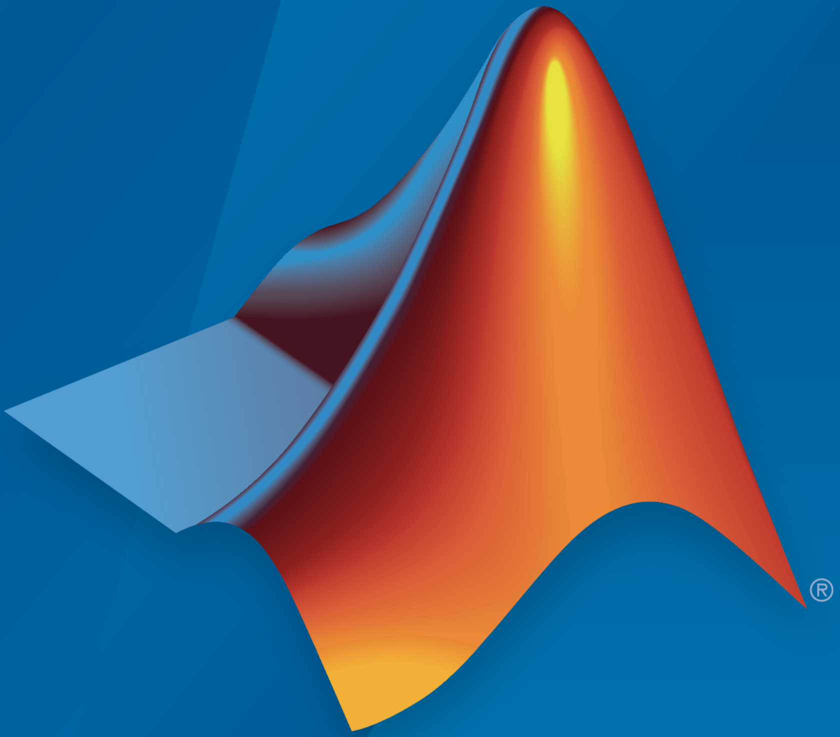
Simulink® Test™

User's Guide



MATLAB® & SIMULINK®

R2015b

MathWorks®

Simulink® Test™ User's Guide

Revision History

March 2015	Online only	New for Version 1.0 (Release 2015a)
September 2015	Online only	Revised for Version 1.1 (Release 2015b)

Contents

7 Test Manager Results and Reports

Test Strategies

Functional Testing in Verification

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

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

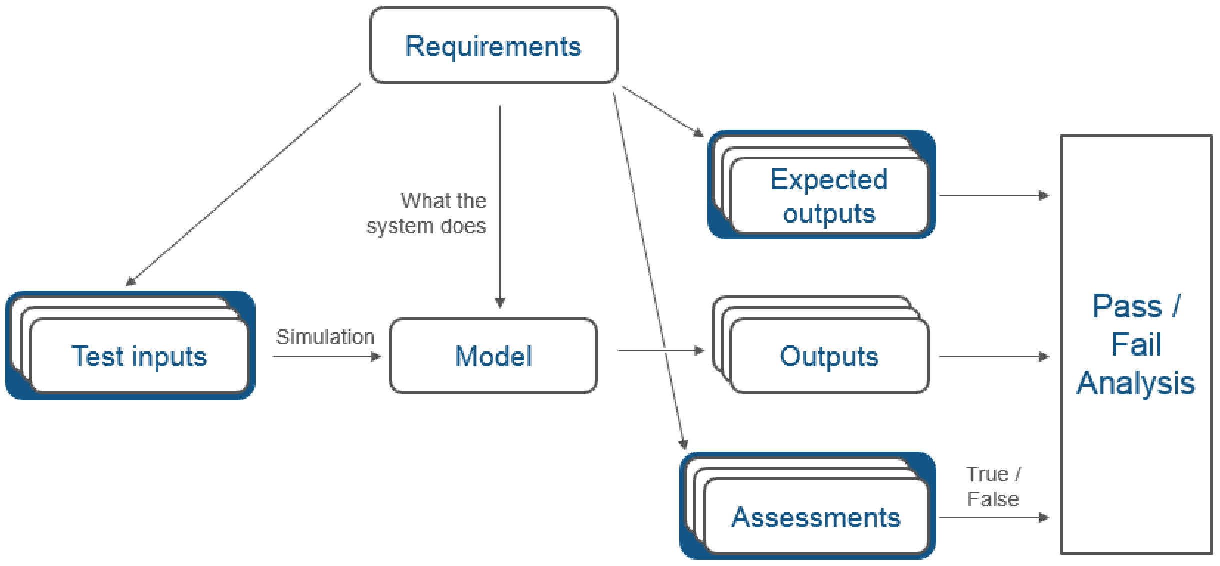
Functional testing can be used to:

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

Common methods of generating test inputs include logging signals from your model, writing test vectors based on requirements, or generating test cases using Simulink® 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.

Requirements-Based Test Cases

Model development begins with detailed requirements that define the system behavior. A well-designed model satisfies requirements without extraneous functionality. Since requirements specify behavior in response to particular conditions, you can build test cases (test inputs, expected outputs, and assessments) from the model requirements. Test cases reproduce specific conditions using test inputs, and assess the actual model output against the expected outputs.



Some requirements apply to multiple test cases. When you create assessments to check such broad requirements, make the assessments reusable. Reusable assessments allow you to manage the same source in multiple test harnesses.

As you develop the model, build test files that check system behavior and link them to corresponding requirements. By defining these test cases in test files, you can periodically check your model and archive results to demonstrate model stability.

Related Examples

- “Organize Test Sequences” on page 3-8

- “Reuse Test Assessments” on page 3-32
- “Requirements-Based Testing for Model Development”

Test Harness

Test Harness and Model Relationship

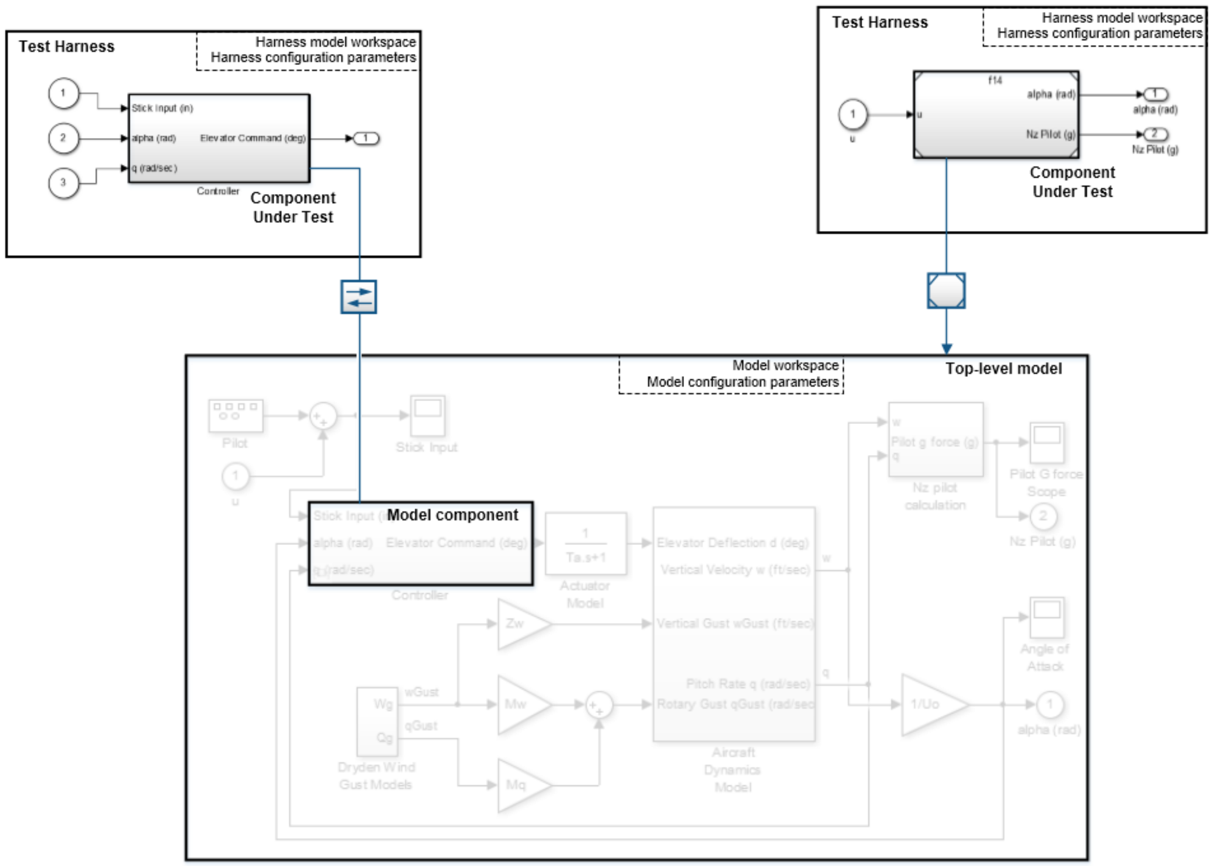
In this section...

Test Harness Description

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

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

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

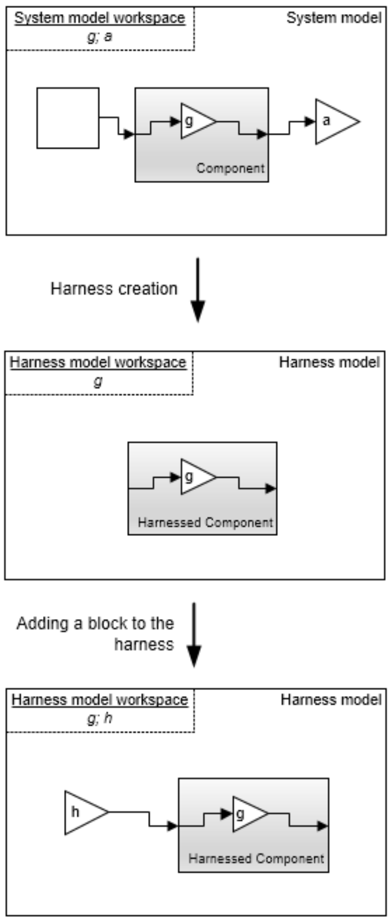


Harness / Model Relationship for a Model Component

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

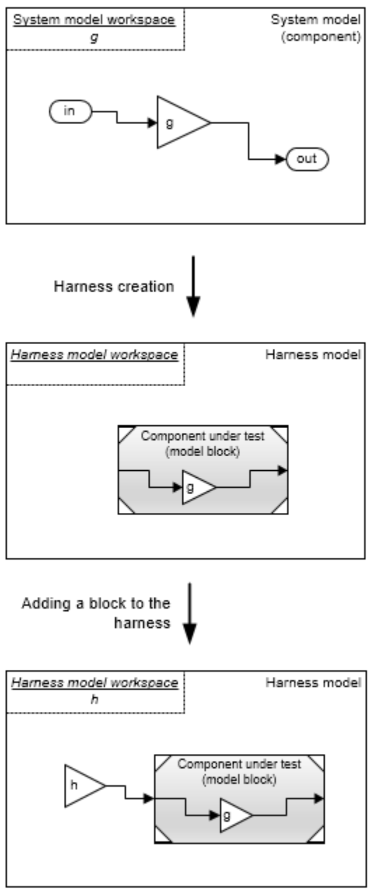
This example shows a test harness for a component that contains a Gain block. The harness model workspace contains a copy of the parameter *g* because *g* defines a part of the component.

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



Harness / Model Relationship for a Top-Level Model

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



Resolving Parameters

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

More About

- "Componentization Guidelines"

Considerations and Limitations

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

Test Harness

- You can open only one test harness at a time per main model.
- Models in MDL format do not support test harness creation. Convert MDL models to SLX format to use test harnesses. Also, SLX models cannot be saved in MDL format. See "Upgrade Model Files to SLX and Preserve Revision History" in the Simulink documentation.
- Do not comment out the component under test in the test harness. Commenting out the component under test can cause unexpected behavior.
- 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" on page 6-22.
- 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.
- The Test Sequence Editor changes the following syntax automatically:

  - Duplicate test step names. For example, if `step_1` already exists, and you change another step name to `step_1`, the step name you change automatically changes to `step_2`.
  - Increment and decrement operations to use MATLAB as the action language, such as `a++` and `a--`. For example, `a++` is changed to `a=a+1`.
  - Assignment operations to use MATLAB as the action language, such as `a+=expr`, `a—=expr`, `a*=expr`, and `a/=expr`. For example, `a+=b` is changed to `a=a+b`.
  - Evaluation operations to use MATLAB as the action language, such as `a!=expr` and `!a`. For example, `a!=b` is changed to `a~=b`.
  - The editor inserts explicit casts for literal constant assignments. For example, if `y` is defined as type `single`, then `y=1` is changed to `y=single(1)`.

Select Test Harness Properties for Your Task

In this section...

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 respective 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 blocks 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 common for model verification. The test harness prevents you from editing the component under test, and the test harness rebuilds every time you open it. In addition to a normal subsystem, you can choose a SIL or PIL block as the component under test (requires Embedded Coder®). See "Verification Modes" on page 2-10.

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.	The test harness might require additional modification for it 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” on page 2-30.
Update Configuration Parameters and Model Workspace data on rebuild	When you select this property, configuration parameters and model workspace data update when you rebuild the harness.	For details on the rebuild process, see “Synchronize Changes Between Test Harness and Model” on page 2-30.
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

Click the badge in the test harness block diagram and click **Test harness properties...** to open the harness properties dialog box.

See Also

Test Sequence | "Synchronize Changes Between Test Harness and Model" on page 2-30

Test Harness Parameters and Signals

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

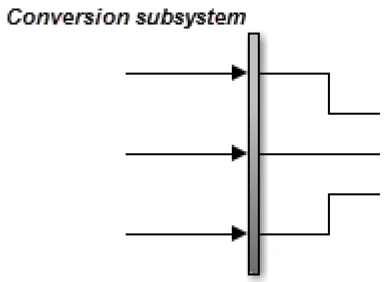
Test Harness Generation Without Compilation

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 2-30.

Signal Conversion Subsystem



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

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

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

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

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

Refine, Test, and Debug a Subsystem

In this section...

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

Model and Requirements

1 Access the model. Enter

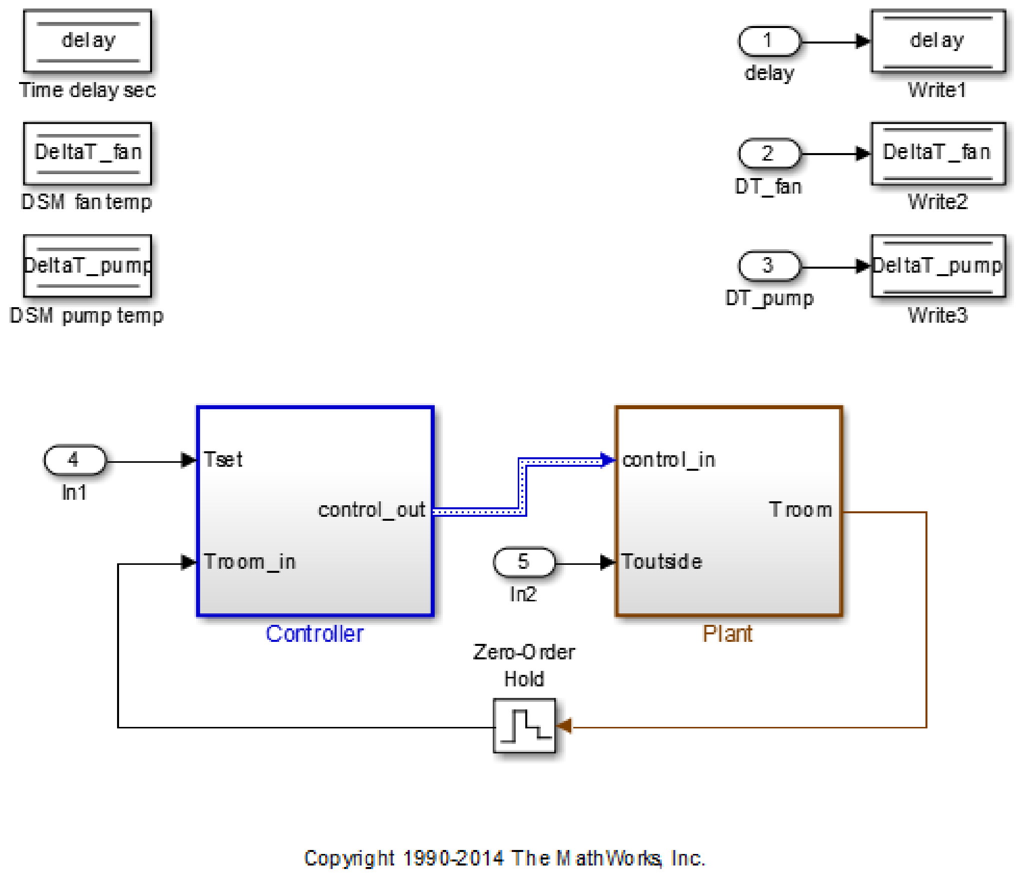
```
cd(fullfile(docroot,'toolbox','sltest','examples'))
```

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

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

3 Open the model.

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



In the example model:

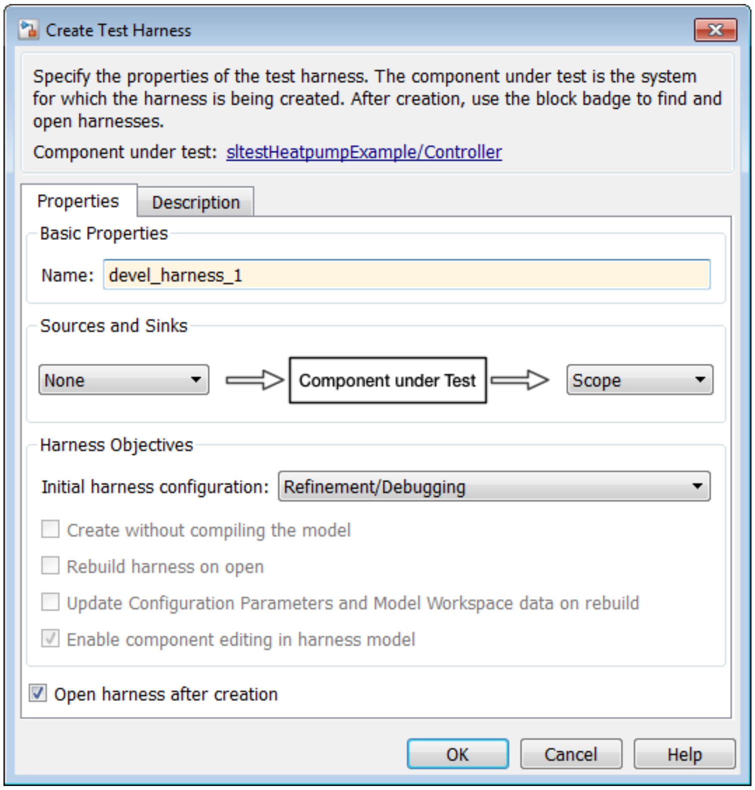
- The controller accepts the room temperature and the set temperature inputs.
- The controller output is a bus with signals controlling the fan, heat pump, and the direction of the heat pump (heat or cool).
- The plant accepts the control bus. The heat pump and the fan signals are Boolean, and the heat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



Create Test Harness

Specify the properties of the test harness. The component under test is the system for which the harness is being created. After creation, use the block badge to find and open harnesses.

Component under test: sltestHeatpumpExample/Controller

Basic Properties

Name: devel_harness_1

Sources and Sinks

None → Component under Test → Scope

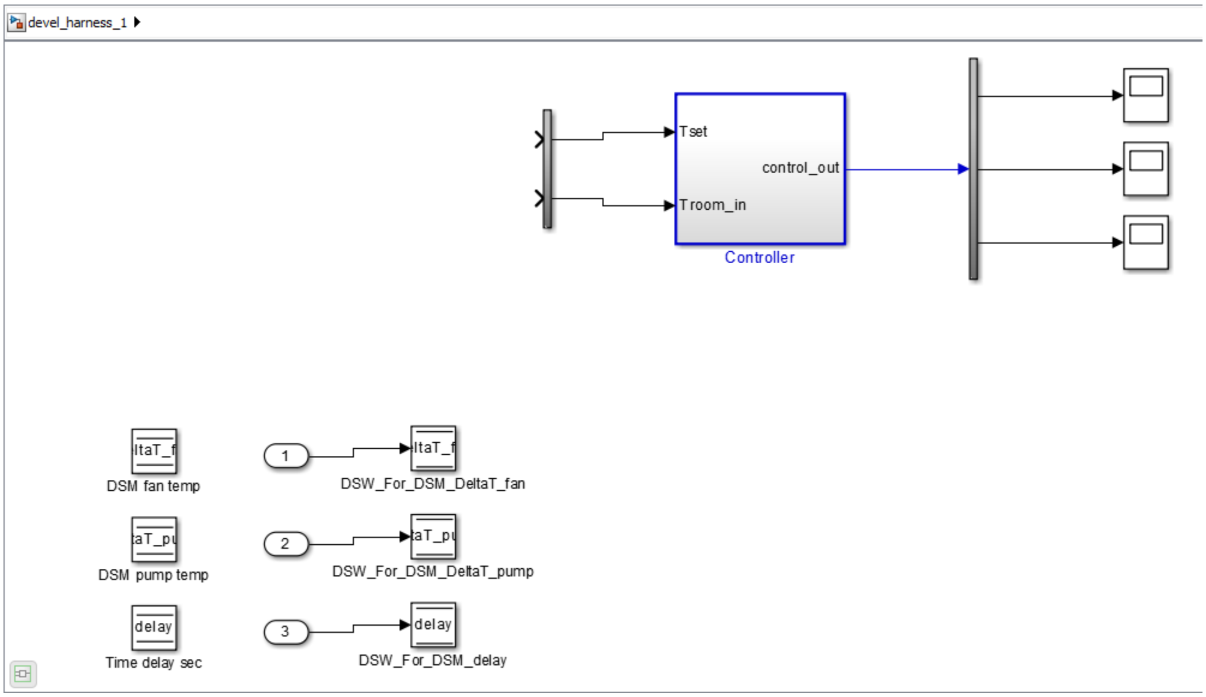
Harness Objectives

Initial harness configuration: Refinement/Debugging

- [ ] Create without compiling the model
- [ ] Rebuild harness on open
- [ ] Update Configuration Parameters and Model Workspace data on rebuild
- [x] Enable component editing in harness model

- [x] Open harness after creation

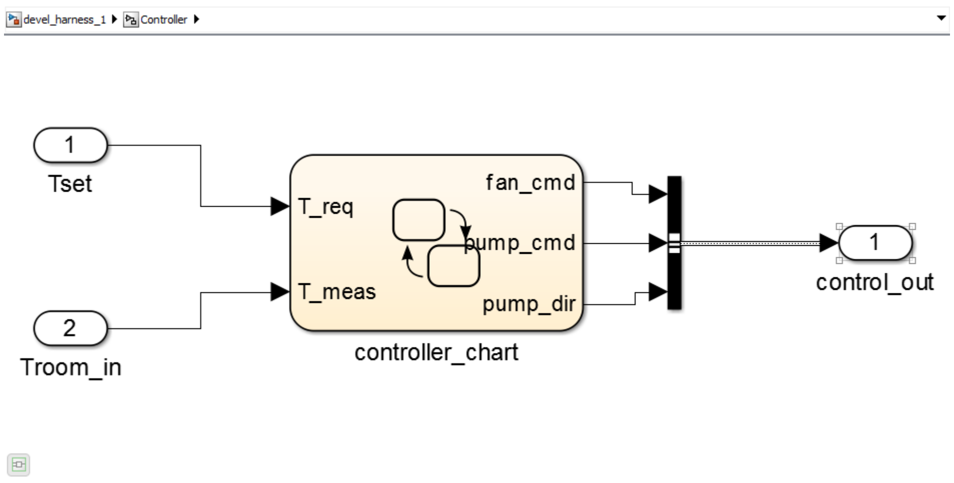
3 Click **OK** to create the test harness.



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

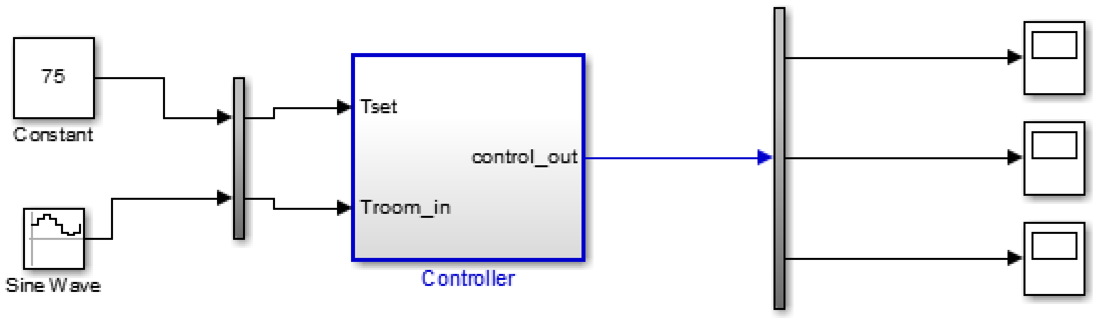
1 Navigate to the top level of `devel_harness_1`.

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`.

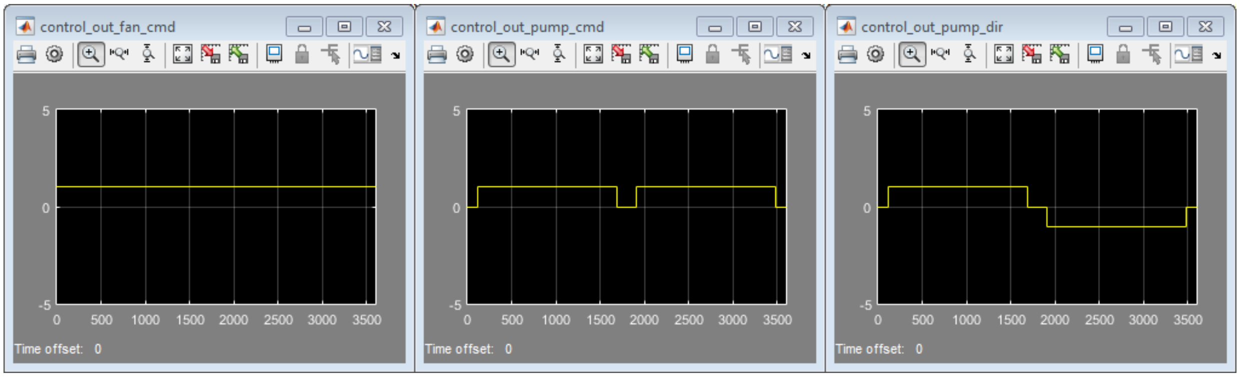
7 Open the three scopes in the harness model.

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`.



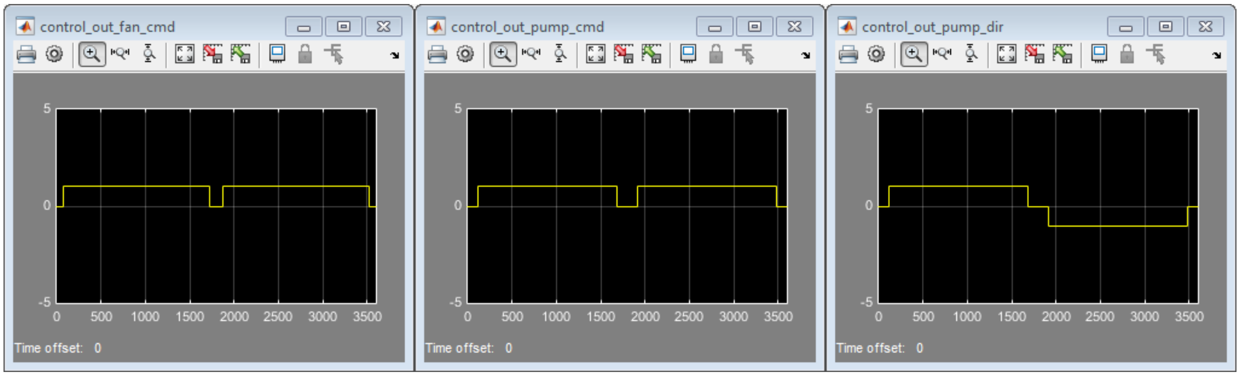
2 In the harness model, open the Controller subsystem.

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

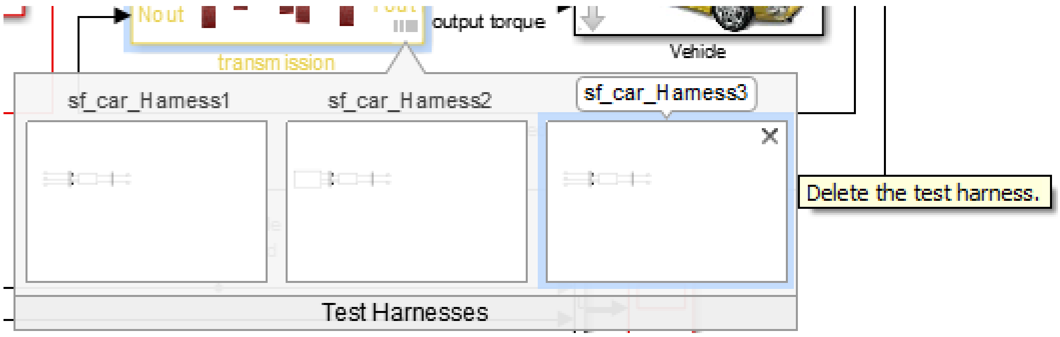
- "Test a Model Component Using Signal Functions" on page 3-22
- "Test Downshift Points of a Transmission Controller" on page 3-26

Manage Test Harnesses

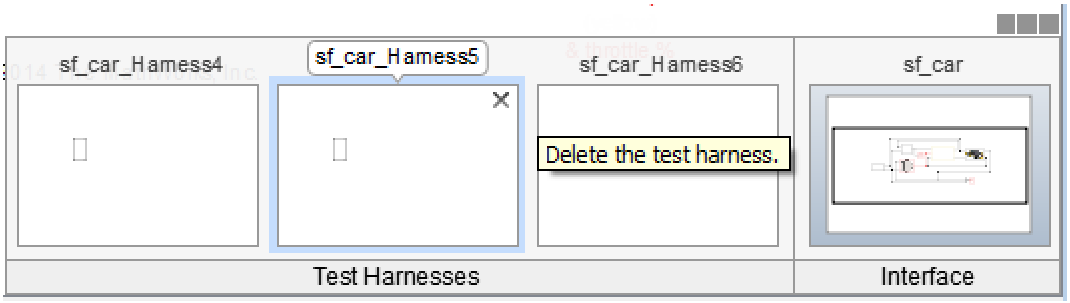
In this section...

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.

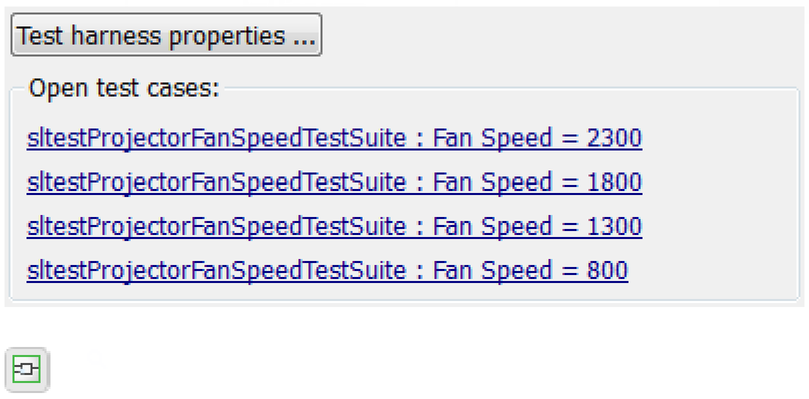


When a model block diagram has a test harness, click the pullout icon in the model canvas to preview the test harnesses. Click a thumbnail to open the harness



Find Test Cases Associated with a Test Harness

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


Test harness properties ...

Open test cases:

sltestProjectorFanSpeedTestSuite : Fan Speed = 2300
sltestProjectorFanSpeedTestSuite : Fan Speed = 1800
sltestProjectorFanSpeedTestSuite : Fan Speed = 1300
sltestProjectorFanSpeedTestSuite : Fan Speed = 800

Delete All Test Harnesses in a Model

You can delete a harness manually, 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 test 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
```

Clone and Export a Test Harness to a Separate Model

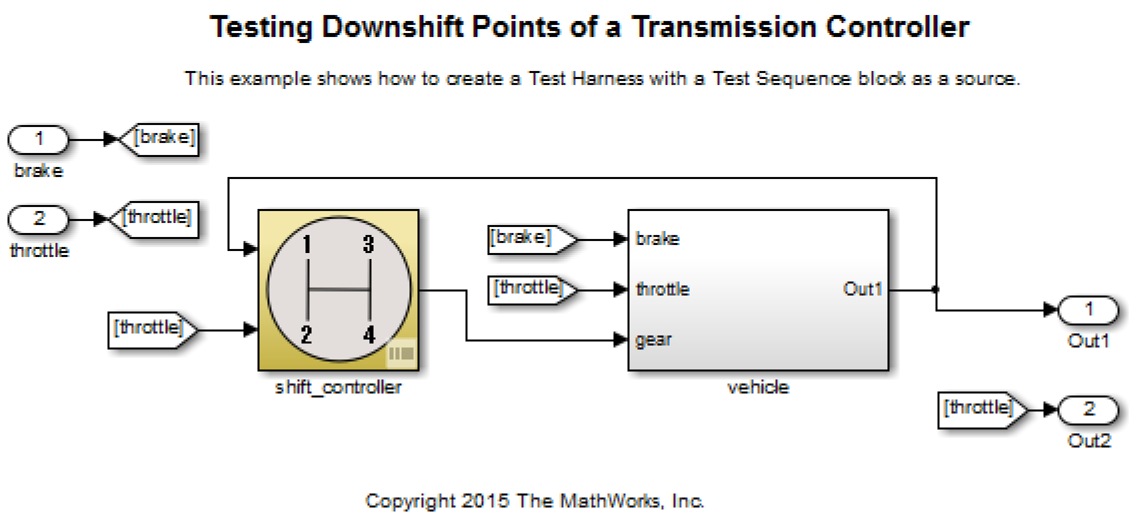
This example demonstrates cloning an existing test harness and exporting the cloned harness to a separate model. This can be useful if you want to create a copy of a test harness as a separate model, but leave the test harness associated with the model component.

High-level Workflow

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

Open the Model and Save a Local Copy

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


Testing Downshift Points of a Transmission Controller
This example shows how to create a Test Harness with a Test Sequence block as a source.
1
brake
[brake]
2
throttle
[throttle]
[throttle]
1
3
2
4
shift_controller
[brake]
[throttle]
brake
throttle
gear
Out1
vehicle
1
Out1
[throttle]
2
Out2
Copyright 2015 The MathWorks, Inc.

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

Get the Properties of the Source Test Harness

```
Properties = sltest.harness.find([Model '/shift_controller'])
```

```
Properties = 

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

Clone the Test Harness

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

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

Save the Model

Before exporting the harness, save changes to the model.

```
save_system(Model)
```

Export the Test Harness to a Separate Model

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

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

clear('Model');
```

```
bdclose all;
```

See Also

Functions

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

Synchronize Changes Between Test Harness and Model

In this section...

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

Maintain SIL or PIL Block Fidelity

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

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

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

Synchronize Changes to the Component Under Test

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

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

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

Rebuild Test Harness

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

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

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

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

Also see “Select Test Harness Properties for Your Task” on page 2-8 and `sltest.harness.rebuild`.

Update Parameters from Test Harness to Model

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

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

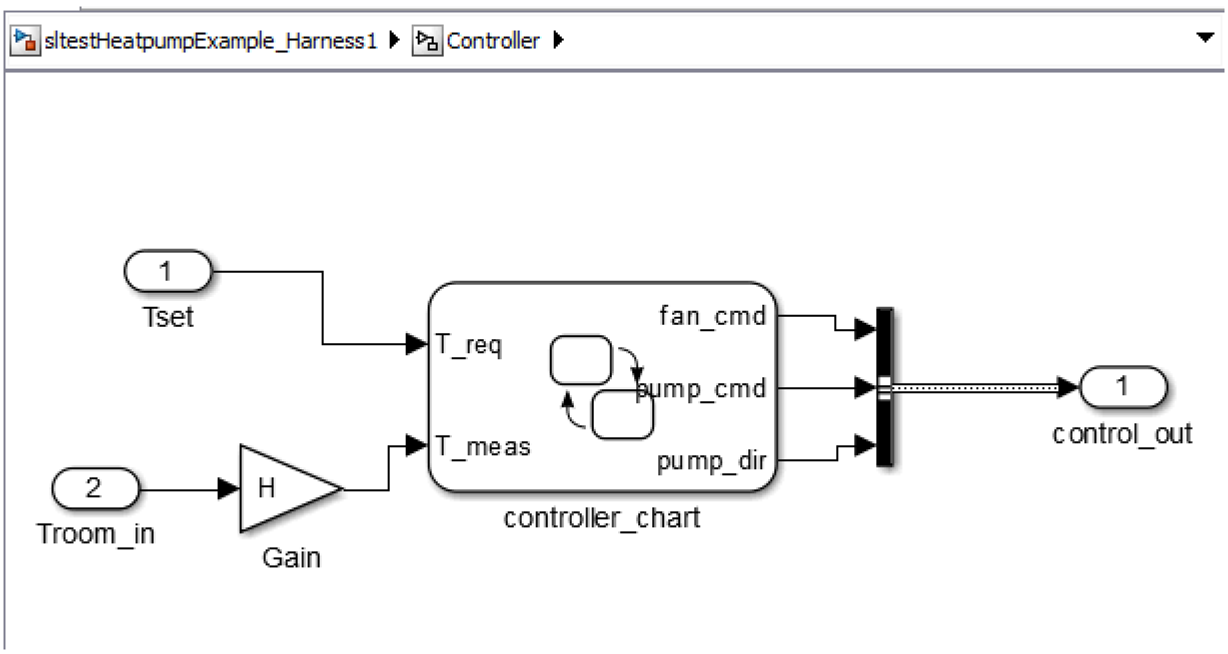
This example shows how to push a new workspace variable to the main model.

1. Access the model. Enter

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

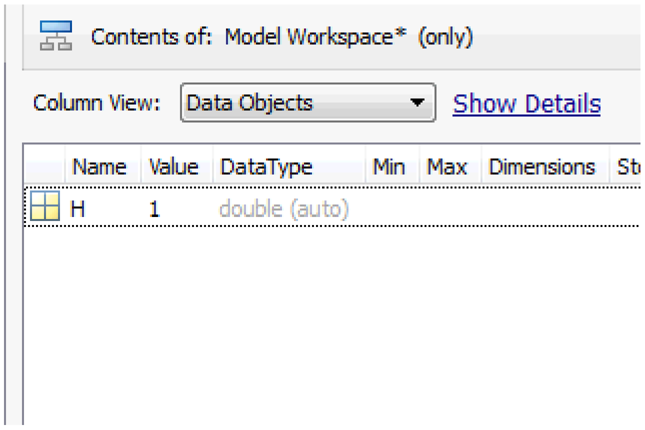
   ```
   sltestHeatpumpExample.slx
   sltestHeatpumpBusPostLoadFcn.mat
   PumpDirection.m
   ```
3. Open the model.

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



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

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


Contents of: Model Workspace* (only)

Column View: Data Objects

Name	Value	DataType	Min	Max	Dimensions	St
H	1	double (auto)				

Related Examples

- "SIL Verification for a Subsystem" on page 4-2

Test Sequences and Assessments

Introduction to Test Sequences

In this section...

You can use the Test Sequence block to specify test steps, actions, and transitions. With time-series inputs, you supply time-defined test vectors. However, the test sequences you create can react to signal and temporal conditions. You can also use them to assess simulation.

Structure of a Test Sequence

A test sequence consists of test steps arranged in a hierarchy. You can use transitions to define the test sequence progression within a hierarchy level.

A test step contains actions and transitions you define using MATLAB as the action language. Actions execute at the beginning of the step. You use actions to define commands for each test step, such as setting signal levels, verifying logical conditions, or setting variables. You use test step transitions to define conditions that determine when the test sequence exits the current step and enters another step.

A standard transition occurs on a condition that you specify. Once the step exits, the next step that you specify executes.

Test Sequence Hierarchy

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

Step Transitions

In a test sequence, the top hierarchy level uses a standard transition. Test sequence execution begins with the top step in the group, and proceeds according to the transition conditions and next steps.

You can change lower-level groups to switch between steps based on signal conditions defined in the step. This switching condition is called a When decomposition. In this case, the parent step evaluates, and then the substeps execute based on their associated conditions. The conditions determine the order in which the substeps execute. For example, the first substep in the table does not necessarily execute first. If multiple steps in a When decomposition group have conditions that are true, the highest step with the true condition is active.

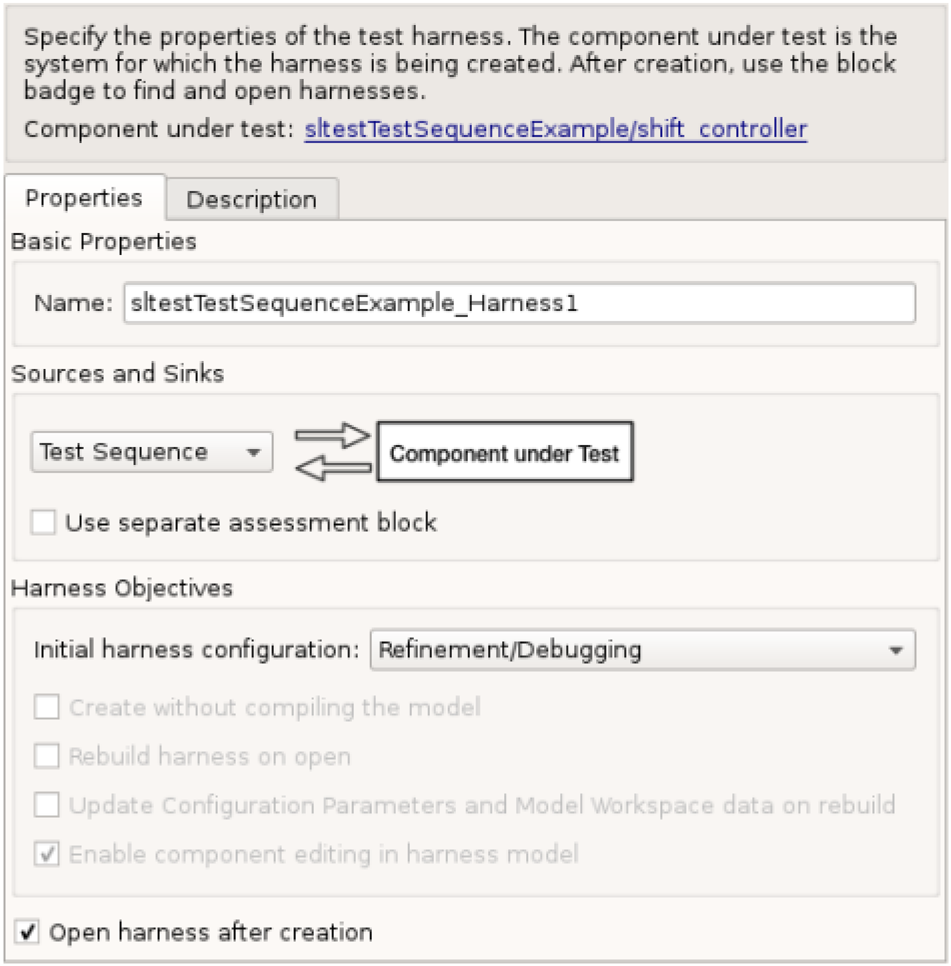
Create a Basic Test Sequence

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

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

   ```
   sltestTestSequenceExample
   ```
2. Right-click the shift_controller subsystem and select **Test Harness** > **Create Test Harness**.
3. In the Create Test Harness dialog box, under **Sources and Sinks**, change `Inport` to `Test Sequence`.

   The schematic displays the closed-loop configuration between the Test Sequence block and the component under test.


Specify the properties of the test harness. The component under test is the system for which the harness is being created. After creation, use the block badge to find and open harnesses.

Component under test: sltestTestSequenceExample/shift_controller

Properties | Description

Basic Properties

Name: sltestTestSequenceExample_Harness1

Sources and Sinks

Test Sequence ⇄ Component under Test

☐ Use separate assessment block

Harness Objectives

Initial harness configuration: Refinement/Debugging

☐ Create without compiling the model

☐ Rebuild harness on open

☐ Update Configuration Parameters and Model Workspace data on rebuild

☑ Enable component editing in harness model

☑ Open harness after creation

4 Click **OK**. The test harness for the `shift_controller` subsystem opens. Double-click the Test Sequence block.

The Test Sequence Editor opens and displays action and transition tips. Click the `X` to close the tips. The first line in a **Step** cell defines the step name.

5 Create the test sequence.

a Rename the first step `Accelerate` and add the step actions:

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

b Rename the second step `Stop` and add the step actions:

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

c Right-click `Accelerate` and select **Add sub-step**. Create a total of four substeps for `Accelerate`.

These steps check the component under test during the test sequence.

d Add a constant to the block. In the **Data Symbols** pane, hover over **Constant** and click **Add**. Enter `Limit` for the constant name.

e Hover over `Limit` and click **Edit**. In the **Initial value** field, enter `2`. Click **OK**.

f In the **Transition** column, enter the transition condition for `Accelerate`. This condition uses the duration operator and transitions to the next step when the system in in fourth gear for 2 seconds.

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

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

g Change the `Accelerate` group to a When decomposition sequence. Right-click `Accelerate` and select **When decomposition**.

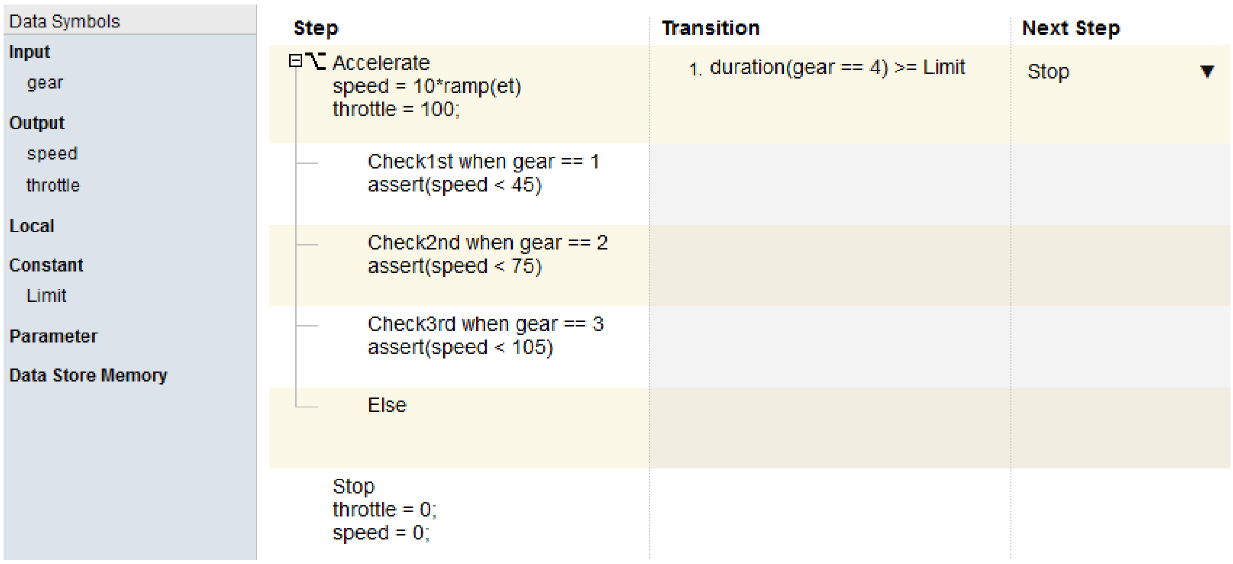
h Enter the names and actions for the substeps. The fourth step `Else` takes no action. It handles conditions that does not match the when statements in the other three.

```
Check1st when gear == 1
assert(speed < 45)

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

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

Else
```



Data Symbols

Input
gear

Output
speed
throttle

Local

Constant
Limit

Parameter

Data Store Memory

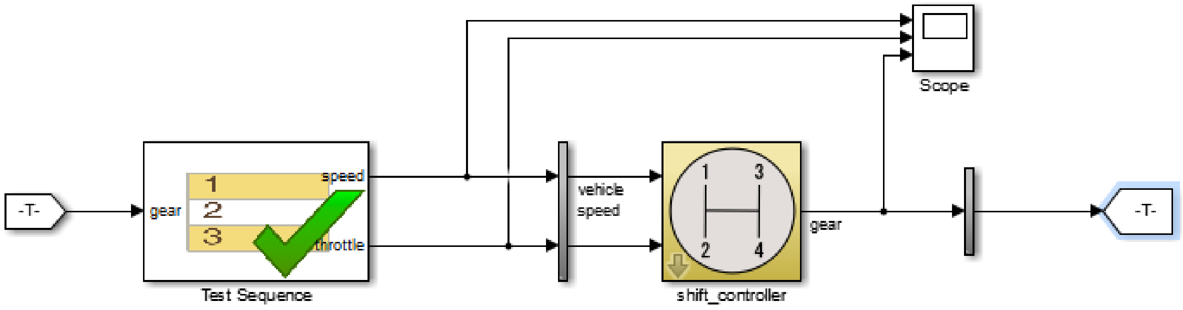
Step	Transition	Next Step

⊟ Accelerate
speed = 10*ramp(et)
throttle = 100;
	1. duration(gear == 4) >= Limit	Stop ▼

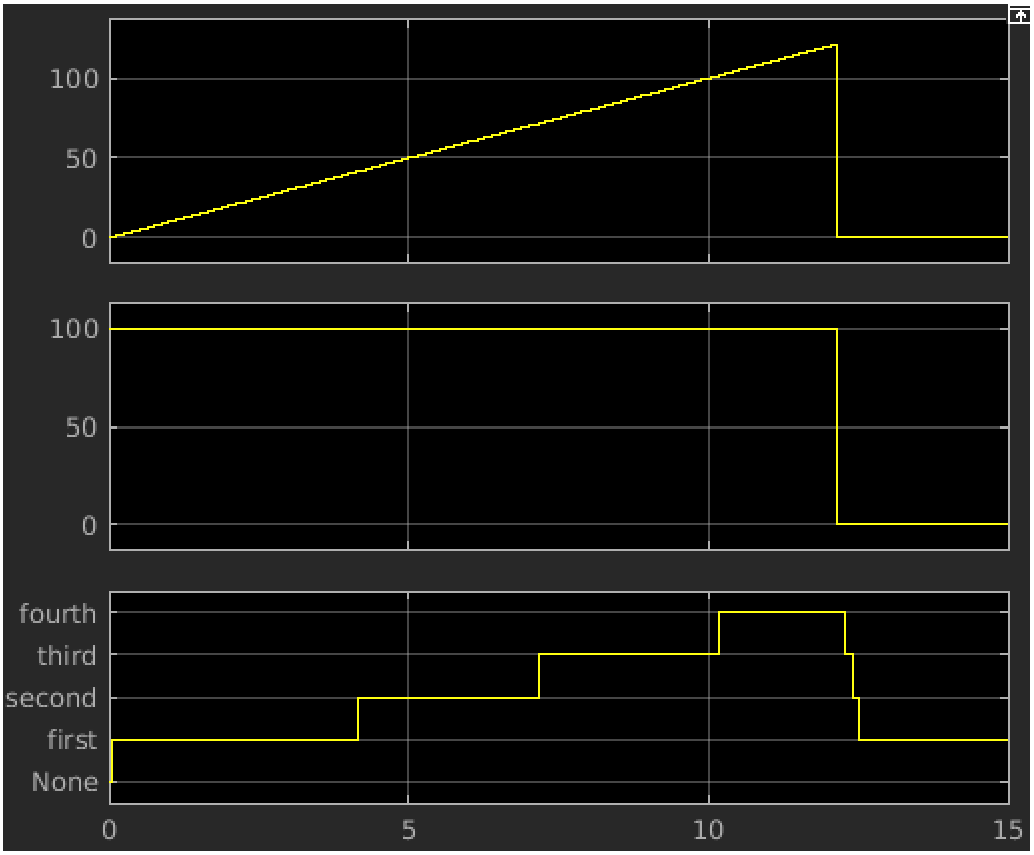
Check1st when gear == 1 assert(speed < 45)		
Check2nd when gear == 2 assert(speed < 75)		
Check3rd when gear == 3 assert(speed < 105)		
Else		

Stop
throttle = 0;
speed = 0;		

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



7 Set the model simulation time to 15 seconds and simulate the test harness.



See Also

“Organize Test Sequences” on page 3-8 | “Reuse Test Assessments” on page 3-32 | “Evaluate Temporal or Signal Conditions in a Test Sequence Transition” on page 3-11 | “Generate Function-Based Test Signals” on page 3-14 | `Test Sequence`

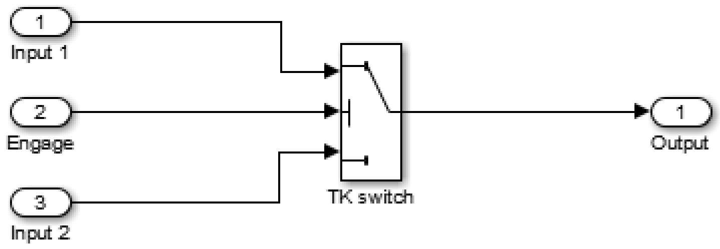
Organize Test Sequences

Compared to using timeseries data, using the Test Sequence block to define your test inputs has these advantages:

- You can organize test scenarios in test step groups, and use hierarchy levels to isolate test scenario execution.
- You can isolate model functionality by separating signal commands into distinct test steps.
- Steps can execute in response to the model, using logical conditions.
- You can author assessments for specific test conditions.
- You can concisely express signal patterns, such as waveforms, using output commands.

Before creating test steps, consider the test sequence organization. Clear organization helps communicate the test sequence intent and structure.

Consider the case of verifying a simple subsystem. The subsystem consists of a switch controlled by the `Engage` signal.



The goal of the test is to complete a simple verification of the switch function. The test does not cover all objectives for full verification, but covers a simple design check. Check that the output equals `Input 1` when the control is engaged, and `Input 2` when the control is not engaged. You organize a test sequence into an initialization step and two test scenarios. Each scenario sets `Input 1` and `Input 2`, then sets `Engage`, then assesses the switch output:

1 **Initialize the signals**

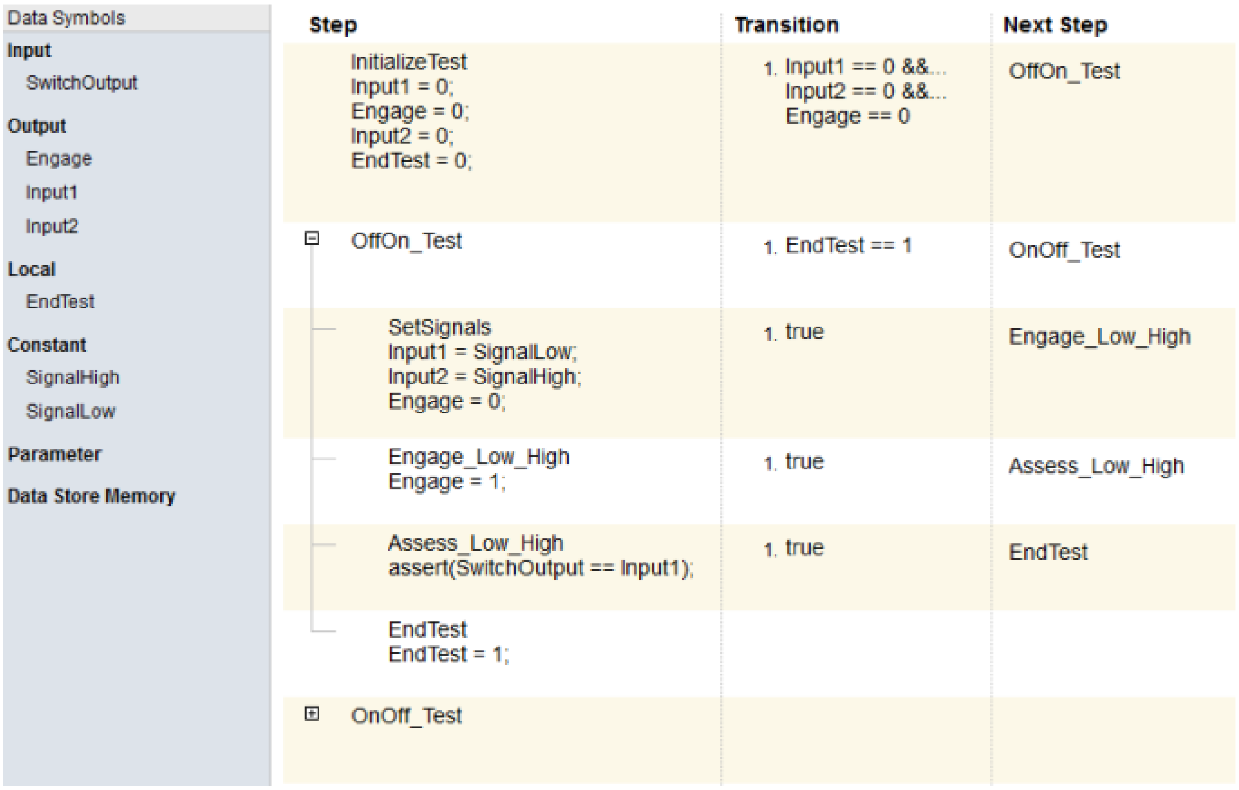
2 **Scenario 1**

   a Set the signal levels
   b Engage the control
   c Assess the result

3 **Scenario 2**

   a Set the signal levels
   b Engage the control
   c Assess the result

In the test sequence editor, the step hierarchy follows the hierarchy of the scenario outline:



Data Symbols
- **Input**
  - SwitchOutput
- **Output**
  - Engage
  - Input1
  - Input2
- **Local**
  - EndTest
- **Constant**
  - SignalHigh
  - SignalLow
- **Parameter**
- **Data Store Memory**

Step	Transition	Next Step

InitializeTest
Input1 = 0;
Engage = 0;
Input2 = 0;
EndTest = 0;
	1. Input1 == 0 &&...
	Input2 == 0 &&...
	Engage == 0
		OffOn_Test

⊟ OffOn_Test	1. EndTest == 1	OnOff_Test

SetSignals
Input1 = SignalLow;
Input2 = SignalHigh;
Engage = 0;
	1. true	Engage_Low_High

Engage_Low_High Engage = 1;	1. true	Assess_Low_High
Assess_Low_High assert(SwitchOutput == Input1);	1. true	EndTest
EndTest EndTest = 1;		
⊞ OnOff_Test		

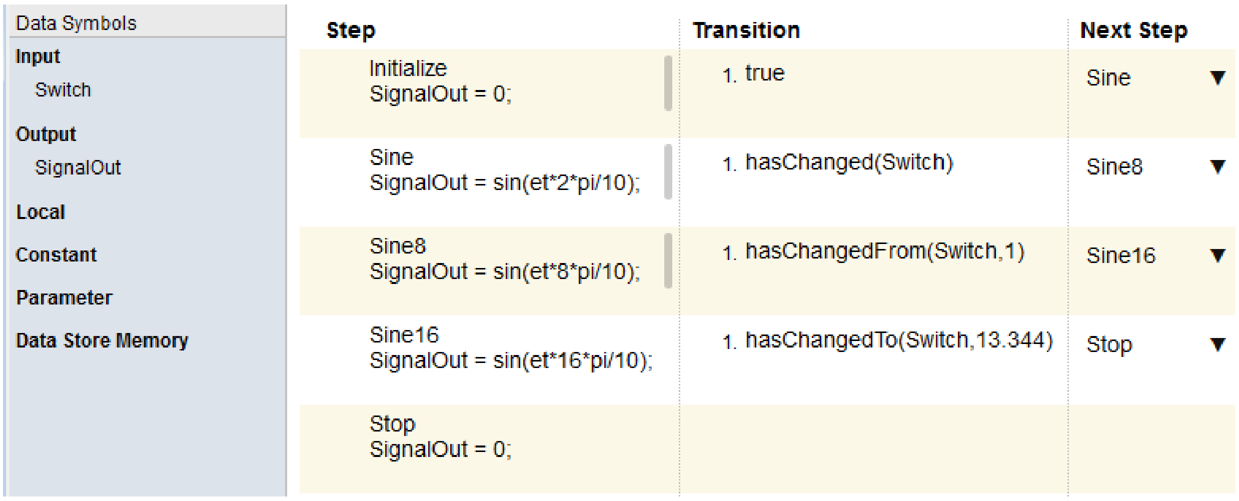
Note: To execute test steps sequentially without using a logical transition condition, use the condition `true`. `true` moves the sequence to the next step after the current step.

Evaluate Temporal or Signal Conditions in a Test Sequence Transition

The Test Sequence block uses MATLAB as the action language. You can transition between test steps by evaluating the component under test. You can use conditional logic, temporal operators, and event operators.

Consider a simple test sequence that outputs a sine wave at three frequencies. The test sequence transitions between steps:

- From `Initialize` to `Sine` when `Switch` changes
- From `Sine` to `Sine8` when `Switch` changes from the value `1`
- From `Sine8` to `Sine16` when `Switch` changes to the value `13.344`



Step	Transition	Next Step
Initialize SignalOut = 0;	1. true	Sine
Sine SignalOut = sin(et*2*pi/10);	1. hasChanged(Switch)	Sine8
Sine8 SignalOut = sin(et*8*pi/10);	1. hasChangedFrom(Switch,1)	Sine16
Sine16 SignalOut = sin(et*16*pi/10);	1. hasChangedTo(Switch,13.344)	Stop
Stop SignalOut = 0;		

You can use these common operators for temporal and event logic.

Operator	Syntax	Description
`after`	`after(n,TimeUnits)`	Returns true if `n` specified units of time have elapsed since the beginning of the current test step. The timer resets if the sequence exits the test step.

Operator	Syntax	Description
`before`	`before(n,TimeUnits)`	Returns true until `n` specified units of time elapse since the beginning of the current test step. The timer resets if the sequence exits the test step.
`duration`	`ElapsedTime = durati`	`duration` uses temporal logic and signal conditions to return the time in seconds since `SignalCondition` becomes true, within the period of the step in which `duration` is used.

`elapsed`
Abbreviation: `et`
	`elapsed(TimeUnits)`
	
	`et(TimeUnits)`
		Returns the elapsed time of the test step in the units specified. Omitting time units returns the value in seconds.

`getSimulationTim`
Abbreviation: `t`
	`getSimulationTime(Ti`
	
	`t(TimeUnits)`
		Returns the elapsed time of the simulation in the units specified. Omitting time units returns the value in seconds.

`hasChanged`	`hasChanged(u)`	Returns true if the argument `u` changes in value since the beginning of the test step, otherwise returns false. In the Test Sequence block, `u` must be an input data symbol. `u` cannot be an expression or other type of variable.
`hasChangedFrom`	`hasChangedFrom(u,A)`	Returns true if the argument `u` changes from the value `A`, otherwise returns false. In the Test Sequence block, `u` must be an input data symbol. `u` cannot be an expression or other type of variable.
`hasChangedTo`	`hasChangedTo(u,B)`	Returns true if the argument `u` changes to the value `B`, otherwise returns false. In the Test Sequence block, `u` must be an input data symbol. `u` cannot be an expression or other type of variable.

Syntax in the table uses these arguments:

`TimeUnits`

The units of time.

Value: `sec|msec|usec`

Examples:

```
msec
```

`SignalCondition`

Logical expression of the condition to trigger the temporal operator. Variables used in the signal condition must be inputs, parameters, or constants in the Test Sequence block.

Examples:

```
u > 0
x <= 1.56
```

See Also

`Test Sequence`

Related Examples

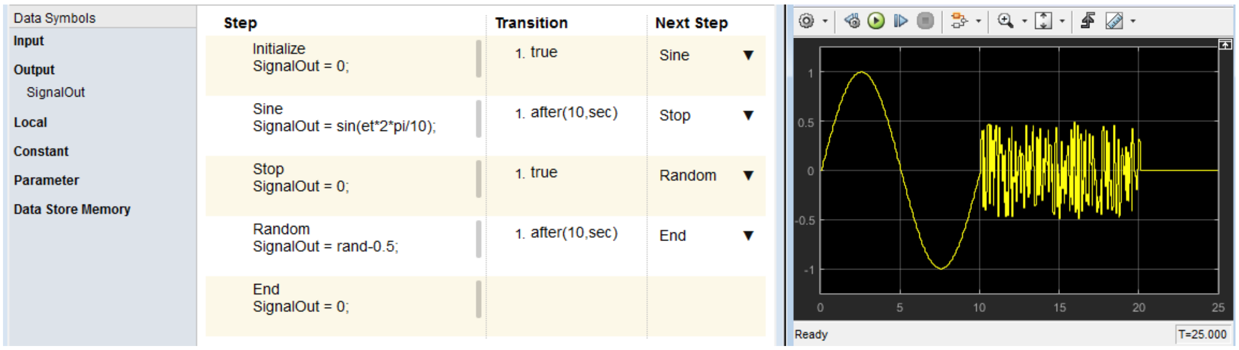
- "Generate Function-Based Test Signals" on page 3-14

Generate Function-Based Test Signals

The Test Sequence block uses MATLAB as the action language. You can use functions to generate signal outputs to the component under test.

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

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

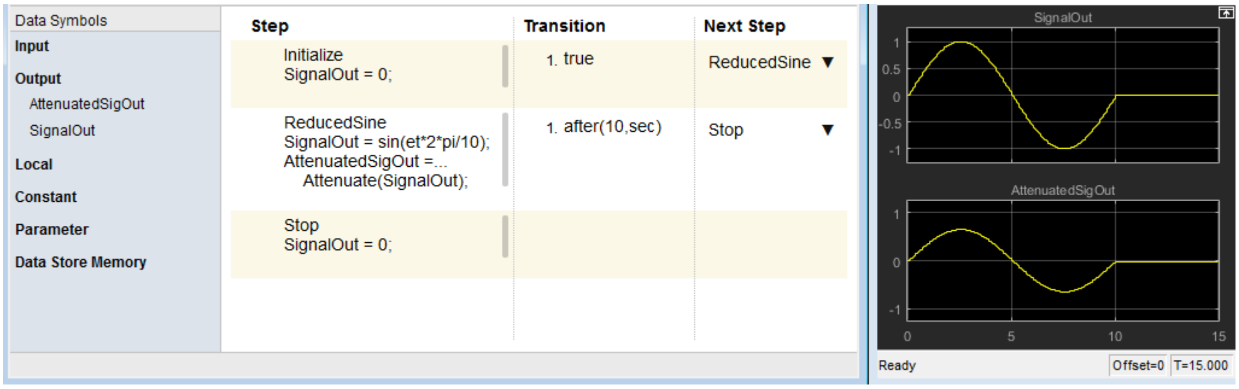


Data Symbols
- **Input**
- **Output**
  - SignalOut
- **Local**
- **Constant**
- **Parameter**
- **Data Store Memory**

Step	Transition	Next Step
Initialize SignalOut = 0;	1. true	Sine ▼
Sine SignalOut = sin(et*2*pi/10);	1. after(10,sec)	Stop ▼
Stop SignalOut = 0;	1. true	Random ▼
Random SignalOut = rand-0.5;	1. after(10,sec)	End ▼
End SignalOut = 0;		

You can also define a function in a script on the MATLAB path, and call the function in the Test Sequence block. In this test sequence, the `ReducedSine` step reduces `SignalOut` using the function `Attenuate`.

```
function[y] = Attenuate(x)
y = 0.65*x;
end
```



Data Symbols
- Input
- Output
  - AttenuatedSigOut
  - SignalOut
- Local
- Constant
- Parameter
- Data Store Memory

Step	Transition	Next Step
Initialize SignalOut = 0;	1. true	ReducedSine ▼

ReducedSine
SignalOut = sin(et*2*pi/10);
AttenuatedSigOut =...
Attenuate(SignalOut);
	1. after(10,sec)	Stop ▼

Stop SignalOut = 0;		

Note: Scaling, rounding, and other approximations can affect signal functions. Signal function output is not constrained to a defined pattern. Consider the effect of scaling, rounding, and other approximations.

You can use these common functions to generate test signals.

Syntax	Description	Additional Information
`square(x)`	Represents a square wave output of period `1` and range `-1` to `1`, returning the value of the square wave at time `x`. Within the period `0 <= x < 1`, `square(x)` returns the value `1` for `0 <= x < 0.5`and `-1` for `0.5 <= x < 1`.	`square(x)` ≡ `4*floor(x)-2*floor(2*x)+1`
`sawtooth(x)`	Represents a sawtooth wave output of period `1`	`sawtooth(x)` ≡ `2*(x-floor(x))-1`

Syntax	Description	Additional Information
	and range `-1` to `1`, returning the value of the sawtooth wave at time `x`. Within the period `0 <= x < 1`, `sawtooth(x)` increases.	
`triangle(x)`	Represents a triangle wave output of period `1` and range `-1` to `1`, returning the value of the triangle wave at time `x`. Within the period `0 <= x < 0.5`, `triangle(x)` increases.	`triangle(x)` ≡ `2*abs(sawtooth(x+0.5))-1`
`ramp(x)`	Represents a ramp signal of slope `1`, returning the value of the ramp at time `x`.	`ramp(x)` ≡ `x`
`heaviside(x)`	Represents a heaviside step signal, returning `0` for `x < 0` and `1` for `x >= 0`.	`heaviside(x)` ≡ `x < 0 ? 0 : 1`

Syntax	Description	Additional Information
`latch(x)`	Returns the current value of `x` and holds that value during the test step.	In this example, `latch(x)` holds the value of `x` upon entry of `TestStep2`, and generates a ramp signal descending from that value.

Step	Transition	Next Step

`TestStep1`
`x = ramp(t)`
	`after(5,sec)`	`TestStep2`

`TestStep2`
`x = latch(x) -`		

Syntax	Description	Additional Information

`sin(x)`
 	
`cos(x)`
	Returns the sine (or cosine) of `x`, where `x` is in radians.	

`rand`	Uniformly distributed pseudorandom number.	`rand` returns a single uniformly distributed value between `0` and `1`.
`randn`	Normally distributed pseudorandom number.	`randn` returns a value selected from a standard normal distribution (`mean = 0`, `stdev = 1`).
`exp(x)`	Returns the natural exponential function, e^x .	
`log(x)`	Natural logarithm of `x`. Complex results are not supported.	

See Also

`Test Sequence`

Related Examples

- "Evaluate Temporal or Signal Conditions in a Test Sequence Transition" on page 3-11

Debug a Test Sequence

In this section...

You can debug a test sequence using tools in the test sequence editor. Debugging involves setting breakpoints to stop simulation, observing data and test sequence progression, and manually stepping through test steps. You can try these features using the model `TestSequenceBlockDebuggingExample`. To open the model, enter

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

Save a copy of the model to a writable location on the MATLAB path. Double-click the Test Sequence block to open the test sequence editor.

View Test Step Execution During Simulation

By default, simulation animates the test sequence by highlighting active steps and transitions. Observing test step execution can help you debug, particularly when manually stepping through the test sequence. Adjust the animation speed using the **Change Animation Speed** button in the toolbar.

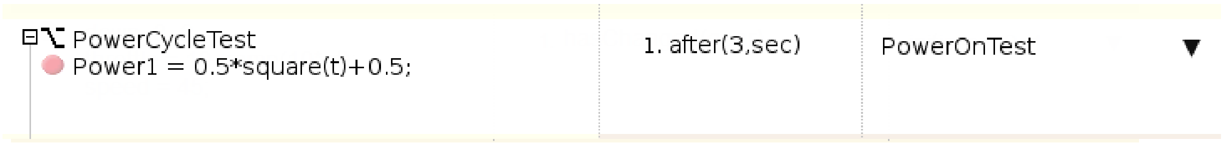
Animation speed affects simulation speed. If you slow down animation speed for debugging, return the speed to **Fast** or **Lightning Fast** when you finish debugging to avoid slowing your simulation. If you do not need the test step highlights and want the fastest simulation, choose **None**.

Set Breakpoints to Enable Debugging

You enable debugging for a test sequence by adding one or more breakpoints. Breakpoints halt simulation every time the test step is evaluated. Therefore, breakpoints on some test steps, such as **When decomposition** parent steps, halt simulation repeatedly because the step is evaluated repeatedly. When simulation halts, you can view data used in the test sequence to investigate the sequence simulation behavior.

You can add breakpoints to test step actions or transitions:

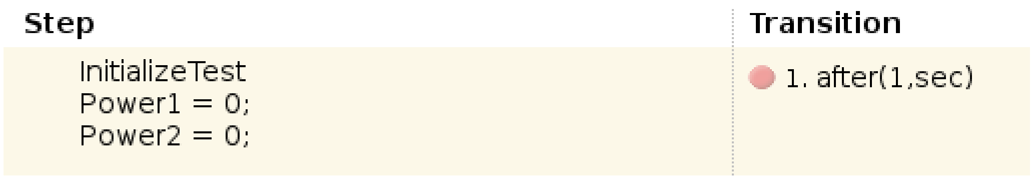
- To add a breakpoint to a test step action, right-click the test step and select **Break while executing step**.



```
PowerCycleTest
Power1 = 0.5*square(t)+0.5;
```

PowerCycleTest Power1 = 0.5*square(t)+0.5;	1. after(3,sec)	PowerOnTest

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



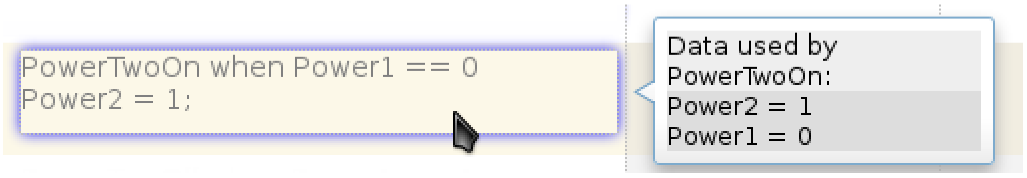
Step	Transition

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

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

View Data Values During Simulation

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



```
PowerTwoOn when Power1 == 0
Power2 = 1;
```

```
Data used by
PowerTwoOn:
Power2 = 1
Power1 = 0
```

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

Step Through Simulation

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

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

See Also

`Test Sequence`

Test a Model Component Using Signal Functions

In this section...
"Create a Test Sequence" on page 3-22
"Simulate the Test Harness" on page 3-24

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

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

Create a Test Sequence

1 Access the model. Enter

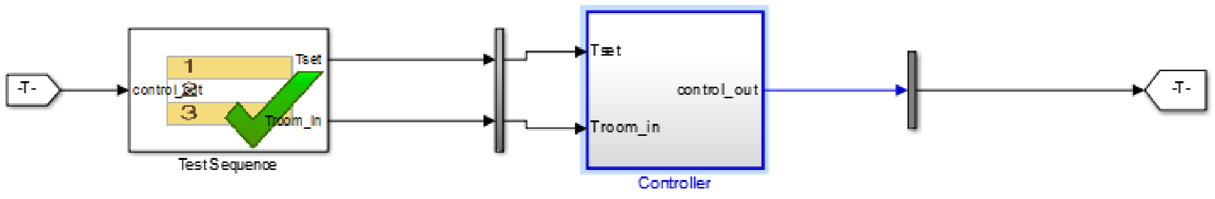
```
cd(fullfile(docroot,'toolbox','sltest','examples'))
```

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

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

3 Open the model, and open the harness.

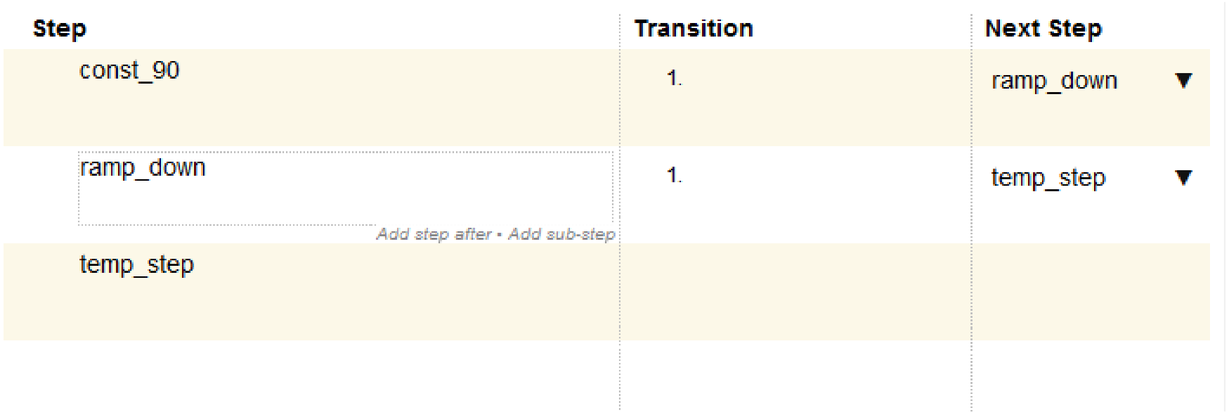
```
open_system('sltestSignalFunctionExample');
sltest.harness.open('sltestSignalFunctionExample/Controller','RampSquareHarness')
```



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

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

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



Step	Transition	Next Step
const_90	1.	ramp_down ▼
ramp_down Add step after • Add sub-step	1.	temp_step ▼
temp_step		

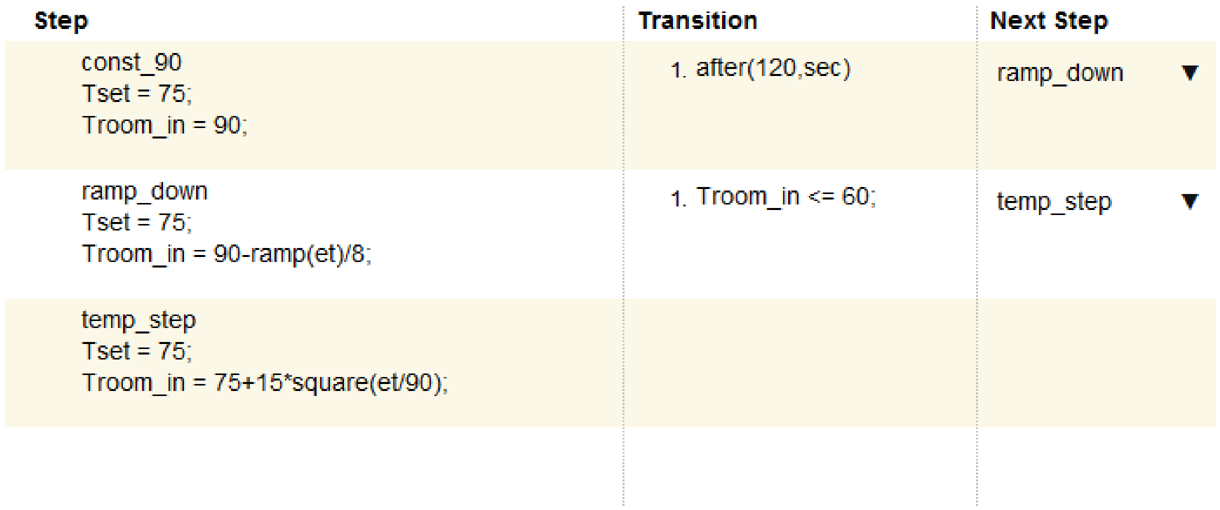
7 Add output conditions and transition fields to the steps. Copy and paste the listings from the table.

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 ▼

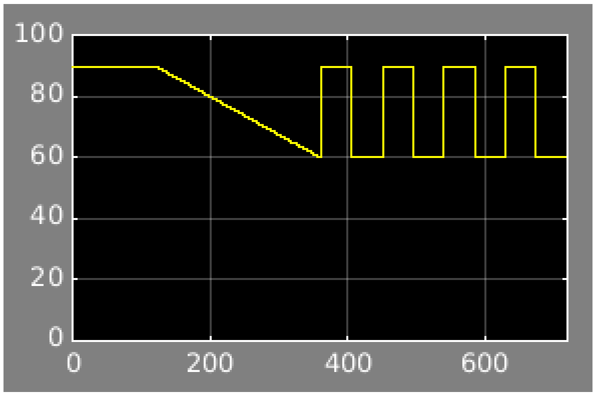
ramp_down
Tset = 75;
Troom_in = 90-ramp(et)/8;
	1. Troom_in <= 60;	temp_step ▼

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

This example demonstrates a test sequence and test assessment for a transmission shift logic controller.

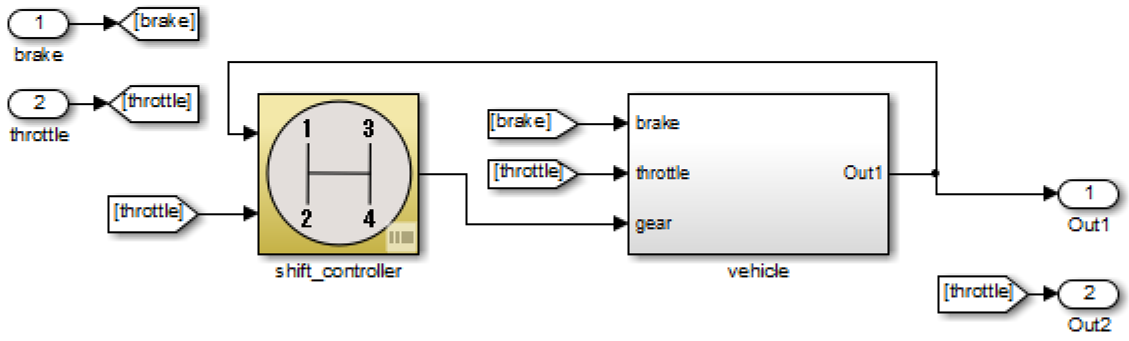
The Model and Controller

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

The Test

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

```
path = fullfile(matlabroot,'examples','simulinktest');
mdl = 'TransmissionDownshiftTestSequence';
harness = 'controller_harness';
open_system(fullfile(path,mdl));
```



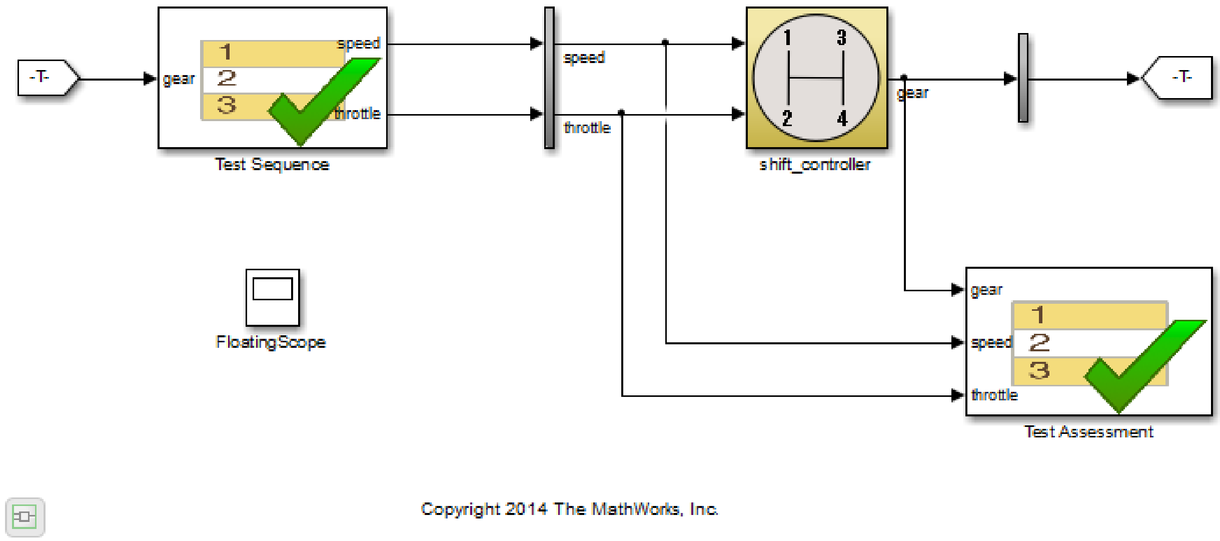
Testing Downshift Points of a Transmission Controller

Copyright 2014 The MathWorks, Inc.

Open the Test Harness

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

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

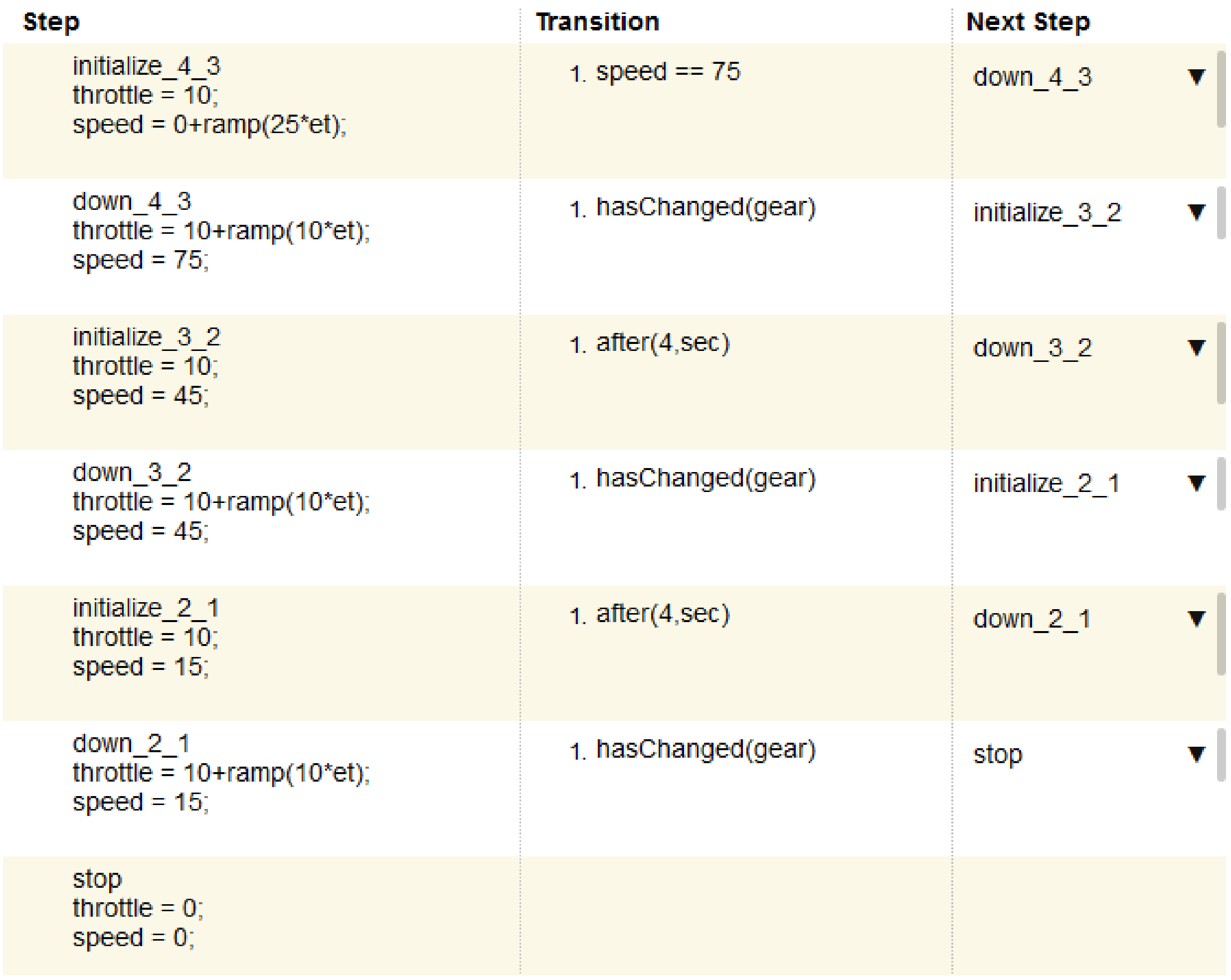


The Test Sequence

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

The test sequence begins by ramping speed to 75 to initialize the controller to fourth gear. Throttle is then ramped at constant speed until a gear change. Subsequent initialization and downshifts execute. After the change to first gear, the test sequence stops.

```
open_system([harness '/Test Sequence']);
```



Step	Transition	Next Step

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;
	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;		

Test Assessments for the Controller

Assume that the requirements for the shift controller include:

- Speed shall never be negative.
- Gear shall always be positive.
- Throttle shall be between 0% and 100%.

- The controller shall not let the engine overspeed.

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

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

The last requirement has three sub-requirements. We assume that the engine cannot overspeed in fourth (top) gear.

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

You can model these assessments with a When decomposition sequence. When decomposition step selection is based on signal conditions defined in the **Step** column, with each condition preceded by the `when` operator. The **Transition** and **Next Step** columns do not affect the transition. The last step `Else` in the when decomposition covers any undefined condition and does not use a `when` declaration.

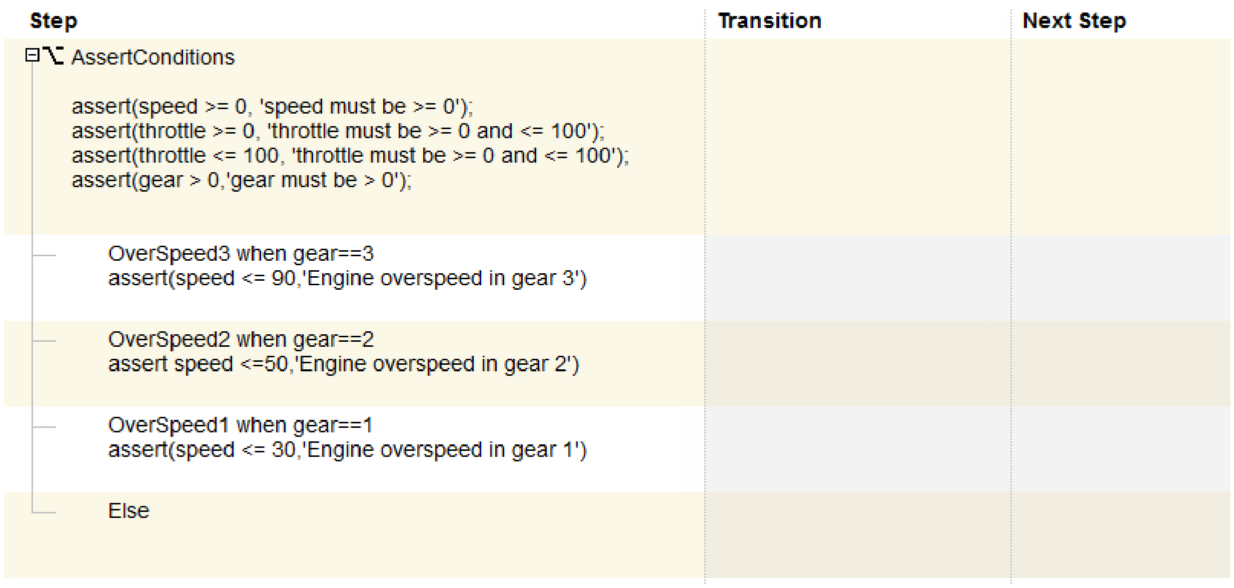
To change a sequence to a When decomposition, right-click a step and select **When decomposition**. Sub-steps of this step then operate using the `when` operator.

`AssertConditions` has sub-steps that assess the controller as follows:

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

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

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



Step	Transition	Next Step

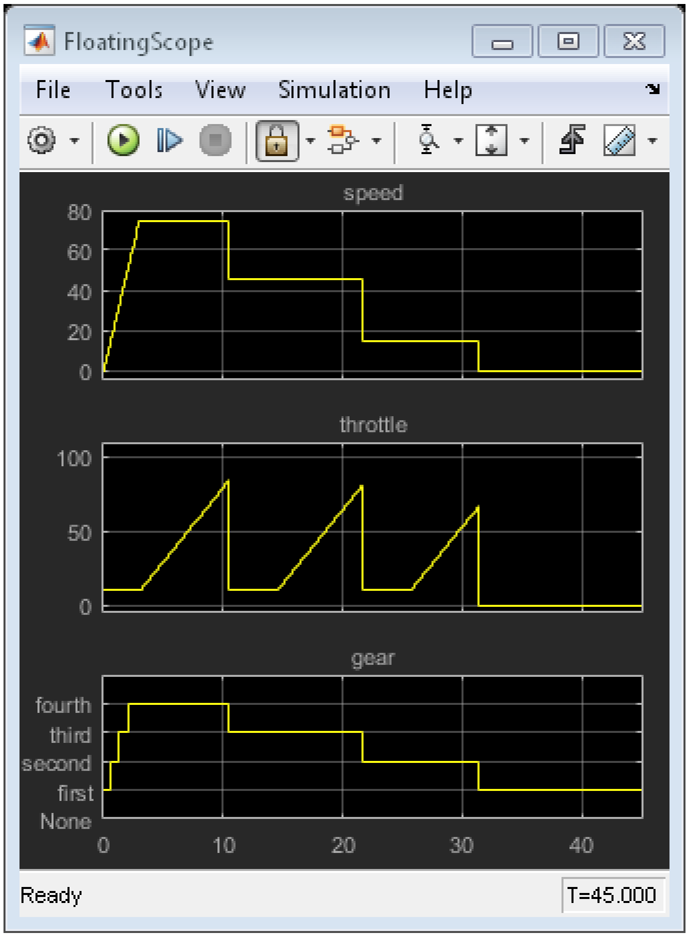
AssertConditions
 	
assert(speed >= 0, 'speed must be >= 0');
assert(throttle >= 0, 'throttle must be >= 0 and <= 100');
assert(throttle <= 100, 'throttle must be >= 0 and <= 100');
assert(gear > 0,'gear must be > 0');		

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		

Testing the Controller

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

```
open_system([harness '/FloatingScope'])
sim(harness);
```


FloatingScope
File
Tools
View
Simulation
Help
speed
80
60
40
20
0
throttle
100
50
0
gear
fourth
third
second
first
None
0
10
20
30
40
Ready
T=45.000

Reuse Test Assessments

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

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

Reuse Test Assessments Using a Library

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

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

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

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

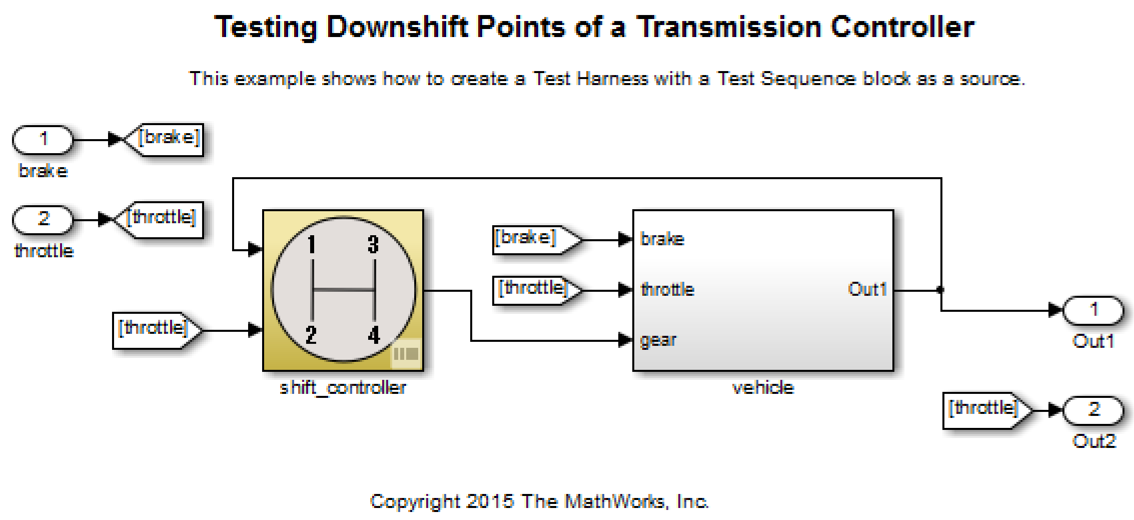
Consider using a library for high-level test assessments that correspond to multiple test cases.

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

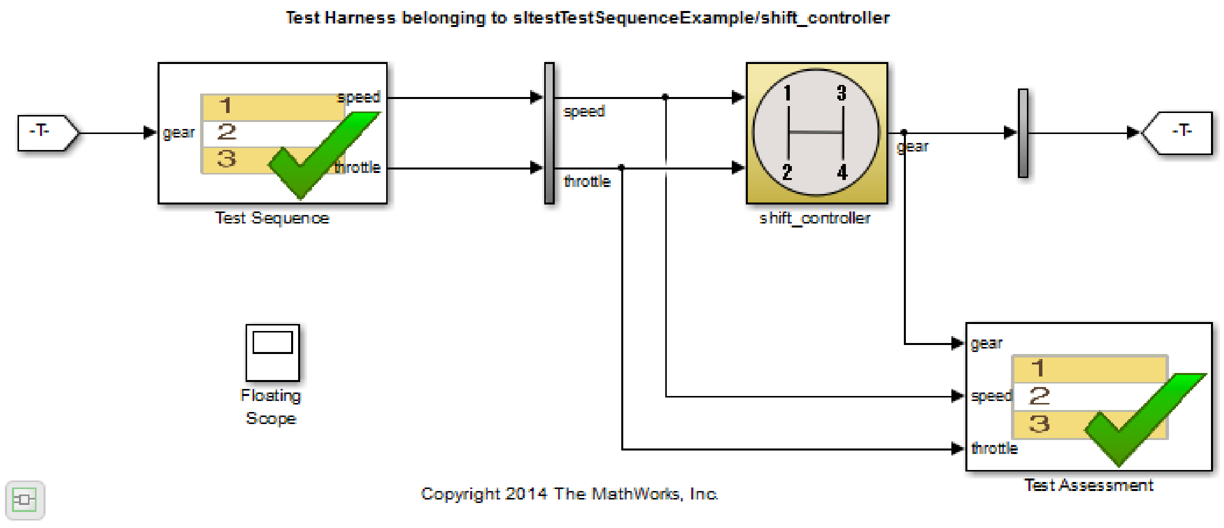
Explore the Test Sequence Example Model

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

```
sltestTestSequenceExample
```


Testing Downshift Points of a Transmission Controller
This example shows how to create a Test Harness with a Test Sequence block as a source.
1
brake
[brake]
2
throttle
[throttle]
[throttle]
1
3
2
4
shift_controller
[brake]
[throttle]
brake
throttle
gear
Out1
vehicle
1
Out1
[throttle]
2
Out2
Copyright 2015 The MathWorks, Inc.

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

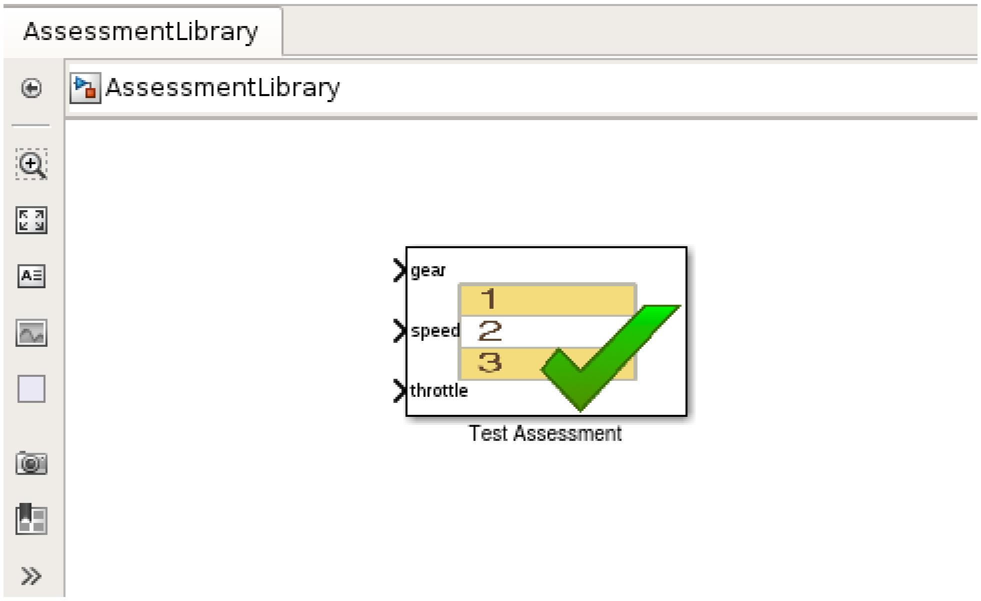


The Test Assessment block contains four assertions that define the assessment criteria:

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

Create a Library for the Test Assessments

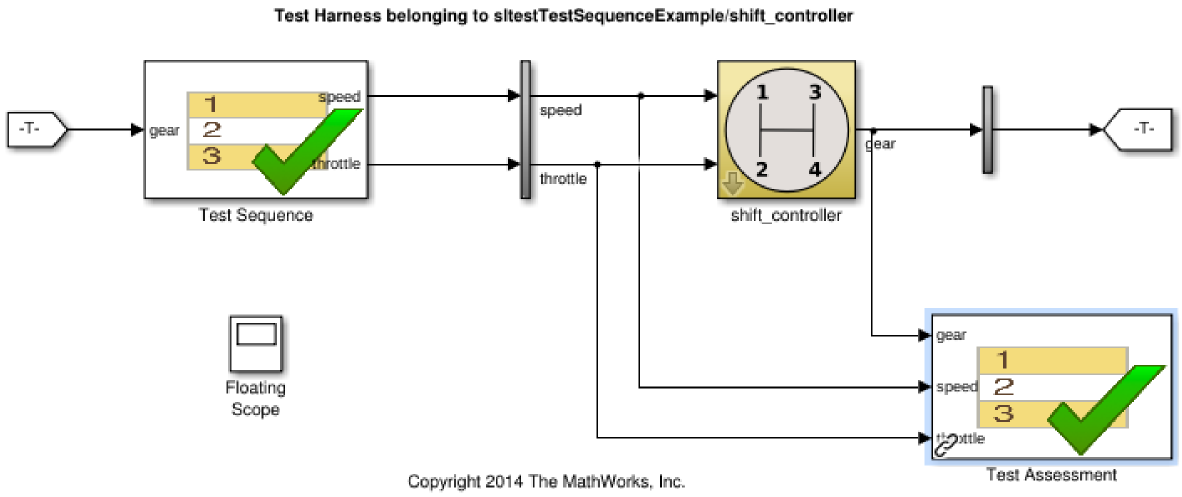
1 In the test harness, select **File > New > Library**.
2 Save the new library as `AssessmentLibrary` in a writable location on the MATLAB® path.
3 Copy the Test Assessment block from the test harness to the library, and then delete the Test Assessment block from the test harness.
4 Save the library.



Create a Linked Test Assessment Block in Test Harnesses

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

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



Edit the Assessment Block in the Library

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

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

SIL Verification for a Subsystem

In this section...

This example shows subsystem verification by ensuring the output of software-in-the-loop (SIL) code matches that of the model subsystem. You generate a SIL verification harness, collect simulation results, and compare the results using the simulation data inspector. You can apply a similar process for processor-in-the-loop (PIL) verification.

With SIL simulation, you can verify the behavior of production source code on your host computer. Additionally, with PIL simulation, you can verify the compiled object code that you intend to deploy in production. You can run the PIL object code on real target hardware or on an instruction set simulator.

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

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

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

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

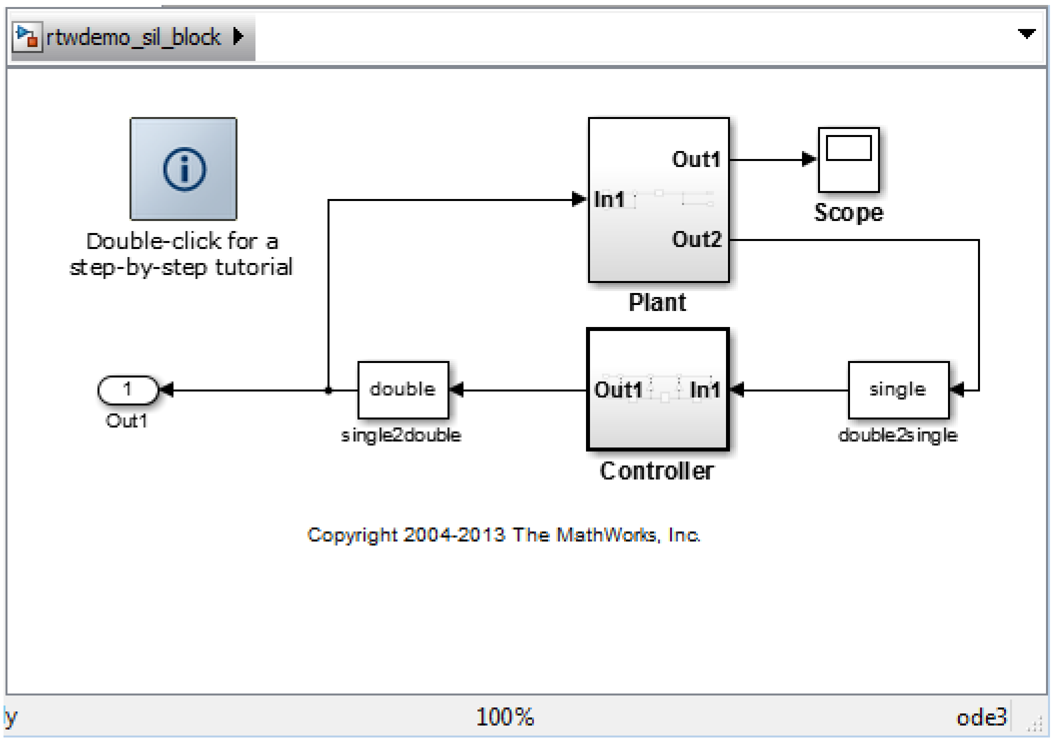
The example models a closed-loop controller-plant system. The controller regulates the plant output.

Create a SIL Verification Harness for a Controller

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

1 Open the example model by entering

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



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

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

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

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

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

6 Simulate the model.

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

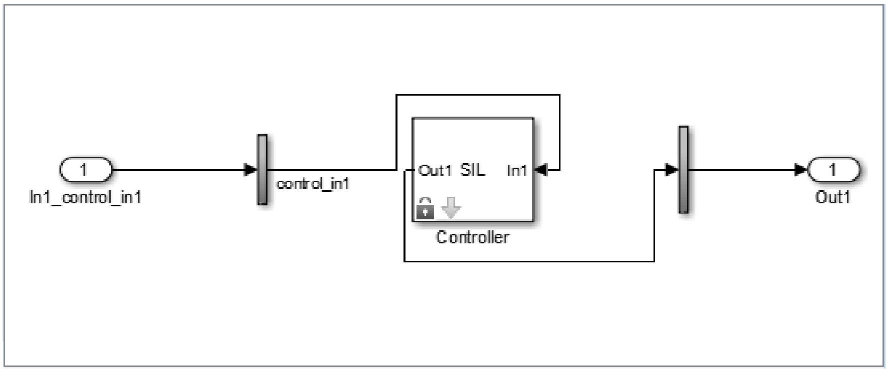
```
out_data = out.get('SIL_signals');
control_in1 = out_data.get('controller_model_input');
control_out1 = out_data.get('controller_model_output');
```

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

9 Set the harness properties:

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

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



Configure and Simulate a SIL Verification Harness

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

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

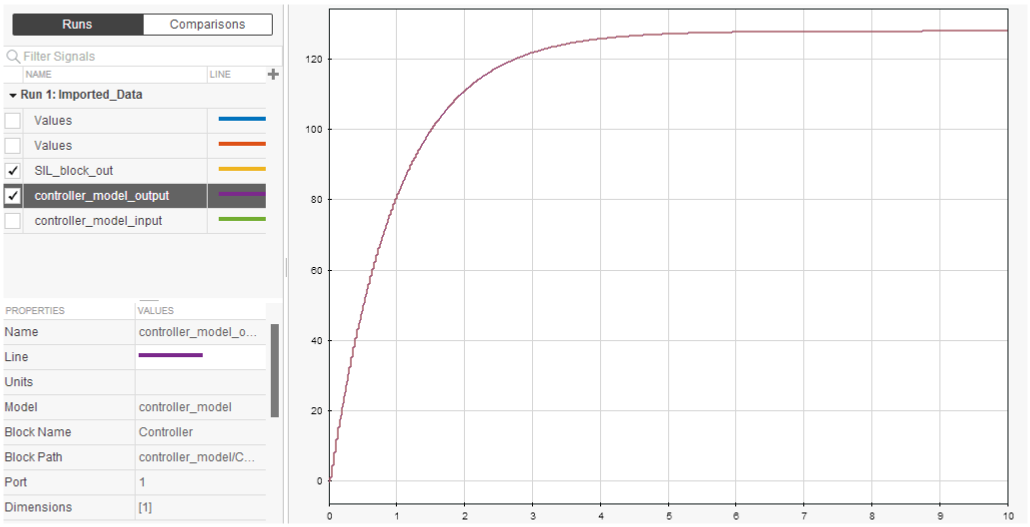
   ```
   set_param('SIL_harness','SignalLogging','on','SignalLoggingName',...
   'harness_signals','SignalLoggingSaveFormat','Dataset')
   ```
3. Right-click the output signal of the SIL block and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `SIL_block_out`.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

Compare the SIL Block and Model Controller Outputs

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

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

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



☐ Values	🟦
☐ Values	🟧
✓ SIL_block_out	🟨
✓ controller_model_output	🟪
☐ controller_model_input	🟩

PROPERTIES	VALUES
Name	controller_model_o...
Line	🟪
Units	
Model	controller_model
Block Name	Controller
Block Path	controller_model/C...
Port	1
Dimensions	[1]

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

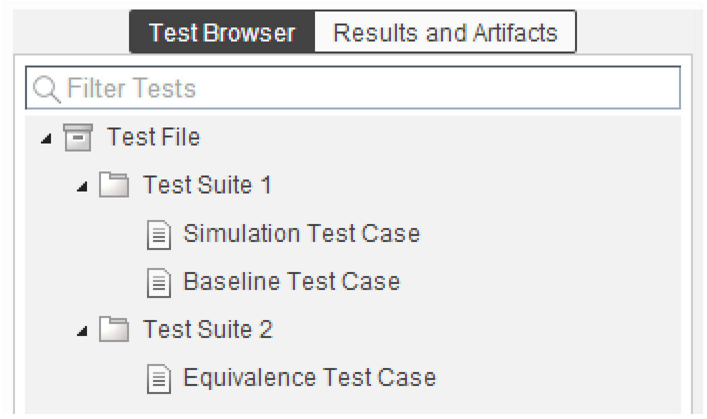
In this section...

Test Manager Description

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.

Test Manager Test Cases

Test Model Output Against a Baseline

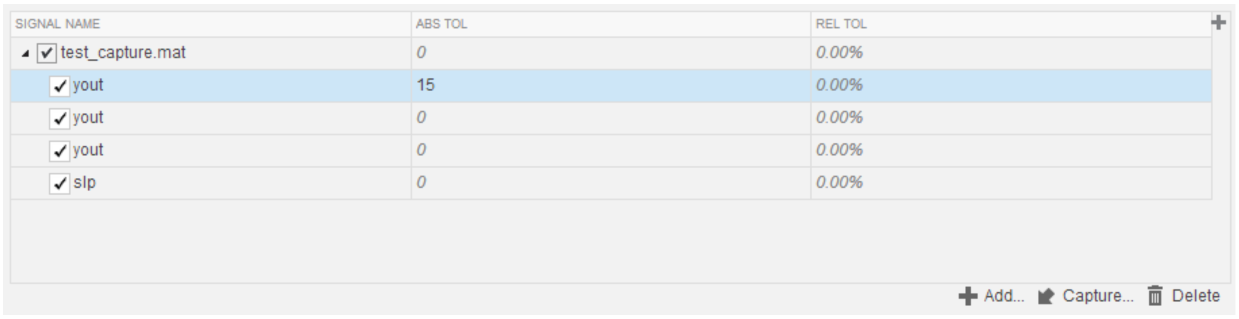
To test the simulation output of a model against a defined baseline data set, use a baseline test case. In this example, use the `sldemo_absbrake` model to compare the simulation output to a baseline that is captured from an earlier state of the model.

Create the Test Case

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

   The new test file consists of a test suite that contains one baseline test case. They appear in the **Test Browser** pane.
4. Right-click the baseline test case in the **Test Browser** pane, and select **Rename**. Rename the test case to `Slip Baseline Test`.
5. Under **System Under Test** in the test case, click the **Use current model** button to load the `sldemo_absbrake` model into the test case.
6. Under **Baseline Criteria**, click **Capture** to record a baseline data set from the model specified under **System Under Test**.

   Save the baseline data set to a location. After you save the baseline MAT-file, the model runs and the baseline criteria appear in the table.
7. Expand the baseline data set. Set the **Absolute Tolerance** of the first `yout` signal to `15`, which corresponds to the `Ww` signal.



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

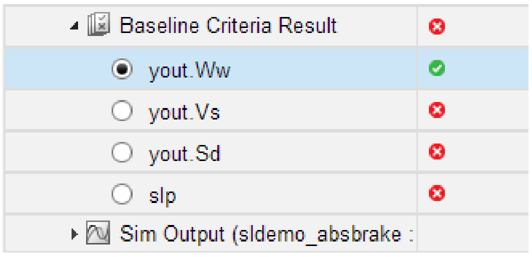
For more information about tolerances and criteria, see “How Tolerances Are Applied to Test Criteria” on page 6-19.

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.

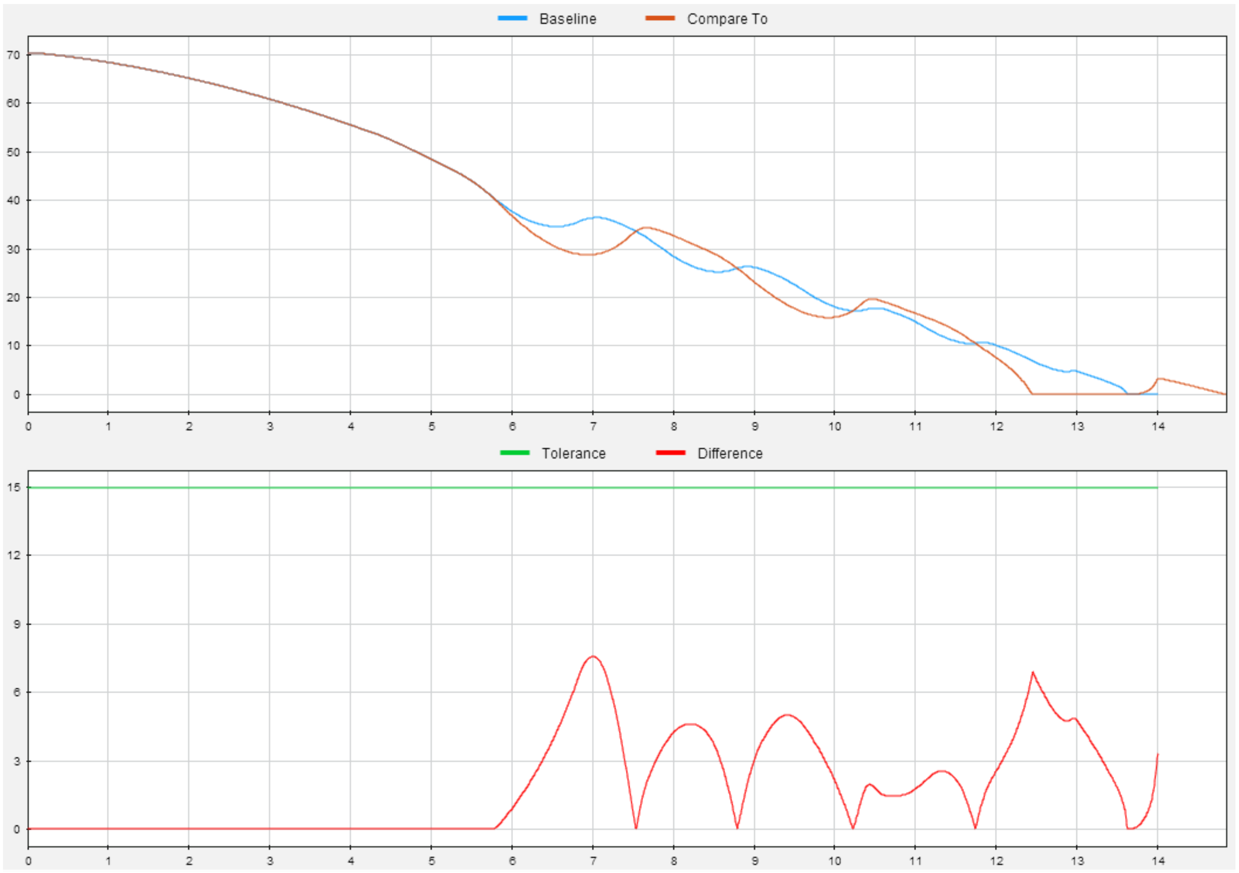
   The test manager switches to the **Results and Artifacts** pane, and the new test result appears at the top of the table.
4. Expand the results until you see the baseline criteria result.

   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.

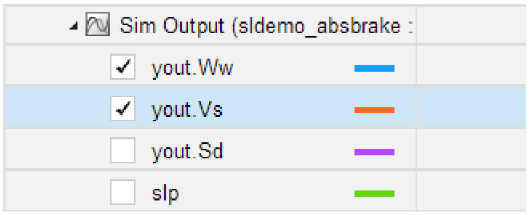


Baseline Criteria Result	✗
● yout.Ww	✓
○ yout.Vs	✗
○ yout.Sd	✗
○ slp	✗
Sim Output (sldemo_absbrake :	

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



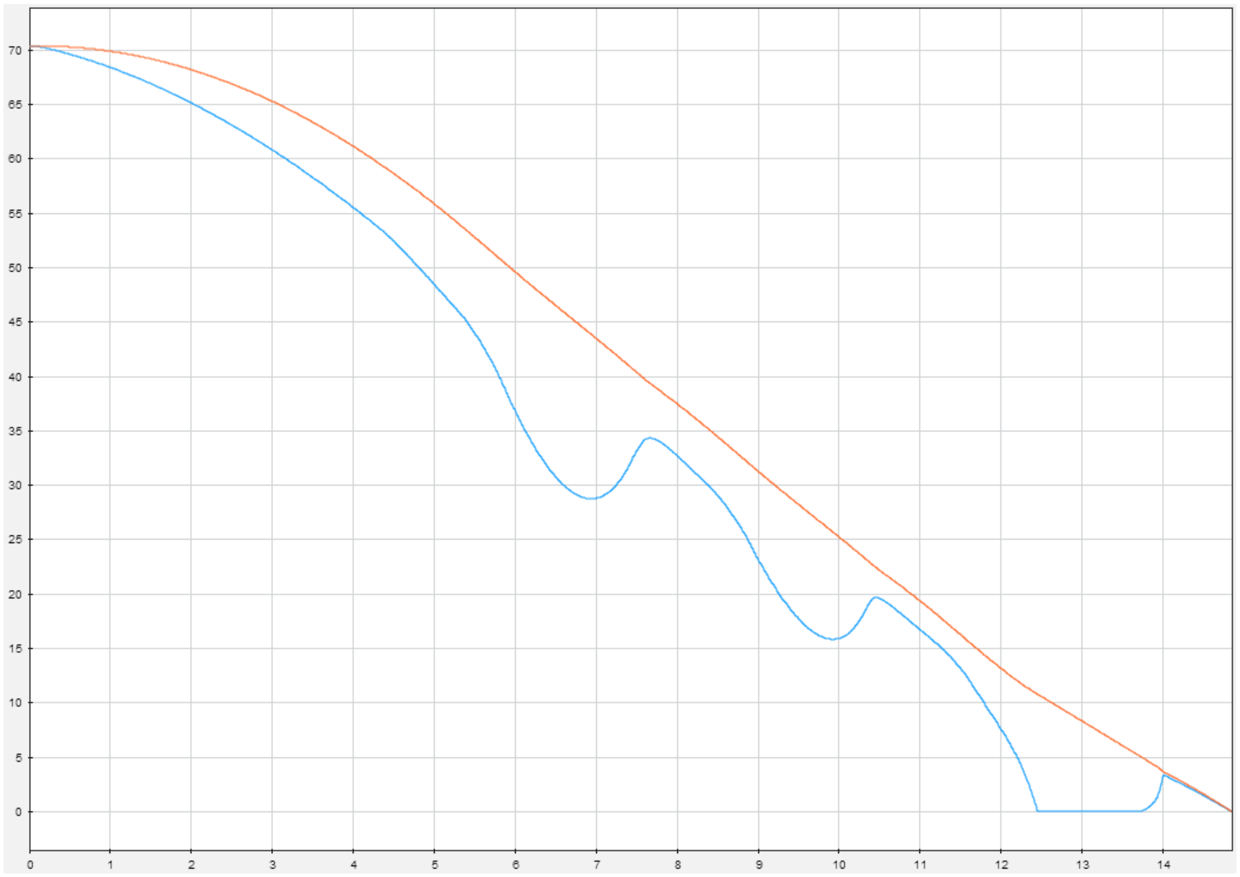
6 You can also view signal data from the simulation. Expand `Sim Output` and select the signals you want to plot.



Sim Output (sldemo_absbrake :

✓	yout.Ww	🟦
✓	yout.Vs	🟧
	yout.Sd	🟪
	slp	🟩

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



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

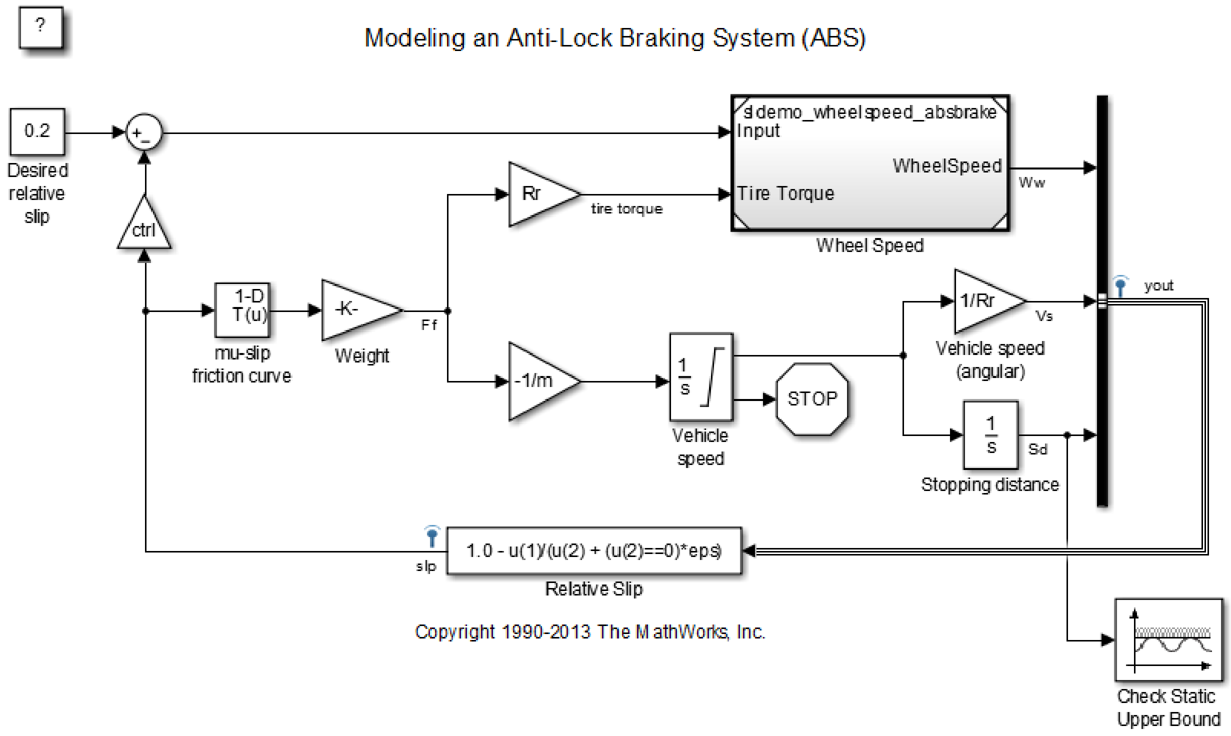
Test a Simulation for Run-Time Errors

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

Configure the Model

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

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



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

Create the Test Case

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

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

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

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

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

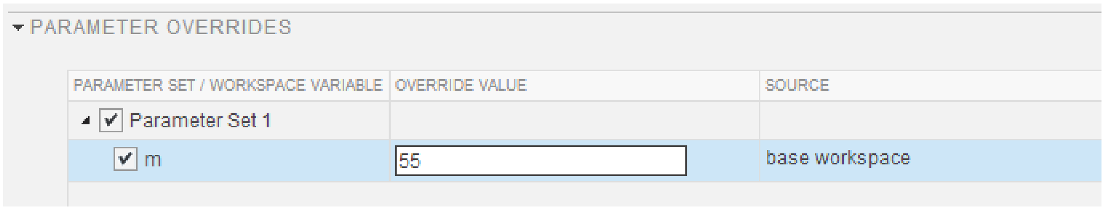
5 In the test case, under **System Under Test**, click the **Use current model** button to assign the `sldemo_absbrake` model to the test case.

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

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

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

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



PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE
✓ Parameter Set 1		
✓ m	55	base workspace

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

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

Run the Test Case

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

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

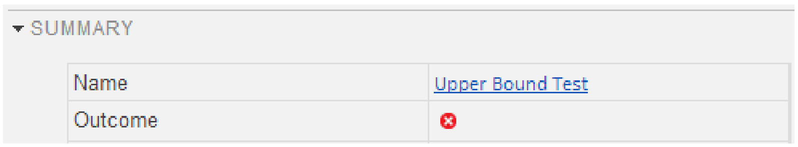
  The test manager switches to the **Results and Artifacts** pane, and the new test result appears at the top of the table.

View Test Results

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

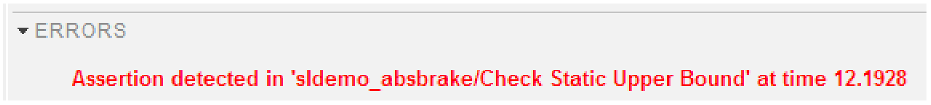
  A new tab opens that displays the outcome and results summary of the simulation test.

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



Name	Upper Bound Test
Outcome	✖

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



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

Generate Test Cases from Model Components

In this section...
"Generate the Test Cases" on page 6-9
"Synchronize Test Cases" on page 6-10

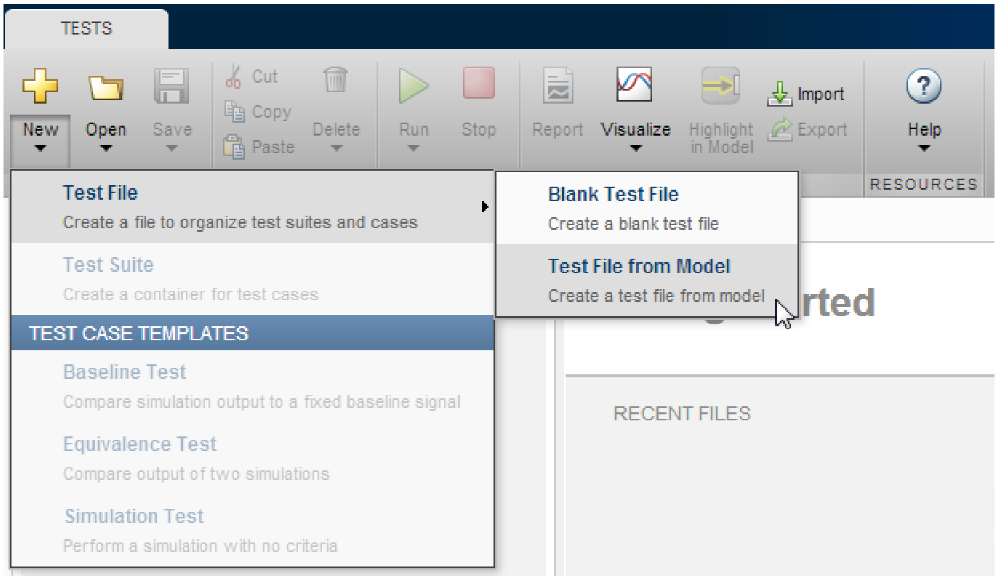
The test manager can generate a list of test cases for you based on the components in your model. Test cases can be generated from:

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

If there are multiple Signal Builder blocks in the top model, then the test manager does not create any test cases from Signal Builder blocks.

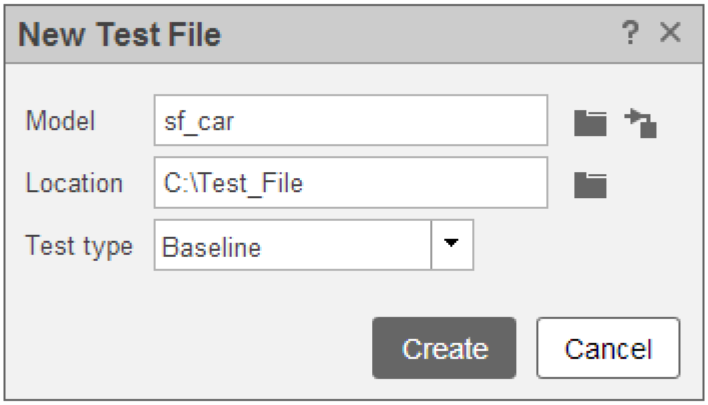
Generate the Test Cases

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



2 In the **New Test File** dialog box, select the model and location. The model must be on the MATLAB path.

3 Select the **Test type** to generate for all test cases.



New Test File

Model: sf_car

Location: C:\Test_File

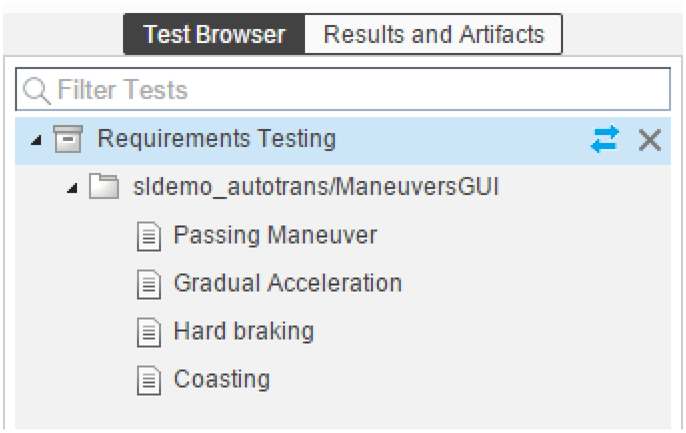
Test type: Baseline

4 Click **Create**.

Synchronize Test Cases

If you add model components to your model, such as Signal Builder groups or test harnesses, then you can generate new test cases in the test manager to synchronize your model. Also, if you remove model components, then you can disable or delete test cases in the test manager when you synchronize. In the test manager **Test Browser** pane, you can synchronize your model and test file using the synchronization button ⇄ next to the test file name.

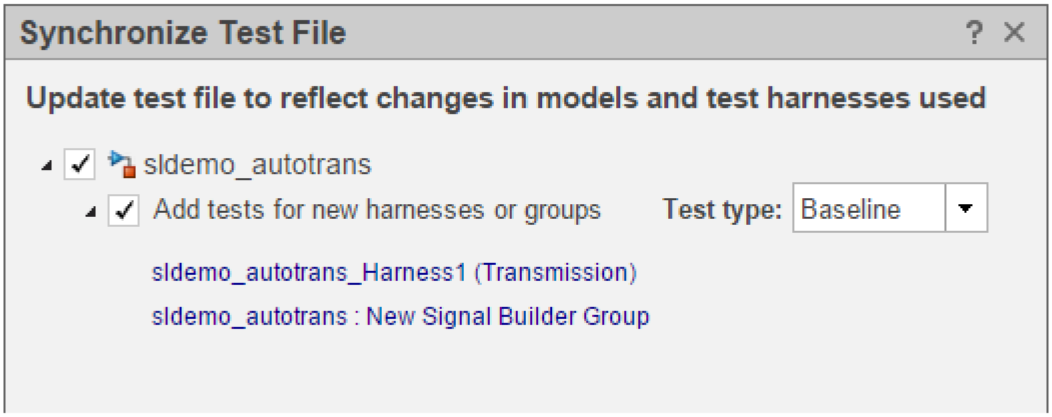
For example, the `sldemo_autotrans` model has a Signal Builder block with four groups by default. If you generate test cases from the model using **New** > **Test File** > **Test File from Model**, then test cases generate using the Signal Builder groups.



Requirements Testing
- sldemo_autotrans/ManeuversGUI
  - Passing Maneuver
  - Gradual Acceleration
  - Hard braking
  - Coasting

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

1. In the test manager, hover over the test file name that you want to synchronize.
2. Click the synchronization button ⇄ next to the test file name.
3. Review the synchronization dialog box to add or remove any test cases, and select the test case type: baseline, equivalence, or simulation.

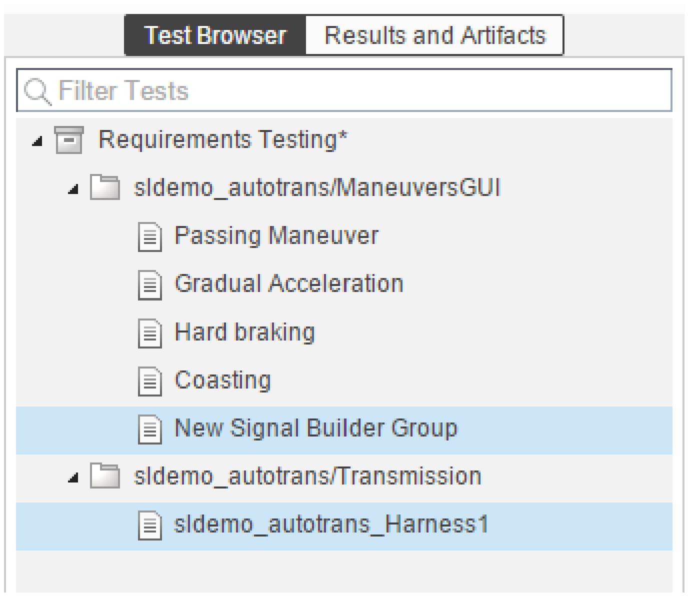


Update test file to reflect changes in models and test harnesses used

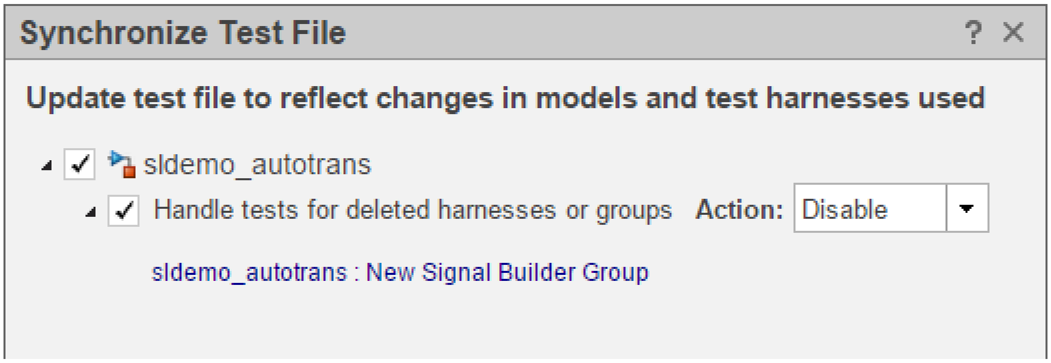
- ✓ sldemo_autotrans
  - ✓ Add tests for new harnesses or groups   Test type: Baseline
    - sldemo_autotrans_Harness1 (Transmission)
    - sldemo_autotrans : New Signal Builder Group

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

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



If you remove model components and synchronize the test file, then you can remove or disable a test case using the **Action** menu. For example, if you remove `New Signal Builder Group` from the model, then the synchronization dialog box shows the deleted Signal Builder group.



Update test file to reflect changes in models and test harnesses used

✓ sldemo_autotrans

✓ Handle tests for deleted harnesses or groups **Action:** Disable

sldemo_autotrans : New Signal Builder Group

Use External Inputs in Test Cases

In this section...

If you have external model inputs from MAT-files or Microsoft® Excel® file sheets, then you can use these as inputs in a test case. External inputs are mapped to the model using root inport mapping under the **Inputs** section. You can import multiple external input files to a test case, but you can select only one external input set to execute when the test runs.

For more information about root inport mapping modes, supported data types or formats, and mapping results, see “Import and Map Root-Level Inport Data”.

Use MAT-File for Inputs

To add a MAT-file as an external input:

1. Expand the **Inputs** section in the test case.
2. Under the **External Inputs** table, click **Add**.
3. Specify a MAT-file.
4. Under **Input Mapping**, choose a mapping mode. For more information about mapping modes, see “Import and Map Root-Level Inport Data”.
5. Click **Map Inputs**. The **Mapping Status** table shows if the port and signals map successfully.

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

Use Microsoft Excel File for Inputs

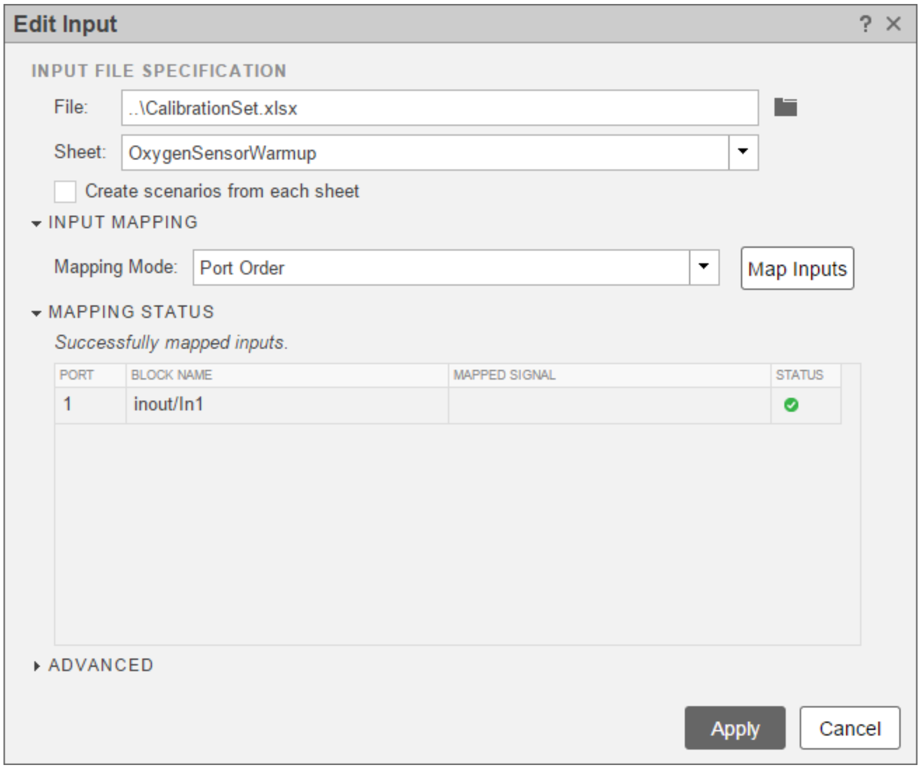
The Root Inport Mapping tool supports Microsoft Excel spreadsheets only for Windows® systems. For Microsoft Excel spreadsheets:

- The tool interprets each worksheet as a Simulink.SimulationData.Dataset data set.
- Each worksheet name must be a valid MATLAB variable name.

- The tool interprets the first row of a worksheet as signal names. If you do not specify a signal name, the tool assigns a default one using the format `Signal#`.
- If all columns do not have signal names, the tool assigns signal names using the format `Signal#`, where # increments with each additional signal.
- All signal-name columns must be filled in. If there are empty signals, the tool returns an error at import.
- The tool interprets the first column as time. In this column, the time values must increase.
- The tool interprets the remaining columns as signals.

To add a Microsoft Excel file as an external input:

1. Expand the **Inputs** section in the test case.
2. Under the **External Inputs** table, click **Add**.



File: ..\CalibrationSet.xlsx

Sheet: OxygenSensorWarmup

☐ Create scenarios from each sheet

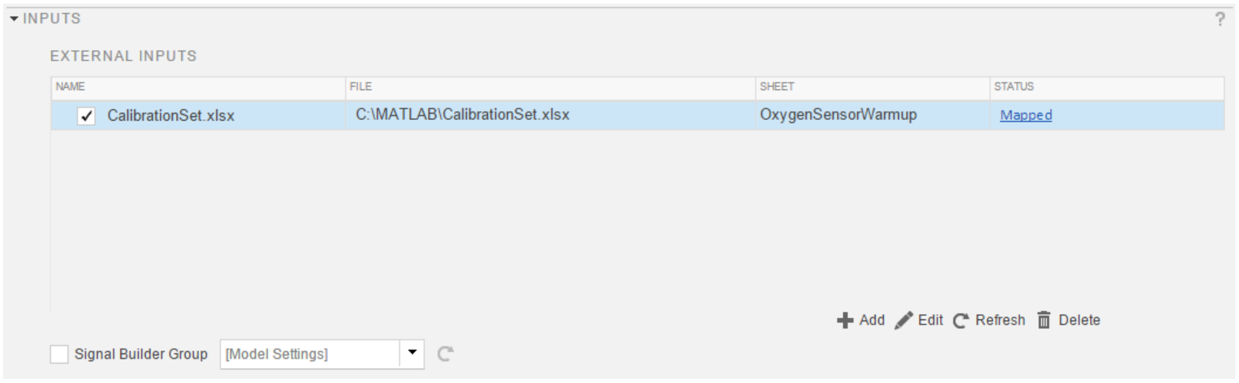
Mapping Mode: Port Order

Successfully mapped inputs.

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	inout/In1		✓

3. Specify a Microsoft Excel file.
4. Select the sheet that contains the input data.
5. If you want to use each sheet to create an input set in the table, select **Create scenarios from each sheet**.
6. Under **Input Mapping**, choose a mapping mode. For more information about mapping modes, see “Import and Map Root-Level Inport Data”.
7. Click **Map Inputs**. The **Mapping Status** table shows if the port and signals map successfully.

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



NAME	FILE	SHEET	STATUS
✓ CalibrationSet.xlsx	C:\MATLAB\CalibrationSet.xlsx	OxygenSensorWarmup	Mapped

Automate Tests Programmatically

In this section...

List of Functions and Classes

Function	Description
`sltest.testmanager.view`	Launch the Simulink Test manager
`sltest.testmanager.createTestsFrom`	Generate test cases from a model
`sltest.import.sldvData`	Create test cases from Simulink Design Verifier results
`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.copyTests`	Copy test cases or test suites to another location
`sltest.testmanager.moveTests`	Move test cases or test suites to a new location
`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.TestFile	Test file object
sltest.testmanager.TestSuite	Test suite object
sltest.testmanager.TestCase	Test case object
sltest.testmanager.ParameterSet	Parameter set object
sltest.testmanager.ParameterOverride	Parameter override object

Class	Description
sltest.testmanager.TestInput	Test input object
sltest.testmanager.BaselineCriteria	Baseline criteria object
sltest.testmanager.EquivalenceCriteria	Equivalence criteria object
sltest.testmanager.SignalCriteria	Signal criteria object
sltest.testmanager.ResultSet	Access results set data
sltest.testmanager.TestSuiteResult	Access test suite results data
sltest.testmanager.TestCaseResult	Access test case results data

Create and Run a Test Case

This example shows how to use the `sltest.testmanager` functions, classes, and methods to automate tests and generate reports. You can create a test case, edit the test case criteria, run the test case, and generate results reports programmatically. The example compares the simulation output of the model to a baseline data set.

Create the test file, test suite, and test case structure.

```
tf = sltest.testmanager.TestFile('API Test File');
ts = tf.createTestSuite('API Test Suite');
tc = ts.createTestCase('baseline','Baseline API Test Case');
```

Remove the default test suite.

```
tsDel = tf.getTestSuiteByName('New Test Suite 1');
remove(tsDel);
```

Assign the system under test to the test case.

```
tc.setProperty('Model','sldemo_absbrake');
```

Capture the baseline criteria.

```
baseline = tc.captureBaselineCriteria('baseline_API.mat',true);
```

Test a new model parameter by overriding it in the test case parameter set.

```
ps = tc.addParameterSet('Name','API Parameter Set');
po = ps.addParameterOverride('m',55);
```

Set the baseline criteria tolerance for a signal.

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

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

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

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

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

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

Generate a report from the results data.

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

The results report is a PDF and opens when it is completed. For more report generation settings, see the `sltest.testmanager.report` function reference page.

See Also

`sltest.testmanager.report`

How Tolerances Are Applied to Test Criteria

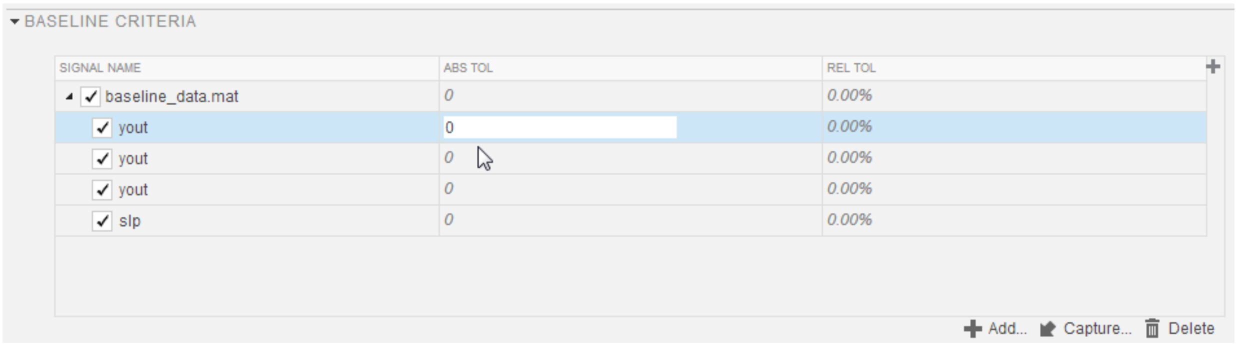
Tolerances can be specified in the **Baseline Criteria** or **Equivalence Criteria** sections of test cases. The default value for the relative tolerance and absolute tolerance for a signal comparison is zero. If you specify tolerances, then the test calculates the tolerances as follows:

```
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.



SIGNAL NAME	ABS TOL	REL TOL
✓ baseline_data.mat	0	0.00%
✓ yout	0	0.00%
✓ yout	0	0.00%
✓ yout	0	0.00%
✓ slp	0	0.00%

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

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

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

Callback Scripts

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

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

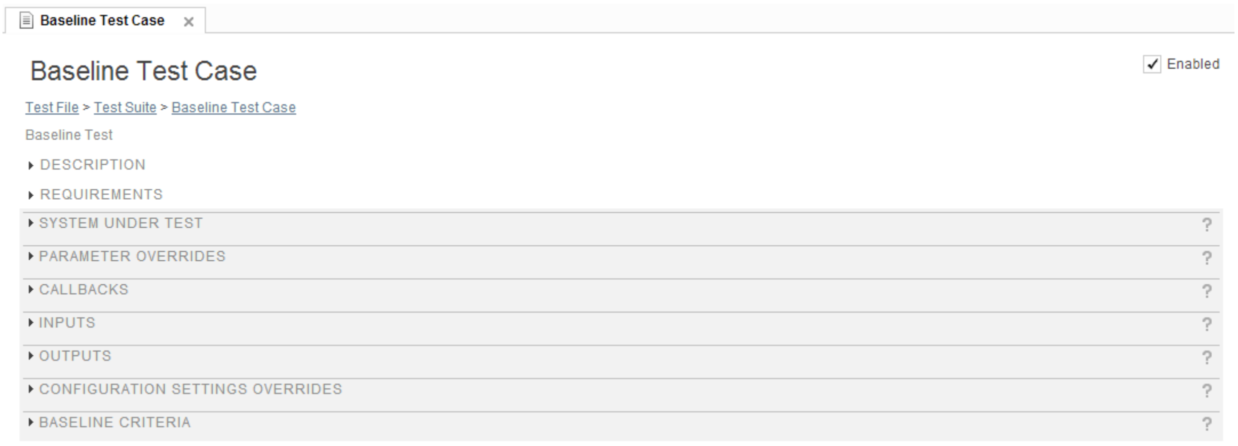
Protected Models

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

Test Case Sections

In this section...

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" on page 5-2.



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 6-24.

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

Test Harness

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

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

Simulation Settings

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

Parameter Overrides

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

1. Click **Add**.

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

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

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

For an example about parameter overrides, see “Overriding Model Parameters in a Test Case”.

Callbacks

Test-Suite Level Callbacks

There are two callback scripts available in each test suite that execute at different times during a test:

- Setup: runs before the test suite executes.
- Cleanup: runs after the test suite executes.

Test-Case Level Callbacks

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

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

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

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

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

See “Test Manager Limitations” on page 6-20 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**. You can use inputs from signal builder groups in the model, or you can use external inputs from MAT-files or Microsoft Excel files. You can use only one external input set in the **External Inputs** table to run when the test case executes. External inputs are mapped using root inport mapping. See "Identify Signal Data to Import and Map" for more information on supported file formats.

For an example of how to use external inputs, see "Use External Inputs in Test Cases" on page 6-13. For more information on the Root Inport Mapping tool see "Import and Map Root-Level Inport Data".

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.

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

Baseline Criteria

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

To capture streamed and logged signal data from the **System Under Test**, click **Capture** to compile and run the system. You are asked to save the signal data to a MAT-file.

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

Test Models Using Inputs Generated by Simulink Design Verifier

In this section...

Using Simulink Design Verifier, you can generate tests that replicate design errors, achieve test objectives, or exercise your model to meet coverage criteria. Over the course of developing your model and generating code, you might repeatedly exercise your model and code with these test inputs. You can simplify repeated testing using Simulink Testto automatically create test cases that use inputs generated using Simulink Design Verifier analysis.

Overall Workflow

Test case generation follows this workflow.

1 Choose an existing Simulink Design Verifier results file, or generate new results by analyzing your model.

   - If you use an existing results file, you can load results by either:
     - Using the Simulink Test command `sltest.import.sldvData`.
     - Using Simulink Design Verifier menu items. In the model, select **Analysis** > **Design Verifier** > **Results** > **Load**. Select the MAT file with the analysis results.
   - If you run a model analysis, the Design Verifier Results Summary window appears after the analysis completes.

2 In the results summary window, click **Export test cases to Simulink Test**.

3 Select an existing test harness, or create a test harness.

   Note: If you create a new test harness, the new harness uses inports to load the inputs from a MAT file. If you select an existing test harness, the existing test harness must use inport sources.

4 Simulink Test generates the test file and test harness. In the test manager, expand the new test file in the **Test Browser** to see the individual test cases.

Test Case Generation Example

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

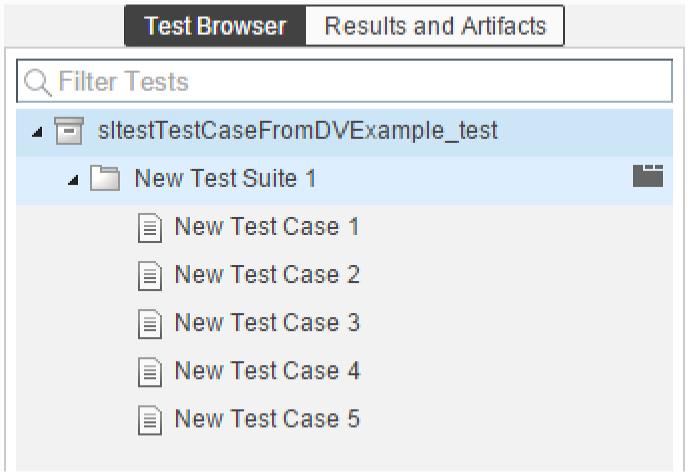
The model is a closed-loop heatpump system. The controller accepts the measured room temperature and set temperature inputs. The controller outputs a bus of three signals controlling the fan, heat pump, and the direction of the heat pump (heat or cool). The model contains a harness that tests heating and cooling scenarios.

1 Open the model.

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

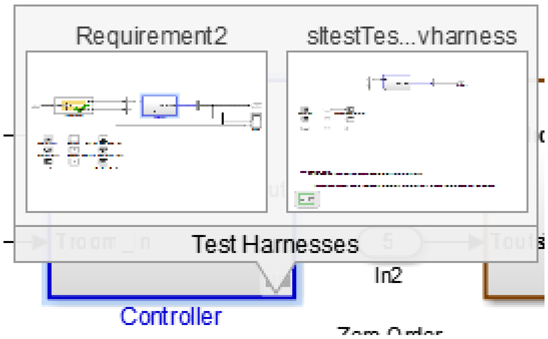
2 Set the current working folder to a writable folder.
3 In the model, generate tests for the Controller subsystem. Right-click the Controller block and select **Design Verifier > Generate Tests for Subsystem**.
4 In the Results Summary window, click **Export test cases to Simulink Test**.
5 In the Harness Selection dialog box, select `New Harness`. Click **OK**.

The test manager displays five new test cases in the test file.



sltestTestCaseFromDVExample_test
New Test Suite 1
New Test Case 1
New Test Case 2
New Test Case 3
New Test Case 4
New Test Case 5

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



7 Add a test case to the other test harness in the model. In the test manager, hover over the new test file name and click the **Synchronize Test File** button ⇄.

8 The test manager prompts you to add tests for the Requirement2 test harness. Select `Simulation` for the test type, and click **Update Test File**.

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

See Also

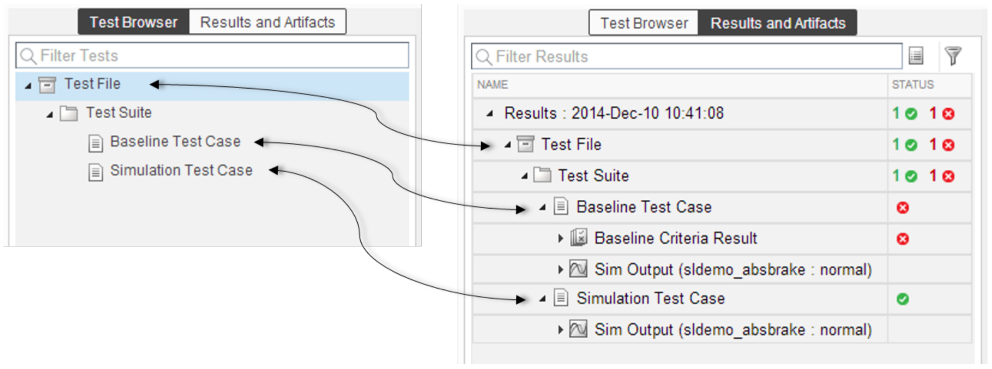
`sltest.import.sldvData`

Test Manager Results and Reports

View Test Case Results

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

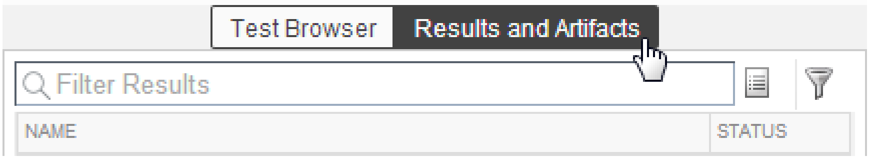
After a test case has finished running in the test manager, the test case result becomes available in the **Results and Artifacts** pane. Test results are organized in the same hierarchy as the test file, test suite, and test cases that were run from the **Test Browser** pane. In addition, the **Results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



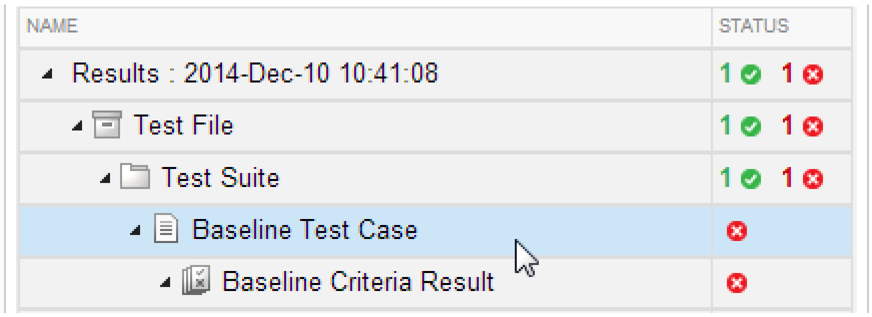
View Results Summary

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

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

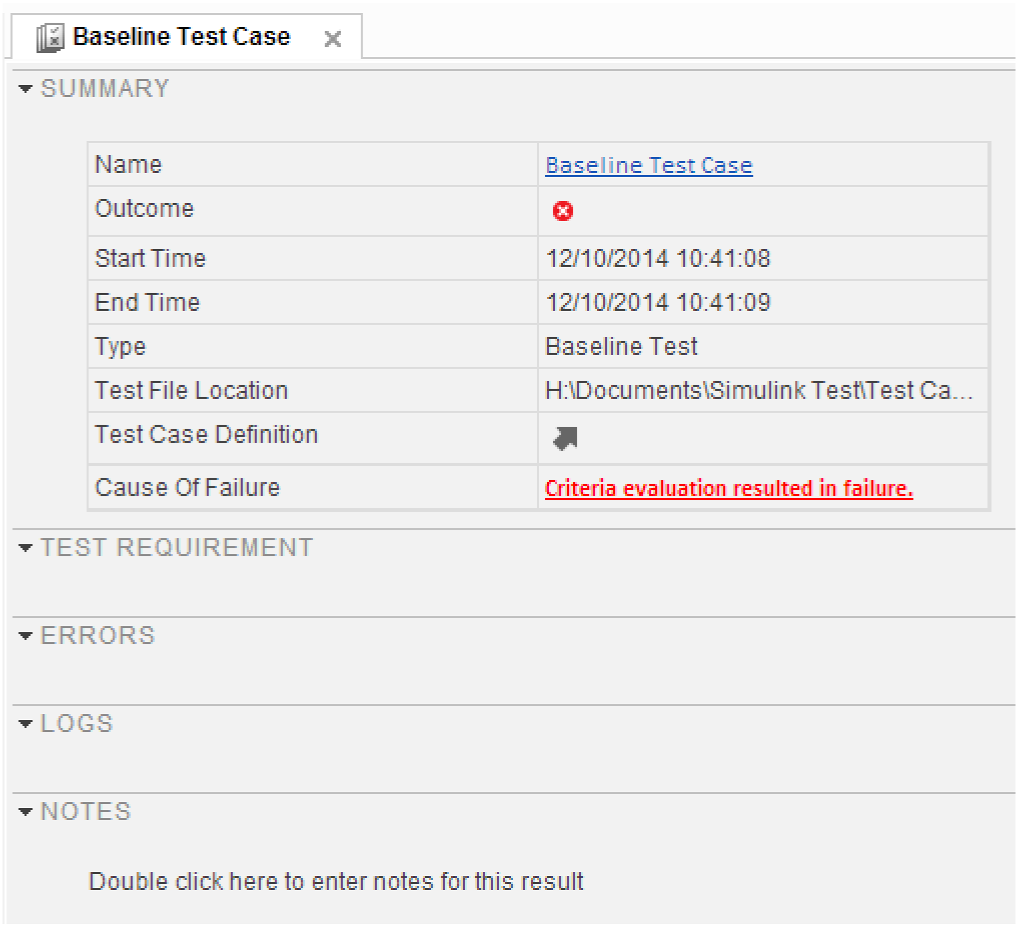


2 Double-click a test case result.



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

A tab opens containing the test case results information.



Name	Baseline Test Case
Outcome	⊗
Start Time	12/10/2014 10:41:08
End Time	12/10/2014 10:41:09
Type	Baseline Test
Test File Location	H:\Documents\Simulink Test\Test Ca...
Test Case Definition	
Cause Of Failure	Criteria evaluation resulted in failure.

Double click here to enter notes for this result

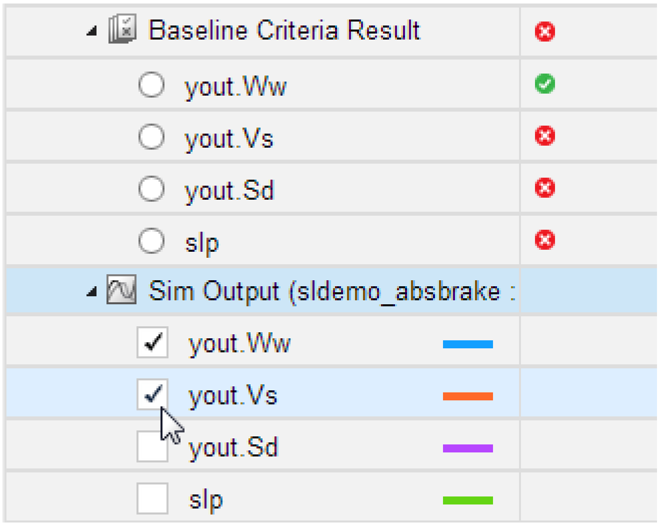
Visualize Test Case Simulation Output and Criteria

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

To view simulation output from a test case:

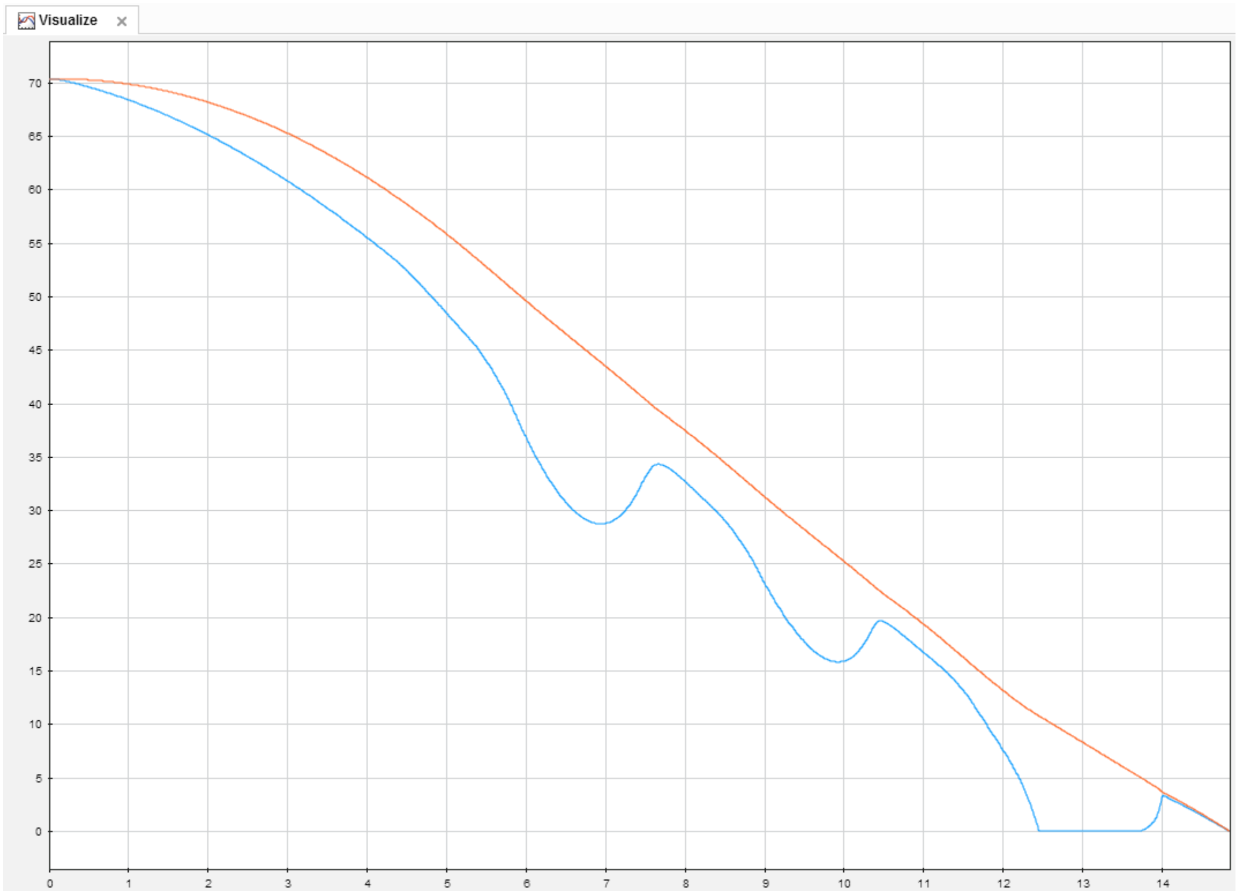
1 Select the **Results and Artifacts** pane.
2 Expand the **Sim Output** section of the test case result.

3 Select the check box of signals you want to plot.



Baseline Criteria Result	✗
○ yout.Ww	✓
○ yout.Vs	✗
○ yout.Sd	✗
○ slp	✗
Sim Output (sldemo_absbrake :	
☑ yout.Ww 🟦	
☑ yout.Vs 🟧	
☐ yout.Sd 🟪	
☐ slp 🟩	

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

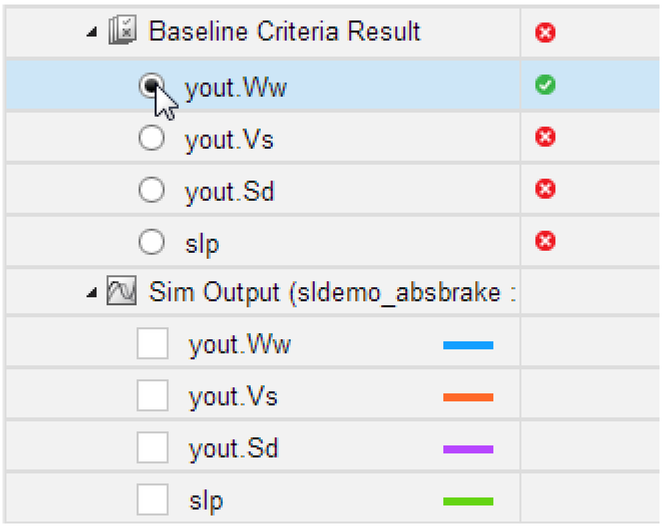


To view equivalence or baseline criteria comparisons:

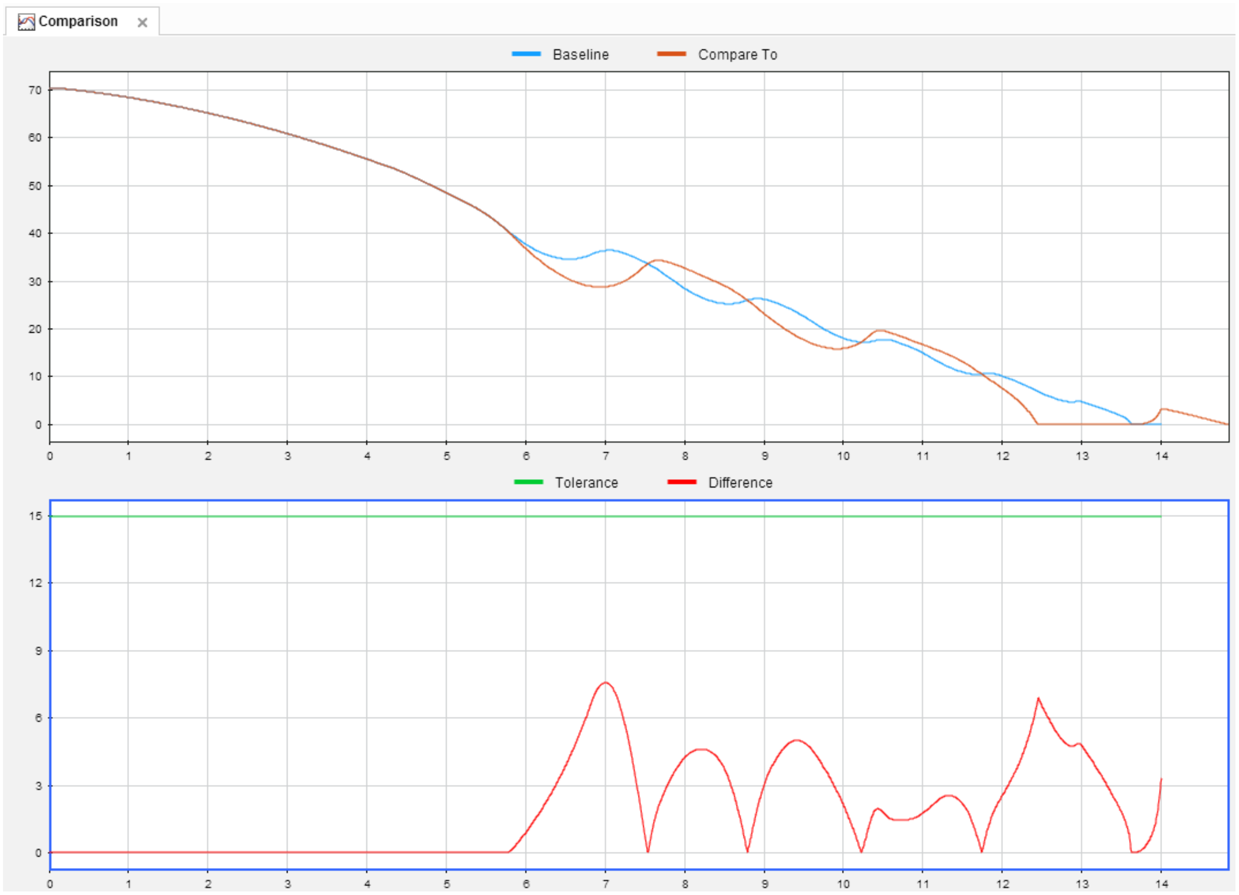
1 Select the **Results and Artifacts** pane.

2 Expand the **Baseline Criteria Result** or **Equivalence Criteria Result** section of the test case result.

3 Select the option button of the signal comparison you want to plot.



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



To see an example of creating a test case and viewing the results, see “Test Model Output Against a Baseline” on page 6-2.

Export Test Results and Generate Reports

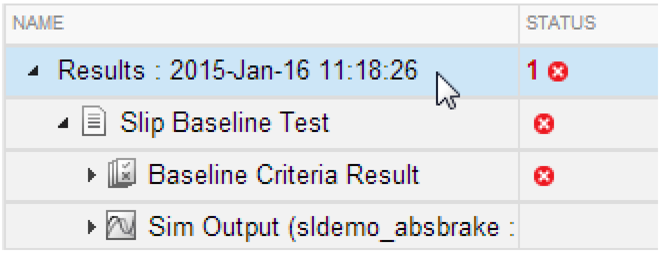
In this section...

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

Export Results

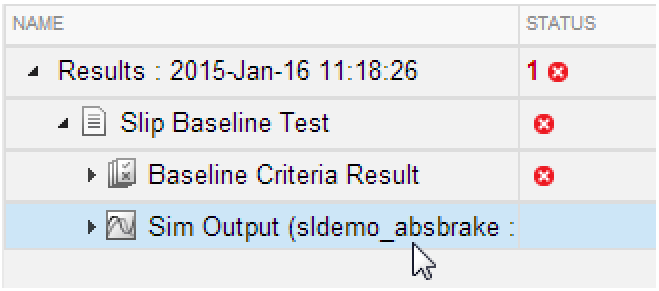
Test results are not saved with the test file. To save results, select the result in the **Results and Artifacts** pane, and click **Export** on the toolstrip.

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



NAME	STATUS
Results : 2015-Jan-16 11:18:26	1 ⊗
Slip Baseline Test	⊗
Baseline Criteria Result	⊗
Sim Output (sldemo_absbrake :	

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



NAME	STATUS
Results : 2015-Jan-16 11:18:26	1 ⊗
Slip Baseline Test	⊗
Baseline Criteria Result	⊗
Sim Output (sldemo_absbrake :	

Create a Test Results Report

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

To generate a report:

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

   Note: You can create a report from multiple results sets, but you cannot create a report from multiple test files, test suites, or test cases within results sets.
3. From the toolstrip, click **Report**.
4. Choose the options of what to include in the report.
5. Select the **File Format** to save the report as.
6. Click **Create** to generate the report.

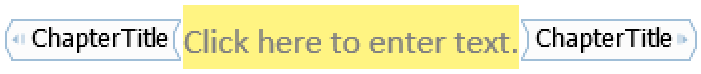
Generate Report Using Microsoft Word Template

If you have a MATLAB Report Generator™ license, then you can create reports from a Microsoft Word template. The report can be generated to a Microsoft Word document or PDF. The report generator in Simulink Test fills in information into rich text content controls in your Microsoft Word template document. For more information on how to use rich text content controls or customize part templates, see the MATLAB Report Generator documentation.

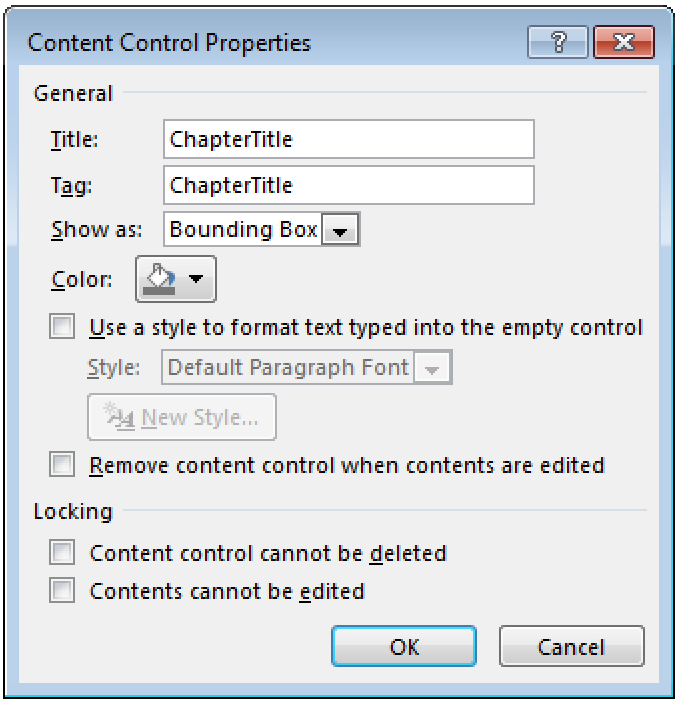
In the Microsoft Word template, you can add rich text content controls. Each Simulink Test report section can be inserted into the rich text content controls. The control names are:

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

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



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



Title: ChapterTitle
Tag: ChapterTitle
Show as: Bounding Box
Color:
☐ Use a style to format text typed into the empty control
Style: Default Paragraph Font
☐ Remove content control when contents are edited
☐ Content control cannot be deleted
☐ Contents cannot be edited

To generate a report from the test manager using a Microsoft Word template:

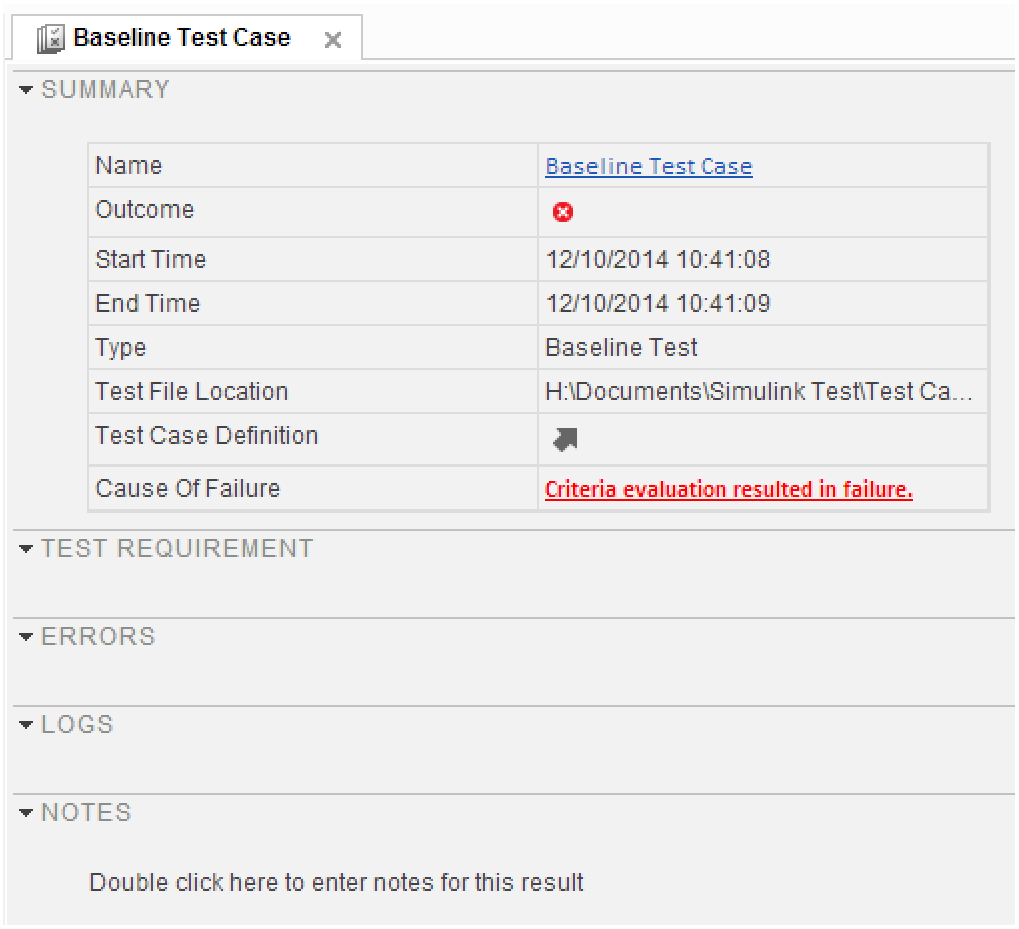
1. In the test manager, select the **Results and Artifacts** pane.
2. Select results for a test file, test suite, or test case in the **Results and Artifacts** pane.
3. From the toolstrip, click **Report**.

4. Choose the options of what to include in the report.
5. Select DOCX or PDF for the **File Format**.
6. Specify the full path and file name of your Microsoft Word template.
7. Click **Create** to generate the report.

Results Sections

In this section...

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.



Name	Baseline Test Case
Outcome	✖
Start Time	12/10/2014 10:41:08
End Time	12/10/2014 10:41:09
Type	Baseline Test
Test File Location	H:\Documents\Simulink Test\Test Ca...
Test Case Definition	
Cause Of Failure	Criteria evaluation resulted in failure.

Double click here to enter notes for this result

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” on page 6-22 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.