Simulink® Test™

User's Guide

MATLAB® & SIMULINK®

R2015a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Simulink® Test™ User's Guide

Revision History

March 2015	Online only	New for Version 1.0 (Release 2015a)

Contents

1 Test Strategies

Functional Testing in Verification **1-2**

Use Requirements-Based Test Cases **1-3**

2 Test Harness

Test Harness and Model Relationship **2-2**
Test Harness Description **2-2**
Harness / Model Relationship for a Model Component **2-3**
Harness / Model Relationship for a Top-Level Model **2-4**
Resolving Parameters **2-5**

Considerations and Limitations **2-6**
Test Harness **2-6**
Test Sequence Block **2-6**

Select Test Harness Properties for Your Task **2-8**
Create a Test Harness **2-8**
Considerations for Selecting Test Harness Properties **2-8**
Choosing Sources and Sinks **2-8**
Use Separate Assessment Block **2-9**
Initial Harness Configuration **2-9**
Verification Modes **2-10**
Change Harness Properties **2-11**

Test Harness Parameters and Signals **2-12**
Test Harness Generation Without Compilation **2-12**
Signal Conversion Subsystem **2-12**

Refine, Test, and Debug a Subsystem **2-14**
Model and Requirements **2-14**
Create a Harness for the Controller **2-16**
Inspect and Refine the Controller **2-18**
Add a Test Case and Test the Controller **2-19**
Debug the Controller **2-20**

Manage Test Harnesses **2-23**
Preview and Open a Test Harness **2-23**
Delete Test Harnesses Programmatically **2-24**
Convert Test Harnesses Into Separate Models **2-25**

Synchronize Changes Between Test Harness and Model . . **2-27**
Maintain SIL or PIL Block Fidelity **2-27**
Synchronize Changes to the Component Under Test **2-27**
Rebuild Test Harness **2-28**
Update Parameters from Test Harness to Model **2-28**

3 Test Sequences and Assessments

Test a Model Component Using Signal Functions **3-2**
Create a Test Sequence **3-2**
Simulate the Test Harness **3-4**

Test Downshift Points of a Transmission Controller **3-6**
Test Objectives and Model **3-6**
The Test Sequence **3-7**
Add Test Assessments for Controller **3-8**
Test the Controller **3-10**

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

SIL Verification for a Subsystem **4-2**
Create a SIL Verification Harness for a Controller **4-3**
Configure and Simulate a SIL Verification Harness **4-4**

Compare the SIL Block and Model Controller Outputs **4-4**

5 Simulink Test Manager Introduction

Introduction to the Test Manager **5-2**
Test Manager Description **5-2**
Test Creation and Hierarchy **5-2**
Test Results **5-3**
Share Results **5-3**

6 Test Manager Test Cases

Test Model Output Against a Baseline **6-2**
Create the Test Case **6-2**
Run the Test Case and View Results **6-3**

Test a Simulation for Run-Time Errors **6-6**
Configure the Model **6-6**
Create the Test Case **6-7**
Run the Test Case **6-7**
View Test Results **6-8**

Generate Test Cases from Model Components **6-9**

Run Test Cases Programmatically **6-11**
List of Functions and Classes **6-11**
Run a Test File and Report Results **6-11**

How Tolerances Are Applied to Test Criteria **6-13**
Modify Criteria Tolerances **6-13**

Test Manager Limitations **6-14**
Simulation Mode **6-14**
Callback Scripts **6-14**
Simulink Design Verifier Input File **6-14**

Protected Models 6-15

Test Case Sections **6-16**
Description **6-17**
Requirements **6-17**
System Under Test **6-17**
Parameter Overrides **6-18**
Callbacks **6-19**
Inputs **6-19**
Outputs **6-20**
Configuration Settings **6-20**
Simulation 1 and Simulation 2 . . . . **6-20**
Equivalence Criteria **6-21**
Baseline Criteria **6-21**

7 Test Manager Results and Reports

View Test Case Results **7-2**
View Results Summary **7-2**
Visualize Test Case Simulation Output and Criteria **7-4**

Export Test Results and Generate Reports **7-9**
Export Results **7-9**
Create a Test Results Report **7-10**

Results Sections **7-11**
Summary **7-12**
Test Requirement **7-12**
Errors **7-13**
Logs **7-13**
Notes **7-13**
Parameter Overrides **7-13**

Test Strategies

- “Functional Testing in Verification” on page 1-2
- “Use Requirements-Based Test Cases” on page 1-3

Functional Testing in Verification

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

Functional testing can be used 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.

Use Requirements-Based Test Cases

Begin your model development with high-level requirements describing the system operation in natural language. To reduce ambiguity and more fully define the system specification, refine high-level requirements into detailed requirements that define operation with regards to the system architecture. Refer to these requirements as you design, refine, and test the system.

Effective functional testing uses test cases that are derived from high-level and detailed requirements. Test cases include inputs to the component under test (test vectors) and expected outputs. Based on expected outputs, you define acceptance criteria including tolerances compared to other data, or assessments on the system behavior. Basing test cases on requirements helps you link pass or fail results of each test to specific line items in your requirements document.

You can link test cases to requirements with the Test Manager, and with Simulink Verification and Validation™ you can link blocks to requirements.

2

Test Harness

- "Test Harness and Model Relationship" on page 2-2
- "Considerations and Limitations" on page 2-6
- "Select Test Harness Properties for Your Task" on page 2-8
- "Test Harness Parameters and Signals" on page 2-12
- "Refine, Test, and Debug a Subsystem" on page 2-14
- "Manage Test Harnesses" on page 2-23
- "Synchronize Changes Between Test Harness and Model" on page 2-27

Test Harness and Model Relationship

In this section...

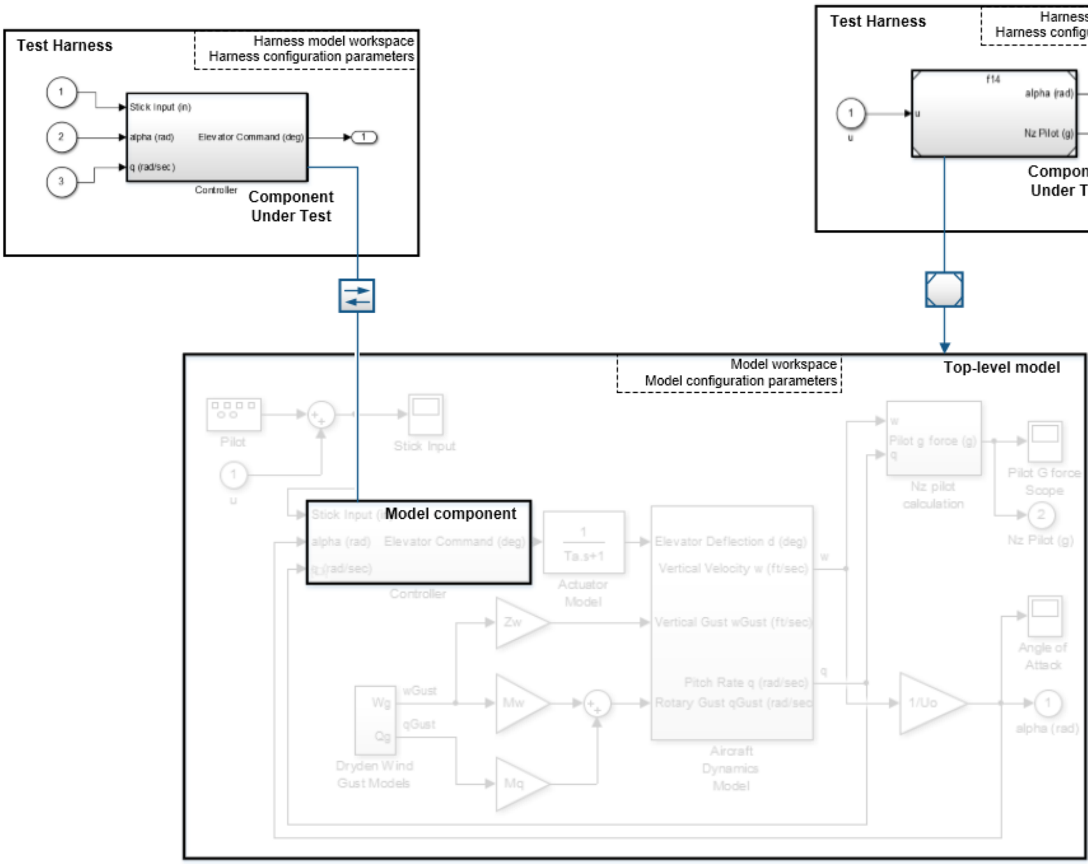
"Test Harness Description" on page 2-2
"Harness / Model Relationship for a Model Component" on page 2-3
"Harness / Model Relationship for a Top-Level Model" on page 2-4
"Resolving Parameters" on page 2-5

Test Harness Description

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

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

- A model component, such as a subsystem. The test harness isolates the component, providing a separate simulation environment from the main model.
- A top-level model. The component under test is a Model block referencing the main model.

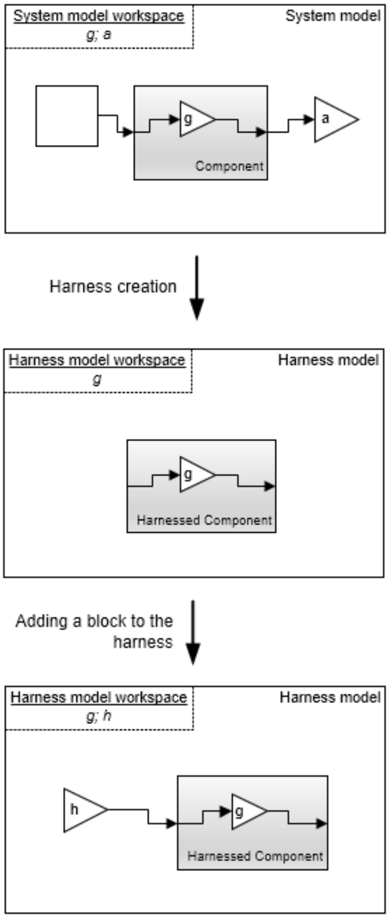


Harness / Model Relationship for a Model Component

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

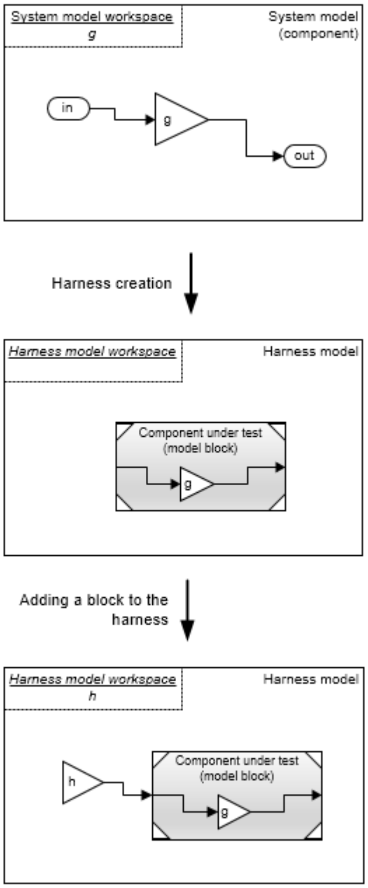
This example shows a test harness for a component that contains a Gain block. The harness model workspace contains a copy of the parameter *g* because *g* defines a part of the component.

The parameter *h* is the gain of a gain block in the harness, outside the component under test (CUT). *h* exists only in the harness model workspace.



Harness / Model Relationship for a Top-Level Model

When you associate a harness with the top level of the main model, the harness model workspace does not contain copies of parameters relevant to the component. The component under test is a Model block referencing the main model, and parameters remain in the main model workspace. In this example, the component under test references the main model, and the variable *g* exists in the main model workspace. The variable *h* is the value of the Gain block in the harness. It exists only in the harness model workspace.



Resolving Parameters

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

More About

- "Componentization Guidelines"

Considerations and Limitations

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

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

Test Harness

- You can open only one test harness at a time per main model.
- Do not comment out the component under test in the test harness. Commenting out the component under test can cause unexpected behavior.
- Requirements linking is not supported for blocks or other objects in test harness models. If you have a Simulink Verification and Validation license, you can link requirements to test cases in the test manager. See "Requirements".
- If a subsystem has a test harness, you cannot expand the subsystem. Delete all test harnesses before expanding the subsystem.
- Test harnesses are not supported for blocks underneath a Stateflow® object.
- For a library, a test harness can only be created for an active top-level library link.
- Test harnesses do not support asynchronous sample times.
- Upgrade advisor and XML differencing are not supported for test harness models.
- A test harness with a Signal Builder block source does not support:
  - Frame-based signals
  - Complex signals
  - Variable-dimension signals
  - Arrays of buses
- For a test harness with a Test Sequence block source, all inputs to the component under test must operate with the same sample time.

Test Sequence Block

- HDL code generation is not supported for the Test Sequence block.

- Code generation reports do not display Test Sequence block contents.
- Requirements linking is not supported from the Test Sequence Editor.

Select Test Harness Properties for Your Task

In this section...
"Create a Test Harness" on page 2-8
"Considerations for Selecting Test Harness Properties" on page 2-8
"Choosing Sources and Sinks" on page 2-8
"Use Separate Assessment Block" on page 2-9
"Initial Harness Configuration" on page 2-9
"Verification Modes" on page 2-10
"Change Harness Properties" on page 2-11

Create a Test Harness

Create the test harness and set the harness properties using the Create Test Harness dialog box. Highlight the subsystem you want to create the harness for, or highlight no blocks to create a harness for the top-level model. From the menu, select **Analysis > Test Harness > Create Test Harness**.

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 results
- 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

You can set sources and sinks only during harness creation. You can set the other properties when you create the harness or change them after you create the harness.

Choosing Sources and Sinks

In the Create Test Harness dialog box, under **Sources and Sinks**, select the source and sink from the appropriate menus. Select a Test Sequence block source to use outputs of the component under test as inputs to the test case. You can build a test harness using any block from the Simulink Sources or Sinks library. Select `Custom` source or sink, and entering the path to the custom block, such as:

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

```
simulink/Sinks/Terminator
```

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

Use Separate Assessment Block

A standalone Test Assessment block can be useful if you want to reuse the same assessments in multiple test harnesses. To build your harness with a separate block, click **Use separate assessment block**.

You can also write test assessments directly in the Test Sequence block.

Initial Harness Configuration

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

- `Prototyping`: Choose this configuration if your model is early in development. You can edit the component under test in the test harness, and control when the harness is rebuilt from the main model. You can use this configuration if your main model does not compile.
- `Refinement/Debugging`: Choose this configuration if you want the test harness to include the configuration set, conversion subsystems, and model parameters for the component under test. This configuration can be useful for a nearly complete model, when you expect limited changes to the design.
- `Verification`: Choose this configuration if you require high fidelity between the main model and the test harness, which is commonly needed for model verification. The test harness prevents you from editing the component under test, and the test harness rebuilds every time you open it. In addition to a normal subsystem, you can choose a SIL or PIL block as the component under test (requires Embedded Coder®). See “Verification Modes”.

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

Property	Description	Additional Information
Create without compiling the model	When you select this property, the main model does not compile when generating the test harness. The test harness does not contain conversion subsystems, configuration parameters, or model workspace data for the component under test.	You might have to take additional steps for the test harness to compile, such as adding signal conversion blocks.
Rebuild harness on open	When you select this property, the test harness rebuilds every time you open it.	For details on the rebuild process, see "Synchronize Changes Between Test Harness and Model".
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".
Enable component editing in harness model	When you select this property, you can edit the component under test in the test harness.	

Verification Modes

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

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

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

Change Harness Properties

You can change properties of the test harness by clicking the badge  in the test harness block diagram.

See Also

Test Sequence | "Synchronize Changes Between Test Harness and Model"

Test Harness Parameters and Signals

In this section...
"Test Harness Generation Without Compilation" on page 2-12
"Signal Conversion Subsystem" on page 2-12

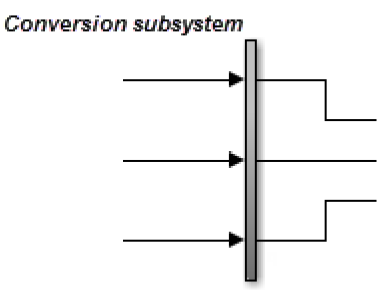
Test Harness Generation Without Compilation

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

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

To execute these processes, you can rebuild the harness when you are ready to compile the main model. For more information, see "Synchronize Changes Between Test Harness and Model" on page 2-27.

Signal Conversion Subsystem



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

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

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

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

Note: When you rebuild the test harness, the signal conversion subsystems are rebuilt. Any changes made to the conversion subsystems are lost.

Refine, Test, and Debug a Subsystem

In this section...

"Model and Requirements" on page 2-14
"Create a Harness for the Controller" on page 2-16
"Inspect and Refine the Controller" on page 2-18
"Add a Test Case and Test the Controller" on page 2-19
"Debug the Controller" on page 2-20

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

Model and Requirements

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

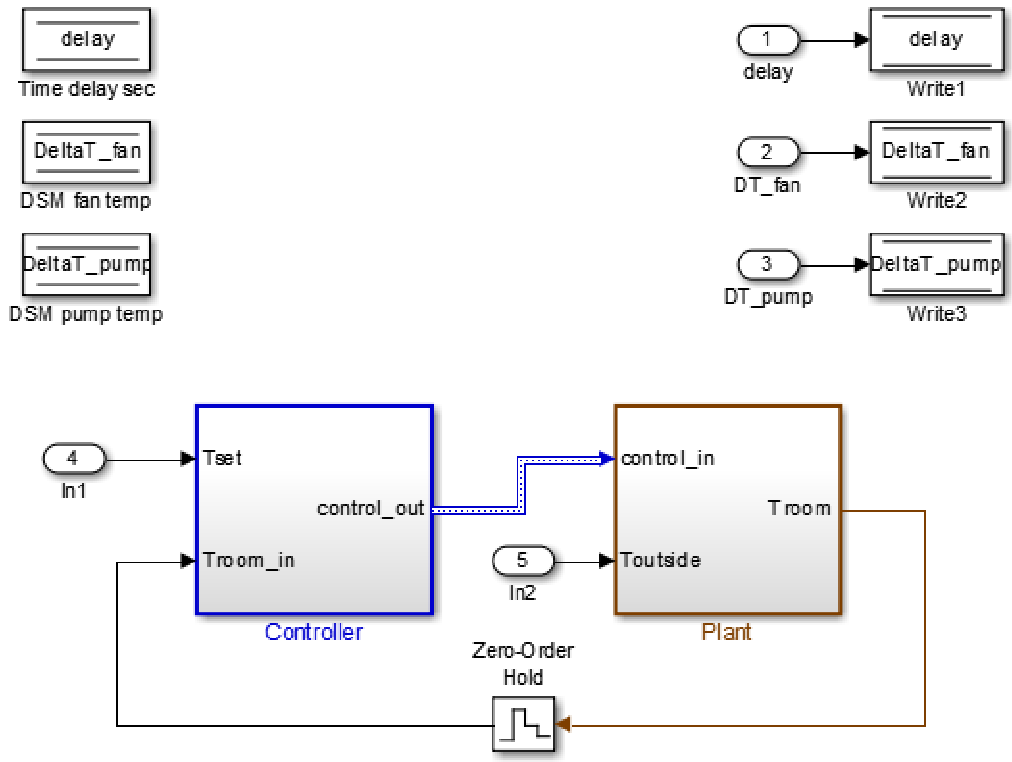
```
cd(fullfile(matlabroot,'help','toolbox','sltest','examples'));
```

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')
```



In the example model:

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

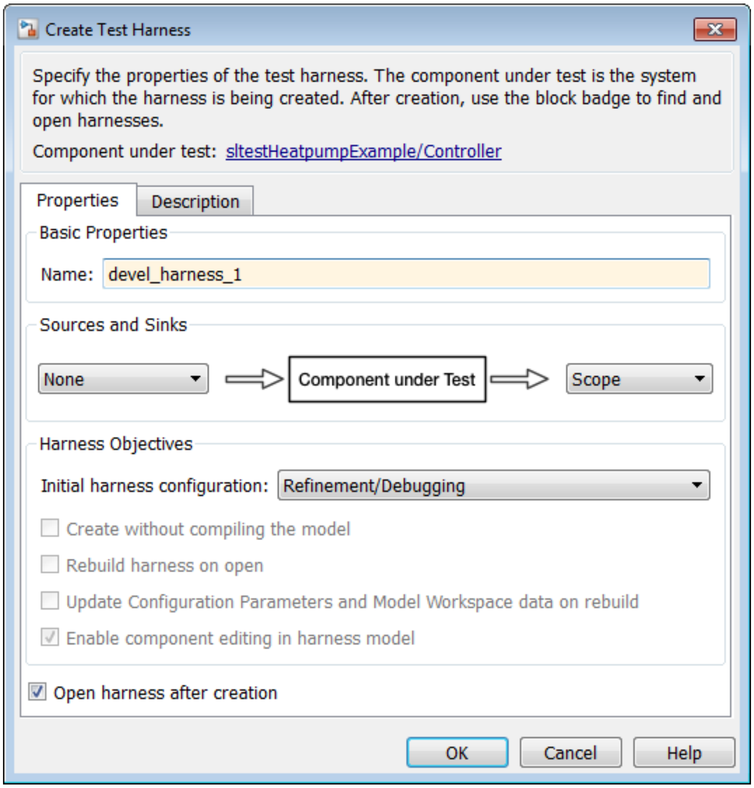
The test covers four temperature conditions. Each condition corresponds to one operating state with fan, pump, and pump direction signal outputs.

Temperature condition	System state	Fan command	Pump command	Pump direction
`|Troom - Tset| < DeltaT_fan`	idle	0	0	0
`DeltaT_fan <= |Troom - Tset| < DeltaT_pump`	fan only	1	0	0
`|Troom - Tset| < DeltaT_pump and Tset < Troom`	cooling	1	1	-1
`|Troom - Tset| < DeltaT_pump and Tset >Troom`	heating	1	1	1

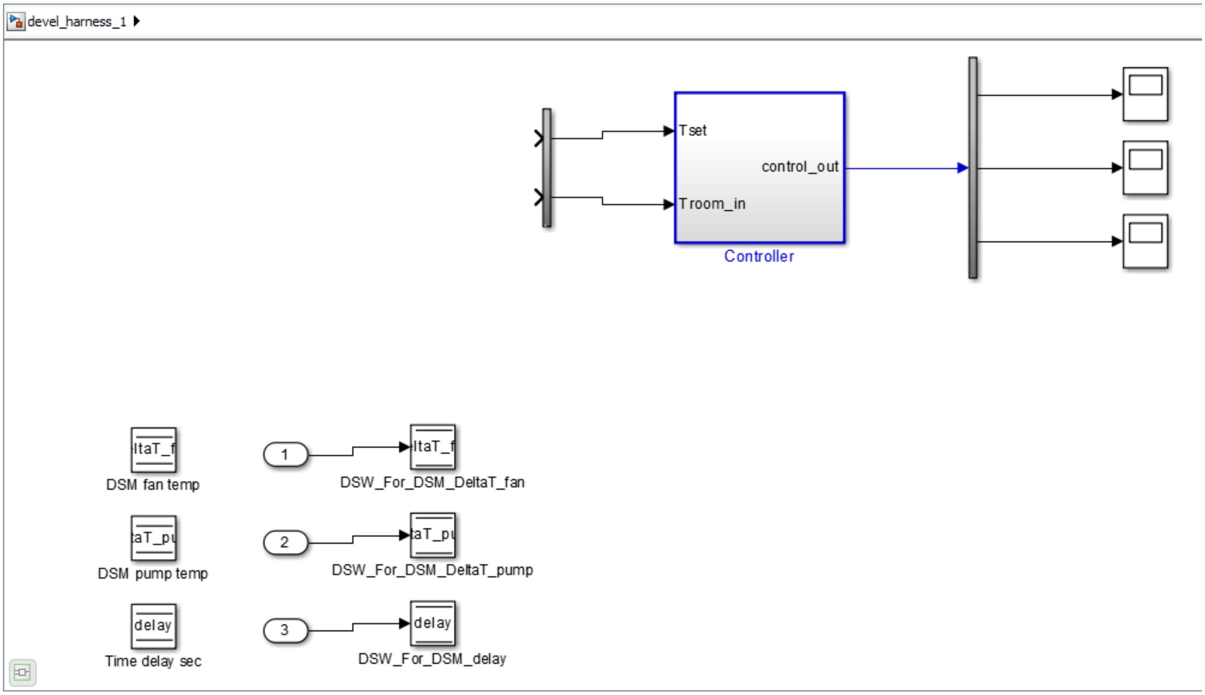
Create a Harness for the Controller

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



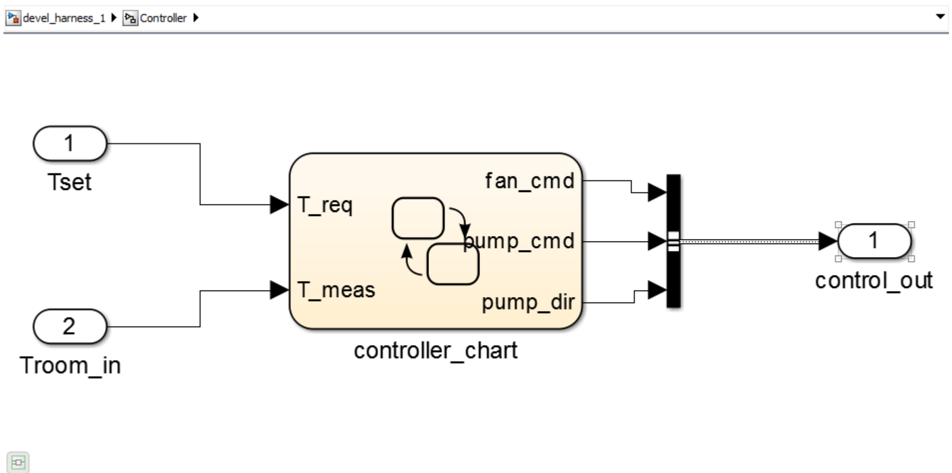
3 Click **OK** to create the test harness.



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

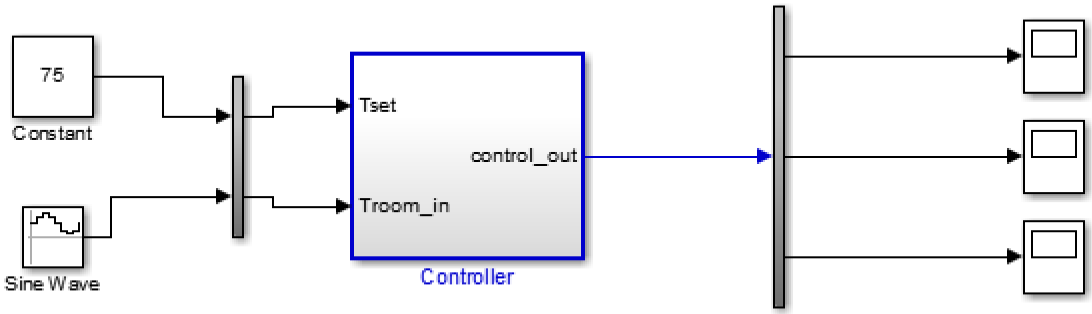
1 Navigate to the top level of `devel_harness_1`.

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`.

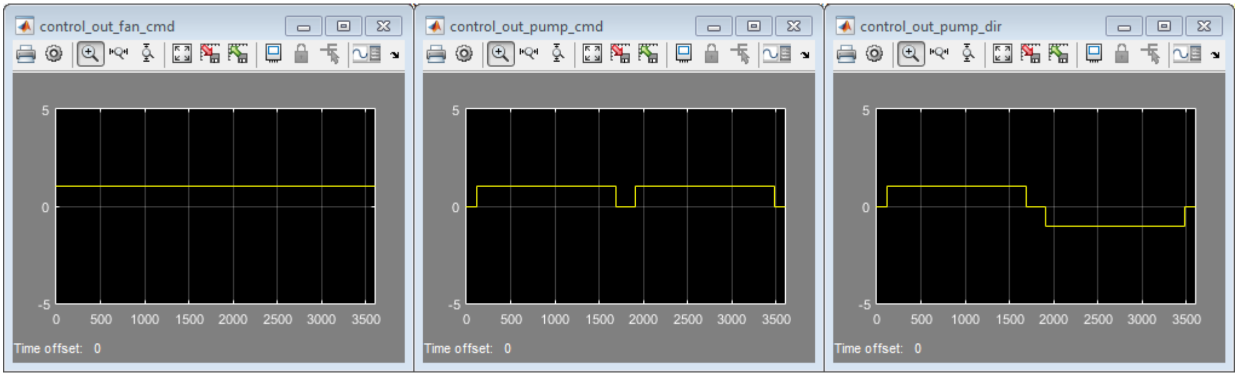
7 Open the three scopes in the harness model.

8 Simulate the harness.

Debug the Controller

1 Observe the controller output. `fan_cmd` is `1` during the IDLE condition where `|Troom - Tset| < DeltaT_fan`.

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



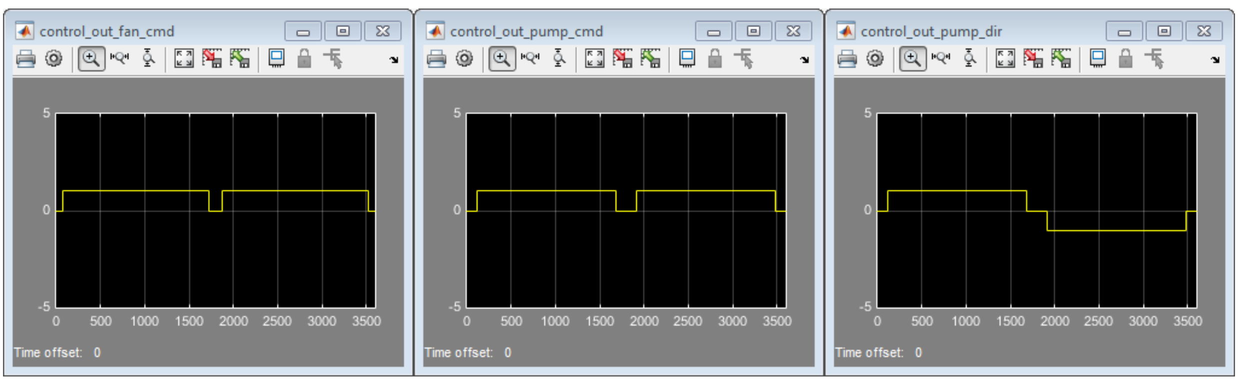
2 In the harness model, open the Controller subsystem.

3 Open controller_chart.

4 In the `IDLE` state, `fan_cmd` is set to return `1`. Change `fan_cmd` to return `0`. `IDLE` is now:

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

5 Simulate the harness model again and observe the outputs.



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

Related Examples

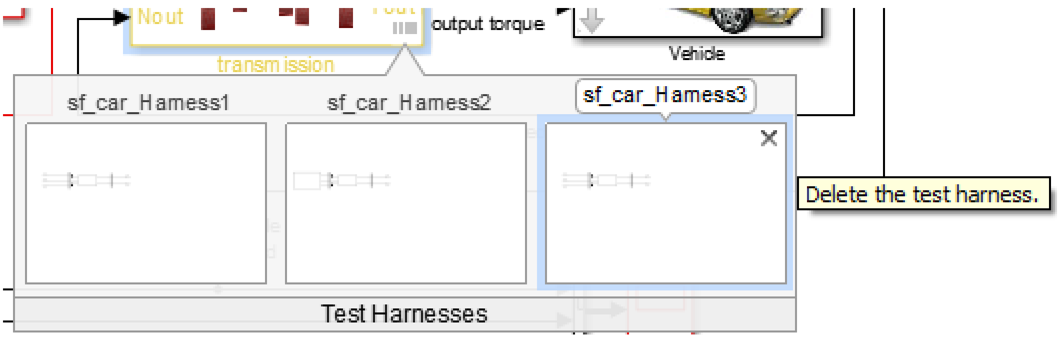
- "Test a Model Component Using Signal Functions"
- "Test Downshift Points of a Transmission Controller"

Manage Test Harnesses

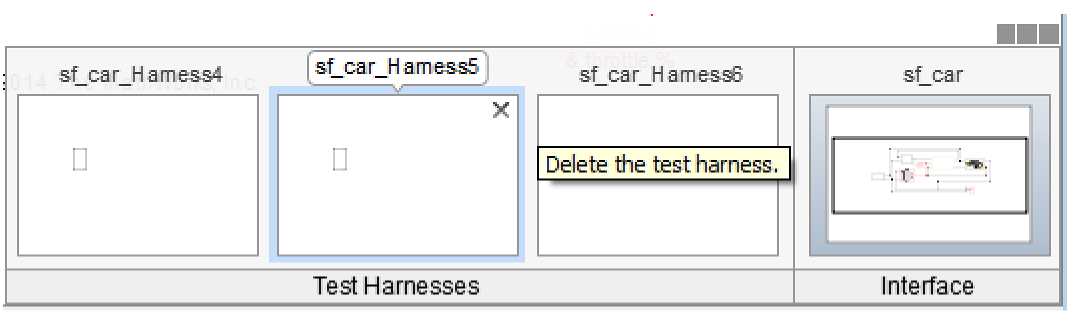
In this section...
"Preview and Open a Test Harness" on page 2-23
"Delete Test Harnesses Programmatically" on page 2-24
"Convert Test Harnesses Into Separate Models" on page 2-25

Preview and Open a Test Harness

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



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



Delete Test Harnesses Programmatically

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

1 Open the model.

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

2 Enter the following at the command line to create two harnesses for the `transmission` subsystem and two harnesses for the `transmission ratio` subsystem.

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

3 Find the harnesses in the `sf_car` model.

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

test_harness_list =

1x4 struct array with fields:

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

4 Delete the harnesses.

```
for k = 1:length(test_harness_list)
```

```
    Simulink.harness.delete(test_harness_list(k).ownerFullPath,...
    test_harness_list(k).name)
end
```

Convert Test Harnesses Into Separate Models

You can convert a test harness block diagram to a separate model, which is useful if you have completed testing but want to preserve the harness design. Select **File > Export Model to > Independent Model For Test Harness**. The harness converts to a separate model containing the blocks from your test harness. Converting removes the harness from your model and breaks the link to the main model.

You can also convert harnesses into separate models programmatically. Programmatic conversion can be useful for handling test harnesses at different hierarchy levels, or for clearing all harnesses from a model without losing the harness designs. This example demonstrates creating four test harnesses for a model and exporting them to separate models.

1. Open the model.

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

2. Enter the following at the command line to create two harnesses for the `transmission` subsystem and two harnesses for the `transmission ratio` subsystem.

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

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

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

   test_harness_list =

   1x4 struct array with fields:

       model
       name
       description
       type
       ownerHandle
   ```

```
    ownerFullPath
    ownerType
    isActive
    canBeActivated
    lockMode
  verificationMode
    saveIndependently
    rebuildOnOpen
    rebuildModelData
```

4 Convert the harnesses into new, separate models. The main model must be saved before each export operation.

```
save_system('sf_car');
for k = 1:length(test_harness_list)
     Simulink.harness.export(test_harness_list(k).ownerFullPath,...
     test_harness_list(k).name,'Name',['test_harness_',num2str(k)]);
     save_system('sf_car');
end
```

See Also

Functions

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

Synchronize Changes Between Test Harness and Model

In this section...

"Maintain SIL or PIL Block Fidelity" on page 2-27
"Synchronize Changes to the Component Under Test" on page 2-27
"Rebuild Test Harness" on page 2-28
"Update Parameters from Test Harness to Model" on page 2-28

A test harness lets you synchronize changes between the test harness and the main model. You can transfer a configuration set and model workspace variables, update the component design, and rebuild the harness to reflect the latest model design. These abilities provide an advantage over isolating a model component in a separate Simulink model.

Maintain SIL or PIL Block Fidelity

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

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

For information about running multiple simulations with unchanged generated code, see "Prevent Code Changes in Multiple SIL and PIL Simulations".

Synchronize Changes to the Component Under Test

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

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

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

Rebuild Test Harness

You can rebuild a test harness to reflect the latest state of the main model. In the test harness, select **Analysis > Test Harness > Rebuild Harness from Main Model**. This operation rebuilds conversion subsystems in the test harness. If the test harness does not have conversion subsystems, this process adds them.

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

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

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

Also see "Select Test Harness Properties for Your Task" and `sltest.harness.rebuild`.

Update Parameters from Test Harness to Model

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

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

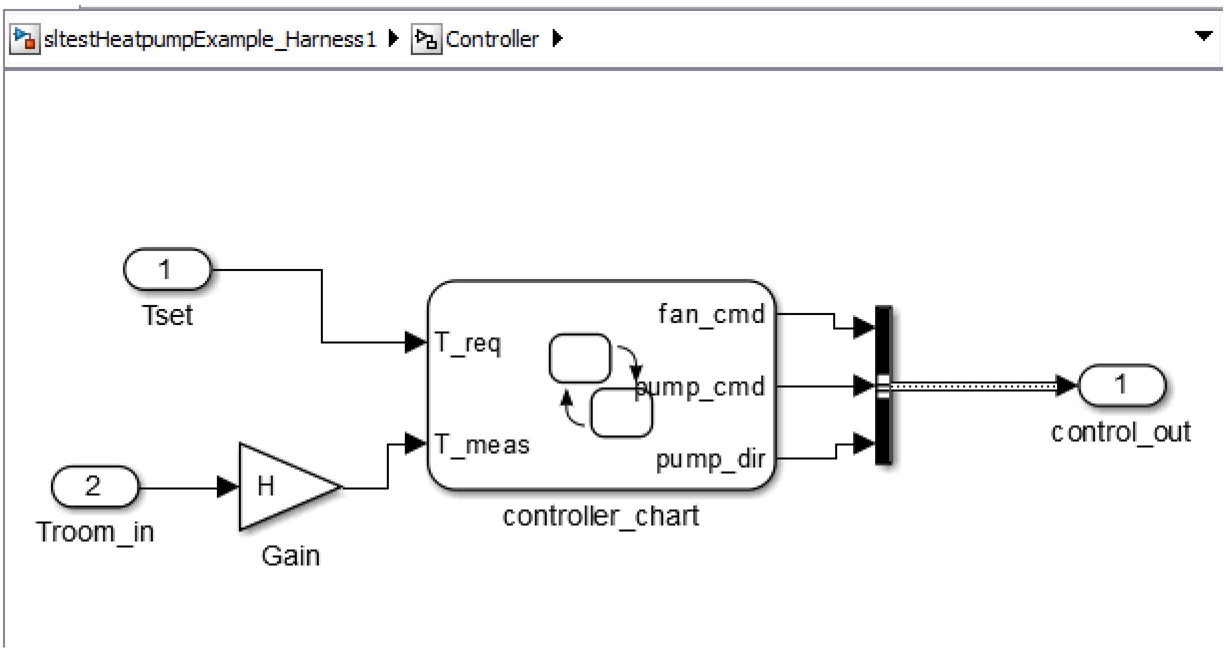
This example shows how to push a new workspace variable to the main model.

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

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

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

   ```
   open_system('sltestHeatpumpExample')
   ```
4. Right-click the `Controller` subsystem and select **Test Harness > Create Test Harness**.
5. In the Create Test Harness dialog box, click **OK** to create a test harness with default properties. The test harness model opens.
6. In the test harness model, select **Tools > Model Explorer** to open the Model Explorer. Expand the items under the test harness name and select **Model Workspace**.
7. Select **Add > MATLAB Variable**. Set the variable name to `H` and the value to `1`.
8. In the top level of the test harness, double-click `Controller` to open the subsystem. Add a Gain block and set the value to `H`. Connect it as shown.



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

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

Related Examples

- "SIL Verification for a Subsystem"

Test Sequences and Assessments

- "Test a Model Component Using Signal Functions" on page 3-2
- "Test Downshift Points of a Transmission Controller" on page 3-6

Test a Model Component Using Signal Functions

In this section...

"Create a Test Sequence" on page 3-2
"Simulate the Test Harness" on page 3-4

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

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

Create a Test Sequence

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

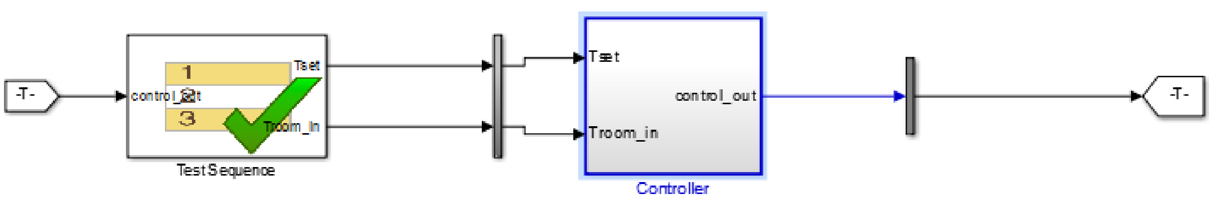
```
cd(fullfile(docroot,'toolbox','sltest','examples'))
```

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

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

3 Open the model, and open the harness.

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



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

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

6 Add a third step to the table. Right-click the `const_90` line, and select **Add step after**. Name the third step `temp_step`.

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

Step	Transition	Next step

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

`ramp_down`
`Tset = 75;`
`Troom_in = 90-ramp(`
	`Troom_in <= 60;`	`temp_step`

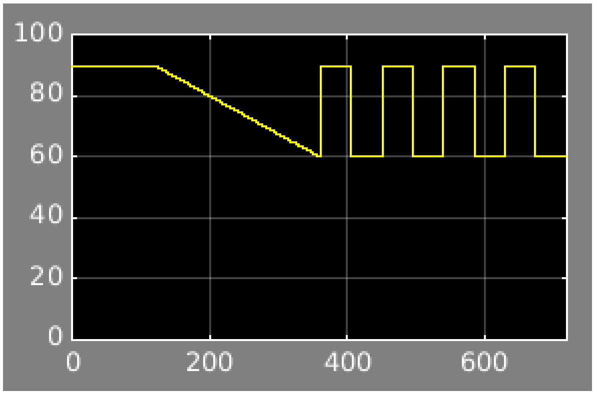
`temp_step`
`Tset = 75;`
`Troom_in = 75+15*s`		

Step	Transition	Next Step
const_90 Tset = 75; Troom_in = 90;	1. after(120,sec)	ramp_down ▼
ramp_down Tset = 75; Troom_in = 90-ramp(et)/8;	1. Troom_in <= 60;	temp_step ▼
temp_step Tset = 75; Troom_in = 75+15*square(et/90);		

Simulate the Test Harness

1 Set the simulation time to `720` sec.

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



See Also

Blocks
Test Sequence

Test Downshift Points of a Transmission Controller

In this section...
"Test Objectives and Model" on page 3-6
"The Test Sequence" on page 3-7
"Add Test Assessments for Controller" on page 3-8
"Test the Controller" on page 3-10

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

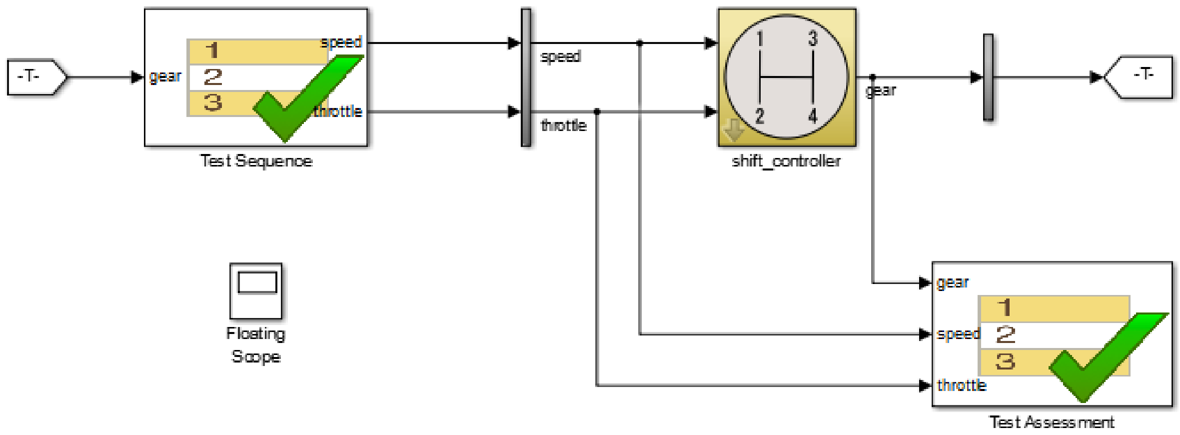
```
cd(fullfile(matlabroot,'help','toolbox','sltest','examples'));
```

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

initialize_4_3
throttle = 10;
speed = 0+ramp(25*et);
	1. speed == 75	down_4_3 ▼

down_4_3
throttle = 10+ramp(10*et);
speed = 75;
	1. hasChanged(gear)	initialize_3_2 ▼

initialize_3_2
throttle = 10;
speed = 45;
Add step after • Add sub-step
	1. after(4,sec)	down_3_2 ▼

down_3_2
throttle = 10+ramp(10*et);
speed = 45;
	1. hasChanged(gear)	initialize_2_1 ▼

initialize_2_1
throttle = 10;
speed = 15;
	1. after(4,sec)	down_2_1 ▼

down_2_1
throttle = 10+ramp(10*et);
speed = 15;
	1. hasChanged(gear)	stop ▼

stop
throttle = 0;
speed = 0;		

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`

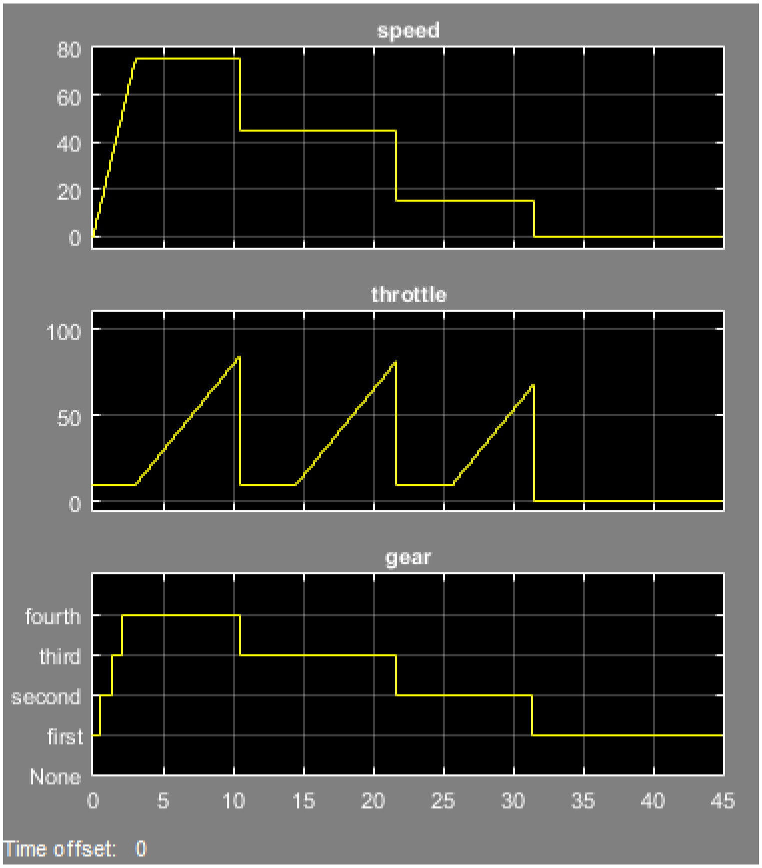
Step	Transition	Next Step

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

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

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 Harness Software- and Processor-in-the-Loop

SIL Verification for a Subsystem

In this section...

"Create a SIL Verification Harness for a Controller" on page 4-3
"Configure and Simulate a SIL Verification Harness" on page 4-4
"Compare the SIL Block and Model Controller Outputs" on page 4-4

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. Additionally, with PIL simulation, you can verify the compiled object code that you intend to deploy in production. You can run the PIL object code on real target hardware or on an instruction set simulator.

If you have an Embedded Coder license, you can create a test harness in SIL or PIL mode for model verification. You can compare the SIL or PIL block results with the model results and collect metrics, including execution time and code coverage. Using the test harness to perform SIL and PIL verification, you can:

- Manage the harness with your model. Generating the test harness generates the SIL block. The test harness is associated with the component under verification. You can save the test harness with the main model.
- Use built-in tools for these test-design-test workflows:
  - Checking the SIL or PIL block equivalence
  - Updating the SIL or PIL block to the latest model design
- View and compare logged data and signals using Simulink Test Manager and Simulation Data Inspector.

For information about running multiple simulations with unchanged generated code, see "Prevent Code Changes in Multiple SIL and PIL Simulations".

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

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

Create a SIL Verification Harness for a Controller

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

1 Open the example model by entering

```
rtwdemo_sil_block
```
at the MATLAB command prompt,

2 Save a copy of the model using the name `controller_model` in a new folder, in a writable location on the MATLAB path.

3 Enable signal logging for the model. At the command prompt, enter

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

4 Right-click the signal into Controller port In1, and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `controller_model_input`. Select **Log signal data** and click **OK**.

5 Right-click the signal out of Controller port Out1, and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `controller_model_output`. Select **Log signal data** and click **OK**.

6 Simulate the model.

7 Get the logged signals from the simulation output into the workspace. At the command prompt, enter

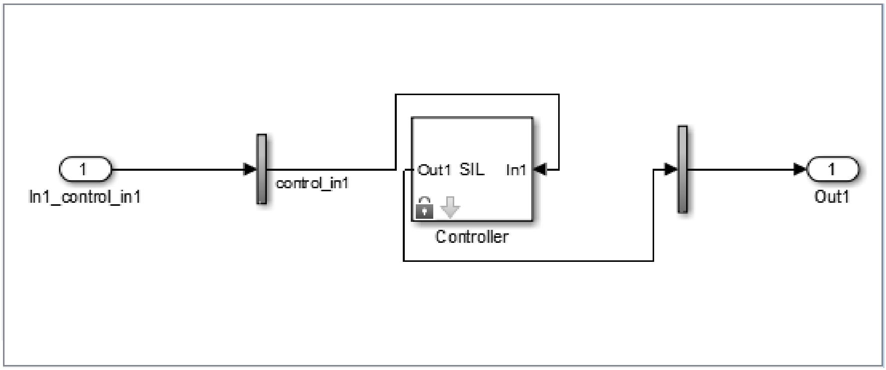
```
out_data = out.get('SIL_signals');
control_in1 = out_data.get('controller_model_input');
control_out1 = out_data.get('controller_model_output');
```

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

9 Set the harness properties:

- **Name**: `SIL_harness`
- **Sources and Sinks**: `Inport` and `Outport`
- **Initial harness configuration**: `Verification`
- **Verification Mode**: `Software-in-the-loop (SIL)`
- Select **Open harness after creation**

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



Configure and Simulate a SIL Verification Harness

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

1. Configure the test harness to import the logged controller input values. From the top level of the test harness, in the model **Configuration Parameters** dialog box, in the **Data Import/Export** pane, select **Input**. Enter `control_in1.Values` as the input and click **OK**.
2. Enable signal logging for the test harness. At the command prompt, enter

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

Compare the SIL Block and Model Controller Outputs

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

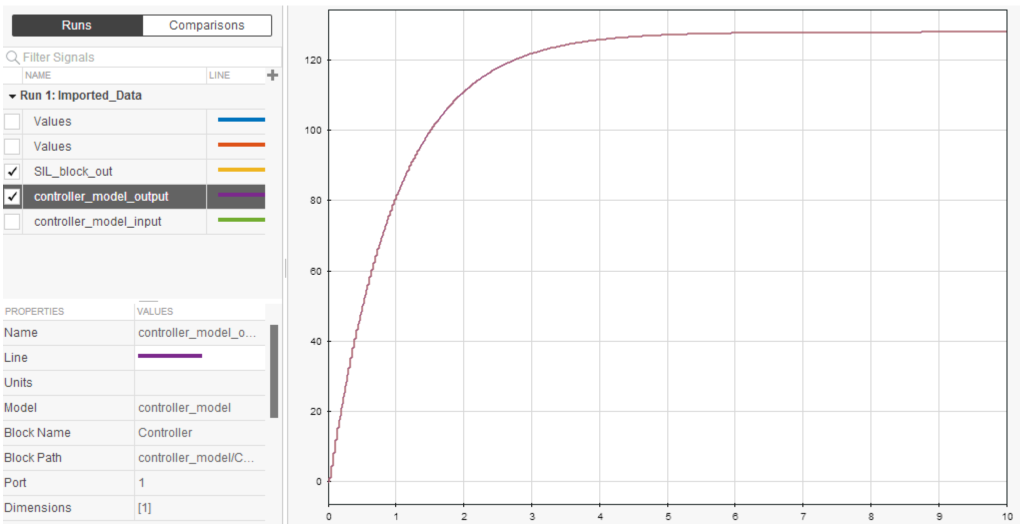
1. In the test harness model, click the Simulation Data Inspector button  to open the Simulation Data Inspector.
2. In the Simulation Data Inspector, click **Import**. In the **Import** dialog box.

- Set **Import from** to: `Base workspace`.
- Set **Import to** to: `New Run`.
- Under **Data to import**, select **Signal Name** to import data from all of the 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 are equivalent. This result means equivalent output and a successful verification for the SIL code. You can plot signal differences using the **Compare** tab in SDI, and perform more detailed analyses for verification. For more information, see "Compare Signal Data from Multiple Simulations" in the Simulink documentation.



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

Simulink Test Manager Introduction

Introduction to the Test Manager

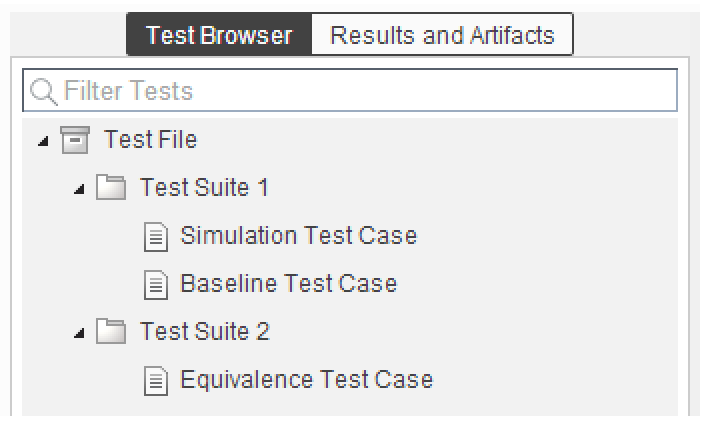
In this section...
"Test Manager Description" on page 5-2
"Test Creation and Hierarchy" on page 5-2
"Test Results" on page 5-3
"Share Results" on page 5-3

Test Manager Description

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

Test Creation and Hierarchy

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



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

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

Test Results

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

Share Results

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

6

Test Manager Test Cases

- "Test Model Output Against a Baseline" on page 6-2
- "Test a Simulation for Run-Time Errors" on page 6-6
- "Generate Test Cases from Model Components" on page 6-9
- "Run Test Cases Programmatically" on page 6-11
- "How Tolerances Are Applied to Test Criteria" on page 6-13
- "Test Manager Limitations" on page 6-14
- "Test Case Sections" on page 6-16

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.

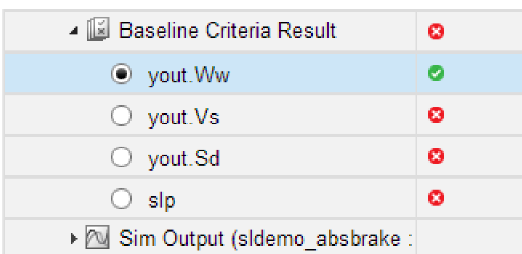
For more information about tolerances and criteria, see “How Tolerances Are Applied to Test Criteria” on page 6-13.

Run the Test Case and View Results

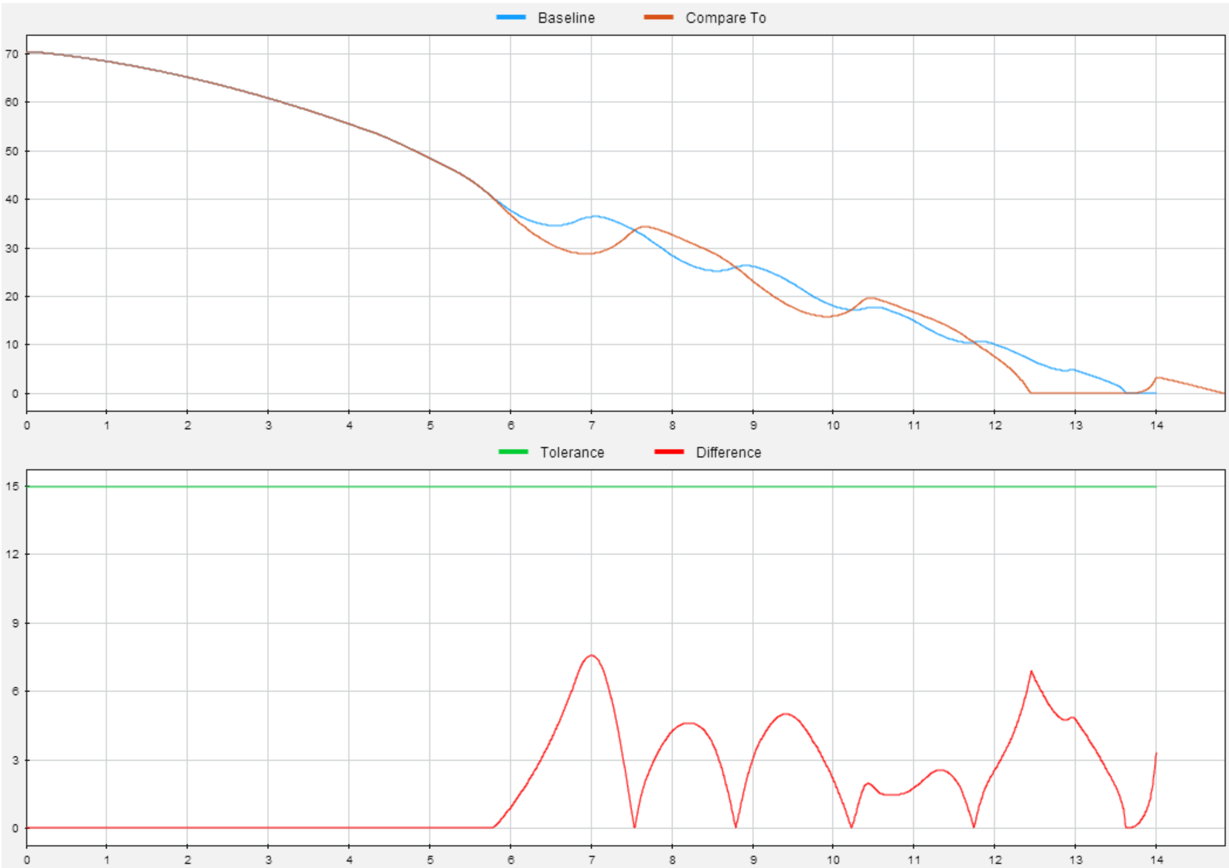
1. In the `sldemo_absbrake` model, set the **Desired relative slip** constant block to `0.22`.
2. In the test manager, select the Slip Baseline Test case in the **Test Browser** pane.
3. On the test manager toolstrip, click **Run** to run the selected test case.

   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.

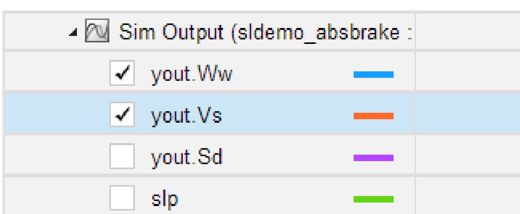
   The signal `yout.Ww` passes, but the overall baseline test fails because other signal comparisons specified in the **Baseline Criteria** section of the test case were not satisfied.
5. To view the `yout.Ww` signal comparison between the model and the baseline criteria, expand `Baseline Criteria Result` and click the option button next to the `yout.Ww` signal.

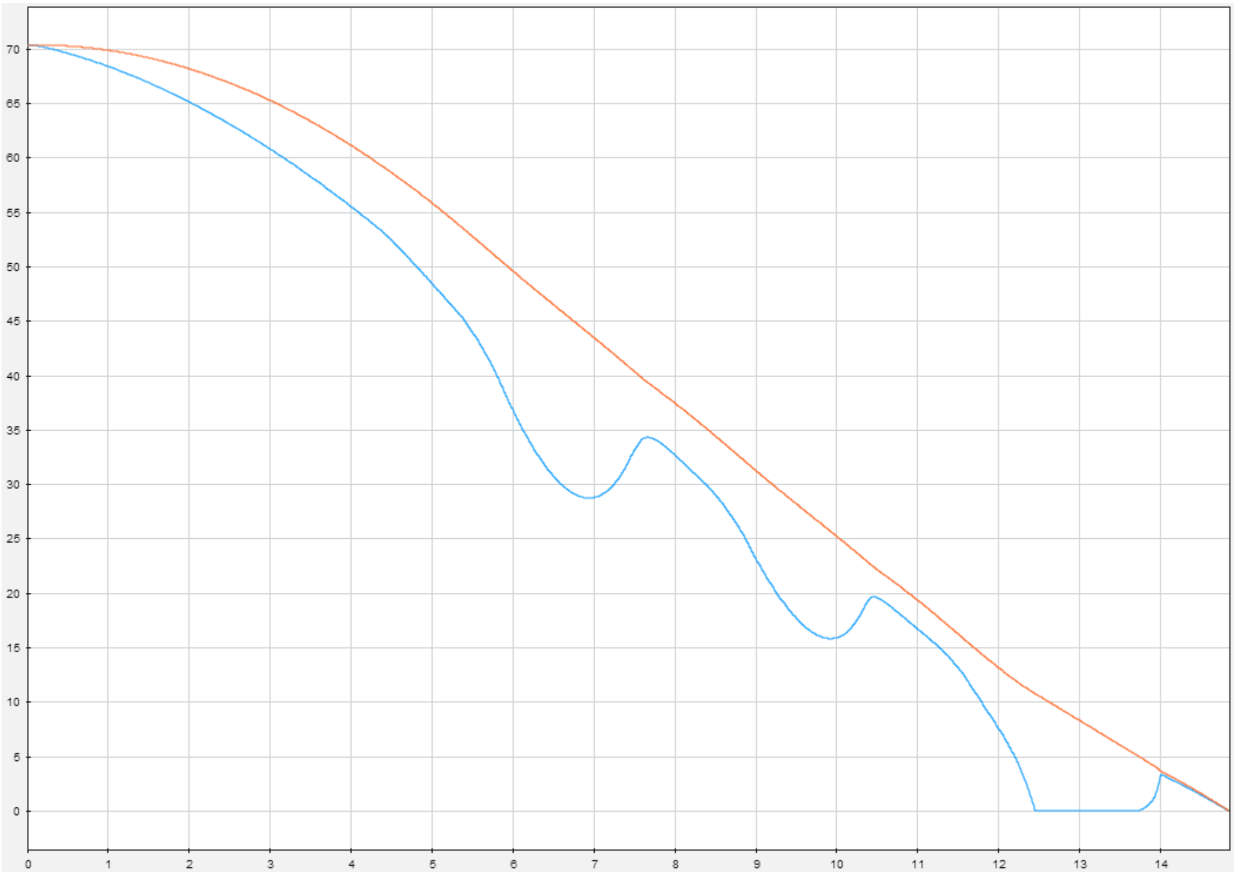
   The **Comparison** tab opens and shows the criteria comparisons for the `yout.Ww` signal.



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”.

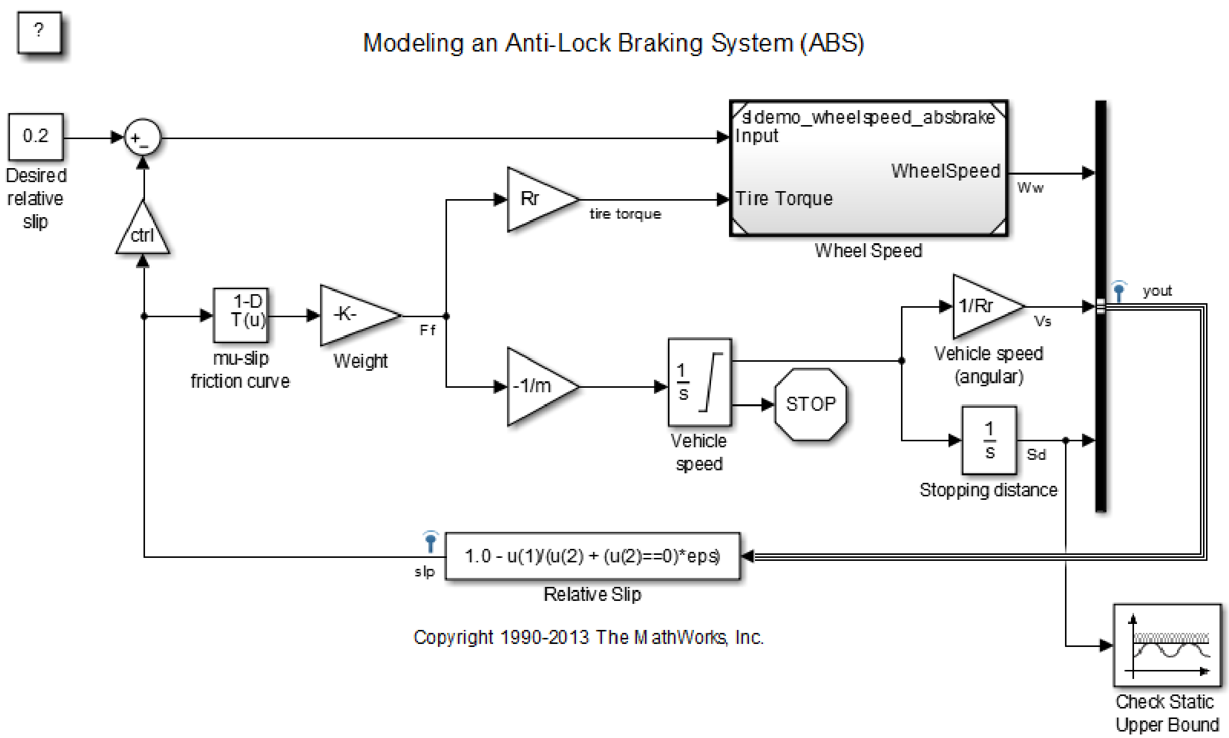
Test a Simulation for Run-Time Errors

In this example, use a simulation test case with the `sldemo_absbrake` model to test for simulation run-time errors. The pass-fail criteria used for a simulation test case is that the simulation finishes without any errors.

Configure the Model

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

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



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

Create the Test Case

1 To open the test manager, from the model, select **Analysis** > **Test Manager**.

2 Click **New** to create a test file. Name and save the test file.

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

3 Select **New** > **Simulation Test**.

4 Right-click the new simulation test case in the **Test Browser** pane, and select **Rename**. Rename the test case to `Upper Bound Test`.

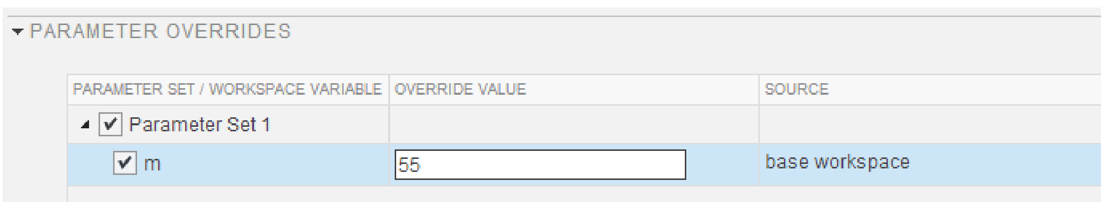
5 In the test case, under **System Under Test**, click the **Use current model** button  to assign the `sldemo_absbrake` model to the test case.

6 Under **Parameter Overrides**, click **Add** to add a parameter set.

7 In the dialog box, click the **Refresh** button  to update the model parameter list.

8 Select the check box next to the workspace variable `m`. Click **OK**.

9 Double-click the **Override Value** and enter `55`.



This value overrides the parameter value in the model when the simulation runs.

Note: To restore the default value of a parameter, clear the value in the **Override Value** column and press **Enter**.

Run the Test Case

1 In the **Test Browser** pane, select the `Upper Bound Test` case.

2 On the test manager toolstrip, click **Run** to run the selected test case.

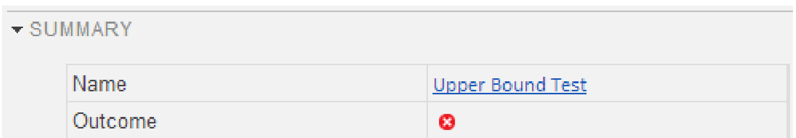
The test manager switches to the **Results and Artifacts** pane, and the new test result appears at the top of the table.

View Test Results

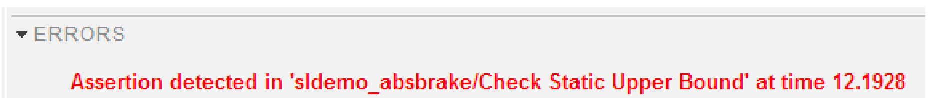
1 Expand the test results, and double-click `Upper Bound Test`.

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

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



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



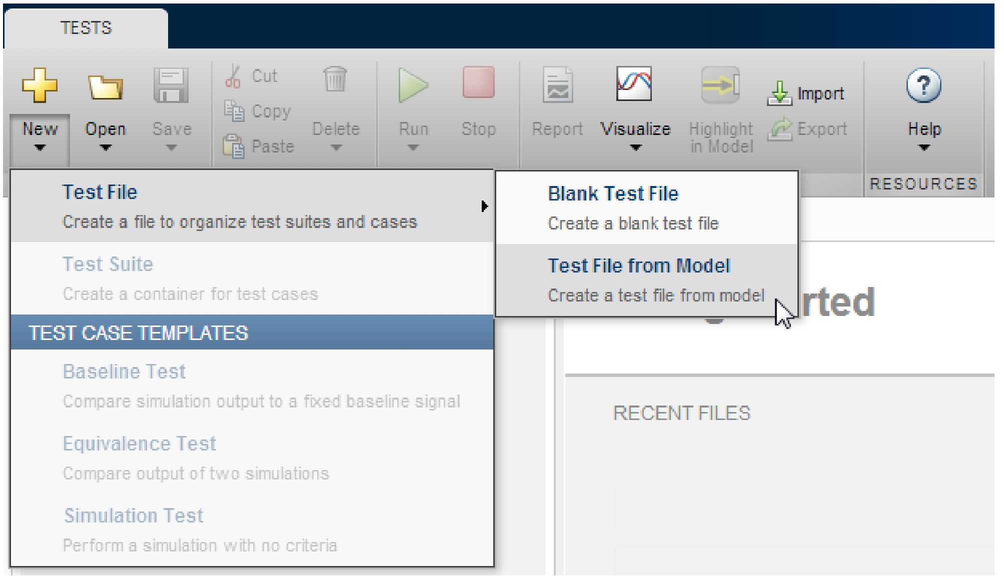
Generate Test Cases from Model Components

The test manager can generate a list of test cases for you based on the components in your model.

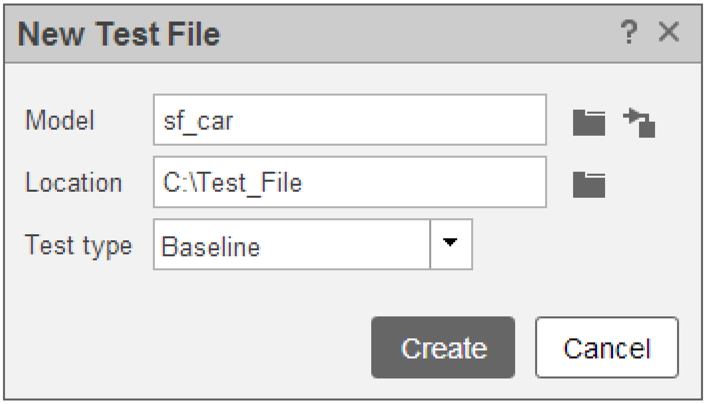
- Signal Builder group in the top model
- Test harnesses from the top model or any subsystem
- Signal Builder group at the top level of a test harness

To generate test cases from your model:

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



2. In the **New Test File** dialog box, specify the **Model**. The model must be on the MATLAB path.
3. Specify the **Location** of the model.
4. Select the **Test type**. All test cases generated will be of the test type specified here.


New Test File
Model
sf_car
Location
C:\Test_File
Test type
Baseline
Create
Cancel

Run Test Cases Programmatically

In this section...
"List of Functions and Classes" on page 6-11
"Run a Test File and Report Results" on page 6-11

List of Functions and Classes

Function	Description
`sltest.testmanager.view`	Launch the Simulink Test manager
`sltest.testmanager.load`	Load a test file in the Simulink Test manager
`sltest.testmanager.run`	Run all test files in the Simulink Test manager
`sltest.testmanager.report`	Generate report of test results
`sltest.testmanager.clear`	Clear all test files from the Simulink Test manager
`sltest.testmanager.close`	Close the Simulink Test manager

Class	Description
`sltest.testmanager.ResultSet`	Access results set data
`sltest.testmanager.TestSuiteResult`	Access test suite results data
`sltest.testmanager.TestCaseResult`	Access test case results data

Run a Test File and Report Results

You can use the `sltest.testmanager` functions for test automation and report generation. If you have authored a test file in the test manager, you can programmatically run the test file, analyze the test results, and generate reports from the results. In this example, the sample test file name is `my_test_file.mldatx`.

Load the test file you authored in the test manager.

```
sltest.testmanager.load('my_test_file.mldatx');
```

Run the test file in the test manager and return an object with results data.

```
resultsObject = sltest.testmanager.run;
```

The results set object gives information about the number of passed, failed, and disabled test cases.

Generate a report from the results data.

```
filePath = 'test_report.pdf';
sltest.testmanager.report(resultsObject,filePath,'Author','Test Engineer');
```

The results report is generated as a PDF and is set to open when it is completed. For more report generation settings, see `sltest.testmanager.report`.

Clear the test manager of all test files.

```
sltest.testmanager.clear;
```

How Tolerances Are Applied to Test Criteria

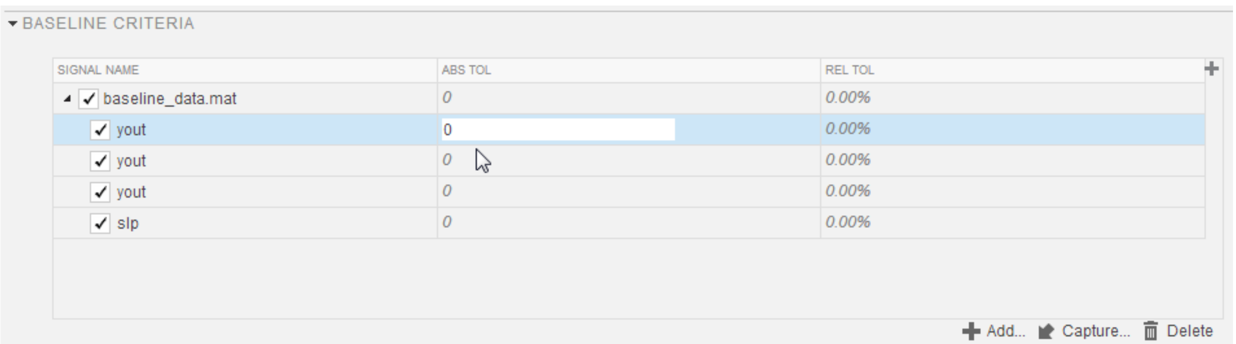
Tolerances can be specified in the **Baseline Criteria** or **Equivalence Criteria** sections of test cases. The default value for the relative tolerance and absolute tolerance for a signal comparison is zero. If you specify tolerances, then the test calculates the tolerances as follows:

```
tolerance = max(absoluteTolerance,relativeTolerance*abs(baselineData));
```

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

Modify Criteria Tolerances

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



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

Test Manager Limitations

In this section...
"Simulation Mode" on page 6-14
"Callback Scripts" on page 6-14
"Simulink Design Verifier Input File" on page 6-14
"Protected Models" on page 6-15

Simulation Mode

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

- The **System Under Test** cannot be in **Fast Restart** mode for test execution.
- A test that is running with the **System Under Test** simulation mode set to **Rapid Accelerator** cannot be stopped using **Stop** on the test manager toolstrip. To stop the test, enter **Ctrl+c** in the MATLAB command prompt.

Callback Scripts

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

- The test manager cannot stop the execution of an infinite loop inside a callback script. To stop execution of an infinite loop from a callback script, press **Ctrl+c** in the MATLAB command prompt.
- `sltest.testmanager` functions are not supported.

Simulink Design Verifier Input File

If you want to use a Simulink Design Verifier simulation data input file in the test manager, then you need to convert the input file to the `DataSet` format using the function `sldvsimdata`. If there are multiple input sets in one Simulink Design Verifier simulation data input file, then you need to separate the input sets into individual input files for each test.

Protected Models

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

Test Case Sections

In this section...
"Description" on page 6-17
"Requirements" on page 6-17
"System Under Test" on page 6-17
"Parameter Overrides" on page 6-18
"Callbacks" on page 6-19
"Inputs" on page 6-19
"Outputs" on page 6-20
"Configuration Settings" on page 6-20
"Simulation 1 and Simulation 2" on page 6-20
"Equivalence Criteria" on page 6-21
"Baseline Criteria" on page 6-21

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

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

Description

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

Requirements

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

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

For more information about the Link Editor, see "Requirements Traceability Link Editor".

System Under Test

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

Note: The model must be available on the path to run the test case. You can set the path programmatically using the pre-load callback. See "Callbacks" on page 6-19.

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

Test Harness

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

For more information about using test harnesses, see "Refine, Test, and Debug a Subsystem".

Simulation Settings

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

Parameter Overrides

You can specify parameter values in the test case to override the parameter values in the model workspace, data dictionary, or base workspace in the **Parameter Overrides** section. Parameters are grouped into sets. Parameter sets and individual parameters overrides can be turned on or off by selecting or clearing the check box next to the set or parameter. To add a parameter override:

1. Click **Add**.

   A dialog box opens with a list of parameters. If the list of parameters is not current, press the **Refresh** button in the dialog box to update the list.
2. Select the parameter you want to override.
3. Click **OK** to add the parameter to the parameter set.
4. Enter the override value in the parameter **Override Value** column.

To restore the default value of a parameter, clear the value in the **Override Value** column and press **Enter**.

You can also add a set of parameter overrides from a MAT-file. Click the **Add** arrow and select `Add File` to create a new parameter set from a MAT-file.

Callbacks

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

- Pre-load: runs before the model loads and any model callbacks.

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

  ```
  addpath(C:\MATLAB\model);
  ```
- Post-load: runs after the model loads and the `PostLoadFcn` model callback.
- Cleanup: runs after simulations and all model callbacks.

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

See "Test Manager Limitations" on page 6-14 for the limitations of callback scripts inside test cases. For information on Simulink model callbacks, see "Model Callbacks".

There are predefined variables available to you in the test case callbacks:

- `sltest_bdroot` available in **Post-Load**: The model simulated by the test case. This can be a harness model.
- `sltest_sut` available in **Post-Load**: The system under test. For a harness, it is the component under test.
- `sltest_isharness` available in **Post-Load**: Returns true if `sltest_bdroot` is a harness model.
- `sltest_simout` available in **Cleanup**: Simulation output produced by simulation.

Inputs

You can override inputs to your **System Under Test**. If you select **Load Inputs from File**, the inputs are mapped using root inport mapping. For more information on root inport mapping see "Import and Map Root-Level Inport Data".

Load Inputs from File

Specify the file name and location of the signal inputs you want to map in the **Location** field. Supported file types are MAT-file and Microsoft® Excel® files. See “Identify Signal Data to Import and Map” for more information on supported file formats.

Select **Map Inputs** to map from the input file specified in **Location**. If you select **Map Inputs**, anything specified in the **Input String** field is overridden.

If **Map Inputs** does not configure the inputs as intended, then select **Mapping Tool** to open the Root Inport Mapping tool for advanced mapping options and detailed information about the mapping status for each signal input.

Signal Builder Group

You can add a signal builder group from a `Signal Builder` in the **System Under Test** model.

Outputs

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

Configuration Settings

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

Note: If you have selected **Override model settings** in the **Outputs** section, then these settings override the output settings in the configuration settings.

Simulation 1 and Simulation 2

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

Equivalence Criteria

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

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

Baseline Criteria

This test case section is only contained in a Baseline test case. Tolerances can be specified in this section to regulate pass-fail criteria of the test. You can specify both absolute and relative tolerances for individual signals or the entire criteria set. Signals in the model marked for streaming and logging are captured. To see tolerances used in an example for baseline criteria, see “Test Model Output Against a Baseline” on page 6-2.

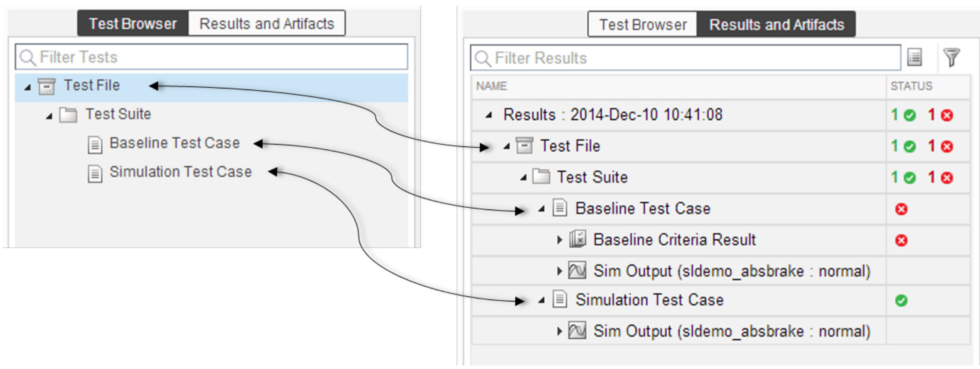
Test Manager Results and Reports

- “View Test Case Results” on page 7-2
- “Export Test Results and Generate Reports” on page 7-9
- “Results Sections” on page 7-11

View Test Case Results

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

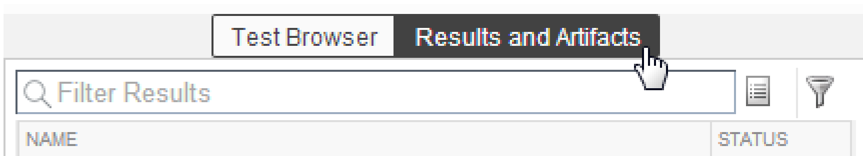
After a test case has finished running in the test manager, the test case result becomes available in the **Results and Artifacts** pane. Test results are organized in the same hierarchy as the test file, test suite, and test cases that were run from the **Test Browser** pane. In addition, the **Results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



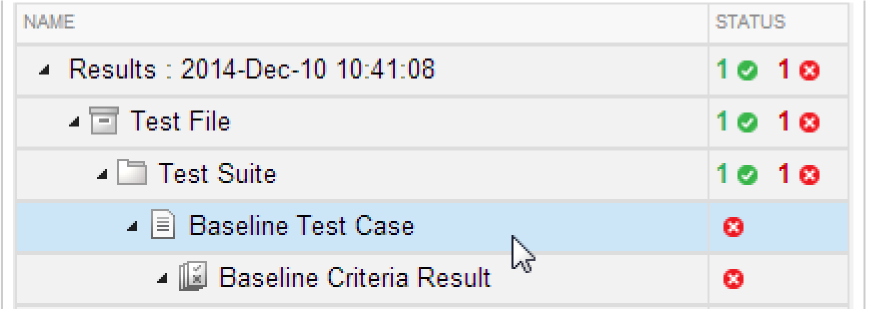
View Results Summary

The test case results tab gives a high-level summary and other information about an individual test case result. To open the test case results tab:

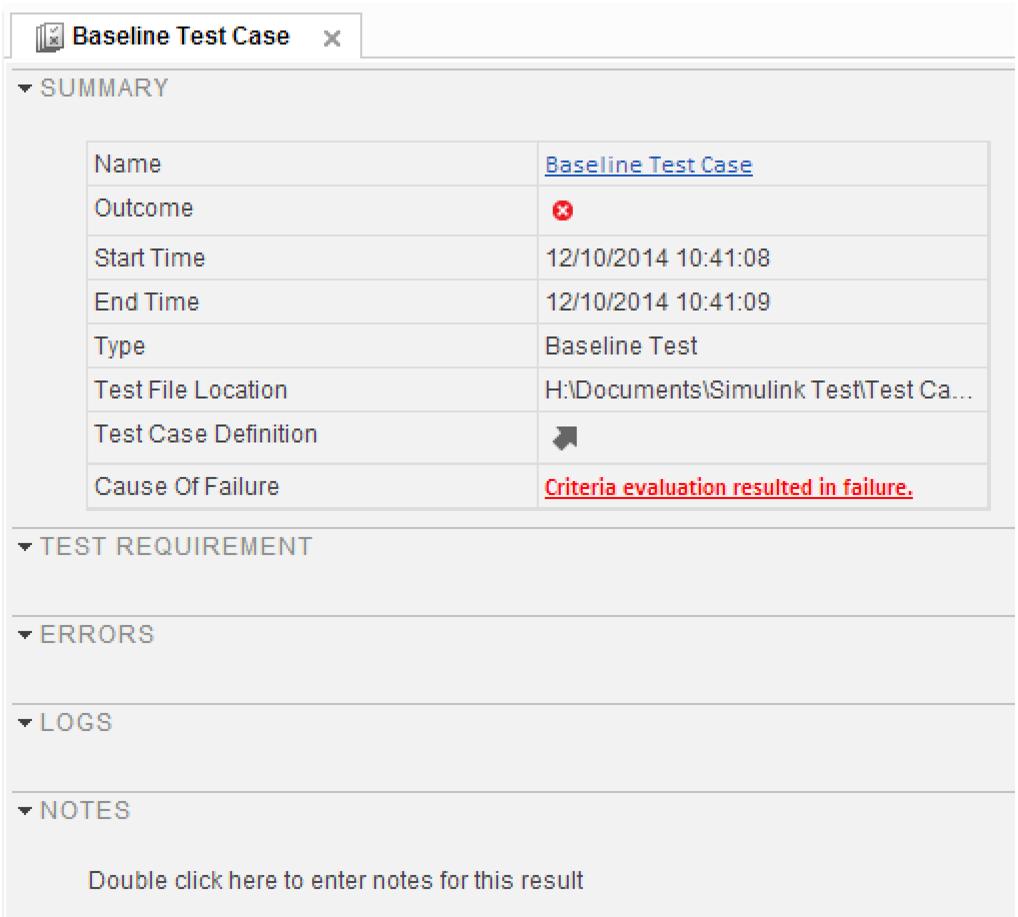
1 Select the **Results and Artifacts** pane.



2 Double-click a test case result.



A tab opens containing the test case results information.



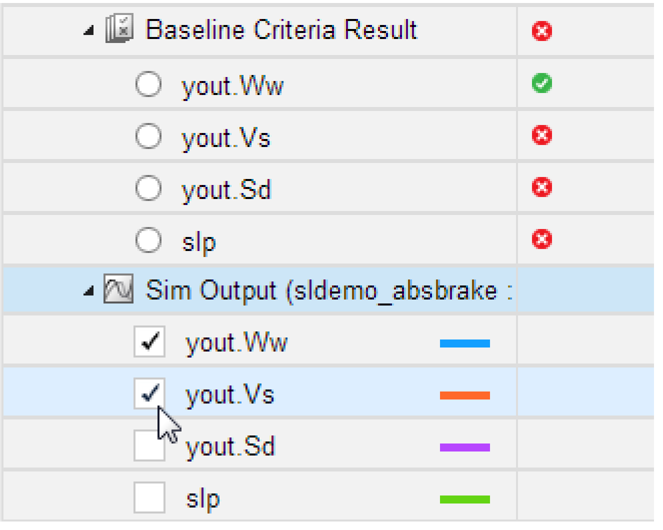
Visualize Test Case Simulation Output and Criteria

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

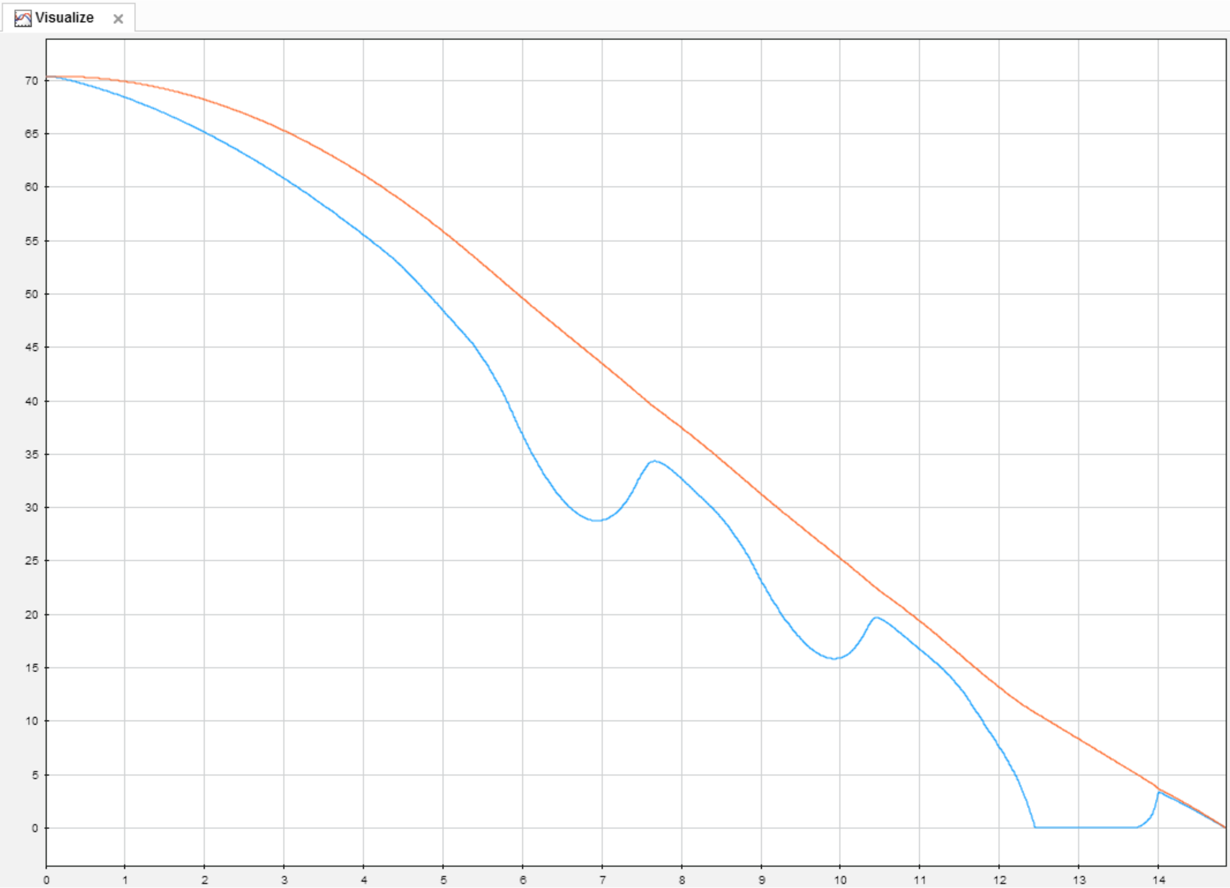
To view simulation output from a test case:

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.

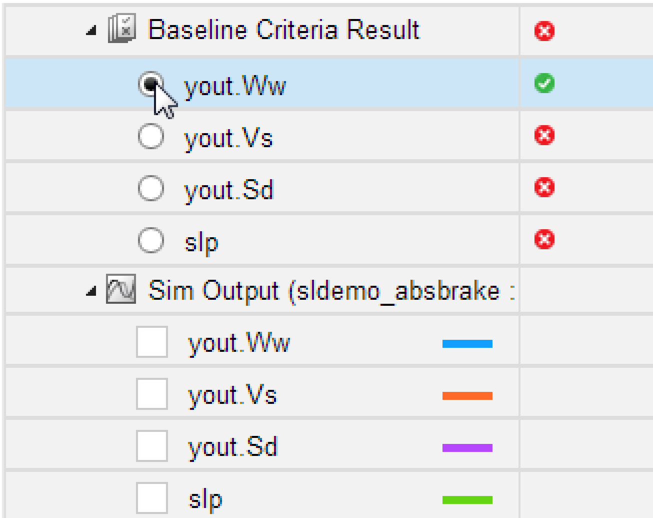


The **Visualize** tab appears and plots the signals.

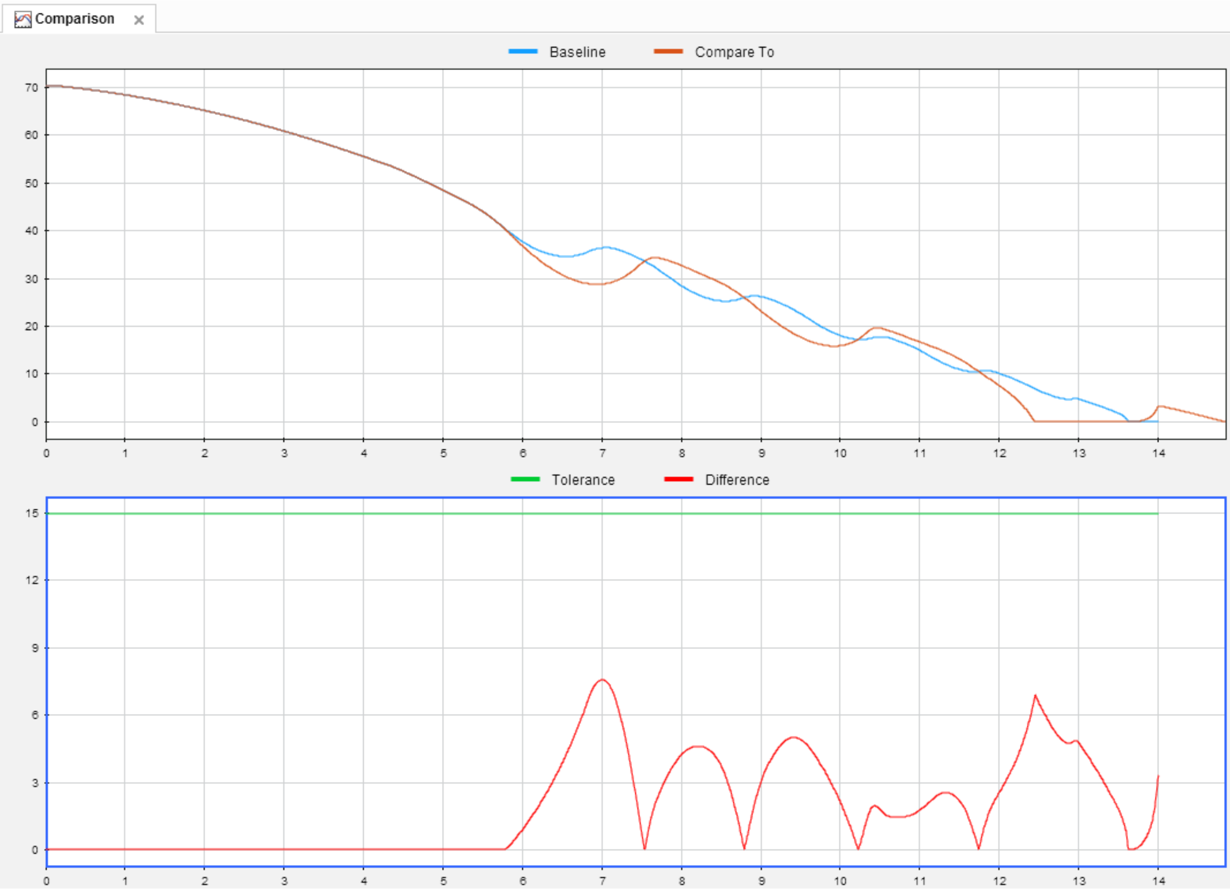


To view equivalence or baseline criteria comparisons:

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.



The **Comparison** tab appears and plots the signal comparison.



To see an example of creating a test case and viewing the results, see “Test Model Output Against a Baseline”.

Export Test Results and Generate Reports

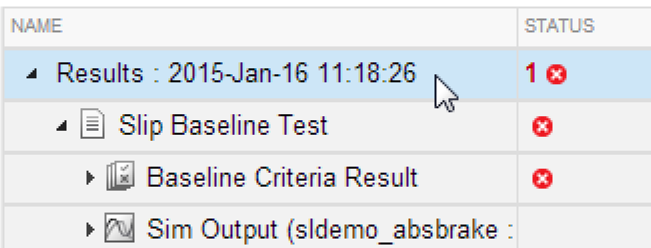
In this section...
"Export Results" on page 7-9
"Create a Test Results Report" on page 7-10

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

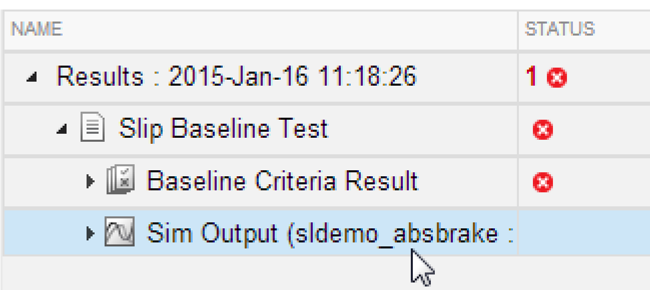
Export Results

Test results are not saved with the test file. To save results, select the result in the **Results and Artifacts** pane, and click **Export** on the toolstrip.

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



- Select criteria comparisons or simulation output to export signal data to the base workspace or to a MAT-file.



Create a Test Results Report

Result reports contain report overview information, the test environment, results summaries with test outcomes, comparison criteria plots, and simulation output plots. You can customize what information is included in the report, and it can be saved in three different file formats: ZIP (HTML), DOCX, and PDF.

To generate a report:

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

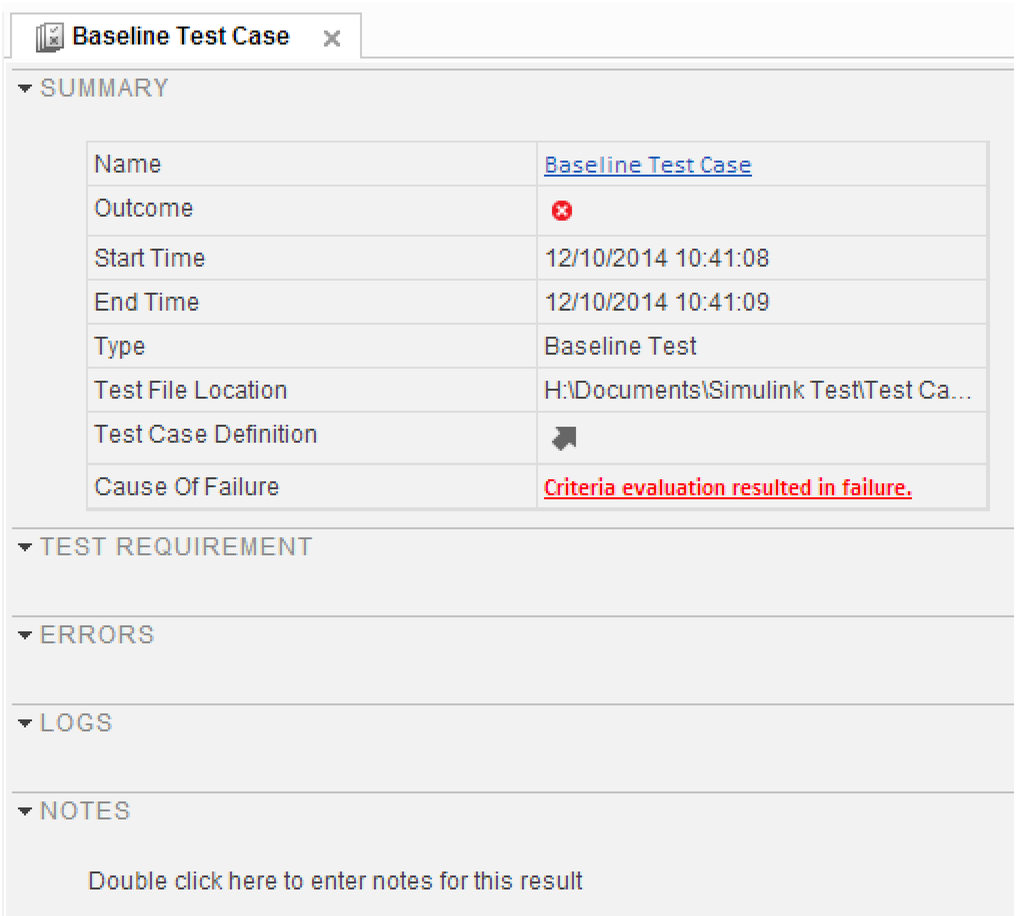
   Note: You can create a report from multiple results sets, but you cannot create a report from multiple test files, test suites, or test cases within results sets.

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

Results Sections

In this section...
"Summary" on page 7-12
"Test Requirement" on page 7-12
"Errors" on page 7-13
"Logs" on page 7-13
"Notes" on page 7-13
"Parameter Overrides" on page 7-13

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



Summary

The **Summary** section includes the basic test information and the test outcome.

Test Requirement

A list of any test requirements linked to the test case. See “Requirements” for more information on linking requirements to test cases.

Errors

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

Logs

These are simulation warnings that are captured from the Simulink Diagnostic Viewer.

Notes

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

Parameter Overrides

A list of any parameter overrides specified in the test case under **Parameter Overrides**. If there are no parameter overrides specified, then this section is not shown in the results summary.