_Requirement1_3` test harness.

- 1 Set your current working folder to a writable folder.
- 2 To open the Test Manager, enter `sltestmgr` on the MATLAB command line.
- 3 Click **Open**.
- 4 In the Open File dialog box, open the `matlab/examples/simulink/main` folder and select `AutopilotTestFile.mldatx`.
- 5 Highlight the Requirement 1.3 Test test case and click **Test Spec Report**.



- 6 In the Create a Test Specification Report dialog box, specify the **Title** as RollAutopilot Model Test Specification Report and the **Author** as John Smith.

Create a Test Specification Report ? X

Title Page Information

Title: RollAutopilot Model Test Specification Report

Author: John Smith

Include in Report

<input checked="" type="checkbox"/> Test Details	<input checked="" type="checkbox"/> Iterations
<input checked="" type="checkbox"/> Logged Signals	<input checked="" type="checkbox"/> External Inputs
<input checked="" type="checkbox"/> Callback Scripts	<input checked="" type="checkbox"/> Parameter Overrides
<input checked="" type="checkbox"/> Coverage Settings	<input checked="" type="checkbox"/> Logical and Temporal Assessments
<input checked="" type="checkbox"/> System Under Test	<input checked="" type="checkbox"/> Configuration Settings
<input checked="" type="checkbox"/> Custom Criteria	

Output Options

File Format: PDF

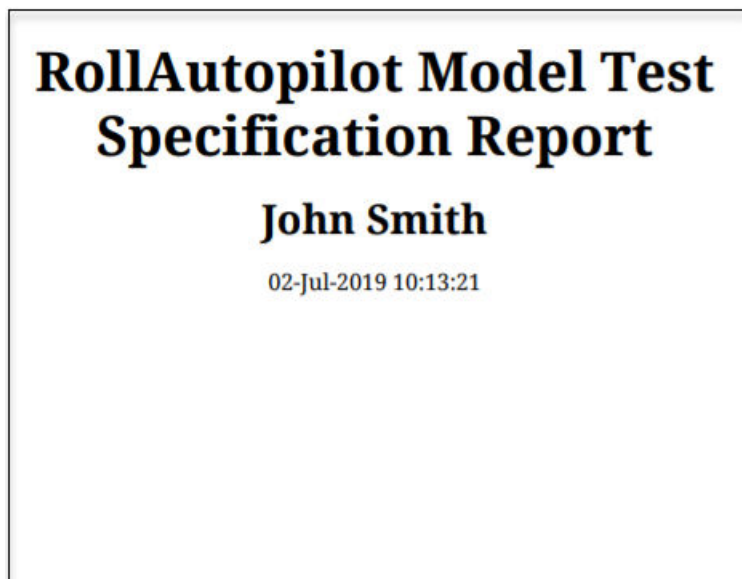
File Name: mynewReport.pdf

Customization Templates

Test Case Reporter: Select TestCaseReporter Template (optional)

Create Cancel

- 7 Leave all the report sections selected by default in the **Include in Report** section.
- 8 Leave PDF as the default output format.
- 9 Specify the filename for the saved report as mynewReport.pdf in a writable folder. If your current working folder is not writable, use a full pathname to a writable folder.
- 10 Leave the Test Case Reporter field blank because this report uses the default test case template.
- 11 Click **Create** to generate the report and open it automatically. These images from the report show the title page, images of the model and harness, and test inputs and assessment information.



1. Requirement 1.3 Test

System Under Test

Model Name: 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®.

RollAutopilotMdlRef.slx
Copyright 2018 The MathWorks, Inc.

Harness Name: RollReference_Requirement1_3

Signal spec. and routing

Test Sequence Data

RollReference_Requirement1_3Requirement1_3Test1/Test Assessment

(Library Link: RollRefAssessLib/Test Assessment)

Symbols

Input

3

1. Requirement 1.3 Test

Port	Name	Class	Data Type*	Size*
1	PhiRef	Data	Inherit: Same as Simulink	-1
2	Phi	Data	Inherit: Same as Simulink	-1
3	APEng	Data	Inherit: Same as Simulink	-1
4	TurnKnob	Data	Inherit: Same as Simulink	-1

***Note:** The model was not compiled during report generation. As a result, the preceding Symbols section does not include information requiring model compilation. See the help for the Simulink Report Generator's Simulink Test Sequence block component for more information.

Step Hierarchy

- [GlobalAssessment](#)
 - [NormalRange](#)
 - [BelowLowLimit](#)
 - [ExceedPosLimit](#)
 - [ExceedNegLimit](#)
 - [TurnKnobAssessments](#)
 - [BelowLowTKLimit](#)
 - [NormalTKLimit](#)
 - [Else2](#)
 - [Else1](#)

GlobalAssessment

NormalRange

Action

```
verify(PhiRef == Phi, 'Simulink:verify_normal', 'PhiRef must equal Phi for normal operation')
```

When Condition

```
(abs(Phi) >= 6 && Phi <= 30) && APEng == true && TurnKnob == false
```

BelowLowLimit

Action

```
verify(PhiRef == 0, 'Simulink:verify_low', 'PhiRef must equal 0 for low angle operation')
```

When Condition

```
abs(Phi) < 6 && APEng == true && TurnKnob == false
```

ExceedPosLimit

Action

```
verify(PhiRef == 30, 'Simulink:verify_high_pos', 'PhiRef must equal 30 for high pos angle operation')
```

When Condition

4

See Also

sltest.testmanager.TestSpecReport

More About

- “Customize Test Specification Reports” on page 7-34

Customize Test Specification Reports

In this section...

“Remove Content or Change Report Formatting and Section Ordering” on page 7-34

“Add Content to a Test Specification Report” on page 7-37

You can customize test specification reports by creating a new test case or test suite template or reporter. The test suite templates and reporter are used for both test suites and test files.

To remove content or change the formatting or section ordering of a report, create a new template. To add new content, create a new reporter and specify new holes to hold that content.

Note To customize a report, you must have a Simulink Report Generator license.

Remove Content or Change Report Formatting and Section Ordering

To change the formatting or section ordering of a Test Specification Report or to remove content, use the `createTemplate` method of the `TestCaseReporter` or `TestSuiteReporter`. The `createTemplate` method applies to one output type at a time (PDF, HTML, or Word).

This example creates a new test case reporter template for PDF output. The process is the same for creating templates for other output types and for creating test suite reporter templates.

- 1 Create a copy of the default `TestCaseReporter` PDF template in the current working folder. This folder must be writable. In this case, the folder name is `myCustomTCTemplate`.

```
sltest.testmanager.TestCaseReporter.createTemplate(...
    'myCustomTCTemplate', 'pdf');
```

For pdf and zip (zip is used for HTML) output, `createTemplate` creates a zipped file. docx (Word) output it creates a `.dotx` template file.

- 2 To access the separate template files, unzip the PDF template file.

```
unzipTemplate('myCustomTCTemplate.pdf');
```

Unzipping the file creates a `docpart_templates.html` file and a `/stylesheets/root.css` file in the new `myCustomTCTemplate` folder. PDF and HTML reports use HTML template files.

- 3 Open and edit the `docpart_templates.html` file using a text editor. This file lists the content holes in the order in which the content appears in the report. In this file, you can reorder the report sections and delete template holes. A portion of the `docpart_templates.html` file is shown.

```
docpart_templates.html x
<html>
  <head>
    <meta charset="utf-8" />
    <title>Document Part Templates</title>
    <link rel="StyleSheet" href="./stylesheets/root.css" type="text/css" />
  </head>

  <body>
    <dplibrary>
      <dptemplate name="TestCaseReporter">
        <!-- filled with test details table -->
        <hole id="TestDetails">TEST_DETAILS</hole>

        <!-- filled with System Under Test -->
        <hole id="SystemUnderTest">SYSTEM_UNDER_TEST</hole>

        <!-- filled with Parameter Overrides -->
        <hole id="ParameterOverrides">PARAMETER_OVERRIDES</hole>

        <!-- filled with Callback scripts -->
        <hole id="Callbacks">CALLBACKS</hole>

        <!-- filled with External inputs table -->
        <hole id="ExternalInputs">EXTERNAL_INPUTS</hole>

        <!-- filled with Simulation Outputs -->
        <hole id="SimulationOutputs">SIMULATION_OUTPUTS</hole>

        <!-- filled with Config Settings Overrides -->
        <hole id="ConfigSettingsOverrides">CONFIG_SETTINGS_OVERRIDES</hole>
      </dptemplate>
    </dplibrary>
  </body>
</html>
```

- 4 In the stylesheets folder, open and edit the root.css file using a text editor. In this file, you can change the table borders, font size, text color, and other styles. For example, to set a font size to 14 pixels, use font-size: 14px;

```

root.css * X
table.TestCase_TestDetailsTable, table.TestCase_CoverageSettingsTable, table.BaselineC
table.ExternalInputsTable, table.ParameterSetsTable, table.TargetSettingsTable, table.
    table-layout: fixed;
    width: 100%;
    border-collapse: collapse;
    border-style:solid;
    margin-top: 15px;
    margin-bottom: 25px;
}

table.TestCase_TestDetailsTable td, table.TestCase_CoverageSettingsTable td, table.Bas
table.ExternalInputsTable td, table.ParameterSetsTable td, table.TargetSettingsTable t
table.SymbolsMetadataTable td {
    border-collapse: collapse;
    border-style:solid;
    white-space: normal;
    text-align: start;
    font-size: 16px;
    font-style: normal;
    font-variant: normal;
    font-weight: normal;
    line-height: normal;
    padding: 10px;
    hyphenation: hyphen;
}

table.BaselineCriteriaTable th, table.EquivalenceCriteriaTable th, table.IterationsTab
table.AssessmentsTable th, table.SymbolsTable th {
    padding: 5px;|
    border-style:solid;

```

To learn more about modifying report styles, see “Modify Styles in PDF Templates” (MATLAB Report Generator). For information on Word or HTML styles, see “Modify Styles in Microsoft Word Templates” (MATLAB Report Generator) or “Modify Styles in HTML Templates” (MATLAB Report Generator), respectively.

- 5 Zip the files into to the myCustomTCTemplate.pdf file.

```
zipTemplate('myCustomTCTemplate.pdf');
```

- 6 Use the custom template for your test specification PDF report by using either of these processes.

- Use `sltestmgr` to open the Test Manager and click **Test Spec Report** to open the Create a Test Specification Report dialog box. Add myCustomTCTemplate.pdf to the **Test Case Reporter** field.
- Specify the myCustomTCTemplate.pdf filename in the TestCaseReporterTemplate property of the `sltest.testmanager.TestSpecReport`.

```

sltest.testmanager.TestSpecReport(test_cases,'testReport.pdf',...
    'Author','John Smith','Title','Autopilot Test Spec Report',...
    'LaunchReport',true,...
    'TestCaseReporterTemplate','MyCustomTCTemplate.pdf')

```

Add Content to a Test Specification Report

To add new content to a report or override how content is added, create a subclass of the `sltest.testmanager.TestCaseReporter` or `sltest.testmanager.TestSuiteReporter` class. Then add properties and methods for the new content in its class definition file. Add holes to hold that content in the test suite or test case templates.

This example describes creating a new test case reporter. Use the same process to create a new test suite reporter.

- 1 To create a new test case reporter class, use the `customizeReporter` method of the `TestCaseCreate` reporter class. This command creates a new class folder in the current working folder. This new reporter inherits from the `TestCaseReporter` class.

```
customTCRptr = ...
    sltest.testmanager.TestCaseReporter.customizeReporter...
    ('@myTCReporter');
```

See “Subclass Reporter Definitions” (MATLAB Report Generator).

The `@myTCReporter` folder has a `myTCReporter.m` class definition file and a `resources` folder. The `resources` folder contains a `templates` folder, which contains folders and files for the report output types:

- `pdf` folder
 - `default.pdf.tx` — Zipped PDF template file. Unzip this file using `unzipTemplate` and then open the template file using a text editor. After editing, use `zipTemplate`.
- `docx` folder
 - `default.dotx` — Word template file. Open this template file by right-clicking and selecting **Open** from the context menu. If you click the filename to open it, the Word file associated with the template opens instead of the template file. See “Open Template Files” (MATLAB Report Generator).
- `html` folder
 - `default.html` — Single-file HTML template. Open this file using a text editor.
 - `default.html.tx` — Zipped HTML template file. Unzip this file using `unzipTemplate` and then open the template file using a text editor. After editing, use `zipTemplate`.

For information on templates, see “Templates” (MATLAB Report Generator).

- 2 In the `@myTCReporter` folder, open the class definition file `myTCReporter.m` in a text editor.

```

myTCReporter.m ×
classdef myTCReporter < sltest.testmanager.TestCaseReporter

    properties
    end

    methods
        function obj = myTCReporter(varargin)
            obj = obj@sltest.testmanager.TestCaseReporter(varargin{:});
        end
    end

    methods (Hidden)
        function templatePath = getDefaultTemplatePath(~, rpt)
            path = myTCReporter.getClassFolder();
            templatePath = ...
                mlreportgen.report.ReportForm.getFormTemplatePath(...
                    path, rpt.Type);
        end
    end

    end

    methods (Static)
        function path = getClassFolder()
            [path] = fileparts(mfilename('fullpath'));
        end

        function createTemplate(templatePath, type)
            path = myTCReporter.getClassFolder();
            mlreportgen.report.ReportForm.createFormTemplate(...
                templatePath, type, path);
        end

        function customizeReporter(toClasspath)
            mlreportgen.report.ReportForm.customizeClass(...
                toClasspath, "myTCReporter");
        end
    end

end
end

```

- 3 To add new content, add a property and define a `get<property>` method in the customized class definition file. Then add the hole to the output type templates.

For example, for a new section named References, add a References property and define a `getReferences` method in the `myTCReporter.m` class definition file.


```

myTCReporter.m ×
classdef myTCReporter < sltest.testmanager.TestCaseReporter

    properties
        References;
    end

    methods
        function obj = myTCReporter(varargin)
            obj = obj@sltest.testmanager.TestCaseReporter(varargin{:});
        end

        function content = getReferences(h,rpt)
            import mlreportgen.dom.*;
            heading = mlreportgen.dom.Heading4('References');
            heading.StyleName = 'ReferencesHeading';
            ol = OrderedList;
            listItem = ExternalLink(
                'https://www.mathworks.com/products/simulink-test.html',...
                'Simulink Test');
            append(ol,listItem);
            listItem = ExternalLink(
                'https://www.mathworks.com/products/SL_reportgenerator.html',...
                'Simulink Report Generator');
            append(ol,listItem);
            content = [{heading},{ol}];
        end
    end
end

```

Then, add `<hole id="References">REFERENCES</hole>` to the template files in the desired location to include the hole content in the generated report for each output type. See “Add Holes in HTML and PDF Templates” (MATLAB Report Generator) and “Add Holes in Microsoft Word Templates” (MATLAB Report Generator).

- 4 To override an existing method, add a function in the customized class definition file that defines the `get` method for the hole.

For example, for the `TestDetails` hole in the `TestCaseReporter`, create a method called `getTestDetails` in the customized `TestCaseReporter` class definition file. You do not need to add a property or hole because they are already specified in the `TestCaseReporter` class from which the customized reporter inherits.

- 5 To generate a report using the custom reporter, use Simulink Report Generator commands (see “Define New Reporters” (MATLAB Report Generator)).

These sample commands create a PDF report for a test case. It uses the `myTCReporter` reporter, which takes a test case array (`test_cases`) as the input object. Then, add the test case reporter object to the report and use `rptview` to display it. The report is saved in the `myCustomTestSpecRpt.pdf` file.

```

myrpt = slreportgen.report.Report('myCustomTestSpecRpt.pdf');
testCaseRptr = myTCReporter('Object',test_cases);

```

```
add(myrpt, testcaseRptr);  
close(myrpt);  
rptview(myrpt);
```

See Also

More About

- “Customize Test Results Reports” on page 7-21
- “Templates” (MATLAB Report Generator)
- “Open Template Files” (MATLAB Report Generator)
- “Subclass Reporter Definitions” (MATLAB Report Generator)
- “Define New Reporters” (MATLAB Report Generator)

Debugging Equivalence Test Failures Using Model Slicer

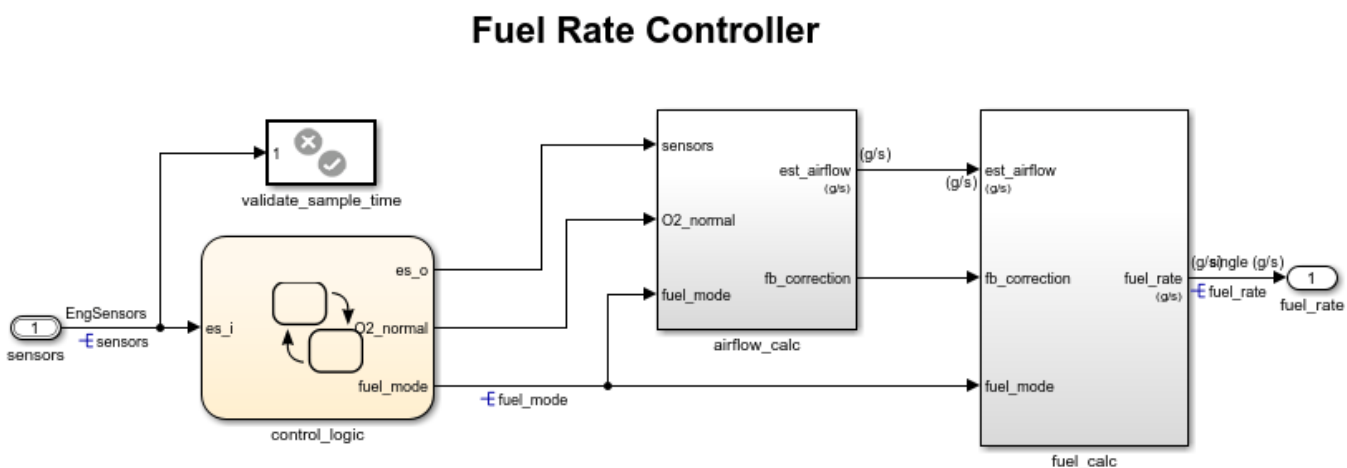
This example shows how to debug Simulink Test equivalence failures by highlighting functional dependencies using Model Slicer. For more information, see “Highlight Functional Dependencies” (Simulink Check). The example files include a predefined equivalence test to use with the `sldemo_fuelsys_dd_controller` model. The sample test case compares the model simulation in normal and software-in-the-loop (SIL) mode. The model includes a numerical discrepancy. In this example, you use Model Slicer to trace the faults and identify the discrepancy. For more information about debugging test failures, see the **Capabilities and Limitations** section of “Debugging Test Failures Using Model Slicer” on page 7-7.

Setting Up the Artifacts

Run the test case and view the results.

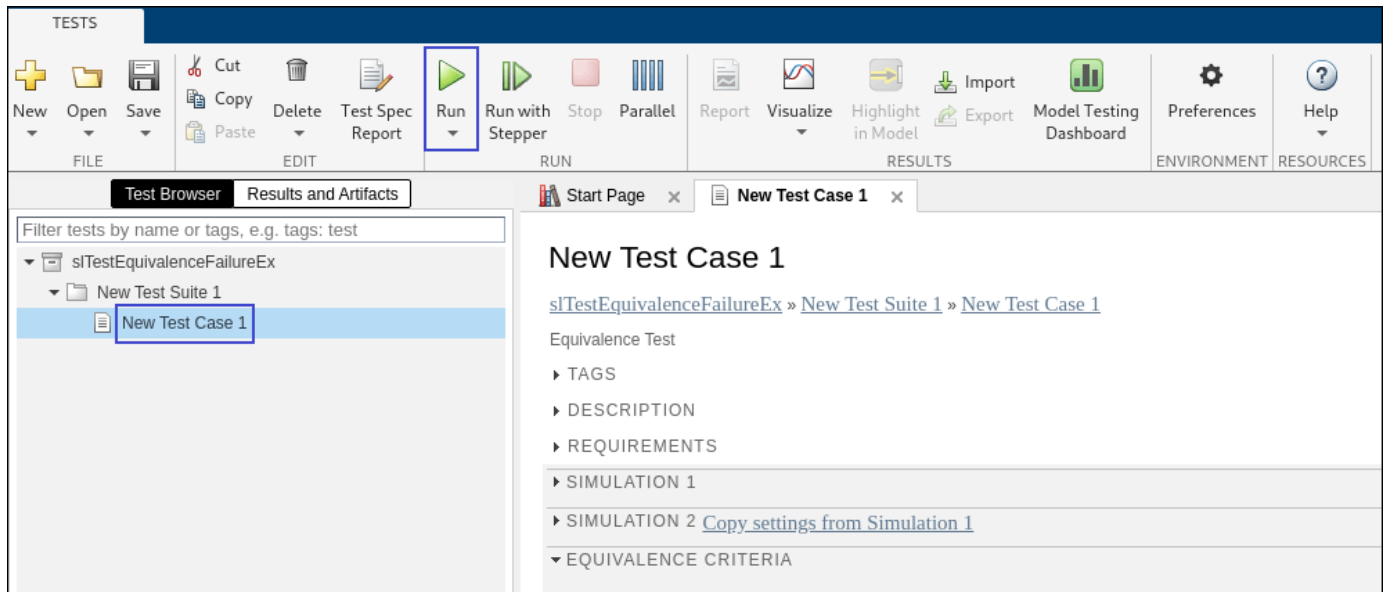
1. Open the `sldemo_fuelsys_dd_controller` model:

```
open_system('sldemo_fuelsys_dd_controller');
```



Copyright 1990-2022 The MathWorks, Inc.

2. Log the output by the signals coming out of `airflow_calc` and `fuel_calc` subsystems and the Stateflow chart `control_logic` to generate a visualization after the analysis. To log signals, click the signal and, in the action bar, click Log Selected Signal.
3. In the **Apps** tab, in the **Model Verification, Validation, and Test** section, click **Simulink Test** to open the Simulink Test toolstrip.
4. In the **Tests** tab, click **Simulink Test Manager** to open the Test Manager.
5. To open the test file, click **Open** and select `slTestEquivalenceFailureEx` from the example folder.
6. After the test file loads, select **New Test Case 1** in the **Test Browser** pane.
7. Click **Run**.



8. The new test results appear in the **Results and Artifacts** pane. Right-click the result and select **Expand All Under** to see the **Equivalence Criteria Result** and **Verify Statements 1** section.

NAME	STATUS
▼ Results: 2021-Nov-16 16:44:40	1 ✖
▼ New Test Case 1	✖
▼ Equivalence Criteria Result	✖
○ airflow_calc:1	✔
● airflow_calc:2	✖
○ control_logic:1.throttle	✔
○ control_logic:1.speed	✔
○ control_logic:1.ego	✔
○ control_logic:1.map	✔
○ control_logic:2	✔
○ fuel_mode	✔
○ fuel_rate	✔
▶ Verify Statements 1	✔
▶ Sim Output 1(sldemo_fuelsys_...	
▶ Sim Output 2(sldemo_fuelsys_...	

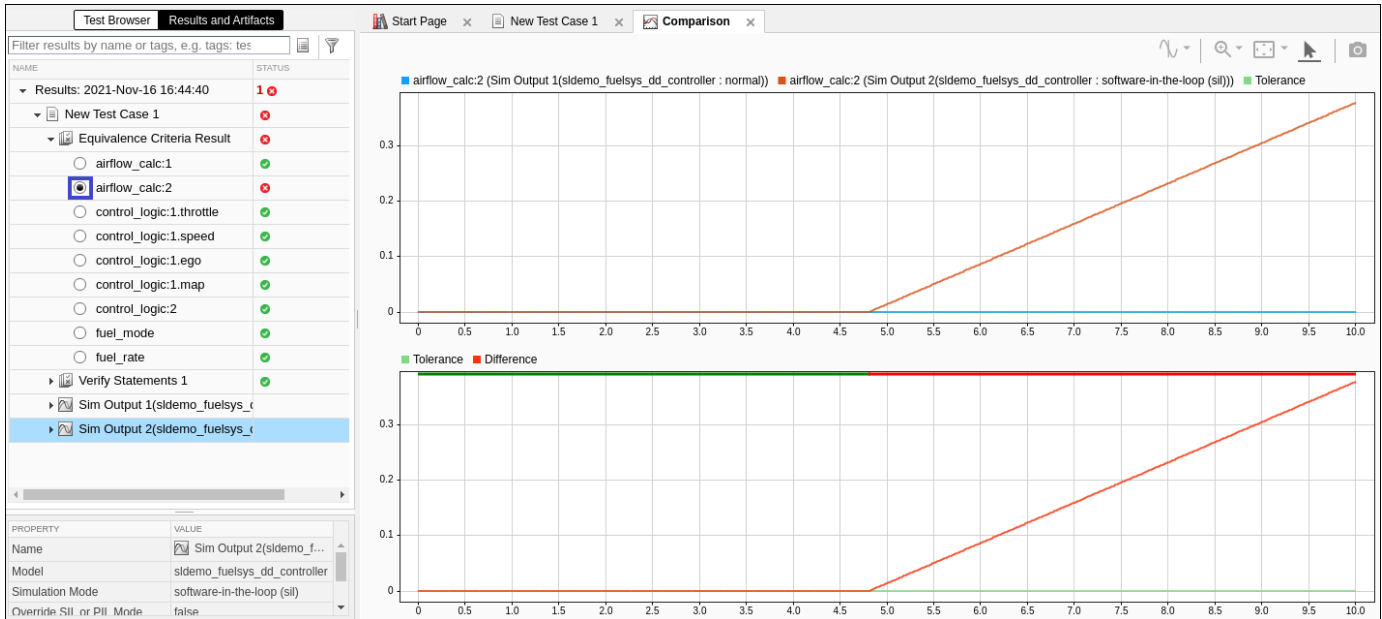
Observe that only one signal failed: `airflow_calc`. Consider changing the **Device details** from 32 bits to 64 bits from **Hardware and Implementation** pane after clicking on **Model Settings** from Simulation tab, if you are not able to see the failure.

Entering Debug Session

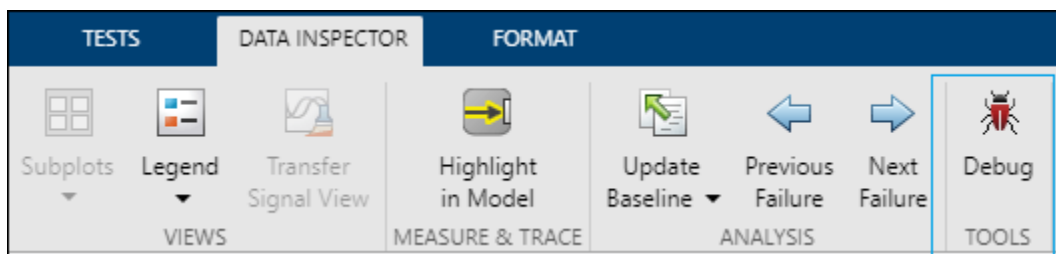
Set up the Model Slicer to debug the failed `airflow_calc` signal.

1. To compare the `airflow_calc` signal in different simulation modes, select the radio button next to the `airflow_calc` signal. Another way to select a failed signal is from the **Signal to Debug** dropdown list in the toolbar.

In the plot area, compare the outputs across simulations.

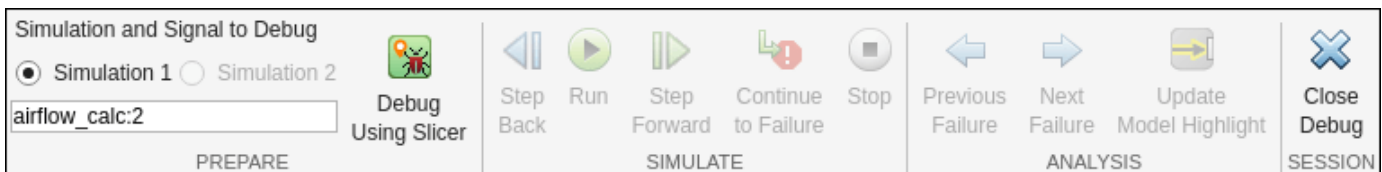


2. In the **Tools** section, click **Debug**. Note that you must plot a failed equivalence or verify signal to enable the **Debug** button.



The **Debug** tab opens and hides. The debug tab hides multiple Test Manager options.

3. To set up the Model Slicer, click **Debug Using Slicer**.



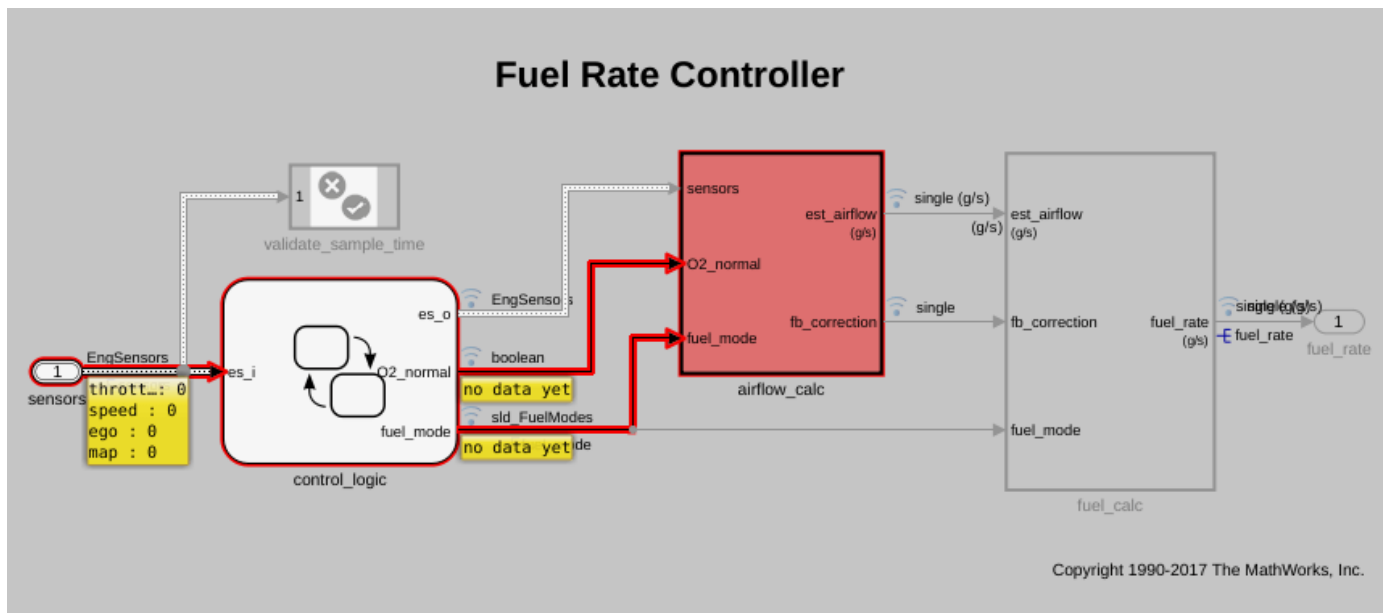
Model Slicer prepares the debugging session by:

- 1 Rerunning the test case and creating new debugging results. This action ensures that the failure still exists in the current state of the test model.
- 2 Launching the Model Slicer on the test model.
- 3 Automatically plotting the selected failed signal in the debugging results and setting the failed signal as the starting time point.
- 4 Pausing the simulation at the model start time to continue debugging.

Debugging Using Model Slicer

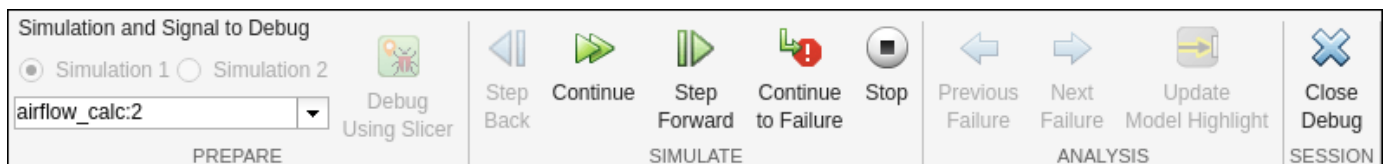
Identify the reason for the failure by using the debugging features of the Model Slicer.

1. Use the **Step Back** or **Step Forward** buttons to move one step back or forward in simulation time. The left data cursor moves to the current simulation time. Observe the changes in the data dependencies by noting the changed model highlighting and port value labels for the active signals at every time stamp.



You can also use **Run**, **Continue**, or **Stop** buttons to run, complete, or stop the current simulation, respectively.

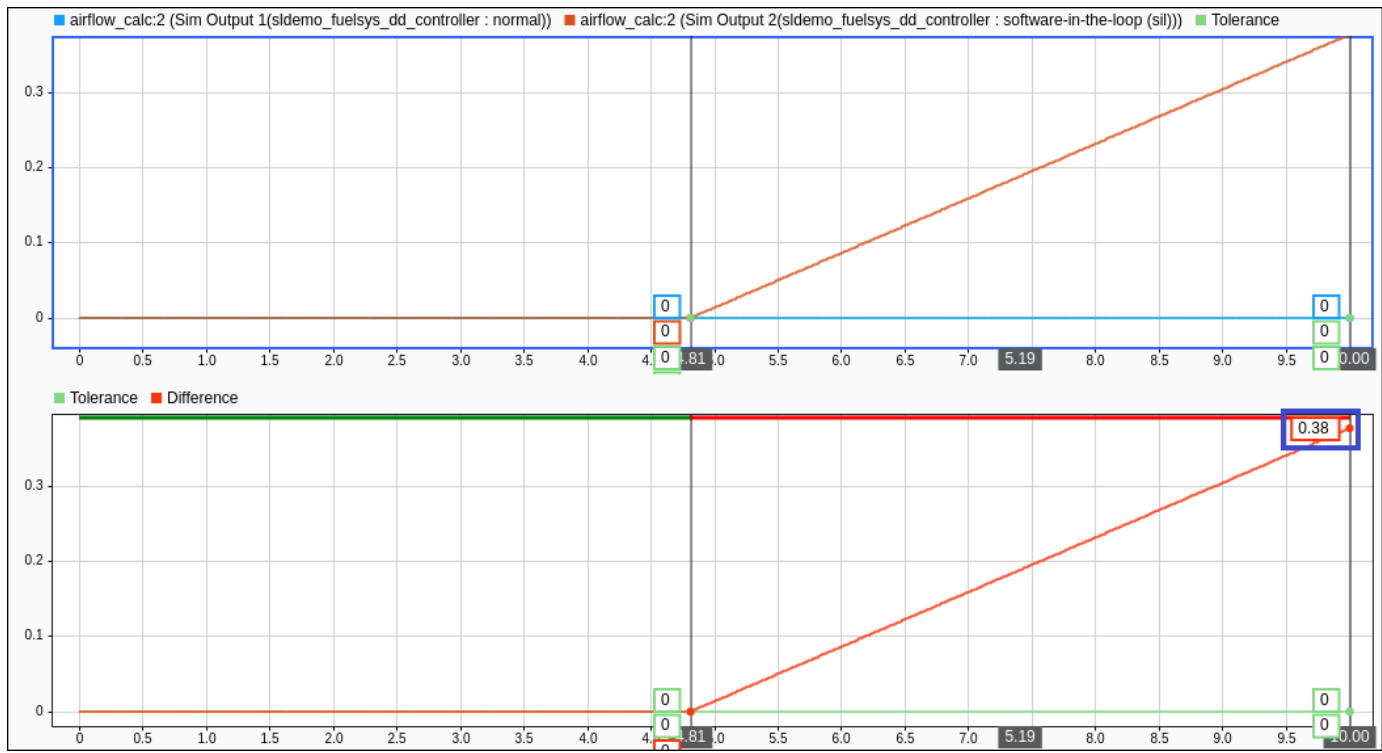
2. In the Test Manager, click **Continue to Failure** to continue the model simulation to the beginning of the next failure region. The data cursors show the bounds of the failure region.



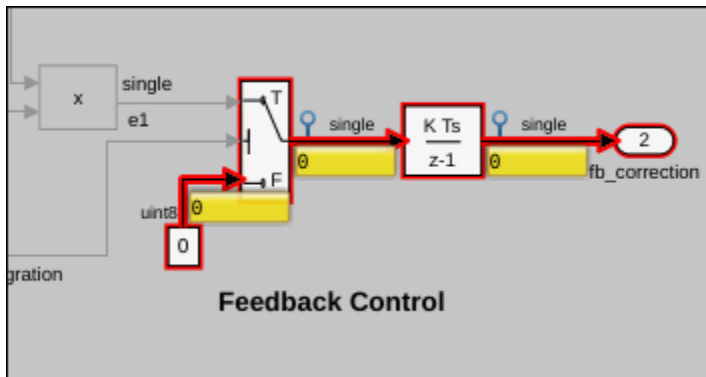
3. Observe these changes at the failure:

- Simulation pauses at $T = 4.81$.

- The data cursors update accordingly.

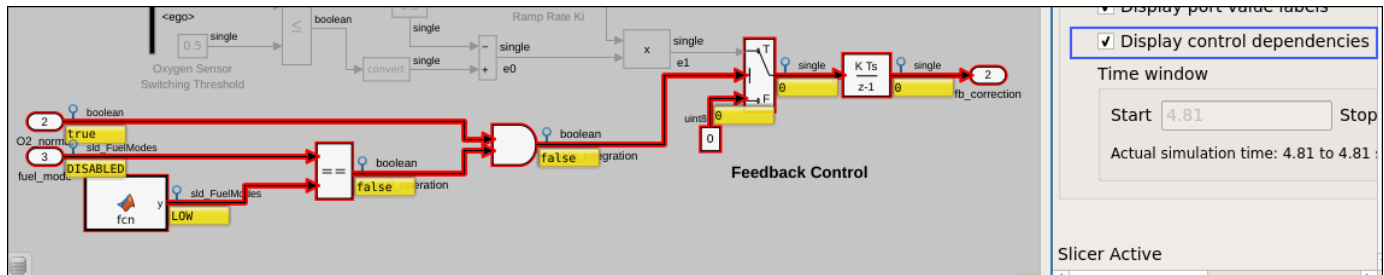


From the Model Slicer highlighting, you can observe the branch of the model that causes the error.



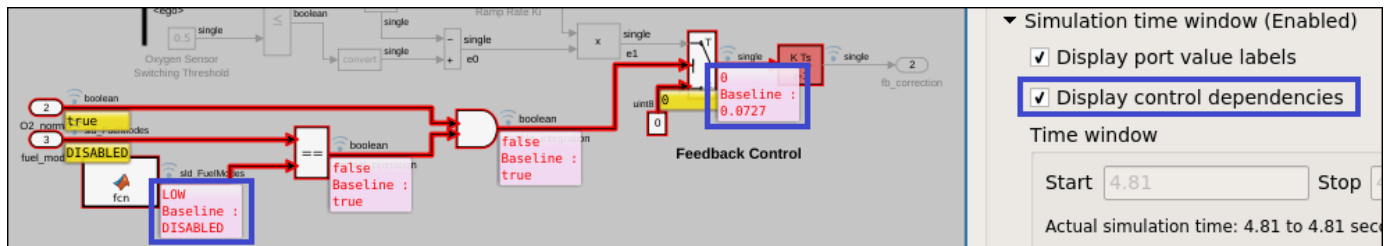
Notice that the constant, 0 passes through the `sldemo_fuelsys_dd_controller/airflow_calc/hold` integrator Switch block. To determine why the control port evaluates to false, highlight the control dependencies in the model.

4. Enable **Display Control Dependencies** from the **Simulation Time Window** section in the Model Slicer Dialog pane. Observe the chain of blocks that the Model Slicer highlights as the probable cause of the discrepancy. To further visualize the numerical differences between the simulation modes, end the debugging session, log the signals in the active chain, and debug the model again.



Refining the Debugging Results

1. Exit the debugging session by clicking **Session > Close Debug**.
2. Open the model and log all signals in the observed path before.
3. Save the model.
4. Repeat steps 1 to 3 from the Debug Using Model Slicer section.



Observe that the Enumerated Constant value is being set in the MATLAB function block based on the simulation mode.

Incorporating the Fix

1. Exit the debugging session by clicking **SESSION > Close Debug**.
2. Open the model and update `sldemo_fuelsys_dd_controller/airflow_calc/MATLAB` function to return the same value irrespective of the simulation mode.
3. Save the model.
4. Run the test case and view the results.

Observe that the test results test results pass.

See Also

- “Refine, Test, and Debug a Subsystem” on page 2-24

Real-Time Testing

- “Test Models in Real Time” on page 8-2
- “Reuse Desktop Test Cases for Real-Time Testing” on page 8-9
- “Install and Set Up the Simulink Test Support Package for ASAM XIL Standard” on page 8-13
- “Real-Time Testing with the Simulink Test Support Package for ASAM XIL Standard” on page 8-15
- “Create Tests Using the Simulink Test Support Package for ASAM XIL Standard” on page 8-19

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 8-3
- “Add Test Cases for Real-Time Testing” on page 8-4
- “Assess Real-Time Execution Using verify Statements” on page 8-8

You can test your system in environments, such as Simulink Real-Time™, 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.

Note The information in this topic does not apply to testing on third-party test benches using the ASAM XIL standard. For information and examples about performing tests using ASAM XIL, see “Install and Set Up the Simulink Test Support Package for ASAM XIL Standard” on page 8-13 and “Real-Time Testing with the Simulink Test Support Package for ASAM XIL Standard” on page 8-15.

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) using Simulink Real-Time, which tests a system that has passed several stages of verification, typically SIL and PIL simulations. You can use Simulink Test on a Windows® or Linux® computer to run real-time test cases using Simulink Real-Time.

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.
- 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.
- You cannot log states or, on some platforms, output ports.
- Your target computer must have a file system to use `verify` statements and test case logging.
- Your target computer must be running Simulink Real-Time.

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 3-75.
- Update your requirements and add corresponding test cases.

Set up the Target Computer

Real-time testing requires a target computer or external hardware test bench. Simulink Test supports target computers running Simulink Real-Time. For more information, see:

- “Target Computer Settings” (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, on the **Apps** tab, under Code Generation, click **Simulink Coder**. In the **C Code** tab, verify that the system target file is `slrealtime.tlc`. If the button in the **Output** section is **Custom Target**, click that button and verify that the **Custom target** is `slrealtime.tlc`. If it isn't, select **Select system target file** and select `slrealtime.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 `sltestProjectorController` model and sets the system target file parameter to `slrealtime.tlc`.

```
openExample('sltestProjectorController');
set_param('sltestProjectorController',...
    'SystemTargetFile','slrealtime.tlc');
```

Data Import/Export Format

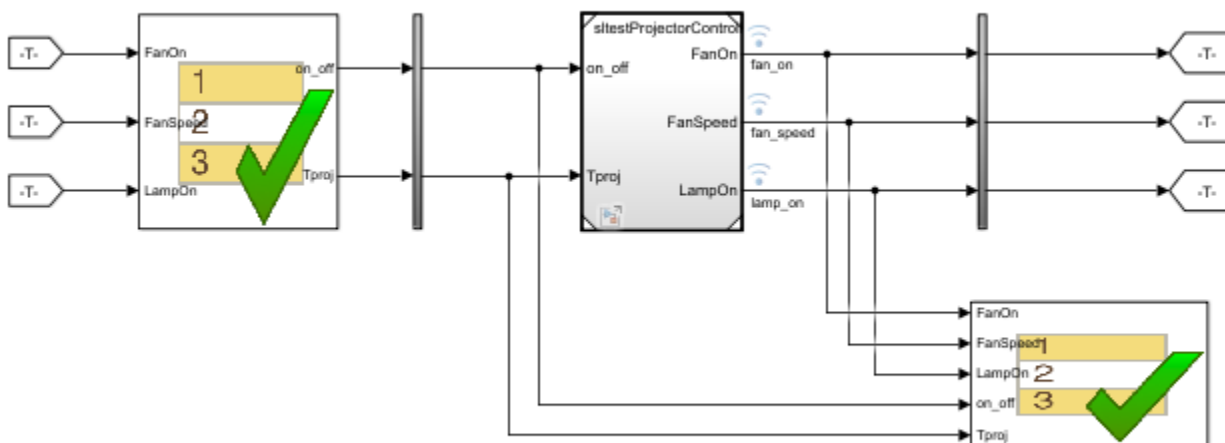
Models must use a data format other than `dataset`. To set the data format:

- 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.
- 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.



Add Test Cases for Real-Time Testing

Use the Test Manager to create real-time test cases.

- 1 In the Simulink toolstrip, on the **Apps** tab under Model Verification, Validation, and Test, select **Simulink Test**.
- 2 Click **Simulink Test Manager**.
- 3 In the Test Manager, select **New > Real-Time Test**.

Test Type

You can select a baseline, equivalence, or simulation real-time test. For simulation test types, `verify` statements serve as pass/fail criteria in the test results. For equivalence and baseline test types, the equivalence or baseline criteria also serve as pass/fail criteria.

- **Baseline** — Compares the signal data returned from the target computer to the baseline in the test case. To compare a real-time execution result to a model simulation result, add the model baseline result to the real-time test case and apply optional tolerances to the signals.
- **Equivalence** — Compares signal data from a simulation and a real-time test, or two real-time tests. To run a real-time test on the target computer, then compare results to a model simulation:
 - Select **Simulation 1 on target**.
 - Clear **Simulation 2 on target**.

The test case displays two simulation sections, **Simulation 1** and **Simulation 2**.

Comparing two real-time tests is similar, except that you select both simulations on target. In the **Equivalence Criteria** section, you can capture logged signals from the simulation and apply tolerances for pass/fail analysis.

- **Simulation**: Assesses the test result using only `verify` statements and real-time execution. If no `verify` statements fail, and the real-time test executes, the test case passes.

Load Application From

Using this option, specify how you want to load the real-time application. The real-time application is built from your model or test harness. You can load the application from:

- **Model** — Choose `Model` if you are running the real-time test for the first time, or your model changed since the last real-time execution. `Model` typically takes the longest because it includes model build and download. `Model` loads the application from the model, builds the real-time application, downloads it to the target computer, and executes it on the target computer.
- **Target Application** — Choose `Target Application` to send the target application from the host to a target computer, and execute the application. `Target Application` can be useful if you want to load an already-built application on multiple targets.
- **Target Computer** — This option executes an application that is already loaded on the real-time target computer. You can update the parameters in the test case and execute using `Target Computer`.

This table summarizes which steps and callbacks execute for each option.

Test Case Execution Step (first to last)	Load Application From		
	Model	Target Application	Target Computer
Executes pre-load callback	Yes	Yes	Yes
Loads Simulink model	Yes	No	No
Executes post-load callback	Yes	No	No
Sets Signal Editor scenario	Yes	No	No

Test Case Execution Step (first to last)	Load Application From		
	Model	Target Application	Target Computer
Builds real-time application from model	Yes	No	No
Downloads real-time application to target computer	Yes	Yes	No
Sets runtime parameters	Yes	Yes	Yes
Runs Test Sequence scenarios	Yes	No	No
Executes pre-start real-time callback	Yes	Yes	Yes
Executes real-time application	Yes	Yes	Yes
Executes cleanup callback	Yes	Yes	Yes

Model

Select the model from which to generate the real-time application.

Test Harness

If you use a test harness to generate the real-time application, select the test harness.

Simulation Settings Overrides

For real-time tests, you can override the simulation stop time, which can be useful in debugging a real-time test failure. Consider a 60-second test that returns a `verify` statement failure at 15 seconds due to a bug in the model. After debugging your model, you execute the real-time test to verify the fix. You can override the stop time to terminate the execution at 20 seconds, which reduces the time it takes to verify the fix.

Callbacks

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

Sequence	Load application from: Model	Load application from: Target application	Load application from: Target computer
Executes first	Preload callback	Preload callback	Preload callback
	Post-load callback	—	—
	Pre-start real-time callback	Pre-start real-time callback	Pre-start real-time callback

Sequence	Load application from: Model	Load application from: Target application	Load application from: Target computer
Executes last	Cleanup callback	Cleanup callback	Cleanup callback

Iterations

You can execute iterations in real-time tests. Iterations are convenient for executing real-time tests that sweep through parameter values or Signal Editor scenarios. Results appear grouped by iteration. For more information on setting up iterations, see “Test Iterations” on page 6-125. 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 Editor scenarios — If your model or test harness uses a Signal Editor input, below the **IterationsTable Iterations** table, click **Auto Generate** and select **Signal Editor Scenario**. If you use a Signal Editor scenario, load the application from the model.
- Tabled iterations from Test Sequence scenarios — If your test harness uses Test Sequence block scenarios, you can create an iteration for each scenario in the Test Manager. Below the **Iterations** table, click **Auto Generate** and select **Test Sequence Scenario**. If you use Test Sequence scenarios, load the application from the model.
- Scripted iterations — Use scripts to iterate using model variables or parameters. For example, assume you are testing an oscillator system and use a Test Sequence block to create a square wave test signal using the parameter frequency.

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

In the test file, you can use real-time test scripted iterations to 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', ...
```

```

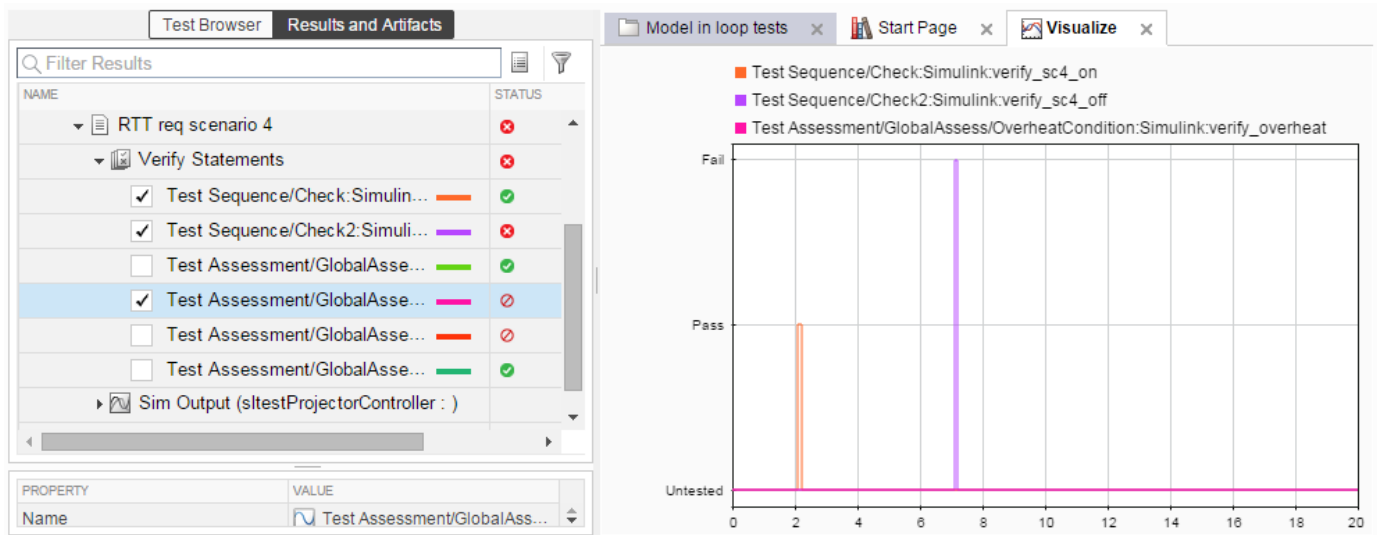
    'Test Sequence', 'Value', freq(i_iter));

    % 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 or, if you have a Stateflow license, in a Stateflow chart. See “Assess Model Simulation Using `verify` Statements” on page 3-18.



See Also

Related Examples

- “Use Simulink Real-Time UI to Create and Execute a 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 Requirements” (Simulink Real-Time).

Use External Data for Real-Time Tests

You can simplify test input data management by defining the input data in an external MAT or Excel file. Map the data to root inports in your model or test harness for desktop simulation. When you convert the desktop simulation test case into a real-time test, the test case uses the same inport mapping.

Using external data depends on how your test case loads the real-time application:

Load Real-Time Application from Model

If you are using external data for a real-time test, loading the real-time application from the model gives you the option of using an Excel file, MAT file, or CSV file. The external data is built into the application, and you can rerun the application from the target application or target computer.

In the **System Under Test** section, set the application to load from **Model**. In the **Inputs** section of the test case, click **Add**, and select an Excel file, MAT file, or CSV file. Map the data to your model inports. For more information on input mapping, see “Use External Excel or MAT-File Data in Test Cases” on page 6-72.

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.

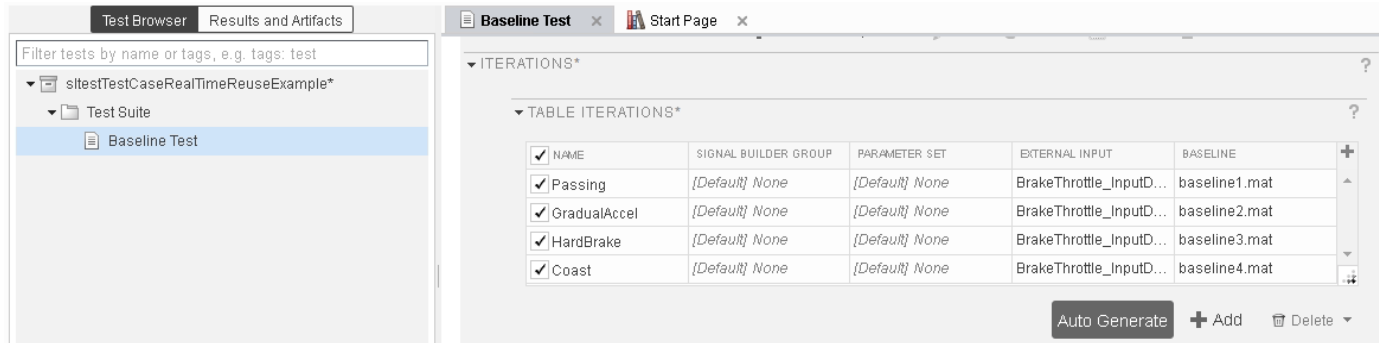
Reuse Desktop Test Case for Real-Time Testing

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. The test file, baseline data, and Excel input data file are provided. This example runs only on Windows systems.

Open the Test Manager and Test File

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. Baseline data associated with each scenario for the signals `vehicle speed` and `output torque`.

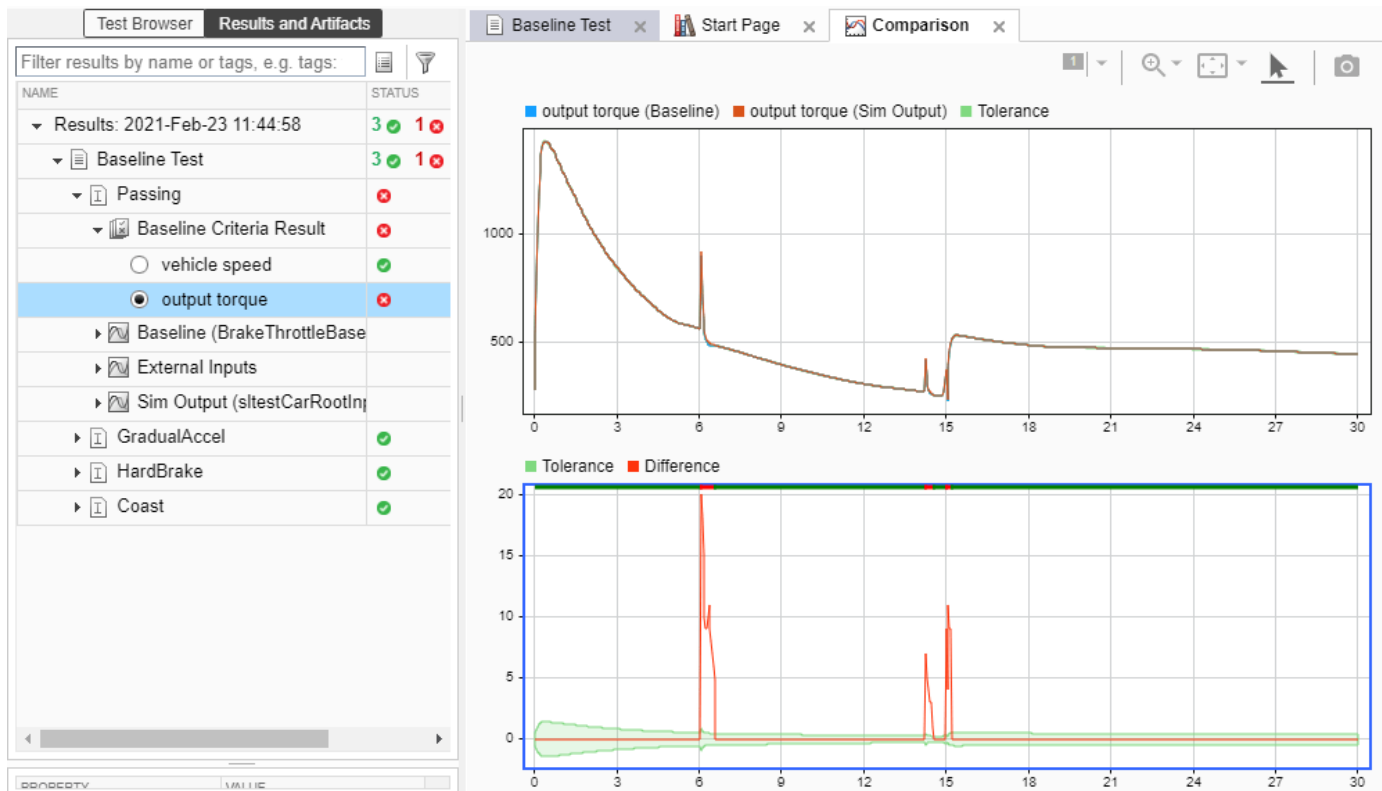
```
tf = sltest.testmanager.TestFile('sltestTestCaseRealTimeReuseExample.mldatx');
sltest.testmanager.load(tf.Name);
sltest.testmanager.view;
```



Run the Baseline Test and View Results

Click **Run** in the toolbar.

When the test finishes running, select **output torque** under **Baseline Criteria Result** to view the comparison. The Passing result fails due to transient signals that fall outside the relative tolerance.



Update the Baseline

Assume that the 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 > Replace Signal Segment in Baseline File** from the toolstrip, and confirm that you want to overwrite the data.
- 3 Repeat this process for the other two failure regions.

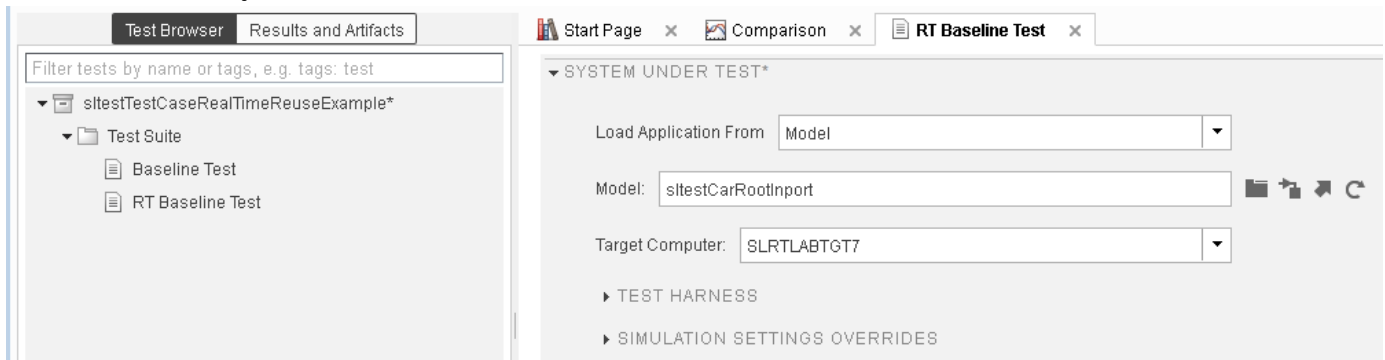
Convert Baseline Test to Real-Time Test

- 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 test case **RT Baseline Test**.
- 4 Right-click **RT Baseline Test** and select **Convert to > Real-Time Test**.

Run the Real-Time Test Case

- 1 Set the **Target Computer**.

2 Set the system under test to load from Model.

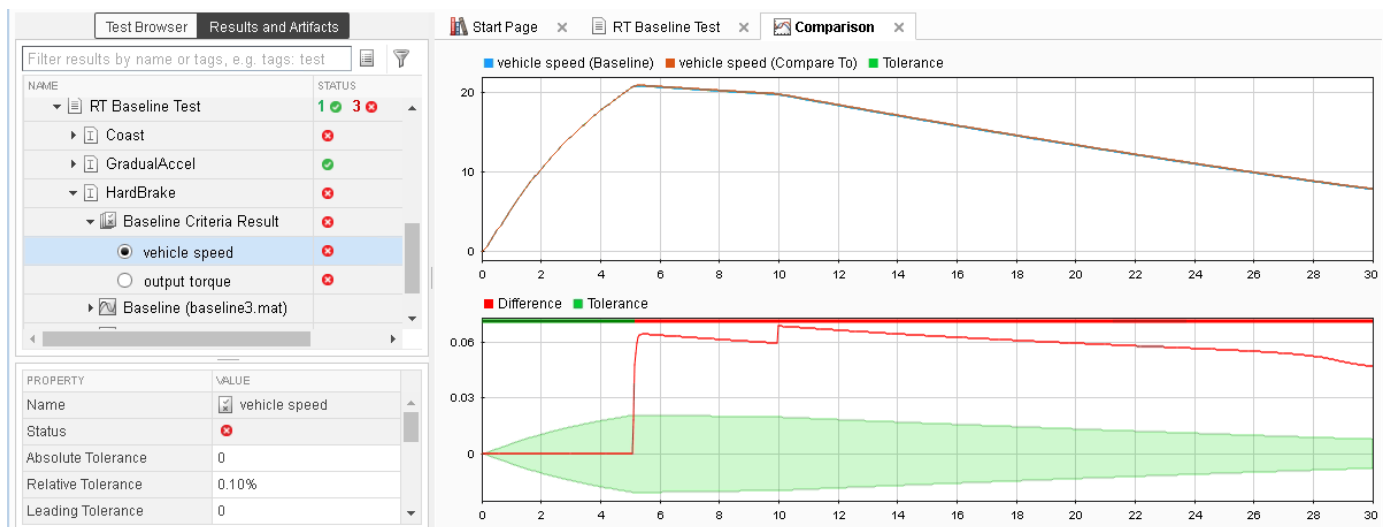


3 Run the RT Baseline Test test case.

Examine and Resolve Test Failures

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

- “Use Simulink Real-Time UI to Create and Execute a Real-Time Application” (Simulink Real-Time)

Install and Set Up the Simulink Test Support Package for ASAM XIL Standard

The ASAM® XIL standard defines communication between test automation tools and test benches. You can use the Simulink Test Support Package for ASAM XIL Standard to use Simulink Test to control a test bench.

If you are using Simulink Real-Time and switching between test benches, or if you are using a third-party test bench, you must install the support package. If you are using Simulink Real-Time as your test bench and do not need to switch between test benches, you do not need to install the Simulink Test Support Package for ASAM XIL Standard. Instead use Simulink Real-Time with the real-time test case provided with Simulink Test.

To use the ASAM XIL API, first install the Simulink Test Support Package for ASAM XIL Standard, and then set up your test bench and Simulink model.

Install the Support Package

- 1 On the MATLAB® **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
- 2 In the Add-On Explorer window, browse or search for the Simulink Test Support Package for ASAM XIL Standard.
- 3 Select the support package and click **Install**.

If you are using Simulink Real-Time as your test bench, you must also install the Simulink Real-Time Support Package for ASAM XIL Standard. See “Install the Simulink Real-Time Support Package for ASAM XIL Standard” (Simulink Real-Time).

- 4 After the support package installation completes, set up your test bench and build the model you want to run on it.

Set Up the Test Bench

- 1 Verify that your version of MATLAB is compatible with your test bench. Depending on test bench compatibility, you might have to use an older version of MATLAB to build the model for your test bench.
- 2 Install and set up your test bench. Refer to the information supplied with your test bench. If you are using Simulink Real-Time as your test bench, see “Install the Simulink Real-Time Support Package for ASAM XIL Standard” (Simulink Real-Time) for setup information.
- 3 Install the test bench compiler toolchain supplied with your test bench on your Windows computer. You use the compiler toolchain when building your Simulink model.

Configure and Build Your Model

Note Use the same version of MATLAB that you verified as compatible with your test bench.

- 1 In your Simulink model, in the **Modeling** tab, click **Model Settings** to open the Configuration Parameters dialog box. In the left pane, click **Code Generation**, then set **Toolchain** to test bench toolchain.

- 2 Build the model for your test bench, creating a binary image that is runnable on the test bench.
- 3 Create a Simulink Real-Time or vendor-specific port configuration file. See “Real-Time Testing with the Simulink Test Support Package for ASAM XIL Standard” on page 8-15 for information and examples of setting up configuration files.

Note After you have installed the support package and set up the test bench, you can build additional models and configurations at any time without having to set up the test bench again.

See Also

`sltest.xil.framework.Framework`

More About

- “Real-Time Testing with the Simulink Test Support Package for ASAM XIL Standard” on page 8-15
- “Create Tests Using the Simulink Test Support Package for ASAM XIL Standard” on page 8-19

External Websites

- ASAM XIL

Real-Time Testing with the Simulink Test Support Package for ASAM XIL Standard

The Simulink Test Support Package for ASAM XIL Standard implements the ASAM XIL API, which is a standard that defines communication between test automation tools, such as Simulink Test, and test benches, such as Simulink Real-Time and third-party test benches. The ASAM XIL API enables running real-time hardware-, software-, and model-in-the-loop (HIL, SIL and MIL, respectively) test cases created in Simulink Test using its XIL framework. The Simulink Test framework includes methods for mapping variables from the test code to the test bench, configuring the ports to use, specifying test bench startup and shutdown order, and other commands to query and control the test bench.

Simulink Test ASAM XIL Standard Workflow

This workflow describes the steps for creating a test that uses the Simulink Test Support Package for ASAM XIL Standard. The workflow tasks are:

- Install the support package and set up the test bench and your model — see “Install and Set Up the Simulink Test Support Package for ASAM XIL Standard” on page 8-13.
- “Configure the Test Bench” on page 8-15
- “Create the Test Body” on page 8-17
- “Run the Test” on page 8-17

If you want to use more than one test bench, repeat the set up and configuring the test bench tasks for each test bench. For detailed examples, see “Create Tests Using the Simulink Test Support Package for ASAM XIL Standard” on page 8-19.

Configure the Test Bench

Follow these steps to configure the test bench port, add the port to the Simulink Test ASAM XIL framework, and map variables from the model to the test bench variables. You can include the code for these steps in the same file as the test body (see “Create the Test Body” on page 8-17).

- 1 Create an instance of the `sltest.xil.framework.Framework` class. Use only one `Framework` object at a time.
- 2 Use the `displayAvailableTestbenches` method of `sltest.xil.framework.Framework` to obtain the names of the available test benches.
- 3 Create an XML port configuration file. The configuration options depend on the test bench. Replace the product version number shown in the sample port configuration file with the version you have. Sample port configuration files are:

- Simulink Real-Time

Use `createPortConfigureFile` to create the file.

- NI™ VeriStand

```
<?xml version="1.0" encoding="UTF-8"?>
<NIVSPortConfig>
  <Version Major="2020" Minor="4" Fix="0" Build="0"/>
  <Project>C:\NIProjects\Project.nivproj</Project>
</NIVSPortConfig>
```

- dSPACE®

```
<?xml version="1.0" encoding="utf-8"?>
<PortConfigurations
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  >
  <MAPortConfig>
    <SystemDescriptionFile>
      C:\DSPACEDProjects\smd_1104_sl.sdf
    </SystemDescriptionFile>
    <PlatformName>DS1104</PlatformName>
  </MAPortConfig>
</PortConfigurations>
```

- 4 Add the port to use with the Simulink Test ASAM XIL framework. See “Create Tests Using the Simulink Test Support Package for ASAM XIL Standard” on page 8-19 for where to include this code in the MATLAB code file with the test body. Replace the product version number shown in the sample port file with the version you have.

- Simulink Real-Time.

To use all of the Simulink Real-Time functionality, add all three ports.

```
framework.Configuration.addModelAccessPort(...
  'MAPort', ...
  'asamxil.v2_1', ...
  'VendorName', 'MathWorks', ...
  'ProductName', 'XIL API', ...
  'ProductVersion', '1.0', ...
  'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'));
framework.Configuration.addECUCalibrationPort(...
  'ECUCPort', ...
  'asamxil.v2_1', ...
  'VendorName', 'MathWorks', ...
  'ProductName', 'XIL API', ...
  'ProductVersion', '1.0', ...
  'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'), ...
  'TargetState', 'started');
framework.Configuration.addECUMeasurementPort(...
  'ECUMPort', ...
  'asamxil.v2_1', ...
  'VendorName', 'MathWorks', ...
  'ProductName', 'XIL API', ...
  'ProductVersion', '1.0', ...
  'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'));
```

- NI™ VeriStand

```
framework.Configuration.addModelAccessPort(...
  'MAPort1', ...
  'asamxil.v2_1', ...
  'VendorName', 'National Instruments', ...
  'ProductName', 'NI VeriStand ASAM XIL Interface', ...
  'ProductVersion', '2020', ...
  'PortConfigFile', fullfile(pwd, 'NIVeriStandPortConfig.xml'));
```

- dSPACE

```
framework.Configuration.addModelAccessPort(...
  'MAPort1', ...
  'asamxil.v2_1', ...
```



```
'vendorName', 'dSPACE GmbH', ...
'productName', 'XIL API', ...
'productVersion', '2021-A', ...
'portConfigFile', fullfile(pwd, 'dSpaceConfig.XML')));
```

Use `framework.Configuration` to display a summary of your configuration.

- 5 Map the test variable names used in the test to the test bench variable names for the specified test bench. Optionally, also specify the task (that is, the logging rate) for the variables. To display the available test bench variable IDs, use the `displayAllTestbenchVariables` method of the `sltest.xil.framework.Framework` class. To view the associated tasks, use the `displayAllTaskInfo` method. See “Create Tests Using the Simulink Test Support Package for ASAM XIL Standard” on page 8-19 for where to include this code in the MATLAB code file with the test body.

- This example maps the RPM test variable name to the `Targets/Controller/Simulation Models/Models/simple_R2020a_2/Outports/Out3` test bench variable name.

```
framework.Configuration.addTestVariableMapping(...
    'RPM', 'MAPort1', ...
    ['Targets/Controller/Simulation Models/Models'...
    '/simple_R2020a_2/Outports/Out3']);
```

Create the Test Body

Write the test body in a MATLAB code file. If you write the test body as a function, you can call it from functions that define different configurations. The test body steps are independent of a test bench. Once you define the test body, you use it as is with any test bench configuration.

The test body typically includes these steps:

- 1 Use `init` to initialize the test bench.
- 2 Instantiate the mapped test variables so you can use them in the rest of the test body.
- 3 Tune parameters in the model to desired values for your test. You can change these values at any time, but if you change values in the middle of the test, the timing is not deterministic.
- 4 Set up the acquisition, including trigger conditions, and start it.
- 5 Set up the stimulation and start it.
- 6 At any point, if desired, call the `sltest.TestCase` methods, such as `sltest.TestCase.verifyThat`, to determine the pass or fail status of the test case.
- 7 Wait for acquisition or simulation to complete, or stop the simulation using the `stop` method.
- 8 Fetch the logged data.
- 9 Push the logged data results to the Test Manager. Include pushing the logged data only if you plan to run your test using the Test Manager.

Run the Test

After you configure the test bench and write the test body, run your test in the Test Manager or at the command line.

- The MATLAB code file inherits from `sltest.TestCase`, which enables you to open and run the MATLAB code test file in the Test Manager. In the test body code, you can include pushing the logged data results to the Test Manager, and then load the file into the Test Manager using **Open**

> **Open MATLAB-Based Simulink Test (.m)**. Then, run the test like you run other test cases in the Test Manager.

- You can run your code at the MATLAB command line, however you cannot push data to the Test Manager from the command line.

Perform additional analysis on the logged data, such as using the `verifySignalsMatch` method to compare the results to the baseline data.

Limitations

These limitations of the ASAM XIL standard apply when doing real-time testing using the Simulink Test Support Package for ASAM XIL Standard.

- The ASAM XIL standard does not provide support for setting the stop time of the model. On some test benches, simulation stops at the stop time the model was built with. Other test benches force the stop time to be `Inf`.
- On some test benches, logging does not start at the same time as simulation starts. This timing difference might cause your tests to not be precisely repeatable. Use an `Acquisition` trigger to repeatedly capture a region of interest in the logged data. The trigger corresponds to `t = 0`.
- The ASAM XIL standard does not provide support for verifying results. Use the `verify` methods from `sltest.TestCase` to analyze your captured data.
- The ASAM XIL standard does not support enumerated, fixed point, or bus data types.

Troubleshooting

You can display information about available test benches, variables, and tasks by using these `sltest.xil.framework.Framework` methods:

- `displayAllAvailableTestbenches`
- `displayAllTestbenchVariables`
- `displayAllTaskInfo`

See Also

`sltest.xil.framework.Framework` | `sltest.xil.framework.FrameworkConfiguration` | `sltest.xil.framework.Acquisition` | `sltest.xil.framework.Stimulation` | `sltest.xil.framework.TestVariable`

Related Examples

- “Install and Set Up the Simulink Test Support Package for ASAM XIL Standard” on page 8-13
- “Create Tests Using the Simulink Test Support Package for ASAM XIL Standard” on page 8-19

External Websites

- ASAM XIL

Create Tests Using the Simulink Test Support Package for ASAM XIL Standard

Note The examples require that you “Install and Set Up the Simulink Test Support Package for ASAM XIL Standard” on page 8-13.

These examples show how you can develop tests using the Simulink Test Support Package for ASAM XIL Standard. One example creates a test that configures one test bench and uses data acquisition triggering. The other test has two different test points, each of which configures a different test bench, but uses the same shared test body.

The examples inherit from `sltest.TestCase`, so you can load them into the Test Manager using **Open > Open MATLAB-Based Simulink Test (.m)**. Then, run them like you run other test cases in the Test Manager. Alternatively, you can run the example files at the MATLAB command line, but you cannot push data to the Test Manager from the command line.

ASAM XIL Test With Data Acquisition Triggering

This example shows a sample MATLAB code file that uses a trigger to control data acquisition. The configuration in the `ABCCoTestPoint` function is for a test bench that supports data acquisition triggering.

```
classdef xilexample_trigger < sltest.TestCase

    methods (Test)
        function ABCCoTestPoint(testCase)
            import sltest.xil.framework.*;

            frm = Framework;

            frm.Configuration.addModelAccessPort(...
                'MAPort1', ...
                'asamxil.v2_1', ...
                'VendorName', 'ABC Co.', ...
                'ProductName', 'ABC Test Bench', ...
                'ProductVersion', '1.7', ...
                'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'));

            frm.Configuration.addTestVariableMapping(...
                'rpm', 'MAPort1', ...
                ['Targets/Controller/Simulation Models/' ...
                'Models/simpleXIL/Outputs/Out3']);
            frm.Configuration.addTestVariableMapping(...
                'temperature', 'MAPort1', ...
                ['Targets/Controller/Simulation Models/' ...
                'Models/simpleXIL/Outputs/Out4']);
            frm.Configuration.addTestVariableMapping(...
                'target_rpm', 'MAPort1', ...
                ['Targets/Controller/Simulation Models/' ...
                'Models/simpleXIL/Parameters/K']);
            frm.Configuration.addTestVariableMapping(...
                'input1', 'MAPort1',
                ['Targets/Controller/Simulation Models/' ...
```

```

        'Models/simpleXIL/Inports/Inport']);

    % Beyond this point the configuration is done and
    % the test is generic with no test bench specifics

    testBody(testCase, frm);
end
end

methods (Access = 'private')
function testBody(testCase, frm)
    import sltest.xil.framework.*;

    frm.init;

    rpm = frm.createVariable('rpm');
    temperature = frm.createVariable('temperature');
    target_rpm = frm.createVariable('target_rpm');
    input1 = frm.createVariable('input1');

    target_rpm.write(50);

    % Start acquisition when rpm reaches more than 10 and
    % stop after 15 seconds.
    frm.Acquisition.setupWithVariables([rpm, temperature], ...
        'triggerVariables', rpm, ...
        'startTriggerType', 'condition', ...
        'startTriggerVal', 'rpm > 10', ...
        'stopTriggerType', 'duration', ...
        'stopTriggerVal', 15);
    frm.Acquisition.start;

    % Set up a stimulation (external input) for the model.
    % Waveform defined here lasts 5 seconds and LoopCount
    % of 2 doubles its duration to 10 seconds.
    tseries = timeseries(cos(2*pi*(0:1000)/200)*10, (0:1000)/200);
    frm.Stimulation.setupWithVariablesAndData(...
        {{input1, tseries}}, 'LoopCount', 2);
    frm.Stimulation.start;

    frm.start;
    disp(temperature.read);

    frm.Acquisition.wait;
    frm.stop;

    result = frm.Acquisition.fetch;
    frm.pushDataToSimulinkTestManager(testCase, result);
    testCase.verifySignalsMatch(result, 'baseline1.mat');
end
end
end

```

The ABCCoTestPoint function:

- Creates a Framework object
- Adds a model access port

- Maps the test variable names to test bench variable names
- Calls the `testBody` function

The `testBody` function:

- Initializes the framework
- Instantiates the variables using the `createVariable` method
- Tunes a parameter by writing a value to the `target_rpm` variable
- Sets up the acquisition and triggering
- Sets up a time series and stimulation
- Starts the simulation and reads and displays a variable
- Waits for acquisition to complete and stops the simulation
- Fetches the result data, pushes it to the Test Manager, and compares it to baseline data

ASAM XIL Test Without Data Acquisition Triggering

This example shows a sample MATLAB code file that does not use data acquisition triggering. A `while` waits for the `temperature` variable to be less than or equal to 50, and then finishes the test. The configuration functions specify two test benches, Simulink Real-Time and ABC Co Test Bench. The test bench that is used when you run the test depends on whether you call the `ABCCoTestPoint` or the `SimulinkRealTimeTestPoint` function.

The `ABCCoTestPoint` and `SimulinkRealTimeTestPoint` functions are the same as described in “ASAM XIL Test With Data Acquisition Triggering” on page 8-19, except the `SimulinkRealTimeTestPoint` function sets up three ports, which is required by Simulink Real-Time.

```
classdef xilexample_polling < sltest.TestCase
    methods (Test)
        function ABCCoTestPoint(testCase)
            import sltest.xil.framework.*;

            frm = Framework;

            % Add the ports
            frm.Configuration.addModelAccessPort(...
                'MAPort1', ...
                'asamxil.v2_1', ...
                'VendorName', 'ABC Co.', ...
                'ProductName', 'ABC Test Bench', ...
                'ProductVersion', '1.7', ...
                'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'));

            % Create the mapping from test variables to
            % test bench port variables
            frm.Configuration.addTestVariableMapping(...
                'rpm', 'MAPort1', ...
                ['Targets/Controller/Simulation Models/'...
                'Models/simpleXIL/Outports/Out3']);
            frm.Configuration.addTestVariableMapping(...
```

```

        'temperature', 'MAPort1',
        ['Targets/Controller/Simulation Models/'...
         'Models/simpleXIL/Outports/Out4']);
frm.Configuration.addTestVariableMapping(...
    'target_rpm', 'MAPort1',...
    ['Targets/Controller/Simulation Models/'...
     'Models/simpleXIL/Parameters/K']);
frm.Configuration.addTestVariableMapping(...
    'input1', 'MAPort1',...
    ['Targets/Controller/Simulation Models/'...
     'Models/simpleXIL/Inports/Inport']);

% Beyond this point the configuration is done and
% the test is generic with no test bench specifics

% Call the generic test body
testBody(testCase, frm);
end

function SimulinkRealTimeTestPoint(testCase)
import sltest.xil.framework.*;

frm = Framework;

frm.Configuration.addModelAccessPort(...
    'MAPort', ...
    'asamxil.v2_1', ...
    'VendorName', 'MathWorks',...
    'ProductName', 'XIL API',...
    'ProductVersion', '1.0',...
    'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'));
frm.Configuration.addECUCalibrationPort(...
    'ECUCPort', ...
    'asamxil.v2_1', ...
    'VendorName', 'MathWorks',...
    'ProductName', 'XIL API',...
    'ProductVersion', '1.0',...
    'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'),...
    'TargetState', 'started');
frm.Configuration.addECUMeasurementPort(...
    'ECUMPort', ...
    'asamxil.v2_1', ...
    'VendorName', 'MathWorks', ...
    'ProductName', 'XIL API', ...
    'ProductVersion', '1.0', ...
    'PortConfigFile', fullfile(pwd, 'myConfigureFile.xml'));

frm.Configuration.addTestVariableMapping(...
    'rpm', 'ECUMPort', 'simpleXIL/Gain:1', 'TaskName', 'SubRate1');
frm.Configuration.addTestVariableMapping('temperature',...
    'ECUMPort', 'simpleXIL/Gain2:1', 'TaskName', 'SubRate2');
frm.Configuration.addTestVariableMapping('target_rpm',...
    'ECUCPort', 'simpleXIL/Gain/Gain');
frm.Configuration.addTestVariableMapping('input1',...
    'MAPort', 'simpleXIL/Inport:1');

% Beyond this point the configuration is done and
% the test is generic with no test bench specifics

```

```

        % Call the generic test body
        testBody(testCase, frm);
    end
end

methods (Access = 'private')
function testBody(testCase, frm)
    import sltest.xil.framework.*;

    frm.init;

    rpm = frm.createVariable('rpm');
    temperature = frm.createVariable('temperature');
    target_rpm = frm.createVariable('target_rpm');
    input1 = frm.createVariable('input1');

    target_rpm.write(100);

    frm.Acquisition.setupWithVariables([rpm, temperature]);
    frm.Acquisition.start;

    % Set up a stimulation (external input) for the model.
    % Waveform defined here lasts 5 seconds and LoopCount
    % of 2 doubles its duration to 10 seconds.
    tseries = timeseries(cos(2*pi*(0:1000)/200)*10, (0:1000)/200);
    frm.Stimulation.setupWithVariablesAndData(...
        {{input1,tseries}}, 'LoopCount', 2);
    frm.Stimulation.start;

    frm.start;
    while(temperature.read > 50)
        pause(1);
    end

    testCase.verifyTrue(rpm.read >= 100);

    frm.stop;
    result = frm.Acquisition.fetch;
    frm.pushDataToSimulinkTestManager(testCase, result);
end
end
end

```

The `testBody` function:

- Initializes the framework
- Instantiates the variables using the `createVariable` method
- Tunes a parameter by writing a value to the `target_rpm` variable
- Sets up the acquisition
- Sets up a time series and stimulation
- Starts the stimulation
- Starts the simulation and waits, using polling, for the temperature to be greater than 50
- Verifies that the rpm value is greater than or equal to 100

- Stops the simulation, fetches the result data, and pushes it to the Test Manager,

See Also

`sltest.xil.framework.Framework` | `sltest.xil.framework.FrameworkConfiguration` |
`sltest.xil.framework.Acquisition` | `sltest.xil.framework.Stimulation` |
`sltest.xil.framework.TestVariable` | `sltest.TestCase`

Related Examples

- “Install and Set Up the Simulink Test Support Package for ASAM XIL Standard” on page 8-13
- “Real-Time Testing with the Simulink Test Support Package for ASAM XIL Standard” on page 8-15

External Websites

- [ASAM XIL](#)

Testing Custom C/C++ Code

- “Importing and Testing Custom C/C++ Code” on page 9-2
- “Import Custom Code for Unit Testing Using API Commands” on page 9-5
- “Conduct Unit Testing on Imported Custom Code by Using the Wizard” on page 9-11

Importing and Testing Custom C/C++ Code

You can test custom C or C++ code by importing it into Simulink using the Code Importer wizard in the Test Manager or API commands at the MATLAB command line. You can perform unit testing to test a subset of your C code or integration testing to test your complete C or C++ code. When you import your code, the code importer:

- Converts the C code functions to Simulink C Caller blocks and saves those blocks in a Simulink library
- Creates an internal harness for each Simulink C Caller block
- Generates a test file

For unit tests, the code importer additionally creates a sandbox to isolate the imported functions.

Import Code Using the Wizard or the API

To import and test custom C or C++ code using the Code Importer wizard, open the Test Manager and select **New > Test for C/C++ Code**. The wizard steps are shown in the “Conduct Unit Testing on Imported Custom Code by Using the Wizard” on page 9-11 example. After you import the code using the wizard, the **Test Browser** pane of the Test Manager shows the generated test file, test suites, and test cases, and automatically fills the library model and test harness fields and coverage settings for each test case. You can customize the test cases in the Test Manager by adding inputs, assessments, links to requirements, or other options.

The “Import Custom Code for Unit Testing Using API Commands” on page 9-5 example shows the classes and methods for importing code. The code importer sets the property values for the generated library, test file, test suites, test cases, and coverage. You can customize the test cases by using API commands to add inputs, assessments, links to requirements, or other options.

Before you run the test cases, either change the current folder to the folder that contains the generated artifacts or add the generated data dictionary to the path. Then run the test cases and view the coverage and other test results.

Code Importer Generated Artifacts

The code importer creates these artifacts:

- A Simulink library with C Caller blocks for each imported custom code function.
- An internal test harness for each C Caller block. For each generated harness, the solver is `FixedStepDiscrete`, and coverage is enabled.
- An MLDATX test file. The test file includes a test suite and test case for each C Caller block. The code importer also sets these coverage types:
 - Decision coverage
 - Condition coverage
 - MCDC coverage
 - Lookup table coverage
 - Signal range coverage
 - Coverage for Simulink Design Verifier blocks

- Relational boundary coverage
- Signal range coverage
- Simulink data dictionary

Additionally, for unit tests only, the code importer creates:

- A sandbox to isolate the functions being tested
- Stubs, if any source files have undefined symbols

Limitations and Workarounds

These limitations and workarounds apply to using the custom C or C++ Code Importer.

General Limitations and Workarounds

- For integration tests, if your code includes C++ functions, add wrappers around them to make them C-compatible before importing the functions into Simulink.
- These C types, and functions with formal arguments that use these types, cannot be imported. Global variables that use these types are not exposed as ports on the C Caller block:
 - Structures with unions or pointer members
 - Functions with inputs that have more than one level of pointer indirection (for example, `>=**`)
 - Functions that return a pointer
 - Types with names longer than 63 characters, and functions and variables that use those types
- If your code has many global variables, use a Stateflow chart instead of an Initialize Function block to set the variables to their initial values in your Simulink model, .
- If a header file or C or C++ file contains assembly code that is defined in the function body, the code importer does not import that function. This limitation applies only if the assembly code is not compatible with the host computer. To import the function,
 - For integration tests, replace the assembly code by using an `#ifdef` directive.
 - For unit tests, the function is moved automatically to the `auto_stub.c` file with an empty body. To import the function into Simulink, manually stub the function in `man_stub.c`.
- For target-specific code, if the code accesses absolute memory addresses, comment out that code to prevent the simulation from failing.

Header Files

- If the same header file is included multiple times and each inclusion is preceded by a different preprocessor directive (such as, `#define X 1`, `#define X 2`), the code might not import correctly.
- If assembly code is defined in the header file, define a compatible macro by using an `#ifdef` directive. For example, if your code is:

```
#define XYZ(K,L) {\
asm("MOVLW " __mkstr(K) ); \
asm("MOVLW " __mkstr(L) ); \
}
```

replace it with:

```
#ifndef IS_SL_IMPORT
#define XYZ(K,L) {\
asm("MOVLW " __mkstr(K) ); \
asm("MOVLW " __mkstr(L) ); \
}
#else
// a valid implementation
#endif
```

Then, add `IS_SL_IMPORT` to the list of defines when you import code.

Unit Tests

These limitations and workarounds apply only to unit tests.

- Only C code is supported for importing.
- If a source or header file contains a function definition that is included multiple times in the source file being imported, update the code so the function definition appears only once.
- All included files, using `#include`, should be self-contained, that is, they compile on their own. Specifically, a header should have header guard and include all other headers that it needs.

See Also

`sltest.CodeImporter` | `sltest.CodeImporter.SandboxSettings` | `createSandbox`

Related Examples

- “Conduct Unit Testing on Imported Custom Code by Using the Wizard” on page 9-11
- “Import Custom Code for Unit Testing Using API Commands” on page 9-5

Import Custom Code for Unit Testing Using API Commands

This example shows how to use the API to import custom C code for a heat pump controller into Simulink for unit testing. Unit tests test one or more functions in isolation from the custom code library. For unit tests, the Simulink Test code importer generates a test sandbox and a library containing a C Caller block from the specified custom code.

Heat Pump Controller Custom Code Files

The complete code for the heat pump controller is in these C code source and header files:

The source files are in the `src` directory:

- `tempController.c`
- `utils.c`

The header files are in the `include` directory:

- `tempController.h`
- `utils.h`
- `controllerTypes.h`

The `tempController.c` file contains the algorithm for the custom C code for a heat pump unit. The `heatpumpController` function in that file uses the room temperature (`Troom_in`) and the set temperature (`Tset`) as inputs. The output is `pump_control_bus` type structure with signals that control the fan, heat pump, and the direction of the heat pump (heat or cool). The `pump_control_bus` structure has these fields: `fan_cmd`, `pump_cmd`, and `pump_dir`. The `pump_control_bus` structure type is defined in the `controllerTypes.h` file. The output of the `heatpumpController` function is:

Temperature Condition	System State	Fan Command	Pump Command	Pump Direction
$ T_{room_in} - T_{set} < \Delta T_{fan}$	Idle	0	0	<i>IDLE</i>
$\Delta T_{fan} \leq T_{room_in} - T_{set} < \Delta T_{pump}$	Fan only	1	0	<i>IDLE</i>
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} < T_{room_in}$	Cooling	1	1	<i>COOLING</i>
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} > T_{room_in}$	Heating	1	1	<i>HEATING</i>

The `heatpumpController` function uses two utility functions, `absoluteTempDifference` and `pumpDirection`, which are defined in the `utils.c` file. The `absoluteTempDifference` function returns the absolute difference between `Tset` and `Troom_in` as a double. The `pumpDirection` function returns one of these `PumpDirection` type enum values:

Temperature Condition	Pump Direction
$T_{set} < T_{room_in}$	COOLING
$T_{set} > T_{room_in}$	HEATING

The `PumpDirection` enum type is defined in the `controllerTypes.h` file.

Import Heat Pump Controller Code and Automatically Create Stubs

This example uses only `tempController.c` to create and import a test sandbox into Simulink. You use the sandbox for performing unit testing only on the `heatpumpController` function and not on

the complete code. Generating the sandbox automatically creates stubs for the utility functions used by the `heatpumpController` function, `absoluteTempDifference` and `pumpDirection`. Since the utility functions are not defined in the `tempController.c` file and the `utils.c` and `utils.h` files are not included, the code importer creates stubs so the code does not error.

Set Up the CodeImporter Object

Create an instance of a `CodeImporter` object for the heat pump controller custom code. Setting the `OutputFolder` property to `pwd` evaluates the string in between the `$` symbols as a MATLAB expression. Set `OutputFolder` to `pwd` to specify the current folder as the output folder. Set the `SourceFiles` property to the `tempController.c` file in the `src` directory. Use the `$` symbols to specify the file location for the `CustomCode` property, too.

```
obj = sltest.CodeImporter('heatpumpController');  
  
obj.OutputFolder = "$pwd$";  
  
obj.CustomCode.SourceFiles = "$fullfile('src','tempController.c')$";  
obj.CustomCode.IncludePaths = fullfile('include');  
obj.CustomCode.GlobalVariableInterface = true;
```

Create the Test Sandbox

Configure the `CodeImporter` object with the desired test type and sandbox settings.

To create a test sandbox for the specified `heatpumpController` function in the custom code, set the `TestType` property `UnitTest`. For this example, use the `GenerateAggregatedHeader` sandbox mode. For information about the different sandbox modes, see `sltest.CodeImporter.SandboxSettings`.

Setting `SandboxSettings.CopySourceFiles` to `true` copies the specified source file into the test sandbox.

Note that you can use `GenerateAggregatedHeader` sandbox mode only with a single source file.

```
obj.TestType = "UnitTest";  
  
obj.SandboxSettings.Mode = "GenerateAggregatedHeader";  
obj.SandboxSettings.CopySourceFiles = true;
```

Create the sandbox. This test sandbox is isolated from the original custom code library. Setting `Overwrite` to `on` overwrites the existing test sandbox, if one exists. By default, `Overwrite` is `off`.

```
obj.createSandbox('Overwrite','on');
```

The `createSandbox` method creates the `heatpumpController_sandbox` directory in the specified output folder, which in this example is the current working folder.

The sandbox directory contains the following subdirectories:

- `src`: This directory contains the copied source file, `tempController.c`.
- `include`: This directory contains the required include files to compile `tempController.c` in the sandbox `src` directory. This directory also contains the `aggregatedHeader.h` file, which contains all the required symbols to compile `tempController.c`.

- `autostub`: This directory contains the `auto_stub.c` and `auto_stub.h` files, which hold the automatically generated stubs for `absoluteTempDifference` and `pumpDirection` utility functions.
- `manualstub`: This directory contains the `man_stub.c` and `man_stub.h` files, which define any manually specified stubs. By default, these files do not define any functions.

Import the Test Sandbox

Import the sandbox code into Simulink.

```
obj.import('Functions', 'heatpumpController');
```

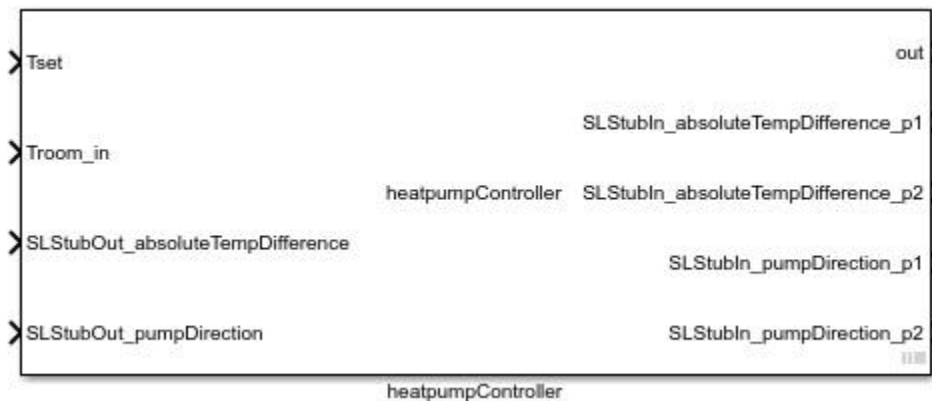
The `import` function creates the sandbox. It also creates a library that contains a C Caller block called `heatpumpController`, which contains an internal test harness that you can use to perform unit testing on the `heatpumpController`. The library is attached to a Simulink Data Dictionary that defines `pump_control_bus` and `PumpDirection` as a `Simulink.Bus` object and a Simulink enumeration signal, respectively.

The C Caller block is attached with an internal test harness for unit testing the `heatpumpController` function.

Input and Output Ports on the C Caller Block

Because you set `CustomCode.GlobalVariableInterface` to `true` before importing, the `import` function creates stubs for the `absoluteTempDifference` and `pumpDirection` global variables in `auto_stub.c` and creates ports for them. For information, see “Enable global variables as function interfaces”.

This is the C Caller block, `heatpumpController`, generated from the `heatpumpController` function:



These are the automatically generated global variables in the `auto_stub.c` file for `absoluteTempDifference` and `pumpDirection`:

```

/*****
/* Generated Global Variables for Stubbed Functions Interface */
/*****
double SLStubIn_absoluteTempDifference_p1;
double SLStubIn_absoluteTempDifference_p2;

```

```

double SLStubOut_absoluteTempDifference;
double SLStubIn_pumpDirection_p1;
double SLStubIn_pumpDirection_p2;
PumpDirection SLStubOut_pumpDirection;

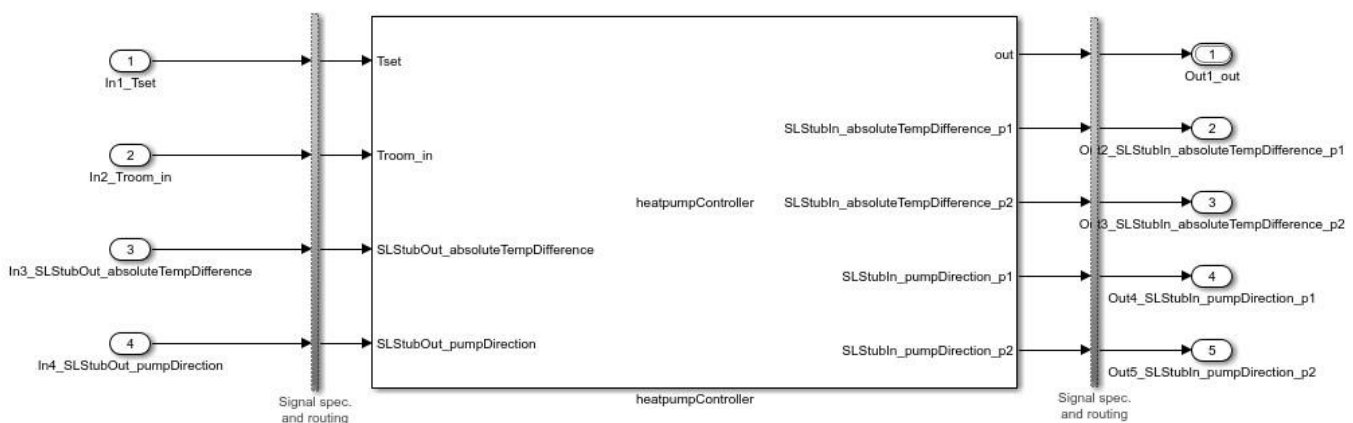
double absoluteTempDifference( double absoluteTempDifference_p1, double absoluteTempDifference_p2)
{
    SLStubIn_absoluteTempDifference_p1 = absoluteTempDifference_p1;
    SLStubIn_absoluteTempDifference_p2 = absoluteTempDifference_p2;
    return SLStubOut_absoluteTempDifference;
}

PumpDirection pumpDirection( double pumpDirection_p1, double pumpDirection_p2)
{
    SLStubIn_pumpDirection_p1 = pumpDirection_p1;
    SLStubIn_pumpDirection_p2 = pumpDirection_p2;
    return SLStubOut_pumpDirection;
}

```

In the automatically generated stubs for the `absoluteTempDifference` function, the global variables `SLStubIn_absoluteTempDifference_p1` and `SLStubIn_absoluteTempDifference_p2` save the input arguments of the function. The function returns the value stored in `SLStubOut_absoluteTempDifference`. Similarly, `pumpDirection` saves the input arguments and returns `SLStubOut_pumpDirection`.

To use the test harness created using automatically created stubs, refer to the next figure. Add buses for the inputs and outputs. To enable connecting a Simulink signal for simulation, connect inputs for `Tset`, `Troom_in` and the expected outputs from the global variables, `SLStubOut_absoluteTempDifference` and `SLStubOut_pumpDirection`. Likewise, connect outputs as shown in the figure. You can use the inputs and outputs to observe the internal values passed by `heatpumpController` to the `absoluteTempDifference` and `pumpDirection` subfunction calls.



Change Automatically Created Stubs to Manual Stubs

In cases where, for example, you want to generate the intended output of the stubs as input to the automatically generated ports, you can substitute manual stubs for the stubs automatically generated

when the sandbox was created. After you switch to using manual stubs, you update the existing sandbox and import it again.

Manually Modify an Automatically Generated Stub Function

You can manually provide definitions for the automatically generated stub functions by updating the `man_stub.c` and `man_stub.h` files in the `manualstub` directory.

```
manualstubpath = fullfile([obj.LibraryFileName.char '_sandbox'], 'manualstub');
helperFunctionToUpdateManualStubs(manualstubpath);
```

The `helperFunctionToUpdateManualStubs` function updates the manual stub files in the test sandbox.

The updated function definition of `absoluteTempDifference` is:

```
double absoluteTempDifference(double Tset, double Troom_in){
    return (double)fabs(Tset - Troom_in);
}
```

The updated function definition of `pumpDirection` is:

```
PumpDirection pumpDirection(double Tset, double Troom_in){
    return Tset > Troom_in ? HEATING : COOLING;
}
```

Update the Existing Test Sandbox

To use the manual stub functions, update the sandbox to reflect your changes. Setting the `Overwrite` option to `off` preserves the changes made to the `manualstub` directory in the test sandbox.

```
obj.createSandbox('Overwrite', 'off');
```

After updating the test sandbox, the `autostub` directory is empty because you defined all of the undefined symbols in the specified custom code.

Import the Updated Test Sandbox

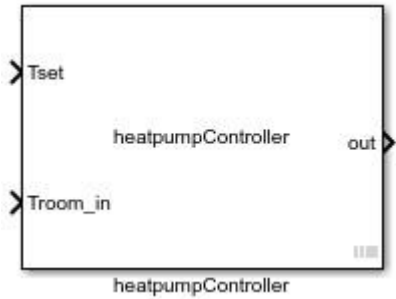
After updating the test sandbox, import the sandbox code into Simulink.

```
obj.import('Functions', 'heatpumpController');
```

The library contains a C Caller block called `heatpumpController` with updated ports, and The internal test harness attached to the C Caller block is also updated.

The import function updates the existing library and imports the `heatpumpController` function. Like when you used automatically generated stubs, the library is attached to a Simulink Data Dictionary that defines `pump_control_bus` and `PumpDirection` as a `Simulink.Bus` object and a Simulink enumeration signal, respectively.

The C Caller block ports reflect the changes made to the stub files. Because the manual implementation of the `absoluteTempDifference` and `pumpDirection` functions does not use any global variables, the C Caller block only has ports for the input arguments and return argument of the `heatpumpController` function. The internal test harness attached to the C Caller block is also updated.



You can use the created MLDATX test file to test the code. See “Conduct Unit Testing on Imported Custom Code by Using the Wizard” on page 9-11 for a testing example.

See Also

`sltest.CodeImporter` | `sltest.CodeImporter.SandboxSettings` | `createSandbox` | `Simulink.CodeImporter.CustomCode` | `import` | `Simulink.CodeImporter` | `Simulink.CodeImporter.ParseInfo` | `Simulink.CodeImporter.Options`

More About

- “Importing and Testing Custom C/C++ Code” on page 9-2

Conduct Unit Testing on Imported Custom Code by Using the Wizard

This example shows how to use the Code Importer wizard to import custom C code for a heat pump controller into Simulink for unit testing. Unit tests test one or more functions in isolation from the custom code library. For unit tests, the Simulink Test Code Importer wizard generates a test sandbox and a library containing a C Caller block for each specified function from the custom code. For more information see “Importing and Testing Custom C/C++ Code” on page 9-2.

This example uses only `tempController.c` file to create and import a test sandbox into Simulink. You use the sandbox for performing unit testing only on the `heatpumpController` function in the `tempController.c` file and not on the complete code. Generating the sandbox automatically creates stubs for the utility functions used by the `heatpumpController` function, `absoluteTempDifference` and `pumpDirection`. Since the utility functions are not defined in the `tempController.c` file, and the `utils.c` and `utils.h` files are not included, the code importer creates stubs so the code does not error.

Heat Pump Controller Custom Code

The complete code for the heat pump controller is in these C code source and header files:

The source files are in the `src` directory:

- `tempController.c`
- `utils.c`

The header files are in the `include` directory:

- `tempController.h`
- `utils.h`
- `controllerTypes.h`

The `tempController.c` file contains the algorithm for the custom C code for a heat pump unit. The `heatpumpController` function in that file uses the room temperature (`Troom_in`) and the set temperature (`Tset`) as inputs. The output is the `pump_control_bus` type structure with signals that control the fan, heat pump, and the direction of the heat pump (heat or cool). The fields in the `pump_control_bus` structure are `fan_cmd`, `pump_cmd`, and `pump_dir`. The `pump_control_bus` structure type is defined in the `controllerTypes.h` file. The output of the `heatpumpController` function is:

The output of the `heatpumpController` algorithm is summarized in the following table:

Temperature Condition	System State	Fan Command	Pump Command	Pump Direction
$ T_{room_in} - T_{set} < \Delta T_{fan}$	Idle	0	0	<i>IDLE</i>
$\Delta T_{fan} \leq T_{room_in} - T_{set} < \Delta T_{pump}$	Fan only	1	0	<i>IDLE</i>
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} < T_{room_in}$	Cooling	1	1	<i>COOLING</i>
$ T_{room_in} - T_{set} \geq \Delta T_{pump}$ and $T_{set} > T_{room_in}$	Heating	1	1	<i>HEATING</i>

The `heatpumpController` function uses two utility functions, `absoluteTempDifference` and `pumpDirection`, which are defined in the `utils.c` file. The `absoluteTempDifference` function

returns the absolute difference between `Tset` and `Troom_in` as a double. The `pumpDirection` function returns one of these `PumpDirection` type enum values:

Temperature Condition Pump Direction

<code>Tset < Troom_in</code>	COOLING
<code>Tset > Troom_in</code>	HEATING

The `PumpDirection` enum type is defined in the `controllerTypes.h` file.

Open the Code Importer Wizard

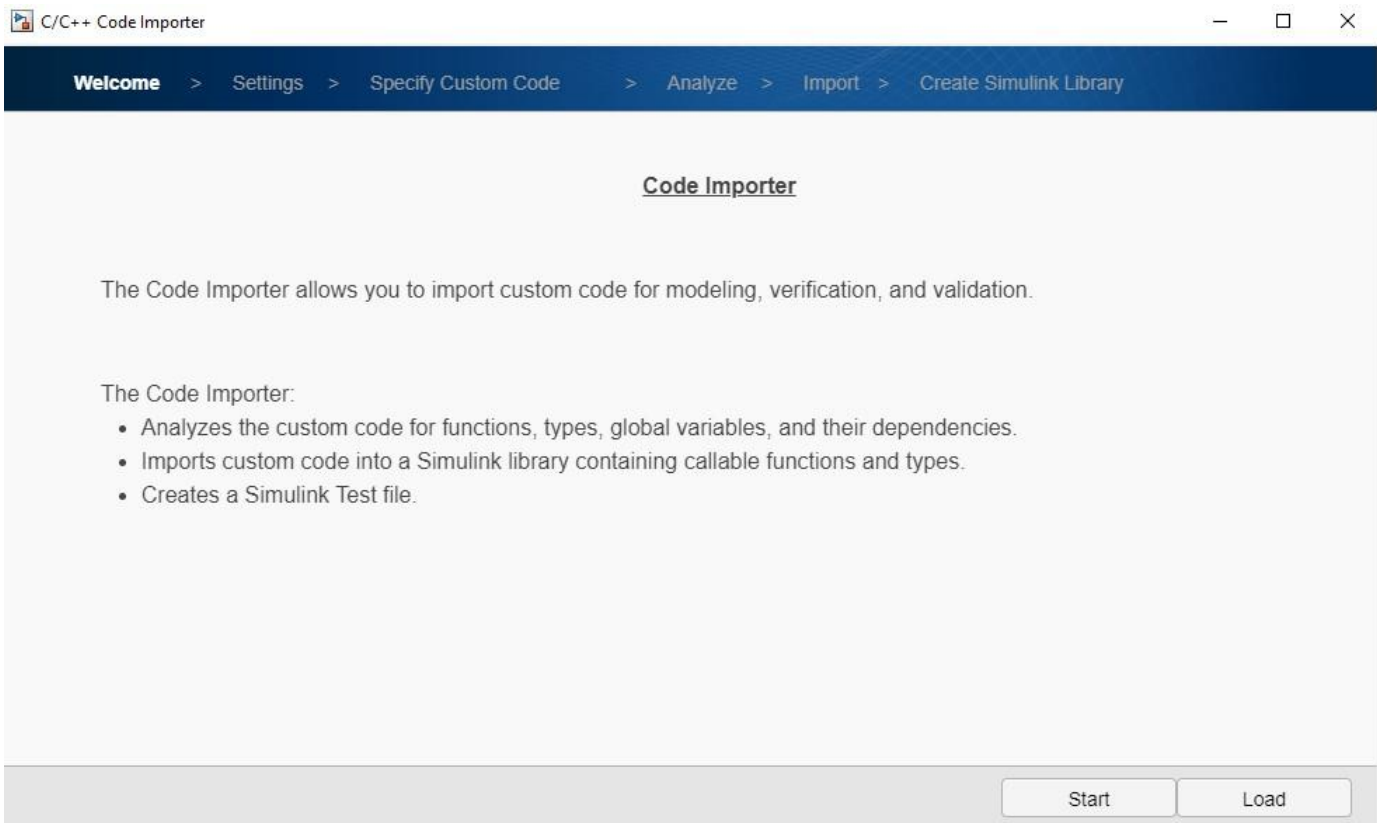
To open the Simulink Test C/C++ Code Importer wizard, first open the Simulink Test Manager.

```
sltest.testmanager.view
```

Then, select **New > Test for C/C++ code**.

Specify the Simulink Library and Testing Method

The wizard opens and displays the **Welcome** tab. Click **Start** to begin the import process.



On the **Settings** tab:

- 1 Enter the **Simulink library file name**. The generated Simulink library, test sandbox directory, and test file (MLDATX) use this name. Enter `heatpumpController`.
- 2 Specify the **Output folder** in which to save the generated artifacts. Enter the name of a writable folder.

3 Select **C Code Unit Testing** as the testing method.

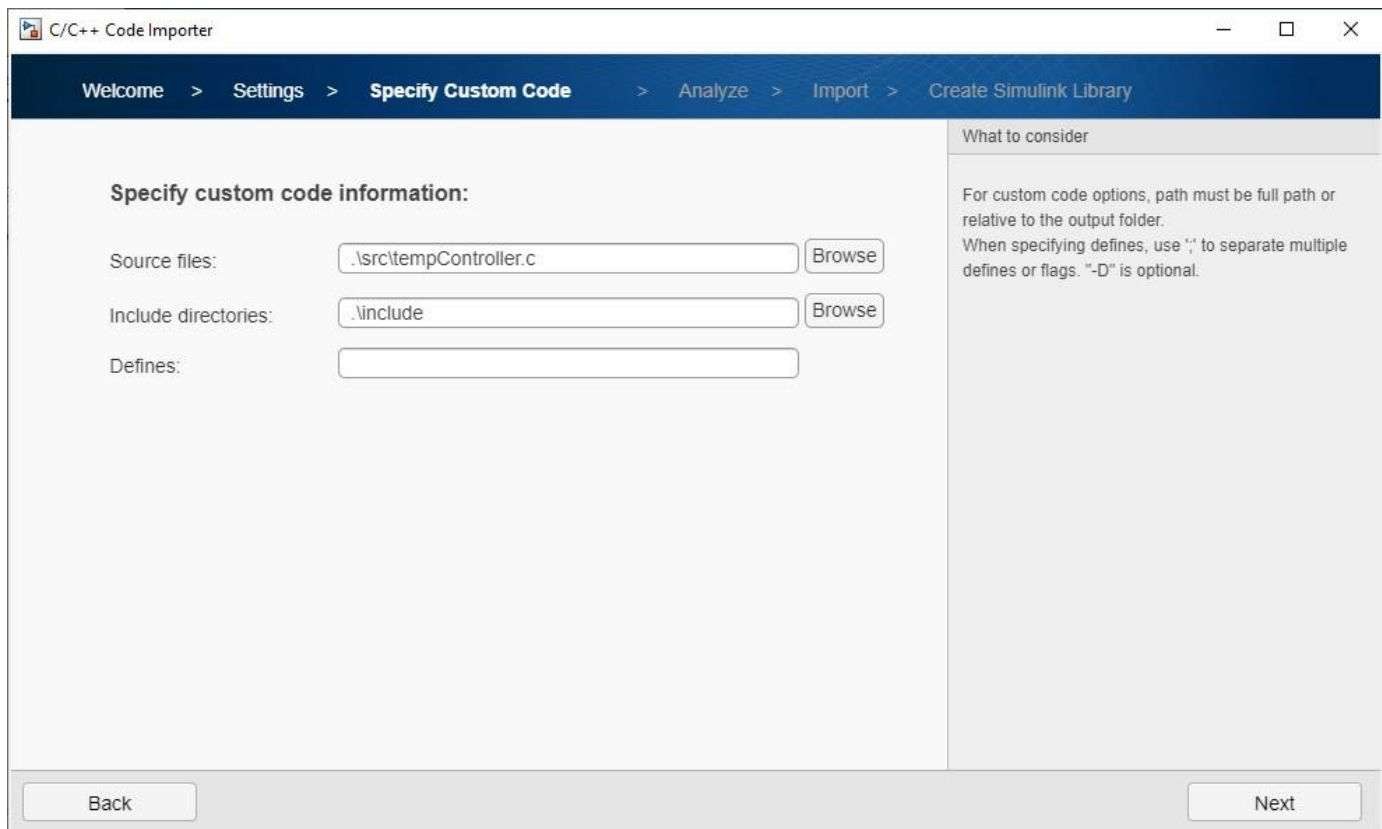
The C Code Unit Testing method tests one C file or a subset of your custom code in isolation. This method creates a test sandbox from the specified source file or files, and imports the test sandbox into Simulink. Because the test sandbox only contains a subset of the C files, the Code Importer wizard automatically creates stubs for all of the undefined symbols used in the C files. This method supports testing C code only. For information on Integration Testing, see the `TestType` property of `sltest.CodeImporter`.

Click **Next**.

Specify the Custom Code to Import

On the **Specify Custom Code** tab:

- 1 In **Source files**, specify the source file that contains the function to import for unit testing. Enter `.\src\tempController.c`.
- 2 In **Include directories**, specify the directories on which the specified source files depend. Enter `.\include`.
- 3 **Defines** specifies the compiler-specific defines. For this example, leave this field blank.



Click **Next**.

Specify the Test Sandbox Settings

On the **Analyze** tab, specify the sandbox settings.

Select the output test sandbox mode

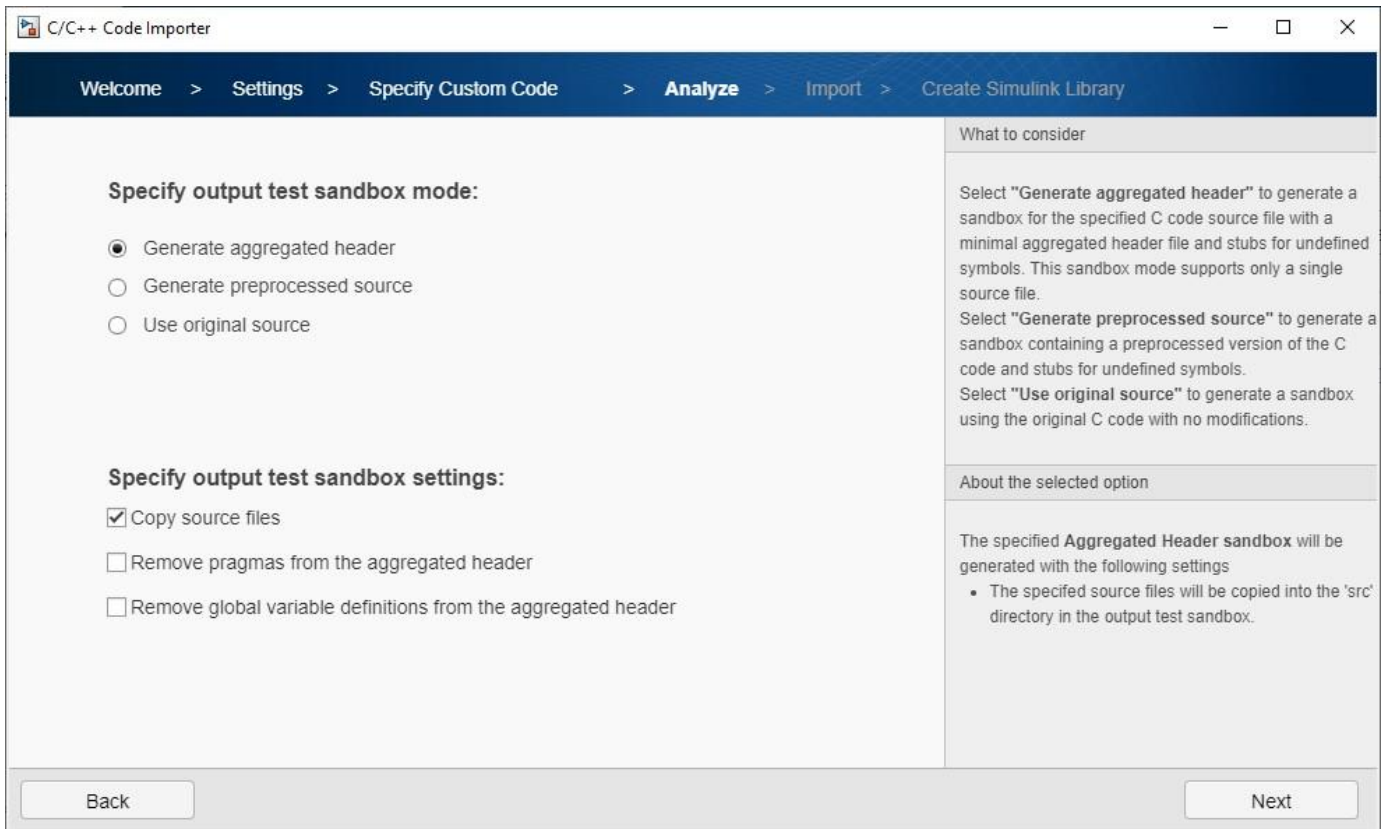
Each of the three test sandbox modes has settings that determine how the Code Importer wizard generates the sandbox and the sandbox artifacts.

Select **Generate aggregated header**. This test sandbox mode generates a minimal aggregated header file for the specified source file. The aggregated header file contains all the declarations of the symbols used by the specified source file.

Select the output test sandbox settings

Select only **Copy source files**. For this option, the Code Importer wizard copies the specified source file to the sandbox `src` directory.

For information on the other output test sandbox modes and settings, see the properties of `sltest.CodeImporter.SandboxSettings`.



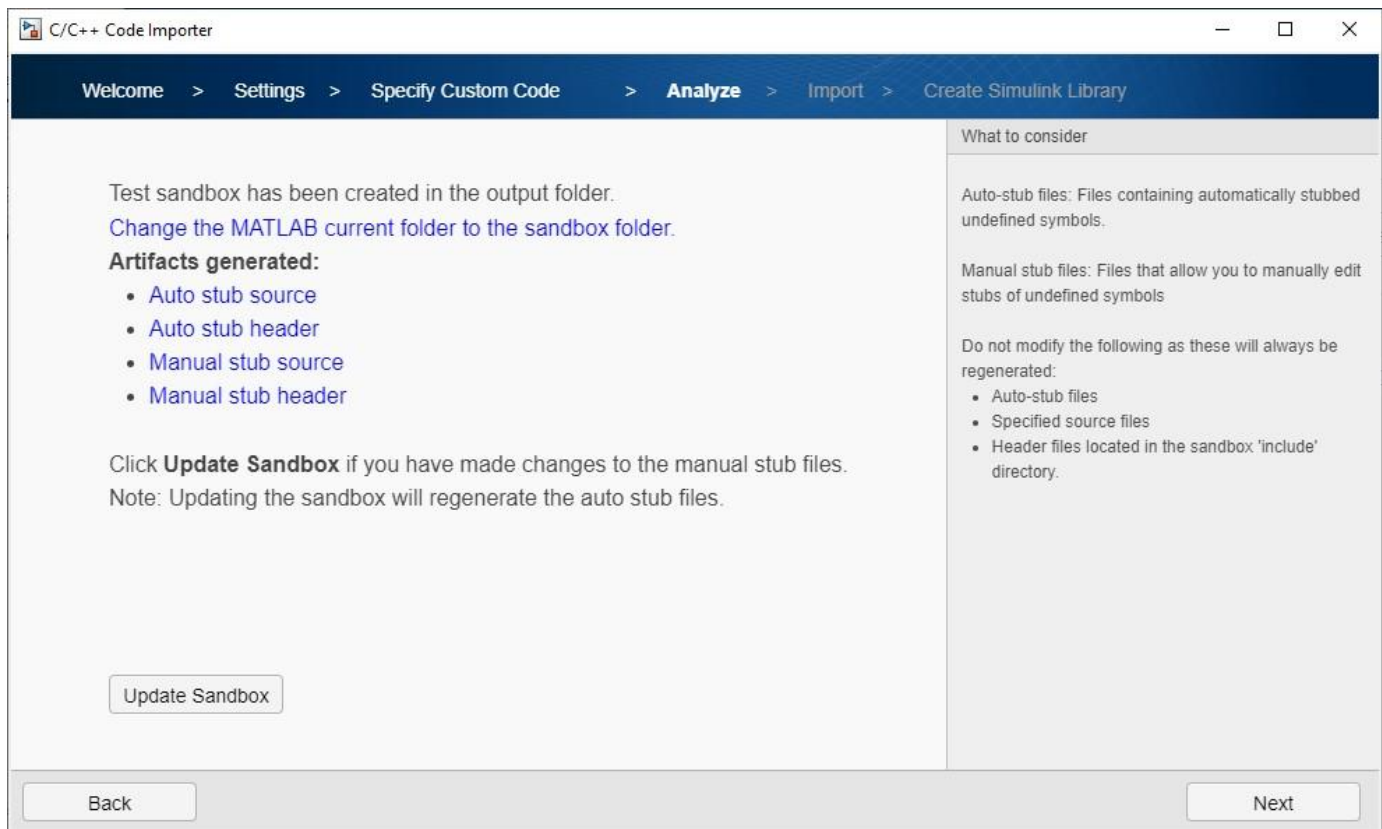
Click **Next** to create the test sandbox with the specified settings.

Create the Test Sandbox

The specified library file name determines the name of the generated sandbox. For this example, the sandbox name is `heatpumpController_sandbox`.

This example uses `tempController.c` and performs unit testing only on the `heatpumpController` function. Generating the sandbox automatically creates stubs for the `absoluteTempDifference` and `pumpDirection` utility functions used by the `heatpumpController` function. Since the utility functions are not defined in the `tempController.c` file, the stubs prevent the code from erroring.

When the sandbox is created, the confirmation screen displays.



The output sandbox folder contains the following subfolders:

- `src`: This folder contains the copied source file or files, which for this example is `tempController.c`.
- `include`: Depending on the test sandbox mode, this folder contains either the `aggregatedHeader.h` or `interfaceHeader.h` file. This example uses the `aggregatedHeader.h` file. The folder also contains and copies of the other header files required for the compilation of the specified source file or files. The `aggregatedHeader.h` file contains all the declarations of the symbols used by the specified source file. The `interfaceHeader.h` contains the declarations for functions, global variables, and types used by the specified source files. Simulink uses the generated interface header file during the import process.
- `autostub`: This folder contains the `auto_stub.c` and `auto_stub.h` files, which hold the automatically generated stubs for `absoluteTempDifference` and `pumpDirection` utility functions.
- `manualstub`: This folder contains the `man_stub.c` and `man_stub.h` files, which define any manually specified stubs. By default, these files do not define any functions.

NOTE: Do not modify the files in the `src`, `include`, or `autosub` folders of the generated sandbox.

After the sandbox has been created, you can view the created files:

- Click **Auto stub source** and **Auto stub header** to open the `auto_stub.c` and `auto_stub.h` files, respectively. In `auto_stub.c`, you can find the stubs for the utility functions used by `heatpumpController`, `absoluteTempDifference` and `pumpDirection`. In `auto_stub.h`, you can find the extern declarations of the stubbed functions.

- Click **Manual stub header** and **Manual stub source** to open the `man_stub.h` and `man_stub.c` files, respectively. Because you selected **Generate aggregated header** for the sandbox mode, `man_stub.h` includes the `aggregatedHeader.h` file. By default, the manual stub files do not have any function definitions.

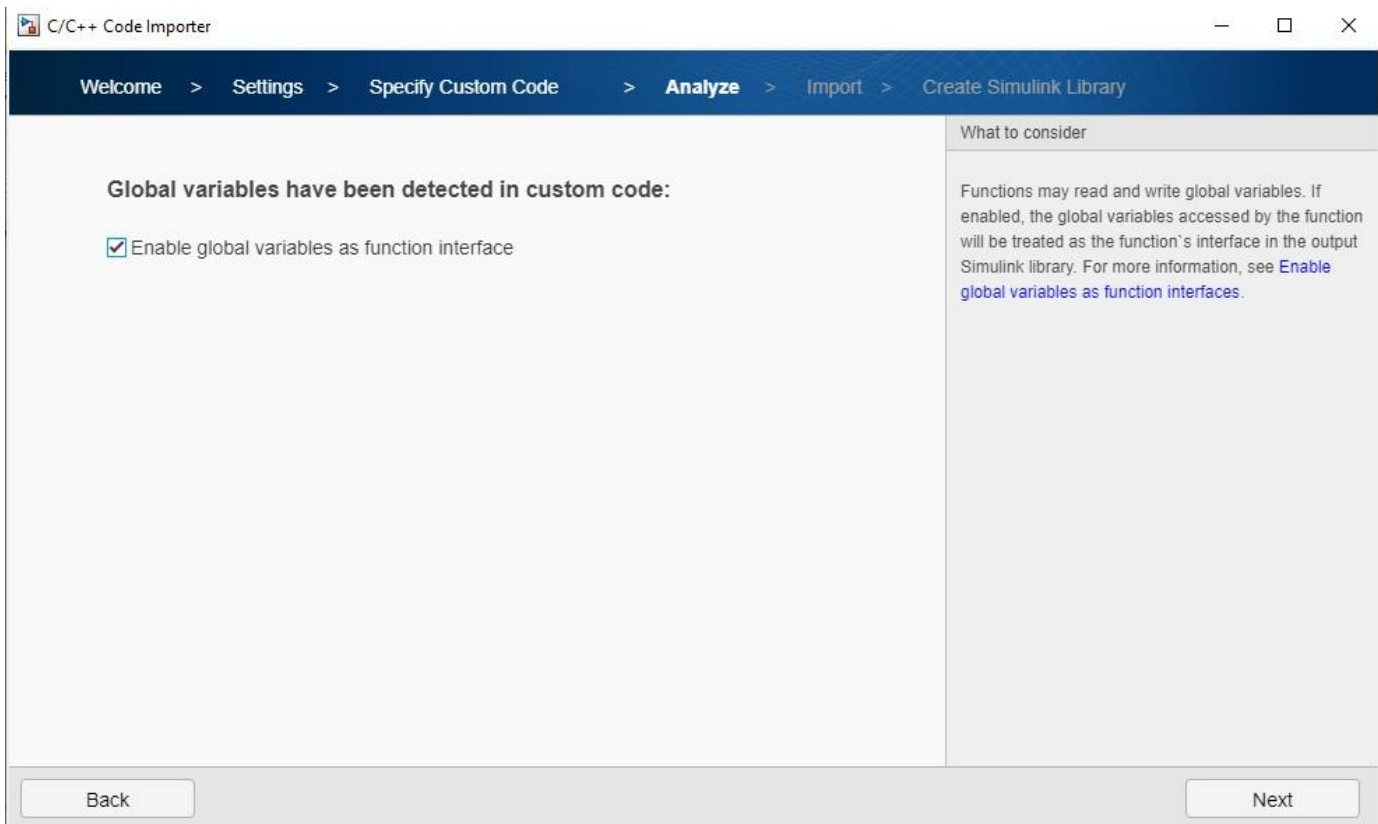
Click **Next**.

Specify Import Settings

Specify Using Global Variables as Function Interfaces

When the wizard detects global variables in the sandbox or custom code, you have the option to use those variables as input or outputs for the function and ports on the created C Caller block. For more information, see “Enable global variables as function interfaces”.

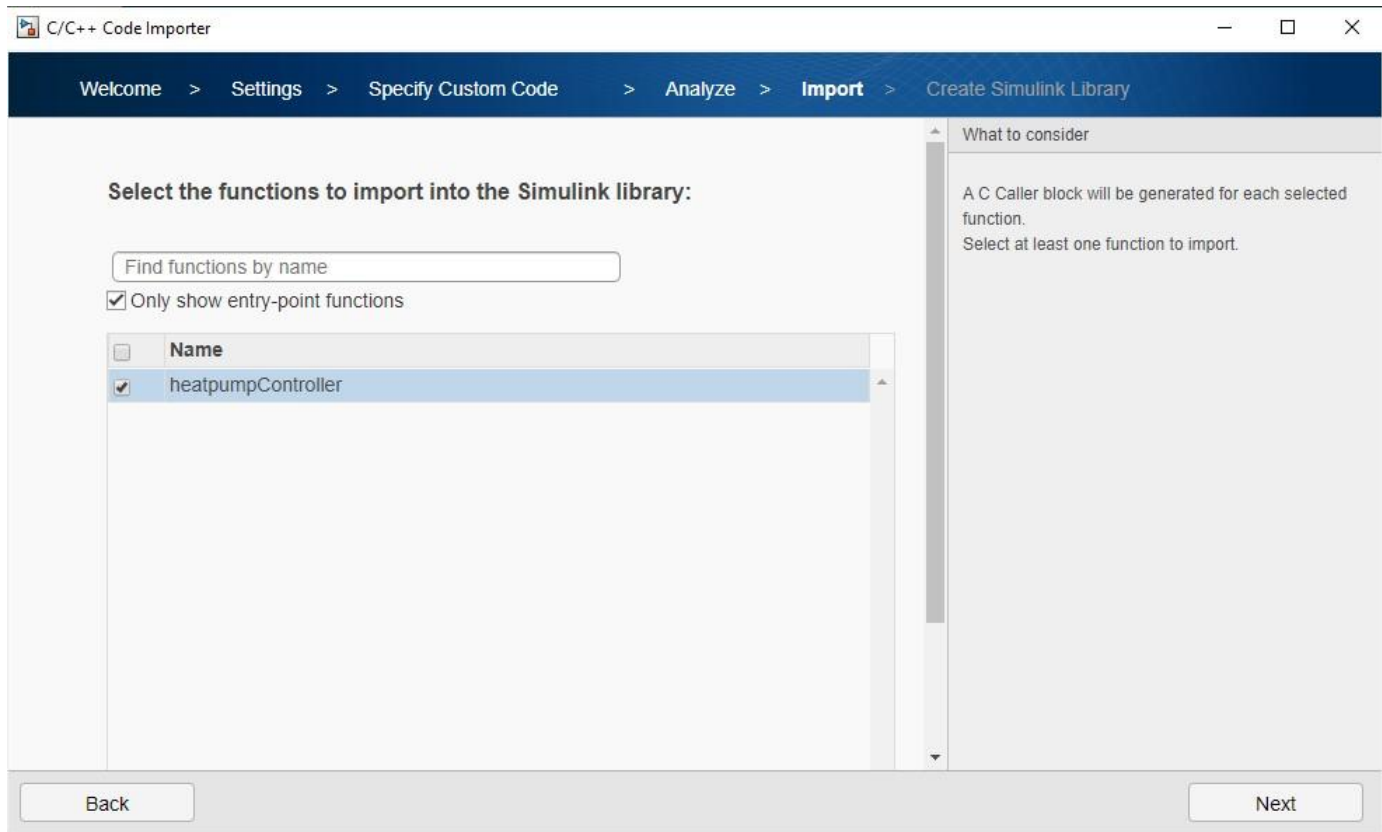
Select **Enable global variables as function interface** option.



Click **Next**.

Select Functions to Import

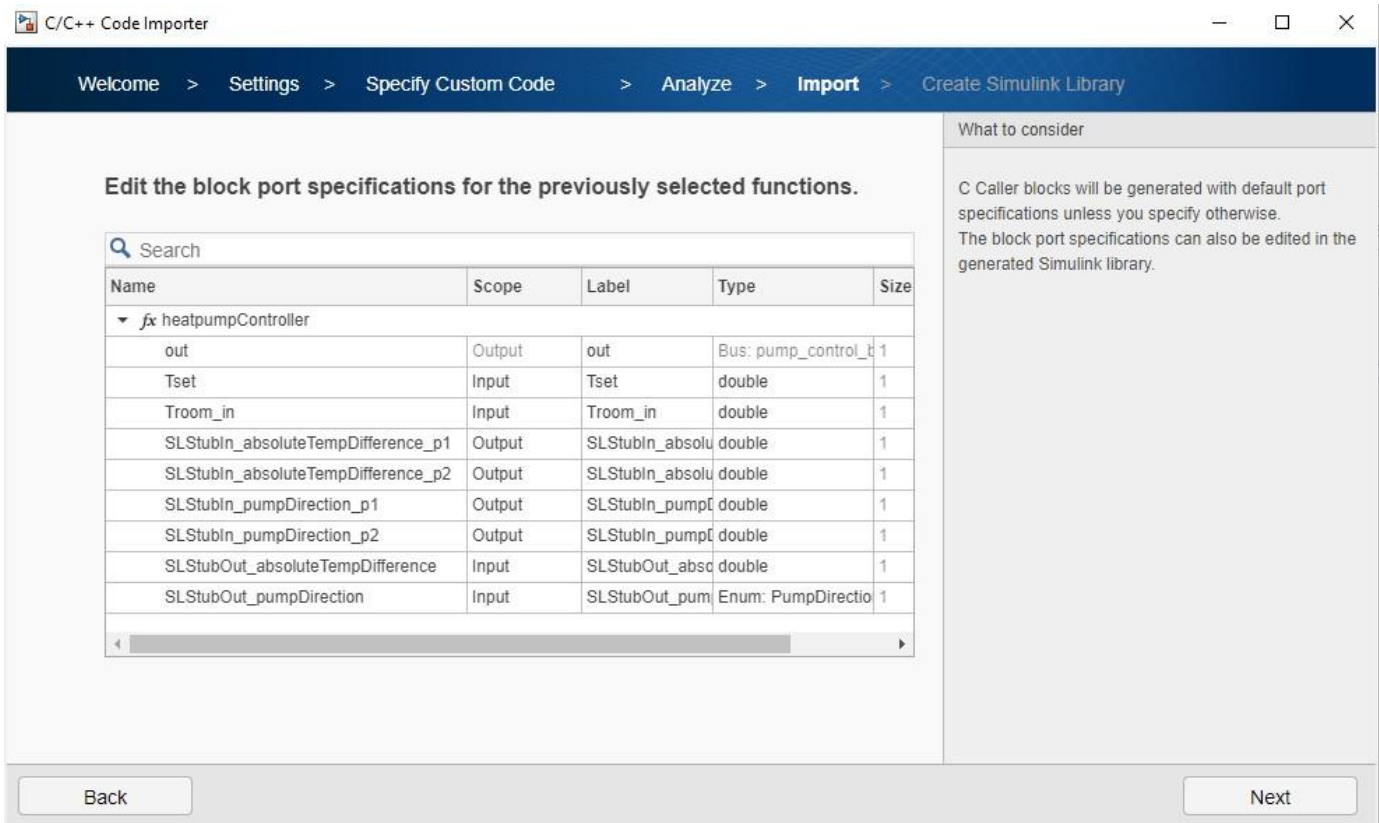
On the **Import** page, select the `heatpumpController` function to import into the Simulink library.



Click **Next**.

Set the Block Port Specifications

For the function you selected on the previous page, the wizard generates a function port specification. The selected ports are used for the generated C Caller block. Note that if the code to import had a function with a pointer output, you would need to specify the size of the output port on this page of the wizard.



In this example, the port specification table lists the formal arguments, `Tset`, `Troom_in`, and `out`, for the `heatpumpController` function, and six more from the automatically generated stubs. Because you selected **Enable global variables as function interface**, the wizard creates ports that are generated from the stubs for the `absoluteTempDifference` and `pumpDirection` functions. The stubs are in `auto_stub.c`.

These are the automatically generated global variables in the `auto_stub.c` file:

```

/*****
/* Generated Global Variables for Stubbed Functions Interface */
/*****
double SLStubIn_absoluteTempDifference_p1;
double SLStubIn_absoluteTempDifference_p2;
double SLStubOut_absoluteTempDifference;
double SLStubIn_pumpDirection_p1;
double SLStubIn_pumpDirection_p2;
PumpDirection SLStubOut_pumpDirection;

double absoluteTempDifference(double absoluteTempDifference_p1, double absoluteTempDifference_p2)
{
    SLStubIn_absoluteTempDifference_p1 = absoluteTempDifference_p1;
    SLStubIn_absoluteTempDifference_p2 = absoluteTempDifference_p2;
    return SLStubOut_absoluteTempDifference;
}

PumpDirection pumpDirection(double pumpDirection_p1, double pumpDirection_p2)
{

```

```

    SLStubIn_pumpDirection_p1 = pumpDirection_p1;
    SLStubIn_pumpDirection_p2 = pumpDirection_p2;
    return SLStubOut_pumpDirection;
}

```

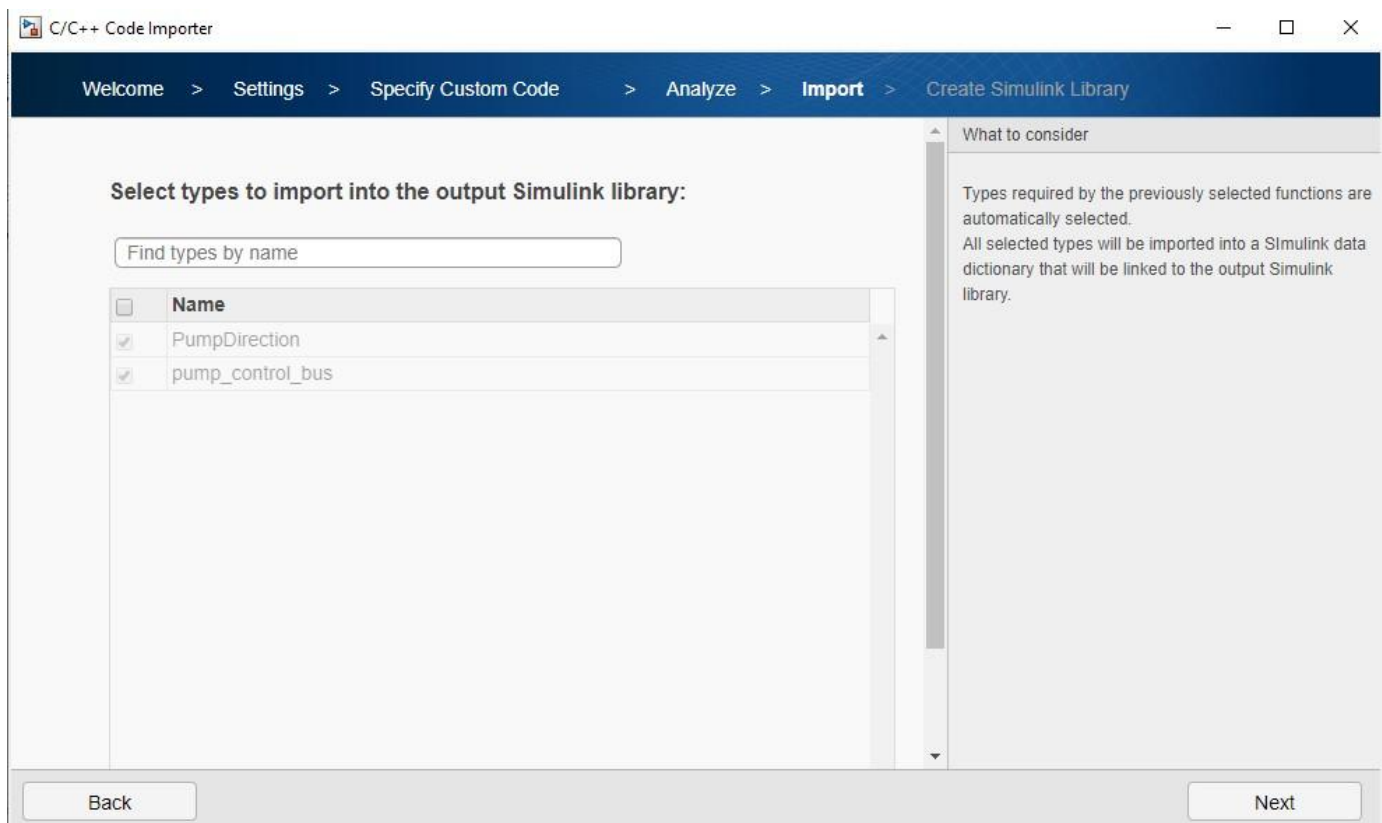
In the automatically generated stubs for `absoluteTempDifference`, the global variables `SLStubIn_absoluteTempDifference_p1` and `SLStubIn_absoluteTempDifference_p2` save the input arguments. The function returns the value stored in `SLStubOut_absoluteTempDifference` as its output. Similarly, `pumpDirection` saves the input arguments and returns `SLStubOut_pumpDirection`.

In this example, `absoluteTempDifference`, `SLStubIn_absoluteTempDifference_p1` and `SLStubIn_absoluteTempDifference_p2` are outputs. The global variable `SLStubOut_absoluteTempDifference` is an input.

Do not make any changes to the port specification. Click **Next**.

Specify Types to Import

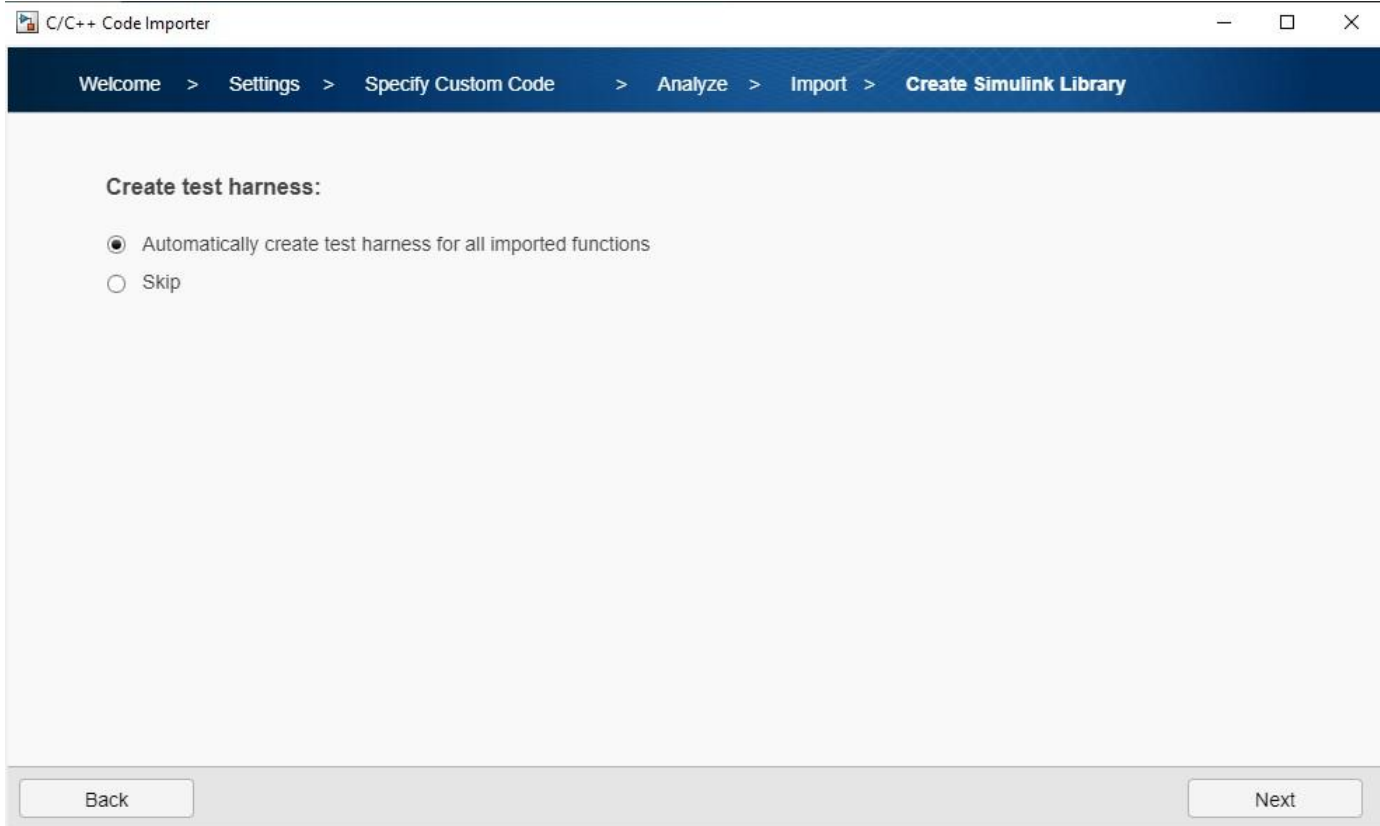
Select the types to import into Simulink. Because the `pump_control_bus` and `PumpDirection` types are required by the `heatpumpController` function, they are selected and dimmed. The wizard creates a Simulink data dictionary containing these types and links the dictionary to the generated library.



Click **Next** to display a summary of the generated library. Click **Next** again to continue.

Create a Test Harness

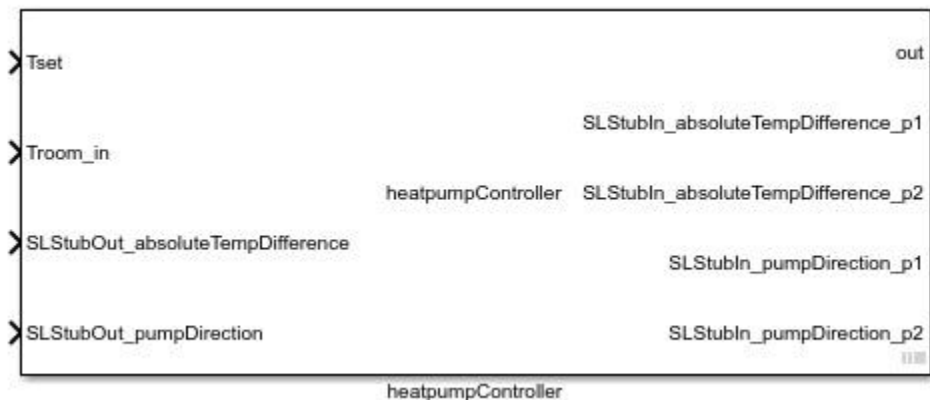
Select **Automatically create test harness for all imported functions**.



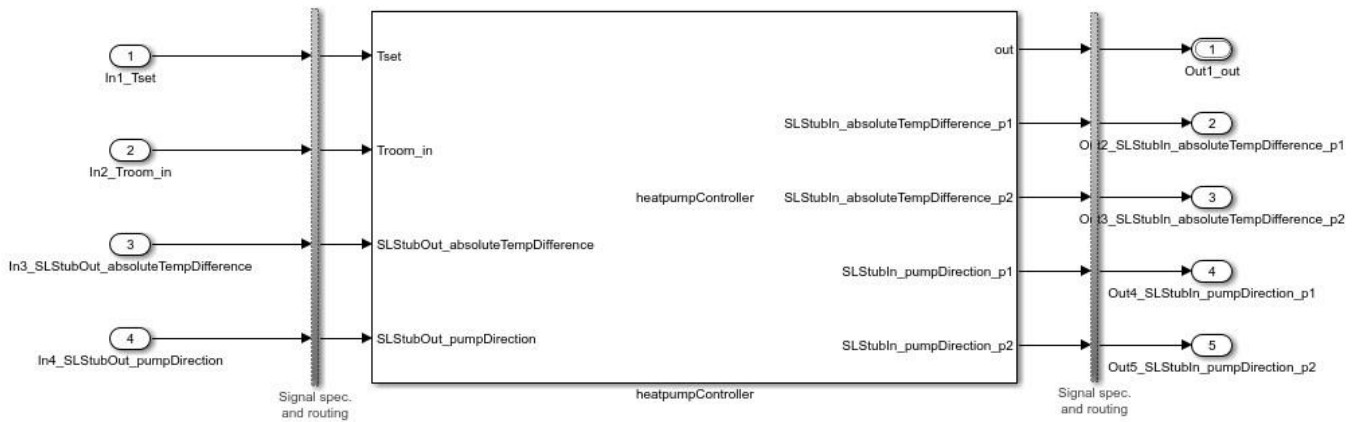
Click **Next** to generate the Simulink library.

After the code imports, the wizard creates a library attached to a Simulink data dictionary that defines `pump_control_bus` and `PumpDirection` as a `Simulink.Bus` object and a Simulink enumeration signal, respectively.

The C Caller block created in the Simulink library is:

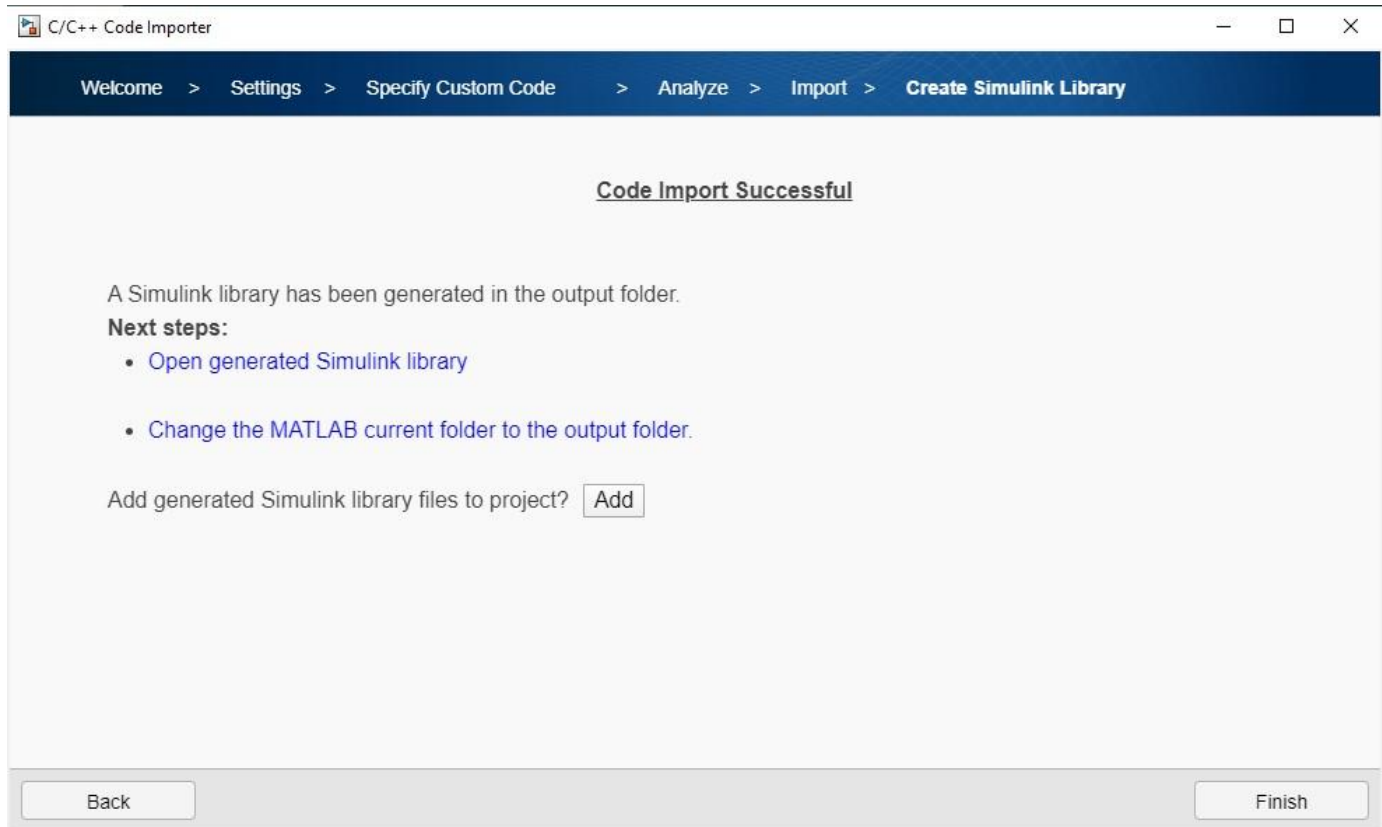


Click in the lower right corner of the block to open its internal test harness:



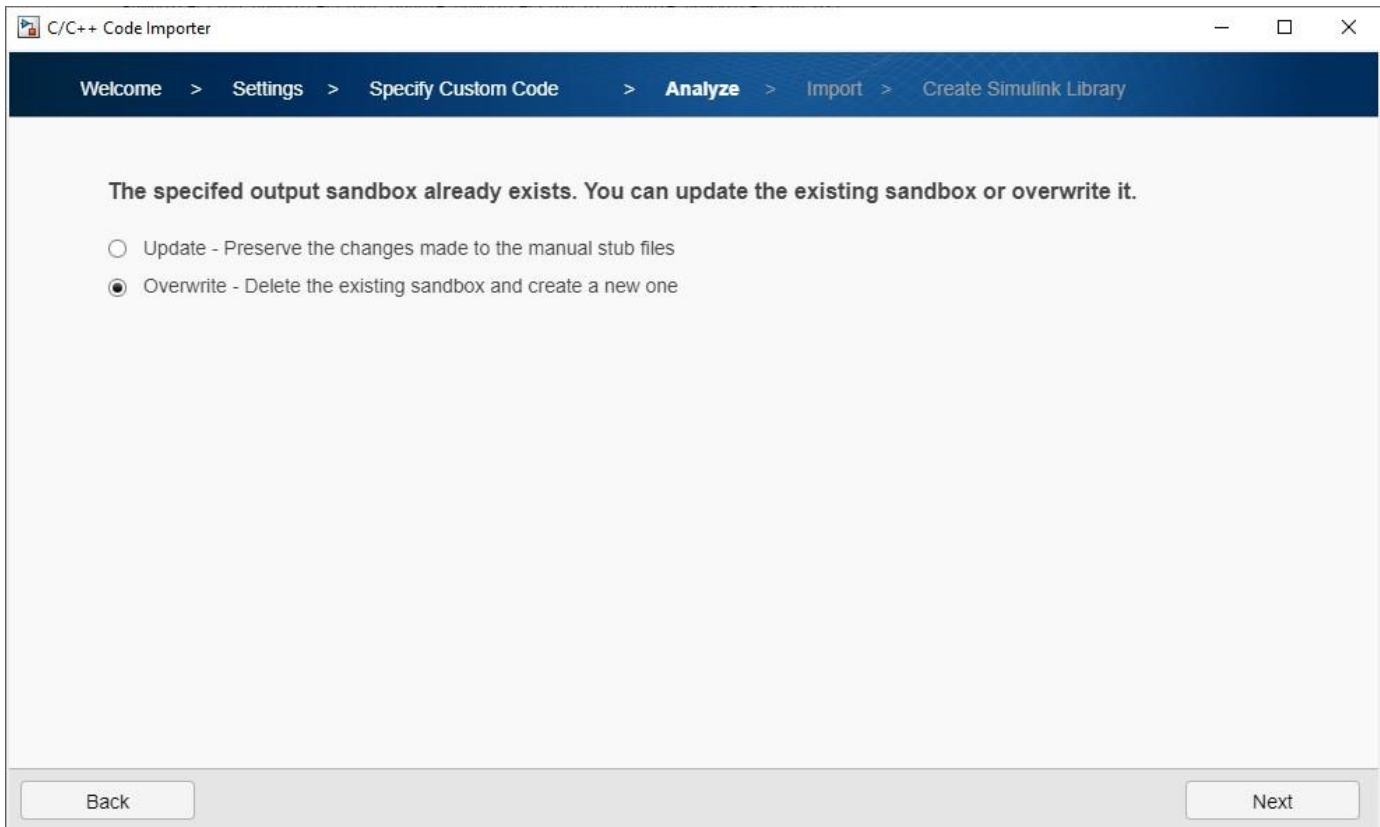
On the Code Import Successful page, click **Change the MATLAB folder to the output folder**.

Do not click **Finish**. Continue to the following section to manually update the stubs for undefined symbols.



Update Test Sandbox with Manual Stubbing

To manually create the stub files for `absoluteTempDifference` and `pumpDirection`, click the **Analyze** tab in the wizard toolbar and then click **Next** twice to display this page:



The Code Importer detects the sandbox you created. Select **Overwrite** and then, click **Next**.

Outside of the wizard, but leaving the wizard open, copy the files in the `updated_manualstub` directory of the sandbox and paste them into the `heatpumpController/manualstub` directory.

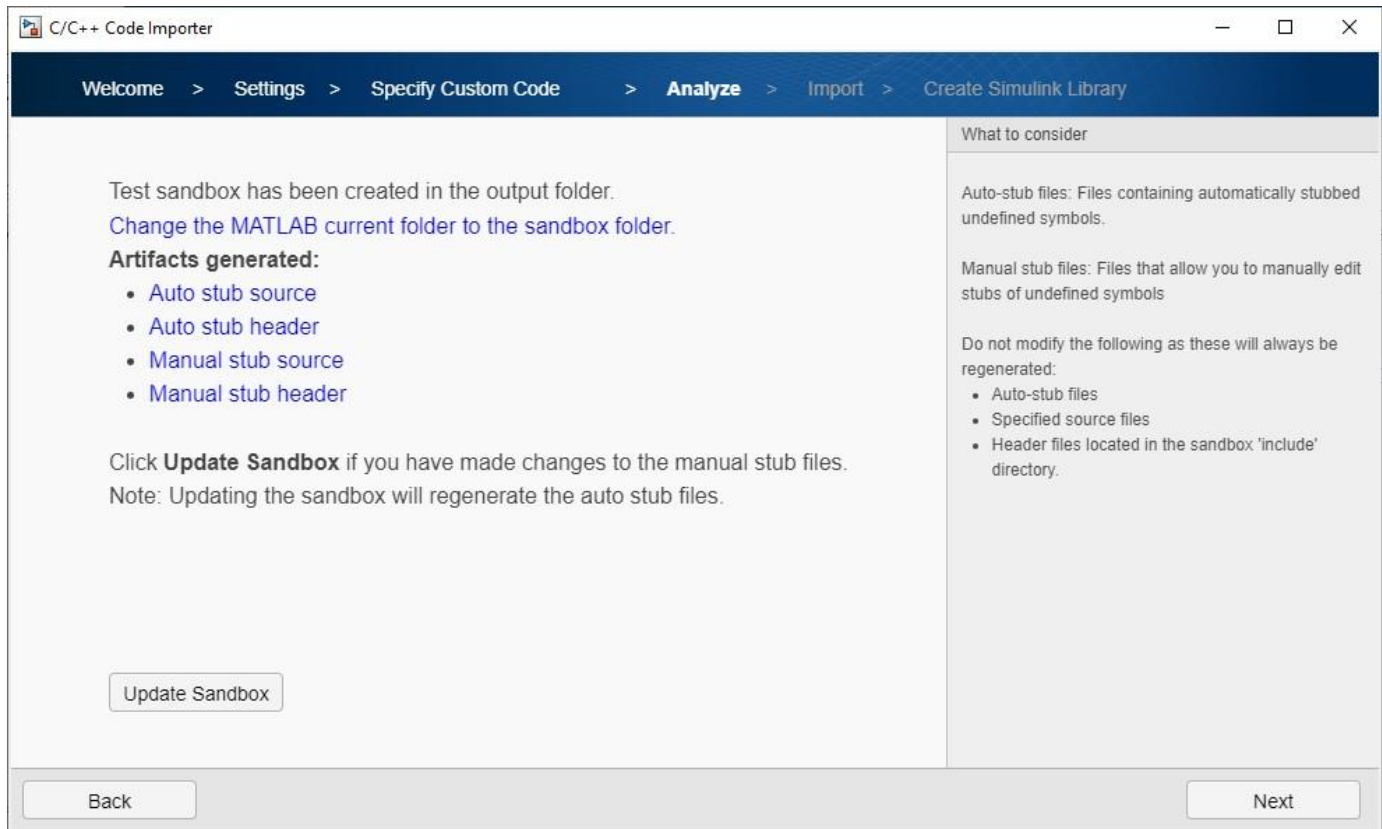
In the `man_stub.c` file, edit the function definition of `absoluteTempDifference`:

```
double absoluteTempDifference(double Tset, double Troom_in){
    return (double)fabs(Tset - Troom_in);
}
```

and the function definition of `pumpDirection`:

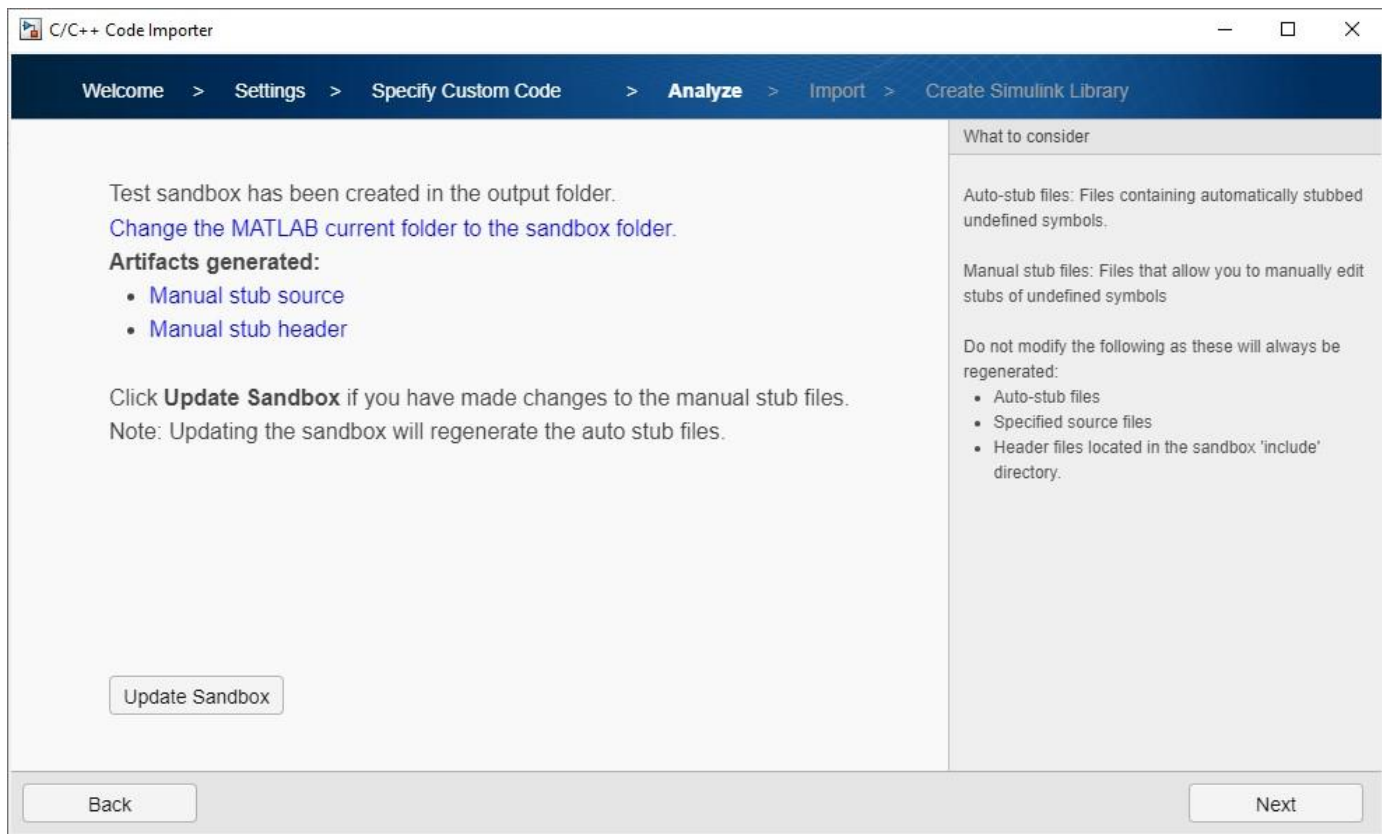
```
PumpDirection pumpDirection(double Tset, double Troom_in){
    return Tset > Troom_in ? HEATING : COOLING;
}
```

Return to the Code Importer wizard.

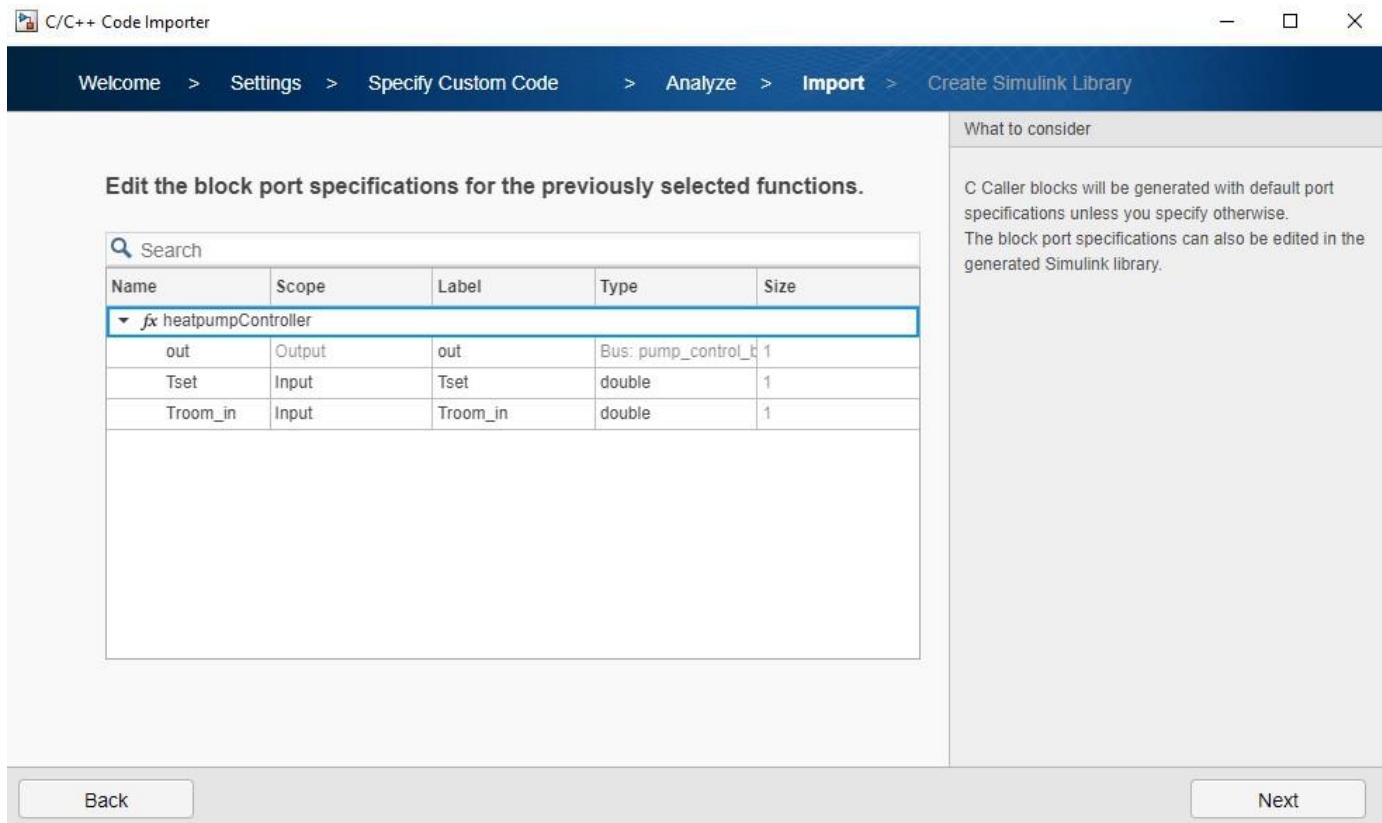


After modifying the manual stub files, return to the wizard and click **Update Sandbox**.

Because you manually defined the definitions for the `absoluteTempDifference` and `pumpDirection` functions, there are no symbols to automatically stub, no artifacts are generated for the `autostub` directory in the sandbox and it is empty.



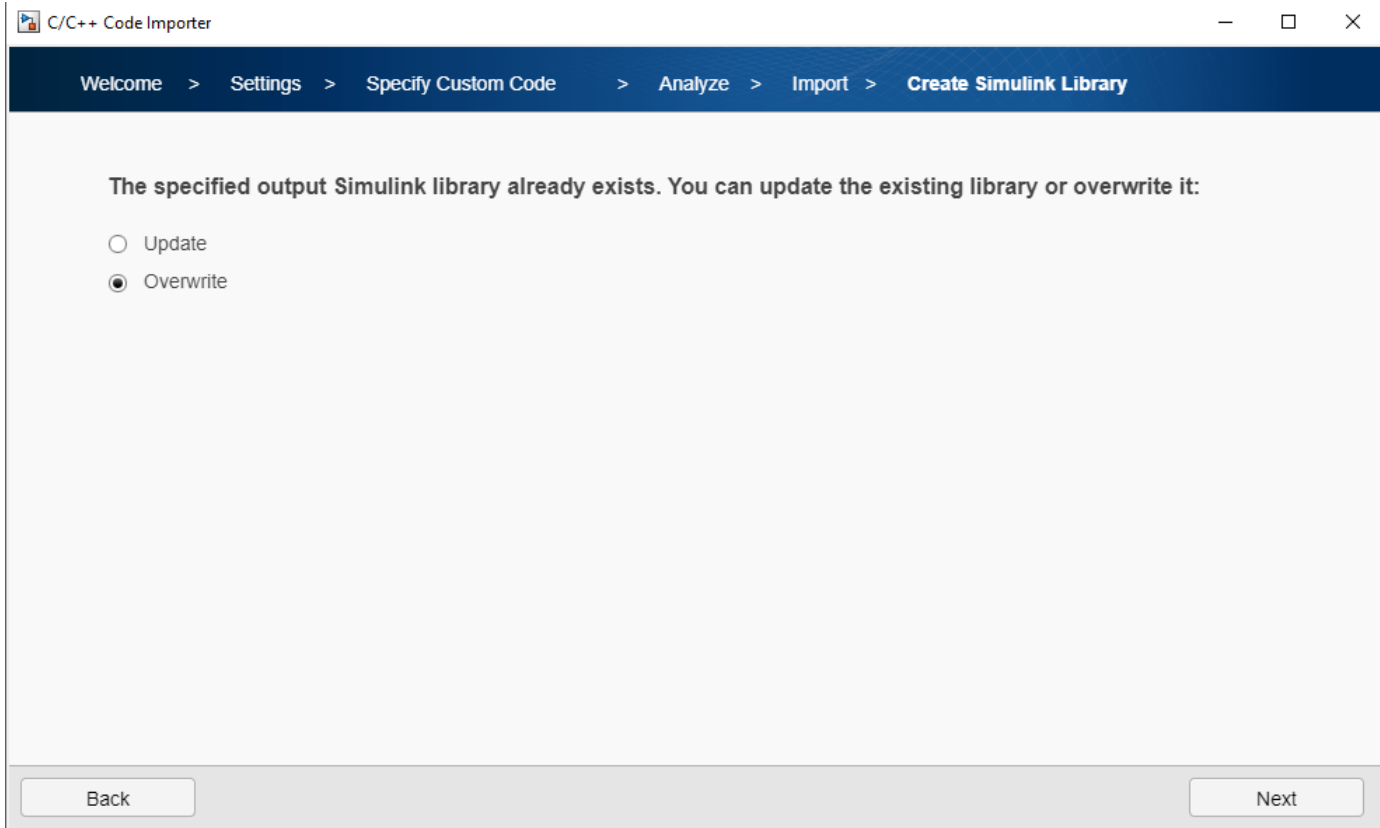
Click **Next** until you reach the block port specification page:



Because the manual implementation of `absoluteTempDifference` and `pumpDirection` does not have any global variables, only the formal arguments and the return argument appear as ports.

Click **Next**.

On the **Create Simulink Library** tab, select **Overwrite**.



Click **Next** through the remaining pages until you reach the **Code Import Successful** page.

Then click **Finish**. If desired, click **Yes** in the dialog box that opens to save the current import settings as a JSON file, which you use to reload the settings into another Code Importer wizard session.





Test the Imported Code

In the Test Manager, in the **Test Browser** pane expand the **heatpumpController** test file and **heatpumpController/heatpumpController** test suite. Then select the **heatpumpController_Harness1** test case.



Specify Simulation Stop Time

In the **System Under Test** section, expand the **Simulation Setting and Release Overrides** section and set the **Stop Time** to 200.


SYSTEM UNDER TEST*

Model:    


TEST HARNESS*

Harness:  

SIMULATION SETTINGS AND RELEASE OVERRIDES*

Simulation Mode: 

Override model blocks in SIL/PIL mode to normal mode

Select releases for simulation: 

Start Time:

Stop Time:

Initial State:


Add Inputs

In the **Inputs** section, at the bottom of the **External Inputs** table, click **Add** to open the Add Input dialog box.

In the **Input File Specification**, enter `heatpumpControllerHarnessInput.xlsx`.

Under **Input Mapping**, select `Block Name` as the **Mapping Mode** and click **Map Inputs**. After the inputs appear in the Mapping Status table, click **OK**.


INPUT FILE SPECIFICATION

File: 

Add iterations to run this input

▶ SHEETS AND/OR RANGE SPECIFICATION





▼ INPUT MAPPING

Mapping Mode: 

Compile the system under test

▼ MAPPING STATUS

Some inports were not assigned any values during mapping and have been set to ground values to enable the model to simulate

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	heatpumpController_Harness1/In...		
2	heatpumpController_Harness1/In...		
3	heatpumpController_Harness1/In...		
4	heatpumpController_Harness1/In...		

Add Assessments

In the **Logical and Temporal Assessments** section, add assessments for each temperature condition:

EN...	NAME	ASSESSMENT	REQUIREMENTS
<input checked="" type="checkbox"/>	Assessment1	▶ At any point of time, whenever $\text{abs}(\text{Troom_in} - \text{Tset}) < 2$ is true then, with no delay, $\text{fan_cmd} == \text{uint8}(0) \ \& \ \text{pump_cmd} == \text{uint8}(0) \ \& \ \text{pump_dir} == \text{IDLE}$ must be true	None
<input checked="" type="checkbox"/>	Assessment2	▶ At any point of time, whenever $\text{abs}(\text{Troom_in} - \text{Tset}) \geq 2 \ \& \ \text{abs}(\text{Troom_in} - \text{Tset}) < 4$ is true then, with no delay, $\text{fan_cmd} == \text{uint8}(1) \ \& \ \text{pump_cmd} == \text{uint8}(0) \ \& \ \text{pump_dir} == \text{IDLE}$ must be true	None
<input checked="" type="checkbox"/>	Assessment3	▶ At any point of time, whenever $\text{abs}(\text{Troom_in} - \text{Tset}) \geq 4 \ \& \ \text{Tset} < \text{Troom_in}$ is true then, with no delay, $\text{fan_cmd} == \text{uint8}(1) \ \& \ \text{pump_cmd} == \text{uint8}(1) \ \& \ \text{pump_dir} == \text{COOLING}$ must be true	None
<input checked="" type="checkbox"/>	Assessment4	▶ At any point of time, whenever $\text{abs}(\text{Troom_in} - \text{Tset}) \geq 4 \ \& \ \text{Tset} > \text{Troom_in}$ is true then, with no delay, $\text{fan_cmd} == \text{uint8}(1) \ \& \ \text{pump_cmd} == \text{uint8}(1) \ \& \ \text{pump_dir} == \text{HEATING}$ must be true	None

For information on assessments, see “Assess Temporal Logic by Using Temporal Assessments” on page 3-93.

In the Symbols pane, add the symbols definitions:

- ▼  Troom_in
 - Name: Input Conversion Subsystem:2
 - Path: heatpumpController_Harness1/Input Conversion Subsystem
 - Port Index: 2
 - Field/Element: <type an expression>

 - ▼  Tset
 - Name: Input Conversion Subsystem:1
 - Path: heatpumpController_Harness1/Input Conversion Subsystem
 - Port Index: 1
 - Field/Element: <type an expression>

 - ▼  fan_cmd
 - Name: heatpumpController:1
 - Path: heatpumpController_Harness1/heatpumpController
 - Port Index: 1
 - Field/Element: .fan_cmd

 - ▼  pump_cmd
 - Name: heatpumpController:1
 - Path: heatpumpController_Harness1/heatpumpController
 - Port Index: 1
 - Field/Element: .pump_cmd

 - ▼  pump_dir
 - Name: heatpumpController:1
 - Path: heatpumpController_Harness1/heatpumpController
 - Port Index: 1
 - Field/Element: .pump_dir

 - ▼  HEATING
 - Expression: PumpDirection.HEATING

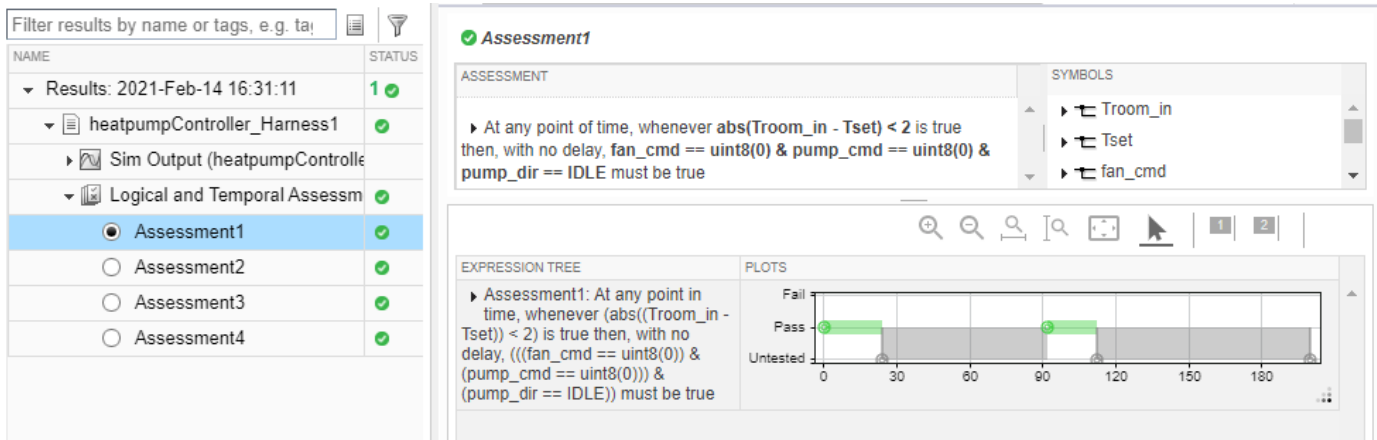
 - ▼  COOLING
 - Expression: PumpDirection.COOLING

 - ▼  IDLE
 - Expression: PumpDirection.IDLE
-

Run the Test and View Results

Click **Run** to run the test.

When the test completes, in the **Results and Artifacts** pane, expand the **Results**. All of the assessments pass.



To view the coverage results, select **heatpumpController_Harness1** under **Results** and expand the **Coverage Results** section. The coverage is 100% for both the `tempController.c` and `man_stub.c` files.

▼ COVERAGE RESULTS

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	EXECUTION
man_stub.c	📄	3	100% 	100%
tempController.c	📄	3	100% 	100%

See Also

More About

- “Importing and Testing Custom C/C++ Code” on page 9-2

Verification and Validation

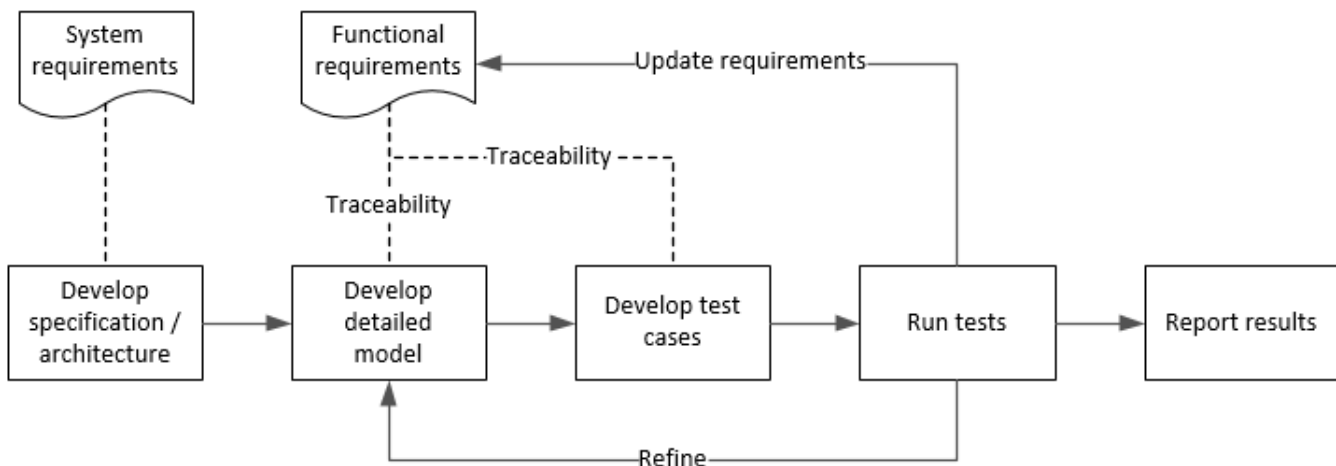
- “Test Model Against Requirements and Report Results” on page 10-2
- “Analyze Models for Standards Compliance and Design Errors” on page 10-7
- “Perform Functional Testing and Analyze Test Coverage” on page 10-9
- “Analyze Code and Test Software-in-the-Loop” on page 10-12

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

- 1 Open the example project.

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

- 2 In the `models` folder, open the `simulinkCruiseAddReqExample` 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 Display the verification and implementation status. Right-click a requirement and select **Verification Status** and **Implementation Status**.

The screenshot displays the Simulink environment with a model diagram on the left, a Requirements table in the center, and a Property Inspector on the right.

Model Diagram: Shows a block named "Compute target speed" with inputs: CruiseOnOff (boolean), Brake (boolean), Speed (single), CoastSetSw (boolean), and AccelResSw (boolean). It has outputs: engaged (boolean) and tspeed (single).

Requirements Table:

Index	ID	Summary	Verified	Implemented
simulinkCruiseChar...				
1	Architecture	Architecture		
1.1	A 1.1	Enable Disable Switch		
1.2	A 1.2	Set Speed / Decelerate Bu...		
1.3	A 1.3	Resume Speed / Accelerat...		
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		

Property Inspector (Requirement: A 1.2):

- Type: Functional
- Index: 1.2
- Custom ID: A 1.2
- Summary: Set Speed / Decelerate Button
- 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)**
- Keywords:
- Revision information:
- Links: Implemented by: CoastSetSw
- Comments:

- 5 In the Project window, open the Simulink Test file `s1ReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

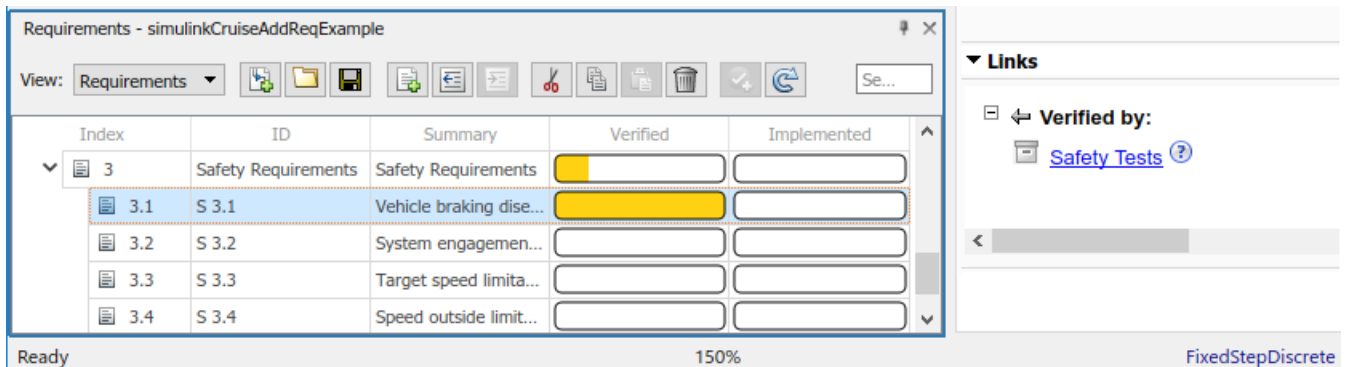
Link Requirements to Tests

Link the requirements to the test case.

- 1 In the Project window, open the Simulink Test file `s1ReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager. Explore the test suite and select `Safety Tests`.

Return to the model. Right-click on requirement `S 3.1` and select **Link from Selected Test Case**.

A link to the `Safety Tests` test case is added to **Verified by**. The yellow bars in the **Verified** column indicate that the requirements are not verified.



- 2 Also add a link for item S 3.4.

Run the Test

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:

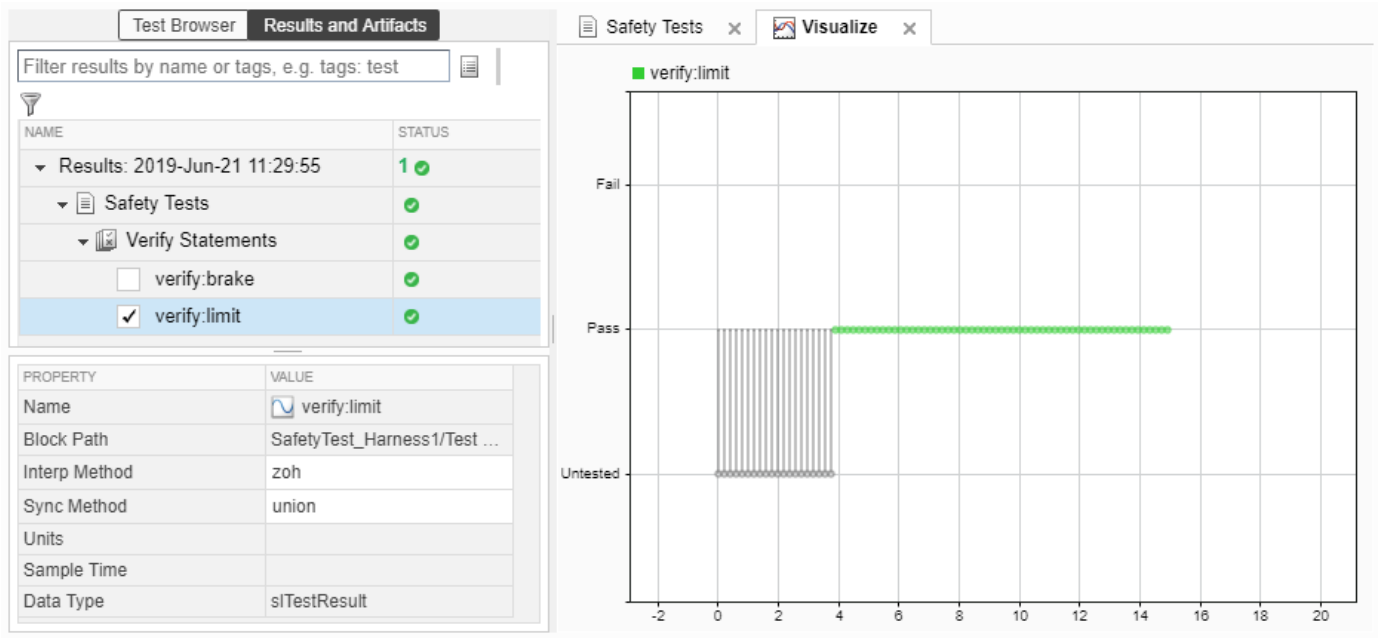
- The `BrakeTest` sequence engages the cruise control, then applies the brake. It includes the `verify` statement

```
verify(engaged == false,...
      'verify:brake',...
      'system must disengage when brake applied')
```

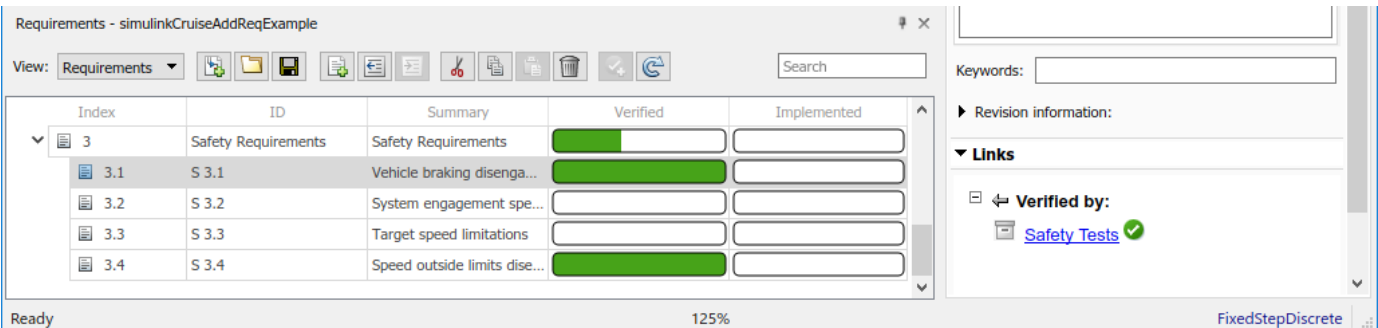
- The `LimitTest` sequence engages the cruise control, then ramps up the vehicle speed until it exceeds the upper limit. It includes the `verify` statement.

```
verify(engaged == false,...
      'verify:limit',...
      'system must disengage when limit exceeded')
```

- 1 Return to the Test Manager. To run the test case, click **Run**.
- 2 When the test finishes, review the results. The Test Manager shows that both assessments pass and the plot provides the detailed results of each `verify` statement.



- 3 Return to the model and refresh the Requirements. The green bar in the **Verified** column indicates that the requirement has been successfully verified.



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 Enter a file name and select a location for the report.
 - d For the **Template File**, select the ReportTemplate.dotx file in the **documents** project folder.
 - e Click **Create**.

- 2 Review the report.
 - a The **Test Case Requirements** section specifies the associated requirements
 - b The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

See Also

Related Examples

- “Link to Requirements” on page 1-2
- “Validate Requirements Links in a Model” (Requirements Toolbox)
- “Customize Requirements Traceability Report for Model” (Requirements Toolbox)

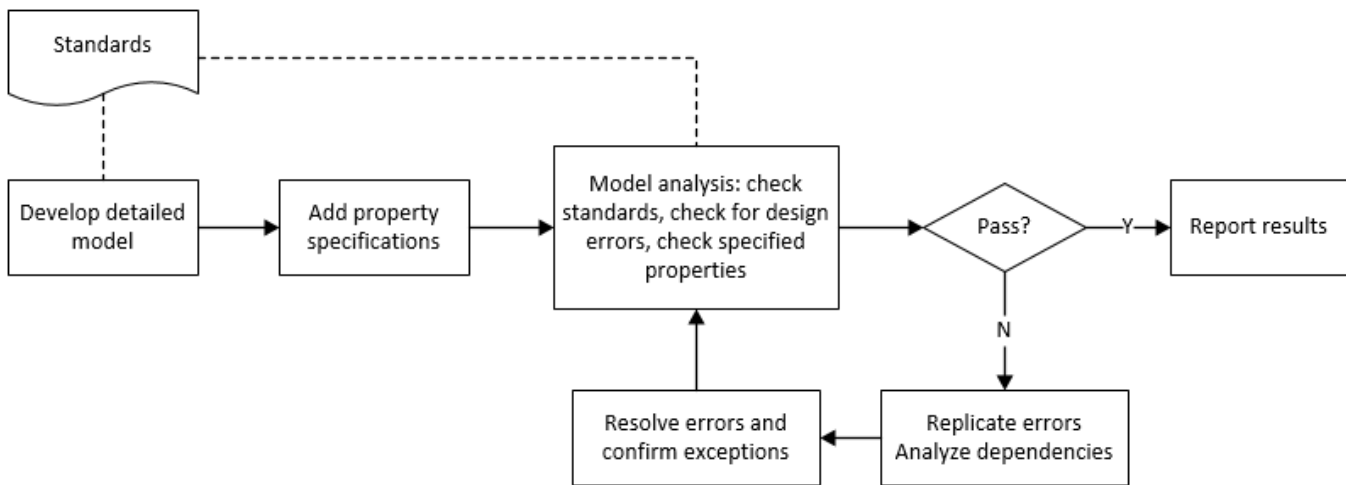
External Websites

- Requirements-Based Testing Workflow

Analyze Models 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 MAB 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® Advisory Board (MAB) 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 MAB Style Guideline Violations

Check that your model complies with MAB guidelines by using the Model Advisor.

- 1 Open the example project. On the command line, enter


```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```
- 2 Open the `simulinkCruiseErrorAndStandardsExample` model.


```
open_system simulinkCruiseErrorAndStandardsExample
```
- 3 In the **Modeling** tab, select **Model Advisor**.
- 4 Click **OK** to select `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
- 5 Check your model for MAB style guideline violations using Simulink Check.
 - a In the left pane, in the **By Product > Simulink Check > Modeling Standards > MAB Checks** folder, select:

- **Check Indexing Mode**
 - **Check model diagnostic parameters**
- b** Right-click on the **MAB Checks** node and select **Run Checks**.
 - c** To review the configuration parameter settings that violate MAB style guidelines, run the **Check model diagnostic parameters** check.
 - d** The analysis results appear in the right pane on the **Report** tab. Report displays the violation details and the recommended action.
 - e** Click the parameter hyperlinks, which opens the Configuration Parameters dialog box, and update the model diagnostic parameters. Save the model.
 - f** To verify that your model passes, rerun the check. Repeat steps from c to e, if necessary, to reach compliance.
 - g** To generate a results report of the Simulink Check checks, select the **MAB Checks** node, and then, from the toolbar, click **Report**.

Check Model for Design Errors

While in the Model Advisor, you can also check your model for hidden design errors using Simulink Design Verifier.

- 1** In the left pane, in the **By Products > Simulink Design Verifier** folder, select **Design Error Detection**.
- 2** If not already checked, click the box beside **Design Error Detection**. All checks in the folder are selected.
- 3** From the tool strip, click **Run Checks**.
- 4** After the Model Advisor analysis, from the tool strip, click **Report**. This generates a HTML report of the check analysis.
- 5** In the generated report, click a **Simulink Design Verifier Results Summary** hyperlink. The dialog box provides tools to help you diagnose errors and warnings in your model.
 - a** Review the analysis results on the model. Click the **Compute target speed** subsystem. 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 or create one if it does not already exist.
 - c** View tests and export test cases.
 - d** Review the analysis report. To see a detailed analysis report, click **HTML** or **PDF**.

See Also

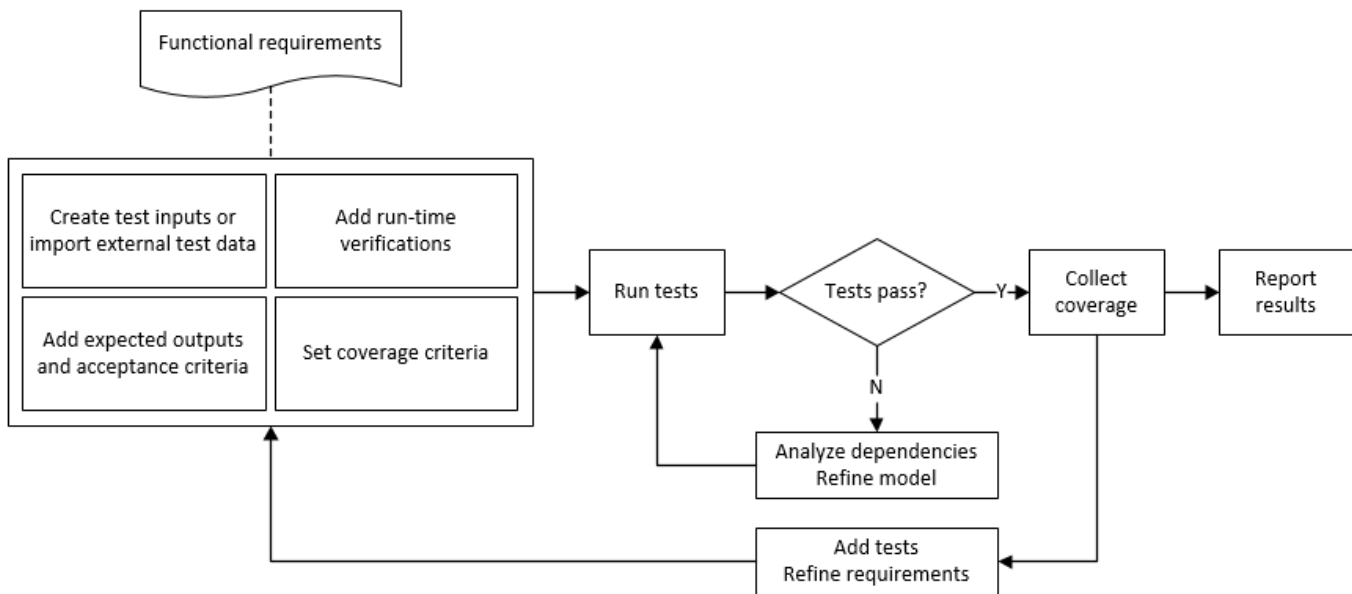
Related Examples

- “Check Model Compliance by Using the Model Advisor” (Simulink Check)
- “Collect Model Metrics Using the Model Advisor” (Simulink Check)
- “Analyze Models for Design Errors” (Simulink Design Verifier)
- “Prove Properties in a Model” (Simulink Design Verifier)

Perform Functional Testing and Analyze Test Coverage

Functional testing begins with building test cases based on requirements. These tests can cover key aspects of your design and verify that individual model components meet requirements. Test cases include inputs, expected outputs, and acceptance criteria.

By collecting individual test cases within test suites, you can run functional tests systematically. To check for regression, add baseline criteria to the test cases and test the model iteratively. Coverage measurement reflects the extent to which these tests have fully exercised the model. Coverage measurement also helps you to add tests and requirements to meet coverage targets.



Incrementally Increase Test Coverage Using Test Case Generation

This example shows how to perform requirements-based tests for a cruise control model. The tests link to a requirements document. You:

- 1 Run the tests.
- 2 Determine test coverage by using Simulink Coverage.
- 3 Increase coverage with additional tests generated by Simulink Design Verifier.
- 4 Report the results.

Open the Test Harness and Model

- 1 Open the project:

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

- 2 Open the model and the test harness. At the command line, enter:

```
open_system simulinkCruiseAddReqExample
sltest.harness.open("simulinkCruiseAddReqExample", "SafetyTest_Harness1")
```

- 3 Load the test suite from “Test Model Against Requirements and Report Results” on page 10-2 and open the Simulink Test Manager.

```
pf = fullfile(pr.RootFolder,"tests","slReqTests.mldatx");
tf = sltest.testmanager.TestFile(pf);
sltest.testmanager.view
```

- 4 Open the Test Sequence block. The sequence verifies system disengagement when either:
 - The brake pedal is pressed.
 - Speed exceeds a limit.

Measure Model Coverage

- 1 In the Simulink Test Manager, select the `slReqTests` test file.
- 2 To enable coverage collection, in the right page under **Coverage Settings**:
 - Select **Record coverage for referenced models**.
 - Specify a coverage filter by using **Coverage filter filename**.
 - Select **Decision**, **Condition**, and **MCDC**.
- 3 Click **Run** on the Test Manager toolstrip.
- 4 After the test completes, select **Results**. The test achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.

ANALYZED MODEL	REPORT CO...	DECISION	CONDITION	MCDC
simulinkCruiseAddReqExample	31	50%	41%	25%

Generate Tests to Increase Model Coverage

- 1 Use Simulink Design Verifier to generate additional tests to increase model coverage. In **Results and Artifacts**, select the `slReqTests` test file and open the **Aggregated Coverage Results** section located in the right pane.
- 2 Right-click the test results and select **Add Tests for Missing Coverage**.
- 3 Under **Harness**, choose **Create a new harness**.
- 4 Click **OK** to add tests to the test suite using Simulink Design Verifier. The model being tested must either be on the MATLAB path or in the working folder.
- 5 On the Test Manager toolstrip, click **Run** to execute the updated test suite. The test results include coverage for the combined test case inputs, achieving increased model coverage.

Alternatively, you can create and use tests to increase coverage programmatically by using `sltest.testmanager.addTestsForMissingCoverage` and `sltest.testmanager.TestOptions`.

See Also

Related Examples

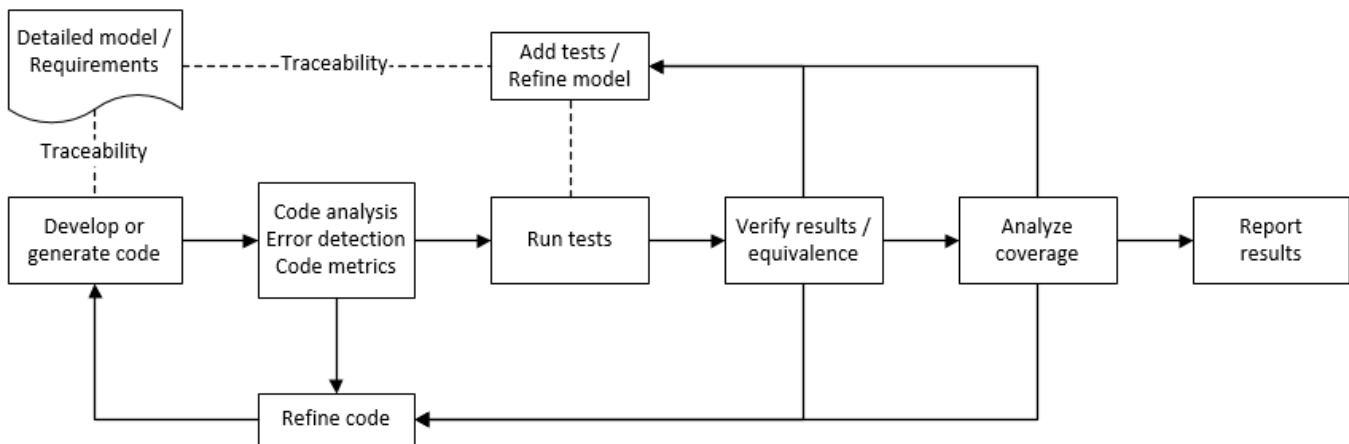
- “Link to Requirements” on page 1-2
- “Assess Model Simulation Using verify Statements” on page 3-18
- “Compare Model Output to Baseline Data” on page 6-7
- “Generate Test Cases for Model Decision Coverage” (Simulink Design Verifier)
- “Increase Test Coverage for a Model” on page 6-147

Analyze Code and Test Software-in-the-Loop

Code Analysis and Testing Software-in-the-Loop Overview

You can analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. For handwritten code, you typically 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, you refine the code and add tests.

In this example, you generate code and demonstrate that the code execution produces equivalent results to the model by using the same test cases and baseline results. Then you compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.



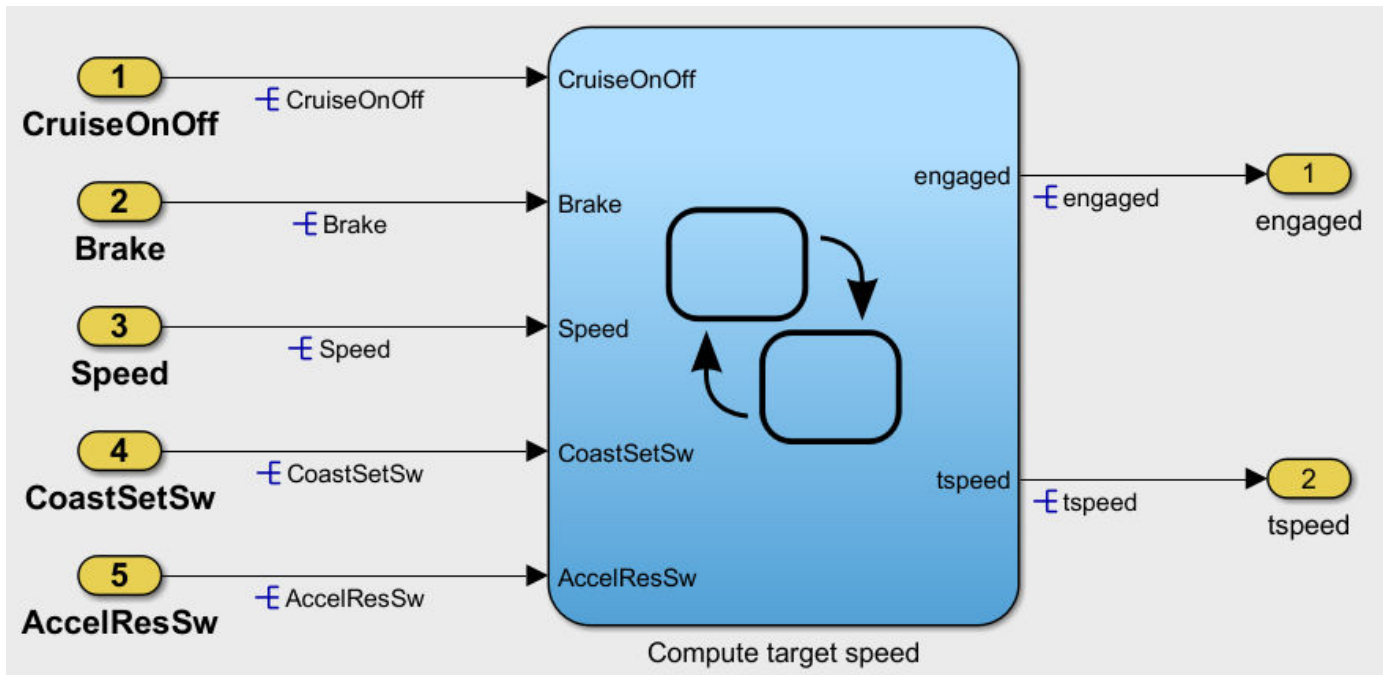
Analyze Code for Defects, Metrics, and MISRA C:2012

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 and defects. To produce more MISRA compliant code from your model, you use the code generation and Model Advisor. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

- 1 Open the project.

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

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

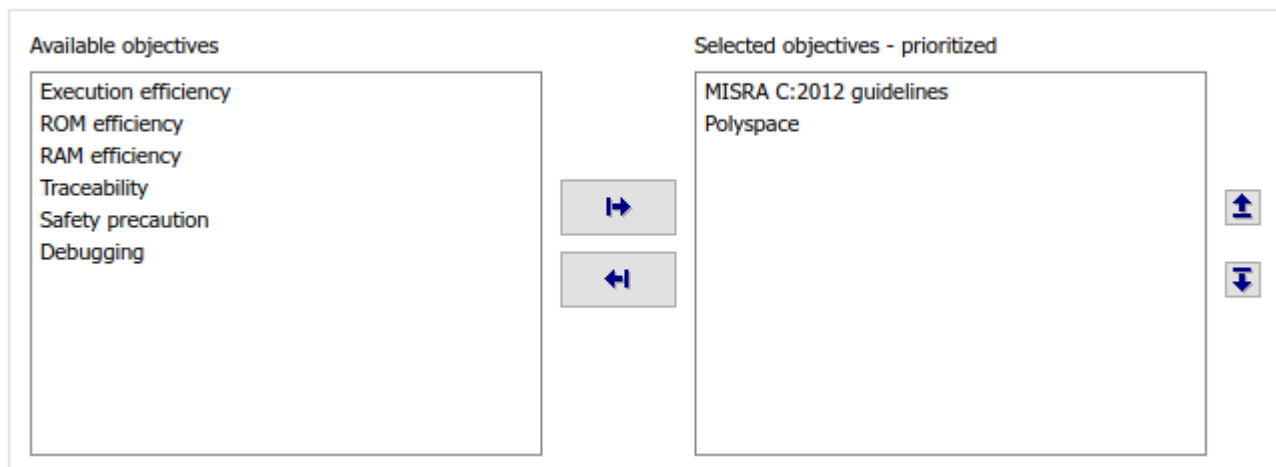


Run Code Generator Checks

Check your model by using the Code Generation Advisor. Configure code generation parameters to generate code more compliant with MISRA C and more compatible with Polyspace.

- 1 Right-click Compute target speed and select **C/C++ Code > Code Generation Advisor**.
- 2 Select the Code Generation Advisor folder. In the right pane, move Polyspace to **Selected objectives - prioritized**. The MISRA C:2012 guidelines objective is already selected.

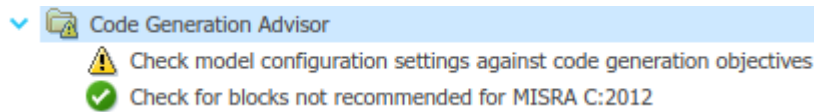
Code Generation Objectives (System target file: ert.tlc)



- 3 Click **Run Selected Checks**.

The Code Generation Advisor checks whether the model includes blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this

model, the check for incompatible blocks passes, but some configuration settings are incompatible with MISRA compliance and Polyspace checking.



- 4 Click the check that did not pass. Accept the parameter changes by selecting **Modify Parameters**.
- 5 Rerun the check by selecting **Run This Check**.

Run Model Advisor Checks

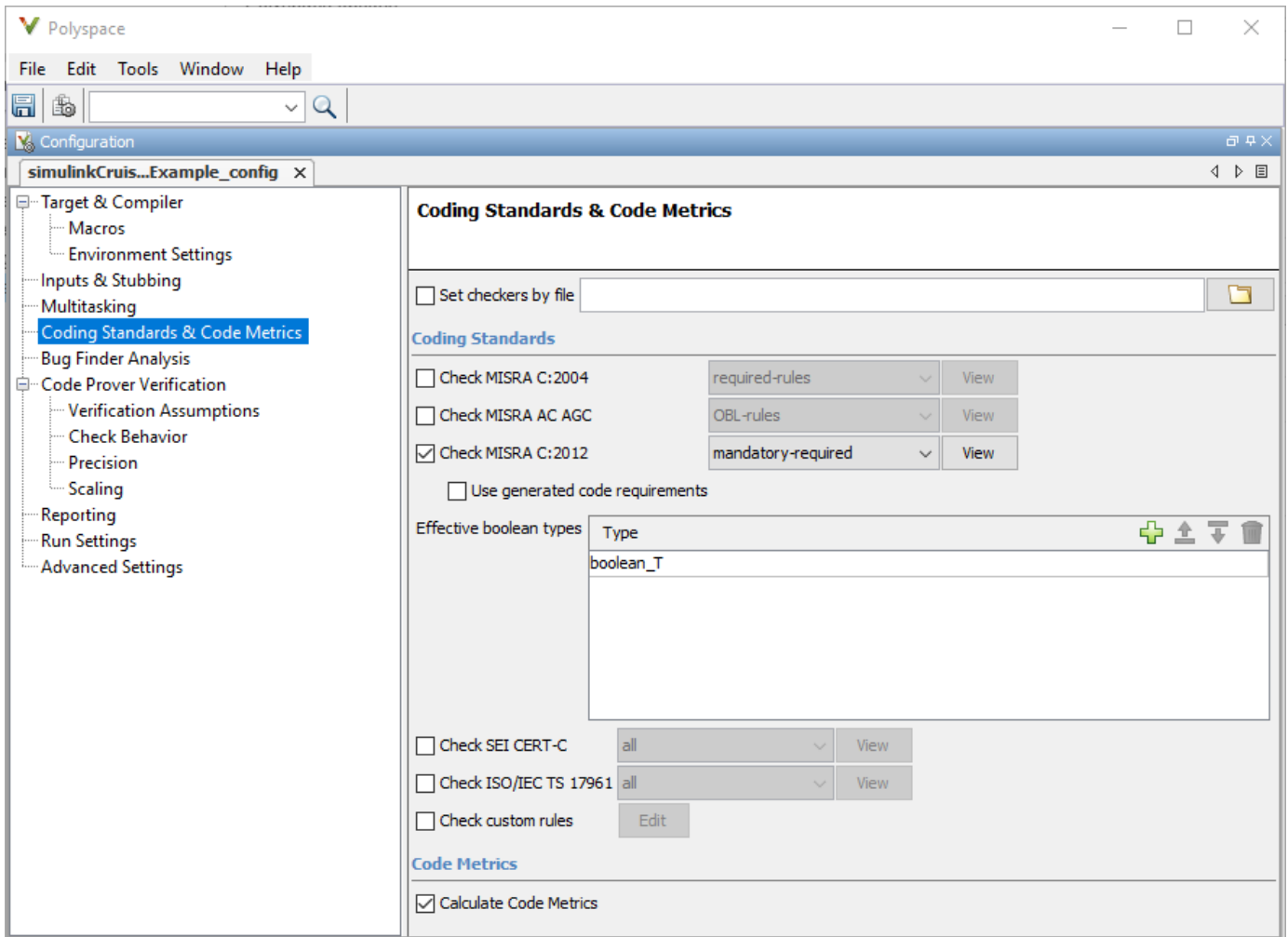
Before you generate code from your model, use the Model Advisor to check your model for MISRA C and Polyspace compliance. This example shows you how to use the Model Advisor to check your model before generating code.

- 1 At the bottom of the Code Generation Advisor window, select **Model Advisor**.
- 2 Under the **By Task** folder, select the **Modeling Standards for MISRA C:2012** advisor checks.
- 3 Click **Run 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 generate the 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++ Code > Build This Subsystem**.
- 2 Use the default settings for the tunable parameters and select **Build**.
- 3 After the code is generated, in the Simulink Editor, right-click Compute target speed and select **Polyspace > Options**.
- 4 Click **Configure** to choose more advanced Polyspace analysis options in the Polyspace configuration window.



- 5 On the left pane, click **Coding Standards & Code Metrics**, then select **Calculate Code Metrics** to enable 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**.

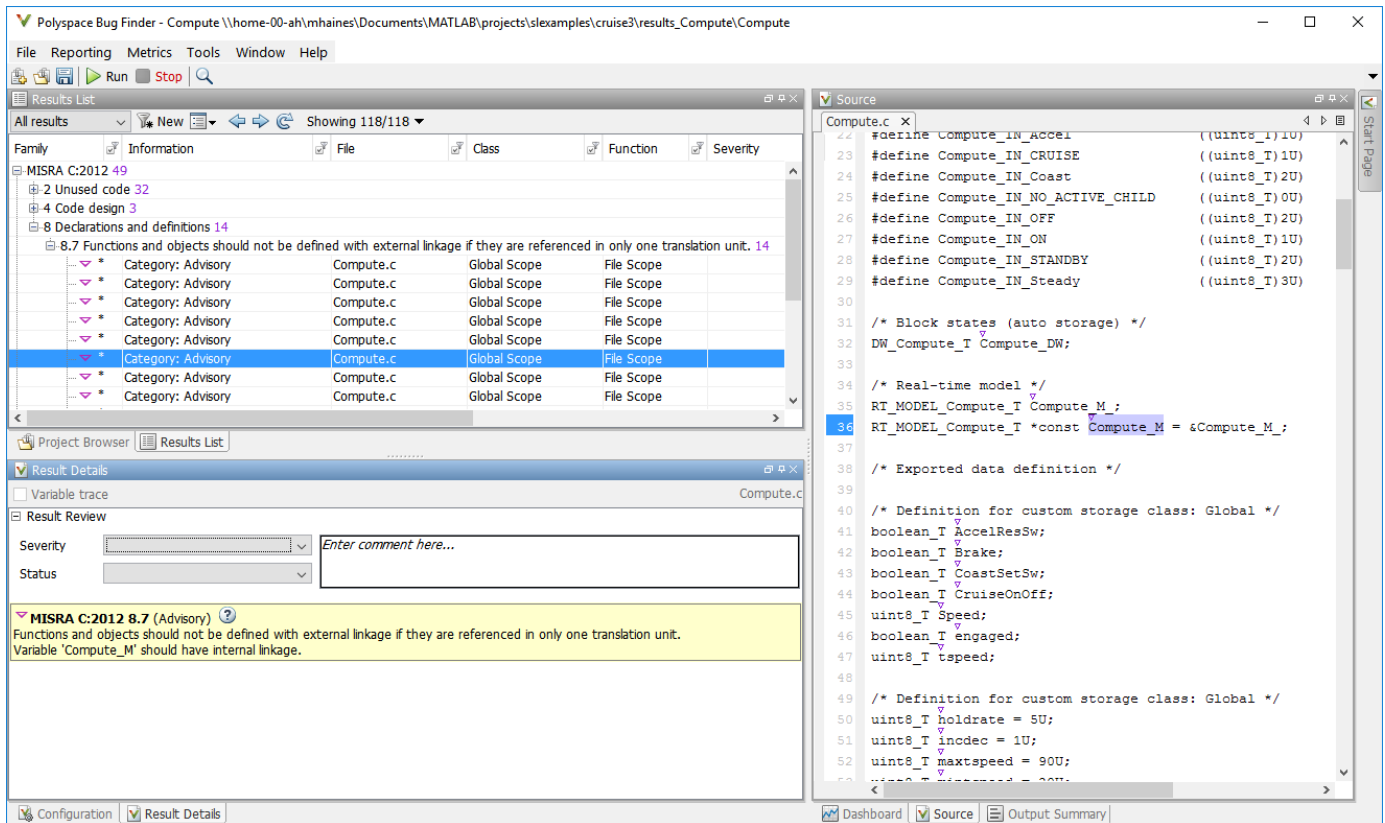
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks. You can see the progress of the analysis in the MATLAB Command Window. After the analysis finishes, the Polyspace environment opens.

Review Results

The Polyspace environment shows you the results of the static code analysis.

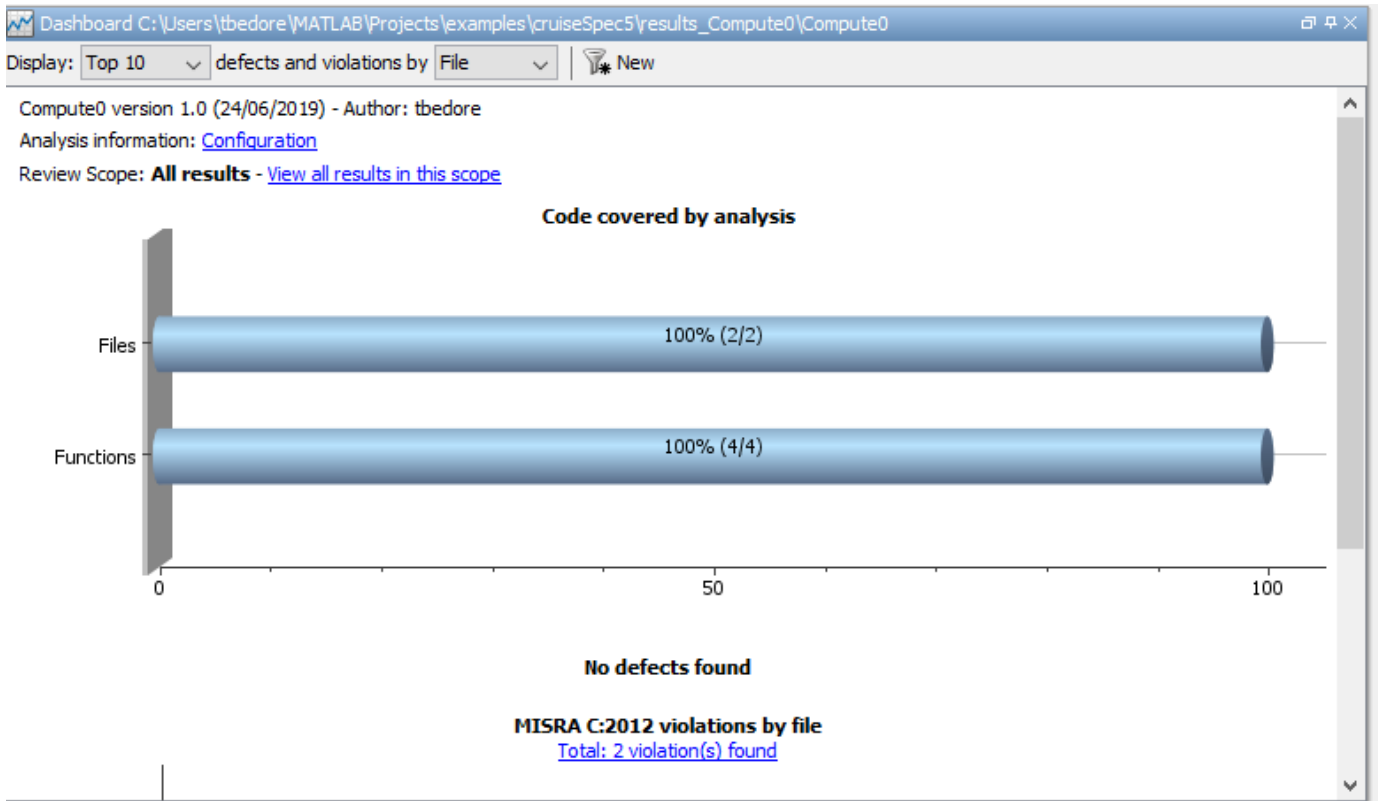
- 1 Expand the tree for rule 8.7 and click through the different results.

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. 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** option to Project configuration to choose a subset of MISRA rules in the Polyspace configuration.
- 4 Click **Configure**.
- 5 In the Polyspace window, on the left pane, click **Coding Standards & Code Metrics**. Then select **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.

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, Polyspace found only two violations.



When you integrate this model 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. If you want to generate a report every time you run an analysis, see [Generate report \(Polyspace Bug Finder\)](#).

- 1 If they are not open already, open your results in the Polyspace environment.
- 2 From the toolbar, select **Reporting > Run Report**.
- 3 Select **BugFinderSummary** as your report type.
- 4 Click **Run Report**.

The report is saved in the same folder as your results.

- 5 To open the report, select **Reporting > Open Report**.

Test Code Against Model Using Software-in-the-Loop Testing

You previously showed that the model functionality meets its requirements by running test cases based on those requirements. Now run the same test cases on the generated code to show that the code produces equivalent results and fulfills the requirements. Then compare the code coverage to the model coverage to see the extent to which the tests exercised the generated code.

- 1 In MATLAB, in the project window, open the `tests` folder, then open `SILTests.mldatx`. The file opens in the Test Manager.

- 2 Review the test case. On the **Test Browser** pane, navigate to SIL Equivalence Test Case. This equivalence test case runs two simulations for the `simulinkCruiseErrorAndStandardsExample` model using a test harness.
 - Simulation 1 is a model simulation in normal mode.
 - Simulation 2 is a software-in-the-loop (SIL) simulation. For the SIL simulation, the test case runs the code generated from the model instead of running the model.

The equivalence test logs one output signal and compares the results from the simulations. The test case also collects coverage measurements for both simulations.

- 3 Run the equivalence test. Select the test case and click **Run**.
- 4 Review the results in the Test Manager. In the **Results and Artifacts** pane, select **SIL Equivalence Test Case** to see the test results. The test case passed and the results show that the code produced the same results as the model for this test case.

The screenshot shows the Test Manager interface with the following components:

- Ribbon:** Contains tabs for FILE, EDIT, RUN, RESULTS, ENVIRONMENT, and RESOURCES. The RUN tab is active, showing options like Run, Run with Stepper, Stop, Parallel, Report, Visualize, Highlight in Model, Export, Testing Dashboard, Preferences, and Help.
- Test Browser:** Shows a list of test cases and results. The 'SIL Equivalence Test Case' is selected and highlighted in blue. Below it are sub-items like 'Equivalence Criteria Result', 'Verify Statements 1', 'Verify Statements 2', and 'Current: Sim Output 1 (simulink)'. A status indicator shows '1' with a green checkmark.
- Results and Artifacts:** Displays the details for the selected test case. It includes sections for SUMMARY, LOGS, DESCRIPTION, and COVERAGE RESULTS. The COVERAGE RESULTS section contains a table with the following data:

ANALYZED MODEL	REPORT	SIM MODE	COM...	DECISION	CONDITION	MCDC	EXECUTION	FUNCTION	FUNCTION...
<code>dlv_su32.c</code>		SIL	2	0%	--	--	0%	0%	--
<code>simulinkCruiseErrorAndStandardsExample</code>		ModelRe...	22	54%	44%	17%	70%	100%	33%
<code>simulinkCruiseErrorAndStandardsExample</code>		Normal	31	50%	41%	25%	--	--	--

At the bottom right of the coverage table, there are buttons for '+ Add Tests for Missing Coverage' and 'Export'.

- 5 Expand the **Coverage Results** section of the results. The coverage measurements show the extent to which the test case exercised the model and the code. When you run multiple test cases, you can view aggregated coverage measurements in the results for the whole run. Use the coverage results to add tests and meet coverage requirements, as shown in “Perform Functional Testing and Analyze Test Coverage” (Simulink Check).

You can also test the generated code on your target hardware by running a processor-in-the-loop (PIL) simulation. By adding a PIL simulation to your test cases, you can compare the test results and coverage results from your model to the results from the generated code as it runs on the target hardware. For more information, see “Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution” (Embedded Coder).

See Also

Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Bug Finder)
- “Test Two Simulations for Equivalence” on page 6-35
- “Export Test Results” on page 7-16

