Simulink® Design Verifier™

User's Guide
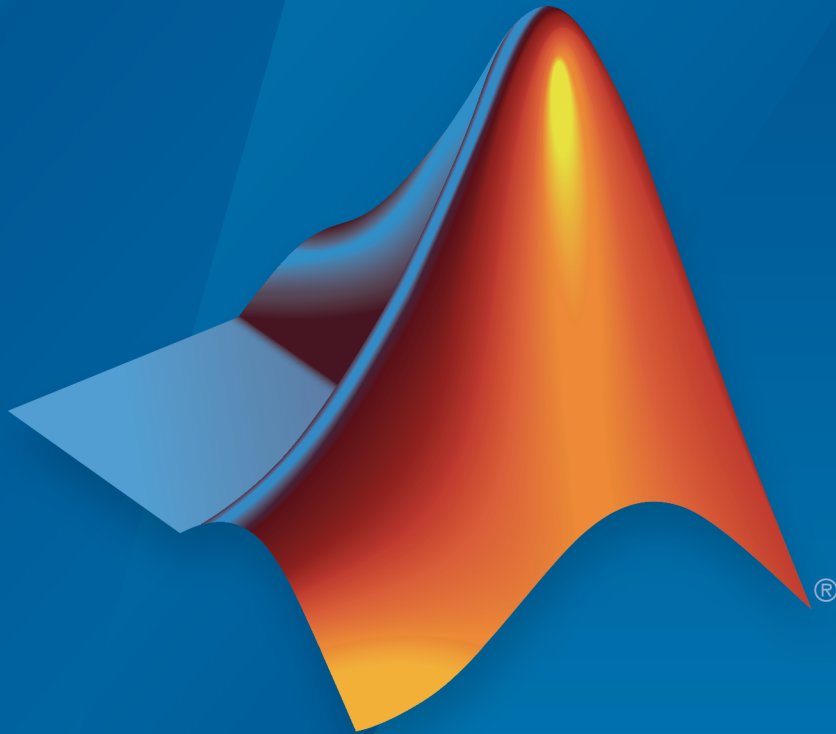
MATLAB&SIMULINK®

MathWorks®

How to Contact MathWorks

	Latest news:	www.mathworks.com
	Sales and services:	www.mathworks.com/sales_and_services
	User community:	www.mathworks.com/matlabcentral
	Technical support:	www.mathworks.com/support/contact_us
	Phone:	508-647-7000

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

Trademarks

Patents

Revision History

Contents

Checking Compatibility with the Simulink Design Verifier Software

3

Working with Block Replacements

4

Specifying Parameter Configurations

5

6

Generating Test Cases

7

Extending Existing Test Cases

8

Achieving Test Cases for Missing Model Coverage

9

Verifying Model Components

10

Considering Specified Minimum and Maximum Values for Inputs During Analysis

11

Proving Properties of a Model

12

Reviewing the Results

13

Analyzing Large Models and Improving Performance

14

15

Simulink Design Verifier Configuration Parameters

Model Slicer

16

Verification and Validation

17

Glossary

Acknowledgments

The Simulink® Design Verifier™ software uses Prover Plug-In® products from Prover® Technology to generate test cases and prove model properties.

Getting Started

Simulink Design Verifier Product Description
Identify and isolate design errors and generate tests

Simulink Design Verifier uses formal methods to identify hidden design errors in models without extensive simulation runs. It detects blocks in the model that result in integer overflow, dead logic, array access violations, division by zero, and requirement violations. For each error it produces a simulation test case for debugging.

Simulink Design Verifier generates test inputs for model coverage and custom objectives. It also lets you augment and extend existing test cases. These test cases drive your model to satisfy condition, decision, modified condition/decision (MCDC), and custom coverage objectives.

The Model Slicer tool in Simulink Design Verifier isolates problematic behavior in a model using a combination of dynamic and static analysis. It lets you highlight and trace functional dependencies of ports, signals, and blocks, and slice a large model into smaller, standalone models for analysis. You can view blocks affecting a subsystem output and trace a signal path through multiple switches and logic. The Variant Reducer tool enables you to simplify models containing multiple variants by creating sliced models based on active variant configurations.

Support for industry standards is available through IEC Certification Kit (for IEC 61508 and ISO 26262) and DO Qualification Kit (for DO-178).

Key Features

- Test case input generation from functional requirements and model coverage objectives, including condition, decision, and MCDC
- Detection of dead logic, integer and fixed-point overflows, array access violations, division by zero, and violations of design requirements
- Verification blocks for modeling functional and safety requirements
- Property proving, with generation of violation examples for analysis and debugging
- Model Slicer for analyzing functional dependencies and problematic behavior in large models
- Variant Reducer for creating sliced models based on active variant configurations
- Polyspace® and Prover formal verification engines for fixed-point and floating-point models

Simulink Design Verifier Block Library

To open the Simulink Design Verifier block library, at the MATLAB® command prompt, type `sldvlib`.
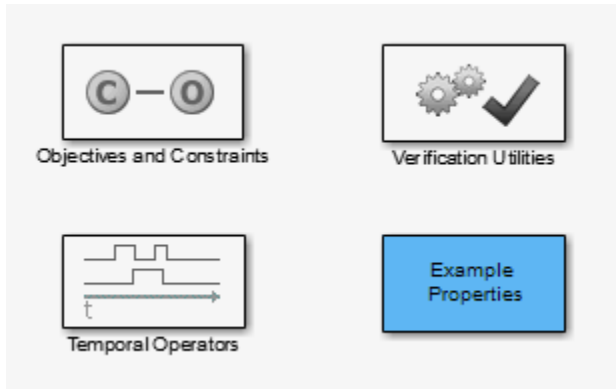
The Simulink Design Verifier block library has three categories of blocks:

·   Objectives and Constraints — Blocks that define custom objectives and constraints
·   Temporal Operators — Blocks that define temporal properties on Boolean signals
·   Verification Utilities — Miscellaneous verification utilities

The block library also has a sublibrary, Example Properties, that includes examples of how to specify common properties in your model. You can easily adapt these examples for use in your models.

Analyze a Model

About This Example

The following sections describe an example model, Cruise Control Test Generation. This example illustrates how to use Simulink Design Verifier to generate test cases that achieve complete model coverage. Through this example, you learn how to analyze models with Simulink Design Verifier and interpret the results.

Open the Model

To open the Cruise Control Test Generation model, at the MATLAB prompt, enter:

```
sldvdemo_cruise_control
```

Simulink Design Verifier
Cruise Control Test Generation
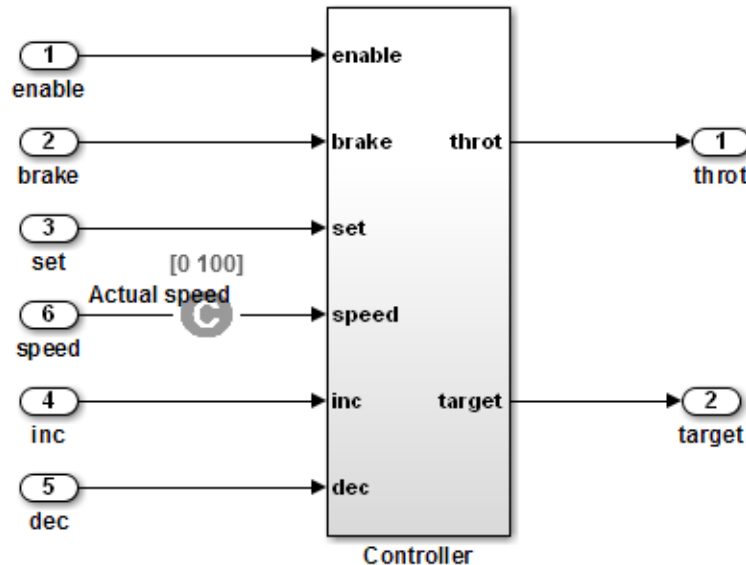
This example shows how to generate test cases that achieve complete model coverage. By default, Simulink Design Verifier generates test cases that satisfy objectives in the fewest steps. One of the test objectives forces the discrete integrator in the PI controller to exceed its upper limit. When you run Simulink Design Verifier without constraints, the limit is exceeded in a single step by forcing speed to be 500. The constraint on speed limits the values in test cases between 0 and 100. This forces the test cases to take several samples to exceed the integrator limit.

Run (double-click)

Run Simulink Design Verifier

Toggle Speed Constraint (double-click)

Toggle Constraint

View Options (double-click)

View Simulink Design Verifier Options

Generate Test Cases

- "Run Analysis" on page 1-6

- "Generate Analysis Results" on page 1-8
- "Highlight Analysis Results on Model" on page 1-9
- "Generate Detailed Analysis Report" on page 1-12
- "Create Harness Model" on page 1-19
- "Simulate Tests and Produce Model Coverage Report" on page 1-23

Run Analysis

To generate test cases for the Cruise Control Test Generation model, open the model window and double-click the block labeled **Run**.

Simulink Design Verifier begins analyzing the model to generate test cases, and the Simulink Design Verifier Results Summary window opens. The Results Summary window displays a running log showing the progress of the analysis.
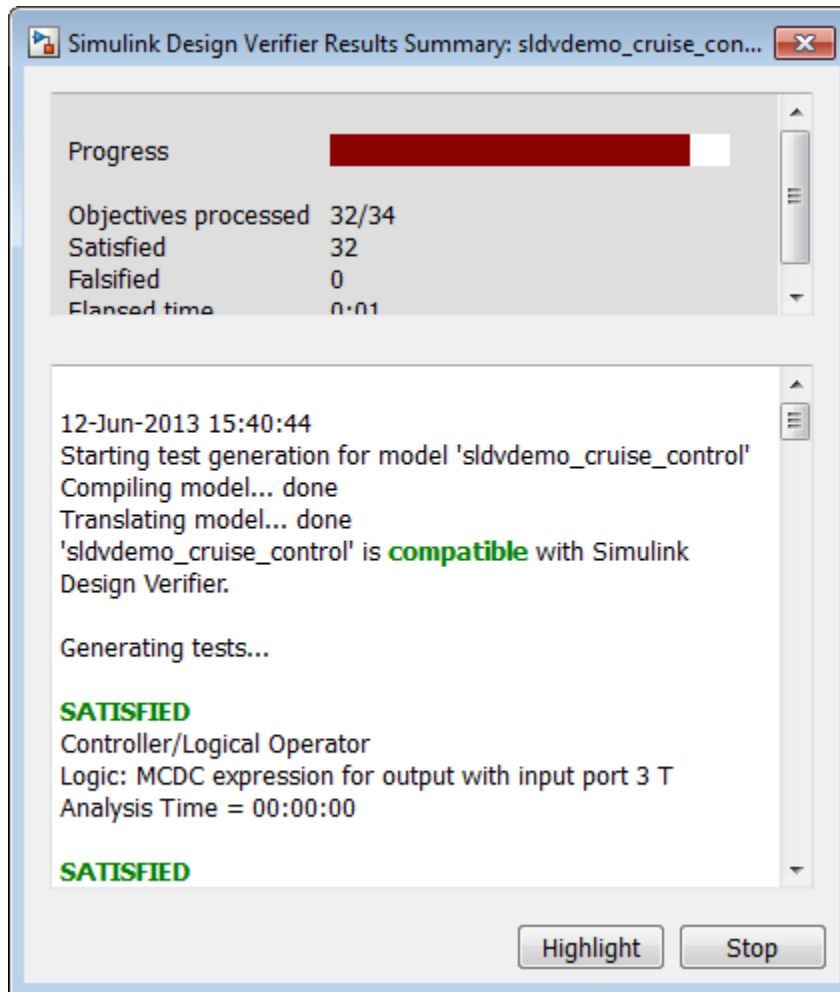
If you need to terminate an analysis while it is running, click **Stop**. The software asks if you want to produce results. If you click **Yes**, the software creates a data file based on the results achieved so far. The path name of the data file appears in the Results Summary window.

The data file is a MAT-file that contains a structure named `sldvData`. This structure stores the data that the software gathers and produces during the analysis.

For more information, see "Simulink Design Verifier Data Files" on page 13-9.

Generate Analysis Results

When Simulink Design Verifier completes its analysis of the `sldvdemo_cruise_control` model, the Results Summary window displays several options:

- **Highlight analysis results on model**
- **Generate detailed analysis report**
- **Create harness model**
- **Simulate tests and produce a model coverage report**

Note: When you analyze other models, depending on the results of the analysis, you may see a subset of these four options.
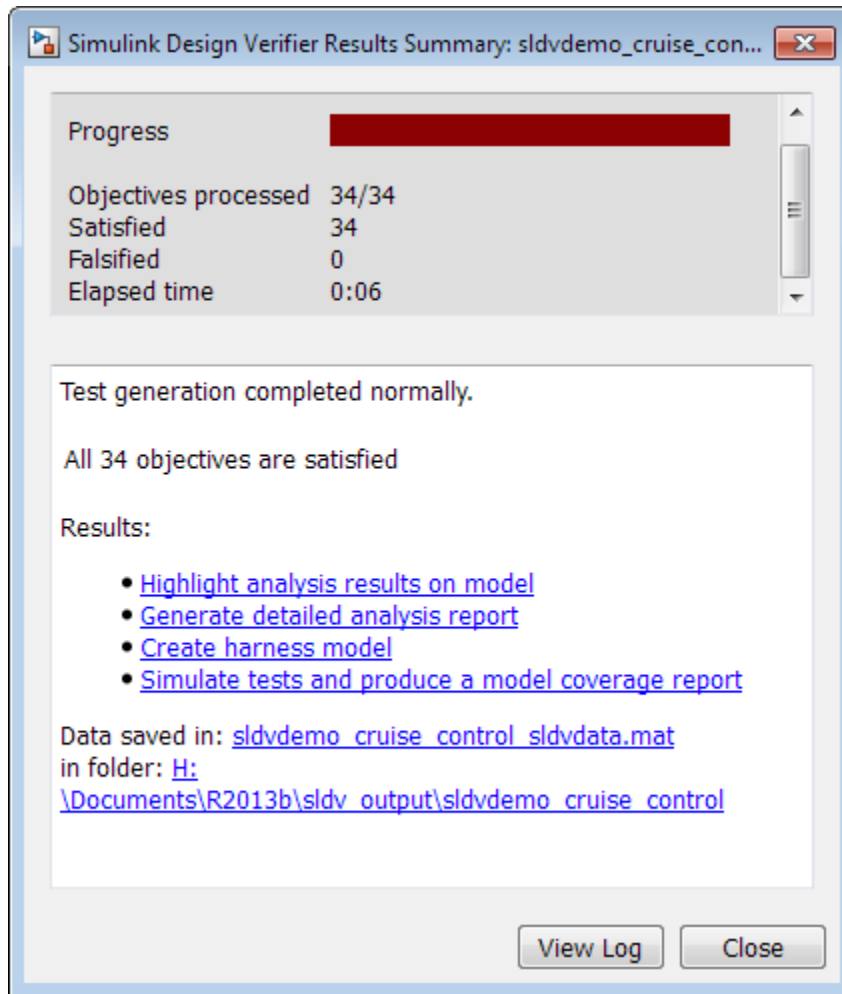
The sections that follow describe these options in detail.

Highlight Analysis Results on Model

In the Simulink Design Verifier Results Summary window, if you click **Highlight analysis results on model**, the software highlights objects in the model in three different colors, depending on the analysis results:

- "Green: Objectives Satisfied" on page 1-10

- "Orange: Objectives Undecided" on page 1-10
- "Red: Objectives Unsatisfiable" on page 1-11

When you highlight the analysis results on a model, the Simulink Design Verifier Results Inspector opens. When you click an object in the model that has analysis results, the Results Inspector displays the results summary for that object.

Green: Objectives Satisfied

Green outline indicates that the analysis generated test cases for all the objectives for that block. If the block is a subsystem or Stateflow® atomic subchart, the green outline indicates that the analysis generated test cases for all objectives associated with the child objects.

For example, in the `sldvdemo_cruise_control` model, the green outline shows that the PI controller subsystem satisfied all test objectives. The Results Inspector lists the two satisfied test objectives for the PI controller subsystem.
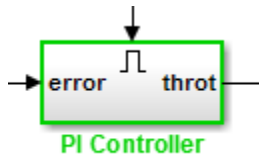
Orange: Objectives Undecided

Orange outline indicates that the analysis was not able to determine if an objective was satisfiable or not. This situation might occur when:

- The analysis times out
- The software satisfies test objectives without generating test cases due to:

    - Automatic stubbing errors
    - Limitations of the analysis engine

In the following example, the analysis timed out before it could determine if one of the objectives for the Discrete-Time Integrator block was satisfiable.
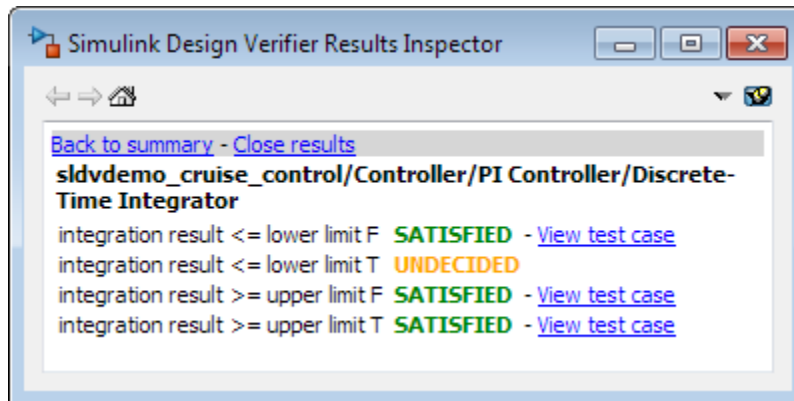
Red: Objectives Unsatisfiable

Red outline indicates that the analysis found some objectives for which it could not generate test cases, most likely due to unreachable design elements in your model.

In the following example, input 2 always satisfies the criterion for the Switch block, so the Switch block never passes through the value of input 3.
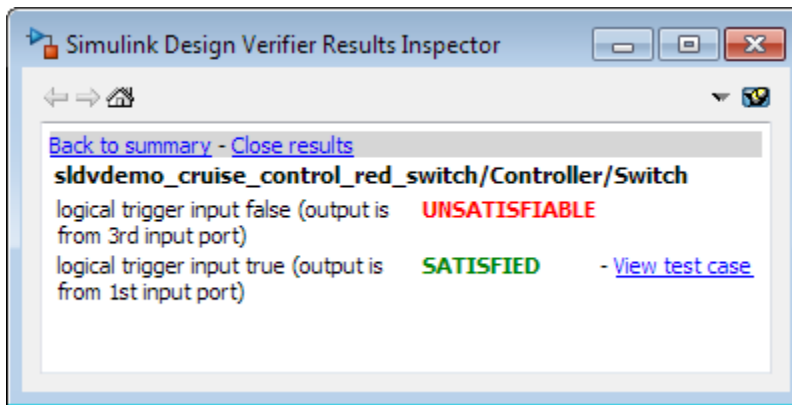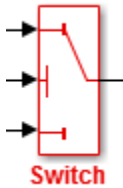
Generate Detailed Analysis Report

In the Simulink Design Verifier Results Summary window, if you click **Generate detailed analysis report**, the software saves and then opens a detailed report of the analysis. The path to the report is:

```
<current_folder>/sldv_output/...
    sldvdemo_cruise_control/sldvdemo_cruise_control_report.html
```

The HTML report includes the following chapters.

Table of Contents

For a description of each report chapter, see:

Summary

In the **Table of Contents**, click **Summary** to display the Summary chapter, which includes the following information:

- Name of the model
- Mode of the analysis (test generation, property proving, design error detection)
- Status of the analysis
- Length of the analysis in seconds
- Number of objectives satisfied

Chapter 1. Summary

Analysis Information

Model:	sldvdemo_cruise_control
Mode:	TestGeneration
Status:	Completed normally
Analysis Time:	7s

Objectives Status

Number of Objectives:	**34**
Objectives Satisfied:	34

Analysis Information

In the **Table of Contents**, click **Analysis Information** to display information about the analyzed model and the analysis options.

Chapter 2. Analysis Information

Table of Contents

Model Information

File:	sldvdemo_cruise_control
Version:	1.56
Time Stamp:	Wed Jul 18 10:45:08 2012
Author:	The MathWorks Inc.

Analysis Options

Mode:	TestGeneration
Test Suite Optimization:	CombinedObjectives
Maximum Testcase Steps:	500 time steps
Test Conditions:	UseLocalSettings
Test Objectives:	UseLocalSettings
Model Coverage Objectives:	MCDC
Maximum Analysis Time:	60s
Block Replacement:	off
Parameters Analysis:	on
Parameters Configuration File:	sldv_params_template.m
Save Data:	on
Save Harness:	off
Save Report:	off

Test Objectives Status

In the **Table of Contents**, click **Test Objectives Status** to display a table of satisfied objectives. The following figure shows a partial list of the objectives satisfied in the Cruise Control Test Generation model.

Chapter 3. Test Objectives Status

Table of Contents

Objectives Satisfied

Simulink Design Verifier found test cases that exercise these test objectives.

#	Type	Model Item	Description	Test Case
1	Decision	Controller/Switch3	logical trigger input false (output is from 3rd input port)	8
2	Decision	Controller/Switch3	logical trigger input true (output is from 1st input port)	4
3	Decision	Controller/Switch2	logical trigger input false (output is from 3rd input port)	1
4	Decision	Controller/Switch2	logical trigger input true (output is from 1st input port)	8
5	Decision	Controller/Switch1	logical trigger input false (output is from 3rd input port)	5
6	Decision	Controller/Switch1	logical trigger input true (output is from 1st input port)	8
7	Condition	Controller/Logical Operator1	Logic: input port 1 T	3
8	Condition	Controller/Logical Operator1	Logic: input port 1 F	8
9	Condition	Controller/Logical Operator2	Logic: input port 1 T	8
10	Condition	Controller/Logical Operator2	Logic: input port 1 F	5
11	Condition	Controller/Logical Operator2	Logic: input port 2 T	6
12	Condition	Controller/Logical Operator2	Logic: input port 2 F	5
13	MCDC	Controller/Logical Operator2	Logic: MCDC expression for output with input port 1 T	8

The **Objectives Satisfied** table lists the following information for the model:

- **#** — Objective number
- **Type** — Objective type
- **Model Item** — Element in the model for which the objective was tested. Click this link to display the model with this element highlighted.
- **Description** — Description of the objective
- **Test Case** — Test case that achieves the objective. Click this link for more information about that test case.

In the row for objective 34, click the test case number (**7**) to display more information about Test Case 7 in the report's **Test Cases** chapter.

Test Case 7

Summary

Length: 0.06 second (7 sample periods)
Objectives
Satisfied: 1

Objectives

Step	Time	Model Item	Objectives
7	0.06	Controller/PI Controller/Discrete-Time Integrator	integration result >= upper limit T

Generated Input Data

Time	0	0.01-0.05	0.06
Step	1	2-6	7
enable	1	1	1
brake	0	0	0
set	1	0	1
inc	1	1	-
dec	0	0	-
speed	97	0	0

In this example, Test Case 7 satisfies one objective, that the integration result be greater than or equal to the upper limit T in the Discrete-Time Integrator block. The table lists the values of the six signals from time 0 through time 0.06.

Model Items

In the **Table of Contents**, click **Model Items** to see detailed information about each item in the model that defines coverage objectives. This table includes the status of the objective at the end of the analysis. Click the links in the table for detailed information about the satisfied objectives.

Chapter 4. Model Items

Table of Contents

This section presents, for each object in the model defining coverage objectives, the list of objectives and their individual status at the end of the analysis. It should match the coverage report obtained from running the generated test suite on the model, either from the harness model or by using the sldvruntests command.

Controller/Switch3

View

#:	Type	Description	Status	Test Case
1	Decision	logical trigger input false (output is from 3rd input port)	Satisfied	8
2	Decision	logical trigger input true (output is from 1st input port)	Satisfied	4

Controller/Switch2

View

#:	Type	Description	Status	Test Case
3	Decision	logical trigger input false (output is from 3rd input port)	Satisfied	1
4	Decision	logical trigger input true (output is from 1st input port)	Satisfied	8

Test Cases

In the **Table of Contents**, click **Test Cases** to display detailed information about each generated test case, including:

· Length of time to execute the test case

· Number of objectives satisfied

· Detailed information about the satisfied objectives

· Input data

For an example, see the section for Test Case 7 in "Test Objectives Status" on page 1-15.
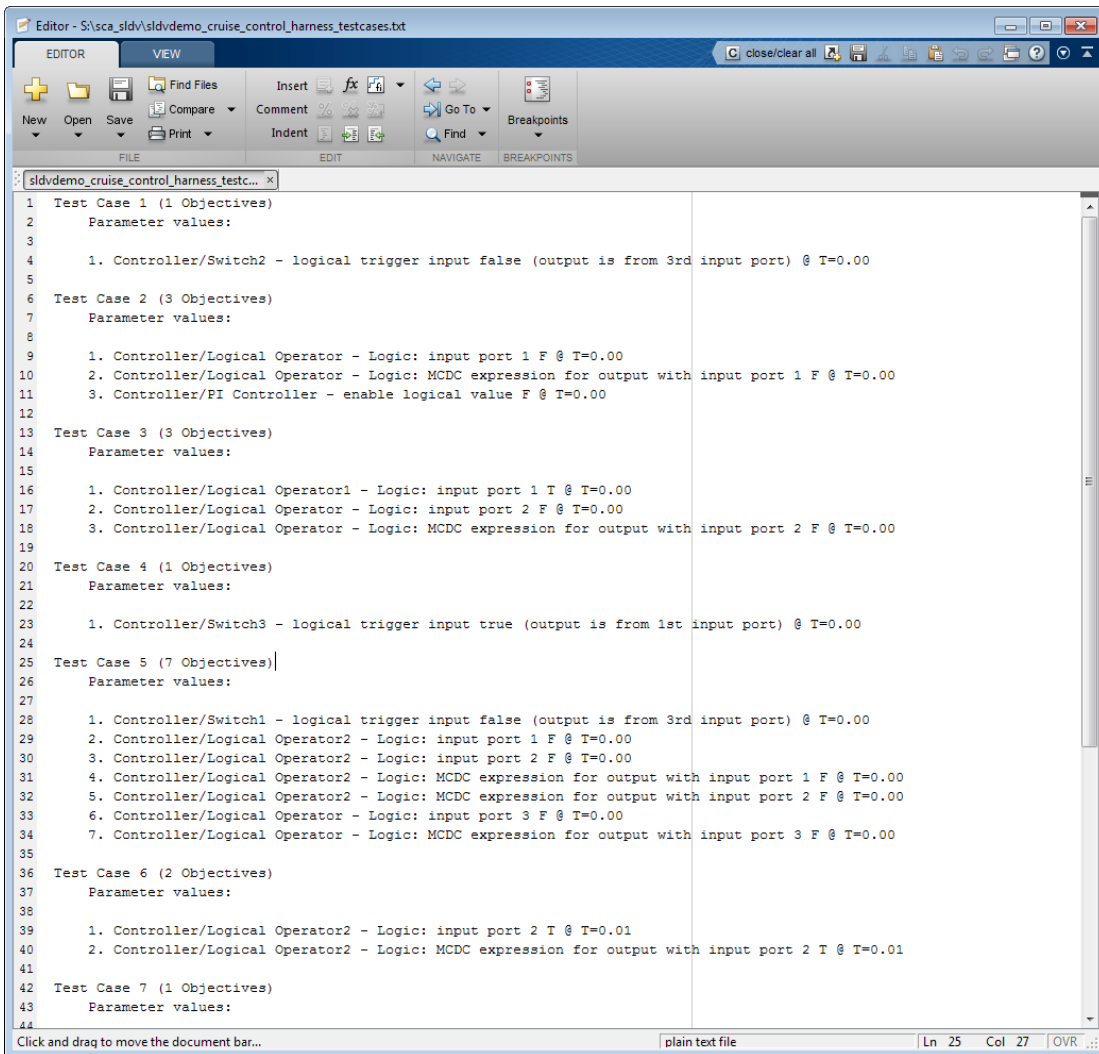
Create Harness Model

In the Simulink Design Verifier Results Summary window, if you click **Create harness model**, the software creates and opens a harness model named `sldvdemo_cruise_control_harness`.
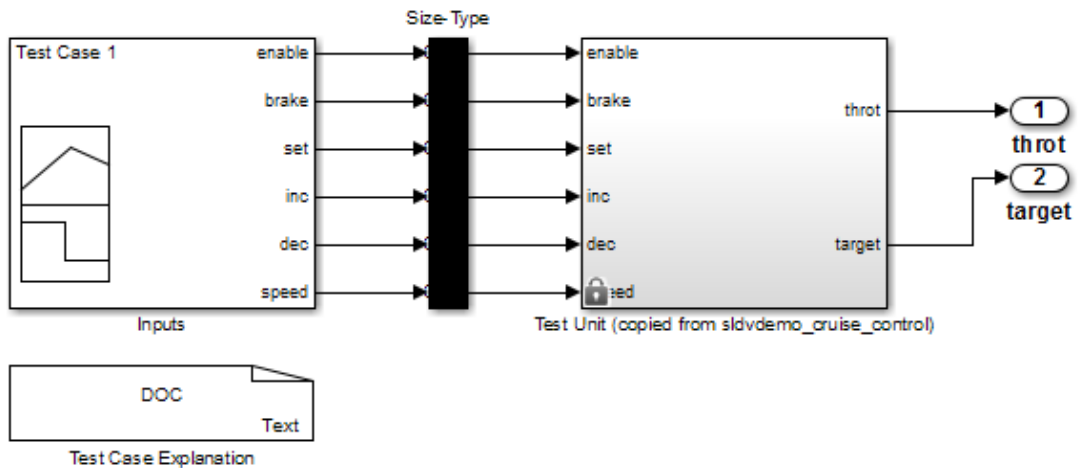
The harness model contains the following blocks:

- The Test Case Explanation block is a `DocBlock` block that documents the generated test cases. Double-click the Test Case Explanation block to view a description of each test case for the objectives that the test case satisfies.

• The Test Unit block is a `Subsystem` block that contains a copy of the original model that the software analyzed. Double-click the Test Unit block to view its contents and confirm that it is a copy of the Cruise Control Test Generation model.

Note: You can configure the harness model to reference the model that you are analyzing using a Model block instead of using a subsystem. In the Configuration Parameters dialog box, on the **Design Verifier > Results** pane, select **Generate separate harness model after analysis** and **Reference input model in generated harness**.

- The Inputs block is a `Signal Builder` block that contains the generated test case signals. Double-click the Inputs block to open the Signal Builder dialog box and view the eight test case signals.

- The Size-Type block is a subsystem that transmits signals from the Inputs block to the Test Unit block. This block verifies that the size and data type of the signals are consistent with the Test Unit block.

The Signal Builder dialog box contains eight test cases.

1 To view Test Case 7, from the **Active Group** list, select `Test Case 7`.

In Test Case 7 at 0.01 seconds:

- The enable and inc signals remain 1.
- The brake and dec signals remain 0.
- The set signal transitions from 1 to 0.
- The speed signal transitions from 100 to 0.
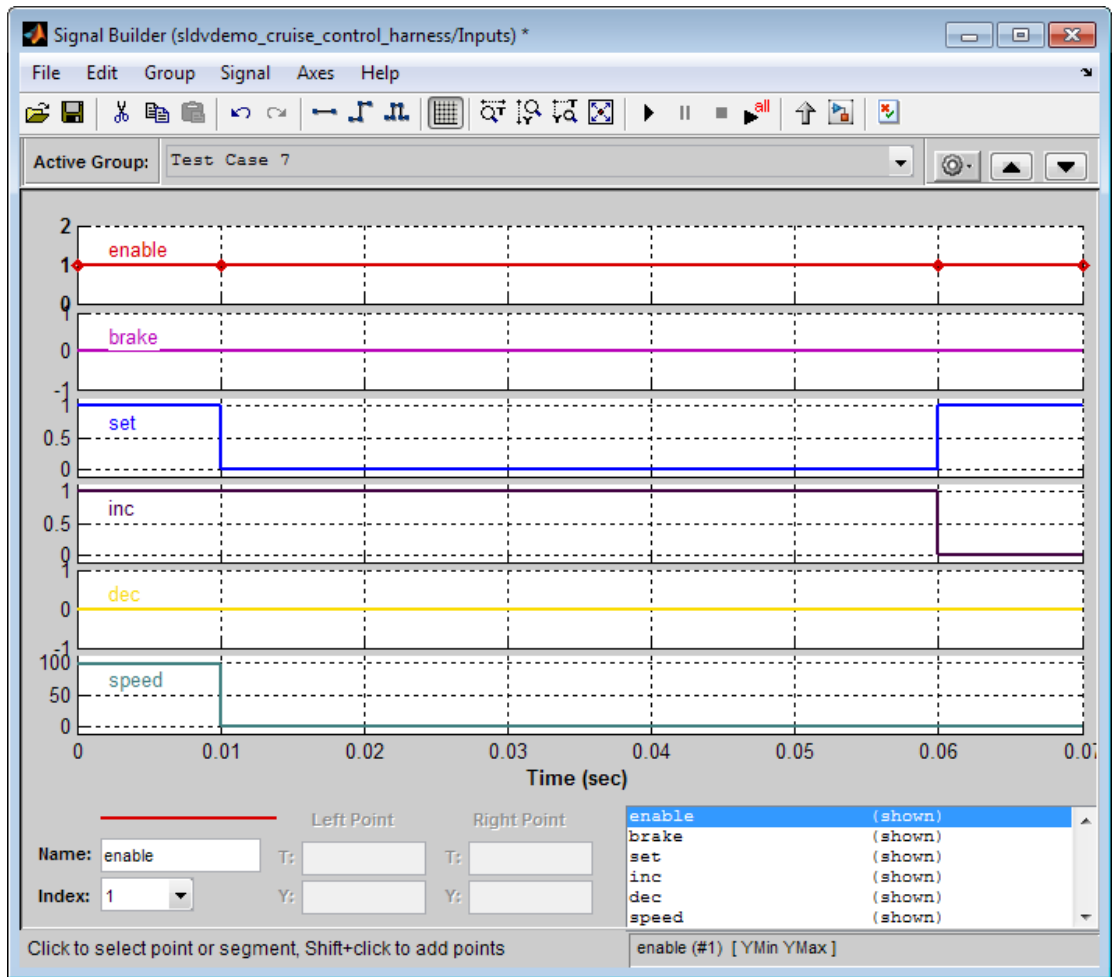
In the Signal Builder block, the signal group satisfies the test objectives described in the Test Case Explanation block.

2 To confirm that Simulink Design Verifier achieved complete model coverage, simulate the harness model using all the test cases. In the Signal Builder dialog box,

click the **Run all and produce coverage** button .

The Simulink software simulates all the test cases. The Simulink Verification and Validation™ software collects coverage data for the harness model and displays a coverage report. The report summary shows that the `sldvdemo_cruise_control_harness` model achieves 100% coverage.

Summary

Model Hierarchy/Complexity:

		D1	C1	MCDC
1. sldvdemo_cruise_control_harness	8	100%	100%	100%
2. . . . Test Unit (copied from sldvdemo_cruise_control)	7	100%	100%	100%
3. . . . . . . Controller	7	100%	100%	100%
4. . . . . PI Controller	4	100%	NA	NA

Simulate Tests and Produce Model Coverage Report

In the Simulink Design Verifier Results Summary window, if you click **Simulate tests and produce a model coverage report**, the software simulates the model and produces a coverage report for the `sldvdemo_cruise_control` model. The software stores the report with the following name:

```
<current_folder>/sldv_output/sldvdemo_cruise_control/...
  sldvdemo_cruise_control_report.html
```

When you click **Run all and produce coverage** to simulate tests in the harness model, you may see the following differences between this coverage report and the report you generated for the model itself:

- The harness model coverage report might contain additional time steps. When you collect coverage for the harness model, the model stop time equals the stop time for the longest test case. As a result, you might achieve additional coverage when you simulate the shorter test cases.

- The cyclomatic complexity coverage for the Test Unit subsystem in the harness model might be different than the coverage for the model itself due to the structure of the harness model.

Combine Test Cases

If you prefer to review results that are combined into a smaller number of test cases, set the **Test suite optimization** parameter to `LongTestcases`. When you use the `LongTestcases` optimization, the analysis generates fewer, but longer, test cases that each satisfy multiple test objectives. This optimization creates a more efficient analysis and results that are easier to review.

Open the `sldvdemo_cruise_control` model and rerun the analysis with the `LongTestcases` optimization:

1 Select **Analysis** > **Design Verifier** > **Options**.

2 In the Configuration Parameters dialog box, in the **Select** tree on the left side, under the **Design Verifier** category, select **Test Generation**.

3 Set the **Test suite optimization** parameter to `LongTestcases`.

4 Click **Apply** and **OK** to close the Configuration Parameters dialog box.

5 In the `sldvdemo_cruise_control` model, double-click the block labeled **Run**.

6 In the Results Summary window, click **Create harness model**.

In the harness model, the Signal Builder block and the Test Case Explanation block now contain one longer test case instead of the eight shorter test cases created earlier in "Generate Test Cases" on page 1-5.

7 Click **Run all and produce coverage** to collect coverage.

The analysis still satisfies all 34 objectives.

Generate Test Cases for a Subsystem

You can analyze a subsystem within a model. This technique is good for large models, where you want to review the analysis in smaller, manageable reports.

This example shows how to analyze the Controller subsystem in the `sldvdemo_cruise_control` model.

1   Open the example model:

`sldvdemo_cruise_control`

2   Right-click the Controller subsystem, and select **Design Verifier** > **Enable 'Treat as Atomic Unit' to Analyze**.

The Function Block Parameters dialog box for the Controller subsystem opens.

3   Select **Treat as atomic unit**.

An *atomic subsystem* executes as a unit relative to the parent model. Subsystem block execution does not interleave with parent block execution. You can extract atomic subsystems for use as standalone models.

You must set the **Treat as atomic unit** parameter to analyze a subsystem with Simulink Design Verifier.

After you set the parameter, other parameters become available, but you can ignore them.

4   Click **OK** to close the dialog box.

5   Select **File** > **Save As** and save the Cruise Control Test Generation model with a new name.

6   To start the subsystem analysis and generate test cases, right-click the Controller subsystem, and select **Design Verifier** > **Generate Tests for Subsystem**.

7   The Simulink Design Verifier software analyzes the subsystem. When the analysis is complete, view the analysis results for the Controller subsystem by clicking one of the following options:

- **Highlight analysis results on model**
- **Generate detailed analysis report**
- **Create harness model**
- **Simulate tests and produce a model coverage report**

Note: After processing a certain number of objectives, if the analysis stops, or if the analysis times out, you can use the Test Generation Advisor to better understand which subsystems are causing the problem. For more information, see "Use Test Generation Advisor to Identify Analyzable Components" on page 7-21.

---

8   Review the results of the subsystem analysis and compare them to the results of the full-model analysis described in "Analyze a Model" on page 1-4:

- The subsystem analysis analyzes the Controller as a standalone model.

- The Controller subsystem contains all the test objectives in the Cruise Control Test Generation model. Both analyses generate the same test cases.

Analyze a Stateflow Atomic Subchart

In a Stateflow chart, an *atomic subchart* is a graphical object that allows you to reuse the same state or subchart across multiple charts and models. You can use Simulink Design Verifier to analyze atomic subcharts individually. You do not have to analyze the chart that contains the atomic subchart, or the model that contains the chart.

If you are having problems analyzing a large model, analyzing an atomic subchart in a controlled environment is helpful. As described in "Bottom-Up Approach to Model Analysis" on page 14-15, by analyzing atomic subcharts or other components in the model hierarchy individually, you can analyze a model to:

- Solve problems that slow down or prevent test generation, property proving, or design error detection.
- Analyze model components that are unreachable in the context of the container model or chart.

Note: For more information about atomic subcharts, see "What Is an Atomic Subchart?" in the Stateflow documentation.

Analyze an Atomic Subchart Using the Simulink Design Verifier Software

The sf_atomic_sensor_pair example model models a redundant sensor pair using atomic subcharts. This example analyzes the Sensor1 subchart in the RedundantSensors chart.

1   Open the sf_atomic_sensor_pair example model:

   sf_atomic_sensor_pair

   This model demonstrates how to model a simple redundant sensor pair using atomic subcharts.

2   Double-click the RedundantSensors chart to open it.
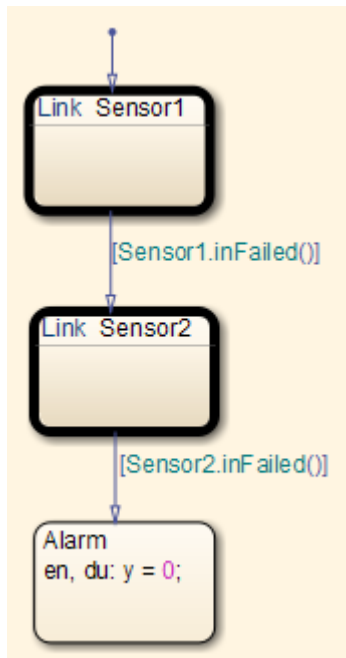
This Stateflow chart has two atomic subcharts:

- Sensor1
- Sensor2

3   To analyze the Sensor1 subchart using Simulink Design Verifier, right-click the subchart and select **Design Verifier** > **Generate Tests for Subchart**.

During the analysis, the software creates a Simulink model named Sensor1 that contains the Sensor1 subchart. The new model contains Inport and Outport blocks that respectively correspond to the data objects u and y in the subchart.
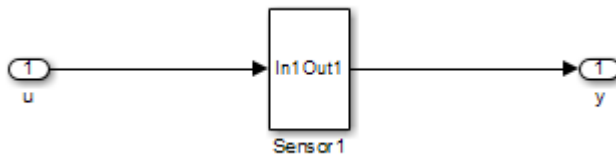
The software saves the new model and other files generated by the analysis in:

<current_folder>/sldv_output/Sensor1

4 When the analysis is complete, view the analysis results for the Sensor1 subchart by clicking one of the following options:

- **Highlight analysis results on model**
- **Generate detailed analysis report**
- **Create harness model**
- **Simulate tests and produce a model coverage report**

Basic Workflow for Simulink Design Verifier

The basic workflow for analyzing your model is described in the following steps, with links to related documentation.

Step	Action	See...
1	Check the compatibility of your model.	"Check Model Compatibility" on page 3-2
2	If you want to work around compatibility limitations in your model or customize model elements for analysis, you can use Simulink Design Verifier block replacement rules. If you want to generate additional values for parameters in your model during analysis, use Simulink Design Verifier parameter configurations.	• "What Is Block Replacement?" on page 4-2 • "Parameter Constraint Values" on page 5-2
3	Set Simulink Design Verifier options.	"Simulink Design Verifier Options" on page 15-2
4	If you plan to generate test cases or prove properties in your model, first run design error detection for integer overflow and division by zero.	• "What Is Design Error Detection?" on page 6-2 • "Detect Integer Overflow and Division-by-Zero Errors" on page 6-25
5	Analyze your model to: • Detect design errors • Generate test cases • Prove properties	• "Run a Design Error Detection Analysis" on page 6-5 • "Workflow for Test Case Generation" on page 7-4 • "Workflow for Proving Model Properties" on page 12-4
6	Generate the results.	"Generate Analysis Results" on page 1-8
7	Interpret the results.	"Results Interpretation and Use"

How the Simulink Design Verifier Software Works

Analyze a Simple Model
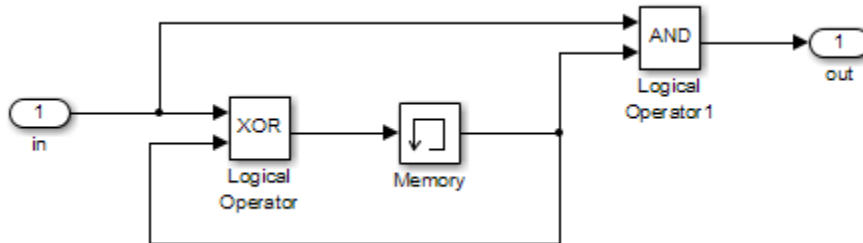
This simple model includes two Logical Operator blocks and a Memory block. The persistent information in this model is limited to the Boolean value of the Memory block. The input to the model is a single Boolean value. The following table describes the complete behavior of the model, including the behavior that results from an arbitrarily long sequence of inputs.

#	Input	Memory Value	Output of XOR Block = Next Memory Value	Output of AND Block
1	false	false	false	false
2	true	false	true	false
3	false	true	true	false
4	true	true	false	true

The test objective is to generate test cases that result in a `true` output. A `true` output results when the input is `true`, and the output of the Memory block is `true`. Test case generation follows a path to reach this condition, which depends on the initial model conditions:

- If the initial memory value is `true`, the test case is a single time step where the input is `true`.

- If the initial memory value is `false`, the test case is two time steps:

  1 The input value is `true` and the memory value is false (row 2). Thus, the output of the XOR block is `true`, making the memory value `true`.

  2 Now that the input value and memory value are both `true` (row 4), the output is `true`, and the analysis achieves the test objective.

An infinite number of test cases can cause the output to be true, and regardless of the state value, the output can be held false for an arbitrary time before making it true. When Simulink Design Verifier searches, it returns the first test case it encounters that satisfies the objective. This case is invariably the simulation with the fewest time steps. Sometimes you may find this result undesirable because it is unrealistic or does not satisfy some other test requirement.

The same basic principles from this example apply to property proving and test case generation. During test case generation, option parameters explicitly specify the search criteria. For example, you can specify that Simulink Design Verifier find paths for all block outputs or find only those paths that cause the block output to be true.

During a property proving analysis, you specify a functional requirement, or property, that you want Simulink Design Verifier to prove, for example, that the output is always true. If the search completes without finding a path that violates the property, the property is proven. If the software finds a path where the output is false, it creates a counterexample that causes the output to be false.

During an error detection analysis, Simulink Design Verifier identifies objectives where data overflow or division-by-zero errors can and cannot occur. The analysis creates test cases that demonstrate how the errors can occur.

Model Blocks

If your model contains Model blocks that reference external models, test creation occurs for the top-level model, considering each referenced model in its execution context.

If multiple Model blocks reference the same model, generated tests attempt to satisfy test objectives for each instance of the referenced model in its individual context in the top-level model. If you have three Model blocks that reference a certain model, the analysis produces results for all three instances.

If you collect coverage using the generated test cases, the cumulative coverage reflects the multiple instances of the same referenced model. The simulation produces one set of coverage results for each referenced model; if you have three Model blocks that reference a certain model, the simulation produces one set of results for that referenced model.

For example, consider a top-level model with three Model blocks referencing the same model. The referenced model has three test objectives. Analyzing the top-level model produces nine test objectives. If you simulate the model with the nine test cases, the coverage results for that referenced model specify three test objectives.

Block Reduction

Block reduction achieves faster execution during model simulation and in generated code. When block reduction is enabled, certain block groups can be collapsed into a single block, or even removed entirely.

With Simulink Design Verifier, block reduction happens automatically, and blocks in unused code paths are eliminated from the model. Simulink Design Verifier results do not include test objectives for blocks that have been reduced.

Consider the Switch block in the following model.
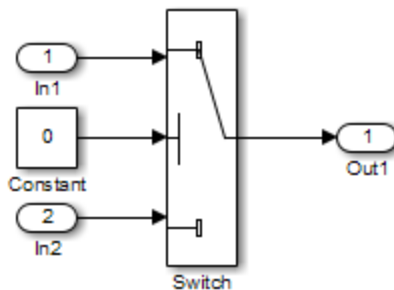
For this Switch block, the control input is always 0. If the **Criteria for passing first input** block parameter is u2 ~= 0, the Switch block always passes the third input through to the output port. When you analyze this model, Simulink Design Verifier removes the Switch block from the model and does not report any test objectives for the Switch block.

For more information about block reduction, see the description of the "Block reduction" parameter.

Inlined Parameters

Abstract

Optimize Simulink models by transforming tunable parameters into constant values.

Setting **Default parameter behavior** to `Inlined` on the **Optimization > Signals and Parameters** pane of the Configuration Parameters dialog box optimizes Simulink models by transforming tunable parameters into constant values. For example, suppose you have a Gain block whose **Gain** parameter is `a`, where `a` is defined in the model workspace. During code generation, Simulink converts that **Gain** parameter to a constant value, as defined in the workspace.

When Simulink Design Verifier translates a model, it transforms all tunable parameters in the model into constant values, even if you set **Default parameter behavior** to `Inlined`.

To tune parameters for an analysis, define parameter values in a parameter configuration file and specify that file in the **Configuration Parameters** > **Design Verifier** > **Parameters** pane to apply those parameter values during the analysis. For example, to constrain the values of a **Gain** parameter `a` to integer values from 4 to 10, in the parameter configuration file, specify the following:

```
params.a = int8([4 10]);
```

The analysis generates the specified values and returns results for those values.

For detailed information about how to specify parameters during a Simulink Design Verifier analysis, see "Define Constraint Values for Parameters" on page 5-4.

Large Models

In larger, more complicated models, Simulink Design Verifier uses mathematical techniques to simplify the analysis:

- It identifies portions of the model that do not affect the desired objectives.
- It discovers relationships within the model that reduce the complexity of the search.
- It reuses intermediate results from one objective to another.

In this way, the problem is reduced to a search though the logical values that describe your model.

For detailed information about analyzing large models, see "Analyze a Large Model" on page 14-3.

Handle Incompatibilities with Automatic Stubbing

In this section...
"What Is Automatic Stubbing?" on page 2-8
"How Automatic Stubbing Works" on page 2-8
"Analyze a Model Using Automatic Stubbing" on page 2-10

What Is Automatic Stubbing?

Automatic stubbing lets you analyze a model that contains objects that Simulink Design Verifier does not support.

When you enable the automatic stubbing option (it is enabled by default), the software considers only the interface of the unsupported objects, not their actual behavior. This technique allows the software to complete the analysis. However, the analysis may achieve only partial results if any unsupported model element affects the simulation outcome.

How Automatic Stubbing Works

If you enable automatic stubbing, when the Simulink Design Verifier analysis comes to an unsupported block, the software "stubs" that block. The analysis ignores the behavior of the block, and as a result, the block output can take any value.

Stub Trigonometric Function Block

The Simulink Design Verifier software does not support Trigonometric Function blocks when the **Function** parameter is set to `acos`, such as the one in the following graphic.

When stubbing this block during analysis, `out_signal` can take any value, with the following results.

Analysis Model	Result of Stubbing out_signal
Design error detection	• If a design-error objective that depends on `out_signal` is proven valid, that objective is valid for all simulations. In this case, the stubbing did not affect the results of the analysis.
	• If a design-error objective that depends on `out_signal` is falsified, the analysis cannot create a test case. The analysis cannot determine which input to the stubbed block produces the output that falsifies the objective.
Test case generation	• If a test objective that depends on the value of `out_signal` is satisfied, the analysis cannot create a test case. The analysis cannot determine which input to the stubbed block produces the output that satisfies the objective.
	• If a test objective that depends on the value of `out_signal` is unsatisfiable, there is no simulation that can satisfy that objective. In this case, the stubbing did not affect the results of the analysis.
Property proving	• If a proof objective that depends on `out_signal` is proven valid, that objective is valid for all simulations. In this case, the stubbing did not affect the results of the analysis.
	• If a proof objective that depends on `out_signal` is falsified, the analysis cannot create a counterexample. The analysis cannot determine which input to the stubbed block produces the output that falsifies the objective.

Stub S-Function Block Containing Function-Call Triggers

The Simulink example model `sfcndemo_sfun_fcncall` has an S-Function block. The S-function `sfun_fcncall` triggers the execution of the function-call subsystems f1 subsys1 and f2 subsys2 on the first and second elements of the first output port.
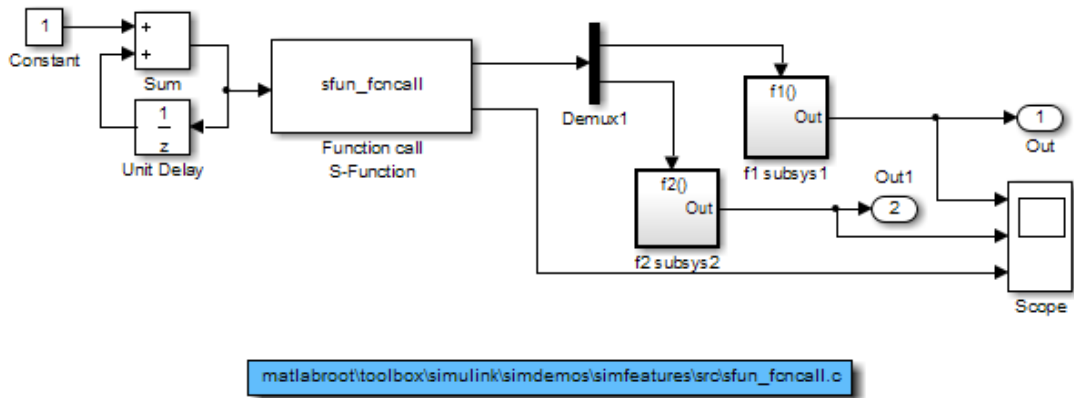
If you do not enable support for an S-function in Simulink Design Verifier and automatic stubbing is enabled, the analysis ignores the behavior of the S-function. As a result, the code that triggers the two function-call subsystems is ignored, resulting in two unsatisfiable objectives. Since the function calls are ignored, the contents of those subsystems are effectively eliminated from the analysis.

To enable support for an S-function in Simulink Design Verifier, see "Support Limitations for S-Functions" on page 3-35

Analyze a Model Using Automatic Stubbing

This section describes a workflow for using automatic stubbing, with a simple Simulink model as an example.

The following model contains a Discrete State-Space block, which is not compatible with Simulink Design Verifier.

Check Model Compatibility

From the Simulink Editor, there are two ways to check whether a model is compatible with Simulink Design Verifier:

- Run the Simulink Design Verifier compatibility check by selecting **Analysis** > **Design Verifier** > **Check Compatibility** > **Model**.
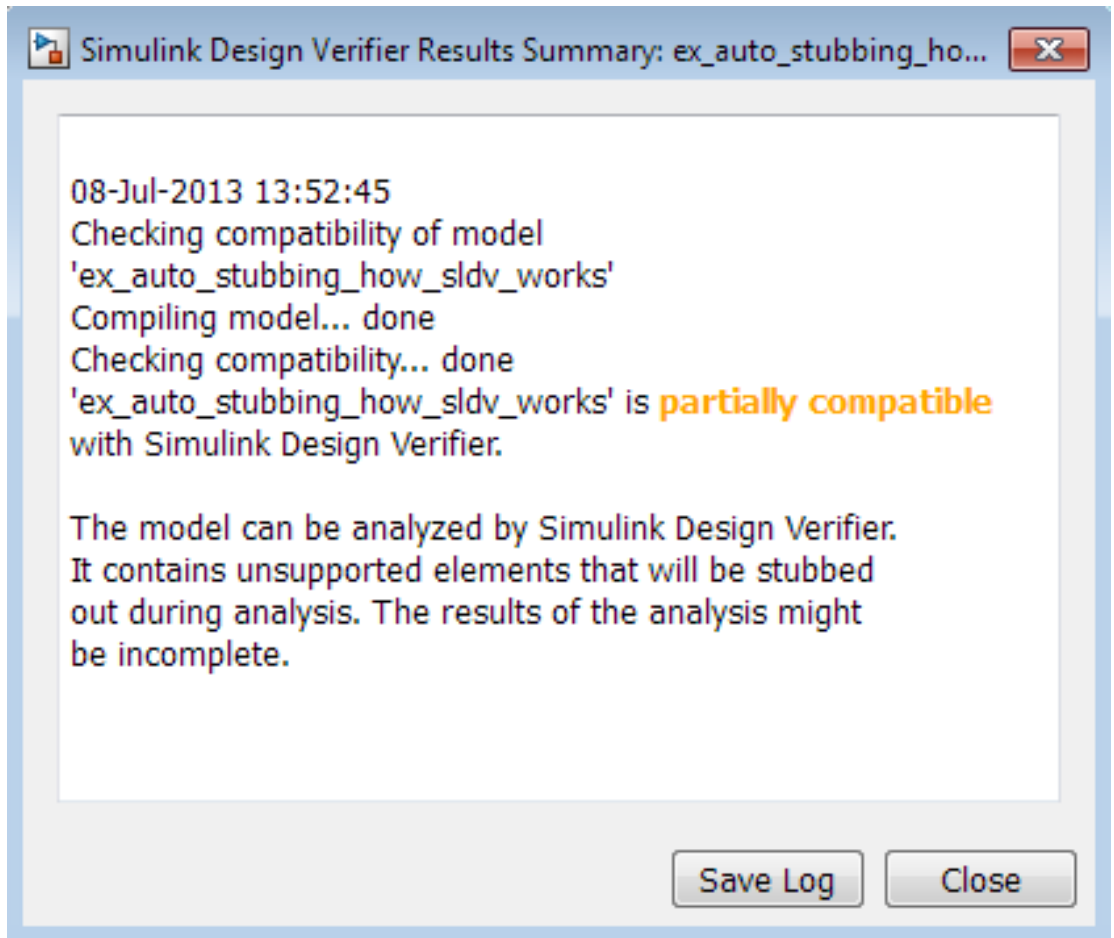
- Select the analysis that you want:

    - **Analysis** > **Design Verifier** > **Detect Design Errors** > **Model**

    - **Analysis** > **Design Verifier** > **Generate Tests** > **Model**

    - **Analysis** > **Design Verifier** > **Prove Properties** > **Model**

The software first checks the compatibility of the model. If the model itself is incompatible, for example, if it uses a variable-step solver, the analysis cannot continue.

If it finds incompatible elements in the model, the software analyzes the model and, by default, stubs out the incompatible elements. The Diagnostic Viewer also opens, listing the incompatibilities.
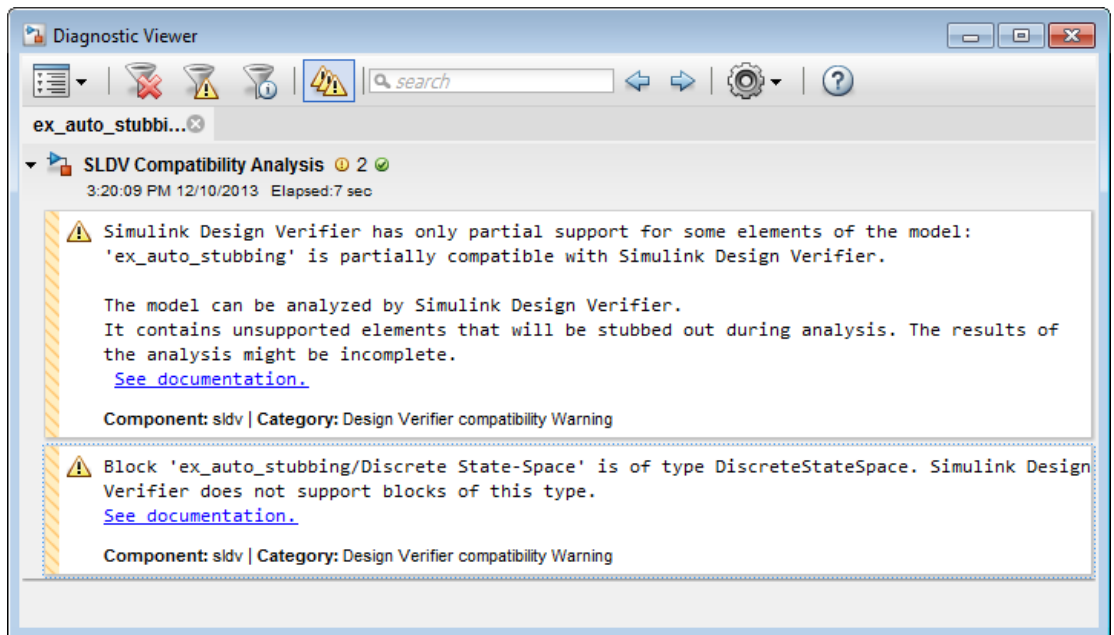
Note: For more information, see "View Diagnostics" in the Simulink documentation.

Turn On Automatic Stubbing

Automatic stubbing is enabled by default. To change the automatic stubbing setting, in the Configuration Parameters dialog box, on the main **Design Verifier** pane, select **Automatic stubbing of unsupported block and functions**. When you run the analysis, the software tells you that stubbing is turned on and the analysis continues.

Review Results

If you run an analysis with automatic stubbing enabled, make sure to review the results. In this report, generated after a test case generation analysis, you see a table of unsupported blocks that the software encountered.

Unsupported Blocks

The following blocks are not supported by Simulink Design Verifier. They were abstracted during the analysis. This can lead Simulink Design Verifier to produce only partial results for parts of the model that depends on the output values of these blocks.

Block	Type
Discrete State-Space	DiscreteStateSpace

The generated analysis report for the example model shows that the objectives are undecided because of stubbing. The software cannot generate test cases because it does not understand the operation of the Discrete State-Space block.

Objectives Undecided Due to Stubbing

Simulink Design Verifier was not able to decide these objectives due to stubbing.

#	Type	Model Item	Description	Analysis Time (sec)
2	Decision	Saturation	input > lower limit F	12
3	Decision	Saturation	input > lower limit T	12
4	Decision	Saturation	input >= upper limit F	12
5	Decision	Saturation	input >= upper limit T	12

Achieve Complete Results

If your analysis does not achieve complete results because of the stubbing, you can define custom block replacements to give a more precise definition of the unsupported blocks. For more information, see "Define Custom Block Replacements" on page 4-10 or follow the steps in "Block Replacements for Unsupported Blocks".

Nonfinite Data

The Simulink Design Verifier software does not support nonfinite data (for example, NaN and Inf) and related operations.

During an analysis, the software handles nonfinite operations as follows:

- In the Relational Operator block:

  - If the **Relational operator** parameter is isFinite, the output is always 1.
  - If the **Relational operator** parameter is isNan or isInf, the output is always 0.
- In the MATLAB Function block:

  - For the isFinite function, the output is always 1.
  - For the isNan and isInf functions, the output is always 0.

Approximations

Approximations During Model Analysis

The Simulink Design Verifier software attempts to generate inputs and parameters to achieve objectives. However, there could be an infinite number of values for the software to search. To create reasonable limits on the analysis, the software performs approximations to simplify the analysis. The software records any approximations it performed in the Analysis Information chapter of the Simulink Design Verifier HTML report. For a description of this chapter, see "Analysis Information Chapter" on page 13-28.

Review the analysis results carefully when the software uses approximations. Evaluate your model to identify which blocks or subsystems caused the software to perform the approximations.

Rarely, an approximation can result in test cases that fail to achieve test objectives or demonstrate a design error, or counterexamples that fail to falsify proof objectives. For example, suppose the software generates a test case signal that should achieve an objective by exceeding a threshold; a floating-point round-off error might prevent that signal from attaining the threshold value.

Types of Approximations

The Simulink Design Verifier software performs the following approximations when it analyzes a model:

- "Floating-Point to Rational Number Conversion" on page 2-17

- "Linearization of Two-Dimensional Lookup Tables for Floating-Point Data Types" on page 2-17
- "Approximation of One- and Two-Dimensional Lookup Tables for Integer and Fixed-Point Data Types" on page 2-18
- "While Loops" on page 2-18

Floating-Point to Rational Number Conversion

The Simulink Design Verifier software simplifies the linear arithmetic of floating-point numbers by approximating them with infinite-precision rational numbers. The software discovers how the logical relationships between these values affects the objectives. This analysis enables the software to support supervisory logic that is commonly found in embedded controls designs.

If your model contains floating-point values in the signals, input values, or block parameters, Simulink Design Verifier converts those values to rational numbers before performing its analysis. As a result of these approximations:

- Round-off error is not considered.
- Upper and lower bounds of floating-point numbers are not considered.
- If your model casts floating-point values to integer values, the integer representation can affect tests generated for the model. In some rare cases the generated tests may not satisfy objectives associated with the floating-point values.

Linearization of Two-Dimensional Lookup Tables for Floating-Point Data Types

The Simulink Design Verifier software does not support nonlinear arithmetic for floating-point data types. If your model contains any 2-D Lookup Table blocks, or n-D Lookup Table blocks where $n = 2$, with all of the following characteristics, the software approximates nonlinear two-dimensional interpolation with linear interpolation by fitting planes to each interpolation interval.

Block	Characteristics
n-D Lookup Table block, $n = 2$:	- **Interpolation method** parameter is `Linear`. - **Extrapolation method** parameter is `Clip` or `Linear`. - The input and output signals both have the floating-point data type.

Approximation of One- and Two-Dimensional Lookup Tables for Integer and Fixed-Point Data Types

If your model contains lookup tables of the following characteristics, Simulink Design Verifier automatically converts your original lookup table into a new lookup table composed of breakpoints that are evenly-spaced in each of their respective dimensions.

Block	Characteristics
n-D Lookup Table block, $n = 1$ or $n = 2$:	• **Interpolation method** parameter is `Linear`. • **Extrapolation method** parameter is `Clip`. • **Index search method** parameter is `Linear search` or `Binary search`. • The input and output signals are both of the same type and are both integer type or fixed-point type.

This approximation allows Simulink Design Verifier to generate tests significantly faster. The time saved is pronounced when you have unsatisfiable test objectives in your model.

If Simulink Design Verifier applies such approximations to your model, the Simulink Design Verifier report includes details of the approximation.

While Loops

If your model or a Stateflow chart in your model contains a `while` loop, Simulink Design Verifier tries to detect a conservative constant bound that allows the `while` loop to exit. If the software cannot find a constant bound, it performs a `while` loop approximation. With this approximation, the analysis does not prove objectives to be valid or unsatisfiable and it does not prove dead logic. The generated analysis report notes this approximation.

The behavior of the `while` loop approximation is consistent in all modes of analysis, as described in the following table.

Analysis Mode	While Loop Approximation
Design Error Detection	Sets number of `while` loop iterations to 3. Does not report dead logic or valid objectives.

Analysis Mode	While Loop Approximation
Test Case Generation	Sets number of `while` loop iterations to 3. Does not report unsatisfiable objectives.
Property Proving	Sets number of `while` loop iterations to 3. Does not report valid objectives.

Logic Operations Short-Circuiting

When Simulink Design Verifier performs an analysis, if possible, the software short-circuits logic blocks. When the previous inputs alone determine the block output, the analysis ignores any remaining block inputs. For example, if the first input to a Logical Operator block whose **Operator** parameter specifies AND is false, the analysis ignores the values of the other inputs.

Consider the following example model, with the **Model coverage objectives** parameter set to Condition Decision.
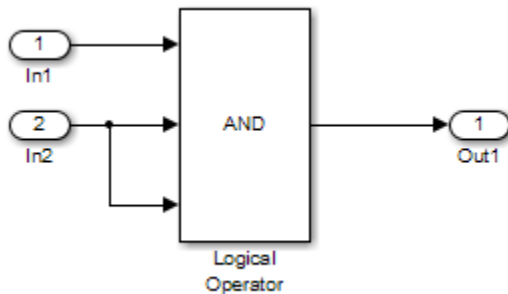
When Simulink Design Verifier analyzes this model for Condition Decision coverage, the analysis can only satisfy five of six objectives for the Logical Operator block inputs. The software cannot generate a test case when the third input to the Logical Operator block is false. If the second input is false, the third input is false, but the software ignores the third input due to the short-circuiting. If the second input is true, the third input is never false.

3

Checking Compatibility with the Simulink Design Verifier Software

Check Model Compatibility

In this section...
"Compatibility with Simulink Design Verifier" on page 3-2
"Run Compatibility Check" on page 3-2
"Compatibility Check Results" on page 3-3

Compatibility with Simulink Design Verifier

The Simulink Design Verifier software analyzes Simulink models to:

- Detect design errors that can occur at run time.
- Generate test cases that achieve model coverage.
- Prove properties and identify property violations.

For these analyses, the models must:

- Compile into an executable form.
- Be compatible with code generation.
- Perform a zero-second simulation with no errors, where the simulation start time and stop time are 0.

The software supports a broad range of Simulink and Stateflow software features in your models. However, there are features that the product does not support, described in "Support Limitations for Simulink Software Features" on page 3-21 and "Support Limitations for Stateflow Software Features" on page 3-26. Avoid using unsupported features in models that you plan to analyze with Simulink Design Verifier.

Run Compatibility Check

Before the software begins an analysis, it automatically checks the compatibility of your model.

Before you start an analysis, you can run a compatibility check on your model. To run a compatibility check on your model, do one of the following:

- From the Simulink Editor, select **Analysis > Design Verifier > Check Compatibility > Model**.

- In the Model Advisor, select either **By Product > Simulink Design Verifier > Check compatibility with Simulink Design Verifier** or **By Task > Simulink Design Verifier Compatibility Check > Check compatibility with Simulink Design Verifier**. Click **Run This Check**.

  For more information, see "Simulink Design Verifier Checks".

- Use the `sldvcompat` function to run the compatibility checker programmatically at the command line or in a MATLAB program. For more information, see the `sldvcompat` reference page.

Compatibility Check Results

There are three outcomes of a compatibility check:

- "Model Is Compatible" on page 3-3
- "Model Is Incompatible" on page 3-4
- "Model Is Partially Compatible" on page 3-7

Model Is Compatible

In the Results Summary window, you see if your model is compatible with the software.
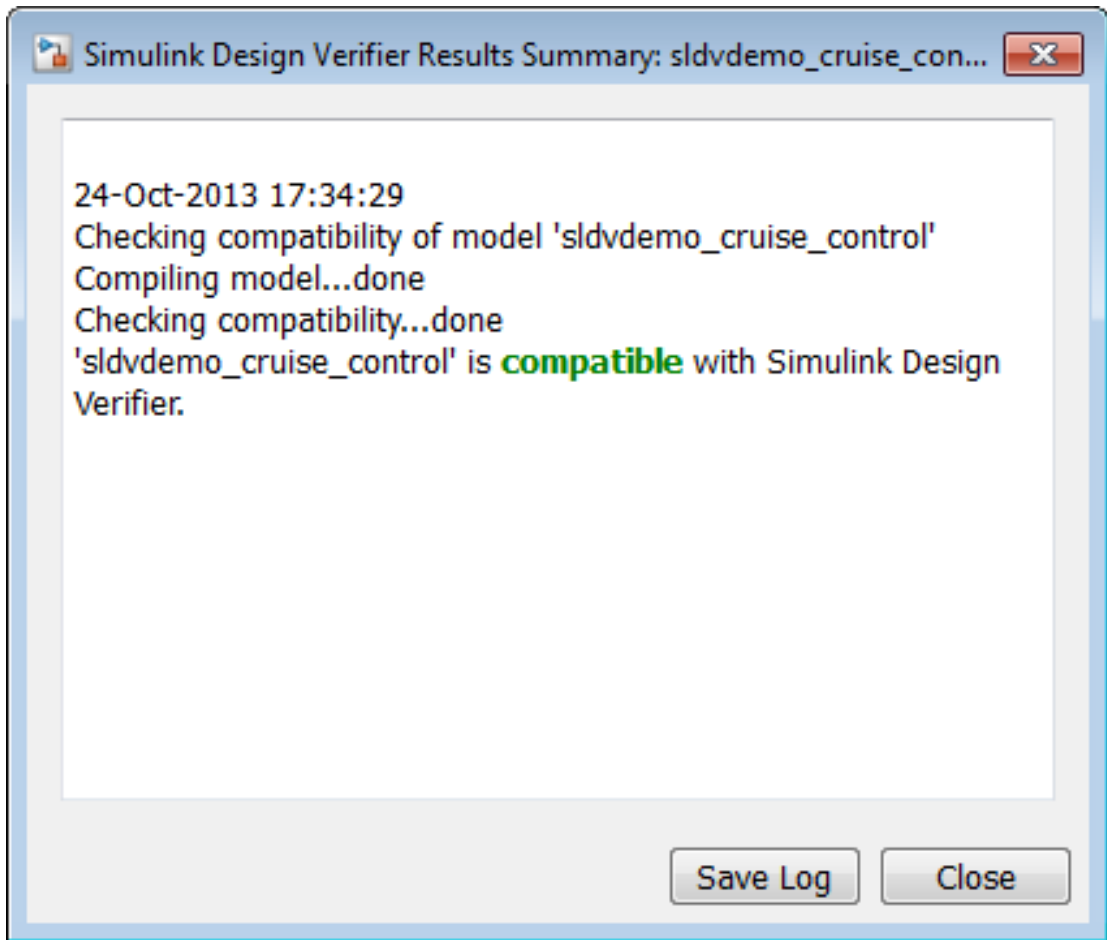
Model Is Incompatible

If the model itself is incompatible with the software, two dialog boxes open:

- Simulink Design Verifier Results Summary
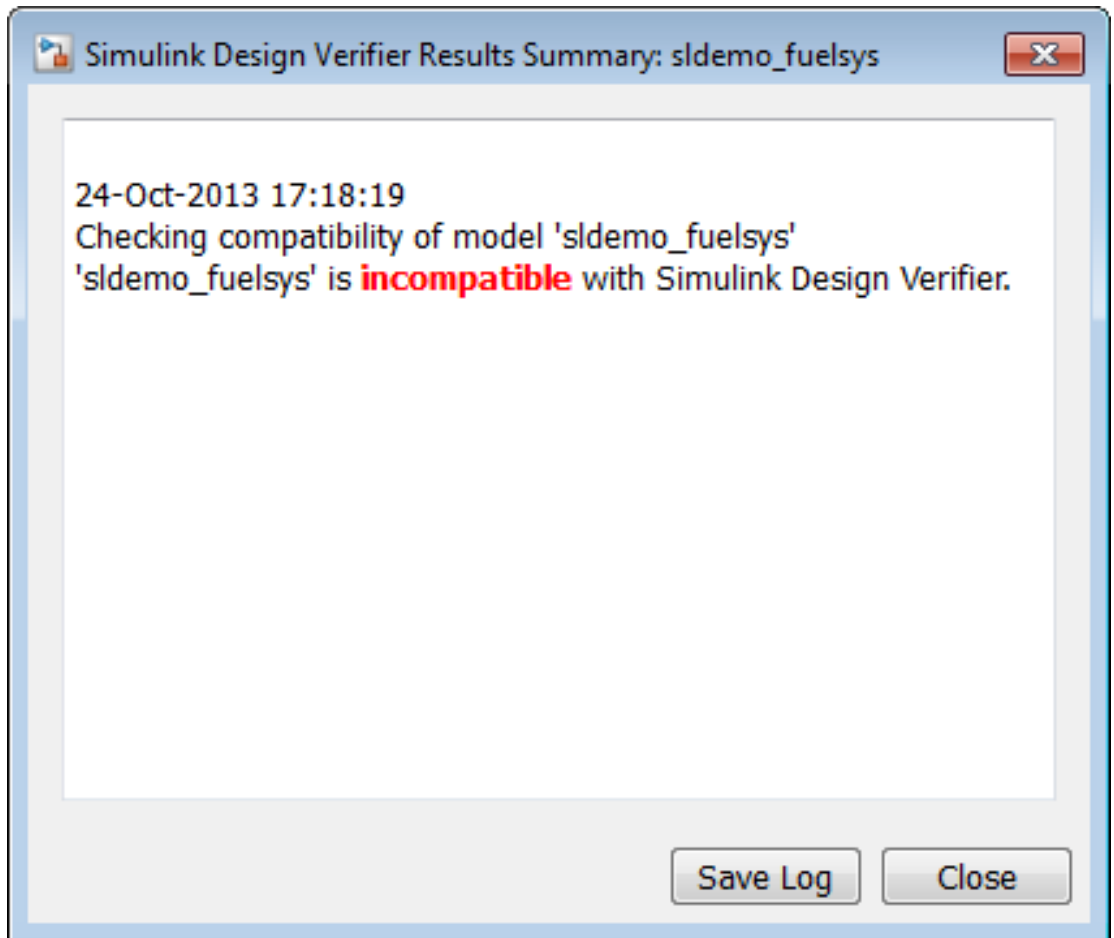
- Diagnostic Viewer. Use the information in this dialog box to identify and fix the incompatibility.

  - If your model uses a variable-step solver, configure the solver options to a fixed step.
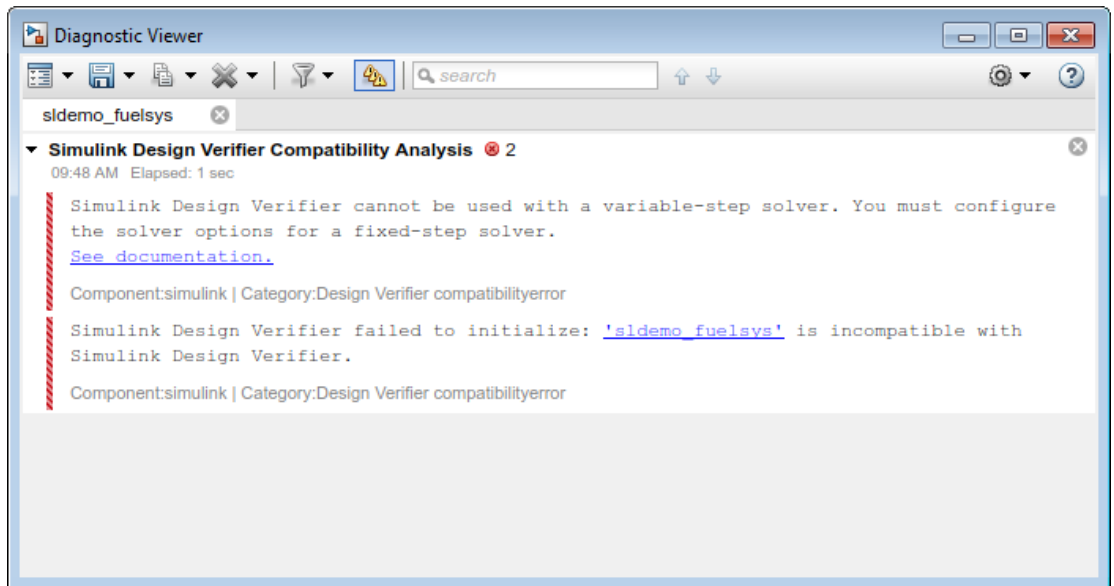
- If your model has nonfinite data, change the value of the data or configure the model so that the data is treated as a variable during Simulink Design Verifier analysis.
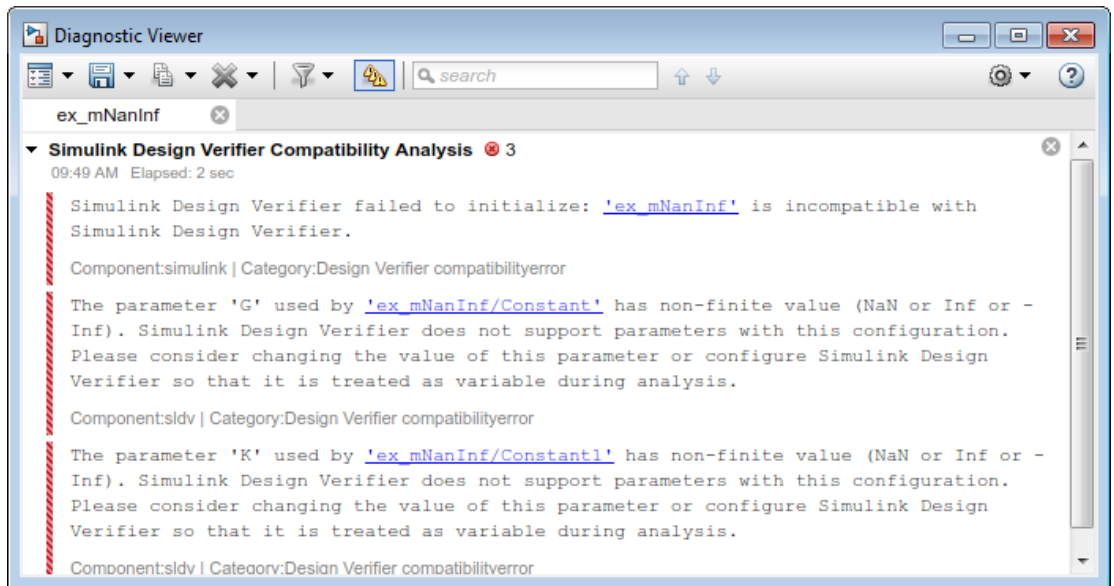
Note: For more information about the Diagnostic Viewer, see "View Diagnostics" in the Simulink documentation.

If your model is large and contains many subsystems, you can use the Test Generation Advisor to determine whether certain subsystems cause the incompatibility. For more information, see "Use Test Generation Advisor to Identify Analyzable Components" on page 7-21.

Model Is Partially Compatible

A model is partially compatible if at least one object in the model is incompatible. Automatic stubbing is enabled by default. If you start an analysis that determines that the model is partially incompatible, you see the following message, but the analysis proceeds.

```
The model can be analyzed by Simulink Design Verifier.
It contains unsupported elements that will be stubbed out
during analysis. The results of the analysis might be
incomplete.
```

If you have disabled automatic stubbing, the analysis stops. The Results Summary window appears as follows.
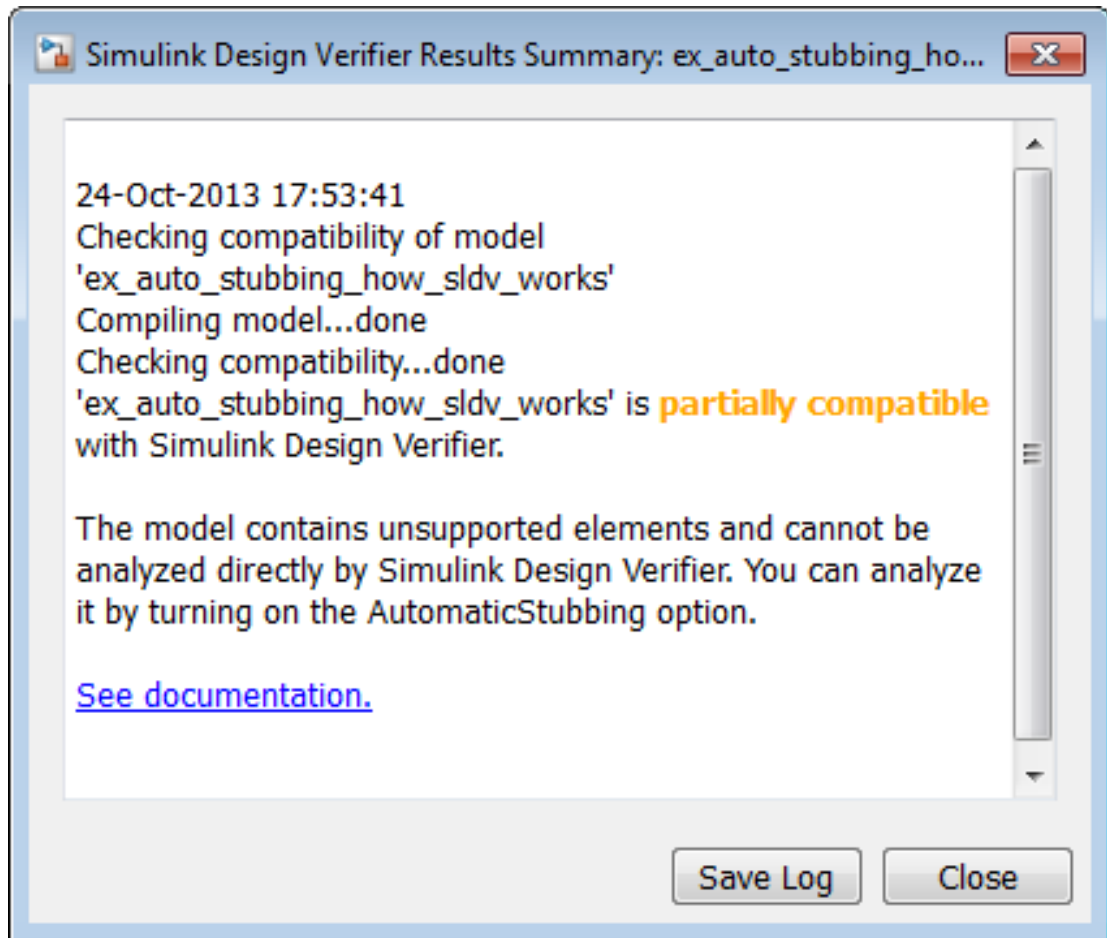
To turn on automatic stubbing:

1 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**.

2 Under **Analysis options**, select **Automatic stubbing of unsupported blocks and functions**.

For more information, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

If your model is large or complex, you can use the Test Generation Advisor to determine whether certain subsystems cause the incompatibility. For more information, see "Use Test Generation Advisor to Identify Analyzable Components" on page 7-21.

Supported and Unsupported Simulink Blocks in Simulink Design Verifier

Abstract

Lists Simulink blocks that Simulink Design Verifier does and does not support.

Simulink Design Verifier provides various levels of support for Simulink blocks:

· Fully supported

· Partially supported

· Not supported

If your model contains unsupported blocks, you can enable automatic stubbing. Automatic stubbing considers the interface of the unsupported blocks, but not their behavior. If any of the unsupported blocks affect the simulation outcome, however, the analysis may achieve only partial results. For details about automatic stubbing, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

To achieve 100% coverage, avoid using unsupported blocks in models that you analyze. Similarly, for partially supported blocks, specify only the block parameters that Simulink Design Verifier recognizes.

The following tables summarize Simulink Design Verifier analysis support for Simulink blocks. Each table lists the blocks in a Simulink library and describes support information for that particular block.

Additional Math and Discrete Library

The software supports all blocks in the Additional Math and Discrete library.

Commonly Used Blocks Library

The Commonly Used Blocks library includes blocks from other libraries. Those blocks are listed under their respective libraries.

Continuous Library

Block	Support Notes
`Derivative`	Not supported
`Integrator`	Not supported and not stubbable

Block	Support Notes
Integrator Limited	Not supported and not stubbable
PID Controller	Not supported
PID Controller (2 DOF)	Not supported
Second Order Integrator	Not supported and not stubbable
Second Order Integrator Limited	Not supported and not stubbable
State-Space	Not supported and not stubbable
Transfer Fcn	Not supported and not stubbable
Transport Delay	Not supported
Variable Time Delay	Not supported
Variable Transport Delay	Not supported
Zero-Pole	Not supported and not stubbable

Discontinuities Library

The software supports all blocks in the Discontinuities library.

Discrete Library

Block	Support Notes
Delay	Supported
Difference	Supported
Discrete Derivative	Supported
Discrete Filter	Supported
Discrete FIR Filter	Supported
Discrete PID Controller	Supported
Discrete PID Controller (2 DOF)	Supported
Discrete State-Space	Not supported
Discrete Transfer Fcn	Supported
Discrete Zero-Pole	Not supported
Discrete-Time Integrator	Supported
First-Order Hold	Supported

Block	Support Notes
Memory	Supported
Tapped Delay	Supported
Transfer Fcn First Order	Supported
Transfer Fcn Lead or Lag	Supported
Transfer Fcn Real Zero	Supported
Unit Delay	Supported
Zero-Order Hold	Supported

Logic and Bit Operations Library

The software supports all blocks in the Logic and Bit Operations library.

Lookup Tables Library

Block	Support Notes
Cosine	Supported
Direct Lookup Table (n-D)	Supported
Interpolation Using Prelookup	Not supported when: • The **Interpolation method** parameter is Linear and the **Number of table dimensions** parameter is greater than 4. or • The **Interpolation method** parameter is Linear and the **Number of sub-table selection dimensions** parameter is not 0.
1-D Lookup Table	Not supported when the **Interpolation method** or the **Extrapolation method** parameter is Cubic Spline.
2-D Lookup Table	Not supported when the **Interpolation method** or the **Extrapolation method** parameter is Cubic Spline.
n-D Lookup Table	Not supported when: • The **Interpolation method** or the **Extrapolation method** parameter is Cubic Spline.

Block	Support Notes
	or • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than 5.
Lookup Table Dynamic	Supported
Prelookup	Supported
Sine	Supported

Math Operations Library

Block	Support Notes
Abs	Supported
Add	Supported
Algebraic Constraint	Supported
Assignment	Supported
Bias	Supported
Complex to Magnitude-Angle	Not supported
Complex to Real-Imag	Supported
Divide	Supported
Dot Product	Supported
Find Nonzero Elements	Supported
Gain	Supported
Magnitude-Angle to Complex	Supported
Math Function	All signal types support the following **Function** parameter settings. | conj	hermitian	magnitude^2	mod
---	---	---	---
rem	reciprocal	square	transpose	 The software does not support the following **Function** parameter settings.

3-13

Block	Support Notes		
	`10^u`	`exp`	`hypot`
	`log`	`log10`	`pow`
Matrix Concatenate	Supported		
`MinMax`	Supported		
`MinMax Running Resettable`	Supported		
`Permute Dimensions`	Supported		
`Polynomial`	Supported		
`Product`	Supported		
`Product of Elements`	Supported		
`Real-Imag to Complex`	Supported		
`Reciprocal Sqrt`	Not supported		
`Reshape`	Supported		
`Rounding Function`	Supported		
`Sign`	Supported		
`Signed Sqrt`	Not supported		
`Sine Wave Function`	Not supported		
`Slider Gain`	Supported		
`Sqrt`	Supported		
`Squeeze`	Supported		
`Subtract`	Supported		
`Sum`	Supported		
`Sum of Elements`	Supported		
`Trigonometric Function`	Supported if **Function** is `sin`, `cos`, or `sincos`, and **Approximation method** is `CORDIC`.		
`Unary Minus`	Supported		
Vector Concatenate	Supported		
`Weighted Sample Time Math`	Supported		

Model Verification Library

The software supports all blocks in the Model Verification library.

Model-Wide Utilities Library

Block	Support Notes
Block Support Table	Supported
DocBlock	Supported
Model Info	Supported
Timed-Based Linearization	Not supported
Trigger-Based Linearization	Not supported

Ports & Subsystems Library

Block	Support Notes
Atomic Subsystem	Supported
Code Reuse Subsystem	Supported
Configurable Subsystem	Supported
Enable	Supported
Enabled Subsystem	Design range checks do not consider specified minimum and maximum values for blocks connected to the outport of the subsystem. For more information on design range checks, see "Check for Specified Intermediate Minimum and Maximum Signal Values" on page 6-30. Simulink Design Verifier treats Enabled Subsystems as short-circuited during test generation.
Enabled and Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type. Design range checks do not consider specified minimum and maximum values for blocks connected to the outport of the subsystem. For more information on design range checks, see "Check for Specified Intermediate Minimum and Maximum Signal Values" on page 6-30.

Block	Support Notes
	Simulink Design Verifier treats Enabled and Triggered Subsystems as short-circuited during test generation.
For Each	Supported with the following limitations: • When For Each Subsystem contains one or more Simulink Design Verifier `Test Condition`, `Test Objective`, `Proof Assumption`, or `Proof Objective` blocks, not supported. • When the mask parameters of the For Each Subsystem are partitioned, not supported.
For Each Subsystem	Supported with the following limitations: • When For Each Subsystem contains one or more Simulink Design Verifier `Test Condition`, `Test Objective`, `Proof Assumption`, or `Proof Objective` blocks, not supported. • When the mask parameters of the For Each Subsystem are partitioned, not supported.
For Iterator Subsystem	Supported
Function-Call Feedback Latch	Supported
Function-Call Generator	Supported
Function-Call Split	Supported
Function-Call Subsystem	Design range checks do not consider specified minimum and maximum values for blocks connected to the outport of the subsystem. For more information on design range checks, see "Check for Specified Intermediate Minimum and Maximum Signal Values" on page 6-30.
If	Parameter configurations are not supported. The analysis ignores parameter configurations that you specify for an If block.
If Action Subsystem	Supported
Inport	Supported
Model	Supported except for the limitations described in "Support Limitations for Model Blocks" on page 3-24.

Block	Support Notes
Model Variants	Supported except for the limitations described in "Support Limitations for Model Blocks" on page 3-24.
Outport	Supported
Resettable Subsystem	Supported
Subsystem	Supported
Switch Case	Supported
Switch Case Action Subsystem	Supported
Trigger	Supported
Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type. Design range checks do not consider specified minimum and maximum values for blocks connected to the outport of the subsystem. For more information on design range checks, see "Check for Specified Intermediate Minimum and Maximum Signal Values" on page 6-30. Simulink Design Verifier treats Enabled Subsystems as short-circuited during test generation.
Variant Subsystem	Not supported when the **Generate preprocessor conditionals** parameter is enabled. Only the active variant is analyzed.
While Iterator Subsystem	Supported

Signal Attributes Library

The software supports all blocks in the Signal Attributes library.

Signal Routing Library

Block	Support Notes
Bus Assignment	Supported
Bus Creator	Supported
Bus Selector	Supported

Block	Support Notes
Data Store Memory	Supported
Data Store Read	Supported
Data Store Write	Supported
Demux	Supported
Environment Controller	Supported
From	Supported
Goto	Supported
Goto Tag Visibility	Supported
Index Vector	Supported
Manual Switch	The Manual Switch block is compatible with the software, but the analysis ignores this block in a model. The analysis does not flag the coverage objectives for this block as satisfiable or unsatisfiable. Model coverage data is collected for the Manual Switch block.
Merge	Supported
Multiport Switch	Supported
Mux	Supported
Selector	Supported
Switch	Supported
Vector Concatenate	Supported

Sinks Library

Block	Support Notes
Display	Supported
Floating Scope	Supported
Outport (Out1)	Supported
Scope	Supported
Stop Simulation	Not supported and not stubbable

Block	Support Notes
`Terminator`	Supported
`To File`	Supported
`To Workspace`	Supported
`XY Graph`	Supported

Sources Library

Block	Support Notes
`Band-Limited White Noise`	Not supported
`Chirp Signal`	Not supported
`Clock`	Supported
`Constant`	Supported unless **Constant value** is `inf`.
`Counter Free-Running`	Supported
`Counter Limited`	Supported
`Digital Clock`	Supported
`Enumerated Constant`	Supported
`From File`	Not supported. When MAT-file data is stored in MATLAB `timeseries` format, not stubbable.
`From Workspace`	Not supported
`Ground`	Supported
`Inport` (In1)	Supported
`Pulse Generator`	Supported
`Ramp`	Supported
`Random Number`	Not supported and not stubbable
`Repeating Sequence`	Not supported
`Repeating Sequence Interpolated`	Not supported
`Repeating Sequence Stair`	Supported
`Signal Builder`	Not supported

Block	Support Notes
`Signal Generator`	Not supported
`Sine Wave`	Not supported
`Step`	Supported
`Uniform Random Number`	Not supported and not stubbable

User-Defined Functions Library

Block	Support Notes
`Fcn`	Supports all operators except `^`, and supports only the mathematical functions `abs`, `ceil`, `fabs`, `floor`, `rem`, and `sgn`. Parameter configurations are not supported. The analysis ignores parameter configurations that you specify for these blocks. Test generation is not supported for relational boundary coverage.
`Interpreted MATLAB Function`	Not supported
`Level-2 MATLAB S-Function`	For limitations, see "Support Limitations for S-Functions" on page 3-35.
`MATLAB Function`	For limitations, see "Support Limitations for MATLAB for Code Generation" on page 3-30.
`S-Function`	For limitations, see "Support Limitations for S-Functions" on page 3-35.
`S-Function Builder`	For limitations, see "Support Limitations for S-Functions" on page 3-35.
`Simulink Function`	Simulink Function blocks with output arguments that are of bus data-type are not supported. Calls to Simulink Functions across model boundaries are not supported.

Support Limitations for Simulink Software Features

Simulink Design Verifier does not support the following Simulink software features. Avoid using these unsupported features.

Not Supported	Description
Variable-step solvers	The software supports only fixed-step solvers. For more information, see "Choose a Fixed-Step Solver" in the Simulink documentation.
Callback functions	The software does not execute model callback functions during the analysis. The results that the analysis generates, such as the harness model, may behave inconsistently with the expected behavior. • If a model or any referenced model calls a callback function that changes any block parameters, model parameters, or workspace variables, the analysis does not reflect those changes. • Changing the storage class of base workspace variables on model callback functions or mask initializations is not supported. • Callback functions called prior to analysis, such as the `PreLoadFcn` or `PostLoadFcn` model callbacks, are fully supported.
Model callback functions	The software only supports model callback functions if the `InitFcn` callback of the model is empty.
Algebraic loops	The software does not support models that contain algebraic loops. For more information, see "Algebraic Loops" in the Simulink documentation.
Masked subsystem initialization functions	The software does not support models whose masked subsystem initialization modifies any attribute of any workspace parameter.
Complex signals	The software supports only real signals.

Not Supported	Description
	For more information, see "Complex Signals" in the Simulink documentation.
Variable-size signals	The software does not support variable-size signals. A variable-size signal is a signal whose size (number of elements in a dimension), in addition to its values, can change during model execution. For more information, see "Variable-Size Signal Basics" in the Simulink documentation.
Multiword fixed-point data types	The software does not support multiword fixed-point data types.
Nonzero start times	Although Simulink allows you to specify a nonzero simulation start time, the analysis generates signal data that begins only at zero. If your model specifies a nonzero start time: • If you do not select the **Reference input model in generated harness** parameter (the default), the harness model is a subsystem. The analysis sets the start time of the harness model to 1 and continues the analysis. • If you select the **Reference input model in generated harness** parameter, a Model block references the harness model. The software cannot change the start time of the harness model, so the analysis stops and you see a recommendation to set the **Start time** parameter to 0.

Not Supported	Description
Nonfinite data	The software does not support nonfinite data (for example, NaN and Inf) and related operations. In the Relational Operator block, the software assigns the output as follows: • If the **Relational operator** parameter is isFinite, the output is always 1. • If the **Relational operator** parameter is isNan or isInf, the output is always 0. In the MATLAB Function block, the software assigns the return value as follows: • For the isFinite function, the output is always 1. • For the isNan and isInf functions, the output is always 0.
Concurrent execution	The software does not support models that are configured for concurrent execution.
Signals with nonzero sample time offset	The software does not support models with signals that have nonzero sample time offsets.
Models with no output ports	The software only supports models that have one or more output ports.
Large floating-point constants outside the range [-realmax/2, realmax/2]	The use of large floating-point constants can cause out of memory errors or substantial loss of precision. Avoid using such constants if possible.
Symbolic Dimensions	The software does not support symbolic dimensions for test generation, property proving, or design error detection.

Support Limitations for Model Blocks

Abstract

Simulink Design Verifier supports the Model block with the some limitations.

Simulink Design Verifier supports the Model block with the following limitations. The software cannot analyze a model containing one or more Model blocks if:

- The referenced model is protected. Protected referenced models are encoded to obscure their contents. This allows third parties to use the referenced model without being able to view the intellectual property that makes up the model.

  For more information, see "Protected Model".

- The parent model or any of the referenced models returns an error when you set the **Configuration Parameters** > **Diagnostics** > **Connectivity** > **Element name mismatch** parameter to `error`.

  You can use the **Element name mismatch** diagnostic along with bus objects so that your model meets the bus element naming requirements imposed by some blocks.

- The Model block uses asynchronous function-call inputs.

- Any of the Model blocks in the model reference hierarchy creates an artificial algebraic loop. If this occurs, take the following steps:

  1 On the **Diagnostics** pane of the Configuration Parameters dialog box, set the **Minimize algebraic loop** parameter to `error` so that Simulink reports an algebraic loop error.

  2 On the **Model Referencing** Pane of the Configuration Parameters dialog box, select the Minimize algebraic loop occurrences parameter.

  Simulink tries to eliminate the artificial algebraic loop during simulation.

  3 Simulate the model.

  4 If Simulink cannot eliminate the artificial algebraic loop, highlight the location of the algebraic loop by selecting **Simulation** > **Update Diagram**.

  5 Eliminate the artificial algebraic loop so that the software can analyze the model. Break the loop with Unit Delay blocks so that the execution order is predictable.

Note: For more information, see "Algebraic Loops".

- The parent model uses the base workspace and the referenced model uses a data dictionary.

- The parent model and the referenced model have mismatched data type override settings. The data type override setting of the parent model and its referenced models must be the same, unless the data type override setting of the parent model is `Use local settings`. You can select the data type override settings for your model in the **Analysis** menu, in the Fixed Point Tool dialog box under the **Settings for selected system** pane.

- The referenced model is a Model Reference block with virtual bus inports, and the signals in the bus do not all have the same sample time at compilation. To make the model compatible with Simulink Design Verifier analysis, convert the port to a nonvirtual bus, or specify an explicit sample time for the port.

Support Limitations for Stateflow Software Features

shSimulink Design Verifier does not support the following Stateflow software features. Avoid using these unsupported features in models that you analyze.

In this section...
"ml Namespace Operator, ml Function, ml Expressions" on page 3-26
"C Math Functions" on page 3-26
"Atomic Subcharts That Call Exported Graphical Functions Outside a Subchart" on page 3-27
"Atomic Subchart Input and Output Mapping" on page 3-27
"Recursion and Cyclic Behavior" on page 3-27
"Custom C or C++ Code" on page 3-29
"Machine-Parented Data" on page 3-29
"Textual Functions with Literal String Arguments" on page 3-29

ml Namespace Operator, ml Function, ml Expressions

The software does not support calls to MATLAB functions or access to MATLAB workspace variables, which the Stateflow software allows. (See "Access Built-In MATLAB Functions and Workspace Data" in the Stateflow documentation.)

C Math Functions

The software supports calls to the following C math functions:

- abs
- ceil
- fabs
- floor
- fmod
- labs
- ldexp
- pow (only for integer exponents)

The software does not support calls to other C math functions, which the Stateflow software allows. If automatic stubbing is enabled, which it is by default, the software eliminates these unsupported functions during the analysis.

For information about C math functions in Stateflow, see "Call C Functions in C Charts" in the Stateflow documentation.

Note: For details about automatic stubbing, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

Atomic Subcharts That Call Exported Graphical Functions Outside a Subchart

The software does not support atomic subcharts that call exported graphical functions, which the Stateflow software allows.

Note: For information about exported functions, see "Export Stateflow Functions for Reuse" in the Stateflow documentation.

Atomic Subchart Input and Output Mapping

If an input or output in an atomic subchart maps to chart-level data of a different scope, the software does not support the chart that contains that atomic subchart.

For an atomic subchart input, this incompatibility applies when the input maps to chart-level data of output, local, or parameter scope. For an atomic subchart output, this incompatibility applies when the output maps to chart-level data of local scope.

Recursion and Cyclic Behavior

The software does not support recursive functions, which occur when a function calls itself directly or indirectly through another function call. Stateflow software allows you to implement recursion using graphical functions.

In addition, the software does not support recursion that the Stateflow software allows you to implement using a combination of event broadcasts and function calls.

> **Note:** For information about avoiding recursion in Stateflow charts, see "Guidelines for Avoiding Unwanted Recursion in a Chart" in the Stateflow documentation.

Stateflow software also allows you to create *cyclic behavior*, where a sequence of steps is repeated indefinitely. If your model has a chart with cyclic behavior, the software cannot analyze it.

> **Note:** For information about cyclic behavior in Stateflow charts, see "Cyclic Behavior in a Chart" in the Stateflow documentation.

However, you can modify a chart with cyclic behavior so that it is compatible, as in the following example.

The following chart creates cyclic behavior. State A calls state A1, which broadcasts a `Clear` event to state B, which calls state B2, which broadcasts a `Set` event back to state A, causing the cyclic behavior.
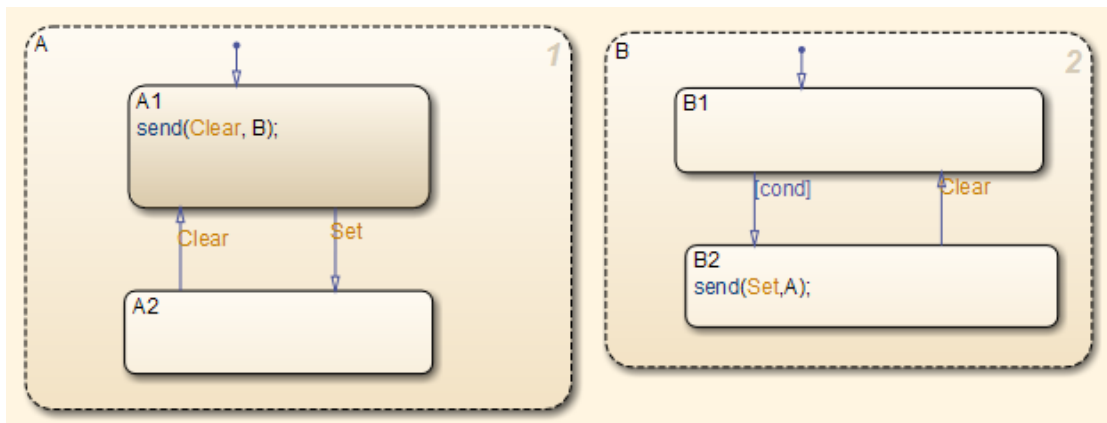
If you change the `send` function calls to use directed event broadcasts so that the Set and Clear events are broadcast directly to the states B1 and A1, respectively, the cyclic behavior disappears and the software can analyze the model.
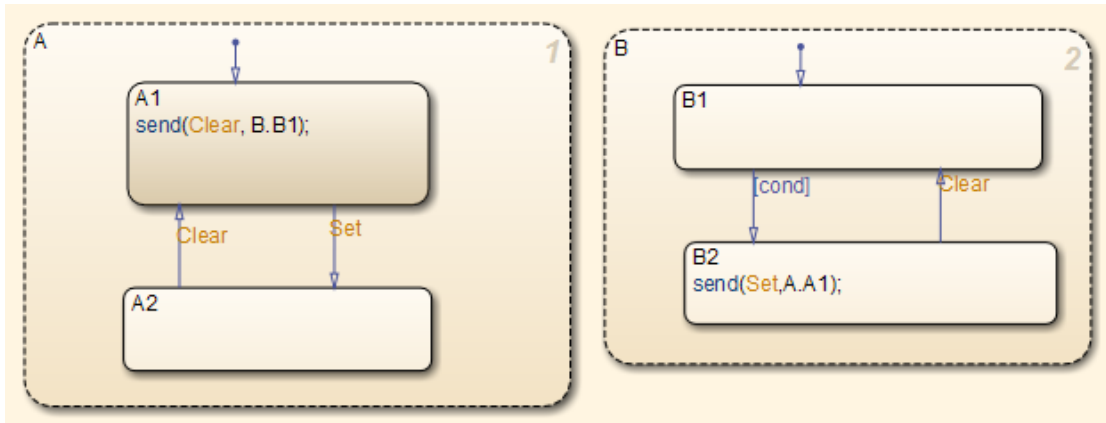
Note: For information about the benefits of directed event broadcasts, see "Broadcast Events to Synchronize States" in the Stateflow documentation.

Custom C or C++ Code

The software does not support custom C or C++ code, which the Stateflow software allows.

Machine-Parented Data

The software does not support machine-parented data (i.e., defined at the level of the Stateflow machine), which the Stateflow software allows.

For more information, see "Best Practices for Using Data in Charts" in the Stateflow documentation.

Textual Functions with Literal String Arguments

The software does not support literal string arguments to textual functions in a Stateflow chart.

Support Limitations for MATLAB for Code Generation

In this section...

"Unsupported MATLAB for Code Generation Features" on page 3-30

"Support Limitations for MATLAB for Code Generation Library Functions" on page 3-30

Unsupported MATLAB for Code Generation Features

Simulink Design Verifier does not support the following features of the MATLAB Function block in the Simulink software and MATLAB functions in the Stateflow software. Avoid using these unsupported features in models that you analyze with Simulink Design Verifier.

Not Supported	Description
Complex numbers	The software supports only real numbers and cannot analyze MATLAB for code generation functions that use complex numbers.
Characters	The software does not support characters, which MATLAB for code generation allows.
C functions	The software does not support calls to external C functions, which MATLAB for code generation allows.
Extrinsic functions	The software supports extrinsic functions only when they do not affect the output of a MATLAB function.
Handle classes	The software does not support handle classes in the MATLAB Function block. The software does support value classes.

Support Limitations for MATLAB for Code Generation Library Functions

Simulink Design Verifier provides various levels of support for MATLAB for code generation library functions. The software either fully or partially supports particular functions. It does not support other functions.

If your model contains unsupported functions, you can turn on automatic stubbing, which considers the interface of the unsupported functions, but not their behavior. However, if any of the unsupported functions affect the simulation outcome, the analysis may achieve

only partial results. For details about automatic stubbing, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

To achieve 100% coverage, avoid using unsupported MATLAB library functions in models that you analyze.

The following table lists Simulink Design Verifier support for categories of library functions in code generation from MATLAB:

- Software supports functions in that category, indicated by a dash (—).
- Software does not support functions in that category.
- Software supports the function in that category with limitations as specified.

For the complete listing of available functions, see "Functions and Objects Supported for C/C++ Code Generation — Alphabetical List".

Function Category	Support Notes		
Aerospace Toolbox functions	Not supported.		
Arithmetic operator functions	Supported with the following limitations:		
	`mldivide` (\)	Supports only scalar arguments.	
	`mpower` (^)	Supports only integer exponents.	
	`mrdivide` (/)	Supports only scalar arguments.	
	`power` (.^)	Supports only integer exponents.	
Bit-wise operation functions	—		
Casting functions	Supported with the following limitations:		
	`char`	Not supported.	
	`typecast`	Not supported.	
Communications System Toolbox™ functions	Not supported.		
Complex number functions	Not supported.		
Computer Vision System Toolbox™ functions	Not supported.		
Data type functions	—		
Derivative and Integral functions	Not supported.		
Discrete math functions	—		
Error handling functions	Supported with the following limitations:		

Function Category	Support Notes	
	`assert`	Supported, but does not behave like a `Proof Objective` block.
Exponential functions	Supported with the following limitations:	
	`exp`	Not supported.
	`expm`	Not supported.
	`expm1`	Not supported.
	`log`	Not supported.
	`log2`	Not supported.
	`log10`	Not supported.
	`log1p`	Not supported.
	`nextpow2`	Not supported.
	`nthroot`	Not supported.
	`reallog`	Not supported.
	`realpow`	Not supported.
	`realsqrt`	Not supported.
	`sqrt`	Not supported.
Filtering and convolution functions	Supported with the following limitations:	
	`detrend`	Not supported.
Fixed-Point Designer functions	Supported with the following limitations:	
	`complex`	Not supported.
Histogram functions	Not supported.	
Image Processing Toolbox™ functions	Not supported.	
Input and output functions	—	
Interpolation and computation geometry	Supported with the following limitations:	
	`cart2pol`	Not supported.
	`cart2sph`	Not supported.
	`pol2cart`	Not supported.
	`sph2cart`	Not supported.

Function Category	Support Notes	
Linear algebra	Not supported.	
Logical operator functions	—	
MATLAB Compiler™ functions	Not supported.	
Matrix and array functions	Supported with the following limitations:	
	`angle`	Not supported.
	`cond`	Not supported.
	`det`	Not supported.
	`eig`	Not supported.
	`inv`	Not supported.
	`invhilb`	Not supported.
	`logspace`	Not supported.
	`lu`	Not supported.
	`norm`	Supported only when invoked using the syntax `norm(A,p)` where `p` is either `1` or `inf`.
	`normest`	Not supported.
	`pinv`	Not supported.
	`planerot`	Not supported.
	`qr`	Not supported.
	`rank`	Not supported.
	`rcond`	Not supported.
	`subspace`	Not supported.
Nonlinear numerical methods	Not supported.	
Polynomial functions	Not supported.	
Relational operations functions	—	
Rounding and remainder functions	—	

Function Category	Support Notes	
Set functions	—	
Signal Processing functions in MATLAB	Not supported.	
Signal Processing Toolbox™ functions	Not supported.	
Special values	Supported with the following limitations:	
	`rand`	Not supported.
	`randn`	Not supported.
Specialized math	Not supported.	
Statistical functions	—	
String functions	Supported with the following limitations:	
	`char`	Not supported.
	`ischar`	Not supported.
Trigonometric functions	Not supported.	

Support Limitations for S-Functions

In this section...
"Enabling S-Functions in Simulink Design Verifier" on page 3-35
"Support Limitations for S-Function Code" on page 3-35
"Considerations for Enabling S-Functions in Simulink Design Verifier" on page 3-36
"Source code protection" on page 3-36

Enabling S-Functions in Simulink Design Verifier

Simulink Design Verifier supports model verification, test case generation, and property proving for S-Functions that:

- The Legacy Code Tool generates, with `def.Options.supportCoverageAndDesignVerifier` set to true.

- The S-Function Builder generates, with **Enable support for Design Verifier** selected on the **Build Info** tab of the S-Function Builder dialog box.

- The function `slcovmex` compiles, with the option `-sldv` passed to the function when compiling the S-function.

For more information on the three approaches, see "Creating C MEX S-Functions"

Support Limitations for S-Function Code

- Simulink Design Verifier does not support S-Functions containing these types of code:

  - S-Functions containing continuous states. Simulink Design Verifier does not analyze such S-Functions.

  - S-Functions containing zero-crossing functions. Simulink Design Verifier ignores such S-Functions during analysis.

  - S-Functions containing constants that describe INF or NaN objects. Simulink Design Verifier considers such S-Functions as containing floating-point overflow errors. Although Simulink Design Verifier analysis cannot determine the type of overflow error for such cases, the analysis can determine which lines of code introduce the incompatibility. Polyspace can provide more information on why the code in your S-Functions contains floating-point overflow errors.

- You must specify that the signal elements entering the ports of S-Functions compiled with `slcovmex` must be contiguous. Use the SimStruct function ssSetInputPortRequiredContiguous.

Considerations for Enabling S-Functions in Simulink Design Verifier

- When analyzing models with enabled S-Functions, Design Verifier assumes that the S-Functions contain no run-time errors. In the case where an enabled S-function contains run-time errors (division by zero, access to noninitialized variables, array out of bounds, and so on), the analysis can produce incorrect results. S-Functions that Polyspace proves to be free of run-time errors provide correct results in Design Verifier analysis.

- If Design Verifier cannot determine the size of arrays (for instance for arrays that are dynamically allocated with nonconstant size), Design Verifier assumes an upper bound for the array. Ensure that the given upper bound is appropriate.

- If you do not enable Design Verifier support for an S-function, Design Verifier stubs the S-function. With S-function support enabled, Design Verifier analyzed the content of the S-function to get more detailed information. In some cases, Design Verifier internally stubs the S-function. Internal stubs can be the result of different C/C++ constructs, such as:

  - Calls to library functions (the library function will be replaced by a stub).
  - Complex pointer operations.
  - Casts to or from incompatible or unknown pointer types.

  Models containing such constructs are labeled **Partially compatible**.

Source code protection

To analyze the contents of an S-function, information about the implementation of the S-function, including information derived from the source code, are stored within the shared object. Although this information is not directly accessible to users, consider disabling Design Verifier support for S-Functions in models that are released externally if the S-Functions contain sensitive source code.

See Also
"Configuring S-Function for Test Case Generation"

4

Working with Block Replacements

What Is Block Replacement?

Using Simulink Design Verifier, you can define rules to replace blocks automatically in your model. For example, you can work around a block that is incompatible with the software by creating a rule that replaces an unsupported Simulink block in your model with a supported block that is functionally equivalent. Or, you can customize blocks for analysis by creating a rule that adds constraints or objectives to particular blocks in your model.

When performing block replacements, the software makes a copy of your model and replaces blocks in the copy, without altering your original model. In this way, you can easily customize a model for analysis.

The Simulink Design Verifier software replaces blocks automatically in a model using:

- Libraries of replacement blocks
- Rules that define which blocks to replace and under what conditions

You replace any block with any built-in block, library block, or subsystem.

Block replacements are extensible, allowing you to define your own libraries of replacement blocks and custom block replacement rules. Using block replacements, you can

- Work around an incompatibility, such as the presence of unsupported blocks in your model.
- Customize a block for analysis, such as:

  - Adding constraints to its input signals
  - Adding objectives to its output signals
  - Eliminating the contents of a subsystem or Model block to simplify your analysis

---

Note: You can use automatic stubbing as an alternative to block replacements to resolve incompatibilities. Automatic stubbing replaces unsupported blocks with elements that have the same interface. For more information, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

Block Replacement Effects on Test Generation

Replacing blocks can affect test case generation if the replaced blocks share functionality with other parts of your model. Before you replace blocks, understand functional dependencies on those blocks or on shared signals. See "Highlight Functional Dependencies" on page 16-2. Replacement blocks can also affect other analysis workflows such as property proving.

For example, you can customize a block for analysis using a replacement block that adds objectives to an input signal. If another subsystem depends on that signal, the replacement block effectively adds an objective for the subsystem.

In this example, the breakpoint range of u1 in the 2-D Lookup Table is 5–7. The switch threshold 8 falls outside the u1 lookup table range.
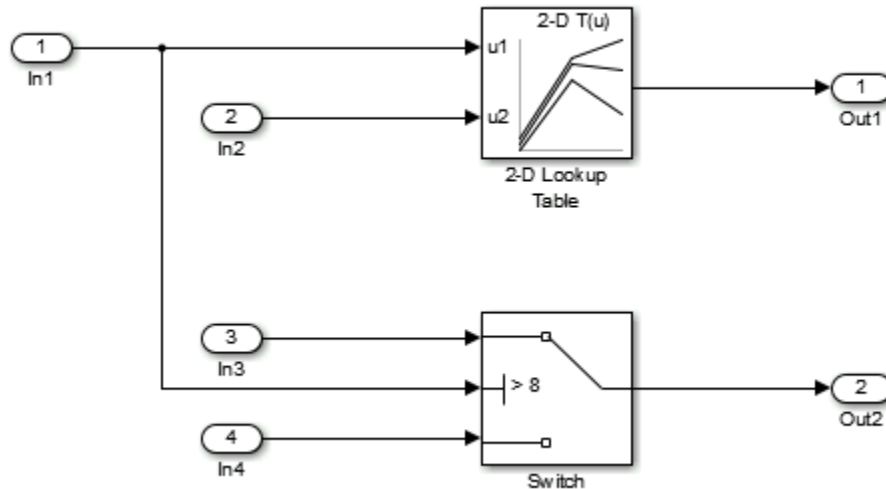
Tests generated without replacing the 2D Lookup Table satisfy two objectives: that the trigger is not greater than the Switch block threshold 8, and that the trigger is greater than the Switch block threshold 8.

Objectives Satisfied

Simulink Design Verifier found test cases that exercise these test objectives.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Switch	trigger > threshold false (output is from 3rd input port)	1	1
2	Decision	Switch	trigger > threshold true (output is from 1st input port)	1	2

Test generation with block replacement returns a different analysis. The `blkrep_rule_lookup2D_normal.m` block replacement rule replaces the 2D Lookup Table with a masked subsystem containing the 2D Lookup Table and a verification subsystem.
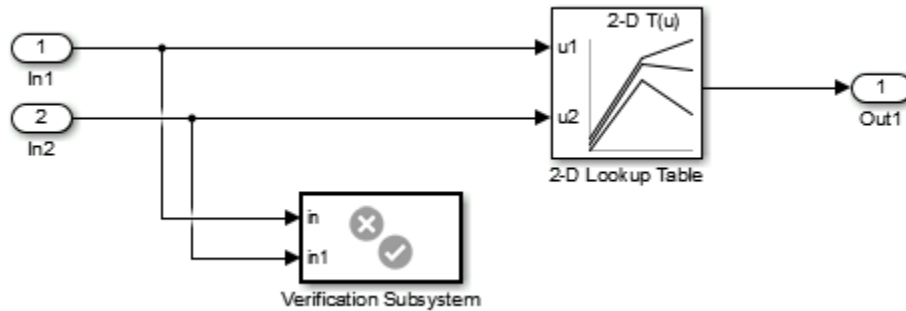
The verification subsystem constrains the analysis within the breakpoint bounds of the table. The additional constraints prevent generating tests that exercise the second objective for the Switch block. The condition that the input signal `In1 > 8` is unsatisfiable.

Objectives Satisfied

Simulink Design Verifier found test cases that exercise these test objectives.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Switch	trigger > threshold false (output is from 3rd input port)	0	1

Objectives Proven Unsatisfiable

Simulink Design Verifier proved that there does not exist any test case exercising these test objectives. This often indicates the presence of dead-logic in the model. Other possible reasons can be inactive blocks in the model due to parameter configuration or test constraints such as given using Test Condition blocks. In rare cases, the approximations performed by Simulink Design Verifier can make objectives impossible to achieve.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
2	Decision	Switch	trigger > threshold true (output is from 1st input port)	0	n/a

Built-In Block Replacements

The Simulink Design Verifier software provides a set of block replacement rules and a corresponding library of replacement blocks. Use these built-in block replacements when analyzing models. They serve as examples that you can examine to learn how to create your own block replacements.

The following table lists the factory default block replacement rules, available in the *matlabroot*\toolbox\sldv\sldv\private folder. There are two implementations of each factory-default block replacement rule. Rules whose file names end with _normal.m replace blocks with Subsystem blocks. Rules whose file names end with _configss.m replace blocks with Configurable Subsystem blocks.

File Name	Description
blkrep_rule_lookup_normal.m blkrep_rule_lookup_configss.m	A rule that replaces 1-D Lookup Table blocks with an implementation that includes test objectives for each breakpoint and interval specified by the **Breakpoints** parameter.
blkrep_rule_lookup2D_normal.m blkrep_rule_lookup2D_configss.m	A rule that adds Test Condition/Proof Assumption blocks to the input ports of 2-D Lookup Table blocks. Each Test Condition/Proof Assumption block constrains signal values to the interval specified by the corresponding breakpoint vector.
blkrep_rule_mpswitch2_normal.m blkrep_rule_mpswitch2_configss.m	A rule that adds a Test Condition/Proof Assumption block to the control input port of Multiport Switch blocks whose **Number of data ports** parameter is 2. The Test Condition/Proof Assumption block constrains signal values to the interval [1, 2] (or [0, 1] if the block uses zero-based indexing).
blkrep_rule_mpswitch3_normal.m blkrep_rule_mpswitch3_configss.m	A rule that adds a Test Condition/Proof Assumption block to the control input port of Multiport Switch blocks whose **Number of data ports** parameter is 3. The Test Condition/Proof Assumption block constrains signal values to the interval [1, 3] (or [0, 2] if the block uses zero-based indexing).

File Name	Description

`blkrep_rule_mpswitch4_normal.m`
 	
`blkrep_rule_mpswitch4_configss.m`
	A rule that adds a Test Condition/Proof Assumption block to the control input port of `Multiport Switch` blocks whose **Number of data ports** parameter is 4. The Test Condition/Proof Assumption block constrains signal values to the interval [1, 4] (or [0, 3] if the block uses zero-based indexing).

`blkrep_rule_mpswitch5_normal.m`
 	
`blkrep_rule_mpswitch5_configss.m`
	A rule that adds a Test Condition/Proof Assumption block to the control input port of `Multiport Switch` blocks whose **Number of data ports** parameter is 5. The Test Condition/Proof Assumption block constrains signal values to the interval [1, 5] (or [0, 4] if the block uses zero-based indexing).

`blkrep_rule_switch_normal.m`
 	
`blkrep_rule_switch_configss.m`
	A rule that replaces `Switch` blocks with an implementation that includes test objectives, requiring that each switch position be exercised when the values of the first and third input ports are different.

`blkrep_rule_selector`
    `IndexVecPort_normal.m`
 	
`blkrep_rule_selector`
    `IndexVecPort_configss.m`
	A rule that adds a Test Condition/Proof Assumption block to the index port of `Selector` blocks whose **Index Option** parameter is `Index vector (port)`. The Test Condition/Proof Assumption block constrains signal values to an interval whose endpoints are derived from the values of the Selector block's **Input port size** and **Index mode** parameters.

`blkrep_rule_selector`
    `StartingIdxPort_normal.m`
 	
`blkrep_rule_selector`
    `StartingIdxPort_configss.m`
	A rule that adds a Test Condition/Proof Assumption block to the index port of `Selector` blocks whose **Index Option** parameter is `Starting index (port)`. The Test Condition/Proof Assumption block constrains signal values to an interval whose endpoints are derived from the values of the Selector block's **Input port size**, **Output size**, and **Index mode** parameters.

The library of replacement blocks that corresponds to the factory default rules is

matlabroot/toolbox/sldv/sldv/sldvblockreplacementlib

Template for Block Replacement Rules

To help you create block replacement rules, Simulink Design Verifier provides an annotated template that contains a skeleton implementation of the requisite callbacks:

matlabroot`/toolbox/sldv/sldv/sldvblockreplacetemplate.m`

To create a block replacement rule, make a copy of the template and edit the copy to implement the desired behavior for the rule you are creating. The comments in the template provide hints about how to use each section. For a tutorial on using the template to write custom block replacements rules, see "Write Block Replacement Rules" on page 4-11.

Define Custom Block Replacements

Basic Workflow for Defining Custom Block Replacements

To replace certain blocks in your model in a way that the factory-default block replacement rules do not handle, create custom block replacement rules by completing the following tasks:

- "Specify Replacement Blocks" on page 4-10
- "Write Block Replacement Rules" on page 4-11

Specify Replacement Blocks

A replacement block can be one of the built-in blocks in the Simulink model library or a block in a user-created library.

In Simulink Design Verifier, replacement blocks have the following restrictions:

- They must be built-in blocks or subsystems.
- They cannot be Model blocks, nor can they include any Model blocks.

  Note: A Model block cannot be a replacement block, but you can replace Model blocks with built-in blocks or subsystems.

- They must reside in a block library that is available on your MATLAB search path.
- If the replacement block is a subsystem, any `Inport` and `Outport` blocks *must* have the default names (`In1` and `Out1`).

After constructing your replacement block, write a custom block replacement rule.

Write Block Replacement Rules

Block replacement rules have the following restrictions:

- The function that represents a block replacement rule must include particular callbacks. Use the block replacement rule template as a starting point for writing a custom rule. (See "Template for Block Replacement Rules" on page 4-9.)
- The function that represents a block replacement rule must be on the MATLAB search path.

Replace Multiport Switch Blocks

- "Why Replace Multiport Switch Blocks?" on page 4-11
- "Create the Library and Replacement Block" on page 4-12
- "Write the Rule for the Replacement Block" on page 4-14

Why Replace Multiport Switch Blocks?

A Multiport Switch block has one control input port and one or more data input ports; the default number of data inputs is 3.
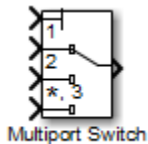
A model may have test objectives on some blocks whose output is directly or indirectly connected to the Multiport Switch block. For example, a Saturation block may send data to the control input port. In this case, the analysis may create test cases that satisfy those objectives. However, those test cases may create values that are out of range for the control input port, regardless of whether the Multiport Switch block uses zero-based indexing or one-based indexing. This causes the simulation to fail.

In this example, you create a rule to replace all Multiport Switch blocks that have two data inputs and do not use zero-based indexing. The replacement block is a subsystem that has a Test Condition block that constrains the value of the control input to 1 or 2, so that the analysis does not create out-of-range data input values. This allows the analysis to satisfy the objectives on blocks that are connected to the control input port of the Multiport Switch block.

Create the Library and Replacement Block

Create a user library and specify the replacement block as a masked subsystem:

1   In the Simulink Library Browser, select **File** > **New** > **Library**.

2   In your library, create a subsystem named `myReplacementBlock` to represent your replacement block.

3   Inside `myReplacementBlock`, add two `Inport` blocks so that the subsystem has three input ports and one output port.
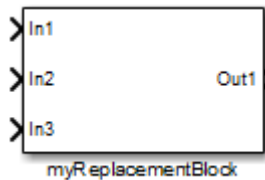
4   Add a `Multiport Switch` block and a `Test Condition` block to the subsystem. Set the block parameters as follows.

   •   In the Multiport Switch block, set the **Number of data ports** parameter to 2.

   •   In the Test Condition block, set the **Values** parameter to `{[1, 2]}`.
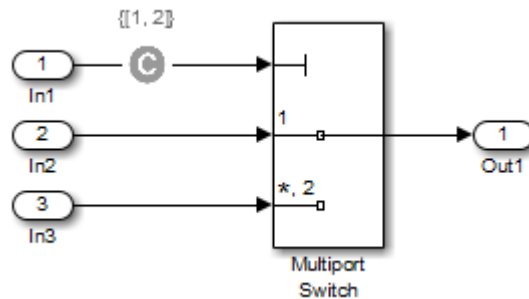
5   To create a mask for your subsystem, from the top-level model, right-click `myReplacementBlock`. From the context menu, select **Mask** > **Create Mask**.

6   In the Mask Editor, specify the following information:

   •   In the **Parameters** pane, define a mask parameter named `InputSameDT` as shown.

This parameter replicates the behavior of the **Require all data port inputs to have the same data type** parameter of the underlying Multiport Switch block.
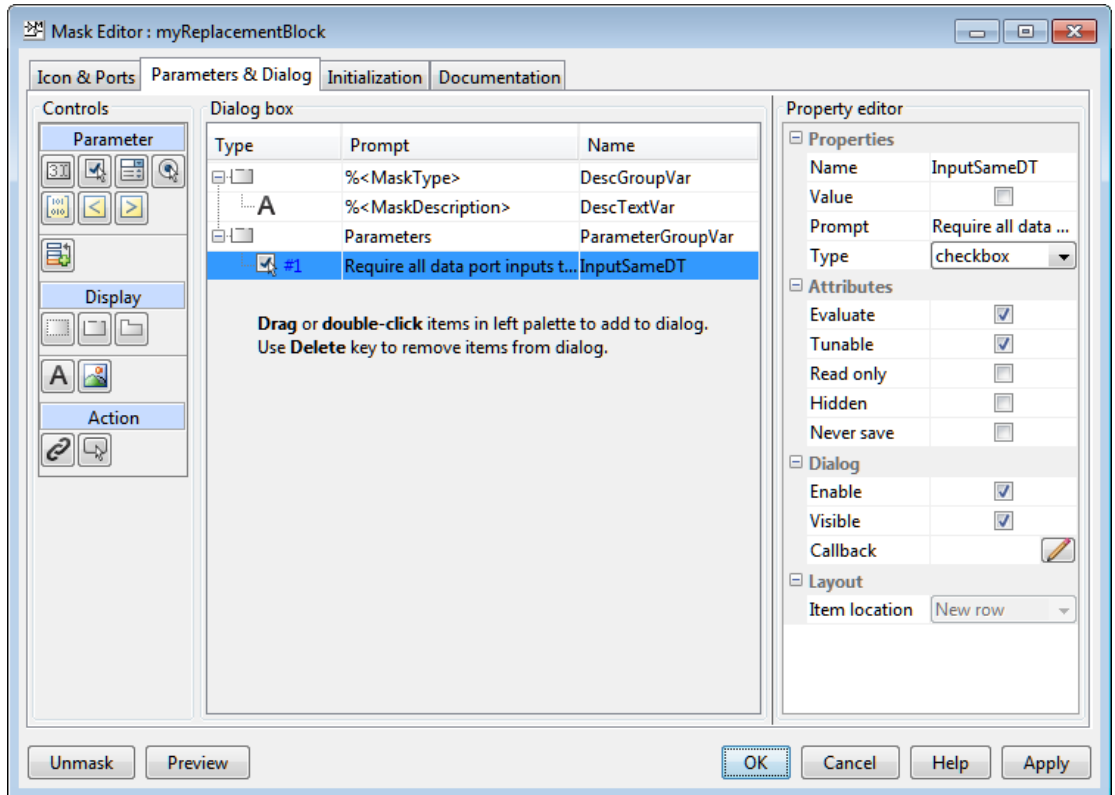
- In the **Initialization** pane, in the **Initialization commands** field, enter commands to specify that the subsystem inherit the `InputSameDT` parameter value of the top-level model:

```
maskInputSameDT = get_param(gcb,'InputSameDT');
blkName = sprintf('/Multiport\nSwitch');
targetBlock = [gcb, blkName];
set_param(targetBlock,'InputSameDT',maskInputSameDT);
```

7 Save your block library as a model file called `custom_rule` in a folder on your MATLAB search path.

Write the Rule for the Replacement Block

To write a rule for the replacement block:

1   Open the block replacement rule template

matlabroot/toolbox/sldv/sldv/sldvblockreplacetemplate.m

2   Make a copy of the file and save it as `custom_rule_switch.m` in a folder on your MATLAB search path.

Note: Execute steps 3 through 11 for the copy of the template that you saved.

---

3   To declare a function `custom_rule_switch` and modify its help, modify the first few lines of the template:

```
function rule = custom_rule_switch
%CUSTOM_RULE_SWITCH Custom block replacement rule for
%Simulink Design Verifier
%
%   This block replacement rule identifies Multiport
%   Switch blocks whose "Number of inputs" parameter
%   specifies '2' and "Use zero-based indexing" parameter
%   specifies 'off'. It replaces such blocks with an
%   implementation that includes a Test Condition block
%   on the control input signal.
```

The function name must match its file name, without the `.m` extension. The comments that follow the function declaration create help for this rule.

4   Specify the type of block that you want to replace in your model by specifying its `BlockType` parameter as the `rule.blockType` object. For this example, change the `rule.blockType` object to `'MultiPortSwitch'`:

```
%% Target Block Type
%
rule.BlockType = 'MultiPortSwitch';
```

Note: You can use the `get_param` function to obtain the value of the `BlockType` parameter for the block that you want to replace.

---

5   Specify the full block path name for the replacement block as the `rule.ReplacementPath` object. For this example, to replace Multiport Switch blocks with the replacement block developed in "Specify Replacement Blocks" on

page 4-10, modify the `rule.ReplacementPath` object using the full block path name:

```
%% Replacement Library
%
rule.ReplacementPath = sprintf('custom_rule/myReplacementBlock');
```

Note: To get the full block path name, use the `gcb` function.

6 To specify the type of subsystem that the software uses when replacing blocks, specify a value for the `rule.ReplacementMode` object. Valid values are:

   - `Normal` — The software replaces blocks with a copy of the subsystem specified by the `rule.ReplacementPath` object. This is the default.

   - `ConfigurableSubSystem` — The software replaces blocks with a Configurable Subsystem block. With the Configurable Subsystem block, you can choose whether it represents the subsystem specified by the `rule.ReplacementPath` object, or the original block before its replacement.

For this example, set `rule.ReplacementMode` to `Normal`:

```
%% Replacement Mode
%
rule.ReplacementMode = 'Normal';
```

7 Specify parameter values that the replacement blocks inherit from the blocks being replaced. You achieve inheritance by mapping the parameter names in a structure. Each field of the structure represents a parameter that the replacement block inherits. Specify the value of each field using the token $original.*parameter*$. *parameter* is the name of the parameter that belongs to the original block.

To define a structure named `parameter` that maps the `InputSameDT` parameter from the original Multiport Switch blocks to their replacement blocks, change the content of the `Parameter Handling` section as follows:

```
%% Parameter Handling
%
parameter.InputSameDT = '$original.InputSameDT$';

% Register the parameter mapping for the rule
rule.ParameterMap = parameter;
```

Note: To determine block parameter names, refer to "Block Libraries" in the Simulink documentation.

---

8 To define the callback functions, keep the following lines in the file:

```
%% Replacement Test Callback
% Customize the local function 'replacementTestFunction' to specify the
% conditions under which Simulink Design Verifier replaces blocks when
% using this rule. Simulink Design Verifier replaces blocks only when this
% local function returns true.
%
rule.IsReplaceableCallBack = @replacementTestFunction;

%% Post Replacement Callback
% Customize the local function 'postReplacementFunction' to specify actions
% that will be performed after a block is replaced.
%
% The usage of this callback in replacement rules is optional. Simulink
% Design Verifier does not enforce its existence in the rule definition.
%
rule.PostReplacementCallBack = @postReplacementFunction;
```

9 Customize `replacementTestFunction` by specifying conditions under which Simulink Design Verifier replaces blocks in your model.

To instruct the software to replace only the Multiport Switch blocks whose `NumInputPorts` parameter is `2` and whose `zeroIdx` parameter is `off`, replace the existing `replacementTestFunction` with the following:

```
function out = replacementTestFunction(blockH)
% Specify the logic that determines when the Simulink Design
% Verifier software replaces a block in your model. For example,
% restrict replacements to only the blocks whose parameters
% specify particular values.
%
out = false;
numInputPorts = eval(get_param(blockH,'NumInputPorts'));
zeroIdx = get_param(blockH,'zeroIdx');
if numInputPorts==2 && strcmp(zeroIdx,'off')
   out = true;
end
```

Because `replacementTestFunction` executes after the model has been compiled, you can access parameters such as `CompiledPortDataTypes` or `CompiledPortDimensions` from `replacementTestFunction`.

For an example of a `replacementTestFunction` that accesses these parameters, open the following file:

matlabroot/toolbox/sldv/sldv/private/blkrep_rule_switch_normal.m

10 Optionally, you can customize `postReplacementFunction` to specify the actions the software performs after a block has been replaced.

For an example of a `postReplacementFunction`, open the following file:

matlabroot/toolbox/sldv/sldv/private/blkrep_rule_selectorIndexVecPort_normal.m

11 Save the edited file and continue to the next section, "Execute Block Replacements" on page 4-18, to execute your replacement rule.

Execute Block Replacements

Configure Block Replacements

You must configure block replacement options before executing block replacements in your model. To specify block replacement options from the model window:

1   Open the sldvdemo_param_identification model.

2   Rename this model to my_sldvdemo_param_identification, and save it in a folder on your MATLAB search path.

3   In the Model Editor, select **Analysis** > **Design Verifier** > **Options**.

   The Configuration Parameters dialog box displays the main pane of the **Design Verifier** category.

4   In the Configuration Parameters dialog box, select **Design Verifier** > **Block Replacements**.

5   On the **Block Replacements** pane, select **Apply block replacements** to enable block replacements.

   Selecting this check box provides access to the **List of block replacement rules (in order of priority)** and **File path of the output model** options.

6   To execute your custom block replacement rule, follow these steps:

   a   In the **List of block replacement rules (in order of priority)** box, delete:

      <FactoryDefaultRules>

   b   Enter:

      custom_rule_switch

   The Simulink Design Verifier software replaces a block in your model only once. If multiple rules apply to the same block, the software replaces the block using the rule with the highest priority.

7 In the **File path of the output model** field, accept the default to create a model named `my_sldvdemo_param_identification_replacement`. This model is a copy of the original model and includes the block replacements.

By default, this software creates a model named `$ModelName$_replacement`, where `$ModelName$` is the name of the model it is analyzing. To use a different name for the model with the block replacements, enter the name in this field. You do not need to include a file extension.

8 Click **Apply**.

9 Save the `my_sldvdemo_param_identification` model.

Replace Blocks in a Model

- "Replace Blocks and Analyze Model with the Block Replacements" on page 4-19
- "Perform the Block Replacements Only" on page 4-20

Replace Blocks and Analyze Model with the Block Replacements

After enabling the **Apply block replacements** option, you can start a Simulink Design Verifier analysis that analyzes the model after executing the block replacements. To trigger block replacements and start the analysis, do one of the following:

- Select **Analysis** > **Design Verifier** > **Options**, and on the **Design Verifier** pane, click **Generate Tests**.
- In the Model Editor, select **Analysis** > **Design Verifier** > **Generate Tests** > **Model**.

Note: If your model has unsaved changes, Simulink Design Verifier asks if you want to save the model before executing the block replacements.

The Simulink Design Verifier software copies your model, replaces blocks in the copy, without altering the original model, and analyzes the model with the replacements.

Upon completing its analysis, you can generate a detailed analysis report that includes information about the block replacements it executed. For each block replacement, you can follow a link from the report to the block replacement in the model copy, saved using the name you specified on the **Design Verifier** > **Block Replacements** pane of the Configuration Parameters dialog box.

4-19

Perform the Block Replacements Only

Replacing the blocks in a model *before* running the analysis can help you debug the custom block replacement rules. Once the block replacement rules are working as you want, analyze the model that contains the block replacements.

To perform only the block replacements, without analyzing the model with the block replacements, at the command line or from a program, use the `sldvblockreplacement` function. Set two parameters of the `sldvoptions` structure related to replacing blocks, and call `sldvblockreplacement` as follows:

```
opts = sldvoptions;
opts.BlockReplacement = 'on'
opts.BlockReplacementRulesList = ...
    'custom_rule_switch, <FactoryDefaultRules>';
[status, newmodelH] = sldvblockreplacement(...
    'my_sldvdemo_param_identification', opts);
```

If you execute block replacements programmatically, in the MATLAB Command Window, Simulink Design Verifier displays a table that lists available block replacement rules and opens the copy of the model that contains the block replacements (`$ModelName $_replacement`).

The table lists all built-in rules and any custom rules that you specified using the **List of block replacement rules (in order of priority)** option (see "Configure Block Replacements" on page 4-18). The table includes the following information:

- `Type`
  Type of rule, either built-in or custom
- `Registration MATLAB File name`
  Name of the file that expresses the rule
- `Block types`
  `BlockType` parameter value of the block that the rule replaces
- `Priority`
  Priority of execution when multiple rules target the same type of block for replacement
- `Active`
  Flag that indicates whether the rule is active (1) or ignored (0)

The output also displays information about the block replacements. For example, the output for this example indicates that the software used the `custom_rule_switch.m` rule to replace a Multiport Switch block (of the same name) at the top level of the model.

5

Specifying Parameter Configurations

Parameter Constraint Values

In this section...
"Parameter Configuration for Analysis" on page 5-2
"Data Types in Parameter Configurations" on page 5-3
"Parameters in Variant Subsystems" on page 5-3

Parameter Configuration for Analysis

Simulink Design Verifier software can treat parameters in your model as variables during its analysis. For example, suppose you specify a variable that is defined in the MATLAB workspace as the value of a block parameter in your model. You can instruct Simulink Design Verifier to use additional values for that parameter in its analysis.

This allows you to, for example:

- Extend the results of a error detection analysis property proof to consider the impact of additional parameter values.
- Generate comprehensive test cases for situations in which parameter values must vary to achieve more complete coverage results. For more information, see "Specify Parameter Constraint Values for Full Coverage" on page 5-10.

If you place a constraint on a parameter in your model, during analysis that parameter takes only your specified constraint value or values. A group of constraints on parameters in the same model is also called a *parameter configuration*.

Use the Parameter Table to manage constraints on your model parameters for analysis. In the Parameter Table, you can:

- Autogenerate value ranges for parameters in your model. See "Autogenerate Parameter Constraint" on page 5-13.
- Enter your own value ranges for parameters in your model. See "Define Constraint Values for Parameters" on page 5-4.
- Highlight objects in your model that have parameters configured to act as variables during analysis. See "Highlight Constrained Parameters in Model" on page 5-9.
- Import and export parameter configurations from MATLAB code files. See "Store Parameter Constraints in MATLAB Code Files" on page 5-21.

> **Note:** When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. In **Analysis** > **Design Verifier** > **Options**, if you specify your model to extend existing test cases with a **Data file** and apply parameter configurations with a **Parameter configuration file** or the Parameter Table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.

Data Types in Parameter Configurations

Consider the following issues related to data types when constraining parameter values:

- "Parameters Cannot Be Structures" on page 5-3
- "Parameters Converted to Fixed Point in the Model" on page 5-3
- "Parameters Defined as Simulink.Parameter and Referenced by Multiple Locations" on page 5-3

Parameters Cannot Be Structures

If the data type of a parameter in the MATLAB workspace is `struct`, Simulink Design Verifier cannot generate values for that parameter during the analysis.

Parameters Converted to Fixed Point in the Model

If your model references a base workspace parameter whose data type is `auto`, `single`, or `double`, and the model converts that parameter to a fixed-point data type, you must define the constraints for that parameter according to its fixed-point type.

Parameters Defined as `Simulink.Parameter` and Referenced by Multiple Locations

For a parameter defined as Simulink.Parameter or an inherited class of `Simulink.Parameter` whose data type is `auto`, if the parameter is referenced by multiple locations with different data types, Simulink Design Verifier cannot generate values for that parameter during the analysis.

Parameters in Variant Subsystems

Parameters can be used to select variants in Variant Subsystem blocks. These parameters are listed in the Parameter Table. However, Simulink Design Verifier only supports analyzing the active variant.

Define Constraint Values for Parameters

In this section...
"Find Parameters and Autogenerate Constraints" on page 5-6
"Edit Parameter Constraints" on page 5-8
"Highlight Constrained Parameters in Model" on page 5-9

Using the Parameter Table, you can find and autogenerate constraints for parameters in your model. This example uses the following model, which contains **Gain** and **Constant** parameters defined as m and b, respectively.
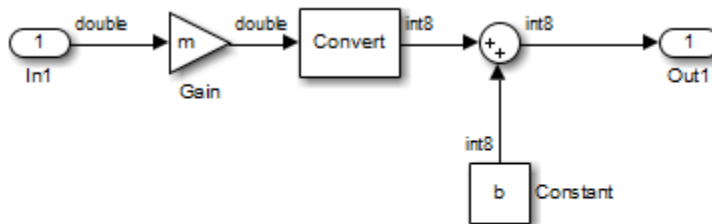
The model callback function PreLoadFcn defines m and b in the MATLAB workspace.
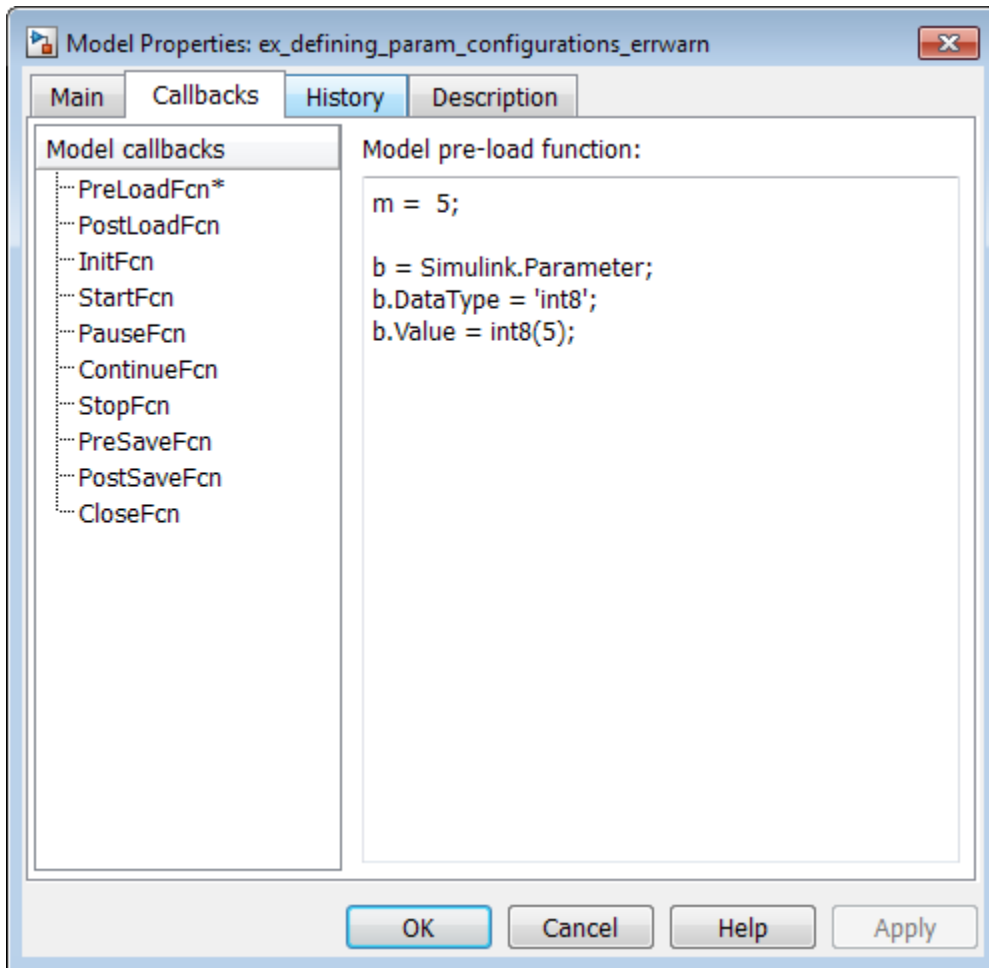
When the model opens:

- `m` is set to 5.
- `b` is a `Simulink.Parameter` object of type `int8` whose value is set to 5.

Find Parameters and Autogenerate Constraints

This example shows how to specify values or ranges of values used for model parameters during Simulink Design Verifier analysis.

Open the Parameter Table.

In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**. In the **Select** tree, choose **Design Verifier** > **Parameters**.

Enable the Parameter Table.

In the **Parameters** pane, select **Enable parameter configuration** and **Use parameter table**.

Find parameters that can be constrained for analysis.

At the bottom of the Parameter Table, click **Find in Model**. The Parameter Table searches your model for parameters that can be configured and loads them in the table.

When possible, the Parameter Table autogenerates constraint values for parameters. You can use these autogenerated values or specify your own constraint.

In this example, in the Parameter Table, rows for model parameters m and b appear.

Each row represents a parameter configuration. You can edit the parameter's constraint value(s) in the field under **Constraint**. To use your specified parameter configuration in analysis, select the check box in the field under **Use**. The following table provides more details about these and other columns in the Parameter Table.

For parameter in row, the column...	Shows...
Use	Whether specified constraint for parameter is used in analysis.
	To include parameter configuration in analysis, select the check box. To exclude

For parameter in row, the column...	Shows...
	parameter configuration from analysis, clear the selection.
Name	Name of parameter.
Constraint	Autogenerated or user-specified constraint value(s) for parameter. To change the specified constraint value(s), double-click in this field and enter new constraint value(s).
Value	Value of parameter. If the parameter is defined in a Simulink data dictionary that is linked to the model, the column shows the value of the parameter in the data dictionary. Otherwise, it shows the value of the parameter in the base workspace.
Min	Specified minimum value for parameter, if parameter is of type `Simulink.Parameter` and has a specified minimum value.
Max	Specified maximum value for parameter, if parameter is of type `Simulink.Parameter` and has a specified maximum value.
Model Element	Path to model component(s) where parameter is used.

Note: If you use a MATLAB variable from a data dictionary as a model parameter, SLDV analysis does not consider the parameter as tunable. If you want the parameter to be tunable for the analysis, use a `Simulink.Parameter` object for the parameter. To create a `Simulink.Parameter` object in the data dictionary:

1   In the Model Explorer, on the **Model Hierarchy** pane, select the workspace under the data dictionary that contains your MATLAB variable.

2   Select **Add** > **Simulink Parameter**. You see a new variable titled `Param` in the workspace.

3 Rename the variable. Assign the same data type as the original MATLAB variable.

4 In your model, use the variable that you just created as your parameter.

Edit Parameter Constraints

For each parameter you want to treat as a variable during analysis, specify constraint values.

In the Parameter Table, in the **Constraint** column, double-click the field for the parameter you want to constrain. Enter your constraint values.

For this example:

- For parameter b, specify the value range [4, 10].
- For parameter m, specify the value 3.

To enable a parameter configuration for analysis, click to select the row that corresponds to the configured parameter. Click **Enable**.

To enable multiple parameter configurations at once, shift-click to select multiple rows, and click **Enable**.

To exclude parameter configurations from analysis, click to select the row that corresponds to the configured parameter. Click **Disable**.

When you disable a parameter configuration, the specified constraint for this parameter is not used in analysis.

To disable multiple parameter configurations at once, shift-click to select multiple rows, and click **Disable**.

To exclude a parameter configuration from analysis and delete its specified constraint, click to select the row that corresponds to the configured parameter. Click **Clear**.

The Parameter Table clears the specified constraint for the parameter, and the parameter configuration is excluded from analysis.

To clear multiple parameter configurations at once, shift-click to select multiple rows, and click **Clear**.

Highlight Constrained Parameters in Model

Highlight model components that use the parameters for which you have specified constraints.

Select the parameter(s) you want to highlight in the model.

To select a parameter, click anywhere inside the **Name** or **Constraint** columns for either parameter. Shift-click to select multiple parameters.

Click **Highlight in Model**.

In the Simulink Editor, model components that use the selected parameters are highlighted.

Specify Parameter Constraint Values for Full Coverage

In this section...
"About This Example" on page 5-10
"Construct Example Model" on page 5-11
"Parameterize Constant Block" on page 5-12
"Preload Workspace Variable" on page 5-12
"Autogenerate Parameter Constraint" on page 5-13
"Analyze Example Model" on page 5-14
"Simulate Test Cases" on page 5-17

About This Example

This example describes how to create and analyze a simple Simulink model, for which you generate test cases that achieve decision coverage. However, in this example, achieving complete decision coverage is possible only when Simulink Design Verifier treats a particular block parameter as a variable during its analysis. This example explains how to specify parameter configurations for use with the analysis.

The following workflow guides you through the process of completing this example.

Task	Description	See...
1	Construct the example model.	"Construct Example Model" on page 5-11
2	Specify a variable as the value of a Constant block parameter.	"Parameterize Constant Block" on page 5-12
3	Constrain the value of the variable that the Constant block specifies.	"Autogenerate Parameter Constraint" on page 5-13
4	Generate test cases for your model and interpret the results.	"Analyze Example Model" on page 5-14
5	Simulate the test cases and measure the resulting decision coverage.	"Simulate Test Cases" on page 5-17

Construct Example Model

Construct a simple Simulink model to use in this example:

1 Create an empty Simulink model.

2 Copy the following blocks into the empty Simulink Editor:

   - From the Sources library:

     - Two `Inport` blocks to initiate the input signals
     - A `Constant` block to control the switch

   - From the Signal Routing library: A `Multiport Switch` block to provide simple logic

   - From the Sinks library: An `Outport` block to receive the output signal

3 Double-click the Multiport Switch block to access its dialog box and specify its **Number of data ports** option as `2`.

4 Connect the blocks so that your model looks like the following.
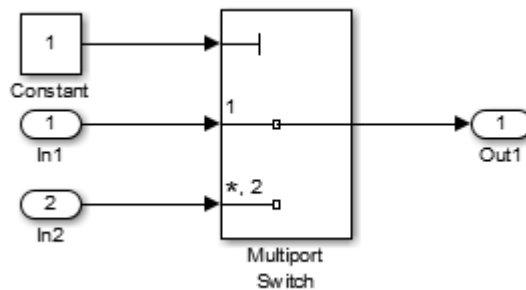
5 Select **Simulation** > **Model Configuration Parameters**.

6 In the **Select** tree on the left side of the Configuration Parameters dialog box, select the **Solver** node. Under **Solver options**, set the **Type** option to `Fixed-step`, and then set the **Solver** option to `discrete (no continuous states)`.

7 In the **Select** tree, select the **Diagnostics** node. Set **Automatic solver parameter selection** to `none`.

8 Click **OK** to apply your changes and close the Configuration Parameters dialog box.

9 Save your model as `ex_defining_params_example` for use in the next procedure.

Parameterize Constant Block

Parameterize the Constant block in your model by specifying a variable as the value of the Constant block's **Constant value** parameter:

1   Double-click the Constant block.

2   In the **Constant value** box, enter A.

3   Click **OK** to apply your change and close the Constant block parameter dialog box.

4   Save your model.

Preload Workspace Variable

Preload the value of the MATLAB workspace variable A referenced by the Constant block:

1   Select **File** > **Model Properties** > **Model Properties**.

2   Click the **Callbacks** tab.

3   In the PreLoadFcn, enter:

```
A = int8(1);
```

4   Click **OK** to close the Model Properties dialog box and save your changes.

5   Close your model.

6   Open your model.

When you open the model, the PreLoadFcn defines a variable A of type int8 whose value is 1.
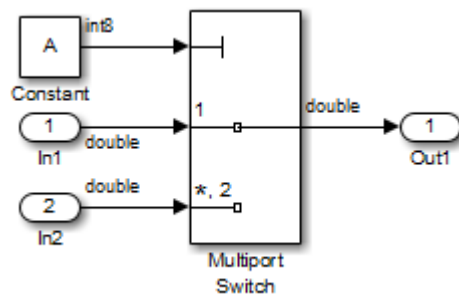
Autogenerate Parameter Constraint

Use the Parameter Table to constrain the variable A to specified values.

1   In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**.

2   In Configuration Parameters dialog box, from the **Select** tree under **Design Verifier**, select **Parameters**.

3   Select **Enable parameter configuration**.

4   Select **Use parameter table**.

5   At the bottom of the Parameter Table, click **Find in Model**.

The Parameter Table is populated with parameters from your model. When possible, it autogenerates constraint values for each parameter, depending on the data type and location of the parameter in the model.

In this case, a row appears for the parameter A that you defined. The table row for A displays the following information:

- In the **Name** column, the parameter name (A).

- In the **Constraint** column, the constraint specified on parameter A. The Parameter Table autogenerates the constraint values {1, 2}.

- In the **Value** column, the value of A in the base workspace. This value is 1.

- In the **Model Element** column, the model component in which A resides (ex_defining_params_example/Constant).

- In the **Use** column, a check box indicating whether the specified constraint values in the table are configured for analysis.
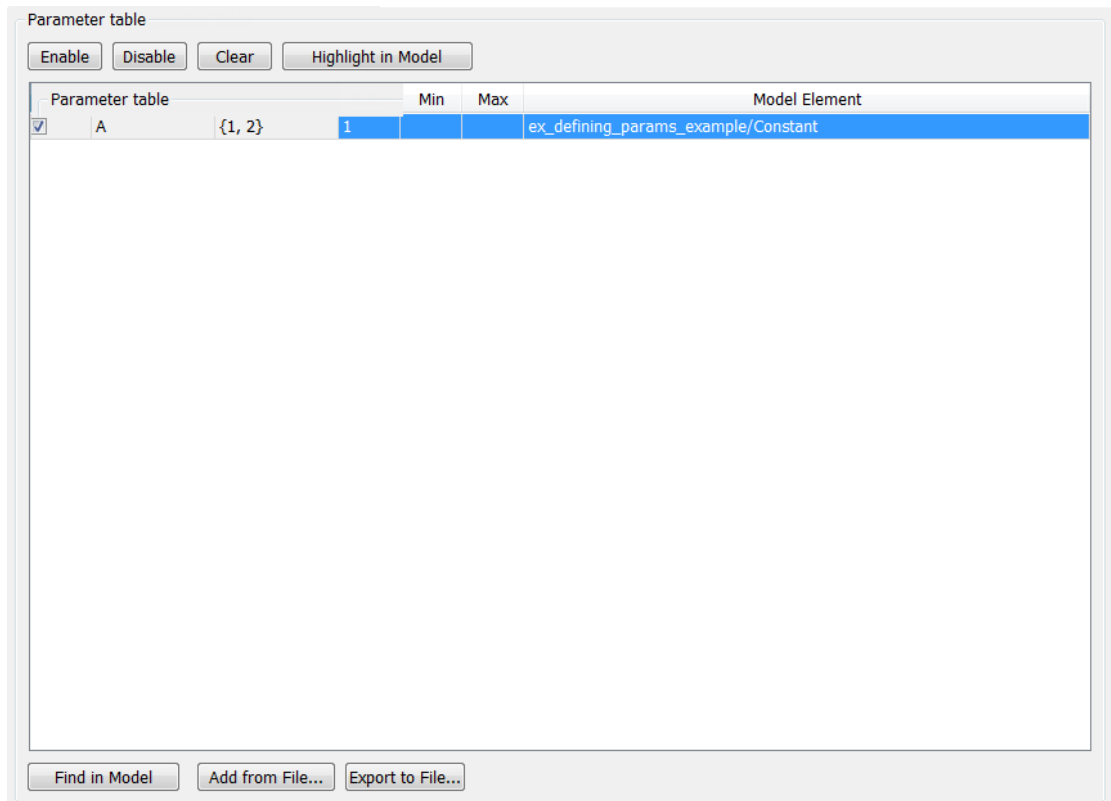
6   In the Parameter Table, in the row for parameter A, make sure that you select the **Use** check box.

When you enable this parameter configuration, during Simulink Design Verifier analysis, the parameter A takes only the `int8` values 1 and 2.

7   In the Configuration Parameters dialog box, click **OK**.

8   Save your model.

Analyze Example Model

Analyze the model using the parameter configuration you just created, and generate the analysis report:

1. In the Simulink Editor, select **Analysis** > **Design Verifier** > **Generate Tests** > **Model**.

   Simulink Design Verifier begins analyzing your model to generate test cases.

2. When the software completes its analysis, in the Simulink Design Verifier Results Summary window, select **Generate detailed analysis report**.

   The software displays an HTML report named `ex_defining_params_example_report.html`.

   Keep the Results Summary window open for the next procedure.

3. In the Simulink Design Verifier report **Table of Contents**, click `Test Cases`.

4. Click `Test Case 1` to display the subsection for that test case.

Test Case 1

Summary

| Length: | 0 second (1 sample period) |
| Objectives Satisfied: | 1 |

Objectives

Step	Time	Model Item	Objectives
1	0	Multiport Switch	integer input value = 1 (output is from input port 1)

Generated Parameter Values

Parameter	Value
A	1

Generated Input Data

Time	0
Step	1
In1	-
In2	-

This section provides details about Test Case 1 that Simulink Design Verifier generated to satisfy a coverage objective in the model. In this test case, a value of 1 for parameter A satisfies the objective.

5 Scroll down to the Test Case 2 section in the **Test Cases** chapter.

Test Case 2

Summary

Length:	0 second (1 sample period)
Objectives Satisfied:	1

Objectives

Step	Time	Model Item	Objectives
1	0	Multiport Switch	integer input value = *,2 (output is from input port 2)

Generated Parameter Values

Parameter	Value
A	2

Generated Input Data

Time	0
Step	1
In1	-
In2	-

This section provides details about Test Case 2, which satisfies another coverage objective in the model. In this test case, a value of 2 for parameter A satisfies the objective.

Simulate Test Cases

Simulate the generated test cases and review the coverage report that results from the simulation:

1   In the Simulink Design Verifier Results Summary window, select **Create harness model**.

    The software creates and opens a harness model named `ex_defining_params_example_harness`.

2   The block labeled Inputs in the harness model is a `Signal Builder` block that contains the test case signals. Double-click the Inputs block to view the test case signals in the Signal Builder block.
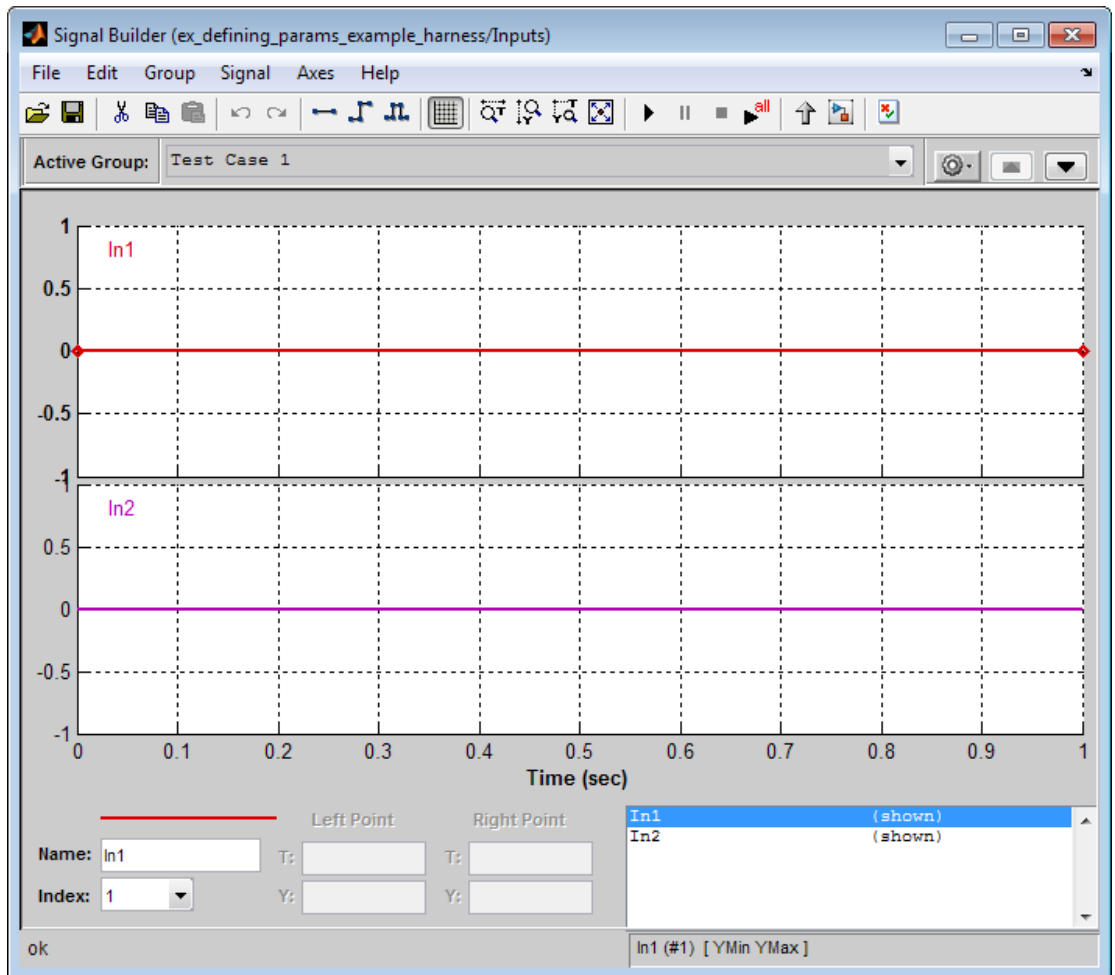
3

In the Signal Builder dialog box, click the **Run all** button .

The Simulink software simulates each of the test cases in succession, collects coverage data for each simulation, and displays an HTML report of the combined coverage results at the end of the last simulation.

4 In the model coverage report, review the **Summary** section:

Summary

Model Hierarchy/Complexity:

D1

1. ex_defining_params_example_harness   2 100% ▬▬▬
2. . . . Test Unit (copied from ex_defining_params_example) 1 100% ▬▬▬

This section summarizes the coverage results for the harness model and its Test Unit subsystem. Observe that the subsystem achieves 100% decision coverage.

5   In the **Summary** section, click the Test Unit subsystem.

The report displays detailed coverage results for the Test Unit subsystem.

2. SubSystem block "Test Unit (copied from ex_defining_param..."

Parent:	/ex_defining_params_example_harness

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	1
Decision (D1)	NA	100% (2/2) decision outcomes

MultiPortSwitch block "Multiport Switch"

Parent:	ex_defining_params_example_harness/Test Unit (copied from ex_defining_params_example)

Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	100% (2/2) decision outcomes

Decisions analyzed:

integer input value	100%
= 1 (output is from input port 1)	2/4
= *,2 (output is from input port 2)	2/4

This section reveals that the Multiport Switch block achieves 100% decision coverage because the test cases exercise each of the switch pathways.

Store Parameter Constraints in MATLAB Code Files

In this section...
"Export Parameter Constraints to File" on page 5-21
"Import Parameter Constraints from File" on page 5-23

You can use the Parameter Table to manage constraints on your model parameters for analysis. If you place a constraint on a parameter in your model, during analysis that parameter takes only your specified constraint value or values. A group of constraints on parameters in the same model is also called a *parameter configuration*. You can store groups of parameter constraints in a MATLAB code file called a parameter configuration file. For more information on configuring parameters for Simulink Design Verifier, see "Define Constraint Values for Parameters" on page 5-4.

To enable parameter configuration, in the Simulink Editor, select **Analysis > Design Verifier > Parameters**. In the **Parameters** pane, select **Enable parameter configuration**.

Export Parameter Constraints to File

Using the Parameter Table, you can export parameter constraint values to a MATLAB code file. If you later want to use the same parameter configuration in a different analysis, you can import your previously specified parameter constraint values from the MATLAB code file.

To export parameter constraint values to a file:

1   Open the Parameter Table. In the Simulink Editor, select **Analysis > Design Verifier > Options**. In the **Select** tree, choose **Design Verifier > Parameters**.

The Parameter Table shows specified constraint values for parameters in your model, as in the following example screen shot.
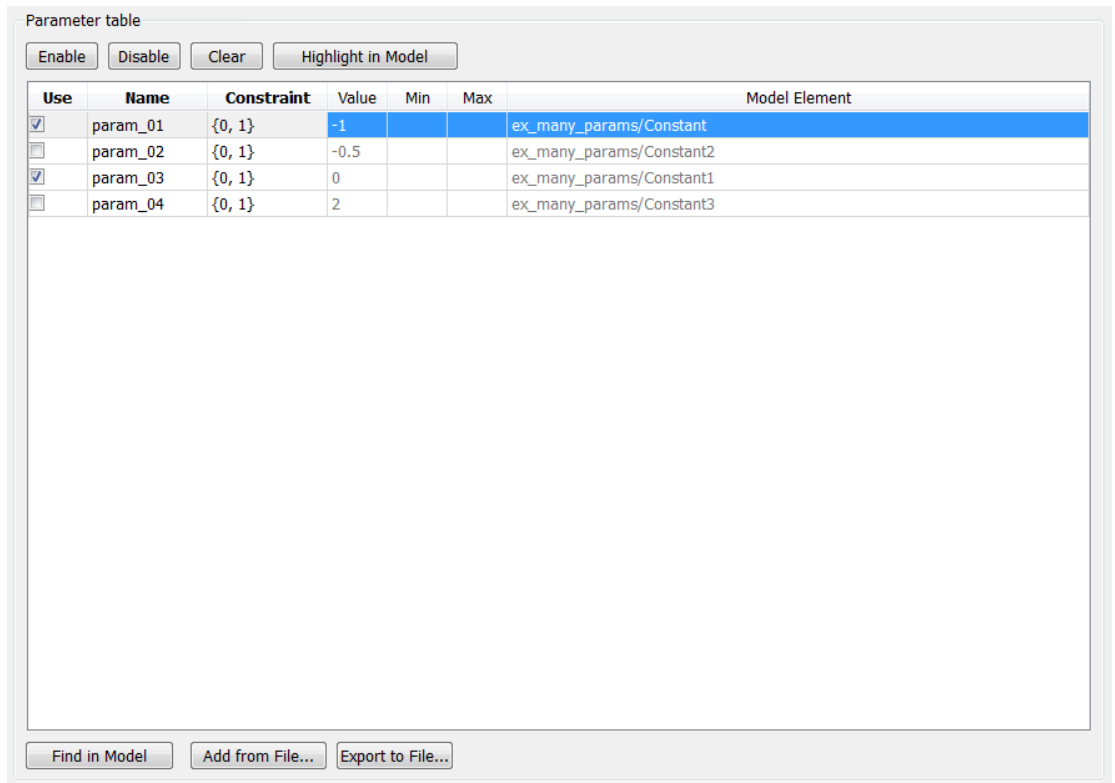
2   Click **Export to File**.

The Parameter Configuration File saves the current parameter configurations to a .m file with the name you specify. Parameters that do not have the **Use** check box enabled appear as commented lines in the parameter configuration file.

In the example shown in the previous step, the parameter configuration file contains the following code:

```
function params = ex_many_params_config
params.param_01 = {0, 1};
% params.param_02 = {0, 01};
params.param_03 = {0, 1};
% params.param_04 = {0, 1};
```

Import Parameter Constraints from File

If you defined parameter configurations for analysis in a release prior to R2014a, you can import corresponding MATLAB files and manage these parameters in the Parameter Table.

To import parameter constraints from a MATLAB code file:

1   Open the Parameter Table. In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**. In the **Select** tree, choose **Design Verifier** > **Parameters**.
2   Click **Add from File**. Choose a parameter configuration file.

   The Parameter Table loads specified parameter constraints from the code, excluding code comments, from the file. If you specify a constraint for a parameter and then load a parameter configuration file containing constraint specification for the same parameter, the constraint specified in the file overwrites the preexisting constraint in the table.

Simulink Design Verifier provides an example parameter configuration file for the example model `sldvdemo_param_identification`:

matlabroot/toolbox/sldv/sldvdemos/sldvdemo_param_ident_config.m

Define Constraint Values for Parameters in MATLAB Code Files

In this section...
"Template Parameter Configuration File" on page 5-24
"Syntax in Parameter Configuration Files" on page 5-24

To specify parameters as variables for analysis, you can use the Parameter Table or define parameter configurations in a MATLAB code file. You can also export parameter configuration files from the Parameter Table. For more information, see "Store Parameter Constraints in MATLAB Code Files" on page 5-21.

This example shows how to define parameter configurations in a MATLAB code file. For an example that shows how to define these parameter configurations using the Parameter Table, see "Define Constraint Values for Parameters" on page 5-4.

Template Parameter Configuration File

The Simulink Design Verifier software provides an annotated template that you can use as a starting point:

matlabroot/toolbox/sldv/sldv/sldv_params_template.m

To create a parameter configuration file, make a copy of the template and edit the copy. The comments in the template explain the syntax for defining parameter configurations.

To associate the parameter configuration file with your model before analyzing the model, in the Configuration Parameters dialog box, on the **Design Verifier** > **Parameters** pane, enter the file name in the **Parameter configuration file** field.

Syntax in Parameter Configuration Files

Specify parameter configurations using a structure whose fields share the same names as the parameters that you treat as input variables.

For example, suppose you want to constrain the **Gain** and **Constant value** parameters, m and b, which appear in the following model:
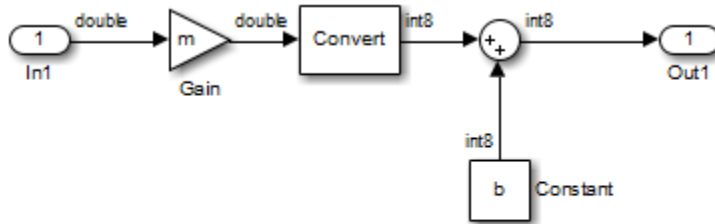
Variables m and b are defined in the MATLAB workspace.

The `PreLoadFcn` callback function defines m and b in the MATLAB workspace when you open the model:

- m is set to 5.

- b is a `Simulink.Parameter` object of type `int8` whose value is set to 5.
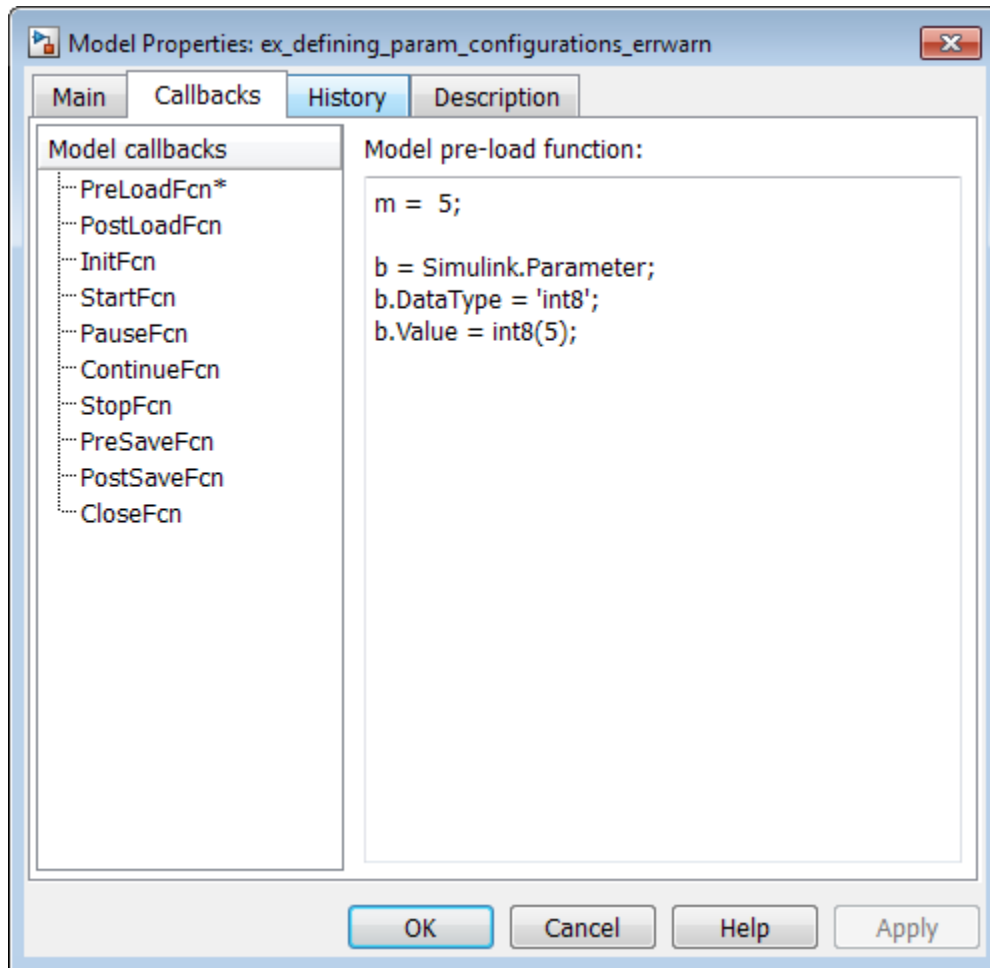
In your parameter configuration file, specify constraints for m and b:

```
params.b = int8([4 10]);
params.m = {};
```

This file specifies:

- b is an 8-bit signed integer from 4 to 10. The constraint type must match the type of the parameter b in the MATLAB workspace, int8, in this example.

- m is not constrained to any values.

Specify points using the `Sldv.Point` constructor, which accepts a single value as its argument. Specify intervals using the `Sldv.Interval` constructor, which requires two input arguments, i.e., a lower bound and an upper bound for the interval. Optionally, you can provide one of the following values as a third input argument that specifies inclusion or exclusion of the interval endpoints:

- `'()'` — Defines an open interval.
- `'[]'` — Defines a closed interval.
- `'(]'` — Defines a left-open interval.
- `'[)'` — Defines a right-open interval.

Note: By default, Simulink Design Verifier considers an interval to be closed if you omit this argument.

The following example constrains m to 3 and b to any value in the closed interval [0, 10]:

```
params.m = Sldv.Point(3);
params.b = Sldv.Interval(0, 10);
```

If the parameters are scalar, you can omit the constructors and instead specify single values or two-element vectors. For example, you can alternatively specify the previous example as:

```
params.m = 3;
params.b = [0 10];
```

Note: To indicate no constraint for an input parameter, specify `params.m = {}` or `params.m = []`. The analysis treats this parameter as free input.

You can specify multiple constraints for a single parameter using a cell array. In this case, the analysis combines the constraints using a logical OR operation.

The following example constrains m to either 3 or 5 and constrains b to any value in the closed interval [0, 10]:

```
params.m = {3, 5};
```

```
params.b = [0 10];
```

You can specify several sets of parameters by expanding the size of your structure. For example, the following example uses a 1-by-2 structure to define two sets of parameters:

```
params(1).m = {3, 5};
params(1).b = [0 10];

params(2).m = {12, 15, Sldv.Interval(50, 60, '()')};
params(2).b = 5;
```

The first parameter set constrains m to either 3 or 5 and constrains b to any value in the closed interval [0, 10]. The second parameter set constrains m to either 12, 15, or any value in the open interval (50, 60), and constrains b to 5.

6

Detecting Design Errors

What Is Design Error Detection?

Design error detection is a Simulink Design Verifier analysis mode that detects the following types of errors:

- Dead logic
- Integer or fixed-point data overflow
- Division by zero
- Intermediate signal values that are outside the specified minimum and maximum values
- Out of bound array access

Before you simulate your model, analyze your model in design error detection mode to find and diagnose these errors. Design error detection analysis determines the conditions that cause the error, helping you identify possible design flaws. Design error detection analysis also computes a range of signal values that can occur for block outports and Stateflow local data in your model.

After the analysis, you can:

- Click individual blocks to view the analysis results for that block.
- Create a harness model containing test cases that demonstrate the errors.
- Create an analysis report that contains detailed results for the entire model.

Derived Ranges in Design Error Detection

When you specify minimum and maximum values for a signal or data in a model, these values define a *design range*.

During design error detection, the software analyzes the model behavior and computes the values that can occur during simulation for:

- Block Outports
- Stateflow local data

The range of these values is called a *derived range*.

The **Use specified input minimum and maximum values** parameter in the Configuration Parameters dialog box, on the **Design Verifier** pane, if enabled, tells the analysis to consider the design ranges on the model input ports as constraints when calculating the derived ranges. By default, the **Use specified input minimum and maximum values** parameter is enabled.

If **Use specified input minimum and maximum values** is disabled, the software does not restrict the signals when computing the derived ranges.

To see how this process works, consider the following model.

In this model, the design ranges are:

- Inport block: [–25..25]
- Abs block output: [10..30]

Given the design range on the Inport block, the only possible values for the Abs block output are values from 0 to 25. Therefore, the derived range for the Abs block is [0..25].

However, if you disable the **Use specified input minimum and maximum values** parameter, the analysis calculates the derived ranges based on unrestricted values of the

input ports of the model. In the preceding model, the only valid outputs of the Abs block are nonnegative numbers. Consequently, the derived range for the Abs block is [0..Inf].

Run a Design Error Detection Analysis

In this section...
"Workflow for Detecting Design Errors" on page 6-5
"Understand the Analysis Results" on page 6-5
"Review the Latest Analysis Results in the Model Explorer" on page 6-8
"Check For Design Errors using the Model Advisor" on page 6-8

Workflow for Detecting Design Errors

To analyze your model for design errors, use the following workflow:

1 Verify that your model is compatible with Simulink Design Verifier software.

2 If you have Stateflow objects in your model, in the Configuration Parameters dialog box, on the **Diagnostics > Stateflow** pane, set **Unreachable execution path** to `error`.

3 Specify options that control how Simulink Design Verifier detects design errors in your model.

4 Execute the Simulink Design Verifier analysis.

5 Review the analysis results.

Note: If you select design error detection for dead logic, you cannot select any other type of design error detection. For dead logic detection, Simulink Design Verifier performs an independent analysis. If you want to detect design errors for dead logic and any of the other types of design errors, you must perform design error detection analysis twice.

Understand the Analysis Results

When you run a design error detection analysis, by default, the software highlights model objects in one of four colors so that the analysis results are easy to review.

Model Object Highlighting Color	Analysis Results
Green	One of the following:

Model Object Highlighting Color	Analysis Results
	• The analysis did not find overflow or division-by-zero errors. • The analysis did not find dead logic. • The analysis did not find intermediate or output signals outside the range of user-specified minimum and maximum constraints. • The analysis did not find out of bound array access errors. **Note:** If your design contains at least one object that Simulink Design Verifier highlights red, other objects in your model that are highlighted green may also contain further design errors. If an object in your design causes run-time errors, Simulink Design Verifier may not be able to determine further errors on objects that are downstream of or rely on the results of the object that causes the run-time errors. Resolve the errors that cause the initial red highlighting and re-run the analysis to determine if Simulink Design Verifier will also highlight other objects in your model as red.
Red	One of the following: • The analysis found at least one test case that causes overflow or division-by-zero errors. • The analysis found dead logic. • The analysis found intermediate or output signals outside the range of user-specified minimum and maximum constraints. • The analysis found at least one test case that causes an out of bound array access error.

Model Object Highlighting Color	Analysis Results

Orange	For at least one objective, the analysis could not determine if the model has dead logic, overflow errors, division-by-zero errors, signals outside the user-specified range, or out of bound array access errors. This situation can occur when:
	
	• The analysis times out.
	
	• The software cannot determine if an error occurred or not. This result is due to:
	
	   • Automatic stubbing errors; for more information, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.
	   • Limitations of the analysis engine.

Gray	The model object was not part of the analysis.

The Simulink Design Verifier Results window initially displays a summary of the analysis results, as in the following example.
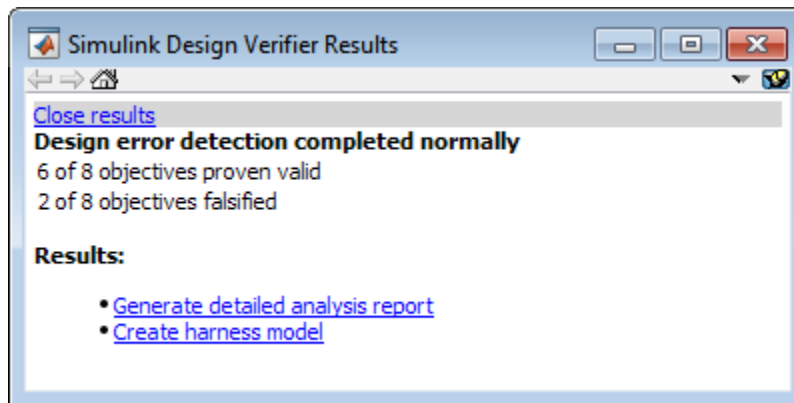
When you click an object in the model, additional details about the results for that object are displayed in the Simulink Design Verifier Results window.

Tip By default, the Simulink Design Verifier Results window is always the topmost visible window. To change that setting, click the 🌐 icon and on the context menu, clear the check mark next to **Always on top**.

Review the Latest Analysis Results in the Model Explorer

If you close the analysis results to fix the cause of the errors in your model, you might need to rereview the analysis results. As long as your model remains open, you can view the results of your most recent analysis results in the Model Explorer.

After you close your model, you can no longer view any analysis results.

To view the latest results, in the model window, select **Analysis** > **Design Verifier** > **Latest Results**. The Model Explorer opens with the results displayed on the right-hand pane.

For any Simulink Design Verifier analysis, from the Model Explorer, you can perform the following tasks:

- Highlight the analysis results on the model.
- Generate a detailed analysis report.
- Create the harness model, or if the harness model already exists, open it.

Note: If no objectives are falsified, you cannot create the harness model.

- View the data file.
- View the log file.

Check For Design Errors using the Model Advisor

You can perform design error detection analysis from the Model Advisor, which is particularly useful if you need to perform other model checks. To analyze your model from the Model Advisor, follow this high-level workflow:

1 Specify options that control how Simulink Design Verifier detects design errors in your model.

2 Open the Model Advisor.

3   From the system hierarchy, select the model or model component you want to analyze

4   Expand the design error detection analysis items. Look for Simulink Design Verifier under either **By Product** or **By Task**.

5   If you have not checked your model for compatibility, enable the compatibility check for Simulink Design Verifier.

6   Select the design error detection checks you want to run.

7   Run the selected checks.

8   Review the analysis results.

More About

- "Consulting the Model Advisor"

Dead Logic Detection

Abstract

Describes the two analysis modes for dead logic detection in Simulink Design Verifier.

In this section...
"Detect Dead Logic Only" on page 6-10
"Detect Dead and Active Logic" on page 6-11

Design error detection for dead logic in Simulink Design Verifier consists of two analysis options:

- Detection of dead logic only. If you select this option, Simulink Design Verifier does not report active logic or undecided objectives. If you select this option, Simulink Design Verifier analyzes your model without floating-point to rational number conversion approximation or while loop approximation. For more information about approximations in Simulink Design Verifier, see "Approximations" on page 2-16.

  This option is available in:

  - Model Advisor. See "Check For Design Errors using the Model Advisor" on page 6-8.
  - The Configuration Parameters dialog box.

- Detection of active logic. Active logic detection runs concurrently with dead logic detection. In rare cases, active logic detection can also find additional dead logic. This option is available in the Configuration Parameters dialog box.

Detect Dead Logic Only

To detect dead logic if you are not using the Model Advisor:

1 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**

2 In the Configuration Parameters dialog box, in the **Select** tree, under **Design Verifier**, select **Design Error Detection**

3 In the **Design Error Detection** pane, select **Dead logic**.

  Clear **Identify active logic** if it is selected.

4 Click **OK** to apply these settings and close the Configuration Parameters dialog box.

5 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Detect Design Errors**.

Detect Dead and Active Logic

1 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**

2 In the Configuration Parameters dialog box, in the **Select** tree, under **Design Verifier**, select **Design Error Detection**

3 In the **Design Error Detection** pane, select **Dead logic** and then **Identify active logic**.

4 Click **OK** to apply these settings and close the Configuration Parameters dialog box.

5 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Detect Design Errors**.

More About

• "Design Verifier Pane: Design Error Detection" on page 15-58

Detect Dead Logic in Example Model

Overview of Dead Logic Detection

During a design error detection analysis, Simulink Design Verifier can detect dead logic in your model. Logic is considered dead if it is proven through analysis to stay inactive during execution. Dead logic might be caused by a design or a requirement error.

1. In the Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane:

   - To detect dead logic, select **Dead Logic**.
   - To detect active logic, select **Identify active logic**.

2. Select **Analysis** > **Design Verifier** > **Detect Design Errors**.

For information on more detailed analysis that identifies both dead logic and active logic, see "Dead Logic Detection" on page 6-10.

Detect Dead Logic in Example Model

This example shows how to analyze the sldvdemo_fuelsys_logic_simple model for dead logic.

1. To open the example model, at the MATLAB command prompt, type:

   ```
   sldvdemo_fuelsys_logic_simple
   ```

2. Select **Analysis** > **Design Verifier** > **Options**.

3. In the Configuration Parameters dialog box, select **Design Verifier** > **Design Error Detection**.

4 On the **Design Error Detection** pane, select **Dead Logic** and **Identify active logic**.

When you select **Dead Logic**, the other **Design Error Detection** analysis options are inactive.

5 Close the Configuration Parameters dialog box.

6 Select **Analysis** > **Design Verifier** > **Detect Design Errors** > **Model**.

After the analysis:

- The software highlights the model with the analysis results.

- The Simulink Design Verifier Results dialog box opens and displays a summary of the analysis.

Review Analysis Results for Example Model

This example shows how to review the analysis results when you detect dead logic in the `sldvdemo_fuelsys_logic_simple` model.

1 To review the model analysis results, at the top level of the `sldvdemo_fuelsys_logic_simple` model, open the `control logic` Stateflow chart.

- Model objects with active logic are highlighted in green.

- Model objects with dead logic are highlighted in red.

2 In the `sldvdemo_fuelsys_logic_simple/control logic` window, click the `Sens_Failure_Counter.INC` transition in the `Speed_Sensor_Mode` state.
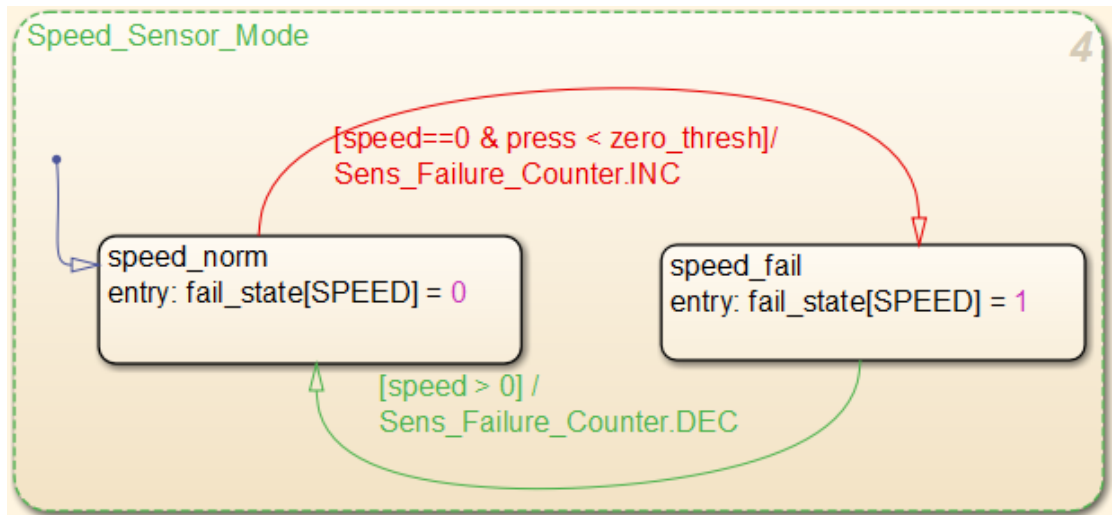
The Simulink Design Verifier Results dialog box displays a summary of the active and dead logic for the transition.
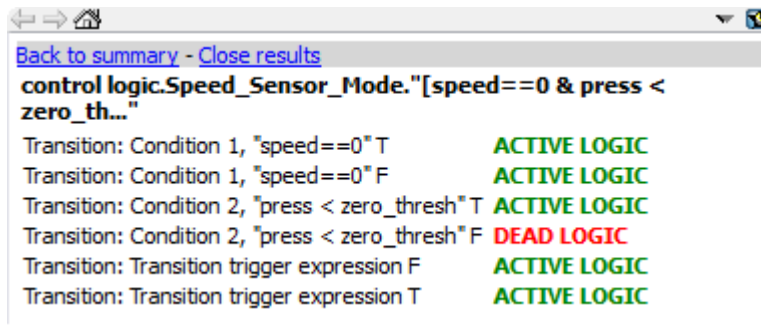
Review Analysis Report for Example Model

This example shows how to generate an analysis report after you detect dead logic in the `sldvdemo_fuelsys_logic_simple` model. The analysis report is an HTML file that provides detailed information about the analysis.

1   In the Simulink Design Verifier Results window, click **Back to summary**.

2 Click **Generate detailed analysis report**.

The software generates a detailed analysis report that opens in a browser.

3 In the analysis report, click **Design Error Detection Objectives Status**.

For `sldvdemo_fuelsys_logic_simple` model, the **Design Error Detection Objectives Status** chapter of the report provides detailed results in two categories:

- **Dead Logic** — Simulink Design Verifier proved the decision and condition outcomes listed in the table that cannot occur and are dead logic.

- **Active Logic** — Simulink Design Verifier found the decision and condition outcomes listed in the table that can occur and are active logic.

4 Investigate the analysis results for `Sens_Failure_Counter.INC`:

- In the analysis report, in the **Analysis Information** chapter, the **Constraints** section lists the constraints for `control logic/Input Data "press"`. The value of `press` is constrained to values between 0 and 2.

- In the Model Explorer window, select **Base Workspace** > **zero_thresh**. The value of `zero_thresh` is 250.

- With the design constraints, `press` is always less than `zero_thresh`. The logic `press < zero_thresh` is never false. The false condition for the transition is never exercised. Therefore, it is dead logic.

5 Close `sldvdemo_fuelsys_logic_simple`.

Model Objects That Receive Dead Logic Detection

Model objects that have decision or condition outcomes receive dead logic detection, as the following table shows. Click a link in the first column to get more detailed information about the outcomes for specific model objects.

Model Object Receiving Dead Logic Detection	Decision Outcomes	Condition Outcomes
"Abs" on page 6-17	●	
"Dead Zone" on page 6-17	●	
"Discrete-Time Integrator" on page 6-18	●	
"Enabled Subsystem" on page 6-18	●	●
"Enabled and Triggered Subsystem" on page 6-19	●	●
"Fcn" on page 6-19		●
"For Iterator, For Iterator Subsystem" on page 6-19	●	
"If, If Action Subsystem" on page 6-20	●	●
"Library-Linked Objects" on page 6-20	●	●
"Logical Operator" on page 6-20		●
"MATLAB Function" on page 6-20	●	●
"MinMax" on page 6-21	●	
"Model" on page 6-21	●	●
"Multiport Switch" on page 6-21	●	
"Rate Limiter" on page 6-21	●	
"Relay" on page 6-22	●	
"Saturation" on page 6-22	●	
"Stateflow Charts" on page 6-23	●	●
"Switch" on page 6-23	●	

Model Object Receiving Dead Logic Detection	Decision Outcomes	Condition Outcomes
"SwitchCase, SwitchCase Action Subsystem" on page 6-23	●	
"Triggered Models" on page 6-23	●	●
"Triggered Subsystem" on page 6-24	●	●
"While Iterator, While Iterator Subsystem" on page 6-24	●	

Abs

The Abs block has decision outcomes based on:

- Input to the block being less than zero.
- Data type of the input signal.

For input to the block being less than zero, there are two decision outcomes:

- Block input is less than zero, indicating a true decision.
- Block input is not less than zero, indicating a false decision.

If the input data type to the Abs block is `uint8`, `uint16`, or `uint32`, the software sets the block output equal to the block input without making a decision. If the input data type to the Abs block is Boolean, an error occurs.

Dead Zone

The Dead Zone block has decision outcomes based on these parameters:

- **Start of dead zone**
- **End of dead zone**

The **Start of dead zone** parameter specifies the lower limit of the dead zone. For the **Start of dead zone** parameter, there are two decision outcomes:

- Block input is greater than or equal to the lower limit, indicating a true decision.
- Block input is less than the lower limit, indicating a false decision.

The **End of dead zone** parameter specifies the upper limit of the dead zone. For the **End of dead zone** parameter, there are two decision outcomes:

- Block input is greater than the upper limit, indicating a true decision.
- Block input is less than or equal to the upper limit, indicating a false decision.

Discrete-Time Integrator

The Discrete-Time Integrator block has decision outcomes based on these parameters:

- **External reset**
- **Limit output**

If you set **External reset** to `none`, the software does not report decision outcomes. Otherwise, there are two decision outcomes:

- Block output is reset, indicating a true decision.
- Block output is not reset, indicating a false decision.

If you do not select **Limit output**, the software does not report decision outcomes. Otherwise, the software reports decision outcomes for the **Lower saturation limit** and the **Upper saturation limit**.

For the **Upper saturation limit**, there are two decision outcomes:

- Integration result is greater than or equal to the upper limit, indicating a true decision.
- Integration result is less than the upper limit, indicating a false decision.

For the **Lower saturation limit**, there are two decision outcomes:

- Integration result is less than or equal to the lower limit, indicating a true decision.
- Integration result is greater than the lower limit, indicating a false decision.

Enabled Subsystem

The Enabled Subsystem block has two decision outcomes:

- Block is enabled, indicating a true decision.
- Block is disabled, indicating a false decision.

The Enabled Subsystem block has two condition outcomes only if the enable input is a vector:

- Element of the enable input is true, indicating a true condition.
- Element of the enable input is false, indicating a false condition.

Enabled and Triggered Subsystem

The Enabled and Triggered Subsystem block has two decision outcomes:

- Trigger edge occurs while the block is enabled, indicating a true decision.
- Trigger edge does not occur while the block is enabled, or the block is disabled, indicating a false decision.

The software determines condition outcomes for the enable input and the trigger input separately.

- For the enable input:
  - Input is true, indicating a true condition.
  - Input is false, indicating a false condition.
- For the trigger input:
  - Trigger edge occurs, indicating a true condition.
  - Trigger edge does not occur, indicating a false condition.

Fcn

The Fcn block has two condition outcomes based on input values or arithmetic expressions that are inputs to Boolean operators in the block:

- Input to a Boolean operator is true, indicating a true condition.
- Input to a Boolean operator is false, indicating a false condition.

For Iterator, For Iterator Subsystem

The For Iterator block and For Iterator Subsystem have two decision outcomes:

- Iteration value being at or below the iteration limit, indicated as true.

• Iteration value being above the iteration limit, indicated as false.

If, If Action Subsystem

The If blocks that causes an If Action Subsystem to execute has:

• Decision outcomes for the `if` condition and all `elseif` conditions defined in the If block.
• Condition outcomes if the `if` condition or any of the `elseif` conditions contains a logical expression with multiple conditions.

Library-Linked Objects

Simulink blocks and Stateflow charts that are linked to library objects receive the same dead logic detection that they would receive if they were not linked to library objects.

Logical Operator

The Logical Operator block has two condition outcomes:

• Input is true, indicating a true condition.
• Input is false, indicating a false condition.

MATLAB Function

The following MATLAB Function block statements have decision outcomes:

• Function header - Function or sub-function that is executed.
• `if` - Expression evaluates to true, indicating a true decision. Expression evaluates to false, indicating a false decision.
• `switch` - Decision outcomes corresponding to every switch case path, including the fall-through case.
• `for` - Loop condition evaluates to true, indicating a true decision. Loop condition evaluates to false, indicating a false decision.
• `while` - Loop condition evaluates to true, indicating a true decision. Loop condition evaluates to false, indicating a false decision.

The following logical conditions have condition outcomes:

- `if` statement conditions
- `while` statement conditions

MinMax

The MinMax block has decision outcomes based on passing each input to the output of the block.

For passing each input to the output of the block, there are two decision outcomes:

- Input passed to block output, indicating a true decision.
- Input not passed to block output, indicating a false decision.

Model

The Model block itself does not have decision or condition outcomes. The model that the block references receive the decision or condition outcomes.

Multiport Switch

The Multiport Switch block has decision outcomes based on passing each input, excluding the first control input, to the output of the block.

For passing each input, excluding the first control input, to the output of the block, there are two decision outcomes:

- Input passed to block output, indicating a true decision.
- Input not passed to block output, indicating a false decision.

Rate Limiter

The Rate Limiter block has decision outcomes based on the **Rising slew rate** and **Falling slew rate** parameters.

For the **Rising slew rate**, there are two decision outcomes:

- Block input changes more than or equal to the rising rate, indicating a true decision.
- Block input changes less than the rising rate, indicating a false decision.

For the **Falling slew rate**, there are two decision outcomes:

- Block input changes less than or equal to the falling rate, indicating a true decision.
- Block input changes more than the falling rate, indicating a false decision.

The software does not have **Falling slew rate** outcomes for a time step when the **Rising slew rate** is true.

Relay

The Relay block has decision outcomes based on the **Switch on point** and the **Switch off point** parameters.

For the **Switch on point**, there are two decision outcomes:

- Block input is greater than or equal to the **Switch on point**, indicating a true decision.
- Block input is less than the **Switch on point**, indicating a false decision.

For the **Switch off point**, there are two decision outcomes:

- Block input is less than or equal to the **Switch off point**, indicating a true decision.
- Block input is greater than the **Switch off point**, indicating a false decision.

The software does not have **Switch off point** decision outcomes for a time step when the switch on threshold is true.

Saturation

The Saturation block has decision outcomes based on the **Lower limit** and **Upper limit** parameters.

For the **Upper limit**, there are two decision outcomes:

- Block input is greater than or equal to the upper limit, indicating a true decision.
- Block input is less than the upper limit, indicating a false decision.

For the **Lower limit**, there are two decision outcomes:

- Block input is greater than the lower limit, indicating a true decision.

•   Block input is less than or equal to the lower limit, indicating a false decision.

The software does not have **Lower limit** decision outcomes for a time step when the upper limit is true.

Stateflow Charts

The Stateflow Chart block has decision outcomes:

•   Transition decision is evaluated as true, indicating a true decision.
•   Transition decision is evaluated as false, indicating a false decision.

The Stateflow Chart block has condition outcomes:

•   Condition is evaluated as true, indicating a true condition.
•   Condition is evaluated as false, indicating a false condition.

Switch

The Switch block has decision outcomes based on the control input to the block.

For the control input to the block, there are two decision outcomes:

•   Control input evaluates to true, indicating a true decision.
•   Control input evaluates to false, indicating a false decision.

SwitchCase, SwitchCase Action Subsystem

The SwitchCase block and SwitchCase Action Subsystem have two decision outcomes:

•   Block evaluates to true, indicating a true decision.
•   Block does not evaluate to true, indicating a false decision.

Triggered Models

The Triggered Models block has two decision outcomes:

•   Referenced model is triggered, indicating a true decision.
•   Referenced model is not triggered, indicating a false decision.

If the trigger input is a vector, then there are two condition outcomes:

- Element of the trigger port is true, indicating a true condition.
- Element of the trigger port is false, indicating a false condition.

Triggered Subsystem

The Triggered Subsystem block has two decision outcomes:

- Block is triggered, indicating a true decision.
- Block is not triggered, indicating a false decision.

If the trigger input is a vector, then there are two condition outcomes:

- Element of the trigger edge is true, indicating a true condition.
- Element of the trigger edged is false, indicating a false condition.

While Iterator, While Iterator Subsystem

The While Iterator block and While Iterator Subsystem have two decision outcomes:

- `while` condition is satisfied, indicating a true decision.
- `while` condition is not satisfied, indicating a false decision.

Detect Integer Overflow and Division-by-Zero Errors

In this section...

About This Example

The following sections describe how to analyze the `sldvdemo_cruise_control_fxp_fixed` model for integer overflow and division-by-zero errors.

Analyze the Model

Open and check model for integer overflow and division-by-zero errors:

1   Open the `sldvdemo_cruise_control_fxp_fixed` model.

2   Select **Analysis** > **Design Verifier** > **Options**.

3   In the Configuration Parameters dialog box, in the **Select** tree under **Design Verifier**, select the **Design Error Detection** node.

4   On the **Design Error Detection** pane, select:

   · **Integer overflow**

   · **Division by zero**

5   In the Configuration Parameters dialog box, on the **Diagnostics** > **Data Validity** pane, set **Signals** > **Wrap on overflow**, **Signals** > **Saturate on overflow** and **Parameters** > **Detect overflow** to `error`.

6   Click **OK** to save these settings and close the Configuration Parameters dialog box.

7   Select **Analysis** > **Design Verifier** > **Detect Design Errors** > **Model**.

When the analysis is complete:

·   The software highlights the model with the analysis results.

- The Simulink Design Verifier Results dialog box opens and displays a summary of the analysis.

Review the Analysis Results

Review the Results on the Model

The derived ranges can help you understand the source of an error by identifying the possible signal values, as you can see by taking the following steps:

1 At the top level of the `sldvdemo_cruise_control_fxp_fixed` model, click the Fixed-Point Controller subsystem.

The Simulink Design Verifier Results window displays the derived range of possible signal values for the Outports, as calculated by the analysis:

- The values of Outport 1 (throt) range from –2.6101 to 2.6096.
- The values of Outport 2 (target) range from 0 to 256.
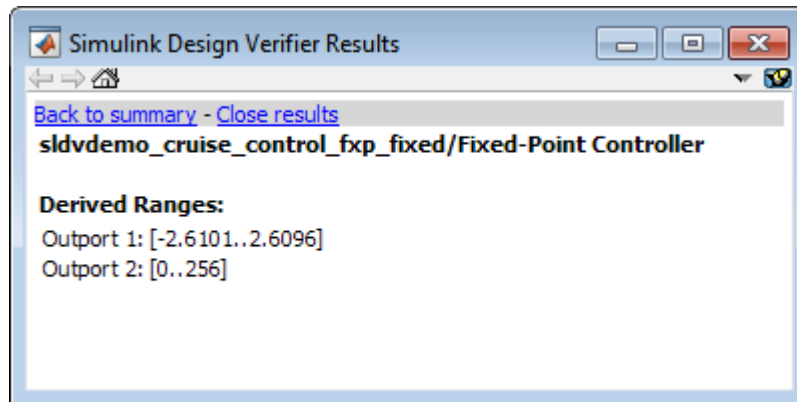
2 Click the Outport blocks of the `sldvdemo_cruise_control_fxp_fixed` model to see the same signal bound values.

3 Open the Fixed-Point Controller subsystem.

Two objects in this subsystem are outlined in red. The PI Controller subsystem is outlined in green.

4 Click the Sum block, outlined in red, that provides the error input to the PI Controller subsystem.
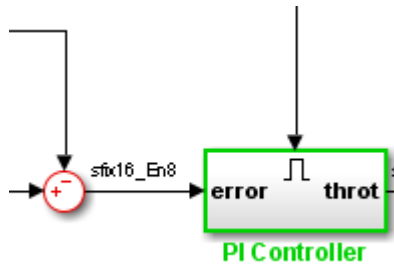
This Sum block can produce an overflow error. The analysis found a test case that can result in a computation where the output of the Sum block exceeds the range [−128..128].
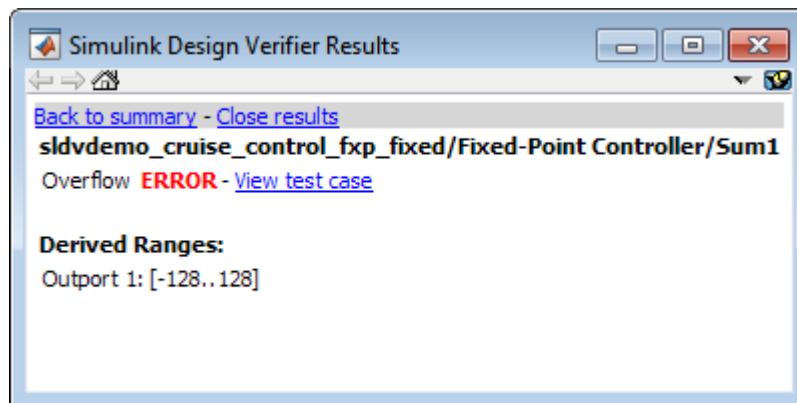
5 To more fully understand this error, click the two blocks that provide the inputs to the Sum block. In the Simulink Design Verifier Results window, view their derived ranges:

- The third Outport from the Bus block has a range of [0..256].
- The Outport from the Switch block has a range of [0..256].

You can see that the sum operation for these signal ranges can compute a value that exceeds the range [–128..128] for the Outport of the Sum block.

The analysis reports the overflow error on the Sum block. The analysis does not propagate this error and assumes that the Sum block output is within the valid range for any subsequent computations.

6   Click the PI Controller subsystem, outlined in green. None of the blocks in the PI Controller subsystem can produce overflow or division-by-zero errors. When the software analyzes the PI Controller subsystem, it ignores the overflow error from the Sum block and assumes that the inputs to the subsystem are valid.

Keep the `sldvdemo_cruise_control_fxp_fixed` model open. In the next section, you create the harness model to see the test case that generates the Sum block overflow error.

Review the Harness Model

To see the test cases that demonstrate the errors, generate the harness model from the Simulink Design Verifier Results window:

1   In the `sldvdemo_cruise_control_fxp_fixed` model, open the Fixed-Point Controller subsystem.

2   Click the Sum block, outlined in red, that provides the error input to the PI Controller subsystem.

The Simulink Design Verifier Results window displays information that an overflow error occurred.

3   In the Simulink Design Verifier Results window, click **View test case**.

The software creates a harness model containing the test case with the signal values that cause this overflow error.

In the harness model, the Signal Builder dialog box opens, with Test Case 2 displayed.

4   Click the Start simulation button to simulate the model with this test case.

As expected, the simulation fails due to an overflow error at the Sum block in the Fixed-Point Controller subsystem.

For more information, see "Simulink Design Verifier Harness Models" on page 13-16.

Review the Analysis Report

To view an HTML report containing detailed information about the analysis report for the `sldvdemo_cruise_control_fxp_fixed` model:

1   In the Simulink Design Verifier Results window, to redisplay the results summary, click **Back to summary**.

2   Click **Generate detailed analysis report**.

    The software generates a detailed analysis report that opens in a browser.

For the `sldvdemo_cruise_control_fxp_fixed` model, the **Design Error Detection Objectives Status** chapter of the report provides detailed results in two categories:

•   **Objectives Proven Valid** — Model objects that did not produce errors

•   **Objectives Falsified with Test Cases** — Model objects for which test cases generated errors

For more information, see "Simulink Design Verifier Reports" on page 13-27.

Check for Specified Intermediate Minimum and Maximum Signal Values

In this section...
"Overview of Specified Minimum and Maximum Signal Values" on page 6-30
"About This Example" on page 6-31
"Create the Example Model" on page 6-31
"Analyze the Model" on page 6-33
"Review the Analysis Results" on page 6-33

Overview of Specified Minimum and Maximum Signal Values

During a design error detection analysis, the software checks the specified minimum and maximum values on intermediate signals throughout the model and on the output ports. These values define the *design ranges*.

The analysis checks for specified minimum and maximum values on:

- Simulink block outputs, with the exception of the limitations described in the next section
- `Simulink.Signal` objects
- Stateflow data objects
- MATLAB for code generation data objects
- Global data store writes

If the analysis detects that a signal exceeds the design range, the results identify where in the model the errors occurred. In addition, you can generate a harness model that contains test cases that demonstrate how the error occurred.

Limitations of Checking Specified Minimum and Maximum Signal Values

If you analyze a model checking if specified minimum and maximum values are exceeded, the software cannot check minimum and maximum values specified on:

- `Simulink.BusElement` objects.

- Any Mux block with an output connected to a Selector block
- Merge block inputs

  To work around this limitation, use a `Simulink.Signal` object on the Merge block output and specify the range on the `Simulink.Signal` object.

Note: For information about how a Simulink Design Verifier analysis handles specified minimum and maximum values on input ports, see "Minimum and Maximum Input Constraints" on page 11-2.

About This Example

In this section, you create and analyze a model that has specified design minimum and maximum values on:

- The input ports
- The output ports of two of the intermediate blocks

The design error detection analysis identifies blocks where the output values exceed the design range. If the analysis detects this error, this example demonstrates how the analysis uses the specified minimum and maximum values when continuing the analysis.

Create the Example Model

Create the model for this example:

1 In the model window, select **File** > **New** > **Model**.
2 From the Simulink Commonly Used Blocks library, add the following blocks to the model and assign the indicated parameter values.

Block	Tab	Parameter	Value
Inport	**Signal Attributes**	**Minimum**	0
Inport	**Signal Attributes**	**Maximum**	5
Gain	**Main**	**Gain**	5
Gain	**Signal Attributes**	**Output minimum**	0

Block	Tab	Parameter	Value
Gain	**Signal Attributes**	**Output maximum**	20
Gain	**Signal Attributes**	**Output data type**	`int16`
Saturation	**Main**	**Upper limit**	25
Saturation	**Main**	**Lower limit**	-25
Saturation	**Signal Attributes**	**Output minimum**	-25
Saturation	**Signal Attributes**	**Output maximum**	25
Outport	No changes		

3 Connect the four blocks as shown.

4 To display the specified minimum and maximum values in the model window, select **Display** > **Signals & Ports** > **Design Ranges**.

5 Select **Analysis** > **Design Verifier** > **Options**.

6 In the Configuration Parameters dialog box, on the **Solver** pane, under **Solver options**:

    a Set **Type** to `Fixed-step`.

       The Simulink Design Verifier software does not support variable-step solvers.

    b Set **Solver** to `discrete (no continuous states)`.

7 On the **Design Verifier** pane, set **Mode** to `Design error detection`.

8 On the **Design Verifier** > **Design Error Detection** pane:

    a Select **Check specified intermediate minimum and maximum values**.

    b Clear the **Integer overflow** and **Division by zero** parameters.

    In this example, you check only for intermediate minimum and maximum violations.

9 To save these settings and exit the Configuration Parameters dialog box, click **OK**.

10 Save the model and name it `ex_interim_minmax`.

Analyze the Model

To analyze the example model to identify any intermediate signals that violate the specified minimum and maximum values, select **Analysis** > **Design Verifier** > **Detect Design Errors** > **Model**.

After the analysis is complete:

- The software highlights the model with the analysis results.
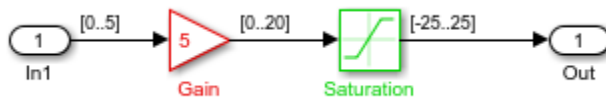
- The Simulink Design Verifier Results dialog box opens and displays a summary of the analysis.
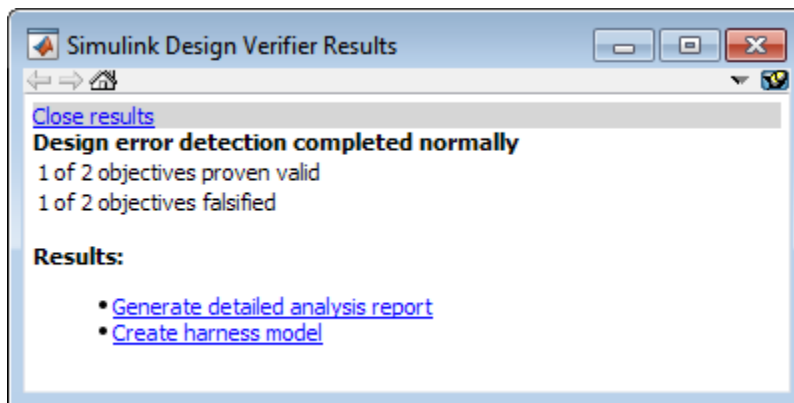
Review the Analysis Results

- "Review Results on the Model" on page 6-34
- "Review the Harness Model" on page 6-35
- "Review the Analysis Report" on page 6-36

Review Results on the Model

In the model window, the Gain block is colored red and the Saturation block is colored green. This indicates that:

- At least one objective associated with the Gain block was falsified. For this example, the analysis falsified exactly one objective.

- All objectives associated with the Saturation block were satisfied. For this example, the analysis satisfied exactly one objective.

To understand these results:

1 Click the Gain block.

The Simulink Design Verifier Results window shows that the design range for the output was [0..20], but the analysis detected an error and generated a test case that demonstrates that error. Because the design range for the input block is [0..5], when the input to the Gain block is 5, the output is 25, which exceeds the specified maximum value on that port.

The analysis computes and displays the derived range to help you understand how the design range was exceeded.
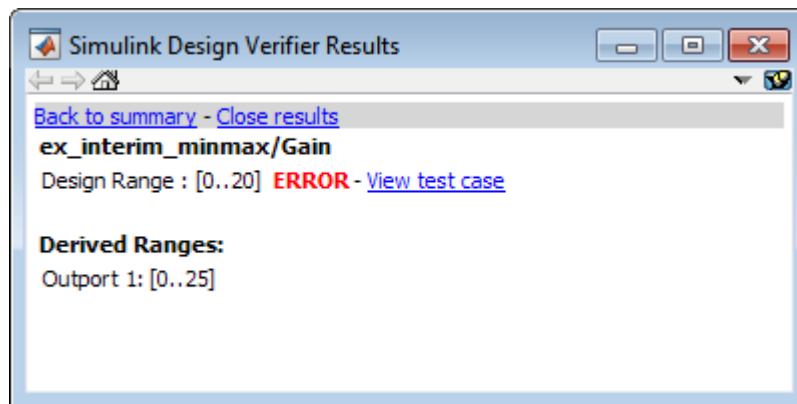
When the software detects an error, the analysis proceeds, but constrains the signal to the tightest intersection of the design range and the derived range, which for the Gain block output in this example is [0..20].

2 Click the Saturation block.

The Simulink Design Verifier Results window shows that the output of the Saturation block never exceeded the design range [–25..25]. The input to the Saturation block never exceeded [0..20], which is the derived range that the analysis propagated from the Gain block.
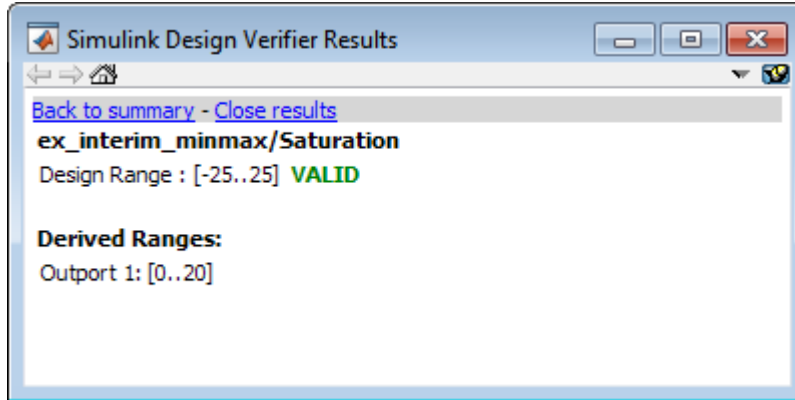
Review the Harness Model

When the analysis completes, you can create a harness model contains the test cases that result in errors.

For the example model, view the test case that caused the design range error in the Gain block:

1 After the analysis completes and the model is highlighted, click the Gain block.

2 In the Simulink Design Verifier Results window, click **View test case**.

   The software creates a harness model named ex_interim_minmax_harness and opens the Signal Builder block in the harness model that contains the test case.

   In the Signal Builder block, one test case, whose signal value is 5, caused the output of the Gain block to be 25, which exceeds the specified maximum of 20.

3 Before you simulate this test case, in the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane, set **Simulation range checking** to warning or error.

Setting this parameter specifies the diagnostic action to take if Simulink detects signals that exceed specified minimum or maximum values during simulation.

- If you specify `warning`, the simulation displays a warning message and continues.
- If you specify `error`, the simulation displays an error message and stops.

4 Click **OK** to save your change and close the Configuration Parameters dialog box.

5 In the Signal Builder block window, click **Start simulation** to simulate the model with this test case.

As expected, in the MATLAB window, the simulation displays a warning or error that the output value of the Gain block exceeds the specified maximum.

Review the Analysis Report

You can also generate an HTML report containing detailed information about the analysis report for the `ex_interim_minmax` model. To create this report, in the Simulink Design Verifier Results window, click **Generate detailed analysis report**. The analysis report opens in a browser.

In the analysis report, the **Design Error Detection Objectives Status** chapter of the report provides detailed results in two categories:

- **Objectives Proven Valid** — The output values for the Saturation block are always within the design range.
- **Objectives Falsified with Test Cases** — The output values for the Gain block violated the design range.

Detect Out of Bound Array Access Errors

In this section...
"Design Error Detection for Out of Bound Array Access" on page 6-37
"Detect Out of Bound Array Access in Example Model" on page 6-38
"Limitations of Support for Out of Bound Array Access Design Error Detection" on page 6-43

Design Error Detection for Out of Bound Array Access

Simulink Design Verifier design error detection analysis detects out of bound array access errors in your model. In simulation, when your model attempts to access an array element using an invalid index, an out of bound array access error occurs.

To detect out of bound array access errors in your model:

1 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**.

   The Configuration Parameters dialog box opens to the **Design Verifier** pane.

2 Under **Analysis options**, from the **Mode** list, select Design error detection.

3 In the **Select** tree, under **Design Verifier**, select **Design Error Detection**.

4 Select **Out of bound array access**.

5 Click **OK**.

6 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Detect Design Errors** > **Model**.

   The Simulink Design Verifier log window opens, showing the progress of the analysis.

   When the analysis is complete:

   • The software highlights the model with the analysis results.

   • The Simulink Design Verifier Results dialog box opens and displays an analysis summary.

Detect Out of Bound Array Access in Example Model

This example shows how to detect out of bound array access errors in the `sldvdemo_array_bounds` example model.

1 At the MATLAB command prompt, type:

    sldvdemo_array_bounds

The example model opens.

Simulink Design Verifier
Design Error Detection for Out of Bound Array Access
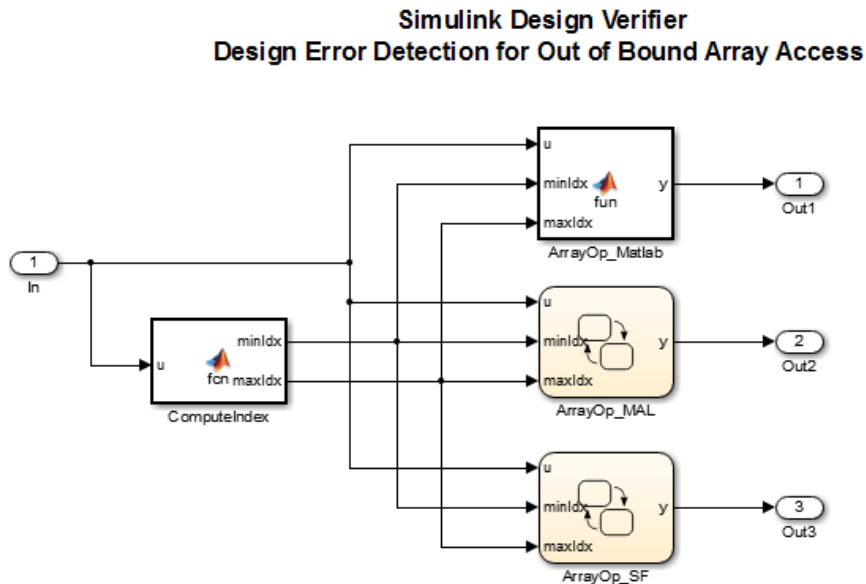
This example shows you how to statically detect out of bound array errors using Simulink Design Verifier.

This model contains errors that result from using 1-based indices in a 0-based Stateflow array.

Run (double-click)
Run Simulink Design Verifier

View Options (double-click)
View Simulink Design Verifier Options

More Information (double-click)
View Documentation

Using input signal values, the ComputeIndex MATLAB Function block determines a range of indices with minimum `minIdx` and maximum `maxIdx`. The ArrayOp_Matlab, ArrayOp_MAL, and ArrayOp_SF blocks use the set of integer indices between `minIdx` and `maxIdx` to access array elements and perform array operations.

In this example model, the analysis options are configured for out of bound array access error detection. To view these options, double-click the **View Simulink Design Verifier Options** button.

2   Start the design error detection analysis by double-clicking the **Run Simulink Design Verifier** button.

The Simulink Design Verifier log window opens, displaying the progress of the analysis.

When the analysis is complete, the example model is highlighted with the analysis results.
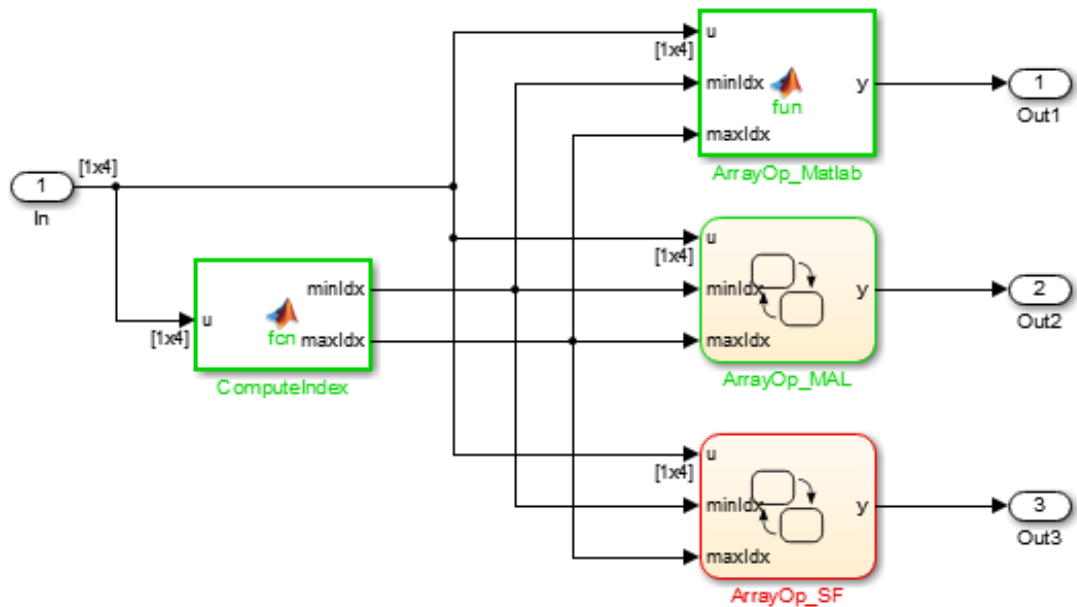
3   View the analysis results inside the chart by double-clicking the ArrayOp_SF Chart
block, highlighted in red.
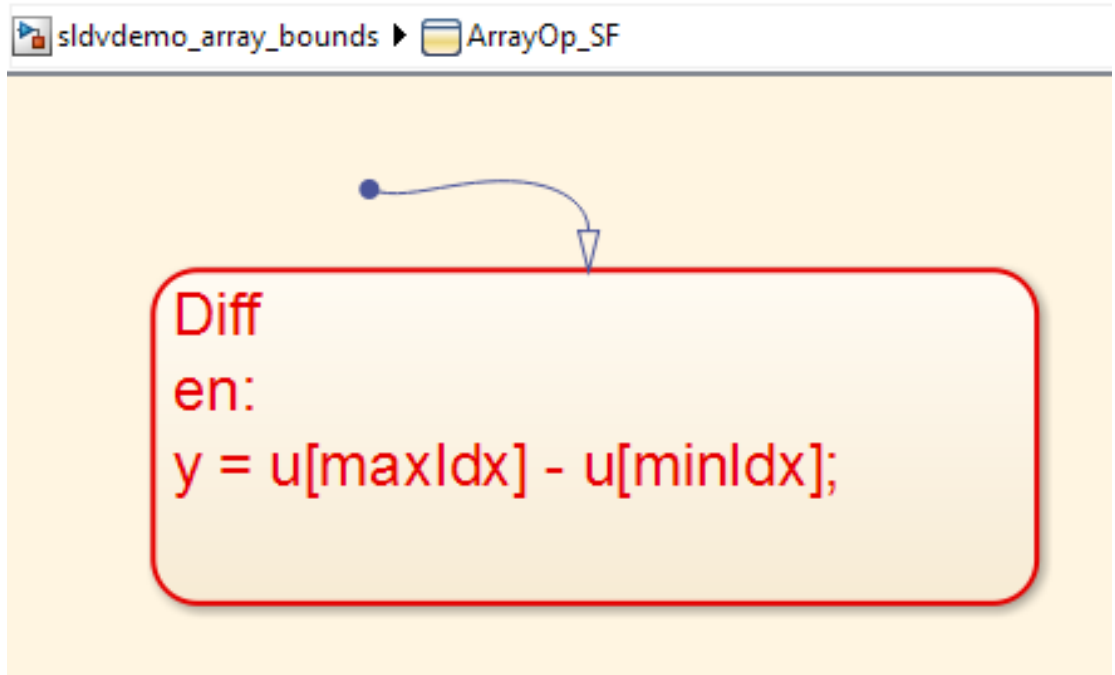
4   See detailed analysis results for the Diff state in the Simulink Design Verifier
Results window by selecting the Diff state. That state is highlighted in red.
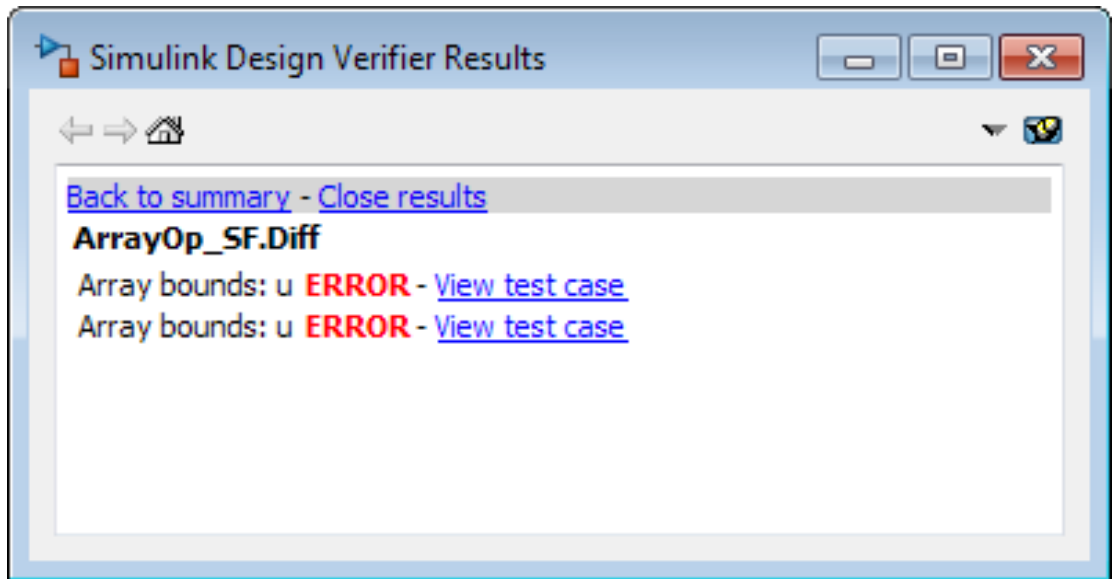
Simulink Design Verifier detected index out of bound errors for array u in state Diff.

5   Click the first **View test case** link.

Simulink Design Verifier creates and opens a harness model that contains test cases, or input signal groups, that demonstrate out of bound array access errors.

6   In the Signal Builder dialog box, click **Start simulation** to simulate the harness model with Test Case 1.

The simulation stops just before entering the state Diff. The Stateflow Debugger opens. The following error is shown:

```
Runtime error: Index into array out of range
Model Name: sldvdemo_array_bounds_harness
Block Name: sldvdemo_array_bounds_harness/...
  Test Unit (copied from sldvdemo_array_bounds)/ArrayOp_SF
Attempted to access 4 element of data u(#188 (0:3:0))...
  The valid index range is 0 to 3
```

Keep the Stateflow Debugger open at this breakpoint.

7 In the `sldvdemo_array_bounds_harness` model, hold your cursor over the Diff state to see the data values at this simulation breakpoint.
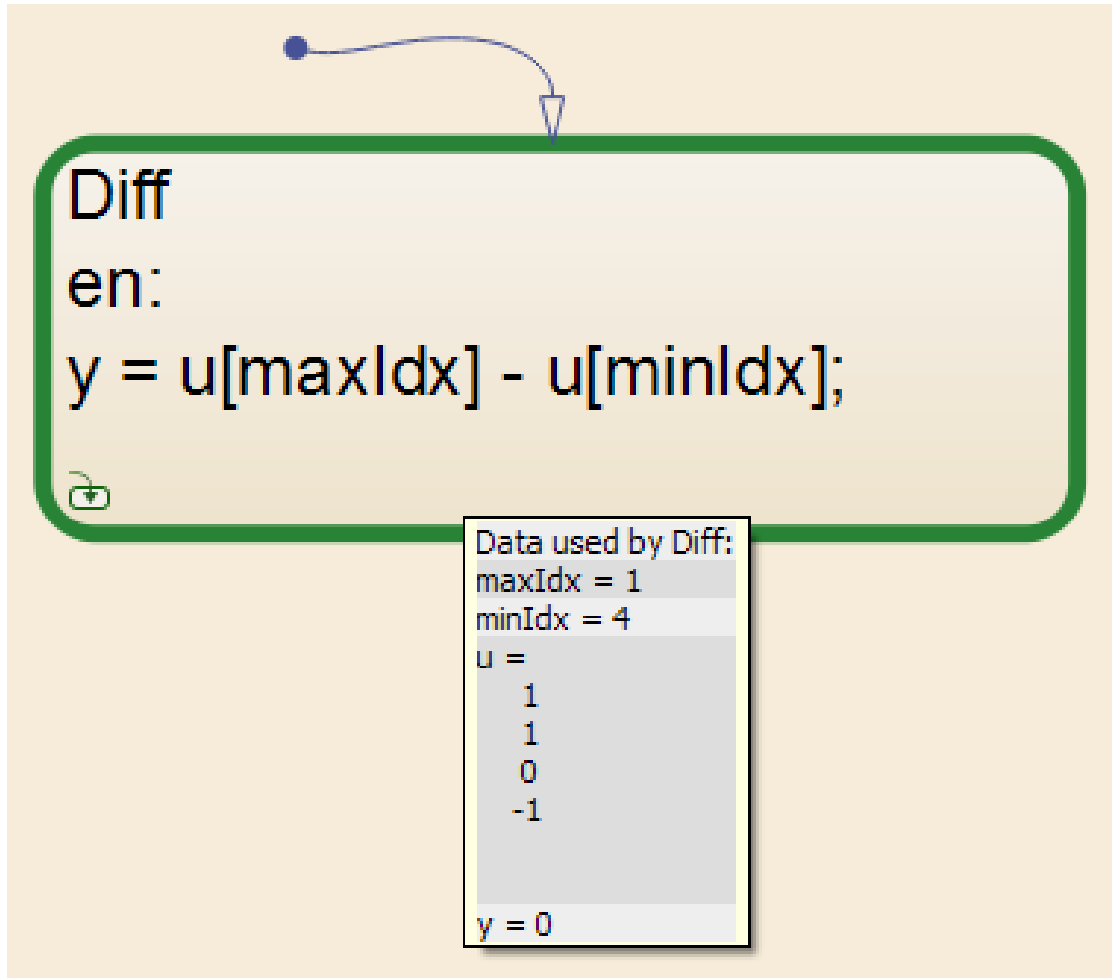
Using Test Case 1 input signal values, the ComputeIndex MATLAB Function block determines the range of array indices to be `1:4`. One-based indexing is consistent with MATLAB syntax, so these indices are valid for the ArrayOp_Matlab MATLAB Function block and the ArrayOp_MAL Stateflow chart.

The ArrayOp_SF Stateflow chart uses C as the action language, which does not support one-based indexing. `1:4` is not a valid index range for array access in that chart. The valid index range for array access in that chart is `0:3`, as the error message reported. When either `maxIdx` or `minIdx` evaluates to `4`, an out of bound array access error occurs in the ArrayOp_SF Chart block.

For more information on zero-based indexing support, see "Differences Between MATLAB and C as Action Language Syntax".

Limitations of Support for Out of Bound Array Access Design Error Detection

`Inf` Index Values

Design error detection does not support indexing by `Inf`. If your model attempts to access an array using an index value that evaluates to `Inf`, design error detection does not report an out of bound array access error, but in simulation, an out of bound array access error occurs.

Index Vector Block with Scalar Data Input

Out of bound array access design error detection does not support `Index Vector` blocks with scalar data inputs. If your model includes an Index Vector block that specifies a scalar data input instead of a vector data input and the control input causes an out of bounds array access, design error detection does not report an error, but an error occurs in simulation.

7

Generating Test Cases

What Is Test Case Generation?

The Simulink Design Verifier software can generate test cases that satisfy coverage objectives for your model, including:

- "Decision" on page 7-27
- "Condition" on page 7-27
- "MCDC" on page 7-28

Test cases help you confirm model performance by demonstrating how the blocks in the model execute in different modes. When generating test cases, the software performs a formal analysis of your model. After completing the analysis, the software provides several ways for you to review the results.

Test Case Blocks

For customizing test cases for your Simulink models, Simulink Design Verifier provides two blocks:

- The `Test Objective` block defines the values of a signal that a test case must satisfy.
- The `Test Condition` block constrains the values of a signal during analysis.

Test Case Functions

To customize test cases for a Simulink model or Stateflow chart, Simulink Design Verifier provides two MATLAB functions. You can use these functions in a MATLAB Function block. Both functions are active in generated code and in Simulink Design Verifier.

- `sldv.test` — Specifies a test objective.
- `sldv.condition` — Specifies a test condition.

These functions:

- Identify mathematical relationships for testing in a form that can be more natural than using block parameters.
- Support specifying multiple objectives, assumptions, or conditions without complicating the model.

- Provide access to the power of MATLAB.
- Support separation of verification and model design.

For an example of how to use these functions, see the `sldv.test` or `sldv.condition` reference page.

Note: Simulink Design Verifier blocks and functions are saved with a model. If you open the model on a MATLAB installation that does not have a Simulink Design Verifier license, you can see the blocks and functions, but they do not produce results.

Workflow for Test Case Generation

To generate test cases for your model, use the following workflow.

Task	Description	For an example, see
1	Verify that your model is compatible for use with Simulink Design Verifier.	"Check Compatibility of the Example Model" on page 7-6
2	Optionally, use the Test Generation Advisor to select model components (atomic subsystems and model blocks) for test generation. Before test generation, you can use the results to better understand your model, particularly large models, complex models, or models for which you are uncertain of the test generation compatibility.	"Use Test Generation Advisor to Identify Analyzable Components" on page 7-21
3	If you have Stateflow objects in your model, in the Configuration Parameters dialog box, on the **Diagnostics** > **Stateflow** pane, set **Unreachable execution path** to `error`.	
4	Optionally, instrument your model with blocks or MATLAB functions that specify test objectives and test conditions.	"Customize Test Generation" on page 7-17
5	Specify options that control how Simulink Design Verifier generates test cases for your model.	"Configure Test Generation Options" on page 7-7
6	Execute the Simulink Design Verifier analysis.	"Analyze the Example Model" on page 7-8 and "Reanalyze the Example Model" on page 7-19
7	Review the analysis results.	"Review Analysis Results" on page 7-8

Generate Test Cases for Model Decision Coverage

Construct the Example Model

Construct a model for this example:

1   Create a Simulink model.

2   Copy the following blocks into your empty model window:

   - From the Sources library, an `Inport` block to initiate the input signal whose value Simulink Design Verifier controls.

   - From the Sources library, two `Constant` blocks to serve as Switch block data inputs.

   - From the Signal Routing library, a `Switch` block to provide simple logic.

   - From the Sinks library, an `Outport` block to receive the output signal.

3   In your model, double-click one of the Constant blocks and specify its **Constant value** parameter as 2.

4   Connect the blocks so that your model appears similar to the following diagram.
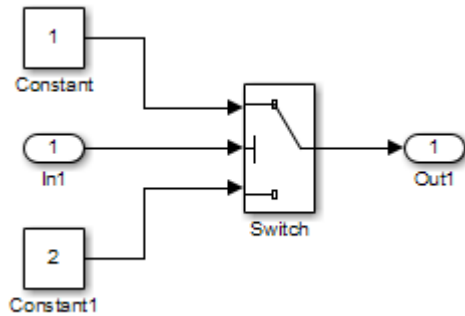
5. In the model window, select **Simulation** > **Model Configuration Parameters**.

6. On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Solver** category. On the right side, under **Solver options**:

   - Set the **Type** option to `Fixed-step`.

   - Set the **Solver** option to `Discrete (no continuous states)`.

   Simulink Design Verifier analyzes only models that use a fixed-step solver.

7. Click **OK** to save your changes and close the Configuration Parameters dialog box.

8. Save your model with the name `ex_generate_test_cases_example`.

Check Compatibility of the Example Model

Every time Simulink Design Verifier analyzes a model, before the analysis begins, the software performs a compatibility check. If your model is not compatible, the software cannot analyze it.

Before you start the analysis, you can also make sure that your model is compatible with Simulink Design Verifier software:

1. Open the `ex_generate_test_cases_example` model.

2. In the model window, select **Analysis** > **Design Verifier** > **Check Compatibility** > **Model**.

   The software displays the log window, which states whether or not your model is compatible for analysis.

The model you just created is compatible.
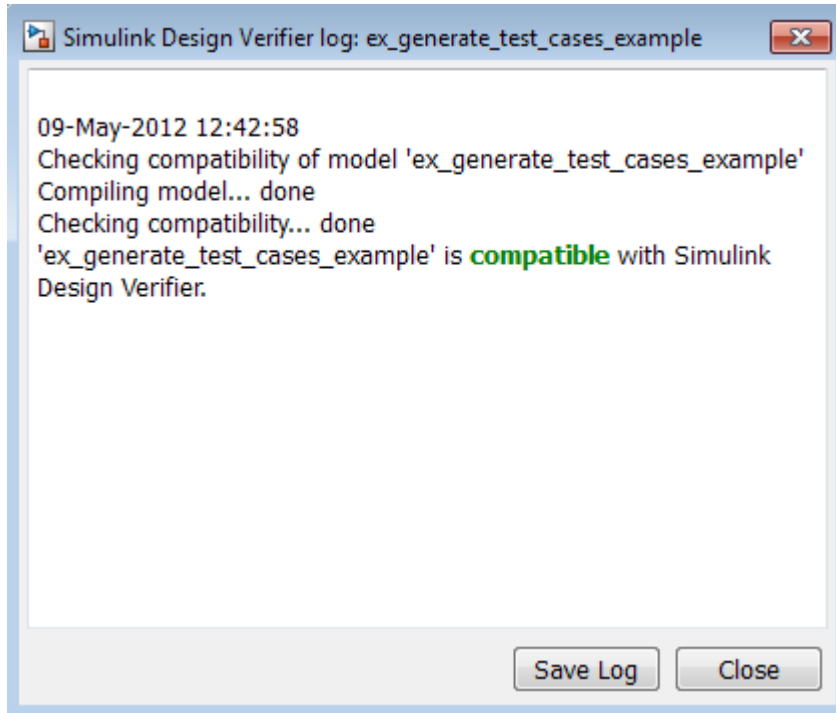
What If a Model Is Partially Compatible?

If the compatibility check indicates that your model is partially compatible, your model contains at least one object that Simulink Design Verifier does not support. You can analyze a partially compatible model, but, by default, the unsupported objects are stubbed out. The results of the analysis can be incomplete.

For detailed information about automatic stubbing, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

Configure Test Generation Options

Configure Simulink Design Verifier to generate test cases that achieve 100% decision coverage for the `ex_generate_test_cases_example` model:

1. Open the `ex_generate_test_cases_example` model.

2. In the model window, select **Analysis** > **Design Verifier** > **Options**.

3. On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Design Verifier** category. Under **Analysis options**, set the **Mode** option to `Test generation`.

4. On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Test Generation** category.

5. On the **Test Generation** pane, set the **Model coverage objectives** parameter to `Decision`.

   For this example, the analysis generates test cases that record only decision coverage.

   The **Test suite optimization** parameter is set by default to `CombinedObjectives`. If you want to generate fewer but longer test cases, select `LongTestcases` for the **Test suite optimization** parameter.

6. Click **OK** to save your changes and close the Configuration Parameters dialog box.

7. Save the `ex_generate_test_cases_example` model.

Analyze the Example Model

To analyze the `ex_generate_test_cases_example` model, in the model window, select **Analysis** > **Design Verifier** > **Generate Tests** > **Model**. The Simulink Design Verifier software begins analyzing your model to generate test cases.

During the analysis, the log window shows the progress of the analysis. It displays information such as the number of test objectives processed and which objectives are satisfied.

Review Analysis Results

When the software completes its analysis, the log window displays the following options for reviewing the results.
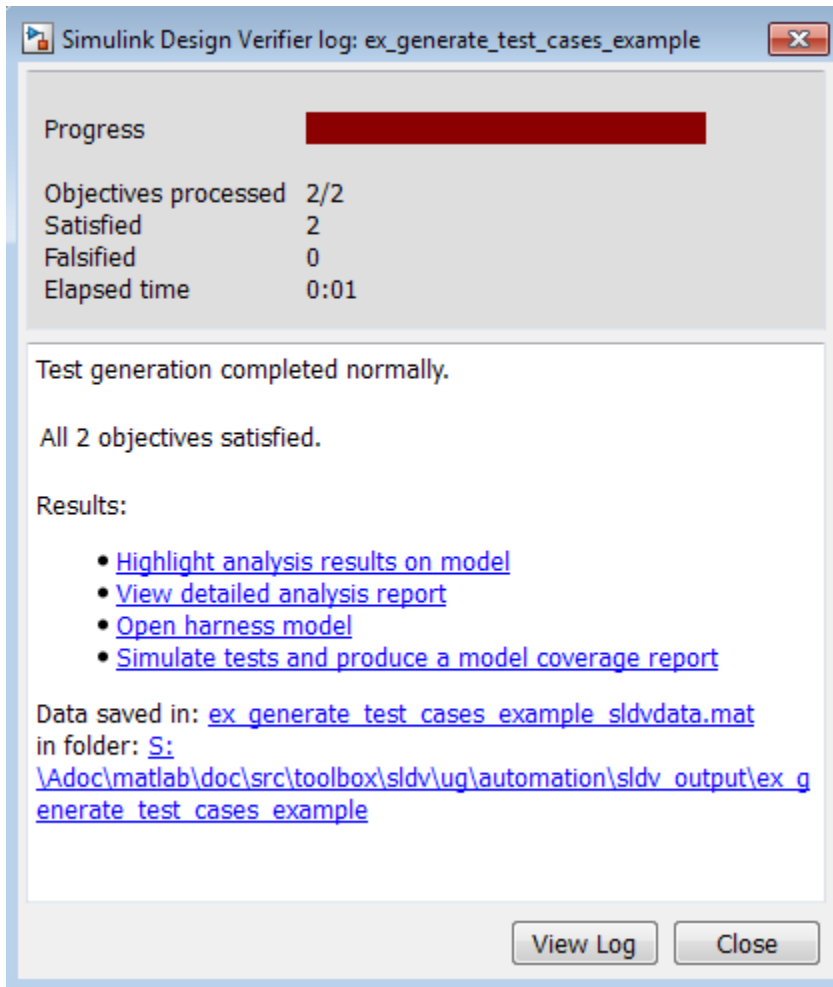
The following sections describe how you can review the analysis results:

- "Review Analysis Results on the Model" on page 7-10
- "Review Detailed Analysis Report" on page 7-12
- "Review Harness Model" on page 7-13
- "Simulate Tests and Produce a Model Coverage Report" on page 7-14
- "View sldvData File" on page 7-16

- "Review Analysis Results in the Model Explorer" on page 7-16

Review Analysis Results on the Model

Highlight the analysis results on the example model:

1 In the log window for the `ex_generate_test_cases_example` analysis, click **Highlight analysis results on model**.
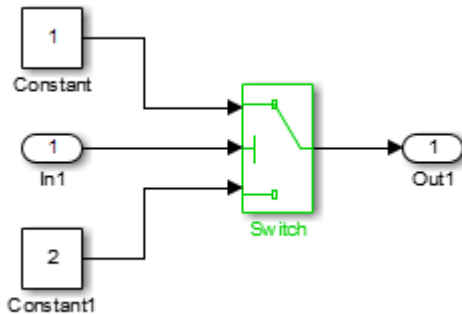
The Switch block is outlined in green, which indicates that the Switch block has test cases that satisfy its test objectives.

The Simulink Design Verifier Results window opens. As you click objects in the model, this window changes to display detailed analysis results for that object. By default, the Simulink Design Verifier Results window is always the topmost visible window. To allow the window to move behind other window, click [image] and clear Always on top.
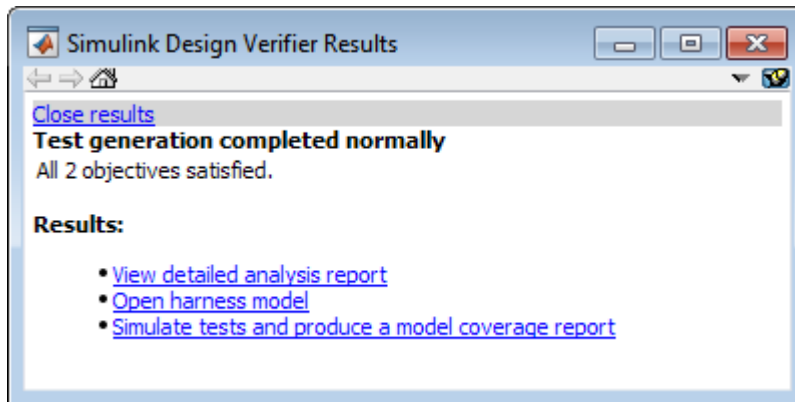
2   Click the highlighted Switch block.

The Simulink Design Verifier Results window indicates that the analysis generated test cases for both test objectives:

- `trigger >= threshold`
- `trigger < threshold`
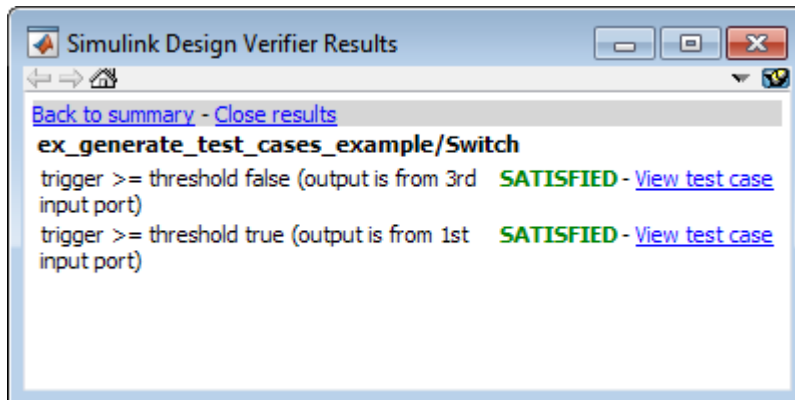
For more information about highlighted analysis results on a model, see "Highlighted Results on the Model" on page 13-2.

Review Detailed Analysis Report

Create a detailed HTML analysis report:

1   In the Simulink Design Verifier log window, click **Generate detailed analysis report**.

    The HTML report opens in a browser window.

2   The report includes the following **Table of Contents**. Click a hyperlink to navigate to a section in the report.

Table of Contents

1. Summary
2. Analysis Information
3. Test Objectives Status
4. Model Items
5. Test Cases

3   In the **Table of Contents**, click `Summary` to display the report's Summary chapter.

    The Summary chapter lists information about the model and the status of the objectives—satisfied or not.

4   In the **Table of Contents**, click `Analysis Information` to display the Analysis Information chapter.

    The Analysis Information chapter provides information about:

    • The model that you analyzed.

    • The options that you specified for the analysis.

    • Approximations the software performed during the analysis.

5   In the **Table of Contents**, click `Test Objectives Status` to display the report's Test Objectives Status chapter.

    This table indicates that the analysis satisfied both test objectives associated with the Switch block in the `ex_generate_test_cases_example` model, for which it generated two test cases.

6   Under the table **Test Case** column, click `2` to display the Test Case 2 section.

This section provides details about a test case that the analysis generated to achieve an objective in your model. This test case achieves test objective 1, when the Switch block passes its third input to its output port. Specifically, the software determines that a value of –1 for the Switch block control signal causes the block to pass its third input as the block output.

For more information about the HTML reports, see "Simulink Design Verifier Reports" on page 13-27.

Review Harness Model

To create a harness model with test cases that satisfy the test objectives in your model, in the Simulink Design Verifier log window, click **Create harness model**.

The software creates a harness model named
`ex_generate_test_cases_example_harness`.
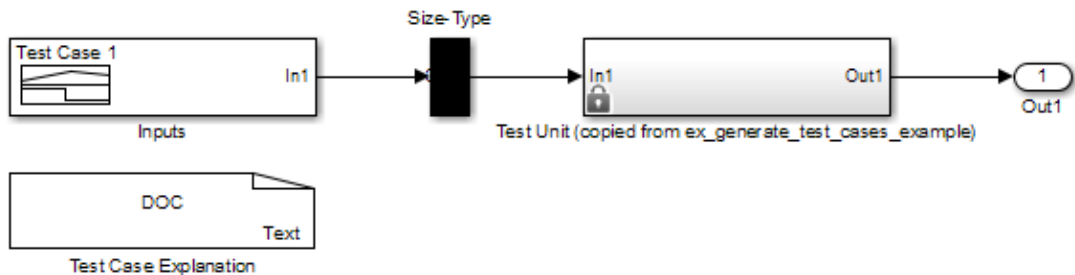
The Signal Builder block named Inputs contains the test cases. Double-click the Inputs block to see the test cases. From the Signal Builder block, you can simulate the model using the test cases and produce a model coverage report, as described in "Simulate Tests and Produce a Model Coverage Report" on page 7-14.

For more information about the harness model, see "Simulink Design Verifier Harness Models" on page 13-16.

If Analysis Generates Many Test Cases

If you have a large model, the analysis might produce a harness model that contains a large number of test cases.

To perform a more efficient analysis and create easier-to-review results:

1   Set the **Test suite optimization** parameter to `LongTestcases`.

2   Rerun the analysis.

In the `LongTestcases` optimization, the analysis generates fewer but longer test cases that each satisfy multiple test objectives.

Simulate Tests and Produce a Model Coverage Report

To simulate the harness model using the generated test cases in the harness model:

1   In the harness model, double-click the Inputs block to open the Signal Builder dialog box.
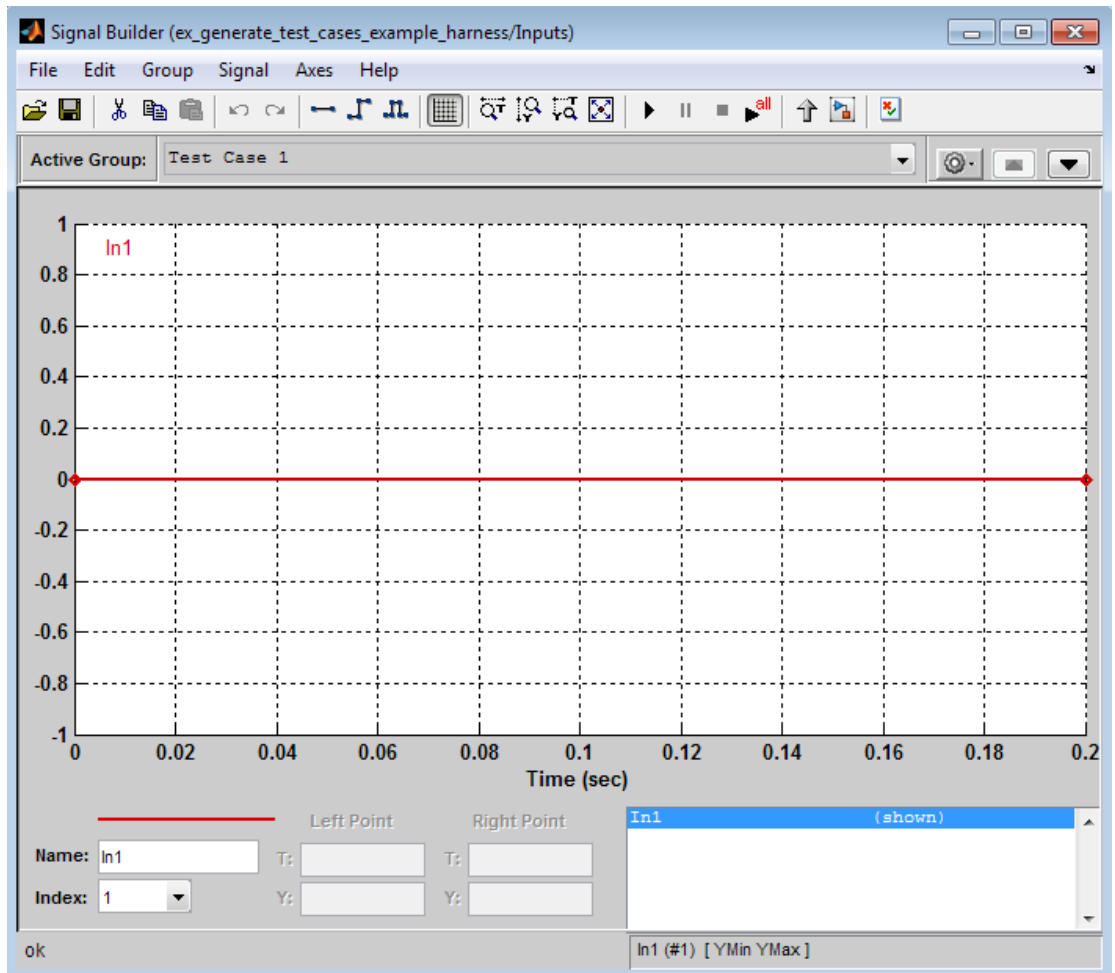
2

In the Signal Builder dialog box, click **Run all** .

The software simulates the harness model using both test cases, collects model coverage information, and displays a coverage report. The coverage report indicates that the test cases record 100% decision coverage for the `ex_generate_test_cases_example` model.

You can also simulate the model without creating a harness model. In the Simulink Design Verifier log window, click **Simulate tests and produce a model coverage report**.

For more information about model coverage, see "Top-Level Model Coverage Report" in the Simulink Verification and Validation documentation.

View `sldvData` File

The Simulink Design Verifier data file is a MAT-file that contains a structure named `sldvData`. This structure stores all the data that the analysis gathers and produces during the analysis. You can use the data file to conduct your own analysis or to generate a custom report.

To view the data file, click the data file name in the log window, in this example, `ex_generate_test_cases_example_sldvdata.mat`. When you click the file name, a copy of the `sldvData` object is instantiated in the MATLAB workspace so that you can review and manipulate the data.

For more information about Simulink Design Verifier data files, see "Simulink Design Verifier Data Files" on page 13-9.

Review Analysis Results in the Model Explorer

As long as your model remains open, you can view the results of your most recent Simulink Design Verifier analysis in the Model Explorer.

In the Simulink Editor, select **Analysis** > **Design Verifier** > **Results** > **Active**. The Model Explorer opens. The results of the latest Simulink Design Verifier analysis appear in the right-hand pane.

For any Simulink Design Verifier analysis, from the Model Explorer, you can perform the following tasks.

Task	For more information
Highlight the analysis results on the model.	"Highlighted Results on the Model" on page 13-2
Generate a detailed analysis report.	"Simulink Design Verifier Reports" on page 13-27
Create the harness model, or if the harness model already exists, open it.	"Simulink Design Verifier Harness Models" on page 13-16

Task	For more information
If no test cases were generated during the analysis, this option is not available.	
View the data file.	"Simulink Design Verifier Data Files" on page 13-9
View the log file.	"Simulink Design Verifier Log Files" on page 13-50

After you close your model, you can no longer view analysis results.

Customize Test Generation

You can use the `Test Condition` block to constrain signals in your model to certain values during the analysis.

1   At the MATLAB command prompt, enter `sldvlib` to display the Simulink Design Verifier library.

2   Open the Objectives and Constraints sublibrary.

3   Copy the Test Condition block to your model by dragging it from the Simulink Design Verifier library to your model window.

4   In the model window, insert the Test Condition block between the Inport and Switch blocks.
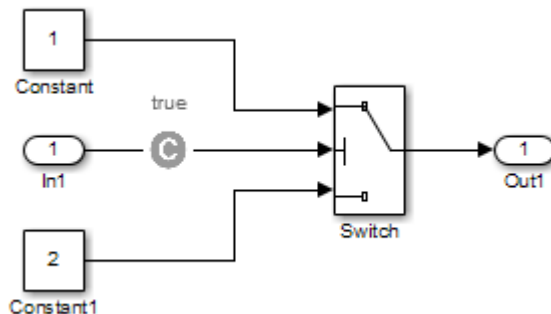
5   Double-click the Test Condition block to access its attributes.

The Test Condition block parameters dialog box opens.

6 In the **Values** box, enter [-0.1, 0.1]. When generating test cases for this model, the analysis constrains the signal values, entering the Switch block control port to the specified range.

Function Block Parameters: Test Condition

Design Verifier Test Condition (mask) (link)

Constrains signal values in Simulink Design Verifier test cases. The 'Values' parameter constrains the block input signal. Two element vectors specify intervals. Cell arrays specify lists. The signal must satisfy at least one of the values or intervals at every time step.
Example Values:
true
{[0 1], 2, [4 5], 6}
{Sldv.Interval(-2, -1), Sldv.Point(0), Sldv.Interval(0, 1, '()'), 1}

Parameters

☑ Enable

Type | Test Condition ▼

Values

[-0.1, 0.1]

☑ Display values

☑ Pass through style (show Outport)

OK   Cancel   Help   Apply

7 Click **OK** to save your changes and close the Test Condition block parameters dialog box.

8 Save your model as ex_generate_test_cases_with_tc_block and keep it open.

Reanalyze the Example Model

Analyze the `ex_generate_test_cases_with_tc_block` model with the Test Condition block. To observe how the Test Condition block affects test generation, compare the result of this analysis to the result that you obtained in "Analyze Example Model" on page 5-14.

1 In the model window, select **Analysis** > **Design Verifier** > **Generate Tests** > **Model**.

The Simulink Design Verifier software displays a log window and begins analyzing your model to generate test cases. When the software completes the analysis, the log window displays the options for reviewing the results.

2 In the Simulink Design Verifier log window, click **Generate detailed analysis report**.

3 To begin reviewing the report, in the **Table of Contents**, click Summary.

The Summary chapter indicates that Simulink Design Verifier satisfied two test objectives in the model.

4 In the **Table of Contents**, click Analysis Information. Scroll to the bottom of this chapter, to the Constraints section.

This section lists the Test Condition block that you added to constrain the value of the Switch block control signal to the interval [–0.1, 0.1].

5 In the **Table of Contents**, click Test Objectives Status.

This table indicates that Simulink Design Verifier satisfied both test objectives for the Switch block through the two test cases generated.

6 Under the table **Test Case** column, click 1.

This section provides details about a test case that the software generated to achieve an objective in your model. This test case achieves test objective 1, when the Switch block passes its third input to its output port. Although the Test Condition block restricts the domain of input signals to the interval [–0.1, 0.1], the software determines that a value of –0.1 for the Switch block control signal satisfies this objective.

7 To confirm that the test case achieves 100% decision coverage, open the harness model.

8 Double-click the Inputs block to open the Signal Builder dialog box.

9

In the Signal Builder dialog box, click **Run all** .

The Simulink software simulates the harness model using both test cases, collects model coverage information, and displays a coverage report. The Summary section of the report indicates that Simulink Design Verifier generated test cases that achieve complete decision coverage for your example model.

Analyze Contradictory Models

If the analysis produces the error `The model is contradictory in its current configuration`, the software detected a contradiction in your model and cannot analyze the model.

You can have a contradiction if your model has Test Objective blocks with incorrect parameters. For example, a contradiction can be an objective that states that a signal must be between 0 and 5 when the signal is the constant 10.

If the software detects a contradiction, all previous results are invalidated and the software reports that some of the objectives cannot be satisfied.

Use Test Generation Advisor to Identify Analyzable Components

In this section...
"Test Generation Advisor" on page 7-21
"Test Generation Advisor Requirements" on page 7-23
"Identify Analyzable Components" on page 7-23
"Analyze and Generate Tests for Model Components" on page 7-23
"Manually Select Components for Testing" on page 7-26

Test Generation Advisor

You can use the Test Generation Advisor to select model components (atomic subsystems and model blocks) for test generation. The Test Generation Advisor summarizes test generation compatibility, condition and decision objectives, and dead logic for the model and model components.

The Test Generation Advisor performs a high-level analysis and fast dead logic detection. You can use the results to better understand your model before test generation, particularly for large models, complex models, or models for which you are uncertain of the test generation compatibility. For example, you can:

· Identify components that are incompatible with test case generation.

· Identify complex components that may be time-consuming to analyze.

· Determine instances of dead logic.

· Get a snapshot of the component hierarchy.

· Get recommended test generation parameters.
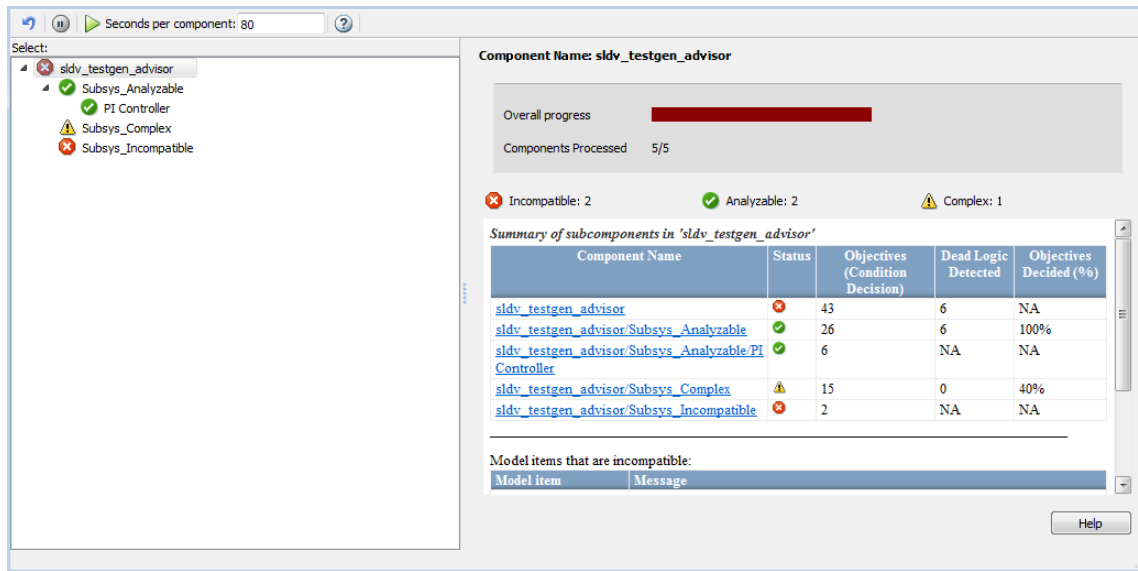
The Test Generation Advisor classifies components as analyzable, complex, or incompatible.

- *Analyzable* components are compatible with Simulink Design Verifier. The preliminary analysis indicates that Simulink Design Verifier might achieve high component coverage.

- *Complex* components are also compatible with Simulink Design Verifier. However, the preliminary analysis indicates that Simulink Design Verifier might require more time and resources to achieve high component coverage due to component complexity or other factors. For more information, see "Sources of Model Complexity" on page 14-2.

- You cannot generate tests for *incompatible* components. For more information, see "Check Model Compatibility" on page 3-2.

The results summary displays specific information about the model and each component:

- **Status**: The compatibility or complexity

- **Objectives**: The number of condition and decision objectives

- **Dead Logic Detected**: The number of instances of dead logic decided during the analysis. This might not include every instance of dead logic.

- **Objectives Decided**: The percentage of condition and decision objectives determined by test cases and dead logic.

Test Generation Advisor Requirements

For analysis, your model must compile. Also, if you change the model name, you must reload the model and reopen the Test Generation Advisor.

Identify Analyzable Components

To analyze your model using the Test Generation Advisor, follow this high-level workflow:

1 Open your model.

2 From the menu bar, click **Analysis > Design Verifier > Generate Tests > Advisor**.

3 Your model compiles, and the Test Generation Advisor opens. It displays the model hierarchy and summary table.

4 Enter a time value for **Seconds per component**, which limits the analysis time per component. This value does not include time for other operations such as compilation.

5 Run the analysis by clicking the Start Analysis button ▶. Track the analysis using the progress indicator.

6 Determine incompatibilities, complexities and characteristics from the component hierarchy tree and the results summary.

7 Trace from the summary to the model using the component hyperlinks.

Analyze and Generate Tests for Model Components

This example demonstrates analysis and test generation using the Test Generation Advisor. The example model has analyzable and incompatible subsystems.

1 At the command line, enter `fuelsys` to open the `fuelsys` model.

2 Save a copy of the model in a writable location on the MATLAB path.

3 Click **Analysis > Design Verifier > Generate Tests > Advisor** to open the Test Generation Advisor.
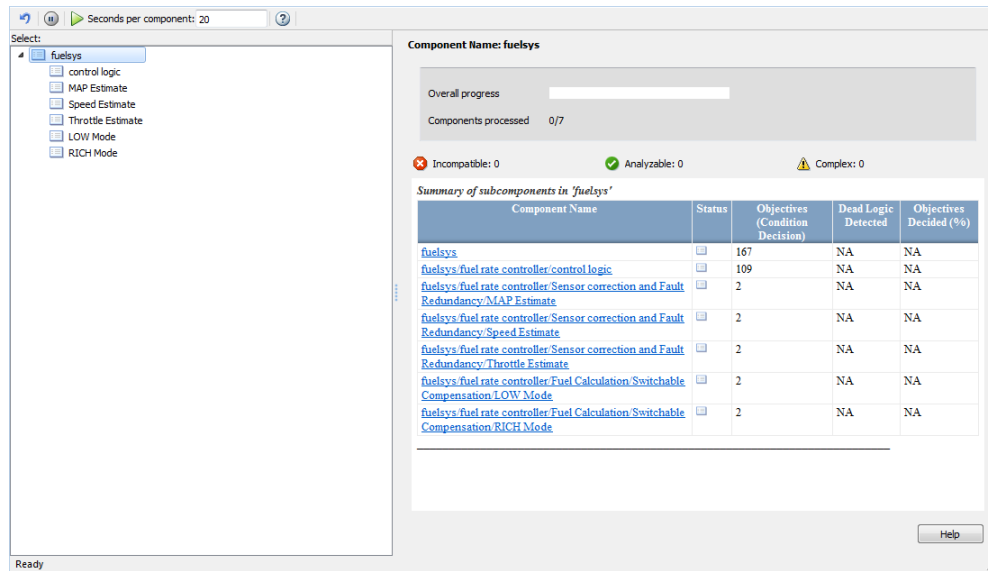
4 In the **Seconds per component** text box, enter 25.

5 Click the Start Analysis button ▷ to begin the model analysis.

6 After the analysis is complete, the component tree displays results for the overall model and each component.

7 Highlight the `control logic` subsystem in the component hierarchy. The analysis was partial, in that it determined 87% of the objectives for `control logic` by test cases and dead logic. To load the test generation summary, click the **Show test generation results summary** link.

At the bottom of the summary, the table lists recommended test generation parameters.

8. Click the **Component name** hyperlink. Simulink traces to the `control logic` Stateflow chart.

9. Generate the full set of tests for the subsystem. In the Test Generation Advisor summary for `control logic`, click **Extract this component and generate tests**.

Manually Select Components for Testing

If you know which model components that you want to test, you can manually select these components. Break down the model into components of 100–1000 objectives each. Use the `sldvextract` function to extract components into a new model. You can then analyze the individual components, starting with the lowest-level subsystems.

Model Coverage Objectives for Test Generation

Decision

Decision coverage in Simulink Design Verifier examines blocks and Stateflow states that represent decision points in a model. For instance, the Switch block involves the decision about whether the control input is greater than a threshold value. For more information, see "Model Objects That Receive Coverage" in Simulink Verification and Validation documentation.

To enable decision coverage, under **Design Verifier** > **Test Generation**, for **Model coverage objectives**, select one of the following:

- Decision
- Condition Decision
- MCDC

For each decision in your model, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see "Decision Coverage (DC)" in Simulink Verification and Validation documentation.

Condition

Condition coverage examines blocks that output the logical combination of their inputs and Stateflow transitions. For more information, see "Model Objects That Receive Coverage" in Simulink Verification and Validation documentation.

To enable condition coverage, under **Design Verifier** > **Test Generation**, for **Model coverage objectives**, select one of the following:

- Condition Decision

- MCDC

For each input to a logical block and each condition in a transition, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see "Condition Coverage (CC)" in Simulink Verification and Validation documentation. .

MCDC

Modifier condition/decision coverage examines blocks that output the logical combination of their inputs and Stateflow transitions. For more information, see "Model Objects That Receive Coverage" in Simulink Verification and Validation documentation.

To enable condition MCDC coverage, under **Design Verifier** > **Test Generation**, for **Model coverage objectives**, select MCDC.

For each input to a logical block and each condition in a transition, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see "MCDC Coverage for Stateflow Charts" in Simulink Verification and Validation documentation.

For information on how MCDC test generation deviates from strict MCDC coverage, see "Modified Condition and Decision Coverage in Simulink Design Verifier" on page 9-15.

Relational Boundary

Relational boundary coverage examines blocks that have an explicit or implicit relational operation and Stateflow transitions. For more information, see "Model Objects That Receive Coverage" in Simulink Verification and Validation documentation. Test generation for relational boundary coverage is not supported for If and Fcn blocks.

To enable relational boundary coverage, under **Design Verifier** > **Test Generation**, select **Include relational boundary objectives**.

For each relational operation in the model, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see "Relational Boundary Coverage" in Simulink Verification and Validation documentation.

8

Extending Existing Test Cases

When to Extend Existing Test Cases

The Simulink Design Verifier software can analyze your model using previously generated test cases that you specify. You can use this feature in the following situations:

- You encounter delays trying to analyze your model, or you see incomplete results. This can happen if your model has any of the following characteristics:
  - Temporal logic
  - Large counters
  - Model objects that are difficult to test due to complex or nonlinear logic

  Analyzing the model and considering the existing test cases allows you to focus the analysis on those parts of the model that are difficult to analyze. You can combine the generated test cases to create a complete test suite for the full model.

  For an example of extending existing test cases for a model that uses temporal logic, see "Extend Test Cases for Model with Temporal Logic" on page 8-5.

- You have a closed-loop simulation model that uses a Model block to include the controller. First, log the data from the Model block and then analyze the model referenced by the Model block. Using this technique, the test cases for the controller can realistically reflect the continuous time behavior expected in the closed-loop system.

  For an example of extending existing test cases for a closed-loop system, see "Extend Test Cases for Closed-Loop System" on page 8-12.

- You change an existing model for which you have already generated test cases . In this situation, you can reanalyze the model, omitting the analysis results from the original version of the model. The combined test cases give you a complete test suite for the new model.

  For an example of extending existing test cases for modified models, see "Extend Test Cases for Modified Model" on page 8-19.

Note: When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. In **Analysis > Design Verifier > Options**, if you specify your model to extend existing test cases with a **Data file** and apply parameter configurations with a **Parameter configuration file**, when you attempt to perform Simulink Design Verifier analysis, the software

reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.

Common Workflow for Extending Existing Test Cases

Use the following workflow for extending existing test cases during a test-generation analysis:

- Create the starting test cases.
- Log the starting test cases.
- Extend the existing test cases during test-generation analysis.
- Verify that you have created a complete test suite.

The examples in this category use some or all of these tasks when extending existing test cases during analysis.

Extend Test Cases for Model with Temporal Logic

In this section...

Create Starting Test Case

This example uses the `sldvdemo_sbr_extend_design` model. This model includes a Stateflow chart `SBR` that uses temporal logic. The transition from the `KEY_OFF` state to the `KEY_ON` state occurs after the Stateflow chart has been simulated 500 times. To test this transition requires a test case with 500 time steps.

In this example, you create a test case that forces the transition to `KEY_ON` by setting the `KEY` input to `1` for the duration of the test case. You simulate the model using this test case, satisfying the objectives for the `KEY_OFF`/`KEY_ON` transition. Then you analyze the model, ignoring the objectives already satisfied by the test case you create.

1 Open the example model:

   `sldvdemo_sbr_extend_design`

2 Open the SBR Stateflow chart to see the `KEY_OFF`/`KEY_ON` transition.
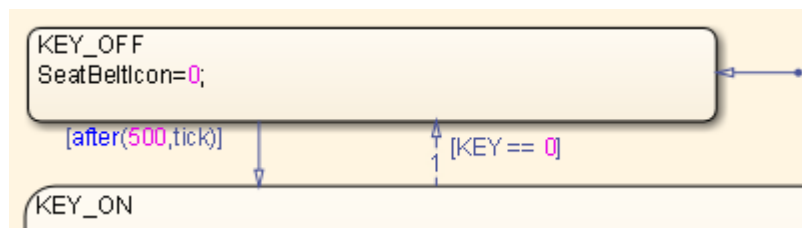
3 Create a model reference harness model:

```
[~, harnessModelFilePath] = ...
    sldvmakeharness('sldvdemo_sbr_extend_design',[],[],true);
```

The harness model, `sldvdemo_sbr_extend_design_harness`, includes:

- A Model block named Test Unit that references the original model, `sldvdemo_sbr_extend_design`.

Test Unit
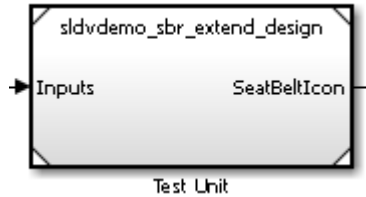
- A Signal Builder block named Inputs that contains the test-case inputs to the model referenced in the Model block.
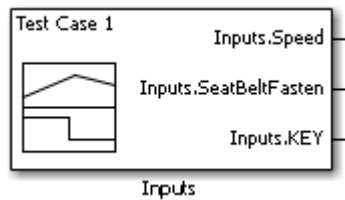
Inputs

Initially, the Signal Builder block contains only the default test case, with all three inputs set to 0.

- A DocBlock block named Test Case Explanation that documents the test case.

Test Case Explanation
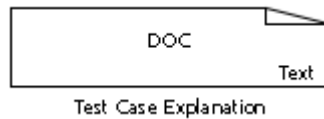
Initially, the Test Case Explanation block does not have any content for the default test case.

4 `sldvmakeharness` returns the path to the harness model file in `harnessModelFilePath`. Extract the name of the harness model file into `harnessModel`, for later use:

```
[~, harnessModel] = fileparts(harnessModelFilePath);
```

In order to analyze the KEY_OFF to KEY_ON state transition, create a test case that makes the transition to the KEY_ON state in 500 time steps:

1   Open the Signal Builder dialog box for the harness model.

2   Select **Axes** > **Change Time Range**.

3   The Signal Builder's time range determines the span of time over which its output is explicitly defined. In the Set the total time range dialog box, set the **Max time** field to 5 seconds, creating 500 time steps of 0.01 seconds duration each.

4   Set the KEY input to 1 for the duration of this starting test case, forcing the transition to the KEY_ON state. Selecting the Inputs.KEY signal requires two clicks. First, click the signal so that dots appear at both ends of the signal.
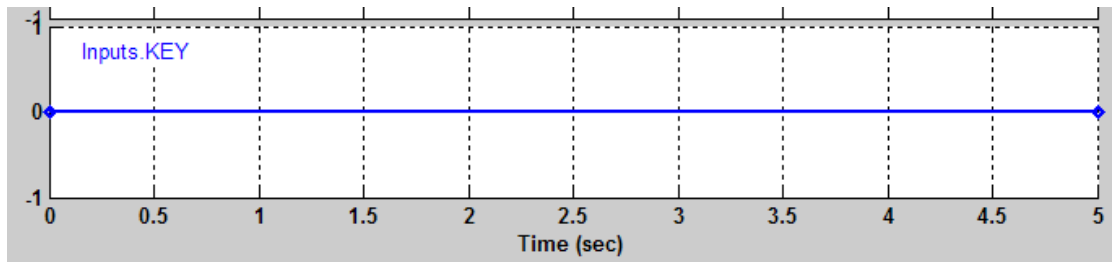
5   Click the Inputs.KEY signal again. The Signal Builder thickens the signal to indicate that it is selected.
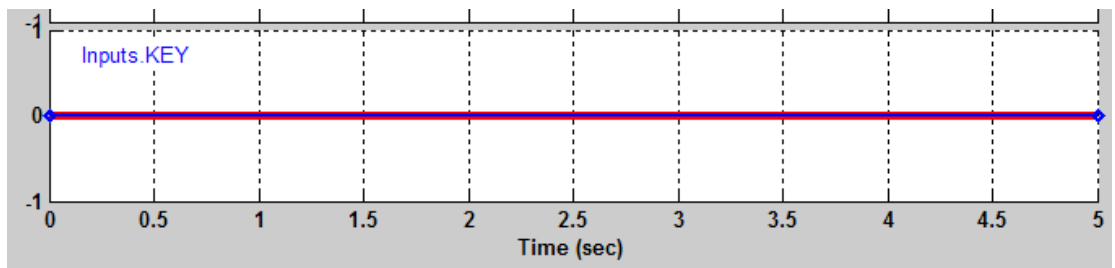
6   At the bottom of the Signal Builder dialog box, under **Left Point**, enter 1 for **Y**.

7   Press **Enter** to apply the change.

The Inputs.KEY signal is set to 1 for the duration of the test case.
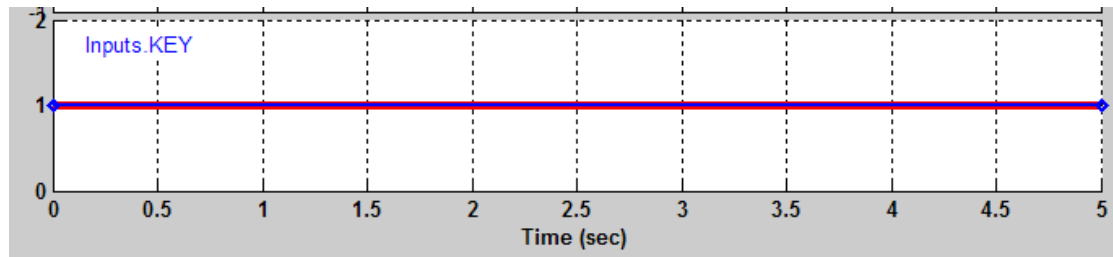
8   Close the Signal Builder dialog box.

Log Starting Test Case

The next step is to log the starting test case that you created. You can then specify that Simulink Design Verifier ignore the objectives satisfied by that test case when performing an analysis.

The `sldvlogsignals` function records the test case data in a MAT-file that contains an `sldvData` structure. This structure stores all the data that the software gathers and produces during the analysis.

To log the starting test cases:

1   Save the name of the Model block in the harness model that references the `sldvdemo_sbr_extend_design` model:

```
[~, modelBlock] = find_mdlrefs(harnessModel, false);
```

2   Simulate the model referenced by the Model block using the new test case, and log the input signals in the workspace variable `loggeddata`:

```
loggeddata = sldvlogsignals(modelBlock{1});
```

3   Save the logged data in a MAT-file named `existingtestcase.mat`:

```
save('existingtestcase.mat', 'loggeddata');
```

You will specify this file when you analyze the `sldvdemo_sbr_extend_design` model.

Extend Existing Test Cases

You can now analyze the `sldvdemo_sbr_extend_design` model and specify that the analysis extend the test cases already satisfied. The analysis uses the existing test-case data as a starting point, and does not try to generate test cases for the KEY_OFF to KEY_ON transition in the SBR Stateflow chart.

Specify the starting test case and analyze the model:

1   In the model window for `sldvdemo_sbr_extend_design`, select **Analysis** > **Design Verifier** > **Options**.

2   In the Configuration Parameters dialog box, in the **Select** tree, under **Design Verifier**, select **Test Generation**.

3   On the **Test Generation** pane, under **Existing test cases**, select **Extend existing test cases**.

4   In the **Data file** field, enter the name of the MAT-file that contains the logged data:

    `existingtestcase.mat`

5   Clear **Ignore objectives satisfied by existing test cases**.

    When you clear this option, the software includes the starting test case in the final test suite. You will see that the complete test suite achieves 100% model coverage.

6   To close the Configuration Parameters dialog box, click **OK**.

7   Save the `sldvdemo_sbr_extend_design` model on the MATLAB path with the name `sldvdemo_sbr_extend_design_test`.

8   In the Model Editor, select **Analysis** > **Design Verifier** > **Generate Tests** > **Model**.

    The log window first lists the objectives that the starting test case satisfied.
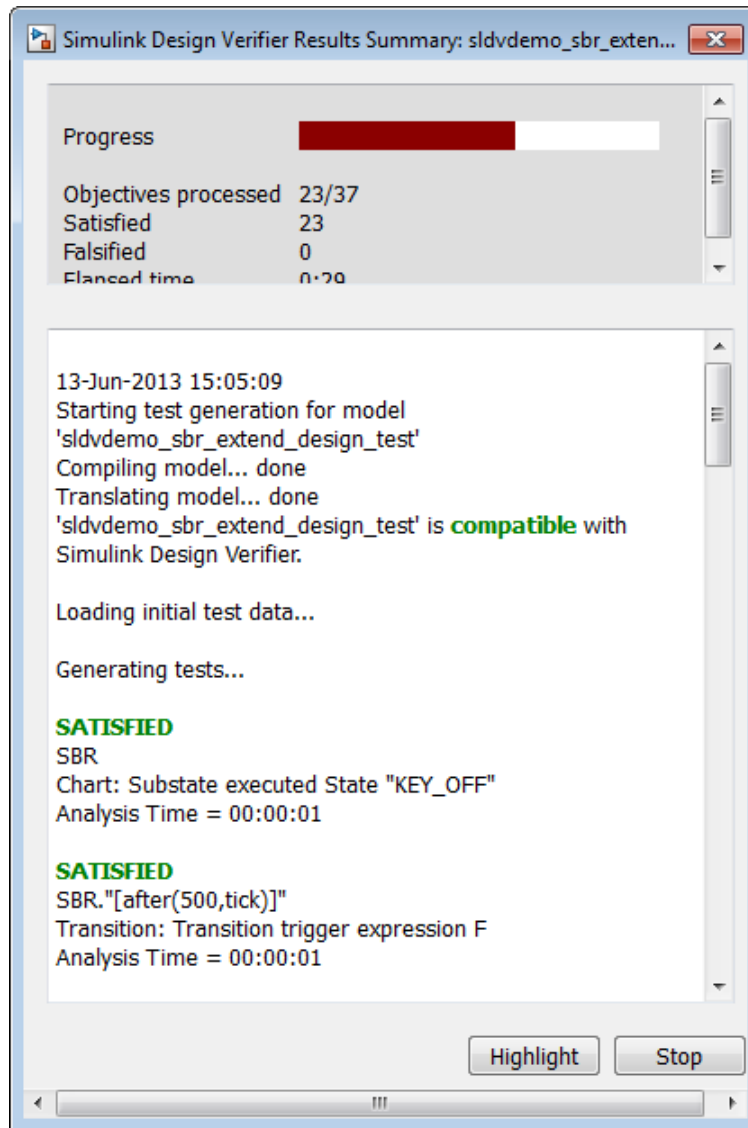
The log window then lists the objectives generated beyond the starting test case.

Verify Analysis Results

To make sure that this analysis creates a complete test suite, generate the harness model so you can simulate the model with the generated test cases:

1   In the log window, click **Create harness model**.

2   In the harness model `sldvdemo_sbr_extend_design_test_harness`, open the Signal Builder block named Inputs.

3   To simulate the model using all the test cases, click the **Run all and produce coverage** button .

    When the simulation is complete, the model coverage report is displayed.

4   View the coverage information for the `sldvdemo_sbr_extend_design_test` model to see that the complete test suite achieves 100% coverage.
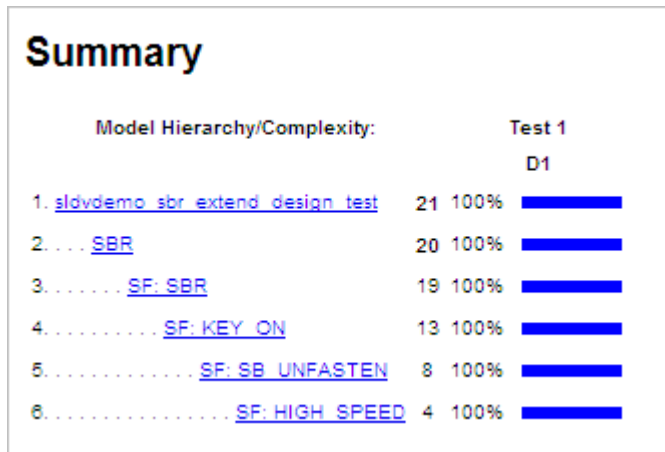
Extend Test Cases for Closed-Loop System

In this section...
"Log Starting Test Case" on page 8-12
"Extend Existing Test Cases" on page 8-15

Suppose that you have a model with a closed-loop controller in a model referenced by a Model block. You do not record 100% coverage for the referenced model. Extending existing test cases can help you achieve 100% coverage. The Simulink Design Verifier software adds time steps to the existing test cases when analyzing the controller implemented by the referenced model. The test cases that result from the analysis realistically reflect the continuous time behavior expected in the closed-loop controller.

A *closed-loop controller* passes instructions to the controlled system and receives information from the environment as the control instructions execute. The controller can adapt and change its instructions as it receives this information.

Log Starting Test Case

This example uses the `sldemo_mdlref_basic` model. The CounterA Model block references the model `sldemo_mdlref_counter`. When you simulate the parent model, `sldemo_mdlref_basic`, and collect coverage, you record only 75% coverage for `sldemo_mdlref_counter`. Log the data from the simulation and extend those test cases to achieve 100% coverage for the referenced model.

1 Open the example model:

   `sldemo_mdlref_basic`

2 In the Simulink Editor, select **Analysis** > **Coverage** > **Settings**.

3 In the **Coverage** pane of the Configuration Parameters, select **Enable coverage analysis**.

4 Select **Referenced Models**

   Note that the analysis records coverage only for referenced models with **Simulation mode** set to `Normal`, `SIL`, or `PIL`. In `sldemo_mdlref_basic`, the CounterC Model block has **Simulation mode** set to `Accelerator`, so you cannot record coverage for it.

5 Under **Coverage metrics**, set the structural coverage level to **Modified Condition Decision (MCDC)** to record decision, condition, and modified condition/decision coverage.

6 Click **OK**.

7 Simulate the model.

When the simulation completes, the generated coverage report opens in a browser window. The report shows the following coverage results for the referenced model:

- Condition: 50% (2/4) condition outcomes

- Decision: 25% (1/4) decision outcomes

- MCDC: 0% (0/2) conditions reversed the outcome

The coverage results are also highlighted in the referenced model, `sldemo_mdlref_counter`. You can select individual model objects to view specific coverage results in the Coverage dialog box, as shown in the following screenshot.
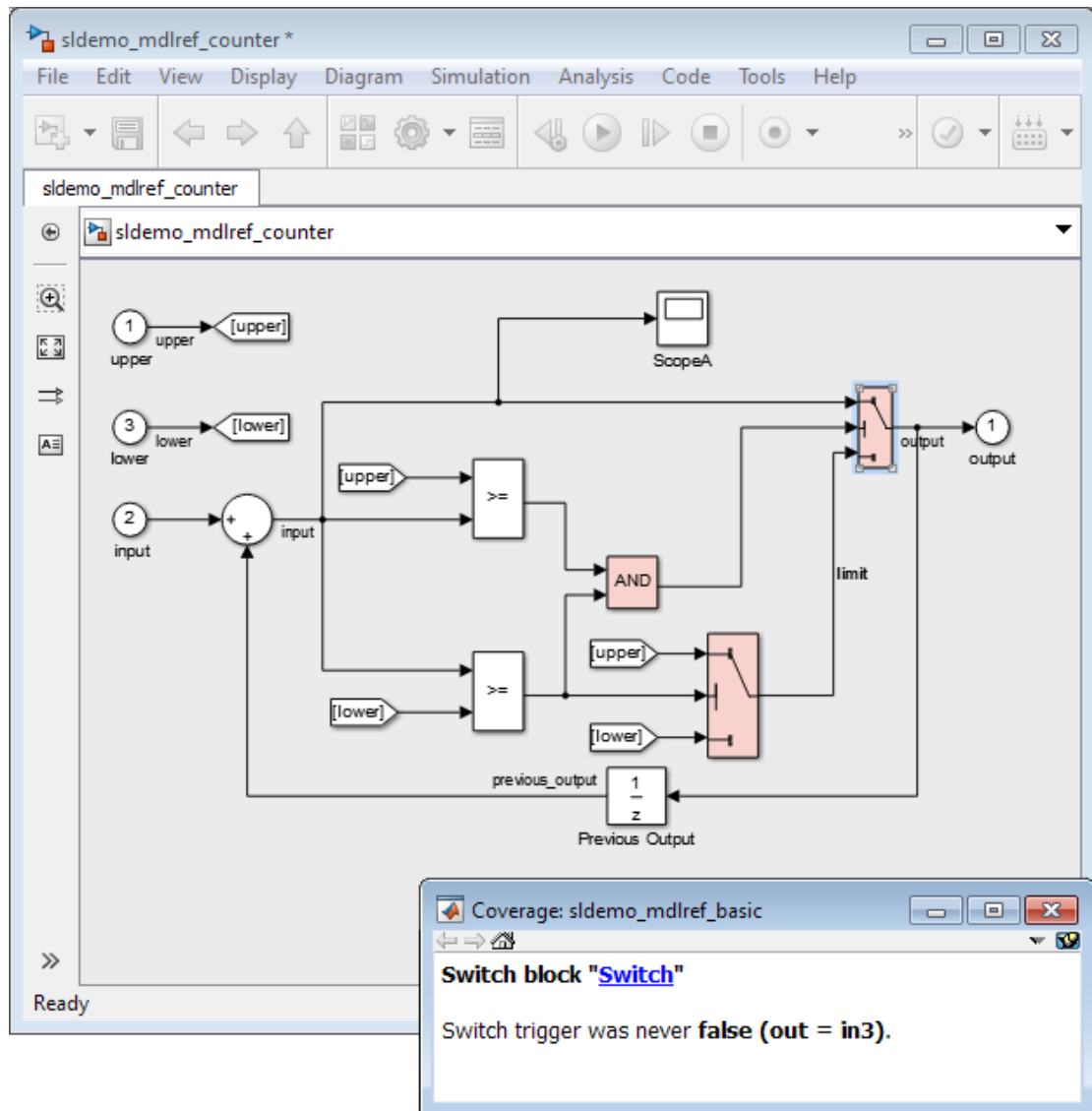
8 To log the input signals for the CounterA Model block in `sldemo_mdlref_basic`
during simulation, at the MATLAB command prompt, enter the following code:

```
logged_data = sldvlogsignals('sldemo_mdlref_basic/CounterA');
```

9 Save the logged data in a MAT-file named `existingtestcase.mat`:

```
save('existingtestcase.mat', 'logged_data');
```

When you analyze the model referenced in CounterA (`sldemo_mdlref_counter`) to extend existing test cases, you specify this MAT-file.

Extend Existing Test Cases

Analyze the `sldemo_mdfref_counter` model, specifying that the analysis extend the test cases already satisfied:

1 To open the `sldemo_mdfref_counter` model, in the `sldemo_mdlref_basic` model, double-click the CounterA Model block.

2 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options**.

3 In the Configuration Parameters dialog box, on the **Select** pane, under **Design Verifier**, select **Test Generation**.

4 On the **Test Generation** pane, in the **Model coverage objectives** box, select MCDC.

5 Under **Existing test cases**, select **Extend existing test cases**.

6 In the **Data file** field, specify the name of the MAT-file that contains the logged data, in this case, `existingtestcase.mat`.

7 Clear **Ignore objectives satisfied by existing test cases**.

When you clear this option, the software includes the test cases recorded in the file `existingtestcase.mat` in the final test suite.

8 Click **OK**.

9 In the Simulink Editor, select **Analysis** > **Design Verifier** > **Generate Tests** > **Model**.

The analysis first loads the objectives satisfied by the logged test cases. Then it adds extra time steps to those test cases and tries to satisfy any missing objectives. When the analysis completes, the Simulink Design Verifier log window opens and indicates that all 12 objectives are satisfied.

10 To view the analysis results on the model, in the Simulink Design Verifier log window, select **Highlight analysis results on model**.

The Simulink Design Verifier results are highlighted in the referenced model, `sldemo_mdlref_counter`. You can select individual model objects to view specific analysis results in the Simulink Design Verifier Results dialog box, as shown in the following screenshot.

11 To verify the results of the analysis and review the generated test cases, in the Simulink Design Verifier log window, select **Generate detailed analysis report**.

12 To collect model coverage using the extended test suite, in the Simulink Design Verifier log window, select **Simulate tests and produce a model coverage report**.

When the simulation completes, the generated coverage report opens in a browser window. The report now shows the following coverage results for the referenced model `sldemo_mdlref_counter`:

- Condition: 100% (4/4) condition outcomes
- Decision: 100% (4/4) decision outcomes
- MCDC: 100% (2/2) conditions reversed the outcome

Extend Test Cases for Modified Model

In this section...
"Create Starting Test Cases" on page 8-19
"Extend Existing Test Cases" on page 8-20

Suppose that you have a model that you have already analyzed using Simulink Design Verifier, and you modify the model. The original test suite may not record 100% coverage for the modified model. Reanalyze the modified model to make sure that it satisfies all the new test objectives. Instead of reanalyzing the entire model, you focus the new analysis on just the modified part of the model. In this way, you leverage the test cases created for the original model, extending them to satisfy any new objectives.

This example uses the `sldvdemo_cruise_control` model. You analyze the model and generate test cases. Then you analyze a modified version of that model, `sldvdemo_cruise_control_mod`, extending the test cases from the original analysis. The analysis returns a complete test suite for the new model.

Create Starting Test Cases

Analyze the `sldvdemo_cruise_control` model and generate test cases that achieve 100% coverage.

1 Open the example model:

```
sldvdemo_cruise_control
```

2 To start a Simulink Design Verifier analysis for the `sldvdemo_cruise_control` model, double-click the Run Simulink Design Verifier block:

Run
(double-click)
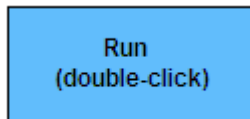
Run Simulink Design Verifier

The analysis satisfies 34 test objectives for the `sldvdemo_cruise_control` model. The software stores the resulting data file in a subfolder of the MATLAB Current Folder:

sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_sldvdata.mat

In the next section, when you analyze the modified model, this data file specifies the starting test cases that you extend.

3 Close the sldvdemo_cruise_control model and all the files created by the analysis. If asked, do not save any changes you made to the model.

Extend Existing Test Cases

The sldvdemo_cruise_control_mod model is a modified version of sldvdemo_cruise_control. The Controller subsystem contains a Saturation block that specifies that the target speed cannot exceed 70.

Open the modified model and analyze it, extending the test cases that you generated when analyzing the sldvdemo_cruise_control model:

1 Open the example model, the modified version of sldvdemo_cruise_control:

sldvdemo_cruise_control_mod

2 Double-click the Controller subsystem to see the change to the original model, a Saturation block that specifies the maximum speed:
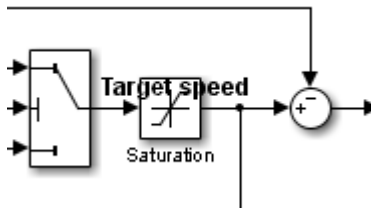
3 Close the Controller subsystem.

4 Select **Analysis** > **Design Verifier** > **Options**.

5 In the Configuration Parameters dialog box, on the **Select** pane, under **Design Verifier**, select **Test Generation**.

6 On the **Test Generation** pane, under **Existing test cases**, select **Extend existing test cases**.

7 In the **Data file** field, click **Browse** and navigate to the MAT-file created in the MATLAB Current Folder when analyzing the original model:

sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_sldvdata.mat

8 Clear **Ignore objectives satisfied by existing test cases**.

When you clear this option, the analysis includes the test cases recorded in the file sldvdemo_cruise_control_sldvdata.mat in the final test suite.

9 Click **Apply** to save these settings.

10 To open the main **Design Verifier** pane, in the **Select** tree, click **Design Verifier**.

11 To start the analysis, click **Generate Tests**.

The analysis first loads the 34 objectives satisfied by the initial test cases. Then it adds extra time steps to those test cases and tries to satisfy any missing objectives.

12 In the log window, click **Generate detailed analysis report**.

The analysis satisfied a total of 38 satisfied objectives for the sldvdemo_cruise_control_mod model. The analysis satisfied four additional objectives that correspond to the Saturation block.

Objectives Satisfied

Simulink Design Verifier found test cases that exercise these test objectives.

#	Type	Model Item	Description	Test Case
1	Decision	Controller/Switch1	logical trigger input false (output is from 3rd input port)	3
2	Decision	Controller/Switch1	logical trigger input true (output is from 1st input port)	1
3	Decision	Controller/Saturation	input > lower limit F	1
4	Decision	Controller/Saturation	input > lower limit T	3
5	Decision	Controller/Saturation	input >= upper limit F	1
6	Decision	Controller/Saturation	input >= upper limit T	10

9

Achieving Test Cases for Missing Model Coverage

Generate Test Cases for Missing Coverage Data

If you simulate your model and record coverage data, but your model does not achieve 100% coverage, Simulink Design Verifier can find test cases that achieve the missing coverage. The software targets the test-generation analysis for the part of the model that is missing coverage, ignoring the model coverage data that was recorded during simulation.

The following examples describe how to focus the test-generation analysis on a part of the model that did not achieve 100% coverage:

- "Achieve Missing Coverage in Referenced Model" on page 9-3
- "Achieve Missing Coverage in Closed-Loop Simulation Model" on page 9-12

Achieve Missing Coverage in Referenced Model

If you simulate a referenced model that does not achieve full coverage, you can use Simulink Design Verifier to generate test cases that achieve full coverage. There are two approaches:

- Programmatically achieve missing coverage: Generate test cases for a referenced model with APIs for test-generation analysis.

- Incrementally increase coverage: Generate test cases for the test harness model with missing coverage analysis features.

Programmatically Achieve Missing Coverage in Referenced Model

- "Record Coverage Data for Example Model" on page 9-3
- "Find Test Cases for the Missing Coverage" on page 9-5
- "Achieve Missing Coverage" on page 9-5
- "Verify Complete Model Coverage" on page 9-6

This example model uses a referenced model that does not achieve full coverage. When you run a test-generation analysis on the referenced model and combine it with previously recorded coverage data, you can achieve 100% coverage for the referenced model.

Record Coverage Data for Example Model

Simulate the example model. Record condition, decision, and MCDC coverage.

1 Open the example model:

   sldemo_mdlref_basic

   The Model blocks CounterA, CounterB, and CounterC reference the model sldemo_mdlref_counter.

2 In the Simulink Editor, select **Analysis** > **Coverage** > **Settings**.

3 On the **Coverage** pane of the Configuration Parameters dialog box, set the following options:

   - Select **Enable coverage analysis**.

- Select **Referenced Models**.

- Click **Select Models**. In the Select Models for Coverage Analysis dialog box, select the check box for the referenced model `sldemo_mdlref_counter`. Click **OK**.

  The check box for `sldemo_mdlref_counter` becomes visible, corresponding to CounterA and CounterB. Coverage is not enabled for CounterC because the reference model CounterC is in `Accelerator` simulation mode.

- Specify which types of coverage to record during simulation. Under **Coverage metrics**, select **MCDC**.

4   In the **Coverage** > **Results** pane of the Configuration Parameters. Set the following options:

- Select **Save last run in workspace variable** to save the collected coverage data from the most recent simulation run in a variable in the MATLAB workspace.

- Select **Generate report automatically after analysis** to specify that the simulation create a coverage report.

- In the **cvdata object name** field, enter `covdata_original` to specify a unique name for the coverage data workspace variable.

5   Click **OK**.

6   To record the coverage data, start the simulation of the `sldemo_mdlref_basic` model.

  After the simulation, the coverage report opens. The report indicates that the following coverage is achieved for the referenced model `sldemo_mdlref_counter`:

- Decision: 25%

- Condition: 50%

- MCDC: 0%

  The simulation saves the coverage data in the MATLAB workspace variable `covdata_original`, a `cvdata` object that contains the coverage data.

7   Save the coverage data in a file on the MATLAB path:

```
cvsave('existingcov',covdata_original);
```

  Keep the model open as you continue through this example.

Find Test Cases for the Missing Coverage

To achieve 100% coverage for the `sldemo_mdlref_counter` model, run a test-generation analysis that uses the existing coverage data.

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

```
open_system('sldemo_mdlref_counter');
```

2 Create an `sldvoptions` object:

```
opts = sldvoptions;
```

When you create the `sldvoptions` object, specify:

- That the analysis ignores satisfied coverage data.
- The file name containing the satisfied coverage data (`existingcov.cvt`)

Enter the following commands to specify these options:

```
opts.IgnoreCovSatisfied = 'on';
opts.CoverageDataFile = 'existingcov.cvt';
```

3 Analyze the referenced model, `sldemo_mdlref_counter`, by using the specified options:

```
[status, fileNames] = sldvrun('sldemo_mdlref_counter',opts,true);
```

The Simulink Design Verifier analysis satisfies seven objectives and creates one test case for the referenced model.

The next procedure simulates the referenced model, `sldemo_mdlref_counter`, with the test case that the analysis created.

Achieve Missing Coverage

To achieve the missing coverage for the referenced model, `sldemo_mdlref_counter`, simulate the model by using the test case from the Simulink Design Verifier analysis.

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

```
open_system('sldemo_mdlref_counter');
```

2 Create a `cvtest` object for the simulation and specify recording decision, condition, and MCDC coverage.

```
cvt = cvtest('sldemo_mdlref_counter');
```

```
cvt.settings.decision = 1;
cvt.settings.condition = 1;
cvt.settings.mcdc = 1;
```

3 Specify recording coverage and set the name of the `cvtest` object.

```
runOpts = sldvruntestopts;
runOpts.coverageEnabled = true;
runOpts.coverageSetting = cvt;
```

4 Simulate the model with the `cvtest` object, `cvt`, and the test case, as defined in `fileNames.DataFile`. Save the recorded coverage data in the workspace variable `covdata_missing`.

```
[~, covdata_missing] = sldvruntest('sldemo_mdlref_counter', fileNames.DataFile, runOpts);
```

Verify Complete Model Coverage

You saved the coverage data from the simulation of the top-level model, `sldemo_mdlref_basic`, in the workspace variable `covdata_original`. To create a report that combines the coverage data from the top-level model with the missing coverage data from the referenced model, `sldemo_mdlref_counter`, enter the following command:

```
cvhtml('Coverage Summary', covdata_original, covdata_missing);
```

The report shows that by analyzing the referenced model and using those results to record coverage, you can achieve 100% decision, condition, and MCDC coverage.

Summary

Model Hierarchy/Complexity:		Test 1			Test 2			Total		
		D1	C1	MCDC	D1	C1	MCDC	D1	C1	MCDC
1. sldemo_mdlref_counter	3	25%	50%	0%	75%	100%	0%	100%	100%	100%
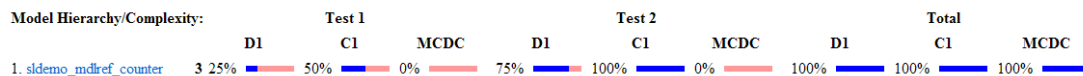
Increase Coverage for Referenced Models in a Test Harness

You can incrementally achieve full coverage for a generated test harness model. This example shows how to first generate a test harness model that does not achieve full

coverage. Next, it shows how to run missing coverage analysis on the test harness model to generate test cases for 100% coverage.

Note: This approach supports only test harness models generated by Simulink Design Verifier that reference the input model. For more information see, "Reference input model in generated harness" on page 15-80.

Generate Test Harness Model and Record Coverage Data

To achieve full coverage for the `sldemo_mdlref_counter` model, run a missing coverage analysis on the Simulink Design Verifier generated harness model.

1   Open the example model:

    ```
    open_system('sldemo_mdlref_counter');
    ```

2   Create a harness model for referenced model `sldemo_mdlref_counter`:
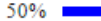
    ```
    [savedHarnessFilePath] = sldvmakeharness('sldemo_mdlref_counter');
    ```

    For more information about the harness model, see "Simulink Design Verifier Harness Models" on page 13-16.

3   In the harness model `sldemo_mdlref_counter_harness`, the **Format** parameter must be `Dataset` to ensure that the referenced model `sldemo_mdlref_counter` and the harness model `sldemo_mdlref_counter_harness` have the same parameter settings. For more information see, "Model Configuration Parameters: Data Import/Export".

4   Simulate the `sldemo_mdlref_counter_harness` model to record the coverage achieved by the test cases in the harness model. After the simulation, the coverage report appears. The report indicates that the following coverage is achieved for `sldemo_mdlref_counter`:

Summary

Model Hierarchy/Complexity	Test 1					
		Decision	Condition	MCDC	Execution	Relational Boundary
1. sldemo_mdlref_counter	3	25%	50%	0%	86%	50%

Generate Test Cases for the Missing Coverage

1 Open the harness model:

   ```
   open_system('sldemo_mdlref_counter_harness');
   ```

   To generate test cases for the missing coverage, in the Simulink Editor, select
   Analysis > **Design Verifier** > **Generate Tests** > **Missing Coverage**. A
   notification indicates the number of new tests that are added.
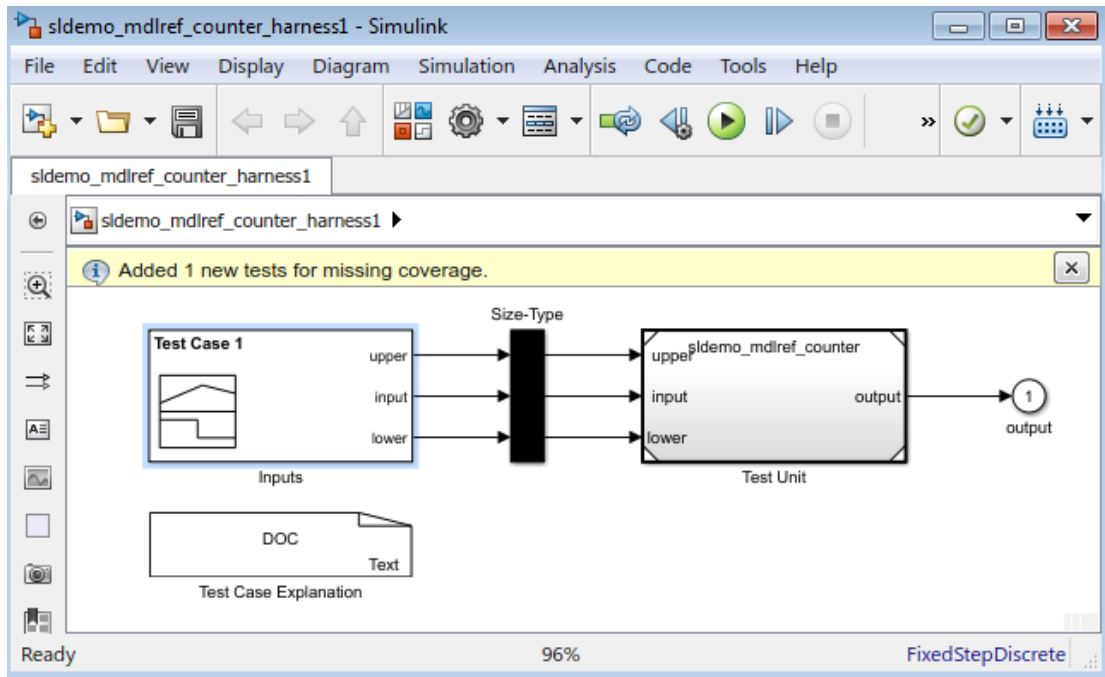
2 The Signal Builder dialog box shows the **Missing coverage test case 1** added to
   the previous **Test Case 1**.

3
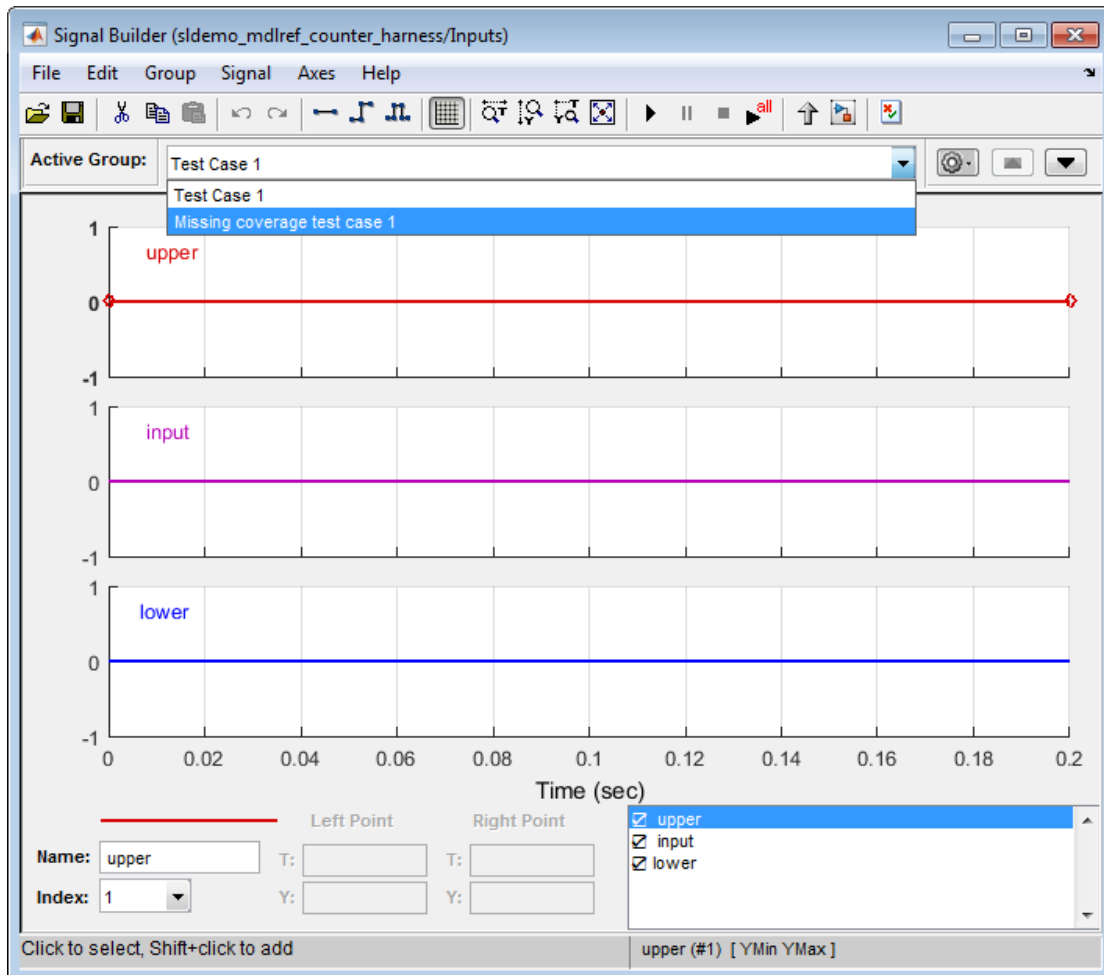
In the Signal Builder dialog box, click **Run all** ![run all icon]. The software simulates the harness model by using all the test cases, collects model coverage information, and displays a coverage report. The coverage report indicates that the missing coverage analysis records 100% coverage for `sldemo_mdlref_counter`.

Summary

Model Hierarchy/Complexity	Test 1				
	Decision	Condition	MCDC	Execution	Relational Boundary
1. sldemo_mdlref_counter 3	100% ▬▬	100% ▬▬	100% ▬▬	100% ▬▬	50% ▬▬

Update Simulink Design Verifier Analysis Options

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

```
open_system('sldemo_mdlref_counter_harness');
```

In the Simulink Editor, select **Analysis** > **Design Verifier** > **Options (sldemo_mdlref_counter)**. The Configuration Parameters dialog box for referenced model `sldemo_mdlref_counter` opens. You can set design verifier options for missing coverage analysis. For more information see, "Options in Configuration Parameters Dialog Box" on page 15-2.

View Active Results for Missing Coverage Analysis

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

```
open_system('sldemo_mdlref_counter');
```

To view active results for missing coverage test cases, in the Simulink Editor, select **Analysis** > **Design Verifier** > **Results** > **Active**. The Results Summary window opens with the missing coverage analysis results. For more information on active results, see "Review Analysis Results" on page 13-52. The missing coverage test cases data is stored in a MAT-file that contains a structure named `sldvData`. For more information see, "Contents of `sldvData` Structure" on page 13-9.

Limitations

1 Missing Coverage analysis is a user interface-based workflow. No command-line functions are available for Missing Coverage analysis.

2 Constraining values for parameters is not supported in the Missing Coverage analysis workflow. For more information see, "Define Constraint Values for Parameters" on page 5-4.

Missing Coverage in Subsystems and Model Blocks

If your model has a Subsystem block that does not achieve full coverage, you can convert it to model referenced in a Model block. "Convert a Subsystem to a Referenced Model" describes how to convert a subsystem to a referenced model. You can then follow the steps described in "Achieve Missing Coverage in Referenced Model" on page 9-3.

You cannot convert some subsystems to Model blocks. To test a subsystem to see if you can convert it to a Model block, use the `Simulink.SubSystem.convertToModelReference` function. If that function cannot convert the subsystem, an error message describes why the conversion failed.

It is possible that you have a Stateflow chart or a MATLAB Function block that does not achieve full coverage. You cannot convert Stateflow charts and MATLAB Function blocks to referenced models.

When you cannot use aModel block, follow the steps described in "Achieve Missing Coverage in Closed-Loop Simulation Model" on page 9-12.

Achieve Missing Coverage in Closed-Loop Simulation Model

In this section...
"Record Coverage Data for the Model" on page 9-12
"Find Test Cases for Missing Coverage" on page 9-13

If you have a subsystem or a Stateflow chart that does not achieve 100% coverage, and you do not want to convert the subsystem or chart to a Model block, follow this example to achieve full coverage.

The example uses a closed-loop controller model. A *closed-loop controller* passes instructions to the controlled system and receives information from the environment as the control instructions are executed. The controller can adapt and change its instructions as it receives this information.

The `sldvdemo_autotrans` model is a closed-loop simulation model. The ShiftLogic Stateflow chart represents the controller part of this model. Test cases designed in the ManeuversGUI Signal Builder block drive the closed-loop simulation.

Record Coverage Data for the Model

To simulate the model, recording condition, decision, and MCDC coverage for the ShiftLogic controller:

1   Open the example model:

    `sldvdemo_autotrans`

2   In the Simulink Editor, select **Analysis** > **Coverage** > **Settings**.

3   On the **Coverage** pane in the Configuration Parameters dialog box. set the following options:

   •   Select **Enable coverage analysis**.

   •   Select **Subsystem** and click **Select Subsystem.**.

   •   In the Subsystem Selection dialog box, select `ShiftLogic` and click **OK**.

4   Under **Coverage metrics**, select `Modified Condition Decision (MCDC)`.

5   Clear the **Other metrics** if they are selected.

6   In the **Coverage** > **Results** pane of the Configuration Parameters dialog box, set the following options:

- In the **cvdata object name** field, enter `covdata_original_controller` to specify a unique name for the coverage data workspace variable.
- Select **Generate report automatically after analysis**.

7 Click **OK**.

8 Start the simulation of the `sldvdemo_autotrans` model to record the coverage data.

   After the simulation, the coverage report opens. The report indicates that the following coverage is achieved for the ShiftLogic Stateflow chart:

   - Decision: 87% (27/31)
   - Condition: 67% (8/12)
   - MCDC: 33% (2/6) conditions reversed the outcome

   The simulation saves the coverage data in the MATLAB workspace variable `covdata_original_controller`, a `cvtest` object that contains the coverage data.

9 Save the coverage data in a file on the MATLAB path:

   ```
   cvsave('existingcov',covdata_original_controller);
   ```

Find Test Cases for Missing Coverage

To find the missing coverage for the ShiftLogic chart, run a subsystem analysis on that block. Use this technique to focus your analysis on an individual part of the model.

To achieve 100% coverage for the ShiftLogic controller, run a test-generation analysis that uses the existing coverage data.

1 Right-click the ShiftLogic block and select **Design Verifier** > **Options**.

2 In the Configuration Parameters dialog box, under the **Select** tree, choose the **Design Verifier** node. Under **Analysis options** in the **Mode** field, select `Test generation`.

3 Under the **Design Verifier** node, select **Test Generation**. Under **Existing coverage data**, select **Ignore objectives satisfied in existing coverage data**.

4 In the **Coverage data file** field, enter the name of the file containing the coverage data that you recorded during simulation:

```
existingcov.cvt
```

5 Click **Apply** to save these settings.

6 Under the **Select** tree, click **Design Verifier**.

7 On the main **Design Verifier** pane, click **Generate Tests**.

The analysis extracts the Stateflow chart into a new model named `ShiftLogic0`. The analysis analyzes the new model, ignoring the coverage objectives previously satisfied and recorded in the `existingcov.cvt` file.

8 When the test-generation analysis is complete, in the Simulink Design Verifier log window, select **Simulate tests and produce a model coverage report**.

The report indicates that the following coverage is achieved for the ShiftLogic chart in simulation with the test cases generated by Simulink Design Verifier:

- Decision: 84% (26/31)
- Condition: 83% (10/12)
- MCDC: 67% (4/6) conditions reversed the outcome

The Simulink Design Verifier report lists six test cases for the extracted model that satisfy the objectives not covered in the `existingcov.cvt` file.

The Simulink Design Verifier report indicates that two coverage objectives in the Stateflow chart ShiftLogic are proven unsatisfiable. The implicit event `tick` is never `false` because the ShiftLogic chart is updated at every time step. The analysis cannot satisfy condition or MCDC coverage for either instance of the temporal event `after(TWAIT, tick)`.

`after(TWAIT, tick)` is semantically equivalent to

```
Event == tick && temporalCount(tick) >= TWAIT
```

If you move `after(TWAIT, tick)` into the condition, as in

```
[after(TWAIT, tick) && speed < down_th]
```

Simulink Design Verifier determines that `tick` is always `true`, so it only tests the `temporalCount(tick) >= TWAIT` part of `after(TWAIT, tick)`. The analysis is able to find test objectives that satisfy condition and MCDC coverage for `after(TWAIT, tick)`.

Modified Condition and Decision Coverage in Simulink Design Verifier

There are some differences in modified condition and decision coverage (MCDC) reporting and test generation between Simulink Verification and Validation and Simulink Design Verifier .

In this section...
"MCDC Definitions for Simulink Verification and Validation and Simulink Design Verifier" on page 9-15
"Cascaded Logic Blocks in Simulink Verification and Validation and Simulink Design Verifier" on page 9-17

MCDC Definitions for Simulink Verification and Validation and Simulink Design Verifier

There is a difference between the definition of modified condition and decision coverage for model coverage analysis in Simulink Verification and Validation and for test case generation analysis in Simulink Design Verifier. This difference can be seen in analysis results for logical expressions containing a mixture of AND and OR operators, as in this Stateflow transition.
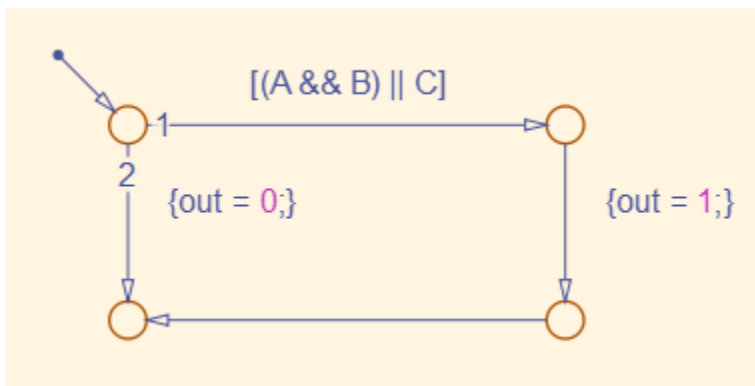
Given that A, B, and C are each separate inputs, there are five possible ways to evaluate the condition on the Stateflow transition, shown in the following table.

	A	B	C	(A && B) || C
1	F	x	F	F
2	F	x	T	T
3	T	F	F	F
4	T	F	T	T
5	T	T	x	T

Satisfying MCDC for a Boolean variable requires a pair of condition evaluations, showing that a change in that variable alone changes the evaluation of the entire expression. In this example, MCDC can be satisfied for C with either the pair 1, 2 or the pair 3, 4. In both of those cases, the value of the expression changed because the value of C changed, while all other variable values stayed the same.

Each pair has a different set of values for A and B which are held constant, but each pair contains one evaluation where C and out are true and one evaluation where C and out are false. To satisfy MCDC for C, Simulink Design Verifier test generation analysis accepts any pair containing one evaluation of true values and one evaluation of false values for C and out. In this example, it accepts not only pair 1, 2 and pair 3, 4 but also pair 1, 4 and pair 2, 3. Simulink Verification and Validation model coverage analysis uses a strict form of MCDC that is satisfied only by pair 1, 2 or by pair 3, 4.

The preceding example assumes that A, B, and C are all separate inputs. When input A is constrained to be the same value as C, as in this model, only a subset of condition evaluations are possible.
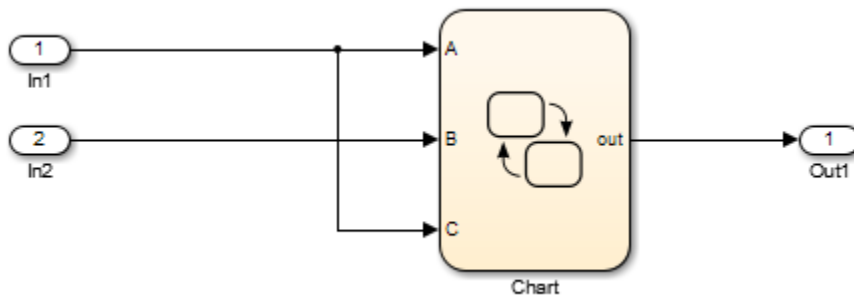
This subset of condition evaluations for the Stateflow transition is shown in the following table.

	A	B	C	(A && B) || C
1	F	x	F	F
4	T	F	T	T
5	T	T	x	T

Evaluations 2 and 3 are no longer possible, so neither pair 1, 2 nor pair 3, 4 is possible. As a result, MCDC for C can no longer be satisfied in Simulink Verification and Validation model coverage analysis. Since pair 1, 4 is still possible, however, Simulink Design Verifier test generation analysis reports that MCDC for C is satisfiable.

The complexity of MCDC analysis for logical expressions with a mixture of AND and OR operators causes this difference between results from Simulink Verification and Validation and Simulink Design Verifier. To minimize this effect, use the IndividualObjectives strategy for test generation analysis in Simulink Design Verifier.

Cascaded Logic Blocks in Simulink Verification and Validation and Simulink Design Verifier

When you generate tests in Simulink Design Verifier to satisfy MCDC objectives for a model containing Simulink logic block cascades, the "Treat Simulink logic blocks as short-circuited" option for your model impacts the tests that Simulink Design Verifier generates. If you enable "Treat Simulink logic blocks as short-circuited", Simulink Design Verifier generates tests for each cascade of logic blocks, but not for individual blocks in the cascades. If you do not enable "Treat Simulink logic blocks as short-circuited", Simulink Design Verifier generates tests for each individual block in the cascades, but not for the cascades of logic blocks. For more information on cascaded logic blocks, see "Logical Operator Cascade Patterns" and "Analyzing MCDC for Cascaded Logic Blocks".

More About

- "MCDC" on page 7-28

Verifying Model Components

What Is Component Verification?

Component Verification Approaches

Component verification lets you test a design component in your model using either of the following approaches:

- **Within the context of the model that contains the component** — Using systematic simulation of closed-loop controllers requires that you verify components within a control system model. Doing so lets you test the control algorithms with your model. This approach is called *system analysis*.

- **As standalone components** — For a high level of confidence in the component algorithm, verify the component in isolation from the rest of the system. This approach is called *component analysis*.

  Verifying standalone components provides three advantages:

  - You can use analysis to focus on portions of the design that you cannot test because of the physical limitations of the system being controlled.

  - You can use this approach for open-loop simulations to test the plant model without feedback control.

  - You can use this approach when the model is unavailable or when you need to simulate a control system model in accelerated mode for performance reasons.

Simulink Design Verifier Tools for Component Verification

By isolating the component to verify, and using tools that Simulink Design Verifier provides, you create test cases that let you expand the scope of the testing for large models. This expanded testing helps you accomplish the following:

- Achieve 100% model coverage — If certain model components do not record 100% coverage, the top-level model cannot achieve 100% coverage. By verifying these components individually, you can create test cases that fully specify the component interface, allowing the component to record 100% coverage.

- Debug the component — To verify that each model component satisfies the specified design requirements, you can create test cases that verify that specific components perform as designed.

- Test the robustness of the component — To verify that a component handles unexpected inputs and calculations properly, you can create test cases that generate data. Then, test the error-handling capabilities in the component.

Functions for Component Verification

The Simulink Design Verifier software provides several functions that facilitate the tasks associated with component verification.

Function	Task
sldvlogsignals	Simulate a Simulink model and log input signals to a Model block in the model. If you modify the test cases in the Signal Builder harness model, use this approach for logging input signals to the harness model itself.
sldvmakeharness	Create a harness model for a component, using logged input signals if specified, or using the default signals. For more information about harness models, see "Simulink Design Verifier Harness Models" on page 13-16.
sldvmergeharness	Merge test cases from several harness models into a single harness model.
sldvextract	Extract an atomic subsystem or atomic subchart into a new model.
sldvruntest	Simulate a model, executing the specified test cases to record model coverage and outport values.
sldvruncgvtest	Invoke the Code Generation Verification (CGV) API, and execute the specified test cases on the generated code for the model. **Note:** To execute a model in different modes of execution, use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see "Programmatic Code Generation Verification".

Component verification functions do not support the following Simulink features:

- Variable-step solvers for sldvruntest
- Component interfaces that contain:
  - Complex signals
  - Variable-size signals

- Array of buses
- Multiword fixed-point data types

Verify a Component for Code Generation

In this section...
"About the Example Model" on page 10-6
"Prepare the Component for Verification" on page 10-8
"Record Coverage for the Component" on page 10-9
"Use Simulink Design Verifier Software to Record Additional Coverage" on page 10-10
"Combine the Harness Models" on page 10-12
"Execute the Component in Simulation Mode" on page 10-13
"Execute the Component in Software-in-the-Loop (SIL) Mode" on page 10-13

About the Example Model

This example uses the `slvnvdemo_powerwindow` model to show how to verify a component in the context of the model that contains that component. As you work through this example, you use the Simulink Design Verifier component verification functions to create test cases and measure coverage for a referenced model. In addition, you can execute the referenced model in both simulation mode and Software-in-the-Loop (SIL) mode using the Code Generation Verification (CGV) API.

Note: You must have the following product licenses to run this example:

- Stateflow
- Embedded Coder®
- Simulink Coder™

The component that you verify is a Model block named `control`. This component resides inside the `power_window_control_system` subsystem in the top level of the `slvnvdemo_powerwindow` model. The `power_window_control_system` subsystem is shown below.
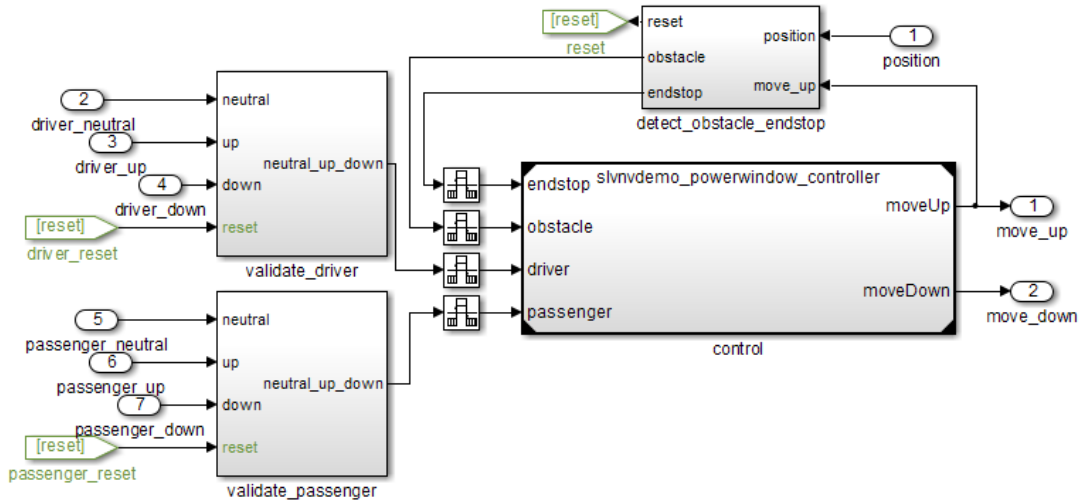
The `control` Model block references the `slvnvdemo_powerwindow_controller` model.
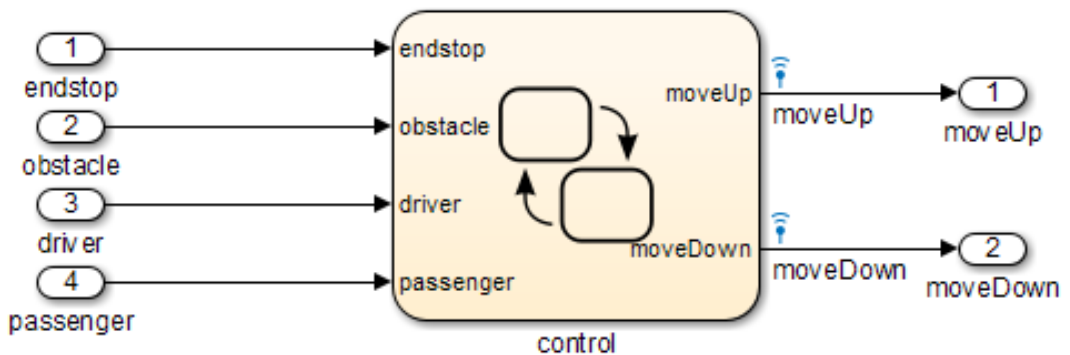
The referenced model contains a Stateflow chart `control`, which implements the logic for the power window controller.
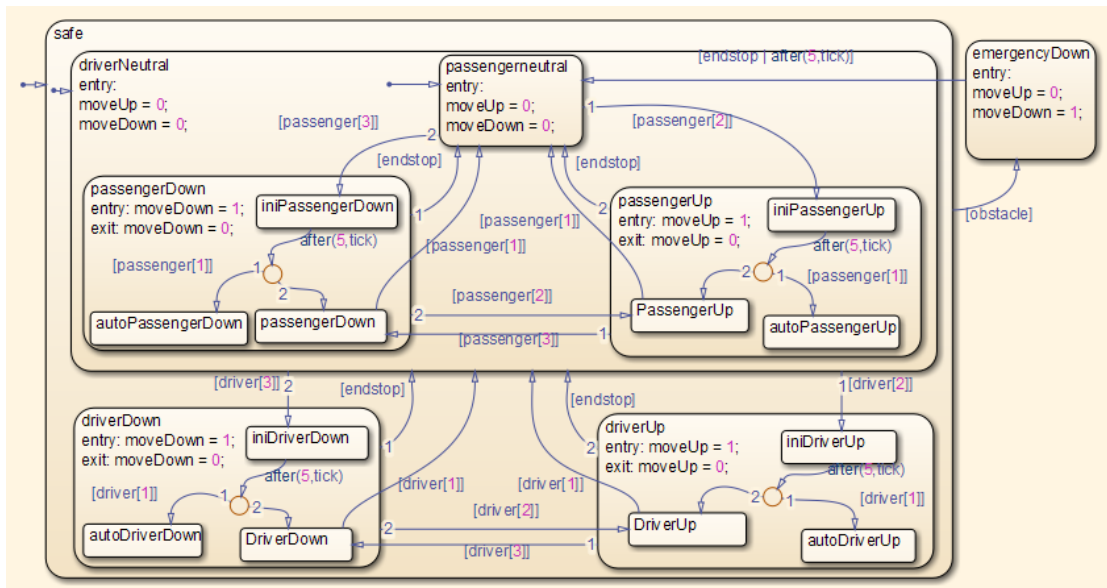
Prepare the Component for Verification

To verify the referenced model slvnvdemo_powerwindow_controller, create a harness model that contains the input signals that simulate the controller in the plant model:

1 Open the slvnvdemo_powerwindow example model and the referenced model:

```
open_system('slvnvdemo_powerwindow');
open_system('slvnvdemo_powerwindow_controller');
```

2 Open the power_window_control_system subsystem in the example model.

The Model block named control in the power_window_control_system subsystem references the component that you verify during this example, slvnvdemo_powerwindow_controller.

3 Simulate the Model block that references the slvnvdemo_powerwindow_controller model and log the input signals to the Model block:

```
loggedSignalsPlant = sldvlogsignals( ...
```

```
'slvnvdemo_powerwindow/power_window_control_system/control');
```

`sldvlogsignals` stores the logged signals in `loggedSignalsPlant`.

4   Generate a harness model with the logged signals:

```
harnessModelFilePath = sldvmakeharness( ...
 'slvnvdemo_powerwindow_controller', loggedSignalsPlant);
```

`sldvmakeharness` creates and opens a harness model named
`slvnvdemo_powerwindow_controller_harness`. The Signal Builder block
contains one test case containing the logged signals.

For more information about harness models, see "Simulink Design Verifier Harness
Models" on page 13-16.

5   For use later in this example, save the name of the harness model:

```
[~, harnessModel] = fileparts(harnessModelFilePath);
```

6   Leave all windows open for the next part of this example.

Next, you will record coverage for the `slvnvdemo_powerwindow_controller` model.

Record Coverage for the Component

Model coverage is a measure of how thoroughly a test case tests a model, and
the percentage of pathways that a test case exercises. To record coverage for the
`slvnvdemo_powerwindow_controller` model:

1   Create a default options object, required by the `sldvruntest` function:

```
runOpts = sldvruntestopts;
```

2   Specify to simulate the model, and record coverage:

```
runOpts.coverageEnabled = true;
```

3   Simulate the referenced model and record coverage:

```
[~, covDataFromLoggedSignals] = sldvruntest( ...
 'slvnvdemo_powerwindow_controller', loggedSignalsPlant, runOpts);
```

4   Display the HTML coverage report:

```
cvhtml('Coverage with Test Cases', covDataFromLoggedSignals);
```

The slvnvdemo_powerwindow_controller model achieved:

- Decision coverage: 40%
- Condition coverage: 35%
- MCDC coverage: 10%

For more information about decision coverage, condition coverage, and MCDC coverage, see "Types of Model Coverage" in the Simulink Verification and Validation documentation.

Because you did not achieve 100% coverage for the slvnvdemo_powerwindow_controller model, next, you will analyze the model to record additional coverage and create additional test cases.

Use Simulink Design Verifier Software to Record Additional Coverage

You can use Simulink Design Verifier to analyze the slvnvdemo_powerwindow_controller model and collect coverage. You can specify that the analysis ignore any previously satisfied objectives and record additional coverage.

To record additional coverage for the model:

1 Save the coverage data that you recorded for the logged signals in a file:

```
cvsave('existingCovFromLoggedSignal', covDataFromLoggedSignals);
```

2 Create a default options object for the analysis:

```
opts = sldvoptions;
```

3 Specify that the analysis generate test cases to record decision, condition, and modified condition/decision coverage:

```
opts.ModelCoverageObjectives = 'MCDC';
```

4 Specify that the analysis ignore objectives that you satisfied when you logged the signals to the Model block:

```
opts.IgnoreCovSatisfied = 'on';
```

5 Specify the name of the file that contains the satisfied objectives data:

```
opts.CoverageDataFile = 'existingCovFromLoggedSignal.cvt';
```

6 Specify that the analysis not display unsatisfiable objectives in the Diagnostic Viewer:

```
opts.DisplayUnsatisfiableObjectives = 'off';
```

For this example, the focus is on satisfying as many objectives as possible.

7 Specify that the analysis create long test cases that satisfy several objectives:

```
opts.TestSuiteOptimization = 'LongTestcases';
```

Creating a smaller number of test cases each of which satisfies multiple test objectives saves time when you execute the generated code in the next section.

8 Specify to create a harness model that references the component using a Model block:

```
opts.saveHarnessModel = 'on';
opts.ModelReferenceHarness = 'on';
```

The harness model that you created from the logged signals in "Prepare the Component for Verification" on page 10-8 uses a Model block that references the slvnvdemo_powerwindow_controller model. The harness model that the analysis creates must also use a Model block that references slvnvdemo_powerwindow_controller. You can append the test case data to the first harness model, creating a single test suite.

9 Analyze the model using Simulink Design Verifier:

```
[status, fileNames] = sldvrun('slvnvdemo_powerwindow_controller', ...
 opts, true);
```

The analysis creates and opens a harness model slvnvdemo_powerwindow_controller_harness. The Signal Builder block contains one long test case that satisfies 74 test objectives.

You can combine this test case with the test case that you created in "Prepare the Component for Verification" on page 10-8, to record additional coverage for the slvnvdemo_powerwindow_controller model.

10 Save the name of the new harness model and open it:

```
[~, newHarnessModel] = fileparts(fileNames.HarnessModel);
open_system(newHarnessModel);
```

Next, you will combine the two harness models to create a single test suite.

Combine the Harness Models

You created two harness models when you:

- Logged the signals to the control Model block that references the `slvnvdemo_powerwindow_controller` model.

- Analyzed the `slvnvdemo_powerwindow_controller` model.

If you combine the test cases in both harness models, you can record coverage that gets you closer to achieving 100% coverage:

1 Combine the harness models by appending the most recent test cases to the test cases for the logged signals:

```
sldvmergeharness(harnessModel, newHarnessModel);
```

The Signal Builder block in the `slvnvdemo_powerwindow_controller_harness` model now contains both test cases.

2 Log the signals to the harness model:

```
loggedSignalsMergedHarness = sldvlogsignals(harnessModel);
```

3 Use the combined test cases to record coverage for the `slvnvdemo_powerwindow_controller_harness` model. First, configure the options object for `sldvruntest`:

```
runOpts = sldvruntestopts;
runOpts.coverageEnabled = true;
```

4 Simulate the model and record and display the coverage data:

```
[~, covDataFromMergedSignals] = sldvruntest( ...
 'slvnvdemo_powerwindow_controller', loggedSignalsMergedHarness, ...
 runOpts);
cvhtml('Coverage with Merged Test Cases', covDataFromMergedSignals);
```

The `slvnvdemo_powerwindow_controller` model now achieves:

- Decision coverage: 100%
- Condition coverage: 80%
- MCDC coverage: 60%

Execute the Component in Simulation Mode

To verify that the generated code for the model produces the same results as simulating the model, use the Code Generation Verification (CGV) API methods.

Note: To execute a model in different modes of execution, use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see "Programmatic Code Generation Verification" in the Embedded Coder documentation.

When you perform this procedure, the simulation compiles and executes the model code using both test cases.

1 Create a default options object for `sldvruncgvtest`:

```
runcgvopts = sldvruntestopts('cgv');
```

2 Specify to execute the model in simulation mode:

```
runcgvopts.cgvConn = 'sim';
```

3 Execute the `slvnv_powerwindow_controller` model using the two test cases and the `runcgvopts` object:

```
cgvSim = sldvruncgvtest('slvnvdemo_powerwindow_controller', ...
 loggedSignalsMergedHarness, runcgvopts);
```

These steps save the results in the workspace variable `cgvSim`.

Next, you will execute the same model with the same test cases in Software-in-the-Loop (SIL) mode and compare the results from both simulations.

For more information about Normal simulation mode, see "Execute the Model" in the Embedded Coder documentation.

Execute the Component in Software-in-the-Loop (SIL) Mode

When you execute a model in Software-in-the-Loop (SIL) mode, the simulation compiles and executes the generated code on your host computer.

In this section, you execute the `slvnvdemo_powerwindow_controller` model in SIL mode and compare the results to the previous section, when you executed the model in simulation mode.

1 Specify to execute the model in SIL mode:

```
runcgvopts.cgvConn = 'sil';
```

2 Execute the slvnv_powerwindow_controller model using the two test cases and the runcgvopts object:

```
cgvSil = sldvruncgvtest('slvnvdemo_powerwindow_controller', ...
 loggedSignalsMergedHarness, runcgvopts);
```

The workspace variable cgvSil contains the results of the SIL mode execution.

3 Compare the results in cgvSil to the results in cgvSim, created from the simulation mode execution. Use the compare (cgv.CGV) method to compare the results from the two simulations:

```
for i=1:length(loggedSignalsMergedHarness.TestCases)
    simout = cgvSim.getOutputData(i);
    silout = cgvSil.getOutputData(i);
    [matchNames, ~, mismatchNames, ~ ] = ...
    cgv.CGV.compare(simout, silout);
end
```

4 Display the results of the comparison in the MATLAB Command Window:

```
fprintf(['\nTest Case(%d):%d Signals match, %d Signals mismatch\r'],...
 i, length(matchNames), length(mismatchNames));
```

As expected, the results of the two simulations match.

For more information about Software-in-the-Loop (SIL) simulations, see "What Are SIL and PIL Simulations?" in the Embedded Coder documentation.

Considering Specified Minimum and Maximum Values for Inputs During Analysis

Minimum and Maximum Input Constraints

In this section...
"Simulink Design Verifier Support for Specified Input Minimum and Maximum Values" on page 11-2
"Limitations of Simulink Design Verifier Support for Specified Minimum and Maximum Values" on page 11-3

When creating a model, you can specify minimum and maximum values on input ports to mimic environmental constraints as part of your design. The Simulink Design Verifier analysis can automatically consider these values as constraints for:

- Design error detection
- Test case generation
- Property proving

Specifying minimum and maximum input values is similar to using the Test Condition block to constrain signals for test case generation or the Proof Assumption block to constrain signals for property proving. The Test Condition and Proof Assumption blocks capture the analysis constraints. The Simulink Design Verifier software can also consider the design constraints captured in the Inport block minimum and maximum parameters as constraints for analysis.

Note: For more information about signal values, see "Signal Values" in the Simulink documentation.

Simulink Design Verifier Support for Specified Input Minimum and Maximum Values

By default, Simulink Design Verifier considers any minimum and maximum input values specified for Inport blocks in your model. To enable this capability:

1   In the model window, select **Analysis** > **Design Verifier** > **Options**.
2   On the **Design Verifier** pane, select the **Use specified input minimum and maximum values** parameter.
3   After the analysis completes, to view the design minimum and maximum constraints for your model, click **Generate detailed analysis reports**.

The constraints are listed in the **Analysis Information** chapter of the Simulink Design Verifier report.

Limitations of Simulink Design Verifier Support for Specified Minimum and Maximum Values

Simulink Design Verifier support for specified minimum and maximum values has the following limitations:

- The analysis considers specified minimum and maximum values on root-level Inport blocks only. The analysis ignores minimum and maximum values specified on other Simulink blocks.

More About

- "Signal Ranges"

Specify Input Range for Inport Blocks

You can specify the output minimum and maximum values on Inport blocks in the Block Parameter dialog box, on the **Signal Attributes** tab. Set the following parameters:

- **Minimum**
- **Maximum**

The following example model restricts the signals from two Inport blocks:

- Input1 block: Minimum: 1, Maximum: 5
- Input2 block: Minimum: –1, Maximum: 1
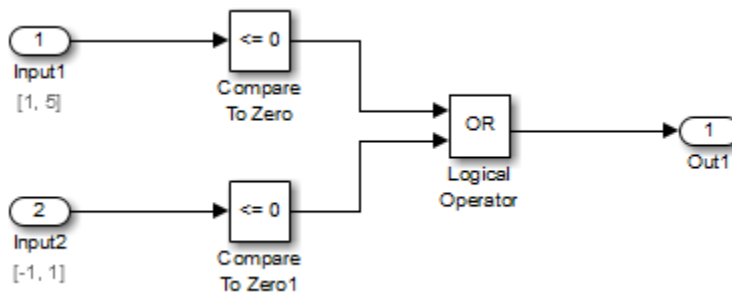
When you analyze this model using Simulink Design Verifier, the analysis uses the minimum and maximum values as constraints. The analysis produces the following results:

- The output from Input1 is never less than 0, so the first input to the Logical Operator block is never `false`. The objective that the first input to the Logical Operator equals `false` is therefore unsatisfiable.
- The Logical Operator block cannot achieve 100% MCDC coverage because the condition where the first input is `false` never occurs.

The detailed analysis report indicates the values that it used as constraints for Input1 and Input2.

Specified Input Ranges in sldvData Fields

When you analyze a model, Simulink Design Verifier generates a data file when it completes its analysis. The data file is a MAT-file that contains an `sldvData` structure. The `sldvData` structure stores all the data that the software gathers and produces during the analysis. You can use the data file to customize your own analysis or to generate a custom report.

If your model contains specified minimum and maximum values on the input ports, the `sldvData` structure contains information about those values. For example, after analyzing the `ex_minmax_on_inports` model in "Specify Input Range for Inport Blocks" on page 11-4, the data file contains the following values:

- For the Input1 block:

  ```
  sldvData.Constraints.DesignMinMax(1).value{1}.low

  ans =

       1

  sldvData.Constraints.DesignMinMax(1).value{1}.high

  ans =

       5
  ```

- For the Input2 block:

  ```
  sldvData.Constraints.DesignMinMax(2).value{1}.low

  ans =

      -1

  sldvData.Constraints.DesignMinMax(2).value{1}.high

  ans =

       1
  ```

Specify Input Range for Simulink.Signal Objects

Using the Model Explorer, in the model workspace, you can specify minimum and maximum values on `Simulink.Signal` objects associated with input signals.

The following example model uses the `Simulink.Signal` objects associated with the input signals `a` and `b` to restrict the signal values:

- Signal `a`: Minimum: 1, Maximum: 5
- Signal `b`: Minimum: –1, Maximum: 1
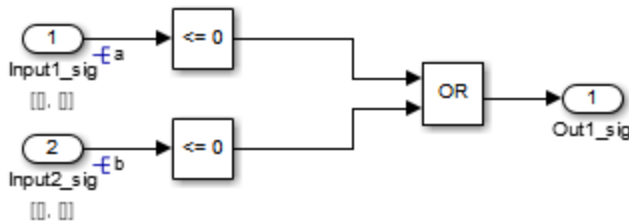
When you analyze this model, the results are the same as if you specified the minimum and maximum values on the input ports.

Specifying Signal Ranges on Inport Blocks and Signals If you specify ranges on the Inport blocks *and* on the signals, the analysis considers the tightest range. For example, if you specify a range of `4..12` on an input port and a range of `1..8` on the signal from the input port, the analysis considers the range `4..8`.

Specify Input Range for Stateflow Data Objects

Using the Model Explorer, you can specify ranges on data objects that are directly connected to the root-level input ports for a Stateflow chart.

In the following example model, the Stateflow chart named Chart has a data object x whose range you specified as 0 < x < 10. In this chart, x must be greater than 15 to trigger the transition from low to high.
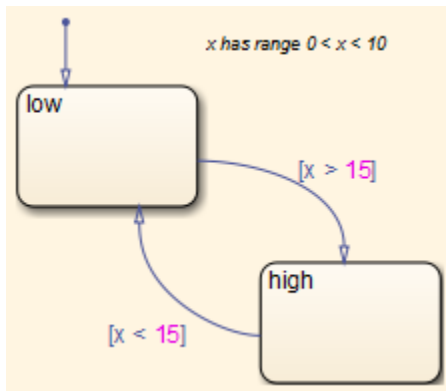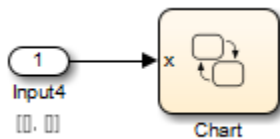
The value of x must fall from 0 through 10, so the transition condition [x > 15] is never true. The transition from low to high never occurs. Because the high state is never entered, the transition condition [x < 15] is never tested, and the transition from high to low never occurs; the chart is always in the low state.

When you analyze this model, the following objectives are proven unsatisfiable:

- The high state is never entered.
- The transition condition [x > 15] is always false, never true.
- The condition [x < 15] is never tested, so it is never true or false.

The analysis report indicates the values that it used as constraints for x: [0, 10].

Specify Input Range for Subsystems

The Simulink Design Verifier software considers specified input minimum and maximum values as constraints only at the top level of a model. You can specify minimum and maximum values on Input ports on subsystems, but when you analyze the top-level model, the software ignores those values.

However, when you perform the subsystem analysis, the software considers specified minimum and maximum values on the input ports of the subsystem.

For example, consider the following model, and its subsystem.
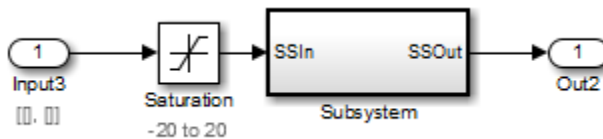
In Subsystem, the specified minimum and maximum values for input port SSIn are –10 and 10, respectively. The lower and upper limits for the Saturation block are –15 and 15, respectively.
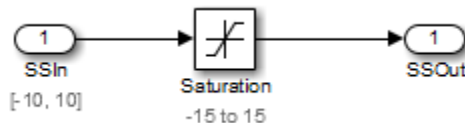
If you right-click Subsystem in the top-level model and select **Design Verifier > Generate Tests for Subsystem**, the analysis considers the specified minimum and maximum values as constraints on the SSIn port.

Constraints

Design Min Max Constraints

Name	Design Min Max Constraint
SSIn	[-10, 10]

The analysis identifies two unsatisfiable objectives:

- input > lower limit F: The input is always greater than the lower limit on the Saturation block (–15).
- input >= upper limit T: The input is never greater than or equal to the upper limit (15).

If you analyze the model that contains Subsystem, the analysis does not consider the values specified on the input port SSIn in the subsystem. The analysis considers only the root-level input ports at the respective level of the hierarchy for analysis.

Specify Input Range for Global Data Stores

A *data store* is a repository to which you can write data and from which you can read data, without having to connect an input or output signal directly to the data store. You create a data store using a `Data Store Memory` block or a `Simulink.Signal` object. You can specify minimum and maximum values for any data store.

As described in "Extract Subsystems for Analysis" on page 14-16, during subsystem analysis, Simulink Design Verifier creates an input port to mimic the execution context for a global data store. If the data store has specified minimum and maximum values, those values are assigned as minimum and maximum values on the new input port. The input minimum and maximum values are used as subsystem-level analysis constraints.

In the following example model, data store `A` has a minimum value of 0 and a maximum value of 10.

The atomic subsystem reads from the data store and checks to see if the input is less than `0`. The Compare To Zero block outputs 1 if the input is less than 0, and outputs 0 if the input is greater than or equal to 0. The Test Objective block checks to see if the output is ever 1.
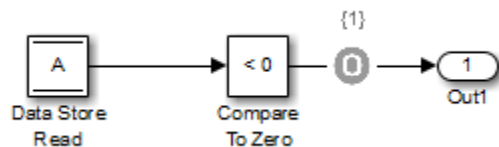
If you right-click Subsystem in the top-level model and select **Design Verifier** > **Generate Tests for Subsystem**, the analysis considers the constraints for data store `A` to be [0, 10].

The analysis does not satisfy the objective specified in the Test Objective block. The input is always greater than or equal to 0, so the output from the Compare To Zero block is always 0.

Specify Input Range for a Bus Element

When you define a bus, you can specify minimum and maximum values for the elements in the bus. Simulink Design Verifier considers these minimum and maximum values when analyzing subsystems and models that use the bus as an input signal.

Consider a subsystem that inputs a bus of three fields, each with a defined minimum and maximum. You can view this subsystem by entering

```
open_system(fullfile(docroot,'toolbox','sldv','examples',
'sldvBusMinMaxExample'))
```
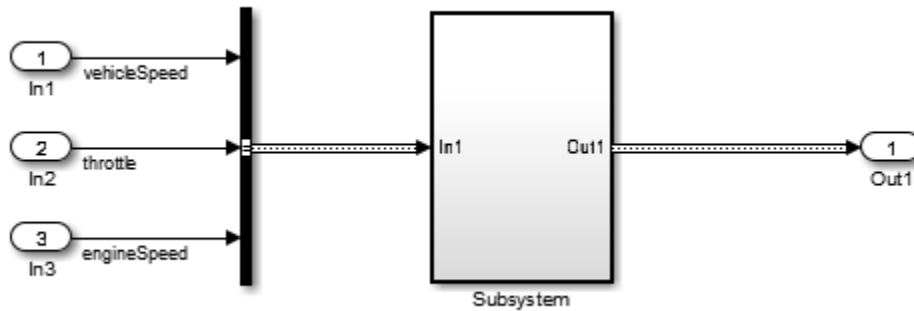

Bus element	Bus element minimum	Bus element maximum
vehicleSpeed	0	125
throttle	0	100
engineSpeed	0	7600

The subsystem has test objectives confirming that each element does not exceed a constant. The `vehicleSpeed` signal is limited to a maximum value lower than the test objective.
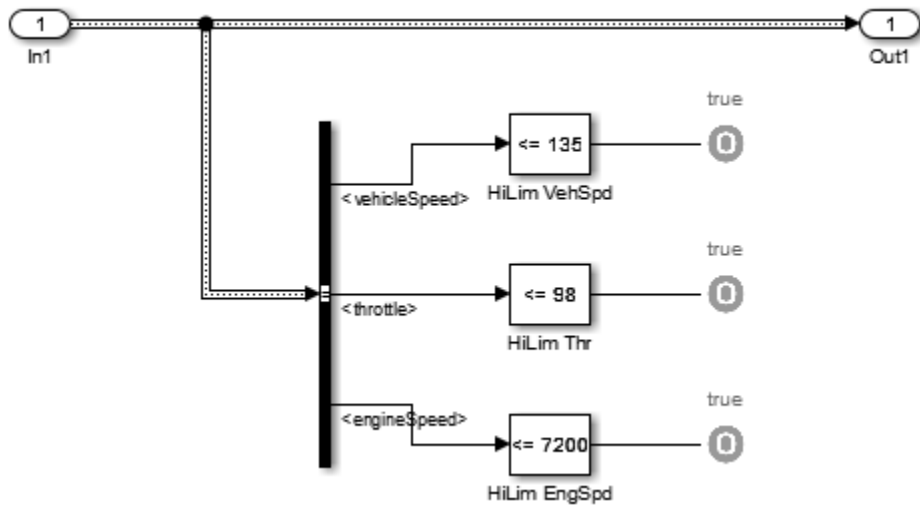
Set the current folder to a writable folder. Right-click Subsystem in the top-level model and select **Design Verifier > Generate Tests for Subsystem**. The Condition Objective for testing `vehicleSpeed > 135` is not satisfiable due to the maximum specification on the `vehicleSpeed` element.

Proving Properties of a Model

What Is Property Proving?

A *property* is a requirement that you model in Simulink or Stateflow, or using MATLAB Function blocks. A property can be a simple requirement, such as a signal in your model that must attain a particular value or range of values during simulation.

A property can also be a requirement on the model that involves a number of input and output signals modeled as a logical expression that needs to be proved.

The Simulink Design Verifier software performs a formal analysis of your model to prove or disprove the specified properties. After completing the analysis, the software offers several ways for you to review the results:

- Highlighted on the model
- A harness model with test cases
- A detailed HTML report

Proof Blocks

The Simulink Design Verifier software provides two blocks so you can specify property proofs in your Simulink models:

- `Proof Objective` — Define the values of a signal to prove
- `Proof Assumption` — Constrain the values of a signal during a proof

Note: Blocks from the Model Verification library in the Simulink software behave like Proof Objective blocks during Simulink Design Verifier proofs. You can use `Assertion` blocks and other Model Verification blocks to specify properties of your model. For more information about these blocks, see "Model Verification" in the Simulink documentation.

Proof Functions

The Simulink Design Verifier software provides two Stateflow and MATLAB for code generation functions to specify property proving for a Simulink model or Stateflow chart:

- `sldv.prove` — Specifies a proof objective
- `sldv.assume` — Specifies a proof assumption

These functions:

- Identify mathematical relationships for proving properties in a form that can be more natural than using block parameters
- Support specifying multiple objectives, assumptions, or conditions without complicating the model.
- Provide access to the power of MATLAB.
- Support separation of verification and model design.

For an example of how to use these proof functions, see the `sldv.prove` reference page.

Note: Simulink Design Verifier blocks and functions are saved with a model. If you open the model on a MATLAB installation that does not have a Simulink Design Verifier license, you can see the blocks and functions, but they do not produce results.

Workflow for Proving Model Properties

To prove properties of your design model, use the following workflow:

1 Determine the verification objectives for your design model, e.g., based on your requirements specifications.

2 Instrument your design model to specify proof objectives and proof assumptions.

- For simple properties, instrument your model with blocks or MATLAB functions that specify the proof objectives.

- For system-level properties, construct a verification model that contains a Model block that references the design model and define the properties on the design model interface using the same inputs and outputs.

3 Define analysis constraints using the Proof Assumption block or `sldv.assume`. These constraints apply to all enabled proof objectives.

Note: The proof assumptions are applied to all enabled proof objectives. Make sure that you do not specify any contradictory assumptions because that might invalidate the entire analysis.

4 Specify options that control how Simulink Design Verifier proves the properties of your model.

5 Execute the Simulink Design Verifier analysis and review the results.

For an exercise that demonstrates this workflow, see "Prove Properties in a Model" on page 12-5.

Prove Properties in a Model

In this section...

About This Example

The following sections describe a Simulink model, for which you prove a property that you specify using a Proof Objective block. This example demonstrates the property-proving capabilities of Simulink Design Verifier.

In this example, you perform the following tasks.

Task	Description	See...
1	Construct the example model.	"Construct Example Model" on page 12-6
2	Verify that your model is compatible with Simulink Design Verifier.	"Check Compatibility of Example Model" on page 12-7
3	Add a Proof Objective block to your model to prepare for its proof.	"Instrument Example Model" on page 12-8

Task	Description	See...
4	Configure Simulink Design Verifier to prove properties.	"Configure Property-Proving Options" on page 12-9
5	Prove a property of your model.	"Analyze Example Model" on page 12-10
6	Review the analysis results.	"Review Analysis Results" on page 12-10
7	Add proof assumptions to specify analysis constraints.	"Customize Example Proof" on page 12-19
8	Prove a property of the customized model and interpret the results.	"Reanalyze Example Model" on page 12-20

Construct Example Model

Construct a Simulink model to use in this example:

1   Create an empty Simulink model.

2   Copy the following blocks into your empty model window:

  - From the Sources library, an `Inport` block to initiate the input signal whose value Simulink Design Verifier controls

  - From the Logic and Bit Operations library, a `Compare To Zero` block to provide simple logic

  - From the Sinks library, an `Outport` block to receive the output signal

3   Connect these blocks such so your model appears similar to the following model:
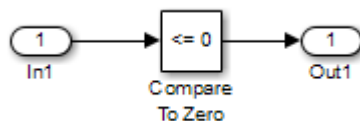
4   In the model window, select **Simulation** > **Model Configuration Parameters**.

5   On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Solver** category. On the right side, under **Solver options**:

  - Set the **Type** option to `Fixed-step`.

  - Set the **Solver** option to `Discrete (no continuous states)`.

The Simulink Design Verifier can analyze only models that use a fixed-step solver.

6 Click **OK** to save your changes and close the Configuration Parameters dialog box.

7 Save your model with the name `ex_property_proving_example_basic`.

Check Compatibility of Example Model

Every time Simulink Design Verifier software analyzes a model, before the analysis begins, the software performs a compatibility check. If your model is not compatible, the software cannot analyze it.

You can also make sure you model is compatible with Simulink Design Verifier before you start the analysis:

1 Open the `ex_property_proving_example_basic` model.

2 In the model window, select **Analysis** > **Design Verifier** > **Check Compatibility** > **Model**.

The Simulink Design Verifier software displays the log window, which states whether or not your model is compatible.

The model you just created is compatible.
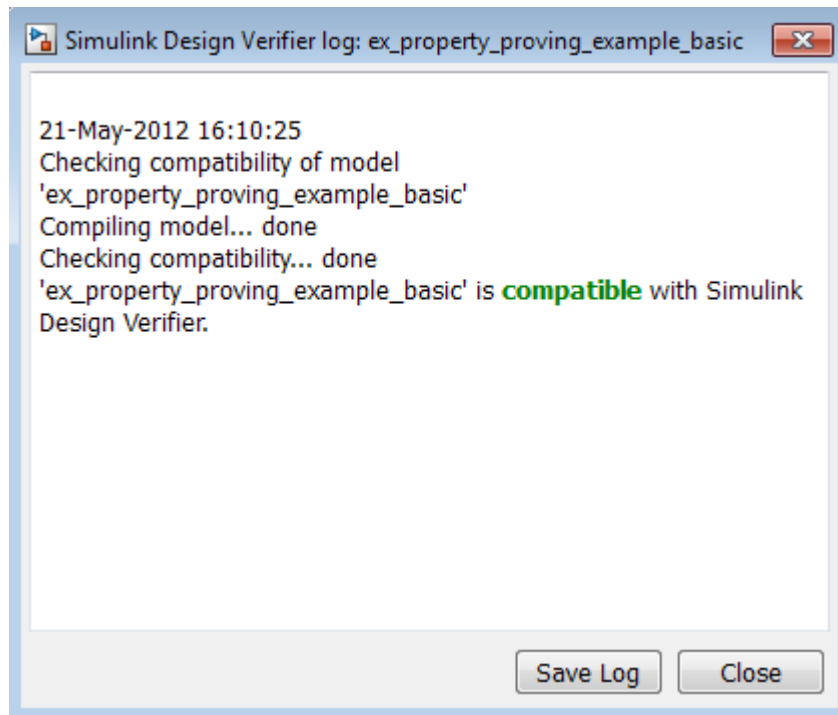
What If a Model Is Partially Compatible?

If the compatibility check indicates that your model is partially compatible, your model contains at least one object that Simulink Design Verifier does not support. You can analyze a partially compatible model, but, by default, unsupported objects are stubbed out. The results of the analysis may be incomplete. For detailed information about automatic stubbing, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

Instrument Example Model

Prepare your example model so that you can prove its properties with Simulink Design Verifier. Specifically, instrument the model by adding and configuring a `Proof Objective` block:

1 In the MATLAB Command Window, enter `sldvlib`.

The Simulink Design Verifier library appears.

2 Open the Objectives and Constraints sublibrary.

3 Copy the Proof Objective block to your model and insert it between the Compare To Zero and Outport blocks.

4 In your model, double-click the Proof Objective block.

The Proof Objective block parameters dialog box opens.

5 In the **Values** box, enter 1.

The Simulink Design Verifier software will attempt to prove that the signal output by the Compare To Zero block always attains this value for any signals that it receives.

6 Click **OK** to apply your changes and close the Proof Objective block parameters dialog box.
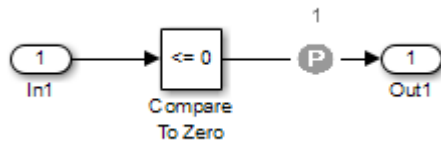
7 Save your model and keep it open.

Configure Property-Proving Options

Configure Simulink Design Verifier to prove properties of the ex_property_proving_example_basic model that you instrumented:

1 Open the ex_property_proving_example_basic model.

2 In your Simulink model window, select **Analysis** > **Design Verifier** > **Options**.

3 On the left side of the Configuration Parameters dialog box, in the **Select** tree, select the **Design Verifier** category. Under **Analysis options** on the right side, set the **Mode** parameter to Property proving.

4 Click **OK** to apply your changes and close the Configuration Parameters dialog box.

> **Note:** On the **Property Proving** pane, you can optionally specify values for other parameters that control how Simulink Design Verifier proves properties of your model. For more information, see "Design Verifier Pane: Property Proving" on page 15-64.

5 Save the `ex_property_proving_example_basic` model.

Analyze Example Model

To analyze the `ex_property_proving_example_basic` model, in the model window, select **Analysis** > **Design Verifier** > **Prove Properties** > **Model**. The Simulink Design Verifier software begins a property-proving analysis.

During the analysis, the log window shows the progress of the analysis. It displays information such as the number of objectives processed and which objectives were satisfied or falsified.

To terminate the analysis at any time, in the log window, click **Stop**.

Review Analysis Results

When the analysis is complete, the log window displays the following options for reviewing the results:

- Highlight the analysis results on the model
- Generate a detailed HTML analysis report
- Create a harness model with test cases
- Simulate the test cases created by the model and produce a model coverage report

You can also view the Simulink Design Verifier data file. For detailed information about the data file, see "Simulink Design Verifier Data Files" on page 13-9.

The following sections describe how you can review the analysis results:

- "Review Results on Model" on page 12-11
- "Review Detailed Analysis Report" on page 12-13
- "Review Harness Model" on page 12-15
- "Simulate Model with Counterexample" on page 12-16
- "Review Analysis Results in the Model Explorer" on page 12-18

Review Results on Model

You can review the analysis results at a glance by viewing the blocks that are highlighted in the model window. The highlighting can have four colors:

- Green — The analysis proved all the proof objectives valid.
- Red — The analysis disproved a proof objective and generated a counterexample that falsified that objective.
- Orange — The analysis disproved a proof objective, but it could not generate a counterexample or the proof objective remained undecided. This result occurs due to:

  - A proof objective on a signal whose value the software cannot control, for example, a Constant block
  - A proof objective that depends on nonlinear computation
  - A proof objective that creates an arithmetic error, such as division by zero
  - Automatic stubbing being enabled, and the analysis encountering an unsupported block whose operation it does not understand but that the analysis requires to generate the counterexample
  - The analysis timing out
  - Limitations of the analysis engine
- Gray — The model object was not part of the analysis.

Highlight the analysis results on the example model:

1   In the log window for the `ex_property_proving_example_basic` analysis, click **Highlight analysis results on model**.
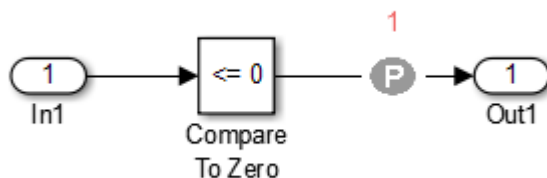
The Proof Objective block is highlighted in red, which indicates that a proof objective was falsified with a counterexample.

12-11

The Simulink Design Verifier Results window appears.
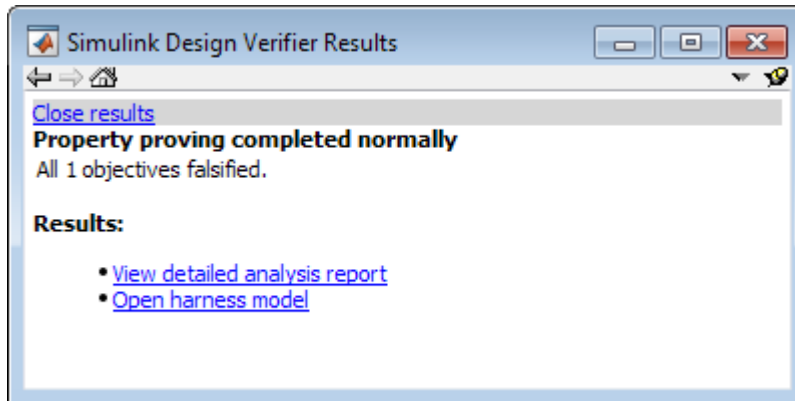
As you click objects in the model, this window changes to display detailed analysis results for that object.
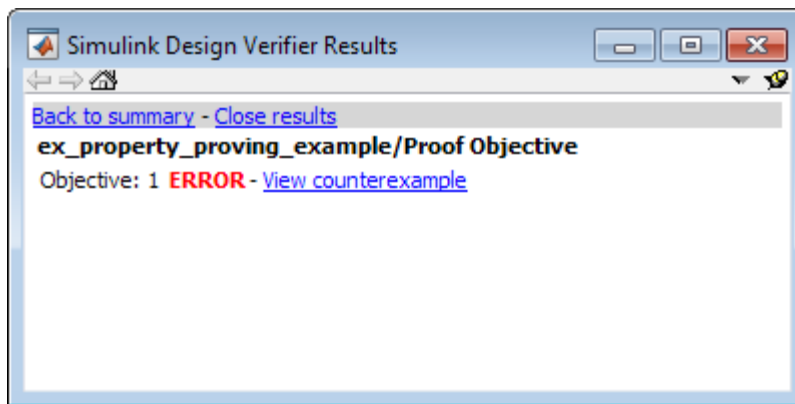
Tip By default, the Simulink Design Verifier Results window is always the topmost visible window. To allow the window to move behind other window, click 📷 and clear Always on top.

---

2 Click the highlighted Proof Objective block.

The Simulink Design Verifier Results window indicates that the proof objective that the output signal from the Compare to Zero was not 1 was disproved with a counterexample.

Review Detailed Analysis Report

To create a detailed HTML analysis report:

1   In the Simulink Design Verifier log window, click **Generate detailed analysis report**.

    The HTML report opens in a browser window.

2   The report includes the following **Table of Contents**. Click a hyperlink to navigate to particular section in the report.

Table of Contents

1. Summary
2. Analysis Information
3. Proof Objectives Status
4. Properties

3   In the **Table of Contents**, click Summary.

Chapter 1. Summary

Analysis Information

Model:	ex_property_proving_example
Mode:	PropertyProving
Status:	Completed normally
Analysis Time:	0s

Objectives Status

Number of Objectives: **1**
Objectives Falsified with Counterexamples: 1

The Summary provides an overview of the analysis results, and it indicates that Simulink Design Verifier identified a counterexample that falsifies an objective in your model.

4   Scroll back to the top of the browser window. In the **Table of Contents**, click `Proof Objectives Status`.

Objectives Falsified with Counterexamples

#:	Type	Model Item	Description	Counterexample
1	Custom Proof Objective	Proof Objective	Objective: 1	1

The Objectives Falsified with Counterexamples table lists the proof objectives that Simulink Design Verifier disproved using a counterexample that it generated. You can locate the objective in your model window by clicking `Proof Objective`; the software highlights the corresponding Proof Objective block in your model window.

5   In the Objectives Falsified with Counterexamples table, under the **Counterexample** column, click `1`.

Proof Objective

Summary

Model Item: Proof Objective
Property: Objective: 1
Status: Falsified

Counter Example

Time	0
Step	1
In1	1

This section displays information about proof objective 1 and provides details about the counterexample that Simulink Design Verifier generated to disprove that objective. In this counterexample, a signal value of 99 falsifies the objective that you specified using the Proof Objective block. That is, 99 is not less than or equal to 0, which causes the Compare To Zero block to return 0 (false) instead of 1 (true).

Review Harness Model

Create a harness model with counterexamples that falsify the proof objectives in your model:

1   In the Simulink Design Verifier log window, click **Create harness model**.

    The software creates a harness model named
    `ex_property_proving_example_basic_harness`.
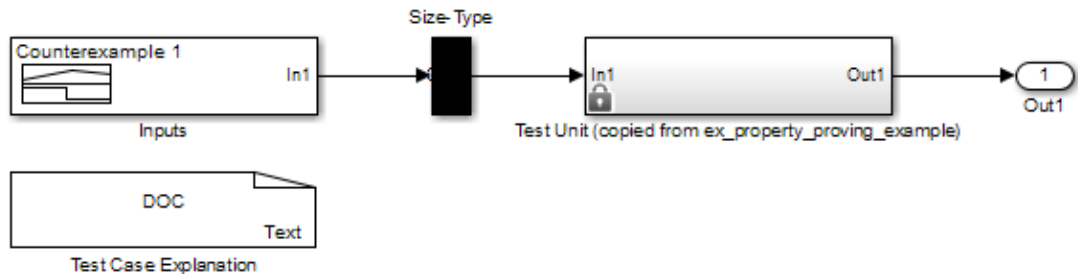
The harness model contains the following items:

- `Signal Builder` block named `Inputs` — A group of signals that falsify proof objectives.
- `Subsystem` block named `Test Unit` — A copy of your model.
- `DocBlock` named `Test Case Explanation` — A textual description of the counterexamples that the analysis generates.
- A Size-Type block — A subsystem that transmits signals from the Inputs block to the Test Unit block. This block verifies that the size and data type of the signals are consistent with the Test Unit block.

2   Double-click the Inputs block.
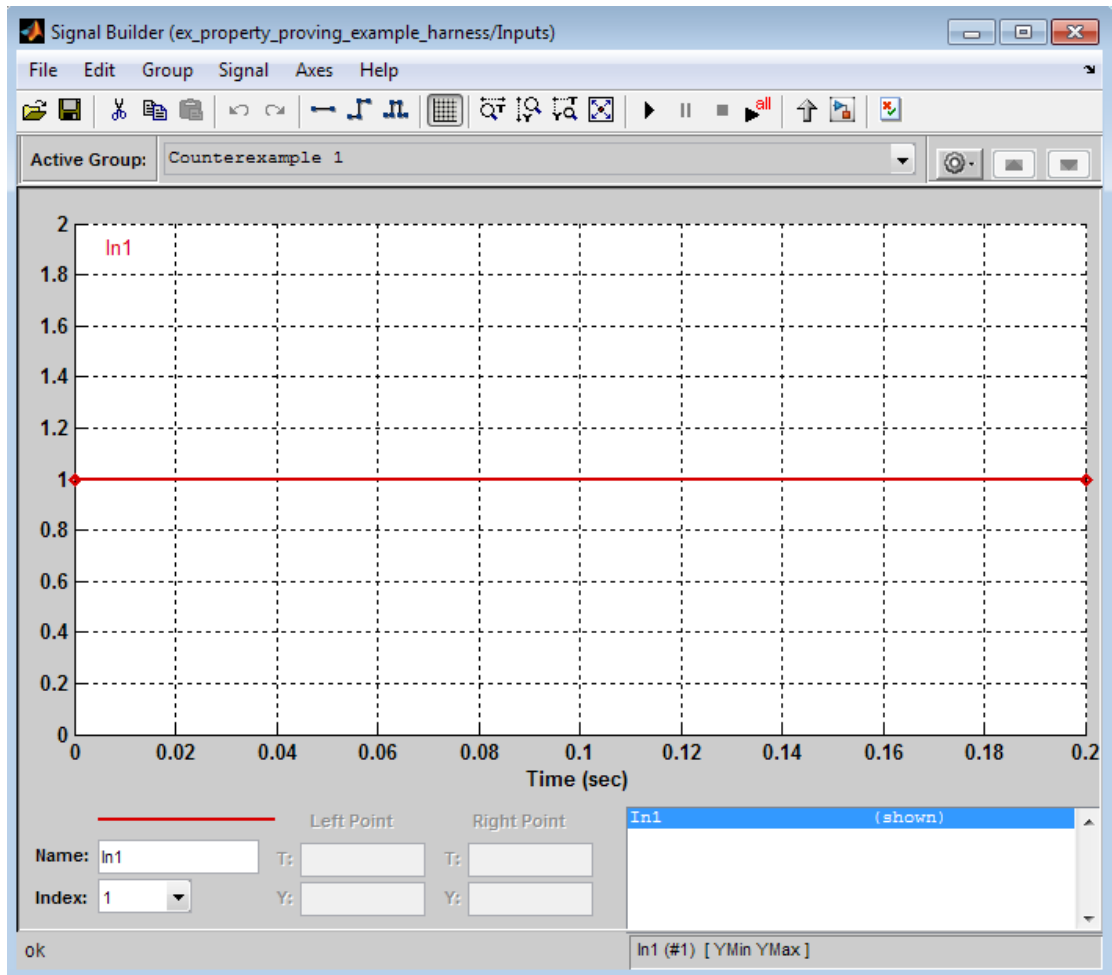
The input signal 1 causes the output of the Compare to Zero block to be 0. This counterexample violates the proof objective that specifies that the output of the Compare to Zero block be 1.

Simulate Model with Counterexample

Simulate the harness model to observe the counterexample that falsifies the proof objective in your model:

1   In the `ex_property_proving_example_basic` model window, select **View** > **Library Browser**

2   From the Sinks library, copy a `Scope` block into your harness model window. The Scope block allows you to see the value of the signal output by the Compare To Zero block in your model.

3   In your harness model window, connect the output signal of the Test Unit subsystem to the Scope block.
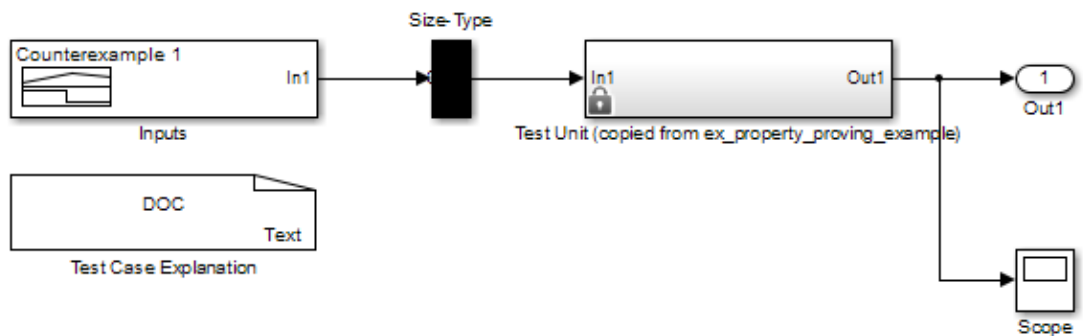
4   In your harness model window, select **Simulation** > **Run** to begin the simulation.

The Simulink software simulates the harness model.

5   In your harness model window, double-click the Scope block to open its display window.
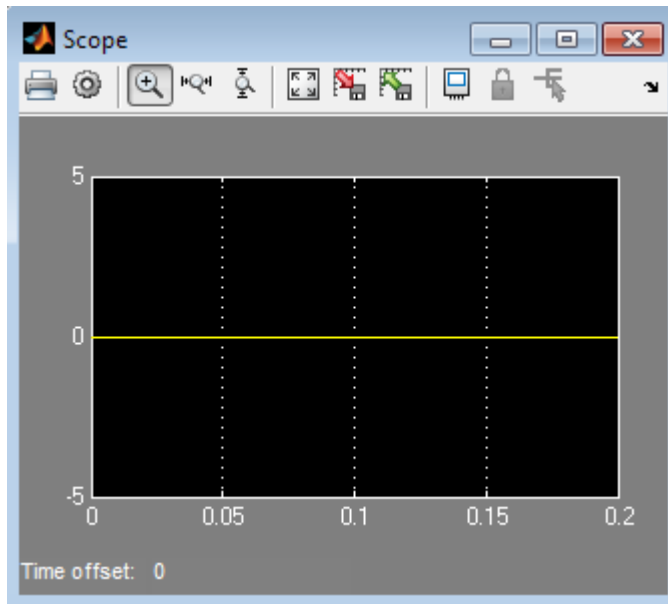
The Scope block displays the value of the signal output by the Compare To Zero block in your model. In this example, the Compare To Zero block returns 0 (false) throughout the simulation, which falsifies the proof objective that the output of the Compare to Zero block be 1 (true). The counterexample that the Signal Builder block supplies falsifies the proof objective.

Review Analysis Results in the Model Explorer

As long as your model remains open, you can view the results of your most recent Simulink Design Verifier analysis results in the Model Explorer.

In the Simulink Editor, select **Analysis** > **Design Verifier** > **Latest Results**. The Model Explorer opens, and the results of the latest Simulink Design Verifier analysis appear in the right-hand pane.

For any Simulink Design Verifier analysis, from the Model Explorer, you can perform the following tasks.

Task	For more information
Highlight the analysis results on the model.	"Highlighted Results on the Model" on page 13-2
Generate a detailed analysis report.	"Simulink Design Verifier Reports" on page 13-27
Create the harness model, or if the harness model already exists, open it. If no counterexamples were created during the analysis, this option is not available.	"Simulink Design Verifier Harness Models" on page 13-16
View the data file.	"Simulink Design Verifier Data Files" on page 13-9
View the log file.	"Simulink Design Verifier Log Files" on page 13-50

After you close your model, you can no longer view the analysis results.

Customize Example Proof

Modify the simple Simulink model whose proof objective Simulink Design Verifier disproved in the previous task. Specifically, customize the proof by adding and configuring a `Proof Assumption` block:

1   In the MATLAB Command Window, type `sldvlib`.

   The Simulink Design Verifier library opens.

2   Open the Objectives and Constraints sublibrary.

3   Copy the Proof Assumption block to your model.

4   In your model window, insert the Proof Assumption block between the Inport and Compare To Zero blocks.

5   In your model, double-click the Proof Assumption block to access its attributes.

   The Proof Assumption block parameter dialog box opens.

6   In the **Values** box, enter `[-1, 0]`. When proving properties of this model, Simulink Design Verifier constrains the signal values entering the Compare To Zero block to the specified range. If the input to the Compare to Zero block is always within this range, the output of the Compare to Zero block will always be 1.

12-19

7 Click **Apply** and then **OK** to apply your changes and close the Proof Assumption block parameter dialog box.
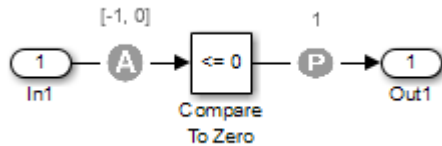
8 Save the `ex_property_proving_example_basic` model and keep it open.

Reanalyze Example Model

Analyze the model that you modified to see how the Proof Assumption block affects the property-proving analysis.

In the `ex_property_proving_example_basic` model window, select **Analysis** > **Design Verifier** > **Prove Properties** > **Model**.

When the analysis is complete, the log window displays the options. There is no option to create a harness model, because the analysis satisfied all proof objectives in your model, so there are no counterexamples.

Review Results of Second Analysis

Review the results of the second analysis:

- "Review Results on the Model" on page 12-20
- "Review Analysis Report" on page 12-21

Review Results on the Model

Highlight the model to see the analysis results:

1 Click **Highlight analysis results on model**.

The Proof Objective is now highlighted in green.
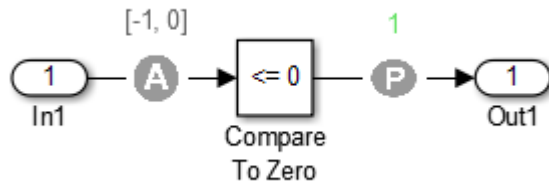
2   Click the Proof Objective block.

The Simulink Design Verifier Results window shows that the proof objective that states that the signal be 1 is valid.
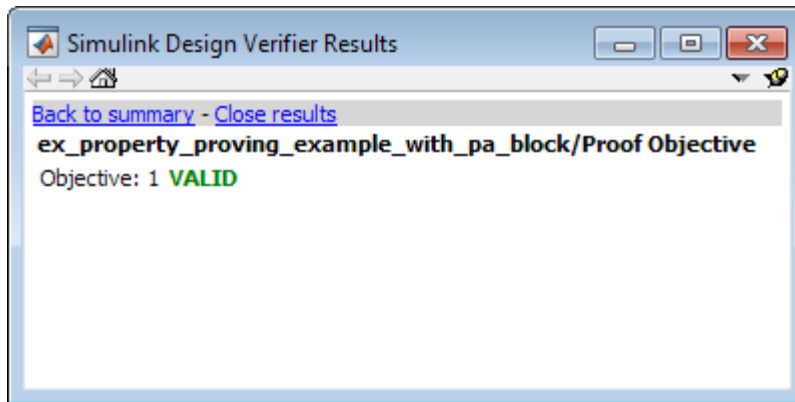
Review Analysis Report

Review the analysis results in the detailed report:

1   Click **Generate detailed analysis report**.

2   In the **Table of Contents**, click Summary.

Chapter 1. Summary

Analysis Information

Model:	ex_property_proving_example_basic
Mode:	PropertyProving
Status:	Completed normally
Analysis Time:	0s

Objectives Status

Number of Objectives:	**1**
Objectives Proven Valid:	1

The Summary chapter indicates that Simulink Design Verifier proved a proof objective in the model.

3 The Constraints section lists the analysis constraint you specified in the Proof Assumption block.

Constraints

Analysis Constraints

Name	Analysis Constraint
Assumption	[-1, 0]

4 Scroll back to the top of the browser window. In the **Table of Contents**, click `Proof Objectives Status`.

Objectives Proven Valid

#	Type	Model Item	Description	Counterexample
1	Proof objective	Proof Objective	Objective: 1	n/a

The Objectives Proven Valid table lists the proof objectives that Simulink Design Verifier proved to be valid.

5 Scroll down to view the Properties chapter or go to the top of the browser window and in the **Table of Contents**, click `Properties`.

Proof Objective

Summary

Model Item: Proof Objective
Property: Objective: 1
Status: Proven valid

The Proof Objective summary indicates that Simulink Design Verifier proved an objective that you specified in your model. The Proof Assumption block restricts the domain of the input signals to the interval [-1, 0]. Therefore, the software proves that this interval does not contain values that are greater than zero, thereby satisfying the proof objective.

Analyze Contradictory Models

If the analysis produces the error `The model is contradictory in its current configuration`, the software detected a contradiction in your model and it cannot analyze the model. You can have a contradiction if your model has Proof Assumption blocks with incorrect parameters. For example, an assumption could state that a signal must be between 0 and 5 when the signal is constant 10.

If the software detects a contradiction, all previous results are invalidated and the software reports that all the properties are falsified.

Prove Properties in a Large Model

A thorough proof of your model requires that Simulink Design Verifier search through all reachable configurations of your model—even the ones that are reached only after long time delays. The computation time and memory required to search a model completely often make an exhaustive proof impractical.

"Prove Properties in Large Models" on page 14-27 gives detailed information about strategies you can use to improve the performance of a property-proving analysis of a large model.

Prove System-Level Properties Using Verification Model

When to Use a Verification Model for Property Proving

If your model has system-wide properties that affect the behavior of the model, you might want to prove the properties without changing the design model. To do this, you create a *verification model* that includes:

- Model block that references the design model
- One or more verification subsystems that define the properties and any required constraints

About this Example

The design model `sldvdemo_sbr_design` models the logic for a seat belt reminder light. If the ignition is turned on, the seat belts are unfastened, and the car exceeds a certain speed, the seat belt reminder light turns on.

The `sldvdemo_sbr_verification` model is a verification model that defines some constraints and verifies the properties in the `sldvdemo_sbr_design` model. The Model block in the verification model references the design model, so that the verification logic exists only in the verification model.

The `sldvdemo_sbr_verification` model contains a property that is falsified, because a constraint is disabled. In the `sldvdemo_sbl_verification_fixed` model, the constraint is enabled and all the properties are proven valid.

Understand the Verification Model

Take these steps to understand how the verification model works:

1   Open the verification model:

    `sldvdemo_sbr_verification`

    The Design Model block is a Model block that references `sldvdemo_sbr_design`. The SBR Stateflow chart in the design model assumes that the KEY input is initially 0.

2   Open the Safety Properties subsystem that specifies the properties of the design model that you want to prove.

    This subsystem contains a MATLAB Function block called **MATLAB Property**. The code in this block specifies the property that the seat belt reminder should be on when the ignition is on, the seat belt is not fastened, and the speed is less than 15:

3   Close the Safety Properties subsystem.

4   Open the Input Constraints subsystem.

    This subsystem defines the following constraints:

    •   The key can have three positions: 0, 1, 2
    •   The speed is constrained to fall between 10 and 30.
    •   The key must start at 0 and can only change by one increment at a time. For example, the key can change from 0 to 1 or 1 to 2, but not from 0 to 2. In this verification model, this constraint is not enabled.

5   Close the Input Constraints subsystem, but keep the `sldvdemo_sbr_verification` model open.

Prove the Properties of the Design Model

Analyze the `sldvdemo_sbr_verification` model to prove the properties:

1   In the `sldvdemo_sbr_verification` model window, to start the analysis, double-click the **Run** button to start the analysis.

    When the analysis completes, the Simulink Design Verifier log window indicates that one objective was falsified.

2   To see which objective was falsified, click **Highlight analysis results on model**.

    The Safety Properties subsystem is highlighted in red.

3   Open the Safety Properties subsystem and click the MATLAB Property block.

The Simulink Design Verifier Results window indicates that the statement

```
sldv.prove(implies(activeCond,SeatBeltIcon))
```

was false during at least one time step.
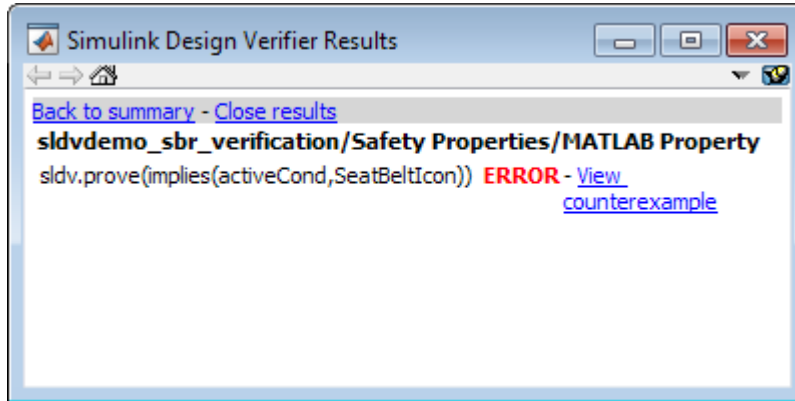
4   Click **View counterexample** to see the signal values that violated this property.

The Signal Builder block opens with the counterexample. The KEY input was initially 2, which is invalid.

To validate the property specified in the Safety Properties subsystem, you have to make sure that the initial value of KEY is 0.

Fix the Verification Model

The Input Constraints subsystem in the verification model contained three constraints. The third constraint, which requires that the initial value of KEY be 0, and that KEY can only change in increments of 1, is disabled.
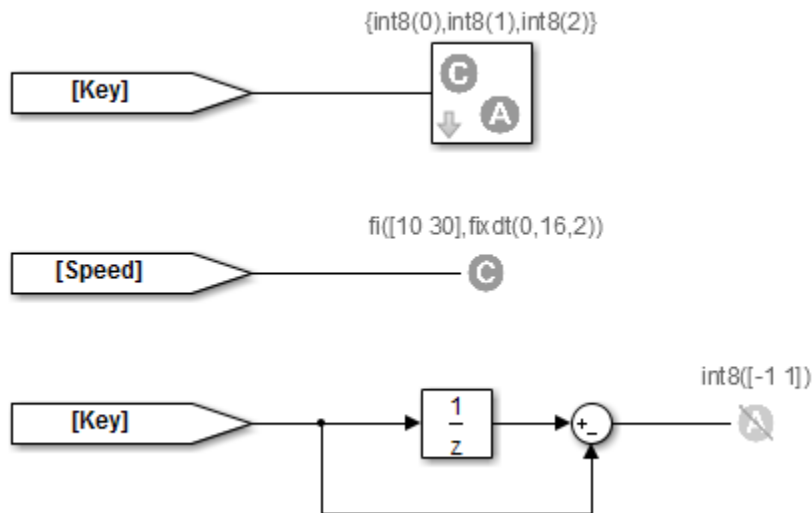
To see how this property is validated when you enable the third constraint:

1   In the `sldvdemo_sbr_verification` model, click **Open Fixed Model**.

    The `sldvdemo_sbr_verification_fixed` verification model opens.

2   Open the Input Constraints subsystem.

    This third constraint is now enabled so that KEY has an initial value of 0 and changes in increments of 1.

3   Close the Input Constraints subsystem.

4   In the `sldvdemo_sbr_verification_fixed` model, to start the analysis, double-click the **Run** block.

    The analysis proves the validity of the property.

Prove Properties in a Subsystem

If you have a large model, you can prove the properties of a subsystem in the model and review the analyses in smaller, manageable reports. The workflow for proving properties in a subsystem is:

1   Open the model that contains the subsystem.

2   Make the subsystem atomic.

3   Run Simulink Design Verifier using the **Prove Properties of Subsystem** option.

4   Review the results.

The tutorial in "Generate Test Cases for a Subsystem" on page 1-26 explains how to generate test cases for the Controller subsystem in the Cruise Control Test Generation model. The steps for proving properties are similar to those for generating test cases, except that you select the **Prove Properties of Subsystem** option instead of the **Generate Tests for Subsystem** option.

Model Requirements

The Simulink Design Verifier block library includes a sublibrary Example Properties. The Example Properties sublibrary includes:

- "Basic Properties" on page 12-30 — Four examples that demonstrate how to prove basic properties.
- "Temporal Properties" on page 12-34 — Four examples that demonstrate how to define temporal properties on Boolean signals

The workflow for using these examples in your model is:

1 Copy these examples into your `Verification Subsystem` block.

2 Adapt them, if required, for the specific properties that you want to prove.

3 Run the Simulink Design Verifier analysis to prove that the assertions in these examples never fail.

4 If the assertion fails, the software creates a counterexample that causes the assertion to fail and then generates a harness model.

5 On the harness model, execute the counterexample to confirm that the assertion fails with that counterexample.

Basic Properties

To view the Basic Properties examples:

1 Open the Simulink Design Verifier block library. Type:

```
sldvlib
```

2 Double-click the Examples sublibrary.

3 Double-click the **Basic Properties** block that contains the examples.

The sections that follow describe each example in the Block Properties sublibrary in detail.

Conditions that Trigger a Result

The Simulink Design Verifier Implies block allows you to test for conditions that trigger a result. This example specifies that if condition A is true, result B must always be true.
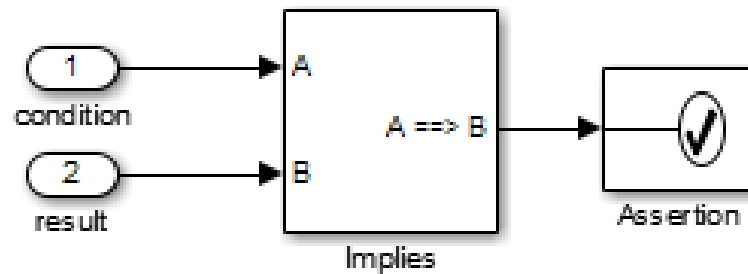
Implies operation describes conditions that should trigger a result.

Increasing or Decreasing Signals

The two examples in this section specify that a signal is either:

- Always increasing or staying constant
- Always decreasing or staying constant
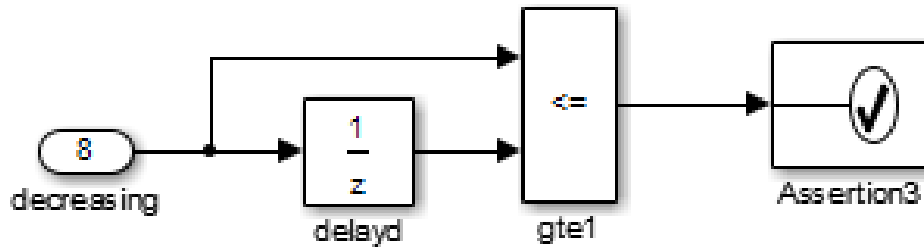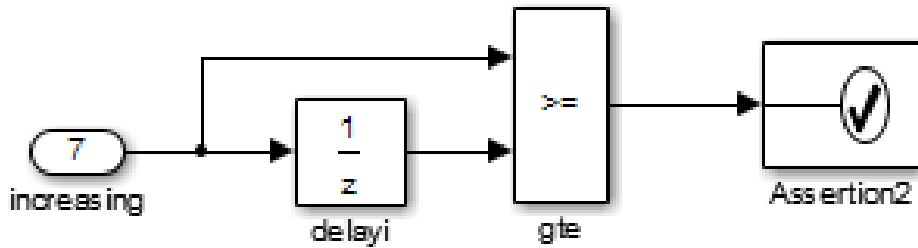
Increasing and decreasing operations describe signals that should increase or decrease.

Exclusivity Operation

This example describes four conditions that should not be true at the same time.
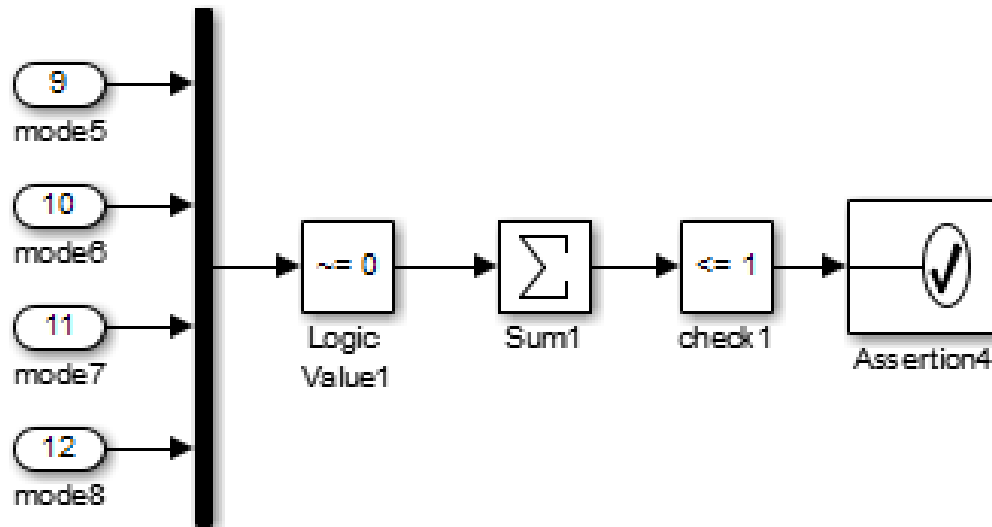
Exclusivity operation describes conditions that should never be true at same time.

Conditions with One True Element

This example specifies that only one of the four input signals can be true.
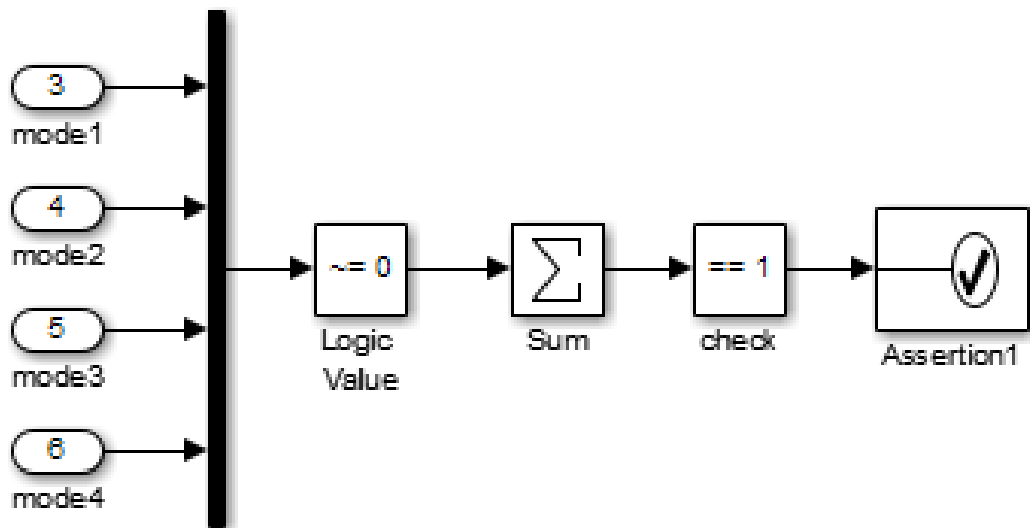
Mutual exclusivity operation describes conditions that should have exactly one true element.

Temporal Properties

To view the Temporal Properties examples:

1 Open the Simulink Design Verifier block library. Type:

   sldvlib

2 Double-click the Temporal Properties sublibrary.

3 Double-click the **Temporal Properties** block that contains the examples.

The sections that follow describe each example in the Temporal Properties sublibrary in detail.

Synchronize the Output with the Input

When the input In1 equals ACTIVE, the input In2 is set to INACTIVE after five time steps.

Whenever In1 becomes ACTIVE, then In2 shall become INACTIVE after a delay of 5 steps.
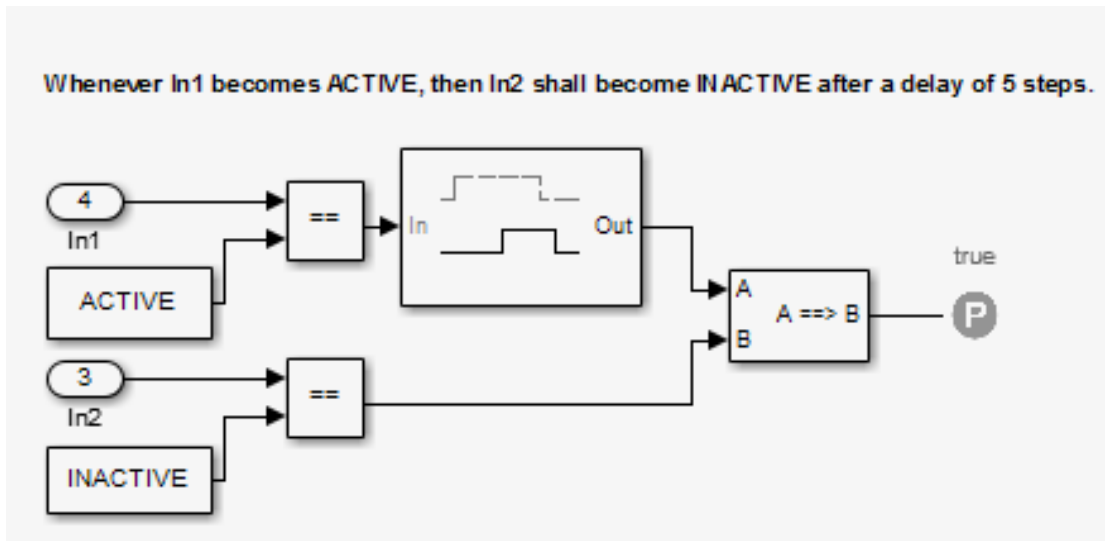
Make a Signal Inactive After a Delay

In this example, after five consecutive time steps where the SENSOR_HIGH input is true, the CMD signal becomes true. CMD is true as long as SENSOR_HIGH is true, unless the block is reset by the MANUAL_RESET signal.

After Sensor is detected at HIGH for 5 consecutive steps, Cmd becomes and stays true for the remaining duration of the Sensor value HIGH unless manual reset is detected.
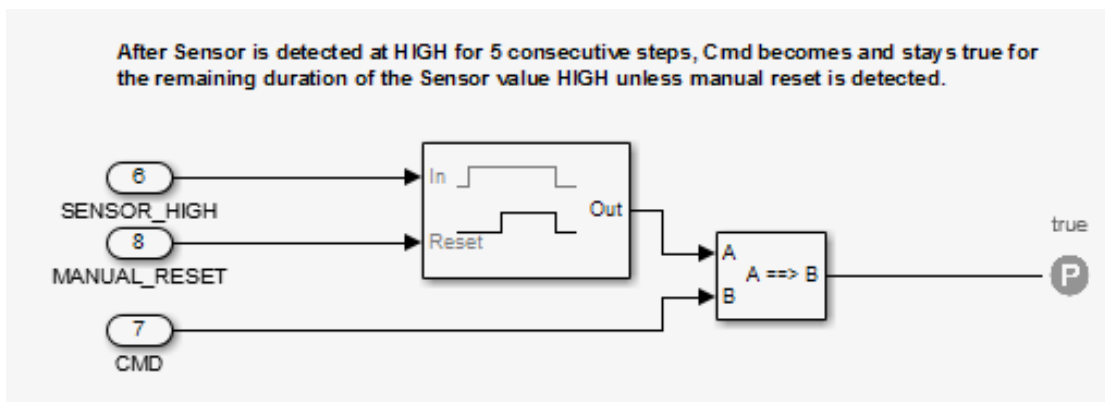
Extend a True Signal

In this example, after the input becomes true, the output becomes true for the number of time steps specified in the Detector block, in this case, 5. The input remains true for 5 time steps as well.

Whenever In becomes true, it shall stay true for the following 5 steps as well.
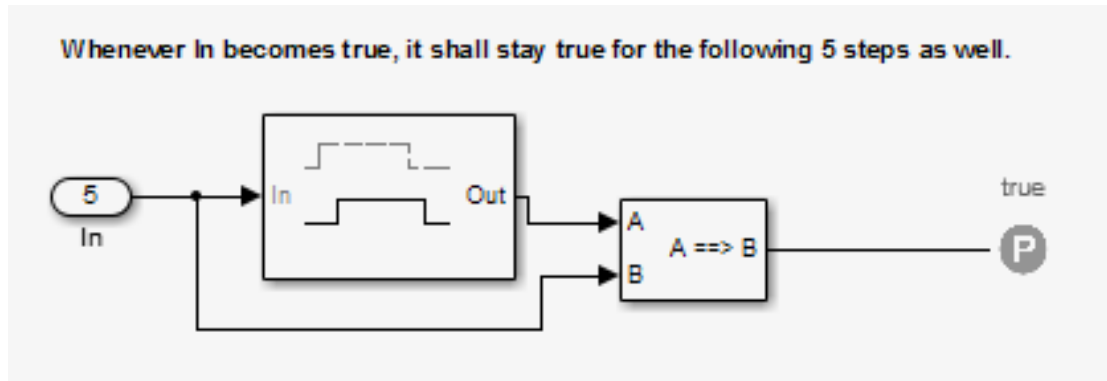
Test the Input Against a Specified Threshold

When the input `In3` equals `ON` and the input `In4` is less than the constant `THRESHOLD`, `In3` is set to `OFF` within five time steps.

Whenever In3 is ON and In4 is less that THRESHOLD, then In3 shall become OFF within 5 steps.

13

Reviewing the Results

Highlighted Results on the Model
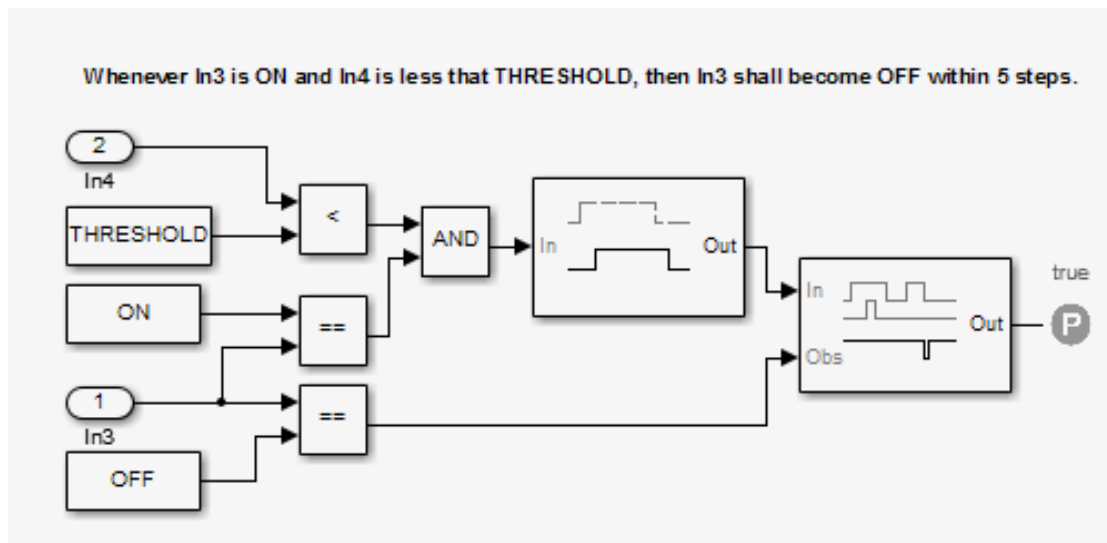
In this section...
"Results Review with Model Highlighting" on page 13-2
"Simulink Design Verifier Results Inspector" on page 13-2
"Enable Highlighted Results on a Model" on page 13-3
"Green Highlighting on Model" on page 13-5
"Red Highlighting on Model" on page 13-6
"Orange Highlighting on Model" on page 13-6
"Gray Highlighting on Model" on page 13-8

Results Review with Model Highlighting

When you analyze a model using Simulink Design Verifier, you can highlight the analyzed model objects in one of four colors:

- Green
- Red
- Orange
- Gray

Model highlighting allows you to review the analysis results at a glance by viewing the objects that are highlighted in the Simulink Editor.

Simulink Design Verifier Results Inspector

When a model is highlighted, you can click an object for which the analysis recorded results. When you click the object, the Simulink Design Verifier Results Inspector displays the detailed analysis results for that object.

The Results Inspector does not display detailed results for Stateflow state transition tables, but analysis data for state transition tables is included in Simulink Design Verifier reports, data files, and log files.

Enable Highlighted Results on a Model

For test case generation or property proving analysis, to enable results highlighting on the model after analysis is complete, do one of the following:

- Before the analysis, in the Configuration Parameters dialog box, on the **Design Verifier** > **Results** pane, select **Display the results of the analysis on the model**.
- After the analysis, in the Simulink Design Verifier log window, select **Highlight analysis results on model**.

For design error detection analysis, results highlighting on the model after analysis is complete is enabled by default.

For all analysis types, you can also highlight results on the model during analysis. To do this, in the Results Summary window, click **Highlight**.

Simulink Design Verifier Results Summary: sldvdemo_fuelsys_logic

Progress

Objectives processed   118/133
Satisfied   115
Falsified   3
Elapsed time   0:45

27-Jun-2013 16:39:29
Starting test generation for model 'sldvdemo_fuelsys_logic'
Compiling model... done
Translating model... done
'sldvdemo_fuelsys_logic' is **compatible** with Simulink Design
Verifier.

Generating tests...

SATISFIED
control
logic.Fueling_Mode.Running.Low_Emissions."[in(O2.normal)]"
Transition: Transition trigger expression F
Analysis Time = 00:00:40

Highlight   Stop

In the Simulink Editor, results highlighting appears on the model. Model objects that the analysis has not yet processed display in orange.

The Results Inspector also opens, displaying the elapsed time of the analysis and a summary of status for analysis objectives.
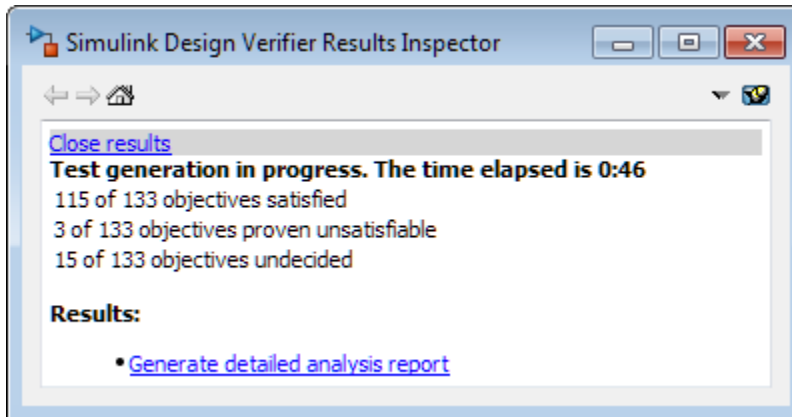
Green Highlighting on Model

Objects that are highlighted in green have the following meaning for each type of analysis.

Analysis mode	Green highlighting means...
Design error detection	One of the following: • The analysis did not find overflow or division-by-zero errors. • The analysis did not find dead logic • The analysis did not find intermediate or output signals outside the range of user-specified minimum and maximum constraints. • The analysis did not find out of bound array access errors.
Test generation	The analysis found test cases that satisfy the test objectives.
Property proving	The analysis proved all the proof objectives valid.

Red Highlighting on Model

Objects that are highlighted in red have the following meaning, depending on the analysis type.

Analysis mode	Red highlighting means...
Design error detection	One of the following: • The analysis found at least one test case that causes overflow or division-by-zero errors. • The analysis found dead logic. • The analysis found intermediate or output signals outside the range of user-specified minimum and maximum constraints. • The analysis found at least one test case that causes an out of bound array access error.
Test generation	The analysis could not satisfy certain test objectives.
Property proving	The analysis disproved a proof objective and generated a counterexample that falsified that objective.

If your model contains at least one object highlighted in red, there may be further design errors in your model that Simulink Design Verifier does not highlight in red. If an object in your design causes run-time errors, Simulink Design Verifier may not be able to determine further errors on objects that are downstream of or rely on the results of the object that causes the run-time errors. Resolve the errors that cause the initial red highlighting and re-run the analysis to determine if Simulink Design Verifier will also highlight other objects in your model as red.

Orange Highlighting on Model

Objects that are highlighted in orange have the following meaning, depending on the analysis type.

Analysis mode	Orange highlighting means...
Design error detection	For the highlighted model object, the analysis did not decide at least one design error detection objective. This situation can occur when:

Analysis mode	Orange highlighting means...
	• The analysis is still in progress.
	• The analysis times out.
	• The analysis cannot decide a design error detection objective because of division by zero or nonlinear arithmetic.
	• The software cannot decide a design error detection objective because of stubbing. For more information, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.
	• The software cannot decide a design error detection objective because of limitations of the analysis engine. For example, if the analysis encounters an unbounded while loop, it performs an approximation. For information about this approximation, see "While Loops" on page 2-18.
Test generation	For the highlighted model object, the analysis did not decide at least one test objective. This situation can occur when:
	• The analysis is still in progress.
	• The analysis times out.
	• The analysis cannot decide a test objective because of division by zero or nonlinear arithmetic.
	• The software cannot decide a test objective because of stubbing. For more information, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.
	• The software cannot decide a test objective because of limitations of the analysis engine. For example, if the analysis encounters an unbounded while loop, it performs an approximation. For information about this approximation, see "While Loops" on page 2-18.

Analysis mode	Orange highlighting means...
Property proving	For the highlighted model object, the analysis did not decide at least one proof objective. This situation can occur when: • The analysis is still in progress. • The analysis times out. • A proof objective exists on a signal whose value the software cannot control, for example, a Constant block. • The analysis cannot decide a proof objective because of division by zero or nonlinear arithmetic. • The software cannot decide a proof objective because of stubbing. For more information, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8. • The software cannot decide a proof objective because of limitations of the analysis engine. For example, if the analysis encounters an unbounded while loop, it performs an approximation. For information about this approximation, see "While Loops" on page 2-18.

Gray Highlighting on Model

Objects that are highlighted in gray have the following meaning.

Analysis mode	Gray highlighting means...
• Design error detection • Test generation • Property proving	The model object was not part of the analysis.

Simulink Design Verifier Data Files

In this section...

Data File Generation

When you enable the **Save test data to file** parameter (see "Design Verifier Pane: Results" on page 15-70), Simulink Design Verifier generates a data file when it completes its analysis. The data file is a MAT-file that contains a structure named `sldvData`. This structure stores all the data the software gathers and produces during the analysis. Although the software displays the same data graphically in the harness model and report, you can use the data file to conduct your own analysis or to generate a custom report.

By default, the **Save test data to file** parameter is enabled.

Contents of `sldvData` Structure

When Simulink Design Verifier completes its analysis, it produces a MAT-file that contains a structure named `sldvData`. To explore the contents of the `sldvData` structure:

1   Generate test cases for the `sldvdemo_flipflop` model:

```
sldvdemo_flipflop;
sldvrun('sldvdemo_flipflop');
```

2   To load the data file, at the MATLAB prompt, enter the following command:

```
load('sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat')
```

The MATLAB software loads the `sldvData` structure into its workspace. This structure contains the Simulink Design Verifier analysis results of the `sldvdemo_flipflop` model.

3 Enter `sldvData` at the MATLAB command line to display the field names that constitute the structure:

```
sldvData =

        ModelInformation: [1x1 struct]
     AnalysisInformation: [1x1 struct]
            ModelObjects: [1x2 struct]
             Constraints: []
              Objectives: [1x12 struct]
                TestCases: [1x4 struct]
                  Version: '2.1'
```

Model Information Fields in `sldvData`

The following sections describe the fields in the `sldvData` structure:

- "ModelInformation Field" on page 13-10
- "AnalysisInformation Field" on page 13-11
- "ModelObjects Field" on page 13-11
- "Constraints Field" on page 13-12
- "Objectives Field" on page 13-12
- "TestCases Field / CounterExamples Field" on page 13-13
- "Version Field" on page 13-14

ModelInformation Field

In the `sldvData` structure, the `ModelInformation` field contains information about the model you analyzed. The following table describes each subfield of the `ModelInformation` field.

Subfield Name	Description
`Name`	The model name.
`Version`	The model number.
`Author`	The user name.
`TimeStamp`	The last date and time the model was updated.
`SubsystemPath`	The full path name of the subsystem (if any) that was analyzed.

Subfield Name	Description
ExtractedModel	The name of the model extracted (if any) to analyze the subsystem (if any) specified in SubsystemPath.
ReplacementModel	The name of the model (if any) that contains the block replacements.
HarnessOwnerModel	The name of the owner model of the Simulink Test™ test harness (if any) being analyzed.

AnalysisInformation Field

In the sldvData structure, the AnalysisInformation field lists settings of particular analysis options and related information. The following table describes each subfield of the AnalysisInformation field.

Subfield Name	Description
Status	The completion status of the Simulink Design Verifier analysis.
AnalysisTime	Double that specifies the length of the analysis in seconds.
Options	Deep copy of the Simulink Design Verifier options object used during the analysis.
InputPortInfo	Cell array of structures that specifies information about each Inport block in the top-level system.
OutputPortInfo	Cell array of structures that specifies information about each Outport block in the top-level system.
SampleTimes	For internal use only.
Parameters	For internal use only.
AbstractedBlocks	For internal use only.
Approximations	A structure that describes the approximations performed during the analysis. For more information about approximations, see "Approximations" on page 2-16.
ReplacementInfo	For internal use only.

ModelObjects Field

In the sldvData structure, the ModelObjects field lists the model items and their associated objectives. The following table describes each subfield of the ModelObjects field.

Subfield Name	Description
descr	The full path to a model object, including objects in a Stateflow chart.
typeDesc	The block type of the model object.
slPath	The full path to a Simulink model object.
sfObjType	The type of a Stateflow object, e.g., S for state and T for transition.
sfObjNum	Integer that represents the unique identifier of a Stateflow object.
sid	For internal use only.
designSid	For internal use only.
replacementSid	For internal use only.
objectives	Vector of integers that represents the indices of objectives associated with a model object.

Constraints Field

In the sldvData structure, the Constraints field lists information about specified minimum and maximum values (if any) on input ports in your model. The following table describes the subfield of the Constraints field.

Subfield Name	Description
DesignMinMax	Cell array of structures that include the name and minimum and maximum values for each input port for which values are specified.

Objectives Field

In the sldvData structure, the Objectives field lists information about each objective, such as its type, status, and description. The following table describes each subfield of the Objectives field.

Subfield Name	Description
type	The type of an objective.

Subfield Name	Description
status	The status of an objective.
descr	The description of an objective.
label	The label of an objective.
outcomeValue	Integer that specifies an objective's outcome.
coveragePointIdx	Integer that represents the index of a coverage point with which an objective is associated.
linkInfo	For internal use only.
range	For internal use only.
modelObjectIdx	Integer that represents the index of a model object with which an objective is associated.
analysistime	Integer that represents the analysis time for an object.
testCaseIdx	Integer that represents the index of a test case or counterexample that addresses an objective.

TestCases Field / CounterExamples Field

In the `sldvData` structure, this field can have two names, depending on the type of check:

- If you set the **Mode** parameter to `Design error detection`, the `CounterExamples` field lists information about each test case that results in an integer overflow or division-by-zero error.
- If you set the **Mode** parameter to `Test generation`, the `TestCases` field lists information about each test case, such as its signal values and the test objectives it achieves.
- If you set the **Mode** parameter to `Property proving`, the `CounterExamples` field lists information about each counterexample and the proof objective it falsifies.

The following table describes each subfield of the `TestCases` / `CounterExamples` field.

Subfield Name	Description
timeValues	Vector that specifies the time values associated with signals in a test case or counterexample.

Subfield Name	Description
`dataValues`	Cell array that specifies the data values associated with signals in a test case or counterexample.
`paramValues`	Structure that specifies the parameter values associated with a test case or counterexample. Its fields include: `name` — The name of a parameter. `value` — Number that specifies the value of a parameter. `noEffect` — Logical value that specifies whether a parameter's value affects an objective.
`stepValues`	Vector that specifies the number of time steps that comprise signals in a test case or counterexample.
`objectives`	Structure that specifies objectives that a test case or a counterexample addresses. Its fields include: `objectiveIdx` — Integer that represents the index of an objective that a test case achieves or a counterexample falsifies. `atTime` — Time value at which either a test case achieves an objective or a counterexample falsifies an objective. `atStep` — Time step at which either a test case achieves an objective or a counterexample falsifies an objective.
`dataNoEffect`	Cell array of logical vectors that specifies whether a signal's data values affect an objective. The vector uses 1 to indicate that a signal's data value does not affect an objective; otherwise, it uses 0.
`expectedOutput`	Cell array of vectors that specifies the output values that result from simulating the model using the test case signals. Each cell represents the output values associated with a different `Outport` block in the top-level system. This subfield is populated if you select **Include expected output values**.

Version Field

In the `sldvData` structure, the `Version` field specifies the version of Simulink Design Verifier that analyzed the model.

Simulate Models with Data Files

The `sldvruntest` function simulates a model using test cases or counterexamples that reside in a Simulink Design Verifier data file:

1   Simulate the `sldvdemo_flipflop` model and generate test cases:

```
sldvdemo_flipflop
```

2   Save the location of the data file generated after analyzing the model:

```
sldvDataFile = 'sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat'
```

3   Use the `sldvruntest` function to simulate the `sldvdemo_flipflop` model using test case 2 in the data file:

```
[ outdata ] = sldvruntest('sldvdemo_flipflop', sldvDataFile, 2)
```

The output from `sldvruntest` is an array of `Simulink.SimulationOutput class` objects.

4   Examine the output data from the first test case using the methods of the `Simulink.SimulationOutput` object:

```
tout_sldvruntest = outData(1).find('tout_sldvruntest');
xout_sldvruntest = outData(1).find('xout_sldvruntest');
yout_sldvruntest = outData(1).find('yout_sldvruntest');
logsout_sldvruntest = outData(1).find('logsout_sldvruntest');
```

Load Results from Data Files

You can load previous analysis results for a model from a data file. For more information, see "Load Previous Results" on page 13-52 and `sldvloadresults`.

Simulink Design Verifier Harness Models

Harness Model Generation

During or after a Simulink Design Verifier analysis, you can create a harness model.

The contents of the harness model depend on the value of the **Mode** parameter, set in the Configuration Parameters dialog box on the **Design Verifier** pane:

- `Design error detection` — The harness model contains test cases that result in errors during simulation.
- `Test generation` — The harness model contains test cases that achieve test objectives.
- `Property proving` — The harness model contains counterexamples that falsify proof objectives.

By default, the **Generate separate harness model after analysis** parameter is disabled.

Note: The Simulink Design Verifier software can generate a harness model only when the top level of the system you are analyzing contains an `Inport` block.

Create a Harness Model

To create a harness model before or after the analysis, do one of the following:

- Before the analysis, in the Configuration Parameters dialog box, on the **Design Verifier** > **Results** pane, select **Generate separate harness model after analysis**.

- After the analysis, in the Simulink Design Verifier log window, select **Create harness model**.

Anatomy of a Harness Model

The Simulink Design Verifier software produces a harness model that looks like this:
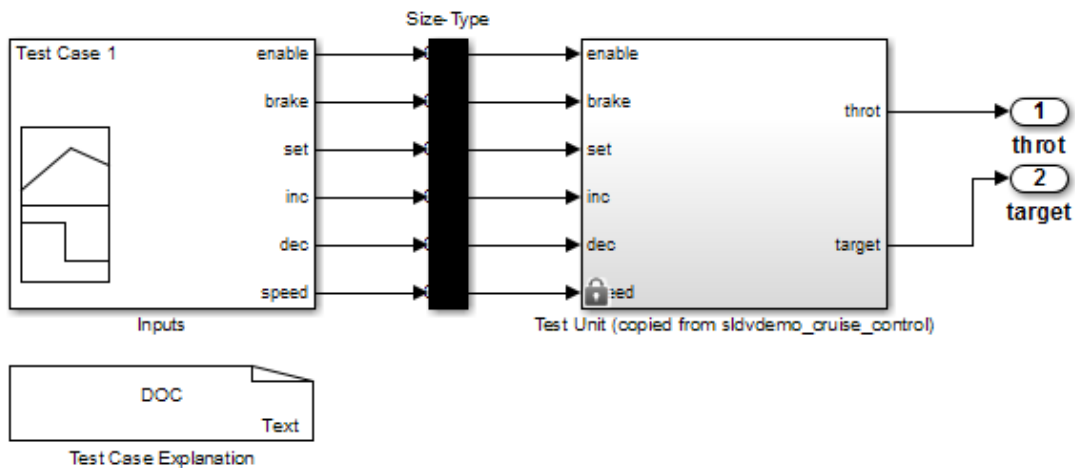
The harness model contains the following items:

- **Inputs** — This `Signal Builder` block contains signals that comprise the test cases or counterexamples that Simulink Design Verifier generated. The Signal Builder block contains signals only for input signals that are used in the model. If an input signal has no effect on the output of the model, that signal is not included in the Signal Builder block.

  Double-click the Inputs block to open the Signal Builder dialog box and view its signals. Each signal group represents a unique test case or counterexample. In the Signal Builder dialog box, select a tab to view the signals associated with a particular test case or counterexample.

  The following Signal Builder block shows the signals for Test Case 7 after Simulink Design Verifier performs test generation analysis on the `sldvdemo_cruise_control` model with the default options.
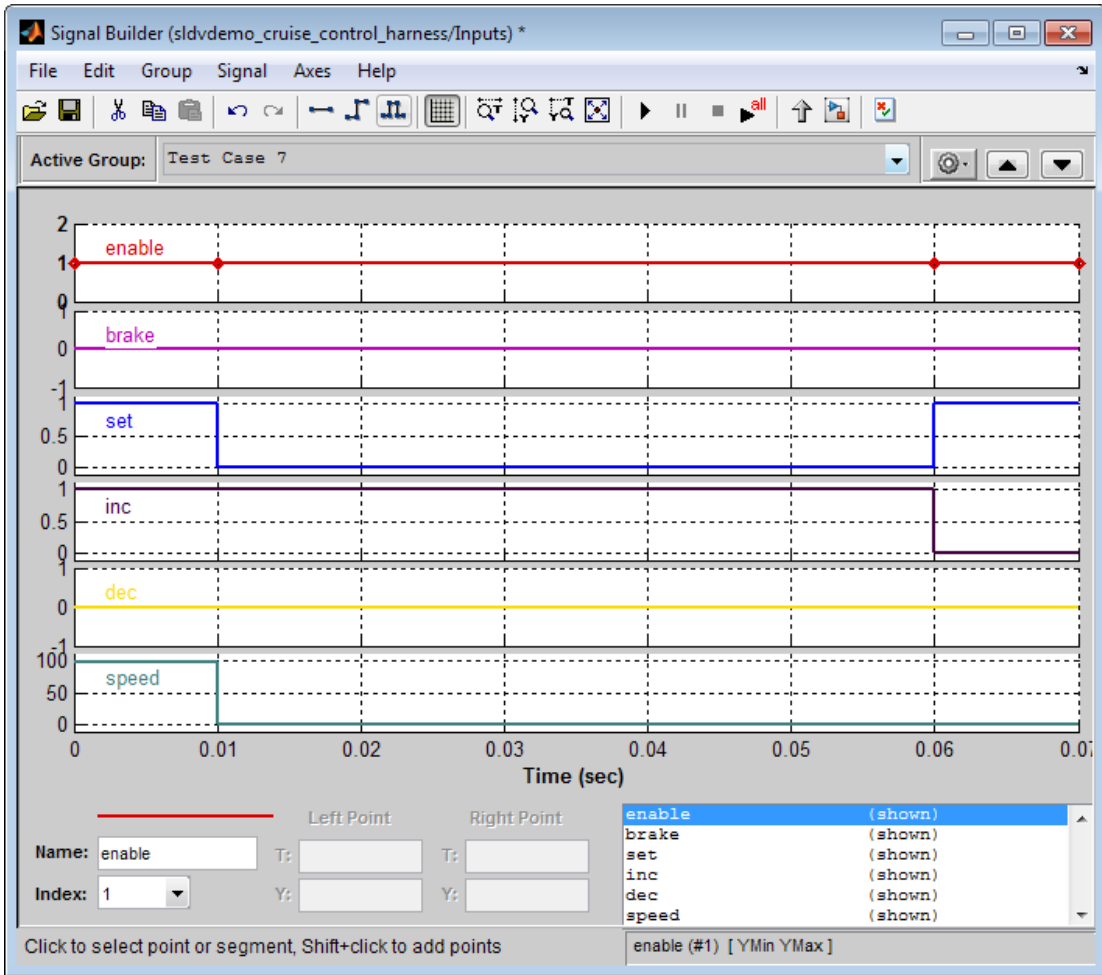
If you select the LongTestcases option of the **Test suite optimization** parameter, the analysis creates fewer, longer test cases. For example, if you select the LongTestcases option for the sldvdemo_cruise_control model, the analysis produces one long test case instead of nine shorter test cases. The following Signal Builder dialog box shows the signals for the long test case.
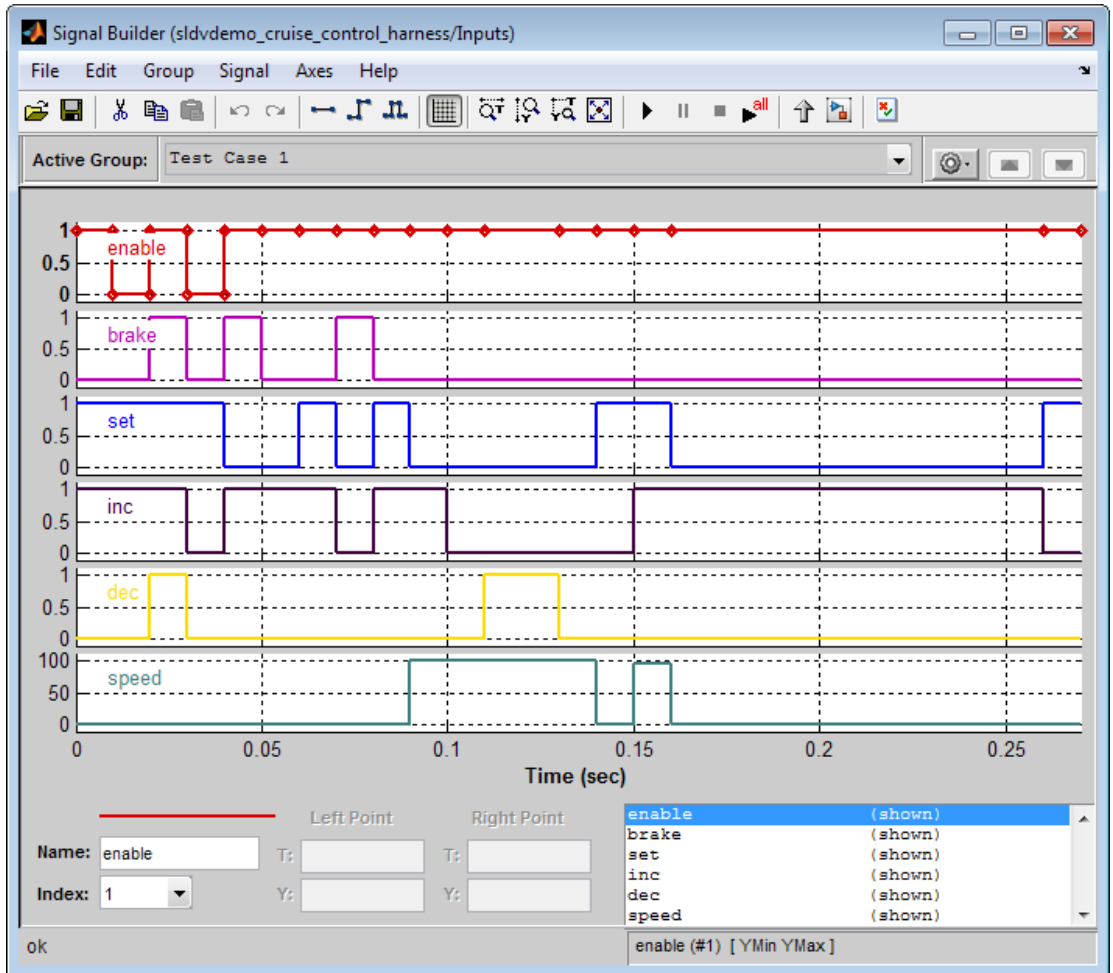
Note: For more information about the Signal Builder dialog box, see "Signal Groups" in the Simulink documentation.

- **Size-Type** — This `Subsystem` block transmits signals from the Inputs block to the Test Unit block. It verifies that the size and data type of the of the signals are consistent with the Test Unit block.

- **Test Unit** — This `Subsystem` block contains a copy of the original model that Simulink Design Verifier analyzed.

  If you select the **Reference input model in generated harness** on the **Design Verifier > Results** pane, the Test Unit is a `Model` block that references the model you are analyzing, not a subsystem.

- **Test Case Explanation** — This `DocBlock` block documents the test cases or counterexamples that Simulink Design Verifier generates. Double-click the Test Case Explanation block to view a description of each test case or counterexample. The block lists either the test objectives that each test case achieves (as in the next graphic) or the proof objectives that each counterexample falsifies.

```
1   Test Case 1 (1 Objectives)
2       Parameter values:
3
4       1. Controller/Switch2 - logical trigger input false (output is from 3rd input port) @ T=0.00
5
6   Test Case 2 (3 Objectives)
7       Parameter values:
8
9       1. Controller/Logical Operator - Logic: input port 1 F @ T=0.00
10      2. Controller/Logical Operator - Logic: MCDC expression for output with input port 1 F @ T=0.00
11      3. Controller/PI Controller - enable logical value F @ T=0.00
12
13  Test Case 3 (3 Objectives)
14      Parameter values:
15
16      1. Controller/Logical Operator1 - Logic: input port 1 T @ T=0.00
17      2. Controller/Logical Operator - Logic: input port 2 F @ T=0.00
18      3. Controller/Logical Operator - Logic: MCDC expression for output with input port 2 F @ T=0.00
19
20  Test Case 4 (1 Objectives)
21      Parameter values:
22
23      1. Controller/Switch3 - logical trigger input true (output is from 1st input port) @ T=0.00
24
25  Test Case 5 (7 Objectives)
26      Parameter values:
27
28      1. Controller/Switch1 - logical trigger input false (output is from 3rd input port) @ T=0.00
29      2. Controller/Logical Operator2 - Logic: input port 1 F @ T=0.00
30      3. Controller/Logical Operator2 - Logic: input port 2 F @ T=0.00
31      4. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 1 F @ T=0.00
32      5. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 2 F @ T=0.00
33      6. Controller/Logical Operator - Logic: input port 3 F @ T=0.00
34      7. Controller/Logical Operator - Logic: MCDC expression for output with input port 3 F @ T=0.00
35
36  Test Case 6 (2 Objectives)
37      Parameter values:
38
39      1. Controller/Logical Operator2 - Logic: input port 2 T @ T=0.01
40      2. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 2 T @ T=0.01
41
42  Test Case 7 (1 Objectives)
43      Parameter values:
44
```

Configuration of the Harness Model

After Simulink Design Verifier generates the harness model, it has the following settings:

- The harness model start time is always 0. If the original model uses a nonzero start time, the software ignores this and uses 0 for the simulation start time for test cases and counterexamples.
- The harness model stop time always equals the stop time of the longest test case in the Signal Builder dialog box.
- By default, the software enables coverage reporting for harness models that contain test cases. Although it enables coverage reporting with particular options selected, you can customize the settings to meet your needs. For more information, see "Customize Requirements Report" in the Simulink Verification and Validation documentation.
- By default, if you select **Ignore objective based on filter** and provide a coverage filter file for the Test Unit, the coverage filter file also applies to the harness model. The coverage objective filter parameters are in the Configuration Parameters dialog box, on the **Test Generation** pane.

Simulate the Harness Model

The harness model enables you to simulate a copy of your original model using the test cases or counterexamples that Simulink Design Verifier generates. Using the harness model, you can simulate:

- A counterexample
- A single test case, for which the Simulink Verification and Validation software collects and displays model coverage information
- All test cases, for which the Simulink Verification and Validation software collects and displays cumulative model coverage information

Note: If you analyze a model that simulates with sample time warnings, when you simulate the harness model, the warnings may be reported as errors, causing the simulation to fail.

To simulate a single test case or counterexample:

1 In the harness model, double-click the Inputs block.

The Signal Builder dialog box appears.

2 In the Signal Builder dialog box, select the tab associated with a particular test case or counterexample.

The Signal Builder dialog box displays the signals that comprise the selected test case or counterexample.

3

In the Signal Builder dialog box, click the **Start simulation** button ▶.

The Simulink software simulates the harness model using the signals associated with the selected test case or counterexample. When simulating a test case, the Simulink Verification and Validation software collects model coverage information and displays a coverage report.

To simulate all test cases and measure their combined model coverage:

1 In the harness model, double-click the Inputs block.

The Signal Builder dialog box appears.

2

In the Signal Builder dialog box, click the **Run all** button ▶.

The Simulink software simulates the harness model using all test cases, while the Simulink Verification and Validation software collects model coverage information and displays a coverage report.

When you click **Run all**, the software simulates all the test cases using the stop time for the harness model. The stop time equals the stop time for the longest test case, so you may accumulate additional coverage when you simulate the shorter test cases.

If the Test Unit in the harness model is a subsystem, the values of the Simulink simulation optimization parameters on the Configuration Parameters dialog box may impact your coverage results.

Note: The simulation optimization parameters are on the following Configuration Parameters dialog box panes:

- **Optimization** pane

- **Optimization > Signals and Parameters** pane
- **Optimization > Stateflow** pane

See "Simulating with Signal Groups" in the Simulink documentation for more information about simulating models containing Signal Builder blocks.

Export Test Cases to Simulink Test

In this section...
"Overall Workflow" on page 13-24
"Test Case Generation Example" on page 13-24

Model verification often requires repeated testing to achieve certain objectives or coverage criteria. If you run repeated tests, consider using the Test Manager in Simulink Test to structure your test cases, archive test results, and generate reports. You can generate test cases using Simulink Design Verifier and export the test inputs to new test cases automatically created in the Simulink Test Manager.

Overall Workflow

Exporting generated inputs to new test cases in Simulink Test 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.

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 Test 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.
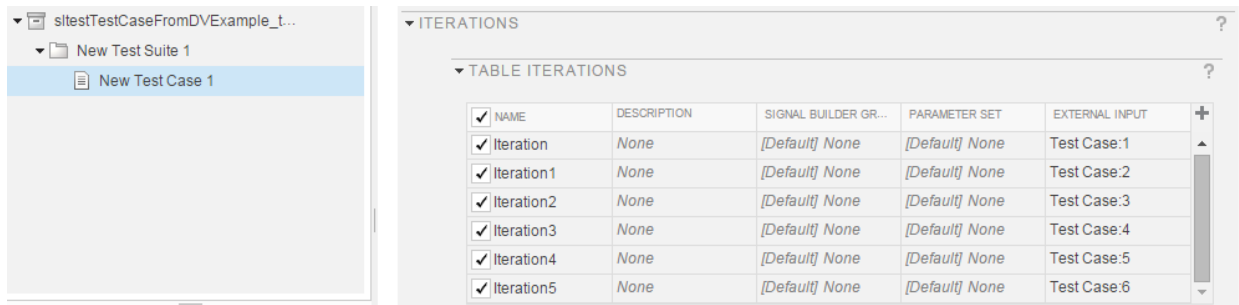
6   Click the harness badge to preview the new test harness.
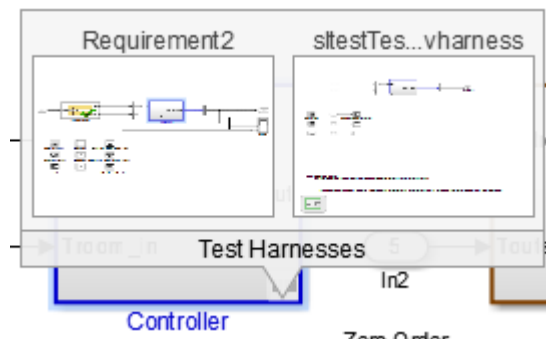
7 Add a test case to the other test harness in the model. In the Test Manager, hover over the new test file name and click the **Synchronize Test File** button .

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`

Simulink Design Verifier Reports

In this section...

Simulink Design Verifier Report Generation

After an analysis, Simulink Design Verifier can generate an HTML report that contains detailed information about the analysis results.

The analysis report contains hyperlinks that allow you to:

- Navigate to a specific part of the report
- Navigate to the object in your Simulink model for which the analysis recorded results

You can also generate an additional PDF version of the Simulink Design Verifier report.

Create Analysis Reports

To create a detailed analysis report before or after the analysis, do one of the following:

- Before the analysis, in the Configuration Parameters dialog box, on the **Design Verifier** > **Report** pane, select **Generate report of the results**. If you want to save an additional PDF version of the Simulink Design Verifier report, select **Generate additional report in PDF format**.

- After the analysis, in the Simulink Design Verifier log window, you can choose HTML or PDF format and **Generate detailed analysis results**.

Front Matter

The report begins with two sections:

- "Title" on page 13-28
- "Table of Contents" on page 13-28

Title

The title section lists the following information:

- Model or subsystem name Simulink Design Verifier analyzed
- User name associated with the current MATLAB session
- Date and time that Simulink Design Verifier generated the report

Table of Contents

The table of contents follows the title section. Clicking items in the table of contents allows you to navigate quickly to particular chapters in the report.

Summary Chapter

The **Summary** chapter of the report lists the following information:

- Name of the model
- Analysis mode
- Analysis status
- Status of objectives analyzed

Analysis Information Chapter

The **Analysis Information** chapter of the report includes the following sections:

- "Model Information" on page 13-29

Model Information

The Model Information section provides the following information about the current version of the model:

- Path and file name of the model that Simulink Design Verifier analyzed
- Model version
- Date and time that the model was last saved
- Name of the person who last saved the model

Analysis Options

The Analysis Options section provides information about the Simulink Design Verifier analysis settings.

The Analysis Options section lists the parameters that affected the Simulink Design Verifier analysis. If you enabled coverage filtering, the name of the filter file is included in this section.

Analysis Options

Mode:	TestGeneration
Test Suite Optimization:	CombinedObjectives
Maximum Testcase Steps:	500 time steps
Test Conditions:	UseLocalSettings
Test Objectives:	UseLocalSettings
Model Coverage Objectives:	MCDC
Maximum Analysis Time:	60s
Block Replacement:	on
Block Replacement Rules:	<FactoryDefaultRules>
Parameters Analysis:	on
Parameters Configuration File:	sldv_params_template.m
Save Data:	on
Save Harness:	on
Save Report:	on

Note: For more information about these parameters, see "Simulink Design Verifier Options" on page 15-2.

Unsupported Blocks

If your model includes unsupported blocks, by default, automatic stubbing is enabled to allow the analysis to proceed. With automatic stubbing enabled, the software considers only the interface of the unsupported blocks, not their actual behavior. This technique

allows the software to complete the analysis. However, the analysis may achieve only partial results if any of the unsupported model blocks affect the simulation outcome.

The Unsupported Blocks section appears only if the analysis stubbed unsupported blocks; it lists the unsupported block in a table, with a hyperlink to the block in the model.

Unsupported Blocks

The following blocks are not supported by Simulink Design Verifier. They were abstracted during the analysis. This can lead Simulink Design Verifier to produce only partial results for parts of the model that depends on the output values of these blocks.

Block	Type
Discrete State-Space	DiscreteStateSpace

For more information about automatic stubbing, see "Handle Incompatibilities with Automatic Stubbing" on page 2-8.

Constraints

The Constraints section provides information about test conditions that Simulink Design Verifier applied when it analyzed a model.

Constraints

Analysis Constraints

Name	Analysis Constraint
constraint	[0, 100]

You can navigate to the constraint in your model by clicking the hyperlink in the Constraints table. The software highlights the corresponding Test Condition block in your model window and opens a new window showing the block in detail.

Block Replacements Summary

The Block Replacements Summary provides an overview of the block replacements that Simulink Design Verifier executed. It appears only if Simulink Design Verifier replaced blocks in a model.

Each row of the table corresponds to a particular block replacement rule that Simulink Design Verifier applied to the model. The table lists the following:

- Name of the file that contains the block replacement rule and the value of the `BlockType` parameter the rule specifies
- Description of the rule that the `MaskDescription` parameter of the replacement block specifies
- Names of blocks that Simulink Design Verifier replaced in the model

To locate a particular block replacement in your model, click on the name for that replacement in the Replaced Blocks column of the table; the software highlights the affected block in your model window and opens a new window that displays the block in detail.

Block Replacements Summary

Table 2.1. Block Replacements

#:	Replacement Rule / Block Type	Rule Description	Replaced Blocks
1	blkrep_rule_switch_normal /Switch	Inserts test objectives for switch blocks that require each switch position be demonstrated when the values of input ports 1 and 3 differ.	Switch1 Switch2 Switch3

Approximations

Each row of the Approximations table describes a specific type of approximation that Simulink Design Verifier used during its analysis of the model.

Approximations

Simulink Design Verifier performed the following approximations during analysis. These can impact the precision of the results generated by Simulink Design Verifier. Please see the product documentation for further details.

#	Type	Description
1	Rational approximation	The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.

Note: Review the analysis results carefully when the software uses approximations. In rare cases, an approximation may result in test cases that fail to achieve test objectives or counterexamples that fail to falsify proof objectives. For example, a floating-point round-off error might prevent a signal from exceeding a designated threshold value.

Derived Ranges Chapter

In a design error detection analysis, the analysis calculates the derived ranges of the signal values for the Outports for each block in the model. This information can help you identify the source of data overflow or division-by-zero errors.

The table in the **Derived Ranges** chapter of the analysis report lists these bounds.

Chapter 3. Derived Ranges

Signal	Derived Ranges
Controller/Constant1- outport 1	1
Controller/Unit Delay- outport 1	[-Inf..Inf]
Controller/Sum- outport 1	[-Inf..Inf]
Controller/Constant3- outport 1	1
Controller/Sum2- outport 1	[-Inf..Inf]
Controller/Switch3/Switch. Defined by block replacement rule 'blkrep_rule_switch_normal'.- outport 1	[-Inf..Inf]
Controller/Switch2/Switch. Defined by block replacement rule 'blkrep_rule_switch_normal'.- outport 1	[-Inf..Inf]
Controller/Switch1/Switch. Defined by block replacement rule 'blkrep_rule_switch_normal'.- outport 1	[-Inf..Inf]
Controller/Sum1- outport 1	[-Inf..Inf]
Controller/Logical Operator1- outport 1	[F..T]
Controller/Unit Delay1- outport 1	[F..T]
Controller/Logical Operator2- outport 1	[F..T]
Controller/Logical Operator- outport 1	[F..T]
throt- outport 1	[-3.5954e+306..3.5954e+306]
target- outport 1	[-Inf..Inf]

Objectives Status Chapters

This section of the report provides information about all objectives in a model, including an objective's type, the model item to which it corresponds, and its description.

- "Design Error Detection Objectives Status" on page 13-34
- "Test Objectives Status" on page 13-36
- "Proof Objectives Status" on page 13-38
- "Objectives Undecided" on page 13-39
- "Objectives Undecided Due to Division by Zero" on page 13-40
- "Objectives Undecided Due to Nonlinearities" on page 13-40
- "Objectives Undecided Due to Stubbing" on page 13-40

Design Error Detection Objectives Status

If you run a design error detection analysis, the **Design Error Detection Objectives Status** section can include the following tables:

- "Active Logic" on page 13-34
- "Dead Logic" on page 13-35
- "Objectives Proven Valid" on page 13-36
- "Objectives Falsified with Test Cases" on page 13-36

Active Logic

The **Active Logic** section lists the model items for which the analysis found active logic. The following image shows a portion of the Active Logic section of the generated analysis report for the sldvdemo_fuelsys_logic_simple example model.

Active Logic

Simulink Design Verifier found that these decision and condition outcomes can occur and are active logic in the model.

#	Type	Model Item	Description	Analysis Time (sec)
5	Decision	control logic.Oxygen_Sensor_Mode	State: Substate executed State "O2_fail"	52
6	Decision	control logic.Oxygen_Sensor_Mode	State: Substate executed State "O2_normal"	52
7	Decision	control logic.Oxygen_Sensor_Mode	State: Substate executed State "O2_warmup"	52
8	Decision	control logic.Oxygen_Sensor_Mode."[Ego < max_ego] / Sens_Fail..."	Transition: Transition trigger expression F	52
9	Decision	control logic.Oxygen_Sensor_Mode."[Ego < max_ego] / Sens_Fail..."	Transition: Transition trigger expression T	52
10	Decision	control logic.Oxygen_Sensor_Mode."[Ego > max_ego]/ Sens_Failu..."	Transition: Transition trigger expression F	52
11	Decision	control logic.Oxygen_Sensor_Mode."[Ego > max_ego]/ Sens_Failu..."	Transition: Transition trigger expression T	52
12	Decision	control logic.Oxygen_Sensor_Mode."[after (5,tick)]"	Transition: Transition trigger expression F	52
13	Decision	control logic.Oxygen_Sensor_Mode."[after (5,tick)]"	Transition: Transition trigger expression T	52

Dead Logic

The **Dead Logic** section lists the model items for which the analysis found dead logic. The following image shows the Dead Logic section of the generated analysis report for the sldvdemo_fuelsys_logic_simple example model.

Dead Logic

Simulink Design Verifier proved that these decision and condition outcomes cannot occur and are dead-logic in the model. Dead-logic in the model can also be a side-effect of parameter configurations, input specified minimum maximum constraints, or in rare cases, the approximations performed by Simulink Design Verifier.

#	Type	Model Item	Description	Analysis Time (sec)
49	Condition	control logic.Speed_Sensor_Mode."[speed==0 & press < zero_th..."	Transition: Condition 2, "press < zero_thresh" F	20
86	Decision	control logic.Fueling_Mode.Fuel_Disabled."[in (Sens_Failure_Counter.Mu..."	Transition: Transition trigger expression F	20

Objectives Proven Valid

The **Objectives Proven Valid** section lists the design error detection objectives that the analysis proved valid. For these objectives, the analysis determined that the described design errors cannot occur.

Objectives Proven Valid

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
20	Overflow	debounce/ SFunction	Overflow	5	n/a

Objectives Falsified with Test Cases

The **Objectives Falsified with Test Cases** section lists the objectives for which the analysis found test cases that demonstrate design errors.

Objectives Falsified with Test Cases

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
12	Overflow	Verify True Output/Sum	Overflow	13	1

Test Objectives Status

If you run a test case generation analysis, the **Test Objectives Status** section can include the following tables:

- "Objectives Satisfied" on page 13-37
- "Objectives Proven Unsatisfiable" on page 13-37

Objectives Satisfied

The **Objectives Satisfied** section lists test objectives that the analysis satisfied.

Objectives Satisfied

Simulink Design Verifier found test cases that exercise these test objectives.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Condition	Controller/Logical Operator1	Logic: input port 1 T	0	3
2	Condition	Controller/Logical Operator1	Logic: input port 1 F	0	8
3	Condition	Controller/Logical Operator2	Logic: input port 1 T	0	8
4	Condition	Controller/Logical Operator2	Logic: input port 1 F	0	5
5	Condition	Controller/Logical Operator2	Logic: input port 2 T	1	6
6	Condition	Controller/Logical Operator2	Logic: input port 2 F	0	5
7	MCDC	Controller/Logical Operator2	Logic: MCDC expression for output with input port 1 T	0	8
8	MCDC	Controller/Logical Operator2	Logic: MCDC expression for output with input port 2 T	1	6
9	MCDC	Controller/Logical Operator2	Logic: MCDC expression for output with input port 1 F	0	5

Objectives Proven Unsatisfiable

The **Objectives Proven Unsatisfiable** section lists the test objectives that the analysis determined could not be satisfied.

In the following example, the Stateflow chart Mode Logic updates at every time step, so the implicit event `tick` is never false. The analysis cannot satisfy condition coverage for the temporal event `after(4, tick)`. For this and other unsatisfiable objectives in the model, the report lists the type, model item, and description.

Objectives Proven Unsatisfiable

Simulink Design Verifier proved that there does not exist any test case exercising these test objectives. This often indicates the presence of dead-logic in the model. Other possible reasons can be inactive blocks in the model due to parameter configuration or test constraints such as given using Test Condition blocks. In rare cases, the approximations performed by Simulink Design Verifier can make objectives impossible to achieve.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
4	Decision	Switchable config	enable logical value T	1	n/a
5	Condition	Logical Operator	Logic: input port 1 T	1	n/a
7	Decision	Normal config (feedthrough)	enable logical value F	1	n/a
9	Decision	Switchable config/Multiport Switch	integer input value = 0 (output is from input port 0)	0	n/a
10	Decision	Switchable config/Multiport Switch	integer input value = 1 (output is from input port 1)	0	n/a
11	Decision	Switchable config/Multiport Switch	integer input value = 2 (output is from input port 2)	0	n/a
12	Decision	Switchable config/Multiport Switch	integer input value = *,3 (output is from input port 3)	0	n/a
22	Condition	Mode Logic."after(4, tick)"	Transition: Condition 1, "tick" F	1	n/a
37	Decision	protected division1/Switch	logical trigger input false (output is from 3rd input port)	1	n/a
41	Decision	protected division/Switch	logical trigger input false (output is from 3rd input port)	1	n/a
45	Decision	Mode Logic.Clipped."[!on]"	Transition: Transition trigger expression F	1	n/a

Proof Objectives Status

If you run a property-proving analysis, the **Proof Objectives Status** section can include:

- "Objectives Proven Valid" on page 13-38
- "Objectives Falsified with Counterexamples" on page 13-39

Objectives Proven Valid

The **Objectives Proven Valid** section lists the proof objectives that the analysis proved valid.

Objectives Proven Valid

#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Verify Output/FoutCorrect	Objective: T	1	n/a
2	Proof objective	Verify Output/ToutCorrect	Objective: T	1	n/a

Objectives Falsified with Counterexamples

The **Objectives Falsified with Counterexamples** section lists the proof objectives that the analysis disproved. In this example, the software generated at least one counterexample that falsifies the specified objective.

Objectives Falsified with Counterexamples

#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Assert	Verify True Output/Assertion	Assert	1	1

Objectives Undecided

For all types of objectives, the **Objectives Undecided** section lists the objectives for which the analysis was unable to determine an outcome in the allotted time.

In the following property-proving example, either the software exceeded its analysis time limit (which the **Maximum analysis time** parameter specifies), or you aborted the analysis before it completed processing these objectives.

Objectives Undecided

Simulink Design Verifier was not able to process these objectives with the current options.

#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Verify Output/FoutCorrect	Objective: T	-1	n/a
2	Proof objective	Verify Output/ToutCorrect	Objective: T	-1	n/a

Objectives Undecided Due to Division by Zero

For all types of objectives, the **Objectives Undecided Due to Division by Zero** section lists objectives that the analysis was unable to decide because of division by zero errors in the associated model items. To detect division by zero errors before running further analysis on your model, follow the procedure in "Detect Integer Overflow and Division-by-Zero Errors" on page 6-25.

Objectives Undecided Due to Division by Zero

Simulink Design Verifier was not able to decide these objectives due to division by zero errors in the model.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Saturation	input > lower limit F	0	n/a
2	Decision	Saturation	input > lower limit T	0	n/a
3	Decision	Saturation	input >= upper limit F	0	n/a
4	Decision	Saturation	input >= upper limit T	0	n/a

Objectives Undecided Due to Nonlinearities

For all types of objectives, the **Objectives Undecided Due to Nonlinearities** section lists the objectives that the analysis was unable to decide because they require computation of nonlinear arithmetic. Simulink Design Verifier does not support nonlinear arithmetic or nonlinear logic.

Objectives Undecided Due to Nonlinearities

Simulink Design Verifier was not able to decide these objectives due to the presence of nonlinear arithmetic in the model.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
30	Decision	BasicRollMode/Integrator	integration result <= lower limit T	2	n/a
32	Decision	BasicRollMode/Integrator	integration result >= upper limit T	2	n/a

Objectives Undecided Due to Stubbing

For all types of objectives, the **Objectives Undecided Due to Stubbing** section lists model items with objectives that the analysis was unable to decide because of stubbing. These can include objectives that, in releases prior to R2013b, were

marked as **Objectives Satisfied – No Test Case** or **Objectives Falsified – No Counterexample**.

Objectives Undecided Due to Stubbing

Simulink Design Verifier was not able to decide these objectives due to stubbing.

#	Type	Model Item	Description	Analysis Time (sec)
2	Decision	Saturation	input > lower limit F	12
3	Decision	Saturation	input > lower limit T	12
4	Decision	Saturation	input >= upper limit F	12
5	Decision	Saturation	input >= upper limit T	12

Model Items Chapter

The **Model Items** chapter of the report includes a table for each object in the model that defines coverage objectives. The table for a particular object lists all of the associated objectives, the objective types, objective descriptions, and the status of each objective at the end of the analysis.

The table for an individual object in the model will look similar to this one for the Discrete-Time Integrator in the PI Controller subsystem of the `sldvdemo_cruise_control` example model.

Controller/PI Controller/Discrete-Time Integrator

View

#:	Type	Description	Status	Test Case
31	Decision	integration result <= lower limit F	Satisfied	3
32	Decision	integration result <= lower limit T	Satisfied	8
33	Decision	integration result >= upper limit F	Satisfied	3
34	Decision	integration result >= upper limit T	Satisfied	9

To highlight a given object in your model, click **View** at the upper-left corner of the table; the software opens a new window that displays the object in detail. To view the details of the test case that was applied to a specific objective, click the test case number in the last column of the table.

Design Errors Chapter

If you run a design error detection analysis, the report includes a **Design Errors** chapter. This chapter includes sections that summarize the design errors the analysis validated or falsified:

- "Table of Contents" on page 13-42
- "Summary" on page 13-42
- "Test Case" on page 13-43

Table of Contents

Each Design Errors chapter contains a table of contents. Each item in the table of contents is a hyperlink to results about a specific design error.

Summary

The Summary section lists:

- The model item
- The type of design error that was detected (overflow or division by zero)
- The status of the analysis (Falsified or Proven Valid)

In the following example, the software analyzed the `sldvdemo_debounce_falseprop` model to detect design errors. The analysis detected an overflow error in the Sum block in the Verification Subsystem named Verify True Output.

Summary

Model Item:	Verify True Output/Sum
Type:	Overflow
Status:	Falsified

Test Case

The Test Case section lists the time step and corresponding time at which the test case falsified the design error objective. The Inport block raw had a value of 255, which caused the overflow error.

Test Case

Time	0-0.01
Step	1-2
raw	255

Test Cases Chapter

If you run a test generation analysis, the report includes a **Test Cases** chapter. This chapter includes sections that summarize the test cases the analysis generated:

- "Table of Contents" on page 13-43
- "Summary" on page 13-43
- "Objectives" on page 13-44
- "Generated Input Data" on page 13-44
- "Expected Output" on page 13-45
- "Combined Objectives" on page 13-46
- "Long Test Cases" on page 13-47

Table of Contents

Each Test Cases chapter contains a table of contents. Each item in the table of contents is a hyperlink to information about a specific test case.

Summary

The Summary section lists:

- Length of the signals that comprise the test case
- Total number of test objectives that the test case achieves

Summary

Length: 0.06 second (7 sample periods)
Objectives Satisfied: 1

Objectives

The Objectives section lists:

- The time step at which the test case achieves that objective.
- The time at which the test case achieves that objective.
- A link to the model item associated with that objective. Clicking the link highlights the model item in the Simulink Editor.
- The objective that was achieved.

Objectives

Step	Time	Model Item	Objectives
7	0.06	Controller/PI Controller/Discrete-Time Integrator	integration result >= upper limit T

Generated Input Data

For each input signal associated with the model item, the Generated Input Data section lists the time step and corresponding time at which the test case achieves particular test objectives. If the signal value does not change over those time steps, the table lists the time step and time as ranges.

Generated Input Data

Time	0	0.01-0.05	0.06
Step	1	2-6	7
enable	1	1	1
brake	0	0	0
set	1	0	1
inc	1	1	-
dec	1	0	-
speed	97	0	0

Note: The Generated Input Data table displays a dash (–) instead of a number as a signal value when the value of the signal at that time step does not affect the test objective. In the harness model, the Inputs block represents these values with zeros unless you enable the **Randomize data that does not affect outcome** parameter (see "Randomize data that do not affect the outcome" on page 15-75).

Expected Output

If you select the **Include expected output values** on the **Design Verifier** > **Results** pane of the Configuration Parameters dialog box, the report includes the Expected Output section for each test case. For each output signal associated with the model item, this table lists the expected output value at each time step.

Expected Output These output values are expected assuming that inputs that do not affect the test objectives (- in the table above) are given a default value - 0 for numeric types, and default value for enumerated types.

Time	0	0.01	0.02	0.03	0.04	0.05	0.06
Step	1	2	3	4	5	6	7
throt	0	1.96	1.9898	2.0197	2.0497	2.0798	0.05
target	97	98	99	100	101	102	0

Combined Objectives

If you set the **Test suite optimization** option to `CombinedObjectives` (the default), the Test Cases chapter may include individual information about many test cases.

Chapter 5. Test Cases

Table of Contents

This section contains detailed information about each generated test case.

Test Case 1

Summary

Length:	0 second (1 sample period)
Objectives Satisfied:	12

Long Test Cases

If you set the **Test suite optimization** option to `LongTestcases`, the Test Cases chapter in the report includes fewer sections about longer test cases.

Chapter 5. Test Cases

Table of Contents

Test Case 1

This section contains detailed information about each generated test case.

Test Case 1

Summary

Length:	0.26 second (27 sample periods)
Objectives Satisfied:	259

Properties Chapter

If you run a property-proving analysis, the report includes a **Properties** chapter. This chapter includes sections that summarize the proof objectives and any counterexamples the software generated:

- "Table of Contents" on page 13-48
- "Summary" on page 13-48
- "Counterexample" on page 13-49

Table of Contents

Each Properties chapter contains a table of contents. Each item in the table of contents is a hyperlink to information about a specific property that was falsified.

Summary

The Summary section lists:

- The model item that the software analyzed
- The type of property that was evaluated
- The status of the analysis

In the following example, the software analyzed the
sldvdemo_cruise_control_verification model for property proving. The analysis
proved that the input to the Assertion block named BrakeAssertion was nonzero.

Summary

Model Item:	Safety Properties/BrakeAssertion
Property:	Assert
Status:	Falsified

Counterexample

The Counterexample section lists the time step and corresponding time at which the
counterexample falsified the property. This section also lists the values of the signals at
that time step.

Counterexample

Time	0	0.01	0.02-0.04
Step	1	2	3-5
InputData.Actual_speed	0	0	0
InputData.Switches.enable	1	1	0
InputData.Switches.brake	0	0	1
InputData.Switches.set	1	0	0
InputData.Switches.setIncDec.inc	1	1	0
InputData.Switches.setIncDec.dec	0	0	0

Simulink Design Verifier Log Files

Every time you analyze a model, Simulink Design Verifier creates a log file. To view the log file, click **View Log** in the Simulink Design Verifier log window.

The log file contains a list of the analysis results for each object in the model. The content of the log file corresponds to the analysis results displayed in the log window during the analysis.

```
 1
 2   12-Jun-2013 11:24:50
 3   Starting test generation for model 'sldvdemo_cruise_control'
 4   Compiling model... done
 5   Translating model... done
 6   'sldvdemo_cruise_control' is compatible with Simulink Design Verifier.
 7
 8   Generating tests...
 9
10   SATISFIED
11   Controller/Logical Operator
12   Logic: MCDC expression for output with input port 3 T
13   Analysis Time = 00:00:01
14
15   SATISFIED
16   Controller/Logical Operator
17   Logic: MCDC expression for output with input port 2 T
18   Analysis Time = 00:00:01
19
20   SATISFIED
21   Controller/Logical Operator
22   Logic: MCDC expression for output with input port 1 T
23   Analysis Time = 00:00:01
24
25   SATISFIED
26   Controller/Logical Operator2
27   Logic: MCDC expression for output with input port 1 T
28   Analysis Time = 00:00:01
29
30   SATISFIED
31   Controller/PI Controller
32   enable logical value T
33   Analysis Time = 00:00:01
```

13-51

Review Analysis Results

Abstract

Review analysis results in the Simulink Design Verifier Results Summary window.

In this section...
"View Active Results" on page 13-52
"Load Previous Results" on page 13-52
"Explore Results" on page 13-53

View Active Results

After analysis is complete, the Simulink Design Verifier Results Summary window opens, showing different ways you can use the results. See "Explore Results" on page 13-53.

If you close the Results Summary window so you can fix the cause of any analysis errors in your model, you might need to review the analysis results again. If you have not closed your model since you ran the analysis, you can reopen the latest analysis results for your model. In the Simulink Editor, select **Analysis** > **Design Verifier** > **Results** > **Active**. The Results Summary window reopens with the latest analysis results for your model.

You can also view Simulink Design Verifier analysis results in the Model Explorer.

Load Previous Results

If you want to review results of a previous analysis on a model, you can load these results from the analysis data file. In the Simulink Editor, select **Analysis** > **Design Verifier** > **Results** > **Load**. Browse and select the data file that corresponds to the analysis you want to review.

For more information on analysis data files, see "Simulink Design Verifier Data Files" on page 13-9.

If you load analysis results for a model from a data file that was generated with a previous version of that model, you might see unexpected effects. To avoid inconsistencies

between your model and analysis results data, when you load results for a model, choose a data file that contains results from the same version of that model.

Explore Results

With active or previous analysis results loaded in the Model Explorer or Results Summary window, you can perform the following tasks.

Task	For more information
Highlight the analysis results on the model.	"Highlighted Results on the Model" on page 13-2
Generate a detailed analysis report.	"Simulink Design Verifier Reports" on page 13-27
Create the harness model, or if the harness model already exists, open it. You will not be able to create the harness model if: • No design error objectives were falsified • No test cases were generated • No counterexamples were created	"Simulink Design Verifier Harness Models" on page 13-16
View the data file.	"Simulink Design Verifier Data Files" on page 13-9
View the log file.	"Simulink Design Verifier Log Files" on page 13-50

14

Analyzing Large Models and Improving Performance

Sources of Model Complexity

Some characteristics of Simulink models can cause problems during a Simulink Design Verifier analysis in the following ways:

- Complexity of model inputs due to:

    - Large number of inputs (The number of inputs can vary, depending on the individual model.)
    - Types of inputs (floating-point values, for example)
    - The way the inputs affect the model state and the objectives of the analysis

- Number of possible simulation paths through a model
- Portions of the model that cannot be reached
- Large counters in the model

The topics in "Complexity Reduction" describe techniques designed to reduce the impact of this complexity and achieve the best performance from Simulink Design Verifier.

Most of these techniques focus on test generation for large models. However, you can use many of them to detect design errors or prove the properties of a large model and generate counterexamples when a property is disproved. In addition, "Prove Properties in Large Models" on page 14-27 describes specific techniques for proving properties in a large model.

Analyze a Large Model

In this section...
"Types of Large Model Problems" on page 14-3
"Summarize Model Hierarchy and Compatibility" on page 14-4
"Use the Default Parameter Values" on page 14-4
"Modify the Analysis Parameters" on page 14-6
"Use the Large Model Optimization" on page 14-6
"Stop the Analysis Before Completion" on page 14-6

Types of Large Model Problems

The Simulink Design Verifier software may encounter some of these problems when analyzing a large model:

- Unsatisfiable objectives — The software proved there are no test cases that exercise these test objectives, and did not generate any test cases.
- Undecided objectives — The software was not able to satisfy or falsify these objectives.
- Objectives with errors — This problem usually occurs when a model component uses nonlinear arithmetic, which can affect a test objective.
- Cannot complete the analysis in the time allotted — This problem may indicate an area of your model where the software encountered problems, or you may need to increase value of the **Maximum analysis time** parameter.
- Analysis hangs — If the number of objectives processed remains constant for a considerable length of time, the software has likely encountered complexity between the model and its objectives.
- Does not achieve a high percentage of model coverage — When you run the test cases on the harness model, the percentage of model coverage is insufficient for your design.

The next few sections describe the initial steps to take when analyzing a large model. Although these steps address test generation, you can use a similar approach when detecting design errors or proving properties in a model.

Summarize Model Hierarchy and Compatibility

You can use the Test Generation Advisor to summarize test generation compatibility, condition and decision objectives, and dead logic for the model and model components.

The Test Generation Advisor performs a high-level analysis and fast dead logic detection. You can use the results to better understand your model, particularly large models, complex models, or models for which you are uncertain of their compatibility with Simulink Design Verifier. For example, you can:

- Identify incompatibilities with test case generation.
- Identify complex components that might be time-consuming to analyze.
- Determine instances of dead logic.
- Get a summary of the component hierarchy.
- Get recommended test generation parameters.

You can access the Test Generation Advisor from the menu bar by clicking **Analysis > Design Verifier > Generate Tests > Advisor**. For more information see "Use Test Generation Advisor to Identify Analyzable Components" on page 7-21.

Use the Default Parameter Values

When you generate test cases, you should generally begin by analyzing the model using the Simulink Design Verifier default parameter values:

1   Check to see if your model is compatible with Simulink Design Verifier, as described in "Check Model Compatibility" on page 3-2.

2   Using the default parameter values, analyze the model. The following table lists the default values for parameters in the Configuration Parameters dialog box that you might change when analyzing large models.

Parameter	Default Value	Description
Maximum analysis time (s)	300 (seconds)	If the analysis does not finish within the specified time, the analysis times out and terminates.
Test suite optimization	CombinedObjectives (Nonlinear extended)	Generates test cases that address more than one test objective, as with the CombinedObjectives option, but with

Parameter	Default Value	Description
		improved support for nonlinear arithmetic. Each test case tends to include many time steps.
Model coverage objectives	`Condition Decision`	Generates test cases that achieve condition and decision coverage.

3 Review the following information in the Simulink Design Verifier log window while the analysis runs:

- Number of objectives processed — How many objectives were processed? Did the analysis hang after processing a certain number of objectives? The answers to these questions might give you a clue about where a problem might lie.

- Number of objectives satisfied/Number of objectives falsified — Which objectives were falsified?

- Time elapsed — Did the analysis time out, or did it finish within the specified maximum analysis time?

4 When the analysis completes, you can highlight the results in the model and individually review the analysis of each model object, as described in "Highlighted Results on the Model" on page 13-2. You can also generate and review the Simulink Design Verifier HTML report. This report contains links to the model elements for satisfied and falsified objectives so you can see what portions of the model might have problems. For more information, see "Simulink Design Verifier Reports" on page 13-27.

5 For a test-generation analysis, if all the test objectives have been satisfied, run the test cases on the harness model to determine model coverage.

If model coverage is enough for your design, you do not need to do anything else. If the coverage is insufficient, take additional steps to improve the analysis performance, as described in the following sections.

Note: A large percentage of falsified objectives and poor model coverage often indicate that you need to change model parameter values to get complete coverage. This can occur when you have tunable parameters in Constant blocks that are connected to enabled subsystems or to the trigger inputs of Switch blocks. In these situations, configure Simulink Design Verifier parameter support as described in the example "Specify Parameter Constraint Values for Full Coverage" on page 5-10.

Modify the Analysis Parameters

If the analysis satisfied most but not all of the objectives, try the following steps:

1   Increase the **Maximum analysis time** parameter. This gives the analysis more time to satisfy all the objectives.

2   Set the **Model coverage objectives** parameter to `Decision`. Selecting this option generates only test cases that achieve decision coverage. These test cases are a subset of the `MCDC` option.

3   Rerun the analysis and review the report.

    If the results are still not satisfactory, try the techniques described in the following sections.

Use the Large Model Optimization

Set the **Test suite optimization** parameter to `LargeModel` or `LargeModel (Nonlinear Extended)`, and rerun the Simulink Design Verifier analysis.

The large model optimization strategies are designed for large, complex models. The `LargeModel (Nonlinear Extended)` strategy includes improved support for nonlinear arithmetic. These two strategies may or may not improve the results of your analysis enough to fully test your design.

If you have outstanding objectives you want the software to generate, continue with the following techniques.

Stop the Analysis Before Completion

Watch the **Objectives processed** value in the log window. If about 50 percent of the **Maximum analysis time** parameter has elapsed and this value does not increase, the model analysis may have trouble processing certain objectives. If the analysis does not progress, take the following steps:

1   Click **Stop** in the log window.

    A dialog box appears, informing you that the analysis was aborted and asking you if you still want to produce results.

2   Click **Yes** to save the results of the analysis so far.

The log window lists the following options, depending on which analysis mode you ran:

- **Highlight analysis results on model**
- **Generate detailed analysis report**
- **Create harness model**
- **Simulate tests and produce a model coverage report**

3 Click **Generate detailed analysis report**.

4 In the HTML report, review the following sections to identify the model elements that are causing problems:

- **Objectives Undecided when the Analysis was Stopped**
- **Objectives Producing Errors**

5 Review the model elements that have undecided objectives or objectives with errors to see if any of the following problems are present. Consult the respective documentation for specific techniques to improve the analysis.

Problem in your model	More information
Floating-point inputs	"Manage Model Data to Simplify the Analysis" on page 14-9
Nonlinear operations	• "Bottom-Up Approach to Model Analysis" on page 14-15 • "Logical Operations" on page 14-23
Large state spaces	"Models with Large Verification State Space" on page 14-24
Large timers and time delays	"Counters and Timers" on page 14-25

Increase Allocated Memory for Analysis Report Generation

When you analyze a model with a large root-level input signal count, you may encounter an insufficient memory error when Simulink Design Verifier is generating the report.

When this occurs, you need to increase the amount of memory the Sun™ Java® Virtual Machine (JVM™) software can allocate. For steps on how to increase this memory, see "Increase the MATLAB JVM Memory Allocation Limit" in the MATLAB Report Generator™ documentation.

Manage Model Data to Simplify the Analysis

In this section...
"Simplify Data Types" on page 14-9
"Constrain Data" on page 14-9

Abstract

Simplify your model to simplify the Simulink Design Verifier analysis.

Simplify Data Types

One way to simplify your model is to use for the designated signal data type a data type requiring the least amount of space for the expected data. For example, do not use an `int` data type for Boolean data, because only one bit is required for Boolean data.

In another example, suppose you have a Sum block with two inputs that are always integers between –10 and 10. Set the **Output data type** parameter to `int8`, rather than `int32` or `double`.

To display the signal data types in the model window, select **Display** > **Signals & Ports** > **Port Data Types**.

Constrain Data

Another effective technique for reducing complexity is to restrict the inputs to a set of representative values or, ideally, a single constant value. This process, called *discretization*, treats the input as if it were an enumeration. Discretization allows you to handle nonlinear arithmetic from multiplication and division in the simplest way possible.

The following model has a Product block feeding a Saturation block.
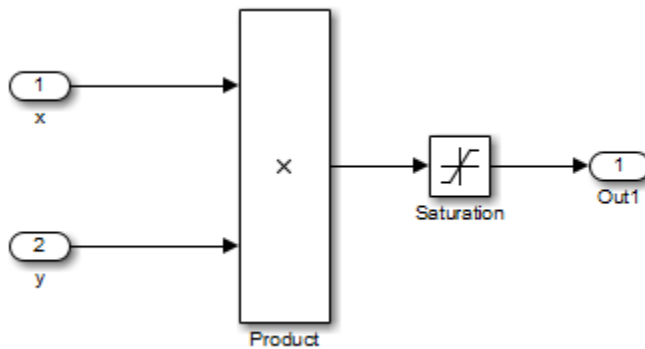
The Simulink Design Verifier software generates errors when attempting to satisfy the upper and lower limits of the Saturation block, because the software does not support nonlinear arithmetic. To work around these errors, restrict one of the inputs to a set of discrete values.

Identify discrete values that are required to satisfy your testing needs. For example, you may have an input for model speed, and your design contains paths of execution that are conditioned on speed above or below thresholds of 80, 150, 600, and 8000 RPM. For an effective analysis, constrain speed values to be 50, 100, 200, 1000, 5000, or 10000 RPM so that every threshold can be either active or inactive.

If you need to use more than two or three values, consider specifying the constrained values using an expression like

```
num2cell(minval:increment:maxval)
```

Using the previous example model, restrict the second input (*y*) to be either 1, 2, 5, or 10 using the Test Condition block as shown in the following model. The Simulink Design Verifier software produces test cases for all inputs.
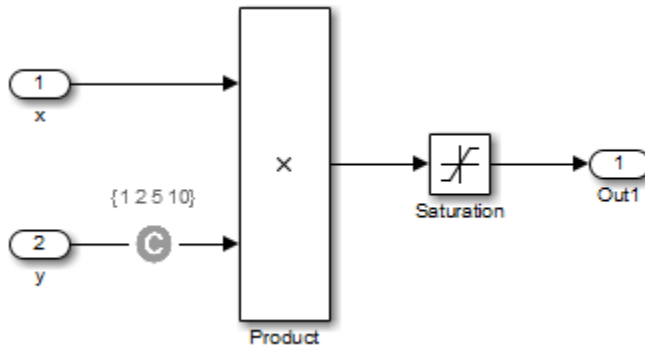
You can also constrain signals that are intermediate or output values of the model. Constraining such signals makes it easier to work around multiplication or division inside lower level subsystems that do not depend on model inputs.

Note: Discretization is best limited to a small number of inputs (less than 10). If your model requires discretization of many inputs, try to achieve model coverage through successive simulations, as described in "Partition Model Inputs for Incremental Test Generation" on page 14-13.

Test Condition blocks do not need to be placed exactly on the inputs. In deciding where to place the constraints in your model, consider the following guidelines:

- Favor constraints on the input values because the software can process inputs easier.

- If you need to place constraints on both the input and the output, for example, to avoid nonlinear arithmetic, one of the constraints should be a range such as `[minval maxval]`. The software first tests the values at both ends of the range and can return a test case, even if the underlying calculations are nonlinear.

- Make sure that constraints at corresponding input and output points are not contradictory. Do not constrain the output signals to values that are not achievable because of the constraints on the input values.

- Avoid creating constraints that contradict the model. Such contradictions occur when a constraint can never be satisfied because it contradicts some aspect of the model or another constraint. Analyzing contradictory models can cause Simulink Design Verifier to hang.

The next model is a simple example of a contradictory model. The second input to the Multiply block is the constant 1, but the Test Condition block constrains it to a value of 2, 5, or 10. The analysis cannot achieve all the test objectives in this model.
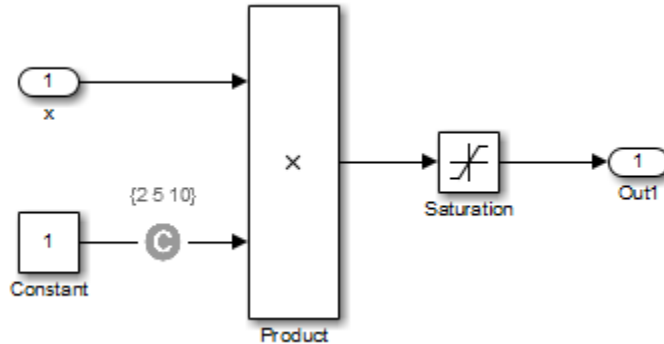
- When you work with large models that have many multiplication and division operations, you may find it easier to add constraints to all of the floating-point inputs rather than to identify the precise set of inputs that require constraints.

Partition Model Inputs for Incremental Test Generation

As described in "Constrain Data" on page 14-9, you can constrain the values of model inputs using the Simulink Design Verifier Test Condition block.

Like other Simulink parameters, constraint values can be shared across several blocks by referencing a common workspace variable; you can initialize constraint values using MATLAB commands. If you have several inputs related to speed, such as desired speed, measured speed, and average speed, you might choose to constrain all of them to the same set of values.

As an advanced technique for experienced MATLAB programmers, you can use parameterized constraints and successive runs of Simulink Design Verifier to implement an incremental test-generation technique:

1 Partition model inputs so that some are held constant, some are constrained to sets of constants using the Test Condition block, and some can have any value.

2 Generate test cases and run those test cases to collect model coverage.

3 Choose new values and partition the inputs with these new values.

4 Generate test cases for missing coverage using the `sldvgencov` function and the current test coverage.

Note: To view an example of extending an existing test suite to achieve missing model coverage, enter the following at the command prompt in the MATLAB Command Window:

    showdemo('sldvdemo_incremental_test_generation')

---

5 Repeat steps 3 and 4 until you have achieved the desired coverage.

Partition the model inputs that enable further simplification when an analysis runs. Consider the following model, which has three mutually independent enabled subsystems:

- Normal Mode
- Shutdown Mode
- Failure Mode
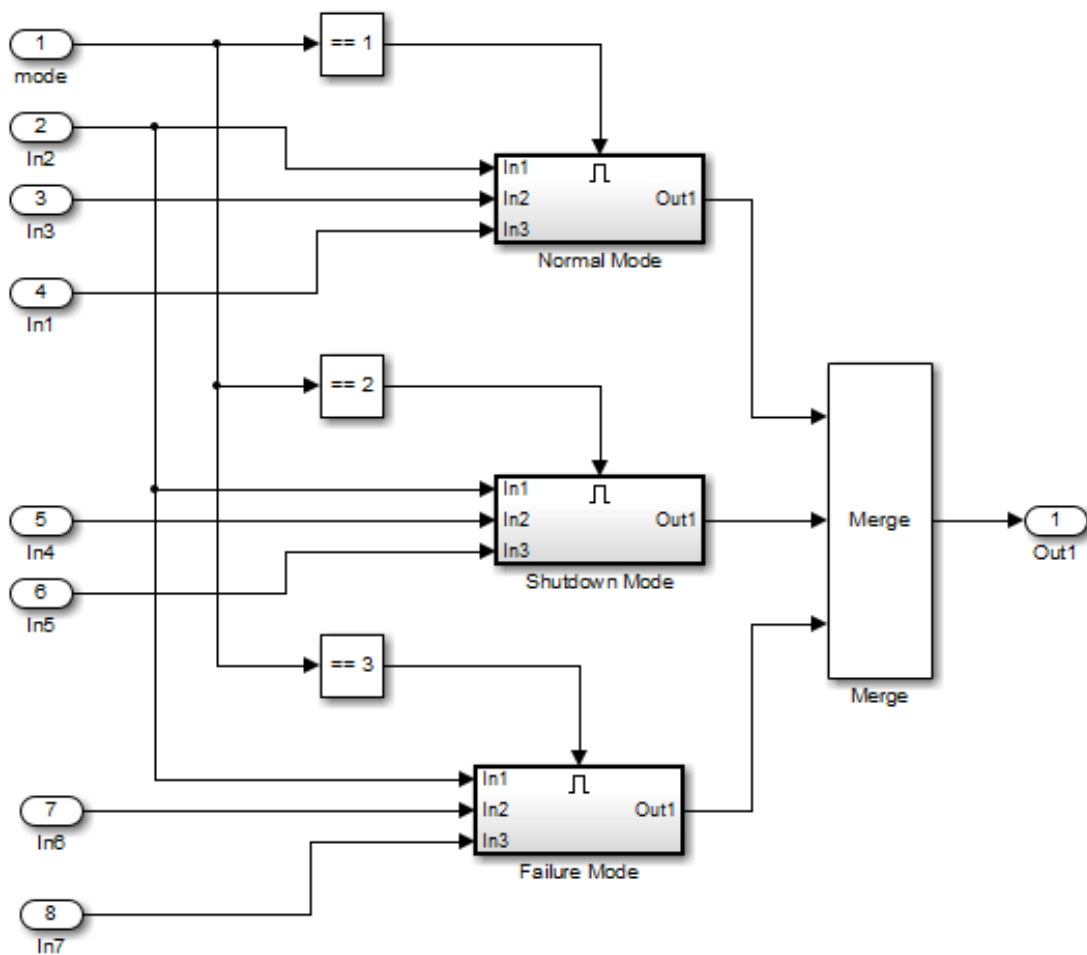
You can incrementally generate test cases for each subsystem by constraining the first input to a constant value before running an analysis. In this way, as you create test cases for each subsystem, the software ignores the complexity of the other two subsystems.

Bottom-Up Approach to Model Analysis

Simulink Design Verifier software works most effectively at analyzing large models using a bottom-up approach. In this approach, the software analyzes smaller model components first, which can be faster than using the `Large model` test suite optimization.

The bottom-up approach offers several advantages:

- It allows you to solve the problems that slow down error detection, test generation, or property proving in a controlled environment.
- Solving problems with small model components before analyzing the model as a whole is more efficient, especially if you have unreachable components in your model that you can only discover in the context of the model.
- You can iterate more quickly—find a problem and fix it, find another problem and fix it, and so on.
- If one model component has a problem—for example, a component is unreachable in simulation—that can prevent the software from generating tests for *all* the objectives in a large model.

Try this workflow with your large model:

1 Use the Test Generation Advisor to identify analyzable model components and generate tests for these components. For more information, see "Use Test Generation Advisor to Identify Analyzable Components" on page 7-21.

2 Fix any problems by adding constraints or specifying block replacements.

3 After you analyze the smaller components, reapply the required constraints and substitutions to the original model. Analyze the full model.

   When you finish a bottom-up analysis, you have a top-level model that Simulink Design Verifier can analyze quickly.

Extract Subsystems for Analysis

In this section...
"Overview of Subsystem Extraction" on page 14-16
"sldvextract Function" on page 14-16
"Structure of the Extracted Model" on page 14-17
"Analyze Subsystems That Read from Global Data Storage" on page 14-17
"Analyze Function-Call Subsystems" on page 14-20

Overview of Subsystem Extraction

If you have a large model that slows down your analysis or has unreachable objectives, you may want to analyze atomic subsystems or Stateflow atomic subcharts using Simulink Design Verifier. This technique allows you to implement a bottom-up approach to analyzing a large model, as described in "Bottom-Up Approach to Model Analysis" on page 14-15.

When you analyze a subsystem or atomic subchart, the software:

- Extracts the subsystem or subchart into a new model.
- If required, adds blocks to the newly created model that replicate the execution context of the subsystem or subchart within its parent model.
- Analyzes the extracted model and produces results.

Note: The Simulink Design Verifier software can only analyze atomic subsystems and atomic subcharts.

For more information about analyzing subsystems, see "Generate Test Cases for a Subsystem" on page 1-26.

For more information about analyzing atomic subcharts, see "Analyze a Stateflow Atomic Subchart" on page 1-28.

---

sldvextract Function

The `sldvextract` function allows you to extract subsystems and atomic subcharts for component verification. By extracting the subsystem or atomic subchart, you can

verify the component in isolation from the rest of the system, allowing you to test the component algorithm. For more information, see "What Is Component Verification?" on page 10-2 and "Functions for Component Verification" on page 10-4.

Structure of the Extracted Model

When you analyze a subsystem or atomic subchart, Simulink Design Verifier creates a new model that contains the subsystem or atomic subchart, and any input and output ports that correspond to the ports connected to the original subsystem. The software assigns the following properties to the ports in the new model, as determined by compiling the original model:

- Data types
- Sample rates
- Signal dimensions

The software names the new model *subsystem_name*, where *subsystem_name* is the name of the subsystem.

The next sections provide examples of how Simulink Design Verifier extracts and analyzes subsystems.

Analyze Subsystems That Read from Global Data Storage

A *data store* is a repository to which you can write data, and from which you can read data, without having to connect an input or output signal directly to the data store.

You create a data store using a `Data Store Memory` block or a `Simulink.Signal` object. The Data Store Memory block or `Simulink.Signal` object represents the data store and specifies its properties. Every data store must have a unique name.

When you analyze a subsystem that reads data from a data store that is accessed outside the subsystem, the analysis:

- Adds a Data Store Memory block to the new model.
- Adds an input port that writes to the data store. Since the input writes to the data store, the data can have any values (within the specified data type) for the purpose of the Simulink Design Verifier analysis.

If the data store specifies minimum and maximum values, those values are assigned to the new input port.

The following example analyzes a subsystem in the `sl_subsys_fcncall8` example model:

1   Open the `sl_subsys_fcncall8` example model:

    `sl_subsys_fcncall8`

    This model defines a data store A, from which the atomic subsystem Reader reads data using a Data Store Read block.

2   Right-click the Reader subsystem and select **Design Verifier** > **Generate Tests for Subsystem**.

    The Simulink Design Verifier log window shows that the software extracts the subsystem into a new model named `Reader`, analyzes the extracted model, and offers you the choice of which results to produce.

3   Open the new `Reader` model that the software created in *<current_folder>*`\sldv_output\Reader`.
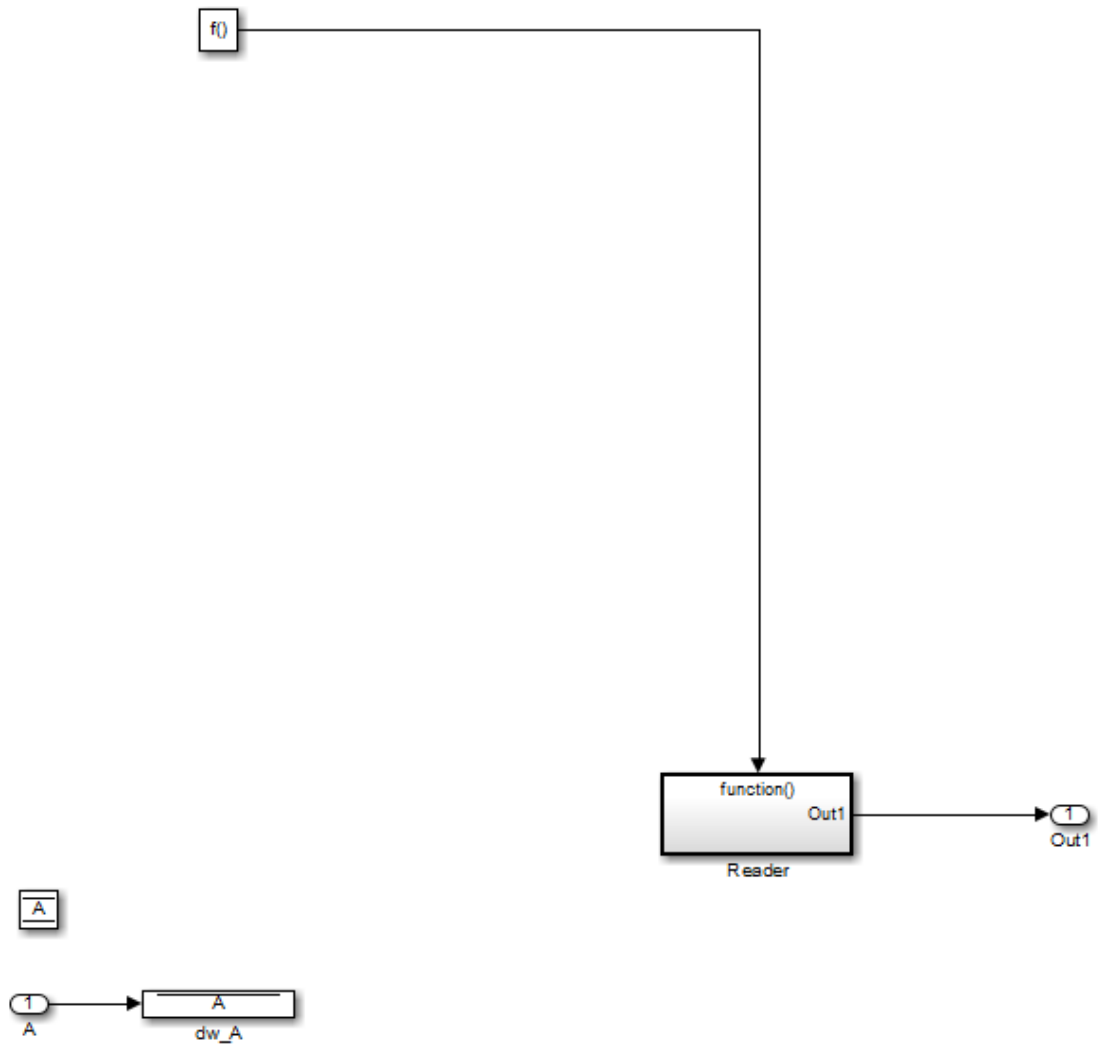
The new Inport block A writes into the data store, which is used by the subsystem Reader in the new model.

Analyze Function-Call Subsystems

A *function-call subsystem* is a triggered subsystem whose execution is determined by logic internal to a C MEX S-function instead of by the value of a signal. Function-call subsystems are always atomic.

Note: For more information, see "Function-Call Subsystems and S-Functions" in the Simulink documentation.

When you analyze a model with a function-call subsystem, Simulink Design Verifier creates a new model with an Inport block that mimics the trigger and a copy of the subsystem. The software then analyzes the new model.

The following example analyzes a function-call subsystem in the `sl_subsys_fcncall2` model:

1   Open the `sl_subsys_fcncall2` example model:

    `sl_subsys_fcncall2`

    This model contains a Stateflow chart named Chart that triggers the function-call subsystem f.

2   Right-click the f subsystem and select **Design Verifier** > **Generate Tests for Subsystem**.

    The software extracts the subsystem into a new model named `f`, analyzes the extracted model, and produces results.
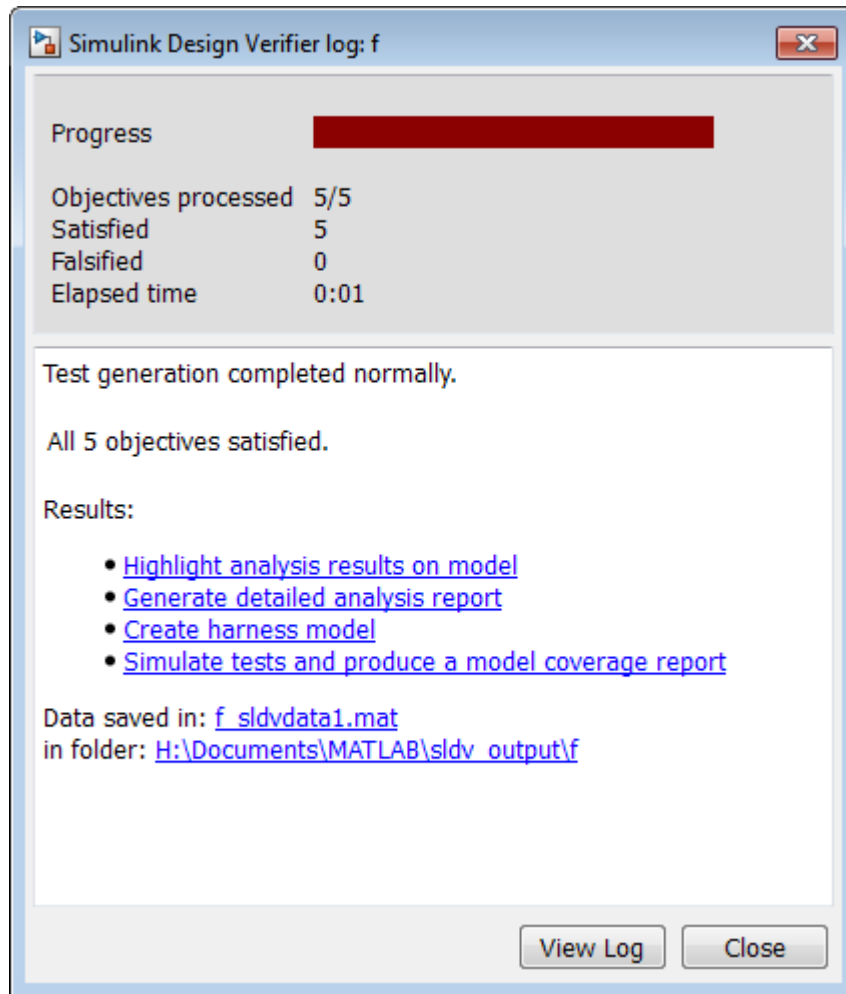
3   Open the f model that the software created in *<current_folder>*\sldv_output \f.

The Inport block and the new subsystem block mimic the trigger for the function-call subsystem f in the new f model.
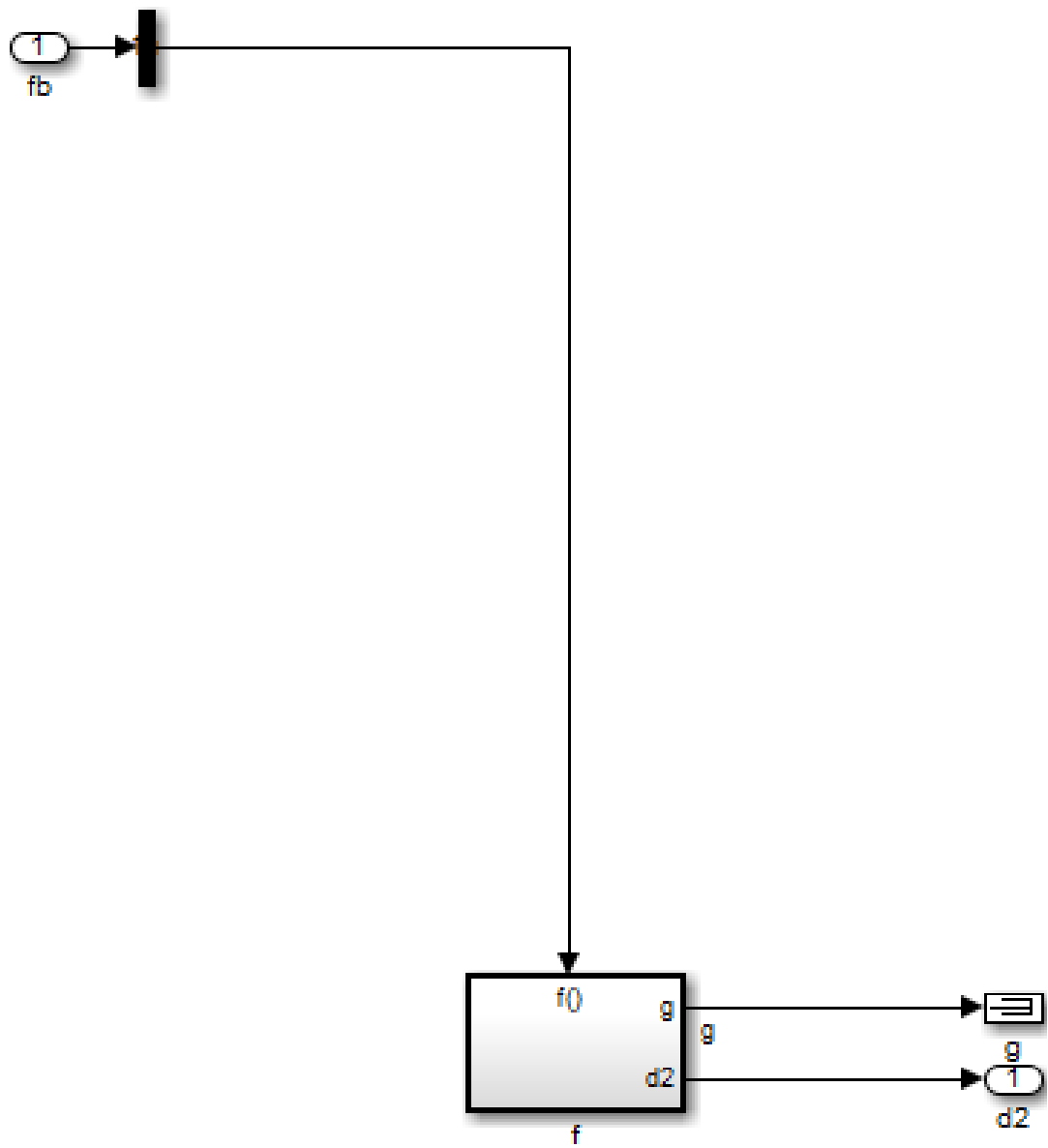
Logical Operations

If you have a Simulink model with both logical and arithmetic operations, consider analyzing only the logical operations.

The Simulink Design Verifier software does not support nonlinear arithmetic of floating-point numbers, as occurs with multiplication or division, unless one of the multiply operands or the divisor is a constant.

To simplify models that contain integers or floating-point numbers, the software maps the model computations into expressions of Boolean variables. For example, the software might represent an eight-bit number as a set of eight Boolean values, with one for each digit. It might represent a bit-wise OR operation of two eight-bit integers as eight separate logical OR operations.

Mapping problems of one data type into Boolean variables is complex, and this complexity increases when the software performs such mapping. The software handles models with predominantly logical signals more efficiently than it does those with large integer or floating-point signals.

Note: Simulink Design Verifier software can handle floating-point inputs when their values impact the design through linear inequalities such as $x < y$ or $a > 0$.

In addition, input complexity can result from certain cast operations. For example, casting a double to an int8 can introduce a nonlinearity in certain situations.

Models with Large Verification State Space

Abstract

Techniques to simplify the complexity of models with large verification state spaces.

Persistent design variables (variables that are assigned in one time step and used in a later time step during simulation) affect the complexity of analysis in much the same way as input complexity. You can use one or more of the following techniques to simplify the complexity of the state space you want to search:

- Apply constraints to input signals that are delayed.

- Constrain the inputs to states that are contained within conditionally executed subsystems.

- Limit the number of test case steps by setting the **Maximum test case step** parameter to `20`.

- Increase the sample time for part or all of the model. (This procedure is similar to reducing timer thresholds, as described in "Counters and Timers" on page 14-25.) A test case that you generate at a lower sample rate often has similarities to the test case with a high sample rate that you need to achieve an objective.

- Use tight variable types where ever possible. For example, if a flag with values of 0 or 1 only is defined as a `double`, restrict the type to `Boolean`.

States that are computed from previous state values present a special challenge. For example, if you want to restrict the integrator value in a PID controller, you can only use a set of values that includes all reachable values from the initial value. Otherwise, the input must be forced to `0`. Neither of these limitations is practical and would probably make the analysis less complete.

Alternatively, you can use existing simulation data to help satisfy your testing needs. If you have existing test data, run it on your model and collect model coverage. For an example of extending an existing test suite to achieve missing model coverage, see Extending an Existing Test Suite.

Counters and Timers

Abstract

Best practices for handling counters and timers in your model to avoid overcomplicating Simulink Design Verifier analysis.

Simulink Design Verifier analysis searches through sequences of states to find input values that drive the analysis to reach a state that satisfies an objective. Each counter value or timer step corresponds to a different state, so the presence of long timers or counters can dramatically increase the size of the state representation. Since analysis complexity depends on the size of the state representation, you must give special consideration to counters and timers in your model to avoid overcomplicating Simulink Design Verifier analysis.

Note: For the purposes of Simulink Design Verifier analysis, the term *configuration* refers to a set of values for all the persistent information in your model.

The search process investigates all configurations that can be reached in a single timer step before considering any of the configurations that can be reached in two timer steps. Likewise, the search investigates all configurations that can be reached in two timer steps before it considers any configuration that requires three or more timer steps, and so on. The number of timer steps required to exhaust the counter directly affects the number of states that the analysis needs to search. Models that contain time delays, such as countdown timers, complicate the analysis by forcing the search to span a large number of states.

You may see similar effects when systems use extensive averaging and filtering to delay the response to a change in inputs. Any aspect of the design that delays the response causes the test sequences to contain more timer steps, resulting in longer test cases that are more difficult to identify.

Some basic techniques you can use to improve analysis performance in models with counters or timers include the following:

• Choose very small values for time delays. A system with a logical error when a time delay is set to 2000 steps usually demonstrates that error if the time delay is changed to 2 steps. If your system has several delays, choose small but unique values for each of them so that your delays are progressively satisfied.

14-25

- Make the initial values of counters and timers parameter values that Simulink Design Verifier can modify. The software finds initial values that allow shorter test cases to exceed thresholds. For more information, see "Parameter Constraint Values" on page 5-2.

- Choose higher frequency cutoffs for filters and fewer samples to average to minimize filtering delays.

Some more advanced techniques you can use to improve analysis performance in models with counters or timers include the following:

- Use `sldvtimer` to identify timer patterns that can be optimized for Simulink Design Verifier test generation.

- Use an existing test case or set of test cases that exhausts the counter or timer, and extend those test cases to create a full test suite. For more information, see Defining and Extending Existing Test Cases.

Prove Properties in Large Models

Abstract

Describes workflows and best practices for proving properties in large models.

Property proving uses the same underlying techniques as design error detection and test generation and suffers from the same performance limitations. However, unlike design error detection or test generation, you often cannot simplify the problem without compromising the validity of the results.

You can quickly prove simple proof objectives that are not affected by model dynamics. However, a thorough proof requires that Simulink Design Verifier search through all reachable configurations of your model—even the ones that are reached only after long time delays. The computation time and memory required to search a model completely often make an exhaustive proof impractical.

There are two techniques you can use to improve the performance of property proving in a large model:

In this section...
"Find Property Violations While Designing Your Model" on page 14-27
"Combine Proving Properties and Finding Proof Violations" on page 14-28

Find Property Violations While Designing Your Model

Simulink Design Verifier software offers a strategy that quickly identifies property violations in larger, more complicated models. While designing your model, analyze your model using this strategy so that you can fix any property violations before finalizing your design.

To identify property violations of a model, on the **Design Verifier > Property Proving** pane of the Configuration Parameters dialog box, specify the value of the **Strategy** parameter as `FindViolation`. When you use this strategy, the **Maximum violation steps** parameter becomes active so that you can specify an upper bound for the number of time steps in the search.

When analyze the model, the software searches only for property violations within the specified number of time steps. By identifying and fixing the property violations first, you improve the performance of a property-proving analysis that uses the `Prove` strategy.

If a violation is not detected, it is impossible to violate the property with any input sequence having fewer time steps than the specified limit. However, you cannot prove that the property is true because there might be a counterexample within more time steps than the specified limit.

Combine Proving Properties and Finding Proof Violations

Use the following technique for proving properties in large model. This technique combines proving and searching for violations:

1 On the **Design Verifier > Property Proving** pane, set the **Strategy** parameter to `Prove`.

2 On the **Design Verifier** pane, use a relatively short value for the **Maximum analysis time** parameter, such as 5–10 minutes. If trivial counterexamples exist — or if your properties do not depend on model dynamics—the analysis should complete in that amount of time.

3 Change the **Strategy** parameter to `FindViolation`, and choose a small bound for the **Maximum violation steps** parameter, such as 4, 5, or 6. If your properties have simple counterexamples, the software should discover them.

4 If you do not find any violations with a small bound, increase the bound and look for longer counterexamples.

   a Increase the bound in several increments, and observe the processing time and memory consumption. System resources might limit the length of violation that can be searched.

   b In addition, consider the dynamics of your model and the number of time steps required to transition between an arbitrary pair of configurations. If you choose too large a bound, the violation search can be more complex than the unbounded proof.

5 If you can run violation searches with relatively large bounds, e.g., 30–50 time steps, switch back to the `Prove` strategy, and use a longer time limit, such as several hours.

15

Simulink Design Verifier Configuration Parameters

Simulink Design Verifier Options

In this section...
"Options in Configuration Parameters Dialog Box" on page 15-2
"Design Verification Options Objects" on page 15-2
"Command-Line Parameters for Design Verification Options" on page 15-2

Options in Configuration Parameters Dialog Box

You can set options for Simulink Design Verifier analysis in the Configuration Parameters dialog box. To view the options, select **Analysis > Design Verifier > Options**. The **Design Verifier** pane of the model configuration parameters opens.

By default, Simulink Design Verifier options do not appear in the Configuration Parameters dialog box. In the Simulink Editor, when you select **Analysis > Design Verifier > Options**, Simulink Design Verifier initially associates its default options with that model. After you save the model, you can access Simulink Design Verifier options directly from the Configuration Parameters dialog box or from the Model Explorer.

See "Configuration Parameters Dialog Box Overview" for more information about working with this interface.

Design Verification Options Objects

You can use the `sldvoptions` function to specify Simulink Design Verifier options at the command line.

To view in the MATLAB Command Window the design verification options associated with a Simulink model, use the following syntax:

```
opts = sldvoptions('model_name');
get(opts)
```

Command-Line Parameters for Design Verification Options

Use the following parameters to configure the behavior of Simulink Design Verifier. Use the `get_param` and `set_param` functions to retrieve and specify values for these parameters programmatically.

For each parameter, the **Location** column indicates where you can set its value in the Configuration Parameters dialog box. The **Values** column shows the type of value required, the possible values (separated with a vertical line), and the default value (enclosed in braces).

Parameter	Location	Values
DVAbsoluteTolerance	Set by the **Floating point absolute tolerance** parameter on the **Design Verifier > Test Generation** pane.	double {'1.0e-05'}
DVAssertions	Set by the **Assertion blocks** parameter on the **Design Verifier > Property Proving** pane.	'EnableAll' | 'DisableAll' | {'UseLocalSettings'}
DVAutomaticStubbing	Set by the **Automatic stubbing of unsupported blocks and functions** parameter on the **Design Verifier** pane.	{'on'} | 'off'
DVBlockReplacement	Set by the **Apply block replacements** parameter on the **Design Verifier > Block Replacements** pane.	'on' | {'off'}
DVBlockReplacement-ModelFileName	Set by the **File path of the output model** parameter on the **Design Verifier > Block Replacements** pane.	character array {'$ModelName$_replacement'}
DVBlockReplacement-RulesList	Set by the **List of block replacement rules** parameter on the **Design Verifier > Block Replacements** pane.	character array {'<FactoryDefaultRules>'}
DVCoverageDataFile	Set by the **Coverage data file** parameter on the **Design Verifier > Test Generation** pane.	character array {''}

Parameter	Location	Values
`DVCovFilter`	Set by the **Ignore objectives based on filter** parameter on the **Design Verifier > Test Generation** pane.	`'on'	{'off'}`
`DVCovFilterFileName`	Set by the **Coverage filter file** parameter on the **Design Verifier > Test Generation** pane.	character array `{''}`
`DVDataFileName`	Set by the **Data file name** parameter on the **Design Verifier > Results** pane.	character array `{'$ModelName $_sldvdata'}`
`DVDesignMinMaxCheck`	Set by the **Check specified intermediate minimum and maximum values** parameter on the **Design Verifier > Design Error Detection** pane.	`'on'	{'off'}`
`DVDesignMinMax-Constraints`	Set by the **Use specified input minimum and maximum values** parameter on the **Design Verifier** pane.	`{'on'}	'off'`
`DVDetectDeadLogic`	Set by **Dead logic** on the **Design Verifier > Design Error Detection** pane.	`'on'	{'off'}`
`DVDetectDivisionByZero`	Set by the **Division by zero** parameter on the **Design Verifier > Design Error Detection** pane.	`{'on'}	'off'`
`DVDetectIntegerOverflow`	Set by the **Integer overflow** parameter on the **Design Verifier > Design Error Detection** pane.	`{'on'}	'off'`

Parameter	Location	Values
DVDetectOutOfBounds	Set by the **Out of bound array access** parameter on the **Design Verifier > Design Error Detection** pane.	'on' | {'off'}
DVDisplayReport	Set by the **Display report** parameter on the **Design Verifier > Report** pane.	{'on'} | 'off'
DVDisplayResultsOnModel	Set by the **Display results of the analysis on the model** parameter on the **Design Verifier > Results** pane.	'on' | {'off'}
DVDisplayUnsatisfiable-Objectives	Set by the **Display unsatisfiable test objectives** parameter on the **Design Verifier** pane.	'on' | {'off'}
DVExtendExistingTests	Set by the **Extend existing test cases** parameter on the **Design Verifier > Test Generation** pane.	'on' | {'off'}
DVExistingTestFile	Set by the **Data file** parameter on the **Design Verifier > Test Generation** pane.	character array {''}
DVHarnessModelFileName	Set by the **Harness model file name** parameter on the **Design Verifier > Results** pane.	character array {'$ModelName $_harness'}
DVIgnoreCovSatisfied	Set by the **Ignore objectives satisfied in existing coverage data** parameter on the **Design Verifier > Test Generation** pane.	'on' | {'off'}

Parameter	Location	Values
`DVIgnoreExistTest-Satisfied`	Set by the **Ignore objectives satisfied by existing test cases** parameter on the **Design Verifier > Test Generation** pane.	`{on'}| 'off'`
`DVIncludeRelational-Boundary`	Set by the **Include relational boundary objectives** parameter on the **Design Verifier > Test Generation** pane.	`{'on'} | 'off'`
`DVMakeOutputFilesUnique`	Set by the **Make output file names unique by adding a suffix** check box on the **Design Verifier** pane.	`{'on'} | 'off'`
`DVMaxProcessTime`	Set by the **Maximum analysis time** parameter on the **Design Verifier** pane.	double `{'300'}`
`DVMaxTestCaseSteps`	Set by the **Maximum test case steps** parameter on the **Design Verifier > Test Generation** pane.	int32 `{'10000'}`
`DVMaxViolationSteps`	Set by the **Maximum violation steps** parameter on the **Design Verifier > Property Proving** pane.	int32 `{'20'}`
`DVMode`	Set by the **Mode** parameter on the **Design Verifier** pane.	`{'TestGeneration'} | 'DesignErrorDetection' | 'PropertyProving'`
`DVModelCoverage-Objectives`	Set by the **Model coverage objectives** parameter on the **Design Verifier > Test Generation** pane.	`'None' | 'Decision' | {'ConditionDecision'} | 'MCDC'`

Parameter	Location	Values
DVModelReferenceHarness	Set by the **Reference input model in generated harness** parameter on the **Design Verifier > Results** pane of the Configuration Parameters dialog box.	`'on'` | `{'off'}`
DVOutputDir	Set by **Output folder** on the **Design Verifier** pane.	character array `{'sldv_output/ $ModelName$'}`
DVParameterConstraints	Set by **Constraint** column in Parameter Table on the **Design Verifier > Parameters** pane.	double array `{[]}`
DVParameterNames	Set by **Name** column in Parameter Table on the **Design Verifier > Parameters** pane.	double array `{[]}`
DVParameterUseInAnalysis	Set by **Use** column in Parameter Table on the **Design Verifier > Parameters** pane.	cell array `{[]}`
DVParameters	Set by **Enable parameter configuration** on the **Design Verifier > Parameters** pane.	`'on'` | `{'off'}`
DVParametersConfigFile-Name	Set by **Parameter configuration file** on the **Design Verifier > Parameters** pane. This parameter is disabled when `DVParametersUseConfig` is set to `'on'`.	character array `{'sldv_params_template.m'}`

Parameter	Location	Values
`DVParametersUseConfig`	Set by **Use parameter table** on the **Design Verifier > Parameters** pane. When set to `'on'`, this parameter disables `DVParametersConfig-FileName`.	`'on'` | `{'off'}`
`DVProofAssumptions`	Set by the **Proof assumptions** parameter on the **Design Verifier > Property Proving** pane.	`'EnableAll'` | `'DisableAll'` | `{'UseLocalSettings'}`
`DVProvingStrategy`	Set by the **Strategy** parameter on the **Design Verifier > Property Proving** pane.	`'FindViolation'` | `{'Prove'}` | `'ProveWithViolationDetection'`
`DVRandomizeNoEffectData`	Set by the **Randomize data that do not affect the outcome** parameter on the **Design Verifier > Results** pane.	`'on'` | `{'off'}`
`DVRelativeTolerance`	Set by the **Floating point relative tolerance** parameter on the **Design Verifier > Test Generation** pane.	double `{'0.01'}`
`DVReportFileName`	Set by the **Report file name** parameter on the **Design Verifier > Report** pane.	character array `{'$ModelName$_report'}`
`DVReportIncludeGraphics`	Set by the **Include screen shots of properties** parameter on the **Design Verifier > Report** pane.	`'on'` | `{'off'}`

Parameter	Location	Values
DVReportPDFFormat	Set by the **Generate additional report in PDF format** parameter on the **Design Verifier > Report** pane.	`'on'` | `{off'}`
DVSaveDataFile	Set by the **Save test data to file** parameter on the **Design Verifier > Results** pane.	`{'on'}` | `'off'`
DVSaveExpectedOutput	Set by the **Include expected output values** parameter on the **Design Verifier > Results** pane.	`'on'` | `{'off'}`
DVSaveHarnessModel	Set by the **Generate separate harness model after analysis** parameter on the **Design Verifier > Results** pane.	`'on'` | `{off'}`
DVSaveReport	Set by the **Generate report of the results** parameter on the **Design Verifier > Report** pane.	`'on'` | `{off'}`
DVSFcnSupport	Set by the **Support S-Functions in the analysis** parameter on the **Design Verifier** pane.	`{'on'}` | `off'`
DVSlTestHarnessName	Set by the **Test Harness Name** parameter on the **Design Verifier > Results** pane.	character array `{'$ModelName $_sldvharness'}`
DVSlTestFileName	Set by the **Test File Name** parameter on the **Design Verifier > Results** pane.	character array `{'$ModelName $_test'}`

Parameter	Location	Values
DVTestConditions	Set by the **Test conditions** parameter on the **Design Verifier > Test Generation** pane.	'EnableAll' | 'DisableAll' | {'UseLocalSettings'}
DVTestObjectives	Set by the **Test objectives** parameter on the **Design Verifier > Test Generation** pane.	'EnableAll' | 'DisableAll' | {'UseLocalSettings'}
DVTestSuiteOptimization	Set by the **Test suite optimization** parameter on the **Design Verifier > Test Generation** pane.	{'CombinedObjectives'} | 'IndividualObjectives' | 'LargeModel' | 'LongTestcases' | 'CombinedObjectives (Nonlinear Extended)' | 'LargeModel (Nonlinear Extended)'

Design Verifier Pane

In this section...

Design Verifier Pane Overview

Specify analysis options and configure Simulink Design Verifier output.

Mode

Specify the analysis mode for Simulink Design Verifier.

Settings

Default: `Test generation`

`Design error detection`

　　Detects integer and fixed-point overflow errors and division-by-zero errors in a model

`Test generation`

　　Generates test cases for a model.

`Property proving`

　　Proves properties of a model.

Tip

The Simulink Design Verifier software specifies the value of this option automatically when you select one of the following menu options:

- **Analysis** > **Design Verifier** > **Generate Tests**
- **Analysis** > **Design Verifier** > **Detect Design Errors**
- **Analysis** > **Design Verifier** > **Prove Properties**

Dependency

Selecting `Test generation` enables the **Display unsatisfiable test objectives** parameter.

When you set the **Mode** parameter, the button below **Check Model Compatibility** changes as follows:

- **Mode**: `Test generation`, button reads: **Generate Tests**
- **Mode**: `Design error detection`, button reads: **Detect Errors**
- **Mode**: `Property proving`, button reads: **Prove Properties**

Command-Line Information

Parameter: `DVMode`
Type: character array
Value: `'TestGeneration'` | `'DesignErrorDetection'` | `'PropertyProving'`
Default: `'TestGeneration'`

See Also

- "Basic Workflow for Simulink Design Verifier" on page 1-31
- "What Is Design Error Detection?" on page 6-2
- "What Is Test Case Generation?" on page 7-2
- "What Is Property Proving?" on page 12-2

Maximum analysis time

Specify the maximum time (in seconds) that Simulink Design Verifier spends analyzing a model.

Settings

Default: 300

The value that you enter represents the maximum number of seconds Simulink Design Verifier analyzes your model.

Command-Line Information
Parameter: DVMaxProcessTime
Type: double
Value: any valid value
Default: 300

Display unsatisfiable test objectives

Specify whether to display warnings if the analysis detects unsatisfiable test objectives.

Settings

Default: Off

☑ On

> Displays a warning in the Simulation Diagnostics Viewer when Simulink Design Verifier is unable to satisfy a test objective.

☐ Off

> Does not display a warning when Simulink Design Verifier is unable to satisfy a test objective.

Tip If you select **Display unsatisfiable test objectives**, on the **Test Generation** pane, set **Test suite optimization** to `CombinedObjectives`. If you perform test-generation analysis on your model and the returned test objectives do not have outcomes, set **Test suite optimization** to `IndividualObjectives` and reanalyze the model. The `IndividualObjectives` strategy analyzes each objective independently and identifies unsatisfiable objectives.

Command-Line Information
Parameter: `DVDisplayUnsatisfiableObjectives`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

Automatic stubbing of unsupported blocks and functions

Specify whether to ignore unsupported blocks and functions during analysis.

Settings

Default: On

☑ On

> Ignores unsupported blocks and functions and proceeds with the analysis.

☐ Off

> Displays a warning when Simulink Design Verifier encounters an unsupported block or function and asks if you want to continue the analysis.

Command-Line Information
Parameter: DVAutomaticStubbing
Type: character array
Value: 'on' | 'off'
Default: 'on'

See Also

"Handle Incompatibilities with Automatic Stubbing" on page 2-8

Support S-Functions in the analysis

Specify whether to enable support for S-Functions that have been compiled to be compatible with Simulink Design Verifier.

Settings

Default: On

☑ On

> Enables support for S-Functions that have been compiled to be compatible with Simulink Design Verifier.

☐ Off

> Simulink Design Verifier automatically stubs S-Functions during analysis.

Command-Line Information

Parameter: DVSFcnSupport
Type: character array
Value: 'on' | 'off'
Default: 'on'

See Also

"Support Limitations for S-Functions" on page 3-35

Configuring S-Function for Test Case Generation

"Handle Incompatibilities with Automatic Stubbing" on page 2-8

Use specified input minimum and maximum values

Specify whether to generate test cases that consider specified minimum and maximum values as constraints for all input signals in your model.

Settings

Default: On

☑ On

> Considers specified minimum and maximum values as constraints for all input signals.

☐ Off

> Ignores any specified minimum and maximum values.

Command-Line Information

Parameter: DVDesignMinMaxConstraints
Type: character array
Value: 'on' | 'off'
Default: 'on'

See Also

"Minimum and Maximum Input Constraints" on page 11-2

Output folder

Specify a path name to which Simulink Design Verifier writes its output.

Settings

Default: `sldv_output/$ModelName$`

- Enter a path that is either absolute or relative to the current folder.
- `$ModelName$` is a token that represents the model name.

Tip

You can use the following parameters to customize the names and locations of Simulink Design Verifier output:

- On the **Results** pane:
    - **Data file name**
    - **Harness model file name**
    - **Simulink Test options > Test File name**
- On the **Report** pane:
    - **Report file name**
    - **File path of the output model**
- On the **Block Replacements** pane:
    - **File path of the output model**

Command-Line Information
Parameter: `DVOutputDir`
Type: character array
Value: any valid path
Default: `'sldv_output/$ModelName$'`

See Also

"Results Interpretation and Use"

Make output file names unique by adding a suffix

Specify whether Simulink Design Verifier makes its output file names unique by appending a numeric suffix.

Settings

Default: On

☑ On

> Appends an incremental numeric suffix to Simulink Design Verifier output file names. Selecting this option prevents the software from overwriting existing files that have the same name.

☐ Off

> Does not append a suffix to Simulink Design Verifier output file names. In this case, the software might overwrite existing files that have the same name.

Command-Line Information

Parameter: `DVMakeOutputFilesUnique`
Type: character array
Value: `'on'` | `'off'`
Default: `'on'`

See Also

"Results Interpretation and Use"

Check Model Compatibility

Run a check to assess your model for compatibility with Simulink Design Verifier. For more information, see "Simulink Design Verifier Checks".

Generate Tests/Detect Errors/Prove Properties

When you set the **Mode** parameter, this button changes as follows:

- **Mode**: `Test generation`, button reads: **Generate Tests**

  For more information, see "What Is Test Case Generation?" on page 7-2.

- **Mode**: `Design error detection`, button reads: **Detect Errors**

  For more information, see "What Is Design Error Detection?" on page 6-2.

- **Mode**: `Property proving`, button reads: **Prove Properties**

  For more information, see "What Is Property Proving?" on page 12-2.

Design Verifier Pane: Block Replacements

In this section...

Block Replacements Pane Overview

Specify options that control how Simulink Design Verifier preprocesses the models it analyzes.

See Also

"Block Replacement"

Apply block replacements

Specify whether Simulink Design Verifier replaces blocks in a model before its analysis.

Settings

Default: Off

☑ On

Replaces blocks in a model before Simulink Design Verifier analyzes it.

☐ Off

Does not replace blocks in a model before Simulink Design Verifier analyzes it.

Dependencies

This parameter enables **List of block replacement rules** and **File path of the output model**.

Command-Line Information

Parameter: DVBlockReplacement
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

"Block Replacement"

List of block replacement rules

Specify a list of block replacement rules that Simulink Design Verifier executes before its analysis.

Settings

Default: `<FactoryDefaultRules>`

- Specify block replacement rules as a list delimited by spaces, commas, or carriage returns.
- The Simulink Design Verifier software processes block replacement rules in the order that you list them.
- If you specify the default value, Simulink Design Verifier uses its factory default block replacement rules.

Dependency

This parameter is enabled when you select **Apply block replacements**.

Command-Line Information
Parameter: `DVBlockReplacementRulesList`
Type: character array
Value: any valid rules
Default: `'<FactoryDefaultRules>'`

See Also

"Block Replacement"

File path of the output model

Specify a folder and file name for the model that results after applying block replacement rules.

Settings

Default: $ModelName$_replacement

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.
- Enter a file name for the model that results after applying block replacement rules.
- $ModelName$ is a token that represents the model name.

Dependency

This parameter is enabled when you select **Apply block replacements**.

Command-Line Information

Parameter: DVBlockReplacementModelFileName
Type: character array
Value: any valid path and file name
Default: '$ModelName$_replacement'

See Also

"Block Replacement"

Design Verifier Pane: Parameters

Parameters
- ☑ Enable parameter configuration   ☐ Use parameter table

Parameter configuration file: sldv_params_template.m   [Browse...] [Edit...]

Parameter table

[Enable] [Disable] [Clear]   [Highlight in Model]

Use	Name	Constraint	Value	Min	Max	Model Element

[Find in Model] [Add from File...] [Export to File...]

Parameters Pane Overview

Specify options that control how Simulink Design Verifier uses parameter configurations when analyzing models.

Enable parameter configuration

Specify whether the software uses parameter configurations when analyzing a model. Select this option to treat parameters as variables in Simulink Design Verifier analysis.

To specify value ranges or constraints for parameters:

- Use a parameter configuration file. Enter the file name in **Parameter configuration file**.
- Use the Parameter Table. Select **Use parameter table**.

Settings

Default: Off

☑ On

The Simulink Design Verifier software uses specified parameter configurations when analyzing a model.

☐ Off

The Simulink Design Verifier software does not use parameter configurations when analyzing a model.

Tips

When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. In **Analysis** > **Design Verifier** > **Options**, if you specify your model to extend existing test cases with a **Data file** and apply parameter configurations with a **Parameter configuration file** or the Parameter Table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.

Dependency

This parameter enables **Parameter configuration file**.

Command-Line Information

Parameter: `DVParameters`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Define Constraint Values for Parameters" on page 5-4

Use parameter table

Enable the Parameter Table to specify value ranges or constraints for parameters.

Settings

Default: Off

☑ On

Use the Parameter Table to define parameters as variables for Simulink Design Verifier analysis.

☐ Off

Do not use the Parameter Table to define parameters as variables for Simulink Design Verifier analysis.

Tips

When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. In **Analysis** > **Design Verifier** > **Options**, if you specify your model to extend existing test cases with a **Data file** and apply parameter configurations with a **Parameter configuration file** or the Parameter Table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.

Dependency

When **Enable parameter configuration** is also selected, this parameter enables the Parameter Table.

This parameter disables **Parameter configuration file**.

Command-Line Information
Parameter: DVParametersUseConfig
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

"Define Constraint Values for Parameters" on page 5-4

Parameter configuration file

Specify a MATLAB function that defines parameter configurations for a model.

Settings

Default: sldv_params_template.m

- The default file, sldv_params_template.m, is a template that you can edit and save. The comments in the template explain the syntax you use to specify parameter configurations.
- Click the **Browse** button to select an existing MATLAB file.
- Click the **Edit** button to open the specified MATLAB file in an editor.

Tips

When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. In **Analysis** > **Design Verifier** > **Options**, if you specify your model to extend existing test cases with a **Data file** and apply parameter configurations with a **Parameter configuration file** or the Parameter Table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.

Dependency

This parameter is enabled by **Enable parameter configuration**. This parameter is disabled by **Use parameter table**.

Command-Line Information
Parameter: DVParametersConfigFileName
Type: character array
Value: any valid MATLAB file
Default: 'sldv_params_template.m'

See Also

"Define Constraint Values for Parameters" on page 5-4

Browse...

Browse to the parameter configuration file.

Dependency

This button is enabled by **Enable parameter configuration**. This button is disabled by **Use parameter table**.

Edit...

Edit the current parameter configuration file.

Dependency

This button is enabled by **Enable parameter configuration**. This button is disabled by **Use parameter table**.

Enable

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this button is enabled.

Disable

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this button is enabled.

Clear

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this button is enabled.

Highlight in Model

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this button is enabled.

Use

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Use** column specifies whether to use this rows's named parameter and specified constraint in the current parameter configuration.

Settings

Default: Off

☑ On

> Use this parameter and its specified constraint in the current parameter configuration.

☐ Off

Do not use this parameter and its specified constraint in the current parameter configuration.

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this column is enabled.

See Also

"Define Constraint Values for Parameters" on page 5-4

Name

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Name** column displays the name of the parameter.

Settings

Default: empty

Tips

To load the model parameters into the Parameter Table, at the bottom of the table, click **Find in Model**. When possible, the software automatically generates constraint values for each parameter.

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this column is enabled.

See Also

"Define Constraint Values for Parameters" on page 5-4

Constraint

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Constraint** column contains the specified value range for the parameter.

Settings

Default: empty

Tips

To autogenerate parameter constraints, at the bottom of the Parameter Table, click **Find in Model**.

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this column is enabled.

See Also

"Define Constraint Values for Parameters" on page 5-4

Value

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Value** column contains the value of the parameter in the base workspace. If the parameter is defined in a Simulink data dictionary that is linked to the model, the **Value** column contains the value of the parameter in the data dictionary.

Settings

Default: empty

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this column is enabled.

See Also

"Define Constraint Values for Parameters" on page 5-4

Min

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

For parameters of type `Simulink.Parameter` with a specified minimum value, the **Min** column contains the specified minimum value for the parameter.

Settings

Default: empty

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this column is enabled.

See Also

- "Define Constraint Values for Parameters" on page 5-4
- Simulink.Parameter

Max

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

For parameters of type `Simulink.Parameter` with a specified maximum value, the **Max** column contains the specified maximum value for the parameter.

Settings

Default: empty

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this column is enabled.

See Also

- "Define Constraint Values for Parameters" on page 5-4

- Simulink.Parameter

Model Element

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Model Element** column displays the path to the model elements where the parameter is used.

Settings

Default: empty

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this column is enabled.

See Also

"Define Constraint Values for Parameters" on page 5-4

Find in Model

The **Parameter Table** searches your model for parameters that can be configured and loads them in the table.

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this button is enabled.

Add from File...

Adds parameters to the **Parameter Table** from a list stored in a file.

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this button is enabled.

Export to File...

Exports the current parameters in the **Parameter Table** to a file.

Dependency

When **Enable parameter configuration** and **Use parameter table** are selected, this button is enabled.

Design Verifier Pane: Test Generation

In this section...

Test Generation Pane Overview

Specify options that control how Simulink Design Verifier generates tests for the models it analyzes.

See Also

Model coverage objectives

Specify the type of model coverage that Simulink Design Verifier attempts to achieve.

Settings

Default: `Condition Decision`

None

> Generates test cases that achieve only the custom objectives that you specified in your model using, for example, Test Objective blocks.

Decision

> Generates test cases that achieve decision coverage. For more information, see "Decision" on page 7-27.

Condition Decision

> Generates test cases that achieve condition and decision coverage. For more information, see "Condition" on page 7-27.

MCDC

> Generates test cases that achieve modified condition/decision coverage (MCDC). For more information, see "MCDC" on page 7-28.

When you set **Model coverage objectives** to `MCDC`, Simulink Design Verifier automatically enables every coverage objective for decision coverage and condition coverage as well. Similarly, enabling coverage for condition coverage causes every decision and condition coverage outcome to be enabled.

Command-Line Information
Parameter: `DVModelCoverageObjectives`
Type: character array
Value: `'None'` | `'Decision'` | `'ConditionDecision'` | `'MCDC'`
Default: `'ConditionDecision'`

See Also

"Workflow for Test Case Generation" on page 7-4

Test conditions

Specify whether Test Condition blocks in your model are enabled or disabled.

Settings

Default: `Use local settings`

`Use local settings`

Enables or disables Test Condition blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

`Enable all`

Enables all Test Condition blocks in the model regardless of the settings of their **Enable** parameters.

`Disable all`

Disables all Test Condition blocks in the model regardless of the settings of their **Enable** parameters.

Command-Line Information

Parameter: `DVTestConditions`
Type: character array
Value: `'UseLocalSettings'` | `'EnableAll'` | `'DisableAll'`
Default: `'UseLocalSettings'`

See Also

- Test Condition
- "Workflow for Test Case Generation" on page 7-4

Test objectives

Specify whether Test Objective blocks in your model are enabled or disabled.

Settings

Default: `Use local settings`

`Use local settings`

Enables or disables Test Objective blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

`Enable all`

Enables all Test Objective blocks in the model regardless of the settings of their **Enable** parameters.

`Disable all`

Disables all Test Objective blocks in the model regardless of the settings of their **Enable** parameters.

Command-Line Information

Parameter: `DVTestObjectives`
Type: character array
Value: `'UseLocalSettings'` | `'EnableAll'` | `'DisableAll'`
Default: `'UseLocalSettings'`

See Also

- Test Objective
- "Workflow for Test Case Generation" on page 7-4

Maximum test case steps

Specify the maximum number of simulation steps Simulink Design Verifier takes when attempting to satisfy a test objective.

The analysis uses the **Maximum test case steps** parameter during certain parts of the test-generation analysis to bound the number of steps that test generation uses. When you set a small value for this parameter, the parts of the analysis that are bounded complete in less time. When you set a larger value, the bounded parts of the analysis take longer, but it is possible for these parts of the analysis to generate longer test cases.

To achieve the best performance, set the **Maximum test case steps** parameter to a value just large enough to bound the longest required test case, even if the test cases that are ultimately generated are longer than this value.

When you also specify LongTestcases for the **Test suite optimization** parameter, the analysis uses successive passes of test generation to extend a potential test case so that it satisfies more objectives. When this happens, the analysis applies the **Maximum test case steps** parameter to each individual iteration of test generation.

Settings

Default: 10000

You can specify a value that represents the maximum number of simulation steps Simulink Design Verifier takes when attempting to satisfy a test objective.

Command-Line Information
Parameter: DVMaxTestCaseSteps
Type: int32
Value: any valid value
Default: 10000

See Also

"Workflow for Test Case Generation" on page 7-4

Test suite optimization

Specify the optimization strategy to use when generating test cases.

Settings

Default: CombinedObjectives (Nonlinear Extended)

CombinedObjectives (Nonlinear Extended)

Analyzes the model using a variation of the CombinedObjectives optimization. This optimization includes improved support for nonlinear arithmetic.

LargeModel (Nonlinear Extended)

Analyzes the model using a variation of the LargeModel optimization. This optimization includes improved support for nonlinear arithmetic.

IndividualObjectives

Maximizes the number of test cases in a suite by generating cases that each address only one test objective. Each test case tends to be short, i.e., it includes only a few time steps.

LongTestcases

Combines test cases to create a smaller number of test cases. This strategy generates fewer, but longer, test cases that each satisfy multiple test objectives and creates a more efficient analysis and easier-to-review results.

CombinedObjectives

Minimizes the number of test cases in a suite by generating cases that address more than one test objective. Each test case tends to be long, i.e., it includes many time steps.

LargeModel

Minimizes the number of test cases in a suite by generating cases that address more than one test objective. This strategy is tailored for large, complex models; consequently, it tends to use all the time that the **Maximum analysis time** option allots.

Tip

If you want to identify unsatisfiable objectives, set this option to IndividualObjectives. The IndividualObjectives strategy analyzes each objective independently, so it has a better chance of identifying unsatisfiable objectives.

If you have many test objectives or you want to create a smaller number of test cases, select `LongTestcases` for a more efficient analysis and an easy-to-review report.

If your model has both of the following characteristics:

- Nonlinearities, such as those that result from multiplying or dividing the model's input signals
- Numerous test objectives, such as those that result when using blocks that receive model coverage

set this option to `LargeModel (Nonlinear Extended)`. The `LargeModel` and `LargeModel (Nonlinear Extended)` strategies perform an analysis that is tailored to large, complex models. However, these strategies tend to use all the time that the **Maximum analysis time** option allots.

Command-Line Information
Parameter: `DVTestSuiteOptimization`
Type: character array
Value: `'CombinedObjectives (Nonlinear Extended)'` | `'LargeModel (Nonlinear Extended)'` | `'IndividualObjectives'` | `'LongTestcases'` | `'CombinedObjectives'` | `'LargeModel'` |
Default: `'CombinedObjectives (Nonlinear Extended)'`

See Also

"Workflow for Test Case Generation" on page 7-4

Extend existing test cases

Extend the Simulink Design Verifier analysis by importing test cases logged from a harness model or a closed-loop simulation model.

Settings

Default: Off

☑ On

Extends the analysis by using the logged test cases specified in **Data file**.

☐ Off

Does not extend the analysis.

Tips

When Simulink Design Verifier is configured to apply parameters specified in **Parameter configuration file**, you cannot use the **Extend existing test cases** option. If you specify your model to extend existing test cases with a **Data file** and apply parameter configurations with a **Parameter configuration file**, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.

Dependency

This parameter enables **Data file** and **Ignore objectives satisfied by existing test cases**.

Command-Line Information
Parameter: DVExtendExistingTests
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

- "When to Extend Existing Test Cases" on page 8-2
- "Common Workflow for Extending Existing Test Cases" on page 8-4

Data file

Specify a folder and file name for the MAT-file that contains the logged test case data.

Settings

Default: ' '

- Specify a folder and file name for the MAT-file that contains the logged test case data in an sldvData object.
- Click the **Browse** button to navigate to and select an existing file.

Tips

When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. In **Analysis >**

Design Verifier > **Options**, if you specify your model to extend existing test cases with a **Data file** and apply parameter configurations with a **Parameter configuration file** or the Parameter Table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.

Command-Line Information
Parameter: `DVExistingTestFile`
Type: character array
Value: any valid path and file name
Default: `''`

See Also

"Simulink Design Verifier Data Files" on page 13-9

Browse...

Browse to the MAT-file that contains the logged test case data.

Dependency

This button is enabled by **Extend existing test cases**.

Ignore objectives satisfied by existing test cases

Ignore the coverage objectives satisfied by the logged test cases in **Data file**.

Settings

Default: On

☑ On

> Generates results, but excludes coverage objectives satisfied by logged test cases in **Data file** from the analysis.

☐ Off

> Generates results for the full test suite, including coverage objectives satisfied by the logged test cases in **Data file**.

Command-Line Information

Parameter: `DVIgnoreExistTestSatisfied`
Type: character array
Value: `'on'` | `'off'`
Default: `'on'`

See Also

- "Extend Test Cases for Closed-Loop System" on page 8-12
- "Simulink Design Verifier Data Files" on page 13-9

Ignore objectives satisfied in existing coverage data

Specify to analyze the model, ignoring satisfied coverage objectives, as specified in **Coverage data file**.

Settings

Default: Off

☑ On

Ignores satisfied coverage objectives in **Coverage data file** during the analysis.

☐ Off

Generates results for all coverage objectives, including those in **Coverage data file**.

Dependency

This parameter enables **Coverage data file**.

Command-Line Information

Parameter: `DVIgnoreCovSatisfied`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

- "Achieve Missing Coverage in Closed-Loop Simulation Model" on page 9-12
- "Test Case Extension"

Coverage data file

Specify a folder and file name for the file that contains data about satisfied coverage objectives.

Settings

Default: ' '

- Specify the name of the folder and file name that contains the satisfied coverage objectives data

Click the **Browse** button to select an existing MATLAB file.

Command-Line Information
Parameter: DVCoverageDataFile
Type: character array
Value: any valid path and file name
Default: ' '

See Also

- "Achieve Missing Coverage in Closed-Loop Simulation Model" on page 9-12
- "Test Case Extension"

Browse...

Browse to the file that contains data about satisfied coverage objectives.

Dependency

This button is enabled by **Ignore objectives satisfied in existing coverage data**.

Ignore objectives based on filter

Specify to analyze the model, ignoring the coverage objectives in the **Coverage filter file**.

Settings

Default: Off

☑ On

> Ignores coverage objectives in the **Coverage filter file** during the analysis.

☐ Off

> Generates results for all coverage objectives, including those in **Coverage filter file**.

Dependency

This parameter enables **Coverage filter file**.

Command-Line Information
Parameter: DVCovFilter
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

"Coverage Filtering" in the Simulink Verification and Validation documentation

Coverage filter file

Specify a folder and file name for the file that contains the coverage objectives you want to ignore. The **Coverage filter file** specifies model objects to exclude from model coverage during test case generation.

Settings

Default: ''

- Specify the name of the folder and file name that contains the coverage objectives you want to ignore.

Click the **Browse** button to select an existing MATLAB file.

Command-Line Information
Parameter: DVCovFilterFileName
Type: character array
Value: any valid path and file name
Default: ''

See Also

"Coverage Filter Rules and Files" in the Simulink Verification and Validation documentation

Browse...

Browse to the file that contains the coverage objectives you want to ignore.

Dependency

This button is enabled by **Ignore objectives based on filter**.

Include relational boundary objectives

Specify generation of test cases that satisfy relational boundary objectives. The objective applies to blocks such as Relational Operator that have an explicit or implicit relational operation. The tests check the relational operations in these blocks with:

- Equal operand values for integer and fixed-point operands.
- Operand values within a certain tolerance for all operands. For integer and fixed-point operands, the tolerance is fixed. For floating-point operands, the tolerance is computed using the inputs and a tolerance value that you specify. If you do not specify a tolerance value, the default values are used.

Settings

Default: Off

☑ On

  For supported blocks, generated test cases satisfy relational boundary objectives.

☐ Off

  Generated test cases do not satisfy relational boundary objectives.

Dependencies

If you select this option, you can use default values or specify values for:

- "Floating point absolute tolerance" on page 15-55
- "Floating point relative tolerance" on page 15-56

Command-Line Information
Parameter: `DVIncludeRelationalBoundary`
Type: character array
Value: `'on'`|`'off'`
Default: `'off'`

See Also

- "Relational Boundary" on page 7-28
- "Model Objects That Receive Coverage" in the Simulink Verification and Validation documentation

Floating point absolute tolerance

Specify a value for absolute tolerance used in relational boundary tests. The relational boundary objectives apply to blocks such as `Relational Operator` that have an explicit or implicit relational operation. The tolerance value applies only if the relational operations in those blocks use floating point operands.

- For integer operands, the tolerance value is fixed at 1.
- For fixed-point operands, the tolerance value is the least significant bit.

Settings

Default: 1.0000e-05

For supported blocks, the relational boundary tests check the relational operations in the block with operand values that differ by a certain tolerance. The software calculates the tolerance value using the following formula

`max(absTol, relTol* max(|lhs|,|rhs|))`, where:

- `absTol` is the absolute tolerance value that you specify.
- `relTol` is a relative tolerance value that you can specify.
- `lhs` is the left operand and `rhs` the right operand.
- `max(x,y)` returns `x` or `y`, whichever is greater.

Dependencies

To enter a value for this option, select "Include relational boundary objectives" on page 15-54.

Command-Line Information
Parameter: DVAbsoluteTolerance
Type: double
Value: Any valid value
Default: 1.0000e-05

See Also

- "Relational Boundary" on page 7-28
- "Model Objects That Receive Coverage" in the Simulink Verification and Validation documentation

Floating point relative tolerance

Specify a value for relative tolerance used in relational boundary tests. The relational boundary objectives apply to blocks such as Relational Operator that have an explicit or implicit relational operation. The tolerance value applies only if the relational operations in those blocks use floating point operands.

- For integer operands, the tolerance value is fixed at 1.
- For fixed-point operands, the tolerance value is the least significant bit.

Settings

Default: 0.01

For supported blocks, the relational boundary tests check the relational operations in the block with operand values that differ by a certain tolerance. The software calculates the tolerance value using the following formula

max(absTol, relTol* max(|lhs|,|rhs|)), where:

- absTol is an absolute tolerance value that you can specify.
- relTol is the relative tolerance value that you specify.
- lhs is the left operand and rhs the right operand.
- max(x,y) returns x or y, whichever is greater.

Dependencies

To enter a value for this option, select "Include relational boundary objectives" on page 15-54.

Command-Line Information
Parameter: `DVRelativeTolerance`
Type: `double`
Value: Any valid value
Default: 0.01

See Also

- "Relational Boundary" on page 7-28
- "Model Objects That Receive Coverage" in the Simulink Verification and Validation documentation

Design Verifier Pane: Design Error Detection

In this section...

Design Error Detection Pane Overview

Specify options that control how Simulink Design Verifier detects runtime errors in the models it analyzes.

Dead logic

Specify whether to analyze your model for dead logic.

Settings

Default: Off

☑ On

Reports dead logic in your model.

☐ Off

Does not report dead logic in your model.

Dependency

Design error detection for dead logic is standalone analysis. When you enable **Dead logic**, **Active logic** is enabled and other design error detection options are disabled.

Command-Line Information

Parameter: `DVDetectDeadLogic`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Dead Logic Detection" on page 6-10

Identify active logic

Specify whether to analyze your model for active logic, in addition to dead logic.

Settings

Default: Off

☑ On

Reports active logic in your model.

☐ Off

Does not report active logic in your model.

Dependency

To enable **Identify active logic**, select **Dead logic**.

Design error detection for dead logic is standalone analysis. When you enable **Dead logic**, **Identify active logic** is enabled and other design error detection options are disabled.

Command-Line Information
Parameter: DVDetectActiveLogic
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

"Dead Logic Detection" on page 6-10

Integer overflow

Specify whether to analyze your model for integer and fixed-point data overflow errors.

Settings

Default: On

☑ On

Reports integer or fixed-point data overflow errors in your model.

☐ Off

Does not report integer or fixed-point data overflow errors in your model.

Dependency

This parameter is disabled by **Dead logic**.

Command-Line Information

Parameter: DVDetectIntegerOverflow
Type: character array
Value: 'on' | 'off'
Default: 'on'

See Also

"Static Run-Time Error Detection"

Division by zero

Specify whether to analyze your model for division-by-zero errors.

Settings

Default: On

☑ On

Reports division-by-zero errors in your model.

☐ Off

Does not report division-by-zero errors in your model.

Dependency

This parameter is disabled by **Dead logic**.

Command-Line Information

Parameter: DVDetectDivisionByZero
Type: character array
Value: 'on' | 'off'
Default: 'on'

See Also

"Static Run-Time Error Detection"

Check specified intermediate minimum and maximum values

Specify whether to check that the intermediate and output signals in your model are within the range of user-specified minimum and maximum constraints.

Settings

Default: Off

☑ On

> Checks that intermediate and output signals are within the range of user-specified minimum and maximum constraints.

☐ Off

> Does not check that intermediate and output signals are within the range of user-specified minimum and maximum constraints.

Dependency

This parameter is disabled by **Dead logic**.

Command-Line Information
Parameter: DVDesignMinMaxCheck
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

"Design Range Checks"

Out of bound array access

Specify whether to analyze your model for out of bound array access errors.

Settings

Default: Off

☑ On

> Reports out of bound array access errors in your model.

☐ Off

> Does not report out of bound array access errors in your model.

Dependency

This parameter is disabled by **Dead logic**.

Command-Line Information
Parameter: `DVDetectOutOfBounds`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

- "Detect Out of Bound Array Access Errors" on page 6-37
- "Static Run-Time Error Detection"

Design Verifier Pane: Property Proving

Property proving	
Assertion blocks:	Enable all
Proof assumptions:	Enable all
Strategy:	FindViolation
Maximum violation steps:	20

Property Proving Pane Overview

Specify options that control how Simulink Design Verifier proves properties for the models it analyzes.

See Also

- "What Is Property Proving?" on page 12-2
- "Workflow for Proving Model Properties" on page 12-4
- "Prove Properties in a Model" on page 12-5

Assertion blocks

Specify whether Assertion blocks in your model are enabled or disabled.

Settings

Default: Use local settings

Use local settings

> Enables or disables Assertion blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

Enable all

> Enables all Assertion blocks in the model regardless of the settings of their **Enable** parameters.

Disable all

> Disables all Assertion blocks in the model regardless of the settings of their **Enable** parameters.

Command-Line Information

Parameter: DVAssertions
Type: character array
Value: 'UseLocalSettings' | 'EnableAll' | 'DisableAll'
Default: 'UseLocalSettings'

See Also

- Assertion
- "Workflow for Proving Model Properties" on page 12-4
- "Prove Properties in a Model" on page 12-5

Proof assumptions

Specify whether Proof Assumption blocks in your model are enabled or disabled.

Settings

Default: Use local settings

Use local settings

> Enables or disables Proof Assumption blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

Enable all

> Enables all Proof Assumption blocks in the model regardless of the settings of their **Enable** parameters.

Disable all

> Disables all Proof Assumption blocks in the model regardless of the settings of their **Enable** parameters.

Command-Line Information

Parameter: DVProofAssumptions
Type: character array
Value: 'UseLocalSettings' | 'EnableAll' | 'DisableAll'
Default: 'UseLocalSettings'

See Also

- Proof Assumption
- "Workflow for Proving Model Properties" on page 12-4
- "Prove Properties in a Model" on page 12-5

Strategy

Specify the strategy that Simulink Design Verifier uses when proving properties.

Settings

Default: `Prove`

`Prove`

> Performs property proofs.

`FindViolation`

> Searches only for property violations within the number of simulation steps specified by the **Maximum violation steps** option.

`ProveWithViolationDetection`

> Searches first for property violations within the number of simulation steps specified by the **Maximum violation steps** option; then it attempts to prove properties for which it failed to detect a violation. This strategy is a combination of the `Prove` and `FindViolation` strategies.

Dependency

Selecting `FindViolation` or `ProveWithViolationDetection` enables the **Maximum violation steps** parameter.

Command-Line Information
Parameter: `DVProvingStrategy`
Type: character array
Value: `'Prove'` | `'FindViolation'` | `'ProveWithViolationDetection'`
Default: `'Prove'`

See Also

- "What Is Property Proving?" on page 12-2
- "Workflow for Proving Model Properties" on page 12-4
- "Prove Properties in a Model" on page 12-5

Maximum violation steps

Specify the maximum number of simulation steps over which Simulink Design Verifier searches for property violations.

Settings

Default: 20

The Simulink Design Verifier software does not search beyond the maximum number of simulation steps that you specify. Therefore, it cannot identify violations that might occur later in a simulation.

Dependency

This parameter is enabled when you set **Strategy** to `FindViolation` or `ProveWithViolationDetection`.

Command-Line Information
Parameter: `DVMaxViolationSteps`
Type: `int32`
Value: any valid value
Default: 20

See Also

- "What Is Property Proving?" on page 12-2
- "Workflow for Proving Model Properties" on page 12-4
- "Prove Properties in a Model" on page 12-5

Design Verifier Pane: Results

Data file options
- ☑ Save test data to file

Data file name: `$ModelName$_sldvdata`
- ☐ Include expected output values
- ☐ Randomize data that do not affect the outcome

Display results on model
- ☐ Display results of the analysis on the model

Harness model options
- ☐ Generate separate harness model after analysis

Harness model file name:
- ☐ Reference input model in generated harness

Simulink Test options

Test File name: `$ModelName$_test`

Test Harness name: `$ModelName$_sldvharness`

In this section...

Results Pane Overview

Specify options that control how Simulink Design Verifier handles the results that it generates.

See Also

"Results Interpretation and Use"

Save test data to file

Save the test data that the Simulink Design Verifier analysis generates to a MAT-file.

Settings

Default: On

☑ On

   Saves the test data that the analysis generates to a MAT-file.

☐ Off

   Does not save the test data that the analysis generates.

Dependency

This parameter enables **Data file name**.

Command-Line Information
Parameter: DVSaveDataFile
Type: character array
Value: 'on' | 'off'
Default: 'on'

See Also

- "Simulink Design Verifier Data Files" on page 13-9
- "Results Interpretation and Use"

Data file name

Specify a folder and file name for the MAT-file that contains the data generated during the analysis, stored in an `sldvData` structure.

Settings

Default: `$ModelName$_sldvdata`

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.
- Enter a file name for the MAT-file.
- `$ModelName$` is a token that represents the model name.

Dependency

This parameter is enabled by **Save test data to file**.

Command-Line Information
Parameter: `DVDataFileName`
Type: character array
Value: any valid path and file name
Default: `'$ModelName$_sldvdata'`

See Also

- "Simulink Design Verifier Data Files" on page 13-9
- "Results Interpretation and Use"

Include expected output values

Simulate the model using test case signals and include the output values in the Simulink Design Verifier data file.

Settings

Default: Off

☑ On

> Simulates the model using the test case signals that the analysis produces. For each test case, the software collects the simulation output values associated with Outport blocks in the top-level system and includes those values in the MAT-file that it generates.

☐ Off

> Does not simulate the model and collect output values for inclusion in the MAT-file that the analysis generates.

Tips

- The `TestCases.expectedOutput` subfield of the MAT-file contains the output values. For more information, see "Contents of `sldvData` Structure" on page 13-9.
- When **Include expected output values** is enabled, Simulink Design Verifier successively simulates the model using each test case that it generates. Enabling this option requires more time for Simulink Design Verifier to complete its analysis.

Dependency

This parameter is enabled by **Save test data to file**.

Command-Line Information
Parameter: `DVSaveExpectedOutput`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

- "Simulink Design Verifier Data Files" on page 13-9

- "Results Interpretation and Use"

Randomize data that do not affect the outcome

Specify whether to use random values instead of zeros for input signals that have no impact on test or proof objectives.

Settings

Default: Off

☑ On

> Assigns random values to test case or counterexample signals that do not affect the outcome of test or proof objectives in a model. This option can enhance traceability and improve your regression tests.

☐ Off

> Assigns zeros to test case or counterexample signals that do not affect the outcome of test or proof objectives in a model.

Tips

- This option replaces default data values with random values when the Simulink Design Verifier internal analysis engine does not specify a value. When a value does not influence the satisfaction of a test or proof objective, the generated analysis report indicates that value with a dash (–).
- Simulink Design Verifier generated analysis reports show the setting of this option.
- Enable this option to enhance traceability when simulating test cases or counterexamples. For instance, consider the following model:
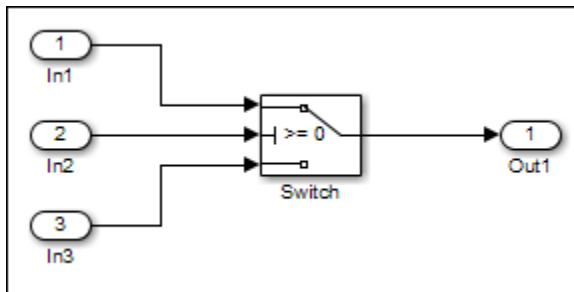
Only the signal entering the Switch block control port impacts its decision coverage. If the **Randomize data that does not affect outcome** parameter is off, Simulink Design Verifier uses zeros to represent the signals from In1 and In3. When inspecting the results from test case or counterexample simulations, it is unclear which of these signals passes through the Switch block because they have the same value. But if the **Randomize data that does not affect outcome** parameter is on, the software uses unique values to represent each of those signals. In this case, it is easier to determine which signal passes through the Switch block.

Dependency

This parameter is enabled by **Save test data to file**.

Command-Line Information

Parameter: `DVRandomizeNoEffectData`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

- "Simulink Design Verifier Data Files" on page 13-9
- "Results Interpretation and Use"

Display results of the analysis on the model

Specify whether to display analysis results by highlighting the model and providing context-sensitive details about the results.

Settings

Default: Off

☑ On

Highlight the model with the analysis results and provide context-sensitive details about the results.

☐ Off

Do not display analysis results on the model.

Command-Line Information
Parameter: DVDisplayResultsOnModel
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

- "Highlighted Results on the Model" on page 13-2
- "Results Interpretation and Use"

Generate separate harness model after analysis

Create a harness model generated by the Simulink Design Verifier analysis.

Settings

Default: Off

☑ On

   Saves the harness model that Simulink Design Verifier generates as a model file.

☐ Off

   Does not save the harness model that Simulink Design Verifier generates.

Dependency

This parameter enables **Harness model file name**.

Command-Line Information
Parameter: DVSaveHarnessModel
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

- "Simulink Design Verifier Harness Models" on page 13-16
- "Results Interpretation and Use"

Harness model file name

Specify a folder and file name for the harness model.

Settings

Default: $ModelName$_harness

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.
- Enter a file name for the harness model.
- $ModelName$ is a token that represents the model name.

Dependency

This parameter is enabled by **Generate separate harness model after analysis**.

Command-Line Information
Parameter: DVHarnessModelFileName
Type: character array
Value: any valid path and file name
Default: '$ModelName$_harness'

See Also

- "Simulink Design Verifier Harness Models" on page 13-16
- "Results Interpretation and Use"

Reference input model in generated harness

Use a Model block to reference the model to run in the harness model.

Settings

Default: Off

☑ On

   Uses a Model block to reference the model to run in the harness model.

☐ Off

   Uses a copy of the model in the harness model.

Tips

- If the Test Unit in the harness model is a subsystem, the values of the Simulink simulation optimization parameters on the Configuration Parameters dialog box can affect the coverage results.

   Note: The simulation optimization parameters are on the following Configuration Parameters dialog box panes:

   - **Optimization** pane
   - **Optimization** > **Signals and Parameters** pane
   - **Optimization** > **Stateflow** pane

- If your model contains bus objects and you select **Reference input model in generated harness**, in the **Configuration Parameters** > **Diagnostics** > **Connectivity** pane, you must set the **Mux blocks used to create bus signals** parameter to error. For more information, see "Prevent Bus and Mux Mixtures".

- On the **Design Verifier** > **Parameters** pane, if you select the **Apply parameters** parameter, Simulink Design Verifier uses a subsystem that contains a copy of the original model in the harness model, even if you select **Reference input model in generated harness**.

Command-Line Information
Parameter: DVModelReferenceHarness
Type: character array

Value: `'on'` | `'off'`
Default: `'off'`

See Also

- "Simulink Design Verifier Harness Models" on page 13-16
- "Results Interpretation and Use"

Test File Name

Name and path for test file name in Simulink Test

Settings

Default: $ModelName$_test

- Enter a file name for the test file containing Simulink Design Verifier results.
- $ModelName$ is a token that represents the model name.
- You can enter an absolute path, or a path relative to that specified by **Output folder** in the Design Verifier pane.

Dependency

This parameter is visible and enabled with a Simulink Test license.

Command-Line Information
Parameter: DVSlTestFileName
Type: character array
Value: any valid path and file name
Default: '$ModelName$_test'

See Also

- "Test Models Using Inputs Generated by Simulink Design Verifier"

Test Harness Name

Name of the test harness in Simulink Test

Settings

Default: $ModelName$_sldvharness

- Enter a valid name for the test harness built to simulate Simulink Design Verifier test cases. The test harness corresponds to the test file specified by the parameter **Test File name**.
- The $ModelName$ token represents the model name.
- Enter a valid MATLAB identifier for the test harness name.

Dependency

This parameter is visible and enabled with a Simulink Test license.

Command-Line Information
Parameter: DVSlTestHarnessName
Type: character array
Value: any valid file name
Default: '$ModelName$_sldvharness'

See Also

- "Test Models Using Inputs Generated by Simulink Design Verifier"

Design Verifier Pane: Report

Report Pane Overview

Specify options that control how Simulink Design Verifier reports its results.

See Also

- "Simulink Design Verifier Reports" on page 13-27
- "Results Interpretation and Use"

Generate report of the results

Generate and save a Simulink Design Verifier report.

Settings

Default: Off

☑ On

> Saves the HTML report that Simulink Design Verifier generates.

☐ Off

> Does not generate a Simulink Design Verifier report.

Dependencies

When this parameter is enabled, you must enable **Generate separate harness model after analysis**.

This parameter enables the following parameters:

- **Generate additional report in PDF format**
- **Report file name**
- **Include screen shots of properties**
- **Display report**

Command-Line Information
Parameter: DVSaveReport
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

- "Simulink Design Verifier Reports" on page 13-27
- "Results Interpretation and Use"

Generate additional report in PDF format

Save an additional PDF version of the Simulink Design Verifier report.

Settings

Default: Off

☑ On

Saves an additional PDF version of the Simulink Design Verifier report.

☐ Off

Does not save an additional PDF version of the Simulink Design Verifier report.

Dependency

This parameter is enabled by **Generate report of the results**.

Command-Line Information

Parameter: DVReportPDFFormat
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

- "Simulink Design Verifier Reports" on page 13-27
- "Results Interpretation and Use"

Report file name

Specify a folder and file name for the report that Simulink Design Verifier analysis generates.

Settings

Default: $ModelName$_report

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.
- Enter a file name for the report that the analysis generates.
- $ModelName$ is a token that represents the model name.

Dependency

This parameter is enabled by **Generate report of the results**.

Command-Line Information
Parameter: DVReportFileName
Type: character array
Value: any valid path and file name
Default: '$ModelName$_report'

See Also

- "Simulink Design Verifier Reports" on page 13-27
- "Results Interpretation and Use"

Include screen shots of properties

Includes screen shots of properties in the Simulink Design Verifier report. Only valid in property-proving mode.

Settings

Default: Off

☑ On

Includes screen shots of properties in the Simulink Design Verifier report. Only valid in property-proving mode.

☐ Off

Does not include screen shots of properties in the Simulink Design Verifier report.

Dependency

This parameter is enabled by **Generate report of the results**.

Command-Line Information
Parameter: DVReportIncludeGraphics
Type: character array
Value: 'on' | 'off'
Default: 'off'

See Also

- "Simulink Design Verifier Reports" on page 13-27
- "Results Interpretation and Use"

Display report

Display the report that the Simulink Design Verifier analysis generates after completing its analysis.

Settings

Default: On

☑ On

> Displays the report that the analysis generates after completing its analysis.

☐ Off

> Does not display the report that the analysis generates after completing its analysis.

Dependency

This parameter is enabled by **Generate report of the results**.

Command-Line Information
Parameter: DVDisplayReport
Type: character array
Value: 'on' | 'off'
Default: 'on'

See Also

- "Simulink Design Verifier Reports" on page 13-27
- "Results Interpretation and Use"

Model Slicer

Highlight Functional Dependencies

Large models often contain many levels of hierarchy, complicated signals, and complex mode logic. You can use Model Slicer to understand which parts of your model are significant for a particular behavior. This example shows how to use Model Slicer to explore the behavior of the `sldvSliceClimateControlExample` model. You first select an area of interest, and then highlight the related blocks in the model. In this example, you trace the dependency paths upstream of `Out1` to highlight which portions of the model affect its behavior.

Open the model and highlight the functional dependencies of a signal in the system:

1 Add the example folder to the search path.

   `addpath(fullfile(docroot,'toolbox','sldv','examples'))`

2 Open the `sldvSliceClimateControlExample` model.

   `sldvSliceClimateControlExample`

3 Select **Analysis** > **Design Verifier** > **Model Slicer...** to open the Model Slice Manager.

   When you open the Model Slice Manager, Model Slicer compiles the model. You then configure the model slice properties.

4 In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.
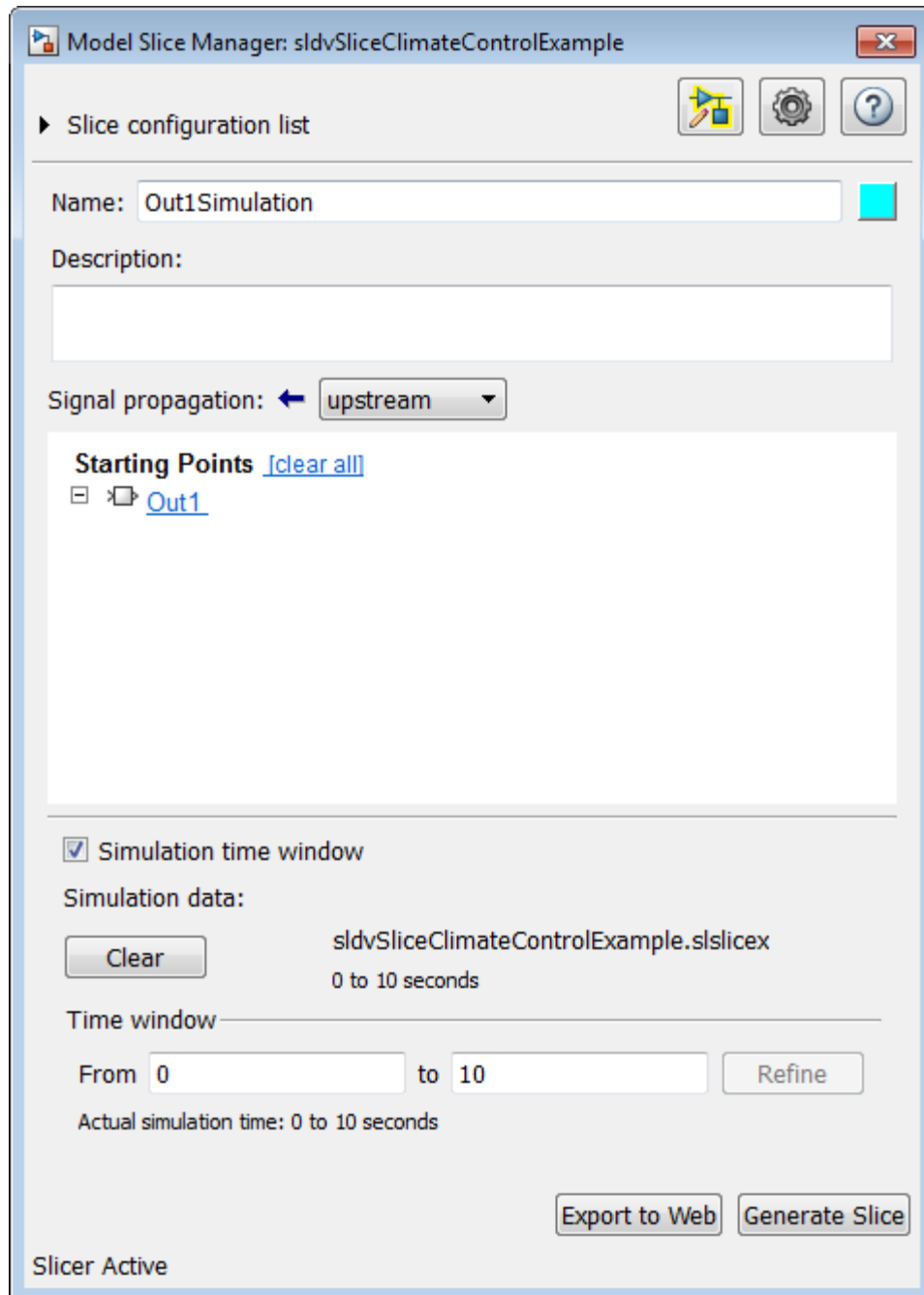
5 Set the slice properties:

   - **Name**: Out1Slice

   - **Color**: ■ (magenta)

   - **Signal Propagation**: upstream

   Model Slicer can also highlight the constructs downstream of or bidirectionally from a block in your model, depending on which direction you want to trace the signal propagation.

6 Add `Out1` as a starting point. In the model, right-click `Out1` and select **Model Slicer > Add as Starting Point**.

The Model Slicer now highlights the upstream constructs that affect `Out1`.
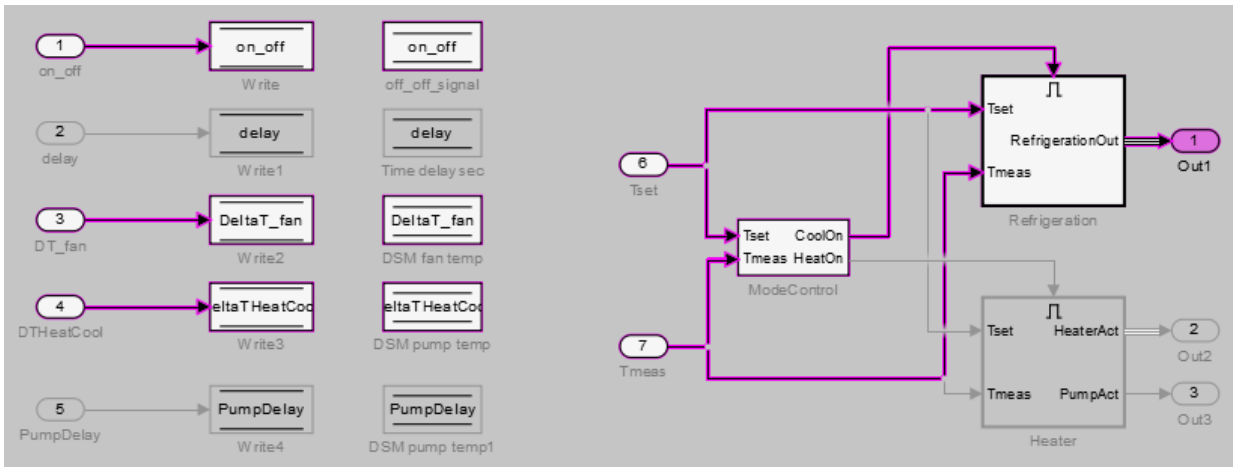
If you create two slice configurations, you can highlight the intersecting portions of their highlights. Create a new slice configuration and view the intersecting portions of the slice configuration you created above and the new slice configuration:

1   Create a new slice configuration with the following properties

   • **Name**: Out3Slice

   • **Color**: (red)

   • **Signal Propagation**: `upstream`

   • **Starting point**: `Out3`
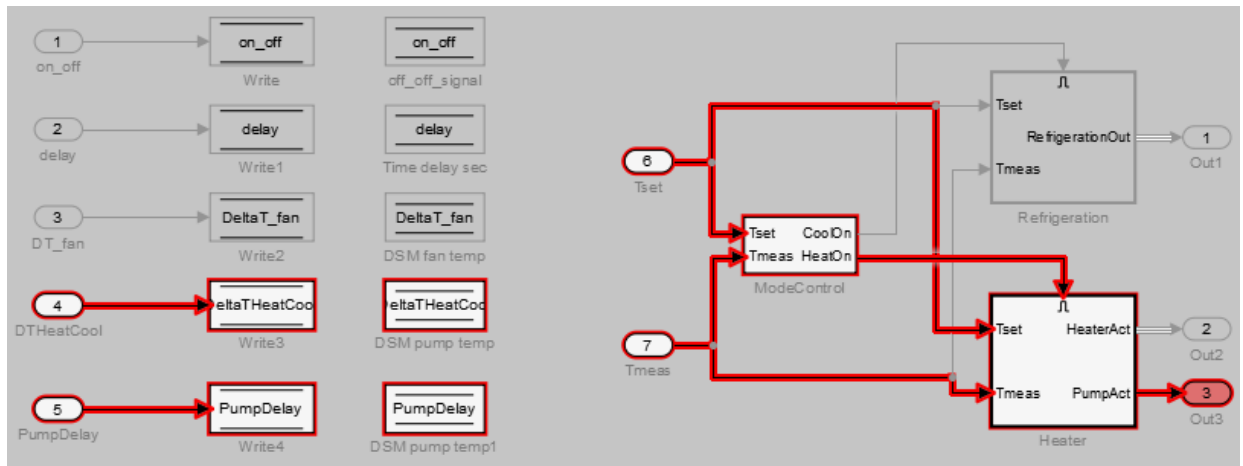
2   In the Model Slice Manager, select both the `Out1Slice` slice configuration and the `Out3Slice` slice configuration.
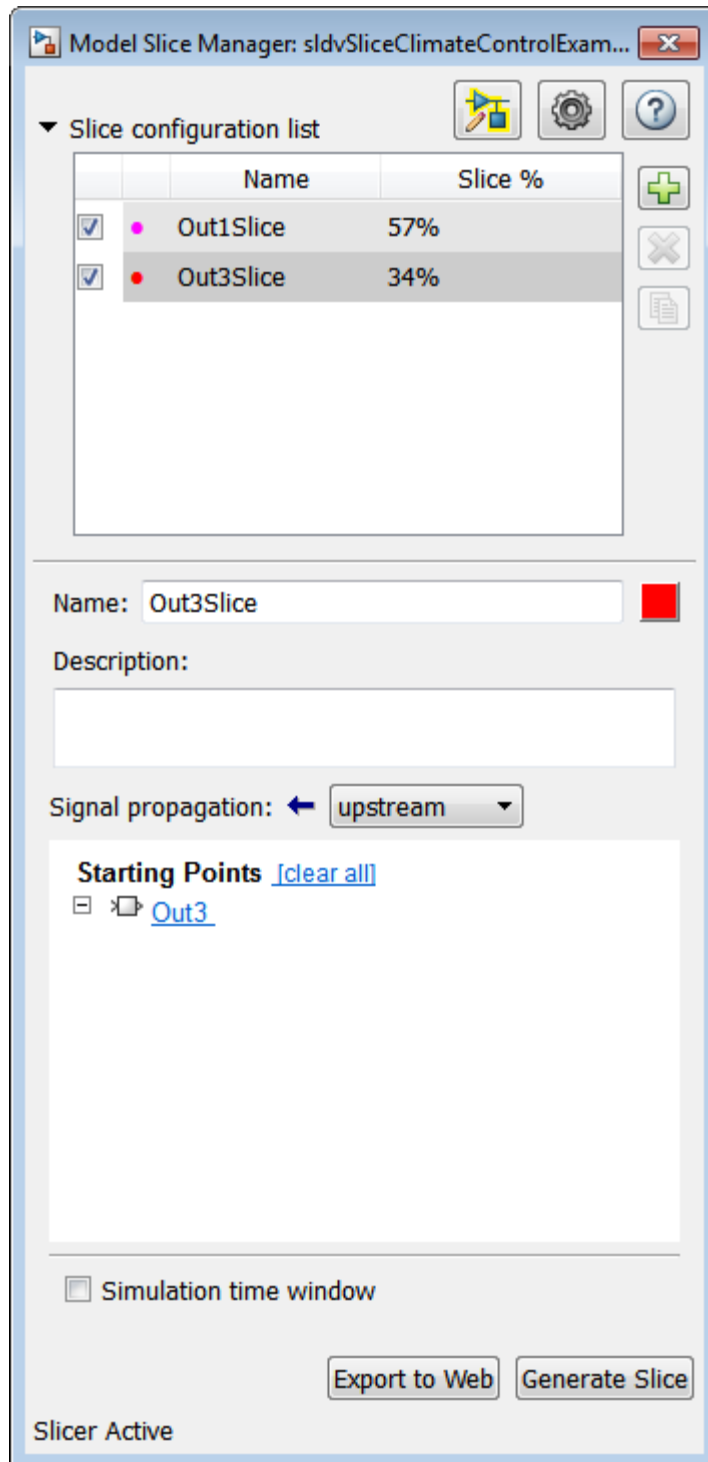
Model Slicer highlights portions of the model as follows:

- The portions of the model that are exclusively upstream of Out1 are highlighted in cyan.
- The portions of the model that are exclusively upstream of Out3 are highlighted in red.
- The portions of the model that are upstream of both Out1 and Out3 are highlighted in black.
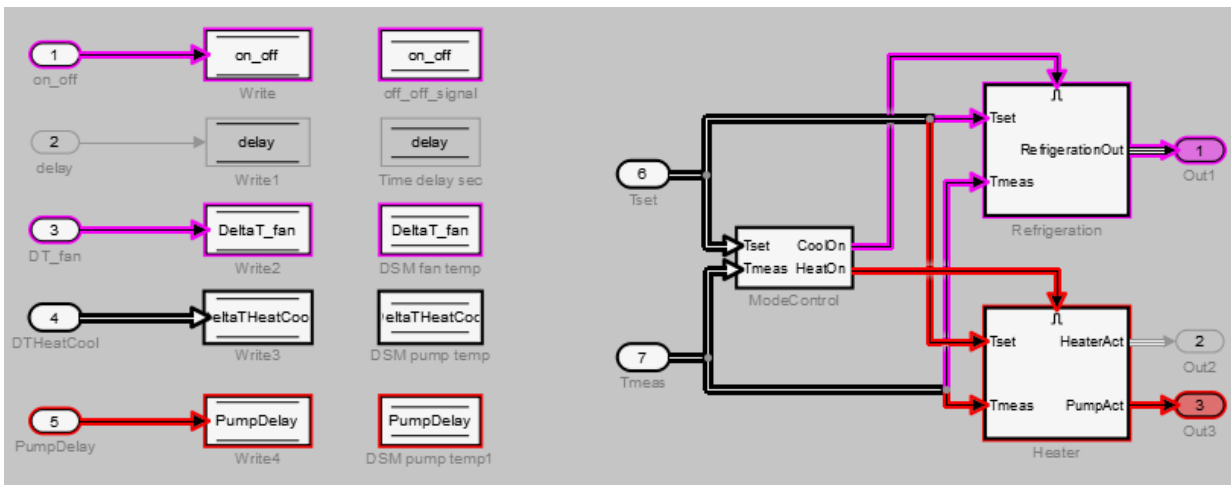
After you highlight a portion of your model, you can then refine the highlighted model to an area of interest. Or, you can create a simplified standalone model containing only the highlighted portion of your model.

More About

- "Refine Highlighted Model" on page 16-8
- "Create a Simplified Standalone Model" on page 16-21
- "Model Slicer Considerations and Limitations" on page 16-36

Refine Highlighted Model

After you highlight a model using Model Slicer, you can refine the dependency paths in the highlighted portion of the model. Using Model Slicer, you can refine a highlighted model by including only those blocks used in a portion of a simulation time window, or by excluding blocks or certain inputs of switch blocks. By refining the highlighted portion of your model, you can include only the relevant parts of your model.

In this section...
"Define a Simulation Time Window" on page 16-8
"Exclude Blocks" on page 16-13
"Exclude Inputs of a Switch Block" on page 16-17

Define a Simulation Time Window

You can refine a highlighted model to include only those blocks used in a portion of a simulation time window. Defining the simulation time window holds some switch blocks constant, and as a result removes inactive inputs.

1 Add the example folder to the search path.

```
addpath(fullfile(docroot,'toolbox','sldv','examples'))
```

2 Open the `sldvSliceClimateControlExample` model.

```
sldvSliceClimateControlExample
```

3 Select **Analysis > Design Verifier > Model Slicer** to open the Model Slice Manager.

When you open the Model Slice Manager, Model Slicer compiles the model. You then configure the model slice properties.

4 In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.

5 Set the slice properties:

   • **Name**: `Out1Simulation`

   • **Color**: (cyan)

   • **Signal propagation**: `upstream`
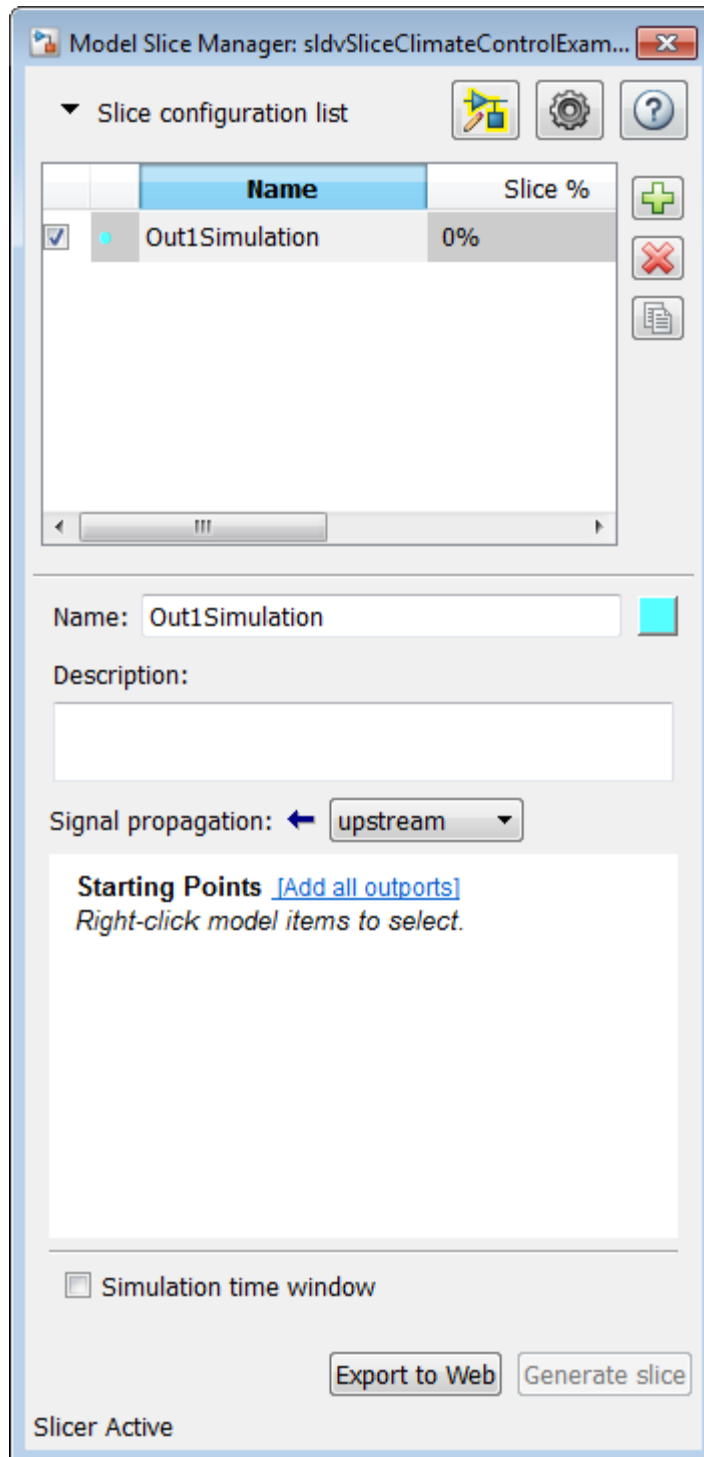
6 In the top level of the model, select the Out1 block as the slice starting point. Right-click the Out1 block and select **Model Slicer > Add as Starting Point**.

The model is highlighted.

7 In the Model Slice Manager, select **Simulation time window**.

8 To specify the stop time of the simulation time window, click the run simulation button in the Model Slice Manager.

9 Set the **Stop time** to 10.

10 Click **OK** to start the simulation.

Model Slice Manager: sldvSliceClimateControlExample

▶ Slice configuration list

Name: Out1Simulation

Description:

Signal propagation: ← upstream ▾

Starting Points [clear all]
☐ ⬛ Out1

☑ Simulation time window

Simulation data:

[Clear]   sldvSliceClimateControlExample.slslicex

0 to 10 seconds

Time window

From 0   to 10   [Refine]

Actual simulation time: 0 to 10 seconds

[Export to Web] [Generate Slice]

Slicer Active

The path is restricted to only those blocks that are active until the stop time that you entered.
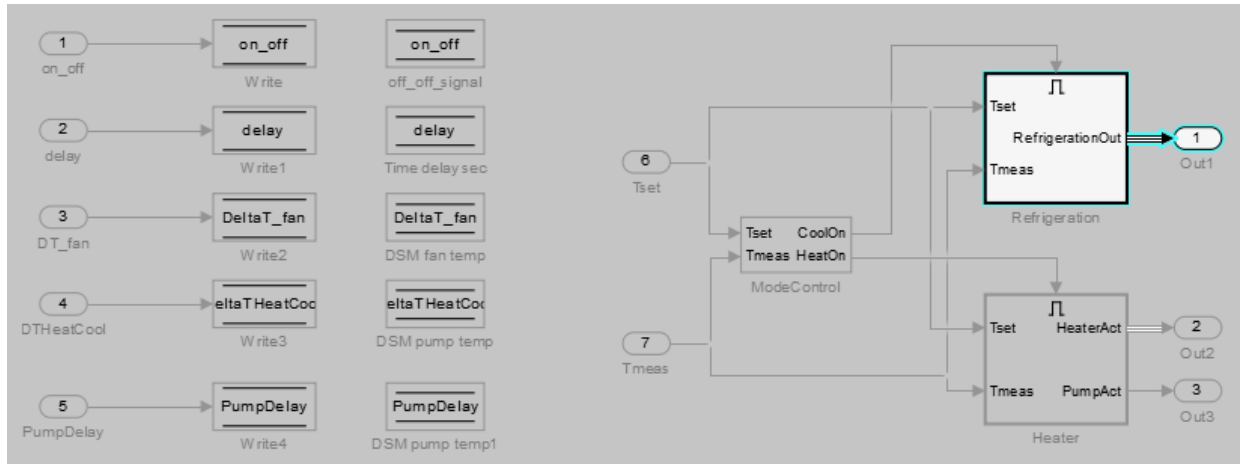
11 To see how this constraint affects the highlighted portion of the model, open the `Refrigeration` subsystem.

The highlighted portion of the model includes only the input ports of switches that are active in the simulation time window that you specified.
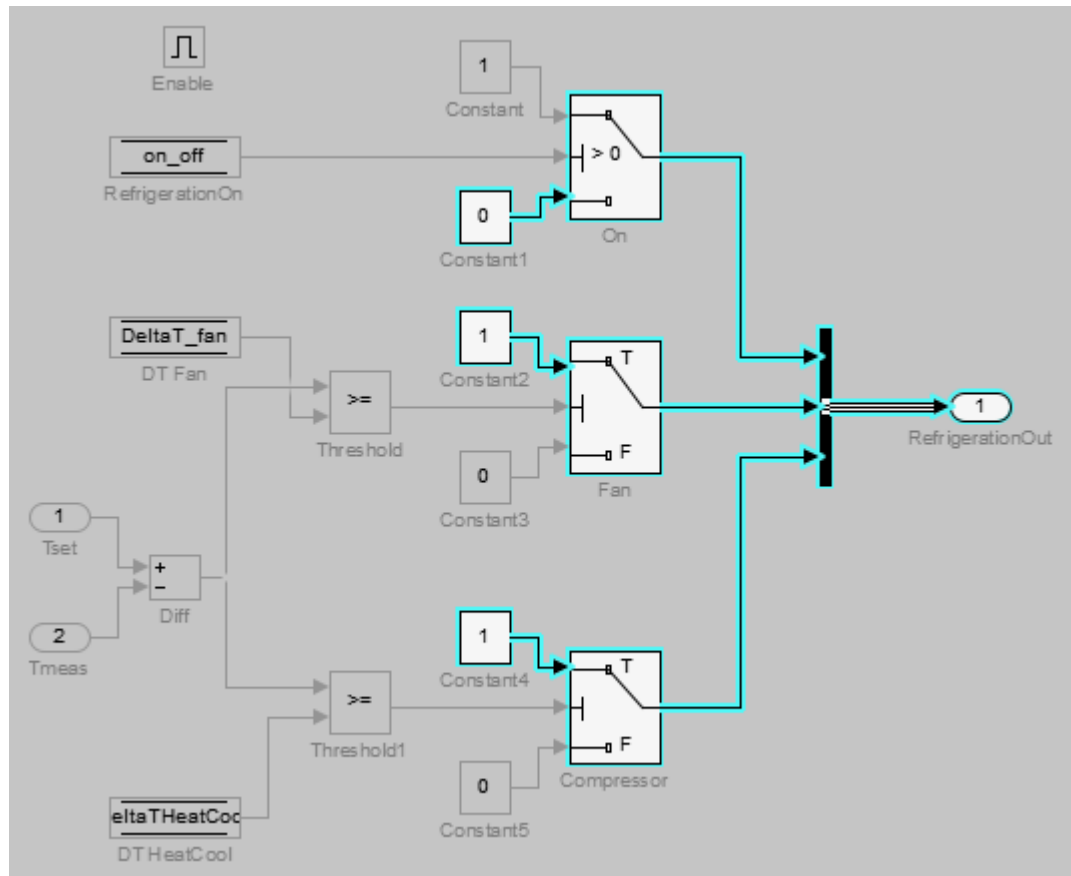
After you refine your highlighted model to include only those blocks used in a portion of a simulation time window, you can then "Create a Simplified Standalone Model" on page 16-21 incorporating the highlighted portion of your model.

Exclude Blocks

You can refine a highlighted model to exclude blocks from the analysis. Excluding a block halts the propagation of dependencies, so that signals and model items beyond the excluded block in the analysis direction are ignored.

Exclusion points are useful for viewing a simplified set of model dependencies. For example, control feedback paths create wide dependencies and extensive model highlighting. You can use an exclusion point to restrict the analysis, particularly if your model has feedback paths.

Note: Simplified standalone model creation is not supported for highlighted models with exclusion points.

---

1   In the Model Slice Manager, click the arrow to expand the **Slice configuration list**.

2   To add a new slice configuration, click the add new button .

3   Set the slice properties:

   - **Name**: Out1Excluded

   - **Color**: (red)

   - **Signal Propagation**: upstream

4   In the top level of the model, select the Out1 block as the slice starting point. Right-click the Out1 block and select **Model Slicer > Add as Starting Point**.

Model Slice Manager: sldvSliceClimateControlExample

▼ Slice configuration list

		Name	Slice %
☐	●	Out1Simulation	13%
☑	●	Out1Excluded	57%

Name: Out1Excluded

Description:

Signal propagation: ← upstream ▼

Starting Points [clear all]
☐ ⊐ Out1

☐ Simulation time window

Export to Web   Generate Slice

Slicer Active

The model is highlighted.

5 To open the subsystem, double-click `Refrigeration`.

6 Right-click the `Fan` switch block, and then select **Model Slicer > Add as Exclusion Point**.

The blocks that are exclusively upstream of the `Fan` switch block are no longer highlighted. The `DT Fan` Data Store Read block is no longer highlighted.
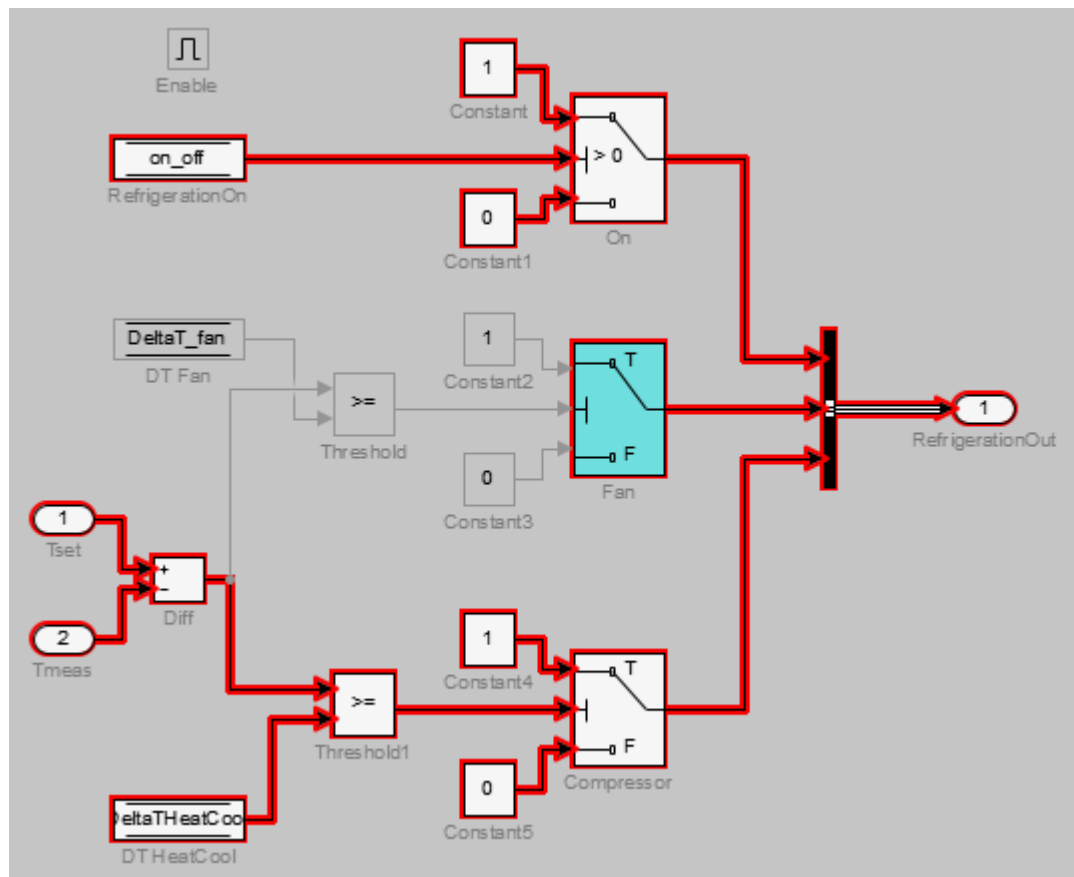
7 To see how this constraint affects the highlighted portion of the model, view the parent system.

The `DSM fan temp` Data Store Memory block and the `Write2` Data Store Write block are no longer highlighted, because the `DT Fan` Data Store Read in the `Refrigeration` subsystem no longer accesses them.
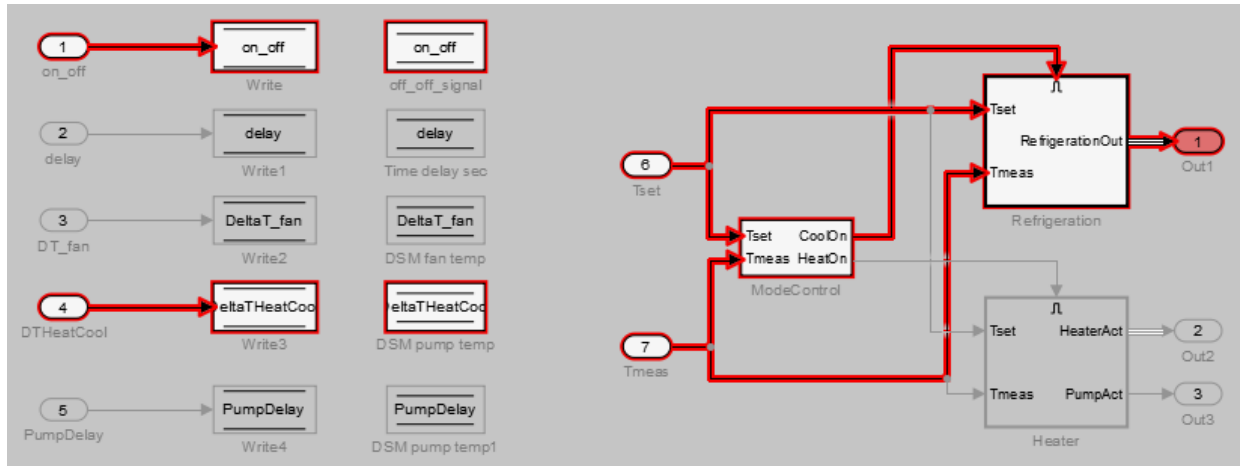
Exclude Inputs of a Switch Block

For complex signal routing, you can constrain the dependency analysis paths to a subset of the available paths through switch blocks. Constraints appear in the Model Slice Manager.

Note: Simplified standalone model creation is not supported for highlighted models with constrained switch blocks.

---

1   Double-click `Refrigeration` to open the subsystem.

2   Constrain the `On` switch block:

   • Right-click the switch block and select **Model Slicer > Add Constraint**.

   • In the Constraints dialog box, select **Port 3**.

   • Click **OK**.
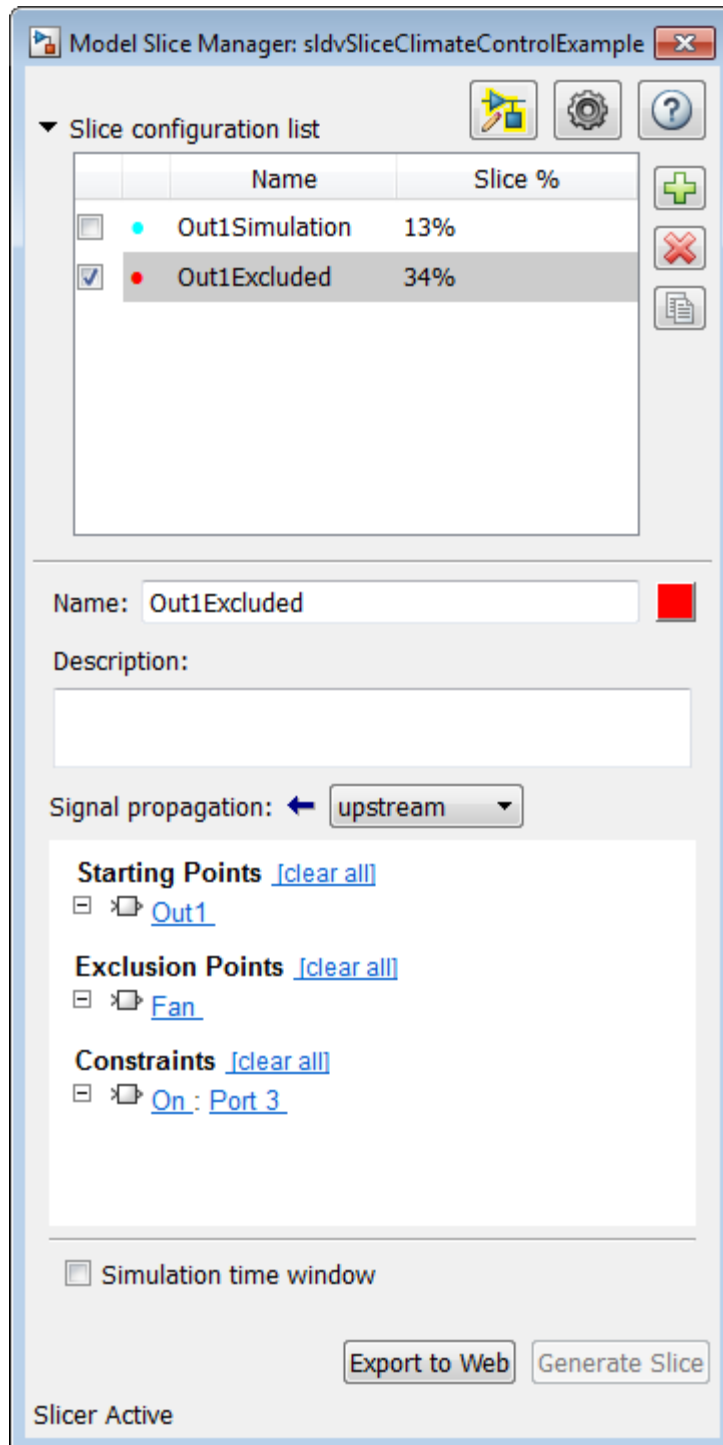
The path is restricted to the `Constant1` port on the switch. The blocks that are upstream of **Port 1** and **Port 2** of the constrained switch are no longer highlighted. Only the blocks upstream of **Port 3** are highlighted.
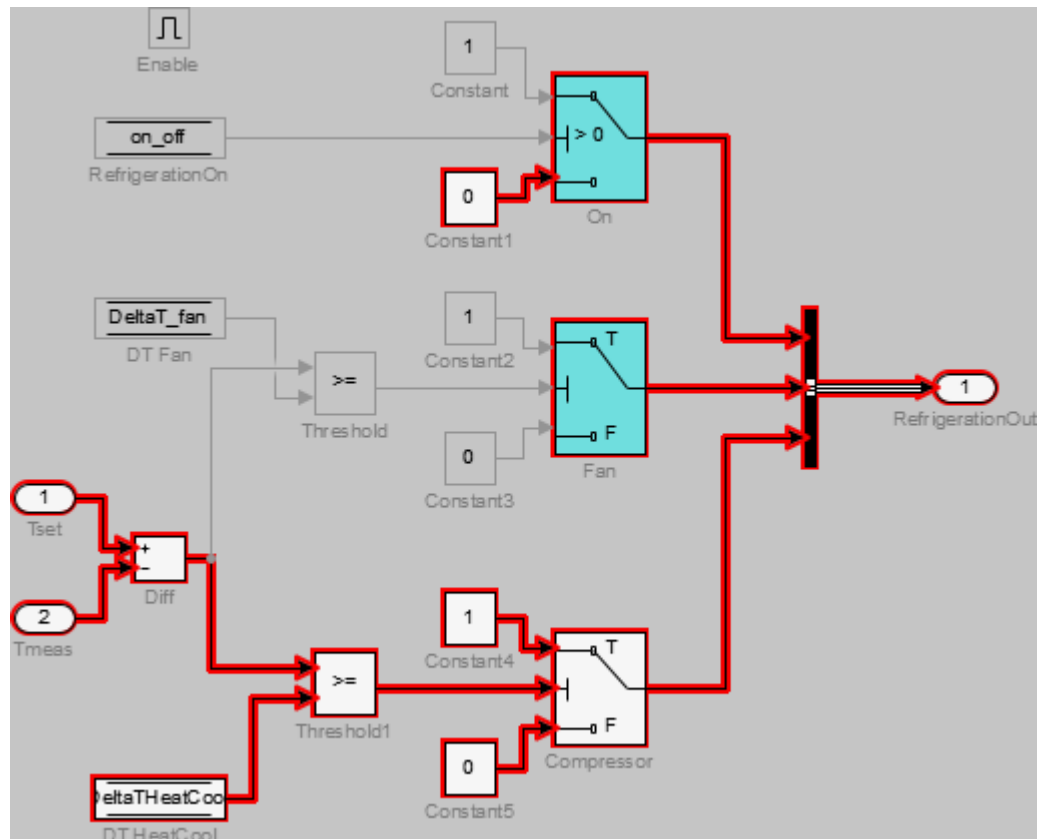
3 To see how this constraint affects the highlighted portion of the model, view the parent system.
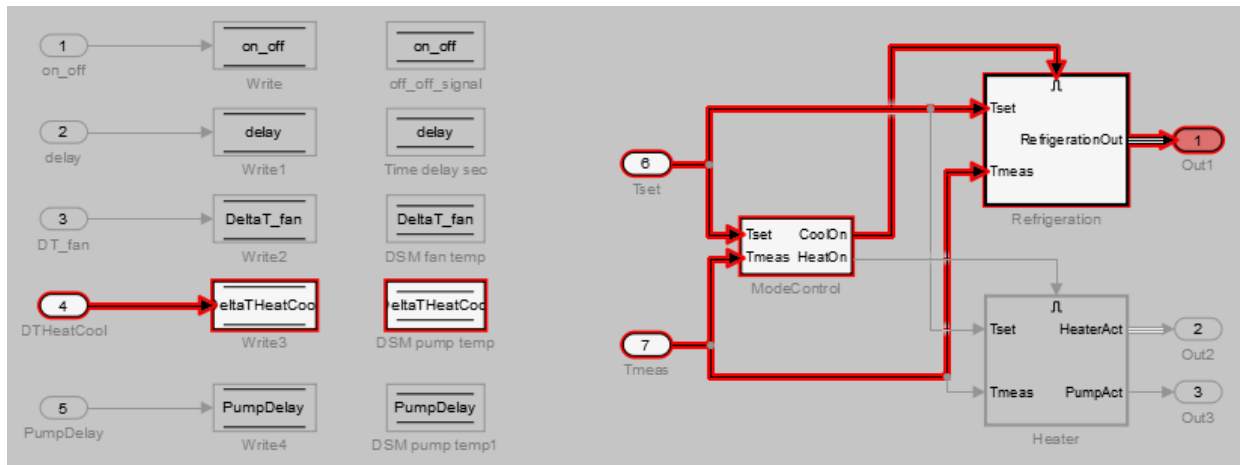
More About

- "Create a Simplified Standalone Model" on page 16-21
- "Model Slicer Considerations and Limitations" on page 16-36

Create a Simplified Standalone Model

You can simplify simulation, debugging, and formal analysis of large and complex models by focusing on areas of interest in your model. After highlighting a portion of your model using Model Slicer, you can generate a simplified standalone model incorporating the highlighted portion of your original model. Apply changes to the simplified standalone model based on simulation, debugging, and formal analysis, and then apply these changes back to the original model.

Note: Simplified standalone model creation is not supported for highlighted models with exclusion points or constrained switch blocks. If you want to view the effects of exclusion points or constrained switch blocks on a simplified standalone model, first create the simplified standalone model, and then add exclusion points or constrained switch blocks.

---

1 Highlight a portion of your model using Model Slicer.

   See "Highlight Functional Dependencies" on page 16-2 and "Refine Highlighted Model" on page 16-8.

2 In the Model Slice Manager, click **Generate slice**.

3 In the **Select File to Write** dialog box, select the save location and enter a model name.

   The simplified standalone model contains the highlighted model items.

4 To remove highlighting from the model, close the Model Slice Manager.

When generating a simplified standalone model from a model highlight, you might need to refine the highlighted model before the simplified standalone model can compile. See the "Model Slicer Considerations and Limitations" on page 16-36 for compilation considerations.

More About

· "Basic Workflow for Simulink Design Verifier" on page 1-31

Simplify a Standalone Model by Inlining Content

You can reduce file dependencies by inlining model content when you generate the sliced model. Inlining brings functional content into the sliced model and can eliminate model references, library links, and variant structures that are often not needed for model refinement or debugging.

If you want to disable inlining for certain block types, open the Model Slice Manager and click the options button [image]. Select only the block types for which you want to inline content. For information on block-specific inlining behavior, see "Inline Content Options" on page 16-34.

This example demonstrates inlining content of a model referenced by a Model block.

1 Add the path to the example and open the model

```
addpath(fullfile(docroot,'toolbox','sldv','examples'))
open_system('sldvSliceEngineDynamicsExample')
```

2 From the menu, select **Analysis > Design Verifier > Model Slicer** to open the Model Slice Manager.

3 In the model, right-click the MAP outport and select **Model Slicer > Add as Starting Point**. The path is highlighted through the Model block.
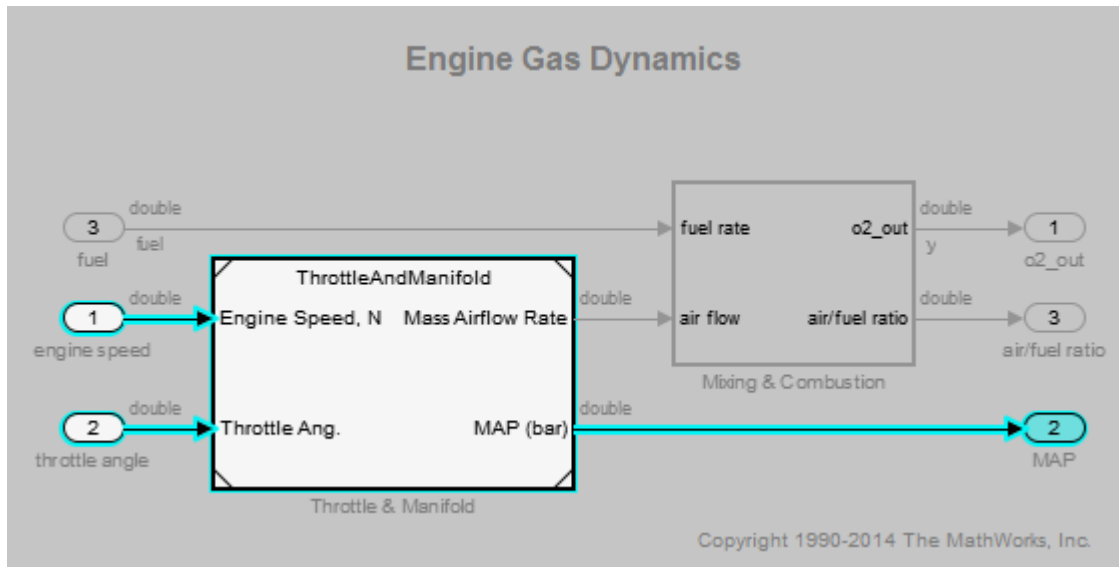
4 Create a sliced model from the highlight. In the Model Slice Manager, click the **Generate slice** button.

5 Enter a file name for the sliced model.

6 The sliced model contains the highlighted model content. The model reference is removed.

Engine Gas Dynamics
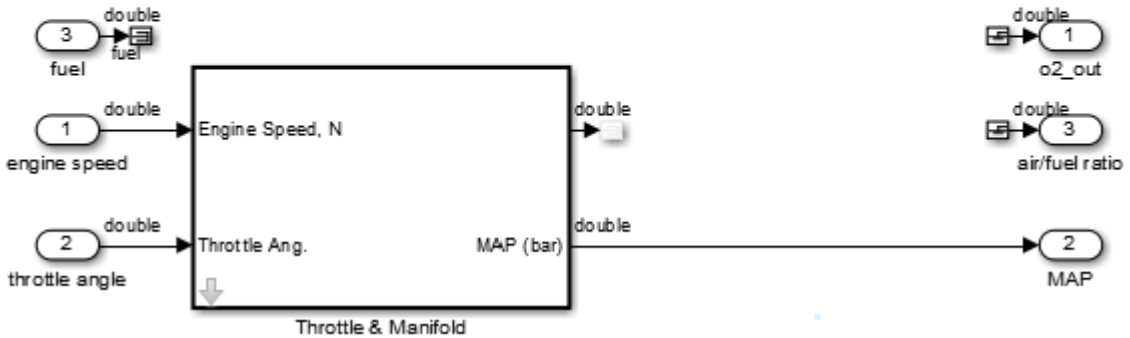
7   Click the arrow to look under the mask of the ThrottleAndManifold subsystem. The content from the referenced model is inlined into the model in the masked subsystem.
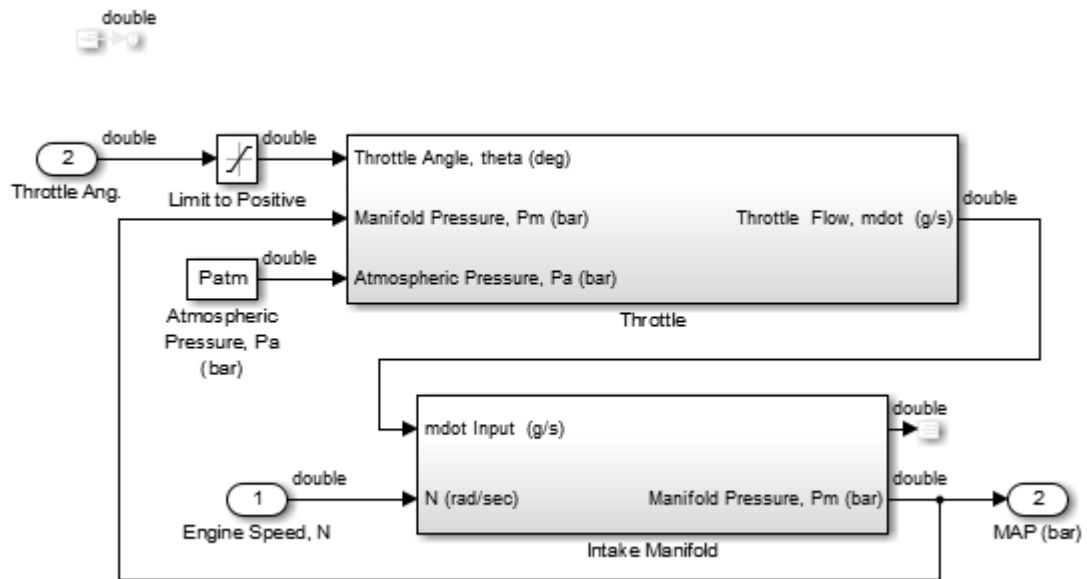
Workflow for Dependency Analysis

Model analysis includes determining dependencies of blocks, signals, and model components. For example, to view blocks affecting a subsystem output, or trace a signal path through multiple switches and logic. Determining dependencies can be a lengthy process, particularly for large or complex models. Use Model Slicer as a simple way to understand functional dependencies in large or complex models. You can also use Model Slicer to create simplified standalone models that are easier to understand and analyze, yet retain their original context.

Dependency Analysis Workflow

The dependency analysis workflow identifies the area of interest in your model, generates a sliced model, revises the sliced model, and incorporates those revisions in the main model.
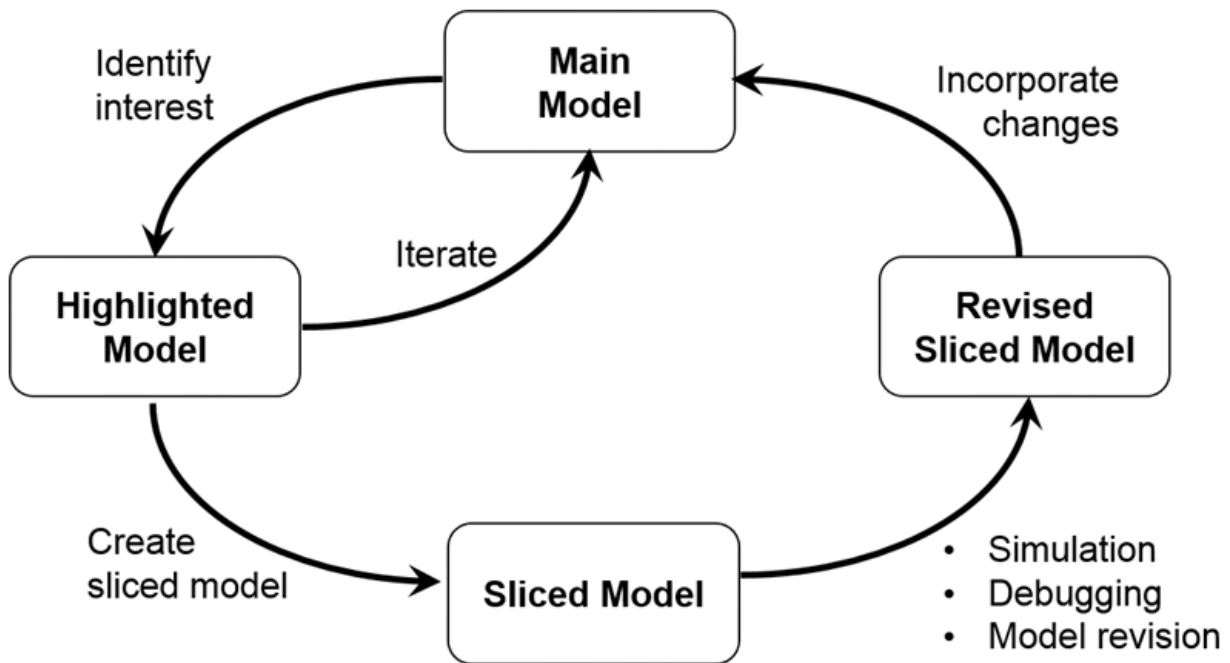
Dependency Analysis Objectives

To identify the area of interest in your model, determine objectives such as:

- What item or items are you analyzing? Analysis begins with at least one starting point.

- In what direction does the analysis propagate? The dependency analysis propagates upstream, downstream, or bidirectionally from the starting points.

- What model items or paths do you want to exclude from analysis?

- What paths do you want to constrain? If your model has switches, you can constrain the switch positions for analysis.

- Is your model a closed-loop system? If so, the highlighted portion of the model can include model dependencies from the feedback loop. Consider excluding blocks from the feedback loop to refine the highlighted portion of the model.

- Do you want to analyze static dependencies, or include simulation effects? Static analysis considers model dependencies for possible simulation paths. Simulation-based analysis highlights only paths active during simulation.

Related Examples

Configure Model Highlight and Sliced Models

Model Slice Manager

Set the properties of your model highlight and standalone sliced model using the Model Slice Manager.
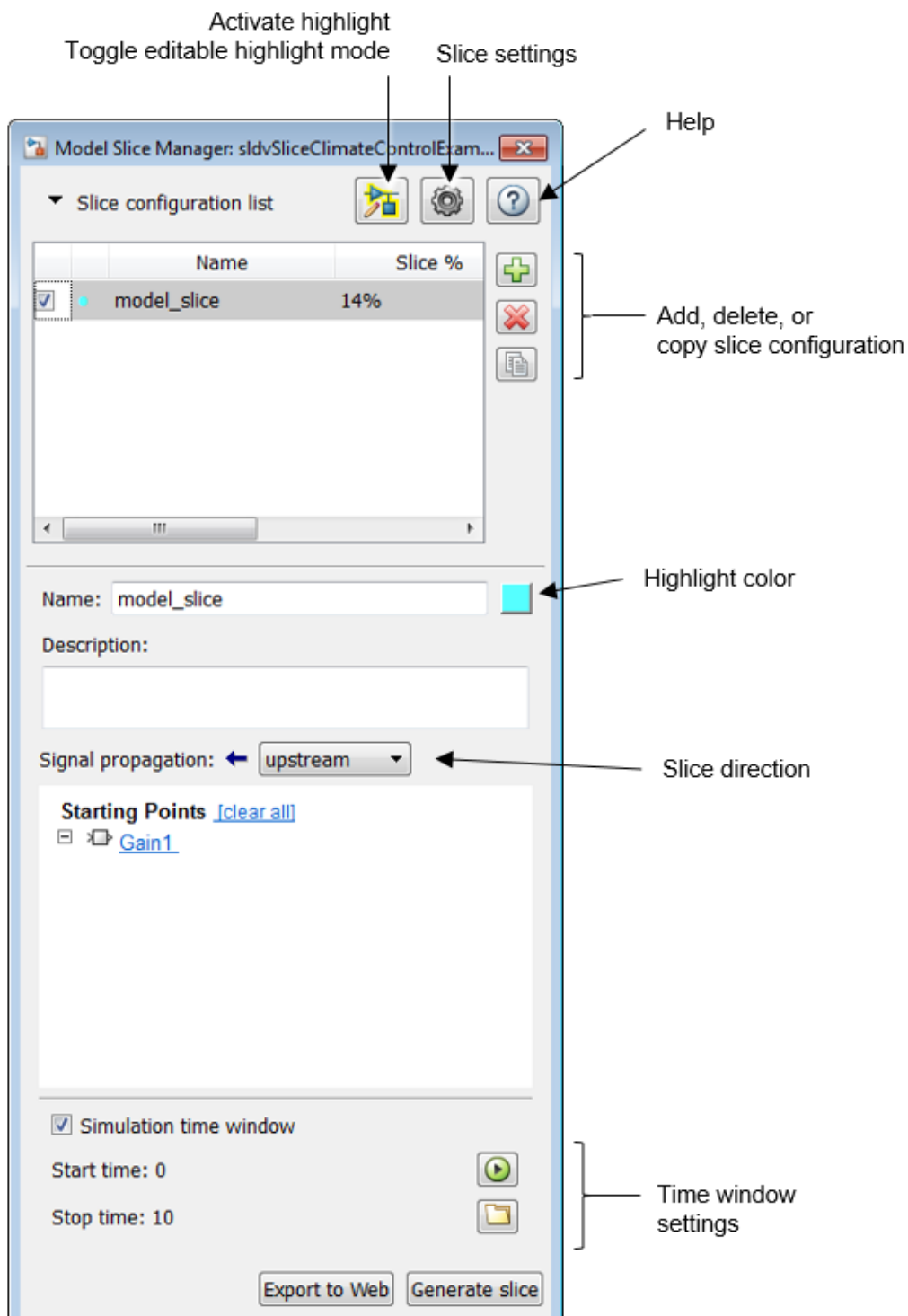
Click the toggle mode button to switch between model edit mode and model highlight mode.

If automatic highlighting is disabled in the slice settings, refresh the model highlight using the refresh button . Refresh the highlight after changing the slice configuration.

Model Slicer Options

You can customize the slice behavior using the options dialog box, which is accessed with the options button .
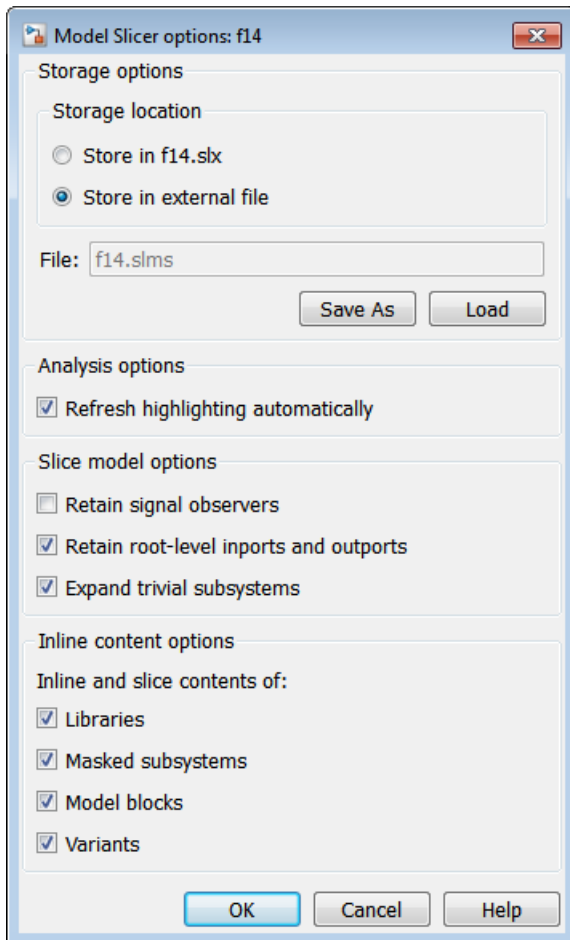
Storage Options

Changes you make to a model slice configuration are saved automatically. You can store the slice configuration in the model SLX file, or in an external SLMS file. Saving the configuration externally can be useful if your SLX file is restricted by a change control system.

To set the storage location, click the options 　　 button in the Model Slice Manager and set the location in the **Storage options** pane.

Settings

Store in <model_name>.slx

Saves the model slice configuration in your model's SLX file

Store in external file

Saves the model slice configuration in a separate SLMS file you specify by clicking the **Save As** button. The model slice configuration filename is shown in **File**.

Refresh Highlighting Automatically

Enables automatic refresh of a model highlight after changing the slice configuration.

Settings

on (default)

Model highlighting refreshes automatically.

off

Model highlighting must be refreshed manually. Click the refresh button 　　 in the Model Slice Manager to refresh the highlight.

Sliced Model Options

You can control what items are retained when you create a sliced model from a model highlight using the options in the **Sliced model options** pane.

Option	On (selected)	Off (cleared)
Retain signal observers	Signal observers, such as scopes, displays, and test condition blocks, are retained in the sliced model.	Signal observers are not retained in the sliced model (default).
Retain root-level inports and outports	Root-level ports are retained in the sliced model (default).	Root-level ports are not retained in the sliced model.

Option	On (selected)	Off (cleared)
Expand trivial subsystems	Trivial subsystems are expanded in the sliced model and the subsystem boundary is removed (default).	Trivial subsystems are not expanded in the sliced model and the subsystem boundary is retained. See "Trivial Subsystems" on page 16-34.

Trivial Subsystems

If a subsystem has all of these characteristics, Model Slicer considers the subsystem trivial:

- If the subsystem is virtual, it contains three or fewer nonvirtual blocks.
- If the subsystem is atomic, it contains one or fewer nonvirtual blocks.
- The subsystem has two or fewer inports.
- The subsystem has two or fewer outports.
- The active inport or outport blocks of the subsystem have default block parameters.
- The system does not contain Goto Tag Visibility blocks.
- In the Block Properties dialog box, the subsystem **Priority** is empty.
- The data type override parameter (if applicable) is set to use local settings.

Note: If you generate a sliced model which does not remove contents of a particular subsystem, the subsystem remains intact in the sliced model.

Inline Content Options

When you create a sliced model from a highlight, model items can be inlined into the sliced model. The **Inline content options** pane controls which model components are inlined in generating a sliced model.

Model Component	Inlining on (selected)	Inlining off (cleared)
Libraries	Model items inside sliced libraries are inlined in the sliced model and the library link is removed. (default)	Model items inside sliced libraries are not inlined in the sliced model and library link remains in place.

Model Component	Inlining on (selected)	Inlining off (cleared)
Masked subsystems	Model items inside sliced masked subsystems are inlined in the sliced model. (default) The mask is retained in the sliced model.	Model items inside sliced masked subsystems are not inlined in the sliced model and the mask is retained.
Model blocks	Model items are inlined to the sliced model from the model referenced by the Model block. The Model block is removed. (default) **Note:** Model Slicer cannot inline model blocks that are not in `Normal` mode.	Model items are not inlined to the sliced model from the model referenced by the Model block. The Model block is retained.
Variants	Model items are inlined to the sliced model from the active variant. Variants are removed. (default)	Model items are not inlined to the sliced model from the variant. The variant is retained.

Related Examples

- "Highlight Functional Dependencies" on page 16-2
- "Refine Highlighted Model" on page 16-8
- "Simplify a Standalone Model by Inlining Content" on page 16-22

Model Slicer Considerations and Limitations

Consider these behaviors and limitations when working with Model Slicer.

Model Highlighting and Model Editing

You cannot edit the model when a slice highlight is active. You can switch into model edit mode and preserve the highlights. When you switch back to slice mode, the slice configuration is recomputed and the highlight updates.

Standalone Sliced Model Generation

Sliced model generation requires one or more starting points for the model highlight. Sliced model generation is not supported for the following:

· Forward-propagating (including bidirectional) dependencies

· Constraints

· Exclusion points

Sliced model generation requires a writable present working folder in MATLAB.

Sliced Model Considerations

When you generate a sliced model from a model highlight, model simplification can change simulation behavior, or prevent the sliced model from compiling. For example:

- Model simplification can change the sorted execution order in a sliced model compared to the original model. The changed sorted execution order can affect the sliced model simulation behavior.

- If you generate a sliced model containing a bus, but not the source signal of that bus, the sliced model can contain unresolved bus elements.

- If you generate a sliced model that inlines a subset of the contents of a masked block, ensure that the subsystem contents resolve to the mask parameters. If the contents and mask do not resolve, it is possible that the sliced model does not compile.

- If the source model uses a bus signal, ensure that the sliced model signals are initialized correctly. Before you create the sliced model, consider including an explicit copy of the bus signal in the source model. For example, you can include a Signal Conversion block with the **Output** option set to `Signal Copy`.

- For solver step sizes set to `auto`, Simulink calculates the maximum time step in part based on the blocks in the model. If the sliced model removes blocks that affect the time step determination, the time step of the sliced model can differ from the source model. The time step difference can cause simulation differences. Consider setting step sizes explicitly to the same values calculated in the source model.

Port Attribute Considerations

Blocks that Model Slicer removes during model simplification can be used to determine compiled attributes, such as inherited sample times, signal dimensions, and data types. Model Slicer can change sliced model port attributes during model simplification to resolve underspecified model port attributes. If Model Slicer cannot resolve these inconsistencies, you can resolve some model port attribute inconsistencies by:

- Explicitly specifying attributes in the source model instead of relying on propagation rules.

- Including in the sliced model the blocks that are responsible for the attribute propagation in your source model. Before you slice the model, add these blocks as additional starting points in the source model highlight.

- Not inlining the model blocks that are responsible for model port attributes into the sliced model. For more information on model items that you can inline into the sliced model, see "Inline Content Options" on page 16-34.

Because of the way Simulink handles model references, you cannot simultaneously compile two models that both contain a model reference to the same model. Model Slicer enters the **Slicer Locked (for attribute checking)** mode when you generate a sliced model if the following conditions are true:

- The parent model contains a referenced model
- The highlighted portion of the parent model contains the referenced model
- The referenced model is not inlined in the sliced model due to one of the following

    - You choose not to inline model blocks in the **Inline content options** pane of the **Model Slicer options**
    - Model Slicer cannot inline the referenced model. For more information on model items that Model Slicer cannot inline, see "Inline Content Options" on page 16-34

To continue refining the highlighted portion of the parent model, you must first activate

slice highlight mode .

Simulation Time Window Considerations

Depending on the step size of your model and the values you enter for the start time and stop time of the simulation time window, Model Slicer may alter the actual simulation start time and stop time used.

- If you enter a stop or start time that falls between time steps for your model's solver, Model Slicer will instead use a stop or start time that matches the time step previous to the value you entered. For more information on step sizes in Simulink, see "Solvers".
- The stop time for the simulation time window cannot be greater than the total simulation time.

Simulation-based Sliced Model Simplifications

When you slice a model using a simulation time window, some blocks in the source model, such as switch blocks, logical operator blocks, and others, can be replaced when creating the simplified standalone model. For example, a switch block that always passes one input is removed, and the active input is directly connected to the output destination. The unused input signal is also removed from the standalone model.

The following table describes the blocks Model Slicer can replace during model simplification:

Block in Source Model	Simplification
Switch Multiport Switch	If only one input port is active, the switch is replaced by a signal connecting the active input to the block output.
Enabled Subsystem or Model	If the subsystem or model is always enabled, remove control input and convert to standard subsystem or model. If the subsystem is never enabled, replace subsystem with a constant value defined by the initial condition.
Triggered Subsystem or Model	If the subsystem or model is always triggered, remove trigger input and convert to standard subsystem or model. If the subsystem is never triggered, replace subsystem with a constant value defined by the initial condition.
Enabled and Triggered Subsystem or Model	If the subsystem is alwaysexecuted, convert to standard subsystem or model If the subsystem is never executed, replace subsystem with a constant value defined by the initial condition.
Merge	If only one input port is active, the merge is replaced by a signal connecting the active input to the block output.
If If Action	If only one action subsystem is active, convert to standard subsystem or mode and remove the If block.
Switch Case Switch Case Action	If only one action subsystem is active, convert to standard subsystem or mode and remove the Switch Case block.

Block in Source Model	Simplification
Logical operator	Replace with constant when the block always outputs true or always outputs false. Replace input signal with constant if input signal is always true or always false.

Starting Points Not Supported

Model Slicer does not support these model items as starting points.

- Virtual blocks
- Output signals from virtual blocks that are not subsystems

Model Slicer Support Limitations for Simulink Software Features

Model Slicer does not support these features.

- Arrays of buses
- Analysis of Simulink Test test harnesses
- Models that contain Simscape™ physical modeling blocks
- Models that contain algebraic loops
- Loading initial states from the source model for sliced model generation, such as data import/export entries. Define initial states explicitly for the sliced model in the sliced model configuration parameters.

Model Slicer Support Limitations for Simulink Blocks

The table lists Model Slicer support limitations for Simulink Blocks:

Block	Limitation
For Each Subsystem block	The simulation impact is ignored for blocks in a For Each subsystem. Therefore, applying a simulation time window returns the same dependency analysis result as a dependency analysis using no simulation time window.
Function Caller block	Model Slicer does not support Function Caller blocks.

Block	Limitation
MATLAB Function block	Model Slicer assumes that any output depends on all inputs in the upstream direction, and any input affects all outputs in the downstream direction.
Merge block	If you generate a slice using a simulation time window, Merge blocks are removed in the standalone model if only a single path is exercised.
Model block	Model Slicer does not support multiple Model blocks with **Simulation mode** set to Normal. Model Slicer does not resolve data dependencies generated by global data store memory in Model blocks with **Simulation mode** set to Accelerator. Model Slicer does not support function-call root-level Inport blocks. For more information, see Export-Function Models. Model Slicer does not analyze the contents within a reference to a "Protected Model". When you slice a model that contains a protected model reference, Model Slicer includes the entire model reference in the sliced model.
Resettable Subsystem block	Model Slicer does not support Resettable Subsystem blocks.
S-function block	Model Slicer assumes that any output depends on all inputs in the upstream direction, and any input affects all outputs in the downstream direction. Model Slicer does not determine dependencies that result from an S-function block accessing model information dependent on a simulation time window.
State Read block	Model Slicer does not support State Read blocks.
State Write block	Model Slicer does not support State Write blocks.

Model Slicer Support Limitations for Stateflow

- When you highlight models containing a Stateflow chart or state transition table, Model Slicer assumes that any output from the Chart block or State Transition Table block depends on all inputs to the Chart block or State Transition Table block.

- When you slice a model with a Stateflow chart or state transition table, Model Slicer does not simplify the chart or table. The chart or table is included in its entirety in the sliced model.

- If you do not "Define a Simulation Time Window" on page 16-8 when you highlight functional dependencies in a Stateflow chart or state transition table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.

- When you highlight functional dependencies in a Stateflow chart or state transition table for a defined simulation time window, Model Slicer does not highlight only the states and transitions that affect the selected starting point. Model Slicer instead all elements that are active in the time window that you specify.

- Model Slicer does not determine dependencies between Stateflow graphical functions and function calls in other Stateflow charts.

- Graphical functions and their contents that were not active during the selected time window can potentially remain highlighted in some cases.

- Entry into states that are preempted due to events can potentially remain highlighted in some cases. For example, after a parent state is entered, an event action can exit the state and preempt entry into the child state. In such a case, Model Slicer highlights the entry into the child state.

- Model Slicer does not support:

  - Embedded MATLAB Function blocks
  - Simulink functions
  - Truth Table blocks
  - Machine-parented data or events in Stateflow.

  .

See Also

"Algebraic Loops" | "Solver Pane"

Using Model Slicer with Stateflow

You can use Model Slicer highlighting to visually verify the logic in your Stateflow charts or tables. After you "Define a Simulation Time Window" on page 16-8, Model Slicer highlights Stateflow elements that are active within the selected time window.

Note: If you do not "Define a Simulation Time Window" on page 16-8 when you highlight functional dependencies in a Stateflow chart or table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.

Model Slicer Highlighting Behavior for Stateflow Elements

Model Slicer highlights a Stateflow element if it was executed in the specified time window. Some examples include:

- A chart, if it is activated in the specified a time window.
- A state, if its entry, exit, or during actions are executed in the specified a time window.
- A parent state, if its child state is highlighted in the specified a time window.
- A transition, if it is taken in the specified time window, such as inner, outer, and default. If the conditions of a transition are evaluated, but the transition is not taken, Model Slicer does not highlight the transition.

Using Model Slicer with Stateflow State Transition Tables

Model Slicer does not directly highlight the contents of Stateflow state transition tables. To view highlighted functional dependencies in a state transition table, you must view the auto-generated diagram for the state transition table. For instructions on how to view the auto-generated diagram for the state transition table, see "Generate Diagrams from State Transition Tables".

Support Limitations for Using Model Slicer with Stateflow

For support limitations when you use Model Slicer with Stateflow, see "Model Slicer Support Limitations for Stateflow" on page 16-42.

More About

- "Highlight Functional Dependencies" on page 16-2
- "Refine Highlighted Model" on page 16-8
- "Chart Simulation Semantics"

Isolating Dependencies of an Actuator Subsystem

This example demonstrates highlighting model items that a subsystem depends on. It also demonstrates generating a standalone model slice from the model highlight.

In this section...
"Choose Starting Points and Direction" on page 16-45
"View Precedents and Generate Model Slice" on page 16-47

Choose Starting Points and Direction

1   Open the f14 example model.

   f14

2   Select **Analysis > Design Verifier > Model Slicer** to open the Model Slice Manager.
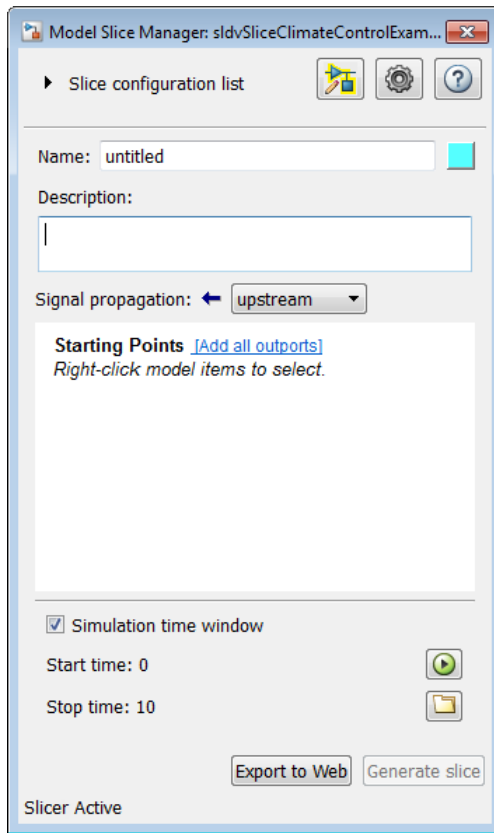
3 In the Model Slice Manager, click the arrow to expand the **Slice configuration list**
   list. Set the slice properties:

   - **Name**: `Actuator_slice`
   - To the right of **Name**, click the colored square to set the highlight color. Choose
     magenta ▮ from the palette.
   - **Signal Propagation**: `upstream`.

4 Add the `Actuator Model` subsystem as a starting point. In the model, right-click
   the `Actuator Model` subsystem and select **Model Slicer > Add as Starting
   Point**.
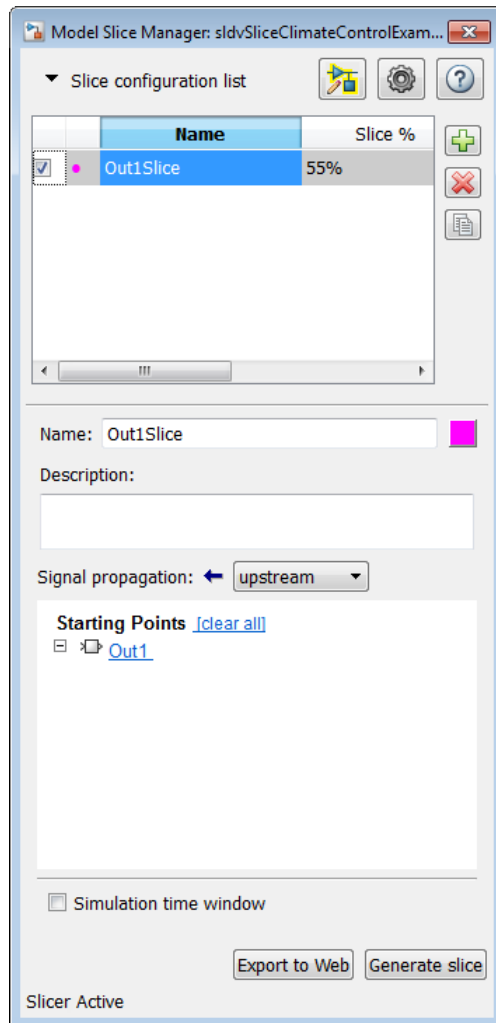
View Precedents and Generate Model Slice

1 The model highlights the upstream dependencies of the `Actuator Model` subsystem.
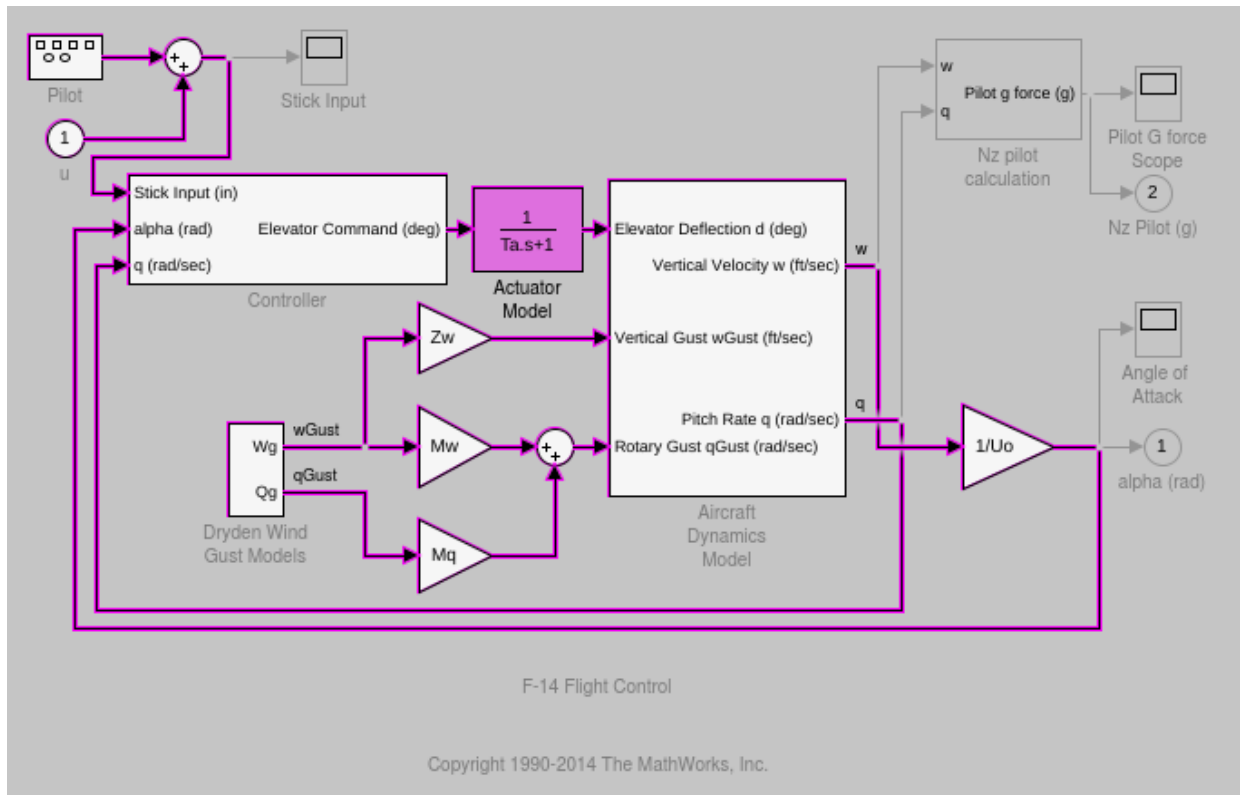
Trace the following dependency path. `Aircraft Dynamics Model` is highlighted via the `Pitch Rate q` signal, which is an input to `Controller`, the output of which feeds `Actuator Model`.

2 Generate a standalone model containing the highlighted model items:

   a In the Model Slice Manager, click **Generate slice**.

   b In the **Select File to Write** dialog box, select the save location and enter `actuator_slice_model`.

   c Click **Save**.

3 The sliced model contains the highlighted model items.
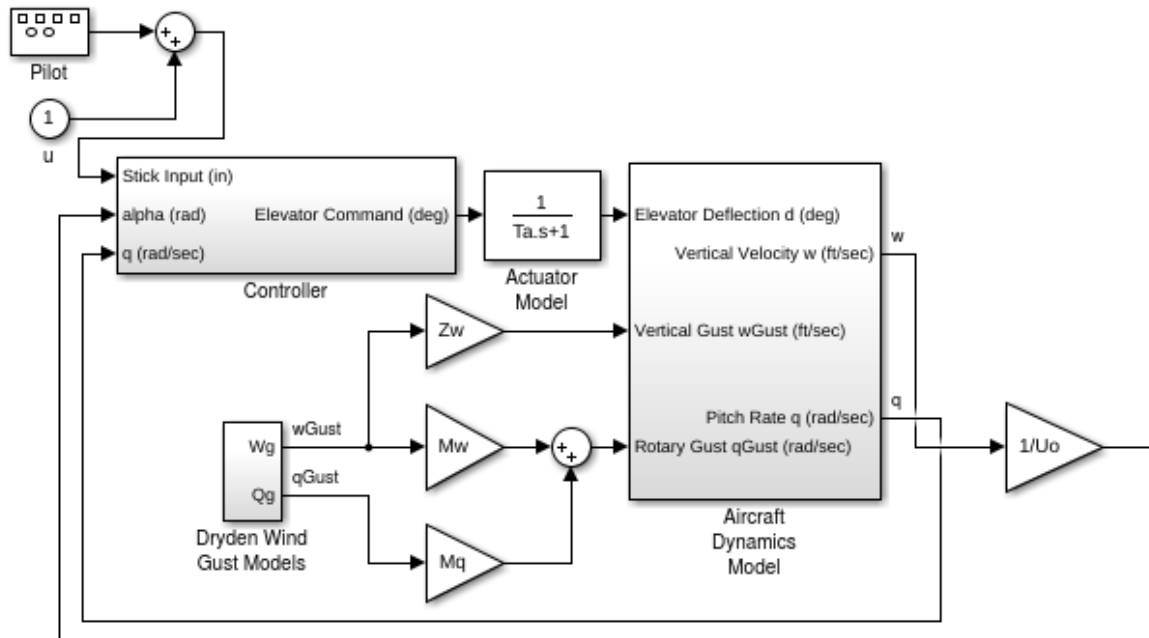
F-14 Flight Control

Copyright 1990-2014 The MathWorks, Inc.

4 To remove highlighting from the model, close the Model Slice Manager.

Isolate Subsystems for Functional Testing

If your model interest centers on a subsystem, you can create a standalone model for subsystem debugging and refinement. The standalone model isolates the subsystem and relevant signals. You can observe the subsystem behavior without simulating the entire source model.

Note: You cannot slice virtual subsystems. If you want to isolate a virtual subsystem, first convert it to an atomic subsystem.

Isolate a Subsystem with Simulation-Based Inputs

If you want to observe the simulation behavior of a subsystem, you can include logged signal inputs in the standalone model. Specify a simulation time window when you configure the model slice. For large models, observing subsystem behavior in a separate model can save time compared to compiling and running the entire source model.

This example demonstrates how to include simulation effects for the controller subsystem of a cruise control system.
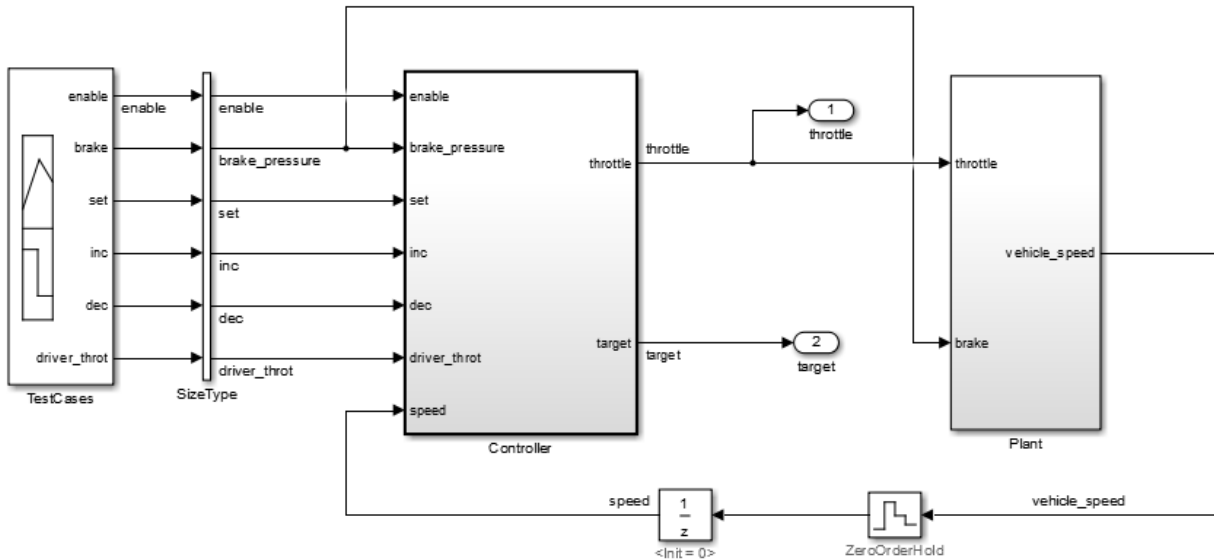
1. Open the model slice manager. From the menu, click **Analysis** > **Design Verifier** > **Model Slicer**.

2. Select the starting point for dependency analysis. Right-click a block, signal, or port, and select **Model Slicer** > **Add as starting point**.

3. Select the subsystem to isolate in the sliced model. Right-click the subsystem, and select **Model Slicer** > **Slice subsystem**.

   In the example model, selecting **Slice subsystem** for the controller limits the dependency analysis to the path between the starting point (the throttle outport) and the Controller subsystem.
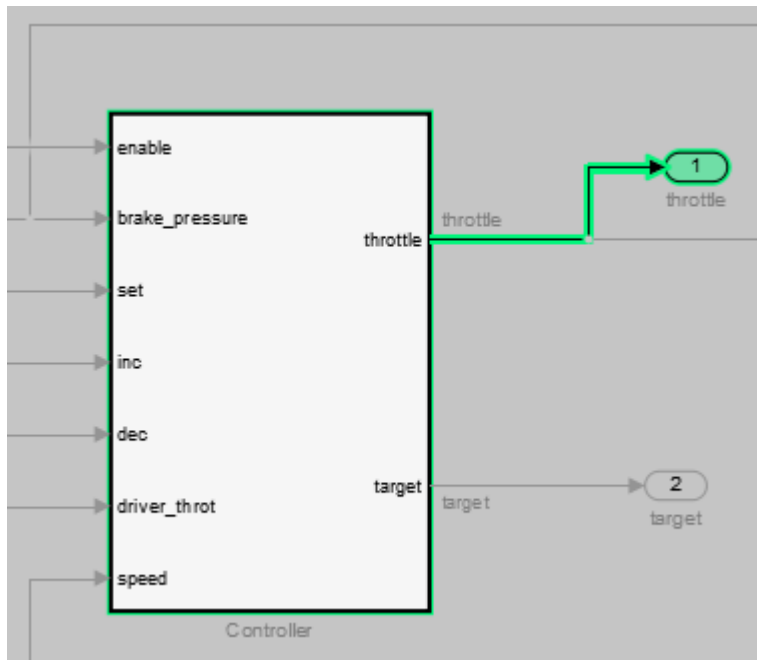
4 Specify the simulation time window:

    a Select **Simulation time window** in the model slice manager.

    b Click the run simulation button in the model slice manager.

    c Enter the simulation stop time, and click **OK**.
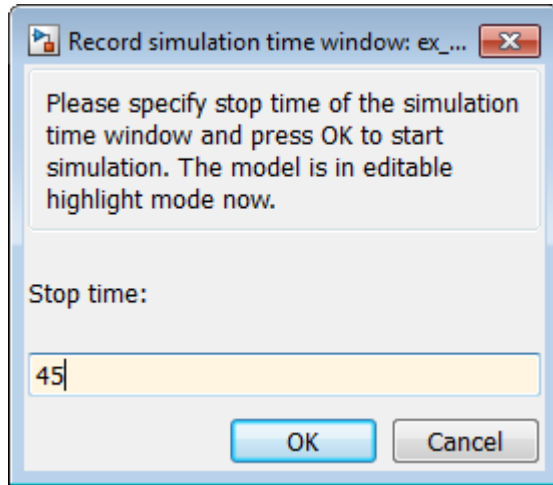
Model slicer analyzes the model dependencies for the simulation interval.

5 Click **Generate slice** to extract the subsystem and logged signals. Enter a file name for the sliced model.

Based on the dependency analysis, signal inputs are supplied to the subsystem with a block.

In the sliced model shown, the sliced model signal builder contains one test case representing the signals into the controller subsystem for simulation time 0–45 seconds.
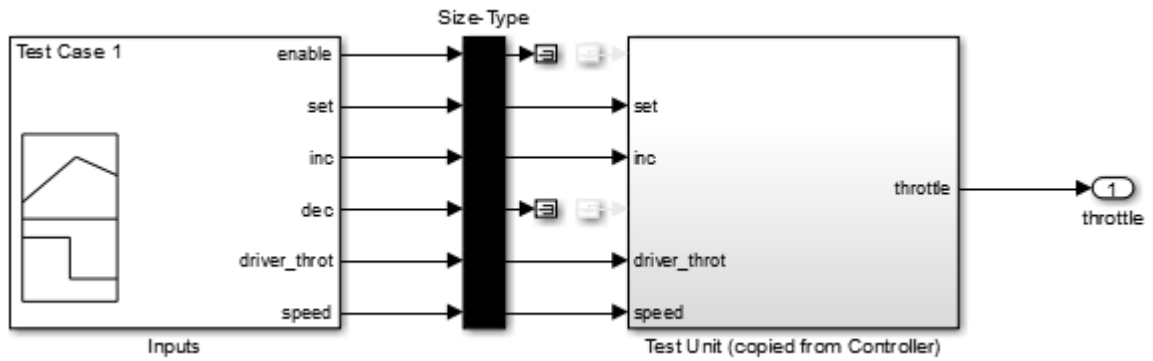
See Also

"Model Slicer Considerations and Limitations" on page 16-36 | "Highlight Functional Dependencies" on page 16-2

Use Existing .slslicex Simulation Time Window Data to Highlight Functional Dependencies

Model Slicer uses simulation results when highlighting functional dependencies in a model using a restricted simulation time window. For very large or complex models, the simulation time can be lengthy. When you highlight functional dependencies in a model using a restricted simulation time window, Model Slicer saves the simulation time window results in a file `<current_folder>\modelslicer\<model_name>\<model_name>.slslicex`.

If you want to highlight functional dependencies in the model again at another time, you can use the existing simulation time window data in `<current_folder>\modelslicer\<model_name>\<model_name>.slslicex` to highlight and refine functional dependencies in the model without needing to resimulate the model.

1 Open the model in Simulink.

2 Select **Analysis** > **Design Verifier** > **Model Slicer...** to open the Model Slice Manager.

3 Select **Simulation time window**.

4 Click **Use existing simulation data** 📂.

5 Navigate to the existing simulation data in `<current_folder>\modelslicer\<model_name>\<model_name>.slslicex` and click **Open**.

Model Slicer then uses the existing simulation data to highlight the model without needing to resimulate the model.

More About

• "Highlight Functional Dependencies" on page 16-2

Programmatically Resolve Unexpected Behavior in a Model with Model Slicer

In this section...
"Prerequisites" on page 16-56
"Find and Isolate the Area of the Model Responsible for Unexpected Behavior" on page 16-56
"Investigate the Sliced Model and Debug the Source Model" on page 16-62

In this tutorial, you evaluate a Simulink model, detect unexpected behavior, and use Model Slicer to programmatically isolate and resolve the unexpected behavior. When you plan to reuse your API commands and extend their use to other models, a programmatic approach is useful.

Prerequisites

Be familiar with the behavior and purpose of Model Slicer and the functionality of the Model Slicer API. "Highlight Functional Dependencies" on page 16-2 outlines how to use Model Slicer user interface to explore models. The `slslicer`, `slsliceroptions`, and `slslicertrace` function reference pages contain the Model Slicer API command help.

Find and Isolate the Area of the Model Responsible for Unexpected Behavior

The `sldvSliceCruiseControlHarness` test harness model contains a cruise controller subsystem `sldvSliceCruiseControl` and a block, `TestCases`, containing a test case for this subsystem. You first simulate the model to execute the test case. You then evaluate the behavior of the model to find and isolate areas of the model responsible for unexpected behavior:

1 Add the example folder to the search path.

    addpath(fullfile(docroot,'toolbox','sldv','examples'))

2 Open the `sldvSliceCruiseControlHarness` test harness for the cruise control model.

    open_system('sldvSliceCruiseControlHarness')
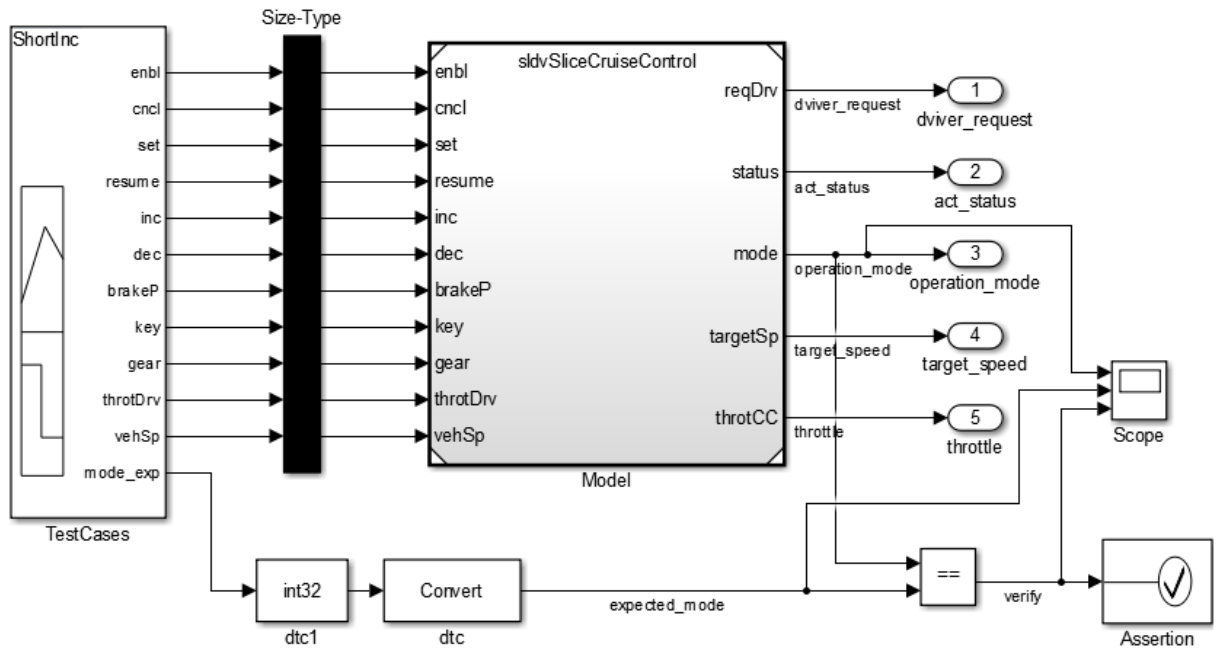
Note: The Assertion block is set to **Stop simulation when assertion fails** when the actual operation mode is not the same as the expected operation mode.

The TestCases block contains several test inputs for sldvSliceCruiseControl.

3　In the TestCases Signal Builder click the **Run all** button ![all] to run all of the included test cases. You receive an error during the ResumeWO test case.
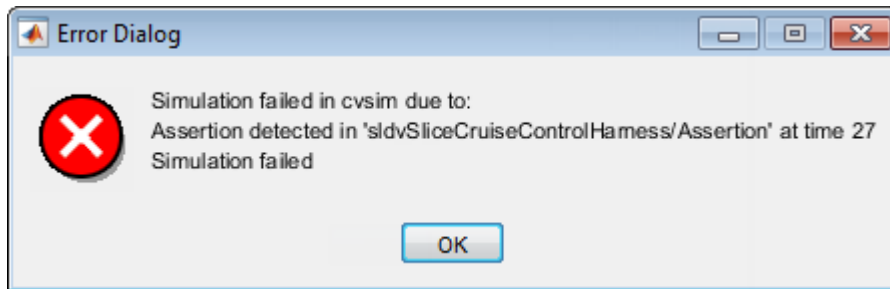
The Assertion block halted simulation at 27 seconds, because the actual operation mode was not the same as the expected operation mode. Click **OK** to close this error message.

4   In the sldvSliceCruiseControlHarness model, double-click the Assertion block, clear **Enable assertion**, and click **OK**.

Caution If you do not clear **Enable assertion**, you encounter an error when you slice the model.

5

Click run to run the simulation again.

The Scope block in the model contains three signals:

- operation_mode – displays the actual operation mode of the subsystem.
- expected_mode – displays the expected operation mode of the subsystem that the test case provides.
- verify – displays a Boolean value comparing the operation mode and the expected mode.
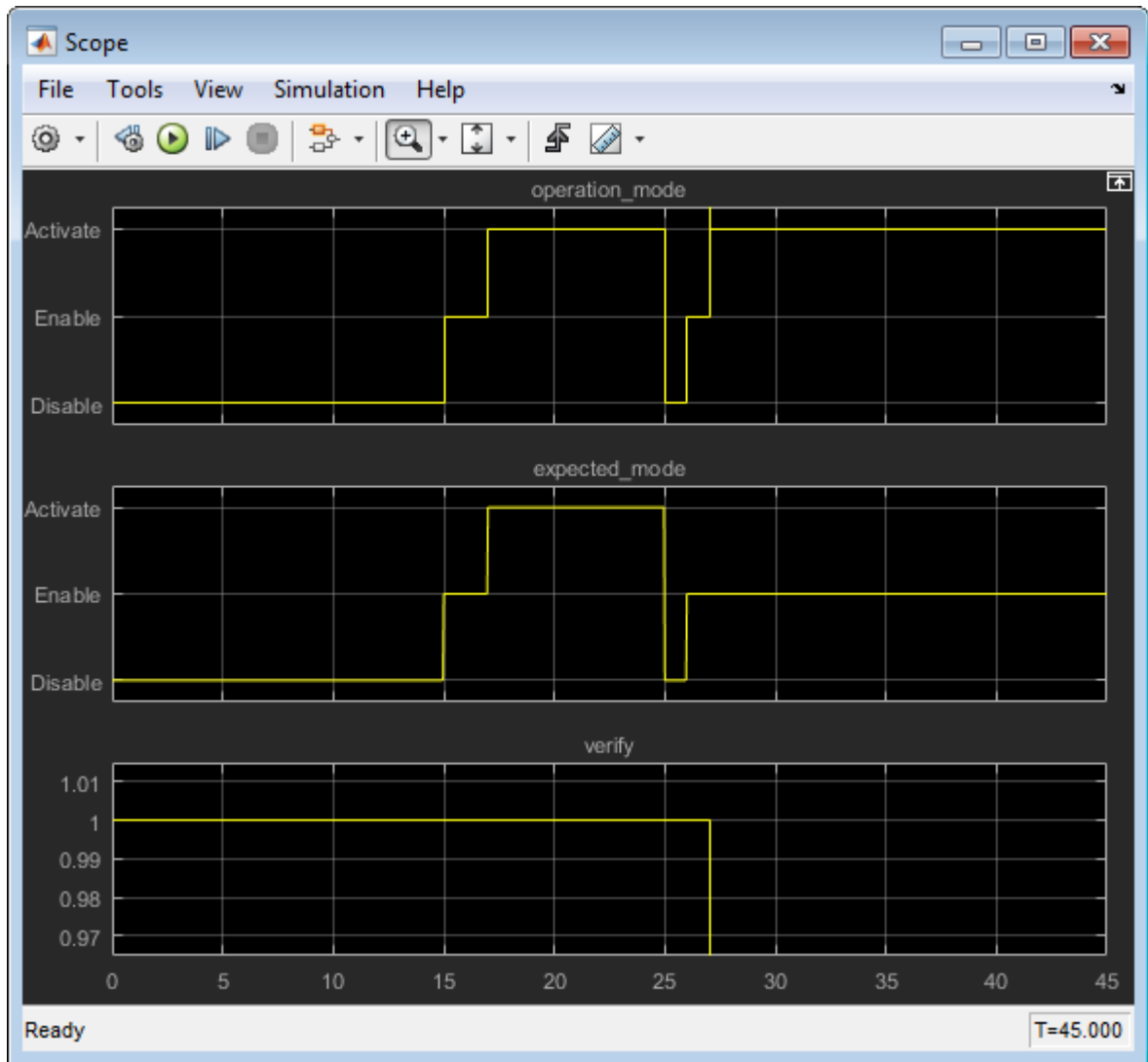
The scope shows a disparity between the expected operation mode and the actual operation mode beginning at time 27. Now that you know the outport displaying the unexpected behavior and the time window containing the unexpected behavior, use Model Slicer to isolate and analyze the unexpected behavior.

6   Create a Model Slicer configuration object for the model using `slslicer`.

```
obj = slslicer('sldvSliceCruiseControlHarness')
```
The Command Window displays the slice properties for this Model Slicer configuration.

7   Activate the slice highlighting mode of Model Slicer to compile the model and prepare it for dependency analysis.

```
activate(obj)
```

8   Add the `operation_mode` outport block as a starting point and highlight it.

```
addStartingPoint(obj,'sldvSliceCruiseControlHarness/operation_mode')
highlight(obj)
```
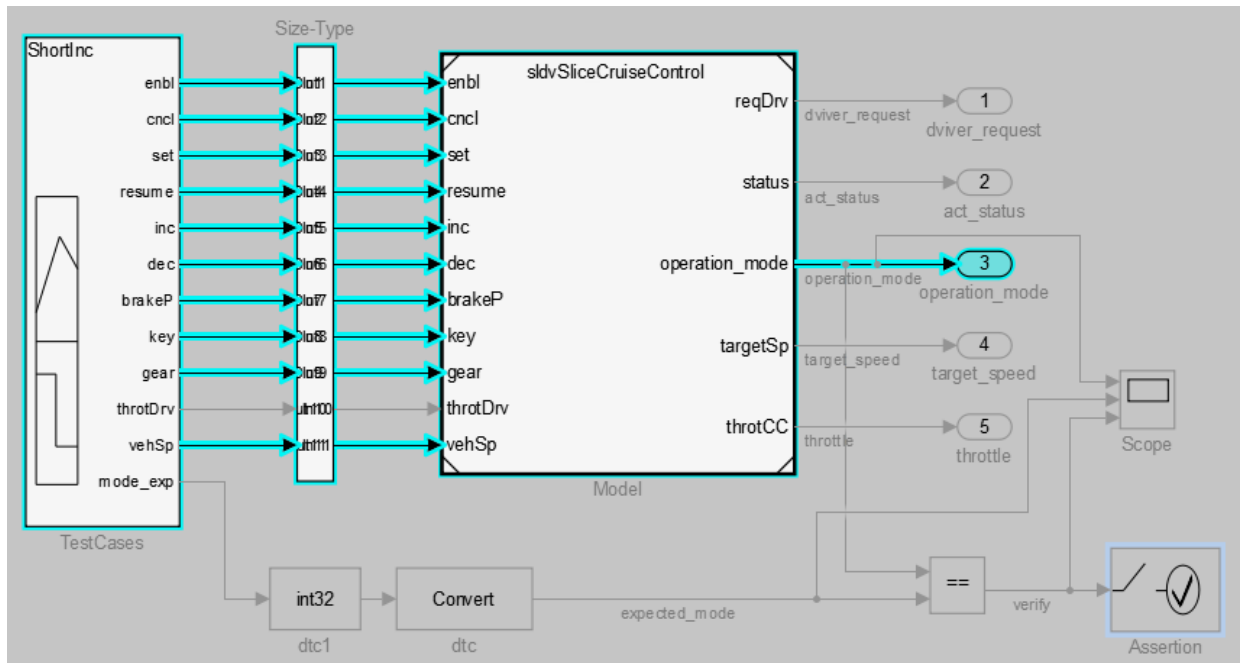

The area of the model upstream of the starting point and active during simulation is highlighted.

9 Simulate the model within a restricted simulation time window (maximum 30 seconds) to highlight only the area of the model upstream of the starting point and active during the time window of interest.

```
simulate(obj,0,30)
```
Only the portion of the model upstream of the starting point and active during the simulation time window is highlighted.
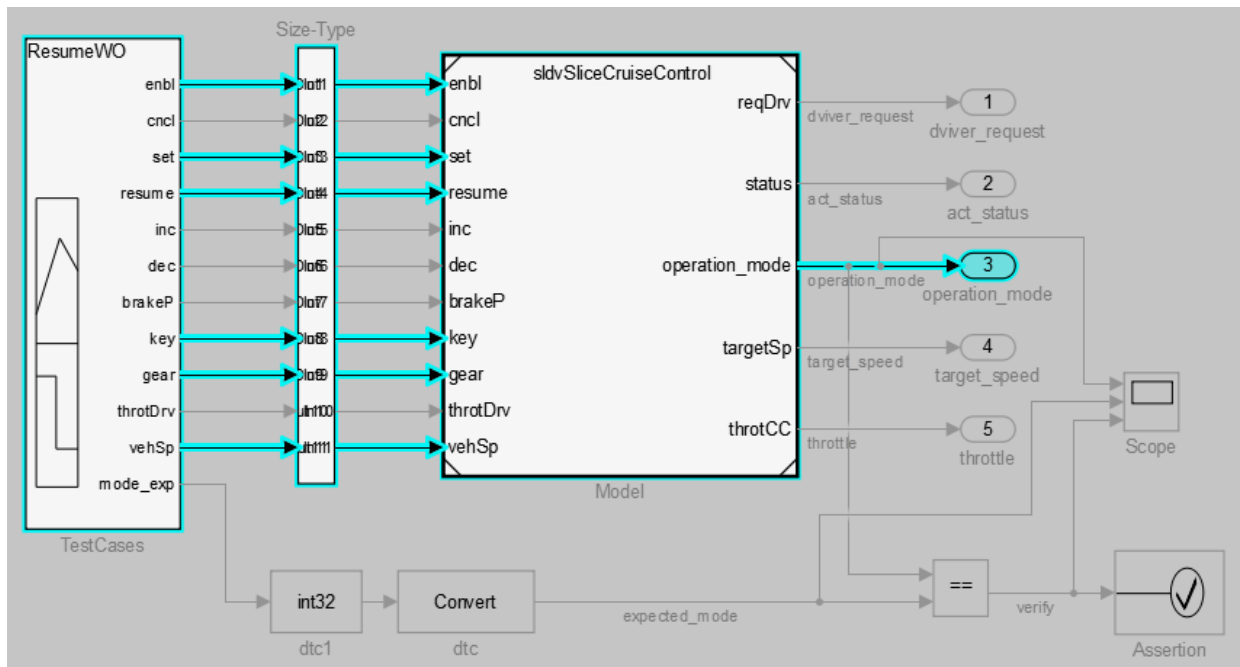
10 Create a sliced model sldvSliceCruiseControlHarness_sliced containing only the area of interest.

```
slicedModel = slice(obj,'sldvSliceCruiseControlHarness_sliced')
```
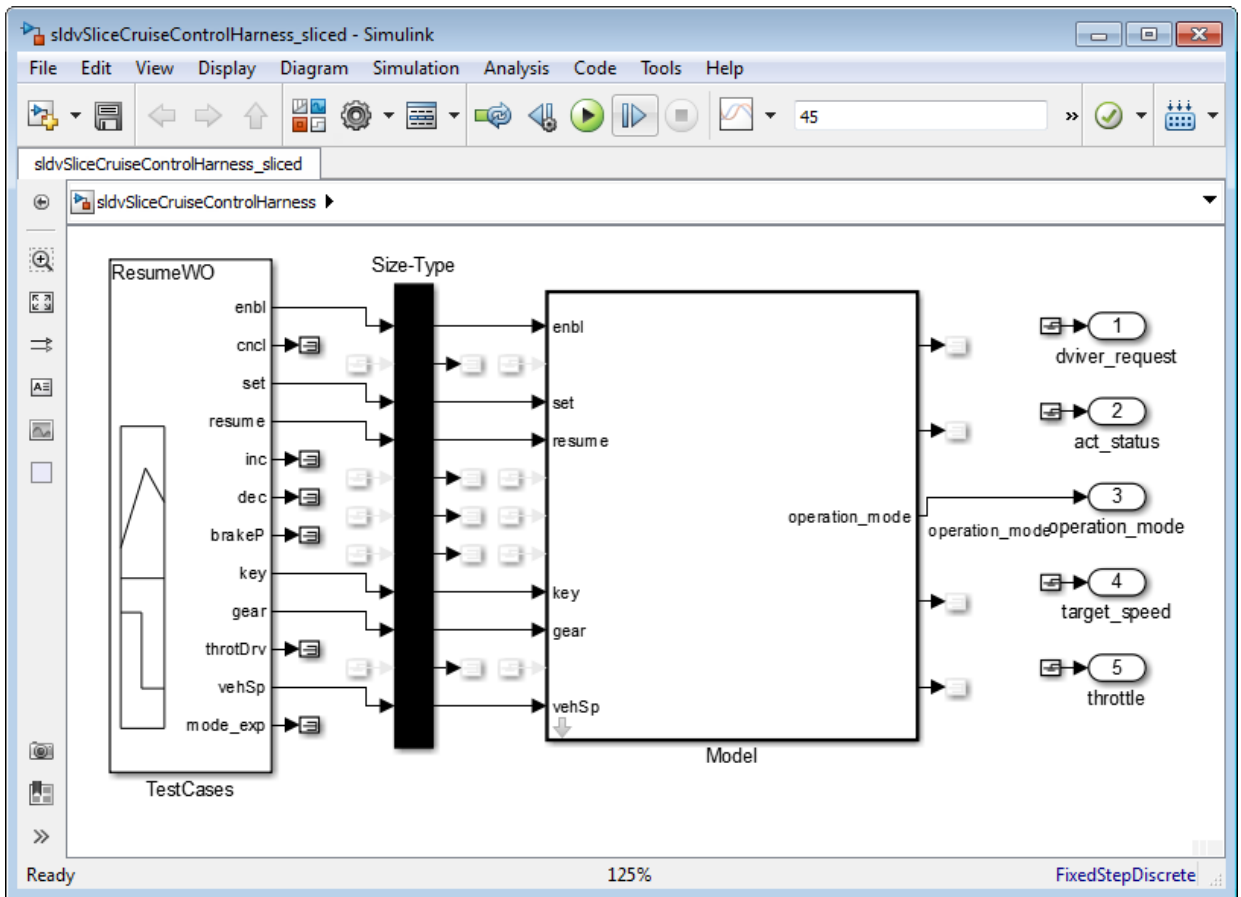
The sliced model `sldvSliceCruiseControlHarness_sliced` now contains a simplified version of the source model `sldvSliceCruiseControlHarness`. The simplified standalone model contains only those parts of the model that are upstream of the specified starting point and active during the time window of interest.

Investigate the Sliced Model and Debug the Source Model

You can now debug the unexpected behavior in the simplified standalone model and then apply changes to the source model.

1 To enable editing the model again, terminate the Model Slicer mode.

```
terminate(obj)
```

2 Navigate to the area of the sliced model that contains the unexpected behavior.

```
open_system('sldvSliceCruiseControlHarness_sliced
/Model/CruiseControlMode/opMode/resumeCondition/hasCanceled')
```
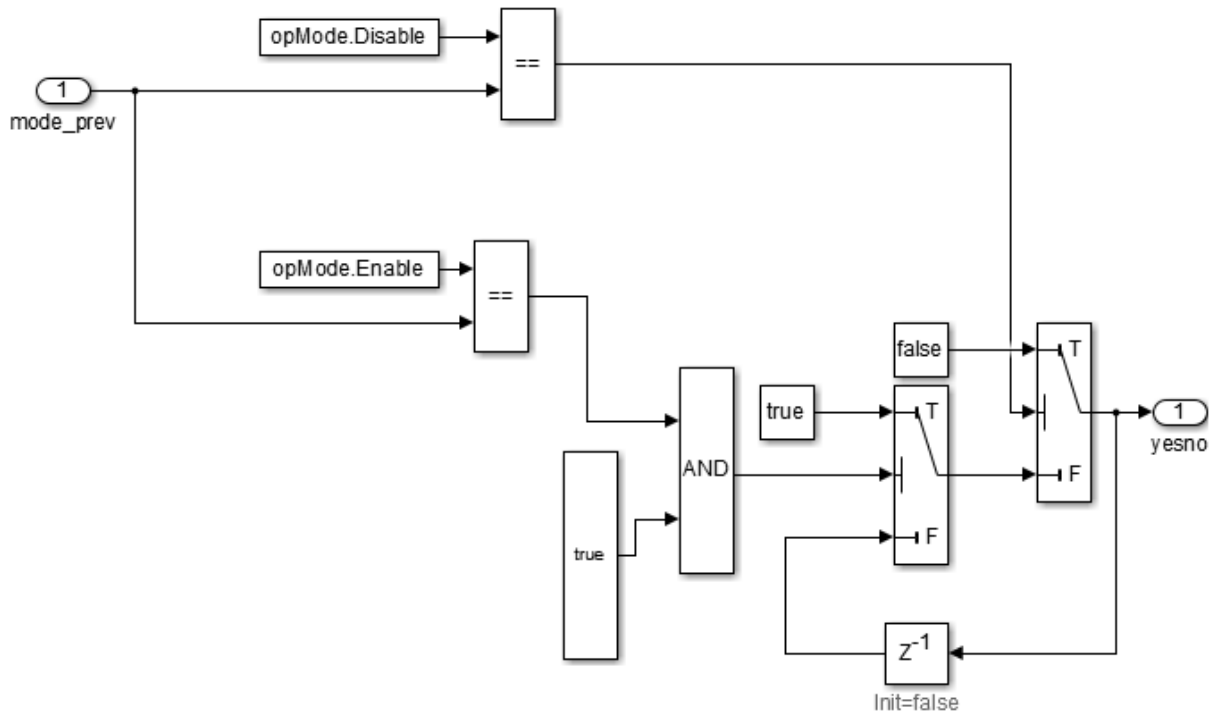

The AND Logical Operator block in this subsystem has a truncated `true` constant attached to its second input port. This `true` constant indicates that the second input port is always `true` during the restricted time window for this sliced model, causing the cruise control system not to enter the "has canceled" state.

3 Navigate to the equivalent AND Logical Operator block in the source system by using `slslicertrace` to view the blocks connected to the second input port.

```
h = slslicertrace('SOURCE', 'sldvSliceCruiseControlHarness_sliced
```

16-63

```
/Model/CruiseControlMode/opMode/resumeCondition/hasCanceled/LogicOp1')
hilite_system(h)
```
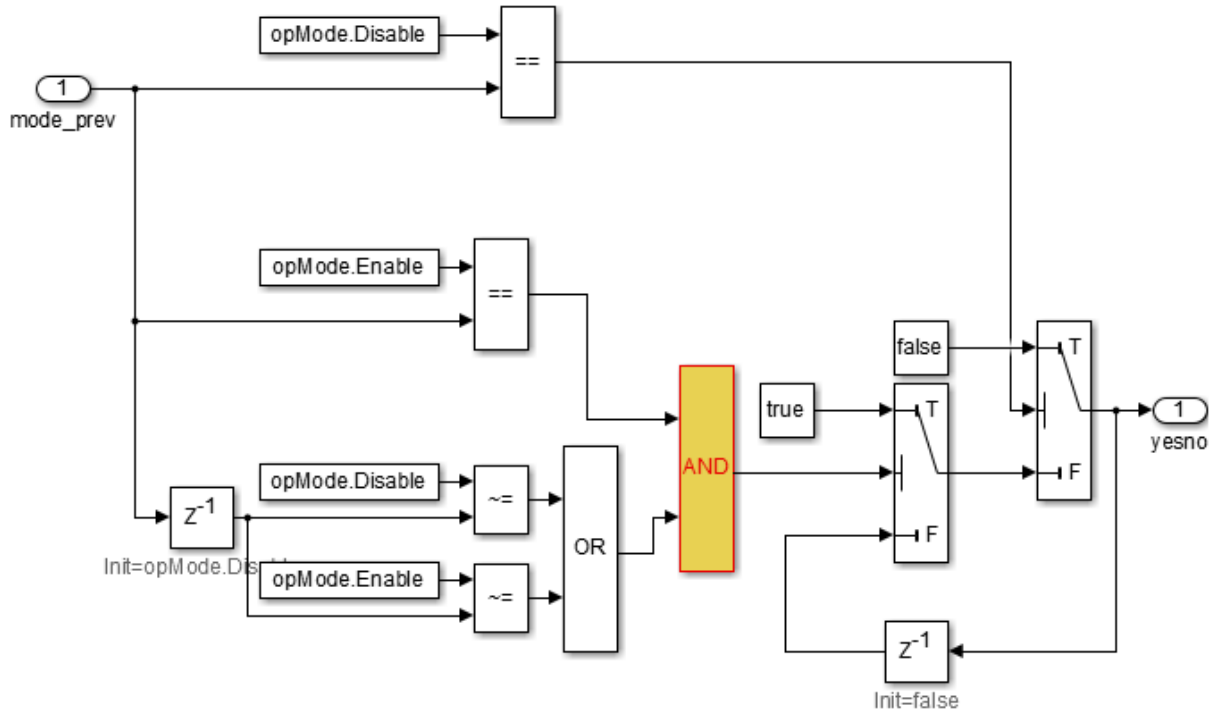

The OR Logical Operator block in this subsystem is always `true` in the current
configuration. Changing the OR Logical Operator block to an AND Logical Operator
block rectifies this error.

4 Before making edits, create new copies of the cruise control model and the test
harness model.

```
save_system('sldvSliceCruiseControl','sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness',
'sldvSliceCruiseControlHarness_fixed')
```

5 Update the model reference in the test harness to refer to the newly saved model.

```
set_param('sldvSliceCruiseControlHarness_fixed/Model',
```

```
'ModelNameDialog','sldvSliceCruiseControl_fixed.slx')
```

6 Use the block path of the erroneous Logical Operator block to fix the error.

```
set_param('sldvSliceCruiseControl_fixed/CruiseControlMode/opMode
/resumeCondition/hasCanceled/LogicOp2','LogicOp','AND')
```
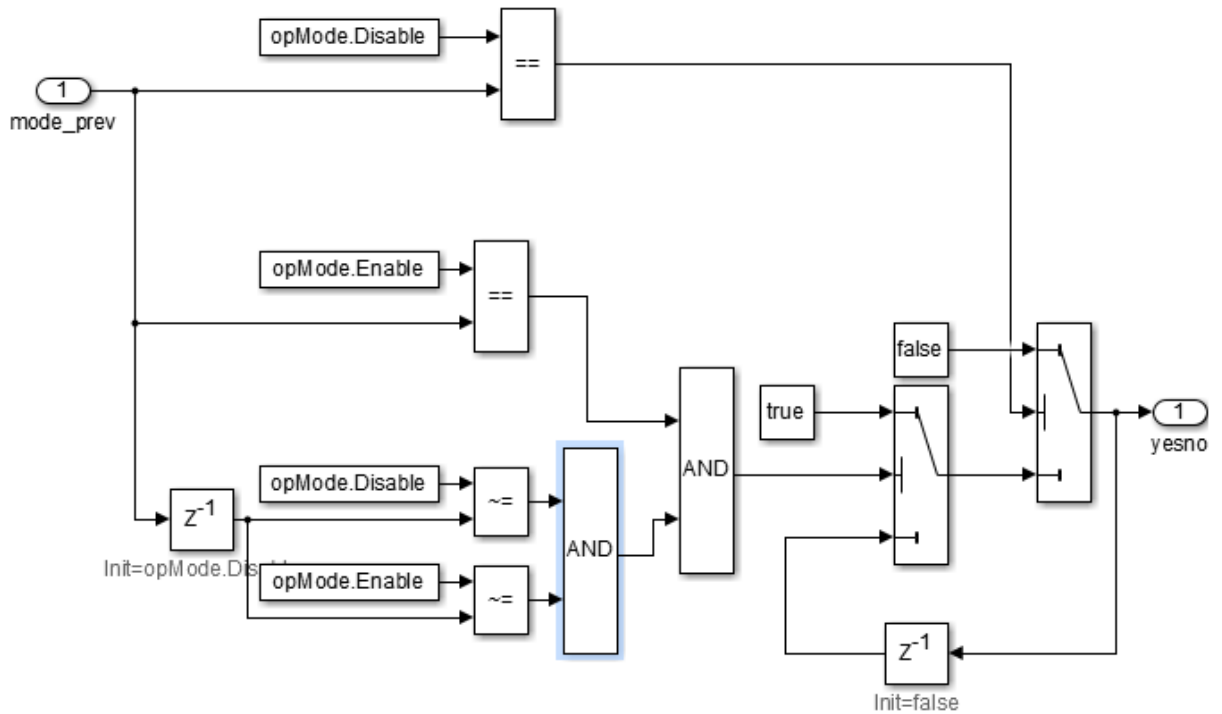

7 Simulate the test harness with the fixed model to confirm the corrected behavior.

```
sim('sldvSliceCruiseControlHarness_fixed')
```
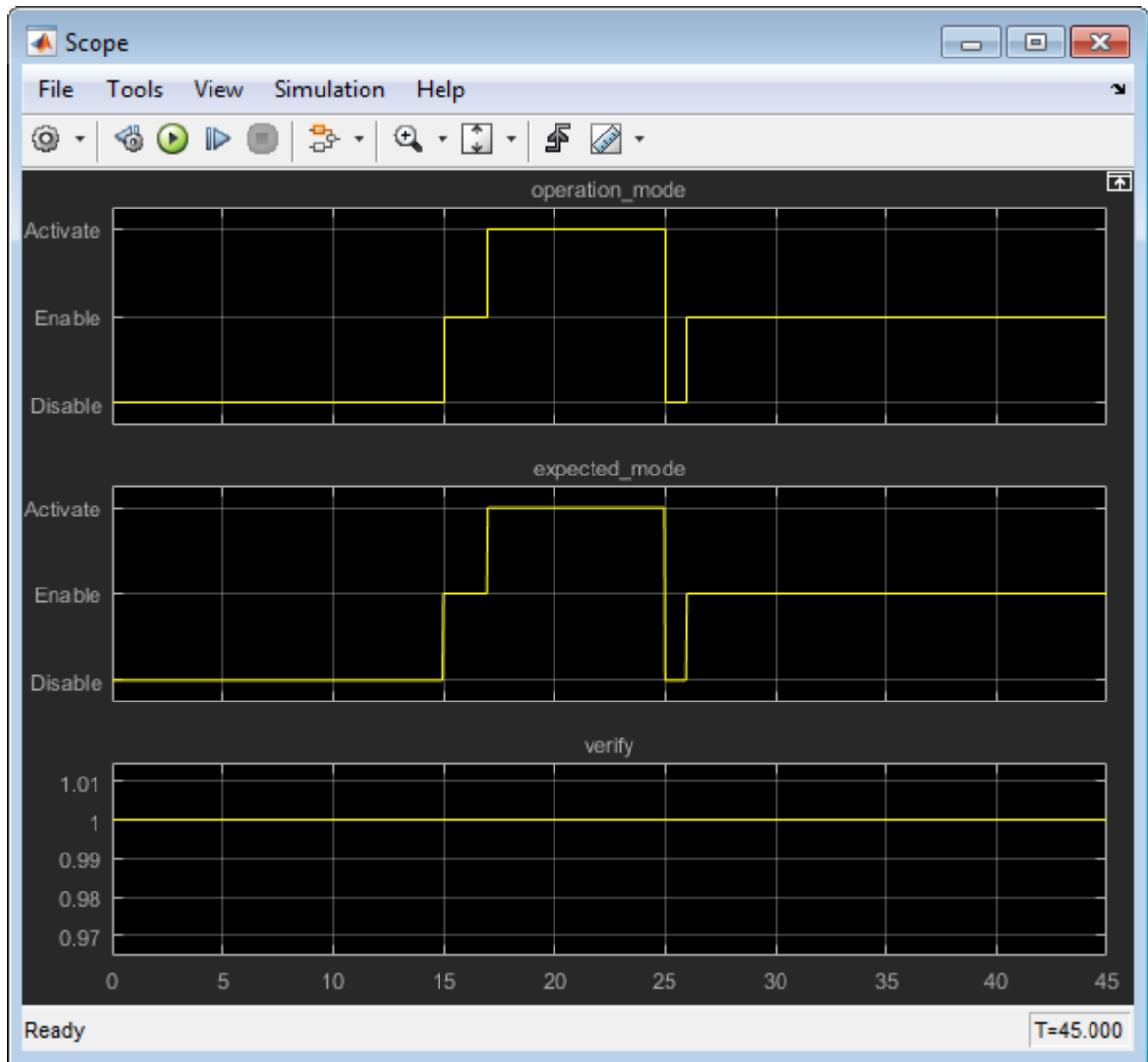
The scope now shows that the expected operation mode is the same as the actual operation mode.

See Also

slslicer | slsliceroptions | slslicertrace

More About

- "Highlight Functional Dependencies" on page 16-2

Simplification of Variant Systems

In this section...
"Use the Variant Reducer to Simplify Variant Systems" on page 16-68
"Use Model Slicer to Simplify Variant Systems" on page 16-68

If your model contains "Variant Systems", you can reduce the model to a simplified, standalone model containing only selected variant configurations.

Use the Variant Reducer to Simplify Variant Systems

After you Add and Validate Variant Configurations, you can reduce the model from the Variant Manager:

1   Open a model containing at least one valid variant configuration.
2   Select **View >> Variant Manager**, or right-click a variant system and select **Variant >> Open in Variant Manager**.
3   Click **Reduce model...**.
4   Select one or more variant configurations.
5   Set the **Output directory**.
6   Click **Reduce** to create a simplified, standalone model containing only the selected variant configurations.

The Variant Reducer creates a simplified, standalone model in the output directory you specified containing only the variant configurations you selected.

Use Model Slicer to Simplify Variant Systems

After you Add and Validate Variant Configurations, you can use Model Slicer to create a simplified, standalone model containing only the active variant configuration. When you "Highlight Functional Dependencies" on page 16-2 in a model containing variant systems, only active variant choices are highlighted. When you "Create a Simplified Standalone Model" on page 16-21 from a model highlight that includes variant systems, Model Slicer removes the variant systems and replaces them with the active variant configurations.

For instructions on how to change the active variant configuration and how to set default variant choices, see "Working with Variant Choices".

More About

- "Create a Simple Variant Model"
- "Define, Configure, and Activate Variants"
- "Introduction to Variant Controls"

17

Verification and Validation

Test Model Against Requirements and Report Results

Requirements Overview

Requirements are the basis for your system architecture, algorithm, and test plan. Traceability between requirements documents, model, code, and tests helps you document relationships, manage design changes, and interpret test results. Required model properties and test objectives enable targeted design analysis and test case generation for specific scenarios. You can evaluate your design and identify incomplete or missing requirements with ad-hoc testing, using simulated user interfaces for your model. Also, you can use rapid prototyping to validate requirements, and connect to physical or simulated environments to test your algorithm. Update the design, adding requirements and requirements links as necessary.
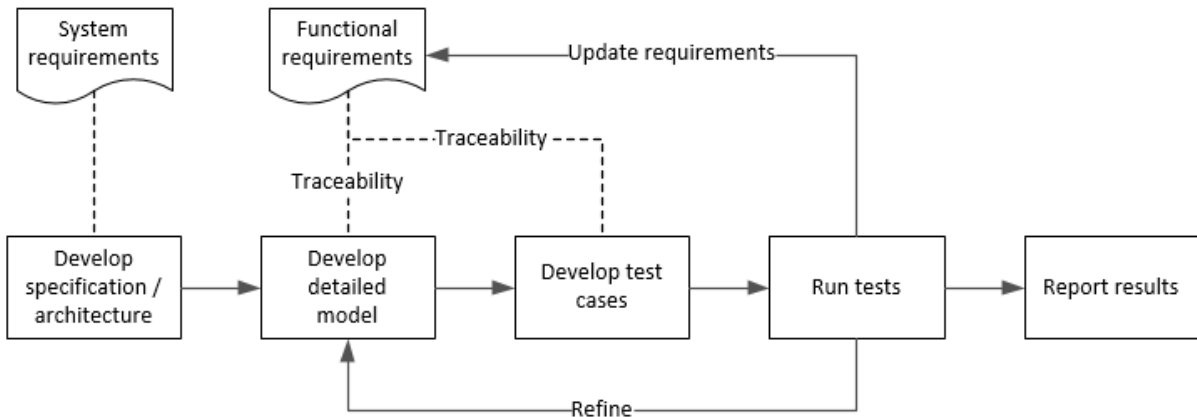
Test a Cruise Control Safety Requirement

This example shows a requirements-based testing workflow for a cruise control model. You start with a model that has traceability to an external requirements document. You add a test to evaluate two safety requirements, checking that the cruise control disengages when the system reaches certain conditions. You add traceability to this test, run the test, and report the results.

1   Create a copy of the project in a working folder. Enter

    slVerificationCruiseStart

2 Open the model and the test harness. On the command line, enter

```
open_system simulinkCruiseAddReqExample
sltest.harness.open('simulinkCruiseAddReqExample','SafetyTest_Harness1')
```

3 Open the Test Sequence block.

- The `BrakeTest` sequence tests that the system disengages when the brake pedal is pressed. It includes a `verify` statement

```
verify(engaged == false,...
    'verify:brake',...
    'system must disengage when brake applied')
```

- The `LimitTest` sequence tests that the system disengages when the speed exceeds a limit. It includes a `verify` statement

```
verify(engaged == false,...
    'verify:limit',...
    'system must disengage when limit exceeded')
```

4 Open the requirements document. In the Simulink Project window, expand the **documents** folder and open **simulinkCruiseChartReqs.docx**.

5 Add links between the test steps and the requirements document.

1 In the requirements document, highlight item 3.1, "Vehicle braking will transition system to disengaged (inactive) when engaged (active)"

2 With item 3.1 highlighted, in the test sequence, right-click the `BrakeTest` step. Select **Requirements traceability** > **Link to Selection in Word**.

3 In the requirements document, highlight item 3.4, "Transition to disengaged (inactive) when vehicle speed is outside the limits of 20 mph to 90 mph"

4 With item 3.4 highlighted, in the test sequence, right-click the `LimitTest` step. Select **Requirements traceability** > **Link to Selection in Word**.

5 Save the requirements document and the model.

6 Create a test case in the Test Manager, and link the test case to the requirements section.

1 Open the Test Manager. In the Simulink menu, select **Analysis** > **Test Manager**.

2 In the Test Manager toolstrip, click **New** > **Test File**. Select the tests folder in the project, and enter a name for the test file. Click **Save**.

A new baseline test is created.

3   Under **System Under Test**, in the **Model** field, click the button 📌 to use the current model. The field displays the model name.

4   Expand the **Test Harness** section. From the drop-down menu, select the test harness name.

5   In the requirements document, highlight section 3.1.

6   In the test case, expand the **Requirements** section. Click the arrow next to the **Add** button and select `Link to Selection in Word`.

7   Use the same process to link the test case to section 3.4 in the requirements document.

7   Highlight the test case. In the Test Manager toolstrip, click **Run**.

8   When the test finishes, expand the **Verify Statements** results. The results show that both assessments pass, and the plot shows the detailed results of each statement.
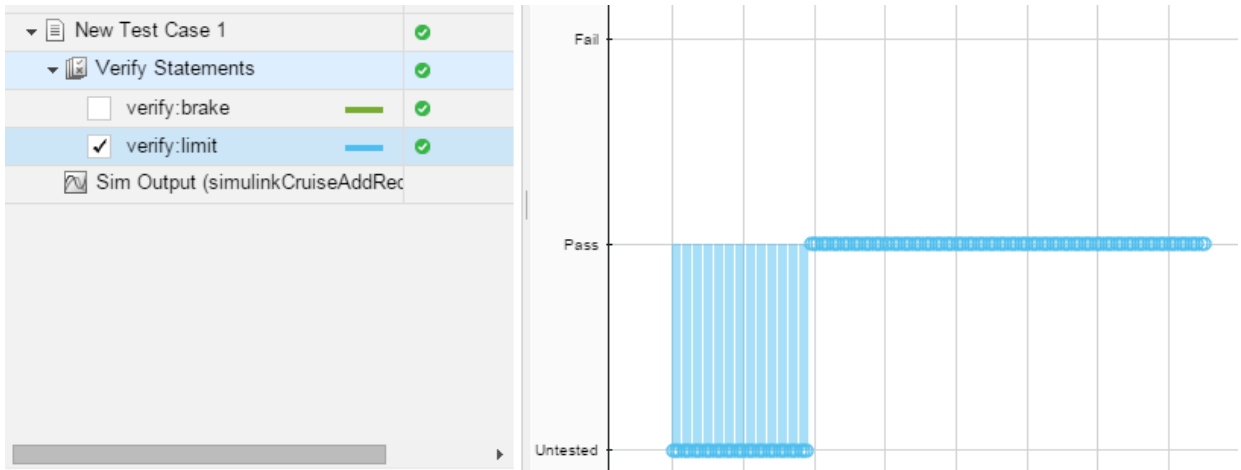
9   Create a report using a custom Microsoft Word template.

1   In the Test Manager, right-click the test case name. Select **Results:** > **Create Report**.

2   In the Create Test Result Report dialog box, set the options:

- Title: `SafetyTest`
- Results for: `All Tests`

- File Format: DOCX
- For the other options, keep the default selections.

3 For the **Template File**, select the ReportTemplate.dotx file in the **documents** project folder.

4 Enter a file name and select a location for the report.

5 Click **Create**.

10 Review the report.

1 In the **Test Case Requirements** section, click the link to trace to the requirements document.

2 The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

Name	Data Type	Units	Sample Time	Interp	Sync	Link to Plot
✅ Test Sequence/.../Verify:verify(engaged == false)	slTestResult			zoh	union	Link
✅ Test Sequence/.../VerifyHigh:verify(engaged == false)	slTestResult			zoh	union	Link

Related Examples

- "Link to Requirements Modeled in Simulink"
- "Link Tests to Requirements"
- "Validate Requirements Links in a Model"
- "Create Requirements Traceability Report for Model"

Analyze a Model for Standards Compliance and Design Errors

Standards and Analysis Overview

During model development, check and analyze your model to increase confidence in its quality. Check your model against standards such as MAAB style guidelines and high-integrity system design guidelines such as DO-178 and ISO 26262. Analyze your model for errors, dead logic, and conditions that violate required properties. Using the analysis results, update your model and document exceptions. Report the results using customizable templates.
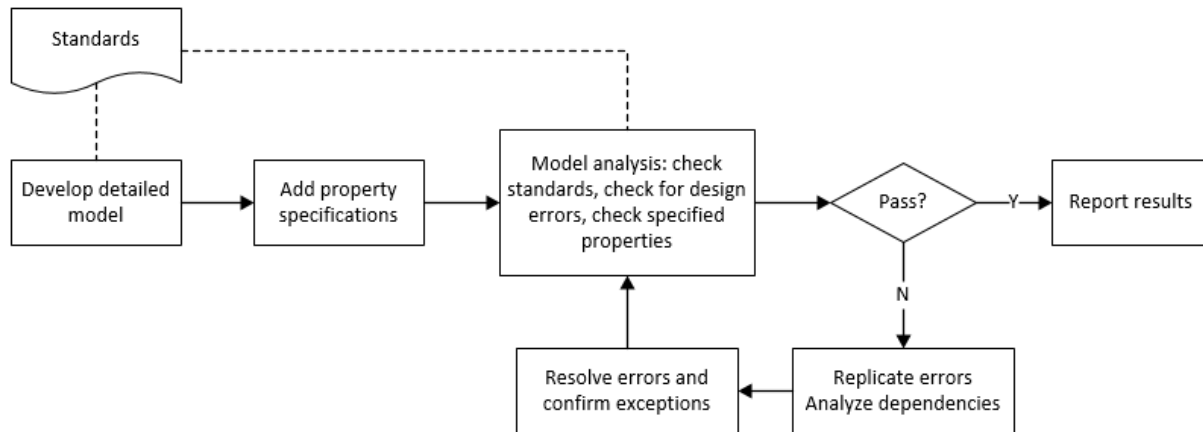
Check Model for Style Guideline Violations and Design Errors

This example shows how to use the Model Advisor to check a cruise control model for MathWorks® Automotive Advisory Board (MAAB) style guideline violations and design errors. Select checks and run the analysis on the model. Iteratively debug issues using the Model Advisor and rerun checks to verify that it is in compliance. After passing your selected checks, report results.

Check Model for MAAB Style Guideline Violations

In Model Advisor, you can check that your model complies with MAAB modeling guidelines.

1 Create a copy of the project in a working folder. On the command line, enter

slVerificationCruiseStart

2 Open the model. On the command line, enter

open_system simulinkCruiseErrorAndStandardsExample

3 In the model window, select **Analysis** > **Model Advisor** > **Model Advisor**.

4 Click OK to choose simulinkCruiseErrorAndStandardsExample from the System Hierarchy.

5 Check your model for MAAB style guideline violations using Simulink Verification and Validation.

    a In the left pane, in the **By Product > Simulink Verification and Validation > Modeling Standards > MathWorks Automotive Advisory Board Checks** folder, select:

        · **Check for indexing in blocks**

        · **Check for prohibited blocks in discrete controllers**

        · **Check model diagnostic parameters**

    b Right-click the **MathWorks Automotive Advisory Board Checks** node, and then select Run Selected Checks.

    c Click **Check model diagnostic parameters** to review the configuration parameter settings that violate MAAB style guidelines.

    d In the right pane, click the parameter links to update the values in the Configuration Parameters dialog box.

    e To verify that your model passes, rerun the check. Repeat steps **c** and **d**, if necessary, to reach compliance.

    f To generate a results report of the Simulink Verification and Validation checks, select the **MathWorks Automotive Advisory Board Checks** node, and then, in the right pane click **Generate Report...**.

Check Model for Design Errors

While in Model Advisor, you can also check your model for hidden design errors using Simulink Design Verifier.

1 In the left pane, in the **By Product > Simulink Design Verifier** folder, select **Design Error Detection**.

2   In the right pane, click **Run Selected Checks**.

3   After the analysis is complete, expand the **Design Error Detection** folder, then select checks to review warnings or errors.

4   In the right pane, click **Simulink Design Verifier Results Summary**. The dialog box provides tools to help you diagnose errors and warnings in your model.

    a   Review the results on the model. Click **Highlight analysis results on model**. Click the `Compute target speed` subsystem, outlined in red. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.

    b   Review the harness model. The Simulink Design Verifier Results Inspector window displays information that an overflow error occurred. To see the test cases that demonstrate the errors, click **View test case**.

    c   Review the analysis report. In the Simulink Design Verifier Results Inspector window, click **Back to summary**. To see a detailed analysis report, click **HTML** or **PDF**.

Related Examples

- "Check for Compliance in Model and Subsystems"
- "Run Checks for Model Metrics"
- "Run a Design Error Detection Analysis" on page 6-5
- "Prove Properties in a Model" on page 12-5

Perform Functional Testing and Analyze Test Coverage

Functional Testing and Coverage Analysis Overview

Functional testing starts with building test cases based on requirements. These tests can cover key aspects of your design and verify that individual model components meet requirements. Test cases include inputs, expected outputs, and acceptance criteria.

By collecting individual test cases within test suites, you can run functional tests systematically. To check for regression, add baseline criteria to the test cases and test the model regularly. Coverage measurement reflects the extent to which these tests have fully exercised the model. Coverage measurement also helps you to add tests and requirements to meet coverage targets.
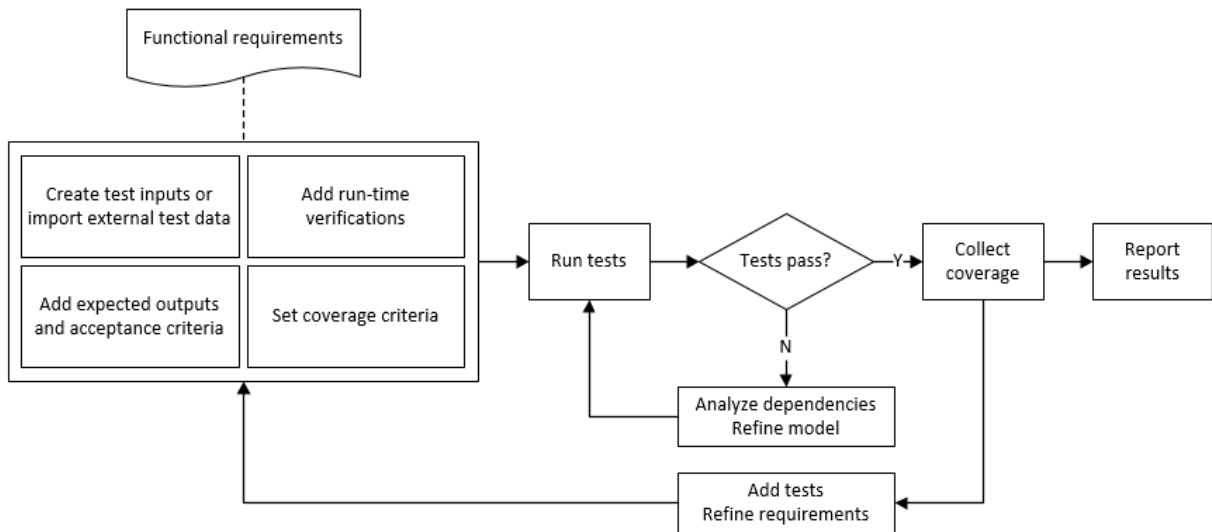
Incrementally Increase Test Coverage Using Test Case Generation

This example shows a functional testing-based testing workflow for a cruise control model. You start with a model that has tests linked to an external requirements document, analyze the model for coverage in Simulink Verification and Validation, incrementally increase coverage with Simulink Design Verifier, and report the results.

Explore the Test Harness and the Model

1 Create a copy of the project in a working folder. At the command line, enter:

```
slVerificationCruiseStart
```

2 Open the model and the test harness. At the command line, enter:

```
open_system simulinkCruiseAddReqExample
sltest.harness.open('simulinkCruiseAddReqExample','SafetyTest_Harness1')
```

3 Load the test suite from "Test Model Against Requirements and Report Results". At the command line, enter:

```
open slReqTests.mldatx
```

4 Open the test sequence block. The sequence tests:

- That the system disengages when the brake pedal is pressed

- That the system disengages when the speed exceeds a limit

Some test sequence steps are linked to a requirements document `simulinkCruiseChartReqs.docx`.

Measure Model Coverage and Save Coverage Results

1 In the test manager, enable coverage collection for the test case.

   a Open the test manager. In the Simulink menu, click **Analysis** > **Test Manager**.

   b In the **Test Browser**, click the `slReqTests` test file.

   c Expand **Coverage Settings**.

   d Under **COVERAGE TO COLLECT**, select **Record coverage for referenced models**.

   e Under **COVERAGE METRICS**, select **Decision**, **Condition**, and **MCDC**.

2   Run the test. On the test manager toolstrip, click **Run**.

3   When the test finishes, in the Test Manager, navigate to the test case. The example model achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.

4   In the coverage results, right-click the `simulinkCruiseAddReqExample` row and click **Export**

5   Select **Export to CVT-file** and export the coverage results to a folder in your working folder.

Generate Tests to Increase Model Coverage

1   Generate additional tests for the referenced model. Specify that the analysis ignore satisfied coverage data from the coverage results recorded in the previous steps.

    a   Open the `simulinkCruiseAddReqExample` model Configuration Parameters ⚙.

    b   In the **Design Verifier > Test Generation** pane, select **Ignore objectives satisfied in existing coverage data**

    c   Click **Browse**, select the Coverage data file that you exported in the previous section, and click **Open**.
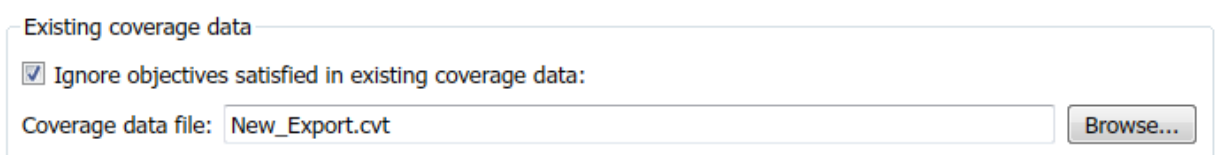
2   To apply the settings and close the Configuration Parameters dialog box, click **OK**.

3 In the `simulinkCruiseAddReqExample` model, right-click the `Compute target speed` subsystem and click **Design Verifier** > **Generate Tests for Subsystem**. The test case generation is executed.

The Simulink Design Verifier Results Summary window opens with the test case generation report.
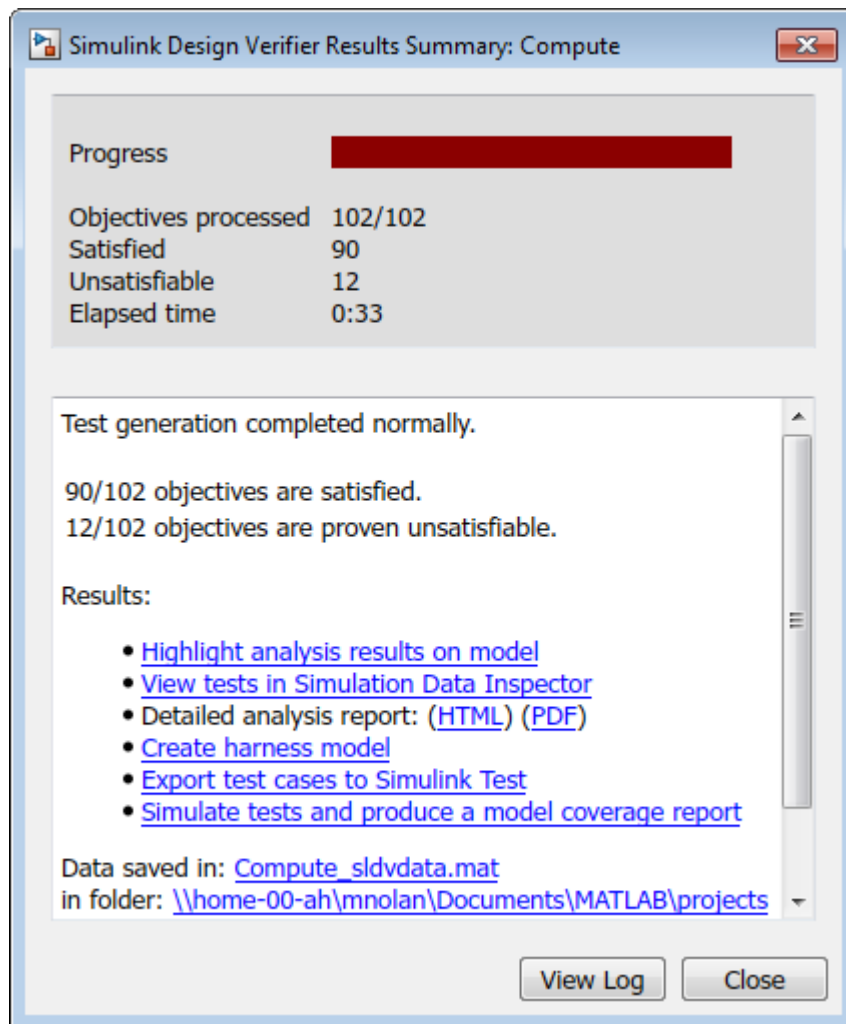
4　In the Simulink Design Verifier Results Summary window, click **Simulate tests and produce a model coverage report**.

The example model now achieves 89% decision coverage, 88% condition coverage, and 88% MCDC coverage.

Summary

Model Hierarchy/Complexity		Test 1		
		Decision	**Condition**	**MCDC**
1. Compute	31	89% ▬▬	88% ▬▬	88% ▬▬
2. . . . Compute target speed	30	89% ▬▬	88% ▬▬	88% ▬▬
3. . . . . . . SF: Compute target speed	29	89% ▬▬	88% ▬▬	88% ▬▬
4. . . . . SF: CRUISE	26	88% ▬▬	88% ▬▬	88% ▬▬
5. . . . . SF: ON	15	80% ▬▬	75% ▬▬	75% ▬▬

Export the Test Cases to the Test Manager

To export the test cases that Simulink Design Verifier generates into the Test Manager, first close the existing Simulink Test test harness.

1　Close the `SafetyTest_Harness1` test harness.

2　In the Simulink Design Verifier Results Summary window, click **Export test cases to Simulink Test**.

3　Select the `simulinkCruiseAddReqExample_sldvharness` test harness and click **OK**.

4　In the test manager, a new test file `simulinkCruiseAddReqExample_test` appears. The test file contains a new test case that uses the inputs generated by Simulink Design Verifier for model coverage.

5　Right-click the new test case in the `simulinkCruiseAddReqExample_test` test file and select **Copy**.

6　In the original `slReqTests` test file, right-click the test suite and select **Paste**.

The test suite in the `slReqTests` test file now contains the original test case and the test cases generated by Simulink Design Verifier.

7 Run the test suite again. Highlight the `slReqTests Test Suite 1` test suite. On the test manager toolstrip, click **Run**.

The test results include coverage for the combined test case inputs.

**8** Close the test harness and the model.

Related Examples

- "Link Tests to Requirements"
- "Assess Simulation Using Logical Statements"
- "Test Model Output Against a Baseline"
- "Highlight Functional Dependencies" on page 16-2
- "Generate Test Cases for Model Decision Coverage" on page 7-5
- "Extend Model Coverage of a Test Suite"

Analyze Code and Test Software-in-the-Loop

Code Analysis and Testing Software-in-the-Loop Overview

Analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. Typically for handwritten code, you check for run-time errors with static code analysis and run test cases that evaluate the code against requirements and evaluate code coverage. Based on the results, refine the code and add tests. For generated code, demonstrate that code execution produces equivalent results to the model by using the same test cases and baseline results. Compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.
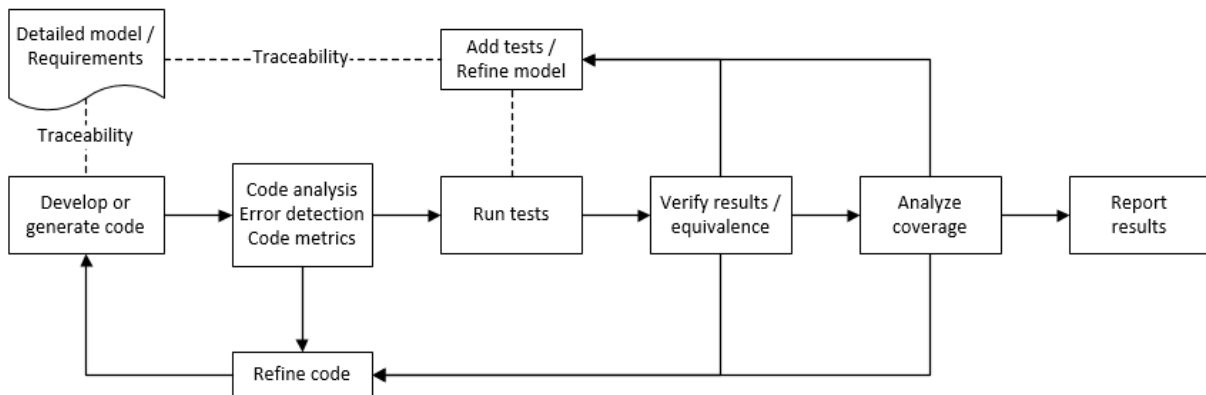

Analyze Code for Defects, Metrics, and MISRA C:2012

This workflow describes how to check if your model produces MISRA® C:2012 compliant code and how to check your generated code for code metrics, code defects, and MISRA compliance. To produce more MISRA compliant code from your model, you use the code generation and model advisors. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

1   Open the Simulink project:

slVerificationCruiseStart

2   From the Simulink project, open the model
    simulinkCruiseErrorAndStandardsExample.
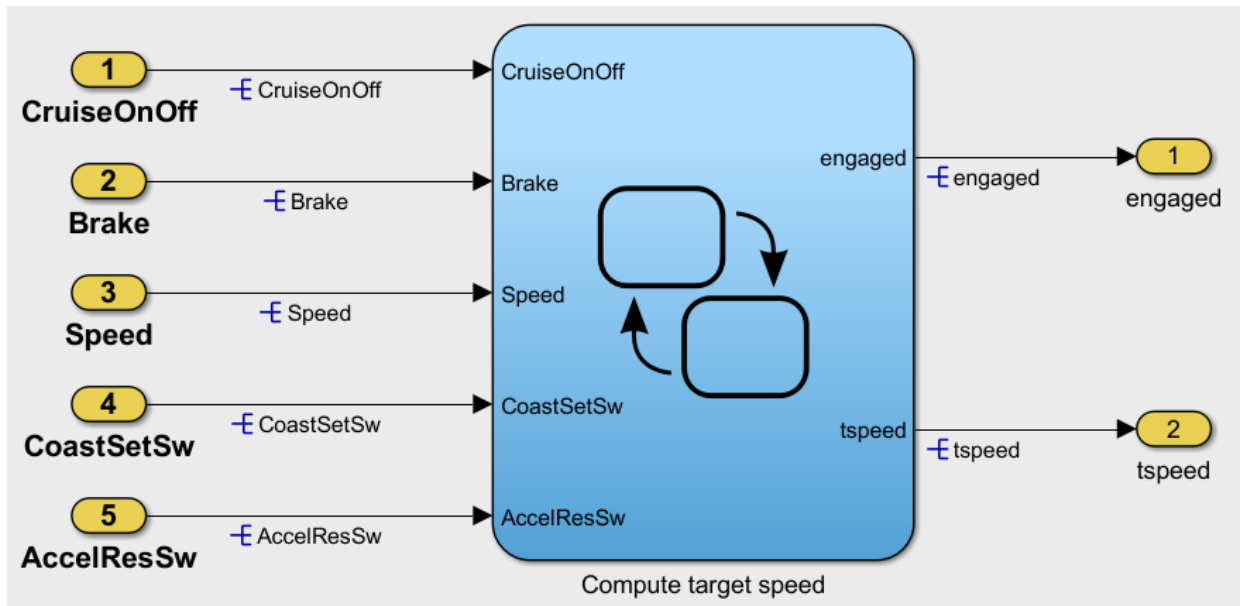
Run Code Generator Checks

Before you generate code from your model, there are steps that you can take to generate
code more compliant with MISRA C and more compatible with Polyspace. This example
shows how to use the Code Generation Advisor to check your model before generating
code.

1   Right-click Compute target speed and select **C/C++** > **Code Generation Advisor**.

2   Select the Code Generation Advisor folder. Add the `Polyspace` objective. The `MISRA`
    `C:2012 guidelines` objective is already selected.

Code Generation Objectives (System target file: ert.tlc)

Available objectives	Selected objectives - prioritized
Execution efficiency ROM efficiency RAM efficiency Traceability Safety precaution Debugging	MISRA C:2012 guidelines Polyspace

3   Click **Run Selected Checks**.

The Code Generation Advisor checks whether there are any blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this mode, the check for incompatible blocks passes, but there are some configuration settings that are incompatible with MISRA compliance and Polyspace checking.

⌄ 🗎 Code Generation Advisor
    ⚠ Check model configuration settings against code generation objectives
    ✓ Check for blocks not recommended for MISRA C:2012

4   Click on check that was not passed. Accept the parameter changes by selecting **Modify Parameters**.

5   Rerun the check by selecting **Run This Check**.

For your own model, you might not want to use all the recommended configuration settings. Using nonrecommended settings can generate less MISRA compliant code.
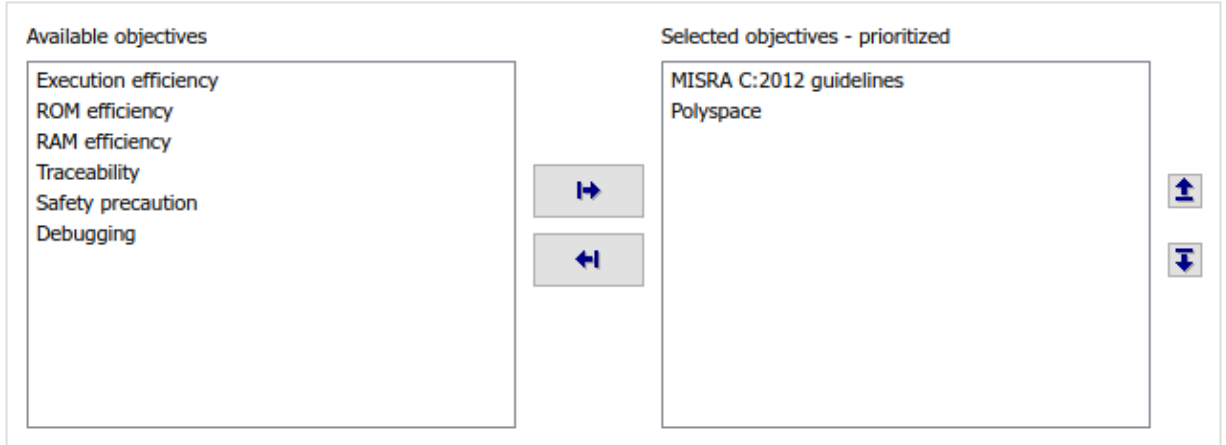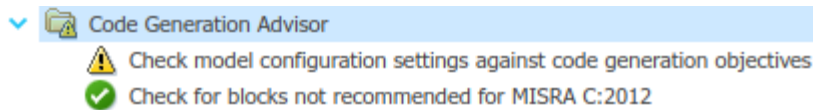
Run Model Advisor Checks

Before you generate code from your model, there are steps you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows you how to use the Model Advisor to check your model further before generating code.

For more checking before generating code, you can also run the Modeling Guidelines for MISRA C:2012.

1   At the bottom of the Code Generation Advisor window, select **Model Advisor**.

2   Under the **By Task** folder, select the **Modeling Guidelines for MISRA C:2012** advisor checks.
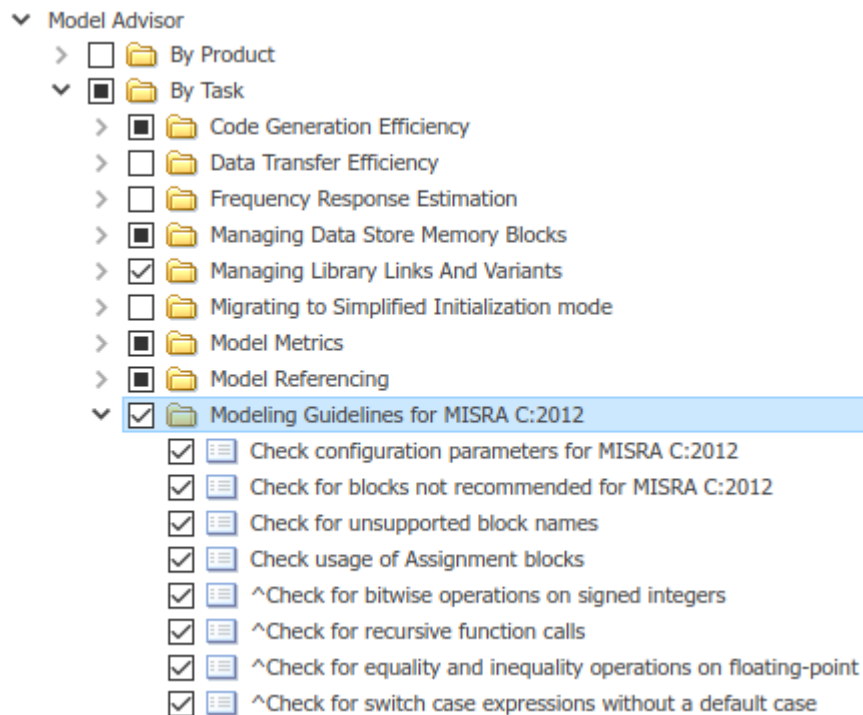
3   Click **Run Selected Checks** and review the results.

4   If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

For your own model, you might not want to use all the recommendations. Using nonrecommended settings or blocks can generate less MISRA compliant code.

Generate and Analyze Code

After you have done the model compliance checking, you can now generate code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

1   In the Simulink editor, right-click Compute target speed and select **C/C++** > **Build This Subsystem**.

2   Use the default settings for the tunable parameters and select **Build**.

3   After the code is generated, right-click Compute target speed and select **Polyspace** > **Options**.
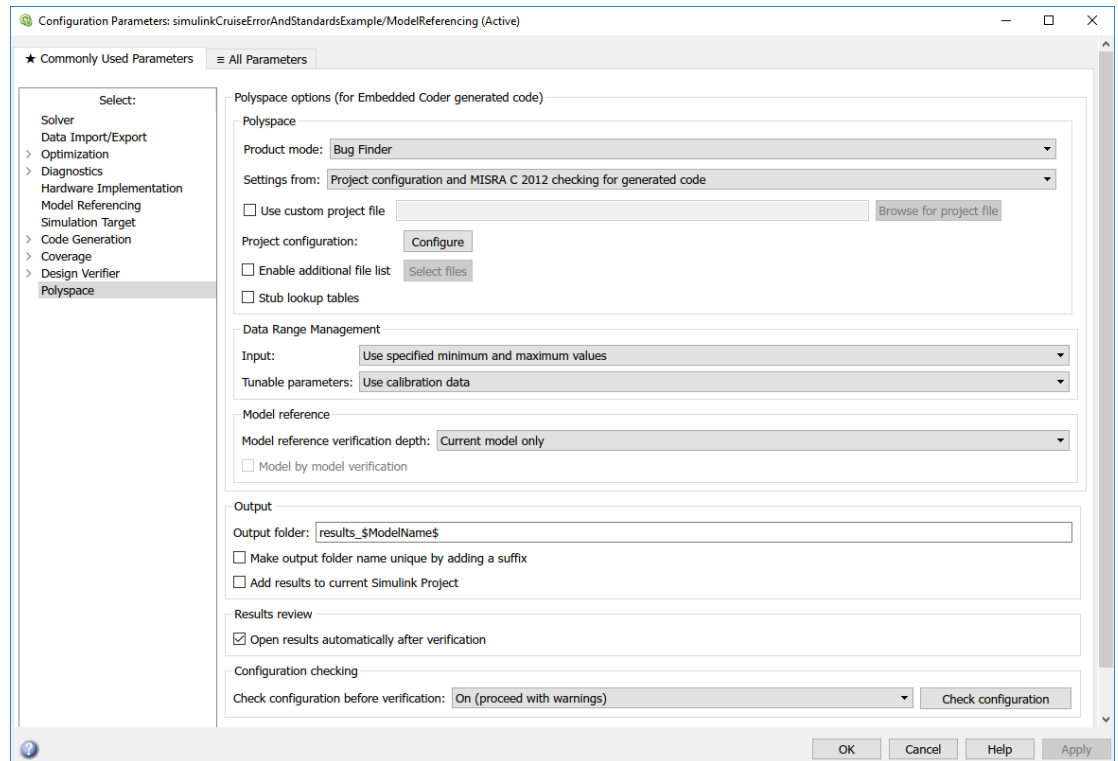
4   Click the **Configure** button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.
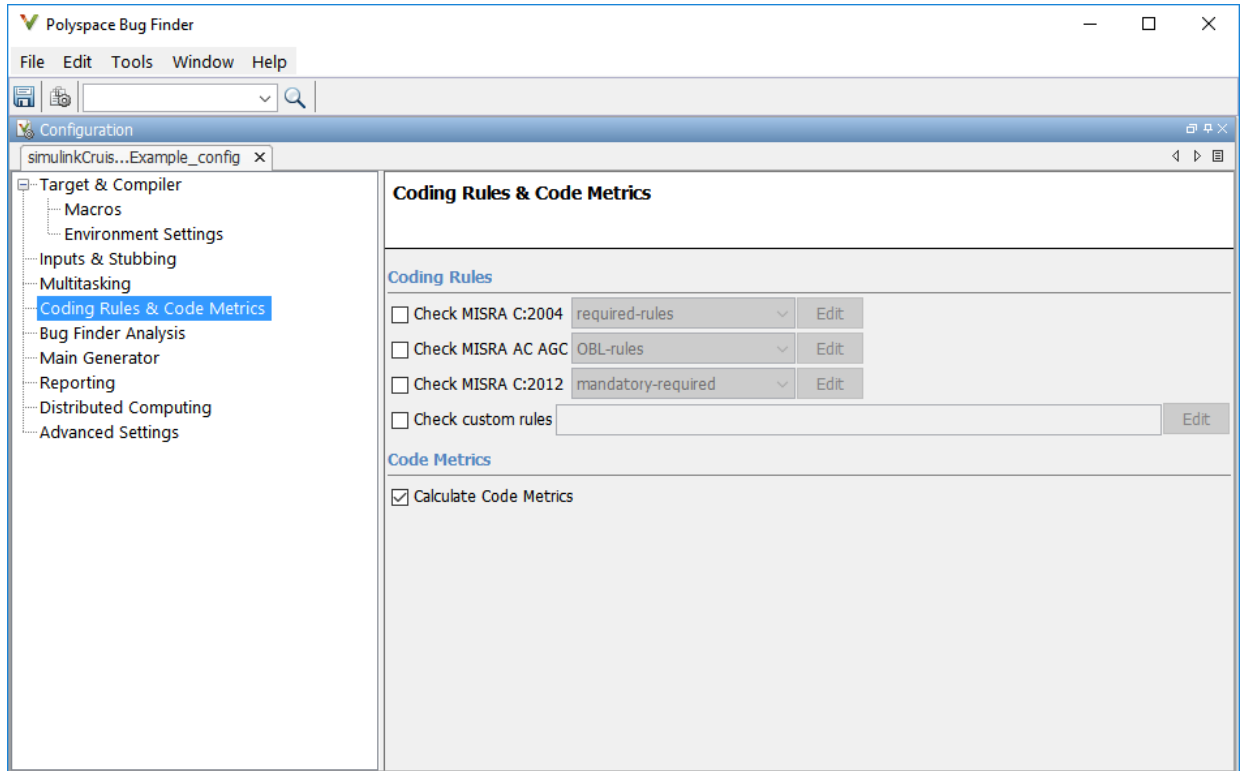
5   On the same pane, select **Calculate Code Metrics**. This option turns on code metric calculations for your generated code.

6   Save and close the Polyspace configuration window.

7   From your model, right-click Compute target speed and select **Polyspace** > **Verify Code Generated For** > **Selected Subsystem**.

Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

Review Results

After you run a Polyspace analysis of your generated code, the Polyspace environment shows you the results of the static code analysis. There are 50 MISRA C:2012 coding rule violations in your generated code.

1   Expand the tree for rule 8.7 and click through the different results.

Rule 8.7 states that functions and objects should not be global if the function or object is local. As you click through the 8.7 violations, you can see that these results refer to variables that other components also use, such as `CruiseOnOff`. You can annotate your code or your model to justify every result. But, because this model is a unit in a larger program, you can also change the configuration of the analysis to check only a subset of MISRA rules.
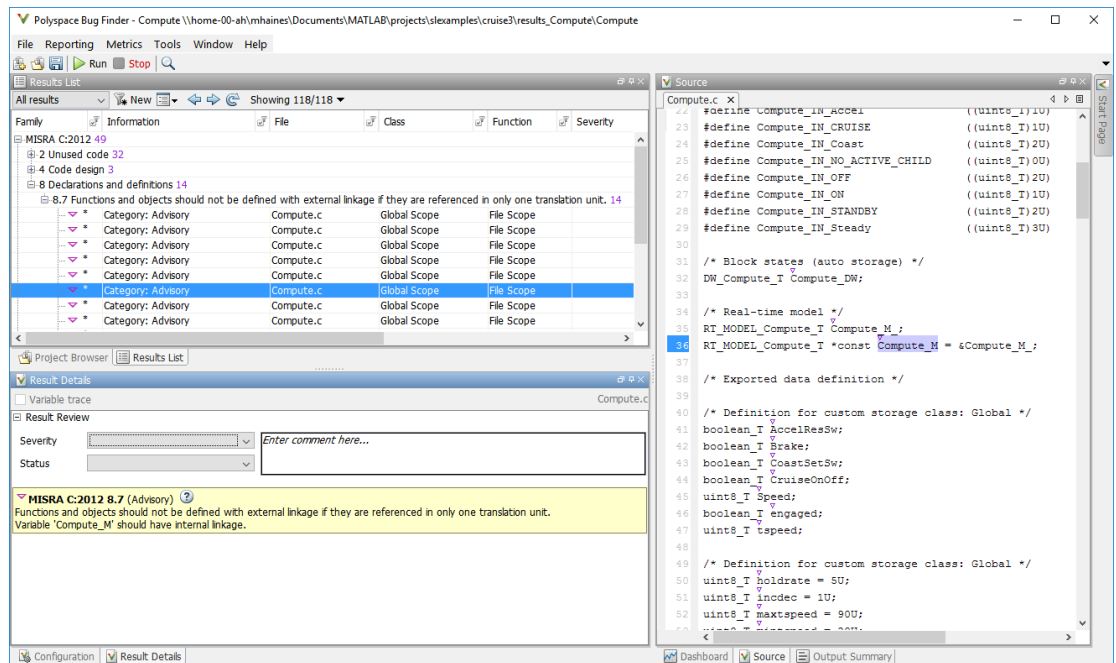
2   In your model, right-click Compute target speed and select **Polyspace** > **Options**.

3   Set the **Settings from** option to `Project configuration`. This option allow you to choose a subset of MISRA rules in the Polyspace configuration.

4   Click the **Configure** button.

5 In the Polyspace Configuration window, on the **Coding Rules & Code Metrics** pane, select the check box **Check MISRA C:2012** and from the drop-down list, select `single-unit-rules`. Now, Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.
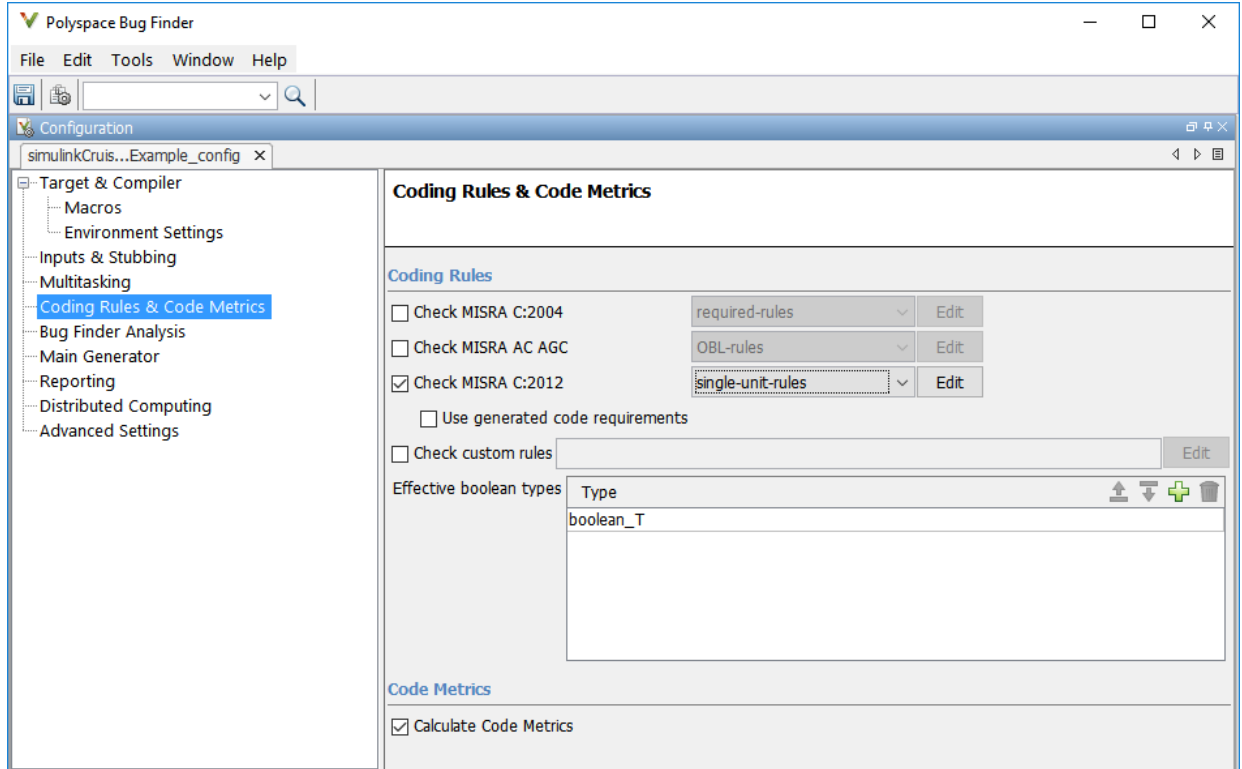
6 Save and close the Polyspace configuration window.

7 Rerun the analysis with the new configuration.

When the Polyspace environment reopens, there are no MISRA results, only code metric results. The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, no violations were found.

When this model is integrated with its parent model, you can add the rest of the MISRA C:2012 rules.

Generate Report

To demonstrate compliance with MISRA C:2012 and report on your generated code metrics, you must export your results. This section shows you how to generate a report after the analysis. If you want to generate a report every time you run an analysis, see Generate report.

1. If they are not open already, open your results in the Polyspace environment.
2. From the toolbar, select **Reporting** > **Run Report**.
3. Select **BugFinderSummary** as your report type.
4. Click **Run Report**.

   The report is saved in the same folder as your results.
5. To open the report, select **Reporting** > **Open Report**.

Related Examples

- "Generate and Analyze Code"
- "Test Two Simulations for Equivalence"
- "Export Test Results and Generate Reports"

Module Verification and Testing Processor-in-the-Loop

Module Verification and Testing Processor-in-the-Loop Overview

Module verification involves testing and analyzing code at a system level, integrating generated code from your model, handwritten code, and legacy code. Module verification often includes generating code that executes on a target object, rather than the desktop environment. Analyze the code to resolve errors and evaluate key metrics. Test the integrated system using new requirements-based tests and system-level tests from your model. Collect coverage on these tests and add tests to meet coverage targets.
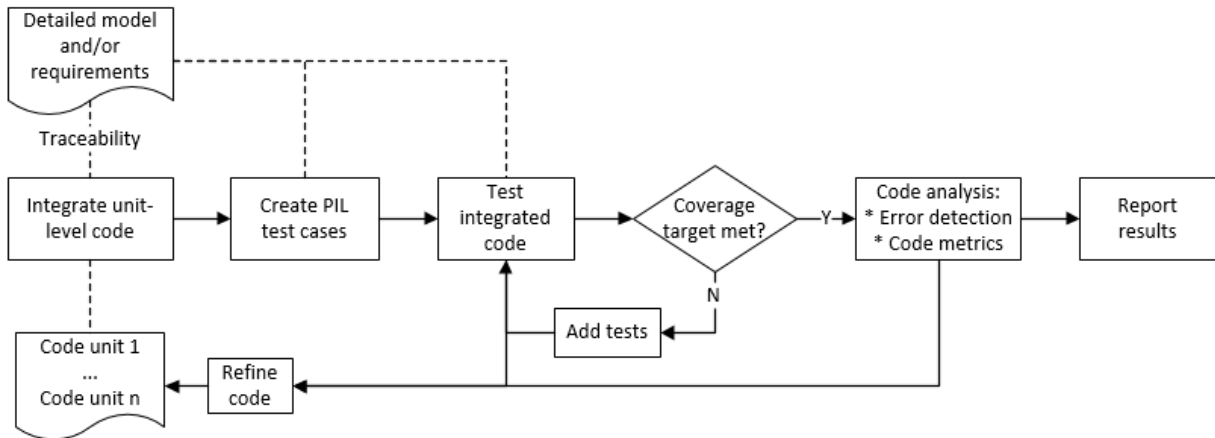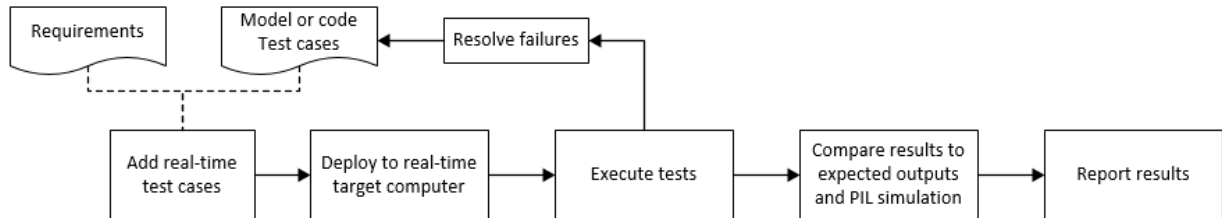

Related Examples

- "Test Two Simulations for Equivalence"
- "Generate and Analyze Code"

Test a Model in Real Time

Real-Time Testing and Testing Production Models Overview

Real-time testing assesses the system while including the effects of timers, physical signals, and target hardware. Sweep through parameter values on the target, verify system operation during execution, and verify expected results in the desktop environment. Systems that have been verified on target hardware often exist in a change-controlled state. You can test these systems without modifying them by using isolated simulation and analysis environments.

Related Examples

- "Create and Run Real-Time Application from Simulink Model"
- "Test Models in Real Time"
- "Assess Simulation Using Logical Statements"

abstraction   The process of ignoring certain aspects of model behavior that do not affect the test objective or property under investigation.

analysis model   The target model for a Simulink Design Verifier analysis. If you select an atomic subsystem for analysis, the analysis model is generated by extracting the subsystem to a new model.

assumption   A property that is assumed to be true during a property proof. The proof result holds only when the assumption is true.

block replacement rule   A rule that is registered with Simulink Design Verifier and defines how instances of specific blocks are replaced by an alternate implementation. The software uses MATLAB commands to define when and how to apply a block replacement rule (see "Define Custom Block Replacements" on page 4-10).

component verification   The process of verifying an individual components in a model. You can verify a component within the execution context of the model, or independently of its parent model.

condition coverage   Measures the percentage of the total number of logic conditions associated with logical model objects that the simulation actually exercised. Enabling condition coverage causes every decision and condition coverage outcome to be enabled. See "Types of Model Coverage" in the Simulink Verification and Validation documentation.

constraint   A property that is forced to be true during test case generation.

counterexample   A test case that demonstrates a property violation.

coverage objective   A test objective that defines when a coverage point results in a particular outcome.

coverage point   A decision, condition, or MCDC expression associated with a model object. Each coverage point has a fixed number of mutually exclusive outcomes.

decision coverage

Measures the percentage of the total number of simulation paths through model objects that the simulation actually traversed. Decision coverage is a subset of modified decision/condition coverage. See "Types of Model Coverage" in the Simulink Verification and Validation documentation.

floating-point approximation

The process of approximating floating-point numbers using rational numbers (i.e., fractions whose numerator and denominator are small integers). The Simulink Design Verifier software performs floating-point approximations during its analysis. It can generate invalid test cases that result from numerical differences. For example, given a large enough floating-point number x, the expression $x==(x+1)$ can be true; however, this expression never holds if x is a rational number.

invalid test case

A test case that does not satisfy its objectives.

modified condition/decision coverage (MCDC)

Measures the independence of logical block inputs and transition conditions associated with logical model objects during the simulation. When you set the coverage objective to MCDC, Simulink Design Verifier automatically enables every coverage objective for decision coverage and condition coverage as well.

Note that MCDC test cases are not generated for XOR configured logic operators. You can achieve MCDC by using the same tests that would be generated from AND configured blocks or OR configured blocks.

See "Types of Model Coverage" in the Simulink Verification and Validation documentation.

nonlinear arithmetic

A computation in the model that cannot be expressed as a combination of mutually exclusive linear expressions. Nonlinear arithmetic can affect a property or test objective, and it can cause the analysis to return an error. In this case, you should apply simplifying approximations and abstractions.

property
A logical expression of the signals and data values, within a model, that is intended to be proven true during simulation. Properties evaluate at specific points in the model.

property violation
The condition during a simulation when a property is false.

test case
A sequence of numeric values and input data time that you input to a model during its simulation.

test harness
A model that runs test cases on an analysis model.

test objective
A logical expression of the signals and data values, within a model, that is intended to be true at least once in the resulting test case during simulation. Test objectives evaluate at specific points in the model.

Test Objective block
The block that you add to a model to define test objectives. In the block mask, define test objectives as values or ranges that an input signal must satisfy during a test case.

unsatisfiable test objective
The status of a test objective that indicates a test case cannot be generated for the specified approximations. This includes floating-point approximations and maximum-step limitations specified in the **Design Verifier > Test Generation** pane of the Configuration Parameters dialog box.

validated property
The status of a property that indicates no counterexample exists, subject to floating-point approximations and the settings specified in the **Property Proving** pane of the Configuration Parameters dialog box.