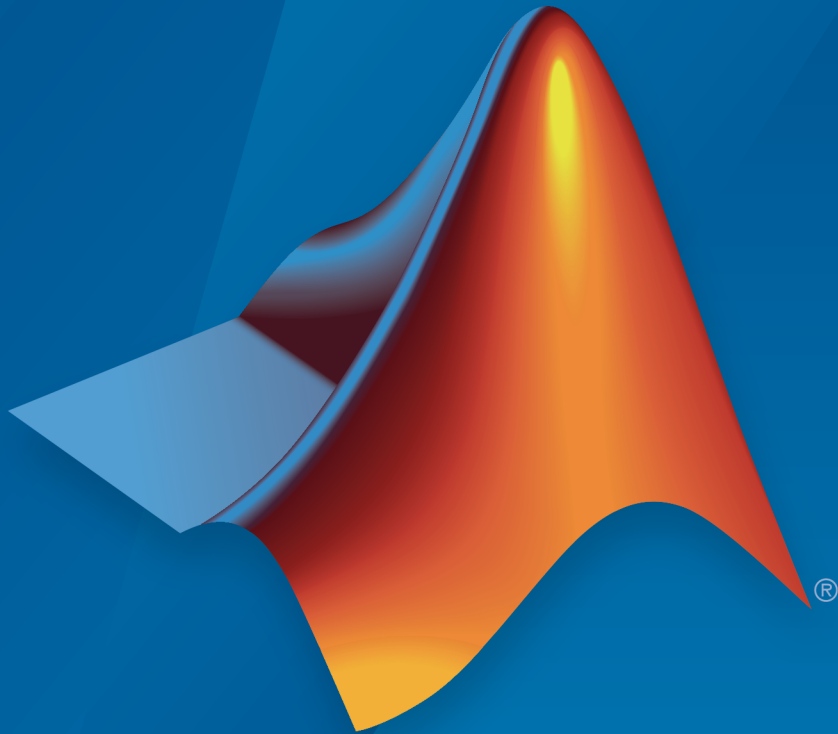


Simulink[®] Test[™]
Getting Started Guide



MATLAB[®]&SIMULINK[®]

R2015a



How to Contact MathWorks



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



Phone: 508-647-7000



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

Simulink[®] Test[™] Getting Started Guide

© COPYRIGHT 2015 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)

Product Overview

1

Simulink Test Product Description	1-2
Key Features	1-2

Introduction

2

Refine, Test, and Debug a Subsystem	2-2
Model and Requirements	2-2
Create a Harness for the Controller	2-4
Inspect and Refine the Controller	2-6
Add a Test Case and Test the Controller	2-7
Debug the Controller	2-8
Test Downshift Points of a Transmission Controller	2-11
Test Objectives and Model	2-11
The Test Sequence	2-12
Add Test Assessments for Controller	2-13
Test the Controller	2-15
Test Model Output Against a Baseline	2-17
Create the Test Case	2-17
Run the Test Case and View Results	2-18
Functional Testing in Verification	2-21
Introduction to the Test Manager	2-22
Test Manager Description	2-22
Test Creation and Hierarchy	2-22
Test Results	2-23

Share Results	2-23
---------------------	------

Product Overview

Simulink Test Product Description

Develop, manage, and execute simulation-based tests

Simulink[®] Test[™] provides tools for authoring, managing, and executing systematic, simulation-based tests. You can create nonintrusive test harnesses to test models and subsystems. Simulink Test includes a test sequence block that lets you construct complex test sequences and assessments, and a test manager that lets you manage and execute tests. It enables functional, baseline, equivalence, and back-to-back testing, including software-in-the-loop (SIL) and processor-in-the-loop (PIL). You can generate reports, archive and review test results, rerun failed tests, and debug the component or system under test.

The test harness in Simulink Test lets you test components without creating a separate test model. You can apply pass and fail criteria that include absolute and relative tolerances, limits, logical checks, and temporal conditions. Test execution can be automated or customized with setup and cleanup scripts. Simulink Test stores test cases and their results, creating a repository for reviewing and investigating failures. You can link requirements to a test case captured in Microsoft[®] Word, IBM[®] Rational[®] DOORS[®], and other documents (with Simulink Verification and Validation[™]).

Key Features

- Test harness for subsystem or model testing
- Test sequence block for running tests and assessments
- Pass-fail criteria, including tolerances, limits, and temporal conditions
- Baseline, equivalence, and back-to-back testing
- Setup and cleanup scripts for customizing test execution
- Test manager for authoring, executing, and organizing test cases and their results
- Automatic report generation to document test outcomes

Introduction

- “Refine, Test, and Debug a Subsystem” on page 2-2
- “Test Downshift Points of a Transmission Controller” on page 2-11
- “Test Model Output Against a Baseline” on page 2-17
- “Functional Testing in Verification” on page 2-21
- “Introduction to the Test Manager” on page 2-22

Refine, Test, and Debug a Subsystem

In this section...
“Model and Requirements” on page 2-2
“Create a Harness for the Controller” on page 2-4
“Inspect and Refine the Controller” on page 2-6
“Add a Test Case and Test the Controller” on page 2-7
“Debug the Controller” on page 2-8

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

Model and Requirements

- 1 Access the model. At the MATLAB command prompt, enter

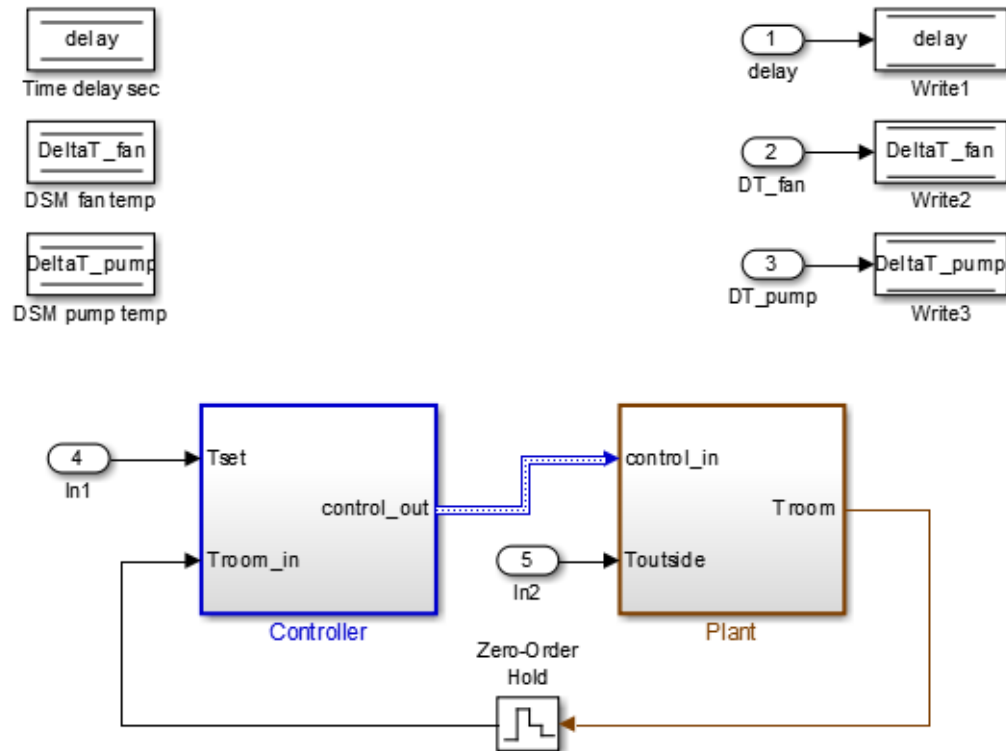
```
cd(fullfile(matlabroot, 'help', 'toolbox', 'sltest', 'examples'));
```

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

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

- 3 Open the model.

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

Copyright 1990-2014 The MathWorks, Inc.

In the example model:

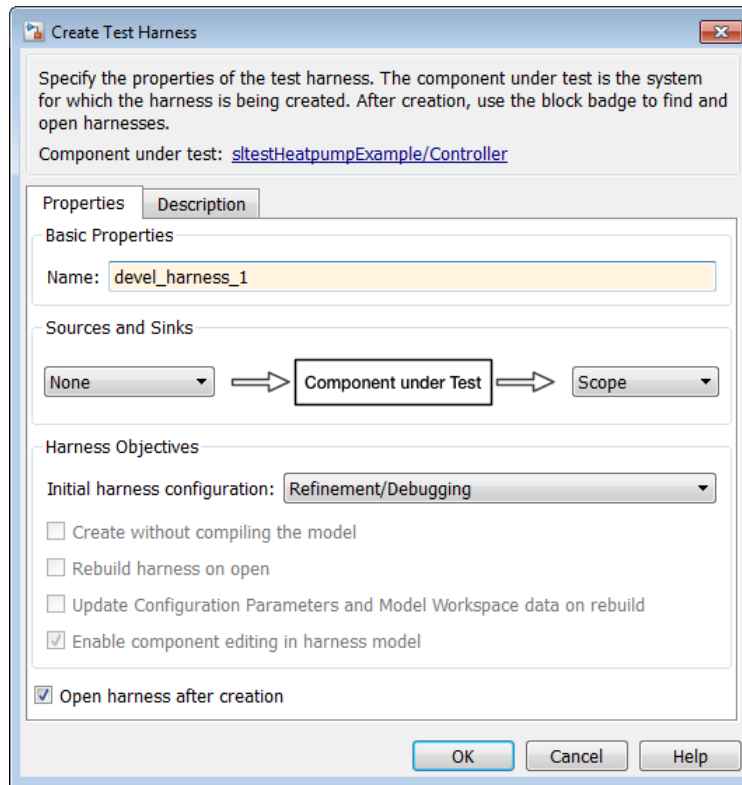
- The controller accepts the room temperature and the set temperature inputs.
- The controller output is a bus with signals controlling the fan, heat pump, and the direction of the heat pump (heat or cool).
- The plant accepts the control bus. The heat pump and the fan signals are Boolean, and the heat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

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

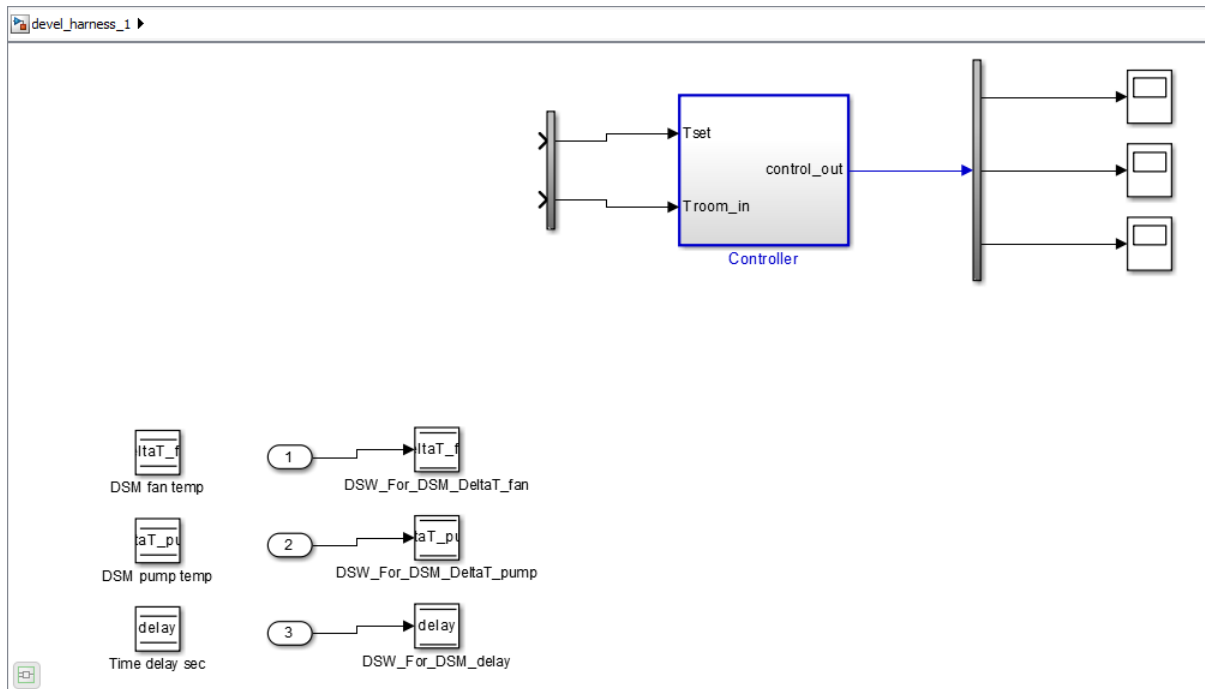
Temperature condition	System state	Fan command	Pump command	Pump direction
$ T_{room} - T_{set} < \Delta T_{fan}$	idle	0	0	0
$\Delta T_{fan} \leq T_{room} - T_{set} < \Delta T_{pump}$	fan only	1	0	0
$ T_{room} - T_{set} < \Delta T_{pump}$ and $T_{set} < T_{room}$	cooling	1	1	-1
$ T_{room} - T_{set} 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

Create a Harness for the Controller

- 1 Right-click the Controller subsystem and select **Test Harness > Create Test Harness (Controller)**.
- 2 Set the harness properties:
 - **Name:** devel_harness_1
 - **Sources and Sinks:** None and **Scope**
 - **Initial harness configuration:** Refinement/Debugging
 - Select **Open harness after creation**.

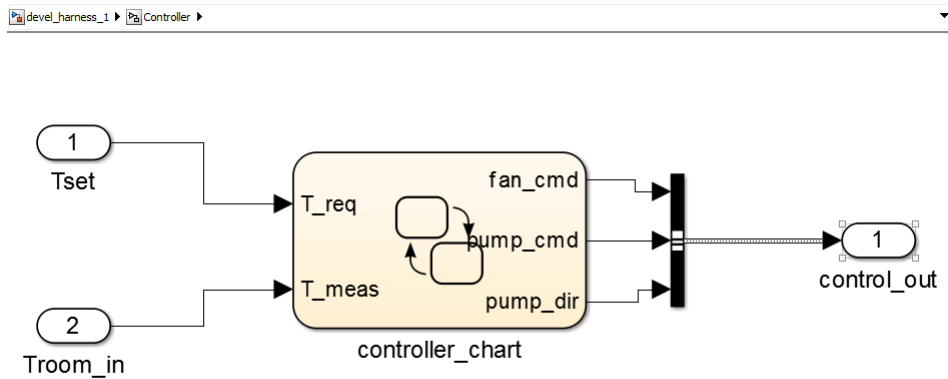


- 3 Click **OK** to create the test harness.



Inspect and Refine the Controller

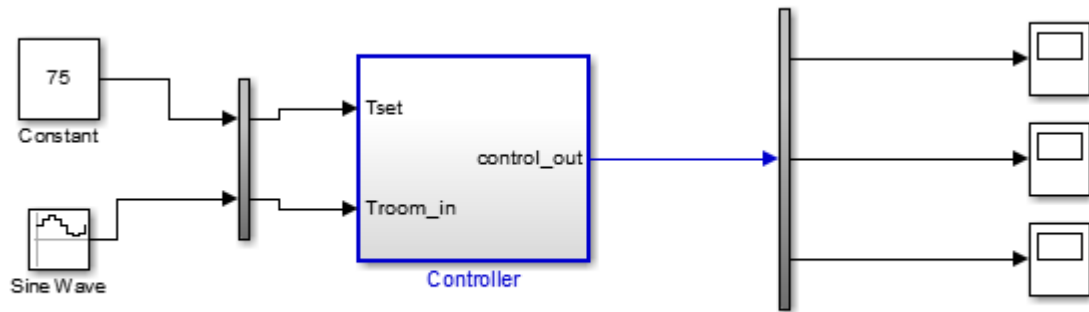
- 1 Double-click Controller to open the subsystem.
- 2 Notice that the state chart is disconnected from its ports. Fix this issue by connecting the chart as shown.



- 3 In the harness, click the save button in the toolbar to save the harness and model.

Add a Test Case and Test the Controller

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

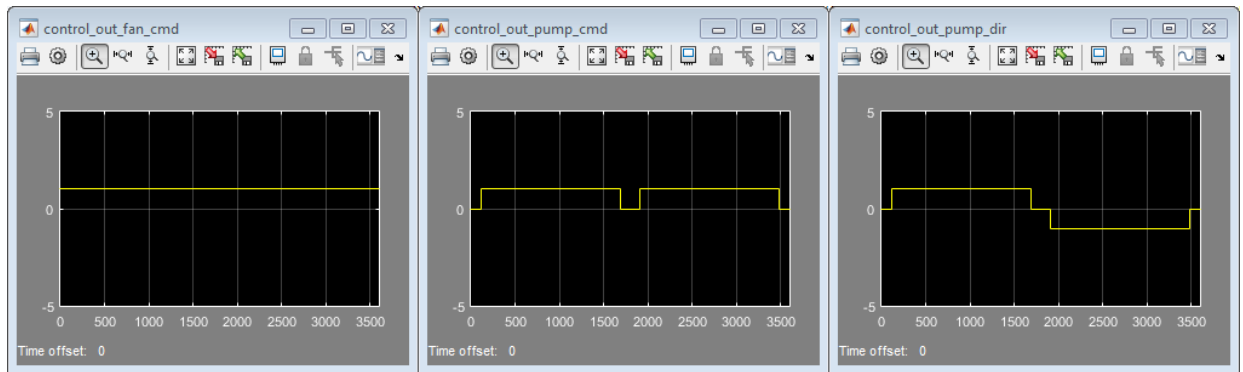


- 5 In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select **Input** and enter `u`. `u` is an existing structure in the MATLAB[®] base workspace.
- 6 In the **Solver** pane, set **Stop time** to 3600.
- 7 Open the three scopes in the harness model.
- 8 Simulate the harness.

Debug the Controller

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

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



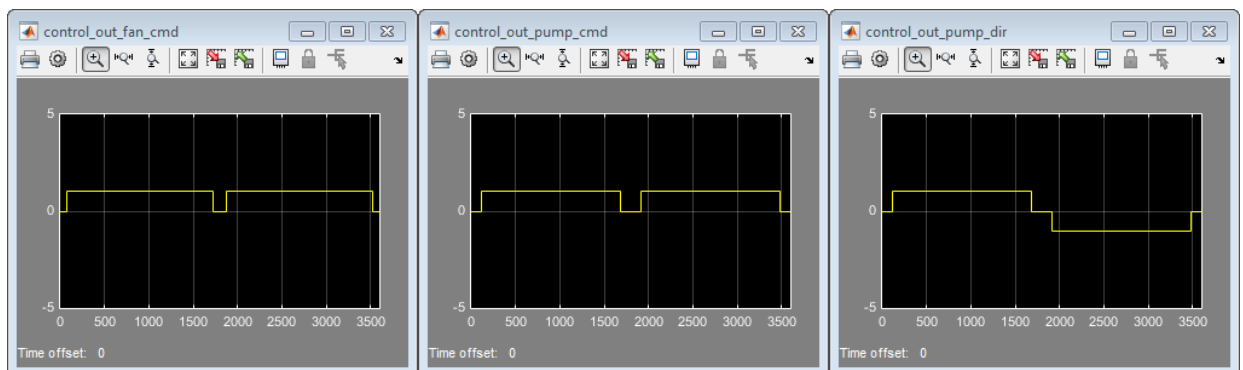
- 2 In the harness model, open the Controller subsystem.
- 3 Open `controller_chart`.
- 4 In the IDLE state, `fan_cmd` is set to return 1. Change `fan_cmd` to return 0. IDLE is now:

```

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

```

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



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

Related Examples

- “Test a Model Component Using Signal Functions”
- “Test Downshift Points of a Transmission Controller”

Test Downshift Points of a Transmission Controller

In this section...

“Test Objectives and Model” on page 2-11

“The Test Sequence” on page 2-12

“Add Test Assessments for Controller” on page 2-13

“Test the Controller” on page 2-15

Test Objectives and Model

This example demonstrates a test sequence and test assessment for a transmission shift logic controller. The controller should downshift between each of its gear ratios in response to a ramped throttle application. As the throttle increases, the vehicle speed is held constant. Based on hypothetical requirements, the controller performance is assessed in a Test Assessment block.

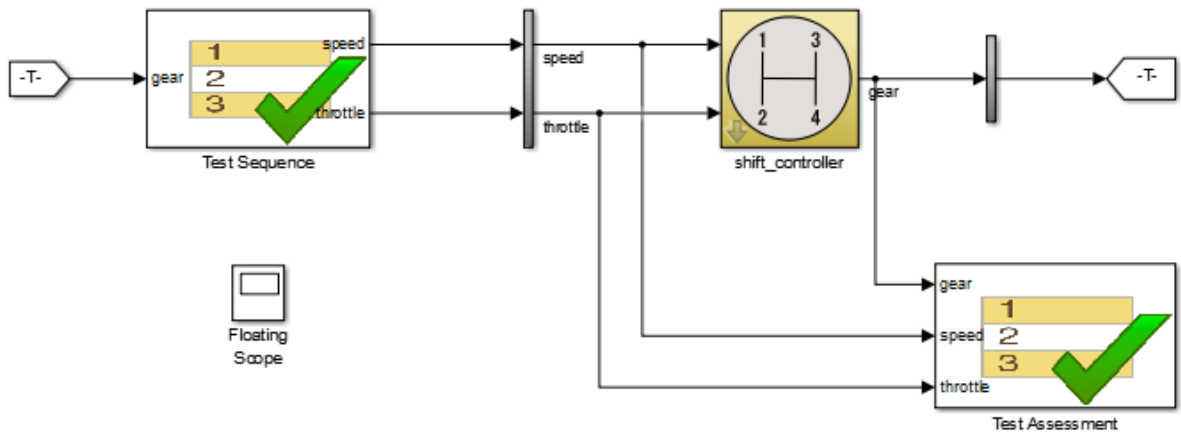
- 1 Access the model. At the MATLAB command line, enter

```
cd(fullfile(matlabroot, 'help', 'toolbox', 'sltest', 'examples'));
```

- 2 Copy the model `sltestTestSequenceDownshift.slx` to a writable location on the MATLAB path.
- 3 Open the model.

```
open_system('sltestTestSequenceDownshift');
```

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



The Test Sequence

- 1 Double-click the Test Sequence block to open the editor.
- 2 The test sequence begins by ramping speed to 75 to initialize the controller to fourth gear. Throttle is then ramped at constant speed until a gear change. Downshifts are performed to second and first gear. 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> <p><i>Add step after - Add sub-step</i></p>	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>		

Add Test Assessments for Controller

Assume that the requirements for the shift controller include:

- Speed shall never be negative.
- Gear shall never be negative.

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

Open the Test Assessment block. The first three requirements correspond to these assertions already in the block. If the controller violates one of the assertions, the simulation fails.

```
assert(speed >= 0, 'Speed < 0');  
assert(throttle >= 0, 'Throttle < 0');  
assert(throttle <= 100, 'Throttle > 100');  
assert(gear > 0, 'Impossible gear');
```

Add additional assessments corresponding to the last requirement that the controller shall not allow the engine to overspeed. Assume that the engine cannot overspeed in top (fourth) gear.

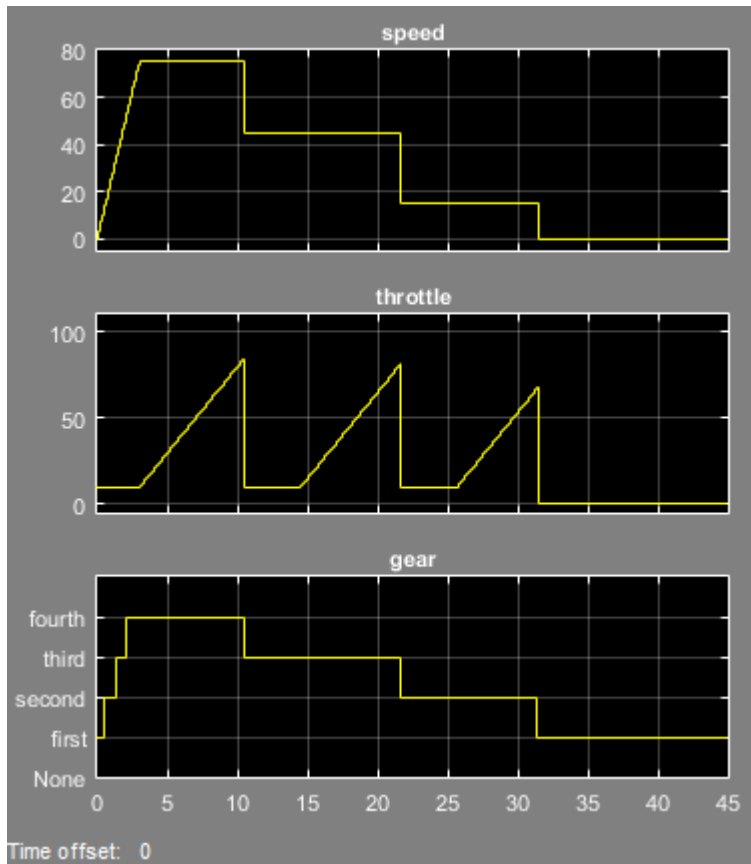
- 1 Add three sub-steps to the AssertConditions step. To add each step, right-click the AssertConditions step and select **Add sub-step**.
- 2 Right-click the AssertConditions step and select **When decomposition**. This changes the switching behavior of the sub-steps of AssertConditions. Switching is based on the signal condition defined in the **Step** column, with each condition preceded by the *when* operator. Because the switching is controlled by *when*, the **Transition** and **Next Step** columns are grayed out. The last step **Else** in the when decomposition covers any condition not defined above it, and is left blank.
- 3 Enter the Step conditions as shown.

Sub-steps of AssertConditions
OverSpeed3 when gear == 3 assert(speed <= 90, 'Engine overspeed in gear 3')
OverSpeed3 when gear == 2 assert(speed <= 50, 'Engine overspeed in gear 2')
OverSpeed3 when gear == 1 assert(speed <= 30, 'Engine overspeed in gear 1')
Else

Step	Transition	Next Step
<div style="border: 1px solid black; padding: 5px;"> ▢ ↗ AssertConditions assert(speed >= 0,'Speed < 0'); assert(throttle >= 0,'Throttle < 0'); assert(throttle <= 100,'Throttle > 100'); assert(gear > 0,'Impossible gear'); </div>		
<div style="border-left: 1px solid black; padding-left: 5px;"> OverSpeed3 when gear == 3 assert(speed <= 90,'Engine overspeed in gear 3') </div>		
<div style="border-left: 1px solid black; padding-left: 5px;"> OverSpeed2 when gear == 2 assert(speed <= 50,'Engine overspeed in gear 2') </div>		
<div style="border-left: 1px solid black; padding-left: 5px;"> OverSpeed1 when gear == 1 assert(speed <= 30,'Engine overspeed in gear 1') </div>		
<div style="border-left: 1px solid black; padding-left: 5px;"> Else </div>		

Test the Controller

- 1 Open the scope.
- 2 Set the test harness model simulation time to 45 sec.
- 3 Simulate the harness. The output shows the progressive throttle ramp at each test step, and the corresponding downshift.



4 The controller passes all of the assessments in the Test Assessment block.

See Also

Blocks

Test Sequence


Test Model Output Against a Baseline

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

Create the Test Case

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




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

- 4 Right-click the baseline test case in the **Test Browser** pane, and select **Rename**. Rename the test case to **Slip Baseline Test**.
- 5 Under **System Under Test** in the test case, click the **Use current model** button  to load the `sldemo_absbrake` model into the test case.
- 6 Under **Baseline Criteria**, click **Capture** to record a baseline data set from the model specified under **System Under Test**.

Save the baseline data set to a location. After you save the baseline MAT-file, the model runs and the baseline criteria appear in the table.

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

SIGNAL NAME	ABS TOL	REL TOL
<input checked="" type="checkbox"/> test_capture.mat	0	0.00%
<input checked="" type="checkbox"/> yout	15	0.00%
<input checked="" type="checkbox"/> yout	0	0.00%
<input checked="" type="checkbox"/> yout	0	0.00%
<input checked="" type="checkbox"/> slp	0	0.00%

 Add...
  Capture...
  Delete

For more information about tolerances and criteria, see .

Run the Test Case and View Results

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

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

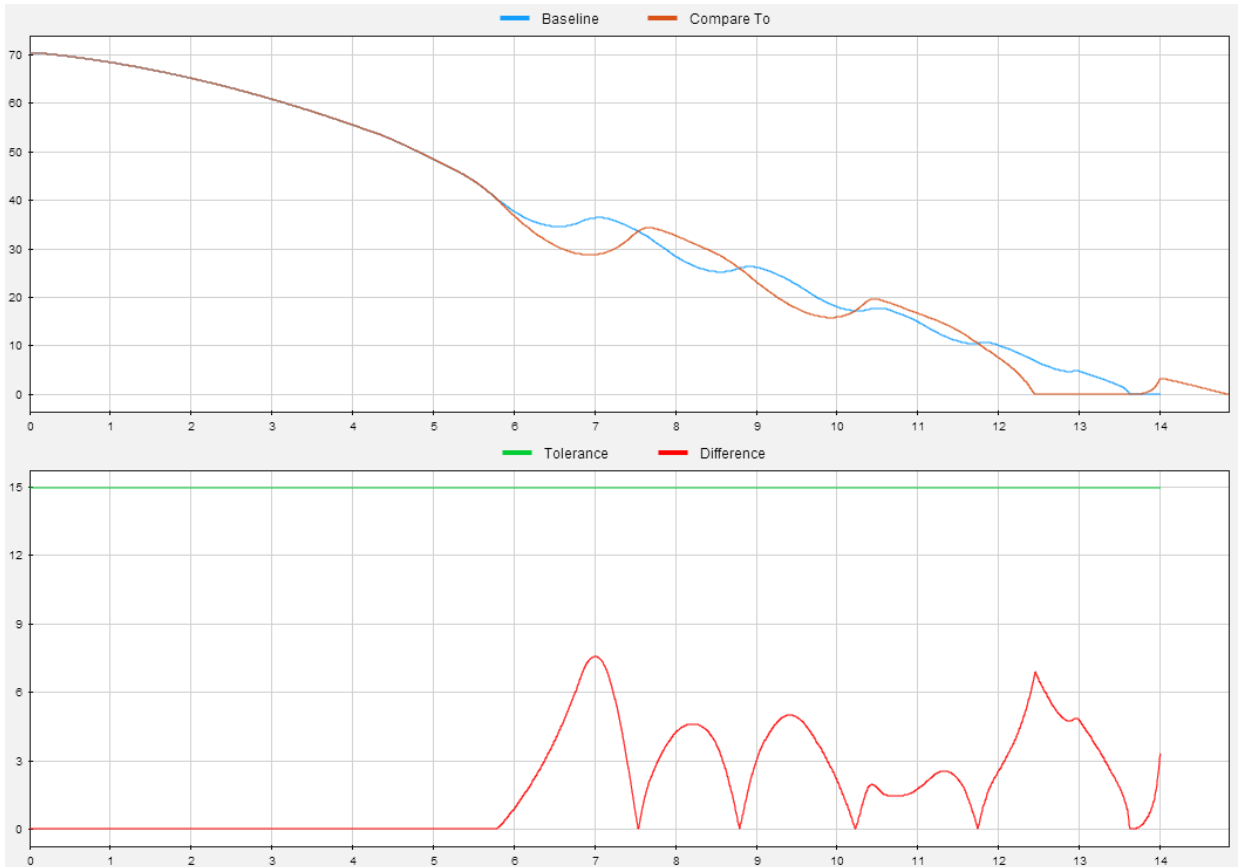
- 4 Expand the results until you see the baseline criteria result.

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.

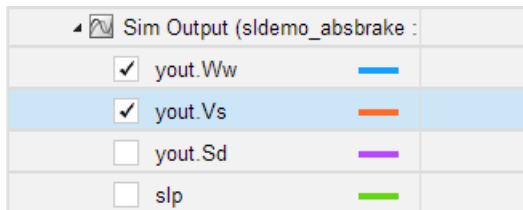
- 5 To view the `yout.Ww` signal comparison between the model and the baseline criteria, expand **Baseline Criteria Result** and click the option button next to the `yout.Ww` signal.

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

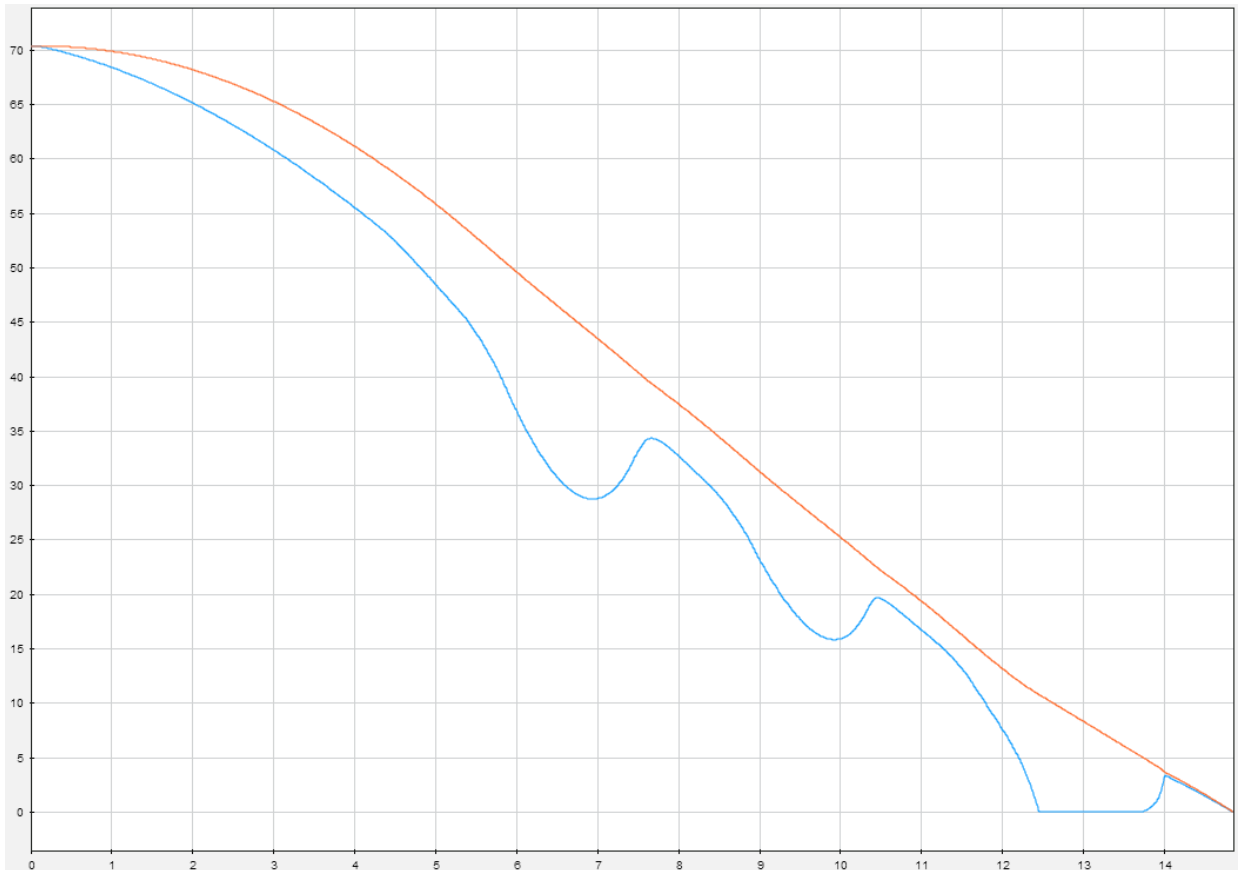
The **Comparison** tab opens and shows the criteria comparisons for the `yout.Ww` signal.



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



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



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

Functional Testing in Verification

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

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

Functional testing can be used to:

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

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

Introduction to the Test Manager

In this section...

“Test Manager Description” on page 2-22

“Test Creation and Hierarchy” on page 2-22

“Test Results” on page 2-23

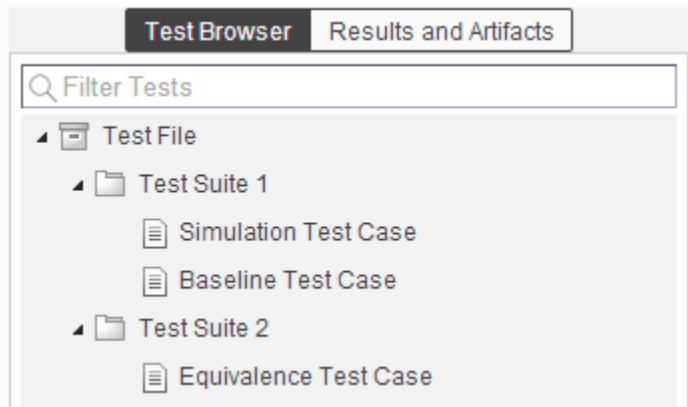
“Share Results” on page 2-23

Test Manager Description

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

Test Creation and Hierarchy

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



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

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

Test Results

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

Share Results

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

