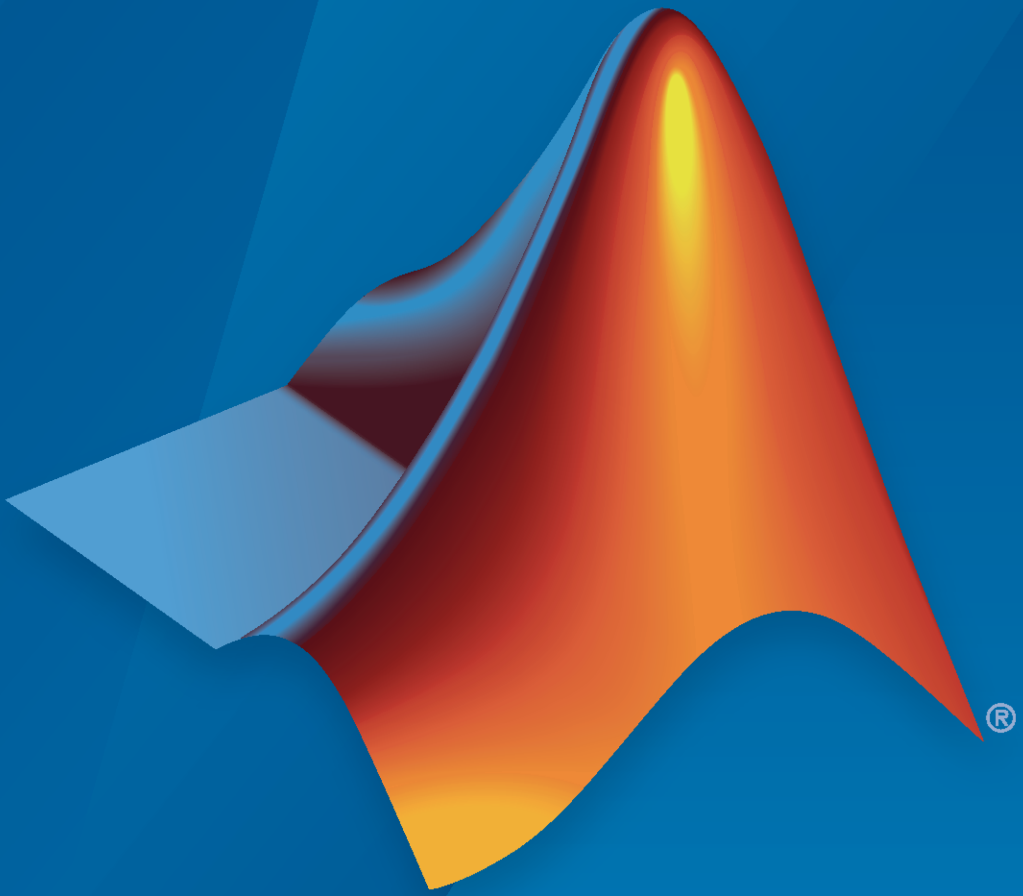
Simulink® Design Verifier™

User's Guide



MATLAB®&SIMULINK®

R2020a

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Design Verifier™ User's Guide

Revision History

May 2007	Online only	New for Version 1.0 (Release 2007a+)
September 2007	Online only	Revised for Version 1.1 (Release 2007b)
March 2008	Online only	Revised for Version 1.2 (Release 2008a)
October 2008	Online only	Revised for Version 1.3 (Release 2008b)
March 2009	Online only	Revised for Version 1.4 (Release 2009a)
September 2009	Online only	Revised for Version 1.5 (Release 2009b)
March 2010	Online only	Revised for Version 1.6 (Release 2010a)
September 2010	Online only	Revised for Version 1.7 (Release 2010b)
April 2011	Online only	Revised for Version 2.0 (Release 2011a)
September 2011	Online only	Revised for Version 2.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.2 (Release 2012a)
September 2012	Online only	Revised for Version 2.3 (Release 2012b)
March 2013	Online only	Revised for Version 2.4 (Release 2013a)
September 2013	Online only	Revised for Version 2.5 (Release 2013b)
March 2014	Online only	Revised for Version 2.6 (Release 2014a)
October 2014	Online only	Revised for Version 2.7 (Release 2014b)
March 2015	Online only	Revised for Version 2.8 (Release 2015a)
September 2015	Online only	Revised for Version 3.0 (Release 2015b)
October 2015	Online only	Rereleased for Version 2.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.1 (Release 2016a)
September 2016	Online only	Revised for Version 3.2 (Release 2016b)
March 2017	Online only	Revised for Version 3.3 (Release 2017a)
September 2017	Online only	Revised for Version 3.4 (Release 2017b)
March 2018	Online only	Revised for Version 3.5 (Release 2018a)
September 2018	Online only	Revised for Version 4.0 (Release 2018b)
March 2019	Online only	Revised for Version 4.1 (Release 2019a)
September 2019	Online only	Revised for Version 4.2 (Release 2019b)
March 2020	Online only	Revised for Version 4.3 (Release 2020a)

Contents

6 Detecting Design Errors

13 Reviewing the Results

14 Analyzing Large Models and Improving Performance

15 Simulink Design Verifier Configuration Parameters

16 Verification and Validation

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 design errors, prove requirements compliance, and generate tests

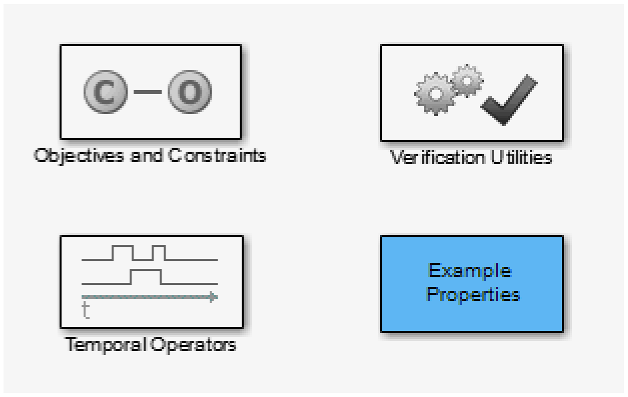
Simulink Design Verifier uses formal methods to identify hidden design errors in models. It detects blocks in the model that result in integer overflow, dead logic, array access violations, and division by zero. It can formally verify that the design meets functional requirements. For each design error or requirements violation, it generates a simulation test case for debugging.

Simulink Design Verifier generates test cases for model coverage and custom objectives to extend existing requirements-based test cases. These test cases drive your model to satisfy condition, decision, modified condition/decision (MCDC), and custom coverage objectives. In addition to coverage objectives, you can specify custom test objectives to automatically generate requirements-based test cases.

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

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

In this section...

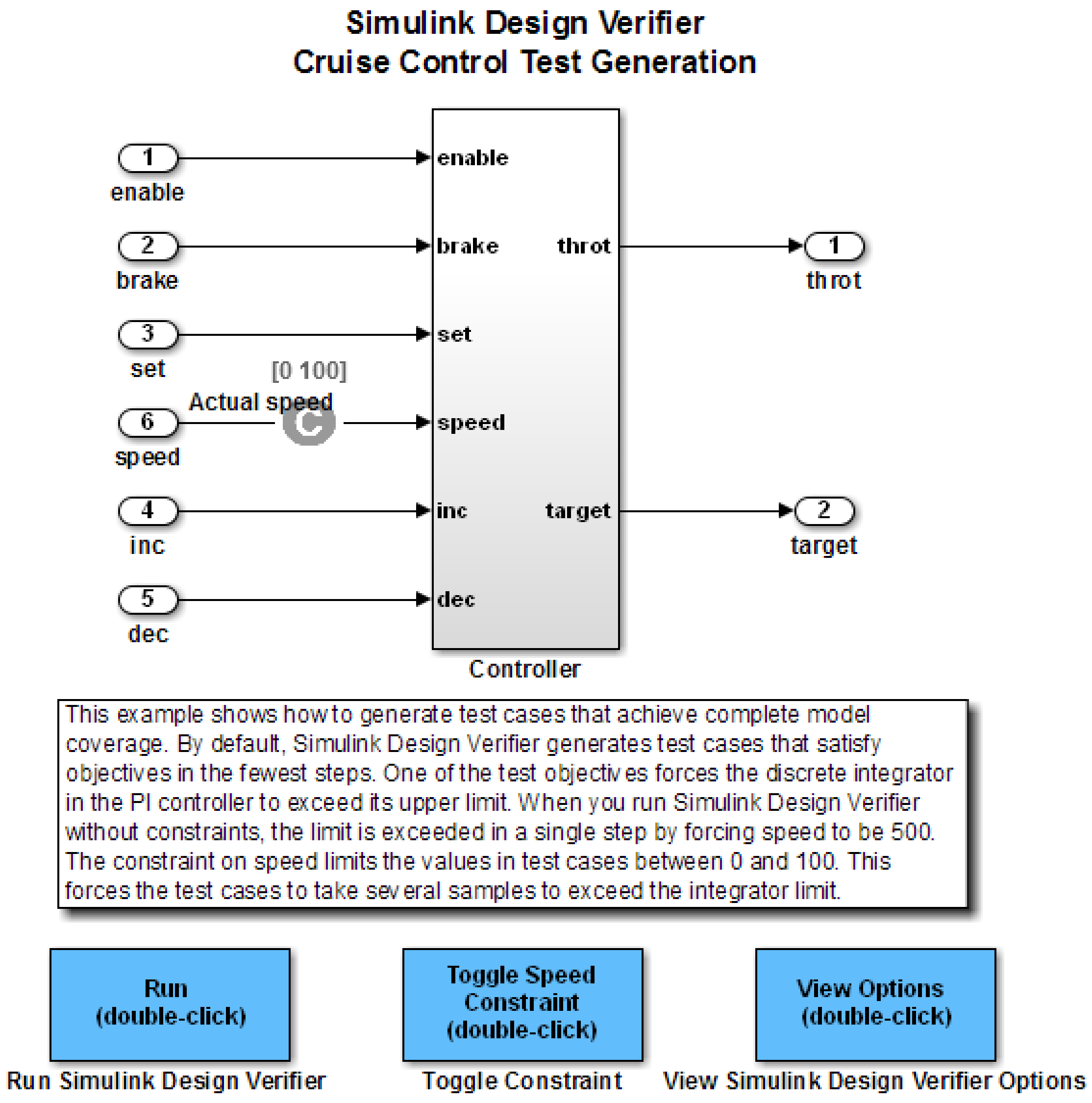
About This Example

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
```



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.

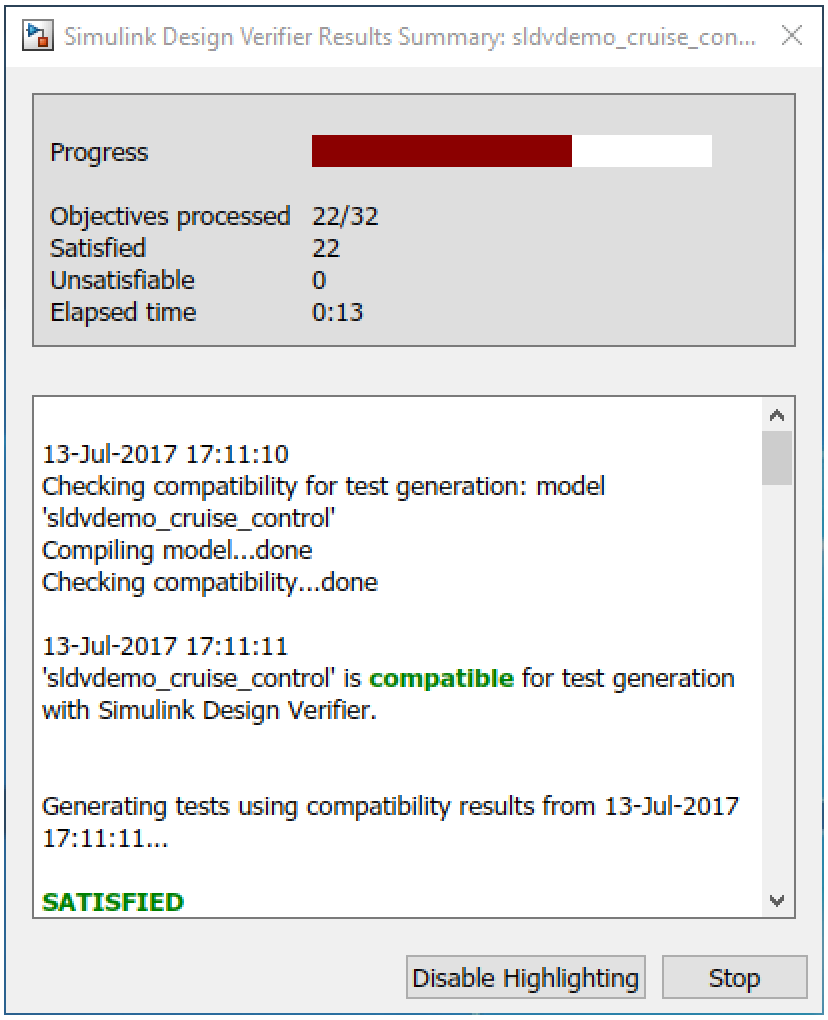
Generate Test Cases

- “Run Analysis” on page 1-5
- “Generate Analysis Results” on page 1-6
- “Highlight Analysis Results on Model” on page 1-7
- “Generate Detailed Analysis Report” on page 1-9
- “Create Harness Model” on page 1-13
- “Simulate Tests and Produce Model Coverage Report” on page 1-17

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.


Simulink Design Verifier Results Summary: sldvdemo_cruise_con...
Progress
Disable Highlighting
Stop

Objectives processed 22/32
Satisfied 22
Unsatisfiable 0
Elapsed time 0:13

```
13-Jul-2017 17:11:10
Checking compatibility for test generation: model
'sldvdemo_cruise_control'
Compiling model...done
Checking compatibility...done

13-Jul-2017 17:11:11
'sldvdemo_cruise_control' is compatible for test generation
with Simulink Design Verifier.


Generating tests using compatibility results from 13-Jul-2017
17:11:11...

SATISFIED
```

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 13-7.

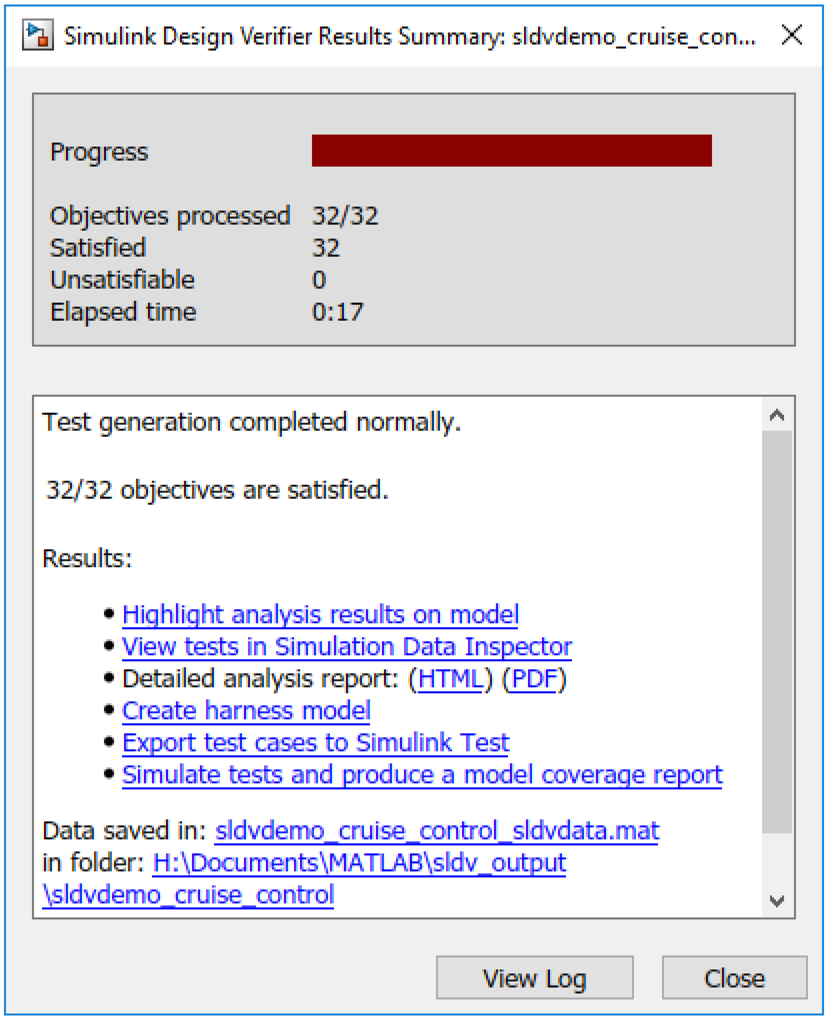
Generate Analysis Results

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.



Simulink Design Verifier Results Summary: sldvdemo_cruise_con...

```
Progress

Objectives processed  32/32
Satisfied             32
Unsatisfiable         0
Elapsed time          0:17
```

Test generation completed normally.

32/32 objectives are satisfied.

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: sldvdemo_cruise_control_sldvdata.mat
in folder: H:\Documents\MATLAB\sldv_output
\sldvdemo_cruise_control

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:

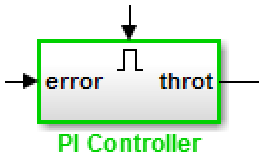
- “Green: Objectives Satisfied” on page 1-8
- “Orange: Objectives Undecided” on page 1-8
- “Red: Objectives Unsatisfiable” on page 1-9

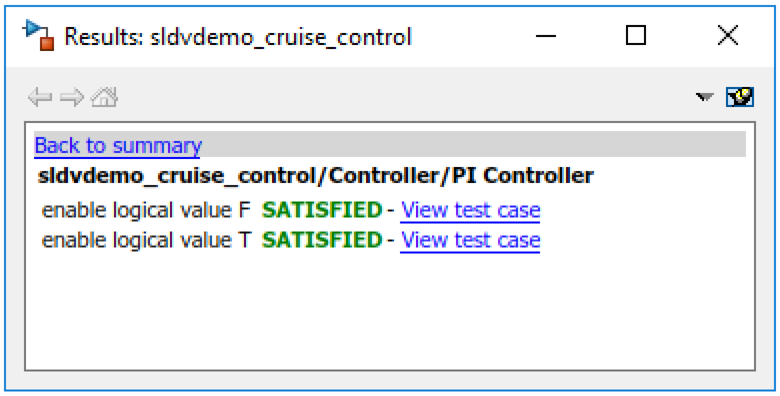
When you highlight the analysis results on a model, the Simulink Design Verifier Results Inspector opens. When you click an object in the model that has analysis results, the Results Inspector displays the results summary for that object.

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.





sldvdemo_cruise_control/Controller/PI Controller

enable logical value F **SATISFIED** - View test case

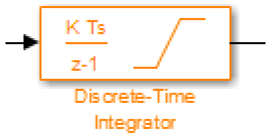
enable logical value T **SATISFIED** - View test case

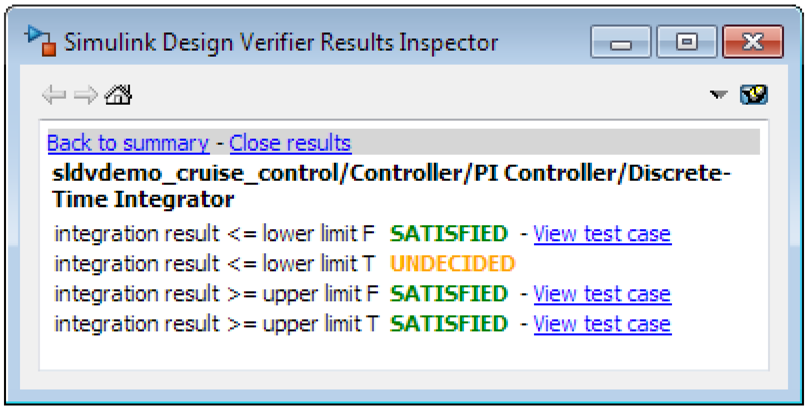
Orange: Objectives Undecided

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.




Simulink Design Verifier Results Inspector

Back to summary - Close results

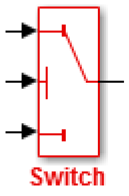
sldvdemo_cruise_control/Controller/PI Controller/Discrete-Time Integrator

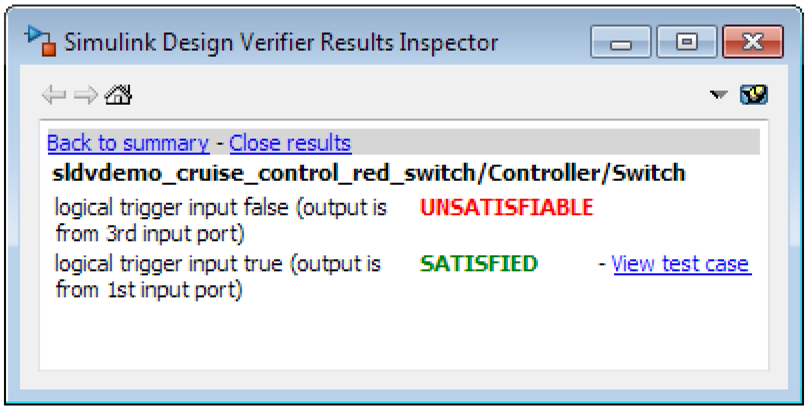
integration result <= lower limit F	SATISFIED	- View test case
integration result <= lower limit T	UNDECIDED	
integration result >= upper limit F	SATISFIED	- View test case
integration result >= upper limit T	SATISFIED	- View test case

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.




Simulink Design Verifier Results Inspector

Back to summary - Close results

sldvdemo_cruise_control_red_switch/Controller/Switch

logical trigger input false (output is from 3rd input port)	UNSATISFIABLE	
logical trigger input true (output is from 1st input port)	SATISFIED	- View test case

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.

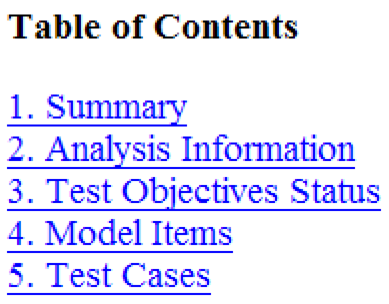


Table of Contents

1. Summary
2. Analysis Information
3. Test Objectives Status
4. Model Items
5. Test Cases

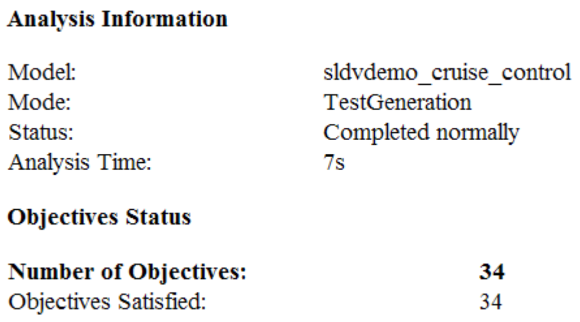
For a description of each report chapter, see:

- "Summary" on page 1-10
- "Analysis Information" on page 1-10
- "Test Objectives Status" on page 1-11
- "Model Items" on page 1-12
- "Test Cases" on page 1-13

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



Analysis Information

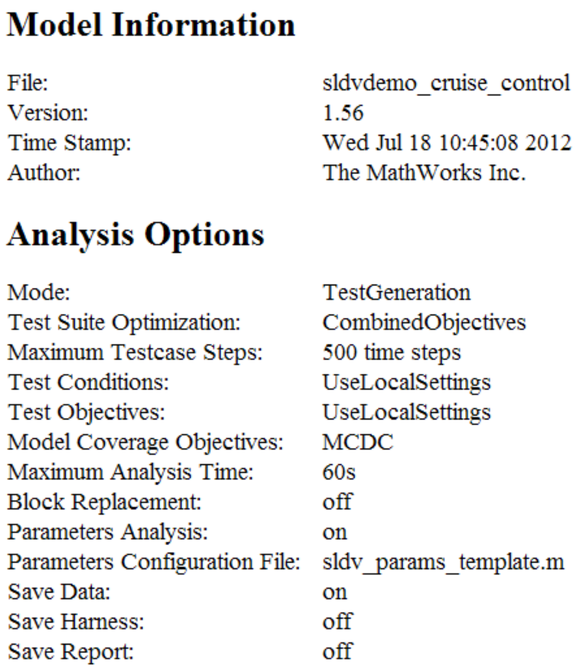
Model:	sldvdemo_cruise_control
Mode:	TestGeneration
Status:	Completed normally
Analysis Time:	7s

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.



Model Information

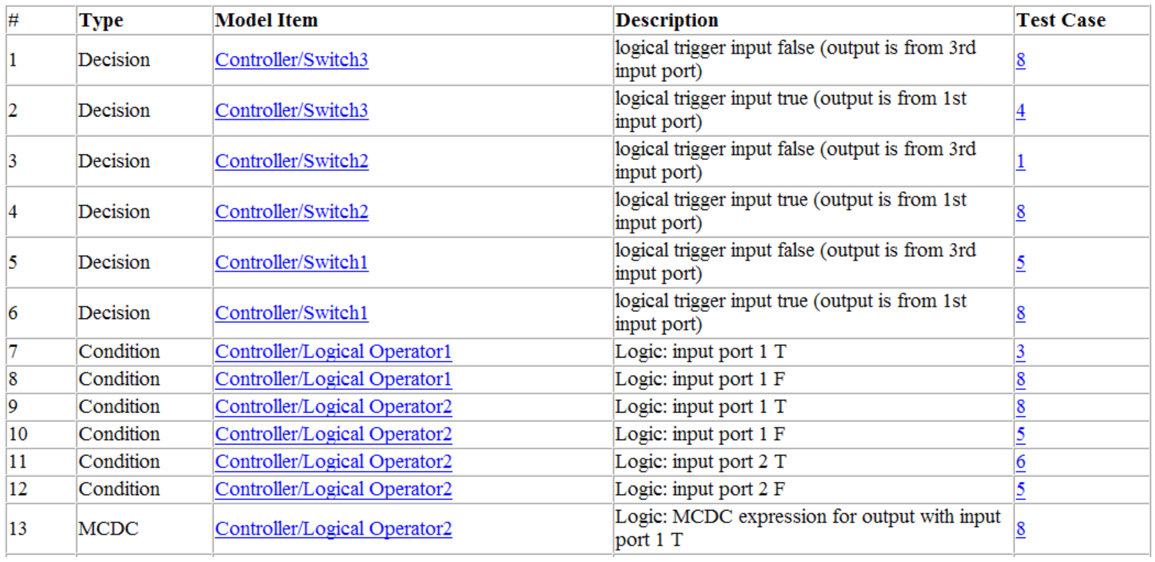
File:	sldvdemo_cruise_control
Version:	1.56
Time Stamp:	Wed Jul 18 10:45:08 2012
Author:	The MathWorks Inc.

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.



#	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

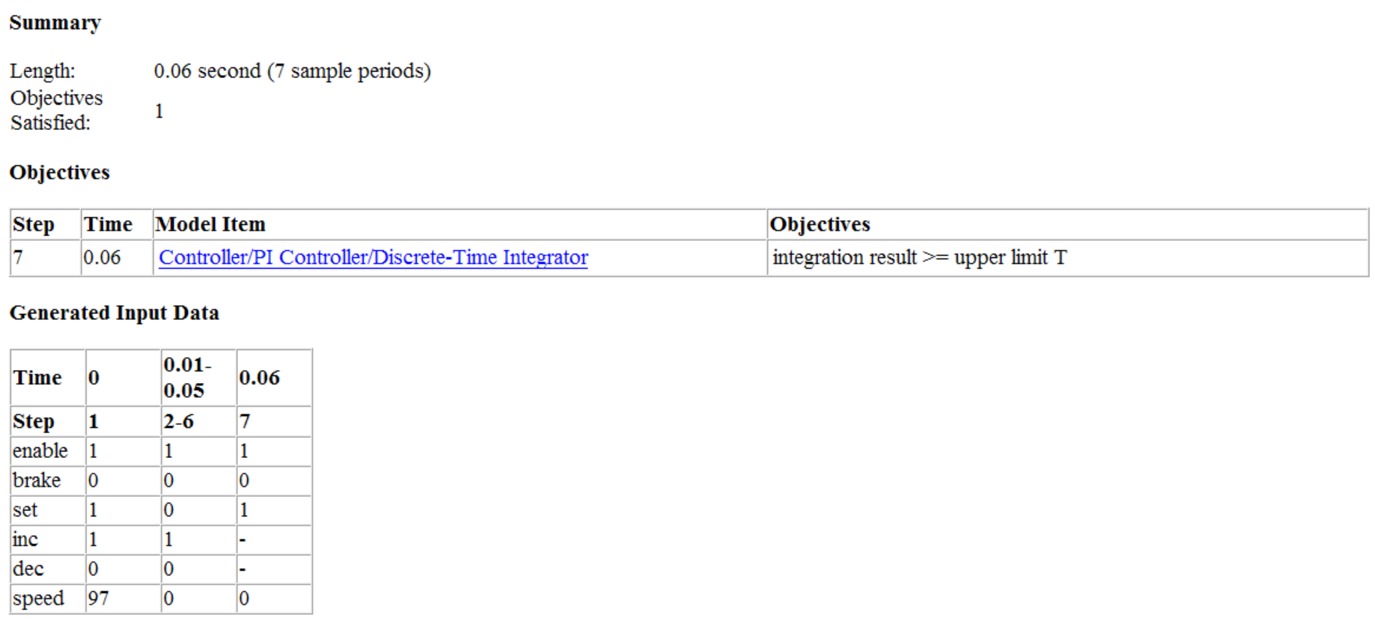
Objectives Status

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.



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

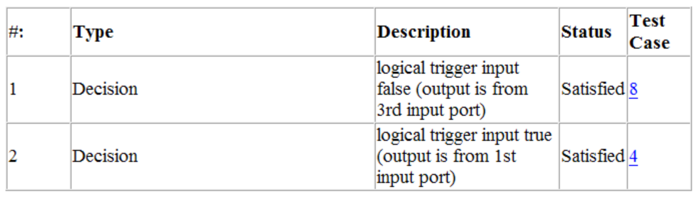
Time	0	0.01-0.05	0.06
Step	**1**	**2-6**	**7**
enable	1	1	1
brake	0	0	0
set	1	0	1
inc	1	1	-
dec	0	0	-
speed	97	0	0

Test Case 7

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.

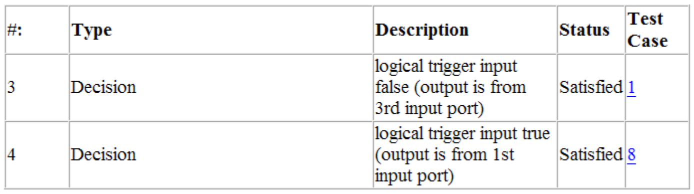
Model Items

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.



#:	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

Model Items - Controller/Switch3



#:	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

Model Items - Controller/Switch2

Test Cases

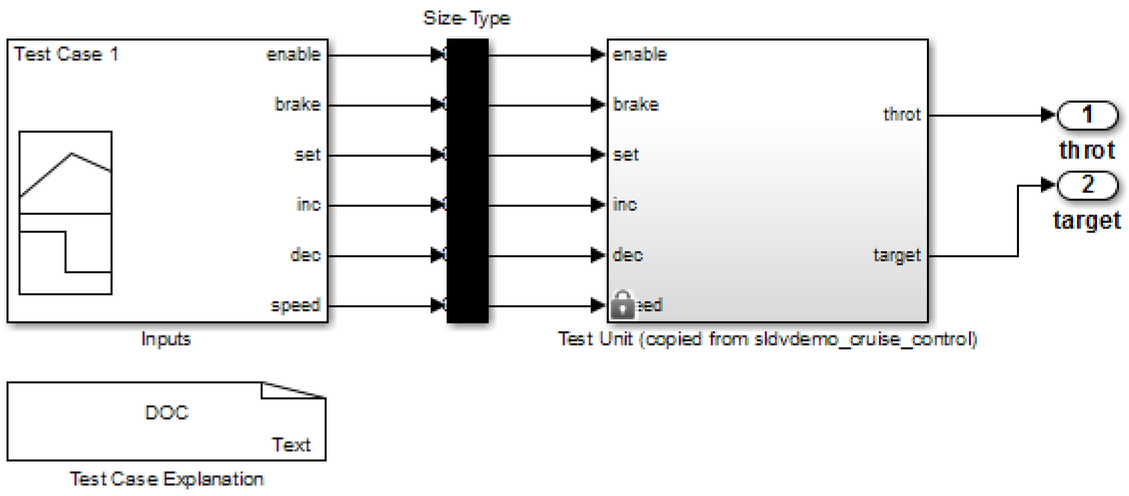
In the **Table of Contents**, click **Test Cases** to display detailed information about each generated test case, including:

- 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 1-11.

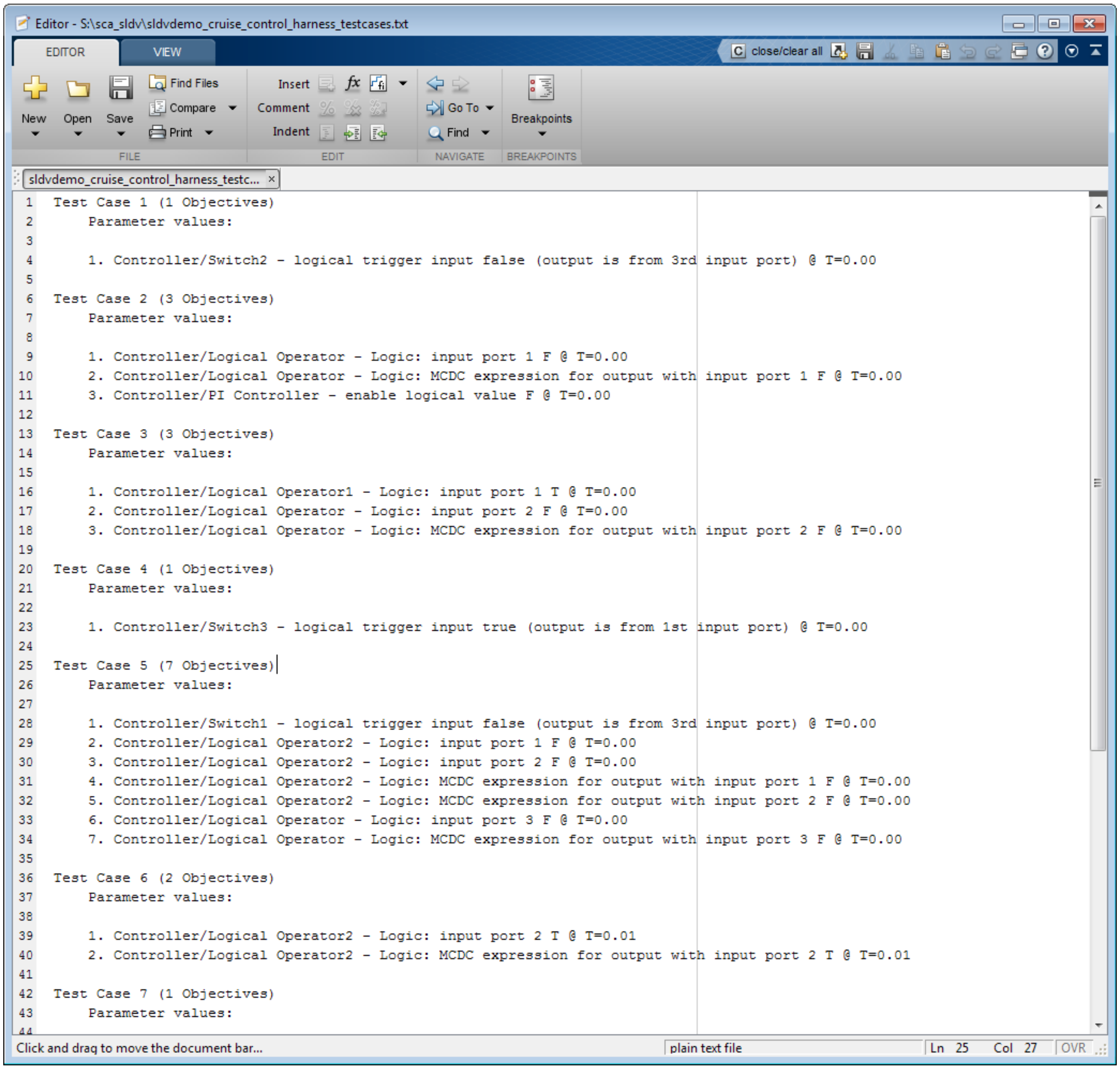
Create Harness Model

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.



```
Test Case 1 (1 Objectives)
    Parameter values:

    1. Controller/Switch2 - logical trigger input false (output is from 3rd input port) @ T=0.00

Test Case 2 (3 Objectives)
    Parameter values:

    1. Controller/Logical Operator - Logic: input port 1 F @ T=0.00
    2. Controller/Logical Operator - Logic: MCDC expression for output with input port 1 F @ T=0.00
    3. Controller/PI Controller - enable logical value F @ T=0.00

Test Case 3 (3 Objectives)
    Parameter values:

    1. Controller/Logical Operator1 - Logic: input port 1 T @ T=0.00
    2. Controller/Logical Operator - Logic: input port 2 F @ T=0.00
    3. Controller/Logical Operator - Logic: MCDC expression for output with input port 2 F @ T=0.00

Test Case 4 (1 Objectives)
    Parameter values:

    1. Controller/Switch3 - logical trigger input true (output is from 1st input port) @ T=0.00

Test Case 5 (7 Objectives)
    Parameter values:

    1. Controller/Switch1 - logical trigger input false (output is from 3rd input port) @ T=0.00
    2. Controller/Logical Operator2 - Logic: input port 1 F @ T=0.00
    3. Controller/Logical Operator2 - Logic: input port 2 F @ T=0.00
    4. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 1 F @ T=0.00
    5. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 2 F @ T=0.00
    6. Controller/Logical Operator - Logic: input port 3 F @ T=0.00
    7. Controller/Logical Operator - Logic: MCDC expression for output with input port 3 F @ T=0.00

Test Case 6 (2 Objectives)
    Parameter values:

    1. Controller/Logical Operator2 - Logic: input port 2 T @ T=0.01
    2. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 2 T @ T=0.01

Test Case 7 (1 Objectives)
    Parameter values:
```

- 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**.

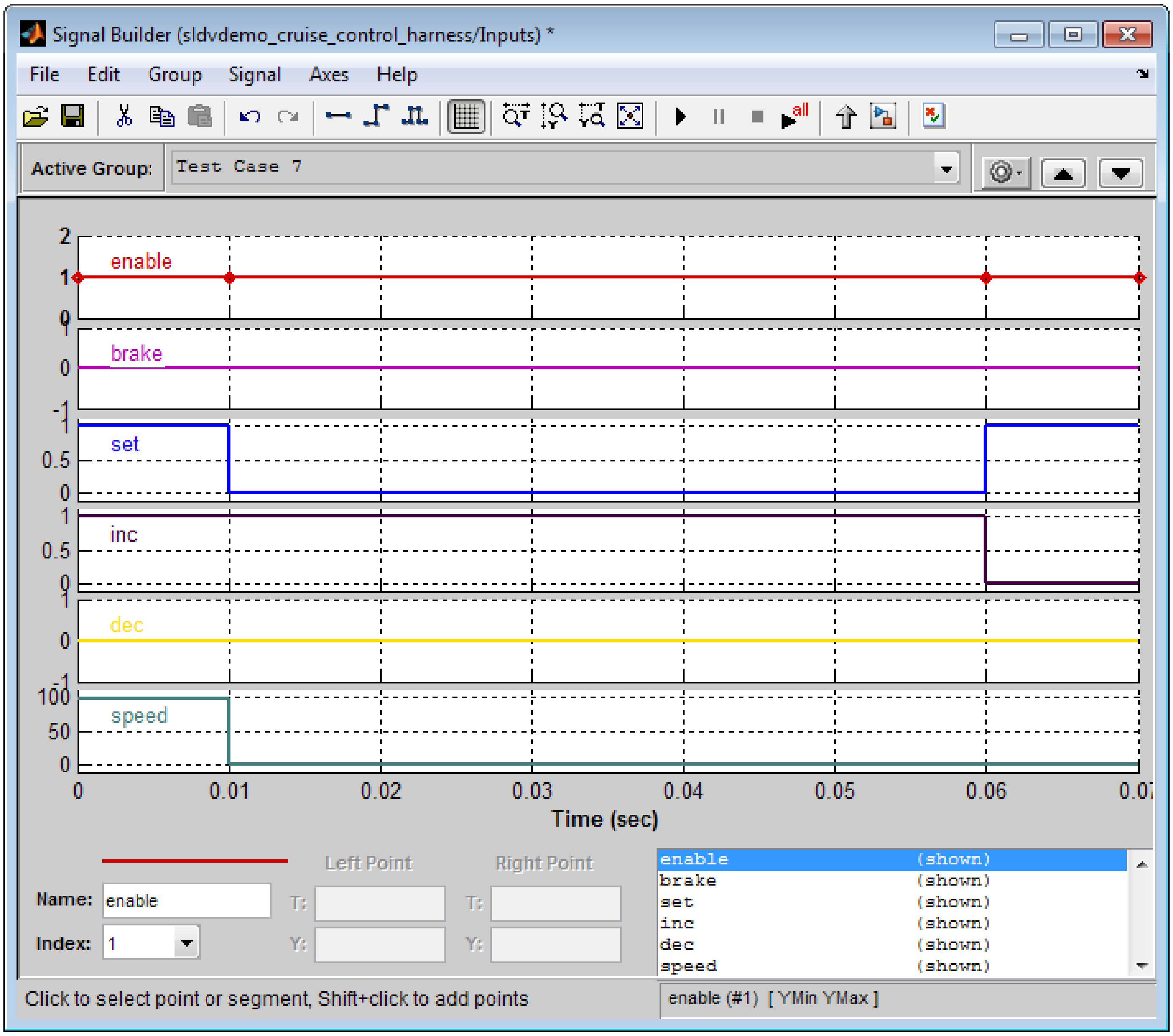
- The Inputs block is a Signal Builder block that contains the generated test case signals. Double-click the Inputs block to open the Signal Builder dialog box and view the eight test case signals.
- The Size-Type block is a subsystem that transmits signals from the Inputs block to the Test Unit block. 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:

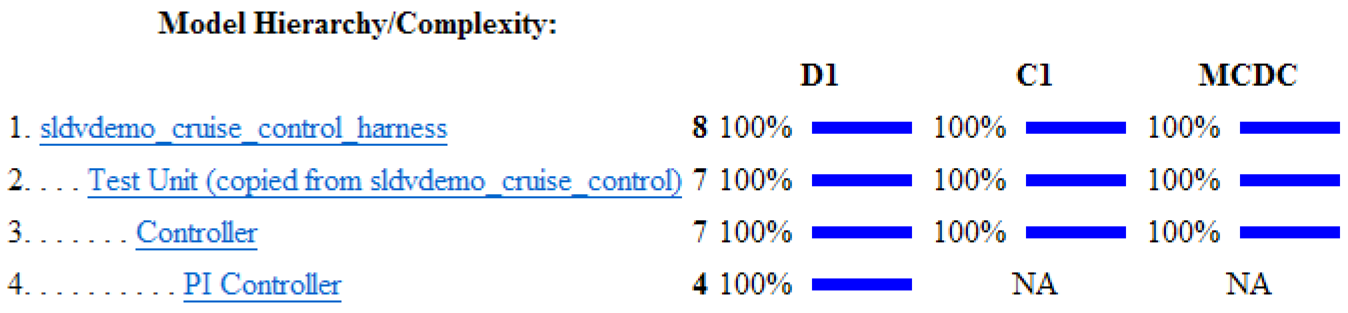
   - The enable and inc signals remain 1.
   - The brake and dec signals remain 0.
   - The set signal transitions from 1 to 0.
   - The speed signal transitions from 100 to 0.


Signal Builder (sldvdemo_cruise_control_harness/Inputs) *
File Edit Group Signal Axes Help
Active Group: Test Case 7
enable
brake
set
inc
dec
speed
Time (sec)
Left Point
Right Point
Name: enable
Index: 1
T:
Y:
enable (shown)
brake (shown)
set (shown)
inc (shown)
dec (shown)
speed (shown)
Click to select point or segment, Shift+click to add points
enable (#1) [YMin YMax]

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 Coverage™ 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.



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

Summary

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.

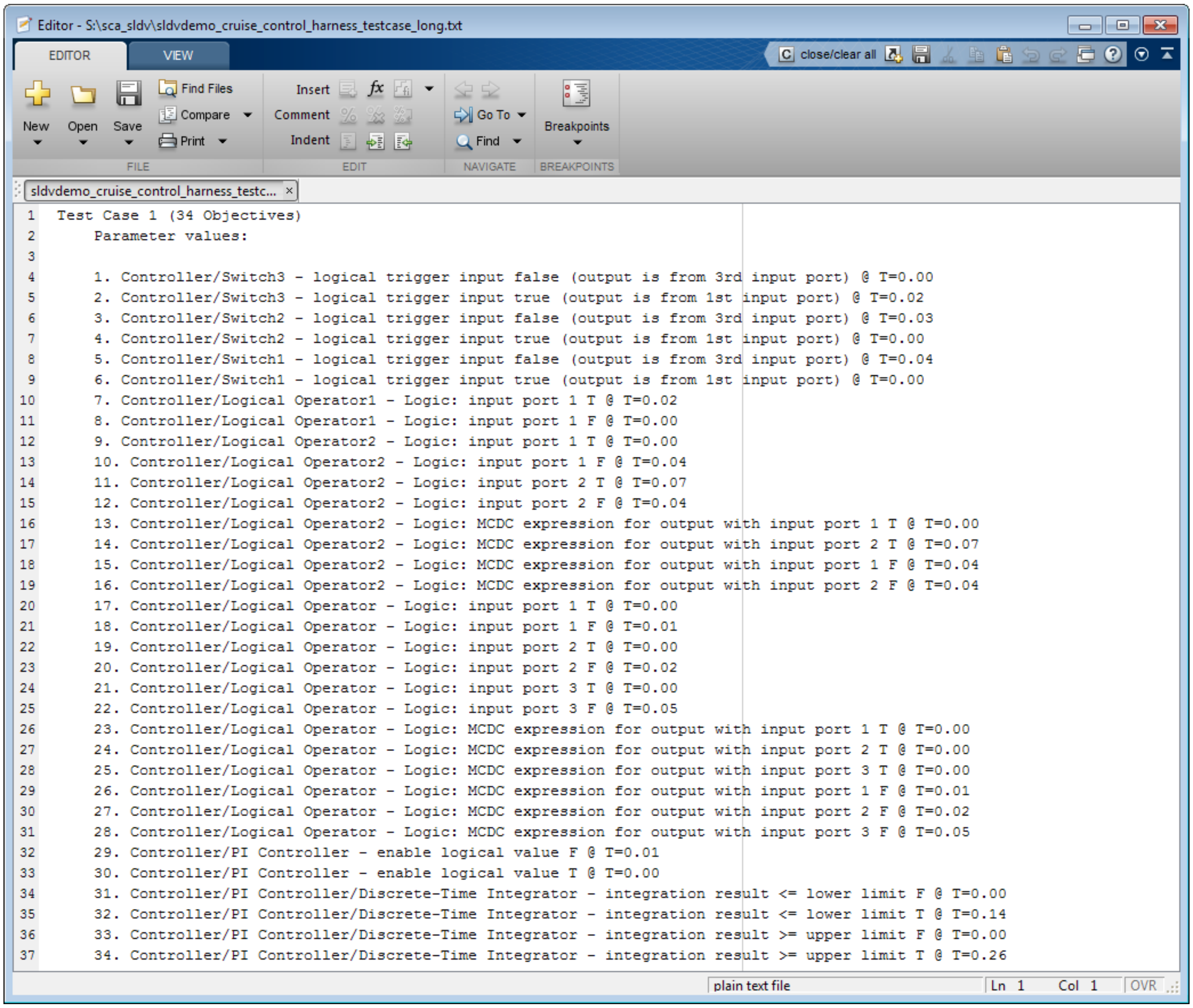
Combine Test Cases

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.

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

1. On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
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.



```
Test Case 1 (34 Objectives)
    Parameter values:

    1. Controller/Switch3 - logical trigger input false (output is from 3rd input port) @ T=0.00
    2. Controller/Switch3 - logical trigger input true (output is from 1st input port) @ T=0.02
    3. Controller/Switch2 - logical trigger input false (output is from 3rd input port) @ T=0.03
    4. Controller/Switch2 - logical trigger input true (output is from 1st input port) @ T=0.00
    5. Controller/Switch1 - logical trigger input false (output is from 3rd input port) @ T=0.04
    6. Controller/Switch1 - logical trigger input true (output is from 1st input port) @ T=0.00
    7. Controller/Logical Operator1 - Logic: input port 1 T @ T=0.02
    8. Controller/Logical Operator1 - Logic: input port 1 F @ T=0.00
    9. Controller/Logical Operator2 - Logic: input port 1 T @ T=0.00
    10. Controller/Logical Operator2 - Logic: input port 1 F @ T=0.04
    11. Controller/Logical Operator2 - Logic: input port 2 T @ T=0.07
    12. Controller/Logical Operator2 - Logic: input port 2 F @ T=0.04
    13. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 1 T @ T=0.00
    14. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 2 T @ T=0.07
    15. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 1 F @ T=0.04
    16. Controller/Logical Operator2 - Logic: MCDC expression for output with input port 2 F @ T=0.04
    17. Controller/Logical Operator - Logic: input port 1 T @ T=0.00
    18. Controller/Logical Operator - Logic: input port 1 F @ T=0.01
    19. Controller/Logical Operator - Logic: input port 2 T @ T=0.00
    20. Controller/Logical Operator - Logic: input port 2 F @ T=0.02
    21. Controller/Logical Operator - Logic: input port 3 T @ T=0.00
    22. Controller/Logical Operator - Logic: input port 3 F @ T=0.05
    23. Controller/Logical Operator - Logic: MCDC expression for output with input port 1 T @ T=0.00
    24. Controller/Logical Operator - Logic: MCDC expression for output with input port 2 T @ T=0.00
    25. Controller/Logical Operator - Logic: MCDC expression for output with input port 3 T @ T=0.00
    26. Controller/Logical Operator - Logic: MCDC expression for output with input port 1 F @ T=0.01
    27. Controller/Logical Operator - Logic: MCDC expression for output with input port 2 F @ T=0.02
    28. Controller/Logical Operator - Logic: MCDC expression for output with input port 3 F @ T=0.05
    29. Controller/PI Controller - enable logical value F @ T=0.01
    30. Controller/PI Controller - enable logical value T @ T=0.00
    31. Controller/PI Controller/Discrete-Time Integrator - integration result <= lower limit F @ T=0.00
    32. Controller/PI Controller/Discrete-Time Integrator - integration result <= lower limit T @ T=0.14
    33. Controller/PI Controller/Discrete-Time Integrator - integration result >= upper limit F @ T=0.00
    34. Controller/PI Controller/Discrete-Time Integrator - integration result >= upper limit T @ T=0.26
```

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

The analysis still satisfies all 34 objectives.

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 14-13, 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 “Create Reusable Subcomponents by Using Atomic Subcharts” (Stateflow).

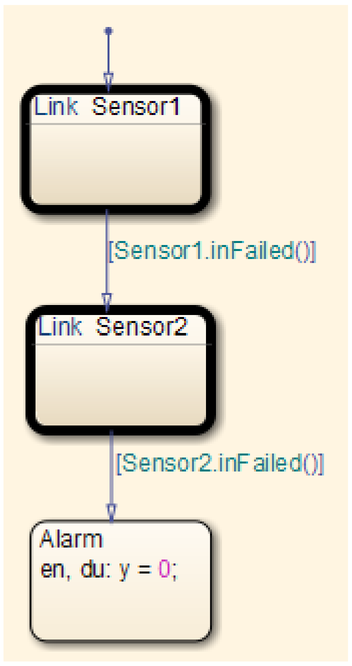
Analyze an Atomic Subchart by Using Simulink Design Verifier

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.

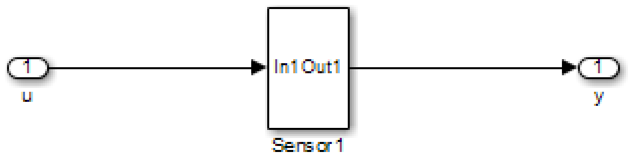


This Stateflow chart has two atomic subcharts:

- `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 For more information on supported blocks and features, see “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” on page 3-7 and “Support Limitations for Simulink Software Features” on page 3-16.
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 6-4 • “Workflow for Test Case Generation” on page 7-2 • “Workflow for Proving Model Properties” on page 12-4
6	Generate the results.	“Generate Analysis Results” on page 1-6
7	Interpret the results.	“Results Interpretation and Use”

See Also

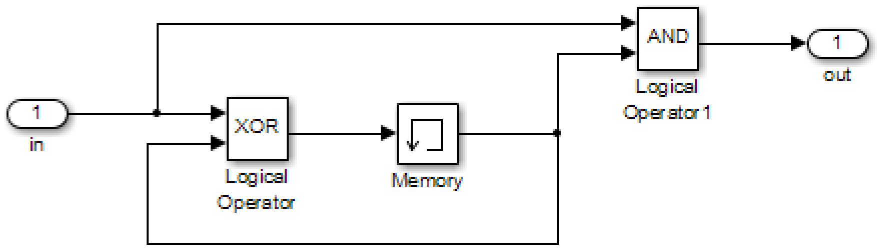
“Support Limitations for Stateflow Software Features” on page 3-20 | “Support Limitations for Model Blocks” on page 3-19

More About

- Systematic Model Verification using Simulink Design Verifier
- “Analyze a Model” on page 1-4

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.

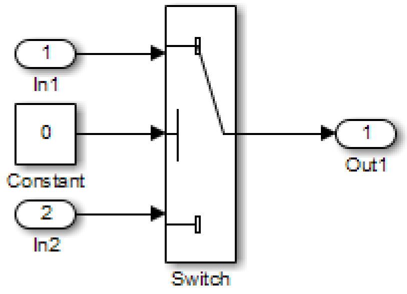
For example, consider a top-level model with three Model blocks referencing the same model. The referenced model has three test objectives. Analyzing the top-level model produces nine test objectives. If you simulate the model with the nine test cases, the coverage results for that referenced model specify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

Inlined Parameters

Setting **Default parameter behavior** to `Inlined` on the **Optimization** pane of the Configuration Parameters dialog box optimizes Simulink models by transforming tunable parameters into constant values. For example, suppose that 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 `Tunable`.

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?

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

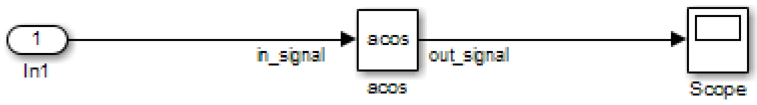
When you enable the automatic stubbing option (it is enabled by default), the software considers only the interface of the unsupported objects, not their actual behavior. This technique allows the software to complete the analysis. However, the analysis may achieve only partial results if any unsupported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

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



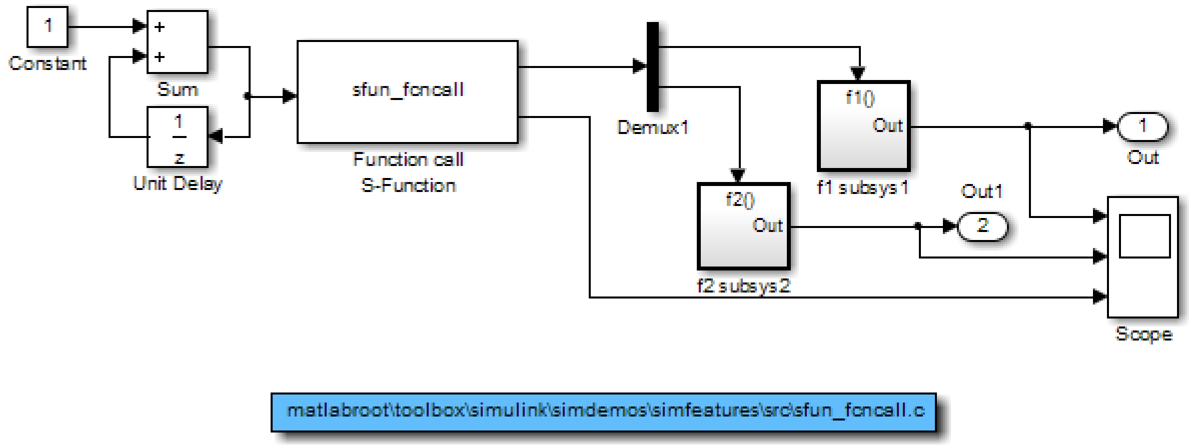
When stubbing this block during analysis, `out_signal` can take any value, with the following results.

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.

Analysis Model	Result of Stubbing out_signal
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.



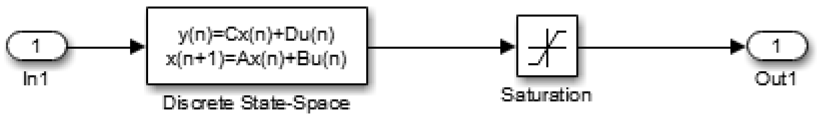
If you do not enable support for an S-function in Simulink Design Verifier and automatic stubbing is enabled, the analysis ignores the behavior of the S-function. As a result, the code that triggers the two function-call subsystems is ignored, resulting in two unsatisfiable objectives. Since the function 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 and Considerations for S-Functions and C/C++ Code” on page 3-27

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.

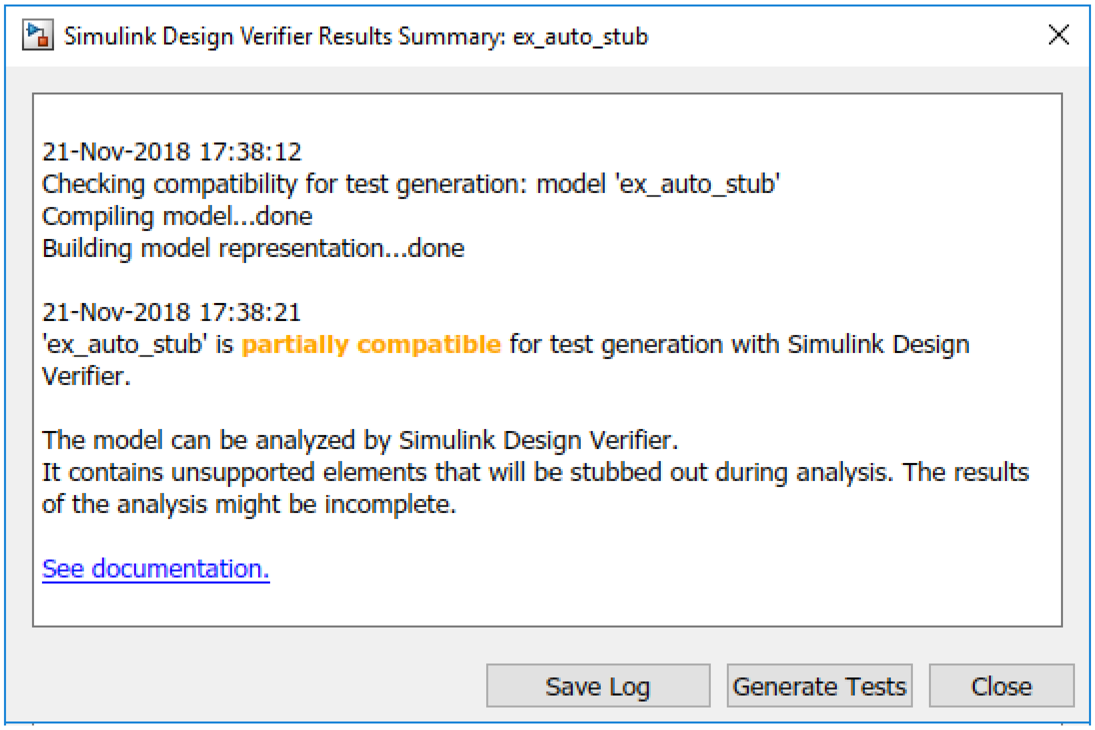


Check Model Compatibility

From the Simulink Editor, there are two ways to check whether a model is compatible with Simulink Design Verifier: by the Simulink Design Verifier compatibility check or by running a Simulink Design Verifier analysis.

To run the Simulink Design Verifier compatibility check:

- On the **Design Verifier** tab, click **Check Compatibility**.



```
21-Nov-2018 17:38:12
Checking compatibility for test generation: model 'ex_auto_stub'
Compiling model...done
Building model representation...done

21-Nov-2018 17:38:21
'ex_auto_stub' is partially compatible for test generation with Simulink Design
Verifier.

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.

See documentation.
```

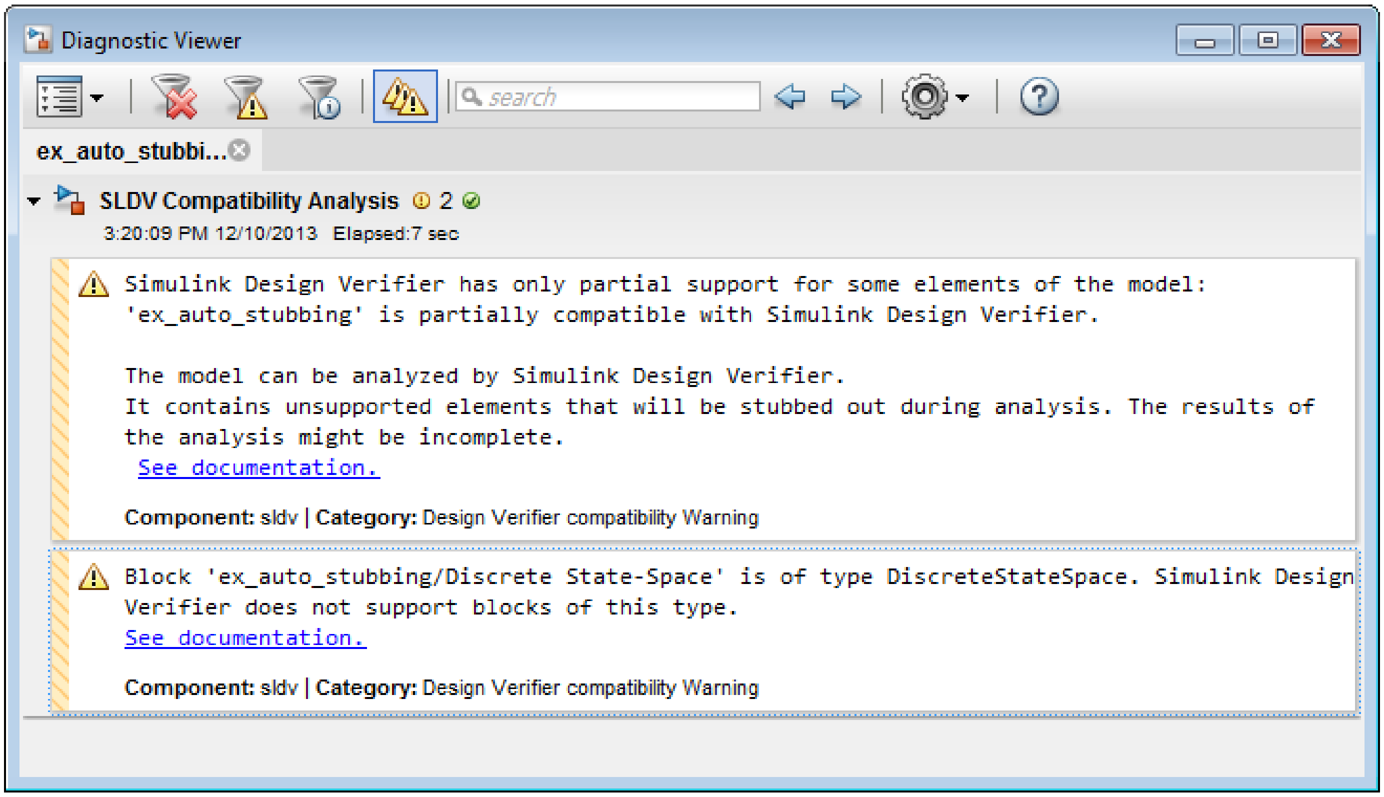
- Select the analysis that you want to perform.

  To run a Simulink Design Verifier analysis, on the **Design Verifier** tab, in the **Mode** section, select any of these options:

  - Select **Design Error Detection**, then click **Detect Design Errors**.
  - Select **Test Generation**, then click **Generate Tests**.
  - Select **Property Proving**, then click **Prove Properties**.

  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.



SLDV Compatibility Analysis 2
3:20:09 PM 12/10/2013 Elapsed:7 sec

```
Simulink Design Verifier has only partial support for some elements of the model:
'ex_auto_stubbing' is partially compatible with Simulink Design Verifier.

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.
 See documentation.
```

Component: sldv | **Category:** Design Verifier compatibility Warning

```
Block 'ex_auto_stubbing/Discrete State-Space' is of type DiscreteStateSpace. Simulink Design
Verifier does not support blocks of this type.
See documentation.
```

Component: sldv | **Category:** Design Verifier compatibility Warning

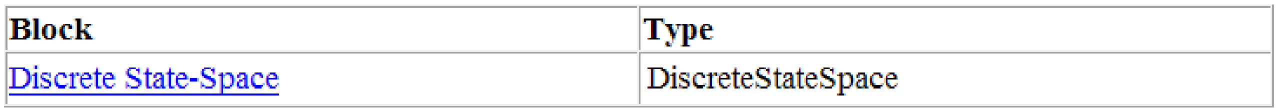
Note For more information, see "View Diagnostics" (Simulink).

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

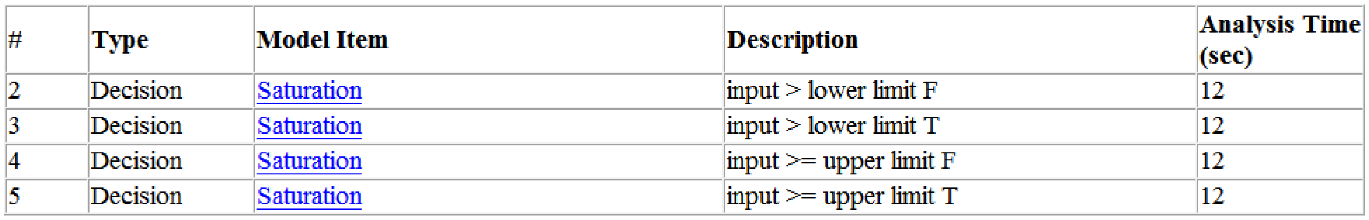
If you run an analysis with automatic stubbing enabled, make sure to review the results. In this report, generated after a test case generation analysis, you see a table of unsupported blocks that the software encountered.



Block	Type
Discrete State-Space	DiscreteStateSpace

Unsupported Blocks

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.



#	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

Objective Undecided Due to Stubbing

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, follow the steps in “Block Replacements for Unsupported Blocks”.

Analyze Export-Function Models

Simulink Design Verifier supports design error detection, test generation, and property proving for export-function models. The software creates schedulers that invoke the export-function models, and then performs the analysis on the scheduler model. You can analyze export-function models with periodic and aperiodic function-call groups. The scheduler invokes the function calls based on the sample times and priorities set in the top model. For more information, see “Export-Function Models Overview” (Simulink).

Analyze an Export-Function Model with Function-Call Subsystems

When you invoke Simulink Design Verifier analysis on a model that consists of export-function models, the software creates a scheduler model and then performs the analysis. By default, the scheduler model that the software creates is saved in this location `<current_folder>\sldv_output\<model_name>\<model_name>_SldvScheduler.slx`

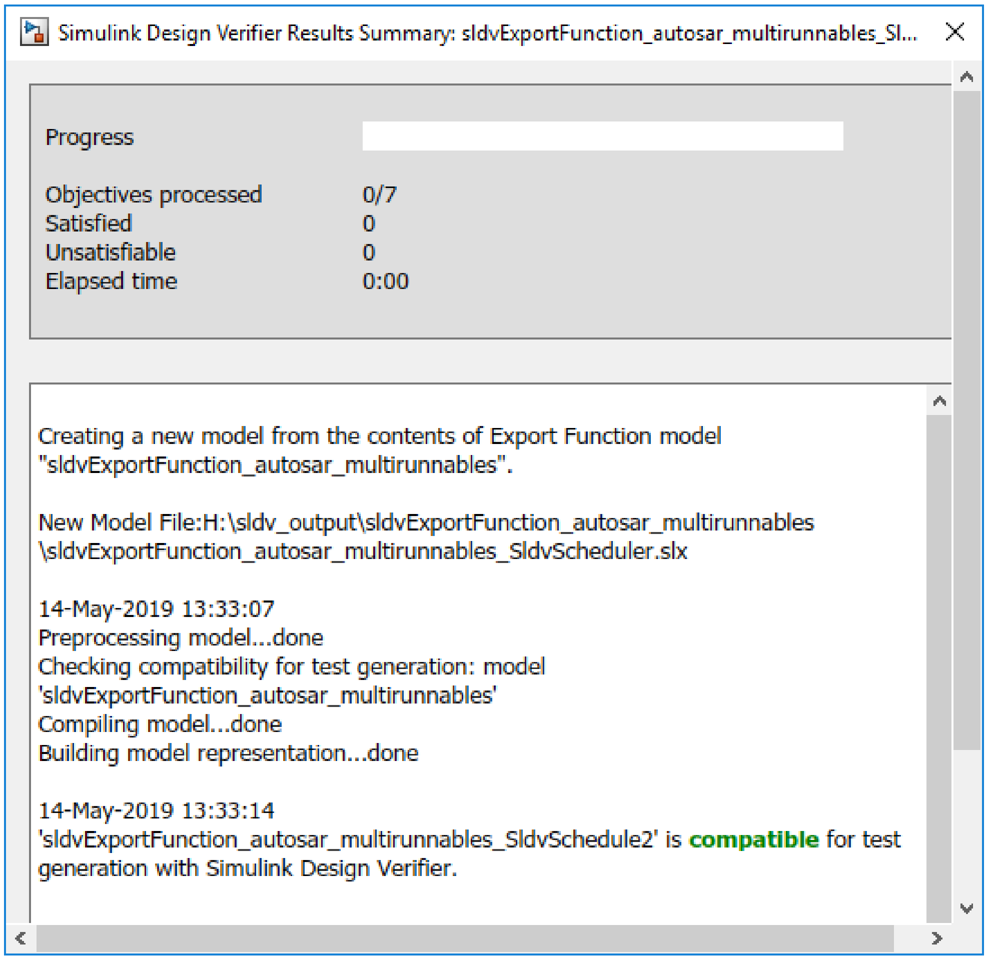
This example shows how to analyze an AUTOSAR example model `sldvExportFunction_autosar_multirunnables` that consists of periodic function-call subsystems.

1. Add the example folder to the search path.

   ```
   addpath(fullfile(docroot,'toolbox','sldv','examples'));
   ```
2. Open the `sldvExportFunction_autosar_multirunnables` model.

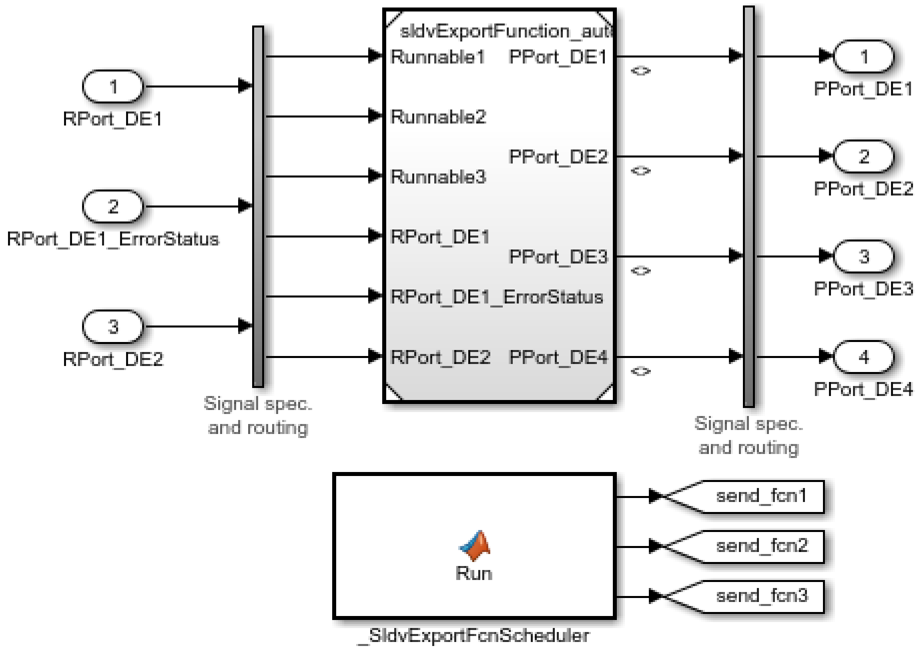
   ```
   open_system('sldvExportFunction_autosar_multirunnables');
   ```
3. To run the test generation analysis, on the **Design Verifier** tab, click **Generate Tests**.

   The Results Summary window indicates that a scheduler model `sldvExportFunction_autosar_multirunnables_SldvScheduler.slx` was created. You can also generate a scheduler model by using `sldvextract`.


Simulink Design Verifier Results Summary: sldvExportFunction_autosar_multirunnables_Sl...

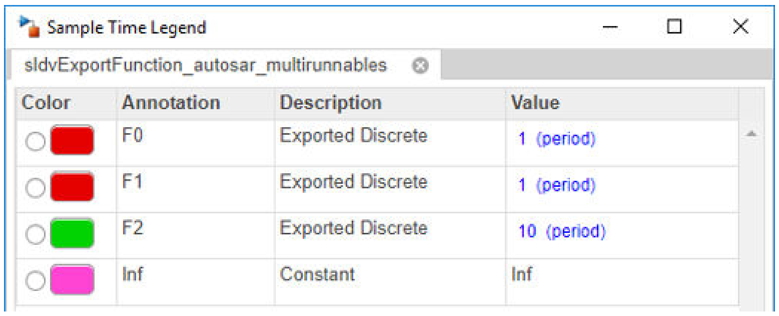
Progress	
Objectives processed	0/7
Satisfied	0
Unsatisfiable	0
Elapsed time	0:00

```
Creating a new model from the contents of Export Function model
"sldvExportFunction_autosar_multirunnables".

New Model File:H:\sldv_output\sldvExportFunction_autosar_multirunnables
\sldvExportFunction_autosar_multirunnables_SldvScheduler.slx

14-May-2019 13:33:07
Preprocessing model...done
Checking compatibility for test generation: model
'sldvExportFunction_autosar_multirunnables'
Compiling model...done
Building model representation...done

14-May-2019 13:33:14
'sldvExportFunction_autosar_multirunnables_SldvSchedule2' is compatible for test
generation with Simulink Design Verifier.
```



The scheduler model consists of a MATLAB function block `_SldvExportFcnScheduler`. The function calls are called periodically as the model consists of periodic function-call subsystem.

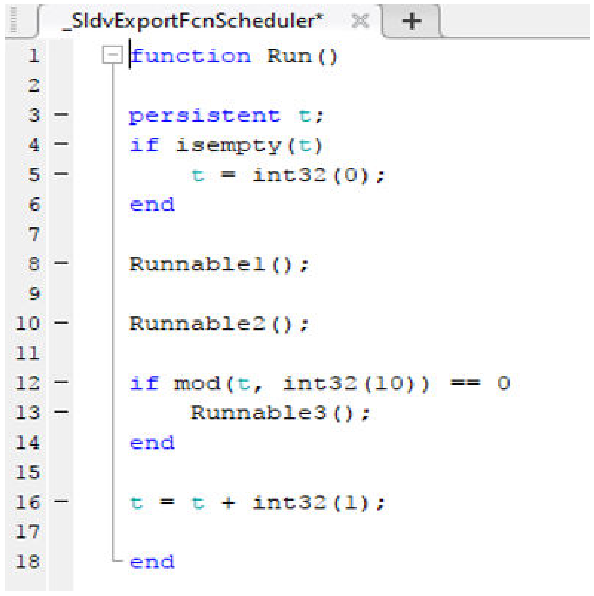
The MATLAB code specifies the order in which the periodic function-call executes. The `Runnable1` and `Runnable2` executes first because the time period is `1` for both of them. After `10` time steps, the `Runnable3` executes.



Sample Time Legend

sldvExportFunction_autosar_multirunnables

Color	Annotation	Description	Value
🟥	F0	Exported Discrete	1 (period)
🟥	F1	Exported Discrete	1 (period)
🟩	F2	Exported Discrete	10 (period)
🟪	Inf	Constant	Inf

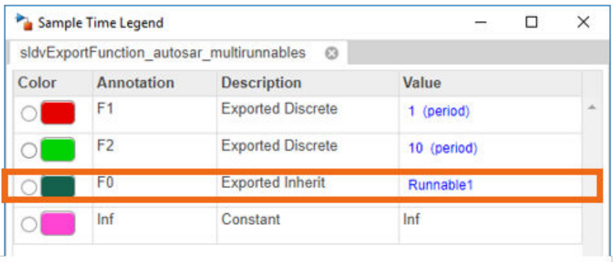


_SldvExportFcnScheduler*

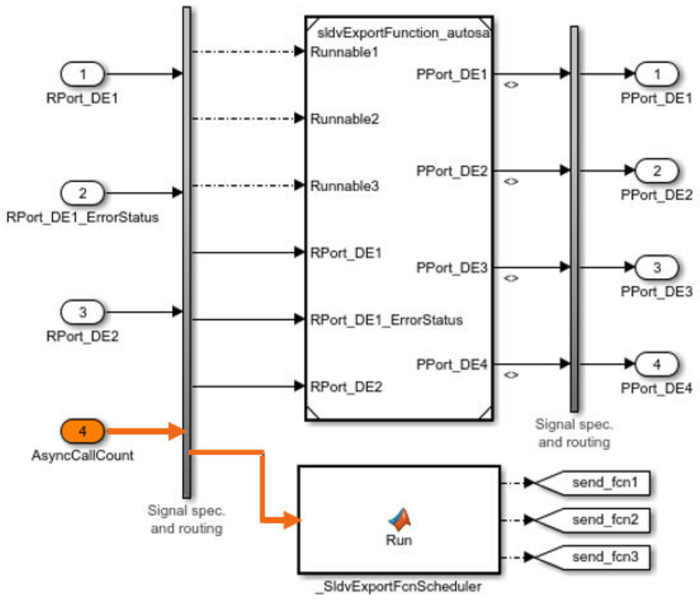
```
function Run()

persistent t;
if isempty(t)
    t = int32(0);
end

Runnable1();

Runnable2();

if mod(t, int32(10)) == 0
    Runnable3();
end

t = t + int32(1);

end
```

If the model consists of aperiodic function-call subsystems, the scheduler consists of an additional inport `AsyncCallCount`. The value of `AsyncCallCount` indicates whether to invoke the function-call or not in a time step.

For example, if the `Runnable1` is an aperiodic function-call subsystem, the scheduler consists of `AsyncCallCount` inport to invoke the scheduler. The Sample Time Legend and the scheduler model for the aperiodic function-call is shown in the graphic.



Color	Annotation	Description	Value
🟥	F1	Exported Discrete	1 (period)
🟩	F2	Exported Discrete	10 (period)
■	F0	Exported Inherit	Runnable1
🟪	Inf	Constant	Inf



After the test generation analysis, in the summary window, you see the results that `7/7` objectives are `Satisfied`.

4 To generate a coverage report by simulating the test cases, in the Results Summary window, click **Simulate tests and produce a model coverage report**.

   The software simulates all the test cases, collects model coverage information, and displays a coverage report.

5 To view the detailed analysis report, click **HTML** in the Results Summary window.

   The **Schedule for Export Function Analysis** section in the **Analysis Information** chapter lists the schedule for invoking the export-functions.

Order	Function-Call Inport	Sample Time(sec)	Number of times invoked per sample hit
1	Runnable1	1	1
2	Runnable2	1	1
3	Runnable3	10	1

Schedule for Export Function Analysis

Limitations

- Simulink Design Verifier analysis does not support a model that consists of export-functions with multiple function-call initiators.

- A masked model block that exports a Simulink Function block is not supported.

See Also

More About

- “Export-Function Models” (Simulink)
- “Analyze a Model” on page 1-4

Nonfinite Data

Simulink Design Verifier 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

In this section...

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 13-29.

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 2-19
- “Linearization of Two-Dimensional Lookup Tables for Floating-Point Data Types” on page 2-20
- “Approximation of One- and Two-Dimensional Lookup Tables for Integer and Fixed-Point Data Types” on page 2-20
- “While Loops” on page 2-20

Floating-Point to Rational Number Conversion

In some cases, 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 some 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.
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.

Reporting Approximations Through Validation Results

Simulink Design Verifier performs approximations during analysis. The software identifies the presence of approximations and reports them at the level of each objective status in the Objective Status Chapter of the Simulink Design Verifier HTML report. For more information, see “Approximations During Model Analysis” on page 2-19 and “Objectives Status Chapters” on page 13-33.

To validate the test cases or counterexamples during simulation, the model is locked in fast restart mode. For more information, see “Fast Restart Methodology” (Simulink).

For example, to ensure the effect of approximations, in the test generation analysis the test cases are validated against the coverage data during analysis.

Impact of Approximations on Objectives Status

The software provides the test cases or counterexamples for the objectives that are impacted due to approximations during analysis. These objectives are reported as “Objectives Undecided with Testcases” on page 13-38 for test generation analysis and “Objectives Undecided with Counterexamples” on page 13-39 for property-proving analysis.

The software confirms the objectives that can be impacted due to approximations as dead logic, valid, or unsatisfiable. This table summarizes these objectives for all analysis modes.

Analysis Mode	Objectives Status
Design error detection	• “Dead Logic under Approximation” on page 13-35 • “Objectives Valid under Approximation” on page 13-36
Test generation	“Objectives Unsatisfiable under Approximation” on page 13-37
Property proving	“Objectives Valid under Approximation” on page 13-38

The software is unable to confirm the objectives status through validation results for these cases:

- The objectives introduced by the block replacement. For more information, see “What Is Block Replacement?” on page 4-2.
- The Verification Subsystem consists of the `sldv.test` or `sldv.prove` function.
- You abort the analysis by using the **Stop** button in the Simulink Design Verifier Results Summary window or the software exceeds its “Maximum analysis time” on page 15-11. Therefore, some objectives remain unvalidated during analysis and the software is unable to confirm the objectives status.
- The block with an objective is inside the Simulink test harness but outside the component under test. For more information, see “Test Harness and Model Relationship” (Simulink Test).

This table summarizes the objectives statuses for the preceding cases. To confirm the status of the objectives, you must run additional simulations of test cases or counterexamples.

Analysis Mode	Objectives Status
Design error detection	• “Active Logic - Needs Simulation” on page 13-35 • “Objectives Falsified - Needs Simulation” on page 13-36

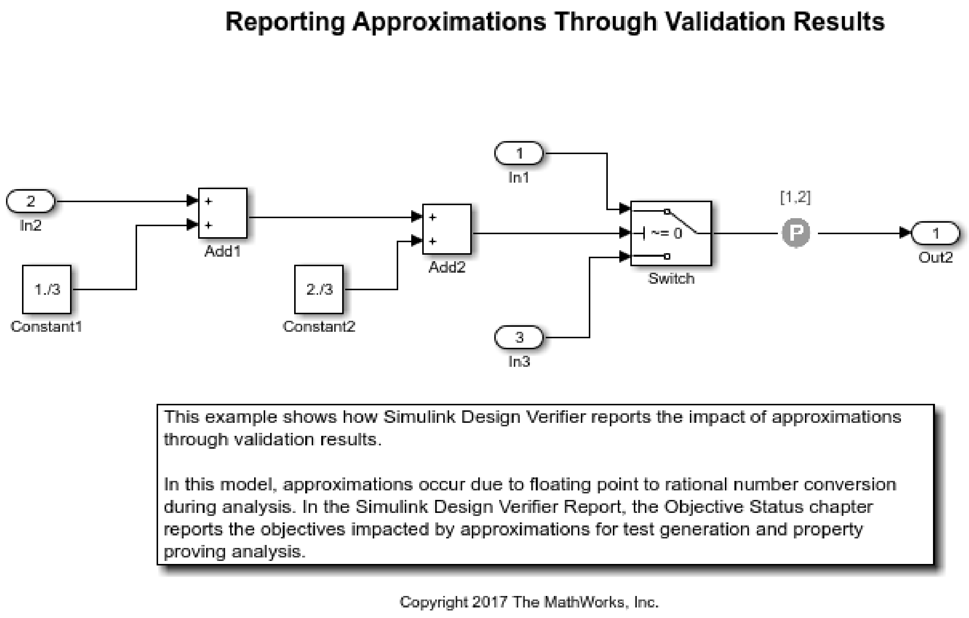
Analysis Mode	Objectives Status
Test generation	“Objectives Satisfied - Needs Simulation” on page 13-37
Property proving	“Objectives Falsified - Needs Simulation” on page 13-39

Identify the Effect of Approximations Through Validation Results

This example shows how approximations affect the objectives status of the Switch block. In the `sldvApproximationsExample` model, the calculations `1./3` and `2./3` in the Constant block result in “Floating-Point to Rational Number Conversion” on page 2-19 during analysis.

For inport `In2` equal to `-1`, the input `2` of the Switch block is not equal to `0` during simulation. Therefore, the Switch does not select inport `In3` as output. For test generation and property-proving analysis, the objective `logical trigger input false(output is from 3rd input port)` for the Switch block is undecided due to the impact of approximations during analysis.

1 Open the model sldvApproximationsExample.


Reporting Approximations Through Validation Results
2
In2
1./3
Constant1
+
+
Add1
2./3
Constant2
+
+
Add2
1
In1
3
In3
~= 0
Switch
[1,2]
P
1
Out2
Copyright 2017 The MathWorks, Inc.

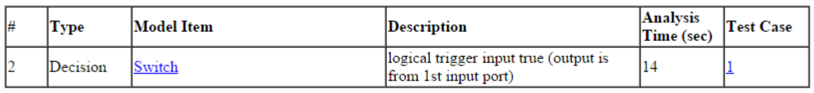
This example shows how Simulink Design Verifier reports the impact of approximations through validation results.

In this model, approximations occur due to floating point to rational number conversion during analysis. In the Simulink Design Verifier Report, the Objective Status chapter reports the objectives impacted by approximations for test generation and property proving analysis.

2 To perform test generation analysis, on the **Design Verifier** tab, click **Generate Tests**. The software simulates the model and validates the test results against coverage data.

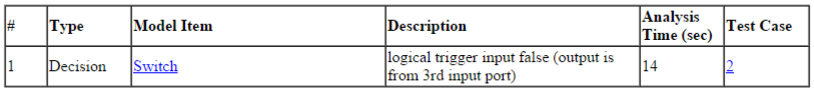
3 To view the detailed analysis report, click HTML in the Simulink Design Verifier Results Summary window.

   This image shows the Test Objectives Status section of the generated analysis report. The software provides two test cases that are impacted by approximations.



#	Type	Model Item	Description	Analysis Time (sec)	Test Case
2	Decision	Switch	logical trigger input true (output is from 1st input port)	14	1

Test Objectives Status - Objective Satisfied

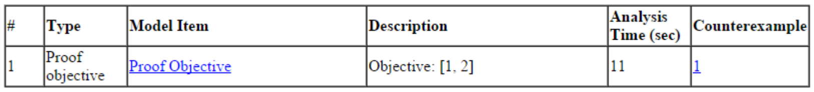


#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Switch	logical trigger input false (output is from 3rd input port)	14	2

Test Objectives Status - Objective Undecided with Testcases

4 To perform property proving analysis, on the **Design Verifier** tab, in the **Mode** section, select **Property Proving**. Click **Prove Properties**.

This image shows the Proof Objectives Status section of the generated analysis report.



#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Proof Objective	Objective: [1, 2]	11	1

Proof Objective Status - Objective Undecided with Conterexamples

The software provides one counterexample that is impacted by approximations.

Note Th `sldvApproximationsExample` example model is preconfigured with the “Run additional analysis to reduce instances of rational approximation” on page 15-15 option set to `Off`. If you enable this option and run the analysis, the `Undecided with Testcases` test objective is reported as `Unsatisfiable` and the proof objective is reported as `Valid`.

See Also

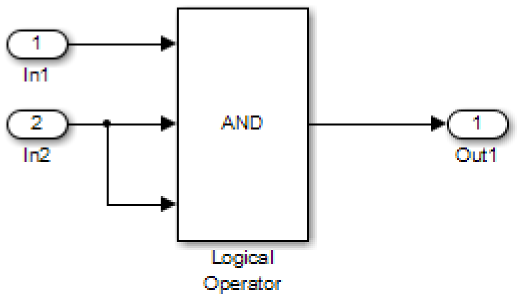
More About

- “Approximations” on page 2-19
- “Simulink Design Verifier Reports” on page 13-28

Logic Operations Short-Circuiting

Simulink Design Verifierconsider logic blocks as short-circuiting during analysis.

Consider the following example model, with the **Model coverage objectives** parameter set to `Condition Decision`. In this case, when a previous input alone determines the block output, the analysis ignores any remaining block inputs. 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.



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.

Model Representation for Analysis

In this section...

When you analyze a model for the first time, Simulink Design Verifier performs a compatibility check and creates a model representation. The model representation contains information about model behavior to use for analysis. By default, the software saves the model representation at the Simulation cache folder (Simulink) location.

If you modify a model and rerun the analysis, Simulink Design Verifier determines whether to rebuild the model representation or to use the existing Simulink cache depending on the "Rebuild model representation" on page 15-13 parameter. A rebuild of the model representation is triggered, when the **Rebuild model representation** option is set to `If change is detected` and the software detects any changes in the model.

Reuse Model Representation for Analysis

The **Rebuild model representation** option is set to `If change is detected` by default and the software validates the model representation against any model changes and Simulink Design Verifier analysis options. The software then determines whether to reuse or to rebuild the model representation for analysis. When you set the option to `Always`, the model representation is rebuilt during every model analysis.

When the **Rebuild model representation** option is set to `If change is detected`, Simulink Design Verifier checks for these changes in a model:

- Simulink Design Verifier Options on page 2-26
- "Structural Checksum of a Model" on page 2-28
- "Additional Dependencies" on page 2-28

Simulink Design Verifier Options

The software validates the model representation against any changes in the Simulink Design Verifier options. This table lists the options that do not affect the model representation, and if you change any of these options the software reuses the model representation.

Design Verifier Options	• “Maximum analysis time” on page 15-11 • “Display unsatisfiable test objectives” on page 15-11 • “Output folder” on page 15-12 • “Make output file names unique by adding a suffix” on page 15-13 • “Run additional analysis to reduce instances of rational approximation” on page 15-15 • “Ignore objectives based on filter” on page 15-17 • “Filter file” on page 15-18
Test Generation options	• “Test conditions” on page 15-33 • “Test objectives” on page 15-34 • “Maximum test case steps” on page 15-34 • “Test suite optimization” on page 15-35 • “Extend existing test cases” on page 15-39 • “Ignore objectives satisfied by existing test cases” on page 15-40 • “Ignore objectives satisfied in existing coverage data” on page 15-40 • “Coverage data file” on page 15-41
Property Proving options	• “Assertion blocks” on page 15-50 • “Proof assumptions” on page 15-51 • “Strategy” on page 15-51 • “Maximum violation steps” on page 15-52
Results generation options	• “Save test data to file” on page 15-55 • “Data file name” on page 15-55 • “Include expected output values” on page 15-56 • “Randomize data that do not affect the outcome” on page 15-56 • “Generate separate harness model after analysis” on page 15-58 • “Harness model file name” on page 15-58 • “Reference input model in generated harness” on page 15-59 • “Harness source” on page 15-60 • “Test File Name” on page 15-60 • “Test Harness Name” on page 15-61

Report generation options	• “Generate report of the results” on page 15-62 • “Generate additional report in PDF format” on page 15-63 • “Report file name” on page 15-63 • “Include screen shots of properties” on page 15-64 • “Display report” on page 15-65

Structural Checksum of a Model

A structural checksum is a computation that detects changes in the model that can affect simulation results. For more information about the kinds of changes that affect the model, see “Rebuild” (Simulink).

Additional Dependencies

In addition to structural checksum, Simulink Design Verifier checks for changes in model dependencies that can impact the analysis results, such as:

- Simulation run-time parameters that are defined in the data dictionary or the MATLAB base, mask, or model workspaces
- External C or C++ source code files that the model uses during simulation
- Minimum and maximum constraints that are specified for block parameters
- Block parameters that are specified for blocks in the “Simulink Design Verifier Block Library” on page 1-3, such as **Values**

Limitations

- The model representation is always rebuilt:
  - When you “Generate Test Cases for Embedded Coder Generated Code” on page 7-21.
  - When Simulink Design Verifier analysis is started from other products such as Simulink Test™, Simulink Coverage, Simulink Check™, and Simulink Requirements™.
- Simulink Design Verifier does not detect changes in the custom block replacement rules that you apply, even if the **Rebuild model representation** option is set to `If change is detected`. In such cases, the Simulink cache is reused for analysis and a warning message is displayed in the Diagnostic Viewer that suggests you to set the **Rebuild model representation** option to `Always`, if you want to rebuild the model representation.

See Also

“Extend Existing Test Cases by Reusing Model Representation” on page 2-31

More About

- Configure Model Representation Options on page 2-35
- “Check Model Compatibility” on page 3-2
- “Simulink Design Verifier Options” on page 15-2

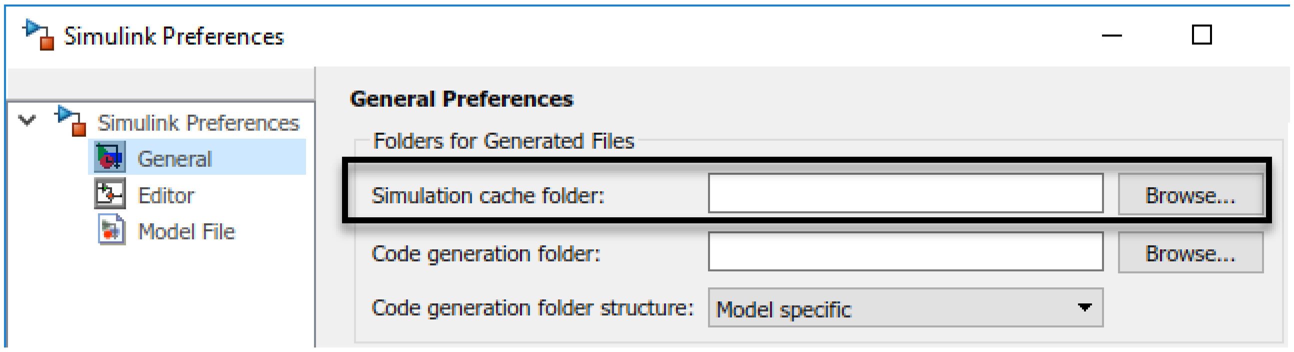
Share Simulink Cache File for Faster Analysis

In this section...
"Store the Simulink Cache File" on page 2-29
"Reuse the Simulink Cache File" on page 2-29

You can share the Simulink cache file for faster Simulink Design Verifier analysis. When you analyze a model, Simulink Design Verifier performs a compatibility check and creates a Simulink cache file that contains the model representation information. If there is no change in the model, Simulink Design Verifier reuses the model representation from the Simulink cache file without performing the compatibility check again. For more information, see "Share Simulink Cache Files for Faster Simulation" (Simulink) and "Model Representation for Analysis" on page 2-26.

Store the Simulink Cache File

The Simulink cache file is stored in the location specified in the **Simulink Preferences** > **General** dialog box, under **Simulation cache folder**. By default, the Simulink cache file is stored in the current working directory.



The file name of the Simulink cache is the same as the file name of the model with an `.slxc` file extension.

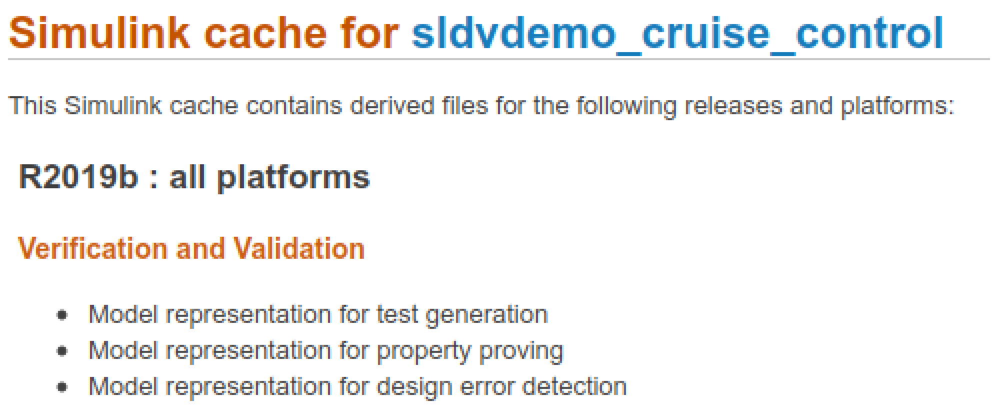
Reuse the Simulink Cache File

You can reuse the Simulink cache file to speed up the Simulink Design Verifier analysis for later use by yourself or others. When you perform Simulink Design Verifier analysis, the software determines whether to rebuild the model representation based on the "Rebuild model representation" on page 15-13 option. By default, this option is set to `If change is detected` and if there is no change in the model, the software reuses the model representation from the Simulink cache file for analysis.

If **Rebuild model representation** is set to `Always` or if the software detects any change in the model during analysis, the software rebuilds the model representation and updates the Simulink cache file.

Note The Simulink cache file accumulates model representation build artifacts for the release in which it was created and supports all platforms. This cache file does not support cross-release compatibility.

For information on what a specific Simulink cache contains, double-click the Simulink cache file. The report contains information of supported releases, platforms, and model representation.


Simulink cache for sldvdemo_cruise_control

This Simulink cache contains derived files for the following releases and platforms:

R2019b : all platforms

Verification and Validation

- Model representation for test generation
- Model representation for property proving
- Model representation for design error detection

For example, suppose a team is working on large models and uses a source control system to manage design files. To reduce the amount of time for Simulink Design Verifier analysis, the team follows these steps:

1. A developer pulls the latest version of the Simulink model from the source control system.
2. Performs Simulink Design Verifier test case generation analysis and shares the latest version of Simulink cache file to the source control system and the generated test cases to the build archive.
3. Test engineer pulls the latest version of the model and the Simulink cache file from the source control systems. Also, pulls the existing test cases from the build archive.
4. Performs test case extension on the same model by using the existing test cases. If no changes are detected in the model, the model representation from the Simulink cache file is reused for analysis. For a detailed example, see “Extend Existing Test Cases by Reusing Model Representation” on page 2-31.

   If the test engineer, changes the model or Simulink Design Verifier options that affects the compatibility results, the model representation is rebuilt and the Simulink cache file is updated. For more information on Simulink Design Verifier options that leverage the reuse of model representation, see “Reuse Model Representation for Analysis” on page 2-26.

See Also

More About

- “Model Representation for Analysis” on page 2-26
- Configure Model Representation Options on page 2-35

External Websites

- Simulink Cache (1 min, 27 sec)

Extend Existing Test Cases by Reusing Model Representation

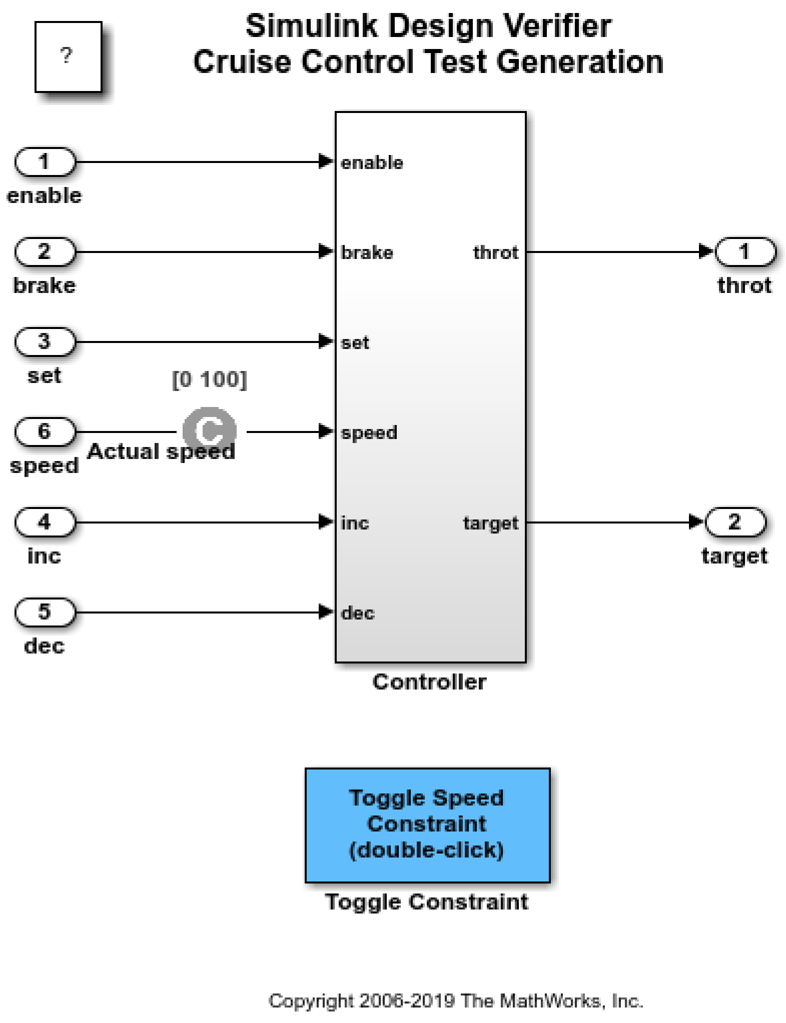
This example shows how to avoid unneeded model representation builds when reanalyzing a model. Consider a case where you perform test generation and the analysis exceeds maximum analysis time. In the specified analysis time, Simulink Design Verifier analyzes some objectives and saves the generated test cases in a MAT-file.

To reanalyze the model, you update the maximum analysis time and select the extend existing test cases option. To speed up the analysis, set the **Rebuild model representation** option to `If change is detected`. Simulink Design Verifier reanalyzes the model by reusing the model representation. For more information, see “Model Representation for Analysis” on page 2-26.

Step 1. Open the model and specify analysis options

Generate test cases for `sldvdemo_cruise_control` model by specifying the `sldvoptions`.

```
model = 'sldvdemo_cruise_control';
open_system(model);
opts = sldvoptions;
opts.Mode = "TestGeneration";
opts.MaxProcessTime = 10;
opts.RebuildModelRepresentation = "IfChangeIsDetected";
```



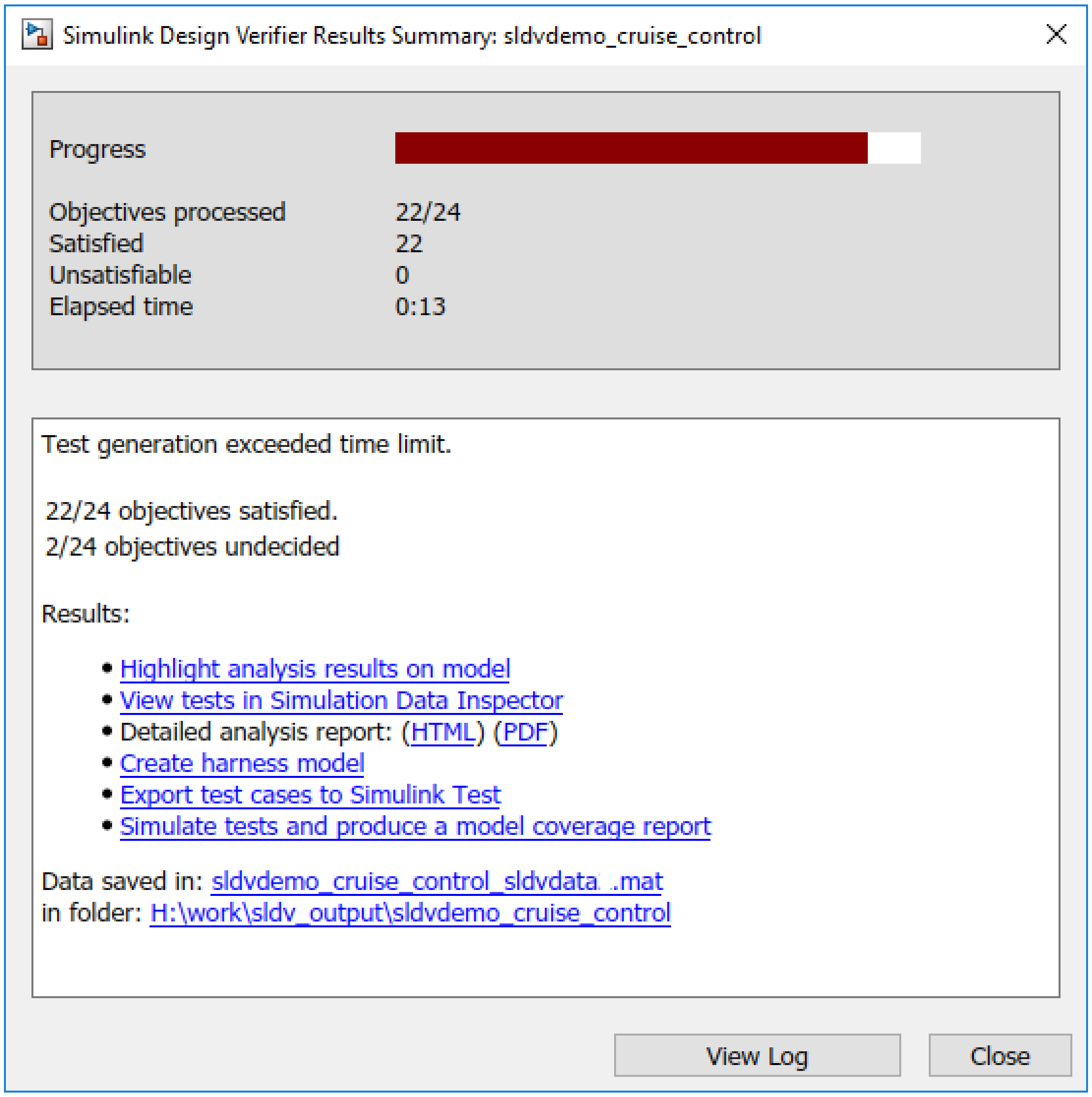
Analyze the model by using this command.

```
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts, true);
```

The Diagnostic Viewer window displays the Test Generation analysis error.

```
Simulink Design Verifier has exceeded the maximum processing time. You can
extend the time limit by modifying the "Maximum analysis time" edit field on
the Design Verifier pane of the configuration dialog or by modifying the
"MaxProcessTime" attribute of the options object.
```

After the analysis is completed, the Results Summary window displays the results. The software reports 22/24 objectives as satisfied and 2/24 objectives as undecided.


Simulink Design Verifier Results Summary: sldvdemo_cruise_control

Progress

Objectives processed 22/24
Satisfied 22
Unsatisfiable 0
Elapsed time 0:13

Test generation exceeded time limit.

22/24 objectives satisfied.
2/24 objectives undecided

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

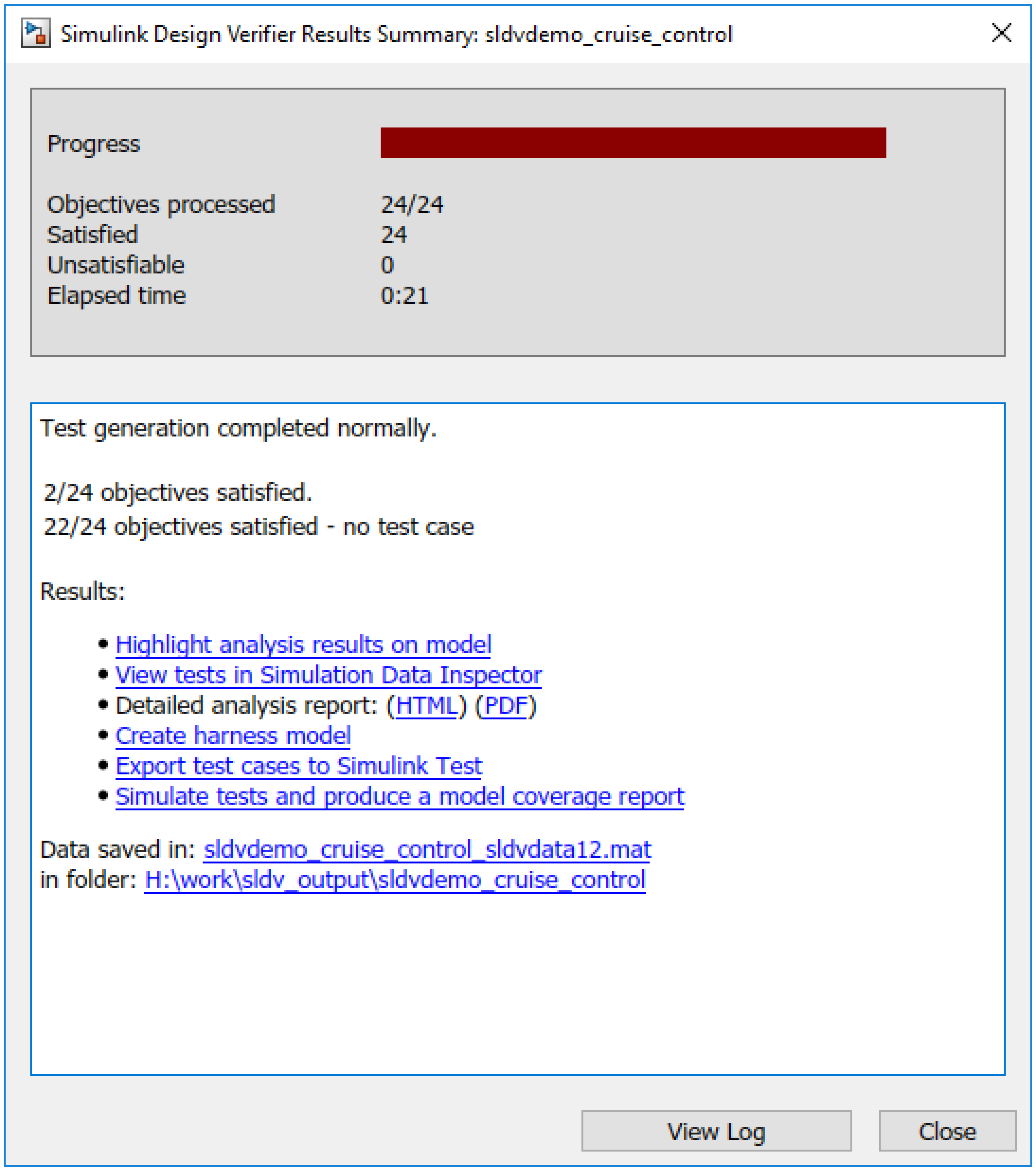
Data saved in: sldvdemo_cruise_control_sldvdata .mat
in folder: H:\work\sldv_output\sldvdemo_cruise_control

Step 2. Reanalyze the model by modifying the sldvptions

To reanalyze the model, you select the extend existing test cases option and update the maximum analysis time. The **Rebuild model representation** option is set to `If change is detected`. The software validates the cache model representation, detects no change, and reuses the model representation for analysis.

```
opts.MaxProcessTime =500;
opts.ExtendExistingTests='on';
opts.IgnoreExistTestSatisfied = 'on';
opts.ExistingTestFile=files.DataFile;
sldvrun('sldvdemo_cruise_control', opts, true);
```

The results show that `24/24` objectives are satisfied and no additional test cases are generated.


Simulink Design Verifier Results Summary: sldvdemo_cruise_control

Progress

Objectives processed 24/24
Satisfied 24
Unsatisfiable 0
Elapsed time 0:21

Test generation completed normally.

2/24 objectives satisfied.
22/24 objectives satisfied - no test case

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: sldvdemo_cruise_control_sldvdata12.mat
in folder: H:\work\sldv_output\sldvdemo_cruise_control

Close the model.

```
close_system('sldvdemo_cruise_control', 0);
```

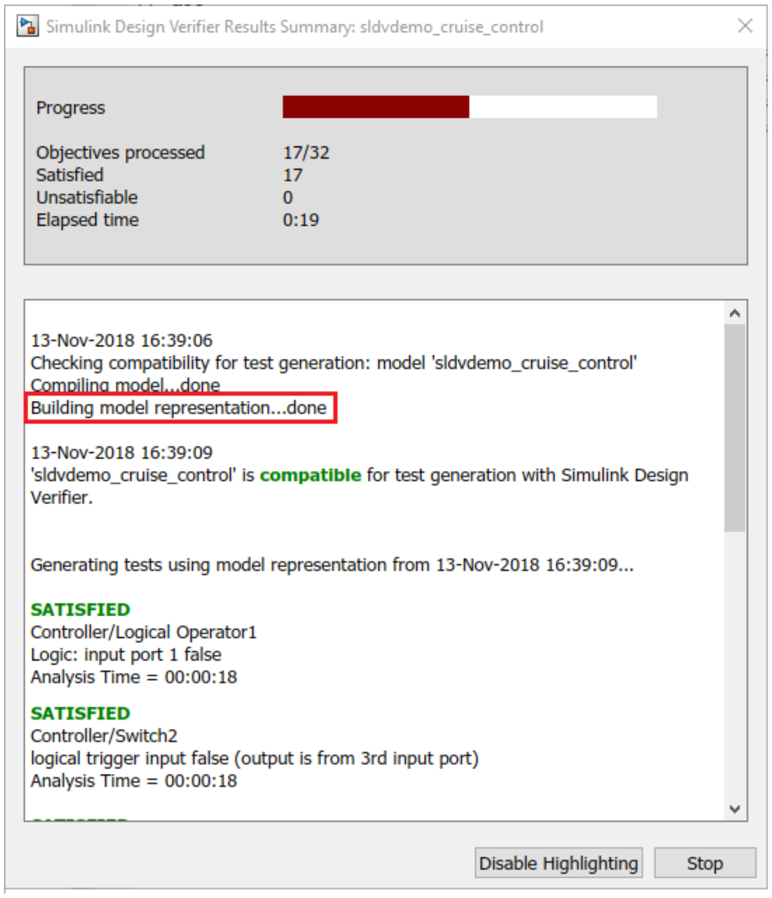
Related Topics

- “Model Representation for Analysis” on page 2-26
- “Extend an Existing Test Suite” on page 7-69

Configure Model Representation Options

You can configure the option to build or reuse the model representation from the **Design Verifier** pane, “Rebuild model representation” on page 15-13 option or by using the `sldvoptions`. By default, the option is set to `If change is detected` and the software reuses the model representation for analysis, if there is no change in the model.

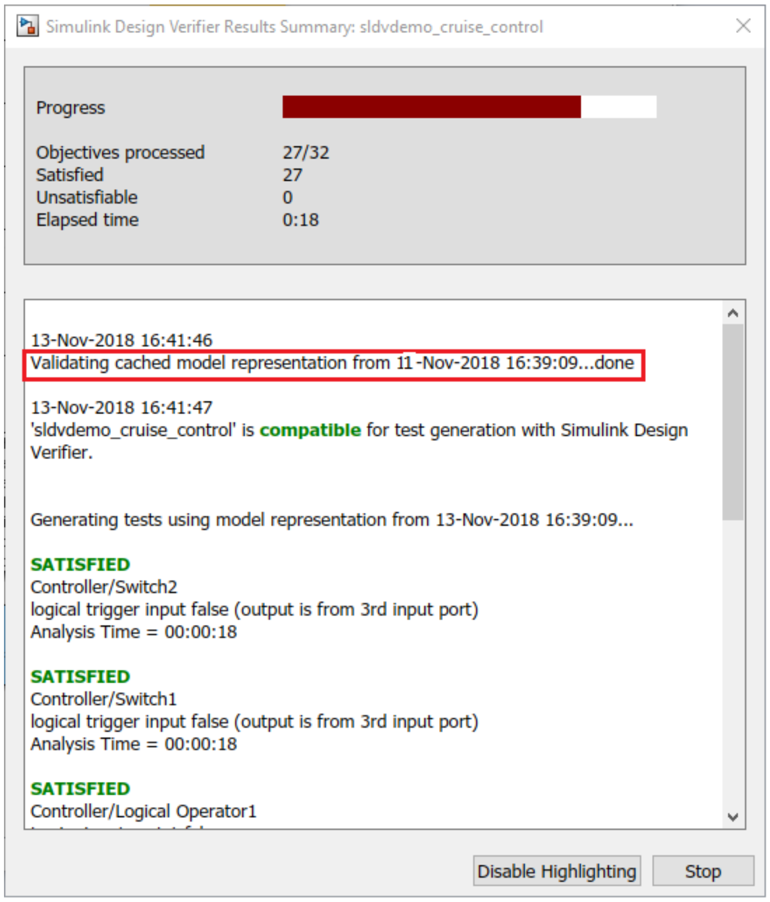
When you perform analysis, the Results Summary window displays the information regarding the model representation. If you select `Always` for the **Rebuild model representation** option, the software rebuilds the model representation during analysis.


Simulink Design Verifier Results Summary: sldvdemo_cruise_control

Progress	
Objectives processed	17/32
Satisfied	17
Unsatisfiable	0
Elapsed time	0:19

```
13-Nov-2018 16:39:06
Checking compatibility for test generation: model 'sldvdemo_cruise_control'
Compiling model...done
Building model representation...done

13-Nov-2018 16:39:09
'sldvdemo_cruise_control' is compatible for test generation with Simulink Design
Verifier.


Generating tests using model representation from 13-Nov-2018 16:39:09...

SATISFIED
Controller/Logical Operator1
Logic: input port 1 false
Analysis Time = 00:00:18

SATISFIED
Controller/Switch2
logical trigger input false (output is from 3rd input port)
Analysis Time = 00:00:18
```

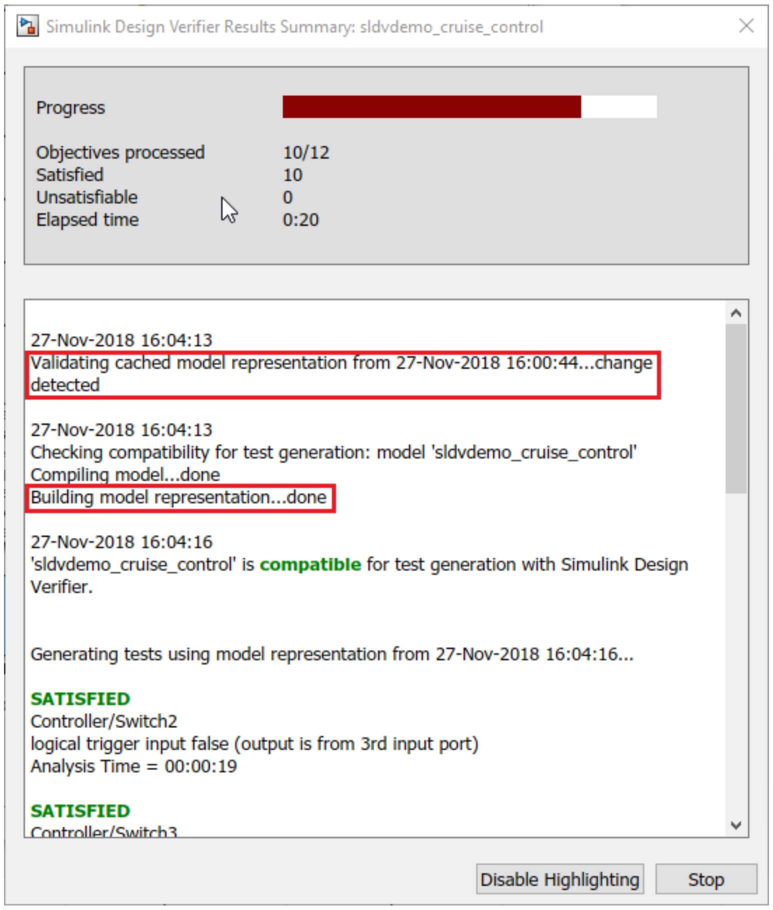
If you select `If change is detected` option, the software validates the existing cached model representation. If the cached model is successfully validated, it is reused for analysis.


Simulink Design Verifier Results Summary: sldvdemo_cruise_control
Disable Highlighting
Stop

```
Progress

Objectives processed    27/32
Satisfied               27
Unsatisfiable           0
Elapsed time            0:18
```

```
13-Nov-2018 16:41:46
Validating cached model representation from 11-Nov-2018 16:39:09...done

13-Nov-2018 16:41:47
'sldvdemo_cruise_control' is compatible for test generation with Simulink Design
Verifier.


Generating tests using model representation from 13-Nov-2018 16:39:09...

SATISFIED
Controller/Switch2
logical trigger input false (output is from 3rd input port)
Analysis Time = 00:00:18

SATISFIED
Controller/Switch1
logical trigger input false (output is from 3rd input port)
Analysis Time = 00:00:18

SATISFIED
Controller/Logical Operator1
```

If change is detected in the model, the model representation is rebuilt. For more information, see Changes That Affect the Model Representation Rebuild on page 2-26.


Simulink Design Verifier Results Summary: sldvdemo_cruise_control
Disable Highlighting
Stop

```
Progress

Objectives processed    10/12
Satisfied               10
Unsatisfiable           0
Elapsed time            0:20

27-Nov-2018 16:04:13
Validating cached model representation from 27-Nov-2018 16:00:44...change
detected

27-Nov-2018 16:04:13
Checking compatibility for test generation: model 'sldvdemo_cruise_control'
Compiling model...done
Building model representation...done

27-Nov-2018 16:04:16
'sldvdemo_cruise_control' is compatible for test generation with Simulink Design
Verifier.


Generating tests using model representation from 27-Nov-2018 16:04:16...

SATISFIED
Controller/Switch2
logical trigger input false (output is from 3rd input port)
Analysis Time = 00:00:19

SATISFIED
Controller/Switch3
```

See Also

More About

- “Model Representation for Analysis” on page 2-26
- “Check Model Compatibility” on page 3-2

Run Additional Analysis to Reduce Instances of Rational Approximation

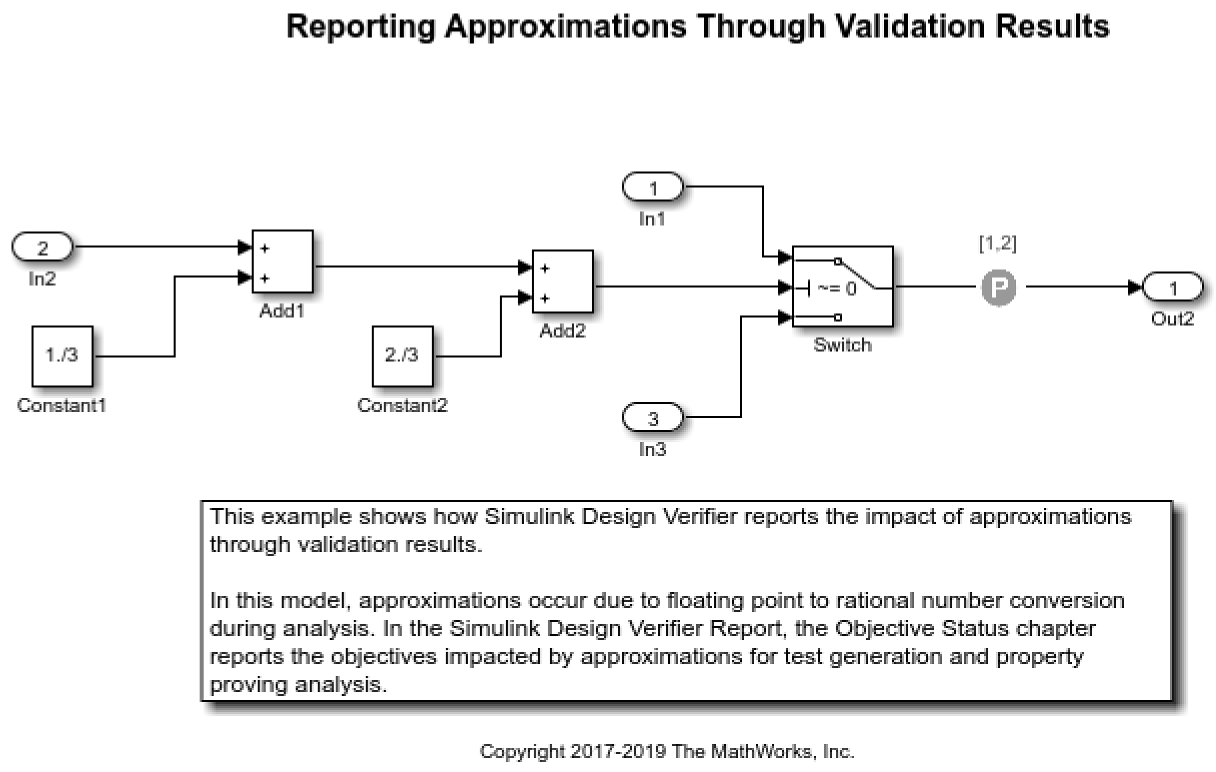
This example shows how to reduce the instances of rational approximation by running additional analysis. You analyze a model and during the analysis, Simulink® Design Verifier™ identifies the presence of approximations and the associated objectives are reported as undecided with test case.

You enable **Run additional analysis to reduce instances of rational approximation** option to perform additional analysis to confirm the undecided objectives. When you rerun the analysis, Simulink cache that contains the model representation information is reused to perform faster analysis. For more information see, “Reuse Model Representation for Analysis” on page 2-26.

Step 1: Open the Model

The `sldvApproximationsExample` model results in approximations due to the calculations 1./3 and 2./3 in the Constant block.

- open_system(docpath(fullfile(docroot,'toolbox','sldv','examples','sldvApproximationsExample')))



This example shows how Simulink Design Verifier reports the impact of approximations through validation results.

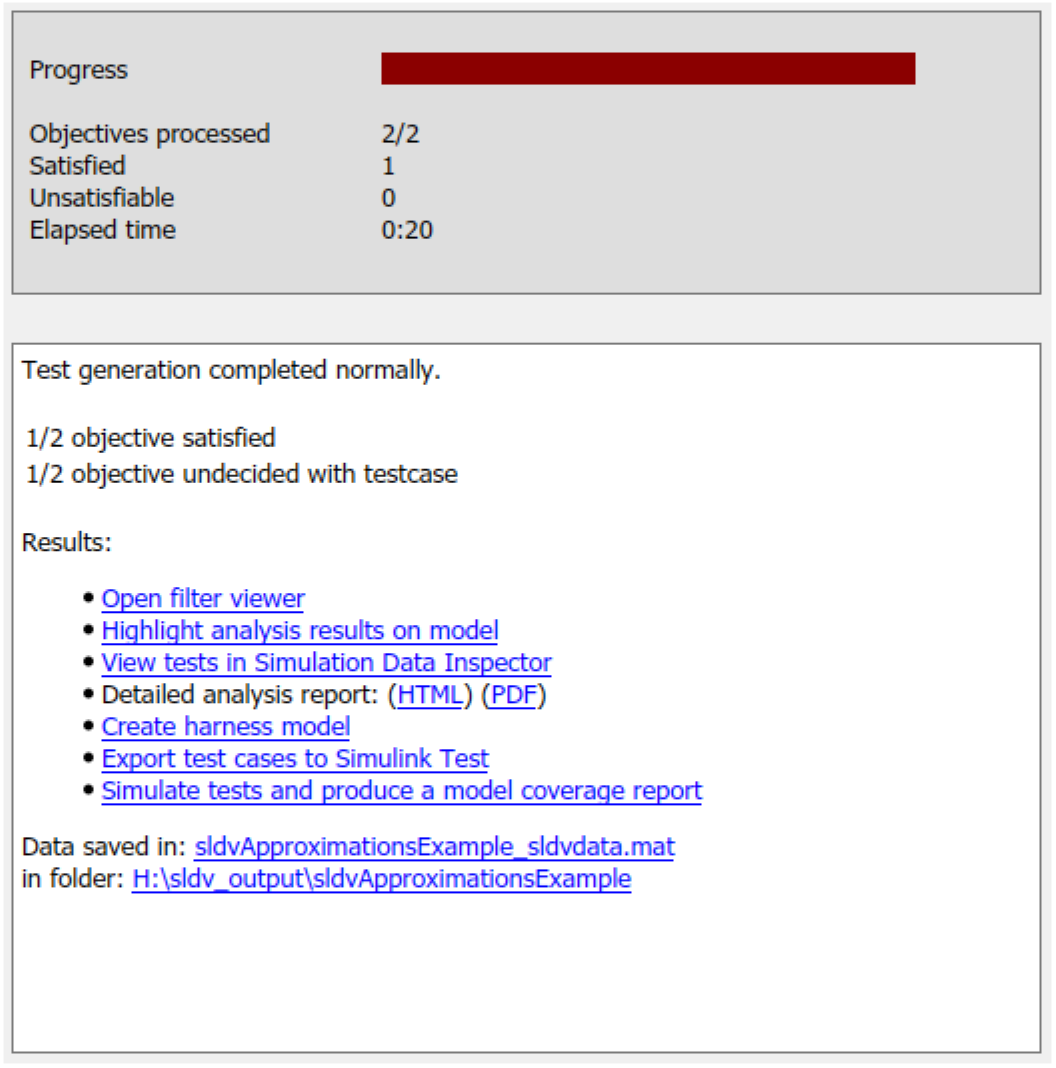
In this model, approximations occur due to floating point to rational number conversion during analysis. In the Simulink Design Verifier Report, the Objective Status chapter reports the objectives impacted by approximations for test generation and property proving analysis.

Copyright 2017-2019 The MathWorks, Inc.

Step 2: Perform Test Case Generation Analysis and Review Results

On the **Design Verifier** tab, click **Generate Tests**.

After the analysis completes, the Results Summary window displays that one objective is satisfied and one objective is undecided with test case.



Progress	
Objectives processed	2/2
Satisfied	1
Unsatisfiable	0
Elapsed time	0:20

Test generation completed normally.

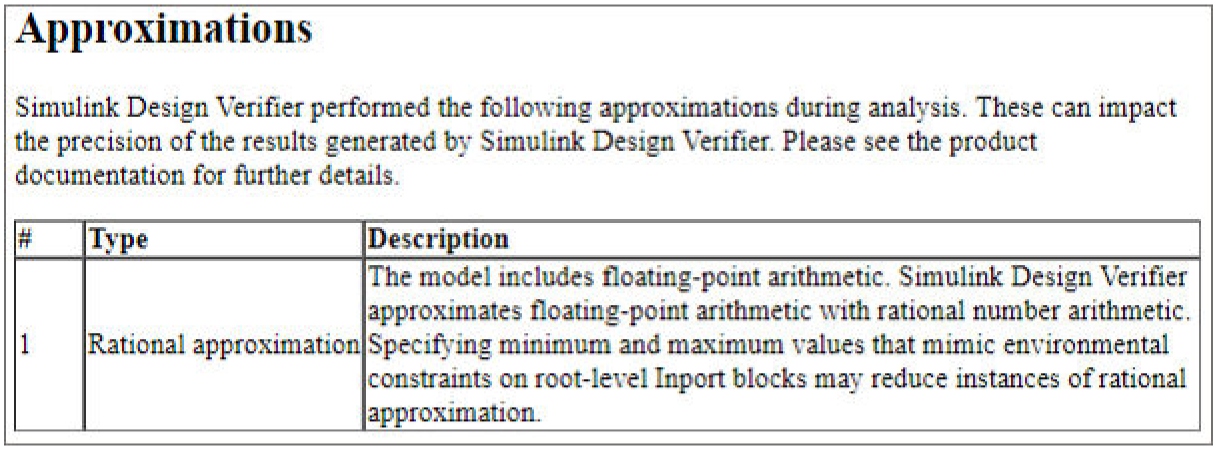
1/2 objective satisfied
1/2 objective undecided with testcase

Results:

- Open filter viewer
- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: sldvApproximationsExample_sldvdata.mat
in folder: H:\sldv_output\sldvApproximationsExample

To view the detailed analysis report, in the Results Summary window, click **HTML**. In the report, the Analysis Information chapter lists the approximations that were performed during analysis

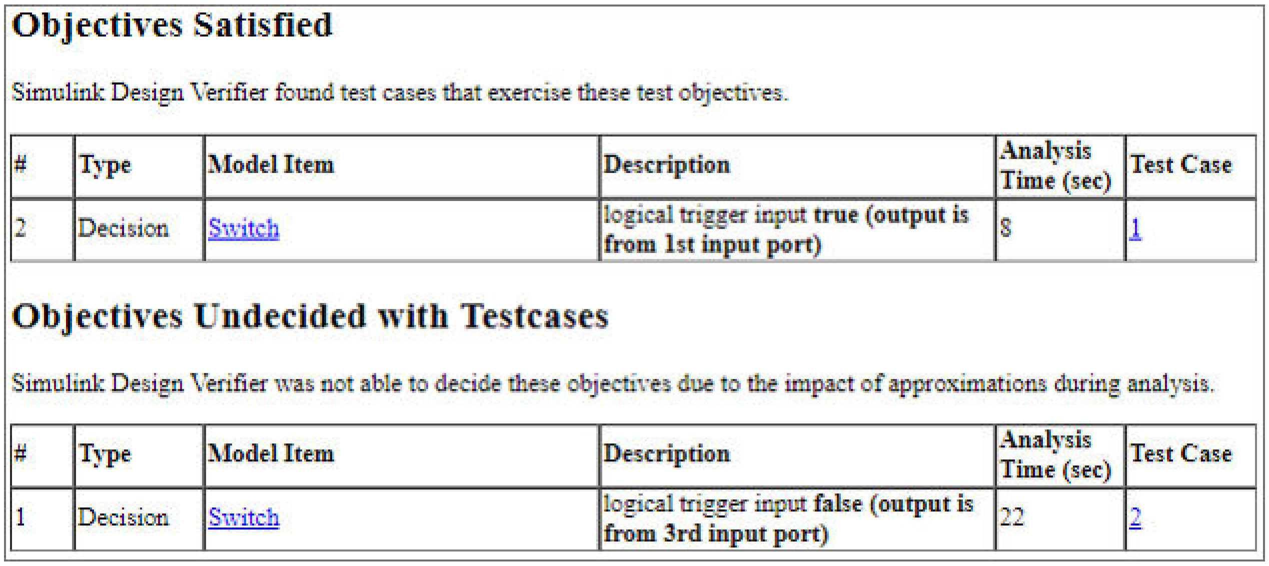


Approximations

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. Specifying minimum and maximum values that mimic environmental constraints on root-level Inport blocks may reduce instances of rational approximation.

The Objective Status chapter gives detailed description of the objectives.



Objectives Satisfied

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

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
2	Decision	Switch	logical trigger input **true (output is from 1st input port)**	8	1

Objectives Undecided with Testcases

Simulink Design Verifier was not able to decide these objectives due to the impact of approximations during analysis.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Switch	logical trigger input **false (output is from 3rd input port)**	22	2

Step 3: Run Additional Analysis by Reusing Cache

The undecided with test case objective is impacted by approximation, and to confirm this objective status you run additional analysis.

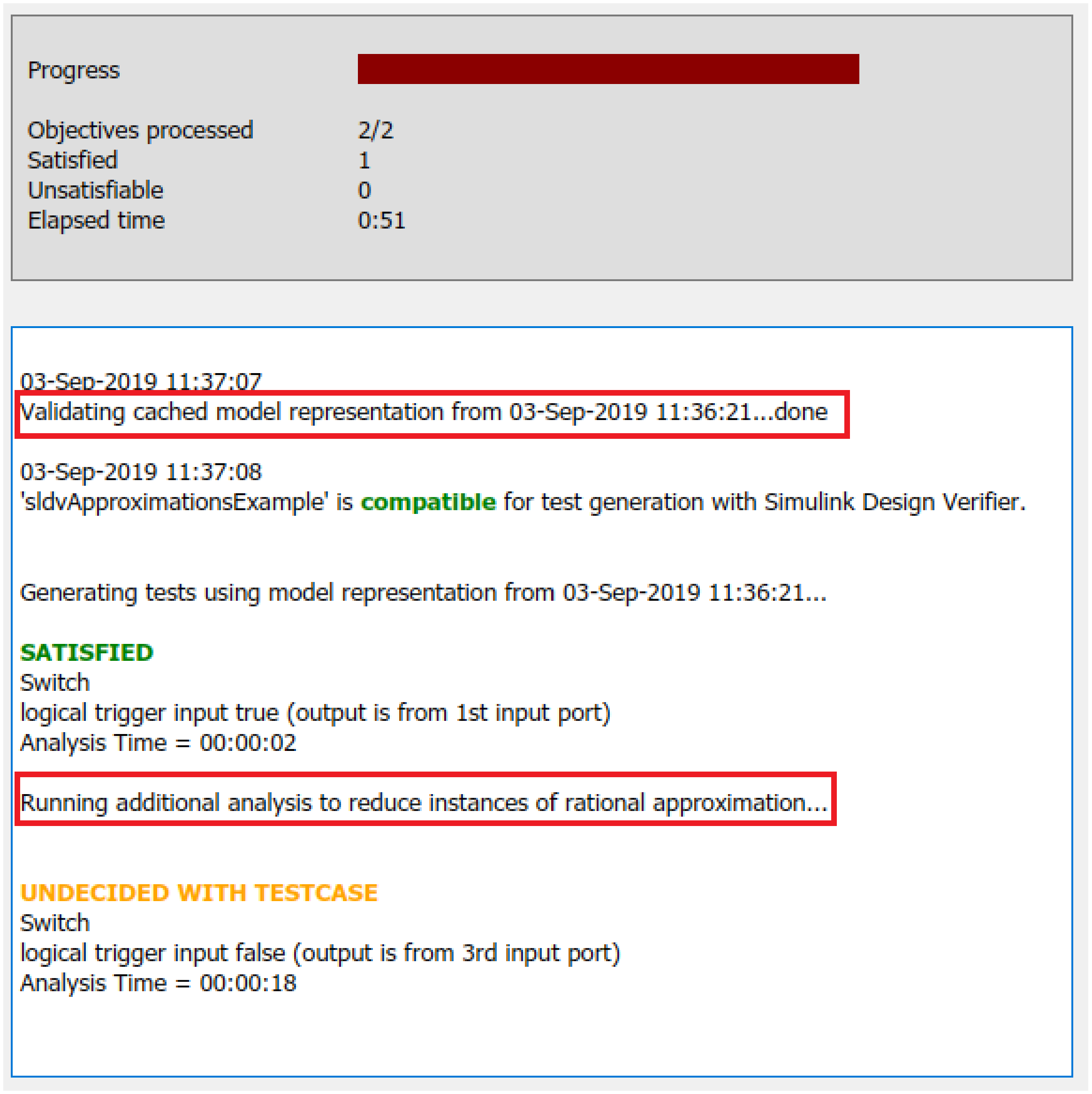
(a) On the **Design Verifier** tab, click **Test Generation Settings** > **Settings**.

(b) In the Configurations Parameters dialog box, on the Design Verifer pane, in Advanced parameters, set the **Rebuild model representation** option to `If change is detected` and enable **Run additional analysis to reduce instances of rational approximation** option. Click **OK**.

Note: If you create a new model, by default, the **Rebuild model representation** option is set to `If change is detected`.

(c) To perform test generation analysis, click **Generate Tests**. The existing cache is validated against the model and the analysis reuses the cache if no change is detected.

The Results Summary window displays that the cached model representation is validated and no change is detected. Hence, the analysis skips the compatibility check and reuses the model representation for analysis.



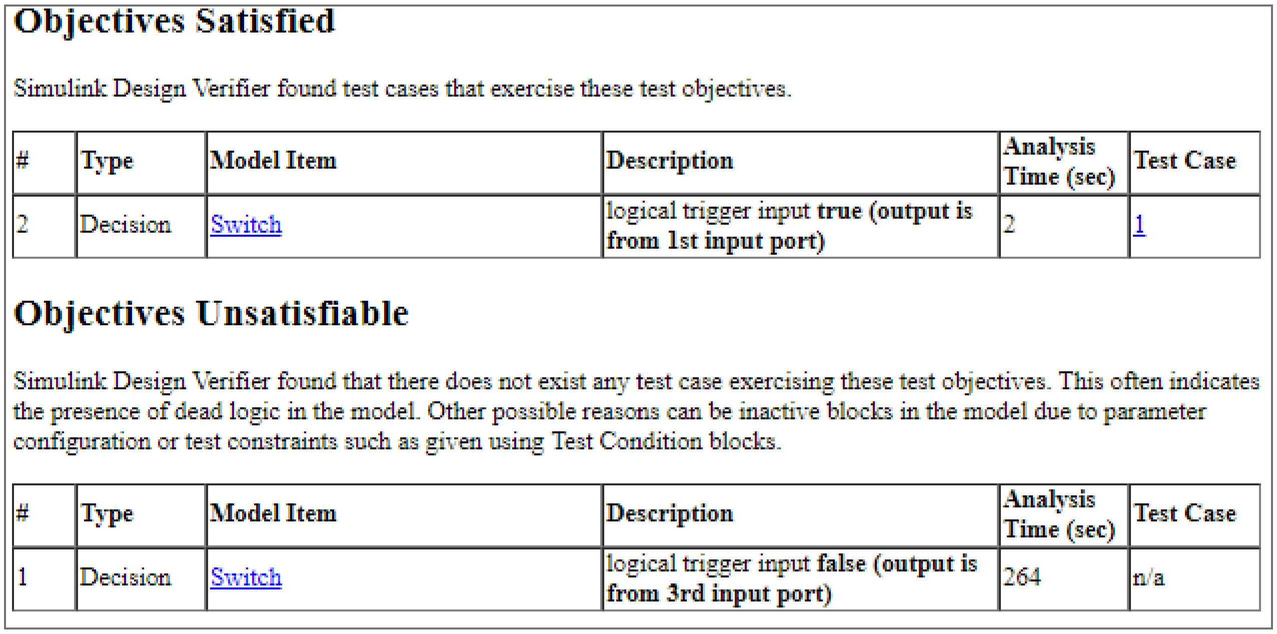
```
Progress

Objectives processed     2/2
Satisfied                1
Unsatisfiable            0
Elapsed time             0:51
```

```
03-Sep-2019 11:37:07
Validating cached model representation from 03-Sep-2019 11:36:21...done

03-Sep-2019 11:37:08
'sldvApproximationsExample' is compatible for test generation with Simulink Design Verifier.


Generating tests using model representation from 03-Sep-2019 11:36:21...

SATISFIED
Switch
logical trigger input true (output is from 1st input port)
Analysis Time = 00:00:02

Running additional analysis to reduce instances of rational approximation...


UNDECIDED WITH TESTCASE
Switch
logical trigger input false (output is from 3rd input port)
Analysis Time = 00:00:18
```

After the analysis completes, the Results Summary window displays that one objective is satisfied and one objective is unsatisfiable.

Step 4: Review Analysis Results

To view the detailed analysis report, in the Results Summary window, click **HTML**. In the report, the Objectives Status chapter gives detailed description of the objectives.


Objectives Satisfied

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

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
2	Decision	Switch	logical trigger input **true (output is from 1st input port)**	2	1

Objectives Unsatisfiable

Simulink Design Verifier found 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.

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Switch	logical trigger input **false (output is from 3rd input port)**	264	n/a

Related Topics

- "Model Representation for Analysis" on page 2-26
- "Run additional analysis to reduce instances of rational approximation" on page 15-15
- "Rebuild model representation" on page 15-13

Checking Compatibility with the Simulink Design Verifier Software

Check Model Compatibility

In this section...

With Simulink Design Verifier, you can analyze Simulink models to:

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

Before Simulink Design Verifier analyzes a model, the software checks whether the model is compatible for analysis. The model is compatible for analysis when:

- The model is compiled into an executable form.
- The model is compatible with code generation.
- The model performs zero-second simulation with no errors, that is the simulation start and stop time is `0`.

The software supports a broad range of Simulink and Stateflow software capabilities in your models. However, there are capabilities that the product does not support, described in "Support Limitations for Simulink Software Features" on page 3-16 and "Support Limitations for Stateflow Software Features" on page 3-20.

For more information on supported Simulink blocks, see "Supported and Unsupported Simulink Blocks in Simulink Design Verifier" on page 3-7.

Run Compatibility Check

Before the software begins an analysis, it checks the compatibility of your model, and then creates a model representation. The model representation includes the model artifacts that are used during analysis. For more information, see "Model Representation for Analysis" on page 2-26.

Before you start an analysis, you can run a compatibility check on your model by using one of these methods. When you use any of these methods, the model representation is always rebuilt.

- On the **Design Verifier** tab, in the **Analyze** section, click **Check Compatibility**.
- 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".
- To run the compatibility check programmatically at the command line or in a MATLAB program, use the `sldvcompat` function . For more information, see `sldvcompat`.
- To check compatibility of a Subsystem, right-click the Subsystem and select **Design Verifier > Check Subsystem Compatibility**.

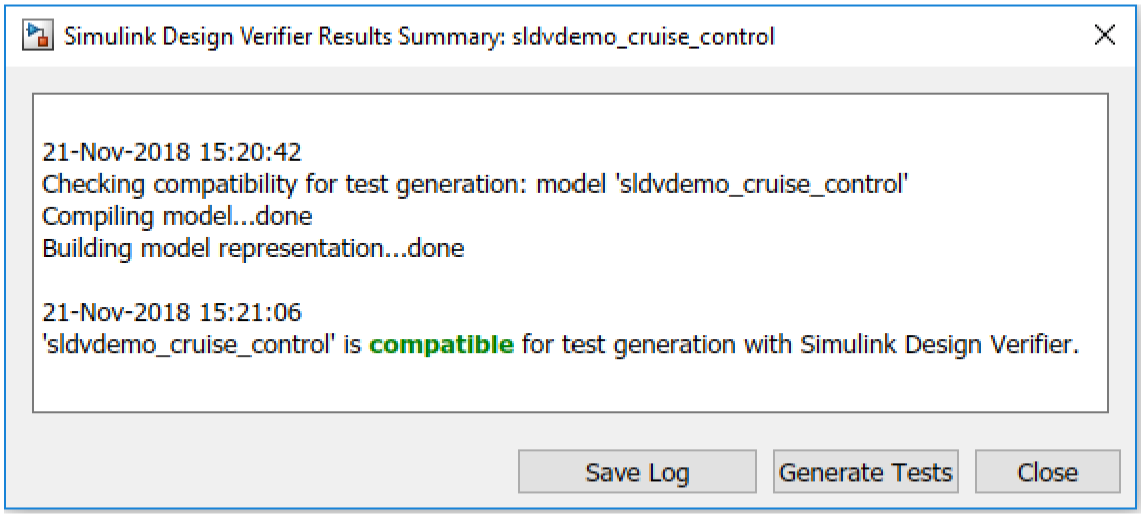
Compatibility Check Results

When you run a compatibility check on a model, the Results Summary window displays one of these results:

- "Model Is Compatible" on page 3-3
- "Model Is Incompatible" on page 3-3
- "Model Is Partially Compatible" on page 3-5

Model Is Compatible

If your model is compatible, you can continue with the analysis in the Results Summary window. For example, to continue the test generation analysis, click **Generate Tests**.

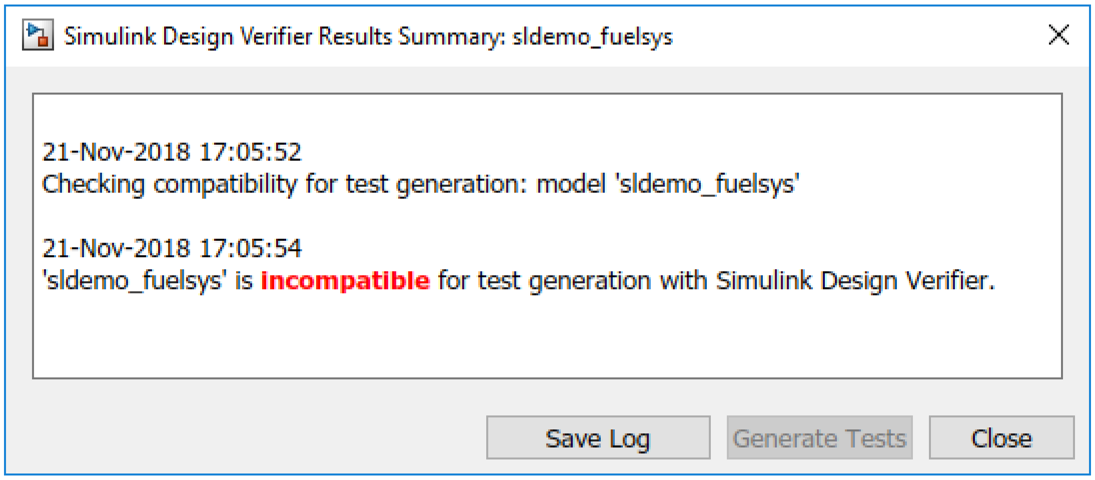

Simulink Design Verifier Results Summary: sldvdemo_cruise_control
Save Log
Generate Tests
Close

```
21-Nov-2018 15:20:42
Checking compatibility for test generation: model 'sldvdemo_cruise_control'
Compiling model...done
Building model representation...done

21-Nov-2018 15:21:06
'sldvdemo_cruise_control' is compatible for test generation with Simulink Design Verifier.
```

Note After you have completed the compatibility check, if you change the model, you cannot continue the analysis in the Results Summary window. If you change your model, rerun the compatibility check for analysis.

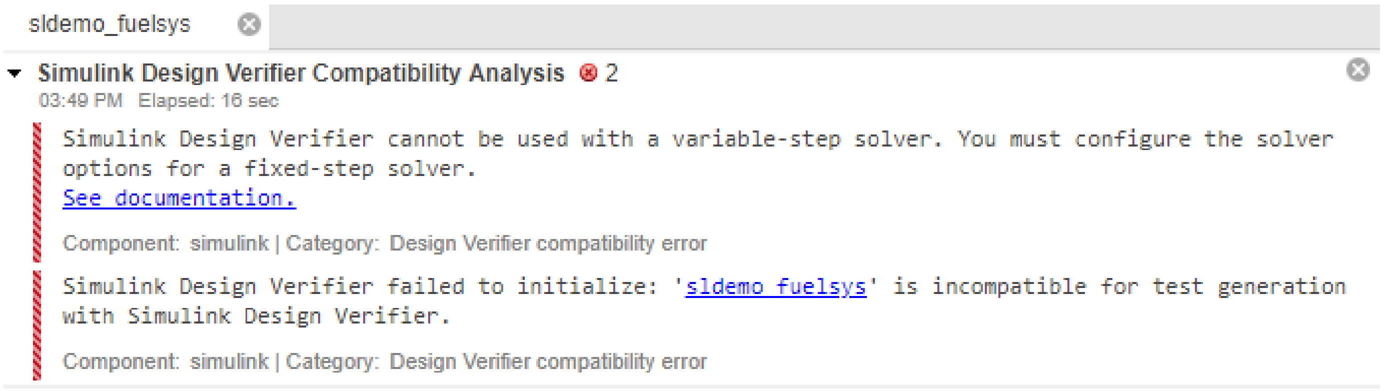
Model Is Incompatible

If the model is incompatible with the Simulink Design Verifier, you can identify and fix the incompatibilities through the Diagnostic Viewer messages. For more information, see "View Diagnostics" (Simulink).


Simulink Design Verifier Results Summary: sldemo_fuelsys

```
21-Nov-2018 17:05:52
Checking compatibility for test generation: model 'sldemo_fuelsys'

21-Nov-2018 17:05:54
'sldemo_fuelsys' is incompatible for test generation with Simulink Design Verifier.
```

- If your model uses a variable-step solver, configure the solver “Type” (Simulink) to `Fixed-step`.



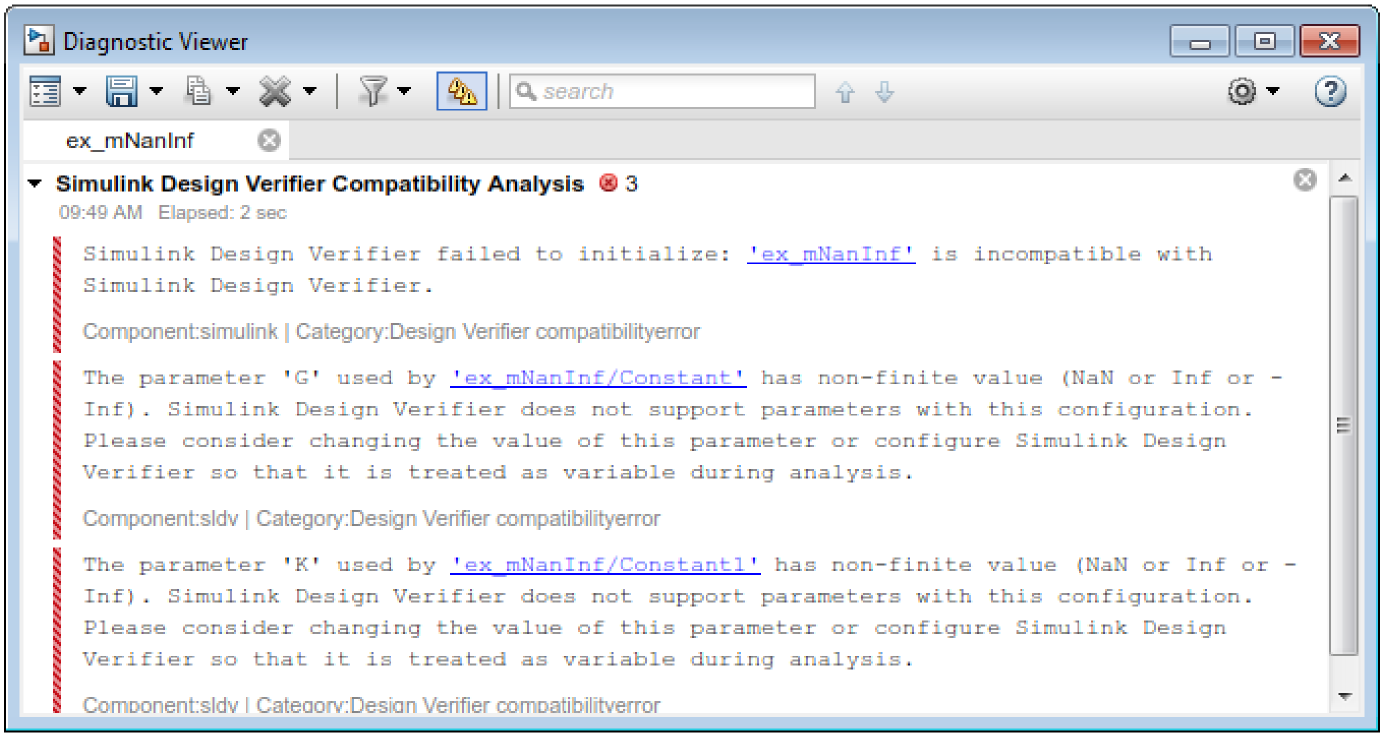
```
Simulink Design Verifier cannot be used with a variable-step solver. You must configure the solver
options for a fixed-step solver.
See documentation.
```

Component: simulink | Category: Design Verifier compatibility error

```
Simulink Design Verifier failed to initialize: 'sldemo_fuelsys' is incompatible for test generation
with Simulink Design Verifier.
```

Component: simulink | Category: Design Verifier compatibility error

- 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. For more information, see “Nonfinite Data” on page 2-18.



Simulink Design Verifier Compatibility Analysis 3
09:49 AM Elapsed: 2 sec

```
Simulink Design Verifier failed to initialize: 'ex_mNanInf' is incompatible with
Simulink Design Verifier.
```

Component:simulink | Category:Design Verifier compatibilityerror

```
The parameter 'G' used by 'ex_mNanInf/Constant' has non-finite value (NaN or Inf or -
Inf). Simulink Design Verifier does not support parameters with this configuration.
Please consider changing the value of this parameter or configure Simulink Design
Verifier so that it is treated as variable during analysis.
```

Component:sldv | Category:Design Verifier compatibilityerror

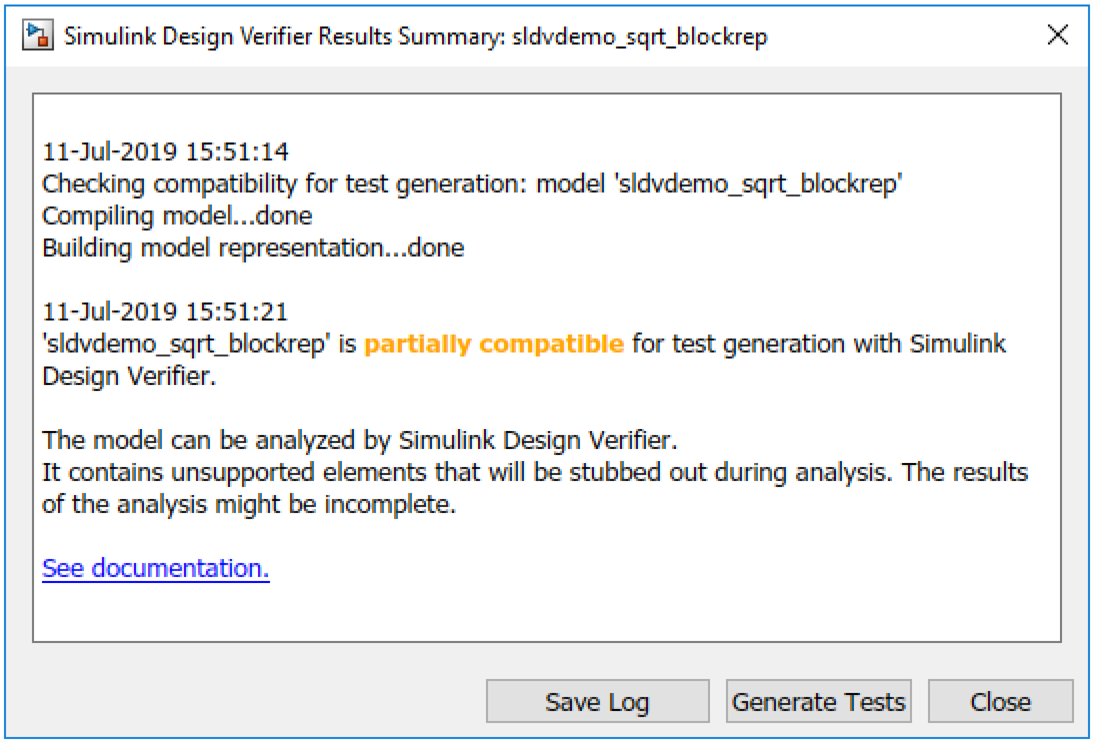
```
The parameter 'K' used by 'ex_mNanInf/Constant1' has non-finite value (NaN or Inf or -
Inf). Simulink Design Verifier does not support parameters with this configuration.
Please consider changing the value of this parameter or configure Simulink Design
Verifier so that it is treated as variable during analysis.
```

Component:sldv | Category:Design Verifier compatibilityerror

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 7-17.

Model Is Partially Compatible

A model is partially compatible if at least one model object in the model is incompatible. Simulink Design Verifier continues the analysis for partially compatible model by stubbing out the unsupported elements. By default, the “Automatic stubbing of unsupported blocks and functions” on page 15-14 option is set to `On`. For more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-8.


Simulink Design Verifier Results Summary: sldvdemo_sqrt_blockrep
Save Log
Generate Tests
Close

```
11-Jul-2019 15:51:14
Checking compatibility for test generation: model 'sldvdemo_sqrt_blockrep'
Compiling model...done
Building model representation...done

11-Jul-2019 15:51:21
'sldvdemo_sqrt_blockrep' is partially compatible for test generation with Simulink
Design Verifier.

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.

See documentation.
```

See Also

“Basic Workflow for Simulink Design Verifier” on page 1-21 | “Block Replacements for Unsupported Blocks” on page 4-8 | “Model Representation for Analysis” on page 2-26

Supported and Unsupported Simulink Blocks in Simulink Design Verifier

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 might 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
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
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`.

Block	Support Notes
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`. or • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than `5`.
Lookup Table Dynamic	Supported
Prelookup	Not supported when output is an array of buses
Sine	Supported

Math Operations Library

Block	Support Notes
Abs	Supported
Add	Supported
Algebraic Constraint	Supported
Assignment	Supported
Bias	Supported
Complex to Magnitude-Angle	Supported
Complex to Real-Imag	Supported
Divide	Supported
Dot Product	Supported
Find Nonzero Elements	Not supported
Gain	Supported
Magnitude-Angle to Complex	Supported
Math Function	Supported
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

Block	Support Notes
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

Block	Support Notes
Enabled Subsystem	Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-29. 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 output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-29. 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 output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-29. Not supported when the Function-Call Subsystem is invoked using function-call triggers passed via root-level Inport blocks. For more information see, “Export-Function Models Overview” (Simulink).

Block	Support Notes
If	Parameter configurations are not supported. The analysis ignores parameter configurations that you specify for an If block.
If Action Subsystem	Supported
In Bus Element	Supported
Inport	Supported
Model	Supported except for the limitations described in “Support Limitations for Model Blocks” on page 3-19.
Out Bus Element	Supported
Outport	Supported
Resettable Subsystem	Supported
Subsystem	Supported
Variant Transitions in Stateflow	Supported. Only the active variant is analyzed.
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 output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-29. 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
Out Bus Element	Supported
Scope	Supported
Stop Simulation	Not supported and not stubbable
Terminator	Supported
To File	Supported
To Workspace	Supported
XY Graph	Supported

Sources Library

Block	Support Notes
Band-Limited White Noise	Not supported

Block	Support Notes
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
In Bus Element	Supported if `Simulink.Bus` type is defined for the In Bus Element.
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
Signal Editor	Not supported
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
Initialize Function	Not supported
Interpreted MATLAB Function	Not supported
Level-2 MATLAB S-Function	For limitations, see “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-27.
MATLAB Function	For limitations, see “Support Limitations for MATLAB for Code Generation” on page 3-24.
Reset Function	Not supported
S-Function Builder	For limitations, see “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-27.

Block	Support Notes
Terminate Function	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 “Fixed Step Solvers in Simulink” (Simulink).
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 Loop Concepts” (Simulink).
Masked subsystem initialization functions	The software does not support models whose masked subsystem initialization modifies any attribute of any workspace parameter.
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” (Simulink).
Multiword fixed-point data types	The software does not support multiword fixed-point data types larger than 128 bits.

Not Supported	Description
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. • Simulink Design Verifier assumes zero start time for analysis and generates signal data that begins at zero. Zero start time might impact the reporting of the objective status. For example, in the test generation analysis, the software might report some objectives as `Undecided with Testcases`. For more information, see “Simulation Basics” (Simulink).
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.

Not Supported	Description
Simulink Strings	Models that contain blocks with string data types as block parameters are not supported. For more information, see “Simulink Strings” (Simulink).
Row-major Algorithms	The software does not support row-major algorithms for block simulation. For more information see, “Use algorithms optimized for row-major array layout” (Simulink). **1** The model is incompatible for Simulink Design Verifier analysis when in **Configuration Parameters**: • In **Code Generation** > **Interface** pane, the **Array layout** parameter is set to `Row-major`. • In **Math and Data types** pane, the parameter `Use algorithms optimized for Row-major array layout` is set to `on`. **2** Simulink Design Verifier will display incompatibility message if the model contains a MATLAB Function block that uses `coder.rowMajor` directive.

Support Limitations for Model Blocks

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 “Reference Protected Models from Third Parties” (Simulink).
- 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. Simulink will remove the algebraic loop if possible. If Simulink cannot eliminate the artificial algebraic loop, highlight the location of the algebraic loop by opening the **Modeling** tab and, in the **Compile** section, clicking **Update Model**.
  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 Loop Concepts” (Simulink).
- 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

Simulink 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

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

C or C++ Operators

The software does not support the `sizeof` operator, which the Stateflow software allows.

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 Library Functions in C Charts" (Stateflow).

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” (Stateflow).

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.

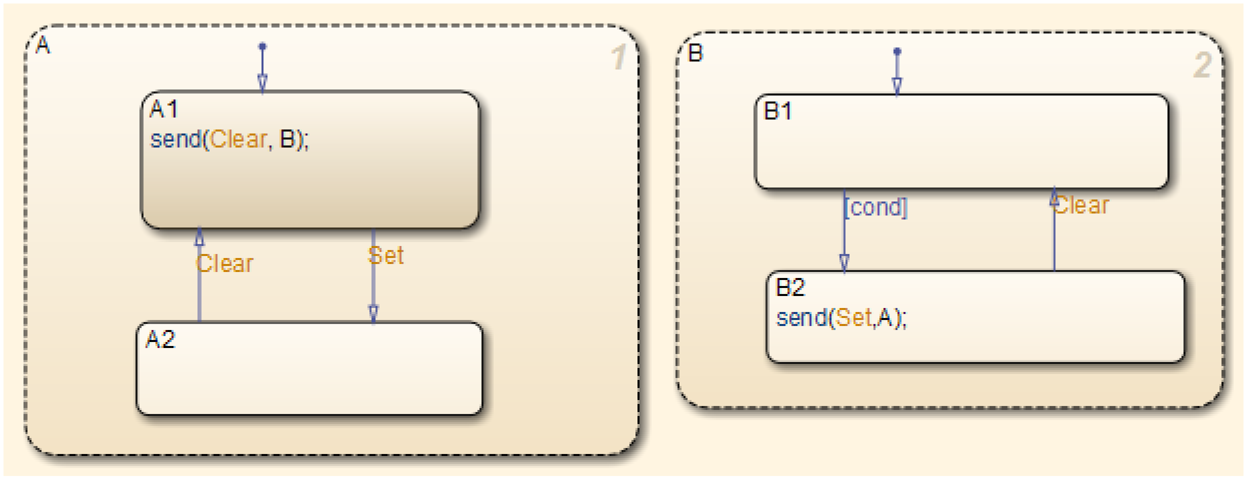
Note For information about avoiding recursion in Stateflow charts, see “Avoid Unwanted Recursion in a Chart” (Stateflow).

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.

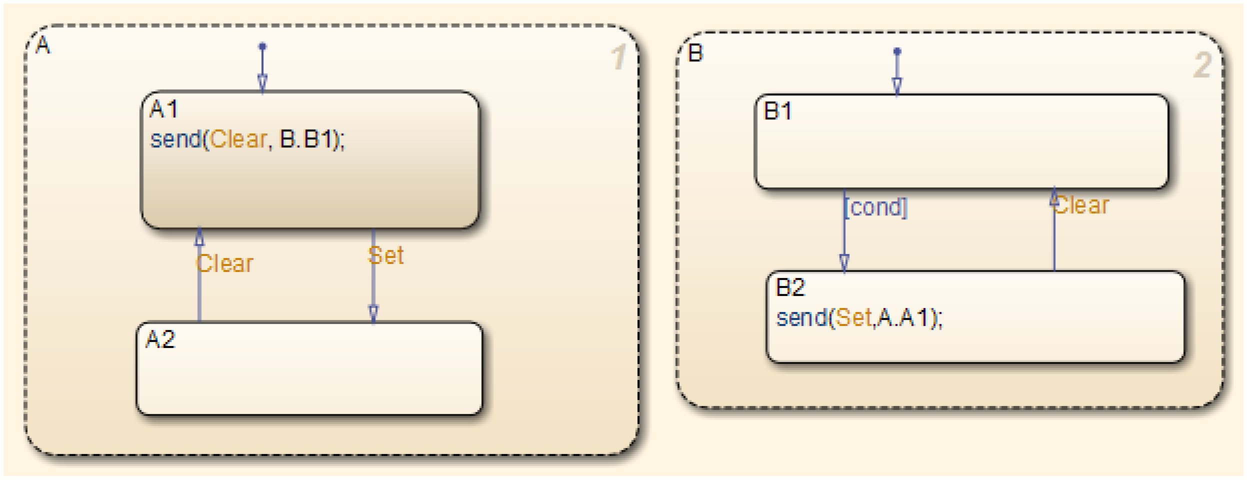
Note For information about cyclic behavior in Stateflow charts, see “Cyclic Behavior” (Stateflow).

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 Local Events to Synchronize Parallel States” (Stateflow).

Custom C/C++ Code

If your model consists of custom C/C++ code, Simulink Design Verifier supports analysis based on these settings:

- If you enable import custom code and custom code analysis options, the software supports custom C/C++ code for analysis. For more information, see “Import custom code” (Simulink) and “Enable custom code analysis” (Simulink).
- If you enable import custom code option and the custom code analysis option is set to `Off`, the model is compatible for analysis, but calls to the custom code are stubbed during analysis.
- If the import custom code option is set to `Off`, the custom code is not supported and the model is incompatible for analysis.

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” (Stateflow).

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 3-24
"Support Limitations for MATLAB for Code Generation Library Functions" on page 3-24

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
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 might 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" (Simulink).

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 Toolbox™ functions	Not supported.	
Complex number functions	Supported.	
Computer Vision Toolbox™ functions	Not supported.	
Data type functions	—	
Derivative and Integral functions	Not supported.	
Discrete math functions	—	
Error handling functions	Supported with the following limitations:	
	`assert`	Supported, but does not behave like a Proof Objective block.
Exponential functions	Supported.	
Filtering and convolution functions	Supported with the following limitations:	
	`detrend`	Not supported.
Fixed-Point Designer functions	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.
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.

Function Category	Support Notes	
	`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	—	
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 and Considerations for S-Functions and C/C++ Code

In this section...

Enabling S-Functions in Simulink Design Verifier

Simulink Design Verifier supports test case generation for code generated with Embedded Coder®. Simulink Design Verifier also supports error detection, 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 “About C MEX S-Functions” (Simulink).

Support Limitations for S-Functions and C/C++ Code

- Simulink Design Verifier does not support S-Functions or C/C++ code containing:
  - Continuous states. Simulink Design Verifier does not analyze such code.
  - Zero-crossing functions. Simulink Design Verifier ignores such code during analysis.
  - Constants that describe INF or NaN objects. Simulink Design Verifier considers such code 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 your code contains floating-point overflow errors.
- You must specify that the signal elements entering the ports of S-Functions compiled with `slcovmex` are contiguous. Use the `SimStruct` function `ssSetInputPortRequiredContiguous`.

Considerations for Enabling S-Functions and C/C++ Code in Simulink Design Verifier

- When performing property proving or test generation analysis for models with enabled S-Functions or C/C++ code generated with Embedded Coder, Simulink Design Verifier assumes that the code contains no run-time errors. If the code contains run-time errors such as division by zero, access to non-initialized variables or array out of bounds, the property proving or test generation analysis can produce incorrect results. Code that has been checked by Polyspace and is free of run-time errors provide correct results in Simulink Design Verifier analysis.

To avoid incorrect results that are produced due to run-time errors, perform design error detection analysis first, and then perform property proving or test generation analysis.

- If Simulink Design Verifier cannot determine the size of arrays in your code (for instance for arrays that are dynamically allocated with non-constant size), Simulink Design Verifier assumes an upper bound for the array. Ensure that the given upper bound is appropriate.
- If you do not enable Simulink Design Verifier support for an S-function, Simulink Design Verifier stubs the S-function. With S-function support enabled, Simulink Design Verifier analyzed the content of the S-function to get more detailed information. Sometimes, Simulink 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 is 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 Simulink 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” | “Generate Test Cases for Embedded Coder Generated Code” on page 7-21

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

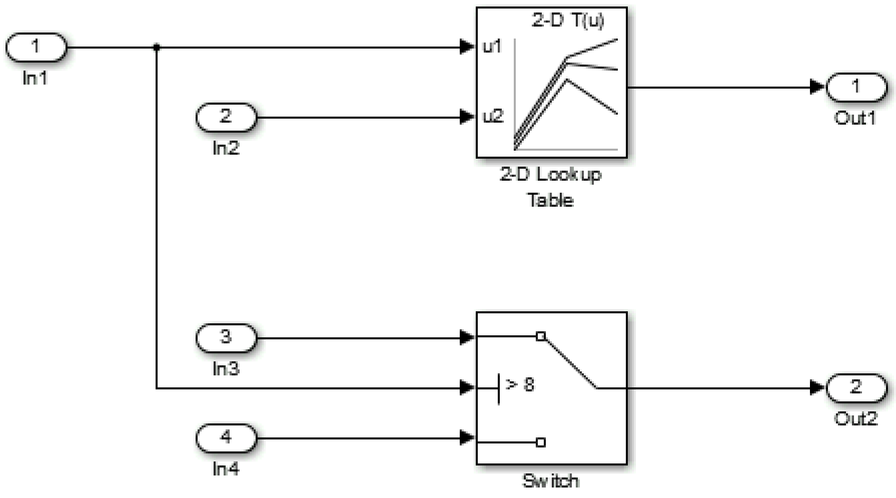
Note You can use automatic stubbing as an alternative to block replacements to resolve incompatibilities. Automatic stubbing replaces unsupported blocks with elements that have the same interface. For more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-8.

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” (Simulink Check). 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.

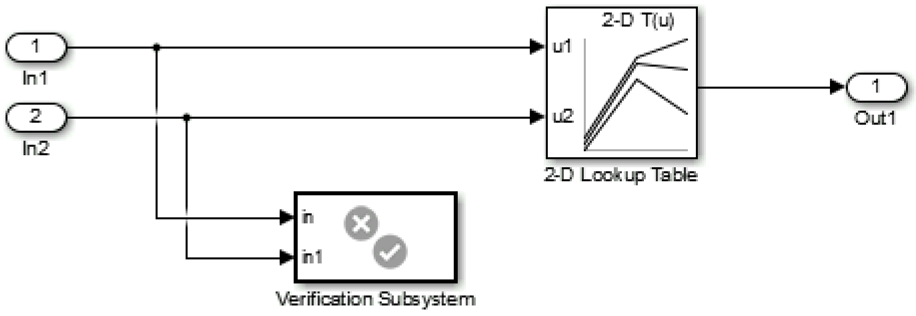


Tests generated without replacing the 2D Lookup Table satisfy two objectives: that the trigger is not greater than the Switch block threshold 8, and that the trigger is greater than the Switch block threshold 8.

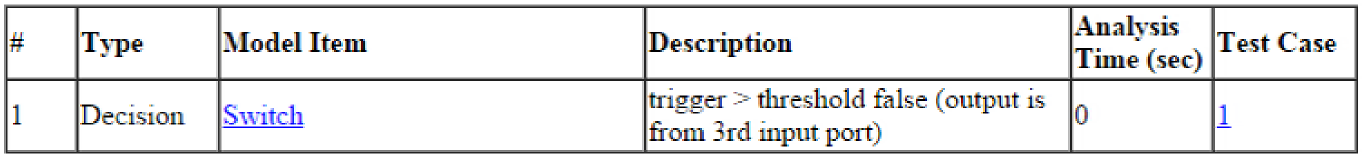
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Switch	trigger > threshold false (output is from 3rd input port)	1	1
2	Decision	Switch	trigger > threshold true (output is from 1st input port)	1	2

Objective Satisfied

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.

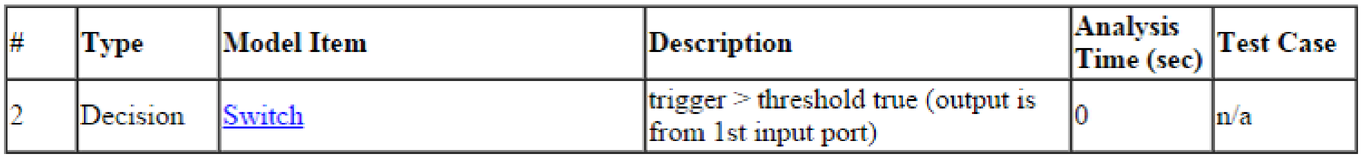


The verification subsystem constrains the analysis within the breakpoint bounds of the table. The additional constraints prevent generating tests that exercise the second objective for the Switch block. The condition that the input signal `In1 > 8` is unsatisfiable.



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

Objectives Satisfied



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

Objectives Proven Unsatisfiable

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).

`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).

File Name	Description

`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.

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 “Block Replacements for Unsupported Blocks” on page 4-8.)
- The function that represents a block replacement rule must be on the MATLAB search path.

Block Replacements for Unsupported Blocks

This example shows how to use Simulink® Design Verifier™ functions to replace unsupported blocks and to how customize test vector generation for specific requirements.

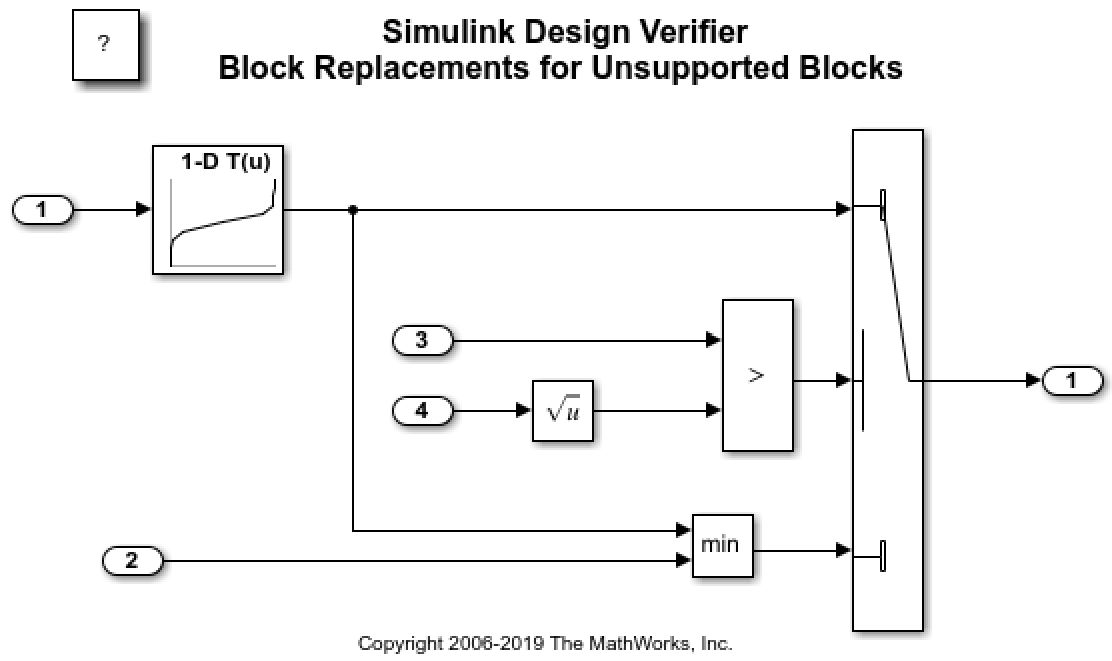
Model with an Unsupported Block

The example model includes a Switch block whose output is controlled by a Sqrt block. For each switch position, the output of the model is calculated by a 1-D Lookup Table block. For this model, the example concentrates on generating test cases that satisfy the following:

1. Achieve 100% lookup table coverage.

2. Test vectors demonstrate each Switch block position when the values of its first and third input ports differ.

```
open_system('sldvdemo_sqrt_blockrep');
```


?
Simulink Design Verifier
Block Replacements for Unsupported Blocks
1-D T(u)
1
3
4
>
√u
1
2
min
Copyright 2006-2019 The MathWorks, Inc.

Checking Model Compatibility

Since the `sqrt` function is not supported, this model is partially compatible with Simulink Design Verifier.

```
sldvcompat('sldvdemo_sqrt_blockrep');
```

```
Checking compatibility for test generation: model 'sldvdemo_sqrt_blockrep'
Compiling model...done
Building model representation...done

'sldvdemo_sqrt_blockrep' is partially compatible for test generation with Simulink Design Verifie
```

```
The model can be analyzed by Simulink Design Verifier.
It contains unsupported elements that will be stubbed out during analysis. The results of the ana
<br> <a href="matlab:helpview([docroot '/sldv/sldv.map'], 'msginfo_stubbing')">See documentation
```

Creating a Custom Block Replacement Rule to Work Around the Incompatibility

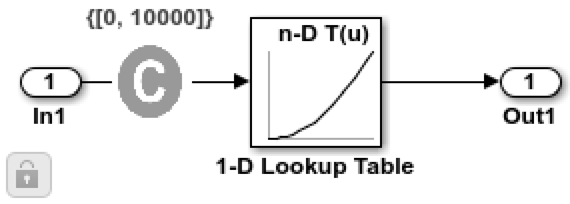
This model can be analyzed for test generation by automatically stubbing the unsupported Sqrt block. However, test cases cannot be generated for the Switch block positions because Simulink Design Verifier does not understand the Sqrt block and the output of this block is effecting the Switch block. Since you want test cases for the Switch block, you need to replace the Sqrt block with a supported block that is functionally equivalent. The library block `sldvdemo_custom_blockreplib` shown below constrains the input signal to the range `[0 10000]` and approximates the `sqrt` function by using a 1-D Lookup Table block.

The table data was calculated to match the values of `sqrt`, with a maximum error of `0.2` in the range `[0 10000]`. Refer to the mask initialization pane of the block `Sqrt_Approx` in the library `sldvdemo_custom_blockreplib` for the values of the lookup table data.

The replacement rule is in defined the MATLAB-file `sldvdemo_custom_blkrep_rule_sqrt.m`. Since the replacement block `sldvdemo_custom_blockreplib` for the Sqrt block is only valid for double or single types, this rule ensures that these conditions are satisfied before allowing a block replacement.

```
function rule = sldvdemo_custom_blkrep_rule_sqrt

    rule = SldvBlockReplacement.blockreprule;
    rule.fileName = mfilename;

    rule.blockType = 'Sqrt';

    rule.replacementPath = sprintf('sldvdemo_custom_blockreplib/Sqrt_Approx');

    rule.replacementMode = 'Normal';

    parameter.OutMin = '$original.OutMin$';
    parameter.OutMax = '$original.OutMax$';
    parameter.OutDataTypeStr = '$original.OutDataTypeStr$';
    rule.parameterMap = parameter;

    rule.isReplaceableCallBack = @replacementTestFunction;

end

function out = replacementTestFunction(blockH)

    out = false;
    acceptedOutDataTypeStr = {'double','single',...
                      'Inherit: Inherit via back propagation',...
                      'Inherit: Same as input'};
    I = strmatch(get_param(blockH,'OutDataTypeStr'),acceptedOutDataTypeStr,'exact');
    if ~isempty(I)

        portDataTypes = get_param(blockH,'CompiledPortDataTypes');

        out = any(strcmp(portDataTypes.Inport,{'double','single'})) &&  ...
              strcmp(portDataTypes.Inport,portDataTypes.Outport);
    end
end
```

```
open_system('sldvdemo_custom_blockreplib');
open_system('sldvdemo_custom_blockreplib/Sqrt_Approx/1-D Lookup Table');
```



Configuring Simulink® Design Verifier™ Options for Block Replacement

You will run Simulink Design Verifier in test generation mode with block replacements enabled. In order to generate test cases for positions of Switch block, you must use the custom replacement rule `sldvdemo_custom_blkrep_rule_sqrt.m`.

Since you are also interested in lookup table coverage, you need the built-in block replacement `blkrep_rule_lookup_normal.m`, which inserts test objectives for each interval and breakpoint value for a 1-D Lookup Table block. Moreover, you need the built-in rule `blkrep_rule_switch_normal.m`, which requires that each switch position be exercised when the values of the first and third input ports differ. Please refer to the Block Replacement in the Simulink Design Verifier documentation for a list of all built-in replacement rules.

The analysis will run for a maximum of 30 seconds and produce a harness model. Report generation is also enabled. Other Simulink Design Verifier options are set to their default values.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.MaxProcessTime = 80;
opts.BlockReplacement = 'on';
opts.BlockReplacementRulesList = ['sldvdemo_custom_blkrep_rule_sqrt.m,' ...
                                  'blkrep_rule_lookup_normal.m,'...
                                  'blkrep_rule_switch_normal.m'];
opts.SaveHarnessModel = 'on';
opts.ModelReferenceHarness = 'on';
opts.SaveReport = 'on';
```

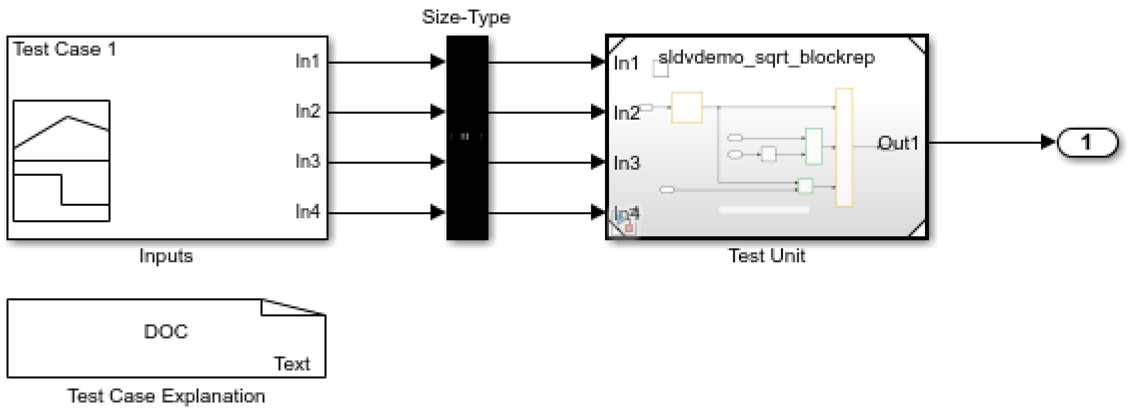
Executing Test Generation with Block Replacements

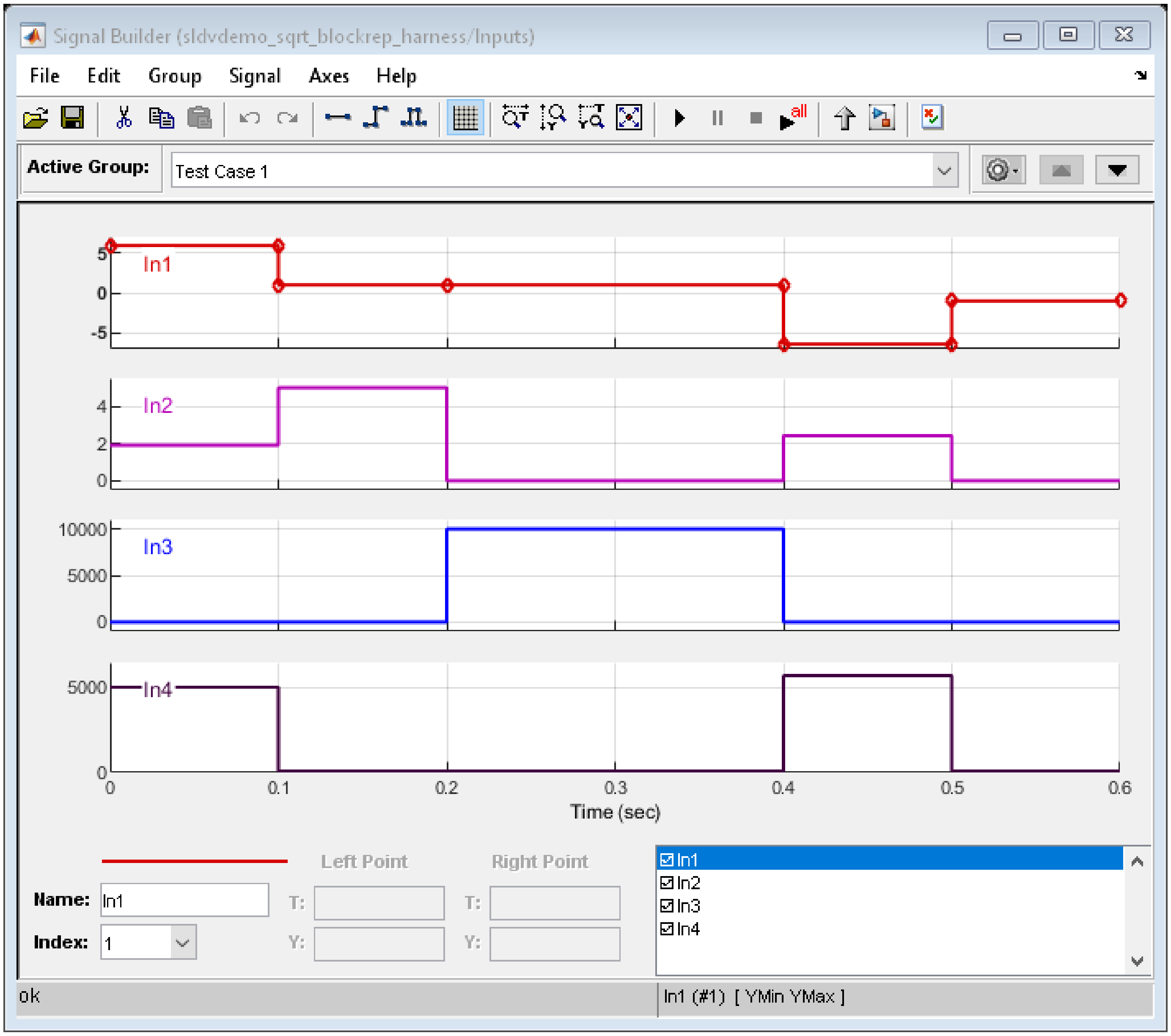
The `sldvrun` function analyzes the model using the settings defined in a `sldvoptions` object `opts`. The generated report includes a chapter summarizing block replacements performed on the model.

```
[status,fileNames] = sldvrun('sldvdemo_sqrt_blockrep', opts, true);
```

```
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
```

```
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
Warning: Cannot tune parameters in block 'sldvdemo_sqrt_blockrep/1-D Lookup
Table' while using the Lookup Table Coverage option. The Coverage tool will
ignore the new parameter values.
```


Test Case 1
In1
In2
In3
In4
Inputs
Size-Type
sldvdemo_sqrt_blockrep
Out1
Test Unit
1
DOC
Text
Test Case Explanation

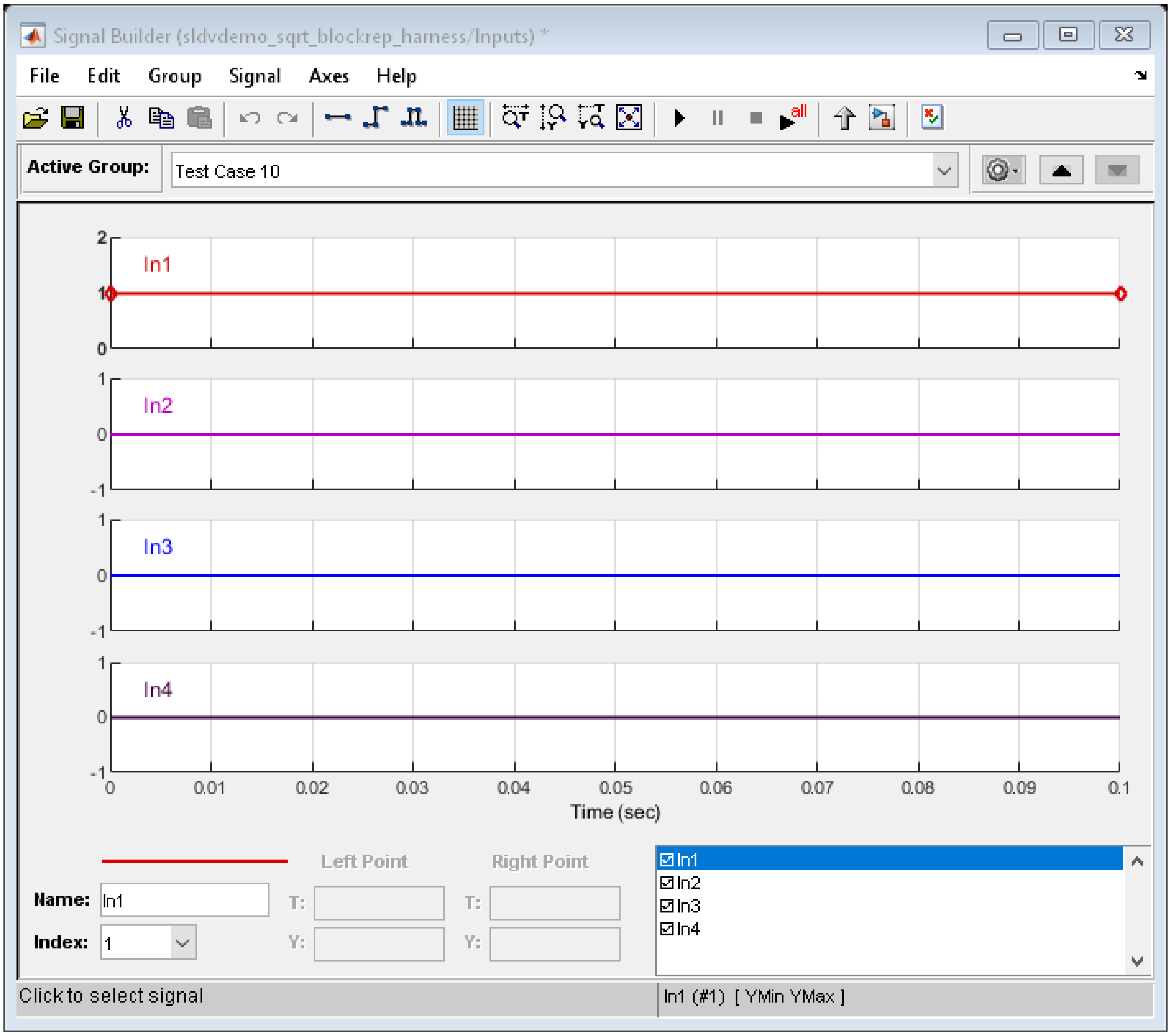


Executing Tests in the Harness Model

Enable the lookup table coverage metric and then run the test cases using the harness model. You can also execute the suite of tests by clicking the "Run all" button on the Signal Builder dialog box after enabling lookup table coverage from the "Analysis" > "Coverage" > "Settings" menu.

The coverage report shown below indicates that you can reach 100% lookup table coverage with the test vectors that Simulink Design Verifier generated.

```
[harnessModelPath,harnessModel] = fileparts(fileNames.HarnessModel);
set_param(harnessModel,'covMetricSettings','dcmte');
sldvdemo_playall(harnessModel);
```



Name: In1

Index: 1

Click to select signal

In1 (#1) [YMin YMax]

Clean Up

To complete the example, close all models and remove the files that Simulink Design Verifier generated.

```
close_system('sldvdemo_custom_blockreplib');
close_system(fileNames.HarnessModel,0);
close_system(fileNames.BlockReplacementModel,0);
close_system('sldvdemo_sqrt_blockrep',0);
delete(fileNames.HarnessModel);
delete(fileNames.BlockReplacementModel);
delete(fileNames.DataFile);
```

Specifying Parameter Configurations

Parameter Constraint Values

In this section...

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 design error detection or property proving analysis 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 5-9.

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 5-11.
- 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 5-8.
- Import and export parameter configurations from MATLAB code files. See "Store Parameter Constraints in MATLAB Code Files" on page 5-18.

Data Types in Parameter Configurations

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

- "Structures as Parameters not Supported" 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
- "Complex Data as Parameters not Supported" on page 5-3

Structures as Parameters not Supported

If the data type of a parameter in the MATLAB workspace is `struct`, Simulink Design Verifier does not support generating 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.

Complex Data as Parameters not Supported

If the data type of a parameter in the MATLAB workspace is complex, Simulink Design Verifier does not support generating 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.

See Also

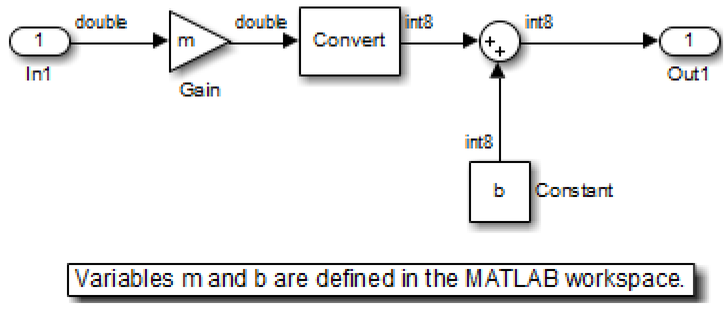
More About

- “Specify Parameter Constraint Values for Full Coverage” on page 5-9
- “Extend Existing Test Cases After Applying Parameter Configurations” on page 5-35

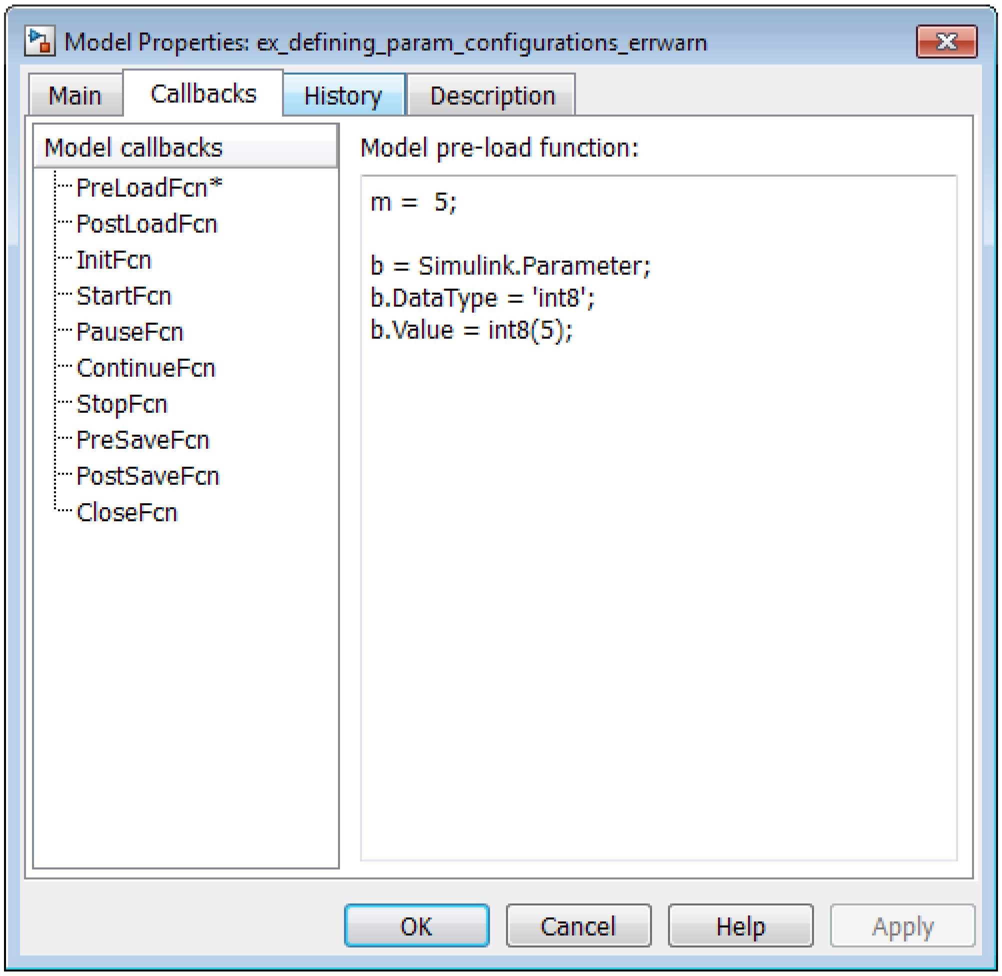
Define Constraint Values for Parameters

In this section...
"Find Parameters and Autogenerate Constraints" on page 5-5
"Edit Parameter Constraints" on page 5-7
"Highlight Constrained Parameters in Model" on page 5-8

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.



```
m =  5;

b = Simulink.Parameter;
b.DataType = 'int8';
b.Value = int8(5);
```

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.

On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.

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

Enable the Parameter Table.

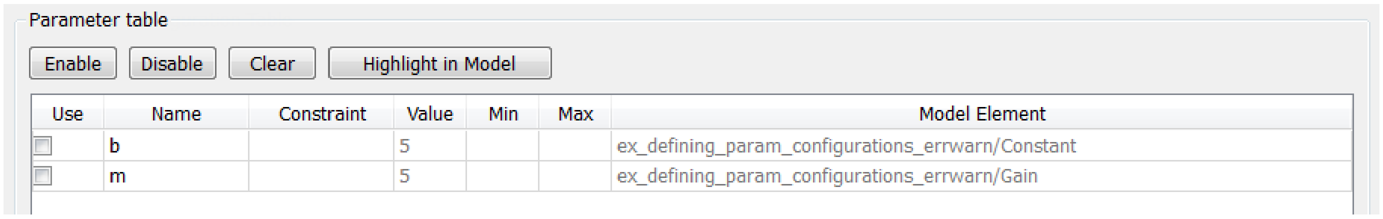
In the **Parameters** pane, select **Enable parameter configuration** and **Use parameter table**.

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.



Use	Name	Constraint	Value	Min	Max	Model Element
☐	b		5			ex_defining_param_configurations_errwarn/Constant
☐	m		5			ex_defining_param_configurations_errwarn/Gain

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 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.

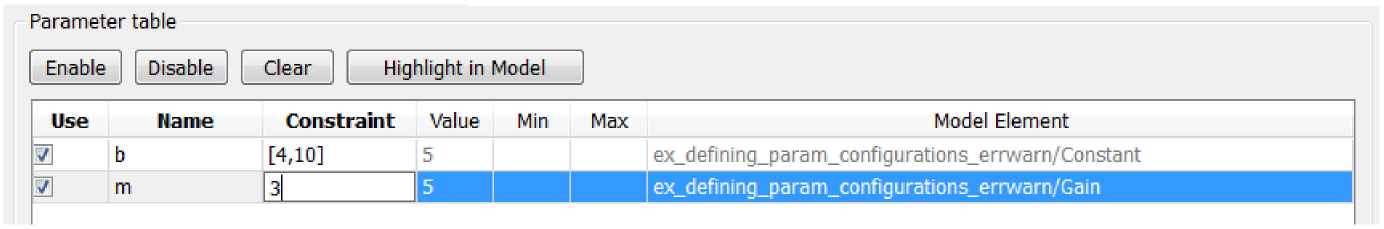
Edit Parameter Constraints

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.



Use	Name	Constraint	Value	Min	Max	Model Element
✓	b	[4,10]	5			ex_defining_param_configurations_errwarn/Constant
✓	m	3	5			ex_defining_param_configurations_errwarn/Gain

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.

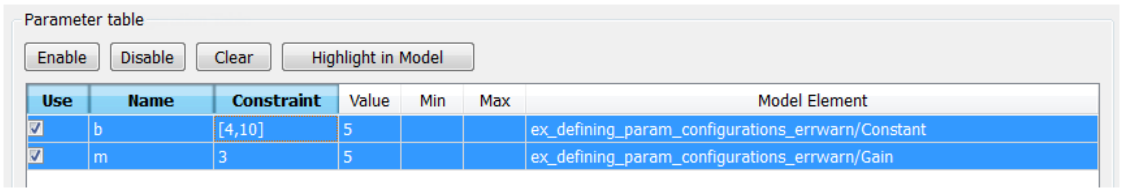
To clear multiple parameter configurations at once, shift-click to select multiple rows, and click **Clear**.

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.

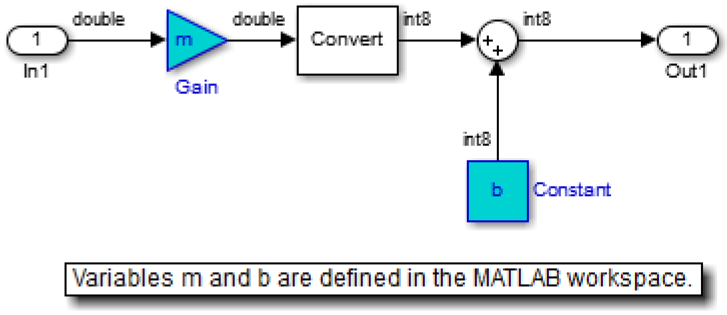
To select a parameter, click anywhere inside the **Name** or **Constraint** columns for either parameter. Shift-click to select multiple parameters.



Use	Name	Constraint	Value	Min	Max	Model Element
✓	b	[4,10]	5			ex_defining_param_configurations_errwarn/Constant
✓	m	3	5			ex_defining_param_configurations_errwarn/Gain

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

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.

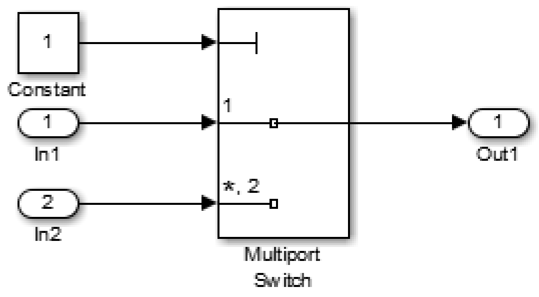
Task	Description	See...
1	Construct the example model.	“Construct Example Model” on page 5-9
2	Specify a variable as the value of a Constant block parameter.	“Parameterize Constant Block” on page 5-10
3	Constrain the value of the variable that the Constant block specifies.	“Autogenerate Parameter Constraint” on page 5-11
4	Generate test cases for your model and interpret the results.	“Analyze Example Model” on page 5-12
5	Simulate the test cases and measure the resulting decision coverage.	“Simulate Test Cases” on page 5-14

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 On the **Simulation** tab, click the arrow on the right of the **Prepare** section and click **Model Settings**.
6 In the Configuration Parameters dialog box, select the **Solver**. Under **Solver selection**, set the **Type** option to `Fixed-step`, and then set the **Solver** option to `discrete (no continuous states)`.
7 In the **Diagnostics** pane, 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

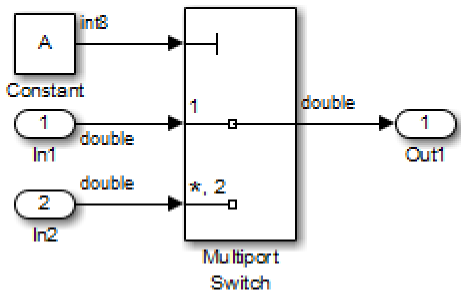
Preload the value of the MATLAB workspace variable `A` referenced by the Constant block:

1 On the **Modeling** tab, select **Model Settings** > **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 variable A to specified values.

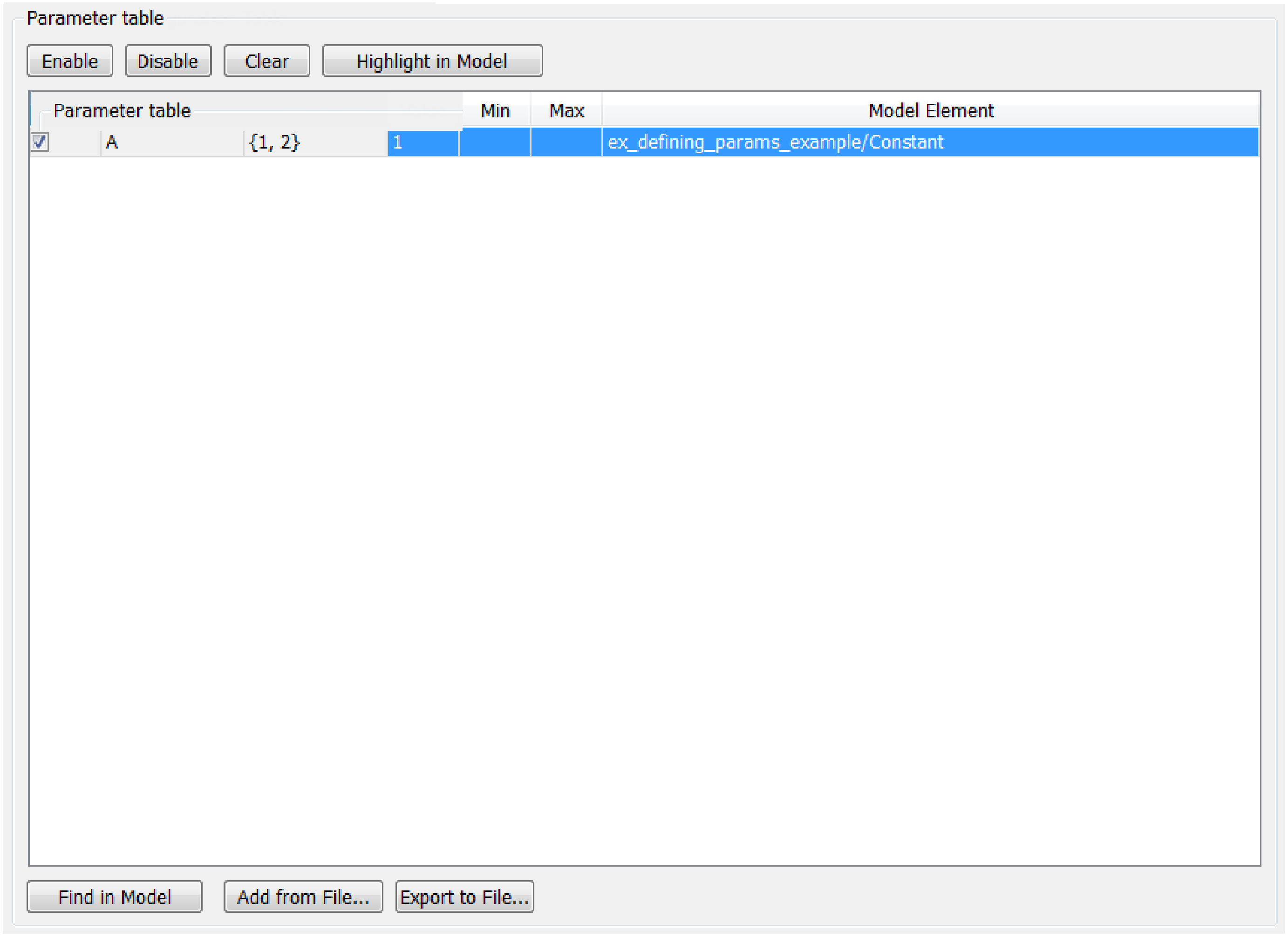
1. On the **Apps** tab, click the arrow on the right of the **Apps** section.

   Under **Model Verification, Validation, and Test**, click **Design Verifier**.
2. On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
3. In Configuration Parameters dialog box, select **Design Verifier** > **Parameters**.
4. Select **Enable parameter configuration**.
5. Select **Use parameter table**.
6. 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.



	Parameter table			Min	Max	Model Element
✓	A	{1, 2}	1			ex_defining_params_example/Constant

7 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.

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

9 Save your model.

Analyze Example Model

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

1 On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**. Click **Generate Tests**.

   Simulink Design Verifier analyzes your model to generate test cases.

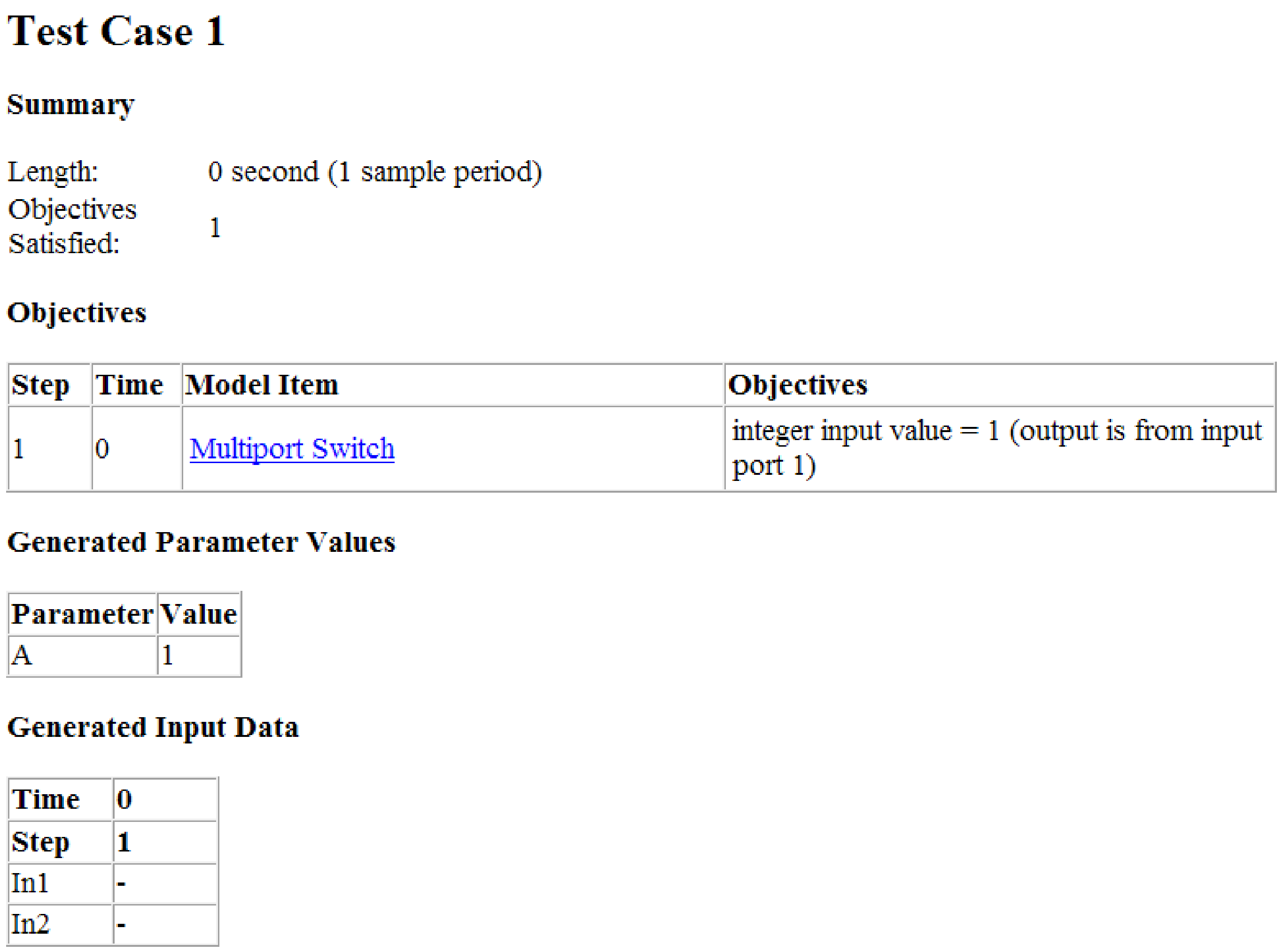
2 When the software completes its analysis, in the Simulink Design Verifier Results Summary window, select **Generate detailed analysis report**.

   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

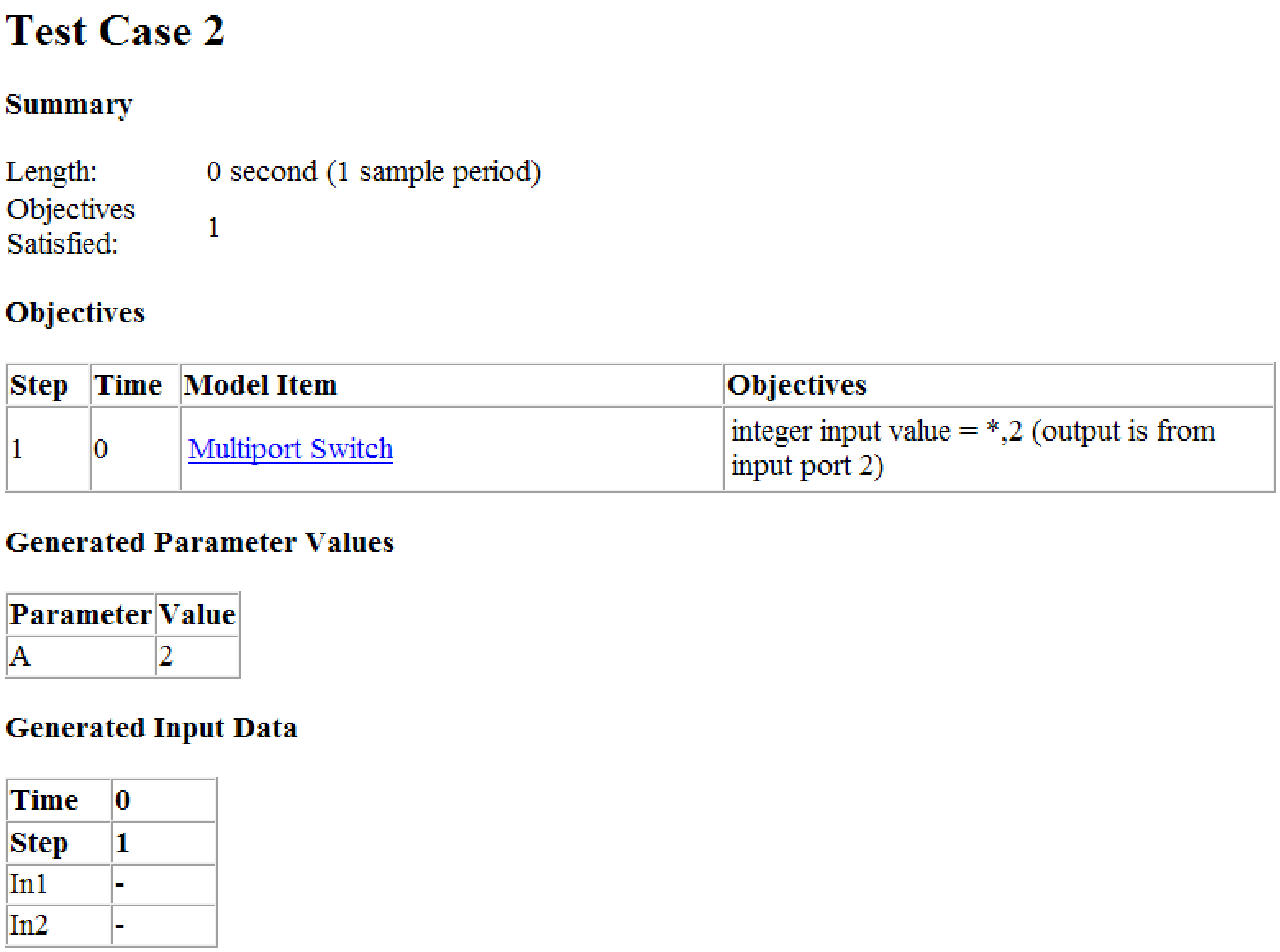
Parameter	Value
A	1

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.

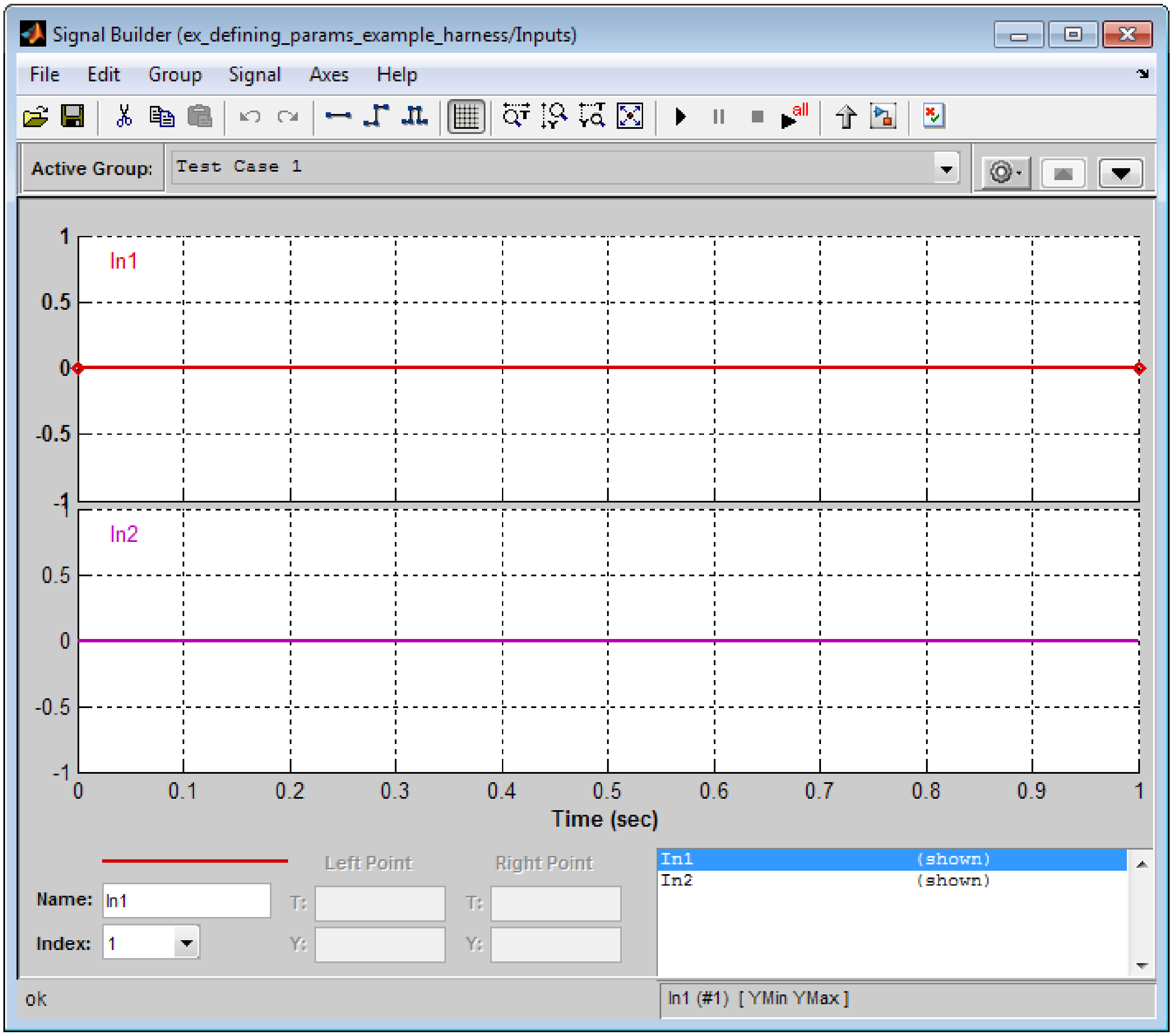
Simulate Test Cases

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`.

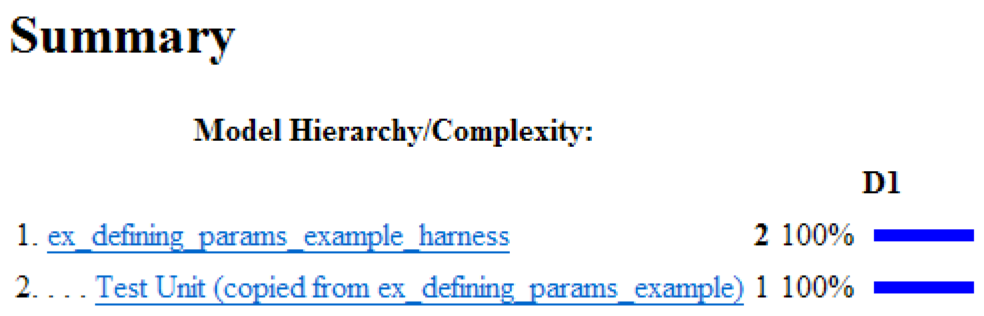
2 The block labeled Inputs in the harness model is a Signal Builder block that contains the test case signals. Double-click the Inputs block to view the test case signals in the Signal Builder block.


Signal Builder (ex_defining_params_example_harness/Inputs)
File Edit Group Signal Axes Help
Active Group: Test Case 1
In1
In2
Time (sec)
Left Point
Right Point
Name: In1
Index: 1
T:
Y:
In1 (shown)
In2 (shown)
ok
In1 (#1) [YMin YMax]

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

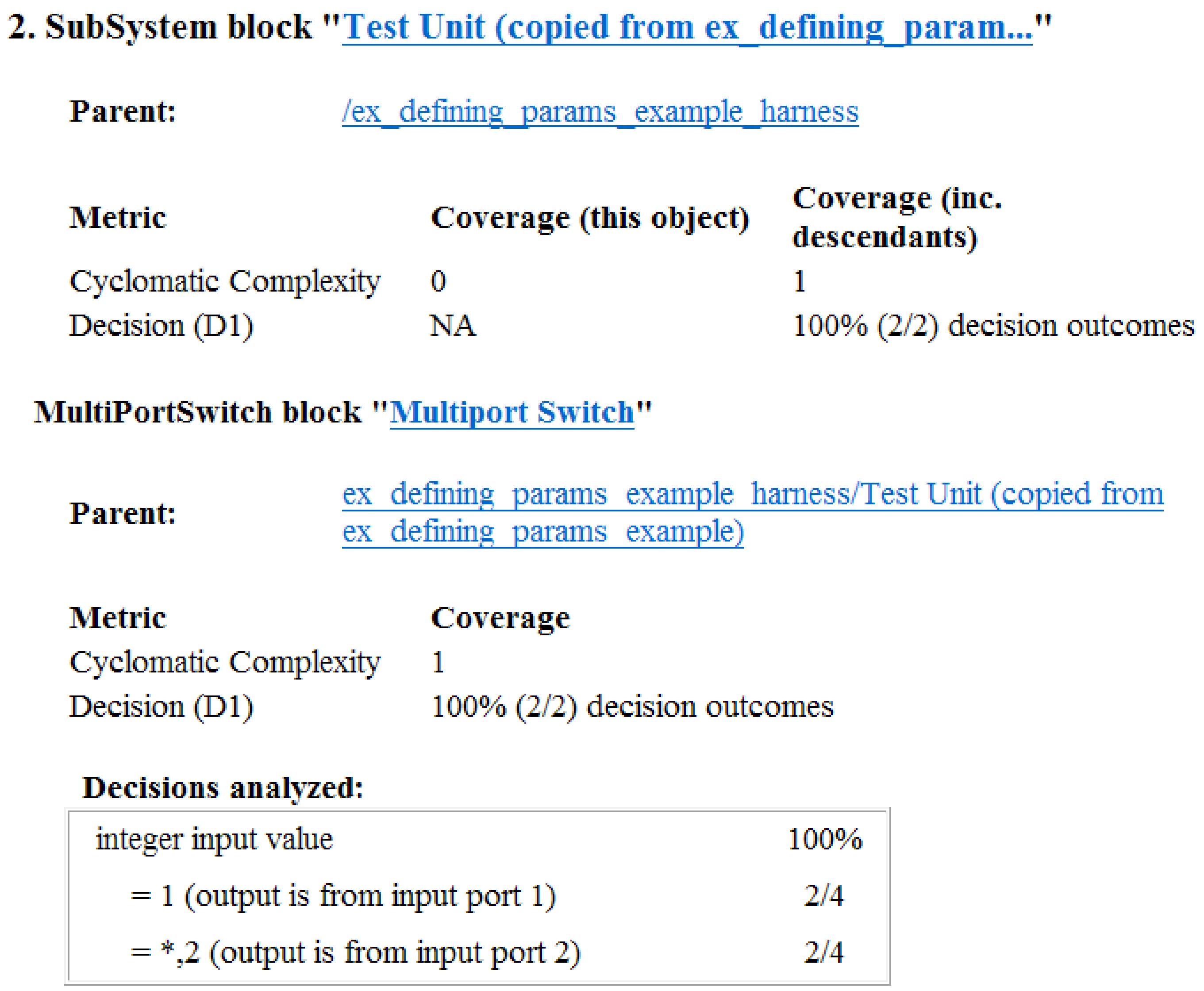
Model Hierarchy/Complexity:

		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.

See Also

"Extend Existing Test Cases After Applying Parameter Configurations" on page 5-35

Store Parameter Constraints in MATLAB Code Files

In this section...
“Export Parameter Constraints to File” on page 5-18
“Import Parameter Constraints from File” on page 5-19

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, on the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**. In the Configuration Parameters dialog box, on the **Design Verifier > Parameters** pane, select **Enable parameter configuration**.

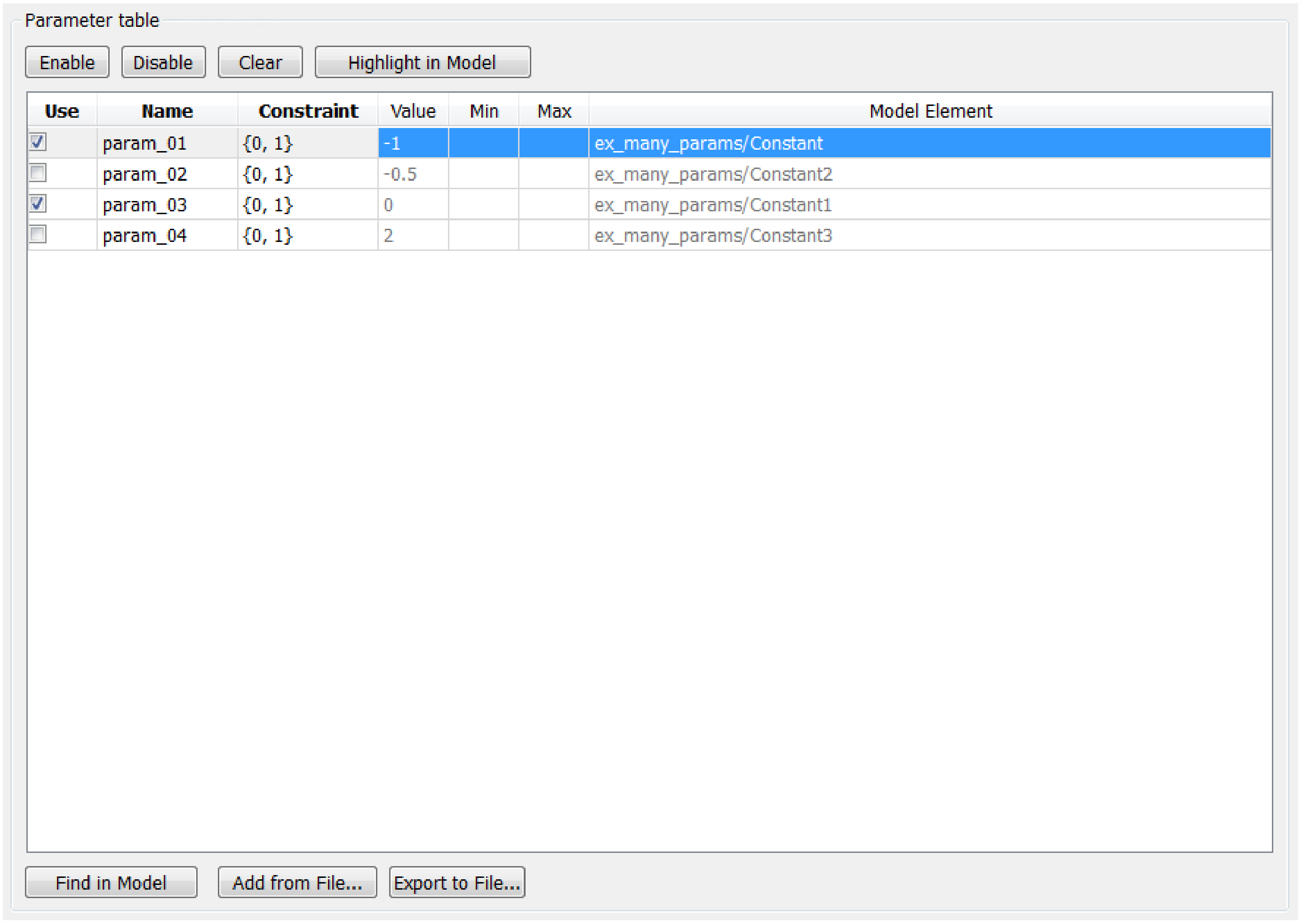
Export Parameter Constraints to File

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 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**. In the Configuration Parameters dialog box, select **Design Verifier > Parameters**.

   The Parameter Table shows specified constraint values for parameters in your model, as in the following example screen shot.



Use	Name	Constraint	Value	Min	Max	Model Element
✓	param_01	{0, 1}	-1			ex_many_params/Constant
☐	param_02	{0, 1}	-0.5			ex_many_params/Constant2
✓	param_03	{0, 1}	0			ex_many_params/Constant1
☐	param_04	{0, 1}	2			ex_many_params/Constant3

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 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**. In the Configuration Parameters dialog box, select **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`

See Also

More About

- "Parameter Identification" on page 5-34

Define Constraint Values for Parameters in MATLAB Code Files

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 5-18.

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`

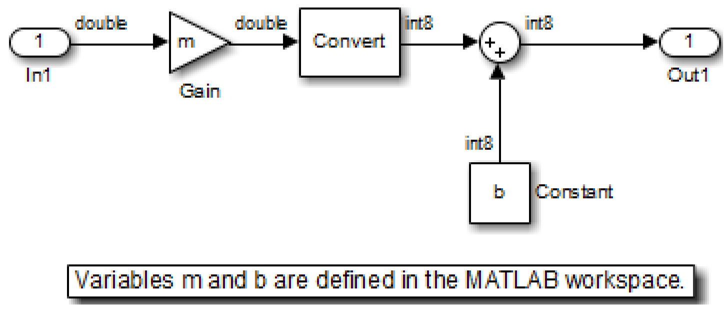
To create a parameter configuration file, make a copy of the template and edit the copy. The comments in the template explain the syntax for defining parameter configurations.

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

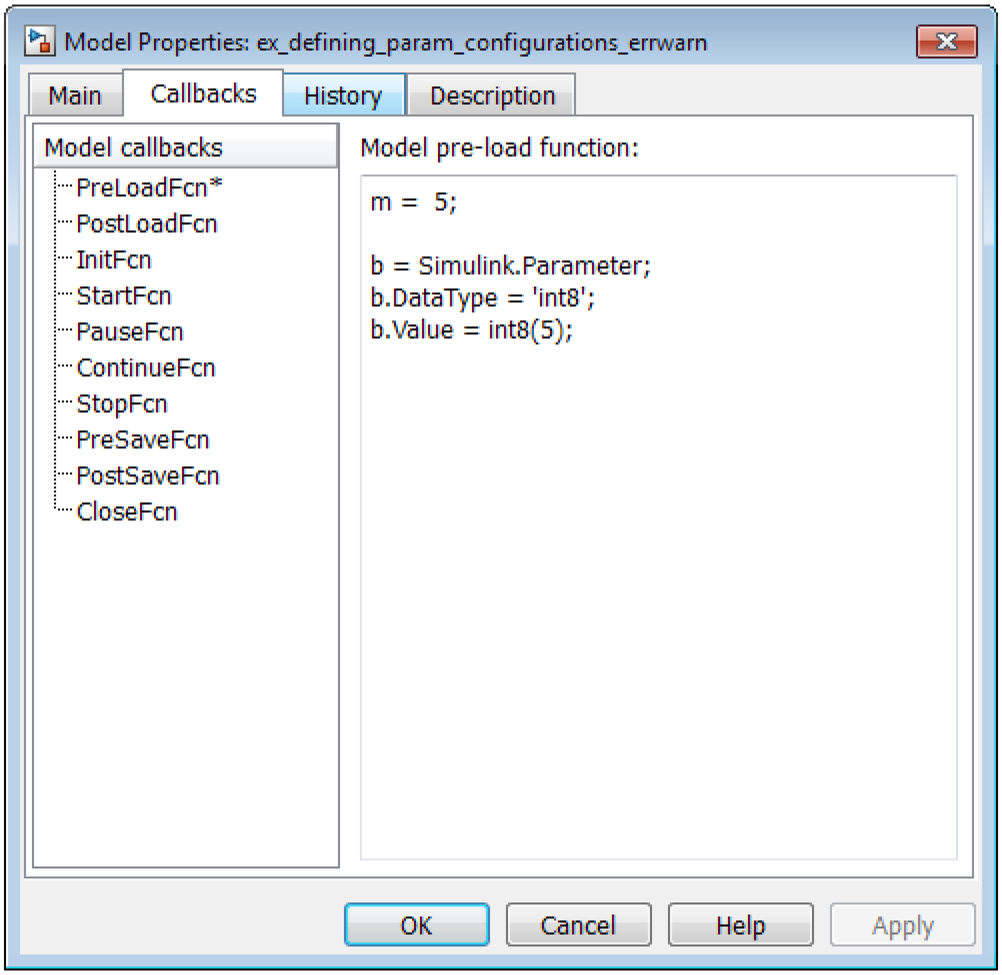
Specify parameter configurations using a structure whose fields share the same names as the parameters that you treat as input variables.

For example, suppose you want to constrain the **Gain** and **Constant value** parameters, `m` and `b`, which appear in the following model:



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.



```
m =  5;

b = Simulink.Parameter;
b.DataType = 'int8';
b.Value = int8(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.

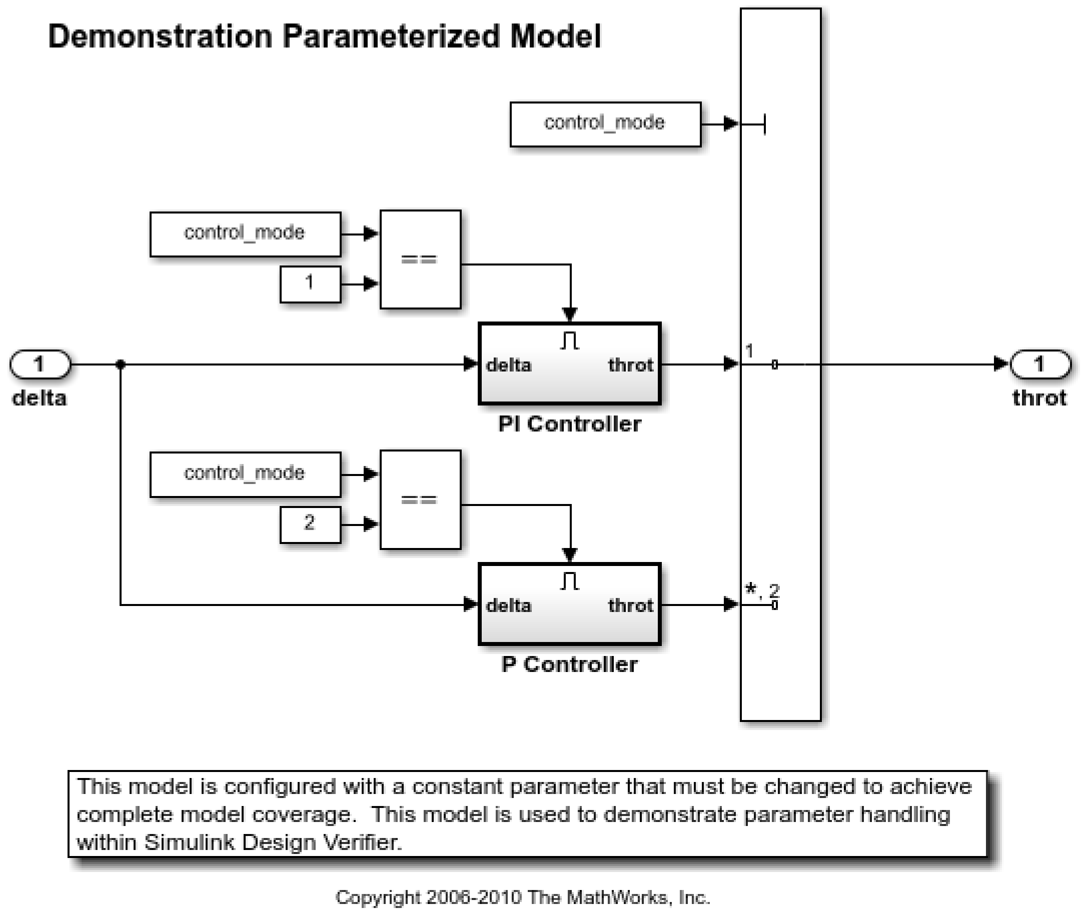
Using Command Line Functions to Support Changing Parameters

This example shows how to use Simulink® Design Verifier™ command-line functions to generate test data that incorporates different parameter values.

Controller Model with an Adjustable Parameter

The example model is a simple controller with a single parameter. The constant parameter 'control_mode' can be either 1 or 2. The parameter must take both values for the test cases to achieve complete coverage. The value determines the switch block output and which enabled subsystem will execute.

```
open_system('sldvdemo_param_controller');
```



This model is configured with a constant parameter that must be changed to achieve complete model coverage. This model is used to demonstrate parameter handling within Simulink Design Verifier.

Specifying Parameter Values for Analysis

Simulink Design Verifier does not identify parameter values. The tool uses the parameter values at the start of analysis for generating tests and proving properties. You can force the tool to incorporate changing parameter values by repeating analysis with different values.

The first iteration of design verifier will use control_mode=1.

```
control_mode = 1;
```

Simulink® Design Verifier™ Options

Simulink Design Verifier functions use options objects created with the `sldvoptions` function to control all aspects of analysis and output.

In this example, we will run Simulink Design Verifier in test generation mode for a maximum of 300 seconds and produce a harness model. We will disable the report generation.

The default values of the remaining options are set correctly to generate tests. You can use the `get` command to display all the options and values.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.MaxProcessTime = 300;
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'off';
opts.HarnessModelFileName = '$ModelName$_harness.slx';

get(opts)
```

```
                             Mode: 'TestGeneration'
                   MaxProcessTime: 300
     DisplayUnsatisfiableObjectives: 'off'
                AutomaticStubbing: 'on'
                      UseParallel: 'off'
          DesignMinMaxConstraints: 'on'
                        OutputDir: 'sldv_output/$ModelName$'
            MakeOutputFilesUnique: 'on'
                 BlockReplacement: 'off'
        BlockReplacementRulesList: '<FactoryDefaultRules>'
    BlockReplacementModelFileName: '$ModelName$_replacement'
                       Parameters: 'off'
         ParametersConfigFileName: 'sldv_params_template.m'
                   ParameterNames: []
             ParameterConstraints: []
           ParameterUseInAnalysis: []
              ParametersUseConfig: 'off'
                    TestgenTarget: 'Model'
          ModelCoverageObjectives: 'ConditionDecision'
                   TestConditions: 'UseLocalSettings'
                   TestObjectives: 'UseLocalSettings'
                 MaxTestCaseSteps: 10000
            TestSuiteOptimization: 'Auto'
                       Assertions: 'UseLocalSettings'
                 ProofAssumptions: 'UseLocalSettings'
              ExtendExistingTests: 'off'
                 ExistingTestFile: ''
         IgnoreExistTestSatisfied: 'on'
               IgnoreCovSatisfied: 'off'
                 CoverageDataFile: ''
                        CovFilter: 'off'
                CovFilterFileName: ''
        IncludeRelationalBoundary: 'off'
                RelativeTolerance: 0.0100
                AbsoluteTolerance: 1.0000e-05
                  DetectDeadLogic: 'off'
               DetectActiveLogic: 'off'
                DetectOutOfBounds: 'on'
             DetectDivisionByZero: 'on'
            DetectIntegerOverflow: 'on'
                     DetectInfNaN: 'off'
                  DetectSubnormal: 'off'
                DesignMinMaxCheck: 'off'
        DetectDSMAccessViolations: 'off'
  DetectBlockInputRangeViolations: 'off'
                  ProvingStrategy: 'Prove'
                MaxViolationSteps: 20
                     SaveDataFile: 'on'
                     DataFileName: '$ModelName$_sldvdata'
               SaveExpectedOutput: 'off'
            RandomizeNoEffectData: 'off'
                 SaveHarnessModel: 'on'
             HarnessModelFileName: '$ModelName$_harness.slx'
            ModelReferenceHarness: 'off'
                    HarnessSource: 'Signal Builder'
                       SaveReport: 'off'
                  ReportPDFFormat: 'off'
                   ReportFileName: '$ModelName$_report'
            ReportIncludeGraphics: 'off'
                    DisplayReport: 'on'
                      SFcnSupport: 'on'
         CodeAnalysisExtraOptions: ''
             ReduceRationalApprox: 'on'
                   SlTestFileName: '$ModelName$_test'
                SlTestHarnessName: '$ModelName$_sldvharness'
              SlTestHarnessSource: 'Inport'
               StrictEnhancedMCDC: 'off'
       RebuildModelRepresentation: 'IfChangeIsDetected'
```

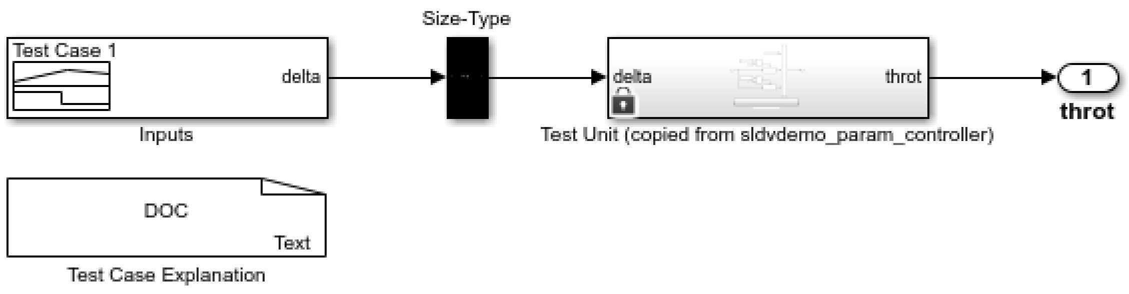
Generating Tests and Collecting Coverage

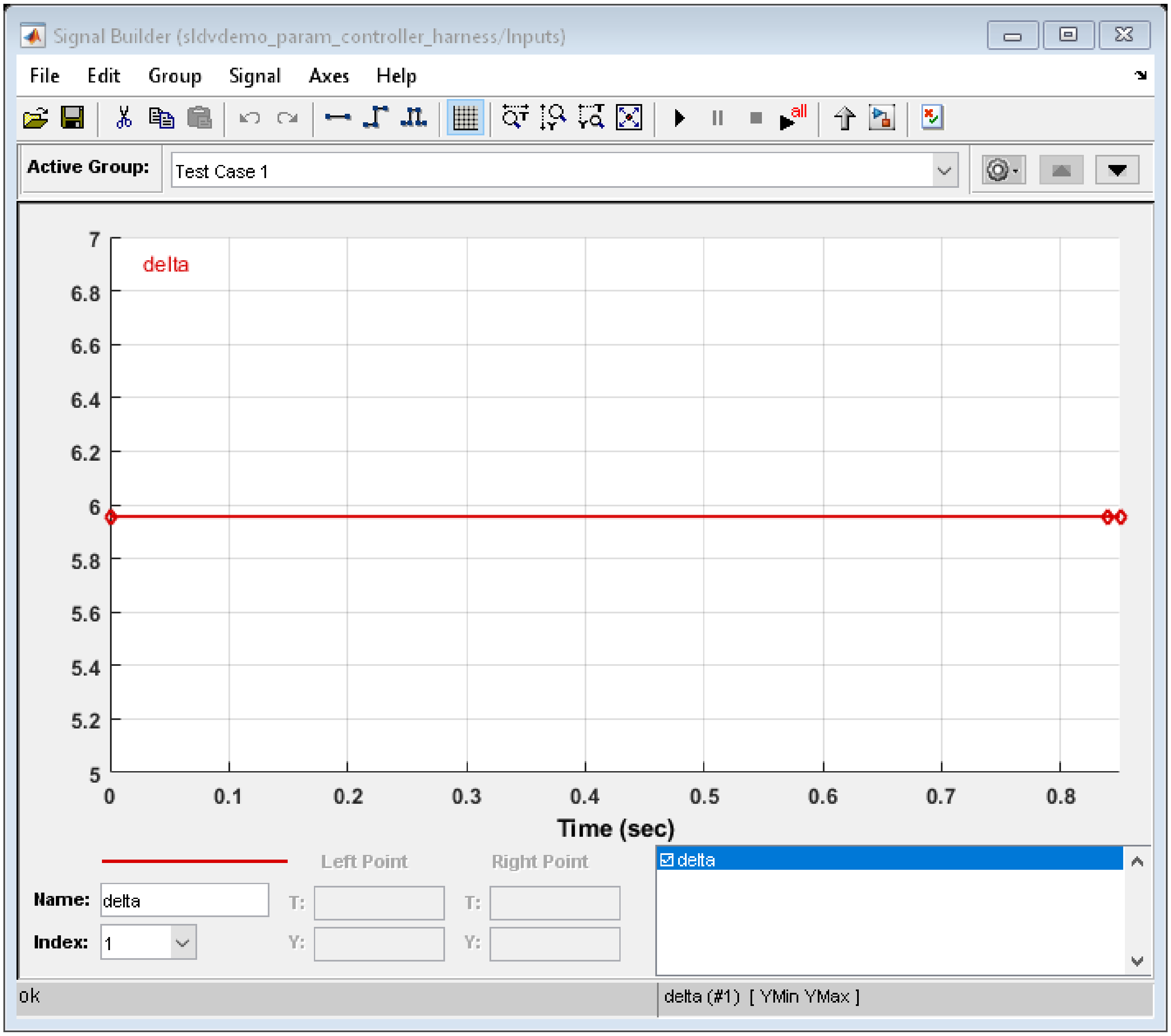
The `sldvgencov` function generates test suites and model coverage together. All tests that can be generated with the current parameter values will be collected into the harness model and the resulting coverage returned in a coverage data object.

```
[status,coverageData,files] = sldvgencov('sldvdemo_param_controller',opts);
```

```
Checking compatibility for test generation: model 'sldvdemo_param_controller'
Compiling model...done
Building model representation...done

'sldvdemo_param_controller' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 29-Feb-2020 11:35:04...
..............

Completed normally.

Generating output files:

    Harness model:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex05697027\sldv_output\sldvdemo_param_co
```

```
Results generation completed.

    Data file:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex05697027\sldv_output\sldvdemo_param_co
```





Integrating Parameter Initialization Into a Test Harness

Generated test cases must be run with the same parameter values used during analysis. An initialization command configures the values during simulation of test cases. The `sldvharnessmerge` function incorporates initialization commands into test harnesses.

```
initCmdStr = 'control_mode=1;'
[path,modelName] = fileparts(files.HarnessModel);
sldvmergeharness(modelName,modelName,initCmdStr);
```

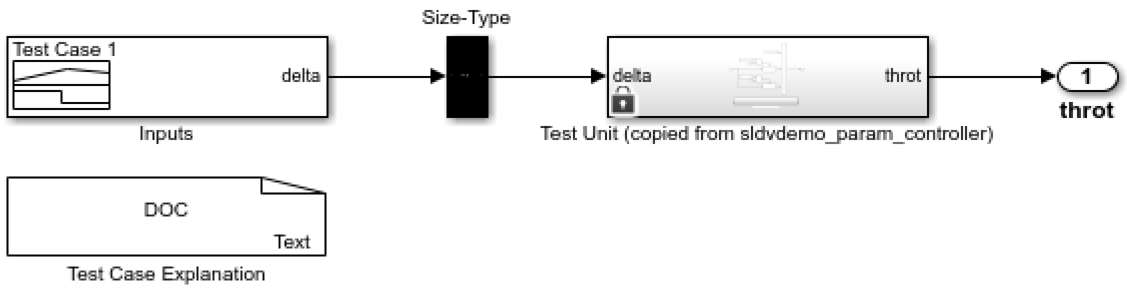
```
initCmdStr =

    'control_mode=1;'
```

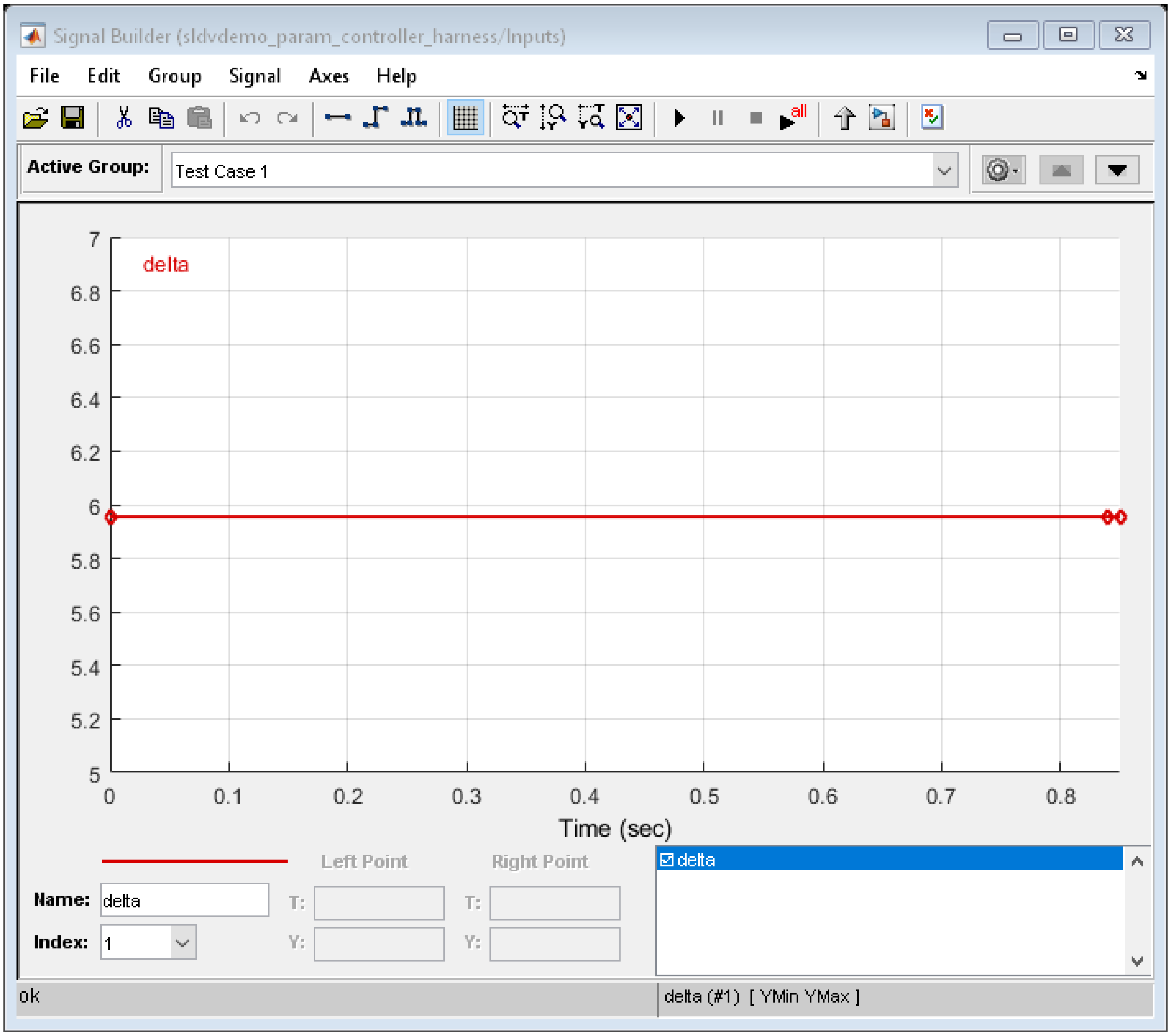
Modifying Parameters and Repeating Test Generation

Modifying parameter values enables additional test generation. Passing a coverage data object as the third input to `sldvgencov` forces the function to ignore all model coverage test objectives that have been satisfied. We use the coverage data that was returned from the earlier call to `sldvgencov` to restrict test generation to unsatisfied test objectives.

```
control_mode=2;
[status,newCov,newFiles] = sldvgencov('sldvdemo_param_controller',opts,false,coverageData);
```

```
Validating cached model representation from 29-Feb-2020 11:35:04...change detected

Checking compatibility for test generation: model 'sldvdemo_param_controller'
Compiling model...done
Building model representation...done

'sldvdemo_param_controller' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 29-Feb-2020 11:36:52...
............

Completed normally.

Generating output files:

    Harness model:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex05697027\sldv_output\sldvdemo_param_co

Results generation completed.

    Data file:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex05697027\sldv_output\sldvdemo_param_co
```

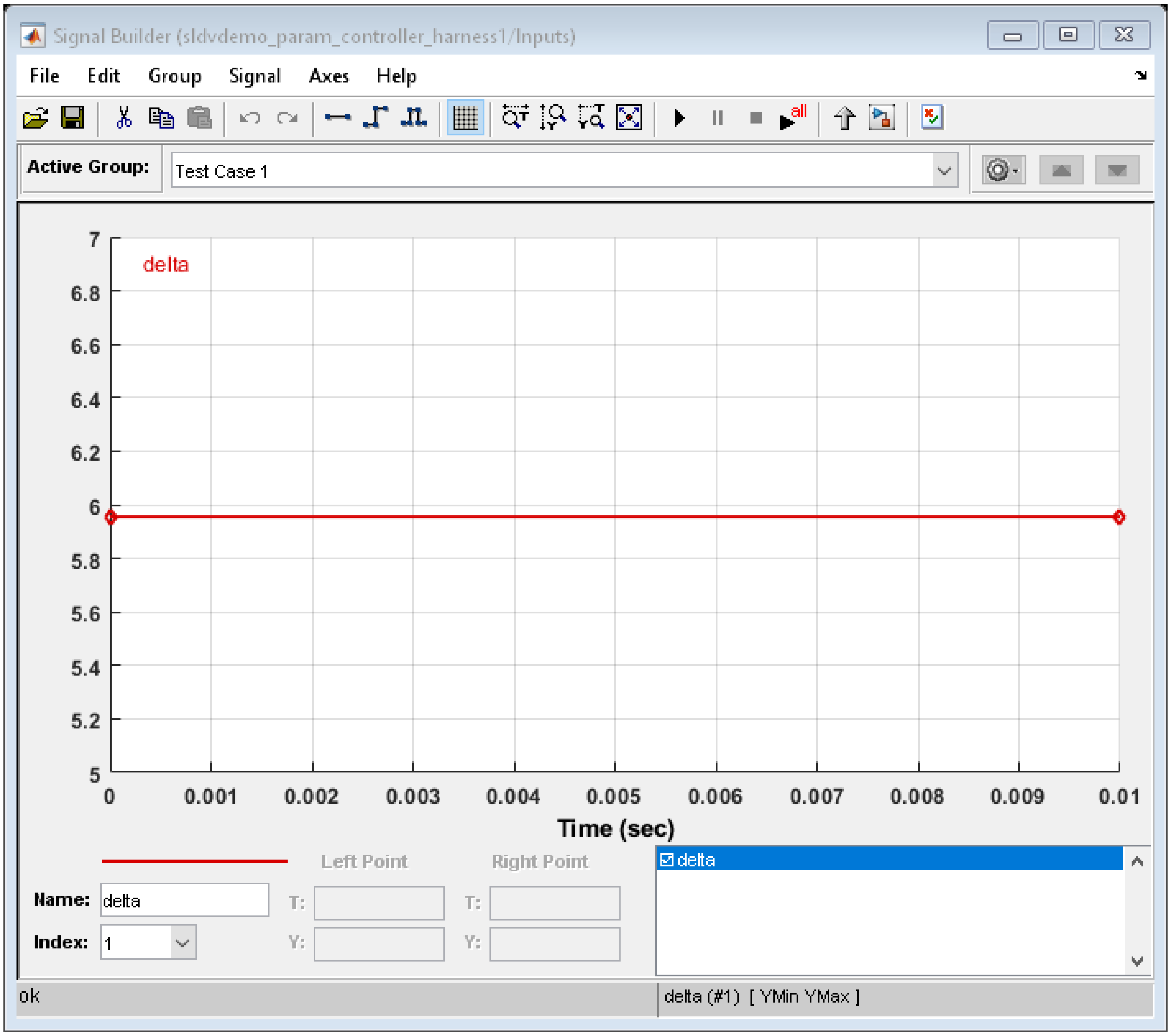




Name: delta
Index: 1

ok

delta (#1) [YMin YMax]

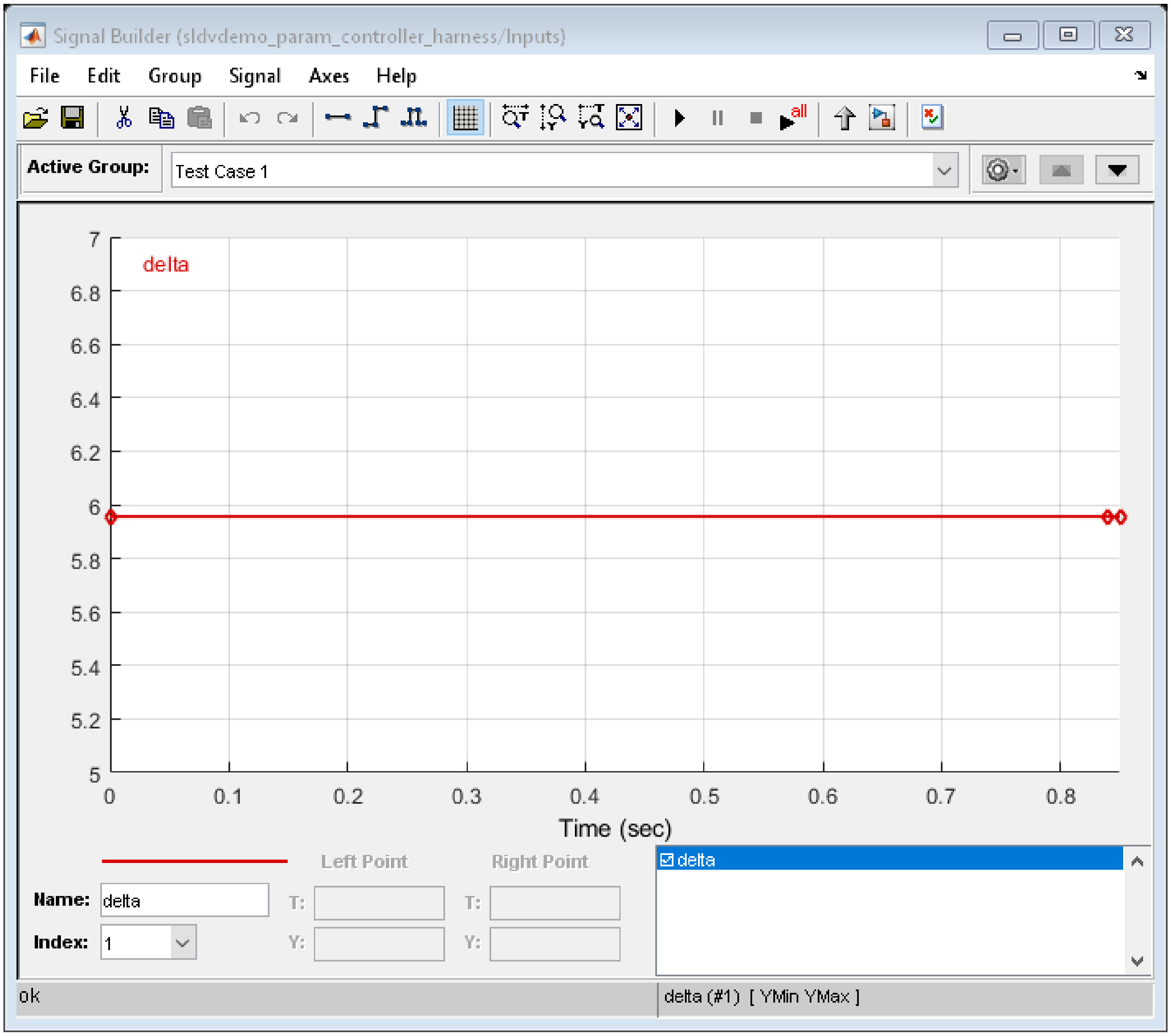

Signal Builder (sldvdemo_param_controller_harness1/Inputs)
File Edit Group Signal Axes Help
Active Group: Test Case 1
delta
Time (sec)
Left Point
Right Point
Name: delta
Index: 1
T:
Y:
☑ delta
ok
delta (#1) [YMin YMax]

Merging Test Harnesses Into a Single Model

Another call to `sldvharnessmerge` merges the test data from the new harness and its initialization command into the existing harness model.

```
newInitCmd = 'control_mode=2;'
[path,newModelName] = fileparts(newFiles.HarnessModel);
sldvmergeharness(modelName,newModelName,newInitCmd);
```

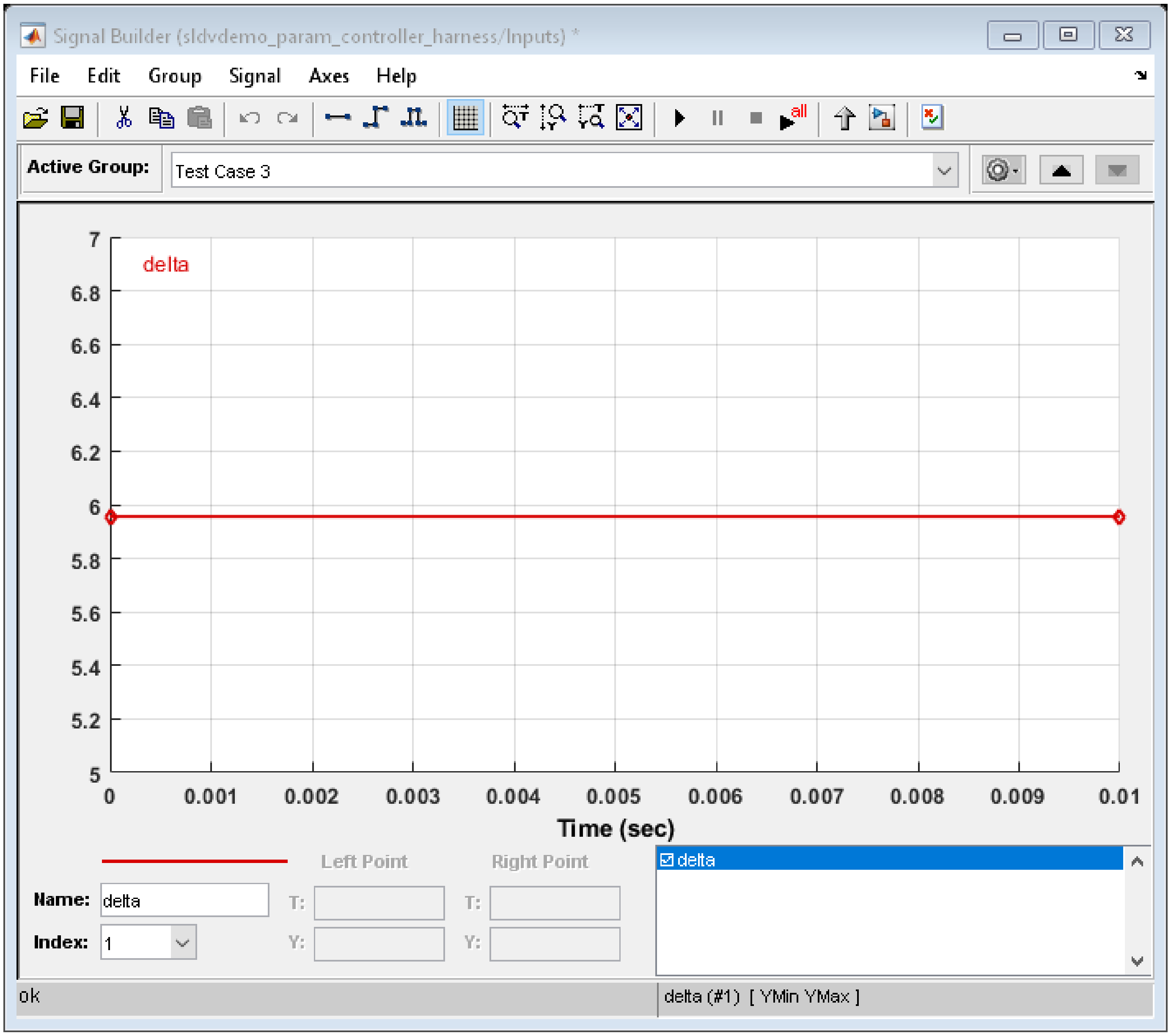
```
newInitCmd =

    'control_mode=2;'
```



Executing the Tests in the Harness Model

We close the second harness model that was created because the test cases have been merged into the first harness model. You can execute the suite of tests by clicking the "Run all" button on the Signal Builder.

```
close_system(newModelName,0);
sldvdemo_playall(modelName);
```



Name: delta
Index: 1
delta (#1) [YMin YMax]

Clean Up

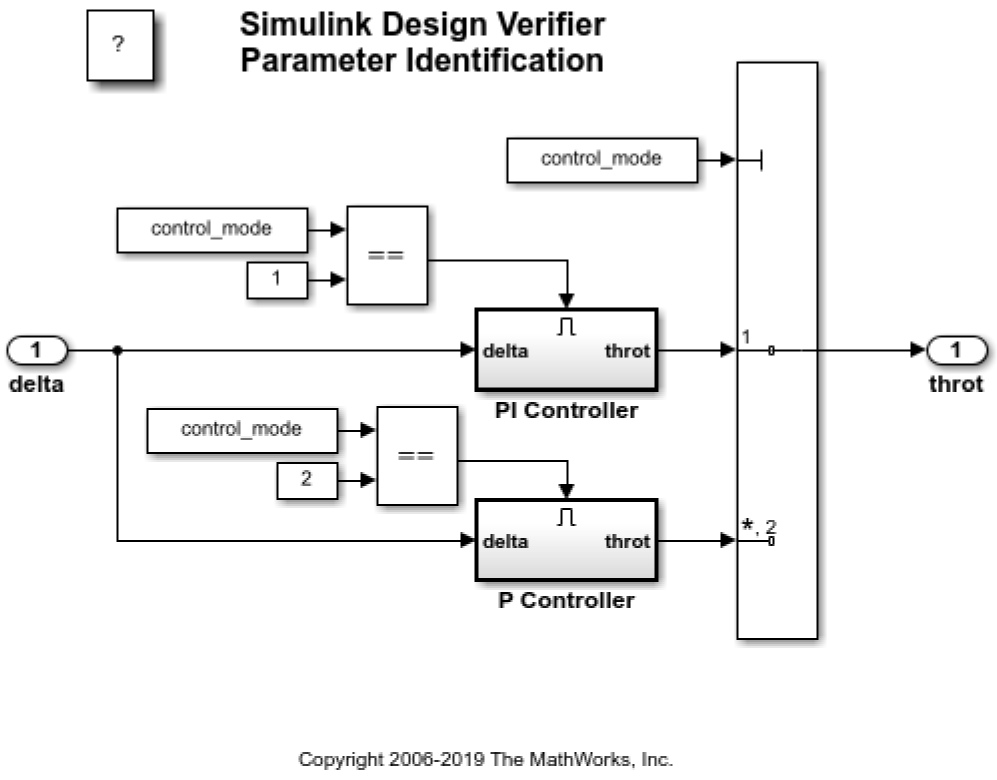
To complete the example, close the models and remove the generated files.

```
close_system(modelName,0);
close_system('sldvdemo_param_controller',0);
delete(files.HarnessModel);
delete(newFiles.HarnessModel);
```

Parameter Identification

This example shows how to tune parameters using parameter configuration file for Simulink Design Verifier analysis. The model contains the parameter `control_mode` that enables the active controller and selects its output to be the model output. Simulink Design Verifier treats this parameter as an input that is constrained to be either 1 or 2 and generates the appropriate value for each test case.

```
open_system('sldvdemo_param_identification');
```



Extend Existing Test Cases After Applying Parameter Configurations

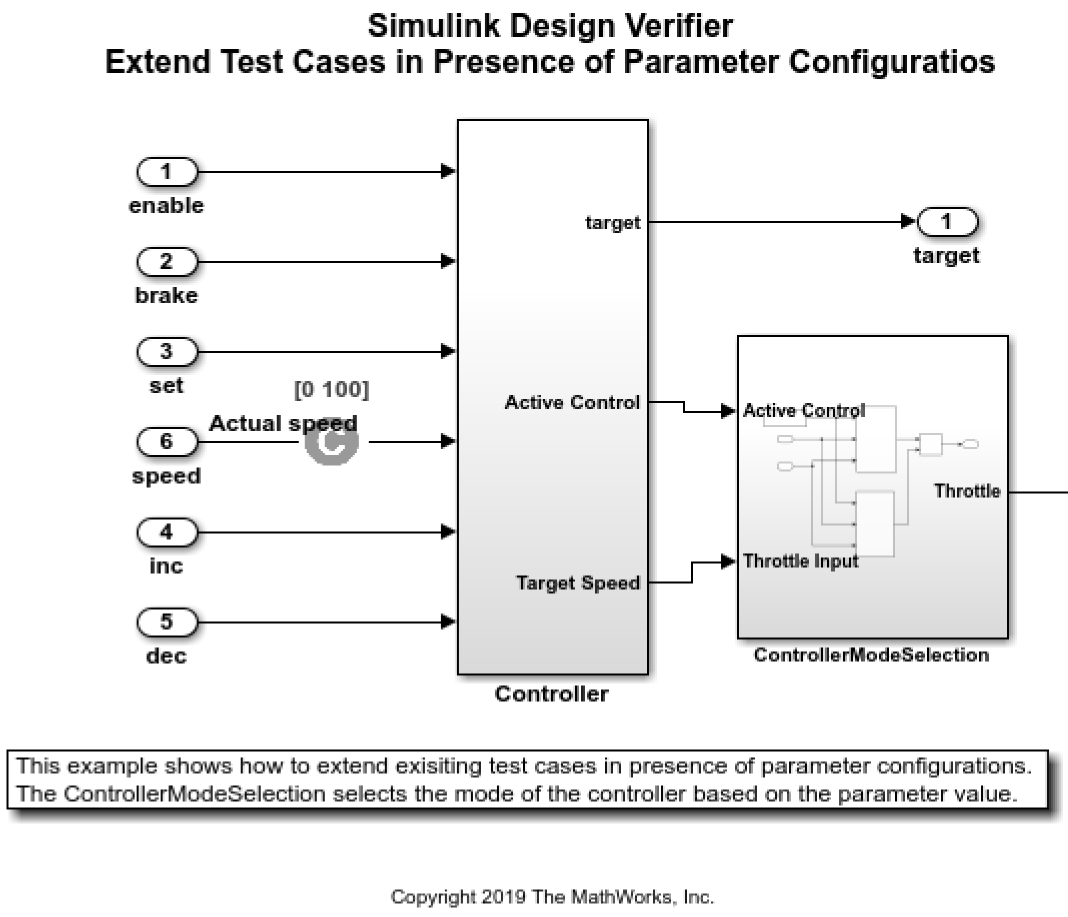
This example shows how to achieve missing coverage by extending existing test cases after applying parameter configurations.

In this example, you generate test cases for a model and review the analysis results. The results show that the model consists of unsatisfiable objectives and does not achieve full coverage. Then, you apply parameter configurations in the model and reuse the previously generated test cases to achieve full model coverage.

Step 1: Generate Initial Test Cases and Review Results

The `sldvexParameterController` model is a cruise control model that controls the throttle speed by selecting a P Controller or PI Controller. The `ControllerModeSelection` subsystem uses the `SelectMode` parameter to select the controller mode. Define the enumerated data type for `Selectmode` by using the function `Simulink.defineIntEnumType`. For more information on enumerated values, see “Use Enumerated Data in Simulink Models” (Simulink).

```
Simulink.defineIntEnumType('EnumForControllerSelection',...
{'Pmode','PImode'},[1;2]);
SelectMode = Simulink.Parameter;
SelectMode.Value = EnumForControllerSelection.Pmode;
model = 'sldvexParameterController';
open_system(model);
```

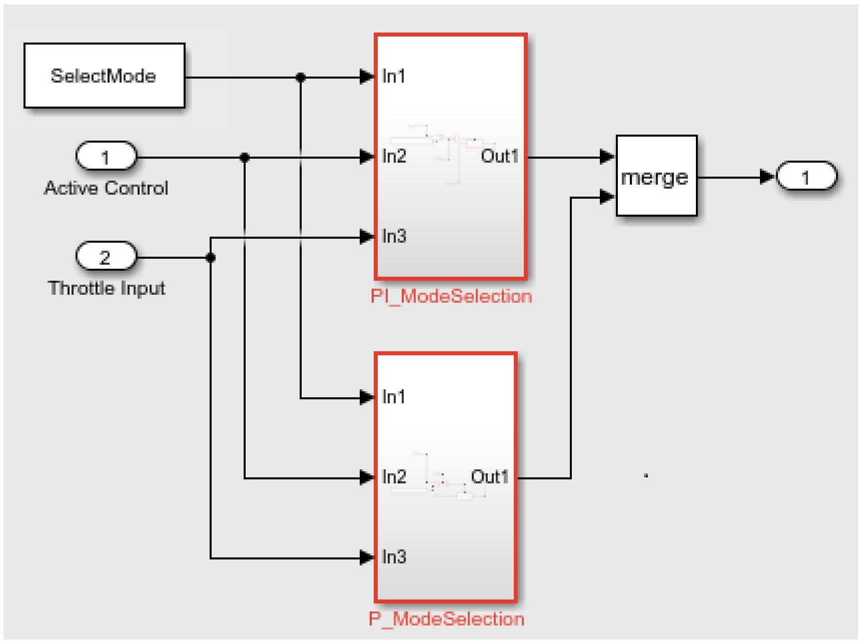

Simulink Design Verifier
Extend Test Cases in Presence of Parameter Configuratios
1
enable
2
brake
3
set
[0 100]
Actual speed
6
speed
4
inc
5
dec
target
Active Control
Target Speed
Controller
1
target
Active Control
Throttle
Throttle Input
ControllerModeSelection
2
throttle
This example shows how to extend exisiting test cases in presence of parameter configurations.
The ControllerModeSelection selects the mode of the controller based on the parameter value.
Copyright 2019 The MathWorks, Inc.

Set the `sldvoptions` and analyze the model by using the specified options.

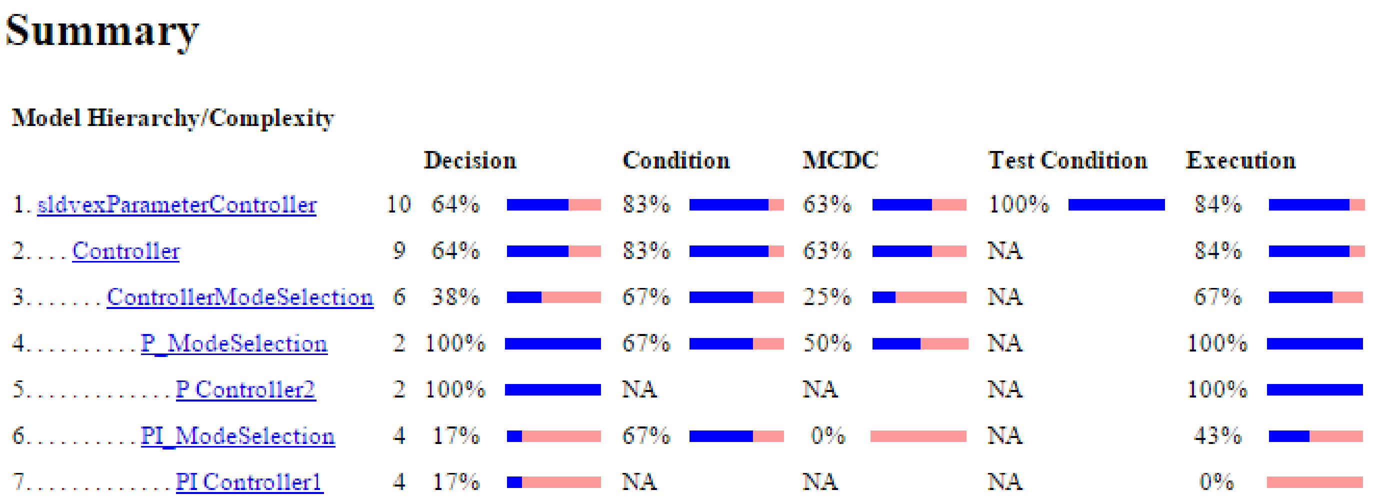
```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'MCDC';
[ status, files ] = sldvrun(model, opts, true);
```

After the analysis completes, the Results Summary window displays that 15 out of 54 objectives are unsatisfiable.

In the Results Summary window, click **Highlight analysis results on model**. Double-click the `ControllerModeSelection` subsystem. The `PI_ModeSelection` and `P_ModeSelection` subsystems are highlighted in red and consist of unsatisfiable objectives.



To view the model coverage report, in the Results Summary window, click **Simulate tests and produce a model coverage report**. The report shows that the model does not achieve full coverage.



Summary

Model Hierarchy/Complexity

		Decision	Condition	MCDC	Test Condition	Execution
1. sldvexParameterController	10	64%	83%	63%	100%	84%
2. . . . Controller	9	64%	83%	63%	NA	84%
3. . . . . . . ControllerModeSelection	6	38%	67%	25%	NA	67%
4. . . . . P_ModeSelection	2	100%	67%	50%	NA	100%
5. . . . . P Controller2	2	100%	NA	NA	NA	100%
6. . . . . PI_ModeSelection	4	17%	67%	0%	NA	43%
7. . . . . PI Controller1	4	17%	NA	NA	NA	0%

Full coverage is not achieved because the parameter value `SelectMode` is restricted to the default value of `EnumForControllerSelection.Pmode`. Consequently, full coverage is not achieved for the `PI_ModeSelection` subsystem.

Step 2: Configure Parameter Configurations and Extend Existing Test Cases

If you apply parameter configurations, Simulink Design Verifier treats the parameter as a variable during analysis and constraints the values based on the constraint values that you specify.

Apply parameter configurations for the `SelectMode` parameter by specifying the constraint values for `parameterValue`.

```
controlParameter = [ {'SelectMode'}];
parameterValue = [ {'[EnumForControllerSelection.Pmode EnumForControllerSelection.PImode]'}];
opts.Parameters = 'on';
opts.ParametersUseConfig = 'on';
opts.ParameterNames = controlParameter;
opts.ParameterConstraints = parameterValue;
opts.ParameterUseInAnalysis = {'on'};
```

To reuse the previously generated test cases, configure the analysis option to extend the existing test cases and specify the existing test file.

```
opts.ExtendExistingTests = 'on';
opts.IgnoreExistTestSatisfied = 'off';
opts.ExistingTestFile = files.DataFile;
```

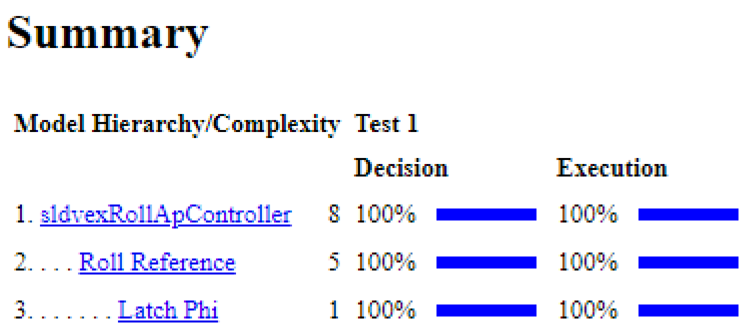
Step 3: Perform Analysis and Review Coverage Report

Analyze the model by using the specified options.

```
[status, fileNames] = sldvrun(model, opts, true);
```

After the analysis completes, the Results Summary window displays that all the objectives are satisfied.

To generate model coverage report, click **Simulate tests and produce a model coverage report**. The report shows that the model achieves full coverage.



Summary

Model Hierarchy/Complexity		Test 1			
		Decision		Execution	
1. sldvexRollApController	8	100%	█	100%	█
2. . . . Roll Reference	5	100%	█	100%	█
3. . . . . . . Latch Phi	1	100%	█	100%	█

To complete this example, close the model.

```
close_system('sldvexParameterController', 0);
```

See also

- “Parameter Constraint Values” on page 5-2
- “When to Extend Existing Test Cases” on page 8-2

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
- Data store access violations
- Specified block input range violations

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.

See Also

"Run a Design Error Detection Analysis" on page 6-4 | "Design Verifier Pane: Design Error Detection" on page 15-43

Derived Ranges in Design Error Detection

When you specify minimum and maximum values for a signal or data in a model (Simulink), 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:

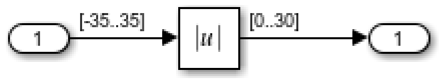
- Block Outports
- Stateflow local data

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: [–35..35]
- Abs block output: [0..30]

Given the design range on the Inport block, the only possible values for the Abs block output are values from 0 to 35. Therefore, the derived range for the Abs block is [0..35].

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

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.

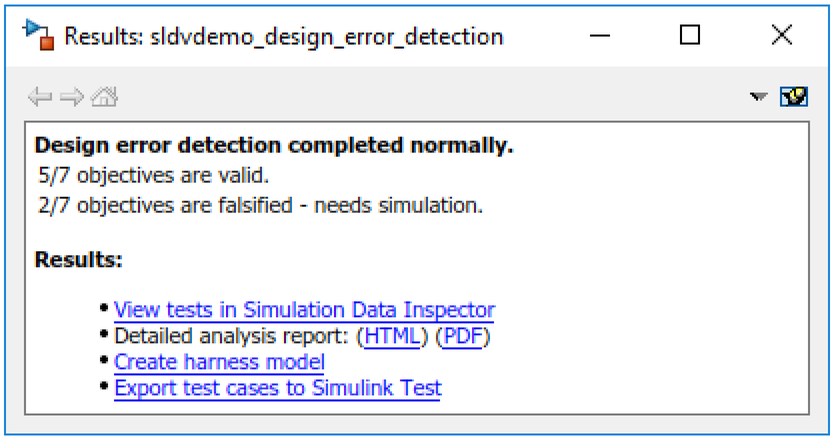
Note If you select design error detection for dead logic, you cannot select any other type of design error detection. For dead logic detection, Simulink Design Verifier performs an independent analysis. If you want to detect design errors for dead logic and any of the other types of design errors, you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


Results: sldvdemo_design_error_detection

Design error detection completed normally.
5/7 objectives are valid.
2/7 objectives are falsified - needs simulation.

Results:

- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test

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 Results Summary Window

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 Results Summary Window.

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

To view the latest results, on the **Design Verifier** tab, in the **Review Results** section, click **Results Summary**.

For any Simulink Design Verifier analysis, from the Results Summary Window, 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.

See Also

More About

- "Check Your Model Using the Model Advisor" (Simulink)

Dead Logic Detection

In this section...

Before you simulate a model, use dead logic detection to analyze the model for dead logic. In Simulink Design Verifier, design error detection for dead logic consists of two analysis options:

- Detection of dead logic only: If you select this option, Simulink Design Verifier analyzes your model without making any approximations, such as rational approximation for floating points, or while loop approximation. For more information, see "Approximations" on page 2-19. With this option, Simulink Design Verifier does not report active logic or undecided objectives and it may not identify some dead logic in your model.

  This option is available in:

  - The Model Advisor. See "Check For Design Errors using the Model Advisor" on page 6-6.
  - The Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane.
- Detection of active logic: Active logic detection runs concurrently with dead logic detection. With this option, Simulink Design Verifier reports active logic in addition to dead logic as well as undecided objectives. This option may in some cases identify or find additional dead logic. The analysis may use approximations and are reported accordingly.

  This option is available in the Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane.

Detect Dead Logic Only

If you are not using the Model Advisor, to detect dead logic:

1. On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
2. Click **Error Detection Settings**.
3. In the Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane:

   a. Enable the "Dead logic" on page 15-43 option.
   b. Clear the "Identify active logic" on page 15-44 option, if it is selected.
4. To apply these settings, click **OK** and close the Configuration Parameters dialog box.
5. Click **Detect Design Errors**.

Detect Dead and Active Logic

1. On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
2. Click **Error Detection Settings**.
3. In the Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane, enable the "Dead logic" on page 15-43 and "Identify active logic" on page 15-44 options.

4 To apply these settings, click **OK** and close the Configuration Parameters dialog box.
5 Click **Detect Design Errors**.

Run a Dead Logic Analysis and Review Results

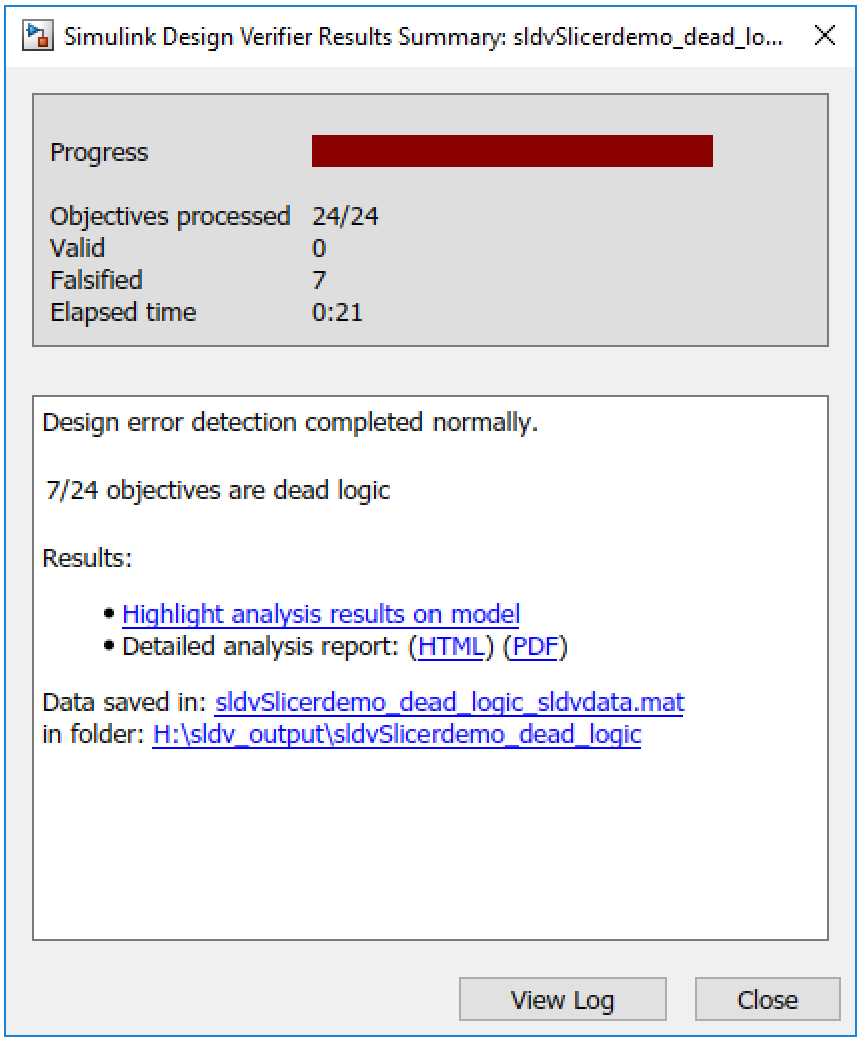
This example shows how to detect dead logic in the `sldvSlicerdemo_dead_logic` example model. Dead logic detection finds the unreachable objectives in the model that cause the model element to remain inactive.

1 Open the `sldvSlicerdemo_dead_logic` model.

```
open_system('sldvSlicerdemo_dead_logic');
```

2 On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
3 Click **Error Detection Settings**.
4 In the Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane, enable “Dead logic” on page 15-43 option and clear “Identify active logic” on page 15-44 option.
5 Click **Detect Design Errors**.

The software analyzes the model for dead logic and displays the results in the Results Summary window. The result indicates that seven of the `24` objectives are dead logic.


Simulink Design Verifier Results Summary: sldvSlicerdemo_dead_lo...

Progress	
Objectives processed	24/24
Valid	0
Falsified	7
Elapsed time	0:21

Design error detection completed normally.

7/24 objectives are dead logic

Results:

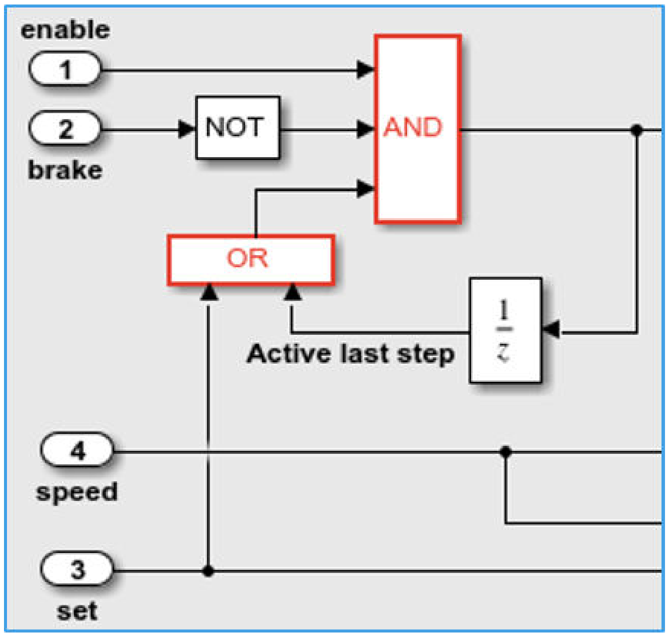
- Highlight analysis results on model
- Detailed analysis report: (HTML) (PDF)

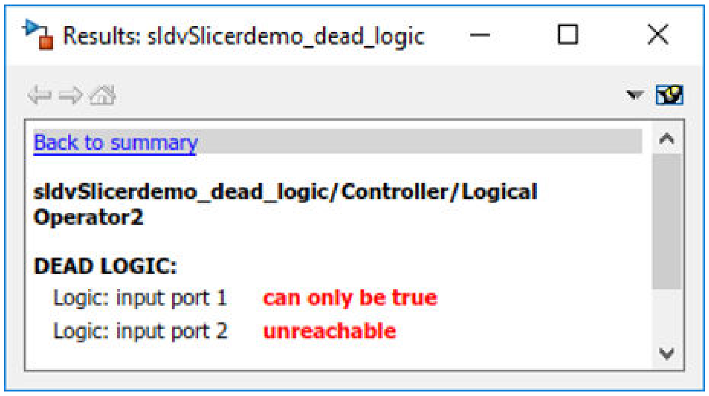
Data saved in: sldvSlicerdemo_dead_logic_sldvdata.mat
in folder: H:\sldv_output\sldvSlicerdemo_dead_logic

6 Click **Highlight analysis results on model**. The dead logic model elements are highlighted in red.

7 Open the `Controller` subsystem, and click the OR block highlighted in red. The Result Inspector displays the summary of the dead logic.

The `set` input is equal to `1`, so the `input port 1` of the OR block **can only be true**. The status implies that the `input port 1` false condition is dead logic. Similarly, the `input port 2` is `unreachable`, as the objective never executes and is dead logic.




Results: sldvSlicerdemo_dead_logic

Back to summary

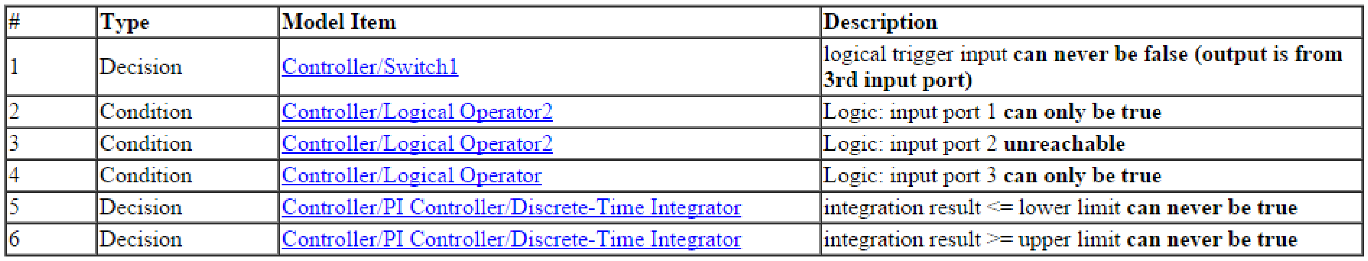
sldvSlicerdemo_dead_logic/Controller/Logical Operator2

DEAD LOGIC:

Logic: input port 1	**can only be true**
Logic: input port 2	**unreachable**

8 To view the detailed analysis report, in the Results Summary window, click **HTML**.

The report displays the summary of all the results that are dead logic in the model.



#	Type	Model Item	Description
1	Decision	Controller/Switch1	logical trigger input **can never be false (output is from 3rd input port)**
2	Condition	Controller/Logical Operator2	Logic: input port 1 **can only be true**
3	Condition	Controller/Logical Operator2	Logic: input port 2 **unreachable**
4	Condition	Controller/Logical Operator	Logic: input port 3 **can only be true**
5	Decision	Controller/PI Controller/Discrete-Time Integrator	integration result <= lower limit **can never be true**
6	Decision	Controller/PI Controller/Discrete-Time Integrator	integration result >= upper limit **can never be true**

Dead Logic

The software stores the detailed analysis results in the `DeadLogic field` in the “Simulink Design Verifier Data Files” on page 13-7. You can use the data file for further analysis of the results.

See Also

More About

- “Design Verifier Pane: Design Error Detection” on page 15-43
- “Model Objects That Receive Dead Logic Detection” on page 6-15

Detect Dead Logic Caused by an Incorrect Value

In this section...

Dead logic detection helps you to identify:

- Model design errors.
- Extraneous model elements.
- Model elements that should be executed, but are not.

In this example, you analyze a fuel rate controller model to determine if the model contains dead logic. Dead logic detection finds the incorrect variable value that causes a transition condition in a Stateflow chart to remain inactive.

Analyze the Fuel System Model

1 Open the model.

```
sldvdemo_fuelsys_logic_simple
```

Ensure that the current folder is writable.

2 Configure dead logic detection.

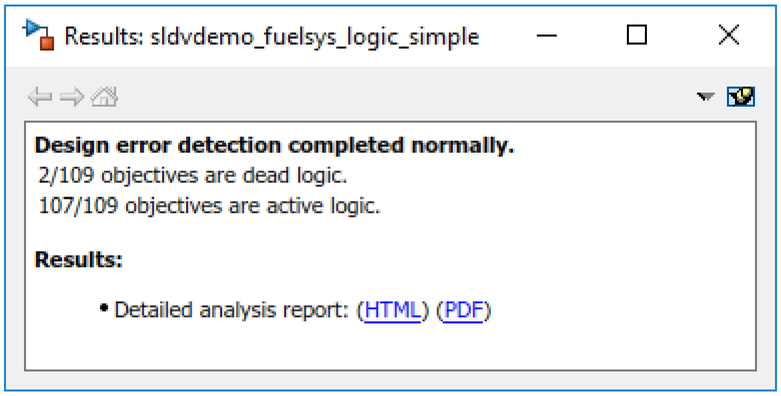
On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.

3 Select **Error Detection Settings**.

4 In the Configuration Parameter dialog box, select **Dead logic**. Clear **Identify active logic**. Click **OK**.

5 Click **Detect Design Errors**.

6 The results dialog box shows that there are 2/109 objectives that are dead logic.



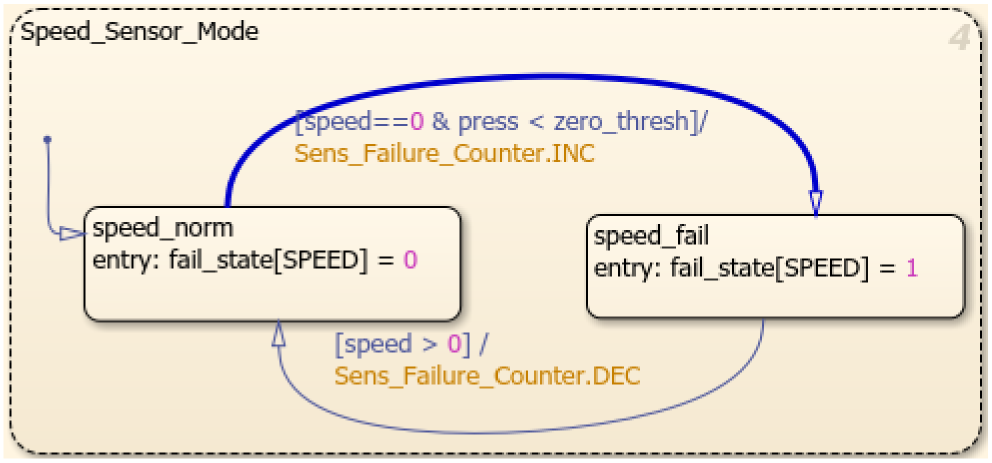
Design error detection completed normally.
2/109 objectives are dead logic.
107/109 objectives are active logic.

Results:

- Detailed analysis report: (HTML) (PDF)

Review the Results and Trace to the Model

1. Create an analysis report. From the results inspector window, click **HTML**.
2. Scroll to the **Dead Logic** section under **Design Error Detection Objectives Status**. The table lists two instances of dead logic.
3. In the **Description** column, one of the dead logic instances is the `false` condition of `press < zero_thresh`. The dead logic result indicates that in the simulation, the `false` condition was not executed. This logic is part of the `Sens_Failure_Counter.INC` transition.
4. Click the **Model Item** link. Simulink highlights the transition in the chart.



Investigate the Cause of the Dead Logic

1. The logical statement controlling the transition is

   ```
   speed==0 & press < zero_thresh
   ```
2. Return to the report. Scroll to the **Constraints** section.
3. The value of the input `control logic/Input Data "press"` is constrained from 0 through 2. Click the link to open the input in the Model Explorer.
4. Select the **Model Workspace** in the Model Explorer. In the contents table, select `zero_thresh`. The value of `zero_thresh` is 250.

   Given the constrained value of `press`, it is always less than `zero_thresh` and therefore, the `false` condition is never exercised.

Update the Input Constraint and Reanalyze the Model

1. Change the value of `zero_thresh` to 0.250.
2. Reanalyze the model. On the **Design Verifier** tab, click **Detect Design Errors**.
3. In the new results, the objective is no longer dead logic.

See Also

Related Examples

- “Dead Logic Detection” on page 6-8

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 6-15	●	
"Dead Zone" on page 6-16	●	
"Discrete-Time Integrator" on page 6-16	●	
"Enabled Subsystem" on page 6-17	●	●
"Enabled and Triggered Subsystem" on page 6-17	●	●
"Fcn" on page 6-17		●
"For Iterator, For Iterator Subsystem" on page 6-17	●	
"If, If Action Subsystem" on page 6-18	●	●
"Library-Linked Objects" on page 6-18	●	●
"Logical Operator" on page 6-18		●
"MATLAB Function" on page 6-18	●	●
"MinMax" on page 6-18	●	
"Model" on page 6-19	●	●
"Multiport Switch" on page 6-19	●	
"Rate Limiter" on page 6-19	●	
"Relay" on page 6-19	●	
"Saturation" on page 6-20	●	
"Stateflow Charts" on page 6-20	●	●
"Switch" on page 6-20	●	
"SwitchCase, SwitchCase Action Subsystem" on page 6-20	●	
"Triggered Models" on page 6-20	●	●
"Triggered Subsystem" on page 6-21	●	●
"While Iterator, While Iterator Subsystem" on page 6-21	●	

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.

Common Causes for Dead Logic

These are a few common modeling patterns that often lead to dead logic in a model:

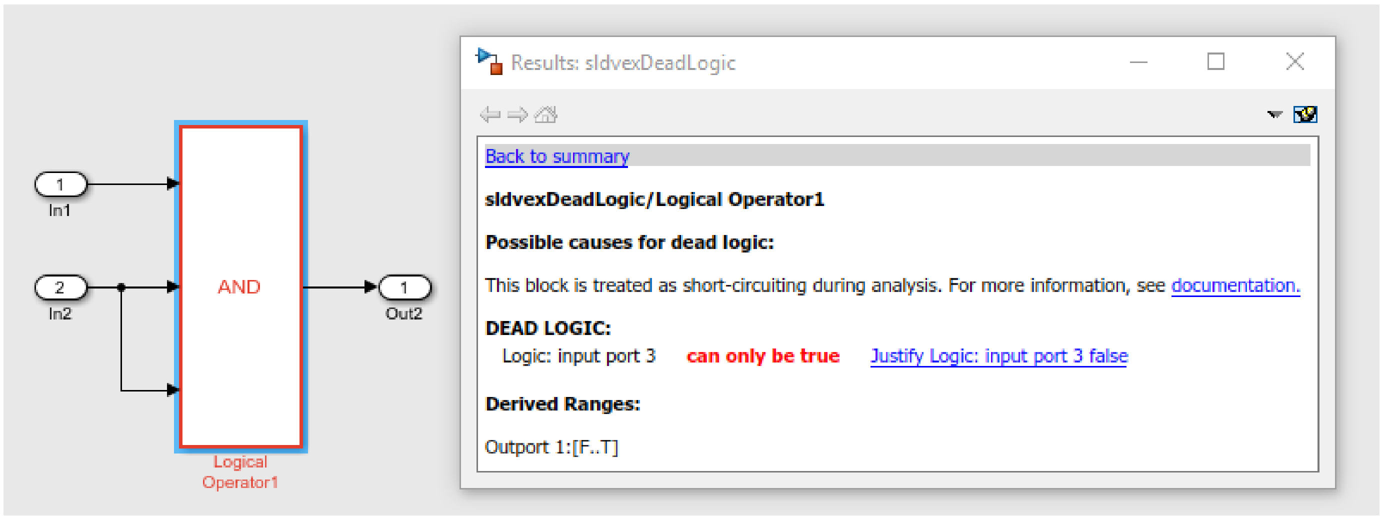
In this section...

When you perform design error detection analysis, Simulink Design Verifier reports the common causes of dead logic in the Results Inspector window.

Short-circuiting of a Logical Operator Block During Analysis

Simulink Design Verifier treats logic blocks as if they are short-circuiting when analyzing for dead logic.

For example, in this model, if `In2` is false, the software ignores the third input due to the short-circuiting. This is suggested as a potential explanation for the dead logic in the Results Inspector window. See "Logic Operations Short-Circuiting" on page 2-25.



sldvexDeadLogic/Logical Operator1

Possible causes for dead logic:

This block is treated as short-circuiting during analysis. For more information, see documentation.

DEAD LOGIC:
Logic: input port 3 **can only be true** Justify Logic: input port 3 false

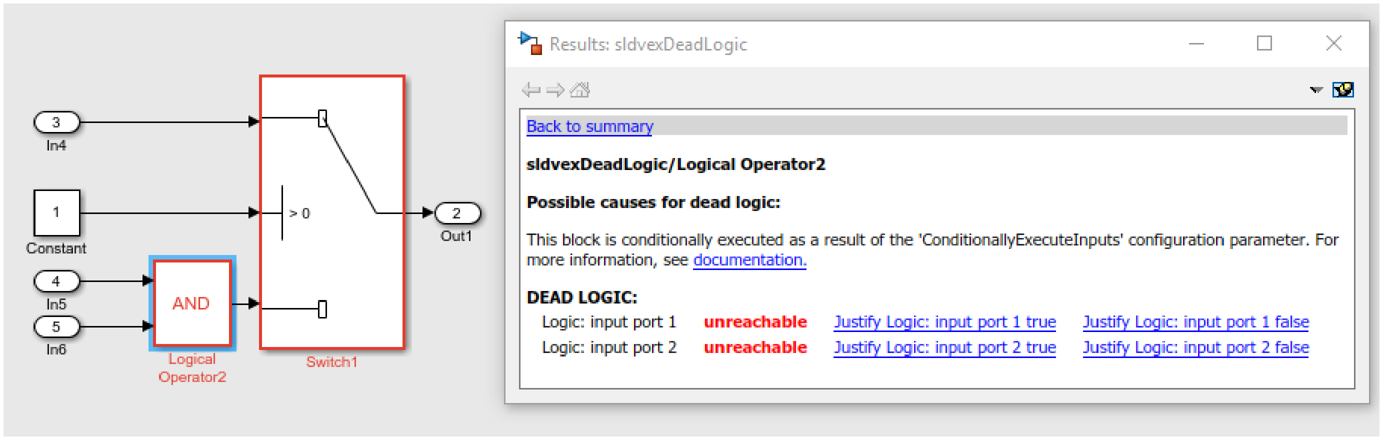
Derived Ranges:

Outport 1:[F..T]

Conditional Execution of a Block

If your model consists of Switch or Multiport Switch blocks and the **Conditional input branch execution** parameter is set to `On`, the conditional execution can often cause unexpected dead logic.

Consider this example model where the **Conditional input branch execution** parameter is set to `On`. The AND Logical Operator block is conditionally executed, which causes the dead logic for the block. For more information, see "Conditional input branch execution" (Simulink).



sldvexDeadLogic/Logical Operator2

Possible causes for dead logic:

This block is conditionally executed as a result of the 'ConditionallyExecuteInputs' configuration parameter. For more information, see documentation.

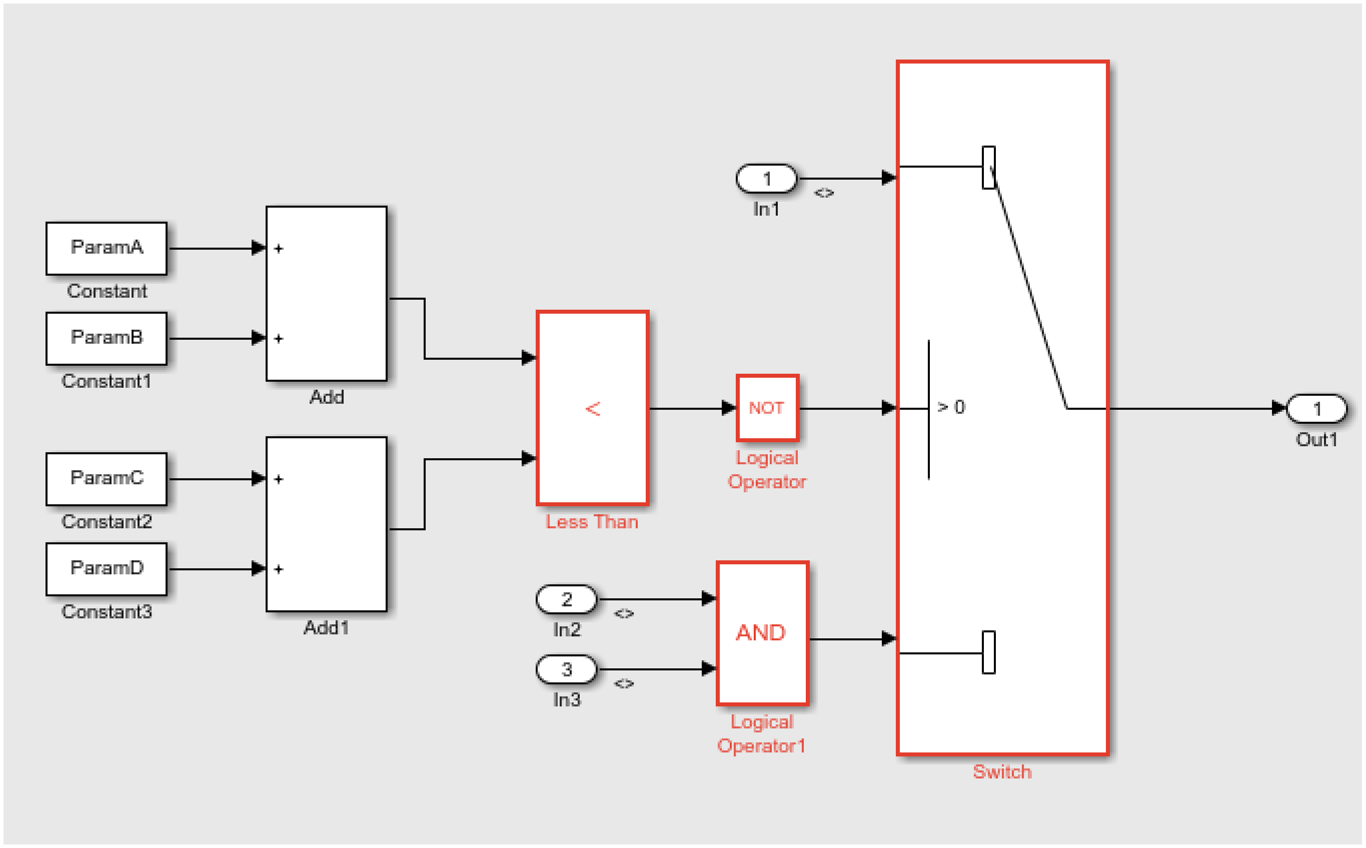
DEAD LOGIC:

Logic: input port 1	**unreachable**	Justify Logic: input port 1 true	Justify Logic: input port 1 false
Logic: input port 2	**unreachable**	Justify Logic: input port 2 true	Justify Logic: input port 2 false

Parameter Values Treated as Constants

If your model contains parameters, Simulink Design Verifier treats the values as constants by default. This might cause dead logic in the model. In these cases, you may wish to consider configuring these parameters to be `tuned` during analysis. For detailed information, see “Inlined Parameters” on page 2-6.

For example, consider this model, where all of the parameters are set to zero. This causes the dead logic for the Less Than block.



Upstream Blocks

When a particular block has dead logic, this often leads to a cascade effect that causes downstream blocks to have dead logic.

Consider the above example model. The dead logic in the Less Than block causes the dead logic in the corresponding downstream blocks. It is therefore often helpful to review the upstream dead logic before reviewing any downstream dead logic.

Library-linked Blocks

Library blocks may be written with defensive conditions that are redundant in some locations where they are used. In some cases, this may cause dead logic. See “Exclude and Justify Objectives for Design Error Detection” on page 6-58.

See Also

More About

- “Run a Dead Logic Analysis and Review Results” on page 6-9
- “Analyzing the Results for a Dead Logic Analysis” on page 6-68

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. On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
3. In the Configuration Parameters dialog box, select **Design Verifier** > **Design Error Detection**.
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. In the **Mode** section, select **Design Error Detection**.
8. Click **Detect Design Errors**.

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" on page 6-25
- "Review the Harness Model" on page 6-27
- "Review the Analysis Report" on page 6-28

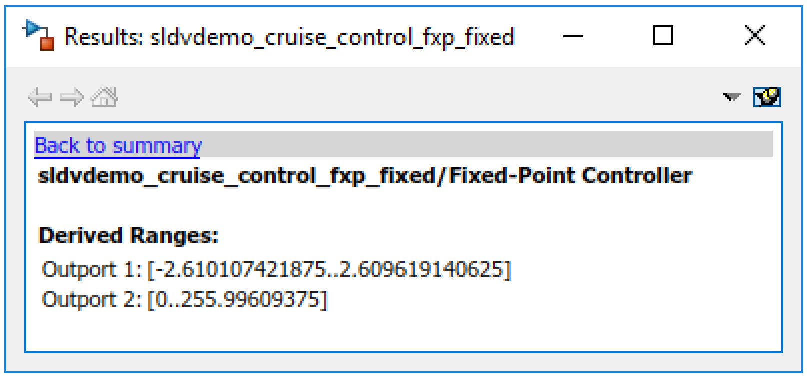
Review the Results on the Model

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 `255.9960`.



Results: sldvdemo_cruise_control_fxp_fixed

Back to summary

sldvdemo_cruise_control_fxp_fixed/Fixed-Point Controller

Derived Ranges:

Outport 1: [-2.610107421875..2.609619140625]

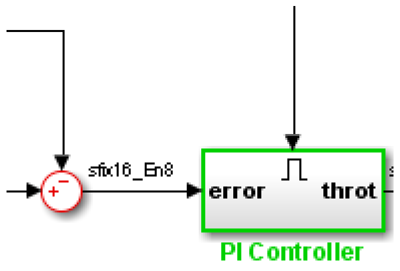
Outport 2: [0..255.99609375]

2 Click the Outport blocks of the `sldvdemo_cruise_control_fxp_fixed` model to see the same signal bound values.

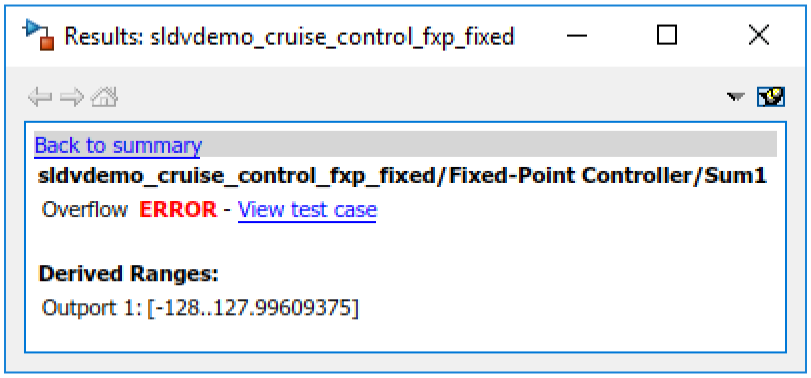
3 Open the Fixed-Point Controller subsystem.

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..127.9960].



Back to summary

sldvdemo_cruise_control_fxp_fixed/Fixed-Point Controller/Sum1

Overflow **ERROR** - View test case

Derived Ranges:

Outport 1: [-128..127.99609375]

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 13-13.

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 13-28.

See Also

More About

- “Detect Integer Overflow Errors” on page 6-50
- “Detect Integer Overflow in a Model with Complex Inputs” on page 6-64

Check for Specified Minimum and Maximum Value Violations

In this section...

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 Value Violations

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

- 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.

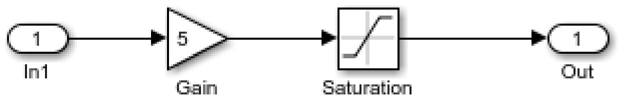
Create the Example Model

Create the model for this example:

1. In the **MATLAB** toolstrip, on the **Home** tab, select **New** > **Simulink 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`
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, on the **Debug** tab, select **Information Overlays** > **Signal Data Ranges**.
5. On the **Modeling** tab, click **Model Settings**.
6. In the Configuration Parameters dialog box, on the **Solver** pane, under **Solver selection**:
   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 **Specified minimum and maximum value violations**.

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`.

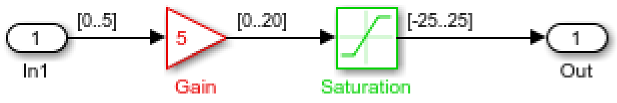
Analyze the Model

To analyze the example model to identify any intermediate signals that violate the specified minimum and maximum values, perform design error detection analysis.

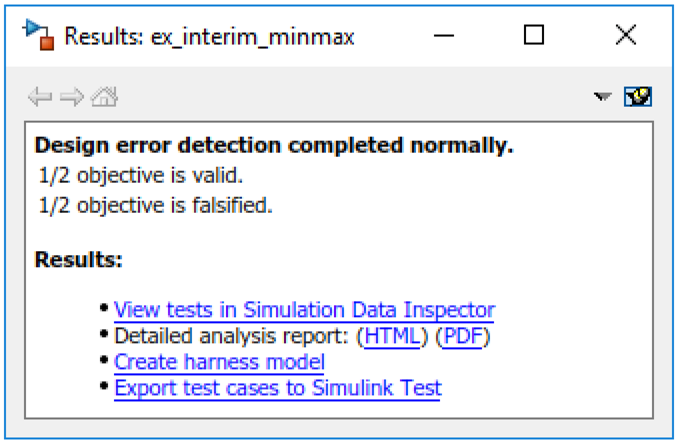
On the **Design Verifier** tab, click **Detect Design Errors**.

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.



Design error detection completed normally.
1/2 objective is valid.
1/2 objective is falsified.

Results:

- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test

Review the Analysis Results

- "Review Results on the Model" on page 6-31
- "Review the Harness Model" on page 6-32
- "Review the Analysis Report" on page 6-33

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.

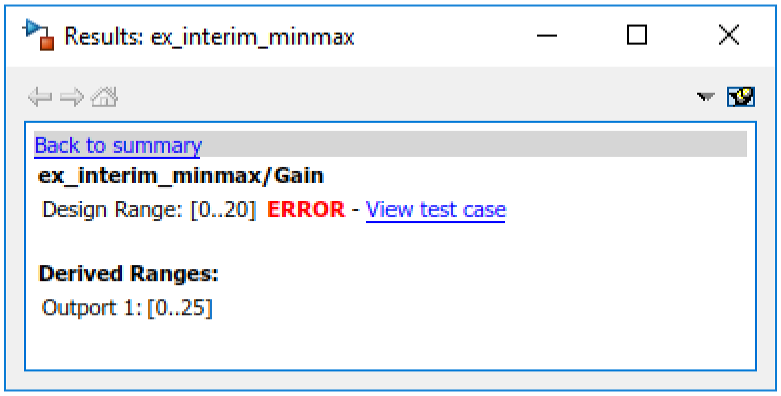
- All objectives associated with the Saturation block were satisfied. For this example, the analysis satisfied exactly one objective.

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.



Back to summary

ex_interim_minmax/Gain

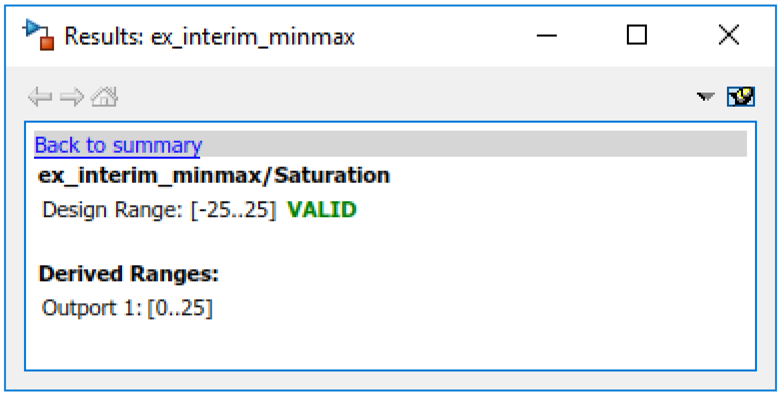
Design Range: [0..20] **ERROR** - View test case

Derived Ranges:

Outport 1: [0..25]

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..25], which is the derived range that the analysis propagated from the Gain block.



Back to summary

ex_interim_minmax/Saturation

Design Range: [-25..25] **VALID**

Derived Ranges:

Outport 1: [0..25]

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

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. On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
2. Click **Error Detection Settings**.
3. In the Configuration Parameters dialog box, in **Design Error Detection** pane, select **Out of bound array access**.
4. Click **OK**.
5. Click **Detect Design Errors**.

   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.

Note If a model contains out of bound array access error, after the first occurrence of array access, Simulink Design Verifier assumes that the array index is within bounds for the remaining analysis. Hence, design error detection objectives that are analyzed after this assumption may be reported as valid, even if the design errors occur in the model.

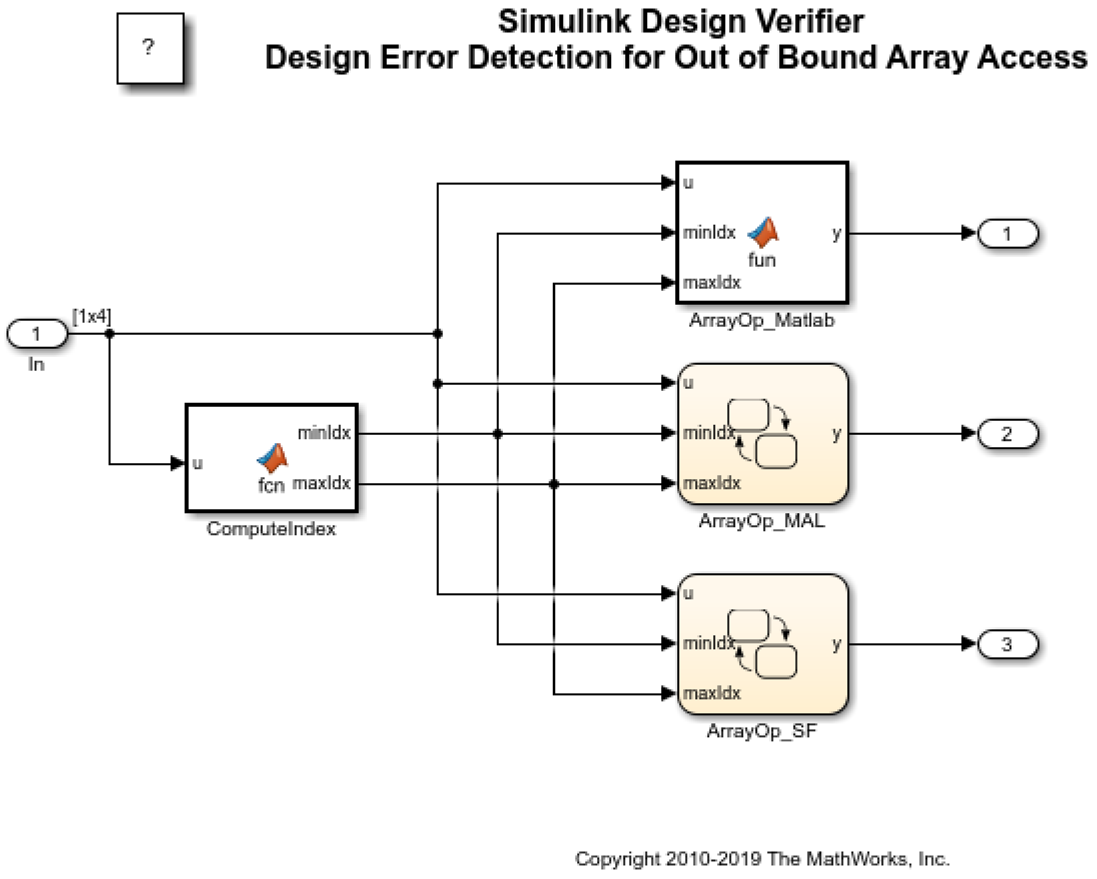
Detect Out of Bound Array Access Example Model

This example shows how to detect out of bound array access errors and review the analysis results. In the `sldvdemo_array_bounds` example model, the ComputeIndex MATLAB Function block uses the input signal values to determine 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.

Step 1: Open the Model

At the command prompt, enter:

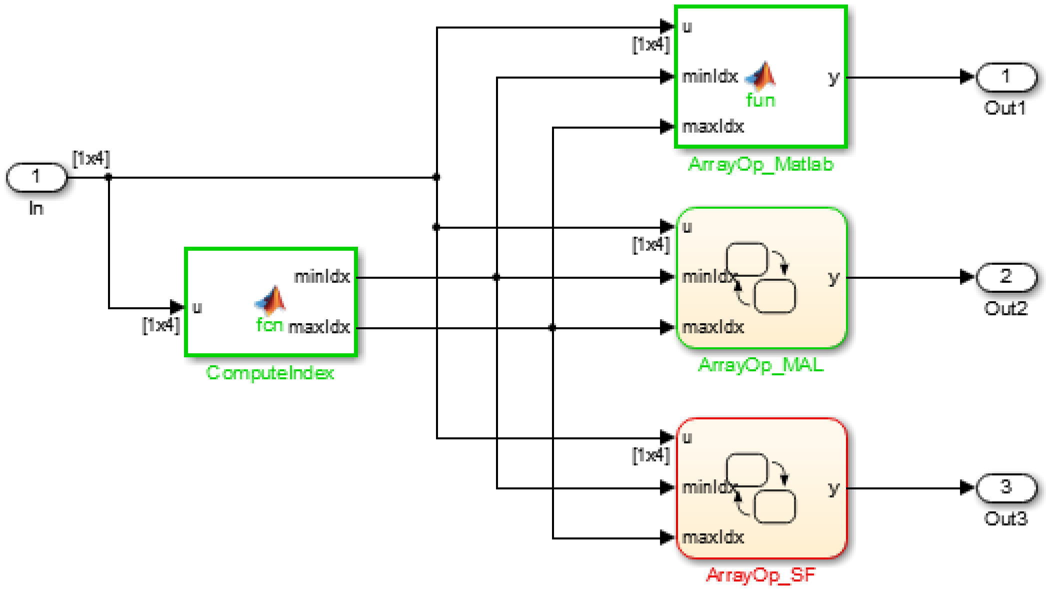
```
open_system('sldvdemo_array_bounds');
```



Step 2: Perform Design Error Detection Analysis

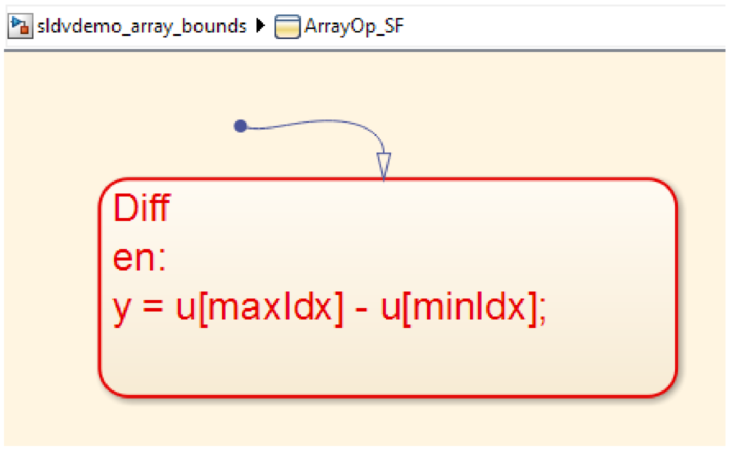
The analysis options in the model are preconfigured for out of bound array access error detection. To view these options, in the Simulink Editor, double-click the **View Options** button.

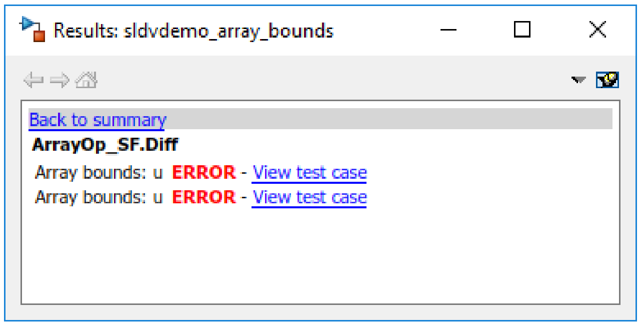
To perform design error detection analysis, in the Simulink Editor, double-click the **Run** button. The Simulink® Design Verifier™ Results Summary window opens that displays the progress of the analysis. When the analysis completes, the example model is highlighted with the analysis results.



Step 3: Review Analysis Results

To view the analysis results inside the chart, double-click the ArrayOp_SF Chart block that is highlighted in red.


sldvdemo_array_bounds ▸ ArrayOp_SF
Diff
en:
y = u[maxIdx] - u[minIdx];



Back to summary

ArrayOp_SF.Diff

Array bounds: u **ERROR** - View test case

Array bounds: u **ERROR** - View test case

Simulink Design Verifier detects that the index out of bound errors occurs in array u in state Diff.

Step 4: Create Harness and Simulate Test Cases

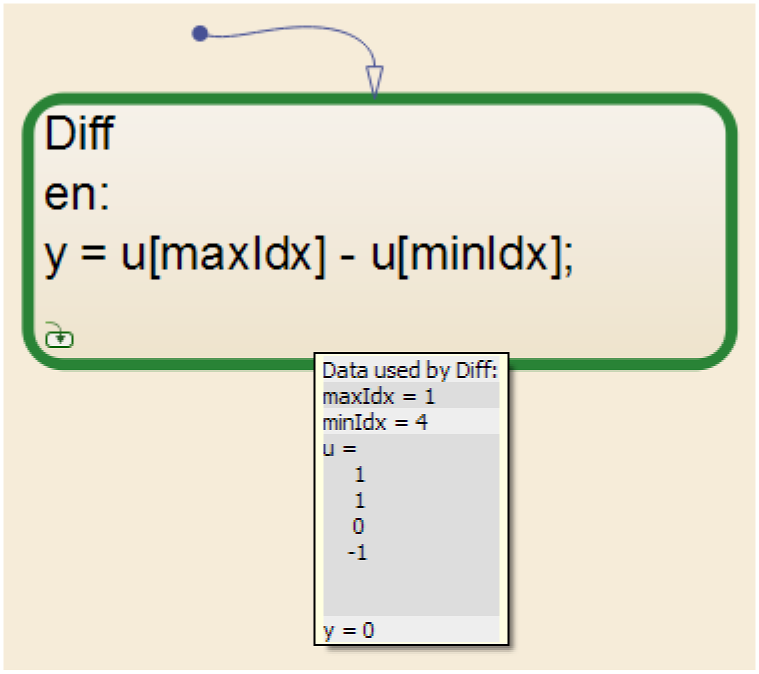
Click the first **View test case** link. Simulink Design Verifier creates and opens a harness model that contains test cases, that demonstrate out of bound array access errors. In the Signal Builder dialog box, click **Start simulation** to simulate the harness model with Test Case 2.

The simulation stops before entering the state Diff. The Stateflow® Debugger opens. The following error is shown:

```
Attempted to access index 4 of u with smaller dimension sizes. The valid
index range is 0 to 3. This error will stop the simulation. State 'Diff' in
```

```
Chart 'sldvdemo_array_bounds_harness/Test Unit (copied from
sldvdemo_array_bounds)/ArrayOp_SF': y = u[maxIdx] - u[minIdx];
```

Keep the Stateflow® Debugger open at this breakpoint. In the `sldvdemo_array_bounds_harness` model, hold your cursor over the Diff state to see the data values at this simulation breakpoint.



```
Data used by Diff:
maxIdx = 1
minIdx = 4
u =
    1
    1
    0
   -1

y = 0
```

Using Test Case 2 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. Thus, 1:4 is not a valid index range for array access in the chart. The valid index range for array access in the chart is 0:3, as reported by the error message. 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” (Stateflow).

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.

See Also

More About

- “Detect Out of Bound Array Access Example Model” on page 6-53

Detect Non-Finite, NaN, and Subnormal Floating-Point Values

To detect occurrences of nonfinite, NaN, and subnormal floating-point values in a model:

1. On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
2. Click **Error Detection Settings**.
3. In the Configuration Parameters dialog box, in **Design Error Detection** pane:
   - a. Select the check box for “Non-finite and NaN floating-point values” on page 15-46.
   - b. Select the check box for “Subnormal floating-point values” on page 15-46.
   - c. To apply these settings, click **OK** and close the Configuration Parameters dialog box.
4. Click **Detect Design Errors**.

Simulink Design Verifier analyzes the model to detect the occurrences of nonfinite, NaN, and subnormal floating-point values.

After the analysis is complete:

- The software highlights the model with the analysis results.
- The Results Summary windows displays the summary of the analysis.

Assumptions and Limitations

When you analyze a model and select “Non-finite and NaN floating-point values” on page 15-46, the software assumes that the floating-point input values and the tunable parameter values are finite.

When you analyze a model and select “Subnormal floating-point values” on page 15-46, the software assumes that the floating-point input values and the tunable parameter values are normal.

Models that use double-precision floating-point signals take more time to analyze than similar models that use single-precision floating-point signals. As a result, models that use double-precision floating-point signals might time out whereas similar models that use single-precision floating-point signals complete their analysis. To improve analysis performance, consider specifying minimum and maximum values that mimic environmental constraints on root-level Inport blocks.

If the model contains cast operations between floating-point signals and multiword fixed-point signals, the analysis might not be able to decide all objectives.

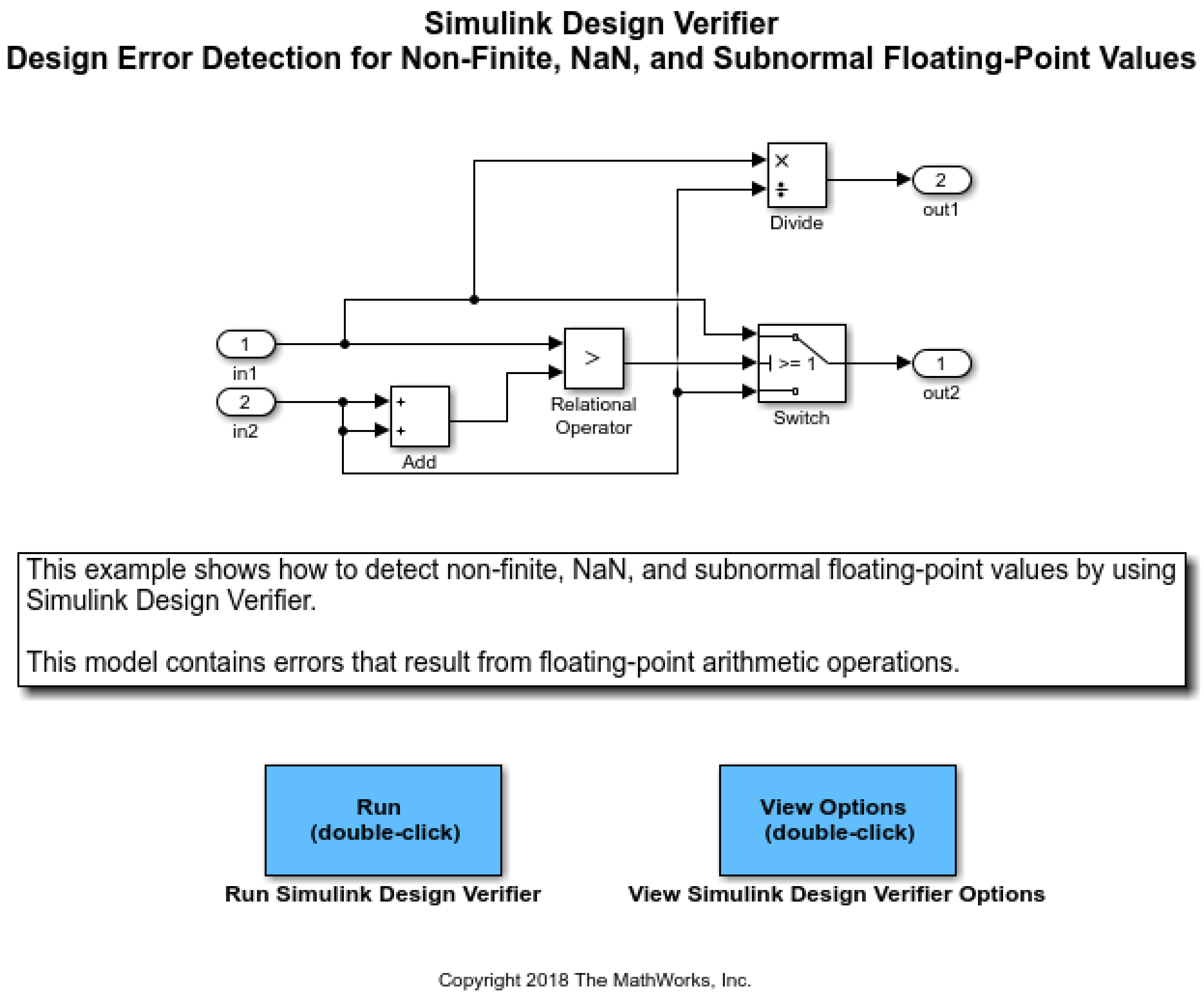
Run Design Error Detection Analysis to Detect Floating-Point Errors

This example shows how to detect nonfinite, NaN, and subnormal floating-point values in the `sldvexFloatingPointErrorChecks` example model. The model consists of floating-point arithmetic operations that result in an error. Perform design error detection analysis to detect these errors in the model.

1. Open the Model

This example model consists of Add and Divide blocks that handle floating-point calculations. The design error detection analysis detects the occurrences of floating-point errors in the model and reports the results.

```
open_system('sldvexFloatingPointErrorChecks');
```


Simulink Design Verifier
Design Error Detection for Non-Finite, NaN, and Subnormal Floating-Point Values

This example shows how to detect non-finite, NaN, and subnormal floating-point values by using Simulink Design Verifier.

This model contains errors that result from floating-point arithmetic operations.

Copyright 2018 The MathWorks, Inc.

Divide
2
out1
1
in1
2
in2
Add
Relational Operator
>= 1
Switch
1
out2
Run (double-click)
Run Simulink Design Verifier
View Options (double-click)
View Simulink Design Verifier Options

2. Perform Design Error Detection Analysis

The model is preconfigured with **Non-finite and NaN floating-point values** and **Subnormal floating-point values** options set to **On**. For more information see “Design Verifier Pane: Design Error Detection” on page 15-43.

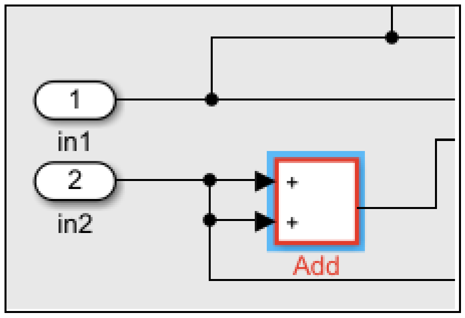
To perform design error detection analysis, on the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**. Click **Detect Design Errors**.

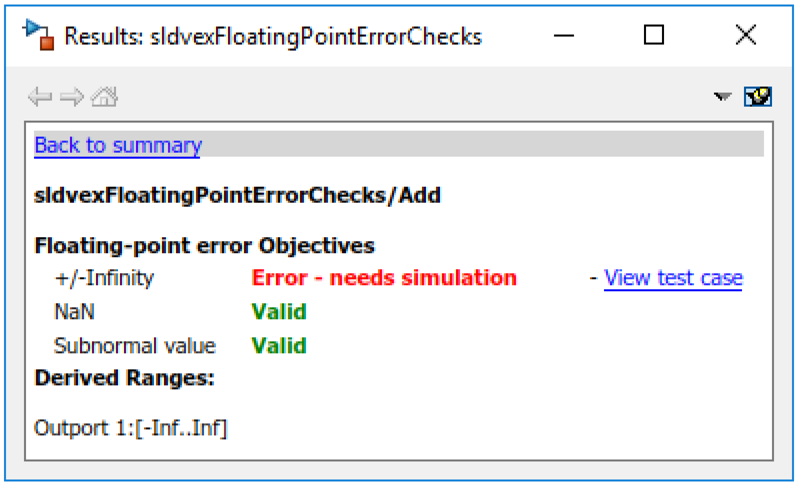
The software analyzes the model for floating-point errors and displays the results in the Results Summary window. The result indicates that 4 out of 6 objectives are falsified.

3. Review Analysis Results

a. Click **Highlight analysis results on model**. The model blocks that result in floating-point errors are highlighted in red.

b. Click the **Add** block highlighted in red. The Result Inspector displays the summary of the floating-point error objectives.





sldvexFloatingPointErrorChecks/Add

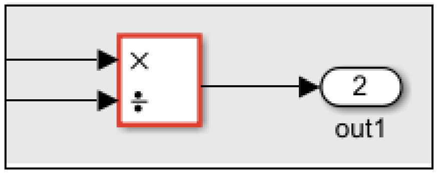
Floating-point error Objectives

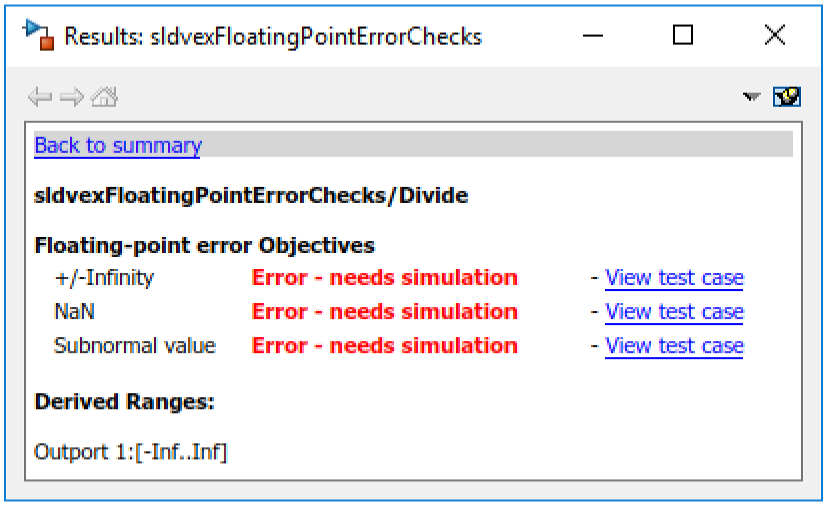
+/-Infinity	**Error - needs simulation**	- View test case
NaN	**Valid**	
Subnormal value	**Valid**	

Derived Ranges:

Outport 1:[-Inf..Inf]

c. Click the **Division** block highlighted in red. The Result Inspector displays the summary of the floating-point error objectives.





sldvexFloatingPointErrorChecks/Divide

Floating-point error Objectives

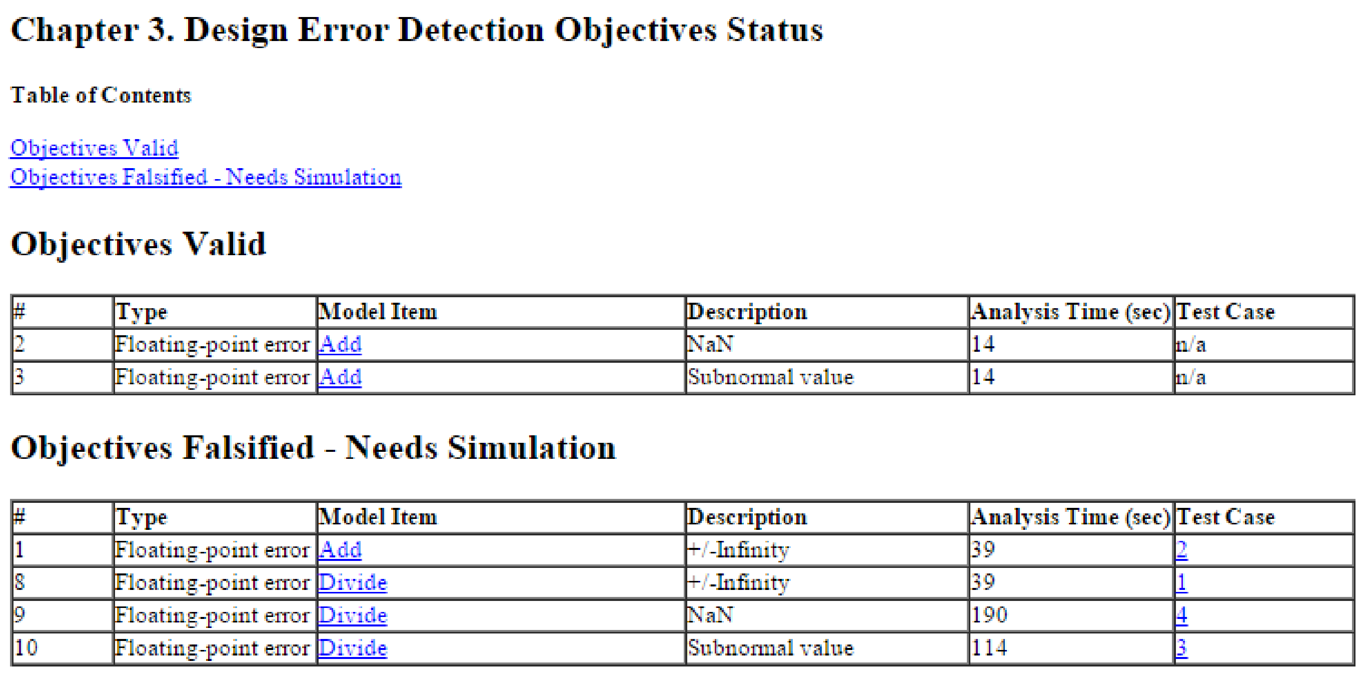
+/-Infinity	**Error - needs simulation**	- View test case
NaN	**Error - needs simulation**	- View test case
Subnormal value	**Error - needs simulation**	- View test case

Derived Ranges:

Outport 1:[-Inf..Inf]

4. View Detailed Analysis Report

To view the detailed analysis report, in the Results Summary window, click **HTML**. The report displays the summary of all occurrences of floating-point errors in the model.



Chapter 3. Design Error Detection Objectives Status

Table of Contents

Objectives Valid

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
2	Floating-point error	Add	NaN	14	n/a
3	Floating-point error	Add	Subnormal value	14	n/a

Objectives Falsified - Needs Simulation

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Floating-point error	Add	+/-Infinity	39	2
8	Floating-point error	Divide	+/-Infinity	39	1
9	Floating-point error	Divide	NaN	190	4
10	Floating-point error	Divide	Subnormal value	114	3

5. Clean Up

To complete this example, close the model.

```
close_system('sldvexFloatingPointErrorChecks', 0);
```

See Also

More About

- "Design Verifier Pane: Design Error Detection" on page 15-43
- "Simulink Design Verifier Options" on page 15-2

Detect Data Store Access Violations

Simulink Design Verifier design error detection analysis identifies unintended sequences of data store reads and writes that occur during simulation. The analysis detects these data store access violations:

- Read-before-write
- Write-after-read
- Write-after-write

To detect data store access violations in your model:

1. On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
2. Click **Error Detection Settings**.
3. In the Configuration Parameters dialog box, in the **Design Error Detection** pane, select “Data store access violations” on page 15-47. Click **OK**.
4. Click **Detect Design Errors**.

After the analysis is complete, the software highlights the model with the analysis results and the Results Summary window displays the summary of the analysis.

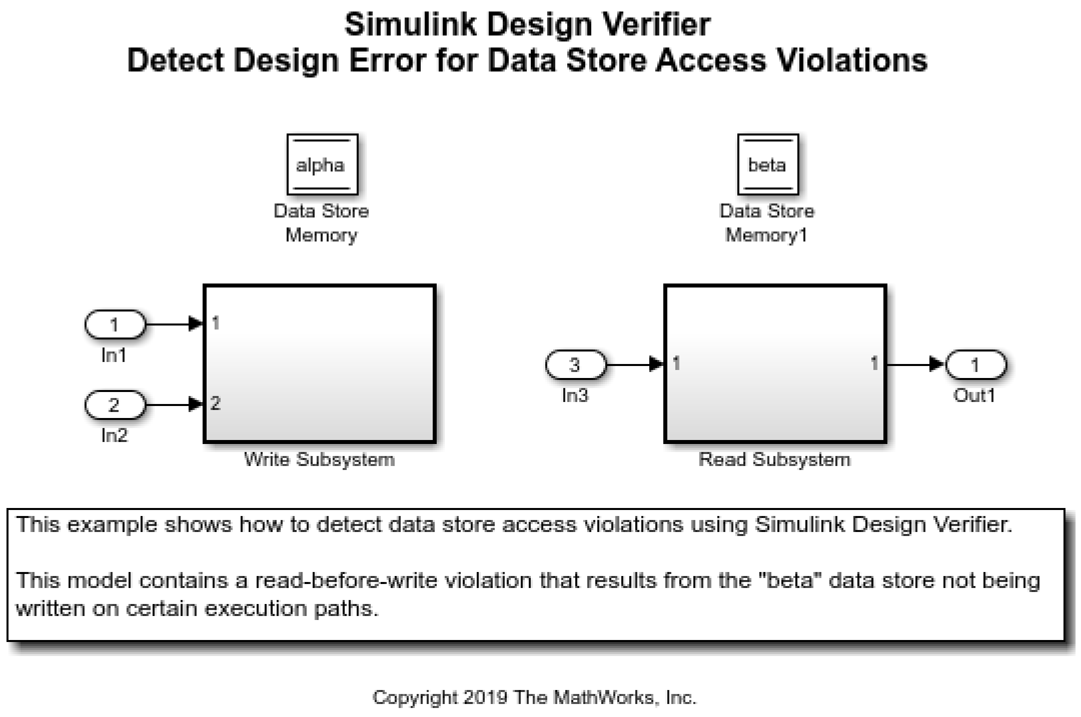
Detect Data Store Access Violations in a Model

This example shows how to detect data store access violations and review the analysis results. The `sldvexDataStoreAccessViolations` example model consists of Data Store Memory blocks that define the `alpha` and `beta` data stores. In the example model, the `Write Subsystem` writes the data to the data store by using Data Store Write blocks and the `Read Subsystem` reads the data from the data store by using the Data Store Read blocks.

Step 1: Open the Model

At the command prompt, enter:

```
open_system('sldvexDataStoreAccessViolations');
```



This example shows how to detect data store access violations using Simulink Design Verifier.

This model contains a read-before-write violation that results from the "beta" data store not being written on certain execution paths.

Step 2: Configure Analysis Options to Detect Data Store Access Violations

The model is preconfigured with the **Data store access violations** parameter set to `On`.

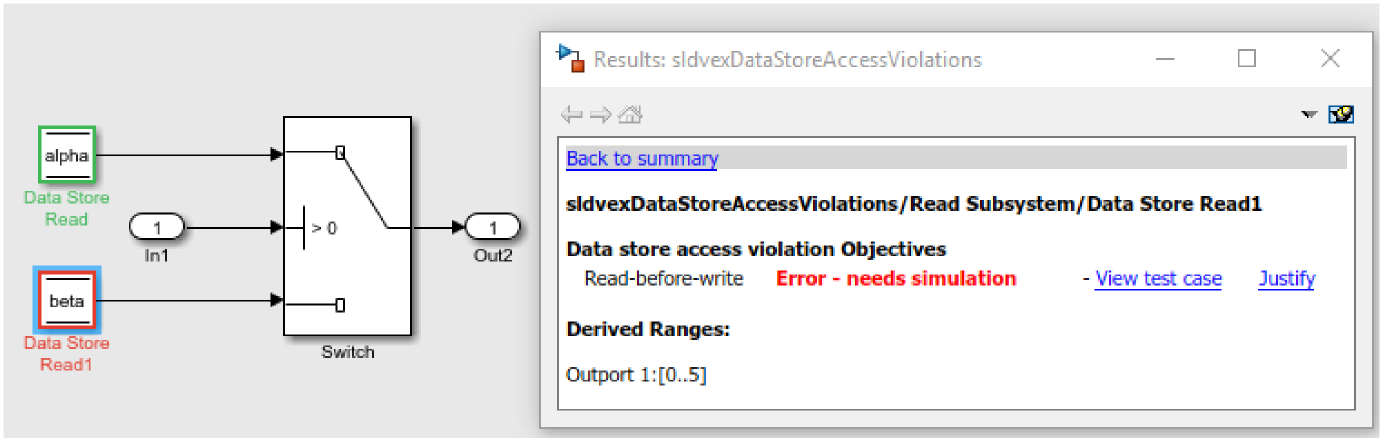
Step 3: Perform Design Error Detection Analysis

On the **Design Verifier** tab, click **Detect Design Errors**. Simulink Design Verifier analyzes the model for data store access violations. After the analysis completes, the Results Summary window displays that one objective was falsified.

Step 4: Review Analysis Results

The model is highlighted with the analysis results.

(1) Open the `Read Subsystem` and click `Data Store Read1` block that is highlighted in red. The Results Inspector window displays the Read-before-write objective that violates the data store access order.



sldvexDataStoreAccessViolations/Read Subsystem/Data Store Read1

Data store access violation Objectives

Read-before-write **Error - needs simulation** - View test case Justify

Derived Ranges:

Outport 1:[0..5]

(2) To view the test case that replicates the error, click **View test case**. The harness model and the Signal Builder block open that displays the test case.

(3) To simulate the test case, in the Signal Builder dialog box, click **Start simulation**. After the simulation completes, the Diagnostic Viewer window displays this warning message:

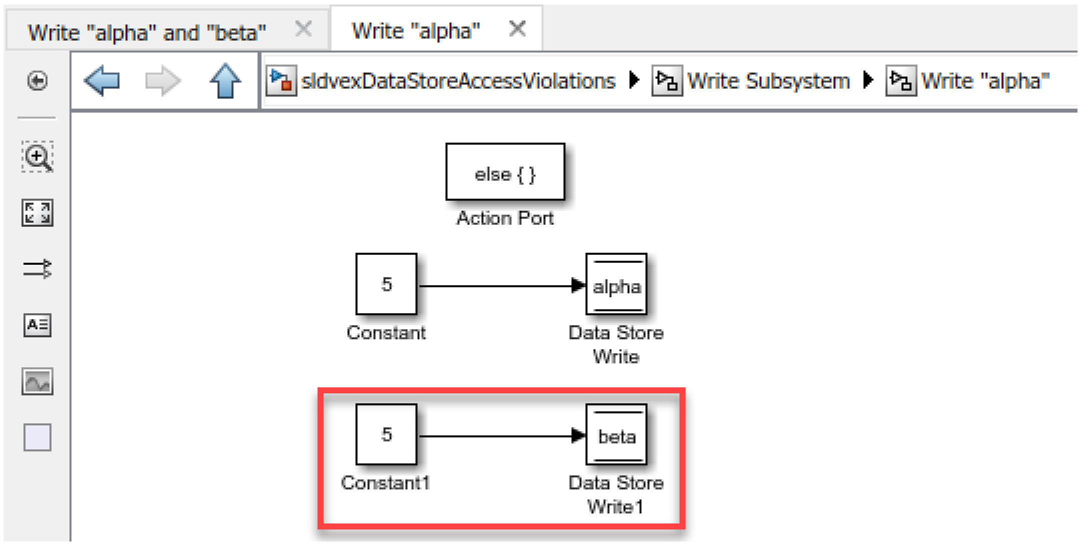
```
The block 'sldvexDataStoreAccessViolations_harness/Test Unit (copied from
sldvexDataStoreAccessViolations)/Read Subsystem/Data Store Read1' is reading
from the data store 'sldvexDataStoreAccessViolations_harness/Test Unit
(copied from sldvexDataStoreAccessViolations)/Data Store Memory1' before any
blocks have written to this entire region of memory at time 0.0. For
performance reasons, occurrences of this diagnostic for this memory at other
simulation time steps will be suppressed.
```

Step 5: Fix the Data Store Access Violation Error

The read-before-write objective results in error because no block has been written to the `beta` data store before the read operation executes.

Open the `Write Subsystem` and double-click `Write "alpha"`. In the `Write "alpha"` subsystem, only the `alpha` data store is written with a constant value. Hence, the read-before-write data store access violation occurs for the "beta" Data Store Read block.

To fix the error, in the `Write "alpha"` subsystem, add a `Constant` block and write its value to `beta` data store by using the Data Store Write block (highlighted in figure below).



On the **Design Verifier** tab, click **Detect Design Errors**. After the analysis completes, the software reports that all the objectives are valid.

See Also

- “Data Store Basics” (Simulink)
- “Detect Data Store Access Violations”

See Also

More About

- “Design Verifier Pane: Design Error Detection” on page 15-43

Filter Objectives by Using Analysis Filter Viewer

Filtering model objects from design error detection or test generation analysis allows you to focus on a subset of objects for Simulink Design Verifier analysis. If you have a large model, there can be model objects that take a long time to analyze or model objects that you can manually prove do not result in errors. You can exclude these objects from analysis by using a coverage filter file. You can add a coverage filter file by opening the Configuration Parameters window, clicking **Design Verifier**, and under **Advanced parameters**, selecting “Ignore objectives based on filter” on page 15-17. Select your coverage filter file for the **Filter file**. For more information on coverage filter file, see “Creating and Using Coverage Filters” (Simulink Coverage).

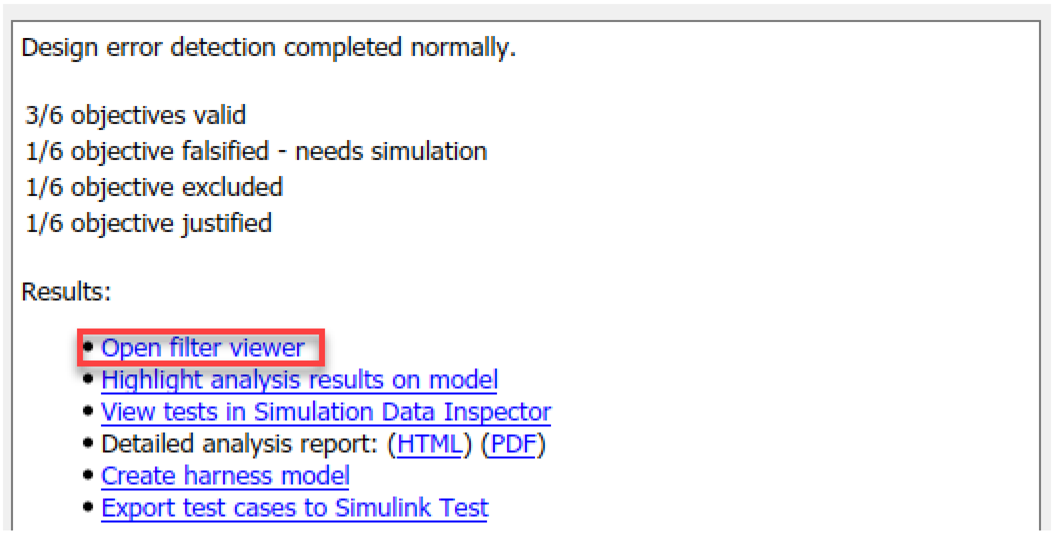
After you perform design error detection or test generation analysis, you can justify the falsified objectives by using the **Analysis Filter** viewer. When you edit the coverage filter by using **Analysis Filter** viewer, you can update the Simulink Design Verifier report and highlight the analysis results on the model without reanalyzing the model. For detailed example on how to filter objectives, see “Exclude and Justify Objectives for Design Error Detection” on page 6-58.

Use the Analysis Filter Viewer to Edit Coverage Filter Files

After analyzing your model, you can use **Analysis Filter** viewer to justify the falsified objectives and update the coverage filter file.

You can open the **Analysis Filter** viewer from the Results Summary window or from the Results Inspector window.

- In the Results Summary window, click **Open filter viewer**.



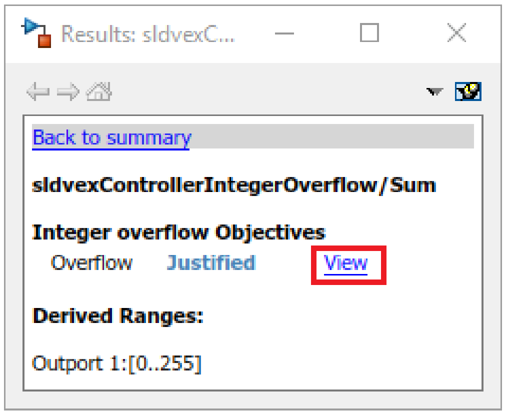
Design error detection completed normally.

3/6 objectives valid
1/6 objective falsified - needs simulation
1/6 objective excluded
1/6 objective justified

Results:

- Open filter viewer
- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test

- In the Results Inspector window,
  - To see a justified objective, click **View**.
  - To justify objective that results in error, click **Justify**.


Results: sldvexC...

Back to summary

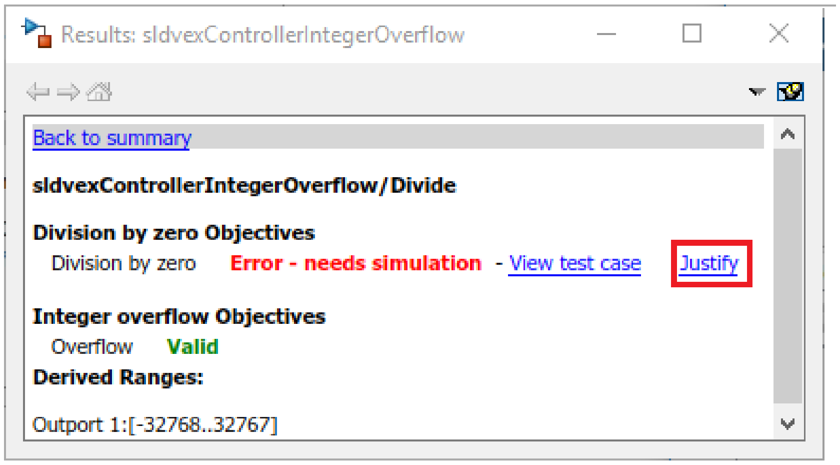
sldvexControllerIntegerOverflow/Sum

Integer overflow Objectives

Overflow **Justified** View

Derived Ranges:

Outport 1:[0..255]


Results: sldvexControllerIntegerOverflow

Back to summary

sldvexControllerIntegerOverflow/Divide

Division by zero Objectives

Division by zero **Error - needs simulation** - View test case Justify

Integer overflow Objectives

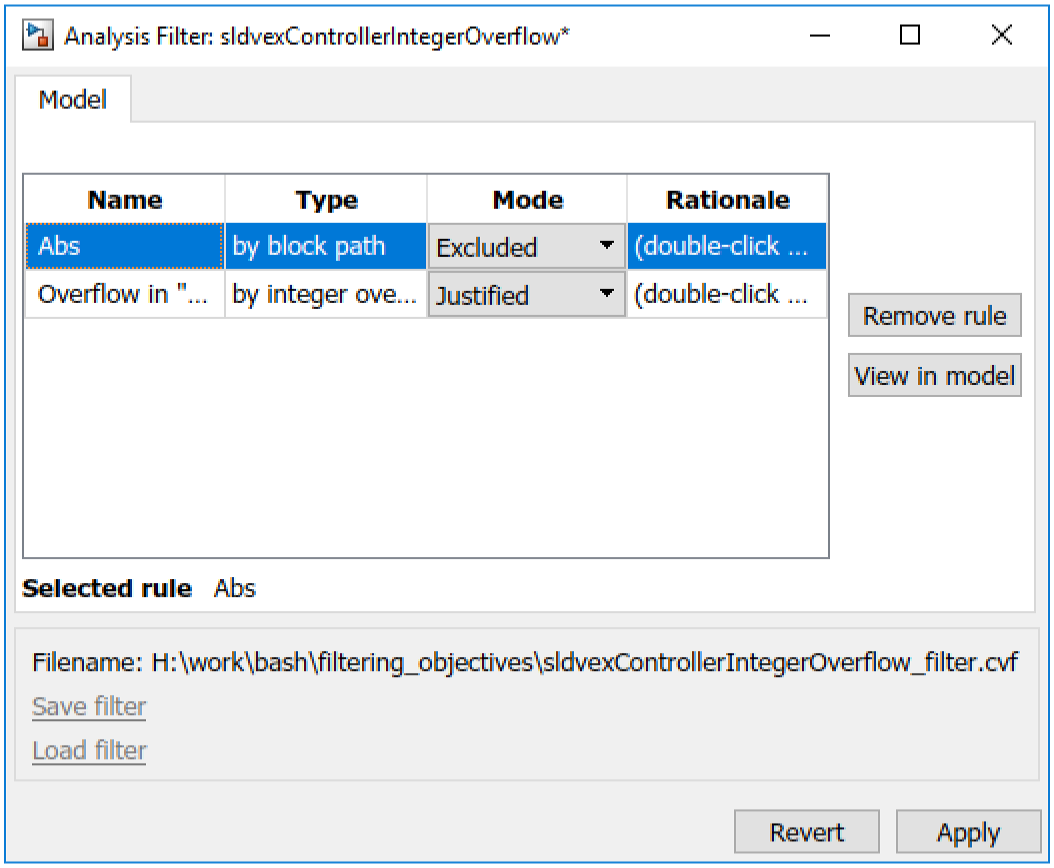
Overflow **Valid**

Derived Ranges:

Outport 1:[-32768..32767]

In the Analysis Filter viewer, you can:

- Review and manage the filter rules for analysis.
- Load or save analysis filter files in your model.
- Navigate to the model to create additional filter rules.
- Add rationale description about why the objective or model object is excluded or justified.


Analysis Filter: sldvexControllerIntegerOverflow*

Model

Name	Type	Mode	Rationale
Abs	by block path	Excluded	(double-click ...
Overflow in "...	by integer ove...	Justified	(double-click ...

Selected rule Abs

Filename: H:\work\bash\filtering_objectives\sldvexControllerIntegerOverflow_filter.cvf

Task	Action
Navigate to a model object associated with a rule.	**1** Select the rule. **2** Click **View in model**. The model object is highlighted in blue.
Delete a rule.	**1** Select the rule. **2** Click **Remove rule**.
Save the current rules to a file.	**1** Click **Apply**. **2** Click **Save filter**. **3** Specify a file name and folder for the filter file and click **Save**.
Load an existing coverage filter file.	**1** Click **Load filter**. **2** Navigate to the filter file and click **Open**.
Highlight the model and update the current analysis report with the current filtering rules.	**1** **Apply** or **Revert** any changes you have made. The model is highlighted with the updated filter rules. **2** In the Results Summary window or in the Results inspector window, click **HTML** or **PDF**.

Limitations

Simulink Design Verifier does not support filtering of these objectives:

- Objectives associated with S-function and custom C/C++ code.
- Objectives associated with property proving analysis.
- Test generation objectives associated with tests for Embedded Coder generated code.
- MCDC and relational boundary objectives. You can exclude these objectives from analysis, but justifying them after the analysis is not supported.

See Also

More About

- “Design Verifier Pane” on page 15-9
- “Create, Edit, and View Coverage Filter Rules” (Simulink Coverage)
- “Simulink Design Verifier Reports” on page 13-28

Detect Integer Overflow Errors

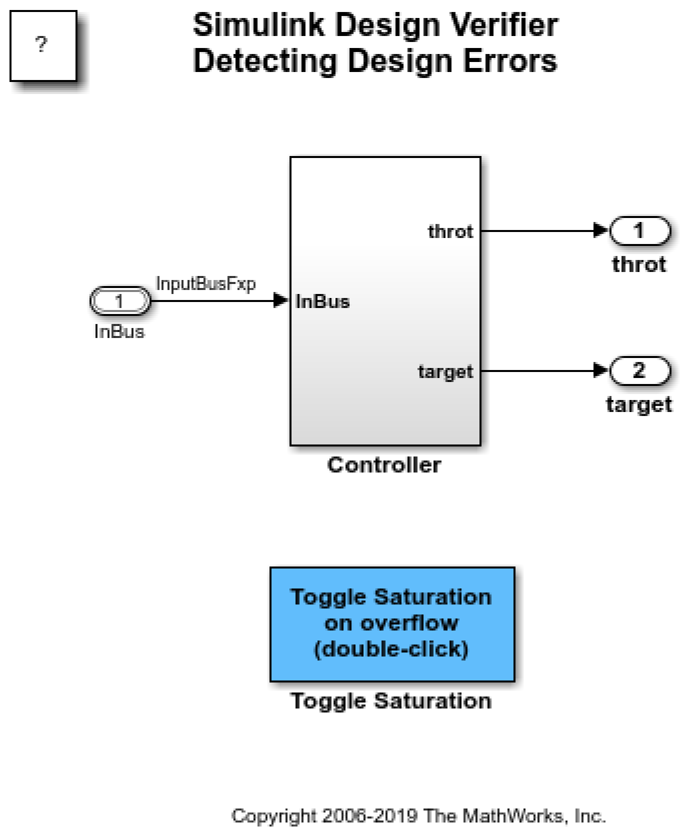
This example shows how to detect integer overflow errors in a model by using design error detection analysis. Simulink® Design Verifier™ identifies the model constructs that may result in integer overflows and then either proves that the integer overflow cannot occur during simulation or generates test cases that demonstrates the integer overflow error.

In this example, you will perform design error detection analysis on a model, then generate a report that shows which integer overflow objectives were valid and which objectives resulted in errors.

Step 1: Open the Model

At the command prompt, enter:

```
open_system('sldvdemo_design_error_detection');
```

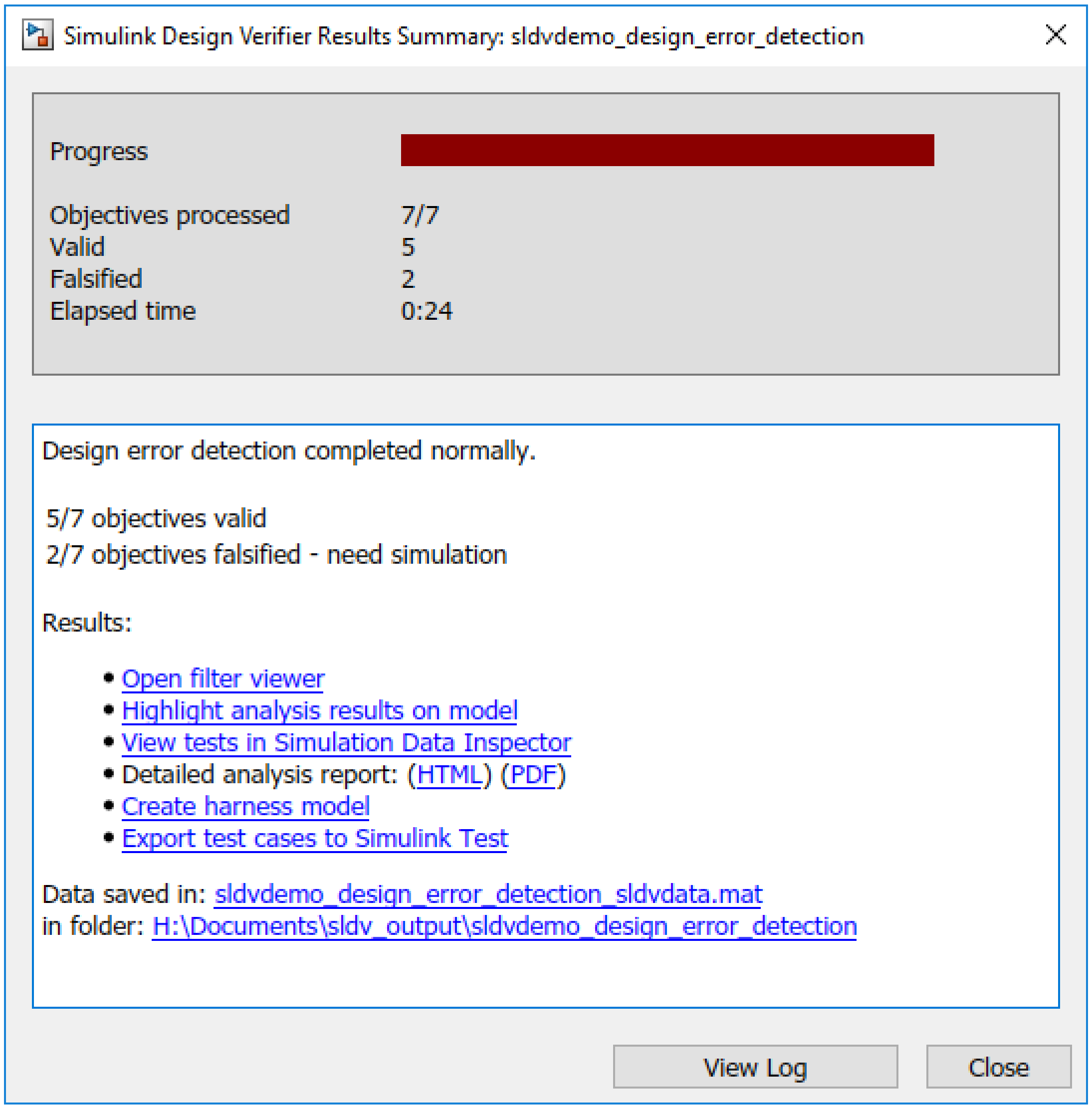


Step 2: Perform Design Error Detection Analysis

The model is preconfigured with the **Integer overflow** option enabled in the Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane.

On the **Design Verifier** tab, click **Detect Design Errors**.

The software analyzes the model for integer overflow errors. After the analysis completes, the Results Summary window reports that five objectives are valid and two objectives are falsified.


Simulink Design Verifier Results Summary: sldvdemo_design_error_detection
View Log
Close

Progress

Objectives processed 7/7
Valid 5
Falsified 2
Elapsed time 0:24

Design error detection completed normally.

5/7 objectives valid
2/7 objectives falsified - need simulation

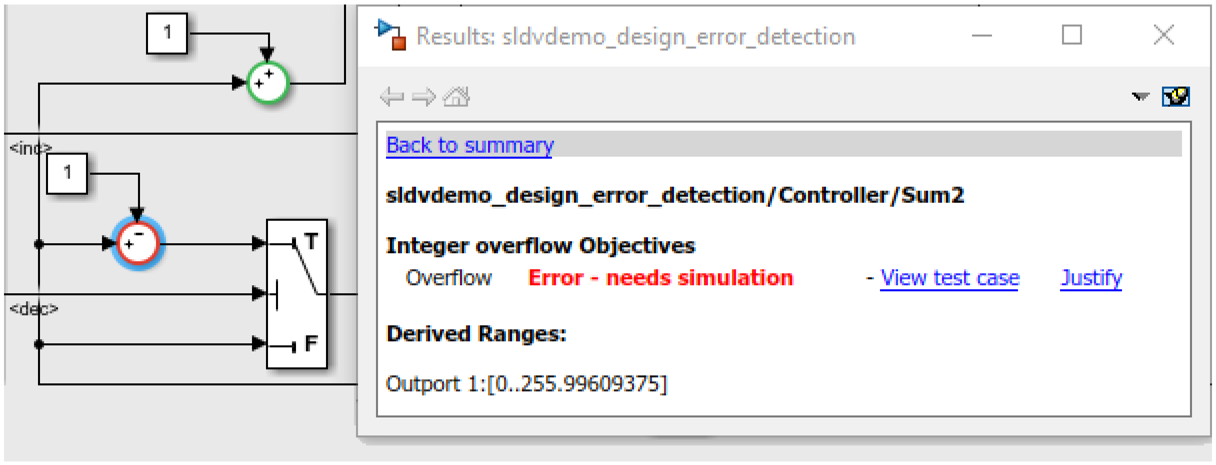
Results:

- Open filter viewer
- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test

Data saved in: sldvdemo_design_error_detection_sldvdata.mat
in folder: H:\Documents\sldv_output\sldvdemo_design_error_detection

Step 3: Review Analysis Results

To highlight the analysis results on the model, in the Results Summary window, click **Highlight analysis results on model**. The valid objectives are highlighted in green and the falsified objectives are highlighted in red.



Results: sldvdemo_design_error_detection

Back to summary

sldvdemo_design_error_detection/Controller/Sum2

Integer overflow Objectives

Overflow Error - needs simulation - View test case Justify

Derived Ranges:

Outport 1:[0..255.99609375]

Double-click the `Controller` subsystem. Click the Sum block that is highlighted in red. The Results Inspector window displays the integer overflow objectives.

To view the test case that results in the error, click **View test case**. The harness model opens and the Signal Builder block displays the test case that results in the error.

Step 4: Fix the Integer Overflow Error

For both the Sum blocks that generated the integer overflow, enable the **Saturate on integer overflow** option. Alternatively, you can double-click the **Toggle Saturation on overflow** button in the Simulink Editor.

To confirm that the integer overflow error was resolved, on the **Design Verifier** tab, click **Detect Design Errors**. After the analysis completes, the software reports that all the objectives are valid.

Related Topics

- "Detect Integer Overflow and Division-by-Zero Errors" on page 6-25
- "Understand the Analysis Results" on page 6-4

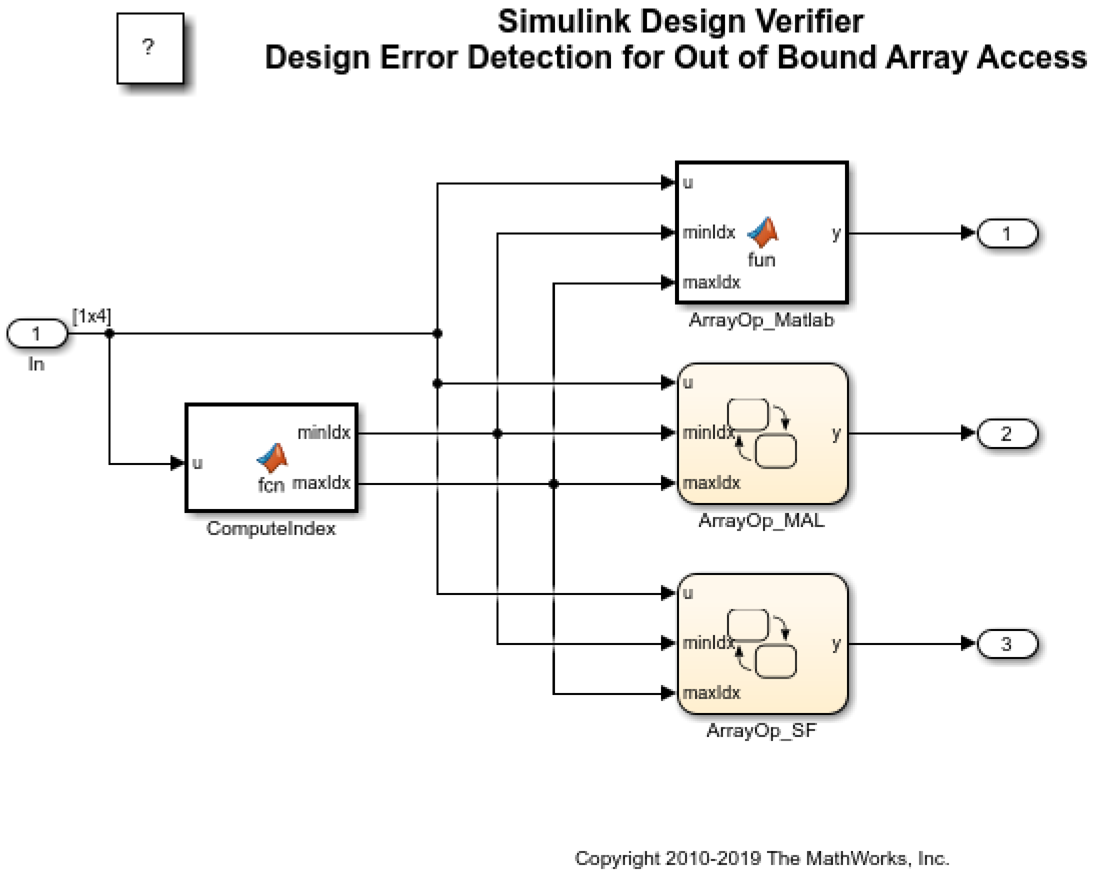
Detect Out of Bound Array Access Example Model

This example shows how to detect out of bound array access errors and review the analysis results. In the `sldvdemo_array_bounds` example model, the ComputeIndex MATLAB Function block uses the input signal values to determine 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.

Step 1: Open the Model

At the command prompt, enter:

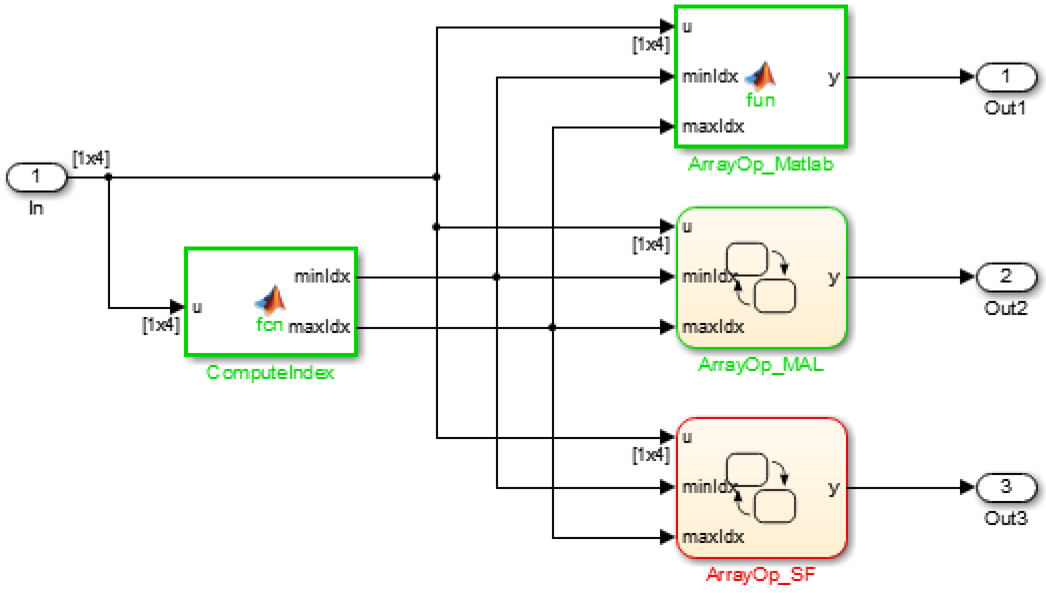
```
open_system('sldvdemo_array_bounds');
```



Step 2: Perform Design Error Detection Analysis

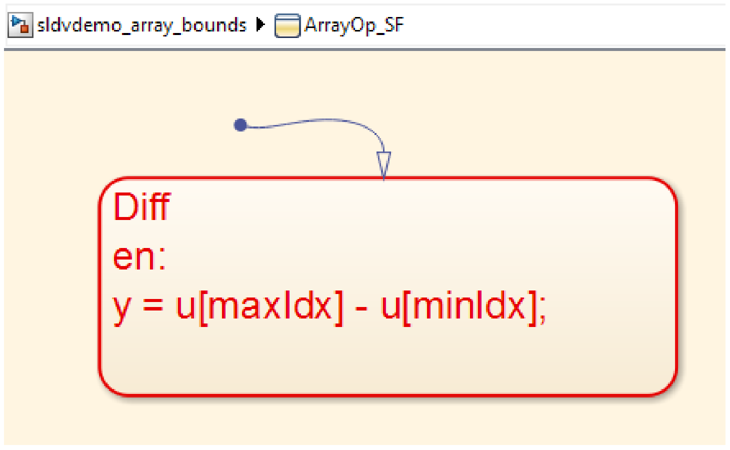
The analysis options in the model are preconfigured for out of bound array access error detection. To view these options, in the Simulink Editor, double-click the **View Options** button.

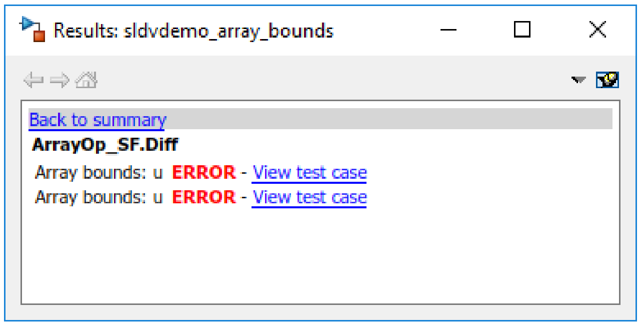
To perform design error detection analysis, in the Simulink Editor, double-click the **Run** button. The Simulink® Design Verifier™ Results Summary window opens that displays the progress of the analysis. When the analysis completes, the example model is highlighted with the analysis results.



Step 3: Review Analysis Results

To view the analysis results inside the chart, double-click the ArrayOp_SF Chart block that is highlighted in red.


sldvdemo_array_bounds ▶ ArrayOp_SF
Diff
en:
y = u[maxIdx] - u[minIdx];



Back to summary

ArrayOp_SF.Diff

Array bounds: u **ERROR** - View test case

Array bounds: u **ERROR** - View test case

Simulink Design Verifier detects that the index out of bound errors occurs in array u in state Diff.

Step 4: Create Harness and Simulate Test Cases

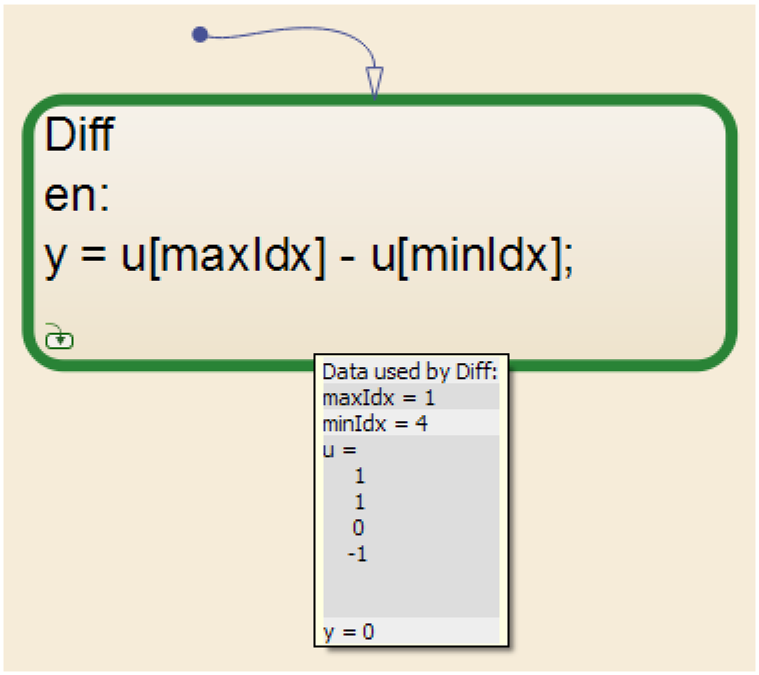
Click the first **View test case** link. Simulink Design Verifier creates and opens a harness model that contains test cases, that demonstrate out of bound array access errors. In the Signal Builder dialog box, click **Start simulation** to simulate the harness model with Test Case 2.

The simulation stops before entering the state Diff. The Stateflow® Debugger opens. The following error is shown:

```
Attempted to access index 4 of u with smaller dimension sizes. The valid
index range is 0 to 3. This error will stop the simulation. State 'Diff' in
```

```
Chart 'sldvdemo_array_bounds_harness/Test Unit (copied from
sldvdemo_array_bounds)/ArrayOp_SF': y = u[maxIdx] - u[minIdx];
```

Keep the Stateflow® Debugger open at this breakpoint. In the `sldvdemo_array_bounds_harness` model, hold your cursor over the Diff state to see the data values at this simulation breakpoint.



```
Diff
en:
y = u[maxIdx] - u[minIdx];
```

```
Data used by Diff:
maxIdx = 1
minIdx = 4
u =
    1
    1
    0
   -1

y = 0
```

Using Test Case 2 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. Thus, 1:4 is not a valid index range for array access in the chart. The valid index range for array access in the chart is 0:3, as reported by the error message. 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” (Stateflow).

Detect Design Errors in C/C++ Custom Code

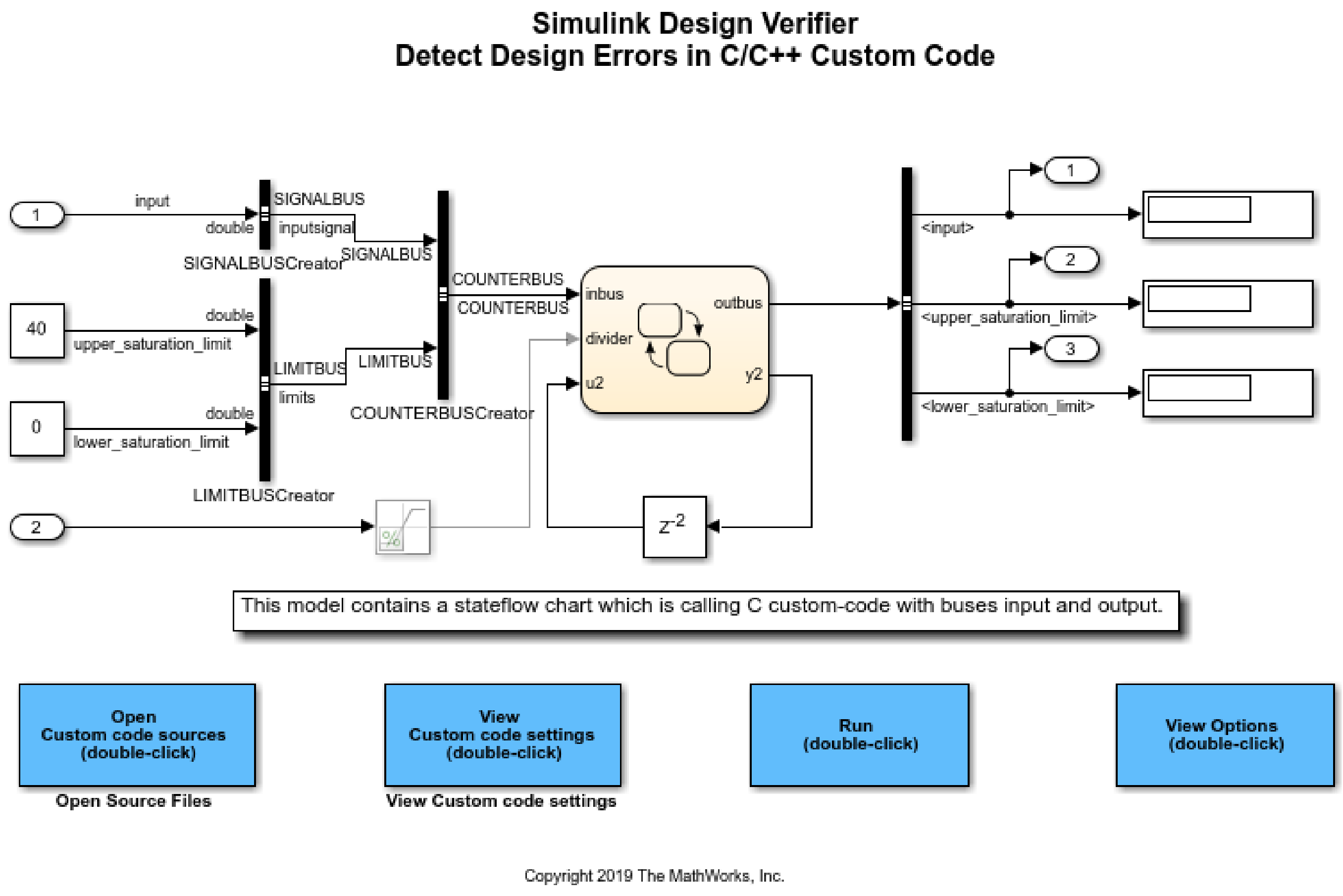
To detect division by zero and out of bound array access errors in a model with C/C++ custom code in model blocks or Stateflow® charts, use design error detection analysis. Simulink Design Verifier identifies the code that results in errors and then either proves that the errors are valid or generates test cases that replicate the error.

This example shows how to detect division by zero errors in a model that consists of C/C++ code in a Stateflow® chart.

Step 1: Open the Model

The example model `sldvexCustomCodeErrorDetectionExample` contains a Stateflow® chart that calls C/C++ custom code that uses input and output buses.

```
open_system('sldvexCustomCodeErrorDetectionExample');
```

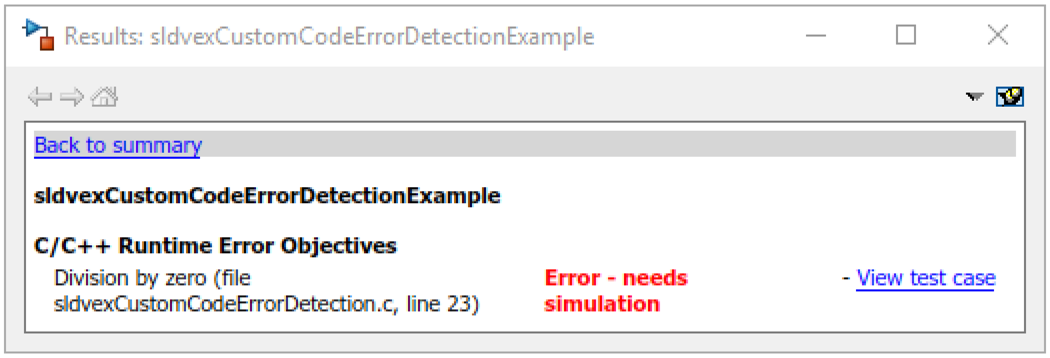


Step 2: Perform Design Error Detection Analysis

To perform design error detection analysis, on the **Design Verifier** tab, click **Detect Design Errors**. After the analysis completes, the Results Summary window indicates that one objective is falsified.

Step 3: Review the Analysis Results

On the **Design Verifier** tab, in the **Review Results** section, click **Highlight in Model**. To view the C/C++ run-time error objectives that resulted in the error, click on the Simulink® Editor. The Results Inspector window displays the division by zero objectives.



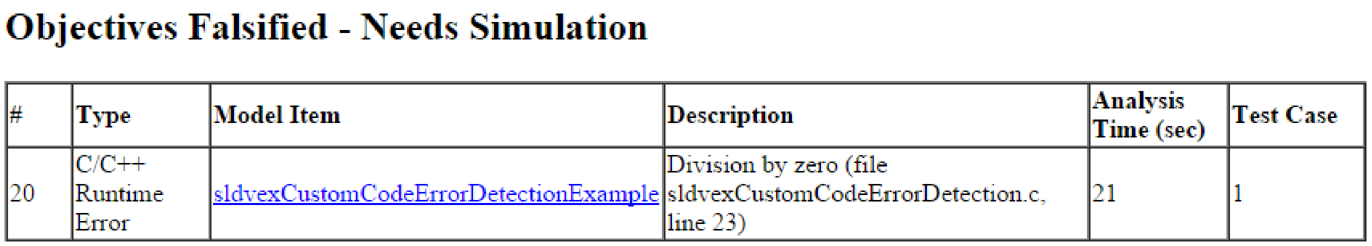
sldvexCustomCodeErrorDetectionExample

C/C++ Runtime Error Objectives

Division by zero (file sldvexCustomCodeErrorDetection.c, line 23)	**Error - needs simulation**	- View test case

Note: When you click **View test case** for the Error - needs simulation objective, Simulink® Design Verifier™ displays the test case that replicates the error. If you simulate the test case, MATLAB® may crash during custom code analysis.

To view the HTML report, on the **Design Verifier** tab, click **HTML Report**. The Design Error Detection Objectives Status section in the report describes the falsified objective.



Objectives Falsified - Needs Simulation

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
20	C/C++ Runtime Error	sldvexCustomCodeErrorDetectionExample	Division by zero (file sldvexCustomCodeErrorDetection.c, line 23)	21	1

Step 4: Fix Design Errors

In the example model, right-click the Saturation block that is greyed out and **Uncomment** the block. Reanalyze the model, by clicking **Detect Design Errors**. The results show that the C/C++ run-time objective is valid.

Step 5: Clean Up

To complete the example, close the model.

```
close_system('sldvexCustomCodeErrorDetectionExample', 0);
```

Related Topics

- “Design Error Detection Objectives Status” on page 13-34
- “Design Verifier Pane: Design Error Detection” on page 15-43

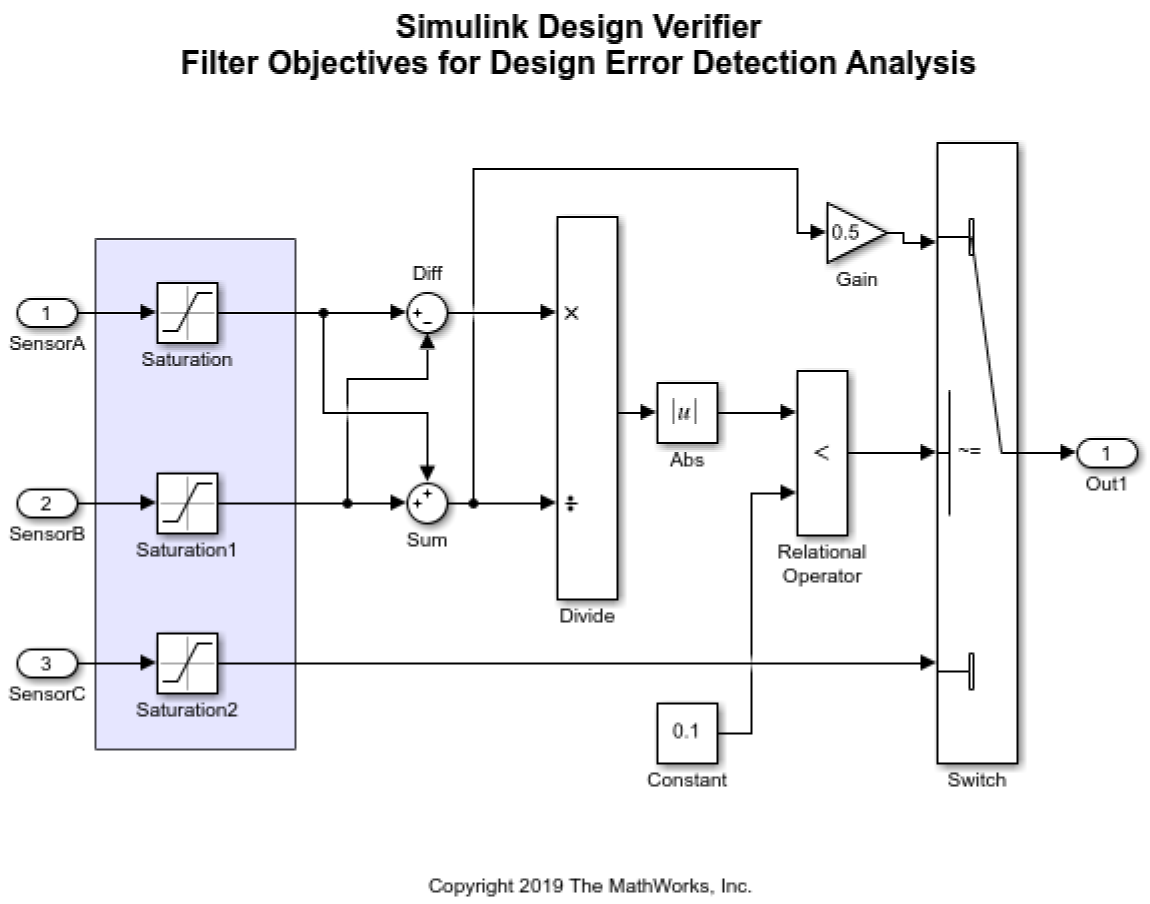
Exclude and Justify Objectives for Design Error Detection

This example shows how to exclude a model object from Simulink® Design Verifier™ analysis by using a coverage filter file. After performing analysis, you can justify objectives by using **Analysis Filter** viewer, update the filter file, and review the analysis results.

Step 1: Open the Model

The example model `sldvexFilterObjectives` is a controller model that operates according to the controller algorithm described in “Detect Design Errors in Controller Model”.

```
open_system('sldvexControllerFilterObjectives');
```

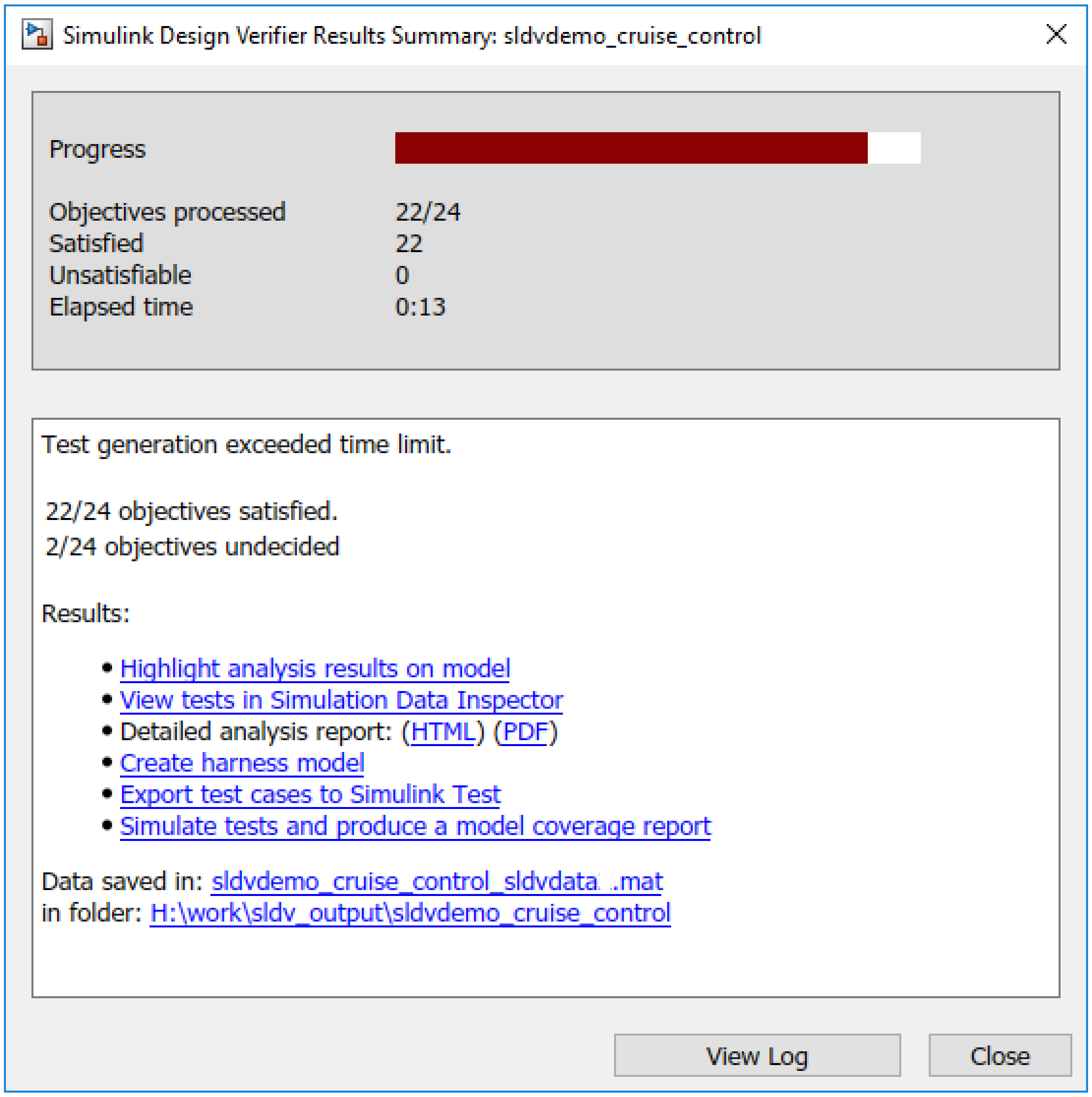


Step 2: Exclude a Model Object from Analysis

The model is preconfigured with the **Ignore objectives based on filter** option set to `On` and a coverage filter file specified by `sldvexControllerFilterObjectives_filter.cvf`. The coverage filter file consists of a rule that excludes the Abs block from the analysis. For more information on coverage filter file, see “Creating and Using Coverage Filters” (Simulink Coverage).

On the **Apps** tab, under **Model Verification, Validation, and Test**, click **Design Verifier**. Then, click **Detect Design Errors**. After the analysis completes, the Results Summary window reports that

5 objectives were processed, out of which, 3 were valid and 2 were falsified. The summary shows that 1 objective was excluded from analysis.


Simulink Design Verifier Results Summary: sldvdemo_cruise_control

Progress

Objectives processed 22/24
Satisfied 22
Unsatisfiable 0
Elapsed time 0:13

Test generation exceeded time limit.

22/24 objectives satisfied.
2/24 objectives undecided

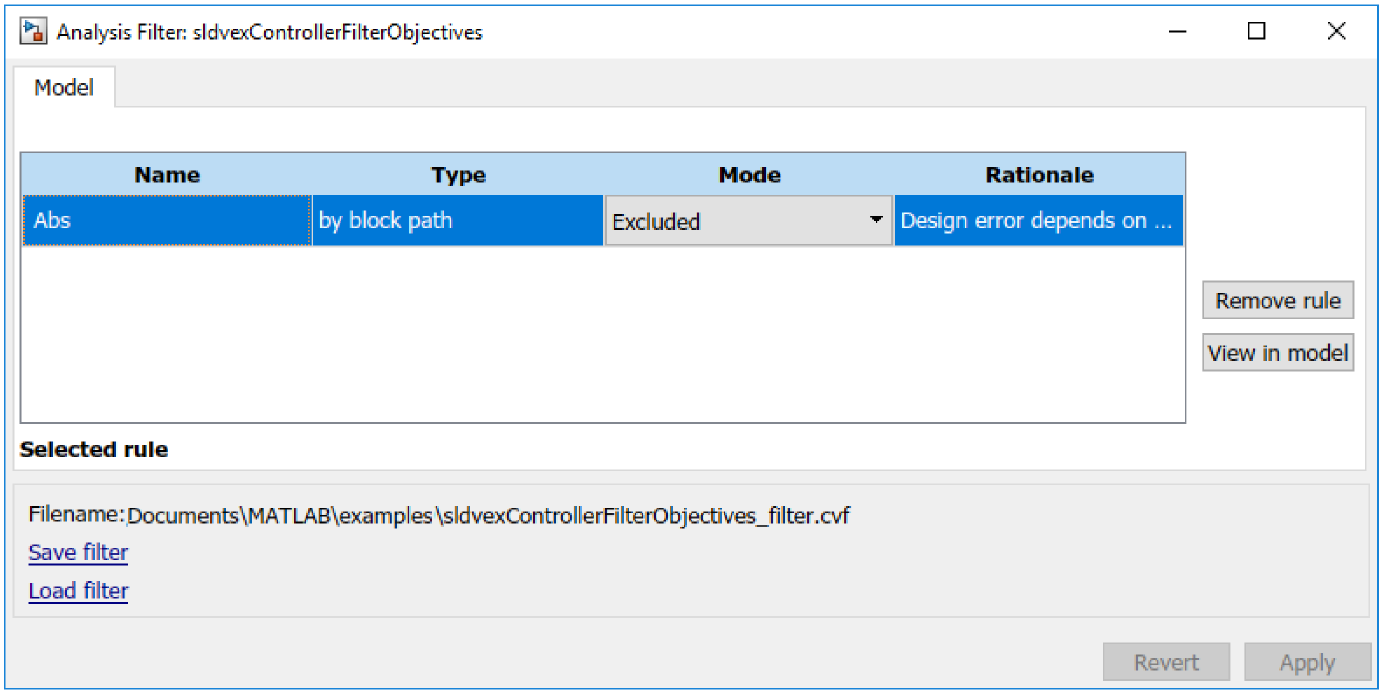
Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: sldvdemo_cruise_control_sldvdata .mat
in folder: H:\work\sldv_output\sldvdemo_cruise_control

Step 3: Open the Analysis Filter Viewer

On the Results Summary window, click **Open filter viewer**. The **Analysis Filter** viewer opens that displays the name, type, and rationale for the excluded objectives specified in the coverage filter file.



Name	Type	Mode	Rationale
Abs	by block path	Excluded	Design error depends on ...

Selected rule

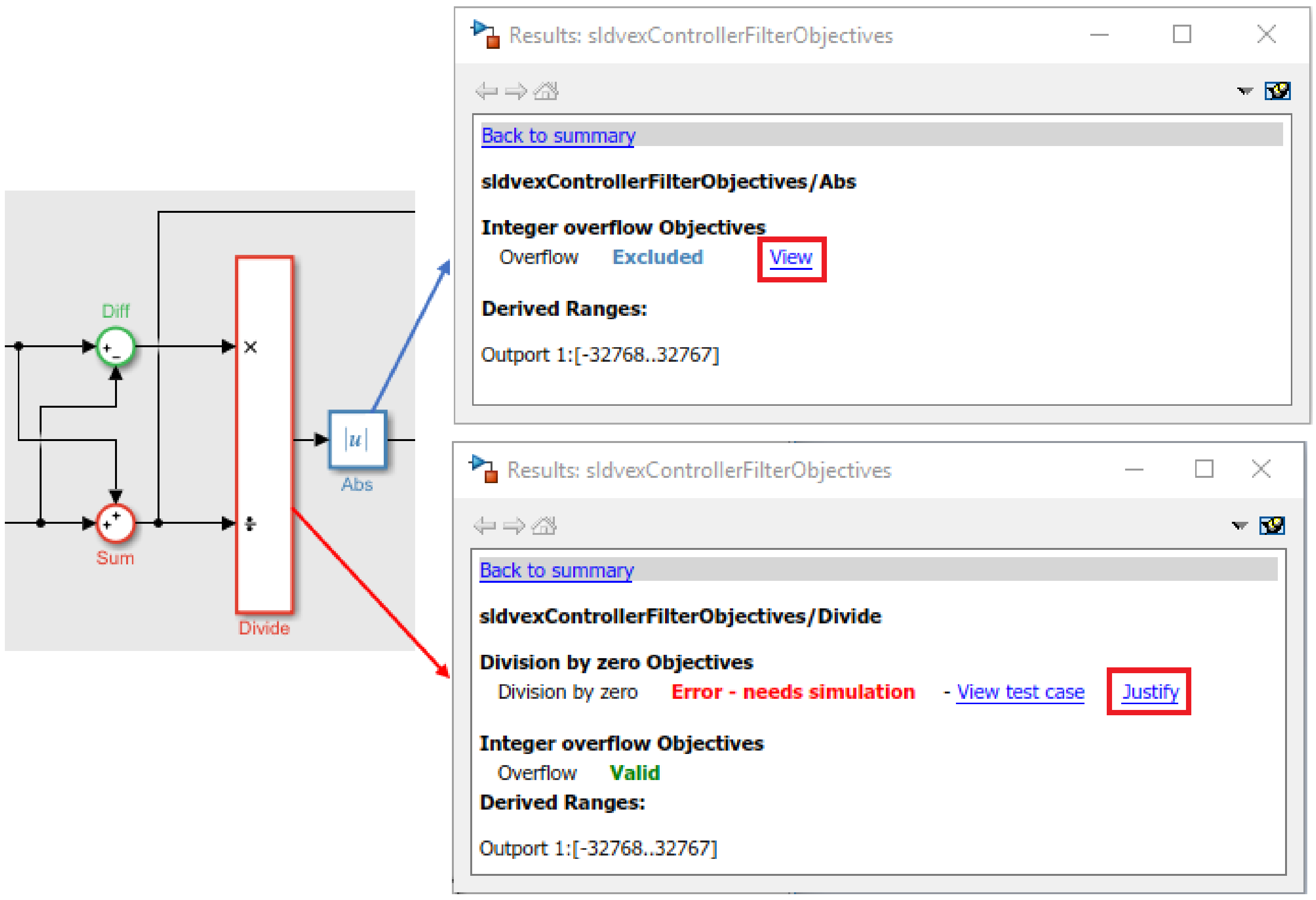
Filename: Documents\MATLAB\examples\sldvexControllerFilterObjectives_filter.cvf

Step 4: Justify Objectives

(a) On the Results Summary window, click **Highlight analysis results on model**. The model is highlighted with the analysis results. The excluded model objects are highlighted in steel blue and the model objects that result in errors are highlighted in red.

(b) To view the excluded objectives, click Abs block and click **View**. The **Analysis Filter** viewer opens.

(c) Click the Divide block. The Results Inspector window displays a summary of the objectives.



sldvexControllerFilterObjectives/Abs

Integer overflow Objectives
Overflow **Excluded** View

Derived Ranges:

Outport 1:[-32768..32767]

sldvexControllerFilterObjectives/Divide

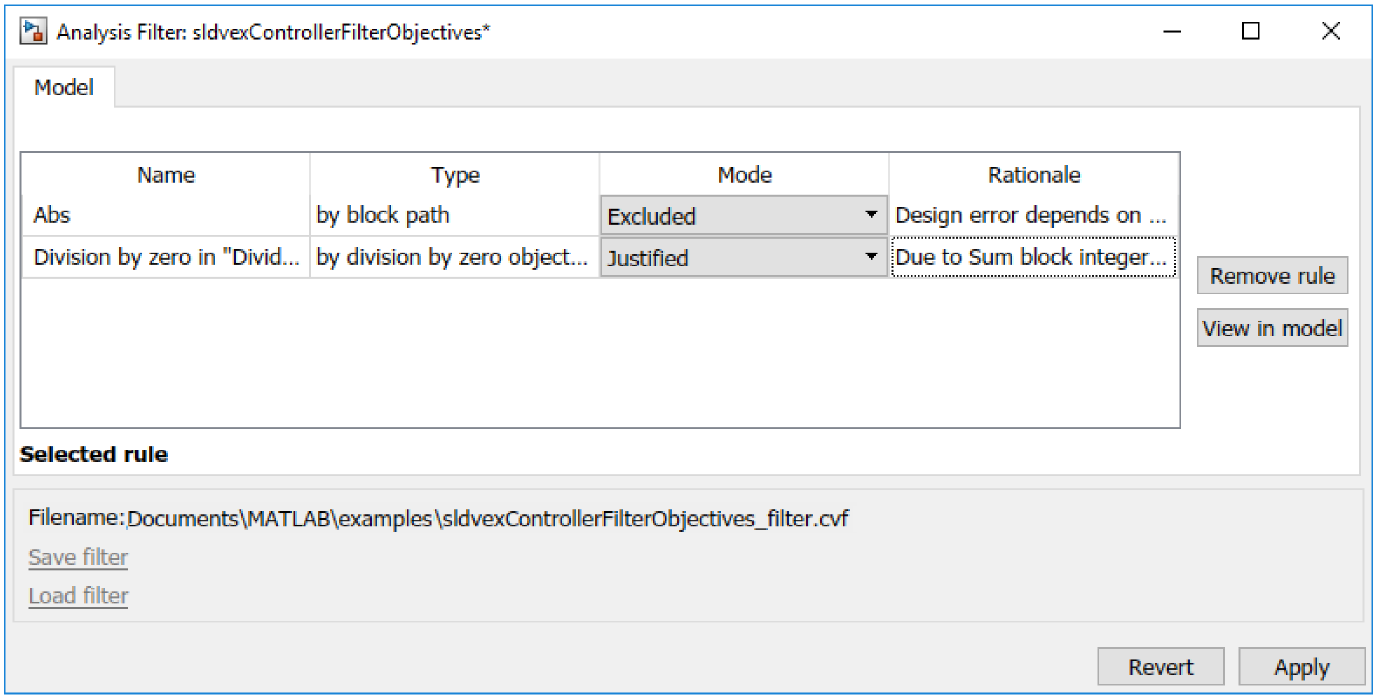
Division by zero Objectives
Division by zero **Error - needs simulation** - View test case Justify

Integer overflow Objectives
Overflow **Valid**

Derived Ranges:

Outport 1:[-32768..32767]

(d) To justify the division by zero objective, click **Justify**. The **Analysis Filter** viewer is updated with a rule that justifies this objective. Optionally, you can update the **Mode** or **Rationale** for the objectives.



Name	Type	Mode	Rationale
Abs	by block path	Excluded	Design error depends on ...
Division by zero in "Divid...	by division by zero object...	Justified	Due to Sum block integer...

Filename: Documents\MATLAB\examples\sldvexControllerFilterObjectives_filter.cvf

Step 5: Apply the Filter File and View Results

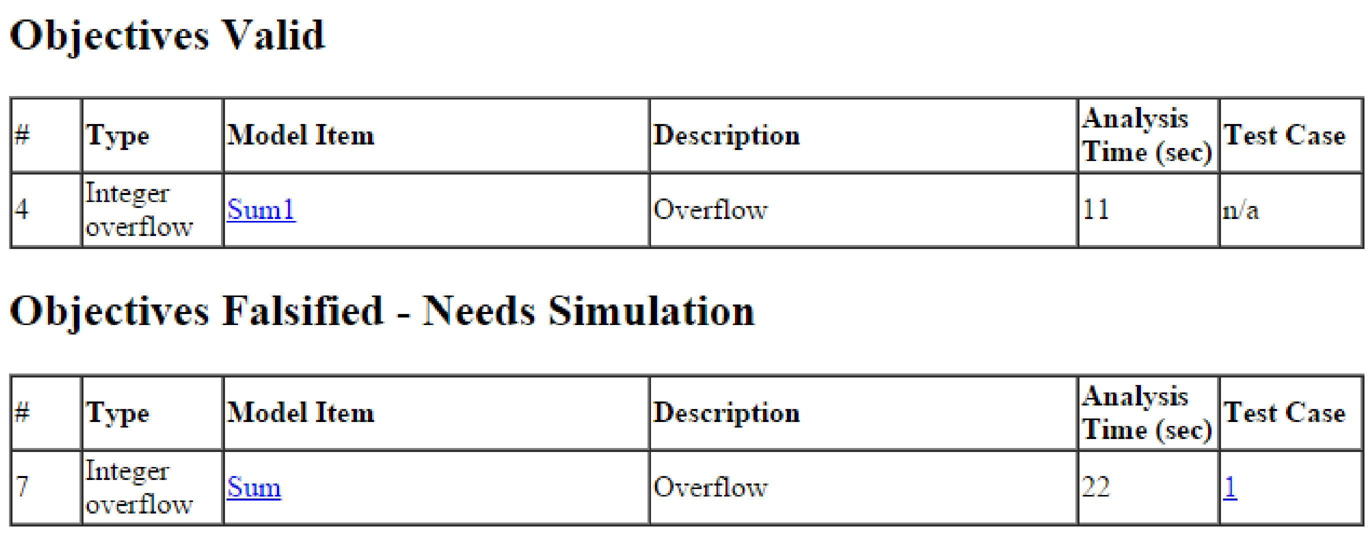
On the **Analysis Filter** viewer, click **Apply**. The model is highlighted with the updated filter. The Divide block is highlighted in green because all the objectives of the block are valid.

To save the updated filter file, in the **Analysis Filter** viewer, click **Save Filter**, enter the name of file, and click **OK**.

Note: After applying the filter, the highlighting of the model objects is as follows:

- If all the objectives of a block are excluded or justified, it is highlighted in steel blue.
- If a block has valid and excluded or justified objectives, it is highlighted in green.
- If a block has falsified and excluded or justified objectives, it is highlighted in red.

For a detailed analysis report, in the Results Summary window, click **HTML** or **PDF**. The Design Error Detection Objectives Status chapter reports the excluded and justified objectives along with the valid and falsified objectives.



Objectives Valid

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
4	Integer overflow	Sum1	Overflow	11	n/a

Objectives Falsified - Needs Simulation

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
7	Integer overflow	Sum	Overflow	22	1

Related Topics

- "Filter Objectives by Using Analysis Filter Viewer" on page 6-47
- "Detect Integer Overflow and Division-by-Zero Errors" on page 6-25

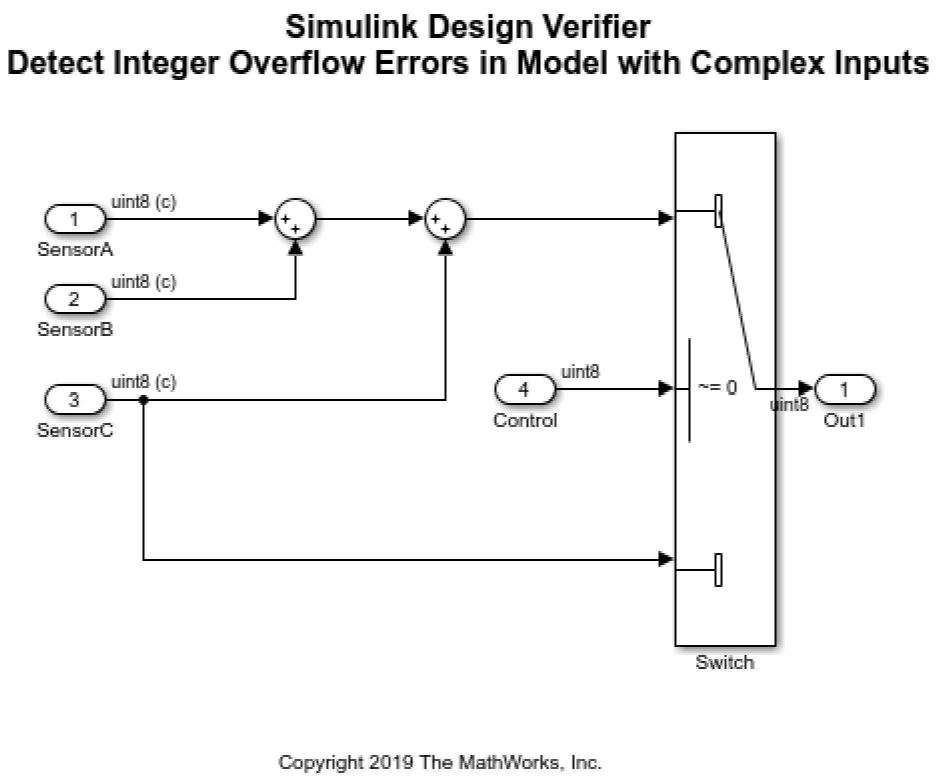
Detect Integer Overflow in a Model with Complex Inputs

This example shows how to detect integer overflow errors in a model that consists of complex type inputs.

Step 1: Open the Model

The `sldvexComplexInputs` model contains SensorA, SensorB, and SensorC complex inputs and a Control input. The SensorA and SensorB inports are constraint to **Maximum output value** equal to `100`.

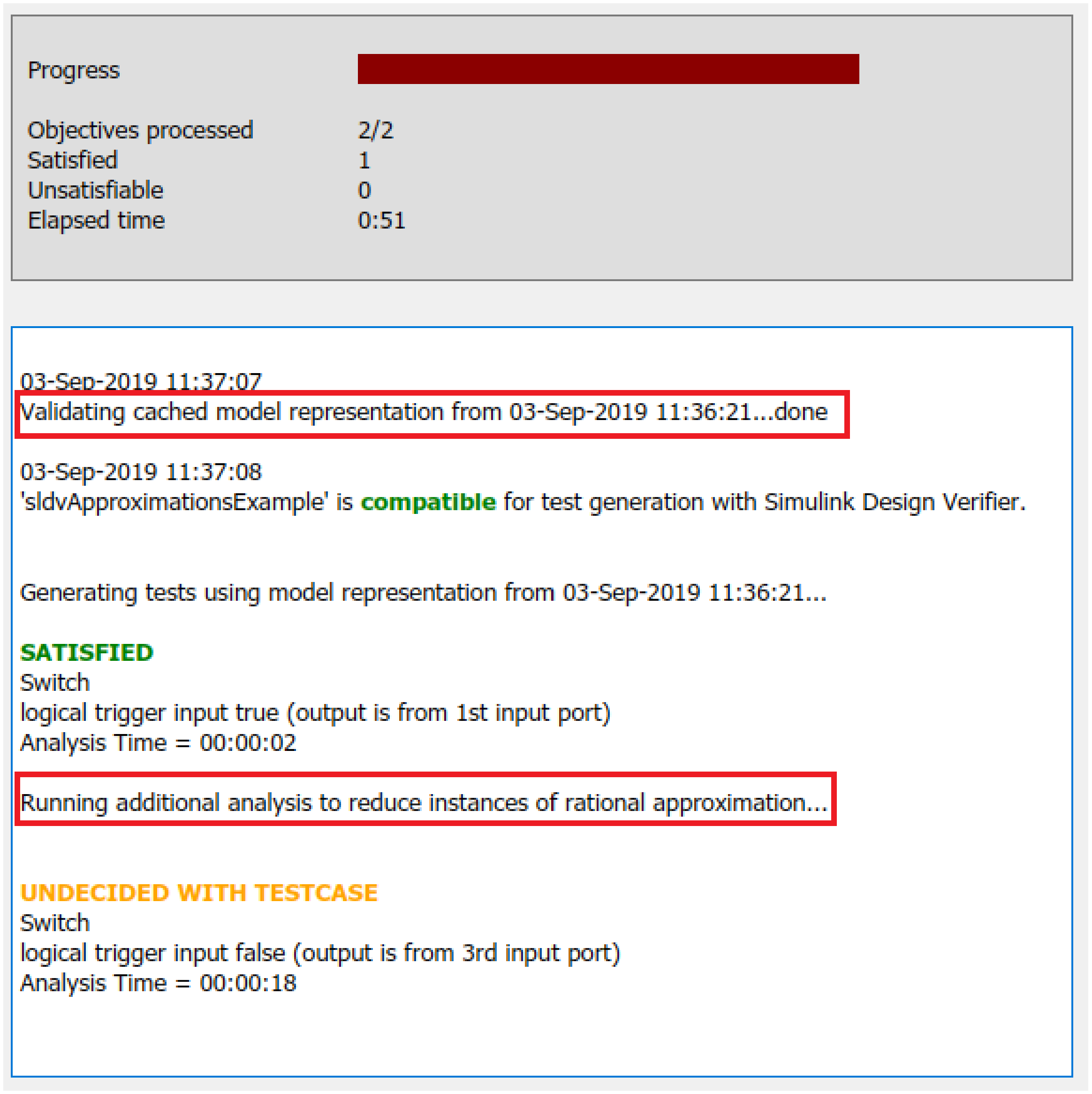
```
open_system('sldvexComplexInputs');
```



Step2: Perform Design Error Detection Analysis

On the **Apps** tab, in the **Model Verification, Validation, and Test** group, select **Design Verifier**.

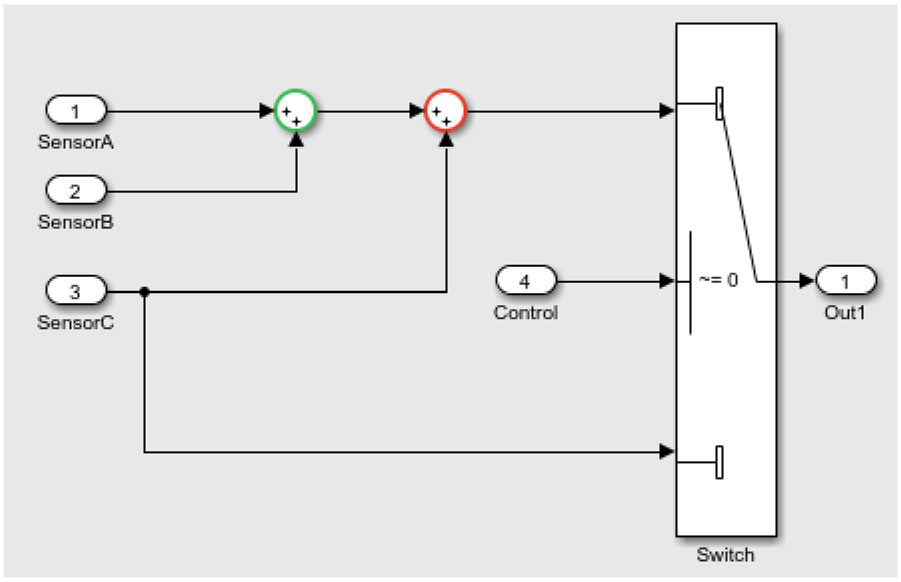
To detect design errors, click **Detect Design Errors**. After the analysis completes, the Results Summary window displays that one objective is valid and one objective is falsified.



```
Progress

Objectives processed    2/2
Satisfied               1
Unsatisfiable           0
Elapsed time            0:51

03-Sep-2019 11:37:07
Validating cached model representation from 03-Sep-2019 11:36:21...done

03-Sep-2019 11:37:08
'sldvApproximationsExample' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 03-Sep-2019 11:36:21...

SATISFIED
Switch
logical trigger input true (output is from 1st input port)
Analysis Time = 00:00:02

Running additional analysis to reduce instances of rational approximation...

UNDECIDED WITH TESTCASE
Switch
logical trigger input false (output is from 3rd input port)
Analysis Time = 00:00:18
```

Step 3: Review Analysis Results

In the Results Summary window, click **Highlight analysis results on model**. The Sum block whose output results in integer overflow error is highlighted in red.



To view the analysis report, click **HTML** or **PDF** in the Results Summary window. The Design Error Detection Objectives Status chapter lists the description of the valid and falsified objectives.

Objectives Valid

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
4	Integer overflow	Sum1	Overflow	11	n/a

Objectives Falsified - Needs Simulation

#	Type	Model Item	Description	Analysis Time (sec)	Test Case
7	Integer overflow	Sum	Overflow	22	1

The Design Errors chapter contains the test case inputs that results in integer overflow.

Time	**0**
Step	**1**
SensorA	16+93i
SensorB	26+78i
SensorC	94+93i
Control	1

See also

- “Detect Integer Overflow Errors” on page 6-50
- “Understand the Analysis Results” on page 6-4

Analyzing the Results for a Dead Logic Analysis

This example demonstrates how to isolate potential causes of dead logic using `sldvexCommonCausesOfDeadLogic` example model. Dead logic detection finds the unreachable objectives in the model that cause the model element to remain inactive.

Workflow

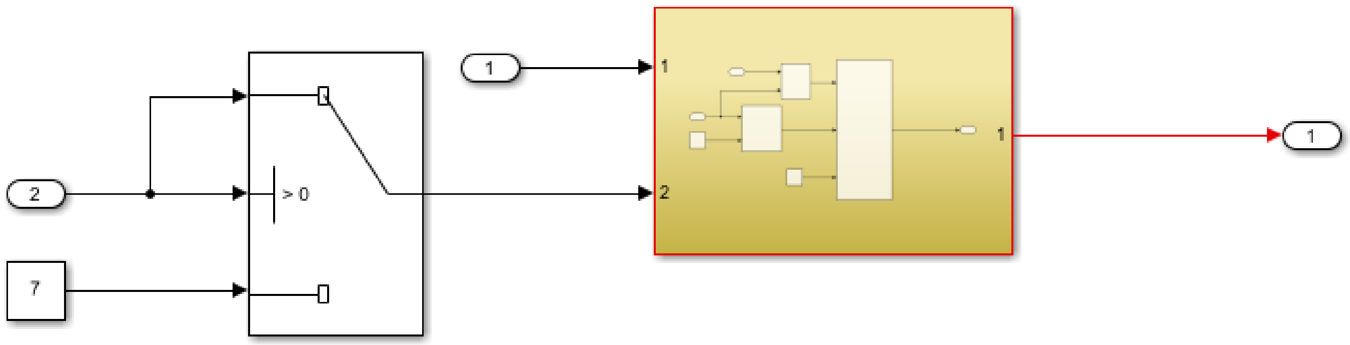
This example model `sldvexCommonCausesOfDeadLogic` explains few common patterns that often leads to dead logic in a model. These patterns are explained by five subsystems contained in `sldvexCommonCausesOfDeadLogic` model.

Section 1 : Run a Dead Logic

Follow these steps to run the dead logic analysis:

Step 1: Open the `sldvexCommonCausesOfDeadLogic`.

```
open_system('sldvexCommonCausesOfDeadLogic');
```



Step 2: On the Design Verifier tab, in the **Mode** section, select **Design Error Detection** .

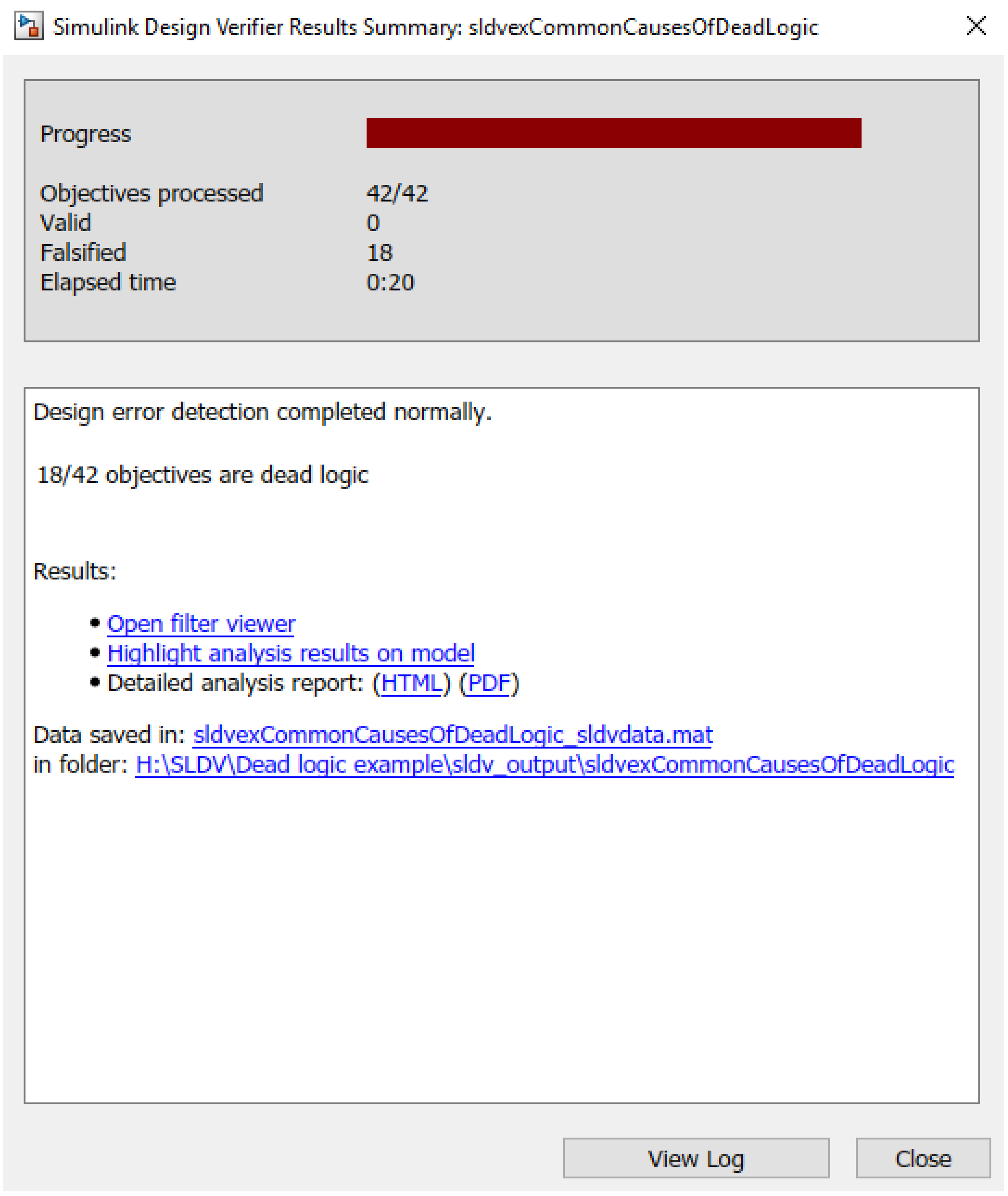
Step 3: Click **Error Detection Settings** .

Step 4: In the Configuration Parameters dialog box, on the **Design Verifier** > **Design Error Detection** pane, enable **Dead logic** option and clear | Identify active logic| option.

Step 5: Click **Detect Design Errors**.

Section 2: Analyze and Review the Results

The software analyzes the model for dead logic and displays the results in the Results Summary window. The result indicates that eighteen of the 42 objectives are dead logic.


Simulink Design Verifier Results Summary: sldvexCommonCausesOfDeadLogic
View Log
Close

Progress	
Objectives processed	42/42
Valid	0
Falsified	18
Elapsed time	0:20

Design error detection completed normally.

18/42 objectives are dead logic

Results:

- Open filter viewer
- Highlight analysis results on model
- Detailed analysis report: (HTML) (PDF)

Data saved in: sldvexCommonCausesOfDeadLogic_sldvdata.mat
in folder: H:\SLDV\Dead logic example\sldv_output\sldvexCommonCausesOfDeadLogic

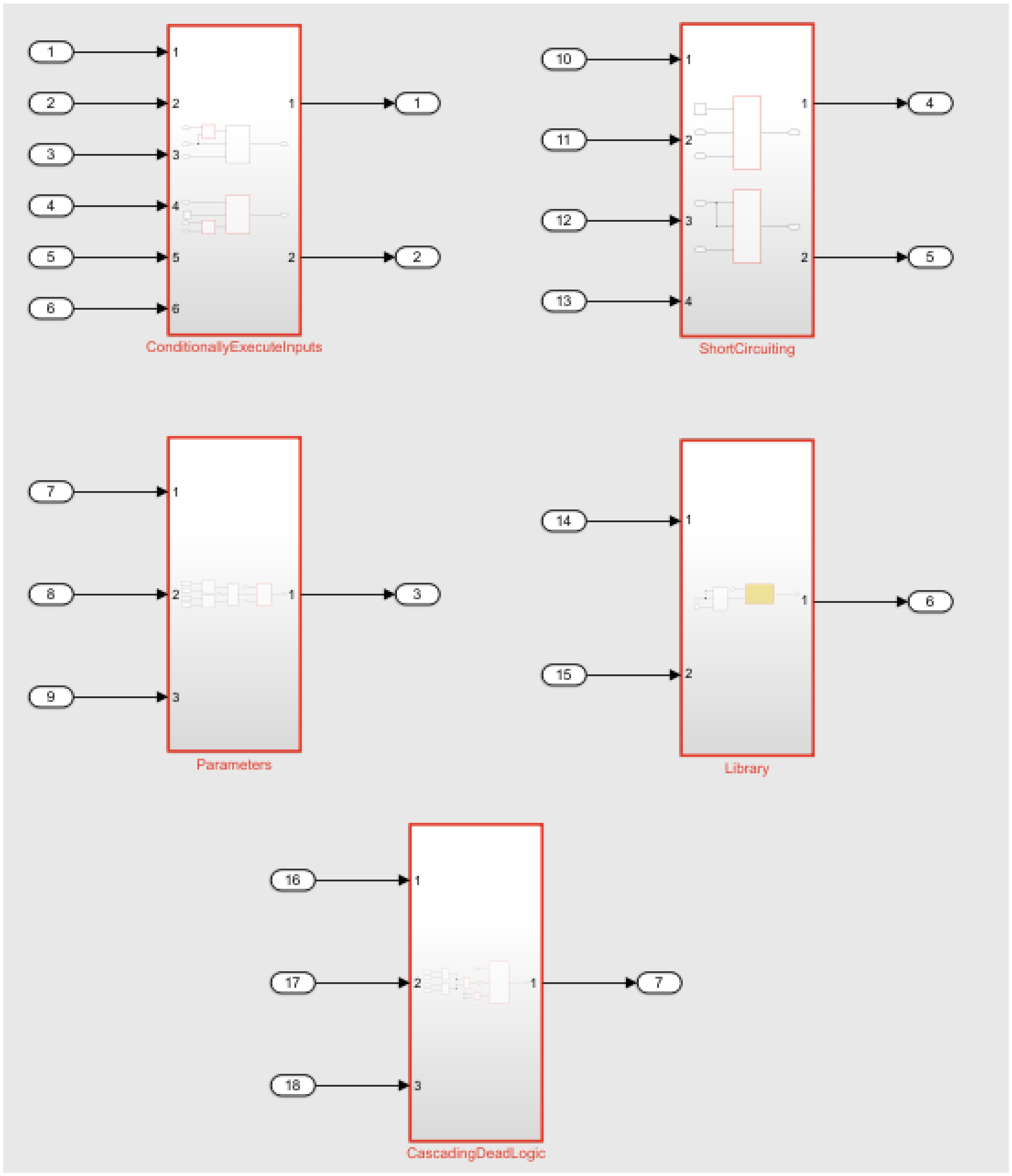
Section 3: Highlight Analysis Results in the Subsystem Blocks

This section explains the common patterns that leads to dead logic in `sldvexCommonCausesOfDeadLogic` model. These patterns can be categorized into:

1. Conditional Execution of a subsystem
2. Short-circuiting of a Logical Operator Block During Analysis
3. Parameter Values Treated as Constants
4. Library-linked Blocks
5. Upstream Blocks

Click **Highlight analysis results on model** in the Results Summary Window. The subsystems with dead logic are highlighted in red. These are:

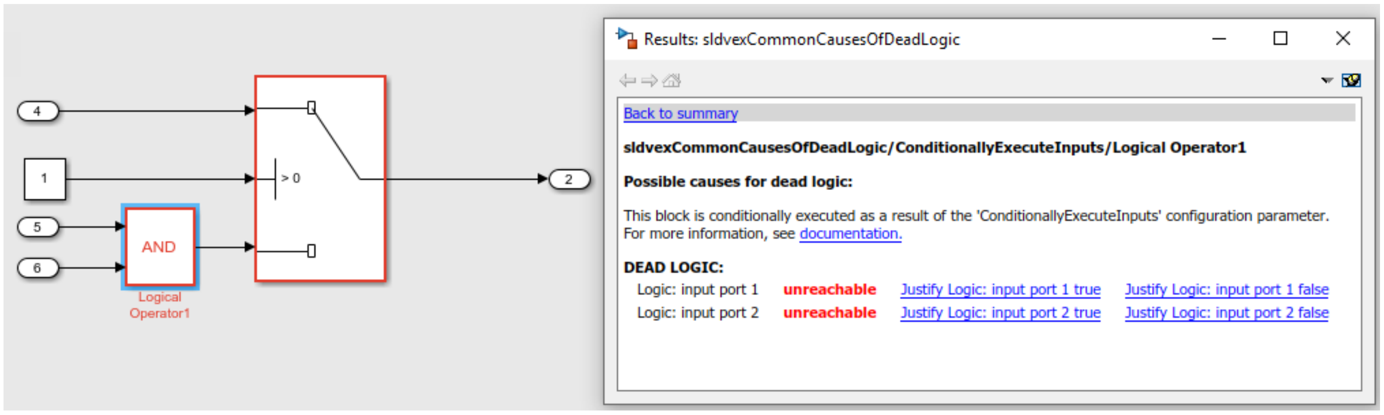
1. **ConditionallyExecuteInputs**
2. **ShortCircuiting**
3. **Parameters**
4. **Library**
5. **CascadingDeadLogic**



The subsystems in `sldvexCommonCausesOfDeadLogic` model explain these patterns. Each subsystem block has a dead logic red. Consider each subsystem one by one to analyze and highlight the results.

1. ConditionallyExecuteInputs Subsystem

If your model consists of Switch or Multiport Switch blocks and the `Conditional input branch execution` parameter is set to `On`, the conditional execution can often cause unexpected dead logic. Open the **ConditionallyExecuteInputs** subsystem, and click the AND block highlighted in red. The Result Inspector displays the summary of the dead logic.



sldvexCommonCausesOfDeadLogic/ConditionallyExecuteInputs/Logical Operator1

Possible causes for dead logic:

This block is conditionally executed as a result of the 'ConditionallyExecuteInputs' configuration parameter. For more information, see documentation.

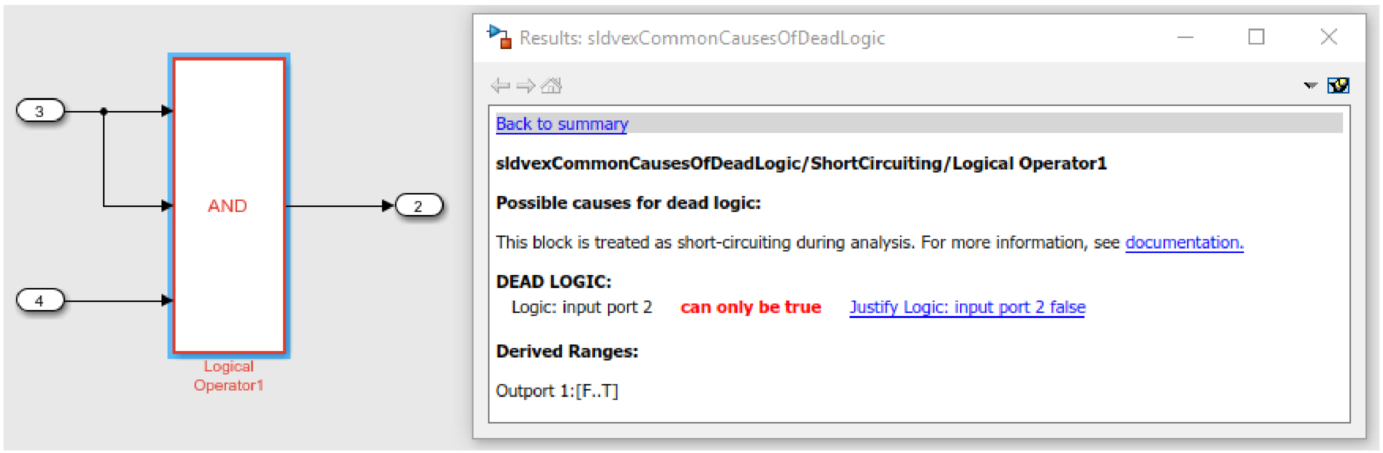
DEAD LOGIC:

Logic: input port 1	**unreachable**	Justify Logic: input port 1 true	Justify Logic: input port 1 false
Logic: input port 2	**unreachable**	Justify Logic: input port 2 true	Justify Logic: input port 2 false

In this subsystem, `Conditional input branch execution` parameter is set to `On`. The AND Logical Operator block is conditionally executed, which causes the dead logic for the subsystem.

2. ShortCircuiting Subsystem

Simulink Design Verifier treats logic blocks as if they are short-circuiting when analyzing for dead logic. Open the **ShortCircuiting** subsystem, and click the AND block highlighted in red. The Result Inspector displays the summary of the dead logic.



sldvexCommonCausesOfDeadLogic/ShortCircuiting/Logical Operator1

Possible causes for dead logic:

This block is treated as short-circuiting during analysis. For more information, see documentation.

DEAD LOGIC:

Logic: input port 2 **can only be true** Justify Logic: input port 2 false

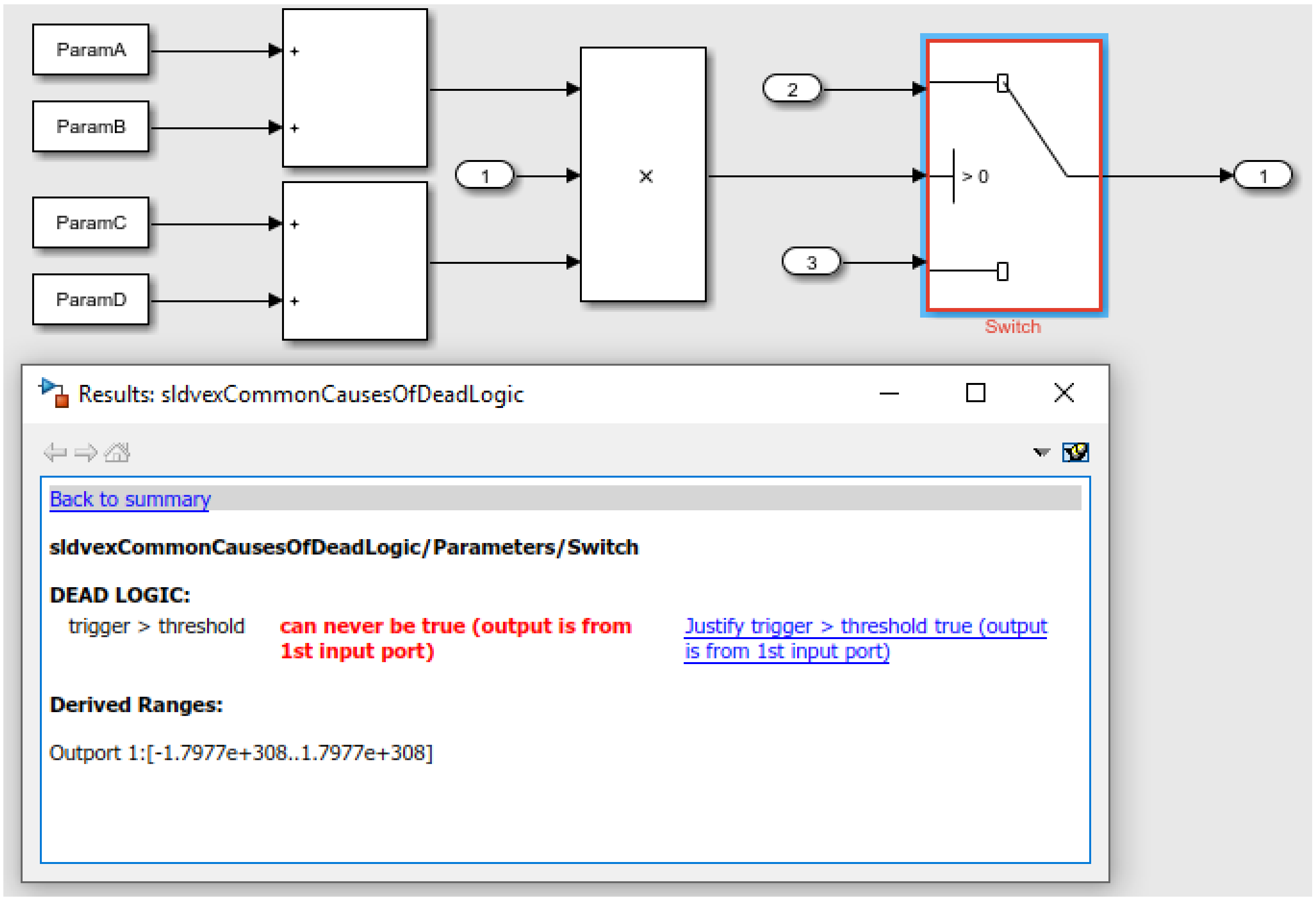
Derived Ranges:

Outport 1:[F..T]

In this model, if `In3` is false, the software ignores the third input due to the short-circuiting. This is suggested as a potential explanation for the dead logic in the Results Inspector window.

3. Parameters Subsystem

If your model contains parameters, Simulink Design Verifier treats the values as constants by default. This might cause dead logic in the model. Open the **ShortCircuiting** subsystem, and click the `Switch` block highlighted in red. The Result Inspector displays the summary of the dead logic.



Back to summary

sldvexCommonCausesOfDeadLogic/Parameters/Switch

DEAD LOGIC:

trigger > threshold	**can never be true (output is from 1st input port)**	Justify trigger > threshold true (output is from 1st input port)

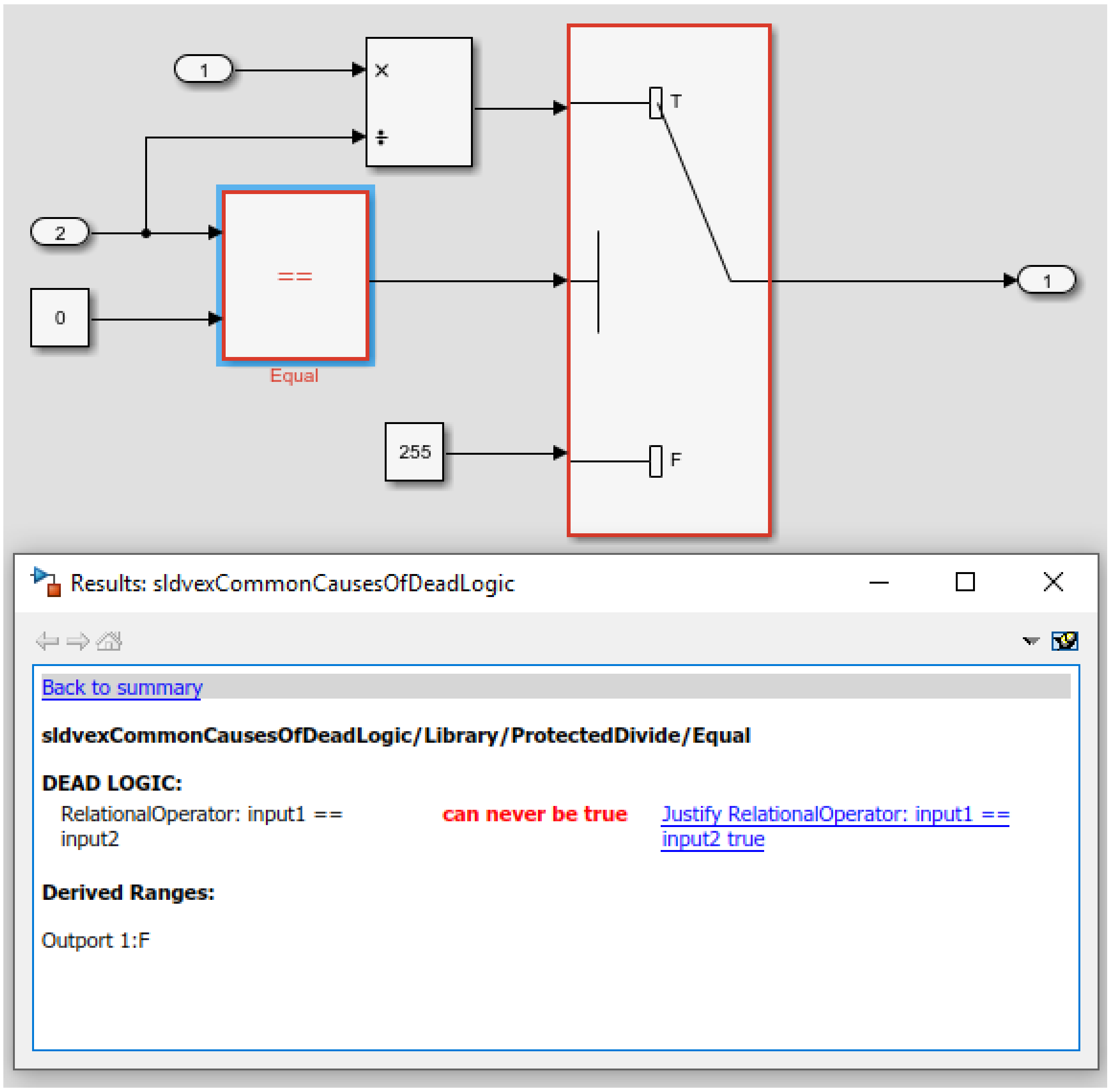
Derived Ranges:

Outport 1:[-1.7977e+308..1.7977e+308]

Here all of the parameters are set to zero. This causes the dead logic for the Less Than block.

4. Library Subsystem

Library blocks may be written with defensive conditions that are redundant in some locations where they are used. In some cases, this may cause dead logic. For example, the **ProtectedDivide** library subsystem has protection for division by zero. In this case, the inputs to the **ProtectedDivide library** subsystem can never experience a division-by-zero. This causes the guarding logic to be dead. Open the **Library** block, and click the **ProtectedDivide** subsystem highlighted in red. The `equal` block shows the dead logic. The Result Inspector displays the summary of the dead logic.



Results: sldvexCommonCausesOfDeadLogic

Back to summary

sldvexCommonCausesOfDeadLogic/Library/ProtectedDivide/Equal

DEAD LOGIC:

RelationalOperator: input1 == input2	**can never be true**	Justify RelationalOperator: input1 == input2 true

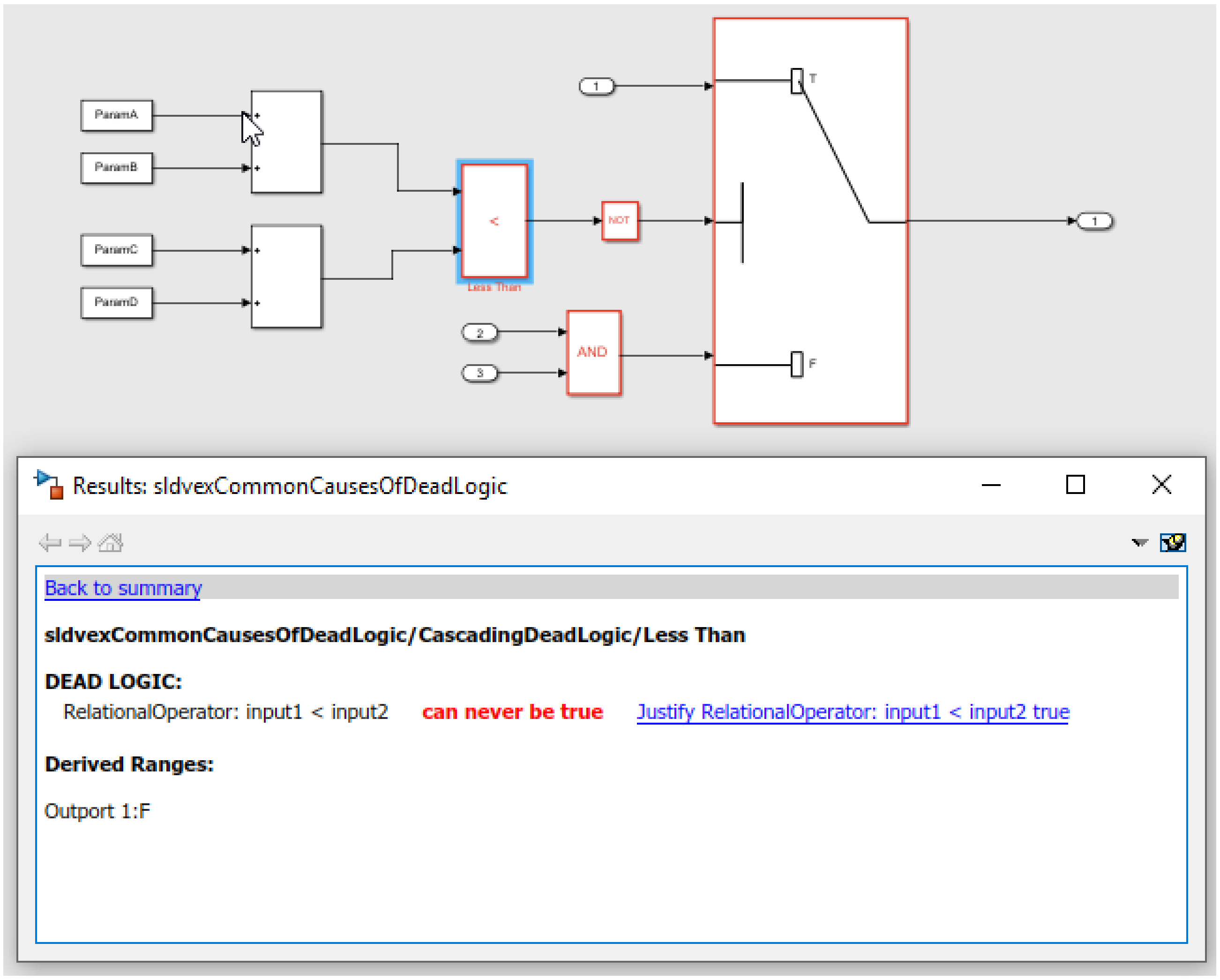
Derived Ranges:

Outport 1:F

You may consider justifying the dead logic that arises from those library blocks.

5.CascadingDeadLogic Subsystem

When a particular block has dead logic, this often leads to a cascade effect that causes downstream blocks to have dead logic. Open the **CascadingDeadLogic** subsystem, and click the `Less Than` block highlighted in red. The Result Inspector displays the summary of the dead logic.



Results: sldvexCommonCausesOfDeadLogic

Back to summary

sldvexCommonCausesOfDeadLogic/CascadingDeadLogic/Less Than

DEAD LOGIC:

RelationalOperator: input1 < input2 **can never be true** Justify RelationalOperator: input1 < input2 true

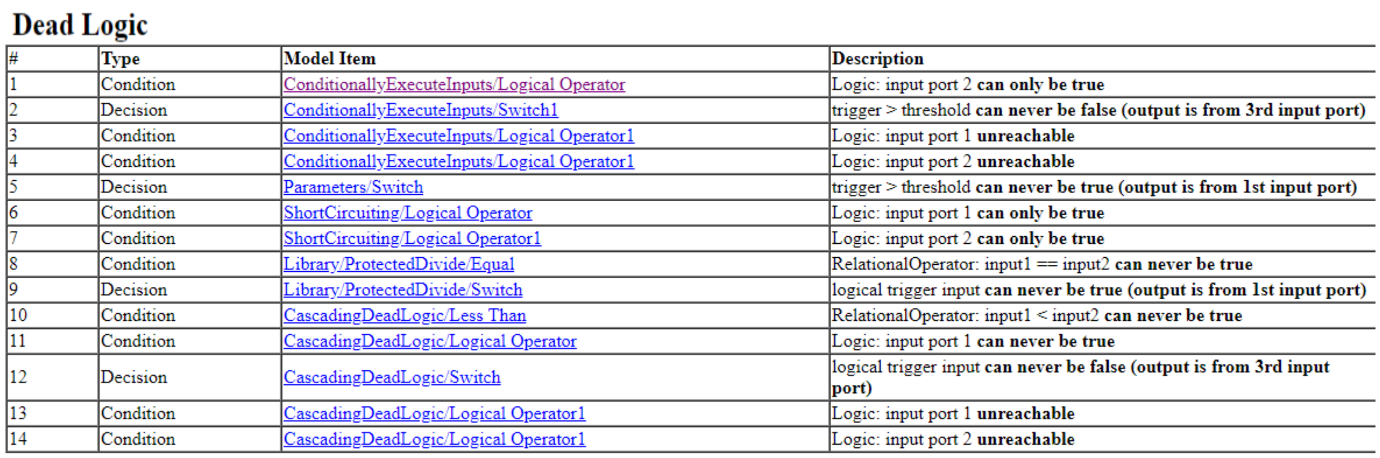
Derived Ranges:

Outport 1:F

The dead logic in the Less Than block causes the dead logic in the corresponding downstream blocks. It is therefore often helpful to review the upstream dead logic before reviewing any downstream dead logic.

Section 4: View the Analysis Report

To view the detailed analysis report, in the Results Summary window, click **HTML**. The report displays the summary of all the results that are dead logic in the model.



Dead Logic

#	Type	Model Item	Description
1	Condition	ConditionallyExecuteInputs/Logical Operator	Logic: input port 2 **can only be true**
2	Decision	ConditionallyExecuteInputs/Switch1	trigger > threshold **can never be false (output is from 3rd input port)**
3	Condition	ConditionallyExecuteInputs/Logical Operator1	Logic: input port 1 **unreachable**
4	Condition	ConditionallyExecuteInputs/Logical Operator1	Logic: input port 2 **unreachable**
5	Decision	Parameters/Switch	trigger > threshold **can never be true (output is from 1st input port)**
6	Condition	ShortCircuiting/Logical Operator	Logic: input port 1 **can only be true**
7	Condition	ShortCircuiting/Logical Operator1	Logic: input port 2 **can only be true**
8	Condition	Library/ProtectedDivide/Equal	RelationalOperator: input1 == input2 **can never be true**
9	Decision	Library/ProtectedDivide/Switch	logical trigger input **can never be true (output is from 1st input port)**
10	Condition	CascadingDeadLogic/Less Than	RelationalOperator: input1 < input2 **can never be true**
11	Condition	CascadingDeadLogic/Logical Operator	Logic: input port 1 **can never be true**
12	Decision	CascadingDeadLogic/Switch	logical trigger input **can never be false (output is from 3rd input port)**
13	Condition	CascadingDeadLogic/Logical Operator1	Logic: input port 1 **unreachable**
14	Condition	CascadingDeadLogic/Logical Operator1	Logic: input port 2 **unreachable**

The software stores the detailed analysis results in the `DeadLogic field` in the Simulink Design Verifier Data Files. You can use the data file for further analysis of the results.

Related Topics

- “Common Causes for Dead Logic” on page 6-22

Generating Test Cases

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 7-4
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 7-17
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 7-11
5	Specify options that control how Simulink Design Verifier generates test cases for your model.	"Configure Test Generation Options" on page 7-5
6	Execute the Simulink Design Verifier analysis.	"Analyze the Example Model" on page 7-5 and "Reanalyze the Example Model" on page 7-13
7	Review the analysis results.	"Review Analysis Results" on page 7-5

See Also

More About

- "Flip Flop Test Generation" on page 7-63
- "Cruise Control Test Generation" on page 7-67
- "Fuel Rate Controller Logic" on page 7-68

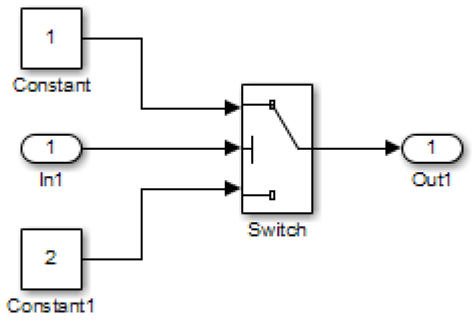
Generate Test Cases for Model Decision Coverage

In this section...

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 On the **Apps** tab, click the arrow on the right of the **Apps** section.

   Under **Model Verification, Validation, and Test**, click **Design Verifier**.
6 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
7 In the Configuration Parameters dialog box, select **Solver** pane. In the **Solver selection**:

- 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.

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

9 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.

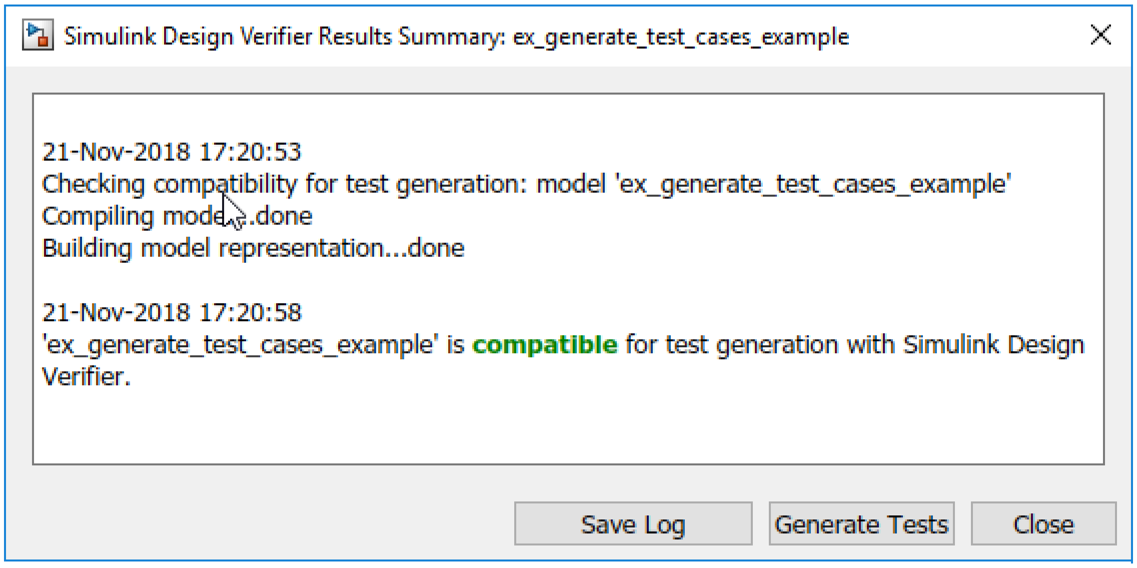
Before you start the analysis, you can also make sure that your model is compatible with Simulink Design Verifier software:

1 Open the `ex_generate_test_cases_example` model.

2 On the **Design Verifier** tab, click **Check Compatibility**.

The software displays the log window, which states whether or not your model is compatible for analysis.

The model you just created is compatible.



Simulink Design Verifier Results Summary: ex_generate_test_cases_example

```
21-Nov-2018 17:20:53
Checking compatibility for test generation: model 'ex_generate_test_cases_example'
Compiling model...done
Building model representation...done

21-Nov-2018 17:20:58
'ex_generate_test_cases_example' is compatible for test generation with Simulink Design
Verifier.
```

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. On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**.
3. Click **Test Generation Settings**.
4. In the Configuration Parameters dialog box, 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 `Auto`. If you want to generate fewer but longer test cases, select `LongTestcases` for the **Test suite optimization** parameter.
5. Click **OK** to save your changes and close the Configuration Parameters dialog box.
6. Save the `ex_generate_test_cases_example` model.

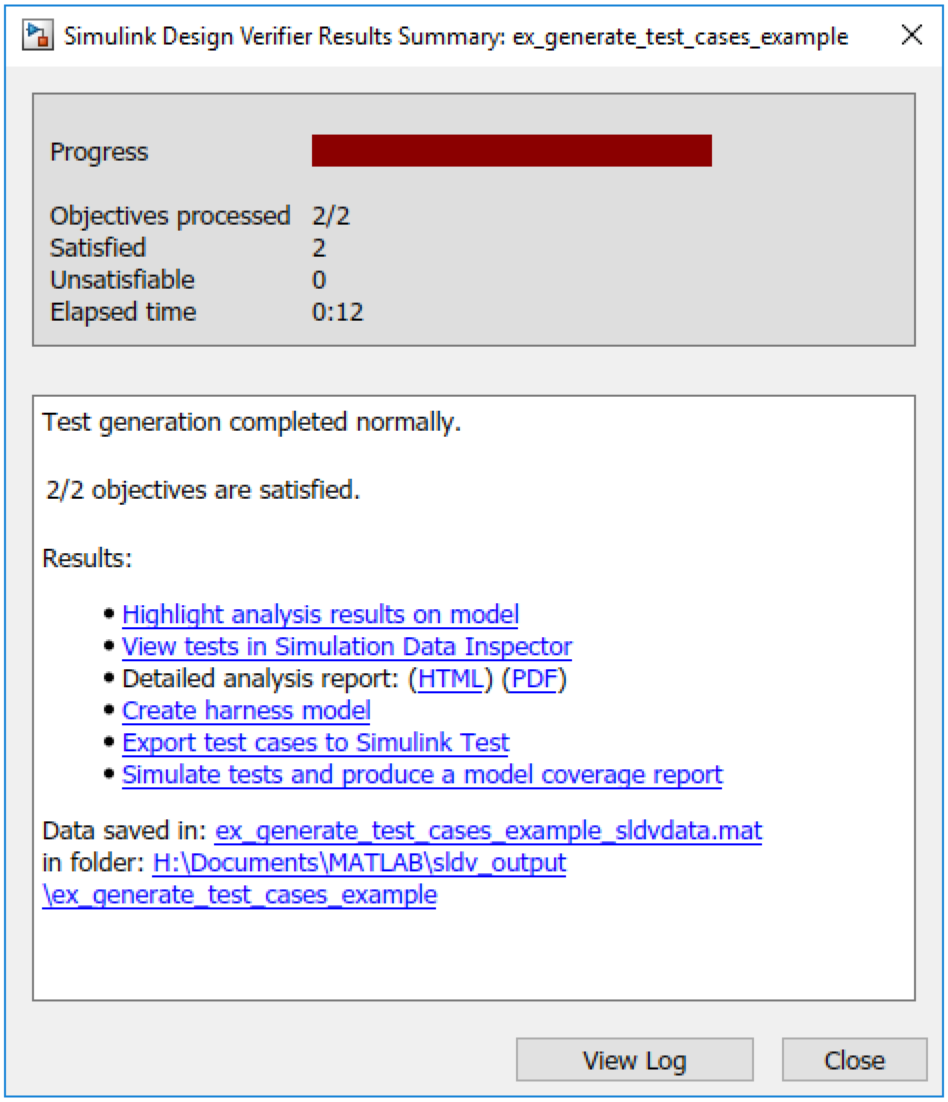
Analyze the Example Model

On the **Design Verifier** tab, click **Generate Tests**. The Simulink Design Verifier analyzes your model to generate test cases.

During the analysis, the Results Summary 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 Results Summary window displays these options for reviewing the results.


Simulink Design Verifier Results Summary: ex_generate_test_cases_example

Progress

Objectives processed 2/2
Satisfied 2
Unsatisfiable 0
Elapsed time 0:12

Test generation completed normally.

2/2 objectives are satisfied.

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: ex_generate_test_cases_example_sldvdata.mat
in folder: H:\Documents\MATLAB\sldv_output
\ex_generate_test_cases_example

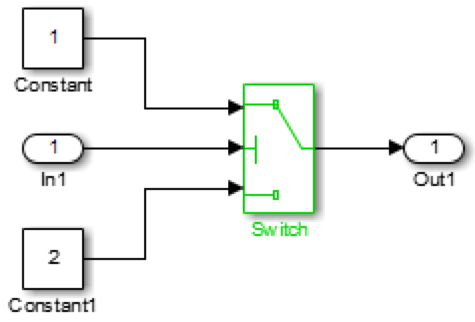
The following sections describe how you can review the analysis results:

- "Review Analysis Results on the Model" on page 7-6
- "Review Detailed Analysis Report" on page 7-8
- "Review Harness Model" on page 7-9
- "Simulate Tests and Produce a Model Coverage Report" on page 7-9
- "View sldvData File" on page 7-11
- "Review Analysis Results in the Results Summary Window" on page 7-11

Review Analysis Results on the Model

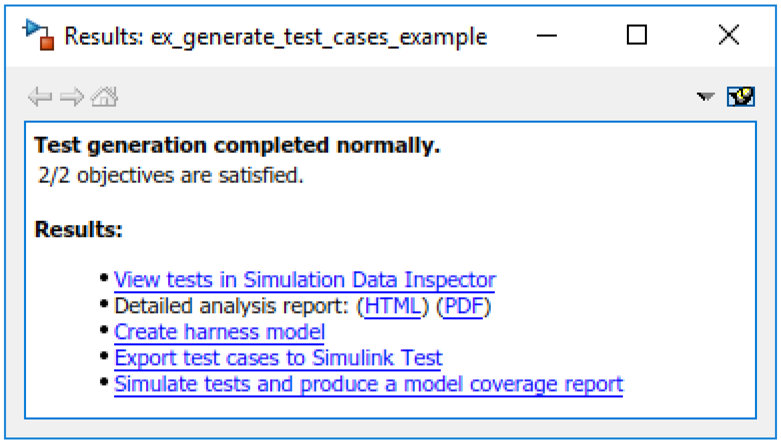
Highlight the analysis results on the example model:

1 In the Results Summary window for the `ex_generate_test_cases_example` analysis, click **Highlight analysis results on model**.



The Switch block is highlighted 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 and clear **Always on top**.


Results: ex_generate_test_cases_example

Test generation completed normally.
2/2 objectives are satisfied.

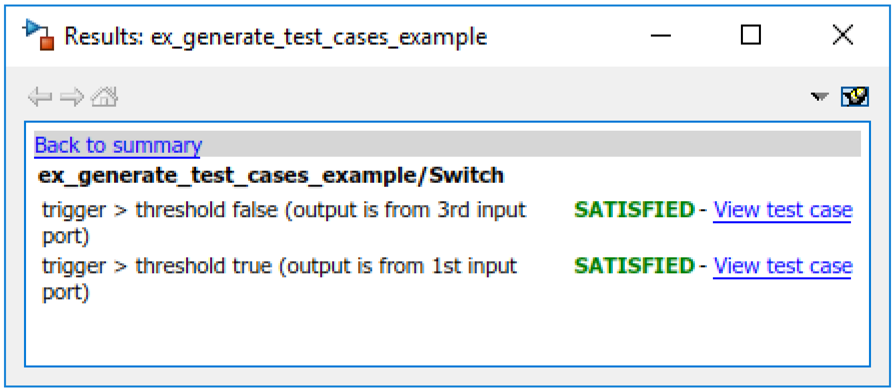
Results:

- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

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`


Results: ex_generate_test_cases_example

Back to summary

ex_generate_test_cases_example/Switch

trigger > threshold false (output is from 3rd input port)	SATISFIED - View test case
trigger > threshold true (output is from 1st input port)	SATISFIED - View test case

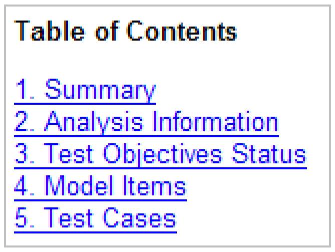
For more information about highlighted analysis results on a model, see “Highlighted Results on the Model” on page 13-2.

Review Detailed Analysis Report

Create a detailed HTML analysis report:

1. In the Simulink Design Verifier Results Summary window, in Detailed analysis report, click **HTML**.

   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.

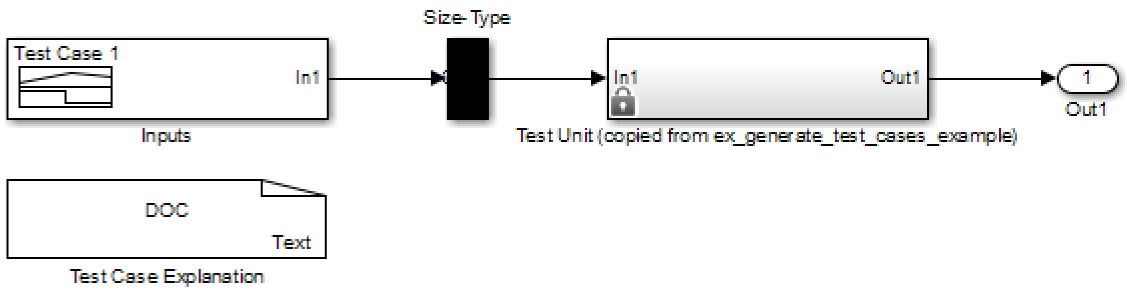
This section provides details about a test case that the analysis generated to achieve an objective in your model. This test case achieves test objective 1, when the Switch block passes its third input to its output port. Specifically, the software determines that a value of –1 for the Switch block control 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 13-28.

Review Harness Model

To create a harness model with test cases that satisfy the test objectives in your model, in the Simulink Design Verifier Results Summary 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 7-9.

For more information about the harness model, see “Simulink Design Verifier Harness Models” on page 13-13.

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 generate fewer test cases:

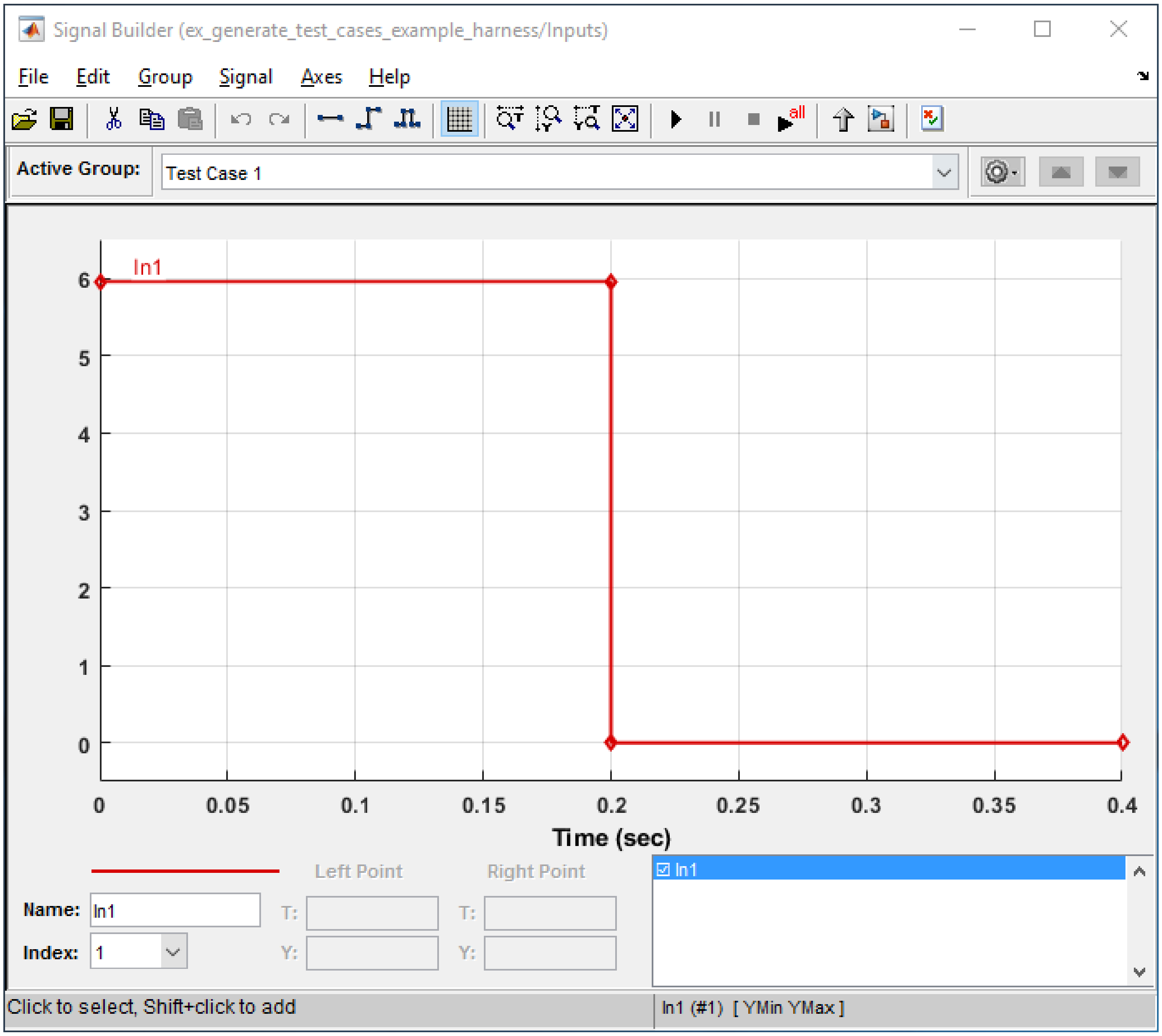
1 Set the **Test suite optimization** parameter to `LongTestcases`.
2 Rerun the analysis.

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.


Signal Builder (ex_generate_test_cases_example_harness/Inputs)
File Edit Group Signal Axes Help
Active Group: Test Case 1
In1
Time (sec)
Left Point
Right Point
Name: In1
Index: 1
☑ In1
Click to select, Shift+click to add
In1 (#1) [YMin YMax]

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” (Simulink Coverage).

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 13-7.

Review Analysis Results in the Results Summary Window

As long as your model remains open, you can view the results of your most recent Simulink Design Verifier analysis in the Results Summary window.

On the **Design Verifier** tab, in the **Review Results** section, click **Load Earlier Results** or **Results Summary** to view the results.

For any Simulink Design Verifier analysis, from the Results Summary window, you can perform these 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 13-28
Create the harness model, or if the harness model already exists, open it. If no test cases were generated during the analysis, this option is not available.	“Simulink Design Verifier Harness Models” on page 13-13
View the data file.	“Simulink Design Verifier Data Files” on page 13-7
View the log file.	“Simulink Design Verifier Log Files” on page 13-46

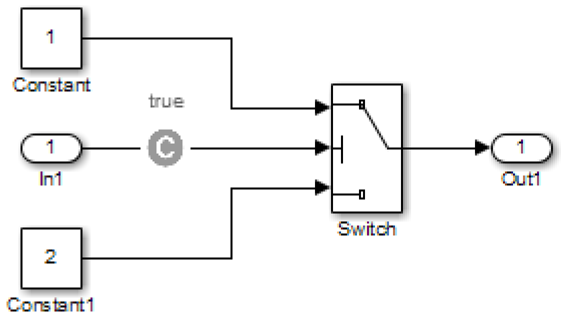
After you close your model, you can no longer view analysis results.

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.

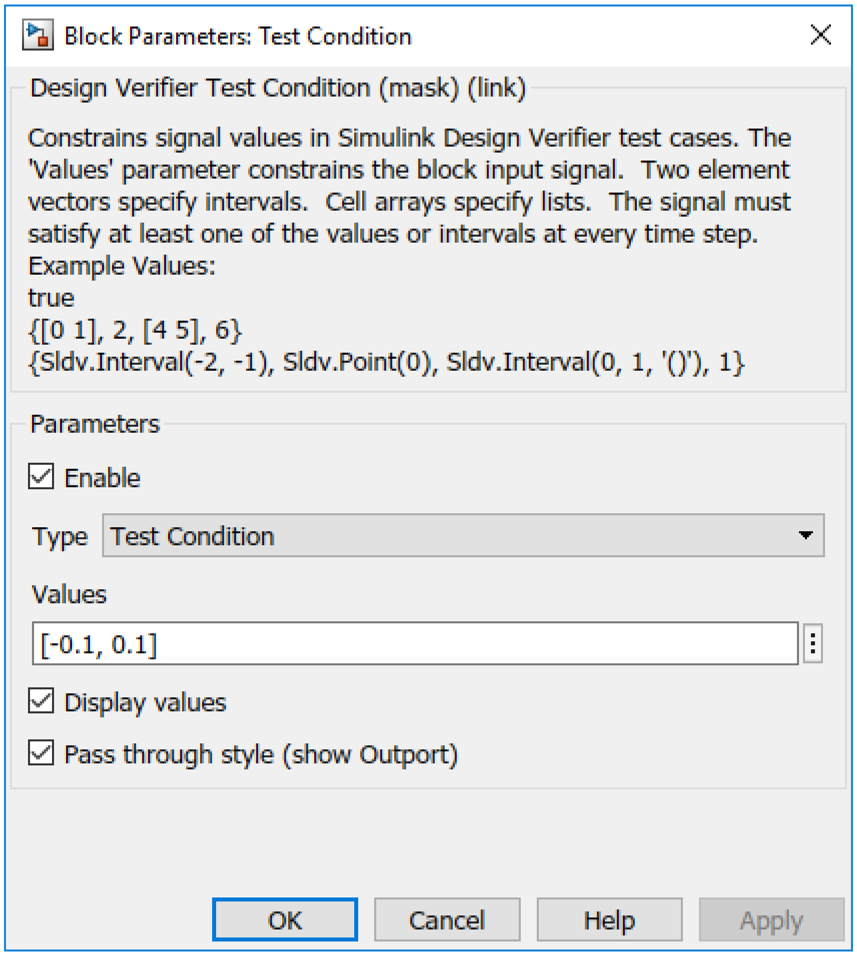
4 In the model window, insert the Test Condition block between the Inport and Switch blocks.



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.



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)

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 5-12.

1. On the **Design Verifier** tab, click **Generate Tests**.

   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 Results Summary window displays the options for reviewing the results.
2. In the Results Summary window, click **HTML 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.

See Also

More About

- Model Coverage Test Generation on page 7-64

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.

   To analyze a subsystem with Simulink Design Verifier, set the **Treat as atomic unit** parameter.

   After you set the parameter, other parameters become available, but you can ignore them.
4. To close the dialog box, click **OK**.
5. On the **Simulation** tab, in the **File** section, select **Save** > **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**
   - **View tests in Simulation Data Inspector**
   - **Detailed analysis report**
   - **Create harness model**
   - **Export test cases to Simulink Test**
   - **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 7-17.

   ---
8. Review the results of the subsystem analysis and compare the results to the results of the full-model analysis as 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 the analyses generate the same test cases.

Use Test Generation Advisor to Identify Analyzable Components

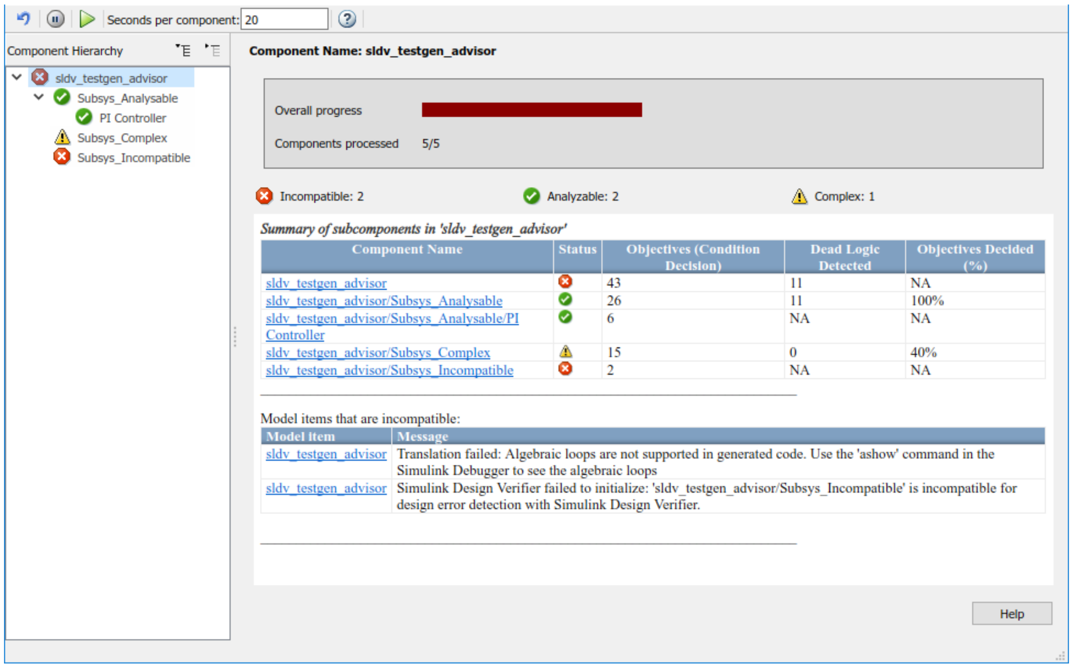
In this section...

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.



Component Name: sldv_testgen_advisor

Overall progress

Components processed 5/5

Incompatible: 2 Analyzable: 2 Complex: 1

Summary of subcomponents in 'sldv_testgen_advisor'

Component Name	Status	Objectives (Condition Decision)	Dead Logic Detected	Objectives Decided (%)
sldv_testgen_advisor	✗	43	11	NA
sldv_testgen_advisor/Subsys_Analysable	✓	26	11	100%
sldv_testgen_advisor/Subsys_Analysable/PI Controller	✓	6	NA	NA
sldv_testgen_advisor/Subsys_Complex	⚠	15	0	40%
sldv_testgen_advisor/Subsys_Incompatible	✗	2	NA	NA

Model items that are incompatible:

Model item	Message
sldv_testgen_advisor	Translation failed: Algebraic loops are not supported in generated code. Use the 'ashow' command in the Simulink Debugger to see the algebraic loops
sldv_testgen_advisor	Simulink Design Verifier failed to initialize: 'sldv_testgen_advisor/Subsys_Incompatible' is incompatible for design error detection with Simulink Design Verifier.

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. On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**, then click **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.

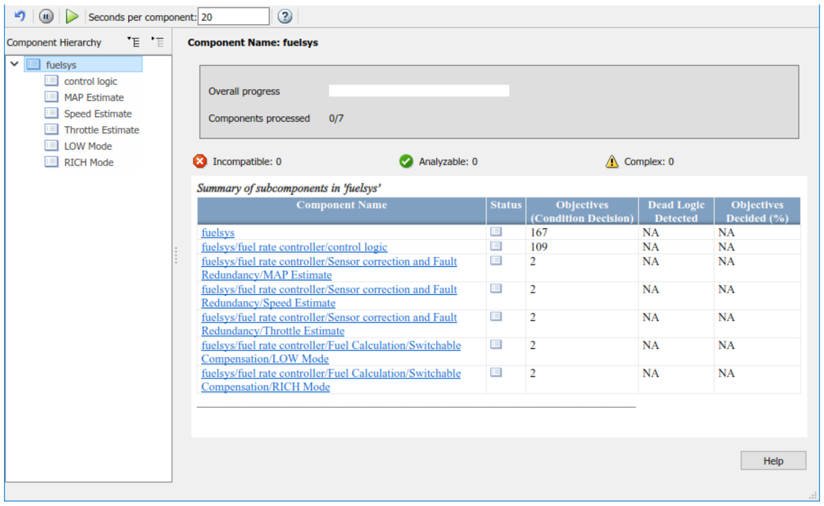
Analyze and Generate Tests for Model Components

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 On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**, then click **Advisor**.



Component Name: fuelsys

Overall progress

Components processed 0/7

Incompatible: 0 Analyzable: 0 Complex: 0

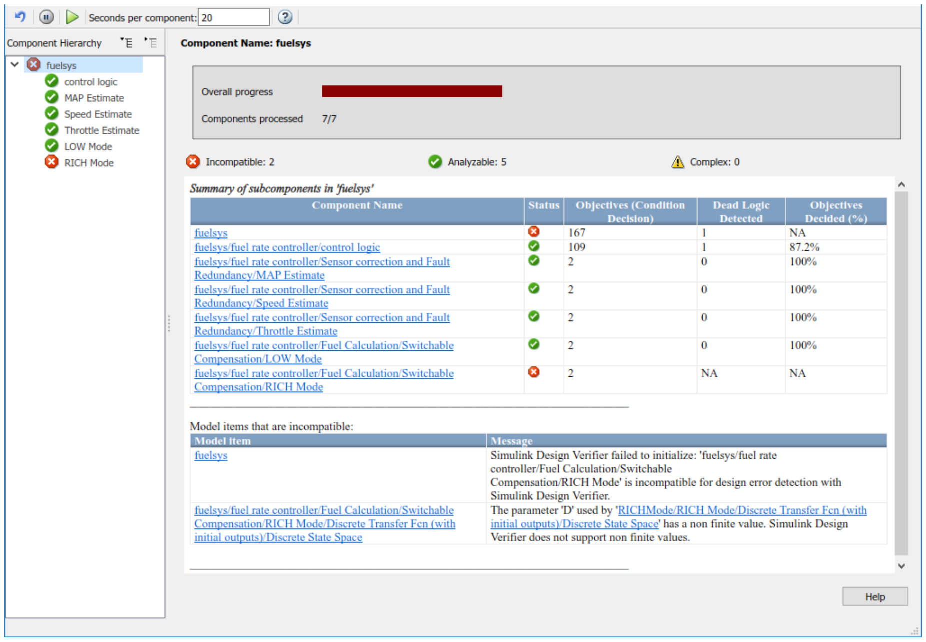
Summary of subcomponents in 'fuelsys'

Component Name	Status	Objectives (Condition Decision)	Dead Logic Detected	Objectives Decided (%)
fuelsys	☐	167	NA	NA
fuelsys/fuel rate controller/control logic	☐	109	NA	NA
fuelsys/fuel rate controller/Sensor correction and Fault Redundancy/MAP Estimate	☐	2	NA	NA
fuelsys/fuel rate controller/Sensor correction and Fault Redundancy/Speed Estimate	☐	2	NA	NA
fuelsys/fuel rate controller/Sensor correction and Fault Redundancy/Throttle Estimate	☐	2	NA	NA
fuelsys/fuel rate controller/Fuel Calculation/Switchable Compensation/LOW Mode	☐	2	NA	NA
fuelsys/fuel rate controller/Fuel Calculation/Switchable Compensation/RICH Mode	☐	2	NA	NA

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.



Component Name: fuelsys

Overall progress

Components processed 7/7

Incompatible: 2 Analyzable: 5 Complex: 0

Summary of subcomponents in 'fuelsys'

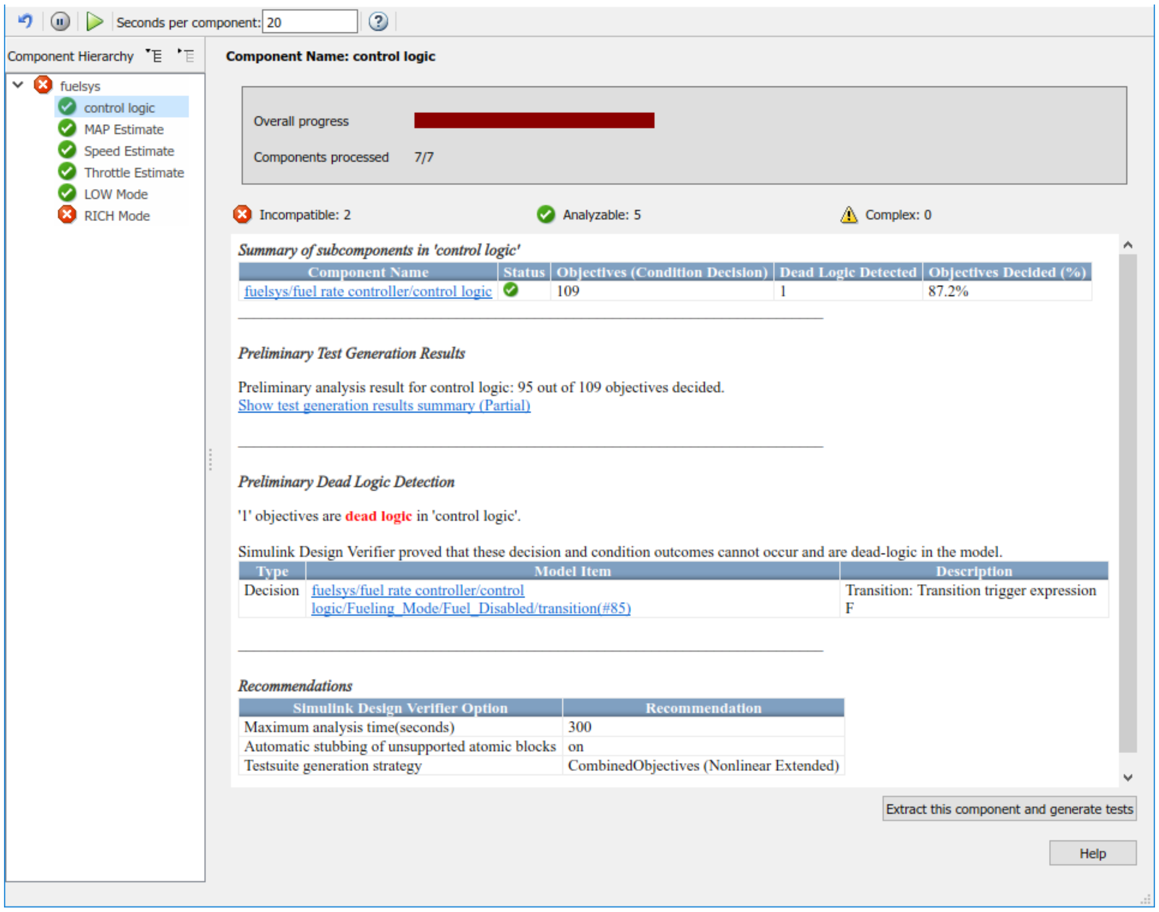
Component Name	Status	Objectives (Condition Decision)	Dead Logic Detected	Objectives Decided (%)
fuelsys	✗	167	1	NA
fuelsys/fuel rate controller/control logic	✓	109	1	87.2%
fuelsys/fuel rate controller/Sensor correction and Fault Redundancy/MAP Estimate	✓	2	0	100%
fuelsys/fuel rate controller/Sensor correction and Fault Redundancy/Speed Estimate	✓	2	0	100%
fuelsys/fuel rate controller/Sensor correction and Fault Redundancy/Throttle Estimate	✓	2	0	100%
fuelsys/fuel rate controller/Fuel Calculation/Switchable Compensation/LOW Mode	✓	2	0	100%
fuelsys/fuel rate controller/Fuel Calculation/Switchable Compensation/RICH Mode	✗	2	NA	NA

Model items that are incompatible:

Model item	Message
fuelsys	Simulink Design Verifier failed to initialize: 'fuelsys/fuel rate controller/Fuel Calculation/Switchable Compensation/RICH Mode' is incompatible for design error detection with Simulink Design Verifier.
fuelsys/fuel rate controller/Fuel Calculation/Switchable Compensation/RICH Mode/Discrete Transfer Fcn (with initial outputs)/Discrete State Space	The parameter 'D' used by 'RICHMode/RICH Mode/Discrete Transfer Fcn (with initial outputs)/Discrete State Space' has a non finite value. Simulink Design Verifier does not support non finite values.

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.



Seconds per component: 20

Component Hierarchy

- ✖ fuelsys
  - ✓ control logic
  - ✓ MAP Estimate
  - ✓ Speed Estimate
  - ✓ Throttle Estimate
  - ✓ LOW Mode
  - ✖ RICH Mode

Component Name: control logic

Overall progress 🟥

Components processed 7/7

✖ Incompatible: 2 ✓ Analyzable: 5 ⚠ Complex: 0

Summary of subcomponents in 'control logic'

Component Name	Status	Objectives (Condition Decision)	Dead Logic Detected	Objectives Decided (%)
fuelsys/fuel rate controller/control logic	✓	109	1	87.2%

Preliminary Test Generation Results

Preliminary analysis result for control logic: 95 out of 109 objectives decided.
Show test generation results summary (Partial)

Preliminary Dead Logic Detection

'1' objectives are **dead logic** in 'control logic'.

Simulink Design Verifier proved that these decision and condition outcomes cannot occur and are dead-logic in the model.

Type	Model Item	Description
Decision	fuelsys/fuel rate controller/control logic/Fueling_Mode/Fuel_Disabled/transition(#85)	Transition: Transition trigger expression F

Recommendations

Simulink Design Verifier Option	Recommendation
Maximum analysis time(seconds)	300
Automatic stubbing of unsupported atomic blocks	on
Testsuite generation strategy	CombinedObjectives (Nonlinear Extended)

Extract this component and generate tests

Help

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.

See Also

More About

- "Model Coverage Objectives for Test Generation" on page 7-23
- "Generate Test Cases for Model Decision Coverage" on page 7-3

Generate Test Cases for Embedded Coder Generated Code

In this section...

When you use Embedded Coder to generate code from a model set to software-in-the-loop (SIL) mode, you can use Simulink Coverage to record coverage metrics on the generated code. However, the same tests that enable you to achieve 100% model coverage might not produce 100% coverage for the generated code. Some differences between the output code and the model can cause gaps in the code coverage compared to the model coverage:

- Extra custom code files
- Shared utility files
- Code transformations, such as:
  - Expression folding
  - Simplified or expanded expressions
  - New decision points due to lookup tables

You can use Simulink Design Verifier to generate test cases to increase coverage for generate code. You generate test cases for generated code from the block diagram, by using the Simulink Design Verifier API, or from the Simulink Test Test Manager. Before you generate test cases, you need to record coverage results at least once.

Generate Test Cases for Generated Code from the Block Diagram

After you Enable SIL Code Coverage for a Model (Simulink Coverage), simulate the model, and record code coverage data, you use Simulink Design Verifier to generate additional test cases for the generated code:

1. If you have not previously recorded coverage results, enable coverage and simulate the model.
2. If you have already recorded coverage results, indicate the existing coverage data. In the Configuration Parameters dialog box, on the “Design Verifier Pane: Test Generation” on page 15-31 pane, select **Ignore objectives satisfied in existing coverage data** and select the existing coverage data file.
3. On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**.

   - To generate tests for code generated as top model, select **Target** > **Code Generated as Top Model**, then click **Generate Tests**.

   - To generate tests for code generated as model reference, select **Target** > **Code Generated as Model Reference**, then click **Generate Tests**.

   Simulink Design Verifier test generation proceeds according to the test generation mode that you choose.

To learn more about the differences between code generated as top model and code generated as model reference, see:

- “Configure and Run SIL Simulation” (Embedded Coder)
- “Code Interfaces for SIL and PIL” (Embedded Coder)
- “Choose a SIL or PIL Approach” (Embedded Coder)

Generate Test Cases for Generated Code by Using the Simulink Design Verifier API

For an example of how to programmatically generate test cases for generated code, see “Code Coverage Test Generation”.

Generate Test Cases for Generated Code from the Simulink Test Test Manager

If you use the Simulink Test Test Manager to record code coverage for a model set to SIL mode, you can incrementally increase coverage for the generated code directly from the Test Manager. For more information, see “Incrementally Increase Test Coverage Using Test Case Generation” on page 16-9.

See Also

More About

- “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-27

Model Coverage Objectives for Test Generation

In this section...

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" (Simulink Coverage).

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)" (Simulink Coverage).

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" (Simulink Coverage).

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)" (Simulink Coverage). .

MCDC

Modified 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" (Simulink Coverage).

To enable 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” (Simulink Coverage).

For information on how MCDC test generation in Simulink Design Verifier can deviate from MCDC coverage recorded by Simulink Coverage, see “Modified Condition and Decision Coverage in Simulink Design Verifier” on page 9-14.

Enhanced MCDC

Enhanced MCDC is an extension of modified condition decision coverage. For a test block, enhanced MCDC generates test cases that avoid masking effects from downstream blocks, so that the test block has an effect on the output.

To enable enhanced MCDC coverage, under **Design Verifier** > **Test Generation**, for **Model coverage objectives**, select `Enhanced MCDC`. For more information, see “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-35.

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” (Simulink Coverage). 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**.

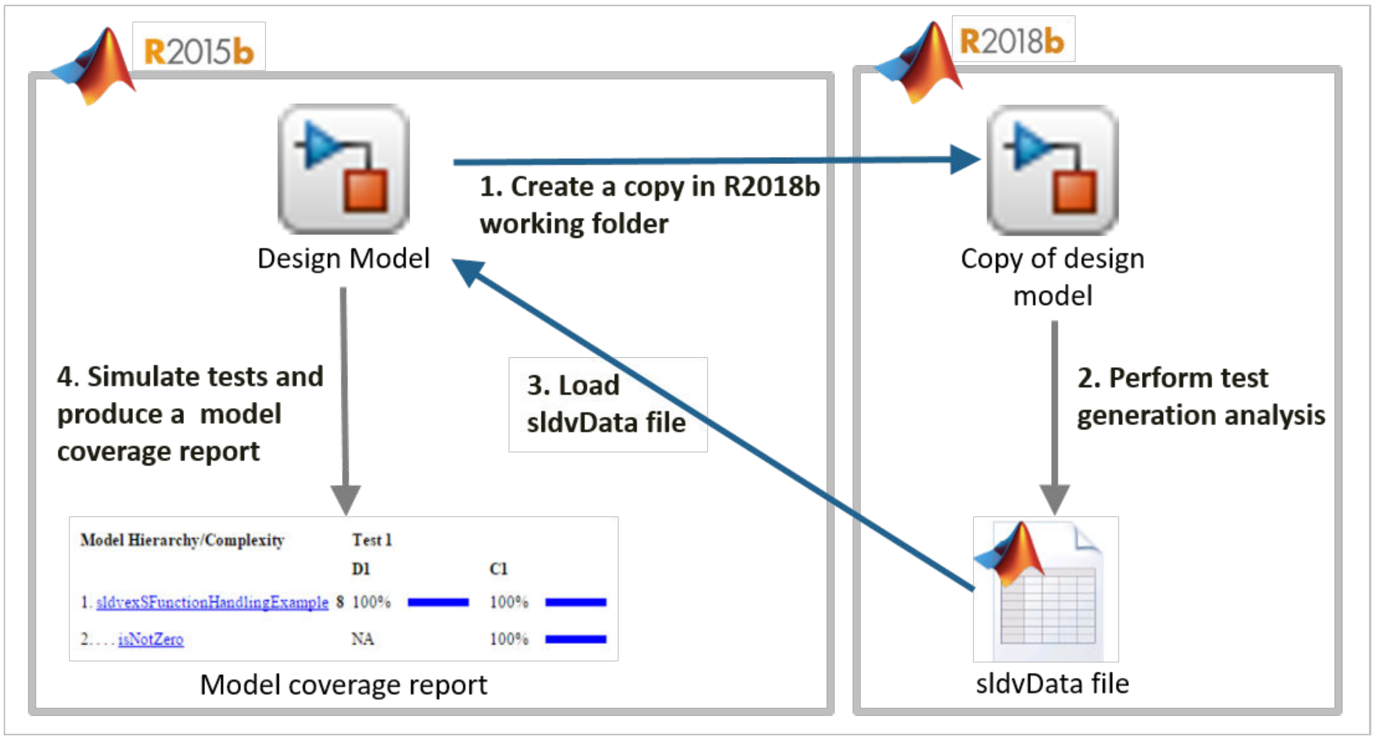
For each relational operation in the model, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see “Relational Boundary Coverage” (Simulink Coverage).

Enhance Model Coverage of Older Release Models

To enhance the model coverage of a model that you created in an older release, use a test generation workflow or a code generation workflow. You can leverage the latest release capabilities of Simulink Design Verifier to generate the test cases for a Model-Based Design.

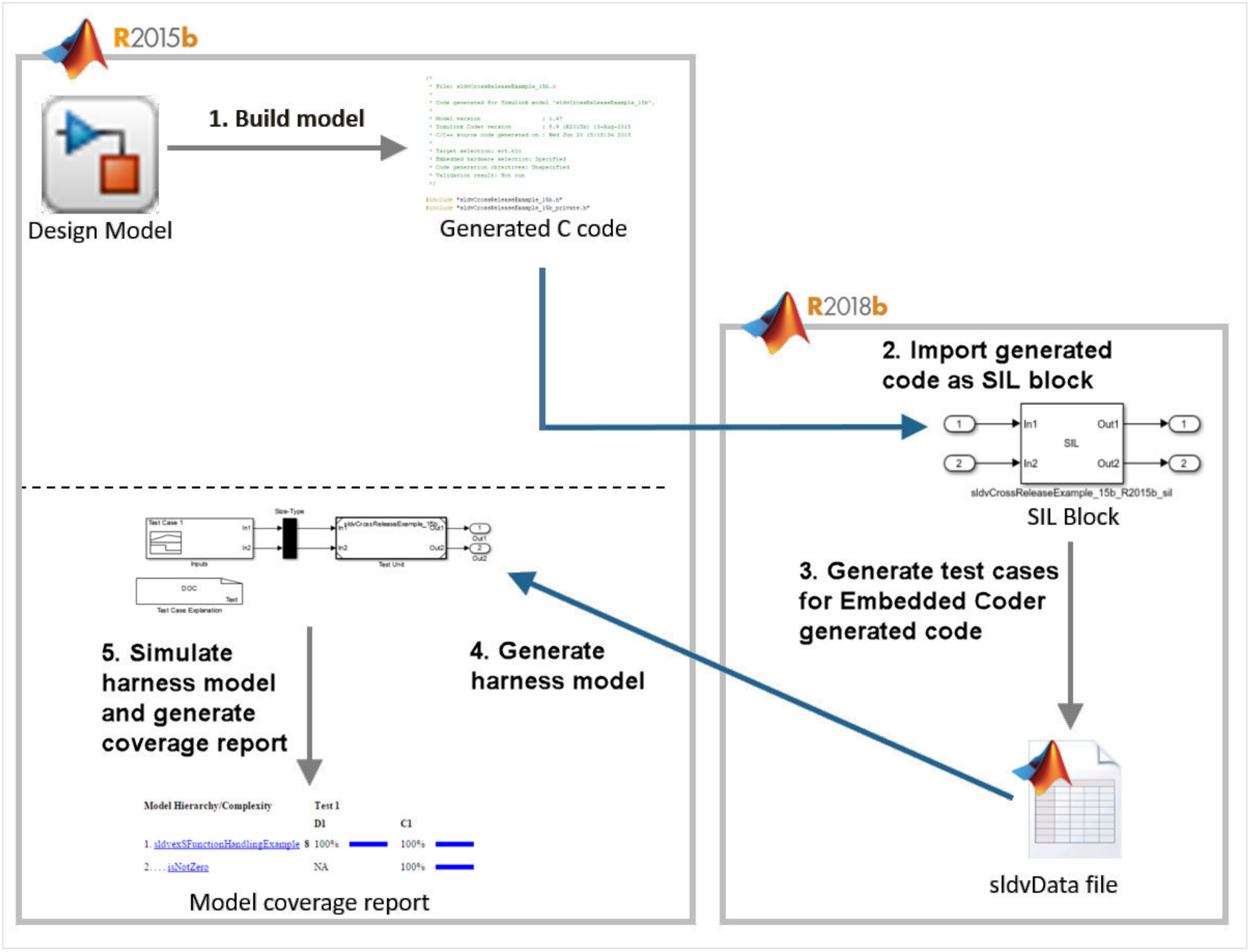
These workflows enhance model coverage.

- “Enhance Model Coverage by Generating Test Cases for Older Release Model” on page 7-26


R2015b
R2018b
Design Model
1. Create a copy in R2018b working folder
Copy of design model
2. Perform test generation analysis
3. Load sldvData file
4. Simulate tests and produce a model coverage report
sldvData file
Model coverage report

Model Hierarchy/Complexity		Test 1			
		D1		C1	
1. sldvexSFunctionHandlingExample	8	100%	▬	100%	▬
2. . . . isNotZero		NA		100%	▬

- “Enhance Model Coverage by Using Generated Code from Older Release” on page 7-29


R2015b
1. Build model
Design Model
Generated C code
R2018b
2. Import generated code as SIL block
sldvCrossReleaseExample_15b_R2015b_sil
SIL Block
3. Generate test cases for Embedded Coder generated code
sldvData file
4. Generate harness model
5. Simulate harness model and generate coverage report
Model coverage report

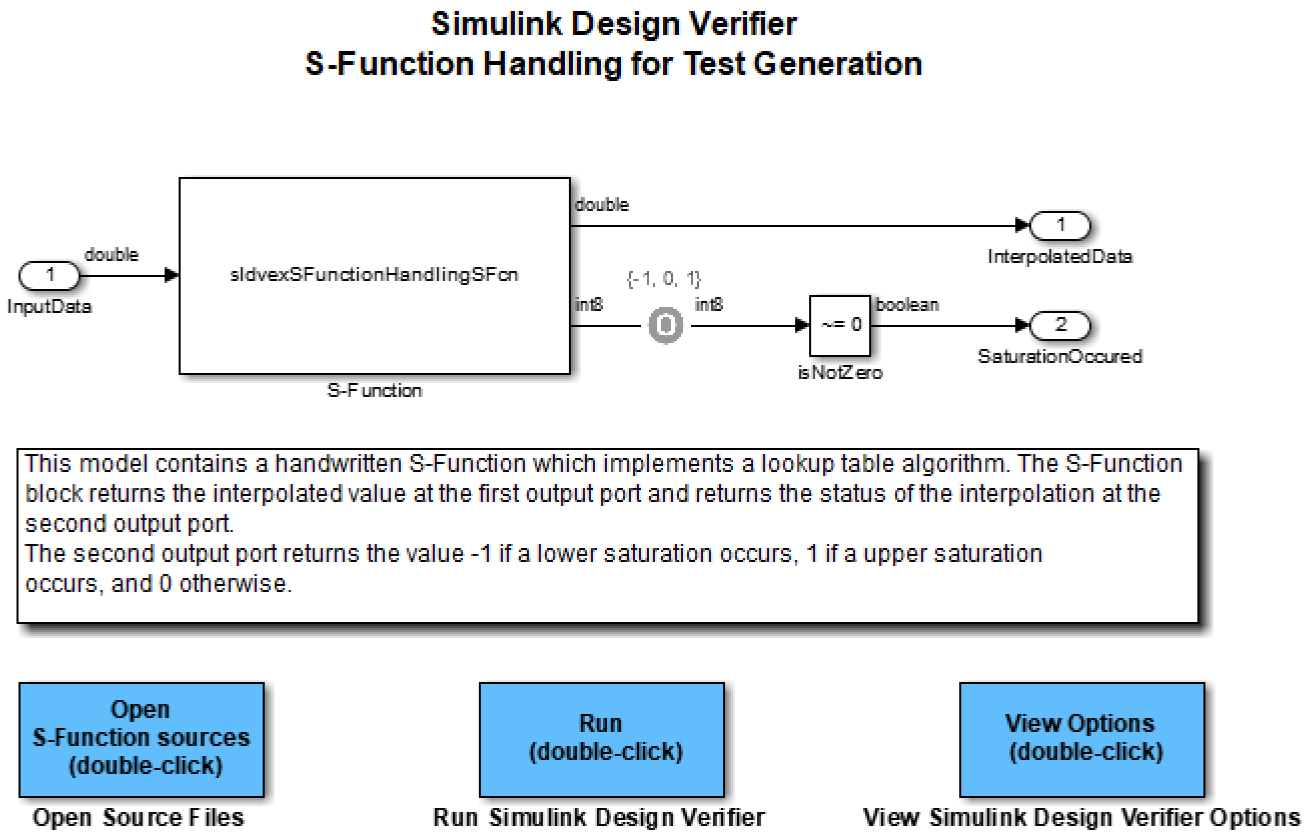
Enhance Model Coverage by Generating Test Cases for Older Release Model

This example shows how to upgrade model coverage of a model created in R2015b. You use test generation for supported S-functions available in the latest release.

The example model sldvexSFunctionHandlingExample contains the handwritten S-Function, which implements a lookup table algorithm. The handwritten S-Function is in the file sldvexSFunctionHandlingSFcn.c. The user source code for the lookup table is in the file sldvexSFunctionHandlingSource.c.

1 In MATLAB R2015b, open the sldvexSFunctionHandlingExample model.

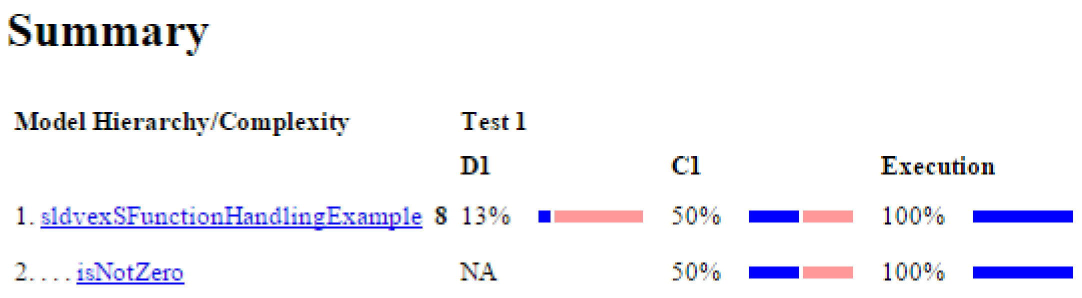
```
open_system('sldvexSFunctionHandlingExample');
```


Simulink Design Verifier
S-Function Handling for Test Generation
double
1
InputData
sldvexSFunctionHandlingSFcn
S-Function
double
1
InterpolatedData
int8
{-1, 0, 1}
int8
~= 0
isNotZero
boolean
2
SaturationOccured
Open S-Function sources (double-click)
Open Source Files
Run (double-click)
Run Simulink Design Verifier
View Options (double-click)
View Simulink Design Verifier Options

This model contains a handwritten S-Function which implements a lookup table algorithm. The S-Function block returns the interpolated value at the first output port and returns the status of the interpolation at the second output port.
The second output port returns the value -1 if a lower saturation occurs, 1 if a upper saturation occurs, and 0 otherwise.

2 To simulate the model and generate the coverage report, in the Simulink Editor, click the Run button. See “View Coverage Results in a Model” (Simulink Coverage).

After the simulation, the coverage report indicates that full coverage is not achieved for `sldvexSFunctionHandlingExample` model.



Summary

Model Hierarchy/Complexity		Test 1 D1	C1	Execution
1. sldvexSFunctionHandlingExample	8	13%	50%	100%
2. . . . isNotZero		NA	50%	100%

3 In MATLAB R2018b or later releases, open the sldvexSFunctionHandlingExample model. The example model `sldvexSFunctionHandlingExample` is available in R2015b and later releases, so you can use the same model for test generation workflow.

```
open_system('sldvexSFunctionHandlingExample');
```

To avoid any potential changes in the model, create a copy of the older release model in the current working folder, and then open the model in R2018b or later releases. To upgrade and improve models that you use in the current release, you can use the `upgradeadvisor`.

4 Compile the S-function to be compatible with Simulink Design Verifier for test case generation by using `slcovmex`. For more information, see “Configuring S-Function for Test Case Generation” on page 7-89.

```
slcovmex('-sldv', ...
        '-output', 'sldvexSFunctionHandlingSFcn',...
        ['-I', fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src')], ...
        fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src',...
        'sldvexSFunctionHandlingSource.c'),...
        fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src',...
         'sldvexSFunctionHandlingSFcn.c'));
```

5 Create an `opts` option for the `sldvexSFunctionHandlingExample` model.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'Condition';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
opts.SFcnSupport = 'on';
```

6 To generate test cases by using the specified `opts` options, use `sldvrun` to analyze the model.

```
[status, fileNames] = sldvrun('sldvexSFunctionHandlingExample', opts);
```

After analysis, the software generates a Simulink Design Verifier data file and stores it in the default location `<current_folder>\sldv_output \sldvexSFunctionHandlingExample_sldvdata.mat`

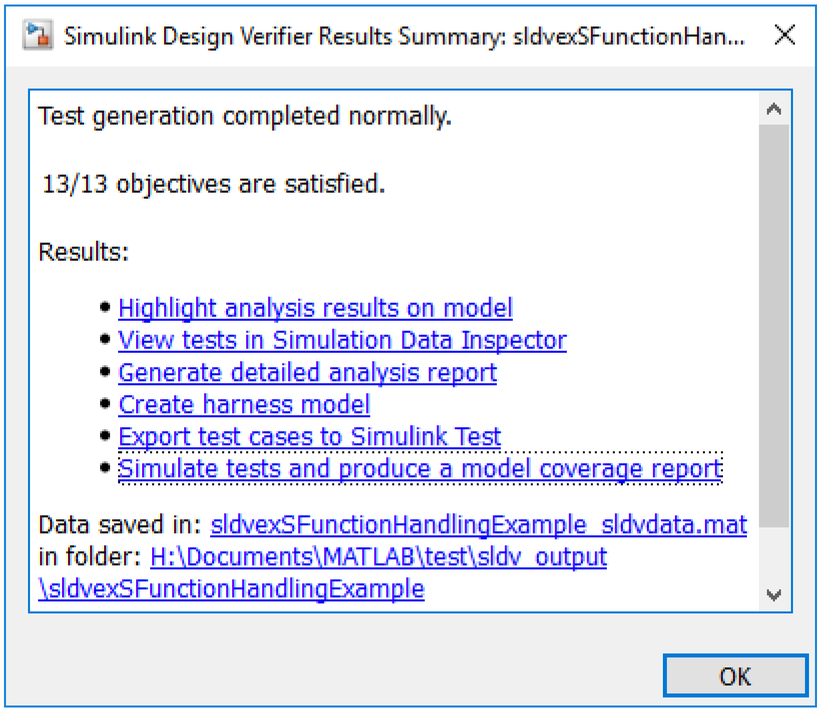
7 In R2015b, open the model.

```
open_system('sldvexSFunctionHandlingExample');
```

8 Load the `sldvData` file created in R2018b or later releases.

   a On the **Design Verifier** tab, click **Load Earlier Results** and browse to the `sldvData` MAT-file generated in R2018b or later releases.

   b Click **Open**.


Simulink Design Verifier Results Summary: sldvexSFunctionHan...
OK

Test generation completed normally.

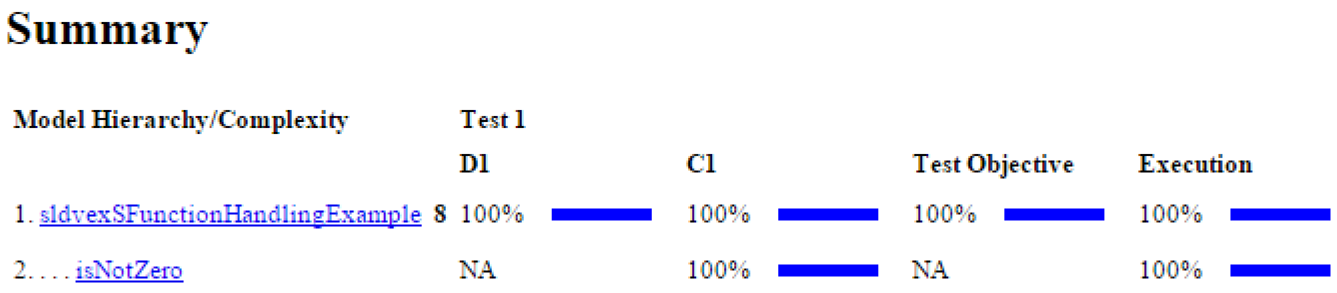
13/13 objectives are satisfied.

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Generate detailed analysis report
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: sldvexSFunctionHandlingExample_sldvdata.mat
in folder: H:\Documents\MATLAB\test\sldv_output
\sldvexSFunctionHandlingExample

9 In the Simulink Design Verifier Results Summary window, click **Simulate tests and produce a model coverage report**. The report indicates that 100% coverage is achieved for `sldvexSFunctionHandlingExample` model.



Summary

Model Hierarchy/Complexity		Test 1			
		D1	C1	Test Objective	Execution
1. sldvexSFunctionHandlingExample	8	100%	100%	100%	100%
2. . . . isNotZero		NA	100%	NA	100%

For more information, see “Simulink Design Verifier Data Files” on page 13-7 and “Simulate Tests and Produce Model Coverage Report” on page 1-17.

Enhance Model Coverage by Using Generated Code from Older Release

This example shows how to upgrade the model coverage of a model created in R2015b by using code generation workflow.

For this workflow, you must have Simulink Coder™ and Embedded Coder.

The example model sldvCrossReleaseExample contains the handwritten S-Function, which implements a relational boundary algorithm. The handwritten S-Function is in the file rel_sfcn.c. The user source code is in the file rel_comp.c.

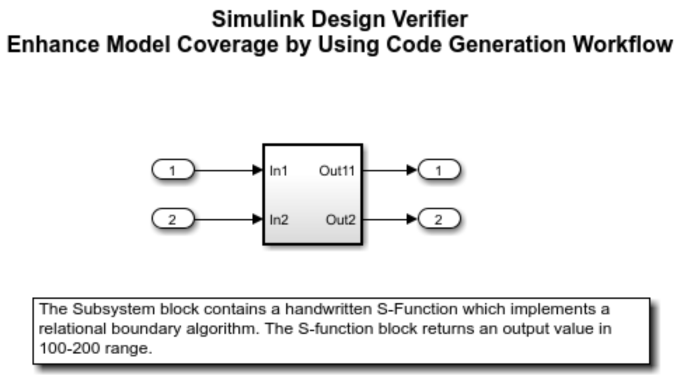
To inline the S-function, use the rel_sfcn.tlc file. For more information, see "Inline S-Functions with TLC" (Embedded Coder).

1 Copy the example model sldvCrossReleaseExample and S-Function files, rel_sfcn.c, rel_comp.c, and rel_sfcn.tlc in the current working folder. Copy the header files rel_comp.h into the current working folder. You use the example model and supporting files in R2015b for a "Cross-Release Code Integration" (Embedded Coder) workflow.

   Note The example model `sldvCrossReleaseExample` is created for example purpose. To perform code generation workflow by using the example model, export `sldvCrossReleaseExample` model to 15b. Save the model as `sldvCrossReleaseExample_15b` in the current working folder. For more information, see "Export a Model to a Previous Simulink Version" (Simulink).

2 In MATLAB R2015b, open `sldvCrossReleaseExample_15b` model from the current working folder.

```
open_system('sldvCrossReleaseExample_15b');
```



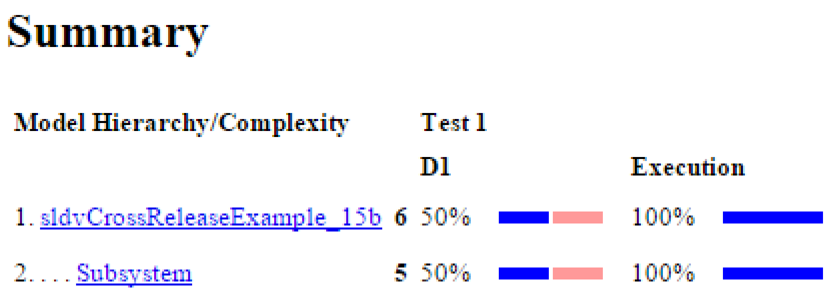
The Subsystem block contains a handwritten S-Function which implements a relational boundary algorithm. The S-function block returns an output value in 100-200 range.

3 Compile the S-function by using the function `legacy_code`.

```
def = legacy_code('initialize');
   def.SFunctionName = 'rel_sfcn';
   def.OutputFcnSpec = 'uint8 y1 = relational_bound(uint8 u1)';
   def.HeaderFiles = {'rel_comp.h'};
   def.SourceFiles = {'rel_comp.c'};
   def.IncPaths = {pwd};
   def.SrcPaths = {pwd};
   def.Options.supportCoverageAndDesignVerifier = true;
   legacy_code('sfcn_cmex_generate', def);
   legacy_code('compile', def);
```

4 To simulate the model and generate the coverage report, in the Simulink Editor, click the **Run** button. See "View Coverage Results in a Model" (Simulink Coverage).

   After the simulation, the coverage report indicates that 50% coverage is achieved for `sldvCrossReleaseExample_15b` model.



Summary

Model Hierarchy/Complexity		Test 1	
		D1	Execution
1. sldvCrossReleaseExample_15b	6	50%	100%
2. . . . Subsystem	5	50%	100%

5 To generate code using Embedded Coder, from the **Apps** tab, select **Embedded Coder**. For more information, see “Generate Code Using Embedded Coder®” (Embedded Coder).

In the **C Code** tab, click **Generate Code**.

The model is preconfigured with these code generation settings.

```
set_param(sldvCrossReleaseExample_15b,'SystemTargetFile','ert.tlc');
set_param(sldvCrossReleaseExample_15b,'PortableWordSizes','on');
set_param(sldvCrossReleaseExample_15b,'SupportNonFinite','off');
set_param(sldvCrossReleaseExample_15b,'GenCodeOnly','on');
set_param(sldvCrossReleaseExample_15b,'SolverMode','SingleTasking');
set_param(sldvCrossReleaseExample_15b,'ProdEqTarget','on');
```

The software generates C code for the model and saves the files in the default folder location `<current_folder>\sldvCrossReleaseExample_15b_ert_rtw`.

6 Save the configuration set of the model `sldvCrossReleaseExample_15b` to a MAT-file. This `ConfigSet` is used to set the configuration set of the model in R2018b and later releases.

```
config_set = getActiveConfigSet('sldvCrossReleaseExample_15b');
copiedConfig = config_set.copy;
save('copiedConfig.mat','copiedConfig');
```

7 In MATLAB R2018b or later releases, import the components exported from R2015b.

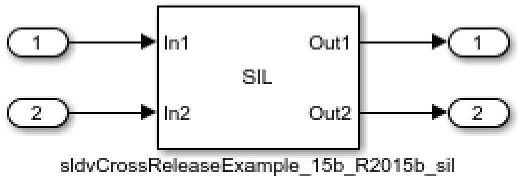
a Before you import components in current release, rename or delete `rtwtypes.h` file available in the folder `<current_folder>\sldvCrossReleaseExample_15b_ert_rtw`. During cross-release import, MATLAB tries to regenerate a file with same name. If you do not delete or rename the file `rtwtypes.h`, MATLAB displays an error.

b Import the generated component code from R2015b as software-in-the-loop (SIL) block.

```
 crossReleaseImport('sldvCrossReleaseExample_15b_ert_rtw',...
'sldvCrossReleaseExample_15b', 'SimulationMode','SIL');
```

The `crossReleaseImport` function creates an untitled model that contains software-in-the-loop (SIL) block `sldvCrossReleaseExample_15b_R2015b_sil`.

8 Add Inport and Outport ports to the `sldvCrossReleaseExample_15b_R2015b_sil` block and save the model as `sldvCrossReleaseExample_sil_18b`.



9 Apply the model configuration set similar to R2015b model.

```
load('copiedConfig.mat');
attachConfigSet('sldvCrossReleaseExample_sil_18b', copiedConfig, true);
setActiveConfigSet('sldvCrossReleaseExample_sil_18b', copiedConfig.Name);
```

10 Set the simulation mode to `Software-in-the-Loop (SIL)`. To simulate the model, in the Simulink Editor, click the **Run** button.

11 To generate test cases for Embedded Coder generated code, on the **Design Verifier** tab, select **Target** > **Code Generated as Top Model** and click **Generate Tests**. For more information, see “Generate Test Cases for Embedded Coder Generated Code” on page 7-21.

   After Simulink Design Verifier analysis, the software generates the test cases and saves the `sldvData` in folder at default location `<current_folder>\sldv_output\sldvCrossReleaseExample_sil_18b`.

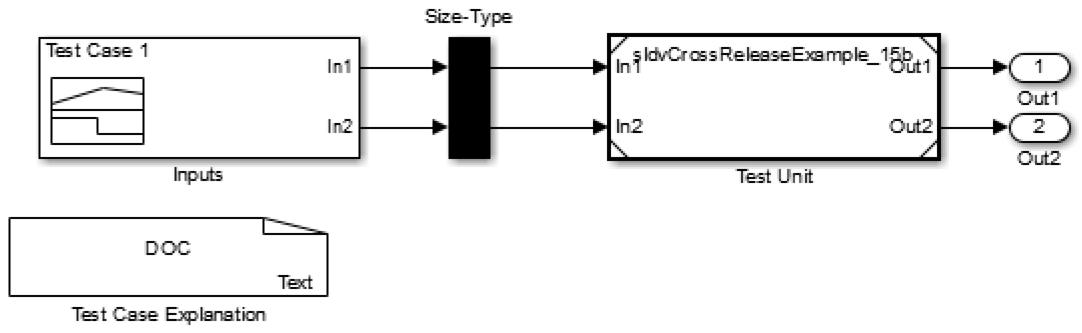
12 In R2015b, open the model.

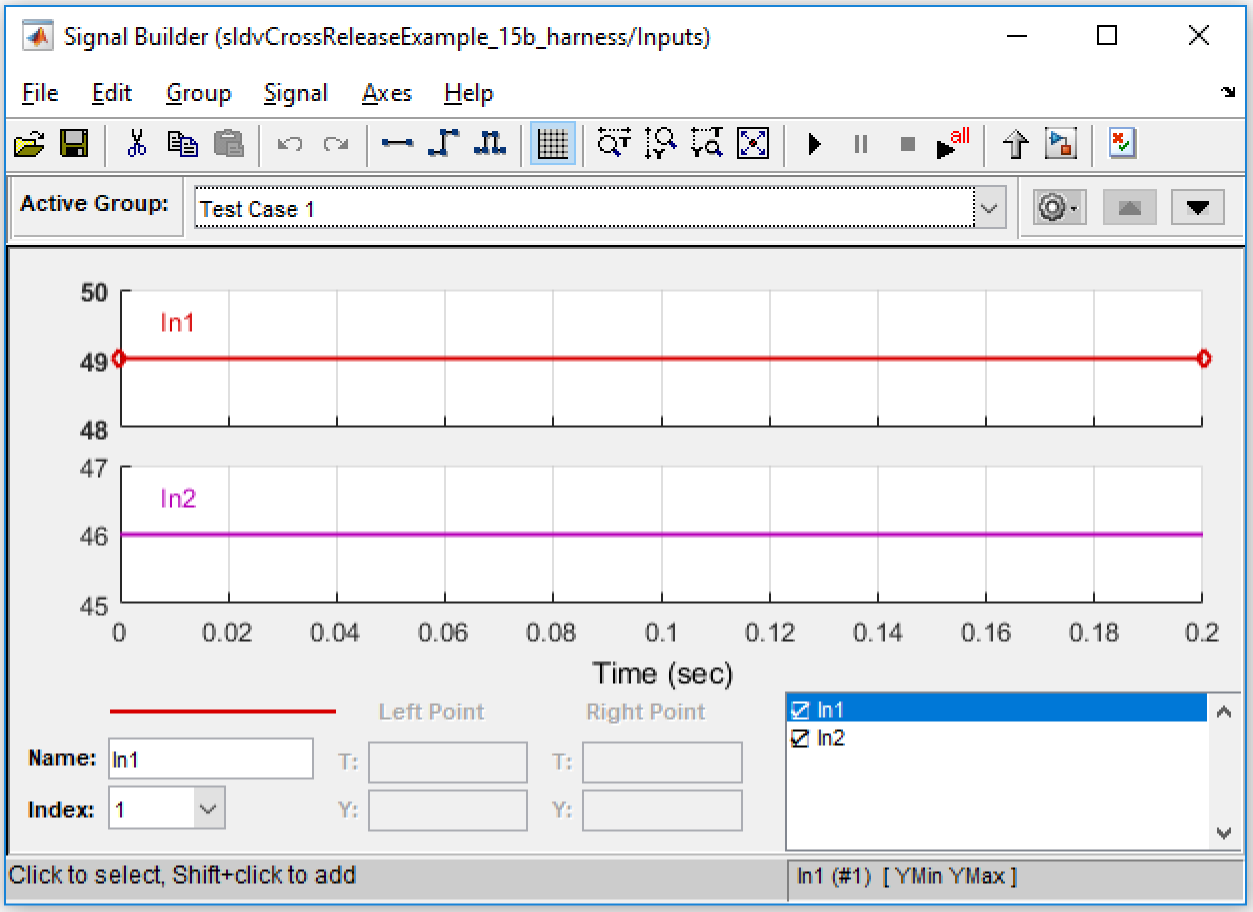
```
open_system('sldvCrossReleaseExample_15b');
```

13 Update the `sldvData.ModelInfomation.Name` field in `sldvData` same as the model name in older release. For example, `sldvCrossReleaseExample_15b.slx`.

14 Create a harness model by using the `sldvData` created in R2018b or later releases. This data consists of test cases generated from Embedded Coder generated code. In the `dataFile`, type the location of the `sldvData` generated for `sldvCrossReleaseExample_sil_18b` model.

```
sldvmakeharness('sldvCrossReleaseExample_15b.slx','dataFile')
```





Name: In1

Index: 1

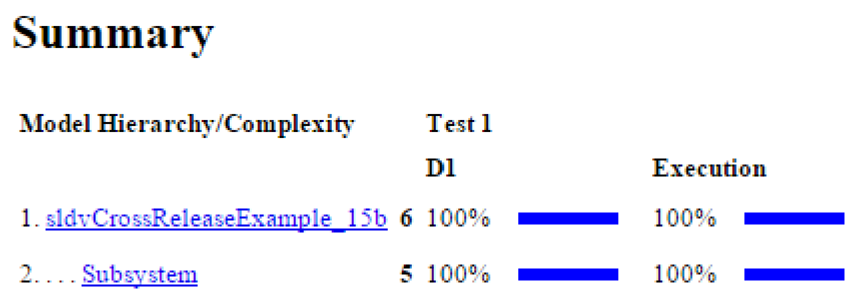
☑ In1
☑ In2

Click to select, Shift+click to add

In1 (#1) [YMin YMax]

15 To simulate the model by using all the test cases, click the **Run all** button ▶all.

The software simulates all the test cases and generates a coverage report. The report indicates that 100% coverage is achieved for `sldvCrossReleaseExample_15b` model.



Summary

Model Hierarchy/Complexity		Test 1	
		D1	Execution
1. sldvCrossReleaseExample_15b	6	100% ▬	100% ▬
2. . . . Subsystem	5	100% ▬	100% ▬

See Also

More About

- “Generate Test Cases for Embedded Coder Generated Code” on page 7-21
- “Cross-Release Code Integration” (Embedded Coder)
- “Simulink Design Verifier Data Files” on page 13-7
- “Simulink Design Verifier Harness Models” on page 13-13

Enhanced MCDC Coverage in Simulink Design Verifier

Enhanced Modified Condition Decision Coverage (MCDC) is an extension of modified condition decision coverage. For a test block, enhanced MCDC generates test cases that avoid masking effects from downstream blocks, so that the test block has an effect on the output.

To detect the effect of a test block by using the enhanced MCDC coverage objective, you can consider a standard model coverage objective of a test block or you can author your own custom test objectives for analysis. For more information, see:

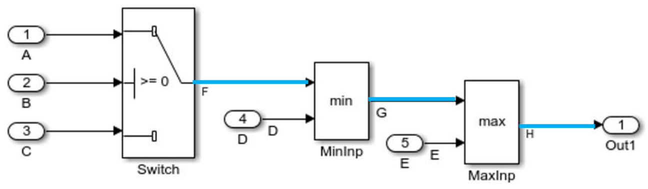
- Use Model Coverage Objectives for Enhanced MCDC Coverage on page 7-35
- Author Custom Test Objectives for Enhanced MCDC Coverage on page 7-36

To generate test cases by using enhanced MCDC model coverage objectives, and then analyzing the results, see Basic Workflow for Enhanced MCDC Analysis on page 7-40.

Use Model Coverage Objectives for Enhanced MCDC Coverage

For a given test block, you can detect the effect on a model coverage objective from the downstream blocks. When you generate test cases by using enhanced MCDC model coverage objectives, the generated test cases avoid the masking effect from the downstream blocks. The model coverage objective is detectable at a detection site.

Consider this model that consists of a cascade of Switch, Min, and Max blocks.



The test cases generated for enhanced MCDC coverage ensure that the decision objective of the “Switch” (Simulink Coverage) test block is not masked by the downstream Min and Max blocks. The generated test cases ensure that these nonmasking conditions for Min and Max blocks are satisfied:

1. F < D, ensures that the Min block does not mask the Switch output.
2. G > E, ensures that the Max block does not mask the Min output.

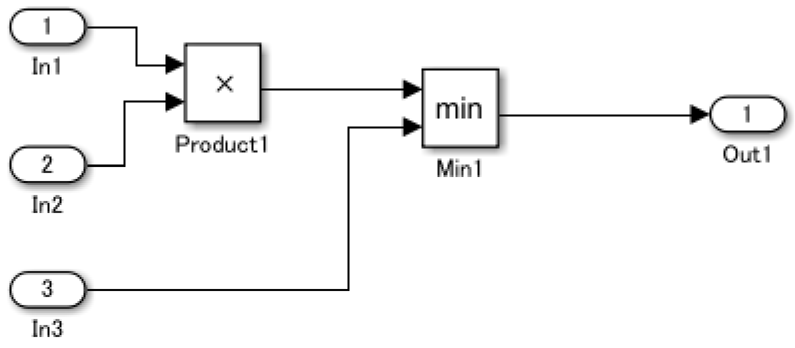
The decision objective of the Switch block and the nonmasking conditions of the Min and Max blocks are satisfied along the path and are detected at the detection site (`Out1`). For example, the path starts from the output signal of the `Switch` block, propagates along the Min block, and ends at the output signal of the Max block (highlighted in the example model).

Enhanced MCDC criteria ensure better quality test cases because the test case detects the effect of a model coverage objective of the test block at the detection site. To analyze a model for enhanced MCDC analysis, see example Analyze a Model for Enhanced MCDC analysis on page 7-37.

Author Custom Test Objectives for Enhanced MCDC Coverage

Enhanced MCDC considers the default coverage objectives of a test block that are detectable at the detection site. To check the detectability status of a custom test objective, you can author the test objective of a model object, and then perform enhanced MCDC analysis.

Consider this model that consists of a Product block and a Min block. The Product block does not have a coverage objective.



You can author a custom test objective for the Product block to render the output value less than `0` and detect the effect of the custom test objective at a detection site.

For more information, see Author Custom Test Objective Workflow on page 7-43.

See Also

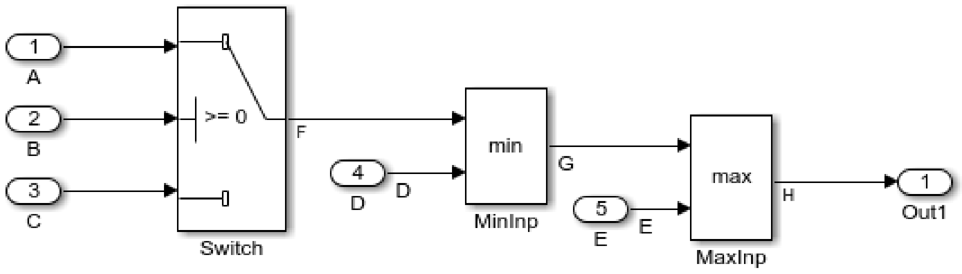
More About

- “Model Coverage Objectives for Test Generation” on page 7-23
- “Design Verifier Pane: Test Generation” on page 15-31

Analyze a Model for Enhanced MCDC Analysis

This example shows how to generate test cases for enhanced Modified Condition Decision Coverage (MCDC) objectives. You generate test cases for enhanced MCDC coverage objectives and review analysis results. The `sldvEnhancedMCDCExample` model consists of Switch, Min, and Max blocks.

1 Open the model sldvEnhancedMCDCExample.



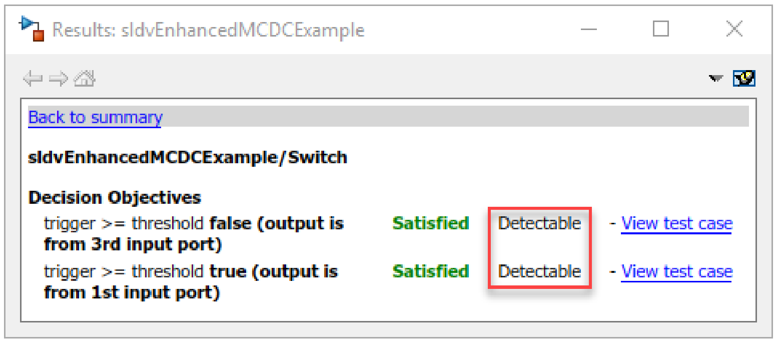
2 To configure the model for Enhanced MCDC objectives, in the Configuration Parameters dialog box, on the**Design Verifier > Test generation** pane, set **Model coverage objectives** to `Enhanced MCDC`. Click **OK**.

3 To generate test cases, on the **Design Verifier** tab, click **Generate Tests**.

After the analysis is completed, the Results Summary window displays the processed objectives and options to review the results.

4 To highlight the analysis results, click **Highlight analysis results on model**.

To analyze whether the model coverage objectives of the Switch test block are detectable, click the Switch block.



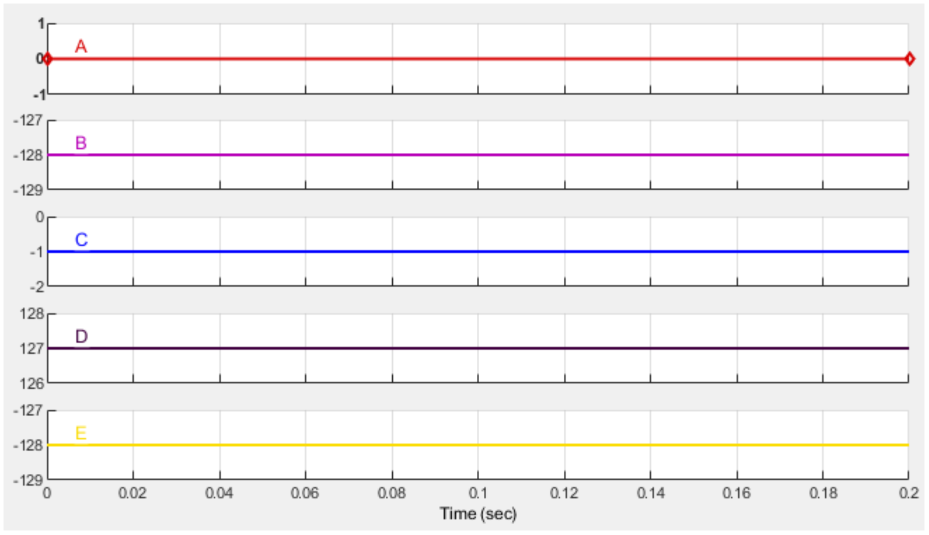
sldvEnhancedMCDCExample/Switch

Decision Objectives

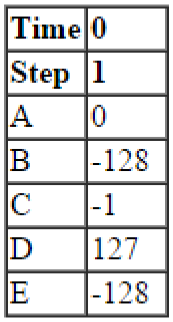
trigger >= threshold **false (output is from 3rd input port)**	**Satisfied**	Detectable	- View test case
trigger >= threshold **true (output is from 1st input port)**	**Satisfied**	Detectable	- View test case

The results show that the decision objectives of the Switch block are detectable.

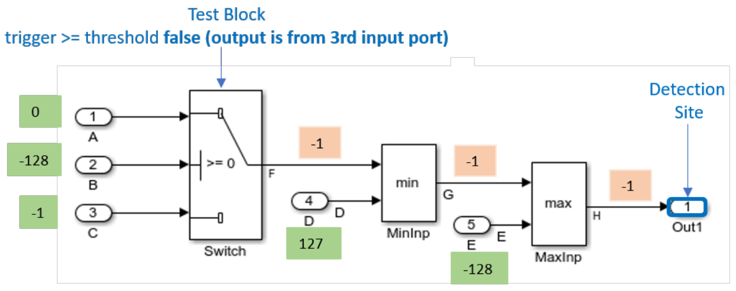
5 Click **View test case**. The harness model opens and the Signal Builder block displays `Test case 4`.



You can also view the test cases from the detailed analysis report.

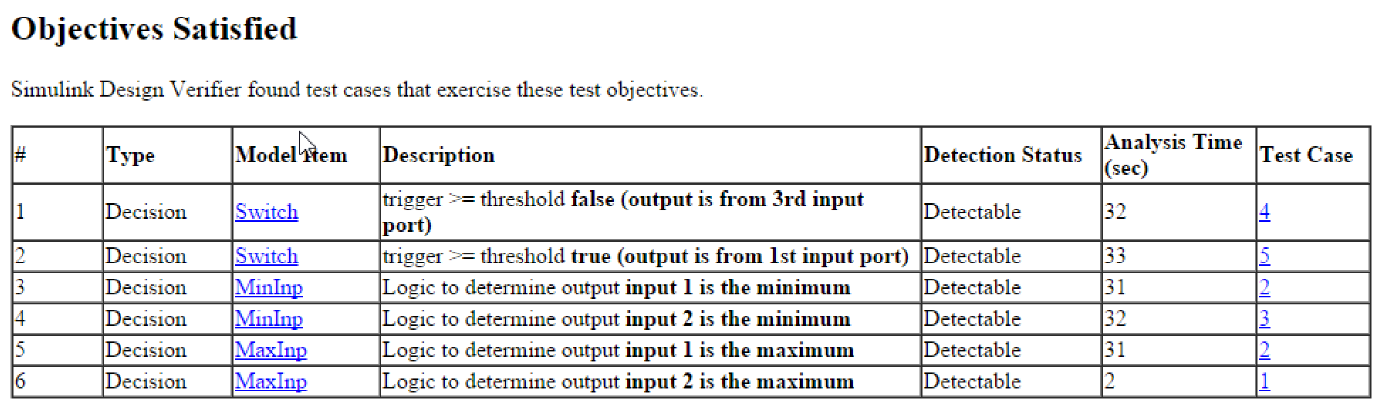


Time	0
Step	1
A	0
B	-128
C	-1
D	127
E	-128



The test case inputs `A`, `B`, and `C` result in `F = -1` and `G = -1`. The value of `E = -128` results in `H = -1`, so the impact of the test objective is detected at the detection site `Out1`. The impact of the model coverage objective of the test block is not masked along the path and is detectable at `Out1`.

6 To view the detailed analysis report, click **HTML** in the Results Summary. The Test Objectives Status section lists the satisfied objectives. The coverage objective that is detectable at the detection site is summarized in the table.



Objectives Satisfied

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

#	Type	Model Item	Description	Detection Status	Analysis Time (sec)	Test Case
1	Decision	Switch	trigger >= threshold **false (output is from 3rd input port)**	Detectable	32	4
2	Decision	Switch	trigger >= threshold **true (output is from 1st input port)**	Detectable	33	5
3	Decision	MinInp	Logic to determine output **input 1 is the minimum**	Detectable	31	2
4	Decision	MinInp	Logic to determine output **input 2 is the minimum**	Detectable	32	3
5	Decision	MaxInp	Logic to determine output **input 1 is the maximum**	Detectable	31	2
6	Decision	MaxInp	Logic to determine output **input 2 is the maximum**	Detectable	2	1

The **Objectives** field in the Simulink Design Verifier data files lists the detectability status and the detection sites for the model coverage objectives. Fore more information, see “Simulink Design Verifier Data Files” on page 13-7.

See Also

More About

- “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-35

Basic Workflow for Enhanced MCDC Analysis

To generate test cases for enhanced Modified Condition Decision Coverage (MCDC) coverage objectives:

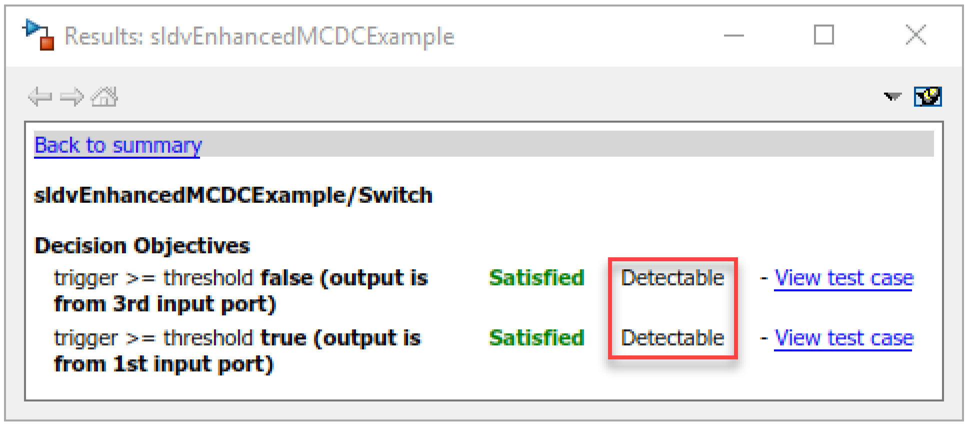
1. On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**.
2. Click **Test Generation Settings**.
3. In the Configuration Parameters dialog box, on the **Design Verifier** > **Test Generation** pane, set **Model coverage objectives** to `Enhanced MCDC`. Click **OK**.
4. Click **Generate Tests**.

Note Enhanced MCDC analysis is not supported when you "Generate Test Cases for Embedded Coder Generated Code" on page 7-21. The software considers MCDC coverage objectives for test generation analysis.

Simulink Design Verifier analyzes the model for Enhanced MCDC coverage objectives.

After the analysis is complete:

- The software highlights the model with the analysis results.
- The Results Inspector window displays the summary of the model coverage objectives including the detectability status.



sldvEnhancedMCDCExample/Switch

Decision Objectives

trigger >= threshold **false (output is from 3rd input port)**	**Satisfied**	Detectable	- View test case
trigger >= threshold **true (output is from 1st input port)**	**Satisfied**	Detectable	- View test case

The Results Inspector window displays these detectability statuses for a model coverage objective:

- Detectable
- Not Detectable
- Undecided

The table lists the possible combinations of the objective status and the detectability statuses.

Objective Status	Detectability Status	Test Case Description
Satisfied	Detectable	The test case satisfies the model coverage objective and is detectable at the detection site.
Satisfied - Needs Simulation	Detectable	The test case satisfies the model coverage objective and is detectable at the detection site. To confirm the satisfied status, you must run additional simulations of test cases. For more information, see “Objectives Satisfied - Needs Simulation” on page 13-37.
Satisfied	Not detectable	The test case satisfies the model coverage objective. However, the test objective is not detectable at any detection site.
Satisfied	Undecided	The test case satisfies the model coverage objective. The software is unable to show the effect of model coverage objective on the downstream blocks.
Unsatisfiable	Not Detectable	The test objective is unsatisfiable and not detectable at any detection site.
Undecided	Undecided	The test objective is undecided and the software is unable to show its effect on the downstream blocks.

- The Simulink Design Verifier data file stores the detectability status and detection site for model coverage objectives. For more information see, “Simulink Design Verifier Data Files” on page 13-7.

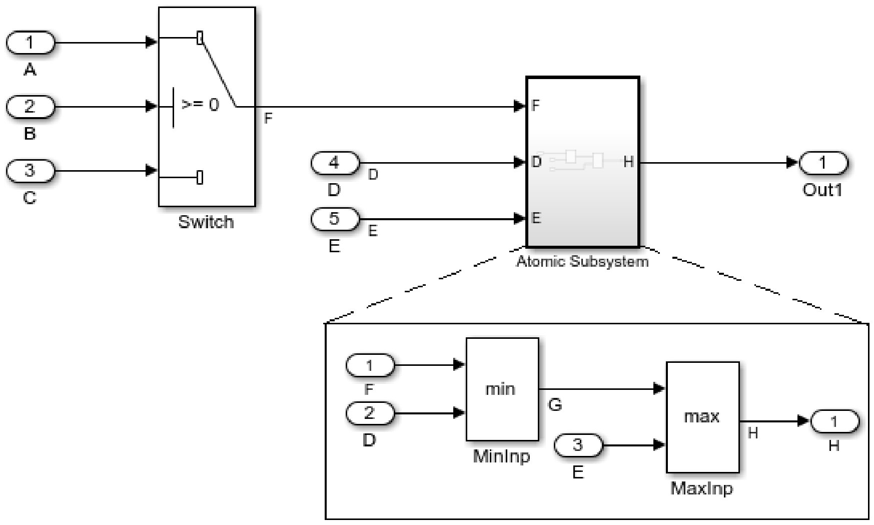
Configure Advanced Options for Enhanced MCDC Analysis

To analyze a model with stricter nonmasking conditions, enable the “Use strict propagation conditions” on page 15-38 option. This option is available in the Configuration Parameters dialog box, on the **Design Verifier > Test Generation** pane, in **Advanced parameters**.

The software evaluates stricter nonmasking conditions to analyze the effect on the test block from the downstream blocks. For example:

- If your model consists of Atomic Subsystem with the Function packaging (Simulink) option set to `Auto` or `Inline`.

  Consider a model that consists of Switch and Atomic Subsystem blocks. The Function packaging (Simulink) option is set to `Auto` and you enable the "Use strict propagation conditions" on page 15-38 option. The effect of the Switch test block is detectable at the detection point `Out1`.

  When you analyze the model with the "Use strict propagation conditions" on page 15-38 option set to `Off`, the software analyzes the model until the effect of the Switch test block reaches the Atomic Subsystem. The Atomic Subsystem is the detection point.

- If your model consists of blocks such as Gain or Product with the **Saturate on integer overflow** option set to `On`.

See Also

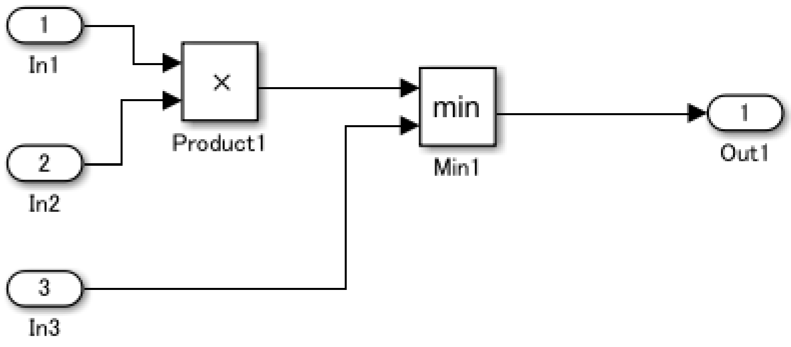
More About

- "Enhanced MCDC Coverage in Simulink Design Verifier" on page 7-35

Author Custom Test Objective Workflow

Enhanced Modified Condition Decision Coverage (MCDC) considers the default coverage objectives of a test block that are detectable at the detection site. To check the detectability status of a custom test objective, you can author the test objective of a model object, and then perform Enhanced MCDC analysis.

Consider this model that consists of a Product block and a Min block. You can author a custom test objective for the Product block to render the output value less than `0` and detect the effect of the custom test objective at a detection site.

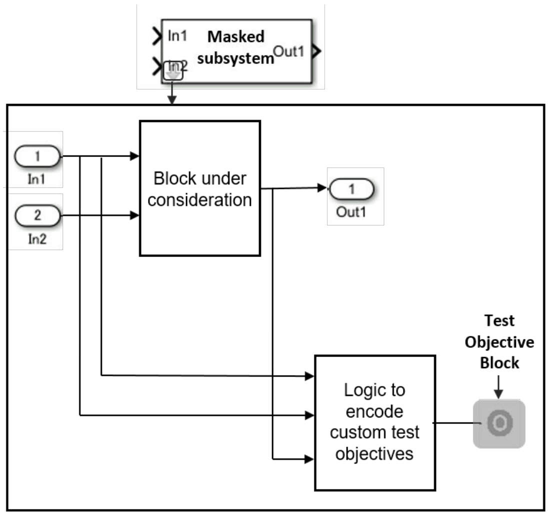


Steps for Authoring Custom Test Objectives

This workflow describes the steps for authoring custom test objectives for a block.

Step 1: Create a library of atomic masked subsystem to author the custom test objectives. The masked subsystem consists of these blocks:

- Block under consideration, for example, a Product block.
- Logic to encode the custom test objective, for example, a MATLAB Function block.
- Simulink Design Verifier Test Objective blocks.


In1
In2
Masked subsystem
Out1
1
In1
2
In2
Block under consideration
1
Out1
Test Objective Block
Logic to encode custom test objectives

Step 2: In the masked subsystem:

- Add `isEnabledForDetectability` parameter and set the parameter to `On`.
- Add the `detectBlock` parameter with the name of the block under consideration.
- Set the `Evaluate` attribute of the `detectBlock` parameter to `Off`.

Step 3: Define the block replacement rule to replace the block under consideration with a masked subsystem.

To author custom test objectives, use blkrep_rule_product_customTestObjective block replacement rule example file. In the block replacement file, you update the `rule.BlockType` and `rule.ReplacementPath` based on your model blocks. For more information, see “Block Replacements for Unsupported Blocks” on page 4-8.

Step 4: Configure your model with the block replacement rule. In the Configuration Parameters dialog box, in **Design Verifier** > **Block Replacements** pane, enter the **List of block replacement rules**.

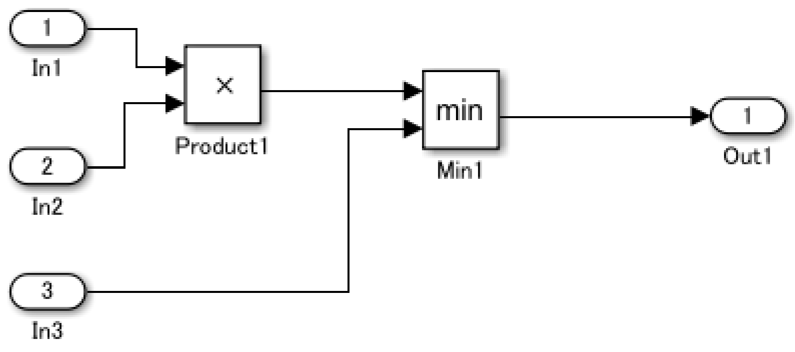
Step 5: Select `Enhanced MCDC` for **Model coverage objectives** and perform test generation analysis.

Analyze Custom Test Objectives in a Model for Enhanced MCDC

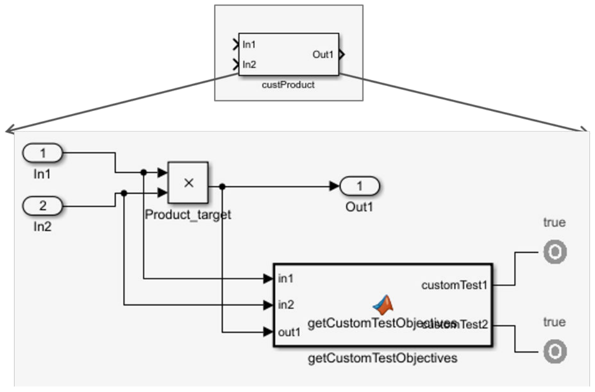
This example shows how to author custom test objectives for the Product block in the `sldvCustomTestObjectiveExample` model. Then, it shows how you can detect the effect of the test objective at a detection site.

1 Open the sldvCustomTestObjectiveExample model.

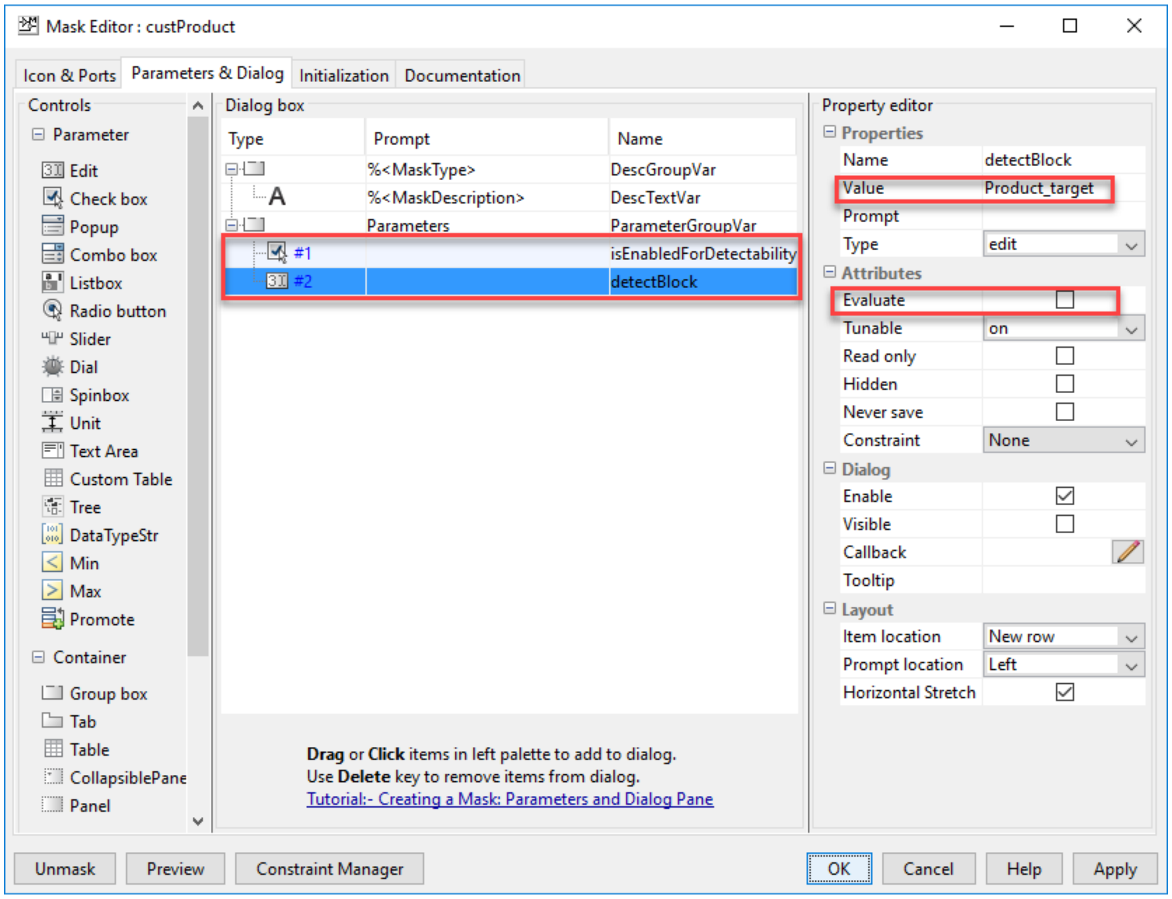
```
addpath(fullfile(docroot,'toolbox','sldv','examples'));
open_system('sldvCustomTestObjectiveExample');
```



Library of atomic masked subsystem: The blkReplacementlib_customTestObjective library consists of the `custProduct` masked subsystem. The logic to encode the custom test objective is defined in the MATLAB Function block. The `getCustomTestObjectives` MATLAB Function block consists of two custom conditions for the Test Objective blocks.



The `custProduct` masked subsystem is preconfigured with these parameters. For more information, see “Mask Editor Overview” (Simulink).


Mask Editor : custProduct

Icon & Ports | Parameters & Dialog | Initialization | Documentation

Dialog box

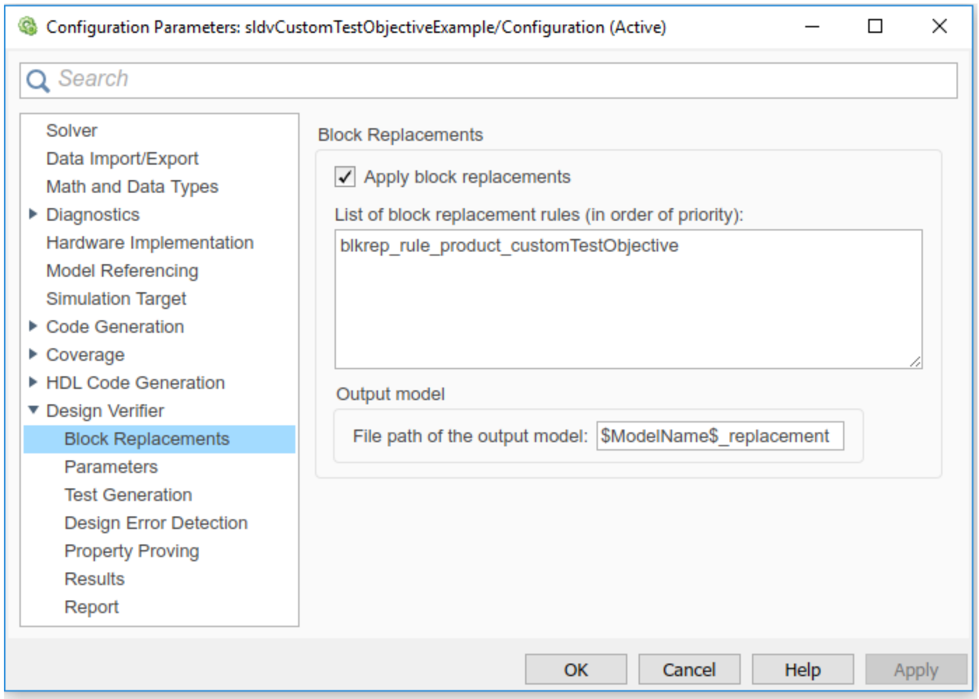
Type	Prompt	Name
	%<MaskType>	DescGroupVar
A	%<MaskDescription>	DescTextVar
	Parameters	ParameterGroupVar
#1		isEnabledForDetectability
#2		detectBlock

Property editor

Properties	
Name	detectBlock
Value	Product_target
Prompt	
Type	edit
Attributes	
Evaluate	☐
Tunable	on
Read only	☐
Hidden	☐
Never save	☐
Constraint	None
Dialog	
Enable	☑
Visible	☐
Callback	
Tooltip	
Layout	
Item location	New row
Prompt location	Left
Horizontal Stretch	☑

Drag or **Click** items in left palette to add to dialog.
Use **Delete** key to remove items from dialog.
Tutorial:- Creating a Mask: Parameters and Dialog Pane

Block replacement rule to replace the block under consideration with a masked subsystem: The `sldvCustomTestObjectiveExample` model is preconfigured with the block replacement options. The block replacement rule is defined in the blkrep_rule_product_customTestObjective file that replaces the Product block with the custProduct masked subsystem.



Block Replacements

☑ Apply block replacements

List of block replacement rules (in order of priority): blkrep_rule_product_customTestObjective

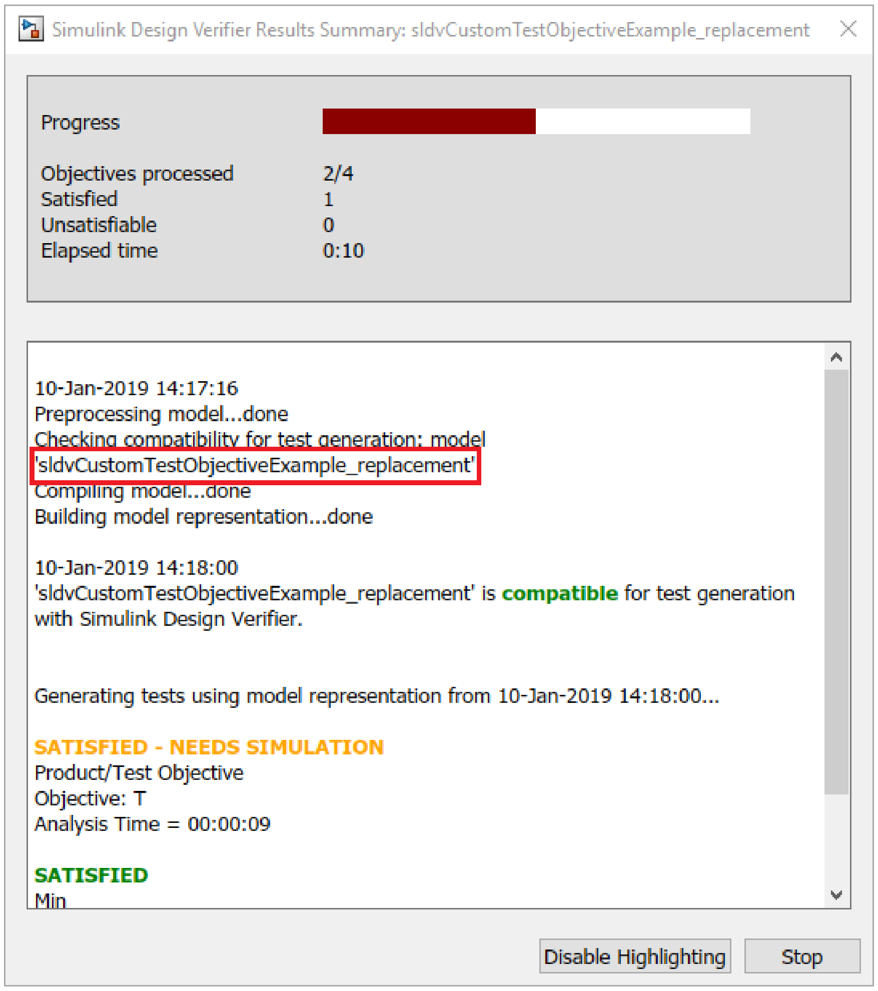
Output model

File path of the output model: $ModelName$_replacement

2 To configure the model for enhanced MCDC objectives, on the **Design Verifier** tab, click **Test Generation Settings**. In the Configuration Parameters dialog box, in **Design Verifier** > **Test Generation** pane, for **Model coverage objectives**, select `Enhanced MCDC`. Click **OK**.

3 To generate test cases, click **Generate Tests**.

The software analyzes the replacement model for test generation.

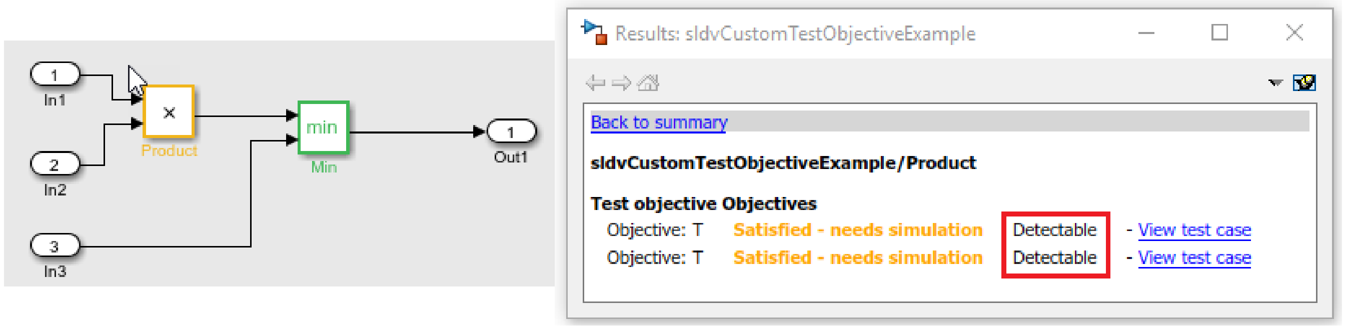

Simulink Design Verifier Results Summary: sldvCustomTestObjectiveExample_replacement

```
Progress

Objectives processed    2/4
Satisfied               1
Unsatisfiable           0
Elapsed time            0:10
```

```
10-Jan-2019 14:17:16
Preprocessing model...done
Checking compatibility for test generation: model
'sldvCustomTestObjectiveExample_replacement'
Compiling model...done
Building model representation...done

10-Jan-2019 14:18:00
'sldvCustomTestObjectiveExample_replacement' is compatible for test generation
with Simulink Design Verifier.


Generating tests using model representation from 10-Jan-2019 14:18:00...

SATISFIED - NEEDS SIMULATION
Product/Test Objective
Objective: T
Analysis Time = 00:00:09

SATISFIED
Min
```

4 Click **Highlight analysis results on model**.

To analyze the detectability of the Product block, click the Product block.



sldvCustomTestObjectiveExample/Product

Test objective Objectives

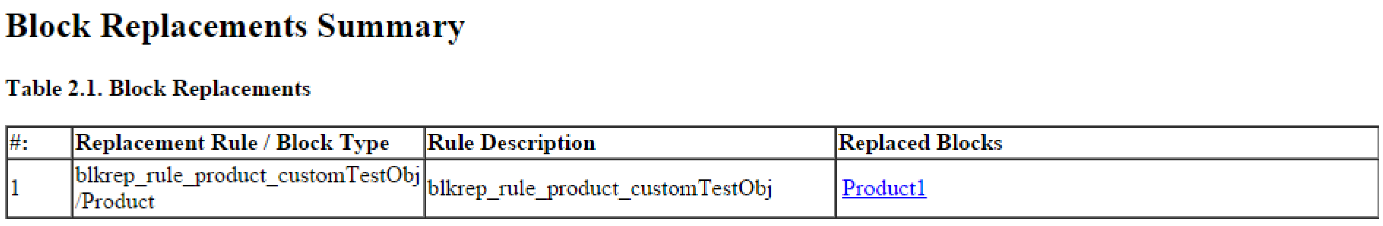
Objective: T	Satisfied - needs simulation	Detectable	- View test case
Objective: T	Satisfied - needs simulation	Detectable	- View test case

The results show that the test objectives of the Product block are detectable. The test case is generated.

Note The software is unable to confirm the objectives status through validation results for the objectives introduced by block replacement. Therefore, the test objective status is reported as satisfied - needs simulation. For more information on validation, see “Reporting Approximations Through Validation Results” on page 2-22.

---

5. Click **View test case**. The harness model opens and the Signal Builder block displays the test case.
6. To view the detailed analysis report, click **HTML** in the Results Summary. The Block Replacement Summary provides details about the replaced blocks.

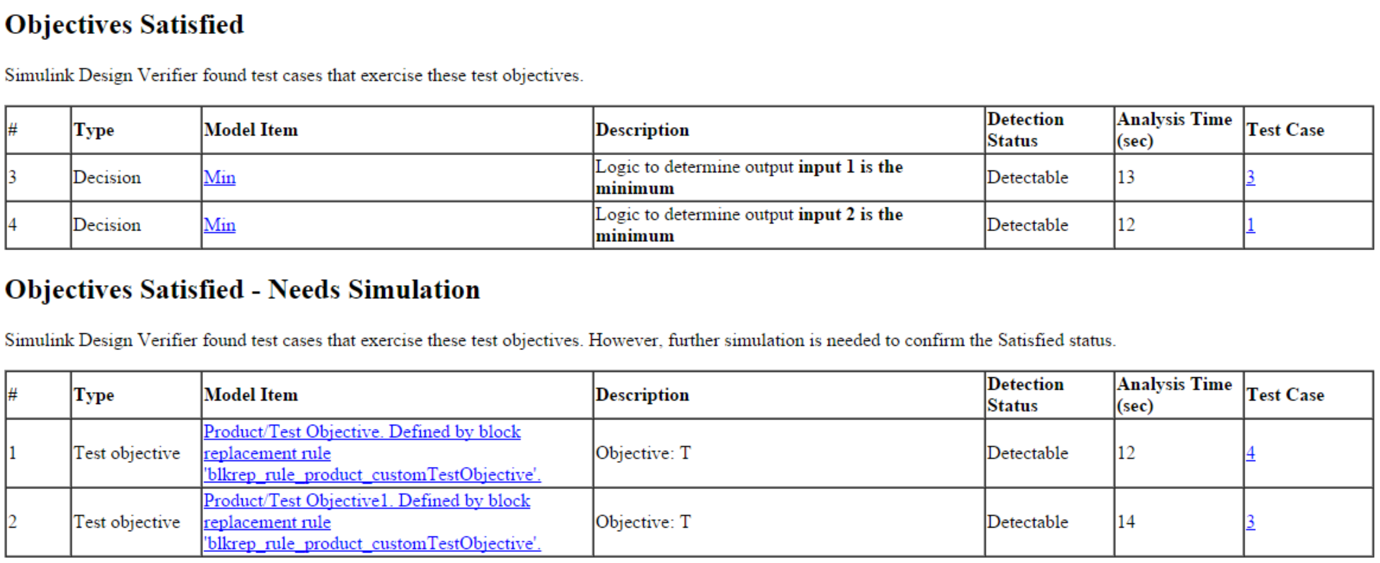


Block Replacements Summary

Table 2.1. Block Replacements

#:	Replacement Rule / Block Type	Rule Description	Replaced Blocks
1	blkrep_rule_product_customTestObj /Product	blkrep_rule_product_customTestObj	Product1

The Test Objectives Status section lists the objectives. The test objective that is detectable at the detection site is summarized in the table.



Objectives Satisfied

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

#	Type	Model Item	Description	Detection Status	Analysis Time (sec)	Test Case
3	Decision	Min	Logic to determine output **input 1 is the minimum**	Detectable	13	3
4	Decision	Min	Logic to determine output **input 2 is the minimum**	Detectable	12	1

Objectives Satisfied - Needs Simulation

Simulink Design Verifier found test cases that exercise these test objectives. However, further simulation is needed to confirm the Satisfied status.

#	Type	Model Item	Description	Detection Status	Analysis Time (sec)	Test Case
1	Test objective	Product/Test Objective. Defined by block replacement rule 'blkrep_rule_product_customTestObjective'.	Objective: T	Detectable	12	4
2	Test objective	Product/Test Objective1. Defined by block replacement rule 'blkrep_rule_product_customTestObjective'.	Objective: T	Detectable	14	3

See Also

More About

- “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-35
- “Block Replacements for Unsupported Blocks” on page 4-8

What is a Specification Model?

When systematically verifying a design model against requirements, the development process involves generating test cases for each requirement. These tests validate the design model which is used for the production code generation and helps to gain the confidence that the design model satisfies requirements. *Specification model* is an executable entity that allows you to perform requirements-based testing by leveraging Simulink Design Verifier capabilities.

If you have a set of requirements that are written in natural language text, you can convert them into formal (executable) specifications using Simulink. These then become a *specification model*. Unlike the design model, a specification model only specifies *What is to be done and not How it is to be done*. It captures the requirements at a higher level and hides the details at lower level.

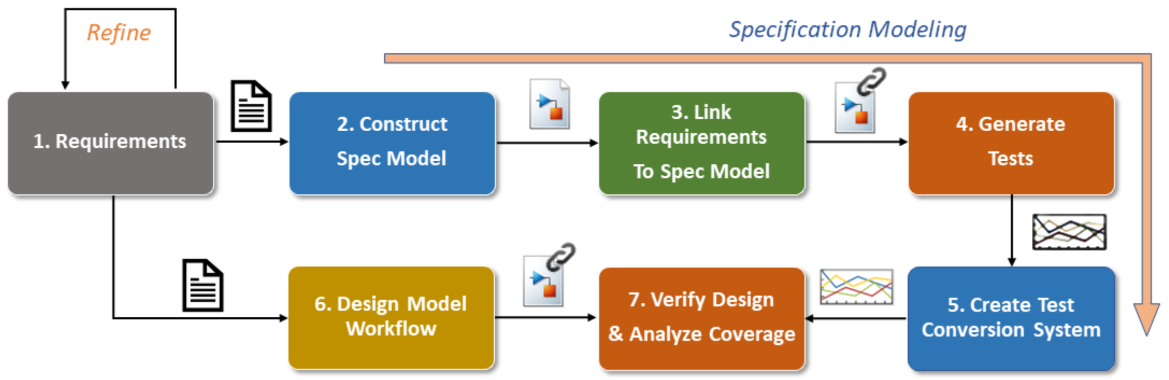
The advantages of using a specification model are:

- It validates the set of requirements in a systematic manner.
- It automates requirements-based testing.
- It helps to identify the missing requirements, design errors, or inconsistencies in your requirements and design model early in the development phase.

Using Specification Models for Requirements-based Testing

For requirements-based testing, test cases generated from the specification model are used to verify the design model against requirements. Follow these steps for the requirements-based testing using a specification model:

1. **Author requirements in the Requirement Editor.** Write your requirements in a natural language text that describes the behavior of the system under design.
2. **Construct a specification model.** Design the specification model as an executable representation of the requirements. This activity may reveal issues that lead to the refinement of the requirements.
3. **Link requirements.** Link the individual requirements or subrequirements to the parts of the specification model.
4. **Generate tests for the specification model.** Generate at least one test per requirement that demonstrates its conformance to that requirement.
5. **Create a test conversion subsystem.** The specification and design models may not use the same input-output interface. Convert the test cases that were generated in step 4 by using a test conversion system.
6. **Develop the design model.** Develop the design model independently by using the requirements document. Link the requirements to the design model.
7. **Verify the design and analyze the coverage.** Run the tests generated in step 5 on the design model that was developed in step 6 and verify whether the results agree with the specification model and requirements. Generate a design model coverage report to identify the missing coverage and refine the requirements, if required.



Specification Model Workflow

Creating a Specification Model

Consider the autopilot controller model described in “Use a Specification Model for Requirements-Based Testing” on page 7-56. For this demonstration, requirements consist of logical and temporal open-loop conditions.

Follow these steps to create a specification model for the requirements:

- “Identify the Specification Model Interface” on page 7-50
- “Use High-Level Representation for Signal Values” on page 7-51
- “Identify High-Level Operating Modes” on page 7-51
- “Identify Preconditions, Effects, and Expected Output for Each Requirement” on page 7-52
- “Final Specification Model” on page 7-53

Identify the Specification Model Interface

List the input and output signals for the specification model that are related to the requirements. You may ignore the signals that are not related to the requirements at hand.

These are the input signals for the autopilot controller that are based on the requirements:

1. **Autopilot Engage Switch**: Enable/Disable the autopilot controller
2. **Heading Engage Switch**: When engaged, enables the `HDG_HOLD_MODE`. Otherwise, `ROLL_HOLD_MODE` is active

3 **Roll Reference Target Turn Knob**: A dial that feeds desired roll angle value to the autopilot controller
4 **Heading Reference Turn Knob**: Gives the set-point value for heading mode
5 **Aircraft Roll Angle**: The current instantaneous roll angle of aircraft

These are the output signals:

1 **Aileron Command**: The output to the aileron actuator
2 **Roll Ref Command**: The output on the display window indicating the set-point value for the aileron actuator

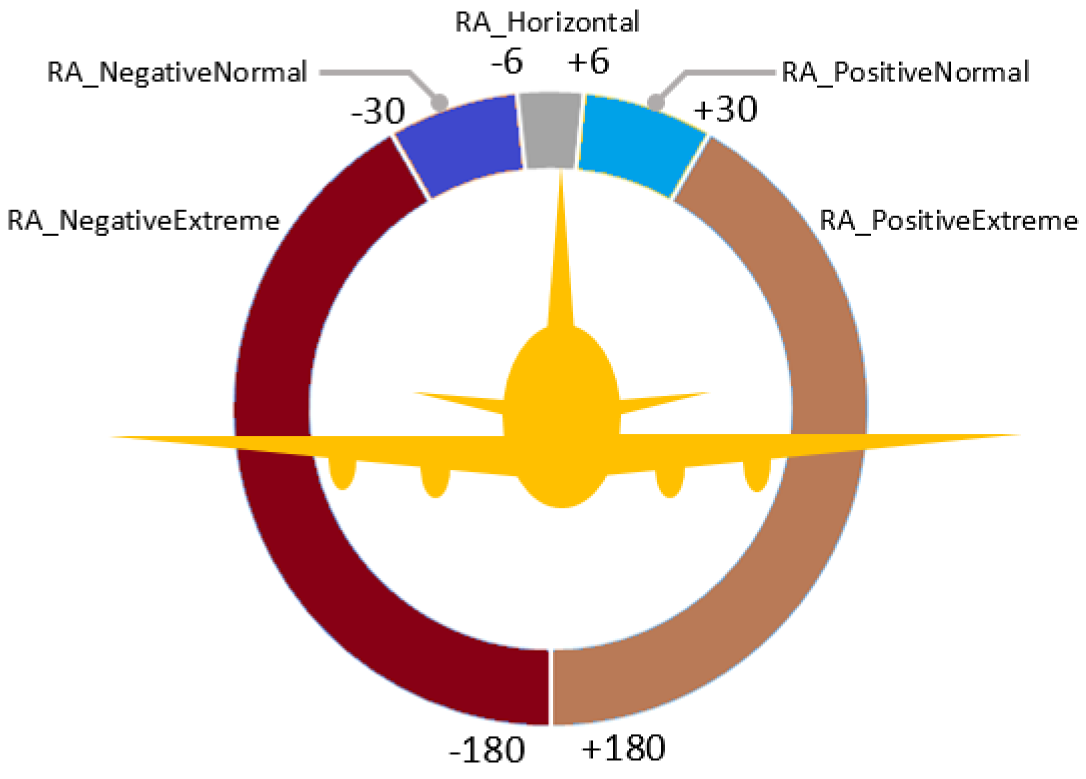
Use High-Level Representation for Signal Values

Some signals are represented at the higher-level in specifications. It is recommended to represent signals using high-level representation like ranges in the specification model.

Consider the input signal Aircraft Roll Angle, which represents the current roll angle of an aircraft and takes any value in the interval -180 to +180 degrees.

The requirements describe the behavior of the autopilot controller in terms of zones. These zones are modelled using the enumeration `Range`.

These five zones are shown in the following figure.



Identify High-Level Operating Modes

The requirements specify the high-level AP Controller modes and their active conditions as follows:

Autopilot Mode	Autopilot Engage Switch	Heading Engage Switch
`OFF`	OFF	Don't care
`ROLL_HOLD_MODE`	ON	OFF
`HDG_HOLD_MODE`	ON	ON

Identify Preconditions, Effects, and Expected Output for Each Requirement

Consider the following requirement:

"Whenever cockpit turn knob **Roll Reference Target Turn Knob** (`Roll_Ref_TK`) is commanding in normal range (between [-30, -3] or [+3, +30] degrees), **Roll Reference** (`Roll_Ref_Cmd`) shall be set to `Roll_Ref_TK`."

Identify the precondition and effects

The above requirement consists of two clauses:

1. **Precondition:** `Roll_Ref_TK` is either in the negative normal range, [-30, -3], or in the positive normal range, [+3, +30].

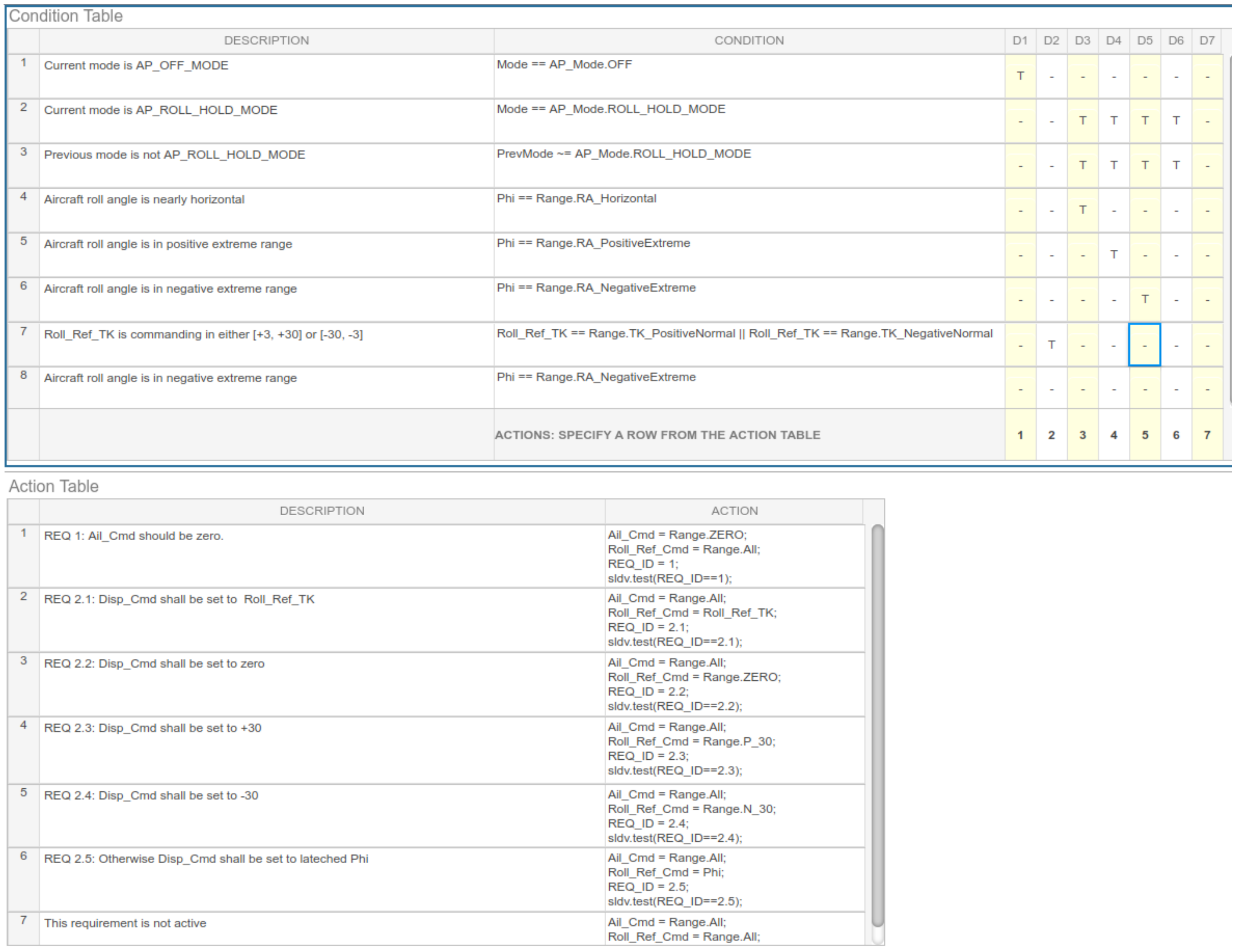
   This precondition is a simple logical-OR expression, so truth tables are used to express the logical pre-conditions.
2. **Effect:** Set `Roll_Ref_Cmd` to `Roll_Ref_TK`.The effect clause specifies the output signal values in expected range.

The precondition clause of a requirement determines when it becomes active, while the effect clause determines what a requirement will do after it becomes active.

The above requirement has no effect on the **Aileron Command** `Ail_Cmd` output, so it is considered as `Range.All` which denotes the set of all possible values.

Create Truth Table for Requirements

- Encode the precondition clause of the requirement into the Condition Table section of the truth table and the effect clause into the Action Table section.
- To track each requirement individually, set the local variable REQ_ID to the corresponding requirement ID 2.1.
- Add a Simulink Design Verifier objective in Action Table by using the statement `sldv.test` (REQ_ID==2.1). Simulink Design Verifier finds a test when REQ_ID 2.1 is satisfied.



Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4	D5	D6	D7
1	Current mode is AP_OFF_MODE	Mode == AP_Mode.OFF	T	-	-	-	-	-	-
2	Current mode is AP_ROLL_HOLD_MODE	Mode == AP_Mode.ROLL_HOLD_MODE	-	-	T	T	T	T	-
3	Previous mode is not AP_ROLL_HOLD_MODE	PrevMode ~= AP_Mode.ROLL_HOLD_MODE	-	-	T	T	T	T	-
4	Aircraft roll angle is nearly horizontal	Phi == Range.RA_Horizontal	-	-	T	-	-	-	-
5	Aircraft roll angle is in positive extreme range	Phi == Range.RA_PositiveExtreme	-	-	-	T	-	-	-
6	Aircraft roll angle is in negative extreme range	Phi == Range.RA_NegativeExtreme	-	-	-	-	T	-	-
7	Roll_Ref_TK is commanding in either [+3, +30] or [-30, -3]	Roll_Ref_TK == Range.TK_PositiveNormal || Roll_Ref_TK == Range.TK_NegativeNormal	-	T	-	-	-	-	-
8	Aircraft roll angle is in negative extreme range	Phi == Range.RA_NegativeExtreme	-	-	-	-	-	-	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	1	2	3	4	5	6	7

Action Table

	DESCRIPTION	ACTION

1	REQ 1: Ail_Cmd should be zero.	Ail_Cmd = Range.ZERO;
		Roll_Ref_Cmd = Range.All;
		REQ_ID = 1;
		sldv.test(REQ_ID==1);

2	REQ 2.1: Disp_Cmd shall be set to Roll_Ref_TK	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Roll_Ref_TK;
		REQ_ID = 2.1;
		sldv.test(REQ_ID==2.1);

3	REQ 2.2: Disp_Cmd shall be set to zero	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.ZERO;
		REQ_ID = 2.2;
		sldv.test(REQ_ID==2.2);

4	REQ 2.3: Disp_Cmd shall be set to +30	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.P_30;
		REQ_ID = 2.3;
		sldv.test(REQ_ID==2.3);

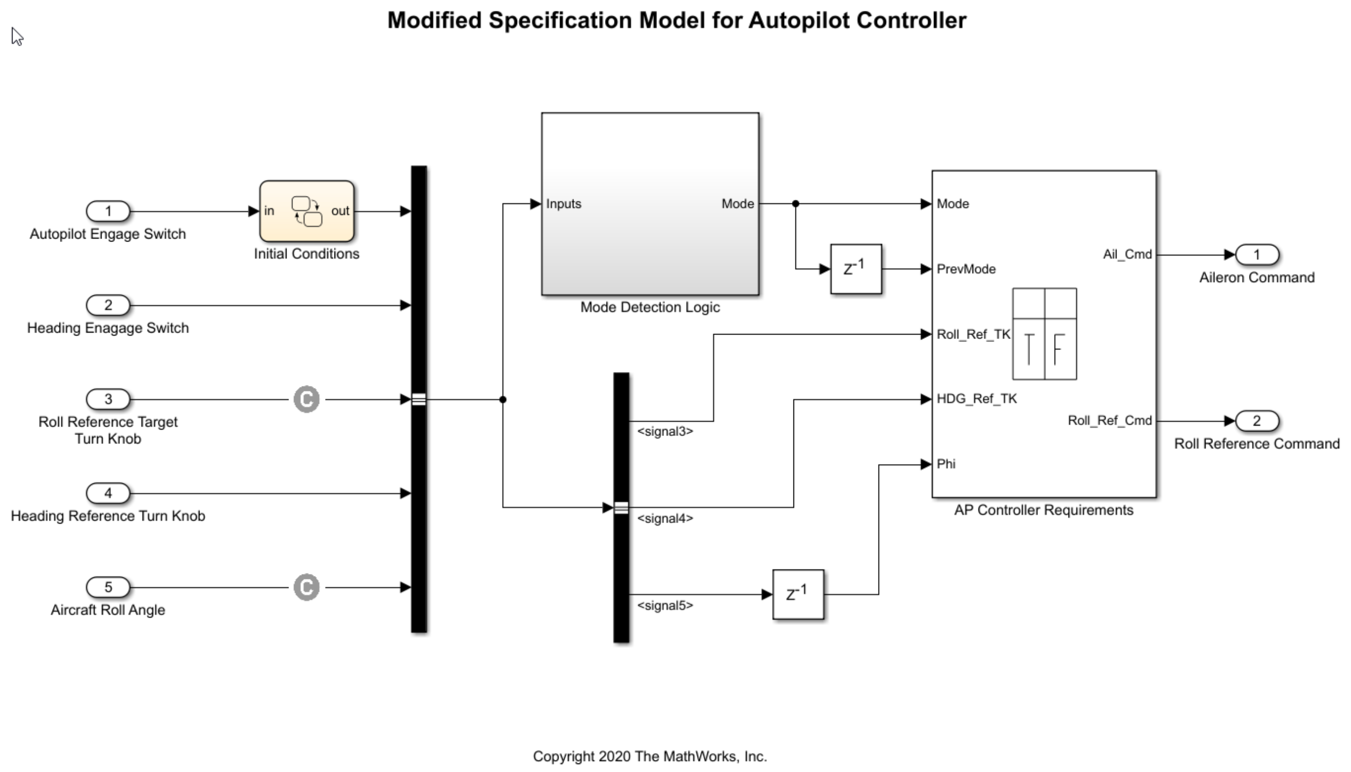
5	REQ 2.4: Disp_Cmd shall be set to -30	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.N_30;
		REQ_ID = 2.4;
		sldv.test(REQ_ID==2.4);

6	REQ 2.5: Otherwise Disp_Cmd shall be set to latched Phi	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Phi;
		REQ_ID = 2.5;
		sldv.test(REQ_ID==2.5);

7	This requirement is not active	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.All;

Final Specification Model

The final specification model obtained by using above workflow looks like this:



See Also

"Use a Specification Model for Requirements-Based Testing" on page 7-56

Test Generation Examples

These test generation examples help you understand and use the test generation capabilities.

Test Generation Capabilities	Related Examples
Generate tests for model coverage analysis	"Cruise Control Test Generation" on page 7-67
	"Fuel Rate Controller Logic" on page 7-68
	"Flip Flop Test Generation" on page 7-63
	"Model Coverage Test Generation" on page 7-64
Functional Requirements Testing	"Test Condition Block" on page 7-66
	"Test Objective Block" on page 7-65
Generate tests for code coverage analysis	"Configuring S-Function for Test Case Generation" on page 7-89
	"Code Coverage Test Generation" on page 7-92
	"Test Generation on Model with C Caller Block" on page 7-104
	"Test Generation for Custom Code in a Stateflow Chart" on page 7-106
Extend existing test cases	"Defining and Extending Existing Tests Cases" on page 7-74
	"Extend an Existing Test Suite" on page 7-69
	"Creating and Executing Test Cases" on page 7-84
	"Extend Existing Test Cases After Applying Parameter Configurations" on page 5-35
Achieve missing coverage	"Achieve Missing Coverage in Referenced Model" on page 9-3
	"Achieve Missing Coverage in Closed-Loop Simulation Model" on page 9-11
	"Using Existing Coverage Data During Subsystem Analysis" on page 7-79
Integrate with other products	"Export Test Cases to Simulink Test" on page 13-26

Use a Specification Model for Requirements-Based Testing

This example shows how to use a specification model to perform requirements-based testing. In this example, you follow a systematic approach to verify your design model against requirements. For a detailed description of specification model, see “What is a Specification Model?” on page 7-49.

Step 1: Author Requirements in the Requirements Editor

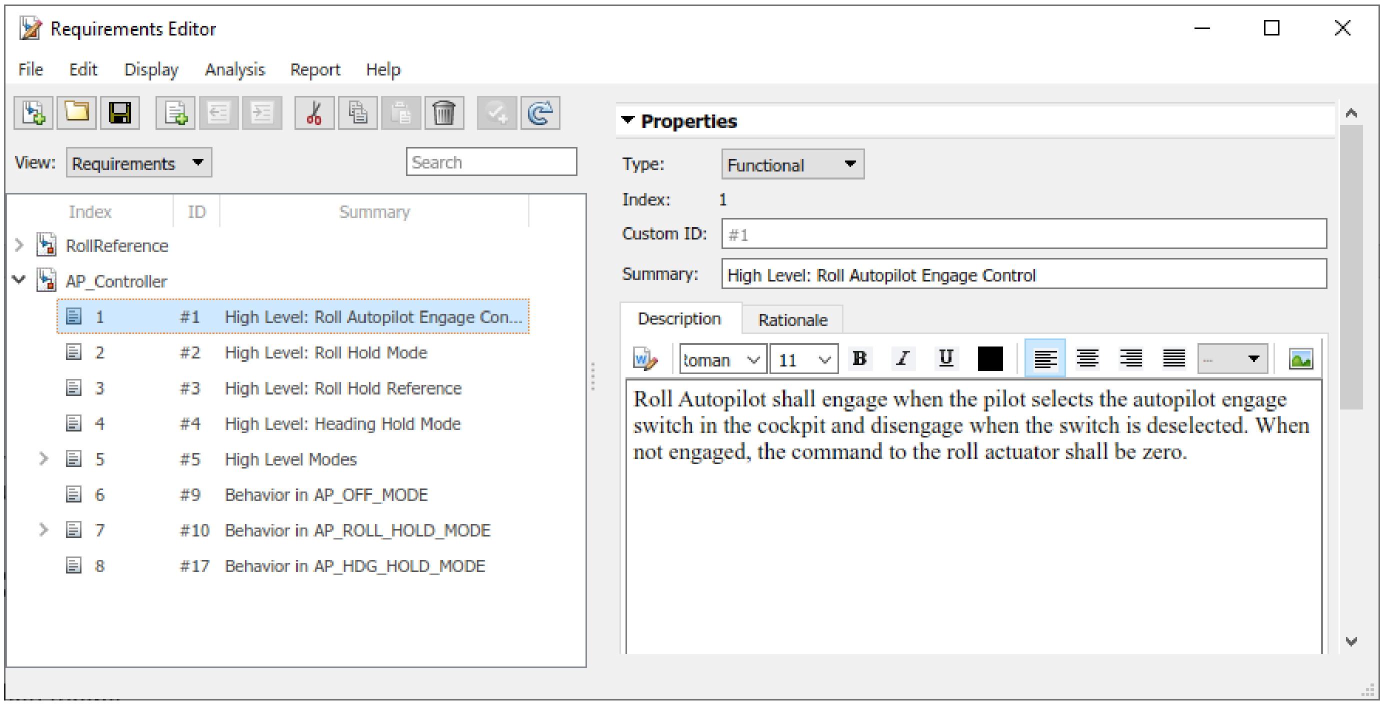
This example uses a roll autopilot controller, `RollAutopilotMdlRef` which is a design model that controls the roll angle of an aircraft. The roll autopilot controller operates in two high-level modes:

1. **Roll hold mode**: This mode either maintains the current roll angle of the aircraft or changes it according to user-specified angle.

2. **Heading hold mode**: This mode either maintains the current heading or rolls the aircraft to achieve the user-specified heading value. For detailed information on the Roll Autopilot Controller system,see “Requirements-Based Testing for Model Development” (Simulink Test).

For the autopilot controller, the requirements describe the system interfaces, high-level system modes, and the expected behavior of the controller. These requirements are authored in the Requirements Editor and saved in the `AP_Controller.slreqx` file. For more information on the Requirements Editor, see “Work with Requirements in the Simulink Editor” (Simulink Requirements). To view the requirements, open the Requirements Editor by entering,

```
slreq.open('AP_Controller');
```



View: Requirements

Index	ID	Summary
RollReference		
AP_Controller		
1	#1	High Level: Roll Autopilot Engage Con...
2	#2	High Level: Roll Hold Mode
3	#3	High Level: Roll Hold Reference
4	#4	High Level: Heading Hold Mode
5	#5	High Level Modes
6	#9	Behavior in AP_OFF_MODE
7	#10	Behavior in AP_ROLL_HOLD_MODE
8	#17	Behavior in AP_HDG_HOLD_MODE

Type: Functional

Index: 1

Custom ID: #1

Summary: High Level: Roll Autopilot Engage Control

Roll Autopilot shall engage when the pilot selects the autopilot engage switch in the cockpit and disengage when the switch is deselected. When not engaged, the command to the roll actuator shall be zero.

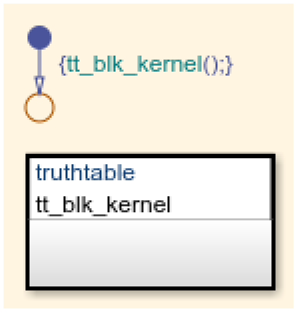
The Requirements Editor displays the high-level requirements for the Roll Hold and Heading Hold modes. When you click on each requirement, the tab lists the details for the requirement.

Step 2: Create a Specification Model

When you create specification model, you need to consider several factors such as the type of requirements, choice of model blocks, and level of abstraction.Follow the guidelines described in “Creating a Specification Model” on page 7-50.

Open the `sldvexSpecPartial` specification model that covers the set of autopilot requirements:

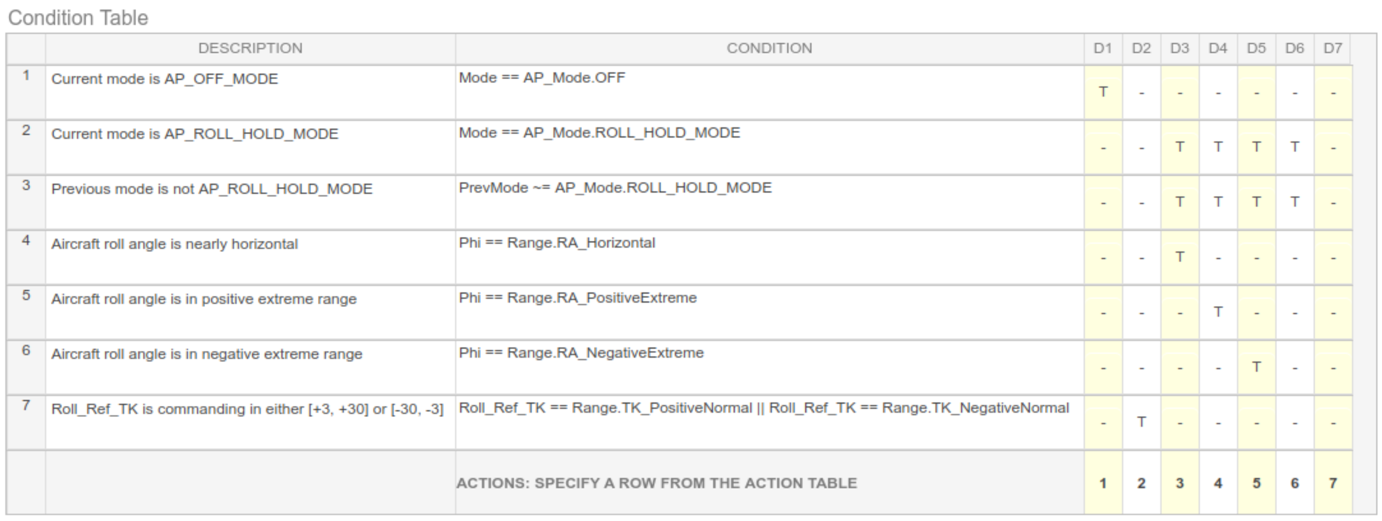
```
spec_model = 'sldvexSpecPartial';
open_system(spec_model);
```



The `sldvexSpecPartial` model consists of input and output interfaces. The truth table captures the requirements.

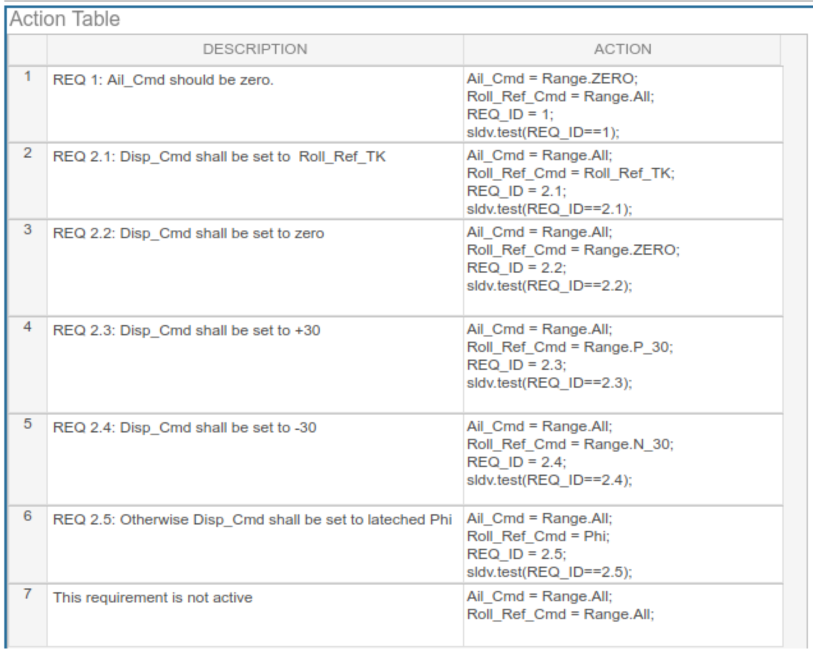
To open the truth table,enter:

```
open_system('sldvexSpecPartial/AP Controller Requirements');
```



Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4	D5	D6	D7
1	Current mode is AP_OFF_MODE	Mode == AP_Mode.OFF	T	-	-	-	-	-	-
2	Current mode is AP_ROLL_HOLD_MODE	Mode == AP_Mode.ROLL_HOLD_MODE	-	-	T	T	T	T	-
3	Previous mode is not AP_ROLL_HOLD_MODE	PrevMode ~= AP_Mode.ROLL_HOLD_MODE	-	-	T	T	T	T	-
4	Aircraft roll angle is nearly horizontal	Phi == Range.RA_Horizontal	-	-	T	-	-	-	-
5	Aircraft roll angle is in positive extreme range	Phi == Range.RA_PositiveExtreme	-	-	-	T	-	-	-
6	Aircraft roll angle is in negative extreme range	Phi == Range.RA_NegativeExtreme	-	-	-	-	T	-	-
7	Roll_Ref_TK is commanding in either [+3, +30] or [-30, -3]	Roll_Ref_TK == Range.TK_PositiveNormal || Roll_Ref_TK == Range.TK_NegativeNormal	-	T	-	-	-	-	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	1	2	3	4	5	6	7



Action Table

	DESCRIPTION	ACTION

1	REQ 1: Ail_Cmd should be zero.	Ail_Cmd = Range.ZERO;
		Roll_Ref_Cmd = Range.All;
		REQ_ID = 1;
		sldv.test(REQ_ID==1);

2	REQ 2.1: Disp_Cmd shall be set to Roll_Ref_TK	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Roll_Ref_TK;
		REQ_ID = 2.1;
		sldv.test(REQ_ID==2.1);

3	REQ 2.2: Disp_Cmd shall be set to zero	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.ZERO;
		REQ_ID = 2.2;
		sldv.test(REQ_ID==2.2);

4	REQ 2.3: Disp_Cmd shall be set to +30	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.P_30;
		REQ_ID = 2.3;
		sldv.test(REQ_ID==2.3);

5	REQ 2.4: Disp_Cmd shall be set to -30	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.N_30;
		REQ_ID = 2.4;
		sldv.test(REQ_ID==2.4);

6	REQ 2.5: Otherwise Disp_Cmd shall be set to latched Phi	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Phi;
		REQ_ID = 2.5;
		sldv.test(REQ_ID==2.5);

7	This requirement is not active	Ail_Cmd = Range.All;
		Roll_Ref_Cmd = Range.All;

Step 3. Link Requirements to the Specification Model

Perform these steps to link the requirements to the specification model.

1. Right-click the truth table named `AP Controller Requirements` in the specification model. In the context menu, click **Requirements** > **Select for linking with Simulink**.

2. Open the requirements in the Requirements Editor. Right-click on the requirement you want to link to the truth table and click **Link from AP Controller Requirements (Truth Table)**. If you have multiple truth tables, each specifying a group of requirements, link them as well.

Step 4: Generate Test Cases for the Specification Model

Use `sldvoptions` to generate test cases for the specification model. Each requirement has been associated with an individual test generation objective using `sldv.test()`.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'None';
[~, files] = sldvrun(spec_model,opts,true);
```

After the analysis completes, the Results Summary window displays that six out of six of the objectives are satisfied.

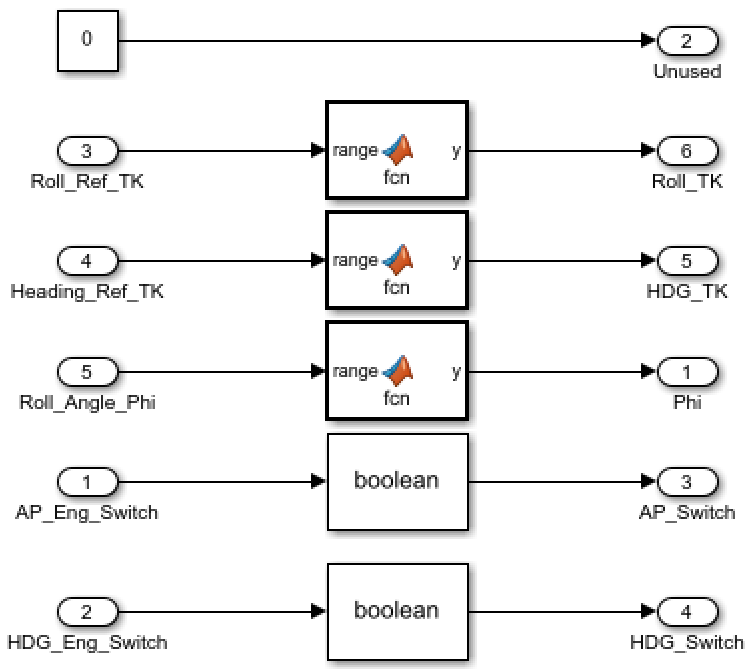
Step 5: Create a Test Conversion System to Run Tests on the Design Model

The autopilot controller specification model and the design model have different interfaces which means that the tests generated in step 4 are not supported for performing simulation on the design model.

For example, **Aircraft Roll Angle** is of the enumeration range type in the specification, but is of double type in the design model.

During the test conversion process, if a signal value, such as RA_Horizontal, is a range, you can choose any value that falls in that range. Various heuristics, such as midpoint (where you can choose midpoint of the range), boundary value (where you can choose lower or upper bound of range), or even random strategy (where you choose a random value in the range) may be used. For the autopilot controller, the subsystem `sldvexDesignHarness/Test Conversion` implements the midpoint strategy in the harness model as shown below:

```
design_model = 'sldvexDesignHarness';
load_system(design_model);
open_system('sldvexDesignHarness/Test Conversion');
```



Step 6: Simulate Test Cases on the Design Model and Identify Missing Requirements

The design model is developed independently by using the requirements document. To verify the design, create a harness model that contains these four subsystems:

(i) The specification model.

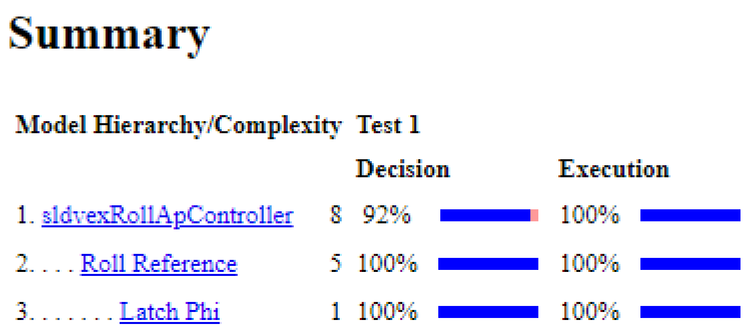
(ii) The design model.

(iii) The test conversion subsystem described in step 5.

(iv) A runtime verification block. This block checks whether the design signal value is in the range specified by the specification model.

Run the tests from step 5 on the design model by using `sldvruntest` and generate a model coverage report.

```
cvopts = sldvruntestopts;
cvopts.coverageEnabled =  true;
[~, initCov] = sldvruntest(design_model,files.DataFile,cvopts);
cvhtml('InitialCov',initCov,'-sRT=0');
```



Summary

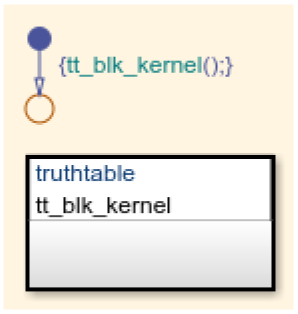
Model Hierarchy/Complexity		Test 1	
		Decision	Execution
1. sldvexRollApController	8	92%	100%
2. . . . Roll Reference	5	100%	100%
3. . . . . . . Latch Phi	1	100%	100%

The analysis results report that full coverage is not achieved for the `roll_ap_mod` Subsystem coverage is achieved for the design model.

Step 7: Update the Specification Model by Adding Missing Requirements

Add the requirement is added to the `sldvexSpecFull` specification model for analysis.

```
spec_model = 'sldvexSpecFull';
open_system(spec_model);
```



(a) Generate Test Cases on the Updated Specification Model

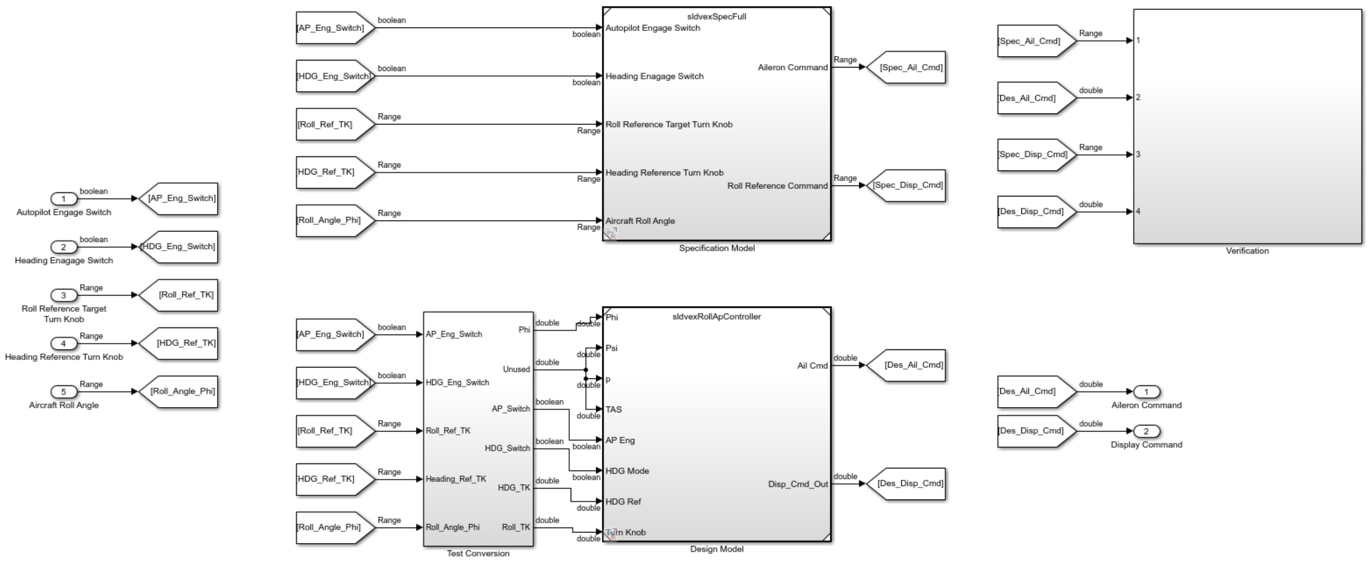
Use `sldvoptions` to generate test cases.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'None';
[~, files] = sldvrun(spec_model,opts,true);
```

(b) Simulate Test Cases on the Design Model and Generate a Coverage Report

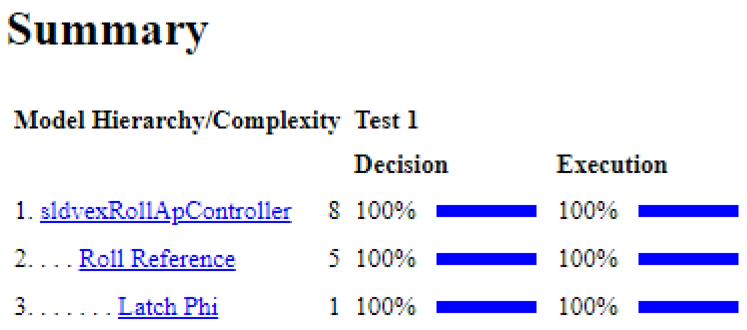
Open the `sldvexDesignHarness` model that contains the design model, specification model, and test conversion subsystem.

```
design_model = 'sldvexDesignHarness';
open_system(design_model);
```



Simulate the test cases by using `sldvruntest` and generate a model coverage report.

```
cvopts = sldvruntestopts;
cvopts.coverageEnabled =  true;
[~, FinalCov] = sldvruntest(design_model,files.DataFile,cvopts);
cvhtml('FinalCov', FinalCov, '-sRT=0');
```


Summary

Model Hierarchy/Complexity		Test 1 Decision		Execution	
1. sldvexRollApController	8	100%	▬	100%	▬
2. . . . Roll Reference	5	100%	▬	100%	▬
3. . . . . . . Latch Phi	1	100%	▬	100%	▬

The coverage report shows that full coverage is achieved of the design model.

```
bdclose('all');
slreq.clear;
```

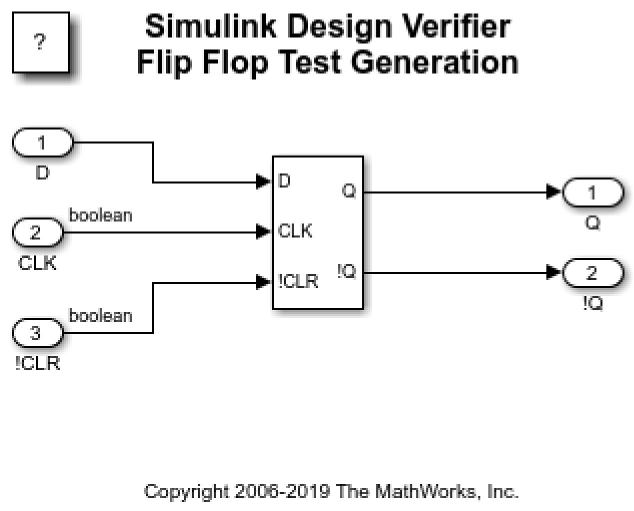
Related Topics

- “Using Specification Models for Requirements-based Testing” on page 7-49

Flip Flop Test Generation

This example shows how to generate test cases that achieve complete model coverage for a flip-flop. The outcome of each model coverage point in this example model is a test objective. If you configure Simulink Design Verifier to generate the fewest test cases, it will satisfy as many objectives as possible in each test case.

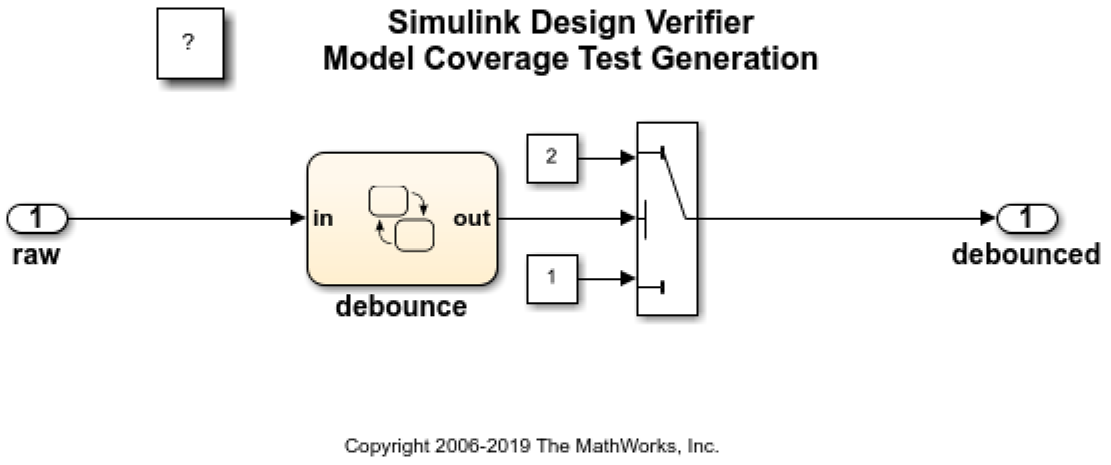
```
open_system('sldvdemo_flipflop');
```



Model Coverage Test Generation

This example shows how to generate test cases that achieve complete model coverage for a debouncer. The outcome of each model coverage point in this example model is a test objective. If you configure Simulink Design Verifier to generate the fewest test cases, it will satisfy as many objectives as possible in each test case.

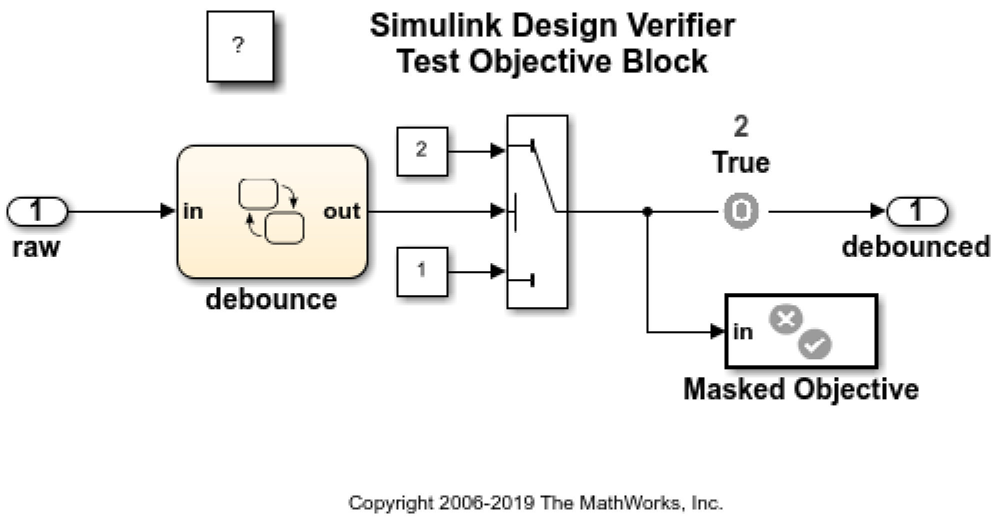
```
open_system('sldvdemo_debounce_modelcov');
```


?
Simulink Design Verifier
Model Coverage Test Generation
1
raw
in
out
debounce
2
1
1
debounced
Copyright 2006-2019 The MathWorks, Inc.

Test Objective Block

This example shows the use of two custom Test Objective blocks. The block "True" forces the output signal to be 2. The block "Edge" inside "Masked Objective" specifies that the output signal transition from 2 to 1.

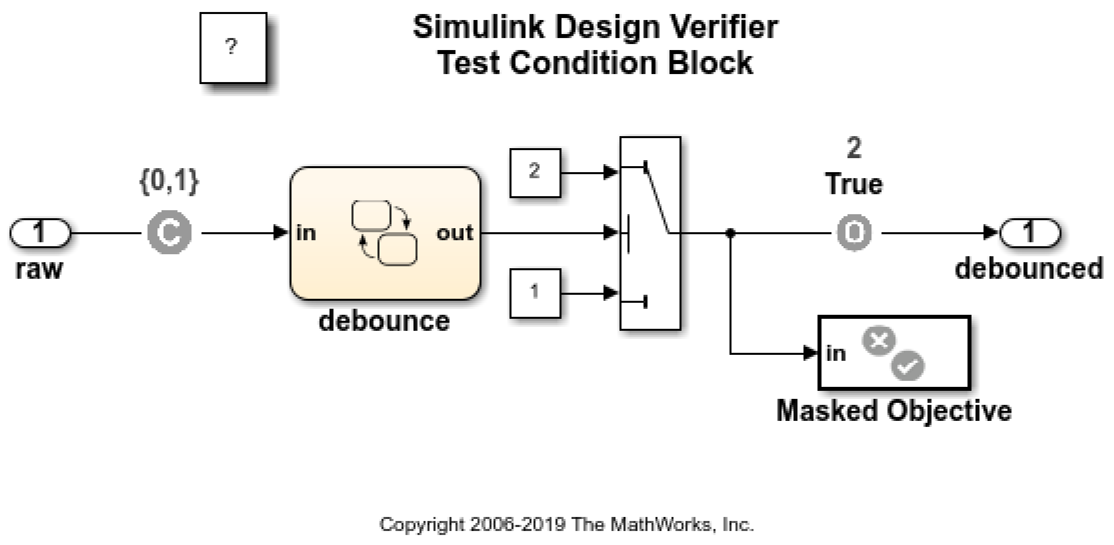
```
open_system('sldvdemo_debounce_testobjblks');
```



Test Condition Block

This example shows how to constrain input values. The Test Condition block forces the input value to be either 0 or 1.

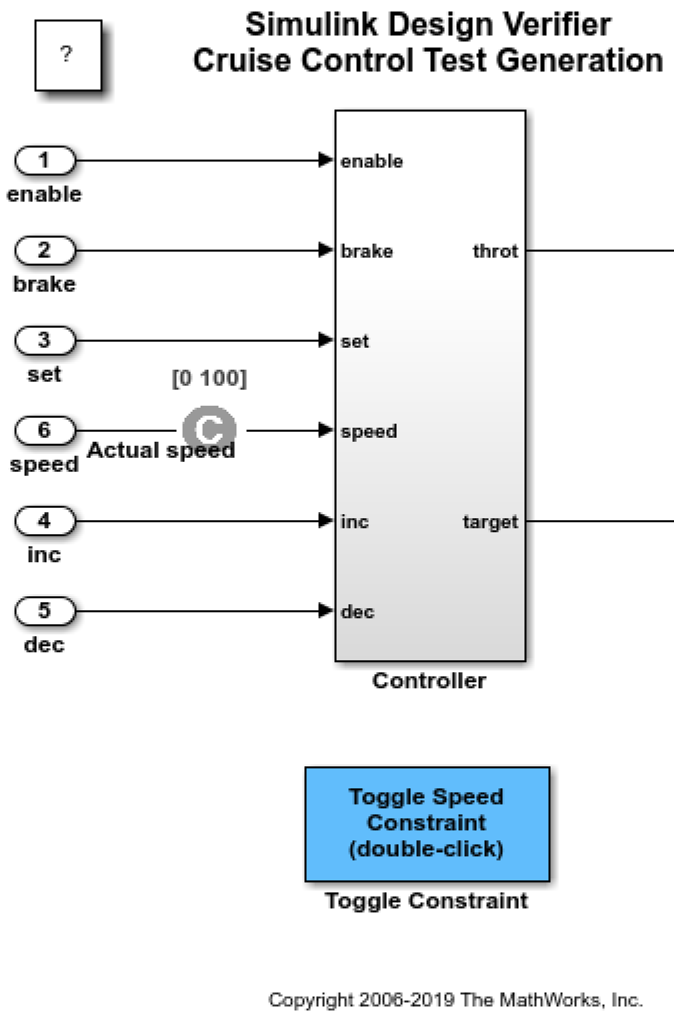
```
open_system('sldvdemo_debounce_testconblk');
```



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.

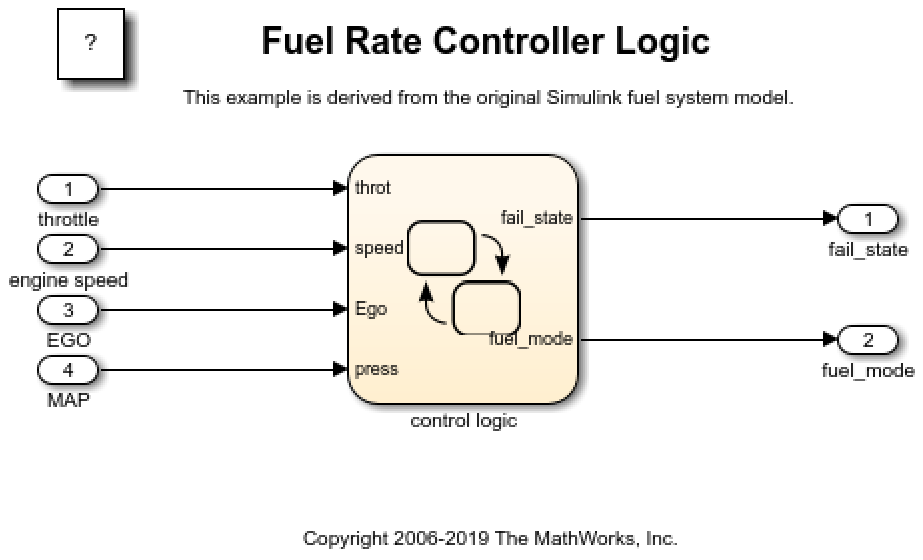
```
open_system('sldvdemo_cruise_control');
```



Fuel Rate Controller Logic

This example shows how to generate test cases that satisfy Decision, Condition, and MCDC coverage. Simulink Design Verifier automatically generates test data and proves properties of models. It produces sequences of input values that satisfy a testing criteria or demonstrate a counterexample of a proof. The configuration options associated with the model specify the objectives of the analysis. When you analyze the model, Simulink Design Verifier uses exhaustive searching techniques to generate input data. When successful, it generates test data and creates a new harness model containing a Signal Builder block with the data values that satisfy the analysis objectives. NOTE: The complexity of this model might prevent test generation from completing in the allotted time. You can stop test generation and generate partial results, or you can extend the time limit by editing the Simulink Design Verifier options.

```
open_system('sldvdemo_fuelsys_logic');
```


?
Fuel Rate Controller Logic
This example is derived from the original Simulink fuel system model.
1
throttle
2
engine speed
3
EGO
4
MAP
throt
speed
Ego
press
fail_state
fuel_mode
control logic
1
fail_state
2
fuel_mode
Copyright 2006-2019 The MathWorks, Inc.

Extend an Existing Test Suite

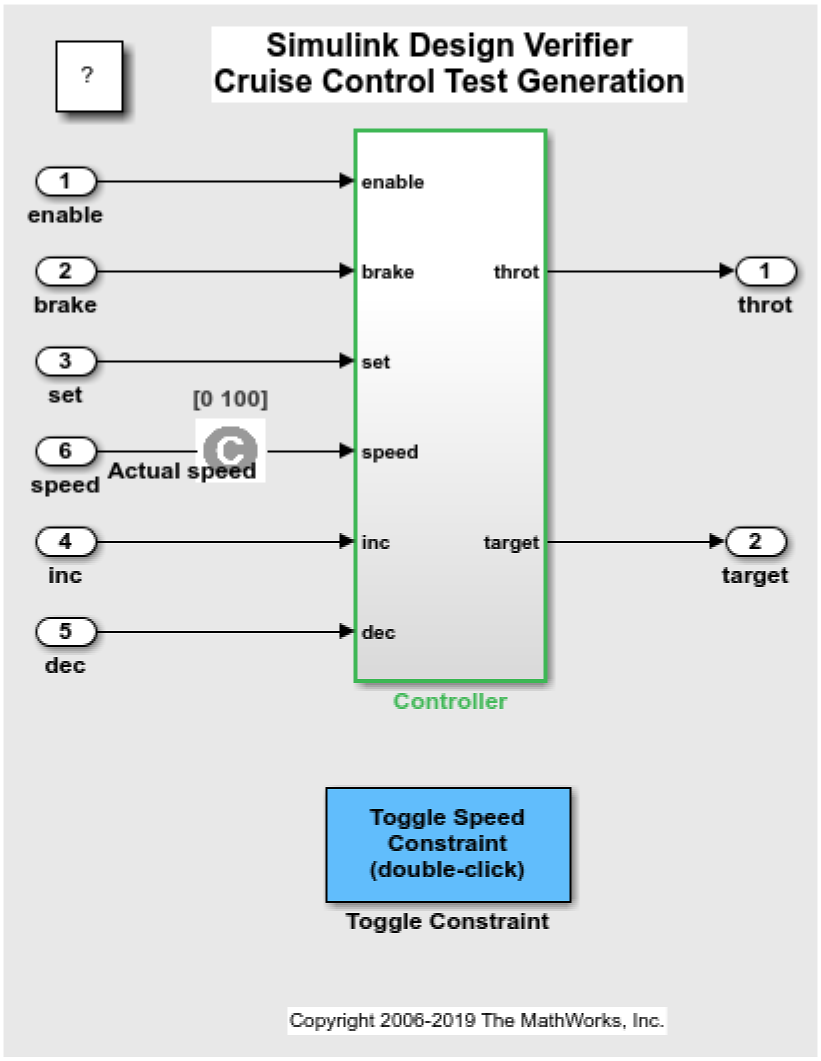
This example shows how to use Simulink® Design Verifier™ to extend an existing test suite to obtain missing model coverage.

You analyze an example model and generate test suite to achieve full coverage. Then, modify the model such that test cases no longer achieve full coverage. Finally, you analyze the modified model to obtain missing coverage by using Simulink® Design Verifier™.

Generate an Initial Test Suite

Analyze the `sldvdemo_cruise_control` model and generate a test suite that achieves full model coverage. To analyze the model to generate test cases that provide model coverage, use the `sldvrun` function. Set the design verification parameters with `sldvoptions`.

```
open_system 'sldvdemo_cruise_control';
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'MCDC';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts, true);
```



The test generation analysis result appears in the Simulink Design Verifier Results Summary window.

```
close_system('sldvdemo_cruise_control',0);
```

Verify Complete Coverage

The `sldvruntest` function simulates the model with the existing test suite. The `cvhtml` function produces a coverage report that indicates the initial coverage of the `sldvdemo_cruise_control` model.

```
open_system 'sldvdemo_cruise_control';
[ outData, initialCov ] = sldvruntest('sldvdemo_cruise_control', files.DataFile, [], true);
cvhtml('Initial coverage',initialCov);
close_system('sldvdemo_cruise_control',0);
```

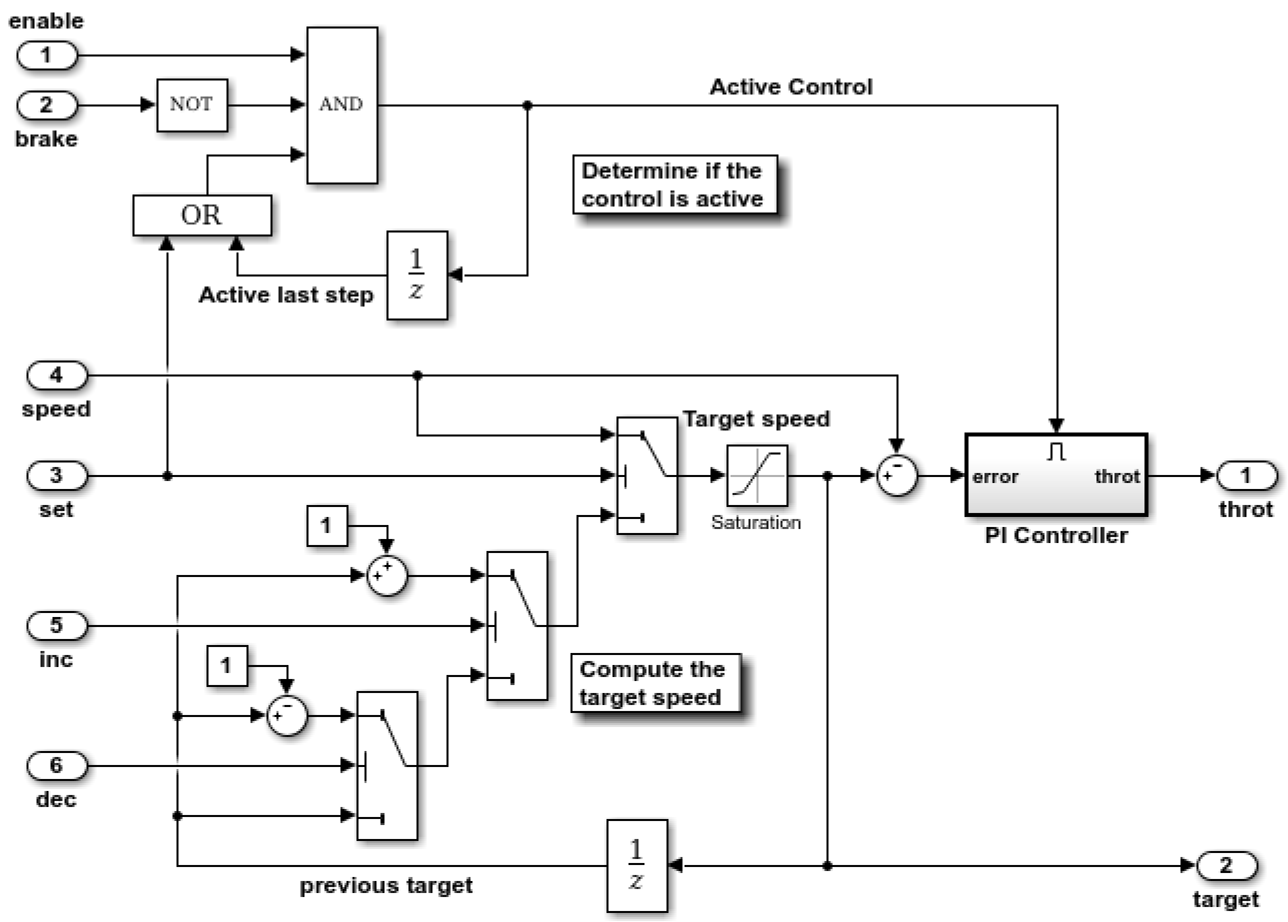
Summary

Model Hierarchy/Complexity		Test 1			
		Decision	Condition	MCDC	Execution
1. sldvdemo_cruise_control	8	100% ▬	100% ▬	100% ▬	100% ▬
2. . . . Controller	7	100% ▬	100% ▬	100% ▬	100% ▬
3. . . . . . . PI Controller	4	100% ▬	NA	NA	100% ▬

Modify the Model

Load the modified `sldvdemo_cruise_control_mod` model. The controller target speed value is limited to 70, by using a `Saturation` block.

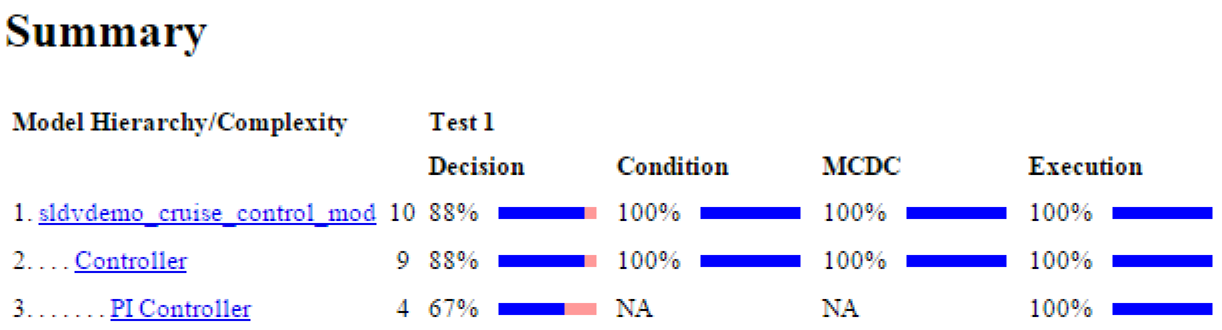
```
load_system 'sldvdemo_cruise_control_mod';
load_system 'sldvdemo_cruise_control_mod/Controller';
```



Measure the Coverage Achieved by the Existing Test Suite

The `sldvruntest` function simulates the modified `sldvdemo_cruise_control_mod` model with an existing test suite and inputs identical to `sldvdemo_cruise_control` model. The `cvhtml` function produces a coverage report that indicates the modified `sldvdemo_cruise_control_mod` model no longer achieves full coverage.

```
[ outData, startCov ] = sldvruntest('sldvdemo_cruise_control_mod', files.DataFile, [], true);
cvhtml('Coverage with the original testsuite',startCov);
```



Summary

Model Hierarchy/Complexity		Test 1			
		Decision	Condition	MCDC	Execution
1. sldvdemo_cruise_control_mod	10	88%	100%	100%	100%
2. . . . Controller	9	88%	100%	100%	100%
3. . . . . . . PI Controller	4	67%	NA	NA	100%

Extend an Existing Test Suite

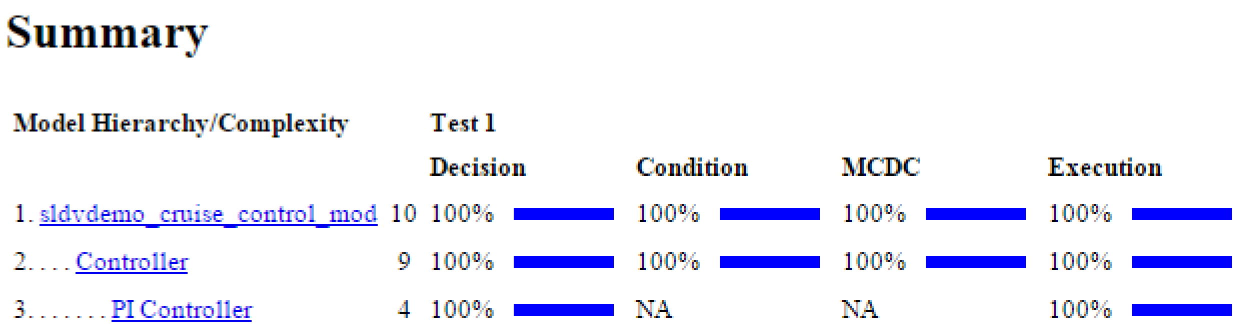
To achieve full model coverage, the `sldvgencov` function analyzes the model and extends the existing test suite.

```
[ status, covData, files ] = sldvgencov('sldvdemo_cruise_control_mod', opts, true, startCov);
```

Verify Complete Coverage

Verify that the new test suite achieves full coverage for the `sldvdemo_cruise_control_mod` modified model. The `sldvruntest` function simulates the modified model with the extended test suite. The `cvhtml` report shows the total coverage achieved by the `sldvdemo_cruise_control_mod` model.

```
[ additionalOut, additionalCov ] = sldvruntest('sldvdemo_cruise_control_mod', files.DataFile, []
totalCov = startCov + additionalCov;
cvhtml('With additional coverage',totalCov);
```



Summary

Model Hierarchy/Complexity		Test 1			
		Decision	Condition	MCDC	Execution
1. sldvdemo_cruise_control_mod	10	100%	100%	100%	100%
2. . . . Controller	9	100%	100%	100%	100%
3. . . . . . . PI Controller	4	100%	NA	NA	100%

To complete the example, close the model.

```
close_system('sldvdemo_cruise_control_mod');
```

Defining and Extending Existing Tests Cases

This example shows how Simulink® Design Verifier™ can extend test cases with additional time steps to efficiently generate complete test suites.

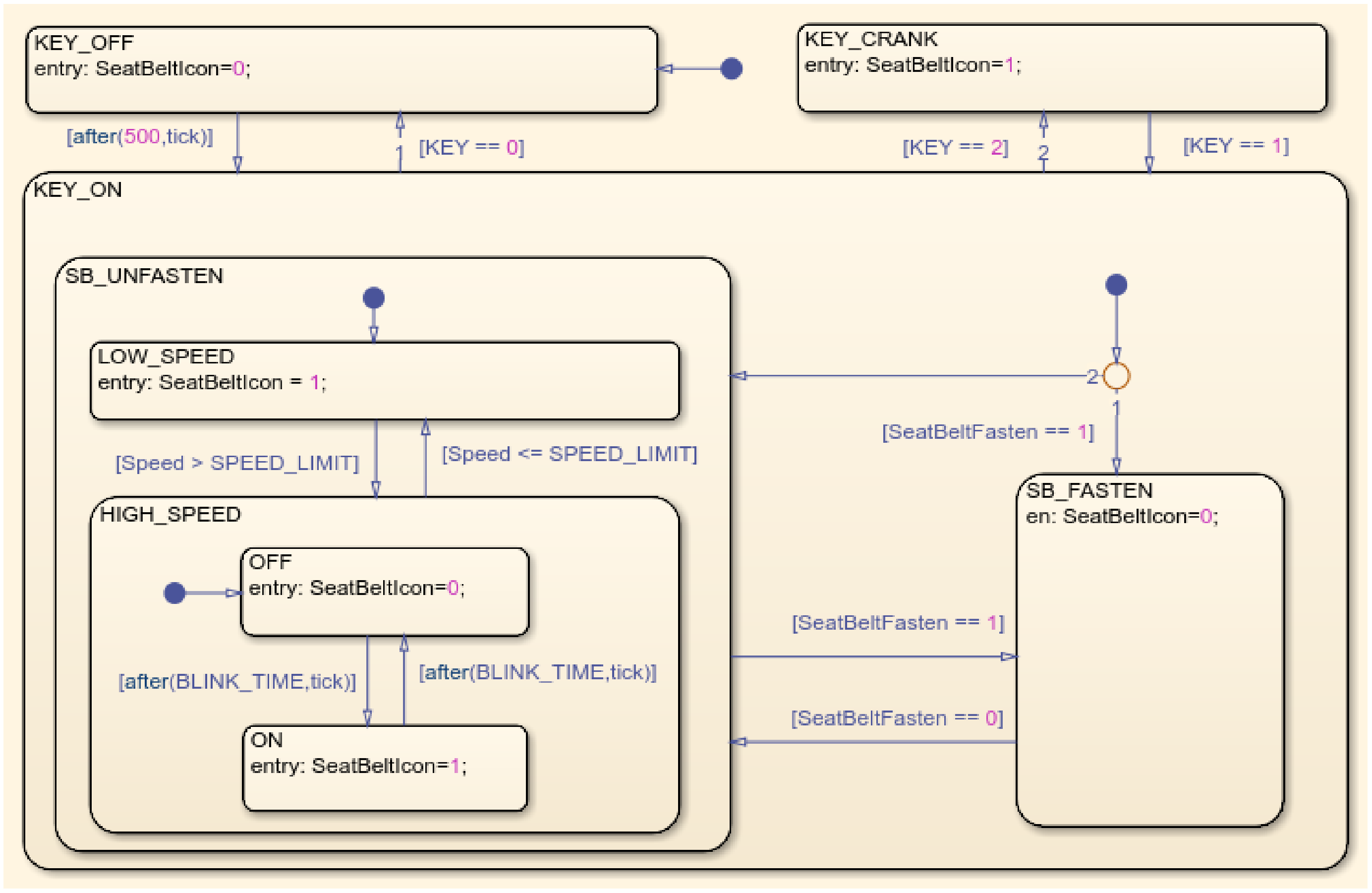
The example starts with a model containing time-delay characteristics that make test generation challenging. By creating a default test harness model and manually authoring one test, the critical obstacle to efficient test generation is removed. Simulink Design Verifier takes as input the logged values from the harness model and efficiently extends this test to create a complete test suite.

Model Characteristics That Motivate Test Case Extension

The `sldvdemo_sbr_extend_design` model includes the Stateflow® Chart `SBR` that uses temporal logic so that very long test cases are required to make a transition from the `KEY_OFF` state to the `KEY_ON` state. This type of time-delay characteristic is common in designs where a delay is used to reject spurious behavior or to wait for a physical system or user to respond. In this design, satisfying the temporal logic in this transition is a common obstacle to testing any of the states and transitions within the `KEY_ON` state.

Fortunately, this type of time-delay characteristic is usually easy to identify and satisfy with a manually authored test case.

```
open_system('sldvdemo_sbr_extend_design');
sf('Open',sldvdemo_ssid_to_sfid('sldvdemo_sbr_extend_design/SBR',11));
```



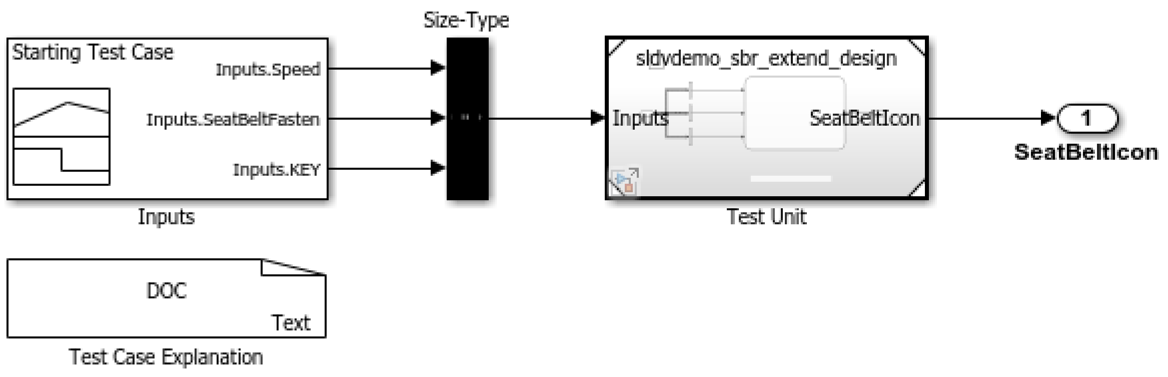
Creating a Harness Model and Defining Starting Tests

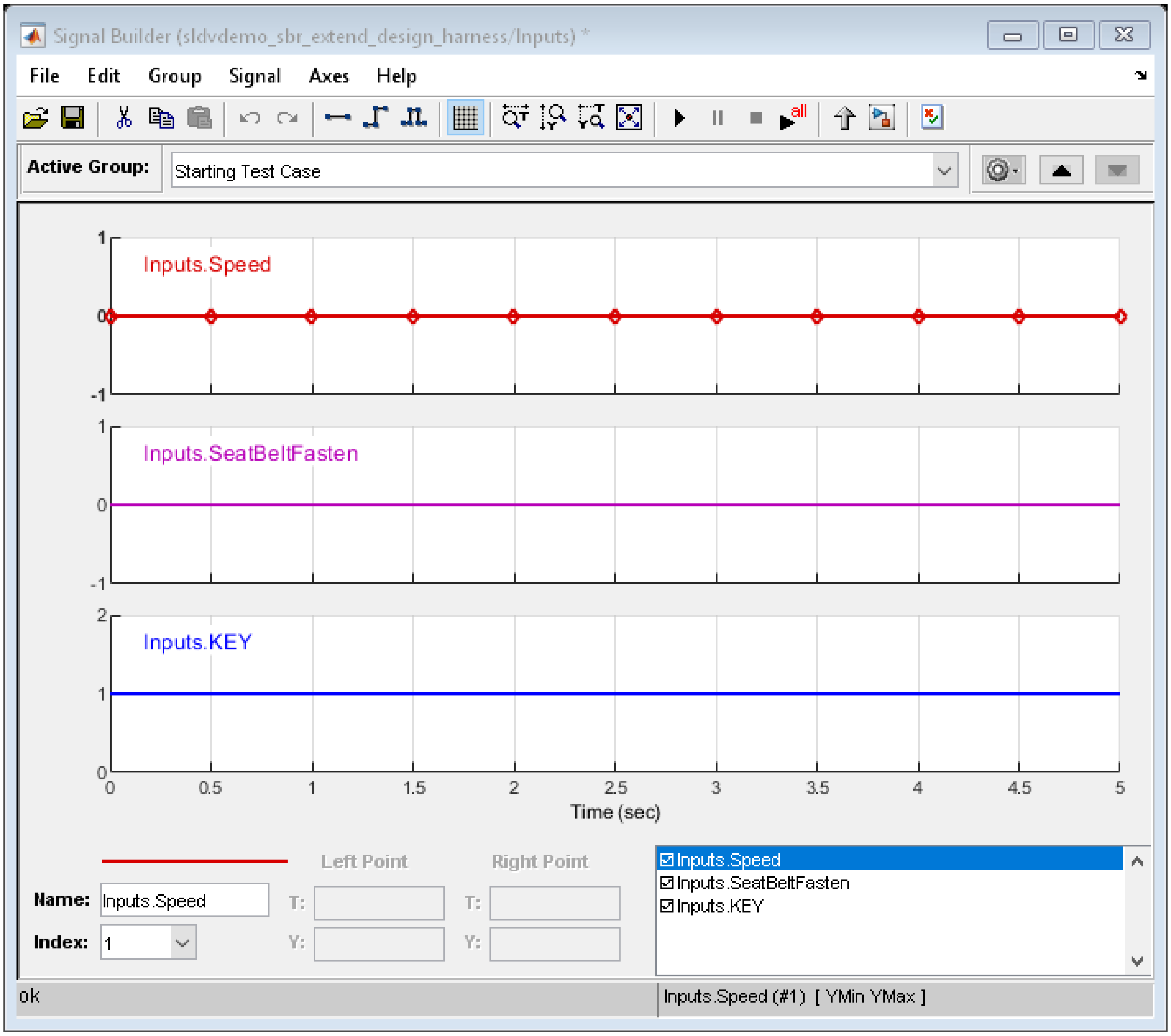
The Simulink Design Verifier function `sldvmakeharness` creates a harness model with a block that generates input values to the test model included by way of a model reference block.

You can modify the test data in a harness model by manually editing the data values using the Signal Builder user interface. You can also add more test cases by creating new signal groups in the block. Alternatively, you can use the `signalbuilder` command to programmatically accomplish the same thing.

In this example, you specify a test case that keeps the system in the `KEY_OFF` state for 5 seconds:

```
[~, harnessModelFilePath] = sldvmakeharness('sldvdemo_sbr_extend_design',[],[],true);
[~, harnessModel] = fileparts(harnessModelFilePath);

startingTestTime = 0:0.5:5;
startingTestData = cell(3, 1);
lengthStartingTest = length(startingTestTime);
startingTestData{1} = zeros(1,lengthStartingTest);
startingTestData{2} = zeros(1,lengthStartingTest);
startingTestData{3} = ones(1,lengthStartingTest);

signalBuilderBlock = sldvdemo_signalbuilder_block(harnessModel);
signalbuilder(signalBuilderBlock,'Append',...
    startingTestTime, startingTestData,...
    {'Inputs.Speed','Inputs.SeatBeltFasten','Inputs.KEY'},'Starting Test Case');

signalbuilder(signalBuilderBlock, 'ActiveGroup', 2);
open_system(signalBuilderBlock);
```





Name: Inputs.Speed
Index: 1

☑ Inputs.Speed
☑ Inputs.SeatBeltFasten
☑ Inputs.KEY

Inputs.Speed (#1) [YMin YMax]

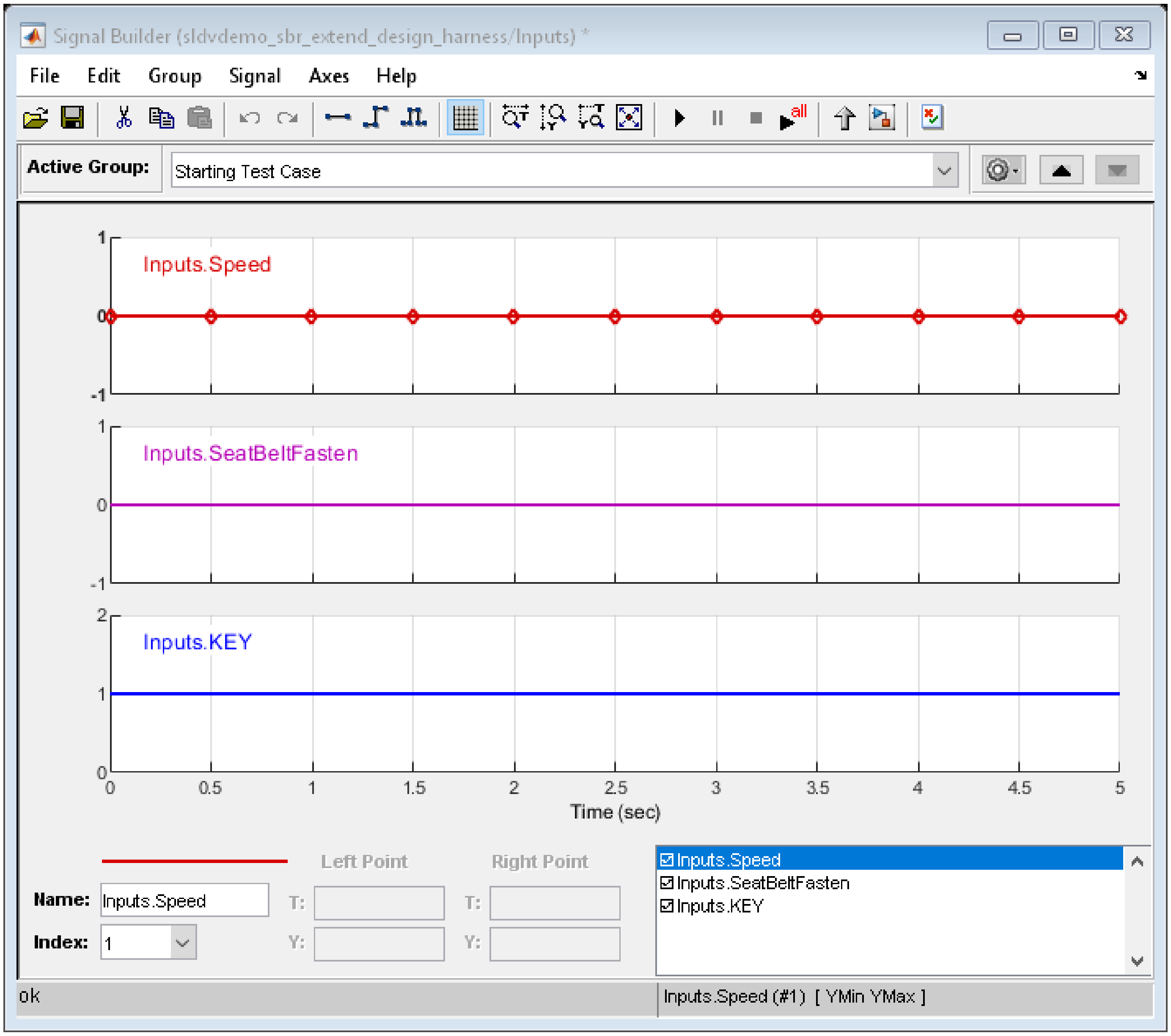
Logging Starting Tests

In order to leverage the starting test case defined above, you use the `sldvlogsignals` function to capture the input values in the necessary logged data format.

The first input to `sldvlogsignals` is the path to a Model block, and the second input is the index of signal group(s) within the harness model. When you invoke `sldvlogsignals`, the parent model that contains the Model block is simulated.

The parent model is not restricted to Simulink Design Verifier harness models. Alternatively, you might log data from a closed-loop simulation model that uses a Model block to include the controller so that controller test cases more realistically reflect the continuous time behavior expected in the closed-loop system.

```
[~, modelBlock] = find_mdlrefs(harnessModel, false);
loggeddata = sldvlogsignals(modelBlock{1},2);
```



Extending Existing Tests During Test Generation

Before you can use existing test data during test generation, the data must be saved to a MAT-file. You enable test case extension in the Test Generation pane of the Simulink Design Verifier configuration parameters. Select **Extend existing test cases**, and specify the MAT file in the **Data file** field.

Generated tests either extend one of the starting test cases with one or more new time steps or will specify one or more time steps starting from the initial, or default, configuration.

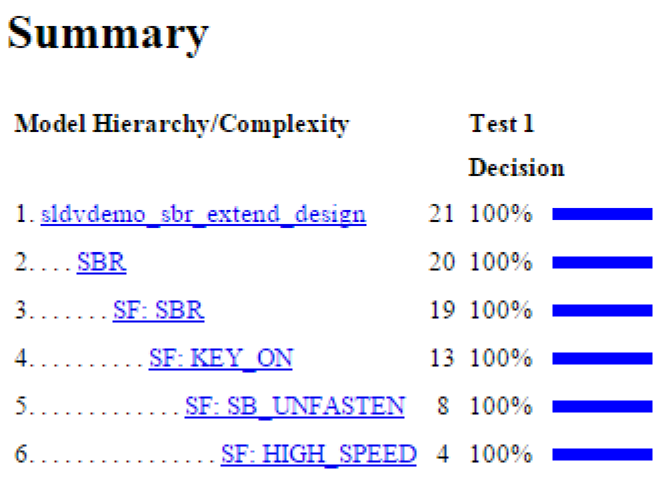
```
save('existingtestcase.mat', 'loggeddata');
```

```
opts = sldvoptions;
opts.ExtendExistingTests = 'on';
opts.ExistingTestFile = 'existingtestcase.mat';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';

[~, fileNames] = sldvrun('sldvdemo_sbr_extend_design', opts, true);
```

Verifying Complete Coverage

The `sldvruntest` function verifies that the new test suite achieves complete model coverage. The `cvhtml` function produces a coverage report that indicates 100% Decision coverage is achieved with the generated test vectors.

```
[~, finalCov] = sldvruntest('sldvdemo_sbr_extend_design', fileNames.DataFile, [], true);
cvhtml('Final Coverage', finalCov);
```



Summary

Model Hierarchy/Complexity		Test 1 Decision	
1. sldvdemo_sbr_extend_design	21	100%	▬
2. . . . SBR	20	100%	▬
3. . . . . . . SF: SBR	19	100%	▬
4. . . . . SF: KEY_ON	13	100%	▬
5. . . . . SF: SB_UNFASTEN	8	100%	▬
6. . . . . SF: HIGH_SPEED	4	100%	▬

Clean Up

To complete the demo, close all models and remove the saved logged data file.

```
close_system(harnessModel,0);
close_system('sldvdemo_sbr_extend_design');
delete('existingtestcase.mat');
```

Using Existing Coverage Data During Subsystem Analysis

This example shows how Simulink® Design Verifier™ can target its analysis to a single subsystem within a continuous-time closed-loop simulation and generate test cases for missing coverage in that subsystem.

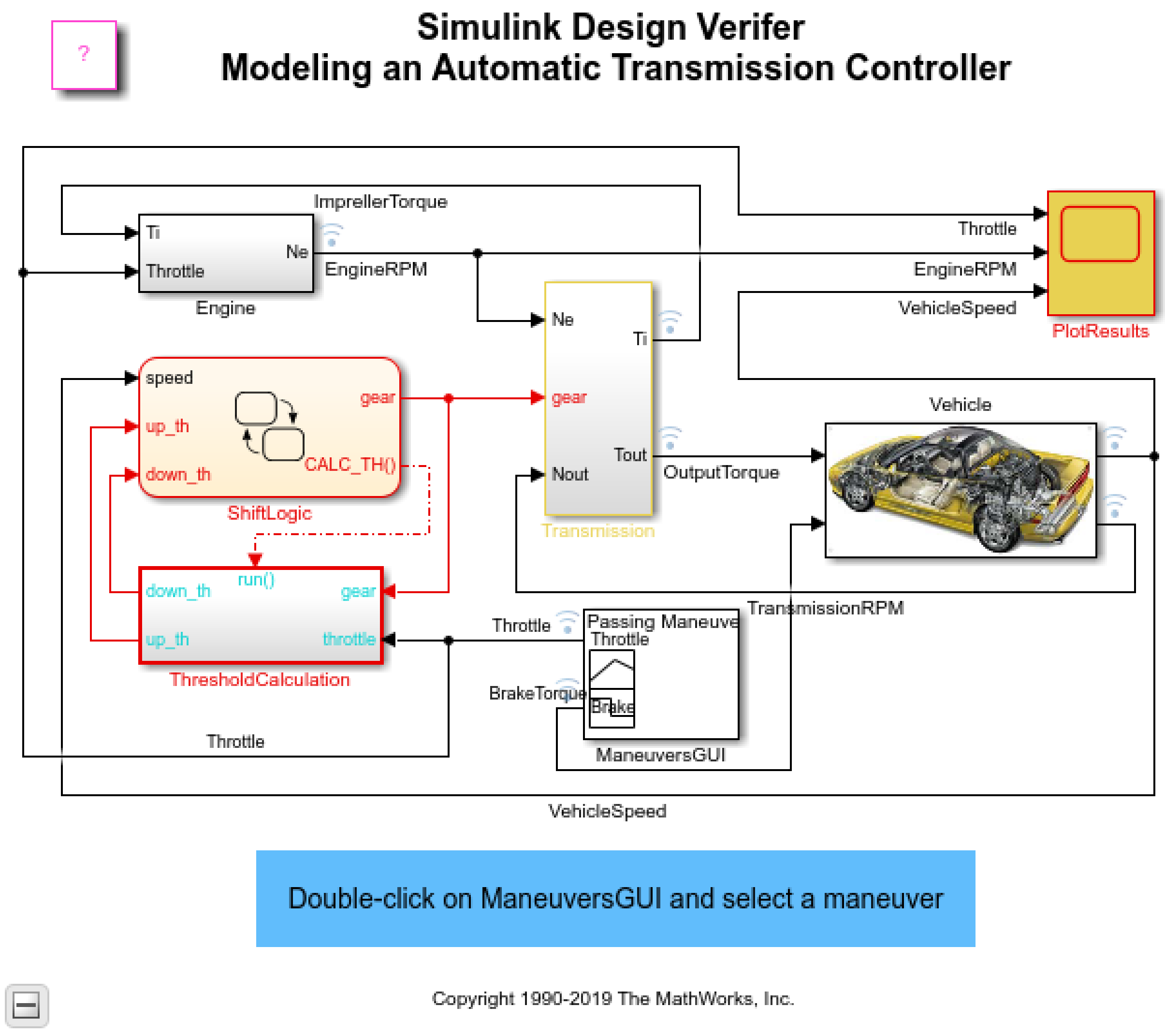
The example starts by measuring the coverage of a subsystem in a closed-loop simulation model. Simulink Design Verifier finds new test cases that achieve the missing coverage of the subsystem.

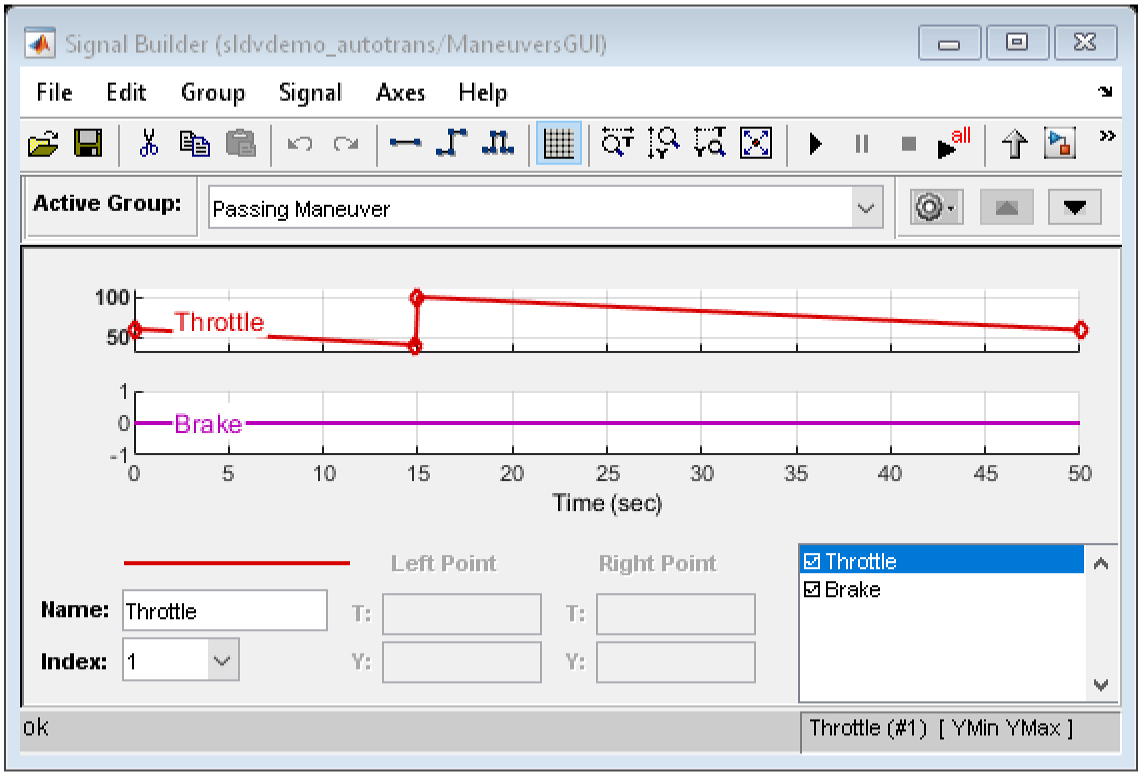
Measure Coverage of the Subsystem

The `sldvdemo_autotrans` model is a closed-loop simulation model. The subsystem `ShiftLogic` is a Stateflow® chart and represents the controller part of this model. Test cases designed in the Signal Builder block `ManeuversGUI` drive the closed-loop simulation. You can use the `cvtest` and `cvsim` functions to measure the model coverage achieved for this subsystem inside the closed-loop simulation model. In this example, specifying the input to `cvtest` as a path to the subsystem rather than to the model name results in measuring the coverage for the subsystem only. Also, the second input to `cvsim` specifies the time interval to simulate the model and it is derived from the time range of the current pane in the block `ManeuversGUI`.

The `cvhtml` function produces a report that indicates that 87% Decision, 67% Condition, and 33% MCDC coverage is achieved by simulating the test case authored in the block `ManeuversGUI`.

```
open_system('sldvdemo_autotrans');
open_system('sldvdemo_autotrans/ManeuversGUI');

test = cvtest('sldvdemo_autotrans/ShiftLogic');
test.settings.decision = 1;
test.settings.condition = 1;
test.settings.mcdc = 1;

signalBuilderBlock = sldvdemo_signalbuilder_block('sldvdemo_autotrans');
signalBuilderTime = signalbuilder(signalBuilderBlock);
simulationStopTime = signalBuilderTime{1,1}(end);

existingCovData = cvsim(test,[0 simulationStopTime]);
cvhtml('Existing Coverage', existingCovData);
```


?
Simulink Design Verifer
Modeling an Automatic Transmission Controller
ImprellerTorque
Ti
Ne
Throttle
Engine
EngineRPM
Throttle
EngineRPM
VehicleSpeed
PlotResults
Ne
Ti
gear
Tout
Nout
Transmission
OutputTorque
Vehicle
speed
up_th
down_th
gear
CALC_TH()
ShiftLogic
run()
down_th
gear
up_th
throttle
ThresholdCalculation
Throttle
Passing Maneuve
Throttle
Brake
BrakeTorque
ManeuversGUI
TransmissionRPM
Throttle
VehicleSpeed
Double-click on ManeuversGUI and select a maneuver
Copyright 1990-2019 The MathWorks, Inc.


Signal Builder (sldvdemo_autotrans/ManeuversGUI)
File Edit Group Signal Axes Help
Active Group: Passing Maneuver
Throttle
Brake
Time (sec)
Left Point
Right Point
Name: Throttle
Index: 1
Throttle
Brake
ok
Throttle (#1) [YMin YMax]

Find Test Cases for Missing Coverage

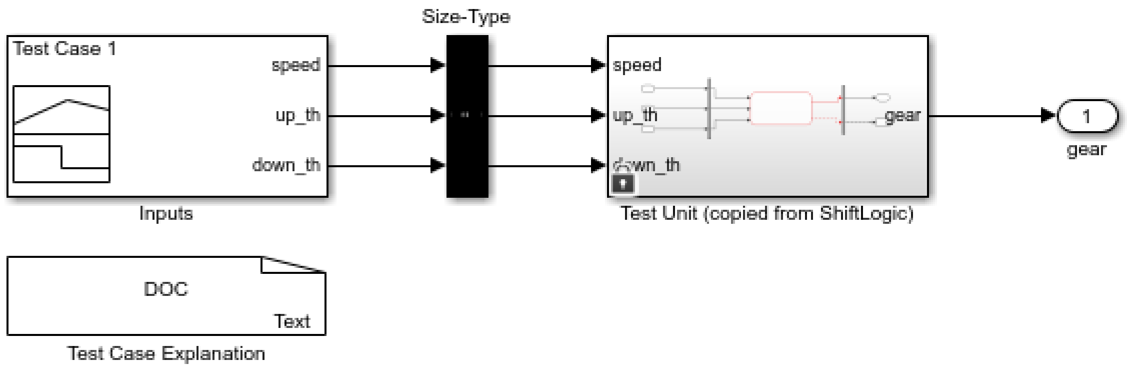
To use existing coverage data during test generation, save existing coverage data to a .cvt coverage data file. You can use existing coverage data by specifying the coverage data path in the **Coverage data file** parameter and setting **Ignore objectives satisfied in existing coverage data** parameter to `on` in the **Test Generation** pane of Simulink Design Verifier configuration parameters.

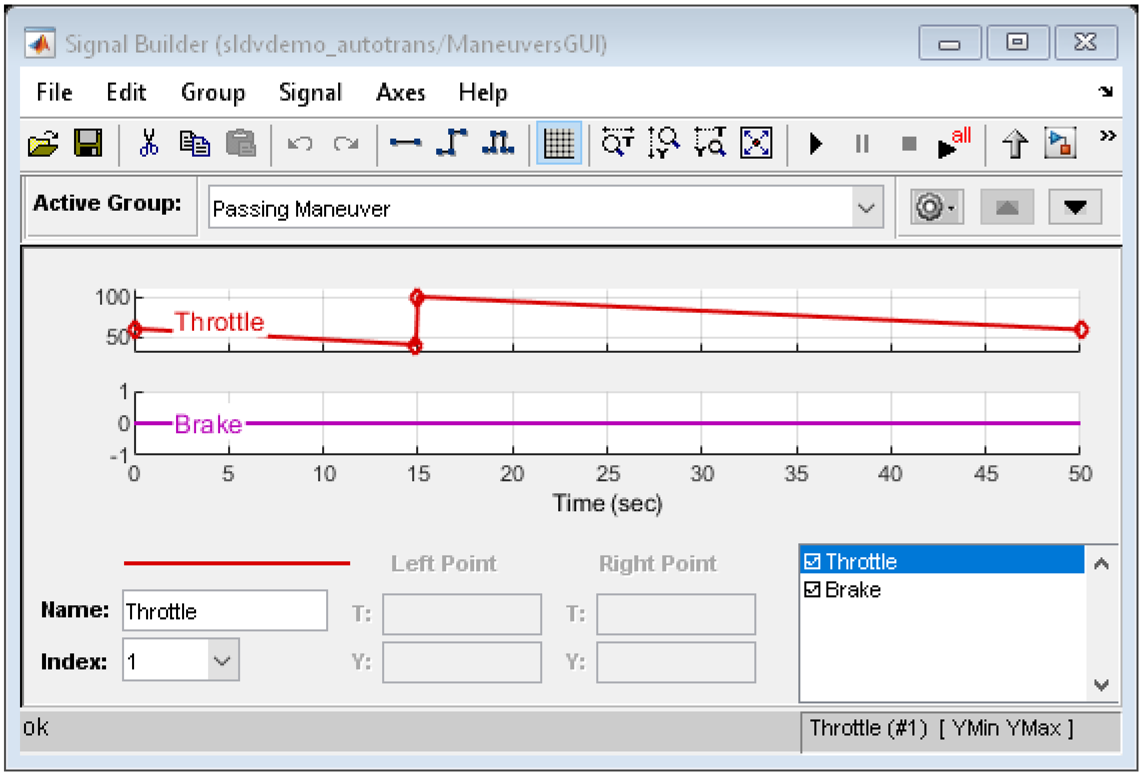
In this example, the first input to `sldvrun` specifies the subsystem to analyze. Instructing Simulink Design Verifier to analyze a subsystem is beneficial when the controller part of a model needs to be tested separately or when you want to divide the analysis of a large model into smaller, manageable parts.

As you can see in the report, Simulink Design Verifier only finds test cases for the coverage objectives that are not covered in the existing coverage file. Notice that 4 coverage objectives in the subsystem `ShiftLogic` are proven to be unsatisfiable. This is expected because the logic inside the Stateflow chart `ShiftLogic` uses temporal events and since this chart updates at every sample time, usage of temporal conditions should be satisfactory. Also note that, dead code within a subsystem will always be a dead code in the model containing that subsystem.

To generate the harness model, Simulink Design Verifier extracts the contents of the subsystem `ShiftLogic` into a Test Unit component fed by a Signal Builder block containing the generated test cases.

```
cvsave('existingcov',existingCovData);

opts = sldvoptions;
opts.IgnoreCovSatisfied = 'on';
opts.CoverageDataFile = 'existingcov.cvt';
opts.ModelCoverageObjectives = 'MCDC';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'on';

[status, fileNames] = sldvrun('sldvdemo_autotrans/ShiftLogic',opts,true);
[~, harnessModel] = fileparts(fileNames.HarnessModel);
open_system(harnessModel);
```





Active Group: Passing Maneuver

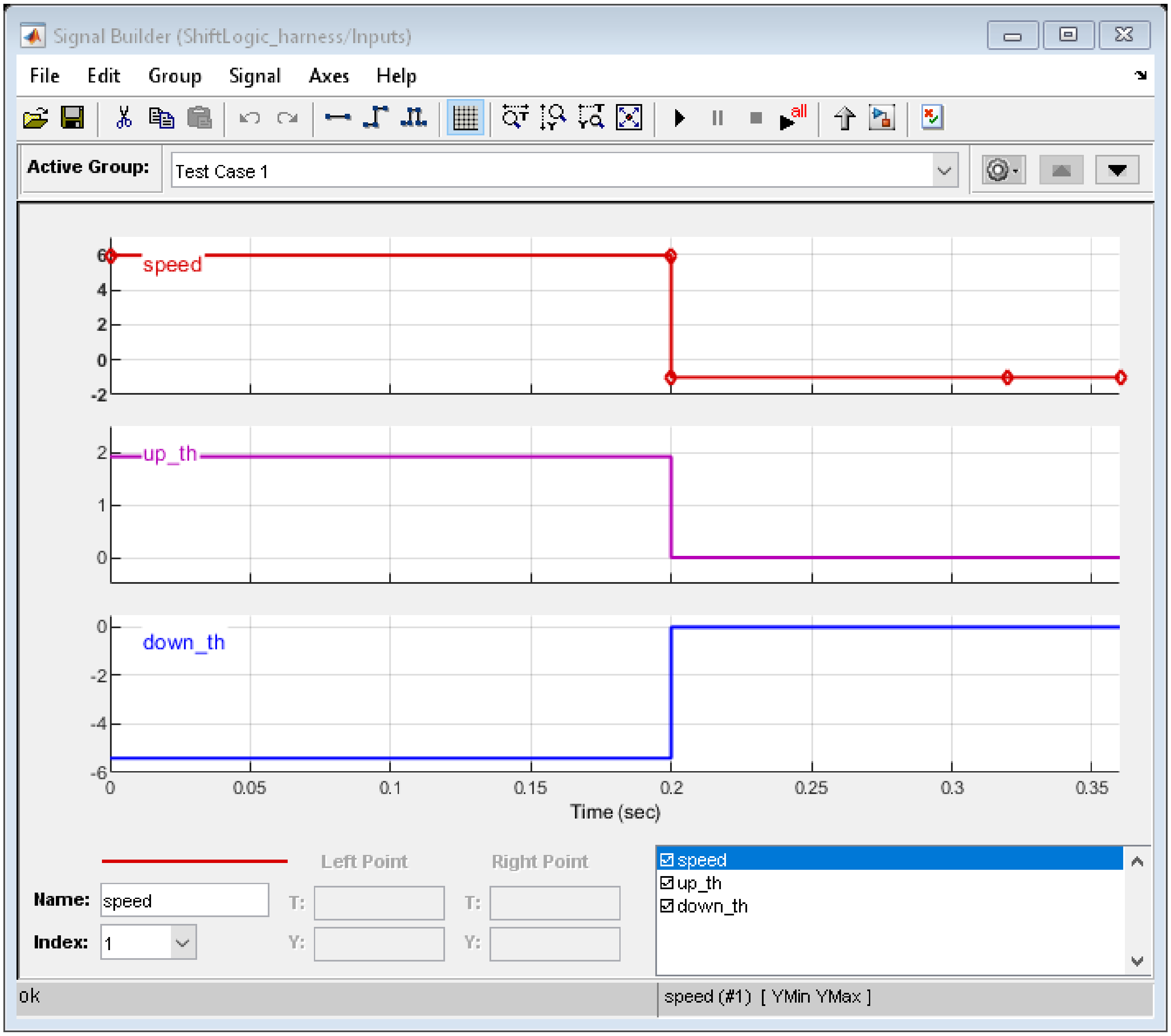
Name: Throttle

Index: 1

☑ Throttle
☑ Brake

ok

Throttle (#1) [YMin YMax]



Active Group: Test Case 1

Name: speed

Index: 1

☑ speed
☑ up_th
☑ down_th

ok

speed (#1) [YMin YMax]

Clean Up

To complete the demo, close all models and remove the saved coverage data file.

```
close_system('sldvdemo_autotrans');
close_system(fileNames.ExtractedModel,0);
close_system(fileNames.HarnessModel,0);
delete('existingcov.cvt');
```

Creating and Executing Test Cases

This example shows how to use Simulink® Design Verifier™ functions to log input signals, create a harness model, generate test cases for missing coverage, merge harness models, and execute test cases.

The example starts by logging input signals to the component that implements the controller in its parent model and creating harness model for the controller from that logged data. You use Simulink Design Verifier to find a new test case that achieves the missing coverage. Then you merge the first harness model with the harness model generated after the Simulink Design Verifier analysis. Finally, you capture all test cases and execute the controller with those test cases in simulation mode and Software-In-the-Loop (SIL) mode, and compare the results using CGV API.

Check Product Availability

This example requires a valid Stateflow® license. To demonstrate test execution in Software-In-the-Loop (SIL) mode it also requires valid Simulink® Coder™ and Embedded Coder™ licenses.

```
if ~license('test','Stateflow')
    return;
end

canUseSIL = license('test','Real-Time_Workshop') && ...
    license('test','RTW_Embedded_Coder');
```

Logging Input Signals to the Component and Creating the Harness Model

The `slvnvdemo_powerwindow` model contains a power window controller and a low-order plant model. The component `slvnvdemo_powerwindow/power_window_control_system/control` is a Model block that references the model `slvnvdemo_powerwindow_controller`, which implements the controller with a Stateflow® chart.

To create a harness model for the controller with the signals that simulate the controller in the plant model, first log the input signals and then invoke harness generation with that logged data.

```
open_system('slvnvdemo_powerwindow');
load_system('slvnvdemo_powerwindow_controller');

loggedSignalsPlant = ...
    sldvlogsignals('slvnvdemo_powerwindow/power_window_control_system/control');

harnessModelFilePath = ...
    sldvmakeharness('slvnvdemo_powerwindow_controller',loggedSignalsPlant);
[~,harnessModel] = fileparts(harnessModelFilePath);
```

Measuring the Coverage with Logged Signals

Use the `cvtest` and `cvsim` functions to measure the model coverage achieved for the controller model `slvnvdemo_powerwindow_controller` with the logged signals that are captured in the harness model.

The `cvhtml` function produces a report that indicates that 40% Decision, 35% Condition, and 10% MCDC coverage is achieved by simulating the test cases captured from the closed-loop model.

```
test = cvtest(harnessModel);
test.modelRefSettings.enable = 'On';
test.modelRefSettings.excludeTopModel = 1;

covDataFromLoggedSignals = cvsim(test);
cvhtml('Coverage with Logged Test Cases',covDataFromLoggedSignals);
```

Finding Test Cases for Missing Coverage

Before you can use existing coverage data during test generation, the data must be saved to a coverage data file(.cvt). You can use the existing coverage data by specifying the coverage data path in the **Coverage data file** parameter and setting the **Ignore objectives satisfied in existing coverage data** parameter to `on` in the **Test Generation** pane of Simulink Design Verifier configuration parameters.

As you can see in the report, Simulink Design Verifier restricts test generation to the coverage objectives that are not covered in the existing coverage file. Notice that 8 coverage objectives in the Stateflow chart `control` are proven to be unsatisfiable. This indicates unnecessary redundant logic that cannot be tested.

```
cvsave('existingCovFromLoggedSignals',covDataFromLoggedSignals);

opts = sldvoptions;
opts.DisplayUnsatisfiableObjectives = 'off';
opts.IgnoreCovSatisfied = 'on';
opts.CoverageDataFile = 'existingCovFromLoggedSignals.cvt';
opts.ModelCoverageObjectives = 'MCDC';
opts.TestSuiteOptimization = 'LongTestcases';
opts.SaveHarnessModel = 'on';
opts.ModelReferenceHarness = 'on';
opts.MaxProcessTime = 500;

[status, fileNames] = sldvrun('slvnvdemo_powerwindow_controller',opts,true);
[~, newHarnessModel] = fileparts(fileNames.HarnessModel);
open_system(newHarnessModel);
```

Merging Test Cases from Harness Models

Now use `sldvmergeharness` to combine generated test cases with logged test case. The command takes a list of harness models as arguments.

```
sldvmergeharness(harnessModel, newHarnessModel);
```

Logging Test Cases of the Harness Model

In order to programmatically execute the model `slvnvdemo_powerwindow_controller` with the test cases captured in the merged harness model, first use the `sldvlogsignals` function to obtain the input values of all test cases in the necessary data format.

```
loggedSignalsMergedHarness = sldvlogsignals(harnessModel);
disp(loggedSignalsMergedHarness);
```

Execute the Model in Simulation Mode with CGV API

Use the `sldvruncgvtest` function to execute the model `slvnvdemo_powerwindow_controller` in simulation mode, with test cases captured from the harness model.

```
runopts = sldvruntestopts('cgv');
disp(runopts);
```

```
runopts.cgvConn = 'sim';
cgvSim = sldvruncgvtest('slvnvdemo_powerwindow_controller',...
    loggedSignalsMergedHarness,runopts);
```

Execute the Model in Software-In-the-Loop (SIL) Mode with CGV API

Now use the `sldvruncgvtest` function to execute the model `slvnvdemo_powerwindow_controller` in SIL mode, with the same test cases.

```
if canUseSIL
    runopts.cgvConn = 'sil';
else
    % When SIL is not possible, the example runs another simulation.
    runopts.cgvConn = 'sim';
end
cgvSil = sldvruncgvtest('slvnvdemo_powerwindow_controller',...
    loggedSignalsMergedHarness,runopts);
```

Compare Results of Simulation and SIL Modes

The `sldvruncgvtest` returns a `cgv.CGV` object after running tests. Use the CGV API to compare the results of executions in simulation and SIL modes for each test case designed in the harness model and show that they are equal.

```
for i=1:length(loggedSignalsMergedHarness.TestCases)
    simout = cgvSim.getOutputData(i);
    silout = cgvSil.getOutputData(i);

    [matchNames, ~, mismatchNames, ~ ] = ...
        cgv.CGV.compare(simout, silout);

    fprintf('\nTest Case(%d):  %d Signals match, %d Signals mismatch', ...
        i, length(matchNames), length(mismatchNames));
end
```

Clean Up

To complete the example, close all models.

```
close_system(harnessModel,0);
close_system(newHarnessModel,0);
close_system('slvnvdemo_powerwindow',0);
close_system('slvnvdemo_powerwindow_controller',0);
```

Using Specified Input Minimum and Maximum Values as Constraints

This example shows how to use input port minimum and maximum values as analysis constraints by Simulink Design Verifier during both test generation and property proving.

This model is preconfigured to generate tests for decision coverage. The specified minimum and maximum values are displayed in square brackets. The constraints in this example prevent some of the coverage objectives from being satisfied. When you generate tests without considering these constraints, all of the coverage objectives are satisfied.

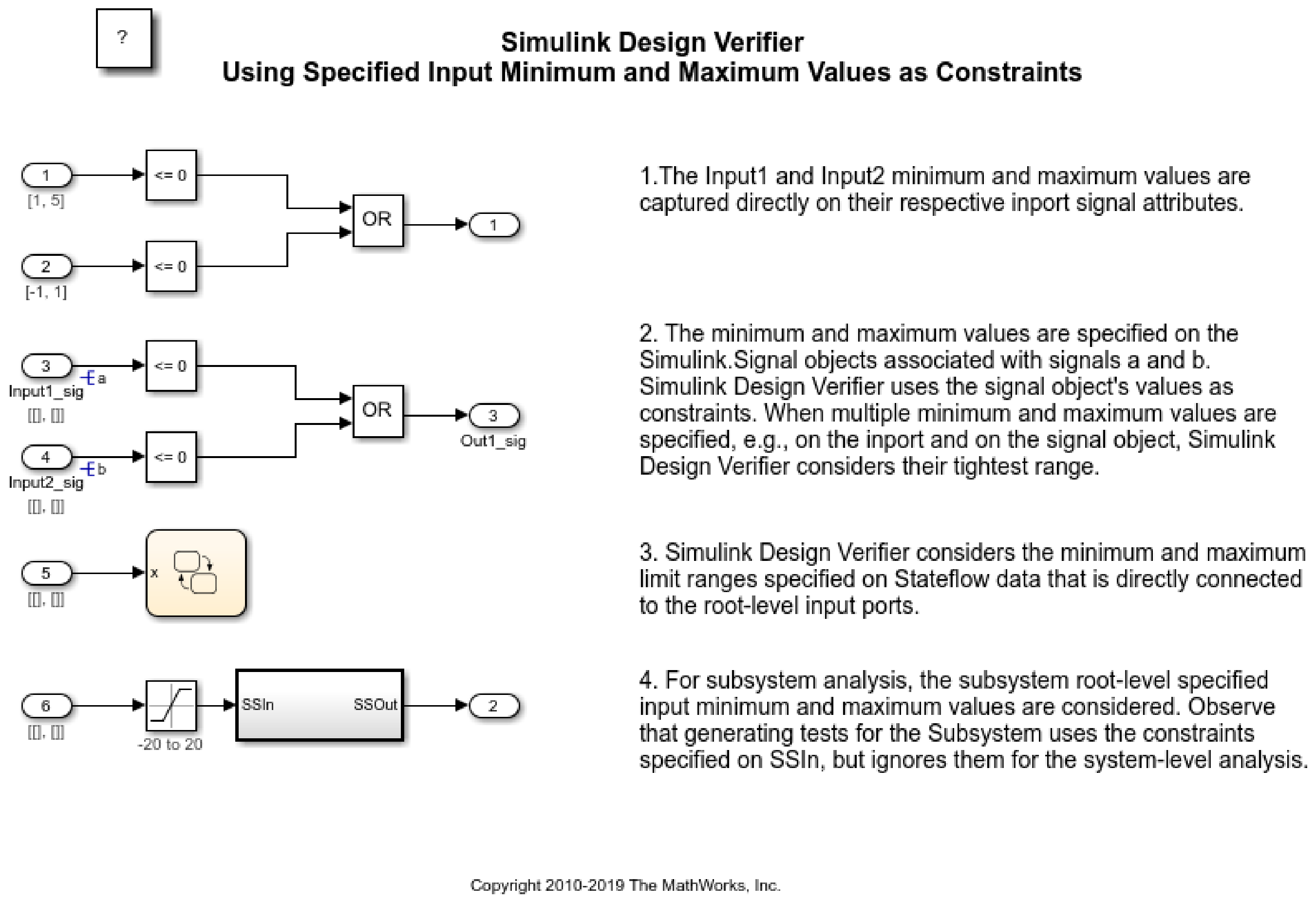
1. The Input1 and Input2 minimum and maximum values are captured directly on their respective inport signal attributes.

2. The minimum and maximum values are specified on the Simulink.Signal objects associated with signals a and b. Simulink Design Verifier uses the signal object's values as constraints. When multiple minimum and maximum values are specified, e.g., on the inport and on the signal object, Simulink Design Verifier considers their tightest range.

3. Simulink Design Verifier considers the minimum and maximum limit ranges specified on Stateflow data that is directly connected to the root-level input ports

4. For subsystem analysis, the subsystem root-level specified input minimum and maximum values are considered. Observe that generating tests for the Subsystem uses the constraints specified on SSIn, but ignores them for the system-level analysis.

```
open_system('sldvdemo_minmaxconstraints');
```



Simulink Design Verifier
Using Specified Input Minimum and Maximum Values as Constraints

1.The Input1 and Input2 minimum and maximum values are captured directly on their respective inport signal attributes.

2. The minimum and maximum values are specified on the Simulink.Signal objects associated with signals a and b. Simulink Design Verifier uses the signal object's values as constraints. When multiple minimum and maximum values are specified, e.g., on the inport and on the signal object, Simulink Design Verifier considers their tightest range.

3. Simulink Design Verifier considers the minimum and maximum limit ranges specified on Stateflow data that is directly connected to the root-level input ports.

4. For subsystem analysis, the subsystem root-level specified input minimum and maximum values are considered. Observe that generating tests for the Subsystem uses the constraints specified on SSIn, but ignores them for the system-level analysis.

Configuring S-Function for Test Case Generation

This example shows how to compile an S-Function to be compatible with Simulink® Design Verifier™ for test case generation. SLDV supports S-Functions that are:

- Generated with the Legacy Code Tool, with `def.Options.supportCoverageAndDesignVerifier` set to `true`,
- Generated with the SFunctionBuilder, with **Enable support for Design Verifier** selected on the **Build Info** tab of the SFunctionBuilder dialog box, or
- Compiled with the function slcovmex, with the option `-sldv` passed.

Compile S-Function to Be Compatible with Simulink® Design Verifier™

The handwritten S-Function is found in the file sldvexSFunctionHandlingSFcn.c, and the user source code for the lookup table is found in the file sldvexSFunctionHandlingSource.c. Call the function slcovmex to compile the C-MEX S-Function and make it compatible with SLDV.

```
slcovmex('-sldv', ...
        '-output', 'sldvexSFunctionHandlingSFcn',...
        ['-I', fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src')], ...
        fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src', 'sldvexSFunctionHandlingSour
        fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src', 'sldvexSFunctionHandlingSFcn
        );
```

```
mex -IB:\matlab\toolbox\sldv\sldvdemos\src C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tpd4673b0e_f
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex -IB:\matlab\toolbox\sldv\sldvdemos\src B:\matlab\toolbox\sldv\sldvdemos\src\sldvexSFunctionHa
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
```

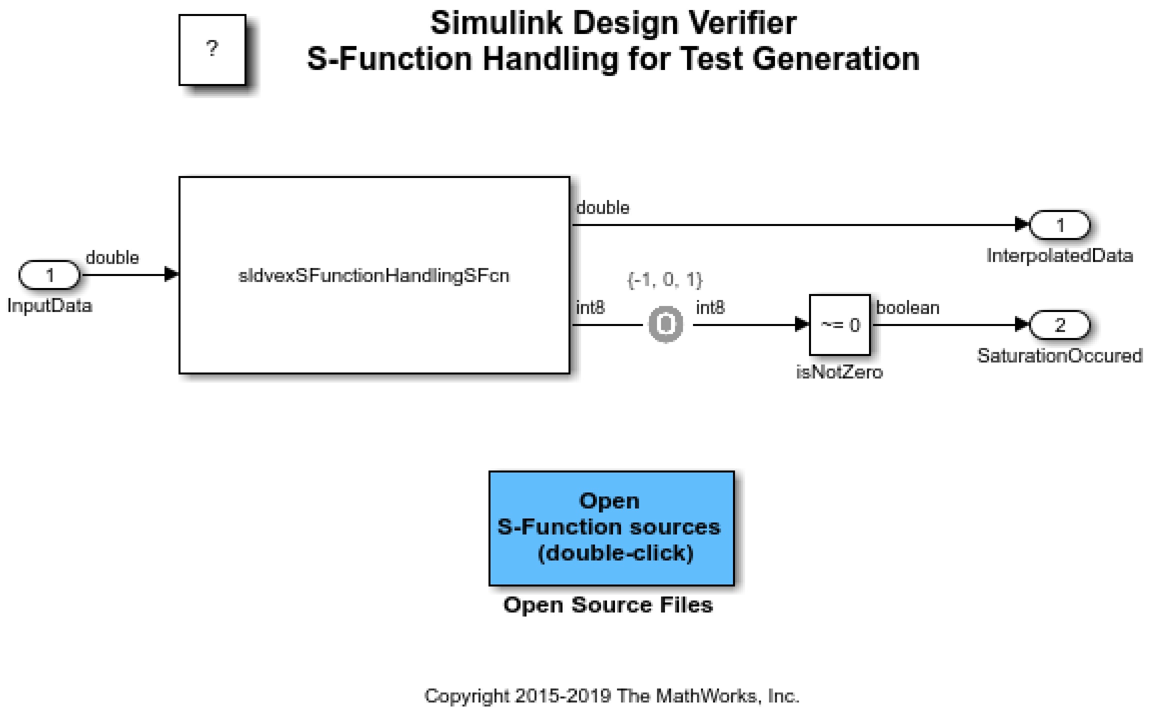
Create Test Suite

The example model sldvexSFunctionHandlingExample example contains the handwritten S-Function, which implements a lookup table algorithm. The S-Function block returns the interpolated value at the first output port and returns the status of the interpolation at the second output port. The second output port returns the value `-1` if a lower saturation occurs, `1` if a upper saturation occurs, and `0` otherwise. Open the sldvexSFunctionHandlingExample model and configure the analysis options by turning on S-Function support for test generation. On running the analysis, Simulink® Design Verifier™ returns a test suite that satisfies all coverage objectives.

```
open_system('sldvexSFunctionHandlingExample');

opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'Condition';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
opts.SFcnSupport = 'on';
opts.MaxProcessTime = 2*opts.MaxProcessTime;

[status, fileNames] = sldvrun('sldvexSFunctionHandlingExample', opts);
```

```
Checking compatibility for test generation: model 'sldvexSFunctionHandlingExample'
Compiling model...done
```

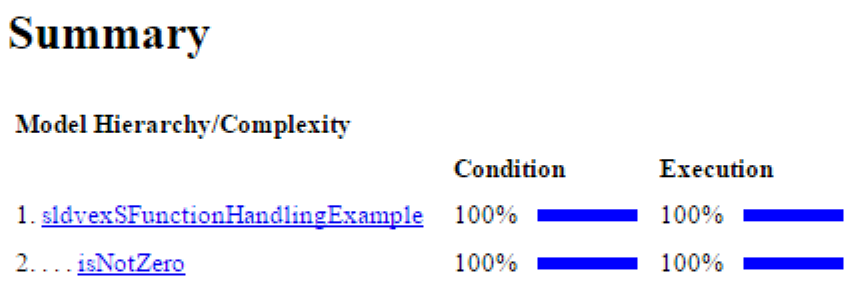
```
Building model representation...done

'sldvexSFunctionHandlingExample' is compatible for test generation with Simulink Design Verifier

Generating tests using model representation from 29-Feb-2020 11:40:47...
.............

Completed normally.

Generating output files:

Results generation completed.

    Data file:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex16001120\sldv_output\sldvexSFunctionHa
```



Verifying Complete Coverage

The `sldvruntest` function verifies that the test suite achieves complete model coverage. The `cvhtml` function produces a coverage report that indicates 100% Condition coverage is achieved with the generated test vectors.

```
[~, finalCov] = sldvruntest('sldvexSFunctionHandlingExample', fileNames.DataFile, [], true);
cvhtml('Final Coverage', finalCov);
```



Summary

Model Hierarchy/Complexity

	Condition		Execution	
1. sldvexSFunctionHandlingExample	100%	▬	100%	▬
2. . . . isNotZero	100%	▬	100%	▬

Clean Up

To complete the example, close the model.

```
close_system('sldvexSFunctionHandlingExample', 0);
```

Code Coverage Test Generation

This example shows how to use Simulink® Design Verifier™ to generate test cases to obtain complete code coverage.

You first collect code coverage for an example model configured for software-in-the-loop (SIL) simulation mode. Then you use Simulink® Design Verifier™ to create a test suite that generates tests cases to achieve the missing coverage. Finally, you execute the generated test cases in SIL simulation mode to verify the complete coverage.

Check Product Availability

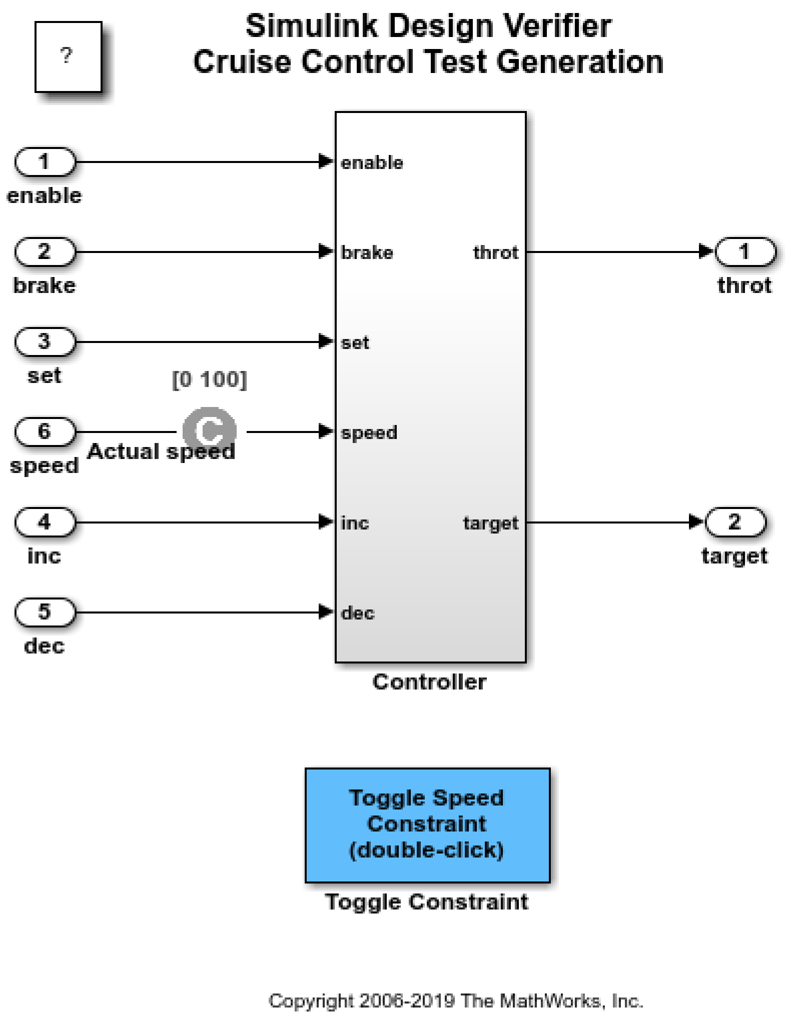
Make sure that you have Simulink® Coder™ and Embedded Coder™ software installed on your machine.

```
if ~(license('test', 'Real-Time_Workshop') && ...
    license('test','RTW_Embedded_Coder'))
    return
end
```

Initial Setup

Make sure that an unedited version of the model is open.

```
model = 'sldvdemo_cruise_control';
close_system(model, 0)
open_system(model)
```



Configure the Model to Measure Code Coverage

Before running the simulation, set the model parameters to run in SIL mode and to collect code coverage metrics by using Simulink® Coverage™.

```
set_param(model,...
    'SimulationMode', 'Software-in-the-Loop (SIL)',...
    'SystemTargetFile', 'ert.tlc',...
    'LaunchReport', 'off',...
    'PortableWordSizes', 'on',...
    'CovEnable', 'on');

% Remove any existing build folders.
buildFolder = RTW.getBuildDir(model);
if exist(buildFolder.BuildDirectory, 'dir')
    rmdir(buildFolder.BuildDirectory, 's');
end
```

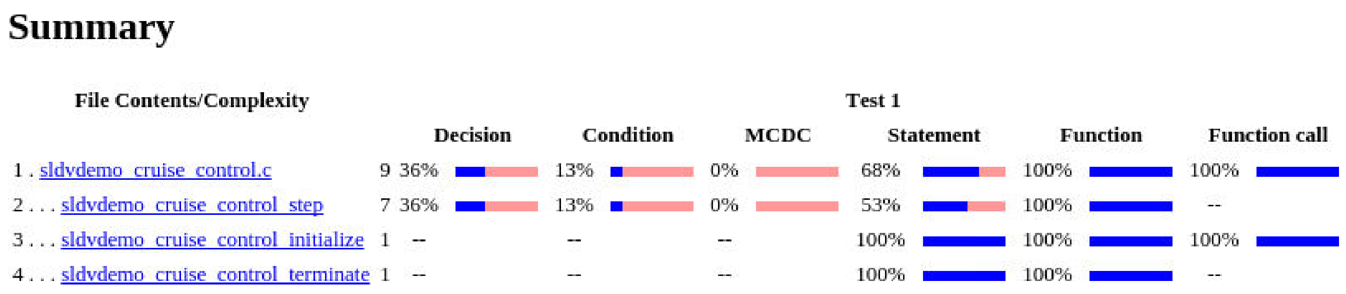
Run Simulations in SIL Mode

Collect code coverage results by using the `cvsim` command and producing a coverage report. The `cvhtml` function produces a coverage report that indicates the initial coverage of the `sldvdemo_cruise_control` model.

```
initialCov = cvsim(model);

cvhtml('sil_initial_coverage', initialCov);
```

```
### Starting build procedure for: sldvdemo_cruise_control
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex155
### Invoking Target Language Compiler on sldvdemo_cruise_control.rtw
### Using System Target File: B:\matlab\rtw\c\ert\ert.tlc
### Loading TLC function libraries
......
### Initial pass through model to cache user defined code
.
### Caching model source code
...............................
    ### Writing header file sldvdemo_cruise_control_types.h
    ### Writing header file sldvdemo_cruise_control.h
    ### Writing header file rtwtypes.h
    ### Writing source file sldvdemo_cruise_control.c
    ### Writing header file sldvdemo_cruise_control_private.h
.
    ### Writing source file ert_main.c
### TLC code generation complete.
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_con
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk buildobj

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTER
sldvdemo_cruise_control.c
### Successfully generated all binary outputs.

### Successful completion of build procedure for: sldvdemo_cruise_control
### Simulink cache artifacts for 'sldvdemo_cruise_control' were created in 'C:\TEMP\Bdoc20a_1326
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_con
### Building 'sldvdemo_cruise_control_ca': nmake  -f sldvdemo_cruise_control_ca.mk all

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\co

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\co

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\co
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DINTEGER_CODE=0
coder_assumptions_hwimpl.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DINTEGER_CODE=0
coder_assumptions_flt.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DINTEGER_CODE=0
sldvdemo_cruise_control_ca.c
### Creating static library ".\sldvdemo_cruise_control_ca.lib" ...
    lib /nologo -out:.\sldvdemo_cruise_control_ca.lib @sldvdemo_cruise_control_ca.rsp
### Created: .\sldvdemo_cruise_control_ca.lib
### Successfully generated all binary outputs.

### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_cont
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk all

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\si

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\si

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\si
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
xil_interface_lib.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
xil_data_stream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
xil_services.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
xil_interface.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
xilcomms_rtiostream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
xil_rtiostream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERF
```

```
rtiostream_utils.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_app.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_data_stream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_rtiostream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
sil_main.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
target_io.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
rtiostream_tcpip.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xil_instrumentation.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
codeinstr_data_stream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
codeinstr_rtiostream.c
### Creating standalone executable ".\sldvdemo_cruise_control.exe" ...
    link /RELEASE  /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsock.lib advapi32.lib -out
### Created: .\sldvdemo_cruise_control.exe
### Successfully generated all binary outputs.

### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier 17652
rtw.connectivity.HostLauncher: stopped executable with host process identifier 17652
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
```



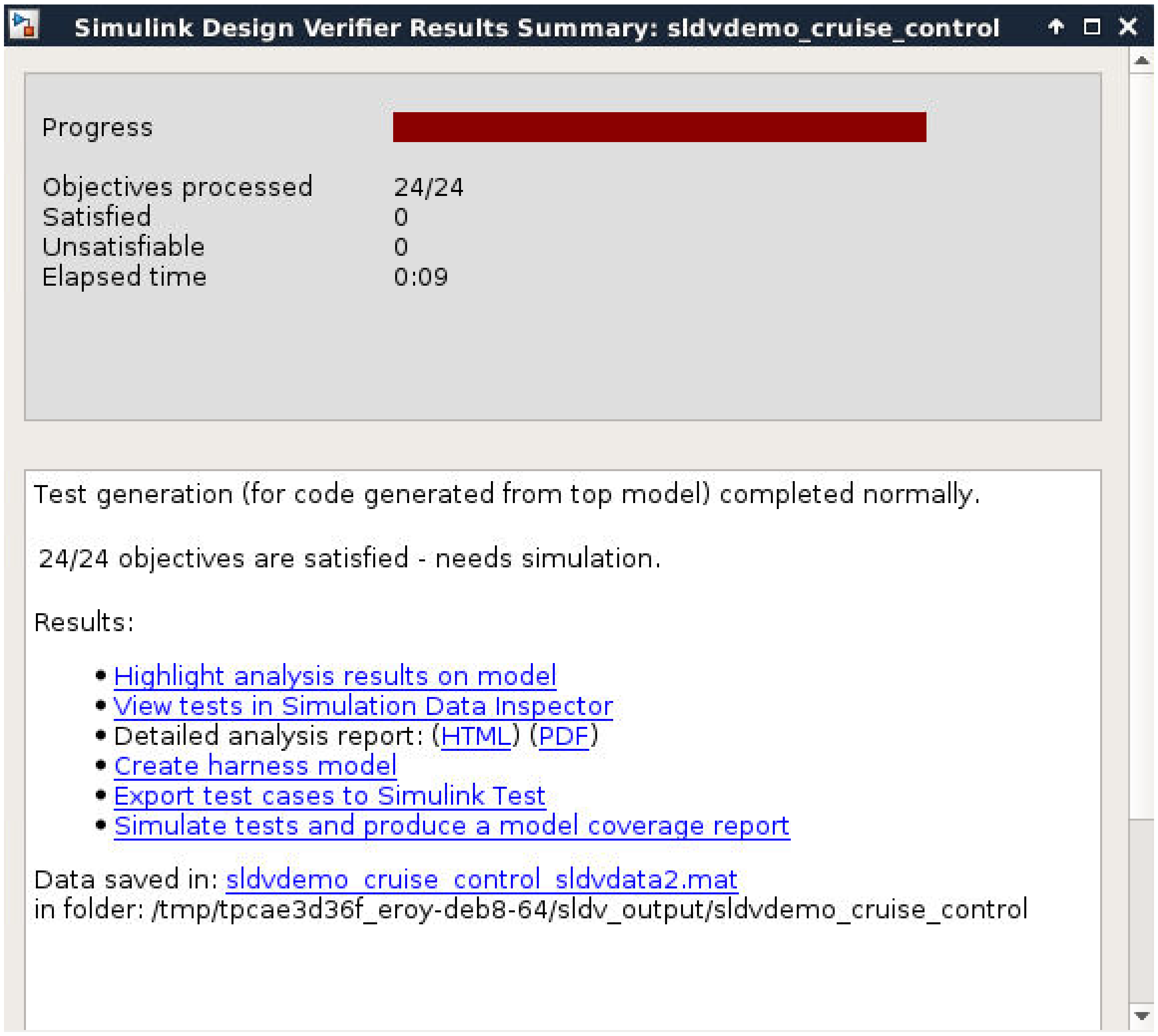
Summary

File Contents/Complexity		Test 1					
		Decision	Condition	MCDC	Statement	Function	Function call
1 . sldvdemo_cruise_control.c	9	36%	13%	0%	68%	100%	100%
2 . . . sldvdemo_cruise_control_step	7	36%	13%	0%	53%	100%	--
3 . . . sldvdemo_cruise_control_initialize	1	--	--	--	100%	100%	100%
4 . . . sldvdemo_cruise_control_terminate	1	--	--	--	100%	100%	--

Find Test Cases for Missing Coverage

Analyze the `sldvdemo_cruise_control` model by using Simulink® Design Verifier™ to generate a test suite that achieves increased code coverage. Set the Simulink® Design Verifier™ options to generate test cases to achieve MCDC coverage for the top model.

```
opts = sldvoptions;
opts.TestgenTarget = 'GenCodeTopModel';
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'MCDC';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
[~, files] = sldvrun(model, opts, true, initialCov);
```


Simulink Design Verifier Results Summary: sldvdemo_cruise_control

Progress	
Objectives processed	24/24
Satisfied	0
Unsatisfiable	0
Elapsed time	0:09

Test generation (for code generated from top model) completed normally.

24/24 objectives are satisfied - needs simulation.

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: sldvdemo_cruise_control_sldvdata2.mat
in folder: /tmp/tpcae3d36f_eroy-deb8-64/sldv_output/sldvdemo_cruise_control

Verify Complete Coverage

The `sldvruntest` function simulates the model by using the generated test suite. The `cvhtml` function produces a coverage report that indicates the final coverage of the `sldvdemo_cruise_control` model.

```
[~, finalCov] = sldvruntest(model, files.DataFile, [], true);
cvhtml('sil_final_coverage', finalCov);
close_system(model, 0);
```

```
### Starting build procedure for: sldvdemo_cruise_control
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex155
### Invoking Target Language Compiler on sldvdemo_cruise_control.rtw
### Using System Target File: B:\matlab\rtw\c\ert\ert.tlc
### Loading TLC function libraries
......
### Initial pass through model to cache user defined code
.
### Caching model source code
................................
```

```
    ### Writing header file sldvdemo_cruise_control_types.h
    ### Writing header file sldvdemo_cruise_control.h
    ### Writing header file rtwtypes.h
    ### Writing source file sldvdemo_cruise_control.c
    ### Writing header file sldvdemo_cruise_control_private.h
.
    ### Writing source file ert_main.c
### TLC code generation complete.
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_con
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk buildobj

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTER
sldvdemo_cruise_control.c
### Successfully generated all binary outputs.

### Successful completion of build procedure for: sldvdemo_cruise_control
### Simulink cache artifacts for 'sldvdemo_cruise_control' were created in 'C:\TEMP\Bdoc20a_1326
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_
### Building 'sldvdemo_cruise_control_ca': nmake  -f sldvdemo_cruise_control_ca.mk all

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\c

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\c

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\c
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DINTEGER_CODE=
sldvdemo_cruise_control_ca.c
### Creating static library ".\sldvdemo_cruise_control_ca.lib" ...
    lib /nologo -out:.\sldvdemo_cruise_control_ca.lib @sldvdemo_cruise_control_ca.rsp
### Created: .\sldvdemo_cruise_control_ca.lib
### Successfully generated all binary outputs.

### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_r
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk all

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\si

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\si

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\si
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xil_interface_lib.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xil_data_stream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xil_services.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xil_interface.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xilcomms_rtiostream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xil_rtiostream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
rtiostream_utils.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_app.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_data_stream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_rtiostream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
sil_main.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
target_io.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
rtiostream_tcpip.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
xil_instrumentation.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
codeinstr_data_stream.c
    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTERI
codeinstr_rtiostream.c
### Creating standalone executable ".\sldvdemo_cruise_control.exe" ...
    link /RELEASE  /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsock.lib advapi32.lib -out
### Created: .\sldvdemo_cruise_control.exe
### Successfully generated all binary outputs.

### Starting SIL simulation for component: sldvdemo_cruise_control
```

```
rtw.connectivity.HostLauncher: started executable with host process identifier 28160
rtw.connectivity.HostLauncher: stopped executable with host process identifier 28160
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting build procedure for: sldvdemo_cruise_control
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex155
### Generated code for 'sldvdemo_cruise_control' is up to date because no structural, parameter 
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_con
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk buildobj

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTER
sldvdemo_cruise_control.c
### Successfully generated all binary outputs.

### Successful completion of build procedure for: sldvdemo_cruise_control
### Preparing to start SIL simulation ...
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk all

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

### Creating standalone executable ".\sldvdemo_cruise_control.exe" ...
    link /RELEASE  /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsock.lib advapi32.lib -out
### Created: .\sldvdemo_cruise_control.exe
### Successfully generated all binary outputs.

### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier 39416
rtw.connectivity.HostLauncher: stopped executable with host process identifier 39416
```

```
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting build procedure for: sldvdemo_cruise_control
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex155
### Generated code for 'sldvdemo_cruise_control' is up to date because no structural, parameter 
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_con
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk buildobj

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTER
sldvdemo_cruise_control.c
### Successfully generated all binary outputs.

### Successful completion of build procedure for: sldvdemo_cruise_control
### Preparing to start SIL simulation ...
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk all

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

### Creating standalone executable ".\sldvdemo_cruise_control.exe" ...
    link /RELEASE  /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsock.lib advapi32.lib -out
### Created: .\sldvdemo_cruise_control.exe
### Successfully generated all binary outputs.

### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier 31124
rtw.connectivity.HostLauncher: stopped executable with host process identifier 31124
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting build procedure for: sldvdemo_cruise_control
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder: C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex155
### Generated code for 'sldvdemo_cruise_control' is up to date because no structural, parameter
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_con
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk buildobj

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

    cl  -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS  /Od /Oy- -DCLASSIC_INTER
sldvdemo_cruise_control.c
### Successfully generated all binary outputs.

### Successful completion of build procedure for: sldvdemo_cruise_control
### Preparing to start SIL simulation ...
### Using toolchain: Microsoft Visual C++ 2017 v15.0 | nmake (64-bit Windows)
### 'C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_
### Building 'sldvdemo_cruise_control': nmake  -f sldvdemo_cruise_control.mk all

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s

C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex15502968\sldvdemo_cruise_control_ert_rtw\s
**********************************************************************
** Visual Studio 2017 Developer Command Prompt v15.8.2
** Copyright (c) 2017 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

Microsoft (R) Program Maintenance Utility Version 14.15.26726.0
Copyright (C) Microsoft Corporation.  All rights reserved.

### Creating standalone executable ".\sldvdemo_cruise_control.exe" ...
    link /RELEASE  /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsock.lib advapi32.lib -out
### Created: .\sldvdemo_cruise_control.exe
### Successfully generated all binary outputs.

### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier 41744
rtw.connectivity.HostLauncher: stopped executable with host process identifier 41744
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
```

Summary

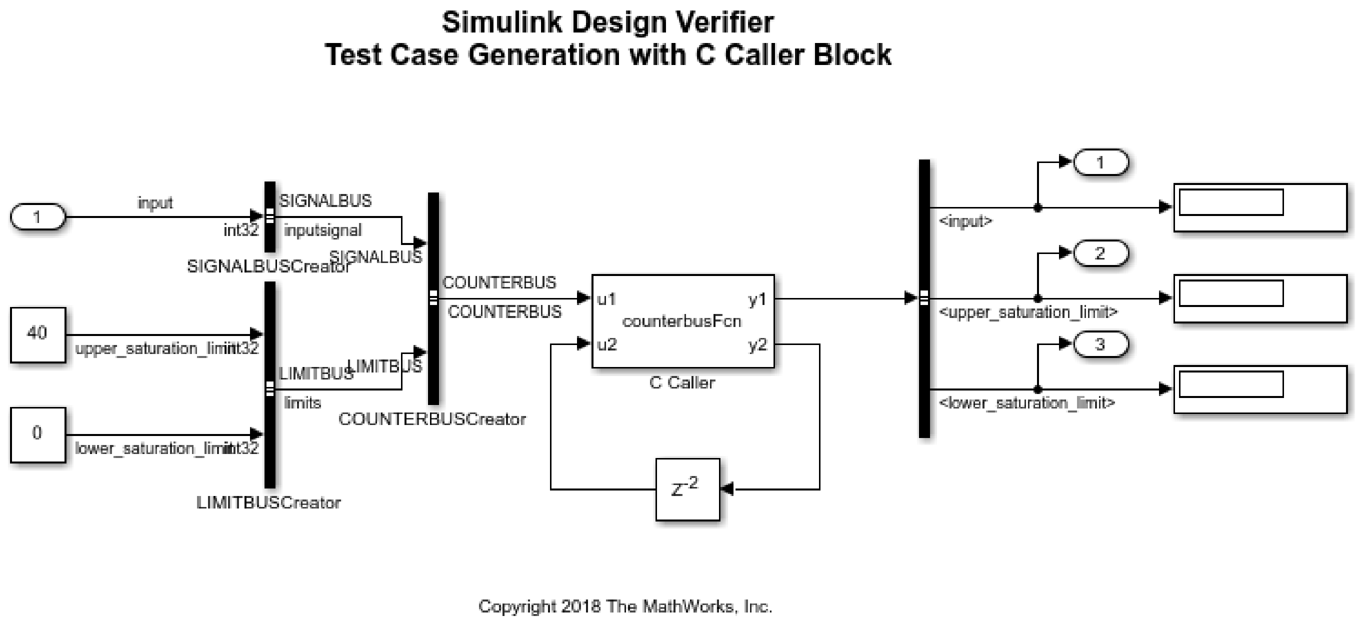
File Contents/Complexity		Test 1					
		Decision	Condition	MCDC	Statement	Function	Function call
1 . sldvdemo_cruise_control.c	10	100%	100%	100%	100%	100%	100%
2 . . . sldvdemo_cruise_control_step	8	100%	100%	100%	100%	100%	--
3 . . . sldvdemo_cruise_control_initialize	1	--	--	--	100%	100%	100%
4 . . . sldvdemo_cruise_control_terminate	1	--	--	--	100%	100%	--

Test Generation on Model with C Caller Block

This example shows how to use test generation on a model with a C Caller block and custom C code

Open the model containing the C Caller block and custom code

```
open_system('sldvexCCallerBlockExample');
```



Generate tests to ensure coverage of the model

Use the `sldvrun` function to run Simulink ® Design Verifier ™ analysis.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'ConditionDecision';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';

[status, fileNames] = sldvrun('sldvexCCallerBlockExample', opts);
```

```
Checking compatibility for test generation: model 'sldvexCCallerBlockExample'
Compiling model...done
Building model representation...done

'sldvexCCallerBlockExample' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 29-Feb-2020 11:43:21...
..........

Completed normally.

Generating output files:
```

```
Results generation completed.

    Data file:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex07804984\sldv_output\sldvexCCallerBloc
```

Verify the coverage

Use the `sldvruntest` function to verify that the test suite achieves complete model coverage.

```
[~, finalCov] = sldvruntest('sldvexCCallerBlockExample', fileNames.DataFile, [], true);
cvhtml('Final Coverage', finalCov);
```

Clean Up

To complete the example, close all models.

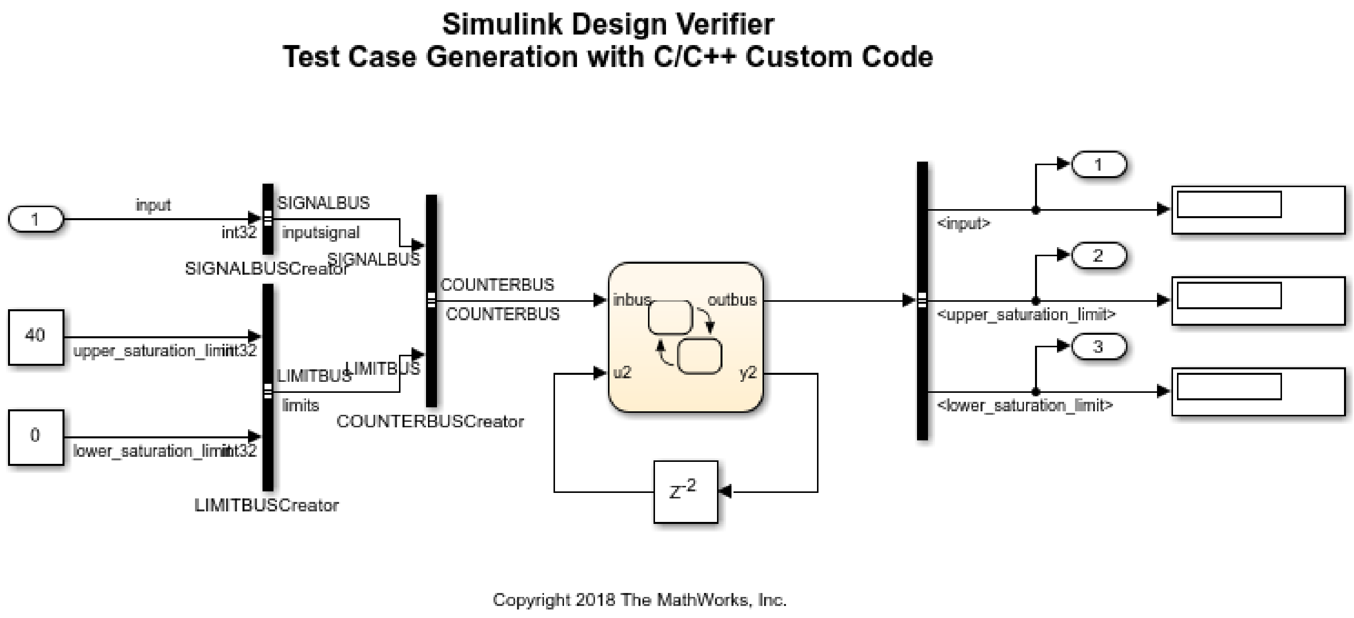
```
close_system('sldvexCCallerBlockExample', 0);
```

Test Generation for Custom Code in a Stateflow Chart

This example shows how to use test generation on a model with custom code in a Stateflow chart.

Open the model containing custom code in a Stateflow chart

```
open_system('sldvexSFCustomCodeExample');
```



Generate tests to ensure coverage of the model

Use the `sldvrun` function to run the Simulink® Design Verifier™ analysis.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'ConditionDecision';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';

[status, fileNames] = sldvrun('sldvexSFCustomCodeExample', opts);
```

```
Checking compatibility for test generation: model 'sldvexSFCustomCodeExample'
Compiling model...done
Building model representation...done

'sldvexSFCustomCodeExample' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 29-Feb-2020 11:22:37...
..........

Completed normally.

Generating output files:
```

```
Results generation completed.

    Data file:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex18712703\sldv_output\sldvexSFCustomCode
```

Verify the coverage

Use the `sldvruntest` function to verify that the test suite achieves complete model coverage.

```
[~, finalCov] = sldvruntest('sldvexSFCustomCodeExample', fileNames.DataFile, [], true);
cvhtml('Final Coverage', finalCov);
```

Clean Up

To complete the example, close all models.

```
close_system('sldvexSFCustomCodeExample', 0);
```

Extending Existing Test Cases

When to Extend Existing Test Cases

In this section...
"Common Workflow for Extending Existing Test Cases" on page 8-2
"Considerations for Starting Test Cases" on page 8-3

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 8-4.
- 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 8-10.
- 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 8-15.
- You apply parameter configurations or update the parameter constraint values of an existing model for which you have generated test cases. In this situation, you can reanalyze the model by reusing the previously generated test cases and extend them to achieve full model coverage. For an example of extending existing test cases when you modify parameter configurations, see "Extend Existing Test Cases After Applying Parameter Configurations" on page 5-35.

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.

Considerations for Starting Test Cases

If the existing test cases are inconsistent with the model, Simulink Design Verifier ignores the test cases during test case extension. For example, if you update the constraint values of parameters and the existing test case violates the specified constraint values, the test case will be ignored.

See Also

More About

- “Extend Test Cases for Model with Temporal Logic” on page 8-4
- “Extend Test Cases for Closed-Loop System” on page 8-10
- “Extend Test Cases for Modified Model” on page 8-15

Extend Test Cases for Model with Temporal Logic

In this section...

Create Starting Test Case

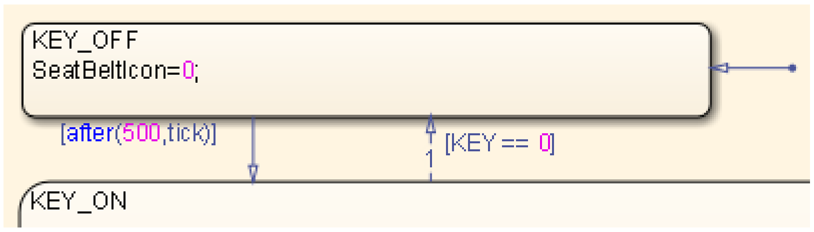
This example uses the `sldvdemo_sbr_extend_design` model. This model includes a Stateflow chart SBR that uses temporal logic. The transition from the KEY_OFF state to the KEY_ON state occurs after the Stateflow chart has been simulated 500 times. To test this transition requires a test case with 500 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.

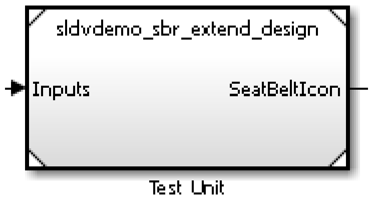


3 Create a model reference harness model:

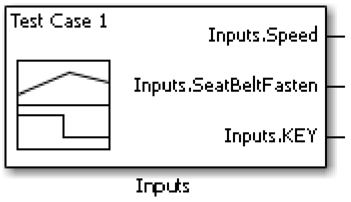
```
[~, harnessModelFilePath] = ...
    sldvmakeharness('sldvdemo_sbr_extend_design',[],[],true);
```

The harness model, `sldvdemo_sbr_extend_design_harness`, includes:

- A Model block named Test Unit that references the original model, `sldvdemo_sbr_extend_design`.

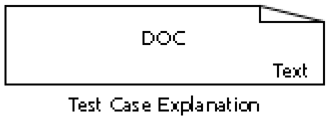


- A Signal Builder block named Inputs that contains the test-case inputs to the model referenced in the Model block.



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.



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);
```

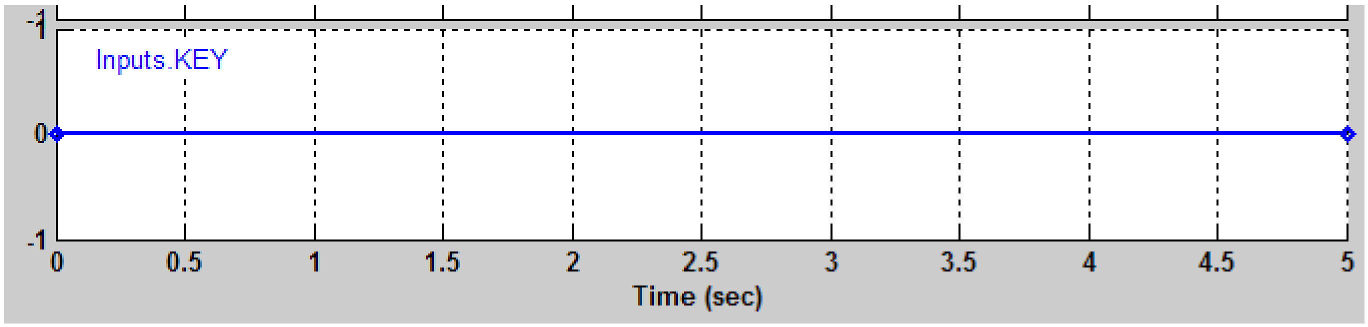
In order to analyze the `KEY_OFF` to `KEY_ON` state transition, create a test case that makes the transition to the `KEY_ON` state in 500 time steps:

1 Open the Signal Builder dialog box for the harness model.

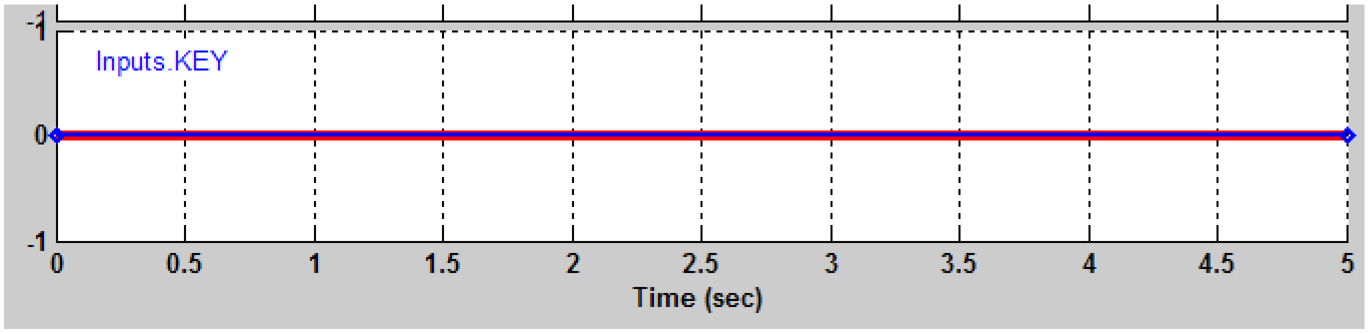
2 Select **Axes > Change Time Range**.

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.



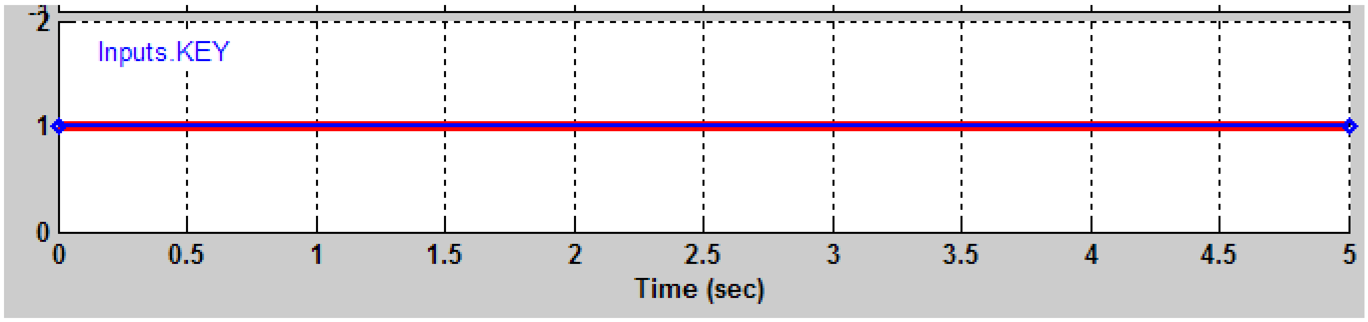
5 Click the `Inputs.KEY` signal again. The Signal Builder thickens the signal to indicate that it is selected.



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 Open the model.

```
open_system('sldvdemo_sbr_extend_design');
```

2 On the **Design Verifier** tab, click **Test Generation Settings**.

3 In the Configuration Parameters dialog box, 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**.

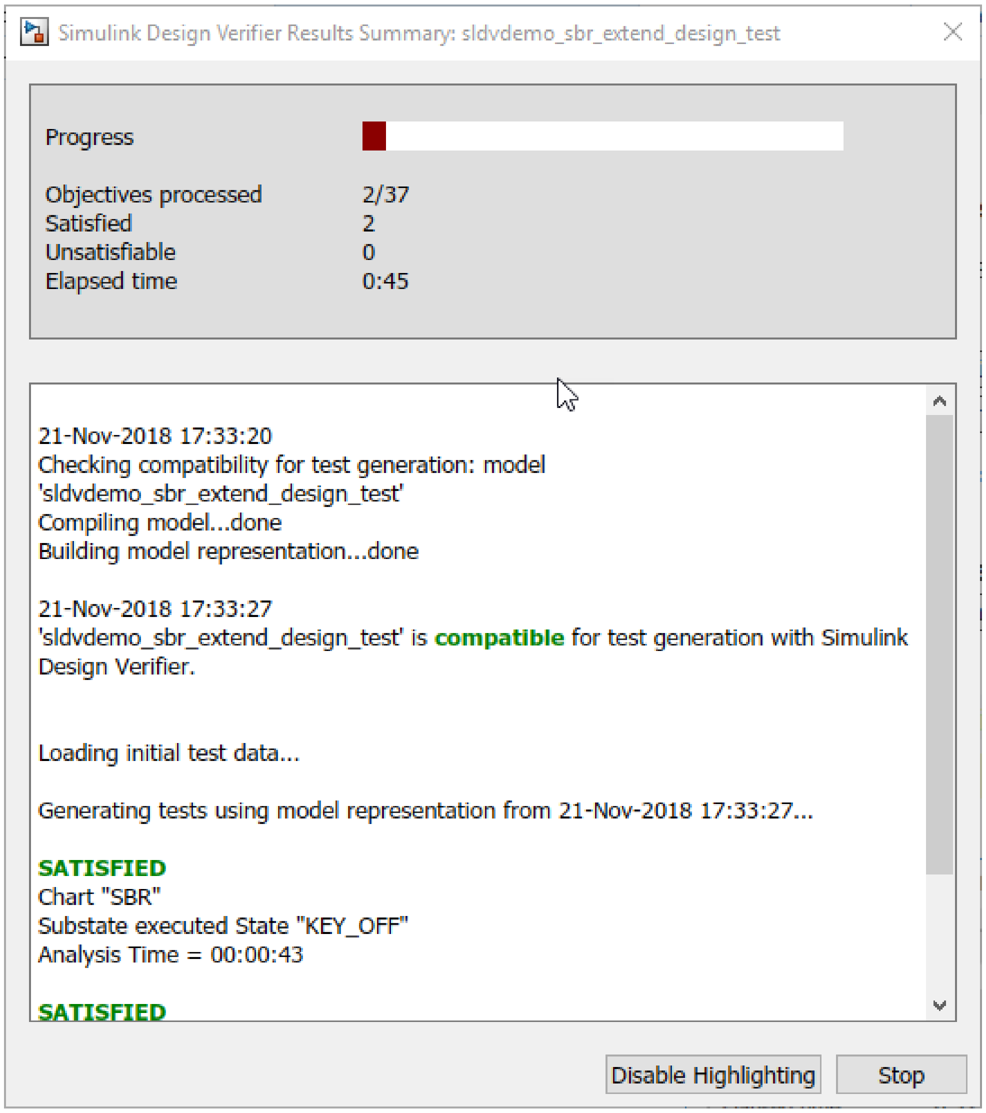
   When you clear this option, the software includes the starting test case in the final test suite. You will see that the complete test suite achieves 100% model coverage.

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 Click **Generate Tests**.

   The log window first lists the objectives that the starting test case satisfied.


Simulink Design Verifier Results Summary: sldvdemo_sbr_extend_design_test
Disable Highlighting
Stop

```
Progress

Objectives processed    2/37
Satisfied               2
Unsatisfiable           0
Elapsed time            0:45
```

```
21-Nov-2018 17:33:20
Checking compatibility for test generation: model
'sldvdemo_sbr_extend_design_test'
Compiling model...done
Building model representation...done

21-Nov-2018 17:33:27
'sldvdemo_sbr_extend_design_test' is compatible for test generation with Simulink
Design Verifier.


Loading initial test data...

Generating tests using model representation from 21-Nov-2018 17:33:27...

SATISFIED
Chart "SBR"
Substate executed State "KEY_OFF"
Analysis Time = 00:00:43

SATISFIED
```

The log window then lists the objectives generated beyond the starting test case.

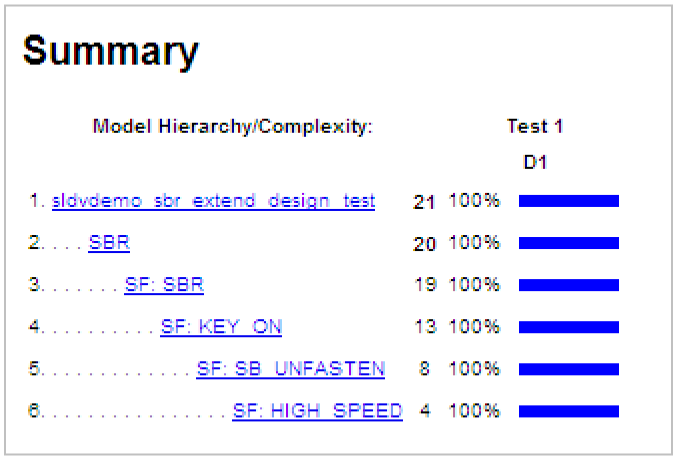
Verify Analysis Results

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. On the **Design Verifier** tab, in the **Review Results** section, click **Create Test 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 ▶all.

   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.



Summary

Model Hierarchy/Complexity:		Test 1 D1	
1. sldvdemo_sbr_extend_design_test	21	100%	▬
2. . . . SBR	20	100%	▬
3. . . . . . . SF: SBR	19	100%	▬
4. . . . . SF: KEY_ON	13	100%	▬
5. . . . . SF: SB_UNFASTEN	8	100%	▬
6. . . . . SF: HIGH_SPEED	4	100%	▬

See Also

More About

- “When to Extend Existing Test Cases” on page 8-2
- “Extend Test Cases for Closed-Loop System” on page 8-10
- “Extend Test Cases for Modified Model” on page 8-15

Extend Test Cases for Closed-Loop System

In this section...

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. On the **Apps** tab, click the arrow on the right of the **Apps** section.

   Under **Model Verification, Validation, and Test**, click **Coverage Analyzer**.
3. On the **Coverage** tab, click **Settings**.
4. In the **Coverage** pane of the Configuration Parameters, select **Enable coverage analysis**.
5. 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.
6. Under **Coverage metrics**, set the structural coverage level to **Modified Condition Decision Coverage (MCDC)** to record decision, condition, and modified condition/decision coverage.
7. Click **OK**.
8. Click **Analyze Coverage**.

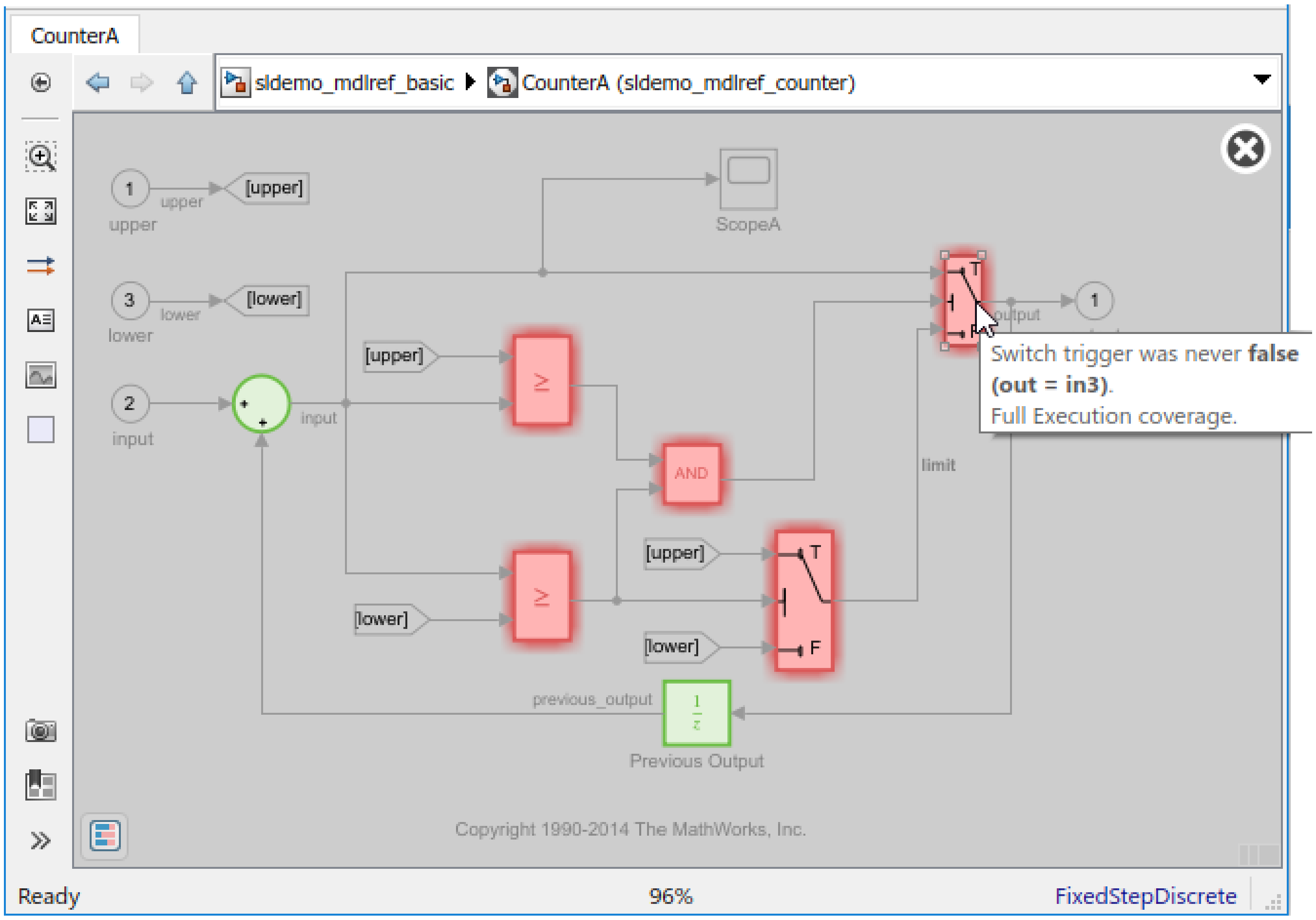
   To open the coverage report, in the **Review Results** section, click **Generate Report**.

   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.



Switch trigger was never **false** **(out = in3)**.
Full Execution coverage.

9 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');
```

10 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

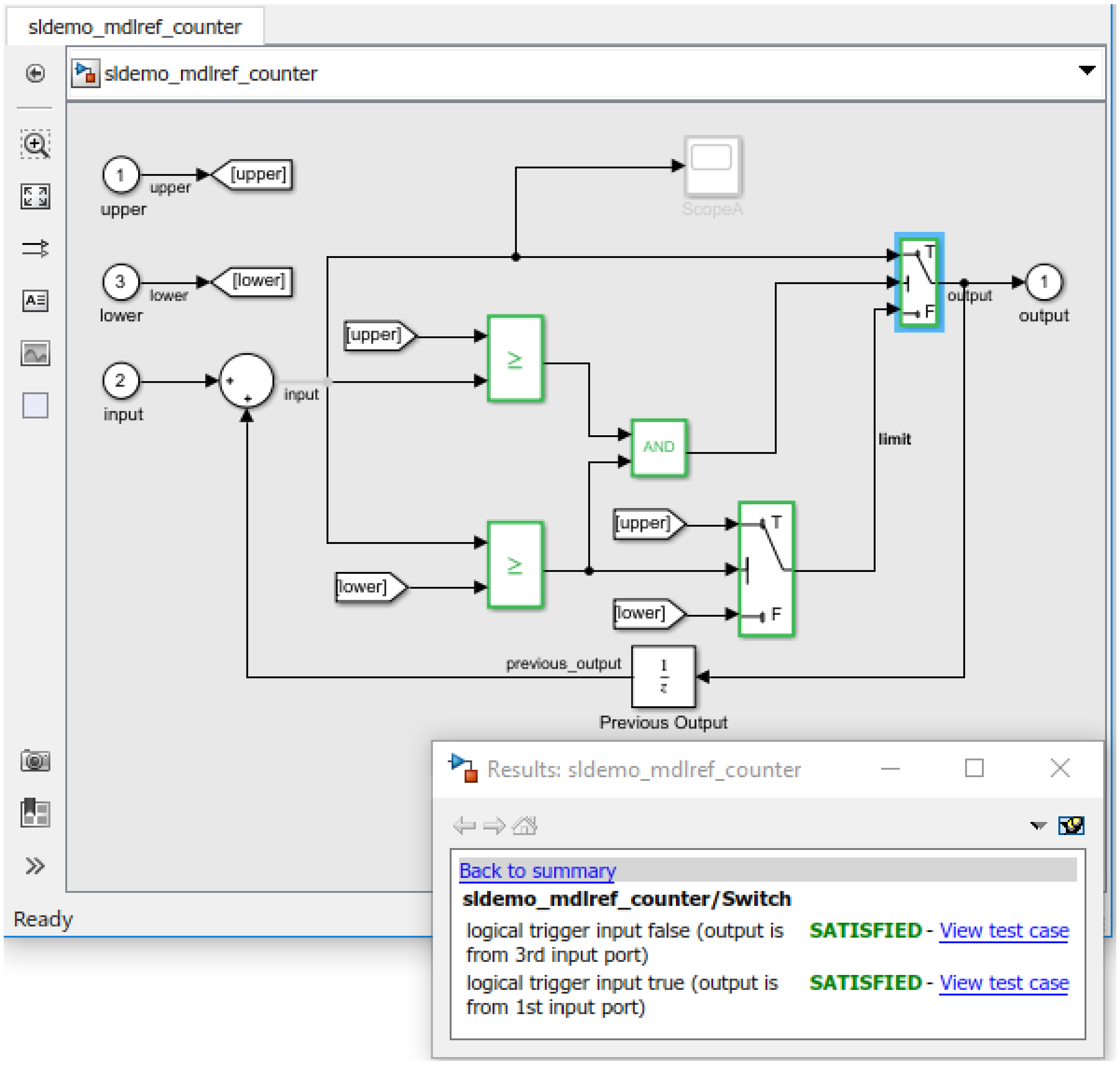
Analyze the `sldemo_mdfref_counter` model, specifying that the analysis extend the test cases already satisfied:

1. To open the `sldemo_mdfref_counter` model, in the `sldemo_mdlref_basic` model, double-click the CounterA Model block.
2. On the **Design Verifier** tab, click **Test Generation Settings**.
3. In the Configuration Parameters dialog box, on the **Test Generation** pane, in the **Model coverage objectives** box, select `MCDC`.
4. Under **Existing test cases**, select **Extend existing test cases**.
5. In the **Data file** field, specify the name of the MAT-file that contains the logged data, in this case, `existingtestcase.mat`.
6. 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.
7. Click **OK**.
8. Click **Generate Tests**.

   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.
9. 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.



Back to summary

sldemo_mdlref_counter/Switch

logical trigger input false (output is from 3rd input port) **SATISFIED** - View test case

logical trigger input true (output is from 1st input port) **SATISFIED** - View test case

10 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**.

11 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

See Also

More About

- “When to Extend Existing Test Cases” on page 8-2
- “Extend Test Cases for Model with Temporal Logic” on page 8-4
- “Extend Test Cases for Modified Model” on page 8-15

Extend Test Cases for Modified Model

In this section...

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.

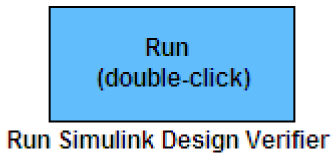
This example uses the `sldvdemo_cruise_control` model. You analyze the model and generate test cases. Then you analyze a modified version of that model, `sldvdemo_cruise_control_mod`, extending the test cases from the original analysis. The analysis returns a complete test suite for the new model.

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.

   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

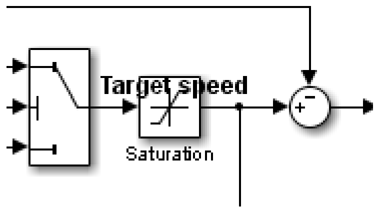
The `sldvdemo_cruise_control_mod` model is a modified version of `sldvdemo_cruise_control`. The Controller subsystem contains a Saturation block that specifies that the target speed cannot exceed 70.

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 On the **Design Verifier** tab, click **Test Generation Settings**.

5 In the Configuration Parameters dialog box, on the **Test Generation** pane, under **Existing test cases**, select **Extend existing test cases**.

6 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
```

7 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.

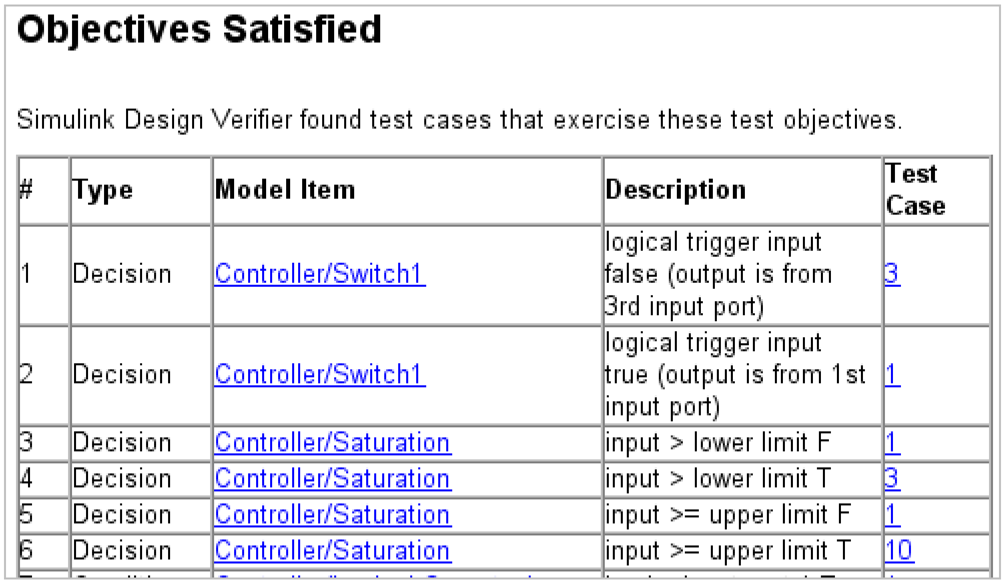
8 Click **Apply** to save these settings.

9 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.

10 In the Results Summary 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

See Also

More About

- “When to Extend Existing Test Cases” on page 8-2
- “Extend Test Cases for Model with Temporal Logic” on page 8-4
- “Extend Test Cases for Closed-Loop System” on page 8-10

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 9-11

See Also

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

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. On the **Apps** tab, click the arrow on the right of the **Apps** section.

   Under **Model Verification, Validation, and Test**, click **Coverage Analyzer**.
3. On the **Coverage** tab, click **Settings**.
4. 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**.
5. 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.

6 Click **OK**.

7 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.

8 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`.

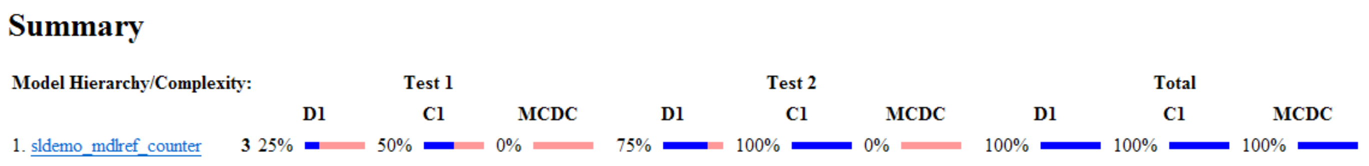
   ```
   [~, covdata_missing] = sldvruntest('sldemo_mdlref_counter', fileNames.DataFile, runOpts);
   ```

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. The **Design Verifier** app is not available for test harness models when the test unit is copied from the top model. For more information see, "Reference input model in generated harness" on page 15-59.

Generate Test Harness Model and Record Coverage Data

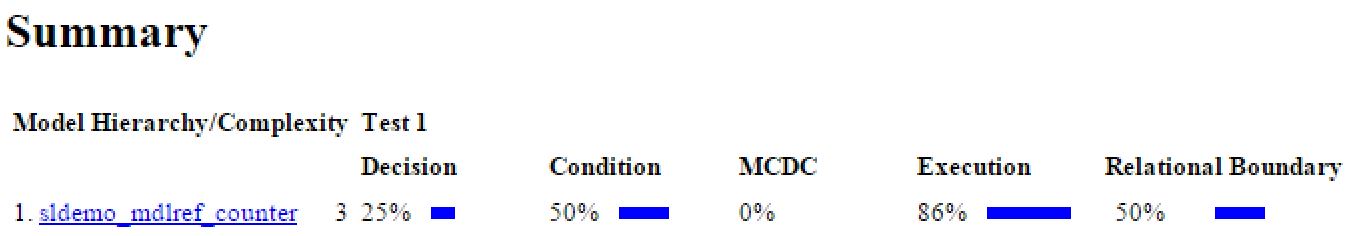
To achieve full coverage for the `sldemo_mdlref_counter` model, run a missing coverage analysis on the Simulink Design Verifier generated harness model.

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 13-13.
3. In the harness model `sldemo_mdlref_counter_harness`, the **Format** parameter must be `Dataset` to make 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" (Simulink).
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

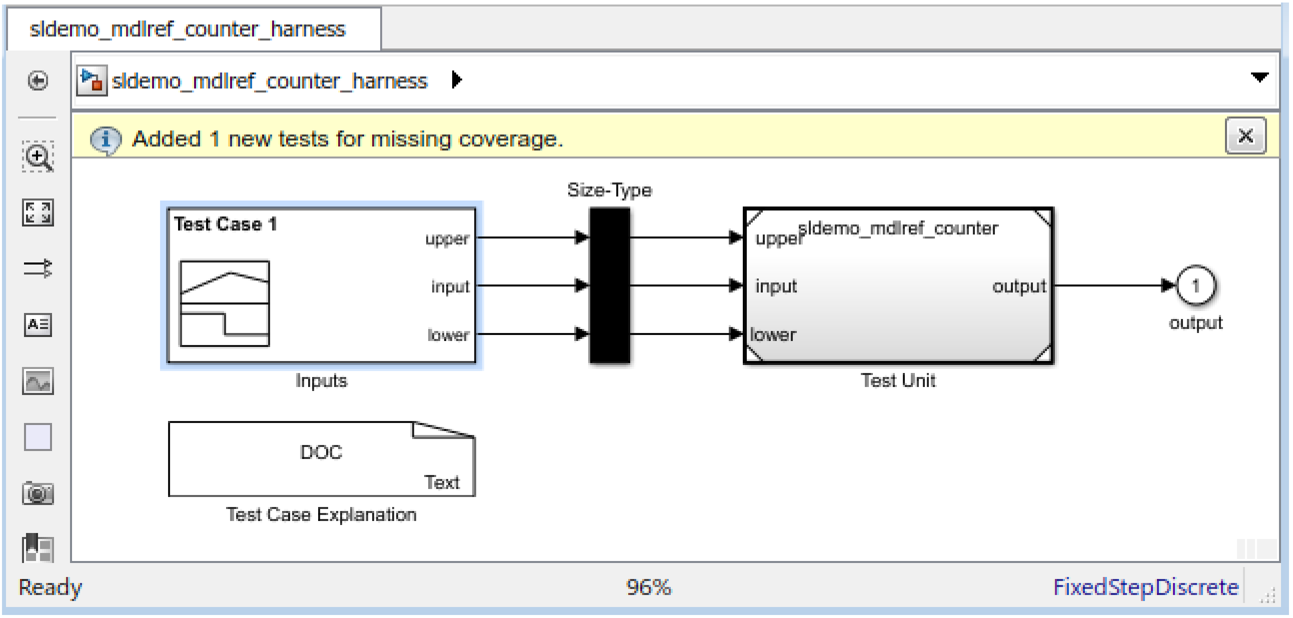
Model Hierarchy/Complexity		Test 1 Decision	Condition	MCDC	Execution	Relational Boundary
1. sldemo_mdlref_counter	3	25%	50%	0%	86%	50%

Generate Test Cases for the Missing Coverage

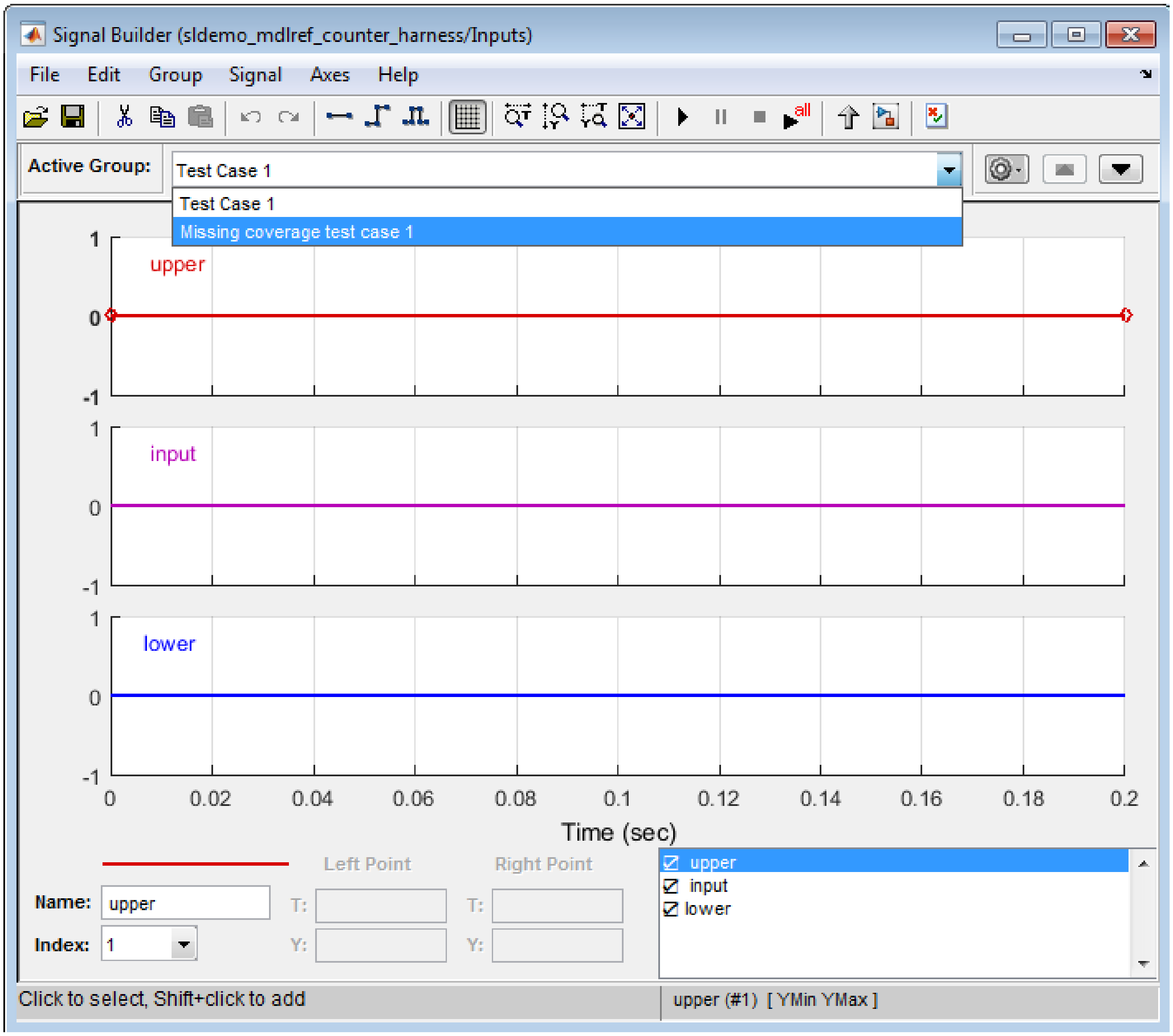
1. Open the harness model:

   ```
   open_system('sldemo_mdlref_counter_harness');
   ```

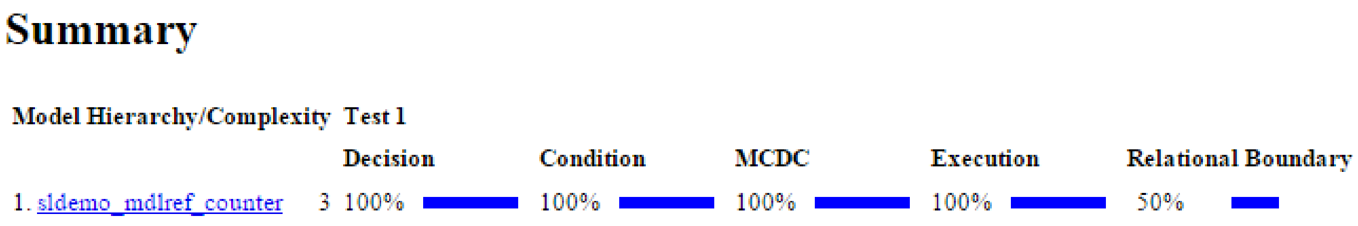
   To generate test cases for the missing coverage, on the **Design Verifier** tab, click **Add Missing Coverage**. A notification indicates the number of new tests that are added.



2 The Signal Builder dialog box shows the **Missing coverage test case 1** added to the previous **Test Case 1**.


Signal Builder (sldemo_mdlref_counter_harness/Inputs)
File Edit Group Signal Axes Help
Active Group: Test Case 1
Test Case 1
Missing coverage test case 1
upper
input
lower
Time (sec)
Left Point
Right Point
Name: upper
Index: 1
T:
Y:
upper
input
lower
Click to select, Shift+click to add
upper (#1) [YMin YMax]

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

   ```
   open_system('sldemo_mdlref_counter_harness');
   ```

   On the **Design Verifier** tab, click **Test Generation Settings**. 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.

   ```
   open_system('sldemo_mdlref_counter');
   ```

   On the **Design Verifier** tab, in the **Review Results** section, click **Load Earlier Results**. Browse to the previously generated data file and click **Open**.

   To view active results for missing coverage test cases, click **Results Summary**. The Results Summary window opens with the missing coverage analysis results. For more information on active results, see “Review Analysis Results” on page 13-47. 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 13-7.

Limitations

1. Missing Coverage analysis is a user interface-based workflow. Command-line functions are not 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.

See Also

More About

- “Generate Test Cases for Missing Coverage Data” on page 9-2
- “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11

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 Subsystems to Referenced Models” (Simulink) 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 9-11.

See Also

More About

- “Achieve Missing Coverage in Referenced Model” on page 9-3
- “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11

Achieve Missing Coverage in Closed-Loop Simulation Model

In this section...
"Record Coverage Data for the Model" on page 9-11
"Find Test Cases for Missing Coverage" on page 9-12

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. On the **Apps** tab, click the arrow on the right of the **Apps** section.

   Under **Model Verification, Validation, and Test**, click **Coverage Analyzer**.
3. On the **Coverage** tab, click **Settings**.
4. 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**.
5. Under **Coverage metrics**, select `Modified Condition Decision Coverage (MCDC)`.
6. Clear the **Other metrics** if they are selected.
7. 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**.
8. Click **OK**.
9. 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.

10. 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)`.

See Also

More About

- “Generate Test Cases for Missing Coverage Data” on page 9-2
- “Achieve Missing Coverage in Referenced Model” on page 9-3

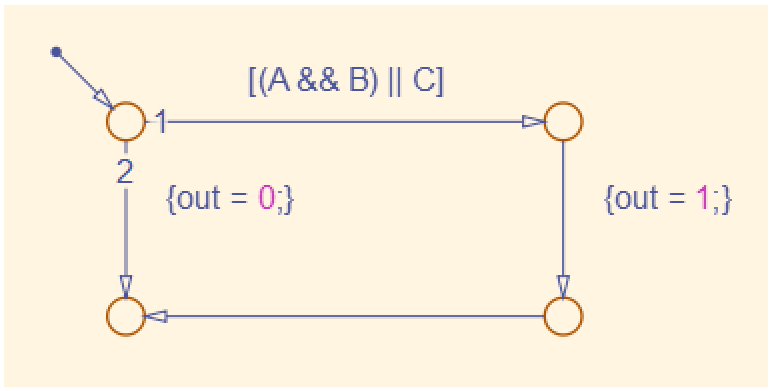
Modified Condition and Decision Coverage in Simulink Design Verifier

Depending on the settings you apply for Simulink Coverage coverage recording, there can be a difference between the definition of modified condition and decision (MCDC) coverage used for model coverage analysis in Simulink Coverage and that used for test case generation analysis in Simulink Design Verifier.

MCDC Definitions for Simulink Coverage and Simulink Design Verifier

Simulink Design Verifier always uses the masking MCDC definition for test case generation. By default, Simulink Coverage also uses the masking MCDC definition when recording coverage. However, if you set the `CovMcdcMode` model configuration parameter to `'UniqueCause'`, Simulink Coverage instead uses the unique-cause MCDC definition when recording coverage. For information on the differences between the masking MCDC definition and the unique-cause MCDC definition, see “Modified Condition and Decision Coverage (MCDC) Definitions in Simulink Coverage” (Simulink Coverage).

Setting the `CovMcdcMode` model configuration parameter to `'UniqueCause'` can result in differences between MCDC reporting in Simulink Coverage and test generation in Simulink Design Verifier. An example of this difference can be seen in analysis results for logical expressions containing a mixture of `AND` and `OR` operators, as in this Stateflow transition.



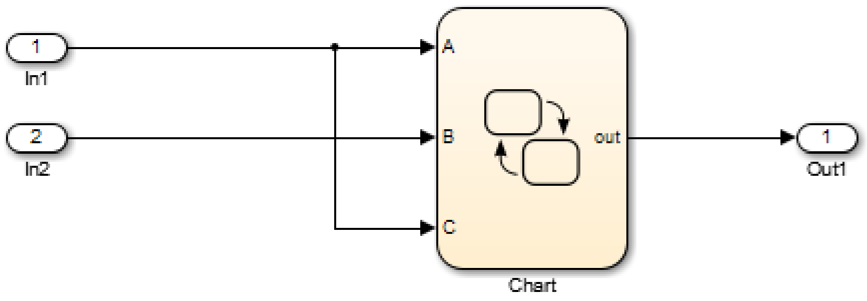
Given that `A`, `B`, and `C` are each separate inputs, there are five possible ways to evaluate the condition on the Stateflow transition, shown in the following table.

	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, Simulink Design Verifier test generation analysis accepts not only pair 1, 2 and pair 3, 4 but also pair 1, 4 and pair 2, 3. Simulink Coverage model coverage analysis using the unique-cause MCDC definition 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, unique-cause MCDC for `C` can no longer be satisfied in Simulink Coverage 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 Coverage set to unique-cause MCDC analysis and Simulink Design Verifier. The default`CovMcdcMode` model configuration parameter value of `'Masking'` does not cause this discrepancy. However, if you require the use of unique-cause MCDC analysis in Simulink Coverage, you can minimize this effect by using the `IndividualObjectives` test suite optimization for test generation analysis in Simulink Design Verifier For more information, see the Tip section of “Test suite optimization” on page 15-35.

See Also

More About

- “MCDC” on page 7-23

Verifying Model Components

What Is Component Verification?

In this section...

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 13-13.
`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” (Embedded Coder).

Component verification functions do not support the following Simulink features:

- Variable-step solvers for `sldvruntest`
- Component interfaces that contain:
  - Variable-size signals
  - Multiword fixed-point data types larger than 128 bits

Verify a Component for Code Generation

In this section...

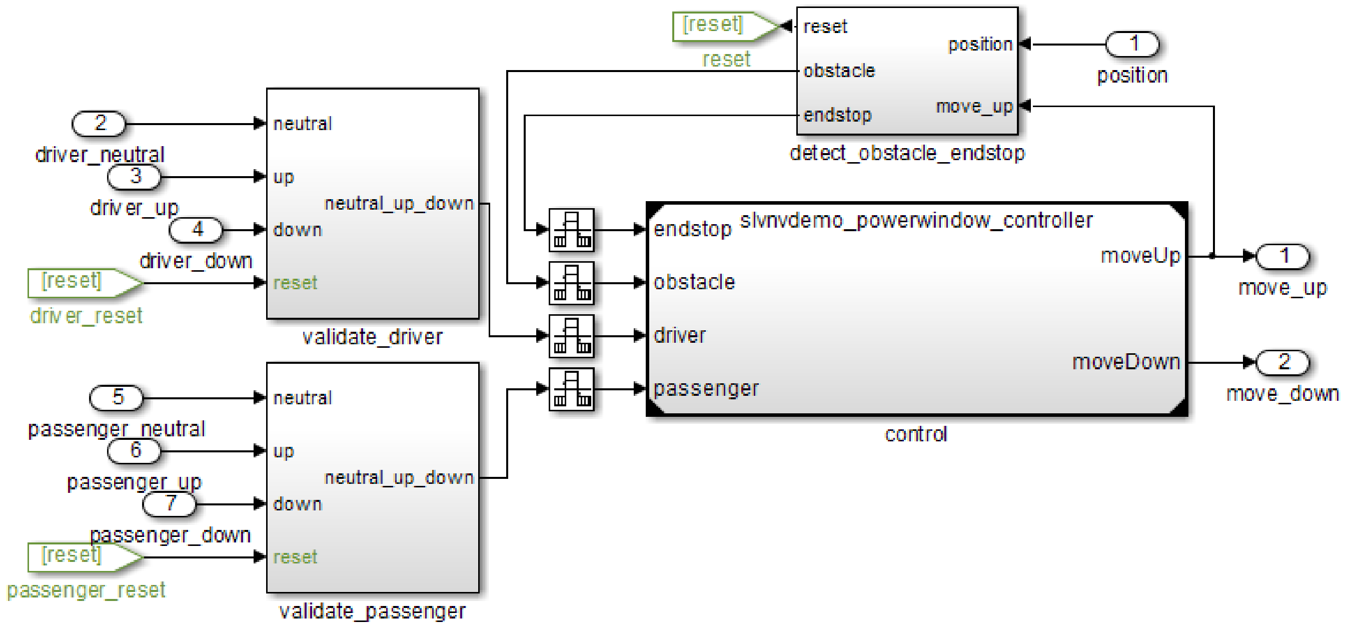
About the Example Model

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.

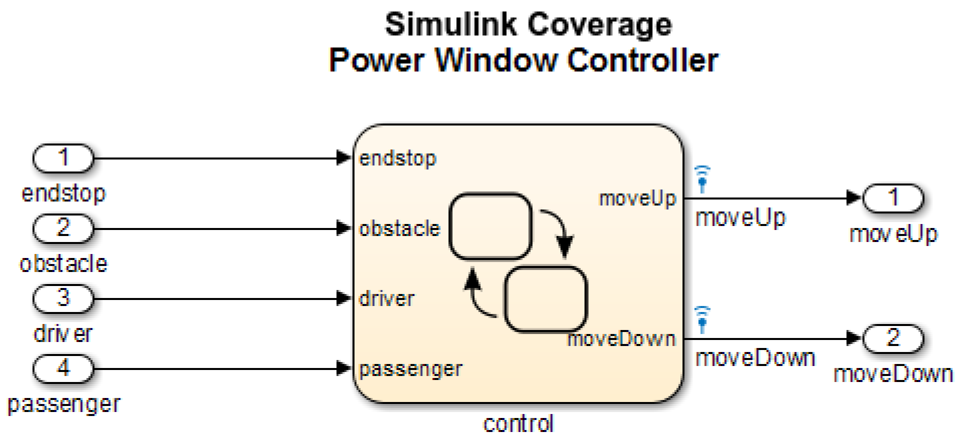
Note You must have the following product licenses to run this example:

- Stateflow
- Embedded Coder
- Simulink Coder

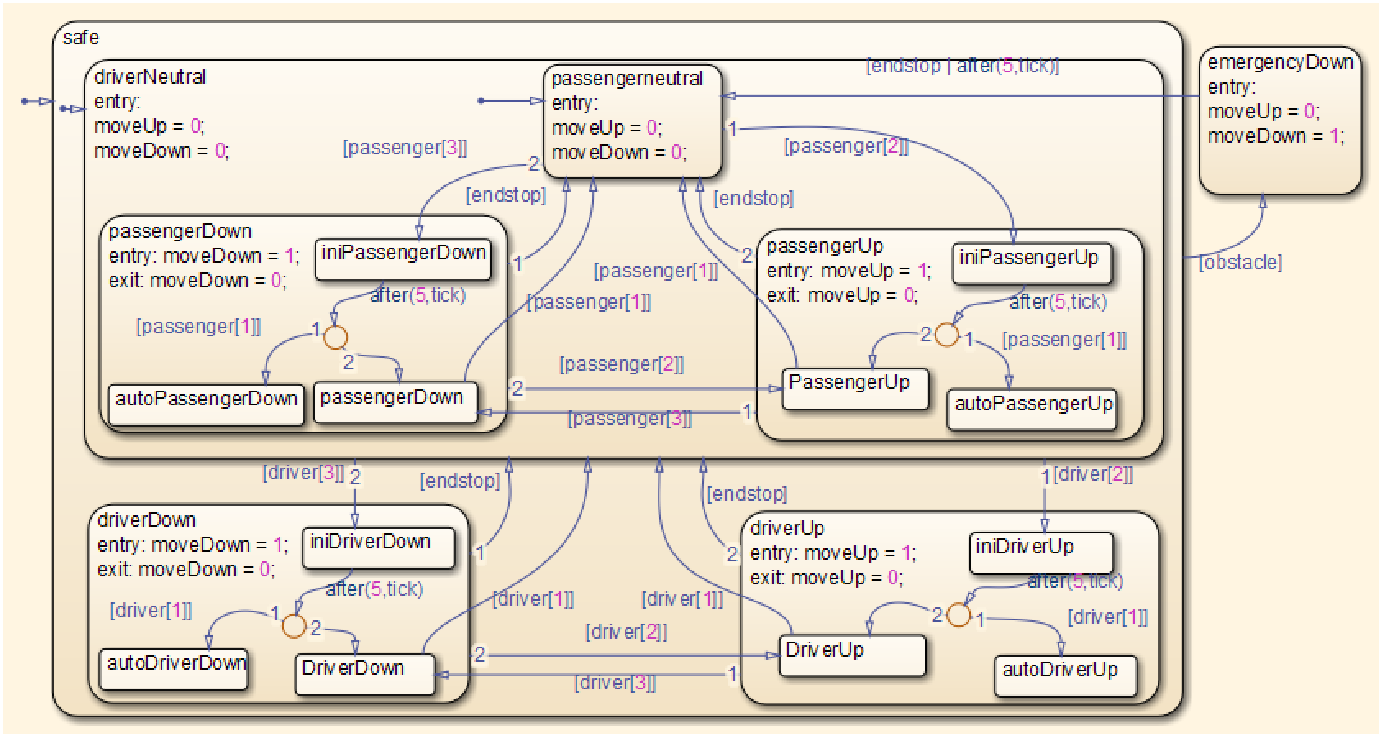
The component that you verify is a Model block named `control`. This component resides inside the `power_window_control_system` subsystem in the top level of the `slvnvdemo_powerwindow` model. The `power_window_control_system` subsystem is shown below.



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 13-13.

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” (Simulink Coverage).

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 10-6 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 10-6, 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” (Embedded Coder).

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” (Embedded Coder).

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` 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?” (Embedded Coder).

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 11-2

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 "Investigate Signal Values" (Simulink).

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. On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
2. In the Configuration Parameters dialog box, 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.

See Also

More About

- “Specify Signal Ranges” (Simulink)

Specify Input Ranges on Simulink and Stateflow Elements

When you specify input range constraints on Simulink and Stateflow elements, Simulink Design Verifier considers these constraints during analysis.

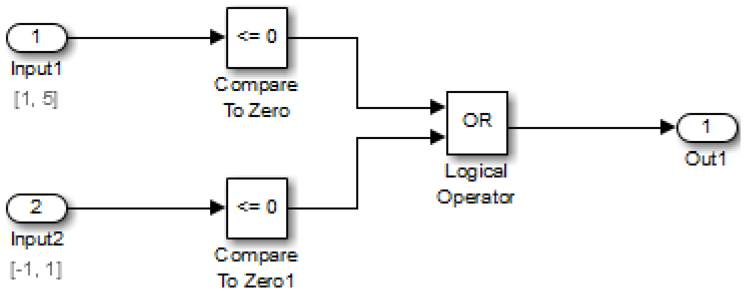
In this section...

Specify Input Ranges for Inport Blocks

After you specify the output minimum and maximum values on Inport blocks (Simulink), Simulink Design Verifier analysis uses the minimum and maximum values as constraints.

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 use Simulink Design Verifier, to analyze this model, the analysis produces these results:

- The output from Input1 is never less than 0, therefore the first input to the Logical Operator block is never `false`. The objective that the first input to the Logical Operator equals `false` is unsatisfiable.
- The Logical Operator block cannot achieve 100% modified condition/decision coverage (MCDC) coverage because the condition where the first input is `false` never occurs.

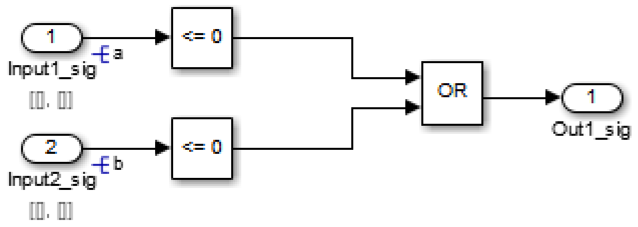
The detailed analysis report shows the values you use as constraints for Input1 and Input2.

Specify Input Ranges for Simulink.Signal Objects

Using the Model Explorer, in the model workspace, you can specify minimum and maximum values (Simulink) 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.

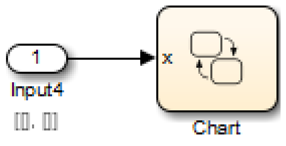
Specifying Signal Ranges on Inport Blocks and Signals

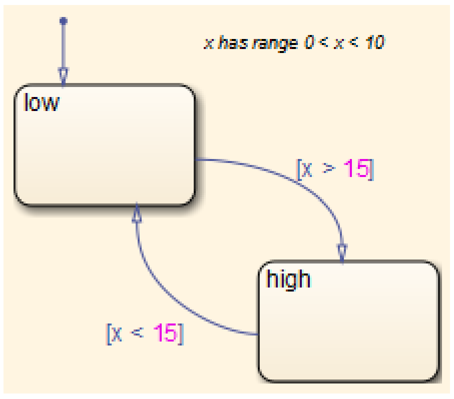
If you specify ranges on the Inport blocks and on the signals, the analysis considers the smallest range for the values. For example, if you specify a range of `4..12` on an input port and a range of `1..8` on the signal from the input port, the analysis considers the range `4..8`.

Specify Input Ranges 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 (Simulink) 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 ranges from 0 through 10, therefore 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, these 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`.

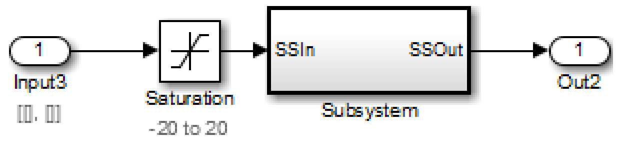
The analysis report indicates the values that you use as constraints for `x`: [0, 10].

Specify Input Ranges for Subsystems

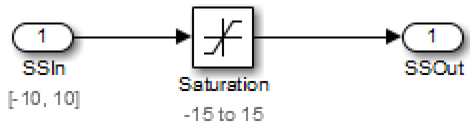
The Simulink Design Verifier software considers specified input minimum and maximum values as constraints only at the top level of a model. You can specify minimum and maximum values on Input ports (Simulink) on subsystems, but when you analyze the top-level model, the software ignores those values.

When you perform the subsystem analysis, the software considers specified minimum and maximum values on the input ports of the subsystem.

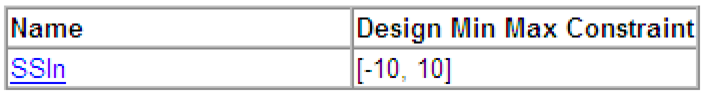
For example, consider the following model and its subsystem.



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.



Name	Design Min Max Constraint
SSIn	[-10, 10]

Constraints: Design Min Max Constraints

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 on the Saturation block (15).

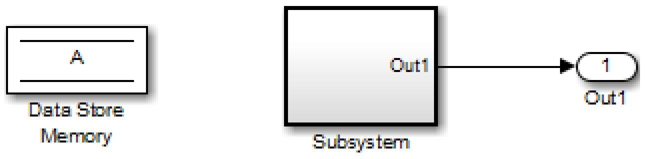
If you analyze the model that contains Subsystem, the analysis does not consider the values specified on the input port SSIn in the subsystem. The analysis considers only the root-level input ports at the respective level of the hierarchy for analysis.

Specify Input Ranges for Global Data Stores

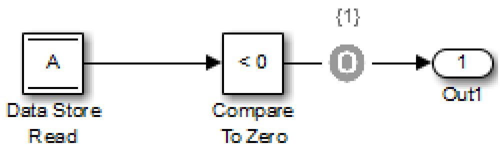
A data store is a repository to which you can write data and from which you can read data, without having to connect an input or output signal directly to the data store. You create a data store by using a Data Store Memory block or a `Simulink.Signal` object. You can specify minimum and maximum values (Simulink) for any data store.

During subsystem analysis, Simulink Design Verifier creates an input port to mimic the execution context for a global data store. For more information, see “Extract Subsystems for Analysis” on page 14-14. If the data store has specified minimum and maximum values, those values are assigned as minimum and maximum values on the new input port. Simulink Design Verifier analysis considers the input minimum and maximum values 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 values 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.



In the top-level model, if you right-click Subsystem and select **Design Verifier** > **Generate Tests for Subsystem**, the analysis considers the constraints for data store A to be [0, 10].

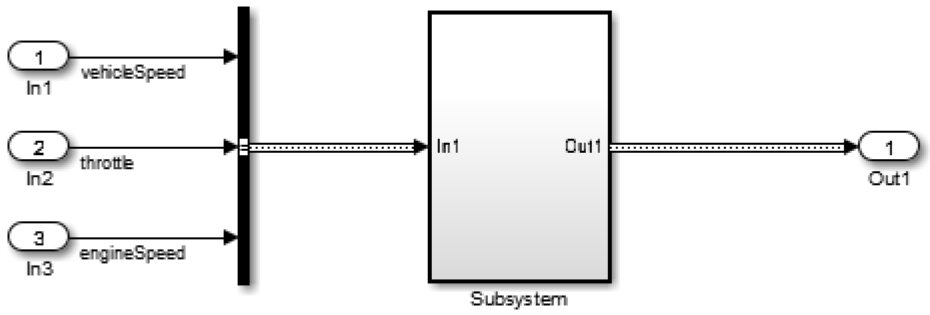
The analysis does not satisfy the objective specified in the Test Objective block. The input is always greater than or equal to 0, therefore the output from the Compare To Zero block is always 0.

Specify Input Ranges for Bus Elements

When you define a bus, you can specify minimum and maximum values for the elements in the bus (Simulink). Simulink Design Verifier considers these minimum and maximum values when analyzing subsystems and models that use the bus as an input signal.

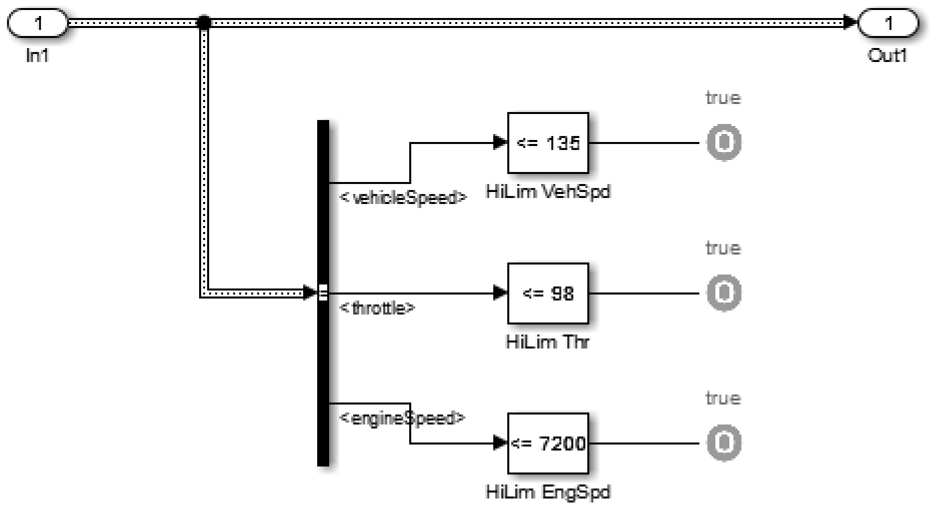
Consider a subsystem that inputs a bus of three fields, each with a defined minimum and maximum. To view this subsystem, at the command line, enter:

```
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 that confirm 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. In the top-level mode, right-click Subsystem 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.

Specify 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 Ranges 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
  ```

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" (Simulink).

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.

See Also

More About

- “Property Proving Workflow for Cruise Control” on page 12-32
- “Property Proving Workflow for Fixed-Point Cruise Control with Block Replacements” on page 12-36
- “Property Proving Workflow for Thrust Reverser” on page 12-39

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 12-5
2	Verify that your model is compatible with Simulink Design Verifier.	"Check Compatibility of Example Model" on page 12-6
3	Add a Proof Objective block to your model to prepare for its proof.	"Instrument Example Model" on page 12-7
4	Configure Simulink Design Verifier to prove properties.	"Configure Property-Proving Options" on page 12-8
5	Prove a property of your model.	"Analyze Example Model" on page 12-8
6	Review the analysis results.	"Review Analysis Results" on page 12-8
7	Add proof assumptions to specify analysis constraints.	"Customize Example Proof" on page 12-15
8	Prove a property of the customized model and interpret the results.	"Reanalyze Example Model" on page 12-16

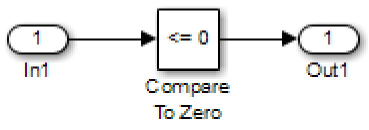
Construct Example Model

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 On the **Modeling** tab, click **Model Settings**.
5 On the Configuration Parameters dialog box, in the **Solver** pane, in the **Solver selection**:

   - 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

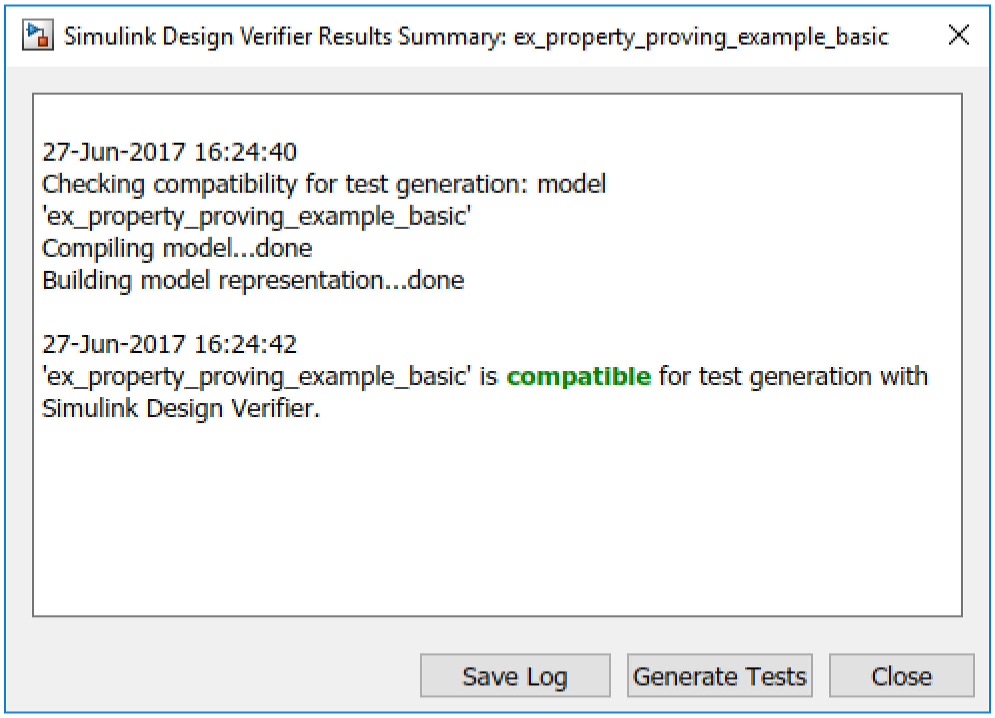
Every time Simulink Design Verifier software analyzes a model, before the analysis begins, the software performs a compatibility check. If your model is not compatible, the software cannot analyze it.

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 On the **Design Verifier** tab, click **Check Compatibility**.

   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.


Simulink Design Verifier Results Summary: ex_property_proving_example_basic
Save Log
Generate Tests
Close

```
27-Jun-2017 16:24:40
Checking compatibility for test generation: model
'ex_property_proving_example_basic'
Compiling model...done
Building model representation...done

27-Jun-2017 16:24:42
'ex_property_proving_example_basic' is compatible for test generation with
Simulink Design Verifier.
```

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

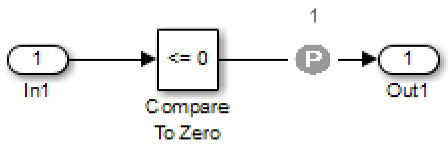
Prepare your example model so that you can prove its properties with Simulink Design Verifier. Specifically, instrument the model by adding and configuring a Proof Objective block:

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. On the **Design Verifier** tab, in the **Mode** section, select **Property Proving**.
3. Click **Property Proving Settings**.
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 15-50.

5. Save the `ex_property_proving_example_basic` model.

Analyze Example Model

To analyze the `ex_property_proving_example_basic` model, on the **Design Verifier** tab, click **Prove Properties**. Simulink Design Verifier 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 13-7.

The following sections describe how you can review the analysis results:

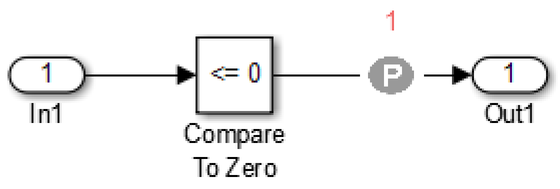
Review Results on Model

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.

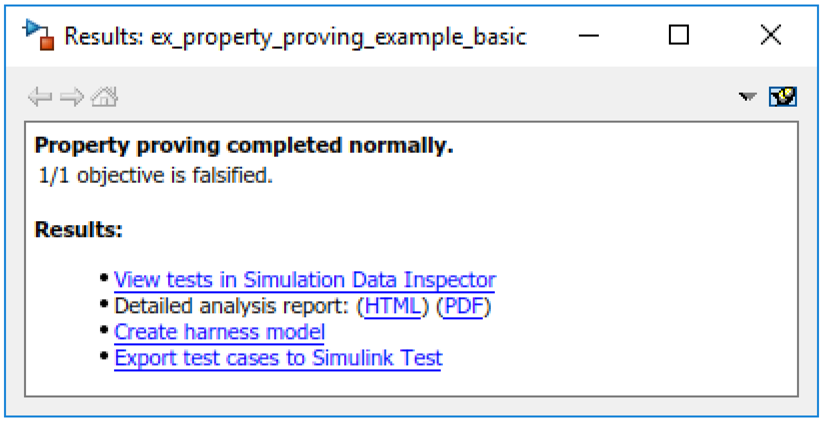
Highlight the analysis results on the example model:

1. In the Results Summary 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.

   The Simulink Design Verifier Results window appears.

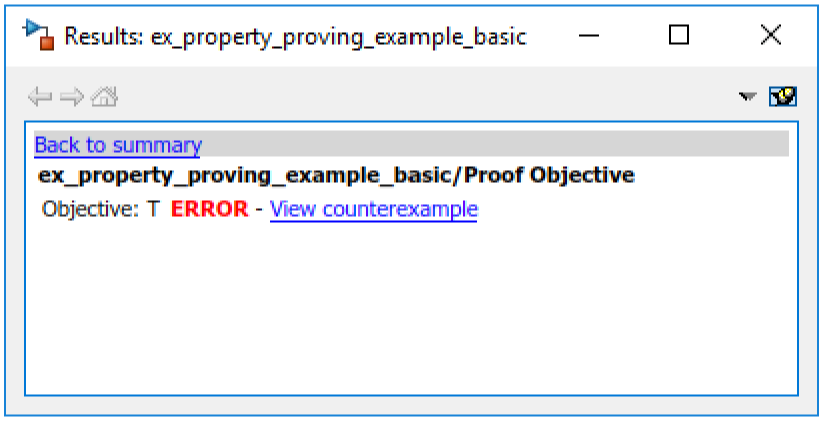

Results: ex_property_proving_example_basic

Property proving completed normally.
1/1 objective is falsified.

Results:

- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test

As you click objects in the model, this window changes to display detailed analysis results for that object.


Results: ex_property_proving_example_basic

Back to summary
ex_property_proving_example_basic/Proof Objective
Objective: T **ERROR** - View counterexample

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 Results Summary 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.

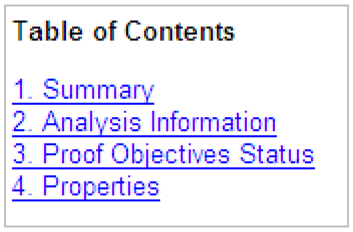
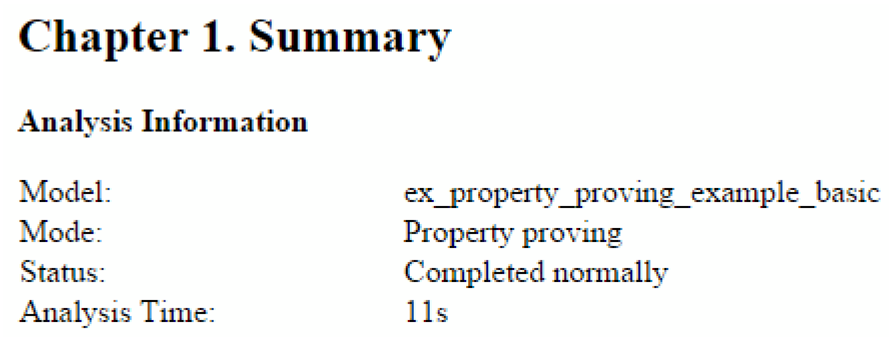


Table of Contents

1. Summary
2. Analysis Information
3. Proof Objectives Status
4. Properties

3 In the **Table of Contents**, click `Summary`.



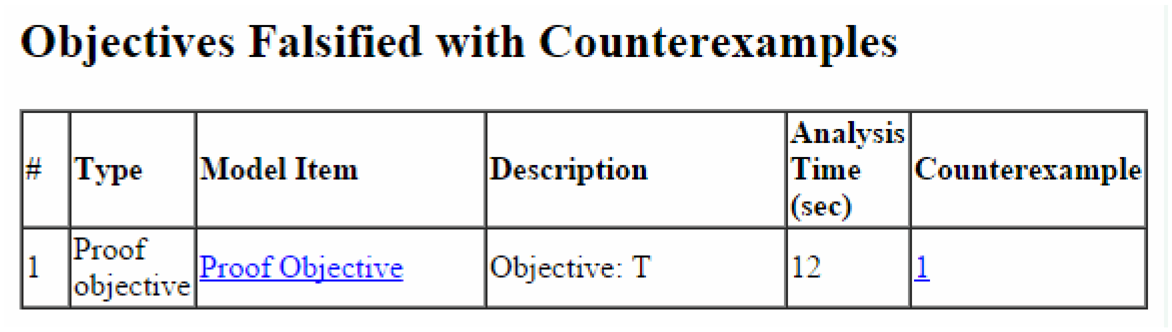
Chapter 1. Summary

Analysis Information

Model:	ex_property_proving_example_basic
Mode:	Property proving
Status:	Completed normally
Analysis Time:	11s

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 In the **Table of Contents**, click `Proof Objectives Status`.

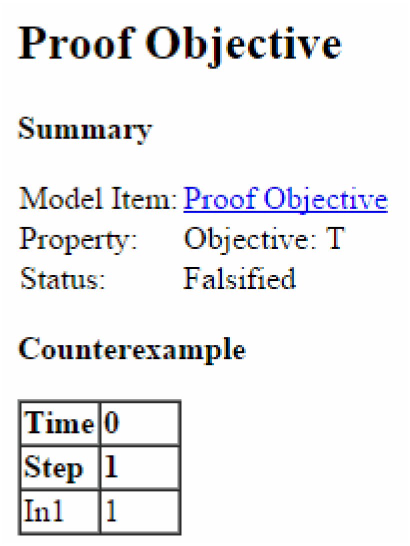


Objectives Falsified with Counterexamples

#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Proof Objective	Objective: T	12	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: T
Status:	Falsified

Counterexample

Time	0
Step	1
In1	1

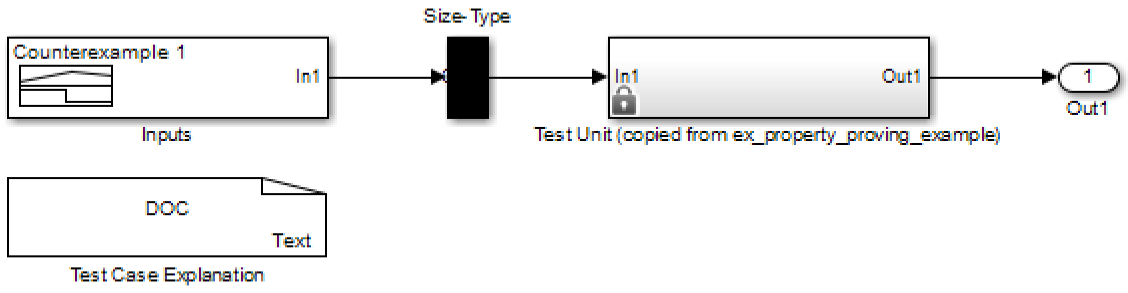
This section displays information about proof objective 1 and provides details about the counterexample that Simulink Design Verifier generated to disprove that objective. In this counterexample, a signal value of 99 falsifies the objective that you specified using the Proof Objective block. That is, 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`.

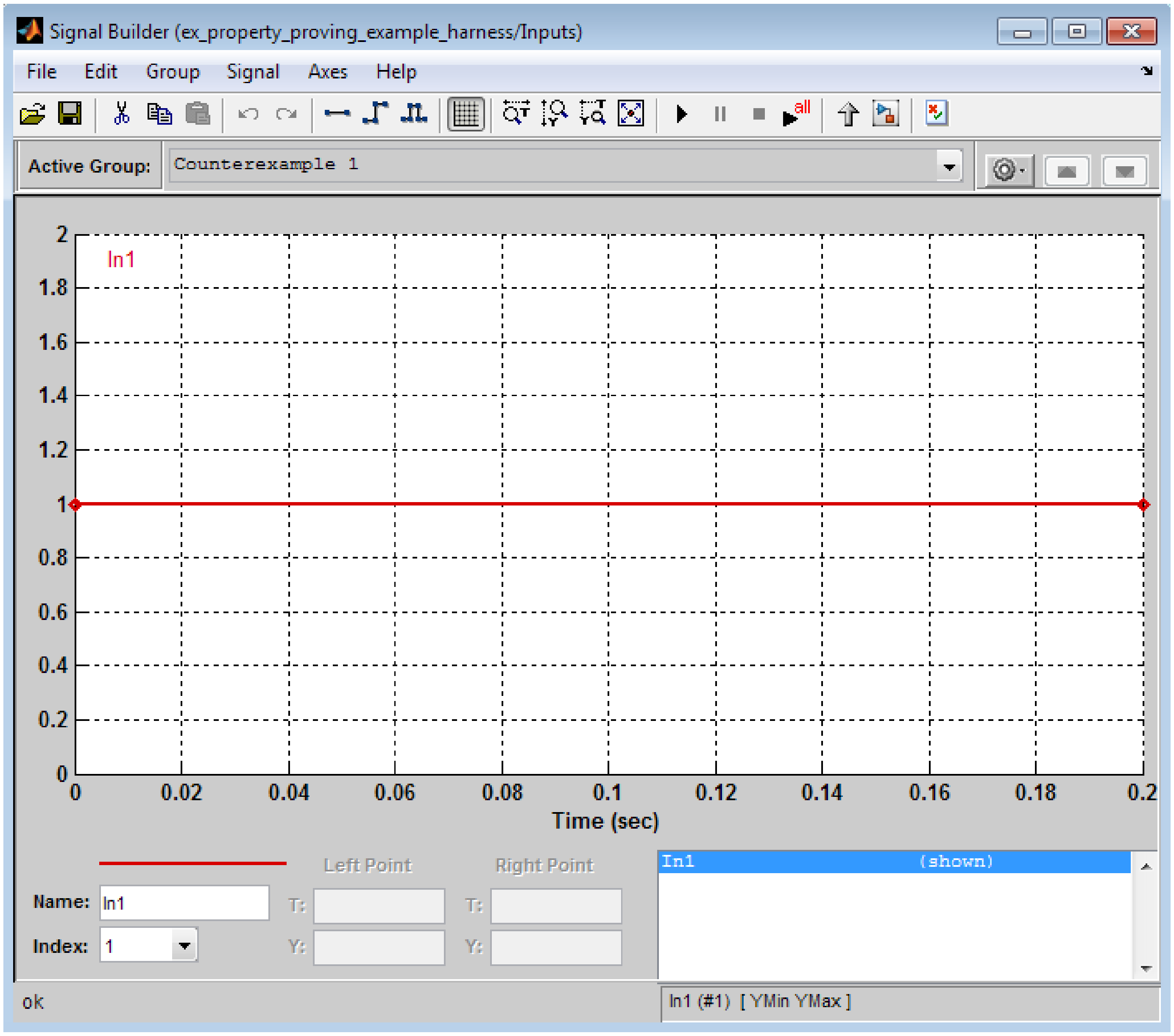


The harness model contains the following items:

- 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.



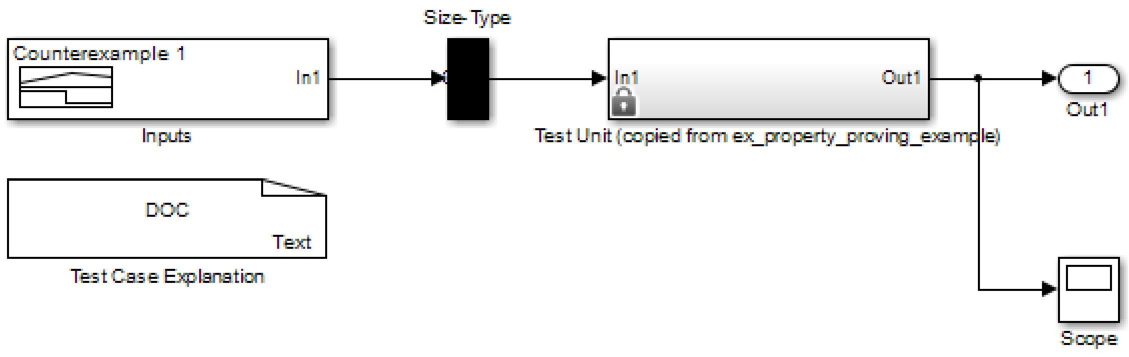
Name: In1
Index: 1

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:

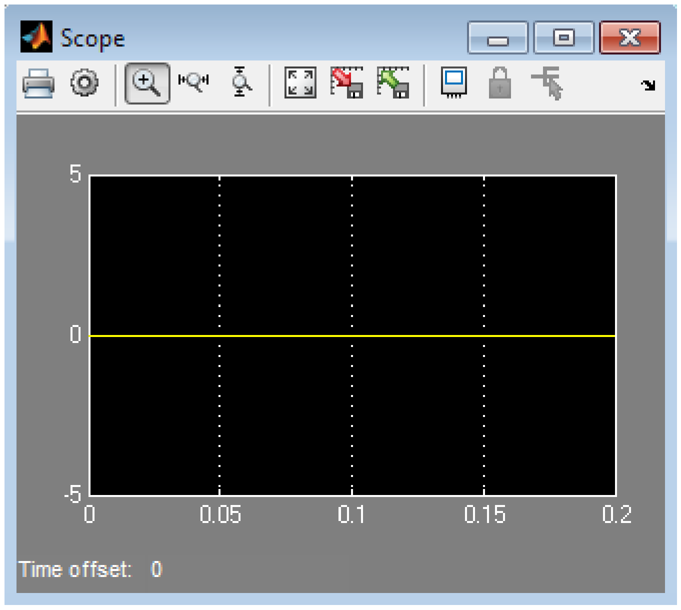
1 Open the `ex_property_proving_example_basic` model.
2 On the **Simulation** tab, click **Library Browser**.
3 From the Sinks library, copy a Scope block into your harness model window. The Scope block allows you to see the value of the signal output by the Compare To Zero block in your model.
4 In your harness model window, connect the output signal of the Test Unit subsystem to the Scope block.



5 To simulate your harness model, on the **Simulation** tab, click **Run**.

The Simulink software simulates the harness model.

6 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

As long as your model remains open, you can view the results of your most recent Simulink Design Verifier analysis results in the Results Summary window.

On the **Design Verifier** tab, in the **Review Results** section, click **Results Summary**. The Results Summary window opens displaying the results of the latest Simulink Design Verifier analysis.

For any Simulink Design Verifier analysis, from the 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 13-28
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 13-13
View the data file.	“Simulink Design Verifier Data Files” on page 13-7
View the log file.	“Simulink Design Verifier Log Files” on page 13-46

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

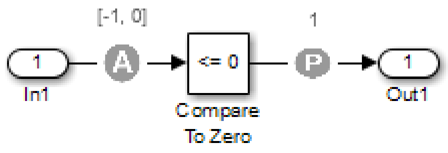
Customize Example Proof

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.
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.

Open the `ex_property_proving_example_basic` model. On the **Design Verifier** tab, click **Prove Properties**.

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:

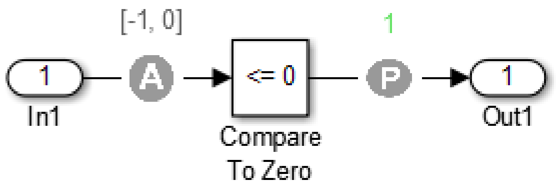
- "Review Results on the Model" on page 12-16
- "Review Analysis Report" on page 12-17

Review Results on the Model

Highlight the model to see the analysis results:

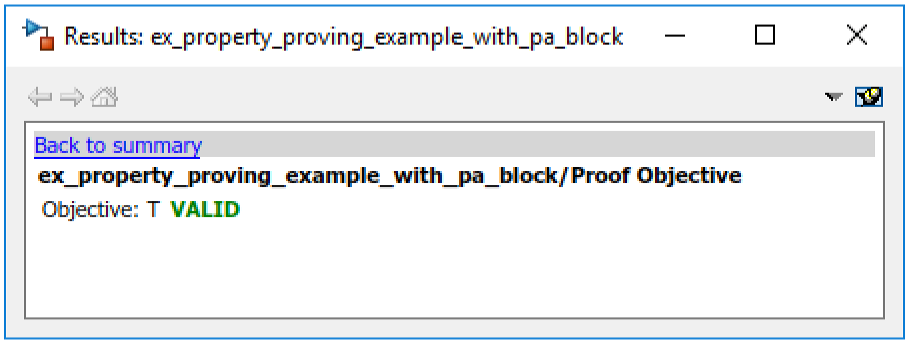
1 Click **Highlight analysis results on model**.

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.


Results: ex_property_proving_example_with_pa_block

Back to summary

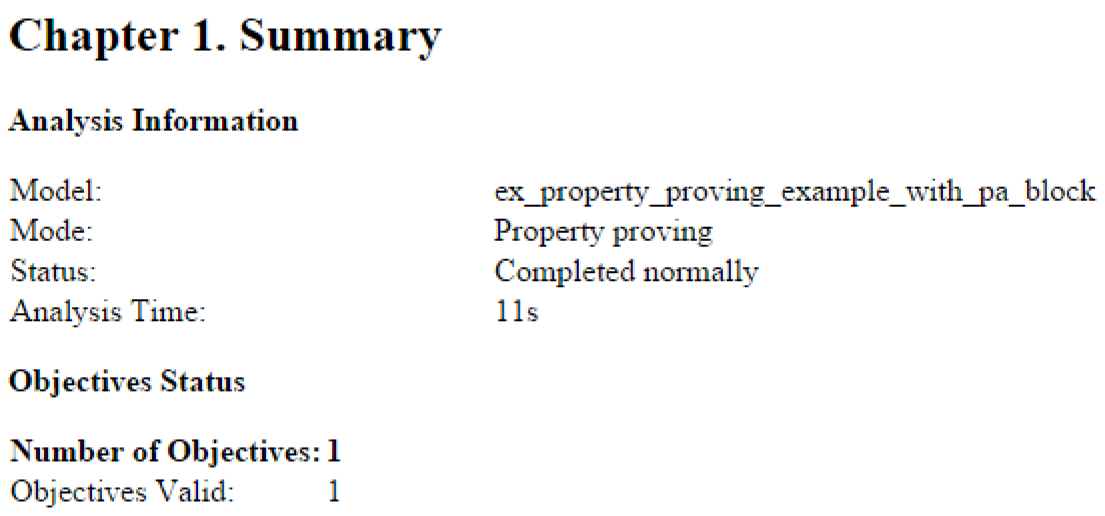
ex_property_proving_example_with_pa_block/Proof Objective

Objective: T **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

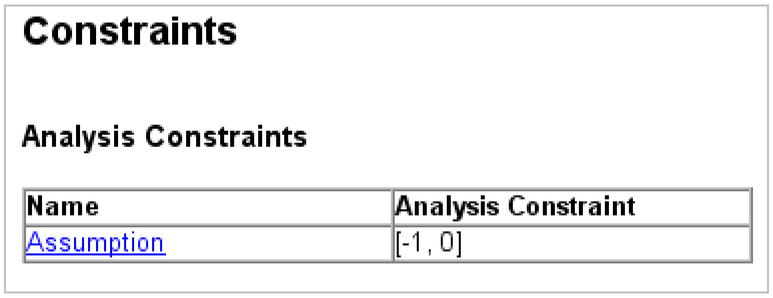
Analysis Information

Model:	ex_property_proving_example_with_pa_block
Mode:	Property proving
Status:	Completed normally
Analysis Time:	11s

Objectives Status

Number of Objectives:	**1**
Objectives Valid:	1

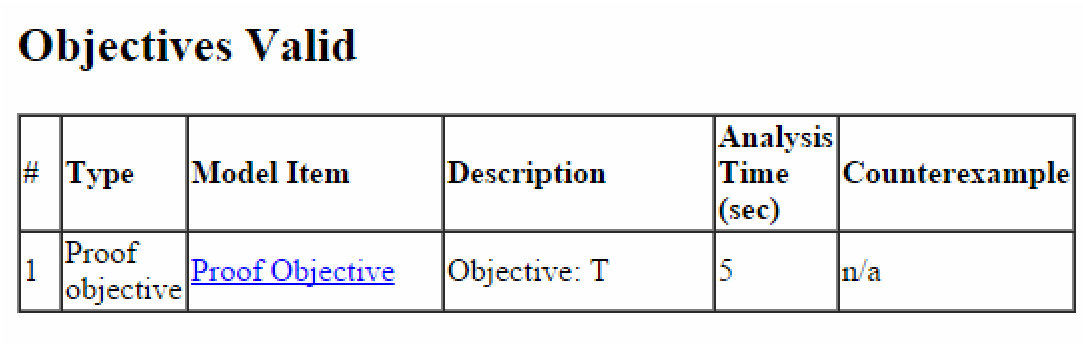
The Summary chapter indicates that Simulink Design Verifier proved a proof objective in the model.
3. The Constraints section lists the analysis constraint you specified in the Proof Assumption block.


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`.

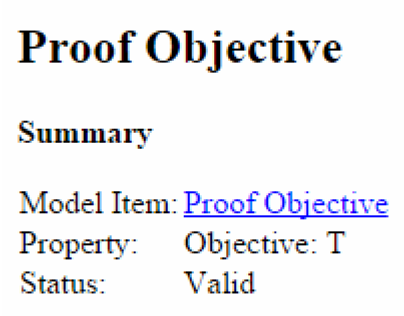


Objectives Valid

#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Proof Objective	Objective: T	5	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: T
Status:	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.

Note Constraints added at the inputs either through design minimum/maximum or test conditions/proof assumptions do not lead to a contradiction. However, if you constrain signals that are downstream of a computation using test conditions/proof assumptions, you must ensure that the constrained condition is feasible through the model computation. Otherwise, the resulting model is contradictory that may produces invalid results with or without an explicit analysis error. To ensure that the constraints are feasible, first try the same condition using a Test Objective. If it can be satisfied, you can use the same condition safely as a constraint.

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 14-22 gives detailed information about strategies you can use to improve the performance of a property-proving analysis of a large model.

See Also

More About

- “Property Proving with an Invalid Property” on page 12-29
- “Property Proving with Multiple Properties” on page 12-30
- “Property Proving with an Assumption Block” on page 12-31

Prove System-Level Properties Using Verification Model

In this section...

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 is falsified - needs simulation. For more information, see “Objectives Falsified - Needs Simulation” on page 13-39.

2 To see which objective was falsified, click **Highlight analysis results on model**.

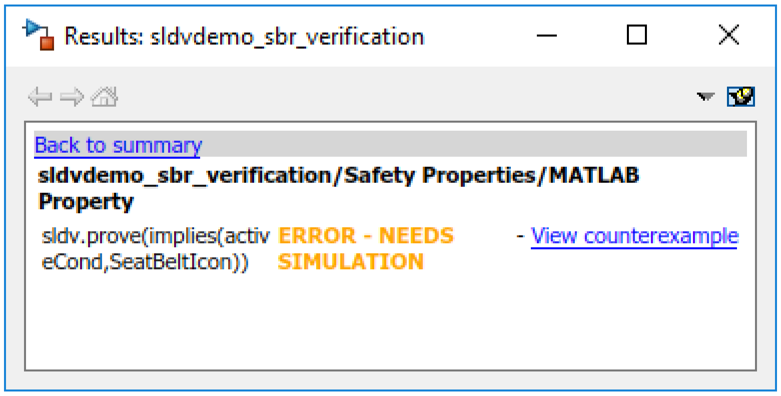
   The Safety Properties subsystem is highlighted in orange.

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.



Results: sldvdemo_sbr_verification

Back to summary

sldvdemo_sbr_verification/Safety Properties/MATLAB Property

sldv.prove(implies(activeCond,SeatBeltIcon)) ERROR - NEEDS SIMULATION - View counterexample

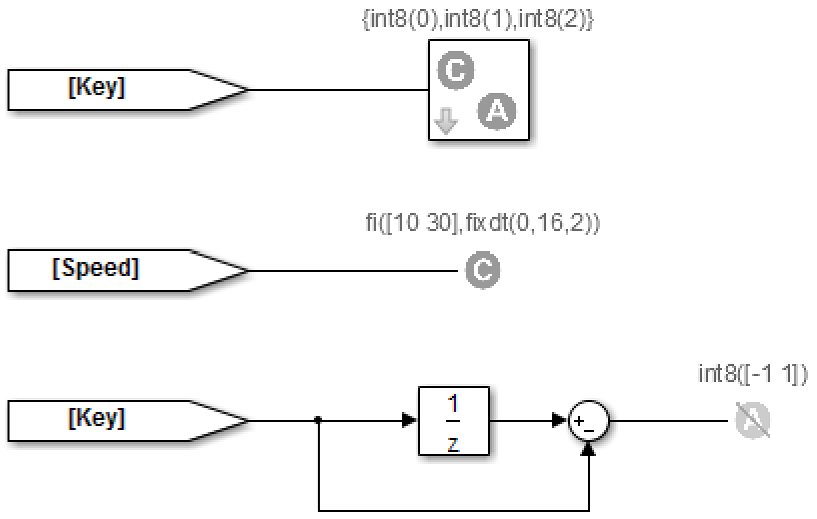
4 Click **View counterexample** to see the signal values that violated this property.

   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.

See Also

More About

- "Property Proving Using MATLAB Function Block" on page 12-37
- "Property Proving Using MATLAB Truth Table Block" on page 12-38

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 7-15 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 12-24 — Four examples that demonstrate how to prove basic properties.
- "Temporal Properties" on page 12-26 — 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:

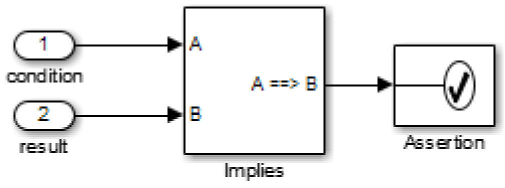
1. Open the Simulink Design Verifier block library. Type:

   ```
   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.

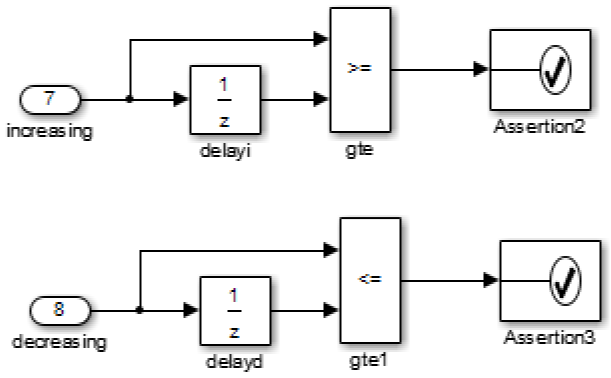


Implies operation describes conditions that should trigger a result.

Increasing or Decreasing Signals

The two examples in this section specify that a signal is either:

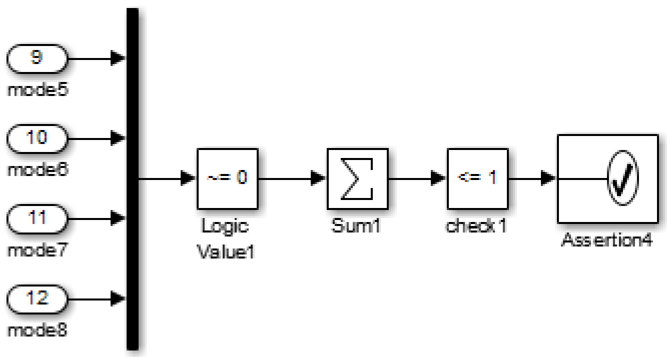
- Always increasing or staying constant
- Always decreasing or staying constant



Increasing and decreasing operations describe signals that should increase or decrease.

Exclusivity Operation

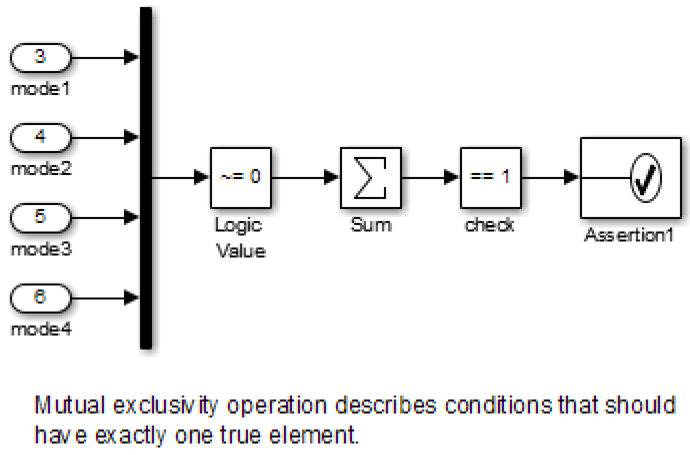
This example describes four conditions that should not be true at the same time.



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.


3
mode1
4
mode2
5
mode3
6
mode4
~= 0
Logic Value
Sum
== 1
check
Assertion1
Mutual exclusivity operation describes conditions that should have exactly one true element.

Temporal Properties

To view the Temporal Properties examples:

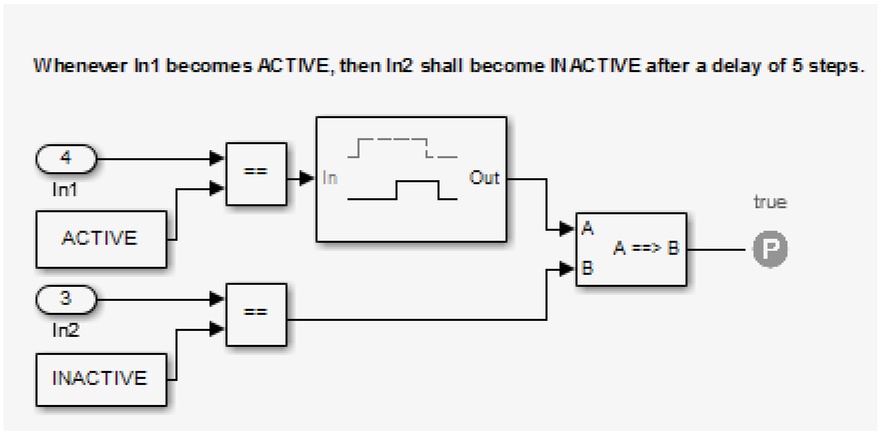
1. Open the Simulink Design Verifier block library. Type:

   ```
   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.

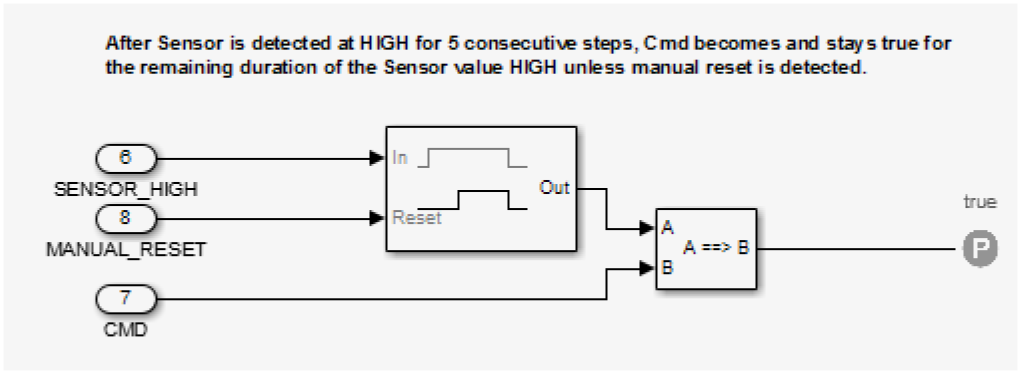
Synchronize the Output with the Input

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.
4
In1
ACTIVE
==
In
Out
3
In2
INACTIVE
==
A
A ==> B
B
true
P

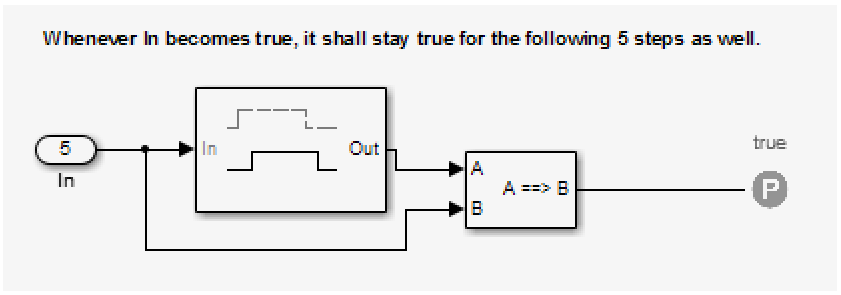
Make a Signal Inactive After a Delay

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.
6
SENSOR_HIGH
8
MANUAL_RESET
7
CMD
In
Reset
Out
A
B
A ==> B
true
P

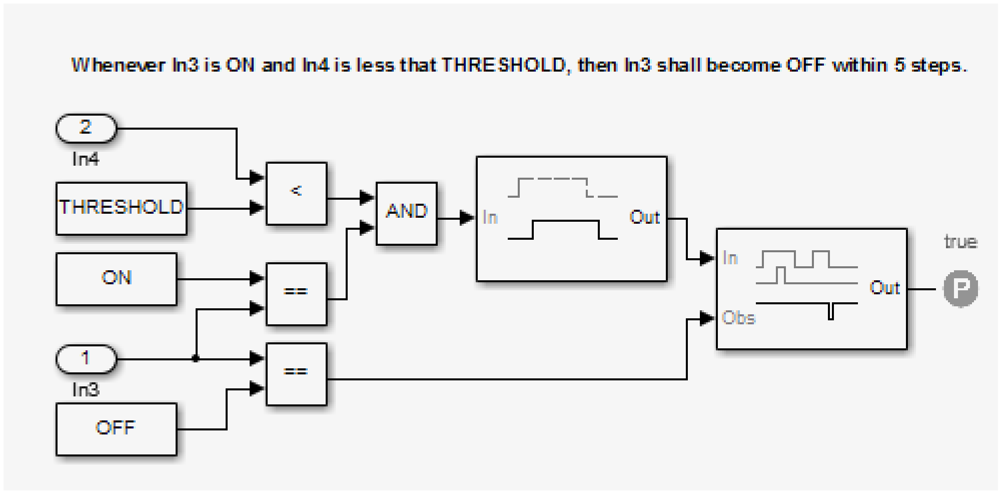
Extend a True Signal

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.
5
In
In
Out
A
B
A ==> B
true
P

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.
2
In4
THRESHOLD
ON
1
In3
OFF
<
==
==
AND
In
Out
In
Obs
Out
true
P

See Also

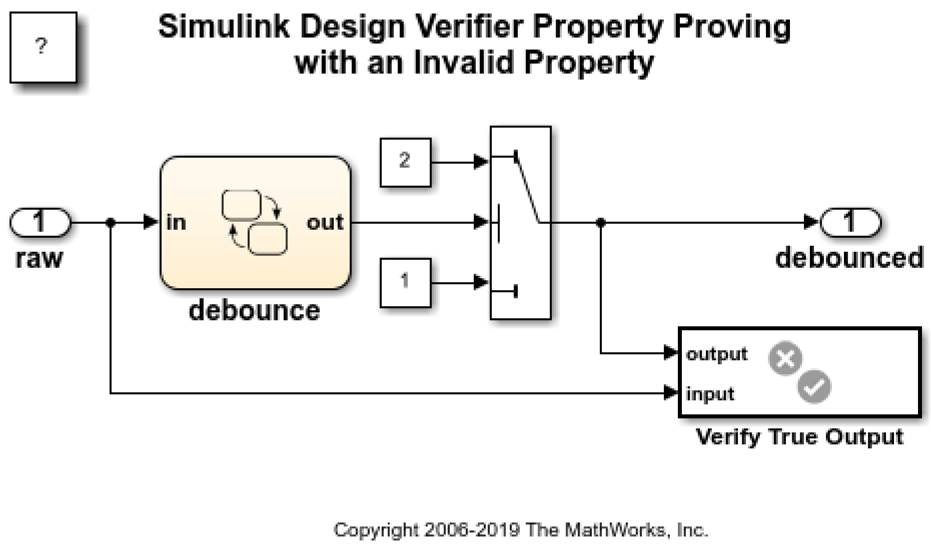
More About

- “Debounce Temporal Properties” on page 12-40
- “Power Window Controller Temporal Properties” on page 12-43

Property Proving with an Invalid Property

This example shows how to find an invalid property using Simulink Design Verifier property proving analysis. It attempts to prove that when the sum of the current and six previous input values is greater than 6, the output equals 2. In this case, the property is invalid because a single large input value (e.g. 255) causes the sum to be greater than 6. Simulink Design Verifier produces a counterexample that demonstrates the violation.

```
open_system('sldvdemo_debounce_falseprop');
```

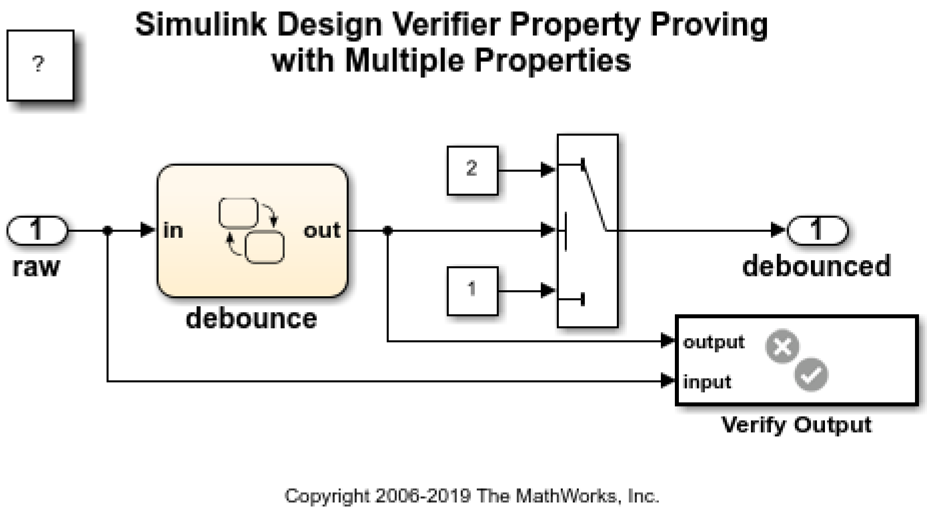


Property Proving with Multiple Properties

This example shows how to perform a property proving analysis with multiple properties. The model is configured for the analysis to attempt to prove that:

- When the current and six previous input values are true, the output will be true.
- When the current and six previous input values are false, the output will be false.

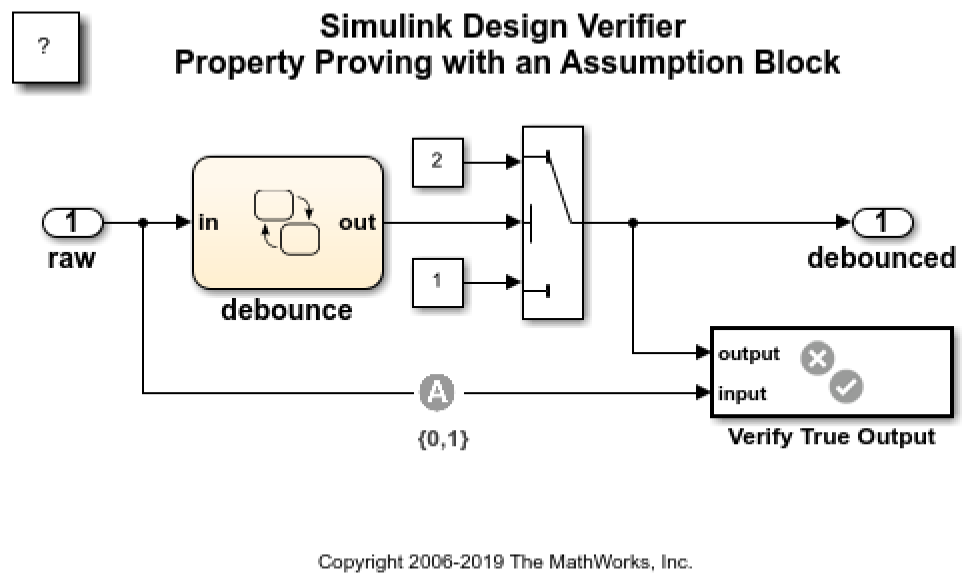
```
open_system('sldvdemo_debounce_validprop');
```


Simulink Design Verifier Property Proving
with Multiple Properties
?
1
raw
in
out
debounce
2
1
1
debounced
output
input
Verify Output
Copyright 2006-2019 The MathWorks, Inc.

Property Proving with an Assumption Block

This example shows how to perform a Simulink Design Verifier property proof using a Proof Assumption block. It attempts to prove that when the sum of the current and six previous input values is greater than 6, the output equals 2. The model includes a Proof Assumption block that constrains the input to be 0 or 1. Simulink Design Verifier searches for violations of 20 or fewer time steps. It is unable to find a violation because the property is valid under the assumption.

```
open_system('sldvdemo_debounce_assumeblk');
```



Property Proving Workflow for Cruise Control

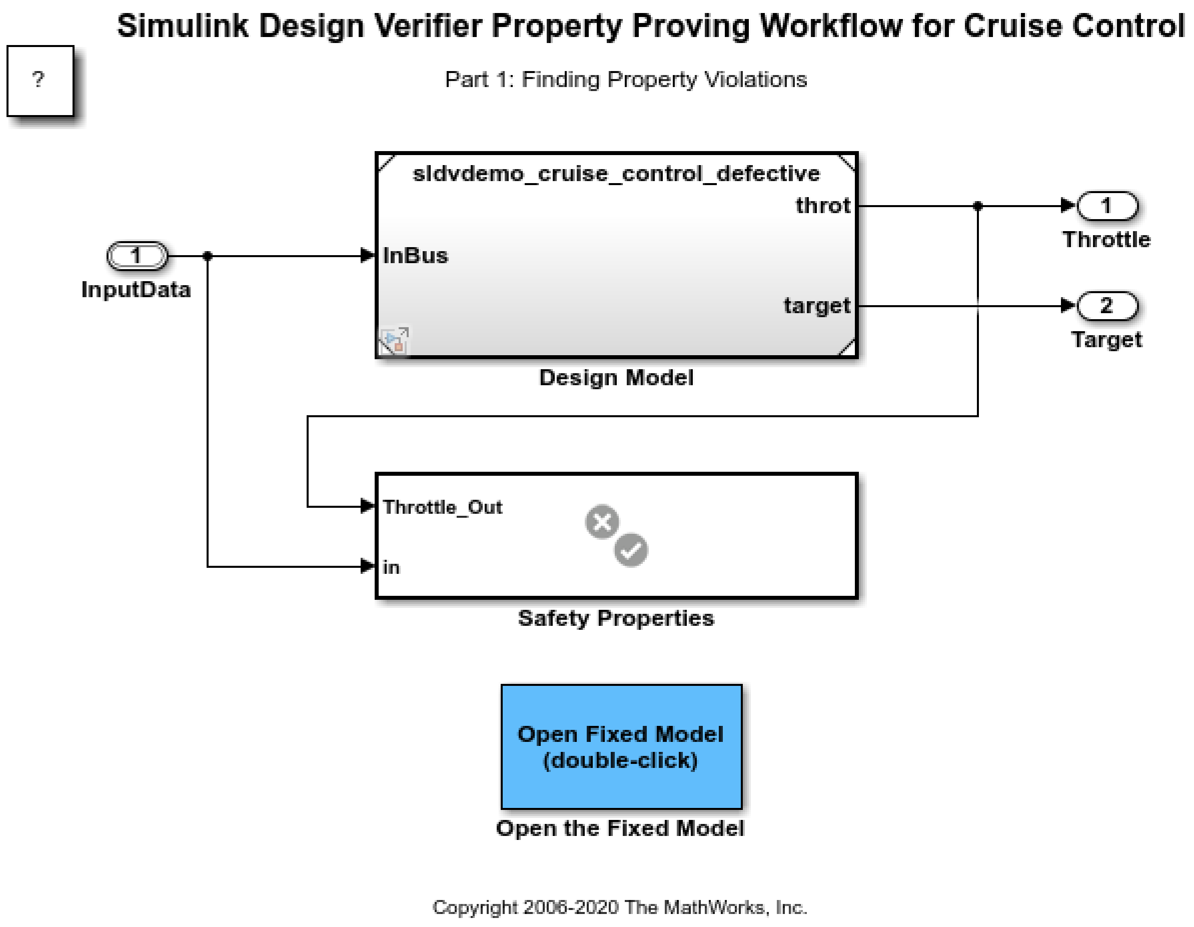
This example shows how to find a property violation by using Simulink® Design Verifier™ property proving analysis. You model safety requirements as properties and then verify the design model against requirements.

When you perform property proving analysis, Simulink Design Verifier generates a counterexample that you use to debug the property violation.

Step 1: Open the Model

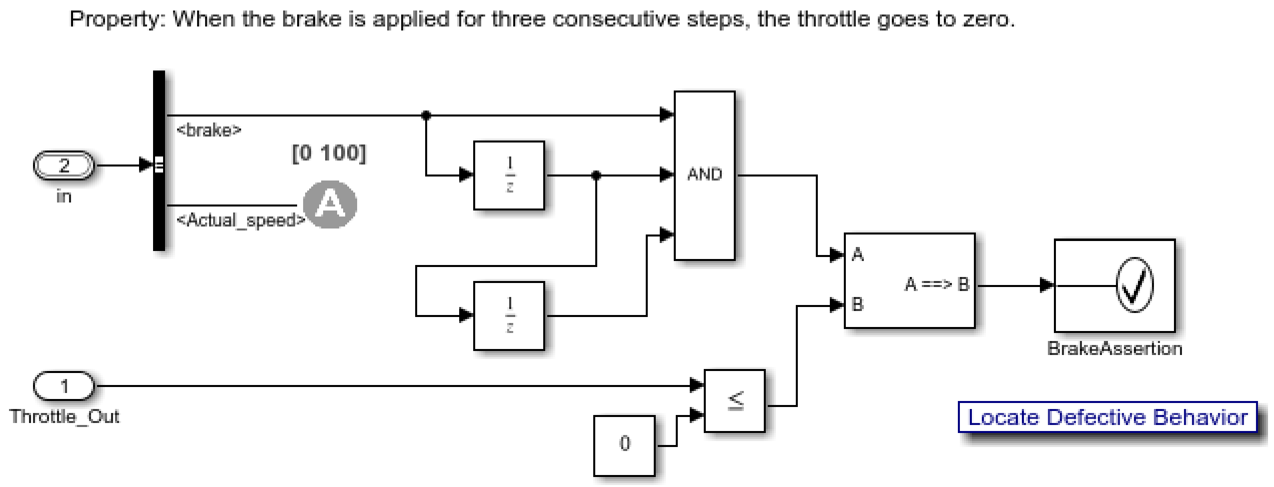
The `sldvdemo_cruise_control_verification` model contains a model reference to the `sldvdemo_cruise_control_defective` design model. The design model is a cruise control system that consists of a PI Controller that computes the throttle output based on the difference between the actual and target speed.

```
open_system('sldvdemo_cruise_control_verification');
```


Simulink Design Verifier Property Proving Workflow for Cruise Control
Part 1: Finding Property Violations
?
sldvdemo_cruise_control_defective
throt
InBus
target
Design Model
1
InputData
1
Throttle
2
Target
Throttle_Out
in
Safety Properties
Open Fixed Model (double-click)
Open the Fixed Model
Copyright 2006-2020 The MathWorks, Inc.

The safety properties for the throttle output are modeled in the `Safety Properties` verification subsystem by the Assertion block.

```
open_system('sldvdemo_cruise_control_verification/Safety Properties');
```

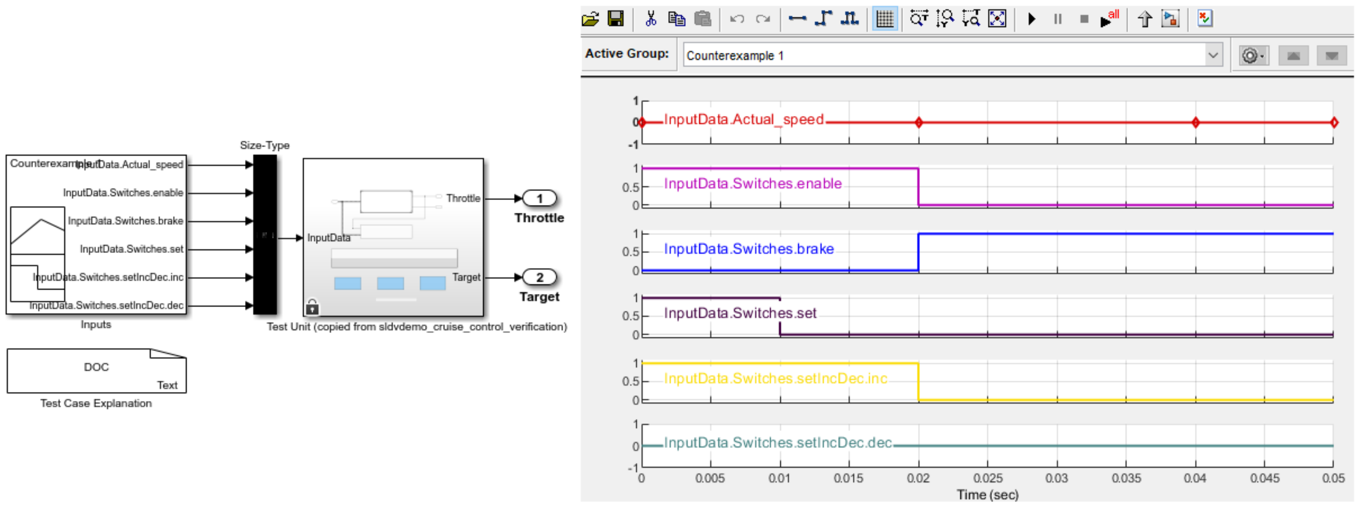

Property: When the brake is applied for three consecutive steps, the throttle goes to zero.
<brake>
[0 100]
<Actual_speed>
A
2
in
1
Throttle_Out
1/z
1/z
AND
A
B
A ==> B
BrakeAssertion
≤
0
Locate Defective Behavior

Step 2: Perform Property Proving Analysis

On the **Design Verifier** tab, click **Prove Properties**.

After the analysis completes, the Results Summary window reports that one objective was falsified.

The harness model is generated and the Signal Builder dialog box opens and displays the counterexample.



Step 3: Simulate the Counterexample to Replicate the Error

In the Signal Builder dialog box, click the **Start simulation** button ▸ .

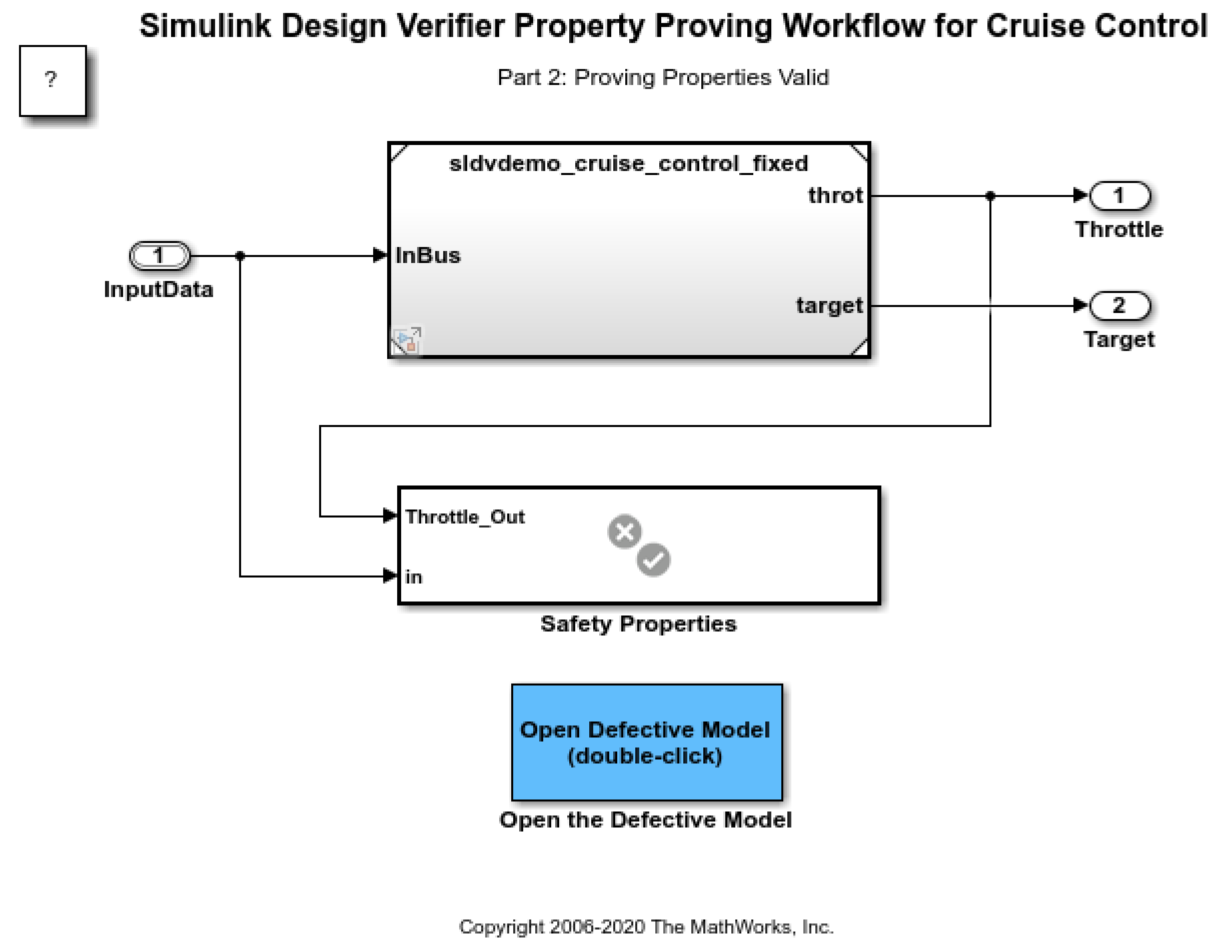
The Diagnostic Viewer window displays an error stating that the simulation was terminated because an assertion occured at time `0.04`.

Optionally, you can debug the property violation by using the Model Slicer. For more information, see "Debug Property Proving Violations by Using Model Slicer" on page 12-52.

Step 4: Open the Fixed Model

The errorneous behavior exhibited by the counter example is fixed in the `sldvdemo_cruise_control_verification_fixed` model.

```
open_system('sldvdemo_cruise_control_verification_fixed');
```


Simulink Design Verifier Property Proving Workflow for Cruise Control
Part 2: Proving Properties Valid
?
sldvdemo_cruise_control_fixed
throt
InBus
target
1
InputData
1
Throttle
2
Target
Throttle_Out
in
Safety Properties
Open Defective Model (double-click)
Open the Defective Model
Copyright 2006-2020 The MathWorks, Inc.

In property proving workflow, you may require to redesign the system and/or redefine the property and perform such iterations.

Open the referenced model `sldvdemo_cruise_control_fixed` and open the `Controller` subsystem. In this subsystem, the updated design model resets the throttle output when Active Control is active.

On the **Design Verifier** tab, click **Prove Properties**. After the analysis completes, the Results Summary window reports that the objective is valid.

See Also

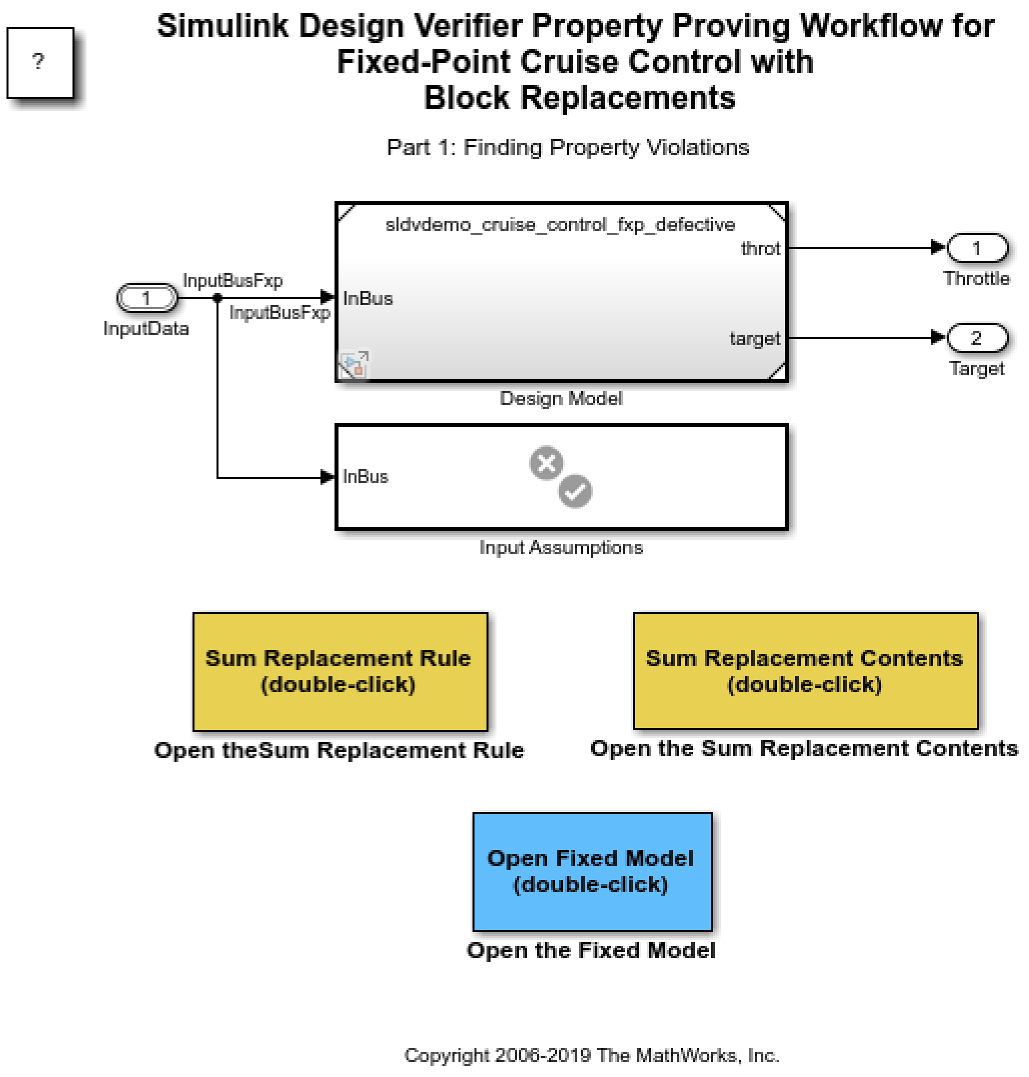
- "Workflow for Proving Model Properties" on page 12-4

- “Prove System-Level Properties Using Verification Model” on page 12-20

Property Proving Workflow for Fixed-Point Cruise Control with Block Replacements

This example shows how to prove properties in a fixed-point cruise control algorithm. It references the design model using model reference so that the original design model is unchanged. A block replacement rule specifies the property that checks if an overflow is possible. The verification subsystem specifies an assumption on the range of the speed input during property proving. This model configures Simulink Design Verifier to apply a block replacement to the Sum block that feeds the outport of the fixed-point PI Controller in the referenced model and return a counterexample that demonstrates an overflow.

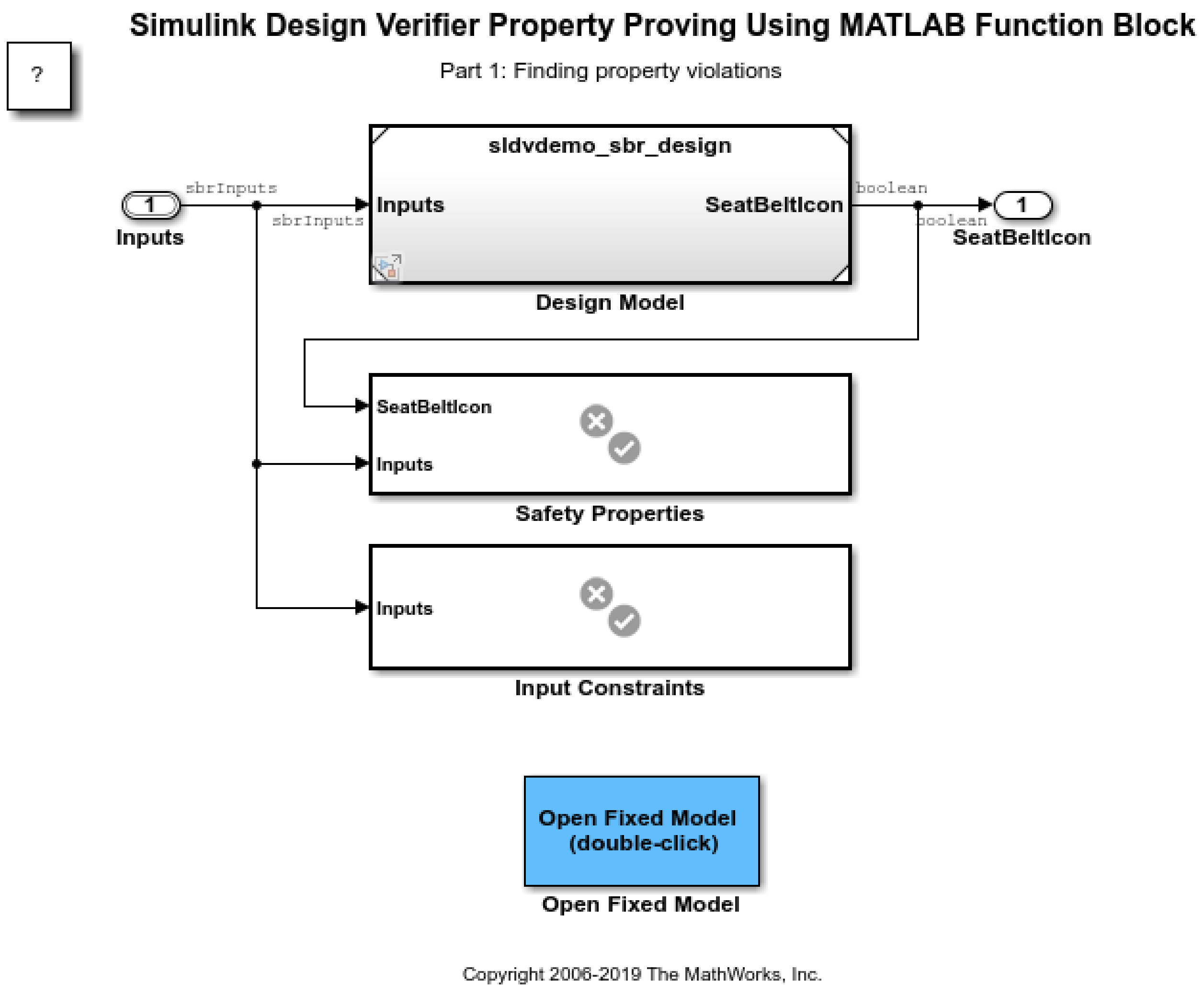
```
open_system('sldvdemo_cruise_control_fxp_verification');
```


?
Simulink Design Verifier Property Proving Workflow for Fixed-Point Cruise Control with Block Replacements
Part 1: Finding Property Violations
sldvdemo_cruise_control_fxp_defective
throt
target
InBus
InputBusFxp
InputBusFxp
1
InputData
1
Throttle
2
Target
Design Model
InBus
Input Assumptions
Sum Replacement Rule (double-click)
Open theSum Replacement Rule
Sum Replacement Contents (double-click)
Open the Sum Replacement Contents
Open Fixed Model (double-click)
Open the Fixed Model
Copyright 2006-2019 The MathWorks, Inc.

Property Proving Using MATLAB Function Block

This example shows how to verify the seat belt reminder design model. The Safety Properties block below it contains a property specified in MATLAB that indicates when the icon should be active. Simulink Design Verifier analyzes the design model and safety property to prove correctness or to identify counterexamples. In this model, the property is violated because the design implicitly assumes that the KEY input starts at 0 and changes by increments of 1.

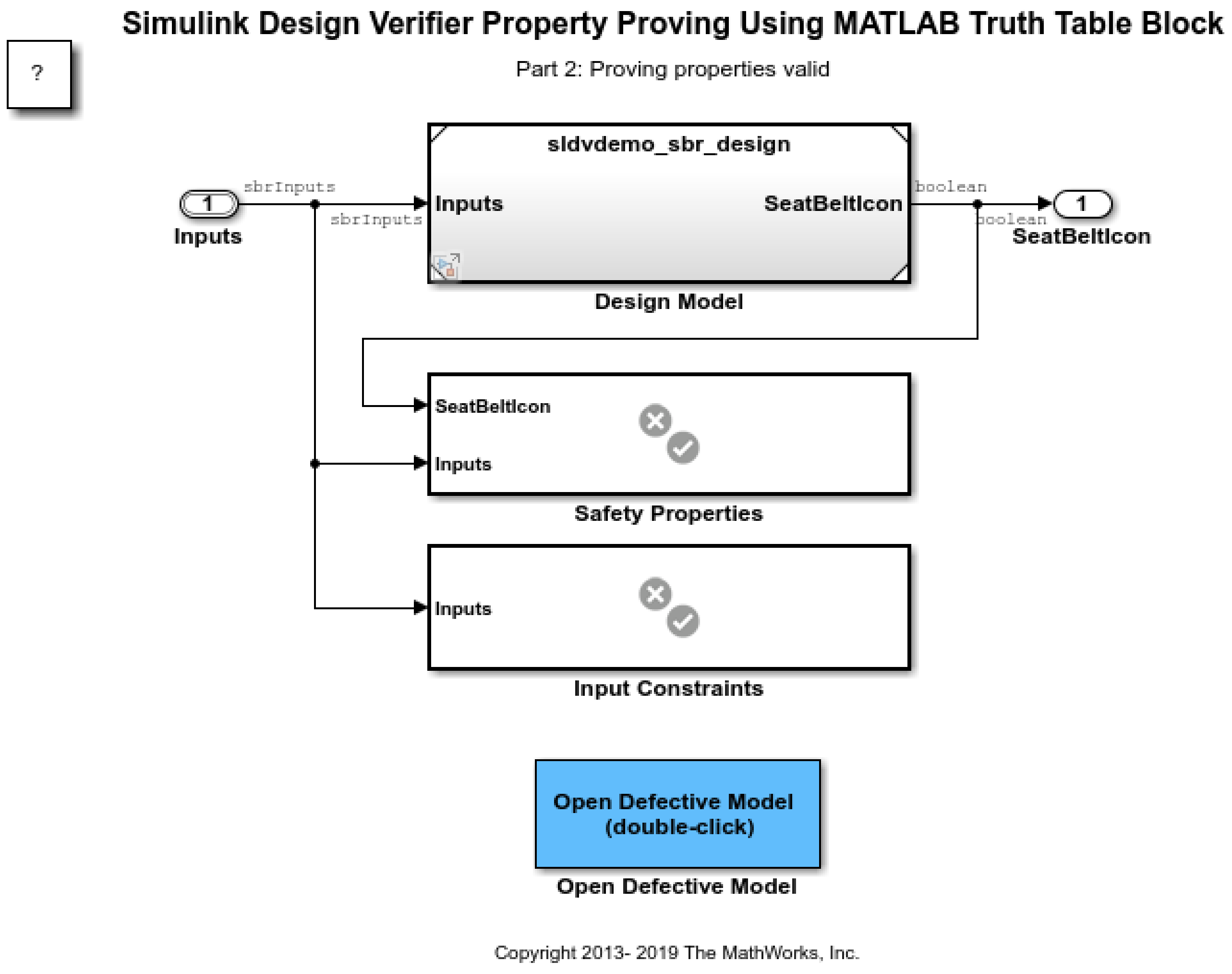
```
open_system('sldvdemo_sbr_verification');
```



Property Proving Using MATLAB Truth Table Block

This example shows how to verify the seat belt reminder design model referenced in the top block above. The Safety Properties block below it contains a property specified in MATLAB Truth Table that indicates when the SeatBeltIcon output should be active. Simulink Design Verifier analyzes the design model and safety property to prove correctness or to identify counterexamples. In this model, the property is proven under the explicit assumption that the KEY input starts at 0 and changes by increments of 1.

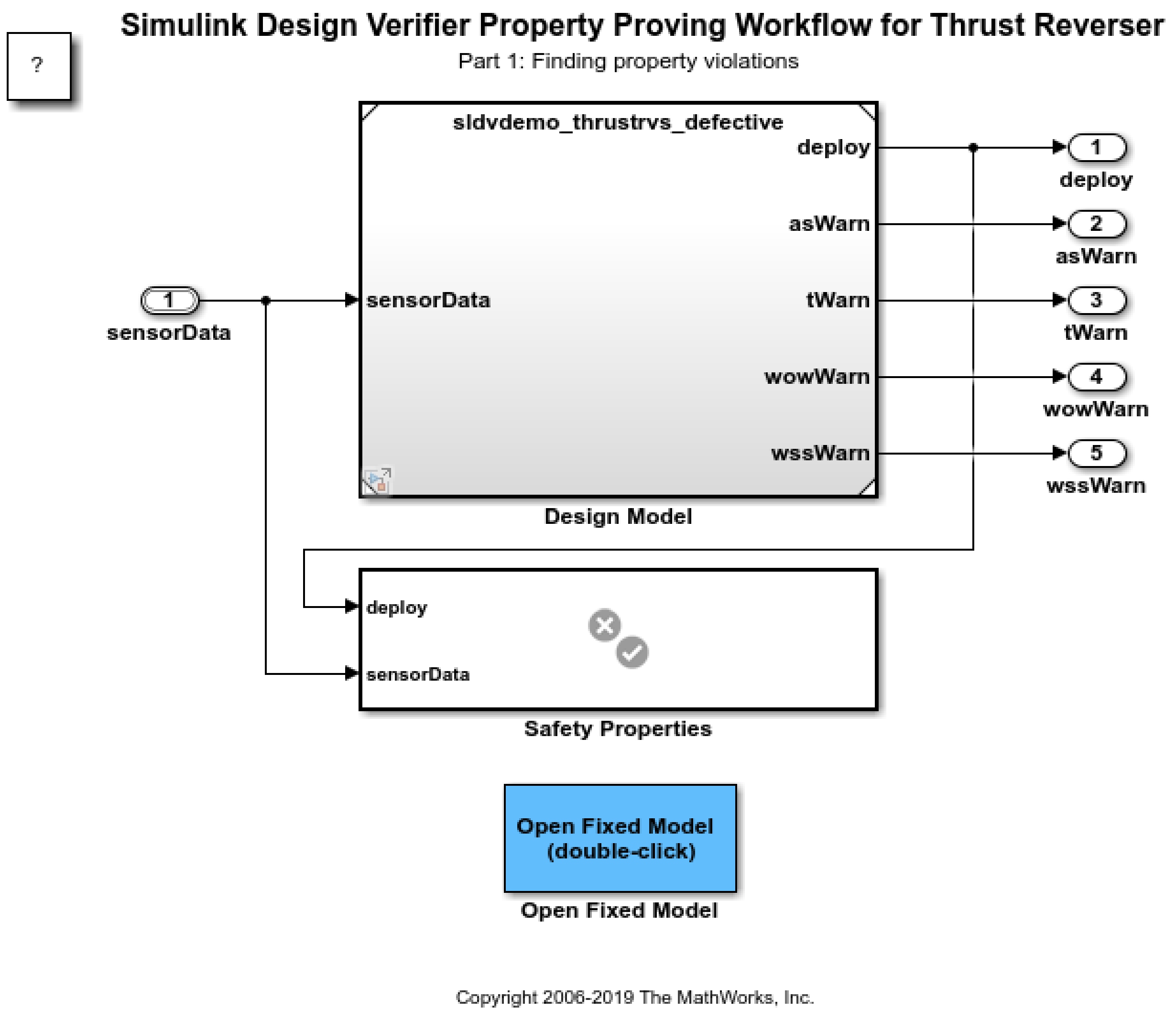
```
open_system('sldvexSBRVerificationTruthTableFixedExample');
```



Property Proving Workflow for Thrust Reverser

This example shows how to verify safety properties in a thrust reverser design model. The Properties block below it contains four safety properties. Simulink Design Verifier analyzes the design model and safety properties to prove correctness or to identify counterexamples. The use of model referencing eliminates the need to add verification content to the design model, allowing the verification content to exist independently from the design.

```
open_system('sldvdemo_thrustrvs_verification');
```


Simulink Design Verifier Property Proving Workflow for Thrust Reverser
Part 1: Finding property violations
?
sldvdemo_thrustrvs_defective
sensorData
deploy
asWarn
tWarn
wowWarn
wssWarn
Design Model
1 sensorData
1 deploy
2 asWarn
3 tWarn
4 wowWarn
5 wssWarn
deploy
sensorData
Safety Properties
Open Fixed Model (double-click)
Open Fixed Model
Copyright 2006-2019 The MathWorks, Inc.

Debounce Temporal Properties

This example shows how to model temporal system requirements for property proving and test case generation using Simulink® Design Verifier™ Temporal Operator blocks.

Temporal Operators

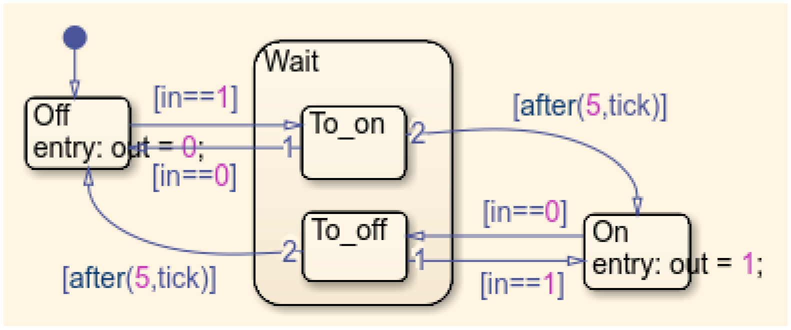
The Simulink® Design Verifier™ library provides three basic temporal operator blocks can be used to model temporal properties. The intent of the temporal operators is to support the specification of temporal requirements, such that the modeled property has a closer co-relation to the actual textual requirement. These blocks are low-level building blocks for constructing more complex temporal properties.

Debounce Model and Requirements

Consider a debounce logic that debounces between values of 0 and 1 based on the input holding a value for a fixed number of time steps.

The debounce functionality is captured in the containing Stateflow® chart.

```
open_system('sldvdemo_debounce_to')
open_system('sldvdemo_debounce_to/debounce')
```



Consider two requirements of the debounce model that you would like to verify.

Requirement 1:

Whenever the input equals 1 for more than 6 steps, the output shall be equal to 2.

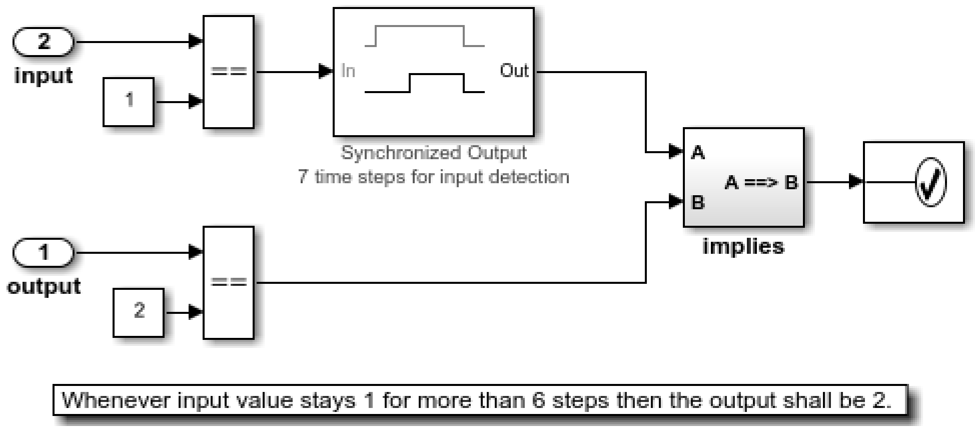
Requirement 2:

Whenever the input becomes 0 for more than 5 steps after the output was 2, the output shall equal 1 as long as the input stays at 0.

Property Specification

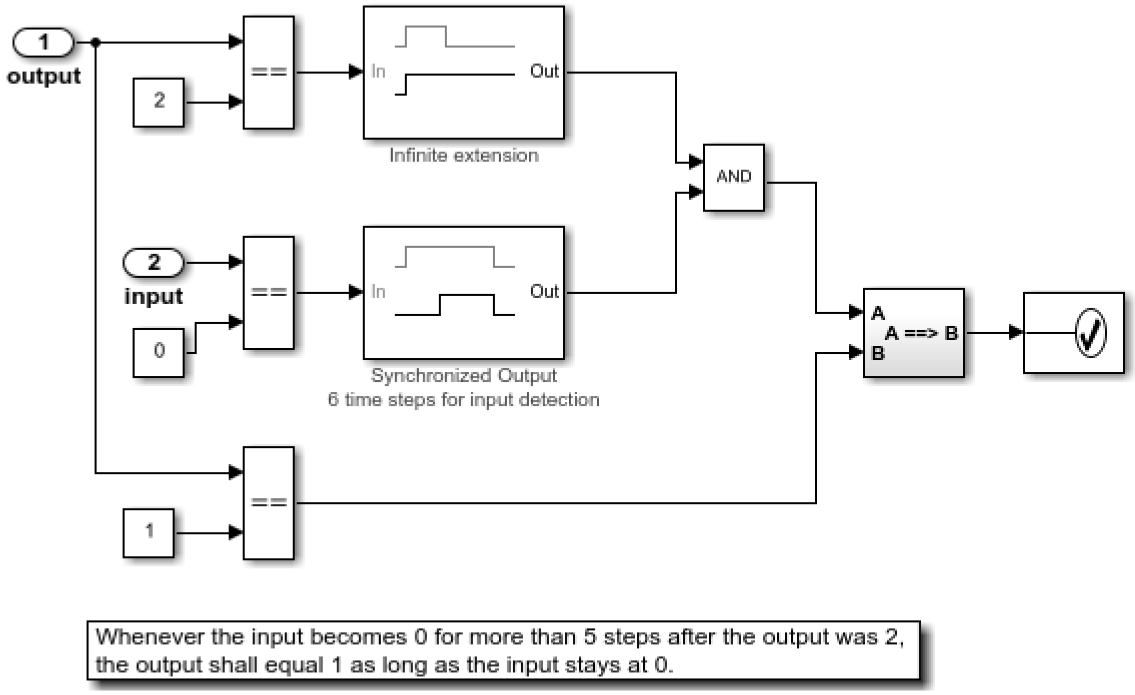
For specifying **Requirement 1**, you first represent the constraint that **input equals 1 for more than 6 steps**. This can be captured by the Detector block from the Temporal Operator Blocks Library. On detecting that the input value is 1 for 7 (or more than 6) time steps, you want to check that the output equals 2 as long as input stays equal to 1 after the detection. Use the "Synchronized" option of the Detector block followed by an Implies block to capture this.

```
open_system('sldvdemo_debounce_to/Verify True Output1')
```


2
input
1
==
In
Out
Synchronized Output
7 time steps for input detection
1
output
2
==
A
B
A ==> B
implies
Whenever input value stays 1 for more than 6 steps then the output shall be 2.

Multiple temporal operator blocks can be combined to construct more complex temporal properties. Consider **Requirement 2**.

```
open_system('sldvdemo_debounce_to/Verify True Output2')
```


1
output
2
==
In
Out
Infinite extension
AND
2
input
0
==
In
Out
Synchronized Output
6 time steps for input detection
1
==
A
B
A ==> B
Whenever the input becomes 0 for more than 5 steps after the output was 2,
the output shall equal 1 as long as the input stays at 0.

For illustration, this requirement is broken down roughly into three pieces of interest:

1. **After the output was 2**: This is an enabling condition for your property. While checking the rest of the constraints, you want to know if this condition was true at some point in the past. This type of an enabling condition is typically followed by an Extender (either "Finite" or "Infinite") that, in combination with other constraints, might form the first input to your implication.
2. **The input becomes 0 for more than 5 steps** and check something **as long as input stays 0**: For the same reason as the first property, you use a Detector with "Synchronized" output ("Time steps for input detection" = 6).

3 **The output shall equal 1**: This is the condition that you want to verify whenever the first two constraints hold. This is captured through a logical Implies block. Note that you cannot use Within Implies block here.

Property Proving

Once the temporal requirements have been modeled, you can perform property proving on these using Simulink Design Verifier.

Clean Up

To complete the example, close all the opened models.

```
close_system('sldvdemo_TOBlocks',0);
close_system('sldvdemo_debounce_to',0);
```

Power Window Controller Temporal Properties

This example shows how to model temporal system requirements in a power window controller model for property proving and test case generation using Simulink® Design Verifier™ Temporal Operator blocks.

Temporal Operators

The Simulink® Design Verifier™ library provides three basic temporal operator blocks which can be used to model temporal properties. The intent of the temporal operators is to support the specification of temporal requirements, such that the modeled property has a closer correlation to the actual textual requirement. These blocks are low-level building blocks for constructing more complex temporal properties.

Power Window Controller

The power window controller responds to the driver and passenger commands by giving the commands for moving the window up or down. It also responds to an obstacle and to reaching the end of the window frame in either direction.

Consider the following two requirements for the power window controller:

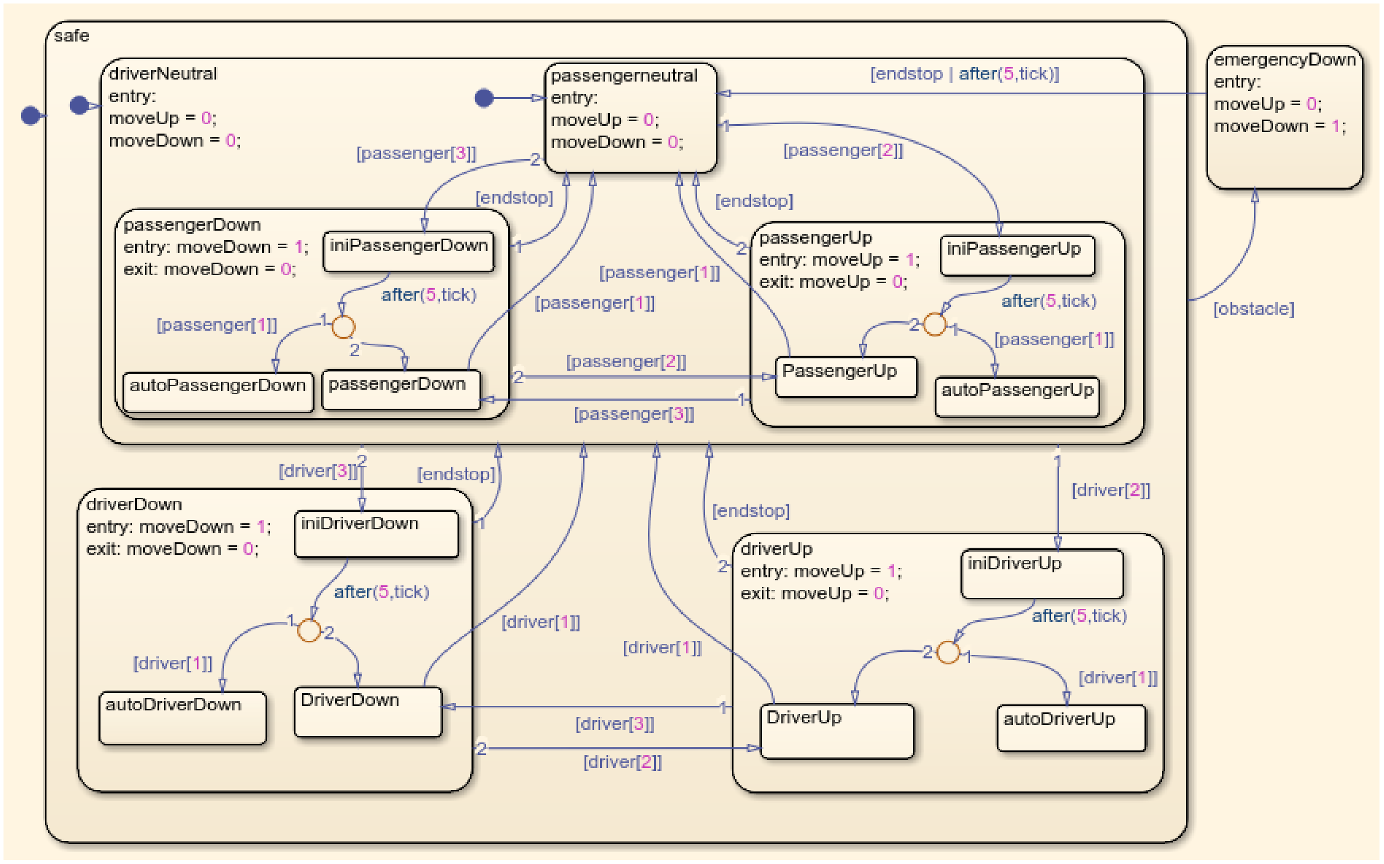
Requirement 1 (Obstacle Response)

Whenever an obstacle is detected, the controller shall give the down command for 1 second.

Requirement 2 (AutoDown feature)

If the driver presses the down button for less than 1 second, the controller keeps giving the down command until the end has been reached or the driver presses the up button.

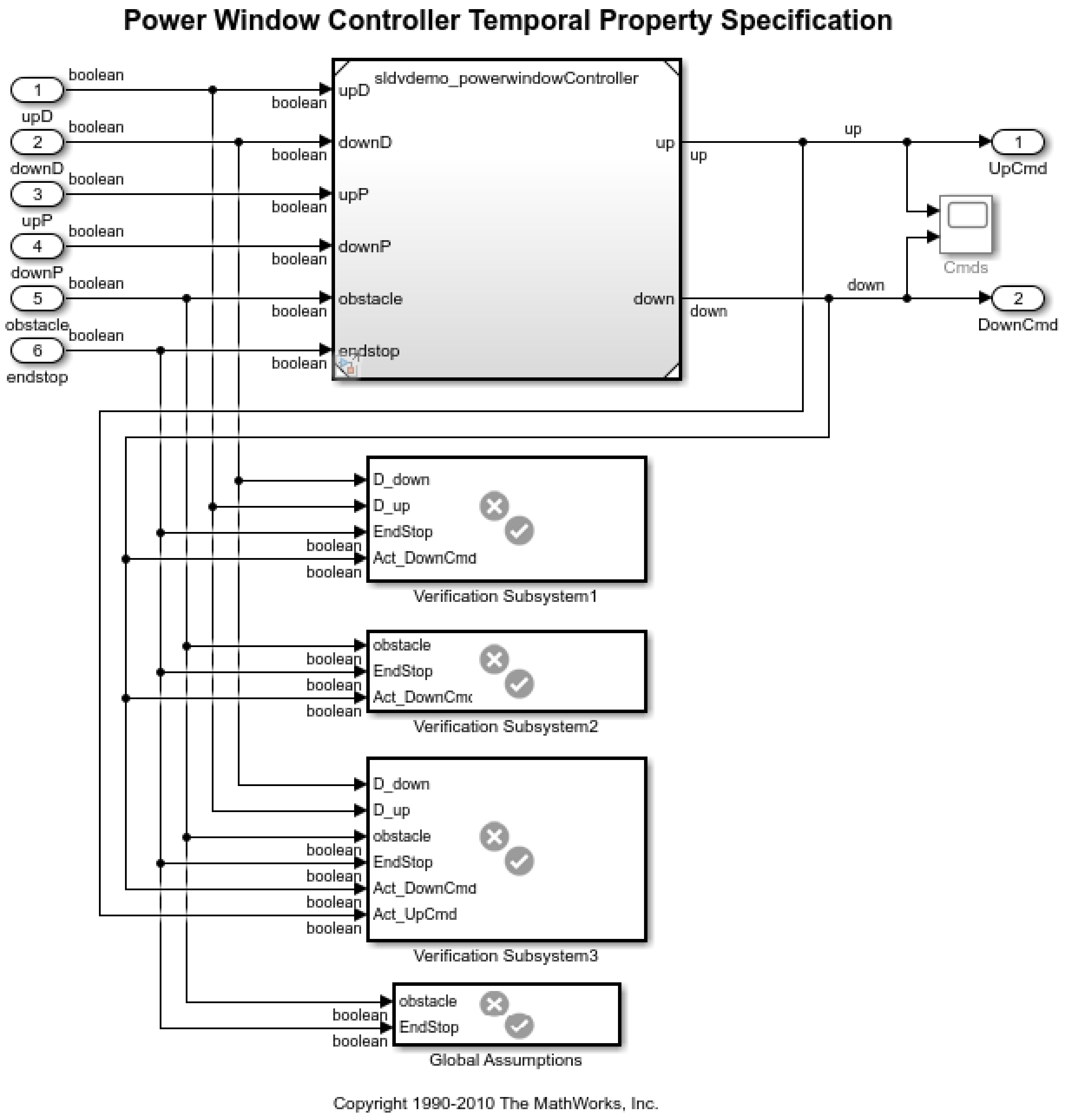
```
%Model of the power window controller
open_system('sldvdemo_powerwindowController')
open_system('sldvdemo_powerwindowController/control')
```



Property Specification

The power window verification system is the top-level model that contains a model reference to the power window controller model specifying the controller behavior and the modeled requirements.

```
%Model of the top-level verification system
open_system('sldvdemo_powerwindow_vs')
```



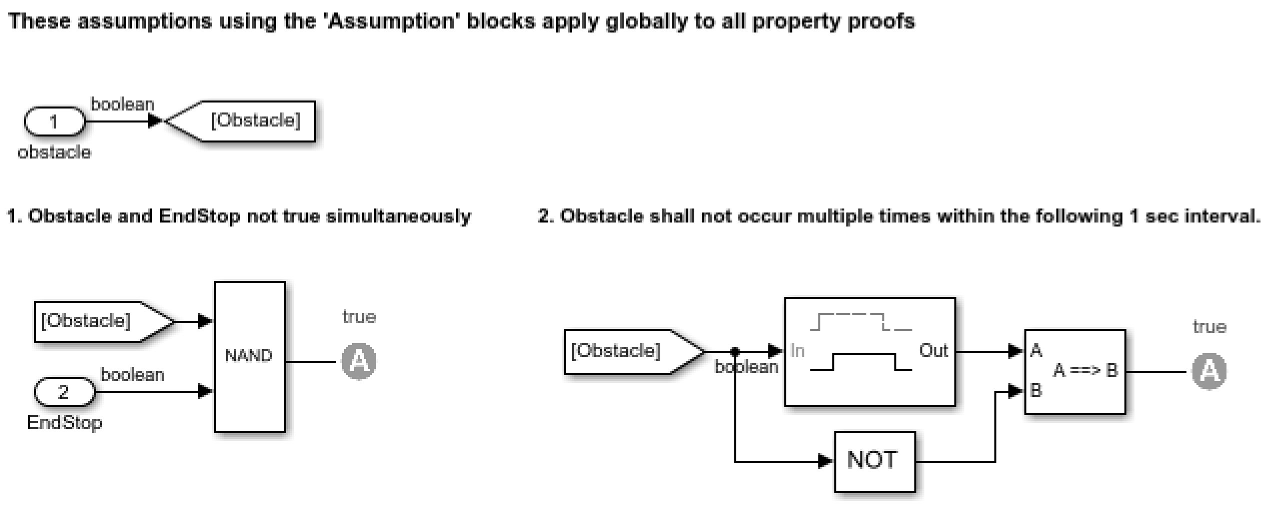
Global Assumptions: The power window controller is an open system. This makes the environment controlled inputs, obstacle and endstop (end of the window frame) to occur freely. To constrain the environment, add two global assumptions for your controller model.

1) The obstacle and the endstop inputs never become true at the same time.

2) The obstacle does not occur multiple times within the following 1-second interval.

For the temporal assumption on obstacle, use a Detector block with output type of "Delayed Fixed Duration" to capture the fixed duration of 1 second (5 time steps with 0.2 sample time).

```
% Global Assumptions
open_system('sldvdemo_powerwindow_vs/Global Assumptions')
```



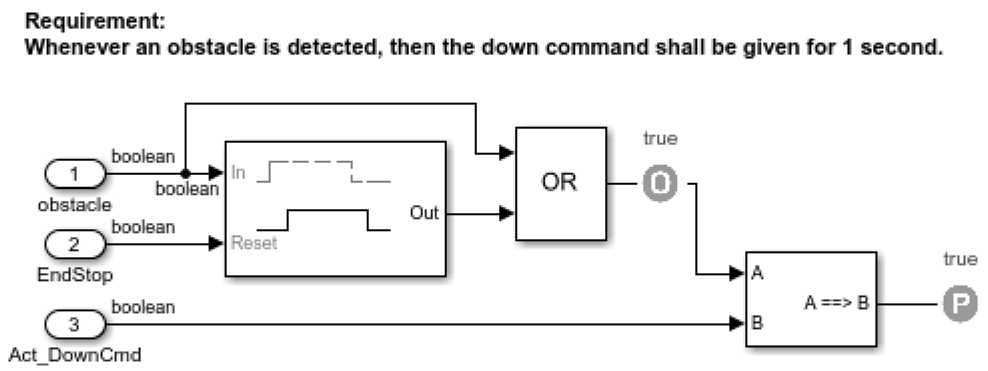
These assumptions using the 'Assumption' blocks apply globally to all property proofs

1. Obstacle and EndStop not true simultaneously

2. Obstacle shall not occur multiple times within the following 1 sec interval.

Now consider the first controller requirement for **Obstacle Response**.

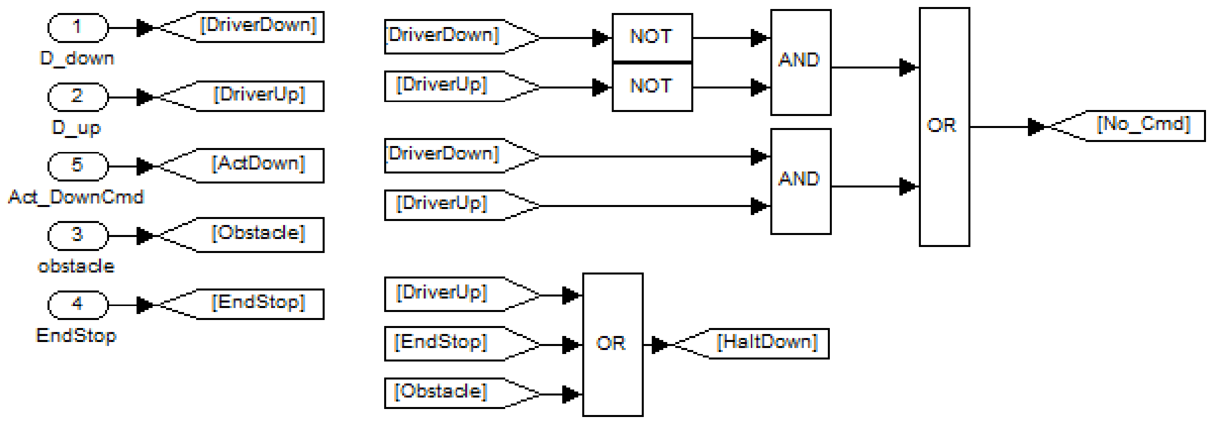
```
% Obstacle Response
open_system('sldvdemo_powerwindow_vs/Verification Subsystem2')
```



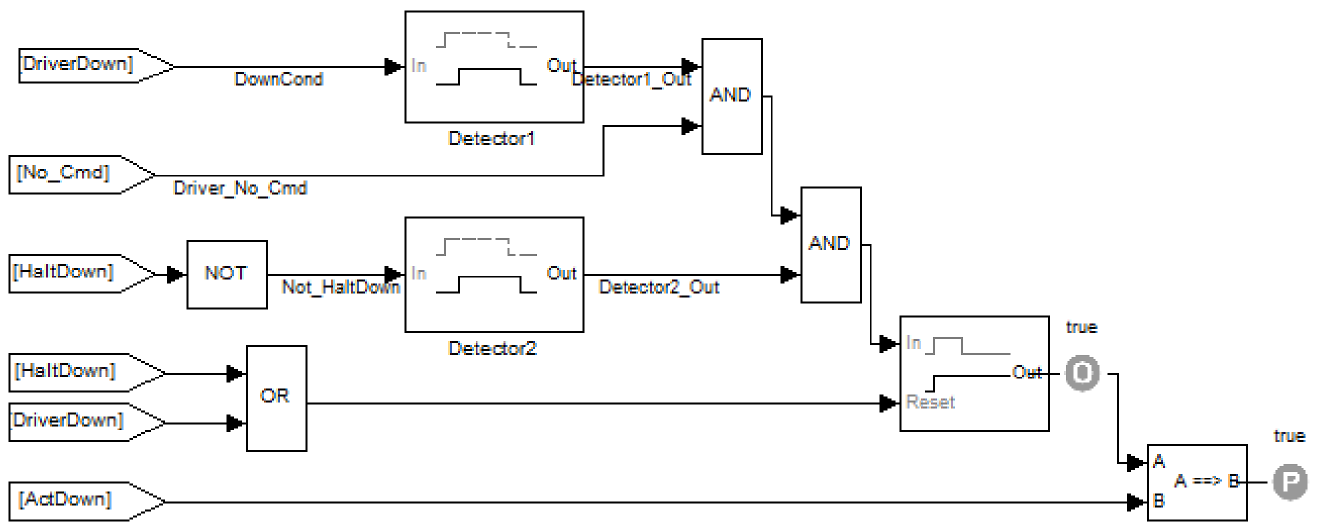
Requirement:
Whenever an obstacle is detected, then the down command shall be given for 1 second.

Here, use the Detector block with output type of "Delayed Fixed Duration" for the property specification. After detection of the obstacle, construct a fixed interval of 4 steps. Note that the input is not observed during the output construction phase for the Detector with "Delayed Fixed Duration" output type. In the case where the obstacle can occur freely in absence of the assumption, you might wish to observe all the intermediate occurrences of the obstacle. This can be achieved through an Extender block with "Finite" extension duration of 4 time steps.

Now consider the **AutoDown feature** of the power window controller.



Requirement (Autodown)
If the driver presses the down button for less than 5 steps, then the controller gives the down command as long as end has not been reached or the driver presses the up button.



For illustration, consider this property specification in smaller parts:

1. The first temporal duration of interest, "driver presses the down button for less than 1 second", is captured by Detector1. At sample rate of 0.2, the 1-second interval is broken down into 5 time steps. On detection of the down signal, Detector1 constructs a 5-step fixed temporal duration at its output, which you will subsequently use in combination with other constraints.
2. For the AutoDown feature, you know that the down signal cannot be pressed for more than 1 second, or 5 time steps. Thus, you want to ensure that both driver up and down are "true" or both are "false" in less than 5 steps after down is pressed. By taking the AND of this driver neutral and the Detector output, enforce the constraint that driver down can be pressed for any number of consecutive time steps less than 5.
3. You also need to ensure that, during this period, other signals such as obstacle, EndStop and DriverUp are not true, since these will take the controller out of responding to the down press. This is captured using Detector2 by enforcing that NOT(HaltDown) is true for 5 time steps.

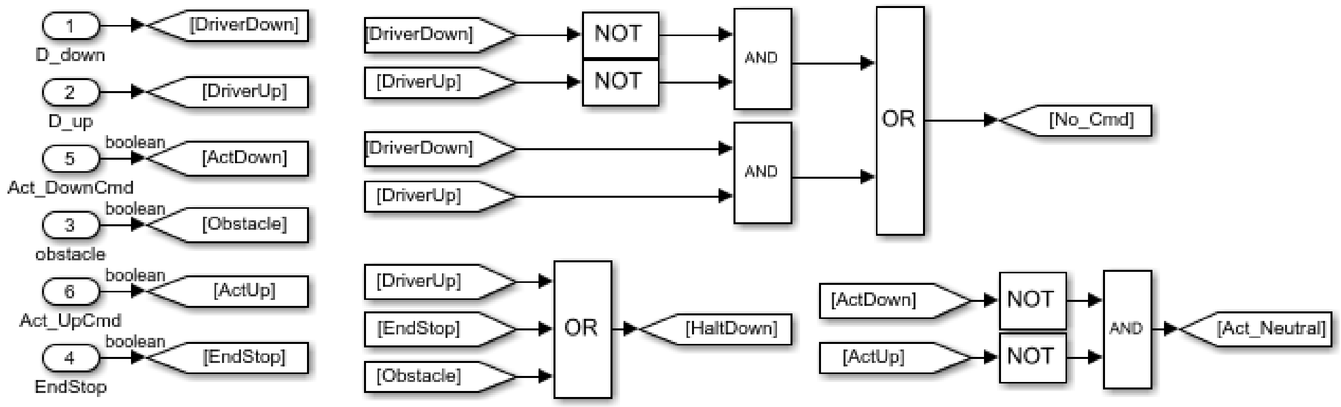
Detector2 has "Delayed Fixed Duration" output type. It also has "Time steps for input detection" = 5 and "Time steps for output duration" = 1.

4. Take the AND of these constructed durations.
5. For the AutoDown feature, you do not want to limit the number of time steps for which the controller gives the down command. You know that you want the controller to keep giving the down command as long as the driver does not press an up or down command again, or an obstacle or the physical end of the window frame is not hit. This behavior can be captured by the Extender block with "Infinite" extension period and an external reset signal that encodes the condition to end the extension.
6. The final piece is an Implies block that takes the temporal duration constructed as explained above and checks if the controller down command is true for every time step of this duration.

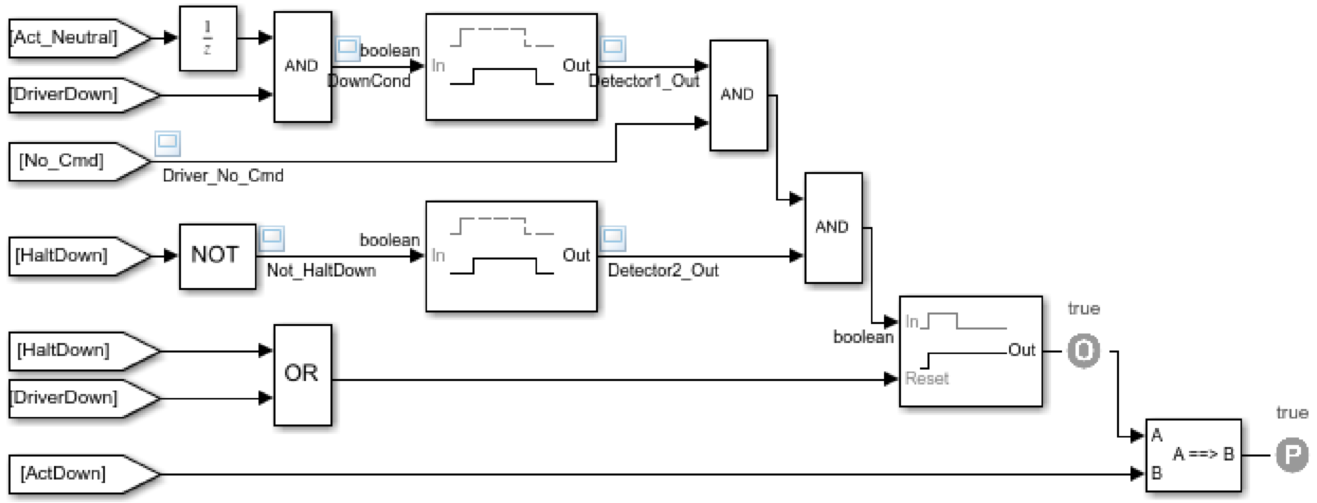
Once you have this initial property specification, you can use it for property proving with Simulink Design Verifier. You will get a counterexample for this property. The counterexample shows a scenario where the down command is given when the controller was in the emergency down state due to the response to an earlier detected obstacle. After you add a constraint to avoid this, you will get another counterexample: if the down button is pressed when previously the up command was being given, the AutoDown feature is disabled and the down command is given only as long as the down button is pressed. Looking at these counterexamples and observing the model, you can see a pattern that the AutoDown feature is enabled only when the controller is in a neutral state to begin with when the driver presses the down button.

Incorporate this constraint by forcing the controller output to be neutral - neither up nor down command is true - as a precondition for the AutoDown property. This property is proven valid.

```
% Valid AutoDown
open_system('sldvdemo_powerwindow_vs/Verification Subsystem3')
```



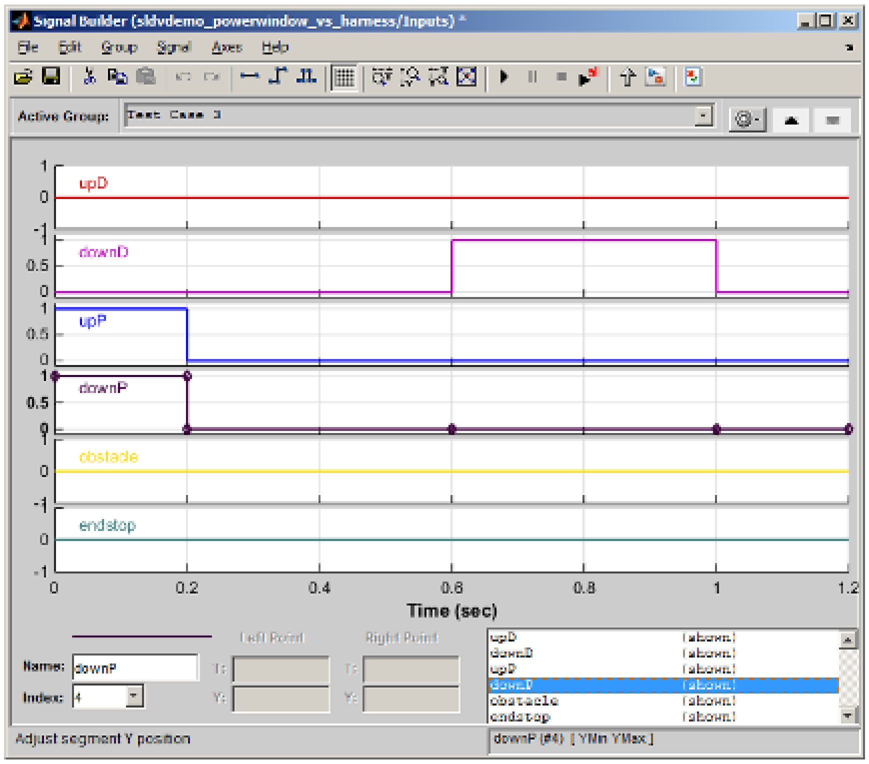
Requirement (Autodown)
If the driver presses the down button for less than 5 steps, then the controller gives the down command as long as end has not been reached or the driver presses the up button.



Test Case Generation for Property Validation

Once the properties are specified, in addition to property proving, you can run Simulink Design Verifier to automatically generate test cases that exercise various conditions in the property. This can be achieved by placing custom Test Objective blocks at appropriate locations in the property.

One such location to place a Test Objective block (with "true" value) is on the signal feeding into the first input of the Implies block (as shown in the above property). On running test generation, this Test Objective is satisfied and you will get a test case exercising the various constraints encoded in the property. Simulink Design Verifier can also create a test harness to simulate this test case. The Signal Builder block with relevant signals is shown below.


Signal Builder (sldvdemo_powerwindow_vs_harness/Inputs) *
File Edit Group Signal Axes Help
Active Group: Test Case 3
upD
downD
upP
downP
obstacle
endstop
Time (sec)
Left Point
Right Point
T:
Y:

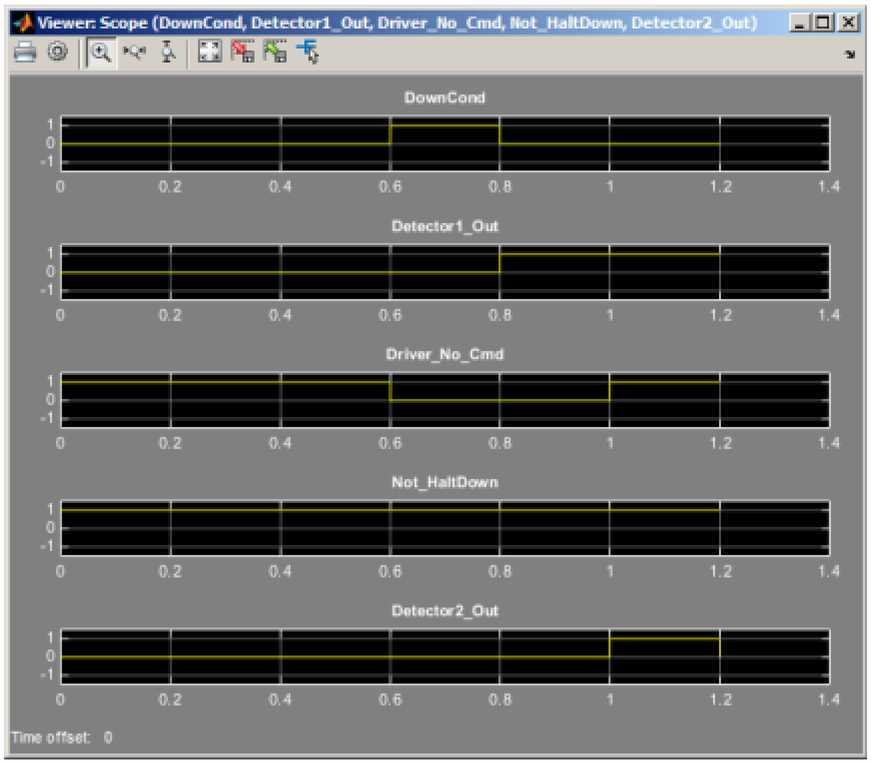
Name: downP
Index: 4

upD	[shown]
downD	[shown]
upP	[shown]
downP	[shown]
obstacle	[shown]
endstop	[shown]

Adjust segment Y position

downP (#4) [YMin YMax]

One can now simulate this test case, and see how the temporal durations are created in the property by placing a scope that displays the input and output values of the two Detector blocks and No_Cmd.


Viewer: Scope (DownCond, Detector1_Out, Driver_No_Cmd, Not_HaltDown, Detector2_Out)
DownCond
Detector1_Out
Driver_No_Cmd
Not_HaltDown
Detector2_Out

Time offset: 0

Manually inspecting the test case values enables you to see if the specified property behaves as intended.

This Test Objective block helps in identifying a scenario where the property is valid while the Implies block is not trivially true. An Implies block is trivially true when its output is true because of its first input being false. When you get a test case satisfying this Test Objective, you know that there is at least one case where the first input to the Implies block is true.

This exercise can help you validate your property specifications by manually inspecting the test cases automatically generated by Simulink Design Verifier.

Clean Up

To complete the example, close all the opened models.

```
close_system('sldvdemo_TOBlocks',0);
close_system('sldvdemo_powerwindowController',0);
close_system('sldvdemo_powerwindow_vs',0);
```

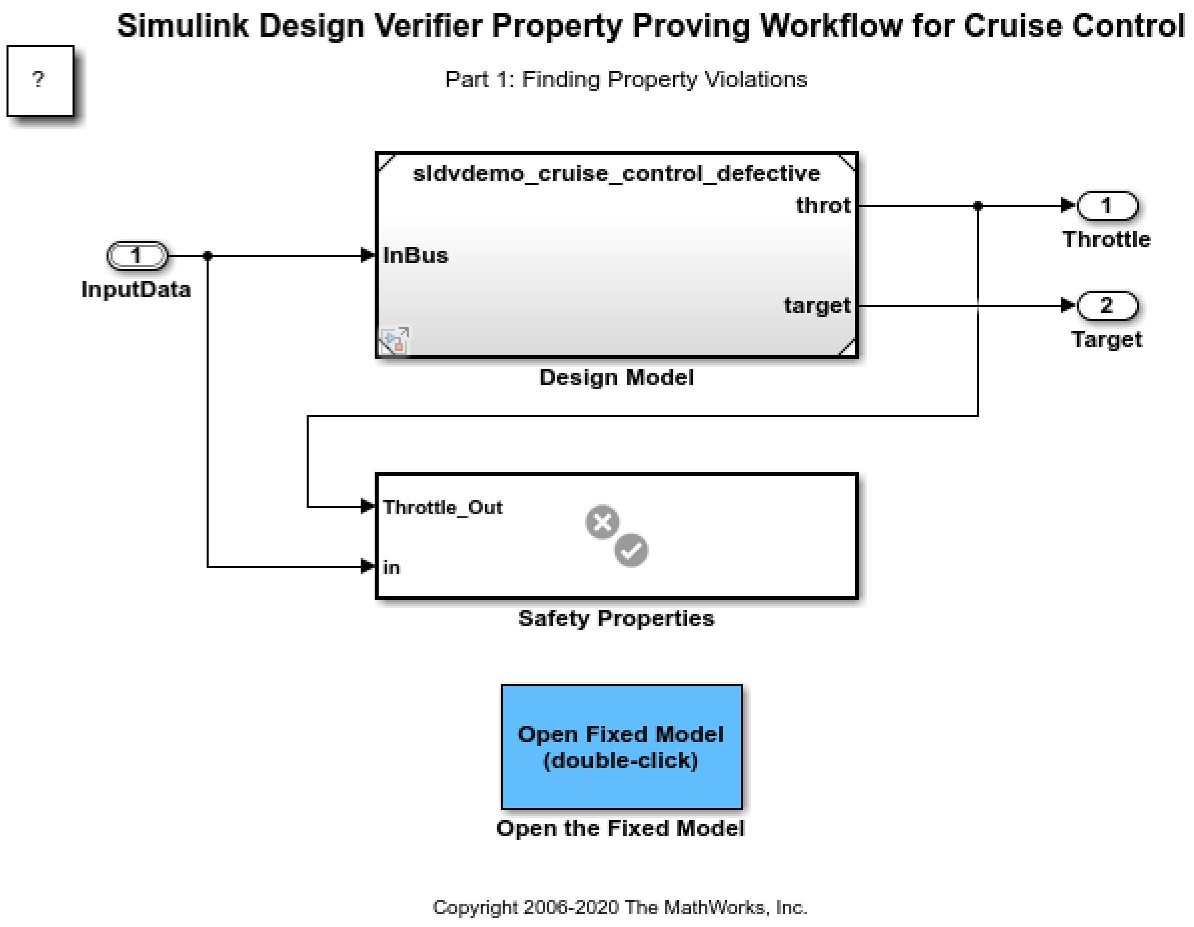
Debug Property Proving Violations by Using Model Slicer

This example shows how to debug property proving violations by using Model Slicer. Consider the model `sldvdemo_cruise_control_verification`. This model contains an **Assertion** block.

The Verification subsystem **Safety Properties** models a property that should hold true for the design model. This subsystem contains an **Assertion** Block (BrakeAssertion) that verifies the property. Simulink Design Verifier Property Proving analysis will try to falsify the assertion. If Simulink Design Verifier is successful it will generate a counterexample falsifying the assertion. We can use Model Slicer to debug this falsified assertion.

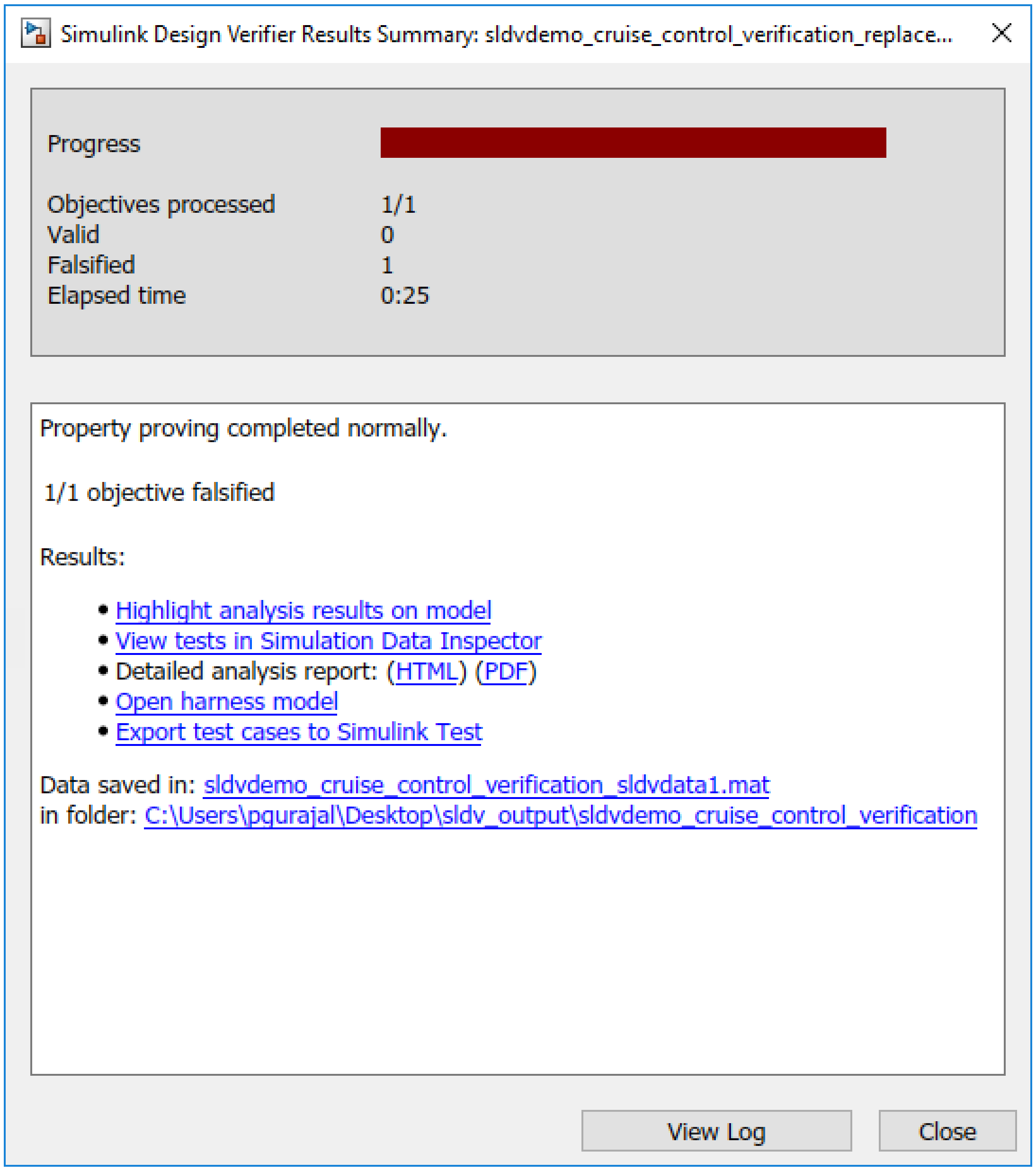
1. Open model **sldvdemo_cruise_control_verification**.

```
open_system ('sldvdemo_cruise_control_verification')
```


Simulink Design Verifier Property Proving Workflow for Cruise Control
?
Part 1: Finding Property Violations
sldvdemo_cruise_control_defective
throt
InBus
target
Design Model
1
InputData
1
Throttle
2
Target
Throttle_Out
in
Safety Properties
Open Fixed Model (double-click)
Open the Fixed Model
Copyright 2006-2020 The MathWorks, Inc.

2. Open Simulink Design Verifier by clicking on **Apps** > **Design Verifier**.

3. Click **Prove Properties**. Simulink Design Verifier analyses the model and displays the results in Results Summary window.


Simulink Design Verifier Results Summary: sldvdemo_cruise_control_verification_replace...

Progress

Objectives processed	1/1
Valid	0
Falsified	1
Elapsed time	0:25

Property proving completed normally.

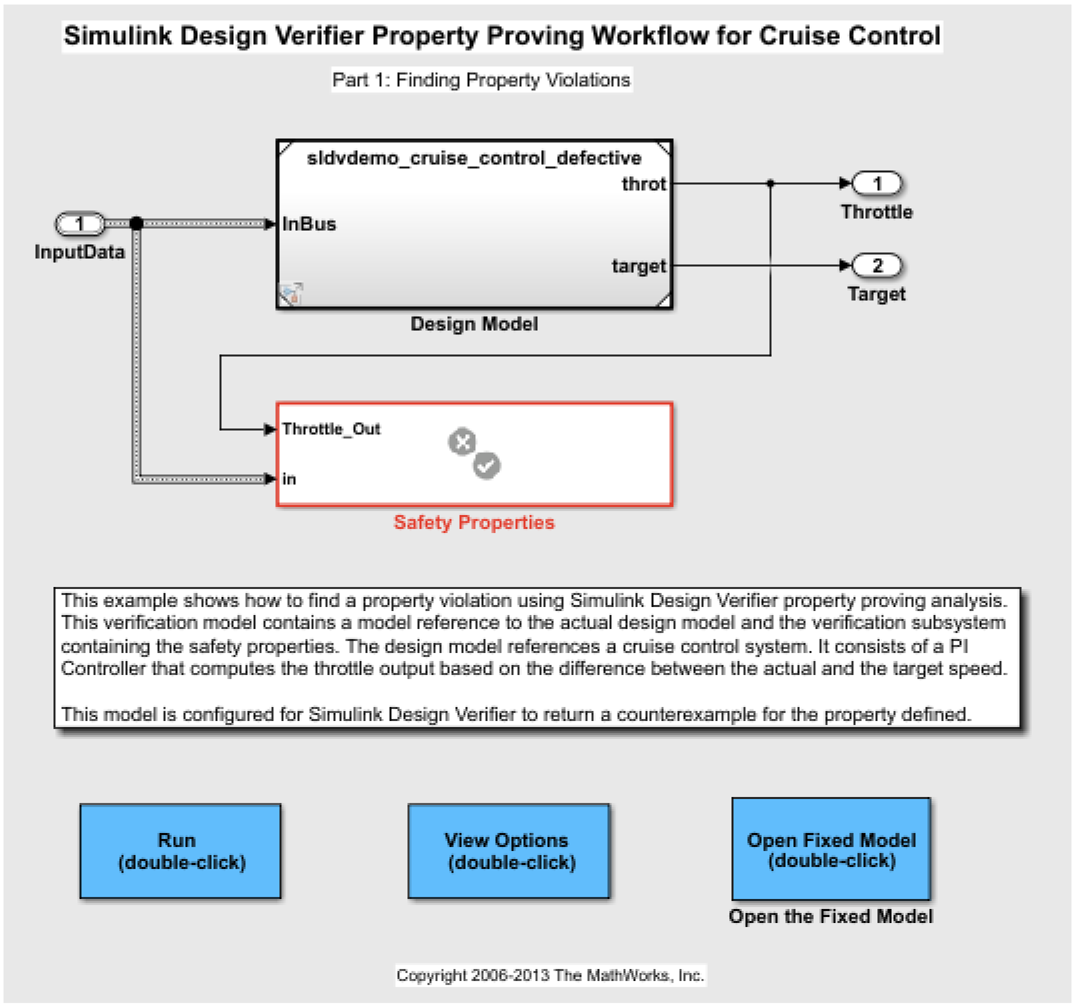
1/1 objective falsified

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Open harness model
- Export test cases to Simulink Test

Data saved in: sldvdemo_cruise_control_verification_sldvdata1.mat
in folder: C:\Users\pgurajal\Desktop\sldv_output\sldvdemo_cruise_control_verification

The model highlights the subsystem where the **Assertion** block is located.



This example shows how to find a property violation using Simulink Design Verifier property proving analysis. This verification model contains a model reference to the actual design model and the verification subsystem containing the safety properties. The design model references a cruise control system. It consists of a PI Controller that computes the throttle output based on the difference between the actual and the target speed.

This model is configured for Simulink Design Verifier to return a counterexample for the property defined.

4. Open Safety Properties subsystem and select the falsified **Assertion** block.

5. Click **Debug Using Slicer** from the toolstrip menu to debug the violation using Model Slicer. Alternatively, you can click **Debug** in the results Inspector window.

On Clicking either of the entry points the following setup is done on the model:

a. The Assertion block is added as a starting point for Model Slicer.

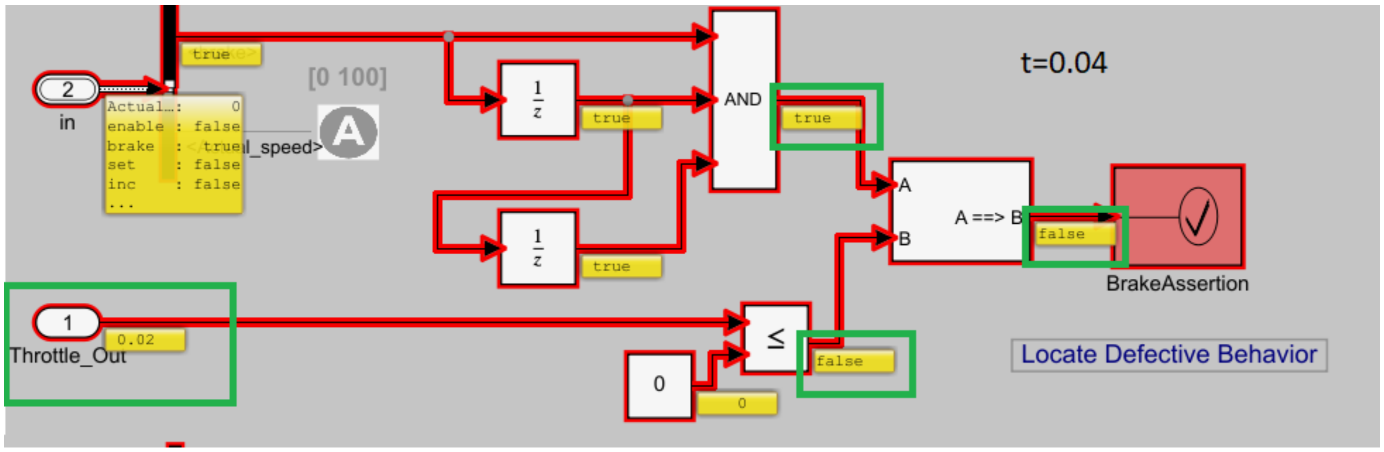
b. The model is highlighted with the counterexample generated by Simulink Design Verifier analysis.

c. The design model is simulated and paused at the time-step of assertion failure.

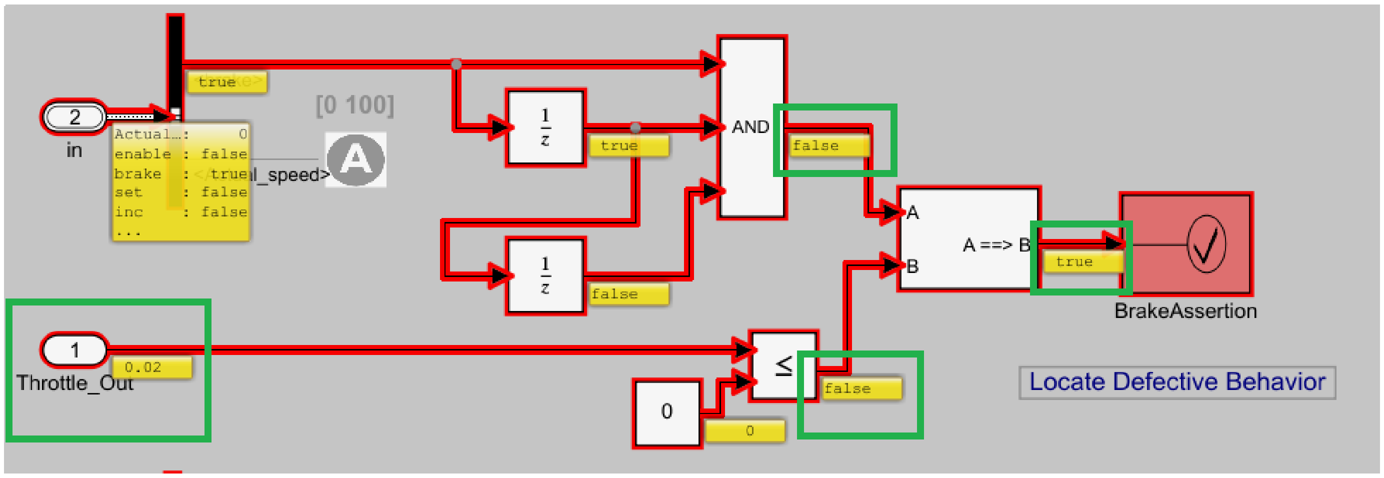
6. Debug and analyze the model by using the Step Back and Step Forward buttons, and inspecting the Port labels.

- The Assert block tests if the output of **A** implies **B** (A==>B) is **false**.
- **A** is **true** when the brake input in is **true** for three consecutive time steps.
- **B** is **true** when the **Throttle_out <= 0**

You can notice that the simulation is stopped at t=0.04 when the condition A==>B is false. This can be observed from the Port labels.

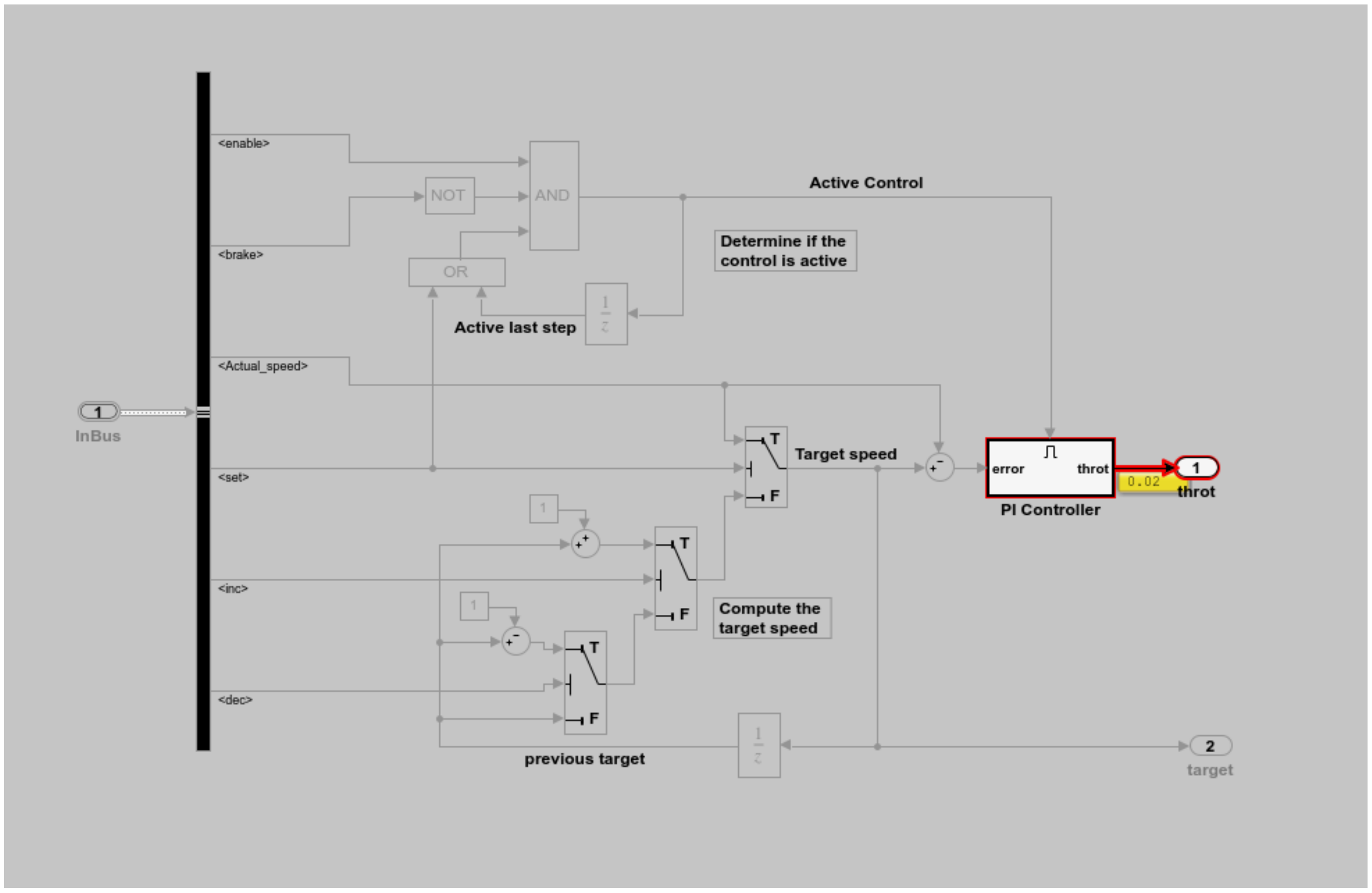


a. On the **Simulation** tab, click the **Step Back** to see the port labels of all the blocks at T = (T-0.1).



You can notice that the Port label of **A** is **false** till T=0.04, when it becomes **true**. At this point the Port label of **B** is **false** (Throttle_Out > 0). The property is falsified because **Throttle_Out** is greater than `0`.

b. To view the blocks that results in the failure, open the **Design Model** > **Controller**. The dependent blocks and path are highlighted.



To view the fix, open `sldvdemo_cruise_control_verification` model and the click the **Open Fixed Model** button on the canvas.

Design and Verify Properties in a Model

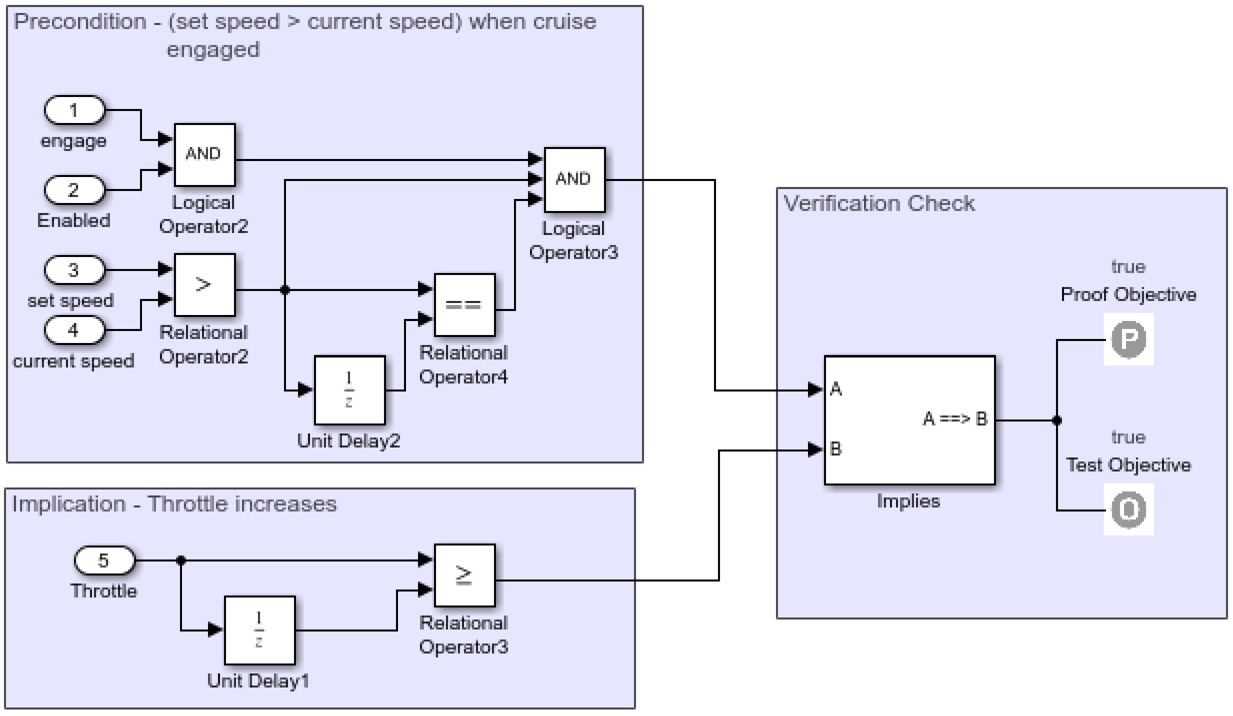
You can use Simulink® Design Verifier™ to model design requirements as properties and then prove properties in a model. To verify that the properties associated with the model requirements hold under all possible input values, use property proving analysis. If the requirement fails, Simulink Design Verifier provides counterexamples to debug the failure.

This example explains how you can model design requirements as properties by using a Proof Objective block and then verify the property for simplified cruise control model discussed in "Analyze a Simple Cruise Control Model".

Step 1: Design Property Using Verification Subsystem

The model `sldvexSimpleCruiseControlProperties` consists of Verification Subsystem, that consists of function requirements modeled by using Proof Objective block.

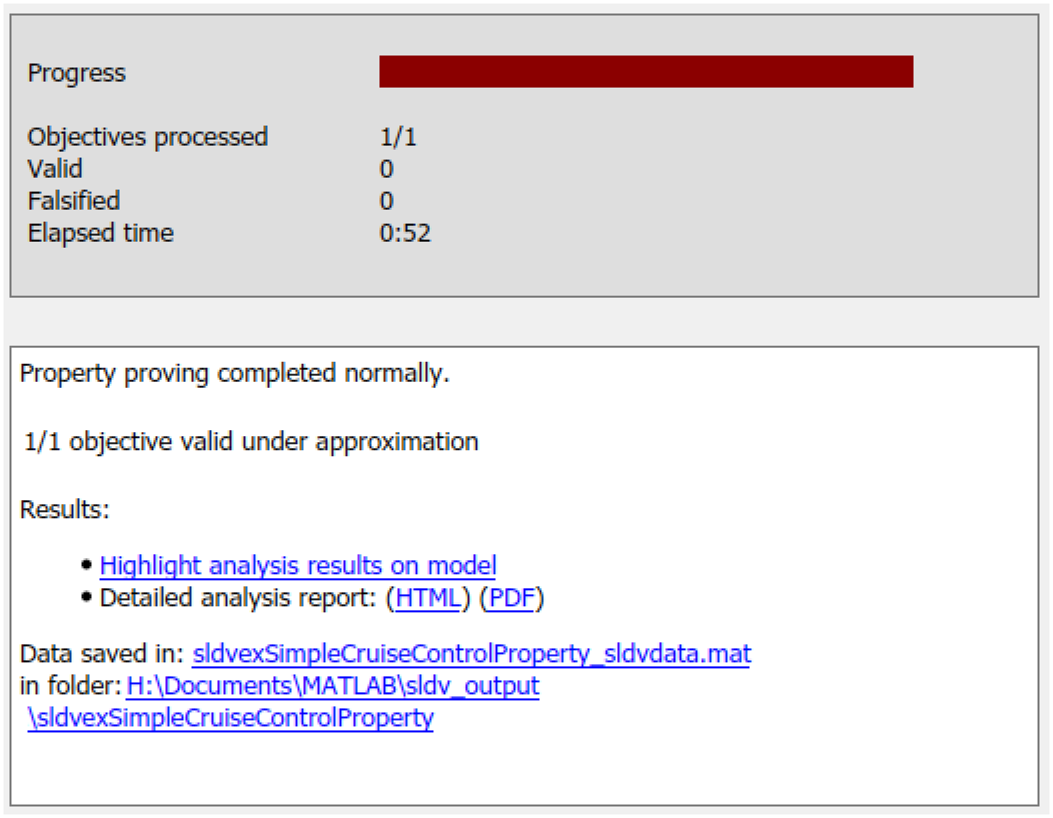
```
load_system('sldvexSimpleCruiseControlProperty');
open_system('sldvexSimpleCruiseControlProperty/Verification Subsystem');
```



Step 2: Perform Property Proving Analysis

On the **Apps** tab, click arrow on the far right of the Apps section. Under **Model Verification, Validation, and Test** gallery, click **Design Verifier**.

To perform property proving analysis, click **Prove Properties**. The software analyzes the model and displays the results in the Results Summary window. The result indicates that one objective is valid under approximation.



Progress

Objectives processed 1/1
Valid 0
Falsified 0
Elapsed time 0:52

Property proving completed normally.

1/1 objective valid under approximation

Results:

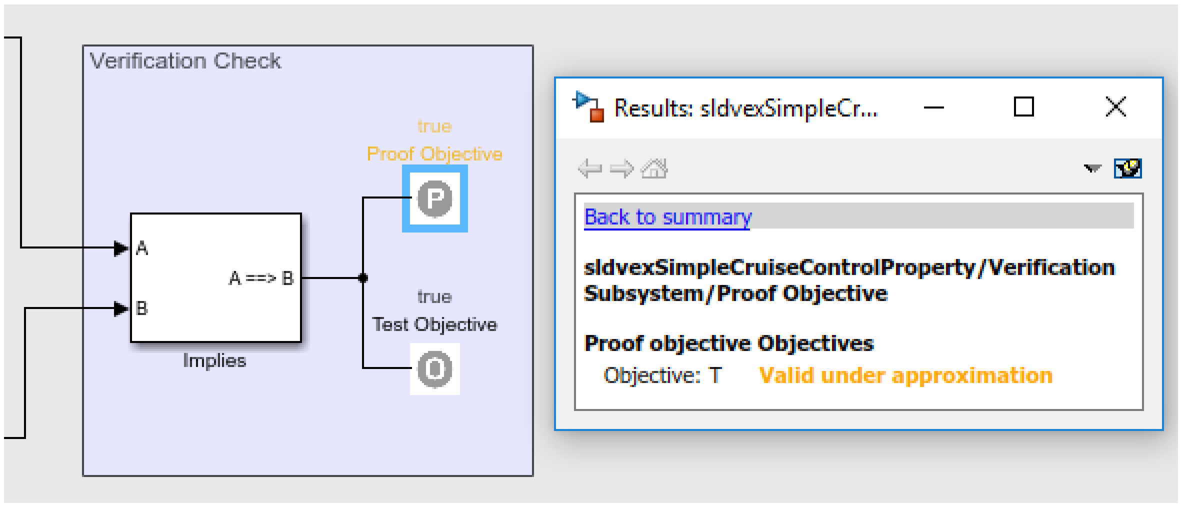
- Highlight analysis results on model
- Detailed analysis report: (HTML) (PDF)

Data saved in: sldvexSimpleCruiseControlProperty_sldvdata.mat
in folder: H:\Documents\MATLAB\sldv_output
\sldvexSimpleCruiseControlProperty

Step 3: Review Analysis Results

On the **Design Verifier** tab, in the Review Results section, click **Highlight in Model**.

The property that is valid under approximation is highlighted in orange. Click the Proof Objective block. The Results Inspector window displays the objectives of the Proof Objective block.

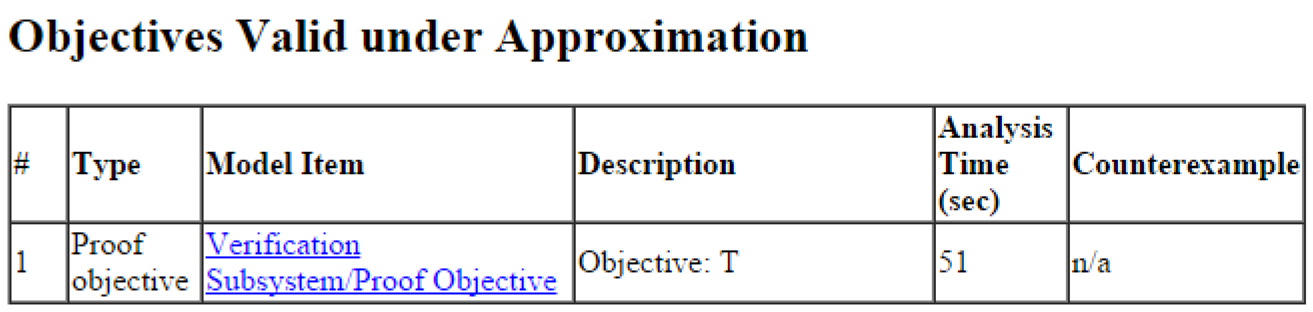


Back to summary

sldvexSimpleCruiseControlProperty/Verification Subsystem/Proof Objective

Proof objective Objectives
Objective: T **Valid under approximation**

To view the HTML report, in the Review Results section, click **HTML Report**. The Proof Objective Status chapter lists the proof objective that is found valid under approximation.



Objectives Valid under Approximation

#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Verification Subsystem/Proof Objective	Objective: T	51	n/a

See also

- “What Is Property Proving?” on page 12-2
- “Prove Properties in a Model” on page 12-5

Reviewing the Results

Highlighted Results on the Model

In this section...

Results Review with Model Highlighting

When you analyze a model by using Simulink Design Verifier, the analyzed model objects are automatically highlighted in one of these colors:

- Green
- Red
- Orange
- Gray

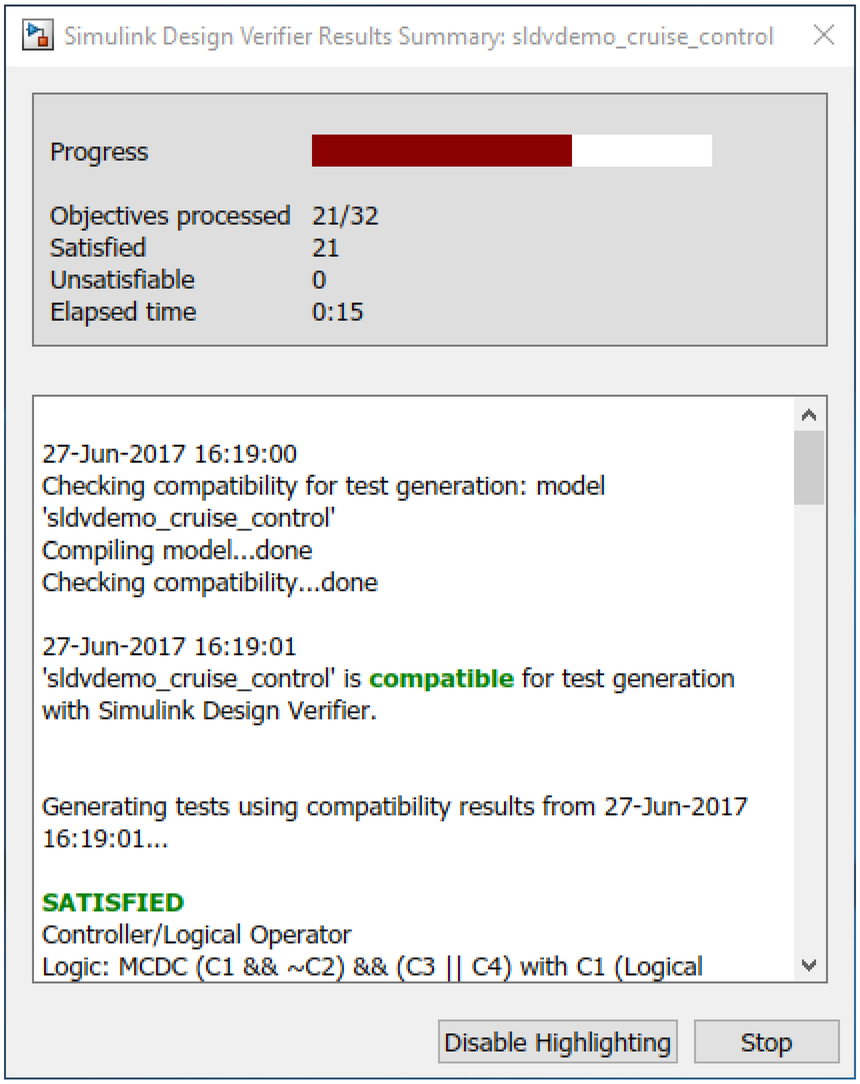
You can review the analysis results at a glance by viewing the objects that are highlighted in the Simulink Editor.

Simulink Design Verifier Results Inspector

When a model is highlighted, you can click an object for which the analysis recorded results. The Simulink Design Verifier Results Inspector then displays the detailed analysis results for that object.

Highlight Results on Model Automatically

During analysis, Simulink Design Verifier highlights the model objects automatically when the objectives status is updated. By default, the automatic highlighting is enabled. To disable the highlighting, click **Disable Highlighting** in the Results Summary window.

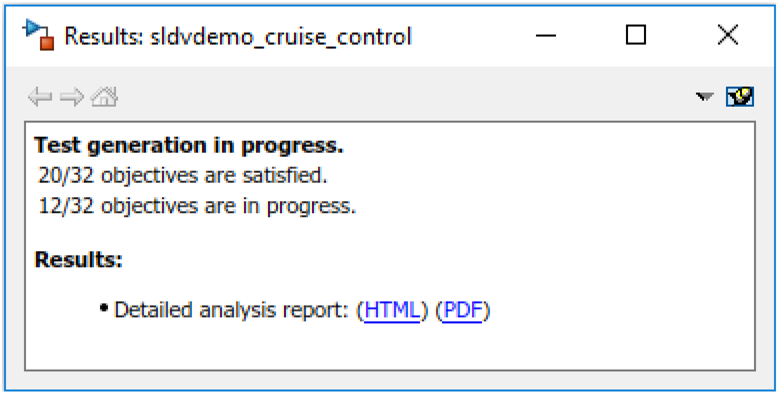

Simulink Design Verifier Results Summary: sldvdemo_cruise_control
Disable Highlighting
Stop

Progress

Objectives processed 21/32
Satisfied 21
Unsatisfiable 0
Elapsed time 0:15

```
27-Jun-2017 16:19:00
Checking compatibility for test generation: model
'sldvdemo_cruise_control'
Compiling model...done
Checking compatibility...done

27-Jun-2017 16:19:01
'sldvdemo_cruise_control' is compatible for test generation
with Simulink Design Verifier.


Generating tests using compatibility results from 27-Jun-2017
16:19:01...

SATISFIED
Controller/Logical Operator
Logic: MCDC (C1 && ~C2) && (C3 || C4) with C1 (Logical
```

In the Simulink Editor, results highlighting appears on the model. When highlighting is enabled, the Results Inspector opens displaying the summary of status for analysis objectives.



Test generation in progress.
20/32 objectives are satisfied.
12/32 objectives are in progress.

Results:

- Detailed analysis report: (HTML) (PDF)

Note Simulink Design Verifier does not highlight the Stateflow state transition tables. The Simulink Design Verifier reports, data files, and log files include the analysis data for the state transition tables. Using the report, you can navigate to the state transition tables.

Green Highlighting on Model

Objects that are highlighted in green have the following meaning for each type of analysis.

Analysis Mode	Green highlighting
Design error detection	• 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 found all the proof objectives as valid.

Red Highlighting on Model

Objects that are highlighted in red have the following meaning, depending on the analysis type.

Analysis Mode	Red highlighting
Design error detection	• 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 did 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 might 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 might 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 rerun the analysis to determine if Simulink Design Verifier highlights 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
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: • 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 more information, see “Approximations” on page 2-19. • The analysis found dead logic that approximations can impact. For more information, see “Reporting Approximations Through Validation Results” on page 2-22. • The analysis found valid objectives that approximations can impact. For more information, see “Reporting Approximations Through Validation Results” on page 2-22.
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 more information, see “Approximations” on page 2-19. • The analysis found unsatisfiable objectives that approximations can impact. For more information, see “Reporting Approximations Through Validation Results” on page 2-22. • The analysis is unable to confirm the satisfied status through validation results. For more information, see “Objectives Satisfied - Needs Simulation” on page 13-37.

Analysis Mode	Orange highlighting
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 more information, see “Approximations” on page 2-19. • The analysis found valid objectives that approximations can impact. For more information, see “Reporting Approximations Through Validation Results” on page 2-22. • The software is unable to confirm the falsified status through validation results. For more information, see “Objectives Falsified - Needs Simulation” on page 13-39.

Gray Highlighting on Model

Objects that are highlighted in gray have the following meaning.

Analysis Mode	Gray Highlighting
• 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

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 13-8
- “AnalysisInformation Field” on page 13-8
- “ModelObjects Field” on page 13-9
- “Constraints Field” on page 13-9
- “Objectives Field” on page 13-9
- “TestCases Field / CounterExamples Field” on page 13-10
- “Version Field” on page 13-11
- “DeadLogic Field” on page 13-11

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.
`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.

Subfield Name	Description
`Approximations`	A structure that describes the approximations performed during the analysis. For more information about approximations, see "Approximations" on page 2-19.
`ReplacementInfo`	For internal use only.
`PreProcessingTime`	Double that specifies the time in seconds to build or reuse the model representation.
`ModelRepresentationInfo`	The date and time of the model representation that is used for analysis.

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. Example: `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.
`status`	The status of an objective.

Subfield Name	Description
`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.
`detectability`	The detectability status of an objective. This field appears in the data file when the analysis “Mode” on page 15-10 is set to `Test Generation` and “Model coverage objectives” on page 15-32 is set to `Enhanced MCDC`.
`detectionSites`	Array of Simulink Identifier (SID) of the detection sites for a detectable objective. The objective is detectable at any one of the detection sites. This field appears in the data file when the analysis “Mode” on page 15-10 is set to `Test Generation` and “Model coverage objectives” on page 15-32 is set to `Enhanced MCDC`.
`modelObjectIdx`	Integer that represents the index of a model object with which an objective is associated.
`analysisistime`	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.
`dataValues`	Cell array that specifies the data values associated with signals in a test case or counterexample.

Subfield Name	Description
`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.

DeadLogic Field

If you analyze your model for dead logic, in the `sldvData` structure, the `DeadLogic` field lists information about each dead logic objective.

This table describes each subfield of the `DeadLogic` field.

Subfield Name	Description
`label`	The description of the dead logic objective.
`descr`	The full path to a model object, including objects in a Stateflow chart.
`modelObjIdx`	Integer that represents the index of a model object that is associated with an objective.
`coverageType`	The type of coverage objective.

Subfield Name	Description
`coverageIdx`	Integer that represents the index of a coverage point that is associated with an objective.
`ObjectiveIdx`	Integer that represents the index of an objective that is associated with a model object.

Simulate Models with Data Files

The `sldvruntest` function simulates a model by 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` 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 13-47 and `sldvloadresults`.

Simulink Design Verifier Harness Models

In this section...

Harness Model Generation

A harness model provides an isolated environment to test design changes. You can create a harness model during Simulink Design Verifier analysis or after the analysis.

The contents of the harness model depends 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 the test cases that result in errors during simulation.
- `Test generation` — The harness model contains the test cases that achieve test objectives.
- `Property proving` — The harness model contains counterexamples that falsify the proof objectives.

By default, the **Generate separate harness model after analysis** parameter is disabled.

Note The Simulink Design Verifier software generates a harness model only when the top-level model that you are analyzing contains an Inport block.

Create a Harness Model

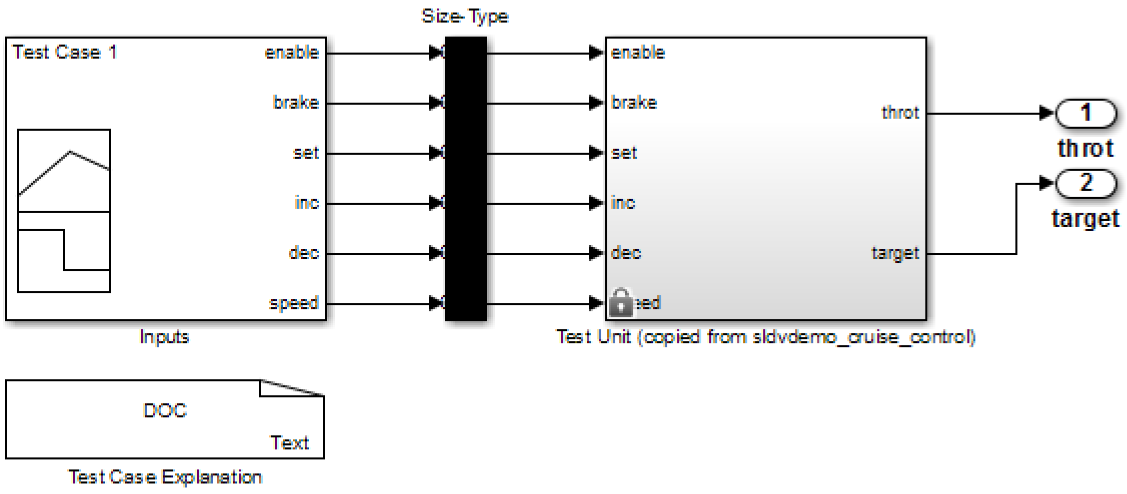
To create a harness model before or after the analysis, use these methods:

- 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 Results Summary window, select **Create harness model**.

Contents of a Harness Model

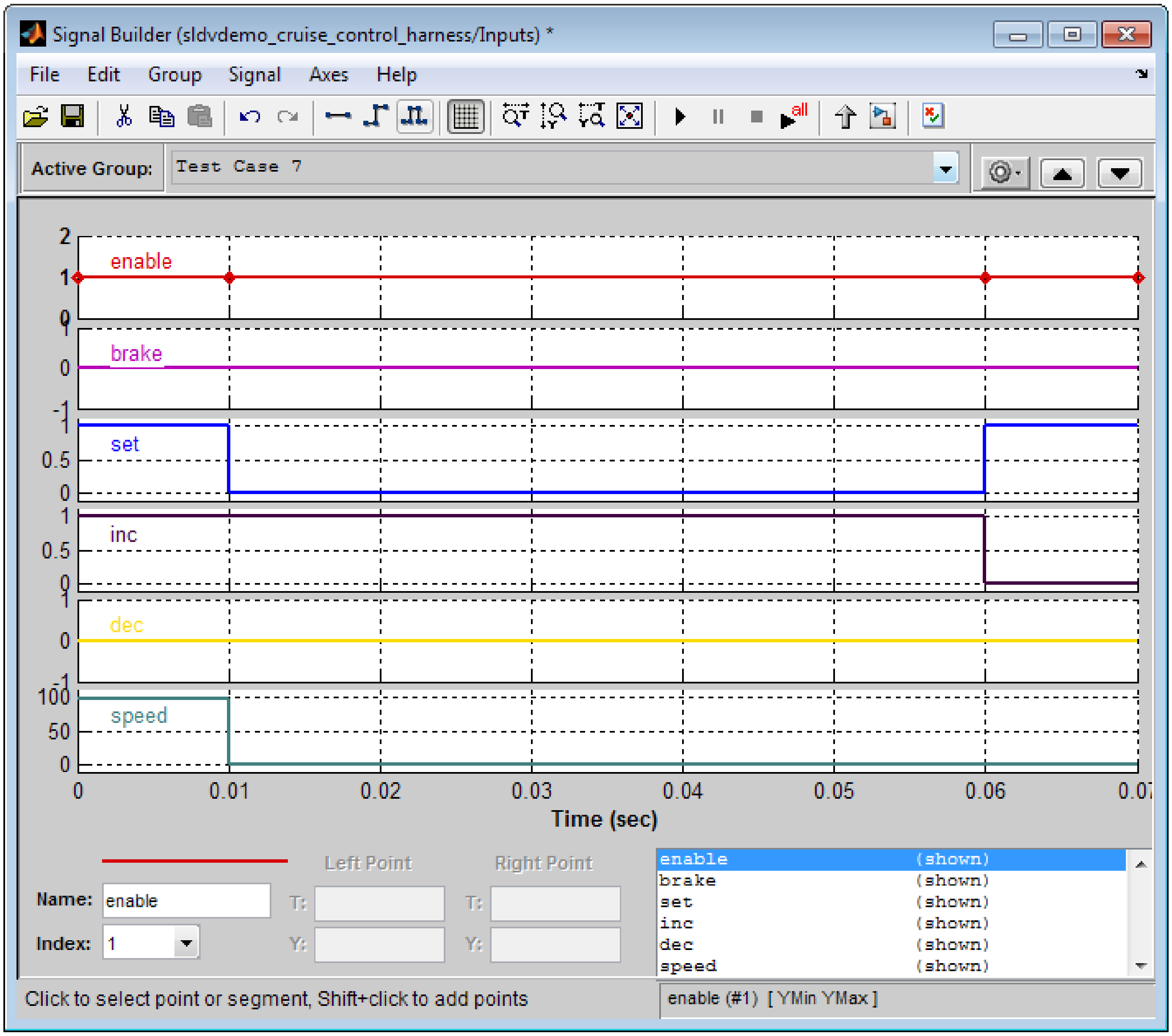
Simulink Design Verifier software creates a harness model that contains these items:

- **Inputs** — The Inputs block is a Signal Builder or Signal Editor block based on the "Harness source" on page 15-60 option set in the **Design Verifier > Results** pane.
  - Signal Builder: This block contains signals that are comprised of the test cases or counterexamples that Simulink Design Verifier generates. 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.



To open the Signal Builder dialog box and view its signals, double-click the Inputs block. Each signal group represents a unique test case or counterexample. To view the signals associated with a particular test case or counterexample, in the Signal Builder dialog box, select **Active Group**.

After Simulink Design Verifier performs test generation analysis on the `sldvdemo_cruise_control` model with the default options, this Signal Builder block shows the signals for `Test Case 7`.



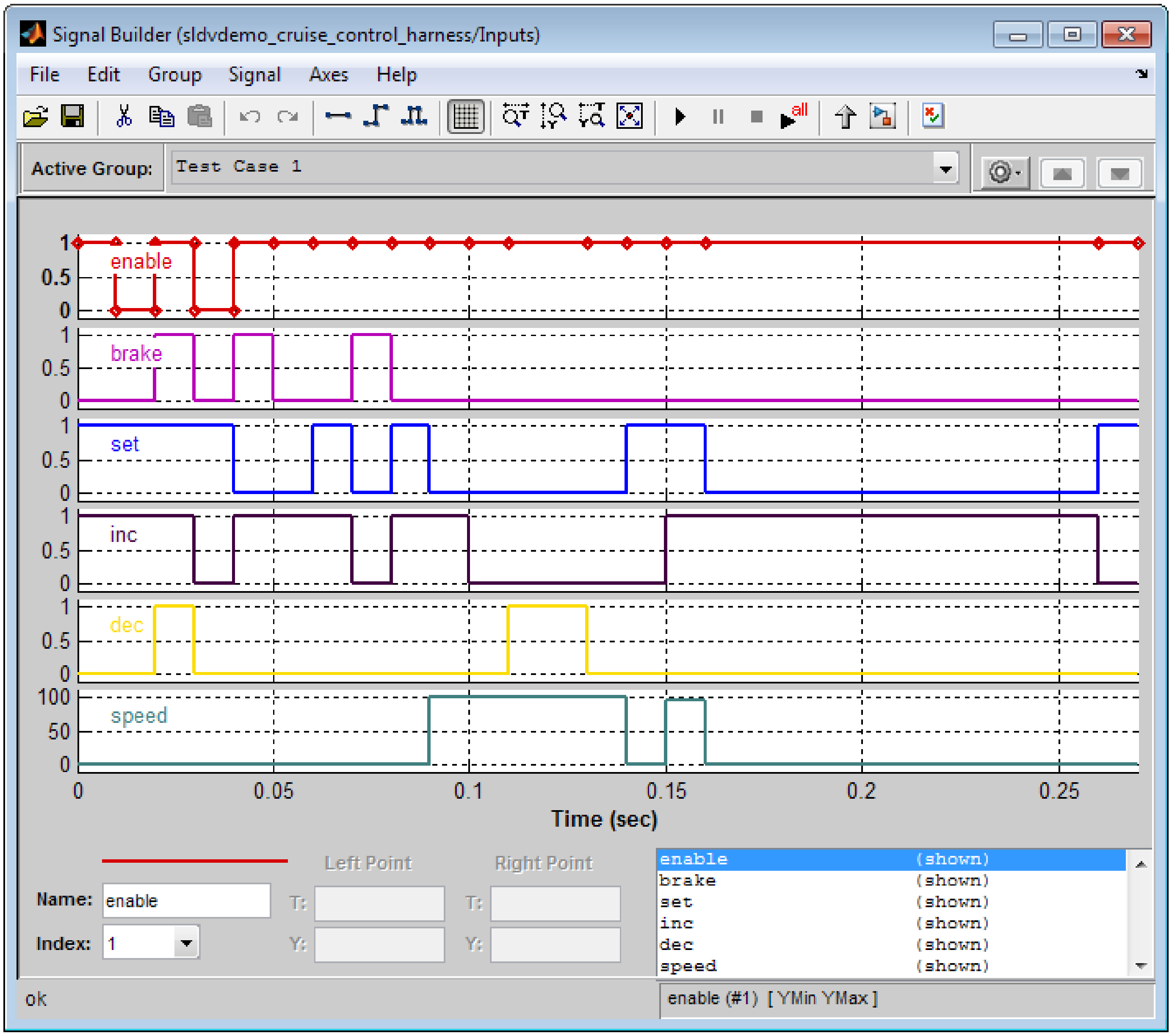
Name: enable
Index: 1

enable	(shown)
brake	(shown)
set	(shown)
inc	(shown)
dec	(shown)
speed	(shown)

Click to select point or segment, Shift+click to add points

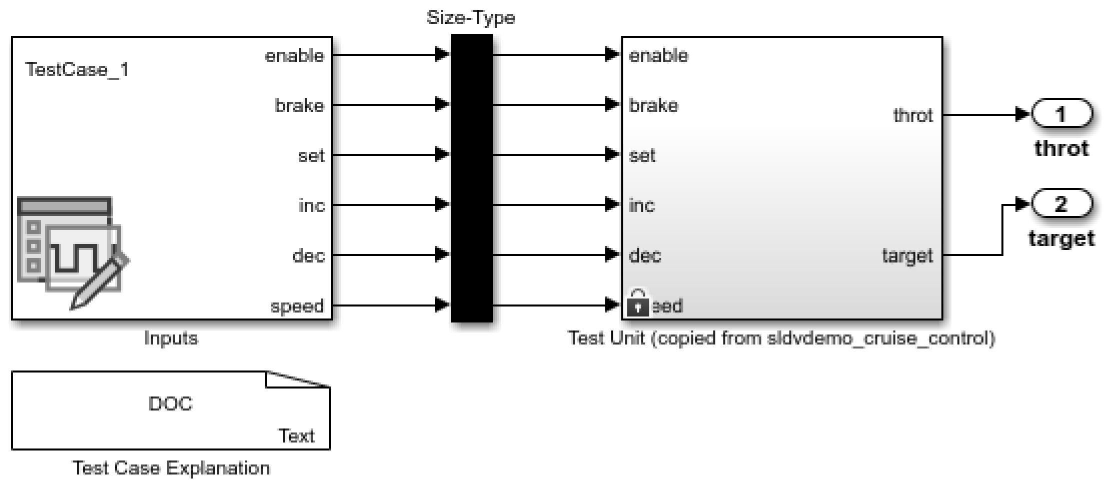
enable (#1) [YMin YMax]

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. This Signal Builder dialog box shows the signals for the long test case. For more information about the Signal Builder dialog box, see “Signal Groups” (Simulink).

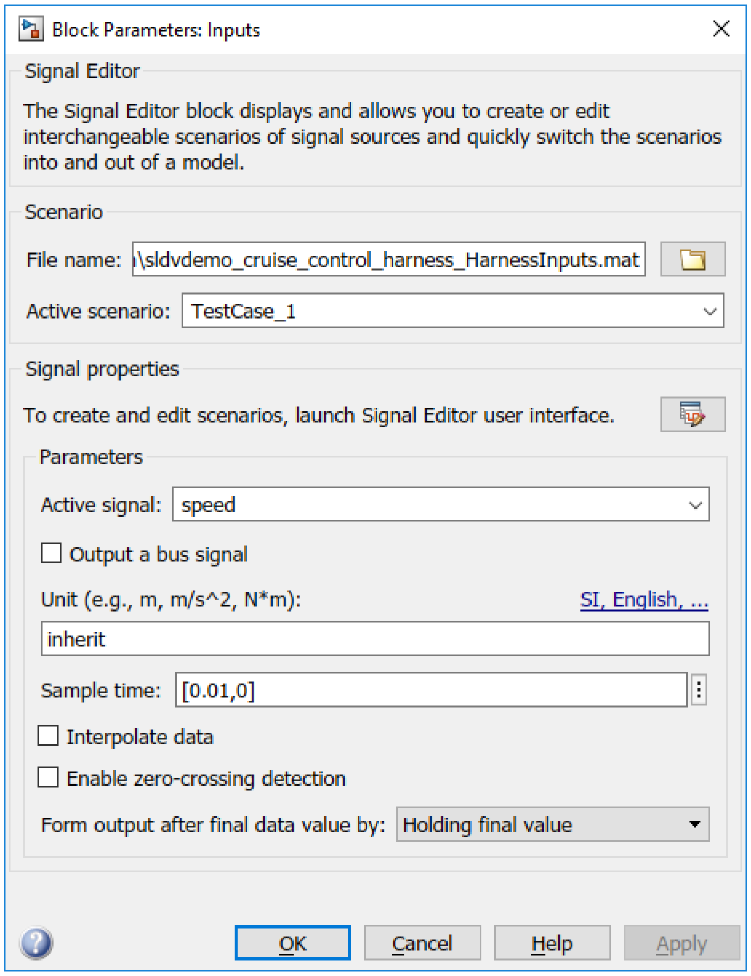


- Signal Editor: This block contains scenarios that are comprised of the test cases or counterexamples that Simulink Design Verifier generates. The Signal Editor block contains signals only for input signals that are used in the model. If an input signal has no effect on the output of the model, that signal is not included in the Signal Editor block.

  After Simulink Design Verifier generates harness model, the input MAT-file for the Signal Editor block is saved at the default location `<current_folder>\sldv_output\<model_name>\<model_name>_harness_HarnessInputs.mat`.



To open the Signal Editor dialog box and view the scenarios of signal sources, double-click the Inputs block. The **Active scenario** lists the test cases or counterexamples. To create and edit scenarios, launch the Signal Editor user interface. For more information, see “Create and Edit Signal Data” (Simulink).



Signal Editor

The Signal Editor block displays and allows you to create or edit interchangeable scenarios of signal sources and quickly switch the scenarios into and out of a model.

Scenario

File name: \sldvdemo_cruise_control_harness_HarnessInputs.mat

Active scenario: TestCase_1

Signal properties

To create and edit scenarios, launch Signal Editor user interface.

Parameters

Active signal: speed

☐ Output a bus signal

Unit (e.g., m, m/s^2, N*m): SI, English, ...

inherit

Sample time: [0.01,0]

☐ Interpolate data

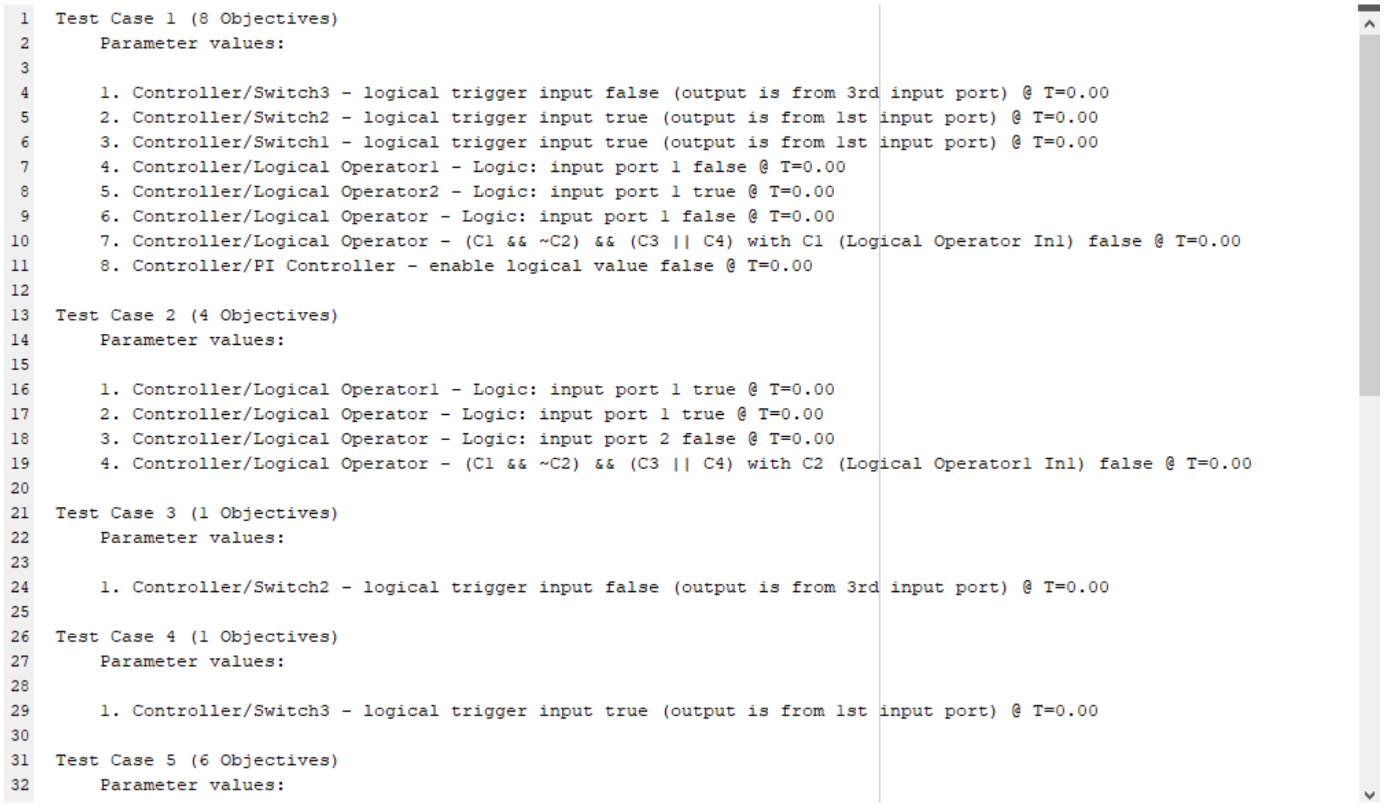
☐ Enable zero-crossing detection

Form output after final data value by: Holding final value

- **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 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 that you are analyzing, not a subsystem.

  If the Test Unit in the harness model is a subsystem, the values of the parameters on the **Optimization** and **Math and Data Types** panes might impact the coverage results.
- **Test Case Explanation** — This DocBlock block documents the test cases or counterexamples that Simulink Design Verifier generates. To view the description of each test case or counterexample, double-click the Test Case Explanation block. The block lists either the test objectives that each test case achieves or the proof objectives that each counterexample falsifies.



```
Test Case 1 (8 Objectives)
    Parameter values:

    1. Controller/Switch3 - logical trigger input false (output is from 3rd input port) @ T=0.00
    2. Controller/Switch2 - logical trigger input true (output is from 1st input port) @ T=0.00
    3. Controller/Switch1 - logical trigger input true (output is from 1st input port) @ T=0.00
    4. Controller/Logical Operator1 - Logic: input port 1 false @ T=0.00
    5. Controller/Logical Operator2 - Logic: input port 1 true @ T=0.00
    6. Controller/Logical Operator - Logic: input port 1 false @ T=0.00
    7. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C1 (Logical Operator In1) false @ T=0.00
    8. Controller/PI Controller - enable logical value false @ T=0.00

Test Case 2 (4 Objectives)
    Parameter values:

    1. Controller/Logical Operator1 - Logic: input port 1 true @ T=0.00
    2. Controller/Logical Operator - Logic: input port 1 true @ T=0.00
    3. Controller/Logical Operator - Logic: input port 2 false @ T=0.00
    4. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C2 (Logical Operator1 In1) false @ T=0.00

Test Case 3 (1 Objectives)
    Parameter values:

    1. Controller/Switch2 - logical trigger input false (output is from 3rd input port) @ T=0.00

Test Case 4 (1 Objectives)
    Parameter values:

    1. Controller/Switch3 - logical trigger input true (output is from 1st input port) @ T=0.00

Test Case 5 (6 Objectives)
    Parameter values:
```

Configuration of the Harness Model

Simulink Design Verifier generates the harness model with these settings.

- The harness model start time is always `0`. If the original model uses a nonzero start time, the software ignores the start time 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 Inputs block.
- By default, the software enables coverage analysis and generates a coverage report for the harness models that contain test cases. The coverage reporting is enabled with default options. You can customize these settings by using “Specify Coverage Options” (Simulink Coverage).
- By default, if you select **Ignore objective based on filter** and provide a coverage filter file for the Test Unit, the coverage filter file applies to the harness model. For more information, see “Coverage data file” on page 15-41.
- The harness model is generated with these Inputs block, regardless of the **Harness source** that you specify:
  - For models that use the complex type Inport block, a Signal Editor block is used as the harness source.
  - For models that use an array of buses as an Inport block, a Signal Builder block is used as the harness source.

Note For models that uses both complex type and array of buses as Inport blocks, harness model generation is not supported.

Simulate the Harness Model

The harness model enables you to simulate a copy of your original model by 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 Coverage software collects and displays model coverage information.
- All the test cases, for which the Simulink Coverage software collects and displays cumulative model coverage information.

Note If you analyze a model that is simulated with sample time warnings, when you simulate the harness model, the warnings might be reported as errors, causing the simulation to fail.

Simulate Harness Model by Using the Signal Builder Source Block

To simulate a single test case or counterexample:

1. In the harness model, double-click the Inputs block.
2. In the Signal Builder dialog box, select the **Active Group** with a particular test case or counterexample.

   The Signal Builder dialog box displays the signals that comprise the selected test case or counterexample.
3. Click the **Start simulation** button ▶.

   The Simulink software simulates the harness model by using the signals associated with the selected test case or counterexample. When simulating a test case, the Simulink Coverage 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.
2. In the Signal Builder dialog box, click the **Run all** button ▶all.

   The Simulink software simulates the harness model by using all test cases, while the Simulink Coverage software collects model coverage information and displays a coverage report.

   When you click **Run all**, the software simulates all the test cases by using the stop time for the harness model. The stop time equals the stop time for the longest test case, so you might accumulate additional coverage when you simulate the shorter test cases.

For more information, see “Simulating with Signal Groups” (Simulink).

Simulate Harness Model by Using the Signal Editor Inputs Block

To simulate a single test case or counterexample:

1. In the harness model, double-click the Inputs block.
2. In the Signal Editor dialog box, select the **Active scenario** with a particular test case or counterexample and click **OK**.
3. In the Simulink editor, click the **Run** button.

   The Simulink software simulates the harness model by using the scenario of signal sources associated with the selected test case or counterexample. When simulating a test case, the Simulink Coverage software collects model coverage information and displays a coverage report.

To simulate all the test cases and measure their combined model coverage, use `cvsim` or `parsim` command. For example, see Simulate Harness Model with Signal Editor Inputs Block on page 13-22.

See Also

"Creating and Executing Test Cases" on page 7-84 | "Create Harness Model" on page 1-13

Simulate Harness Model with Signal Editor Inputs Block

This example shows how to generate model coverage report by simulating the test harness model with the Signal Editor Inputs block. You can simulate a single test case or counterexample by selecting the active scenario in the Signal Editor dialog box. For more information see, “Simulate Harness Model by Using the Signal Editor Inputs Block” on page 13-21.

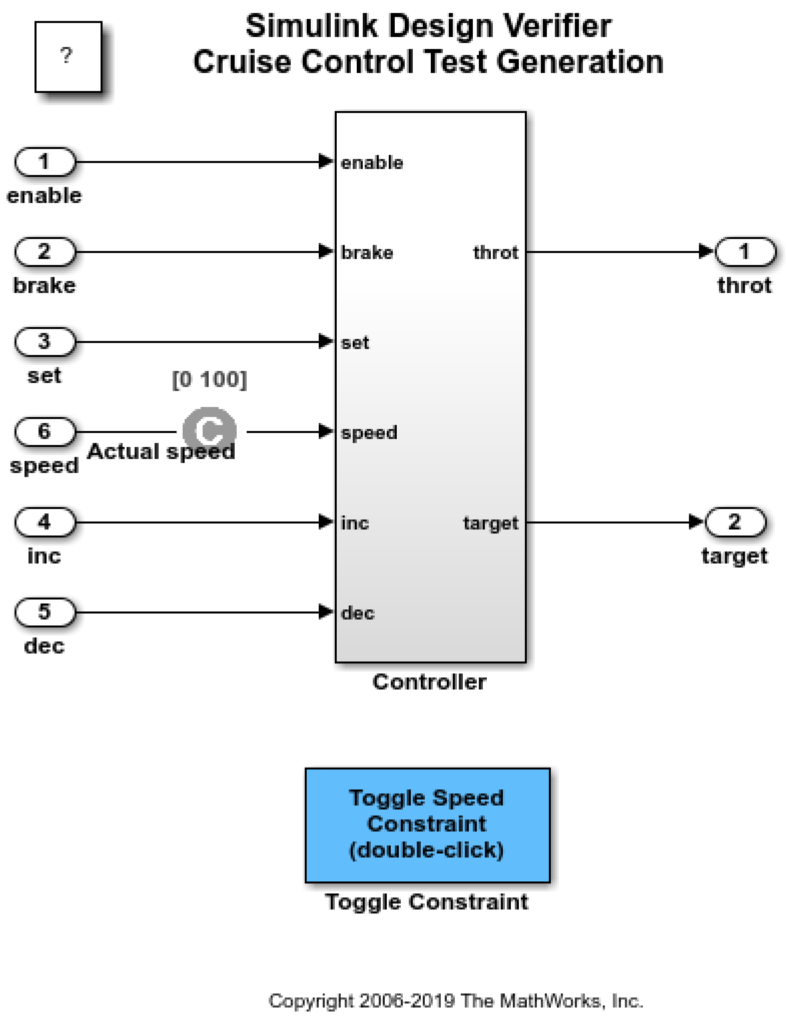
To simulate all the test cases and measure their combined model coverage, use the `cvsim` or the `parsim` command.

In this example, you generate a harness model by selecting the Signal Editor as the harness source. The Signal Editor scenarios consists of signal sources that are associated with the test cases or counterexamples. Then, to generate combined model coverage report, you simulate all the scenarios by using the `cvsim` or `parsim` function.

1. Open the model and configure harness options

Create a harness model for the `sldvdemo_cruise_control` model by using the `sldvharnessopts` options. Set the `HarnessSource` option to `Signal Editor`.

```
model = 'sldvdemo_cruise_control';
open_system(model);
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.SaveHarnessModel = 'on';
opts.HarnessSource = 'Signal Editor';
opts.HarnessModelFileName = 'sldvdemo_cruise_control_harness';
opts.SaveReport = 'off';
```

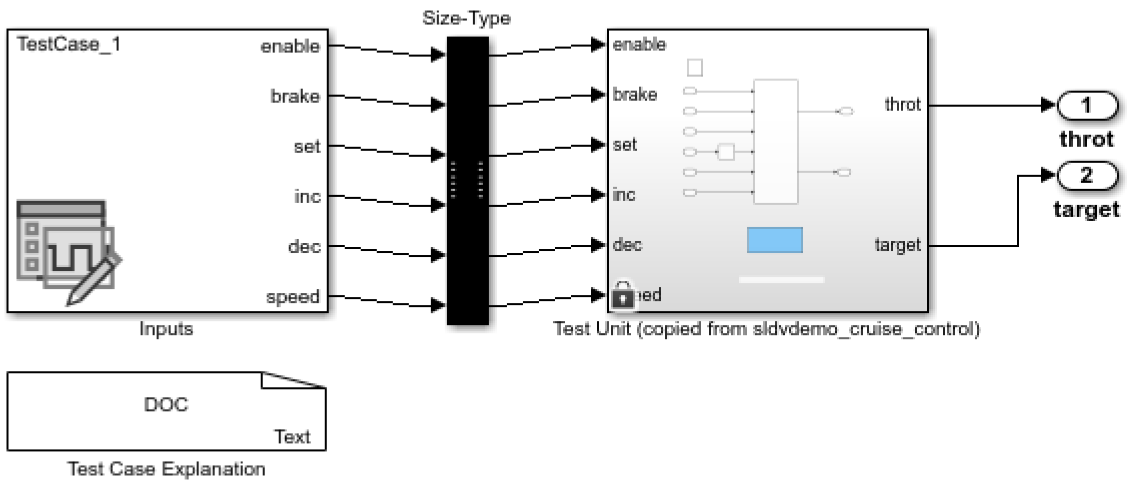


2. Generate test cases

Analyze the model by using the `sldvrun` function and `sldvoptions`.

```
sldvrun('sldvdemo_cruise_control', opts);
save_system('sldvdemo_cruise_control_harness');
```

```
Checking compatibility for test generation: model 'sldvdemo_cruise_control'
Compiling model...done
Building model representation...done

'sldvdemo_cruise_control' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 29-Feb-2020 11:23:46...
........................

Completed normally.

Generating output files:
    Harness model:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex99648832\sldv_output\sldvdemo_cruise_co

Results generation completed.

    Data file:
    C:\TEMP\Bdoc20a_1326390_10420\ibC22023\0\tp86b730ee\ex99648832\sldv_output\sldvdemo_cruise_co
```



3. Generate combined model coverage report

After the analysis generates the harness model, use this code that uses `cvtest` and `cvsim` functions to generate the combined model coverage report.

```
signalEditorBlock = 'sldvdemo_cruise_control_harness/Inputs';
numOfScenarios = str2double(get_param(signalEditorBlock,'NumberOfScenarios'));
harnessModel = 'sldvdemo_cruise_control_harness';
test = cvtest(harnessModel);
test.modelRefSettings.enable = 'On';
test.modelRefSettings.excludeTopModel = 1;
covData = [];
for id = 1:numOfScenarios
set_param(signalEditorBlock,'ActiveScenario',id);
aCovData = cvsim(harnessModel);
if isempty(covData)
covData = aCovData;
else
covData = covData + aCovData;
end
end
save_system('sldvdemo_cruise_control_harness');
cvhtml('Coverage_Harness',covData);
```

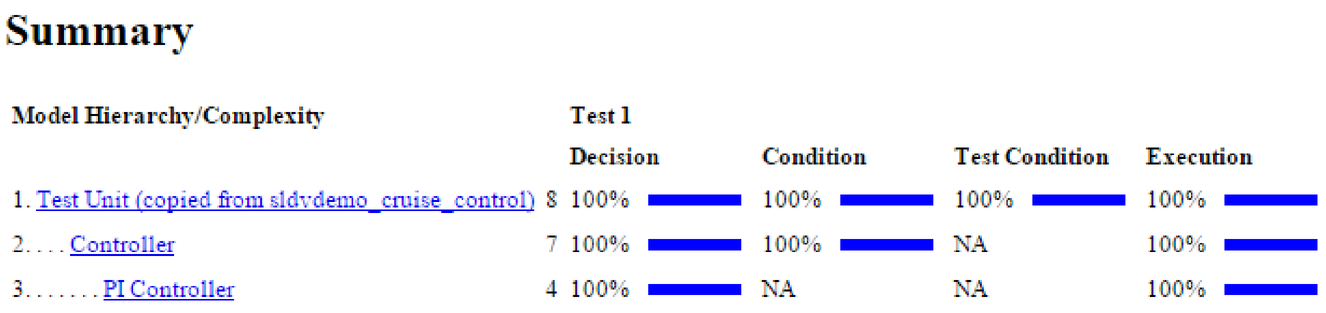
Optionally, you can use this code that uses the `parsim` function to generate the combined model coverage report.

```
signalEditorBlock = 'sldvdemo_cruise_control_harness/Inputs';
numOfScenarios = str2double(get_param(signalEditorBlock,'NumberOfScenarios'));
harnessModel = 'sldvdemo_cruise_control_harness';
```

```
simIn  = Simulink.SimulationInput.empty(0,numOfScenarios);
for id = 1:numOfScenarios
    simIn(id) = Simulink.SimulationInput(harnessModel);
    simIn(id) = simIn(id).setBlockParameter(signalEditorBlock,'ActiveScenario', id);
    simIn(id) = simIn(id).setModelParameter('CovEnable', 'on');
    simIn(id) = simIn(id).setModelParameter('CovSaveSingleToWorkspaceVar', 'on');
end

simOut = parsim(simIn);
cvhtml('Coverage_Harness',simOut.covdata);
```

```
[29-Feb-2020 11:24:42] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).
[29-Feb-2020 11:27:28] Starting Simulink on parallel workers...
[29-Feb-2020 11:27:48] Configuring simulation cache folder on parallel workers...
[29-Feb-2020 11:27:48] Loading model on parallel workers...
[29-Feb-2020 11:27:57] Running simulations...
[29-Feb-2020 11:28:07] Completed 1 of 3 simulation runs
[29-Feb-2020 11:28:07] Completed 2 of 3 simulation runs
[29-Feb-2020 11:28:07] Completed 3 of 3 simulation runs
[29-Feb-2020 11:28:07] Cleaning up parallel workers...
```

The coverage report indicates that 100% coverage is achieved by simulating all the test cases for `sldvdemo_cruise_control_model`.



Summary

Model Hierarchy/Complexity		Test 1			
		Decision	Condition	Test Condition	Execution
1. Test Unit (copied from sldvdemo_cruise_control)	8	100%	100%	100%	100%
2. . . . Controller	7	100%	100%	NA	100%
3. . . . . . . PI Controller	4	100%	NA	NA	100%

5. Clean Up

```
% To complete this example, close the models.
close_system('sldvdemo_cruise_control_harness', 0);
close_system('sldvdemo_cruise_control', 0);
```

Export Test Cases to Simulink Test

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 **Load Earlier Results** in the **Design Verifier** tab. 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.

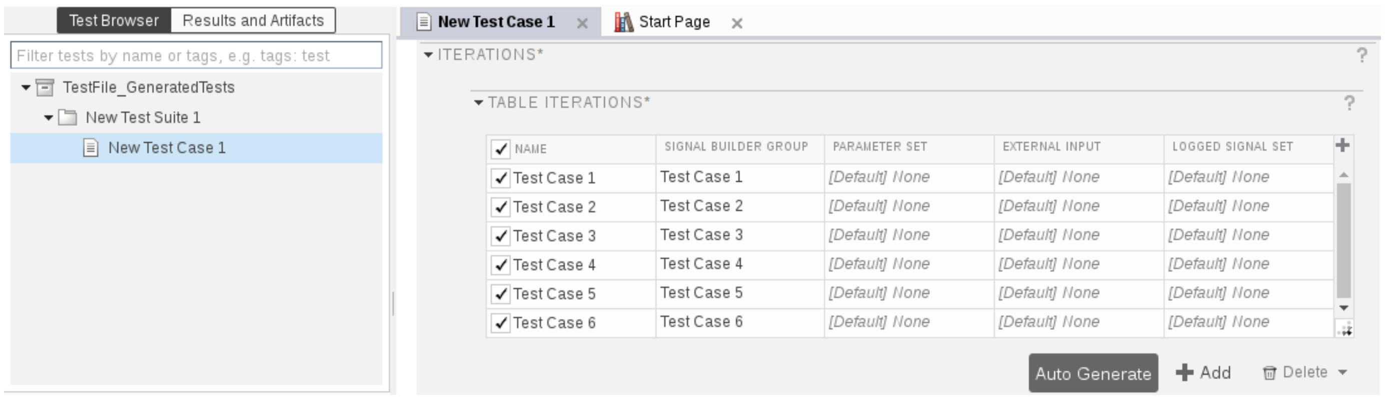
1. Open the model.

   ```
   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 six new test cases in the test file.



Test Browser | Results and Artifacts

Filter tests by name or tags, e.g. tags: test

- TestFile_GeneratedTests
  - New Test Suite 1
    - New Test Case 1

New Test Case 1 | Start Page

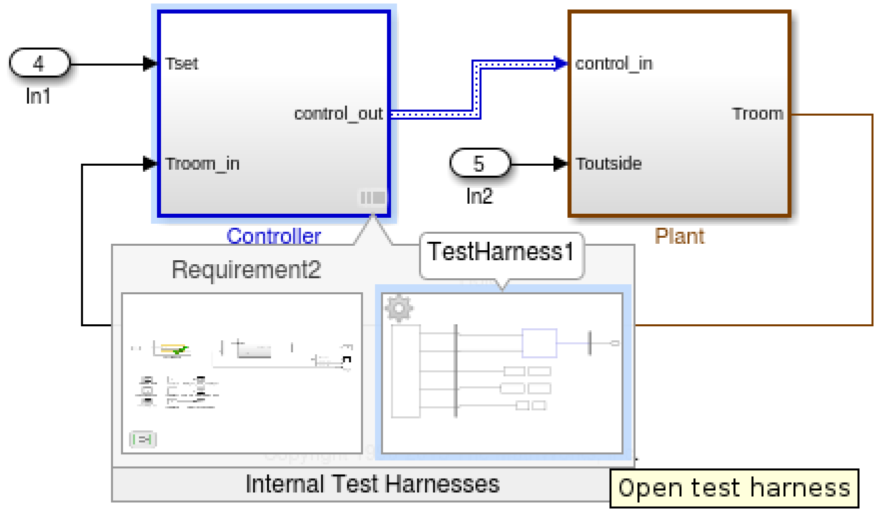
ITERATIONS*

TABLE ITERATIONS*

✓ NAME	SIGNAL BUILDER GROUP	PARAMETER SET	EXTERNAL INPUT	LOGGED SIGNAL SET
✓ Test Case 1	Test Case 1	[Default] None	[Default] None	[Default] None
✓ Test Case 2	Test Case 2	[Default] None	[Default] None	[Default] None
✓ Test Case 3	Test Case 3	[Default] None	[Default] None	[Default] None
✓ Test Case 4	Test Case 4	[Default] None	[Default] None	[Default] None
✓ Test Case 5	Test Case 5	[Default] None	[Default] None	[Default] None
✓ Test Case 6	Test Case 6	[Default] None	[Default] None	[Default] None

Auto Generate | Add | Delete

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 report**.

Front Matter

The report begins with two sections:

- “Title” on page 13-28
- “Table of Contents” on page 13-29

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

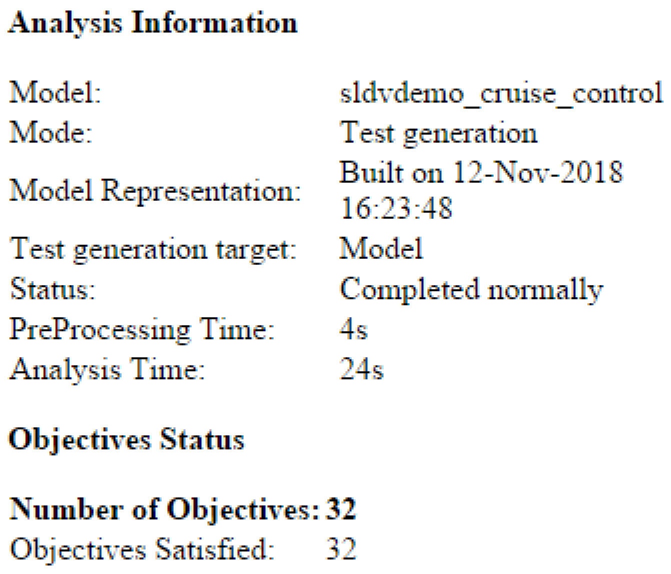
Table of Contents

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
- Model Representation
- Analysis status
- Preprocessing time
- Analysis time
- Status of objectives analyzed



Analysis Information

Model:	sldvdemo_cruise_control
Mode:	Test generation
Model Representation:	Built on 12-Nov-2018 16:23:48
Test generation target:	Model
Status:	Completed normally
PreProcessing Time:	4s
Analysis Time:	24s

Objectives Status

Number of Objectives:	**32**
Objectives Satisfied:	32

Analysis Information Chapter

The **Analysis Information** chapter of the report includes the following sections:

- “Model Information” on page 13-30
- “Analysis Options” on page 13-30
- “Unsupported Blocks” on page 13-31
- “Constraints” on page 13-31

- “Block Replacements Summary” on page 13-31
- “Approximations” on page 13-32

Model Information

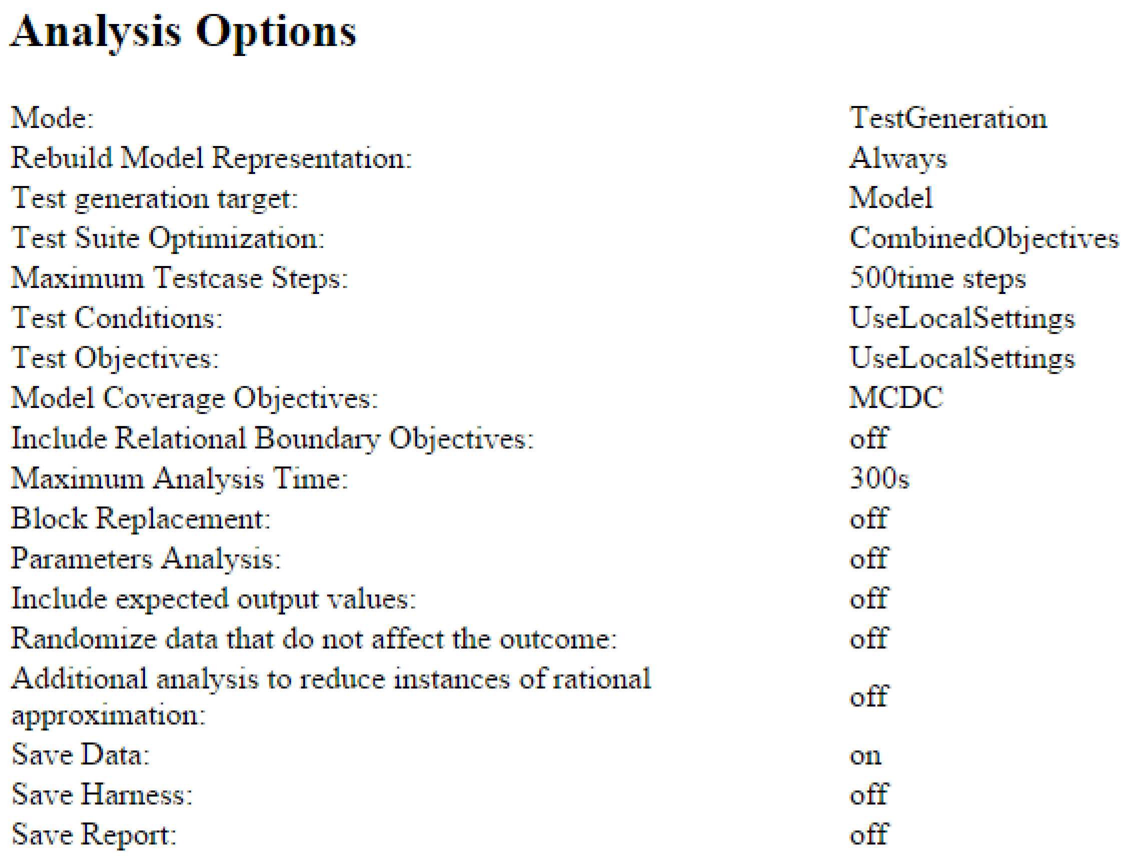
The Model Information section provides the following information about the current version of the model:

- 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

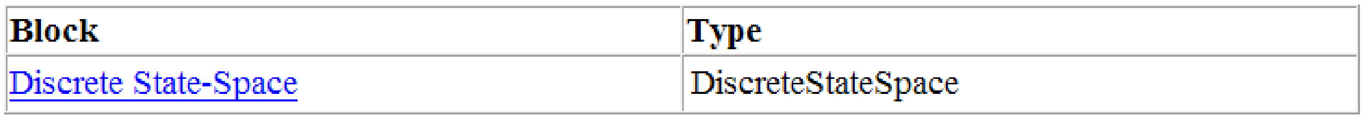
Mode:	TestGeneration
Rebuild Model Representation:	Always
Test generation target:	Model
Test Suite Optimization:	CombinedObjectives
Maximum Testcase Steps:	500time steps
Test Conditions:	UseLocalSettings
Test Objectives:	UseLocalSettings
Model Coverage Objectives:	MCDC
Include Relational Boundary Objectives:	off
Maximum Analysis Time:	300s
Block Replacement:	off
Parameters Analysis:	off
Include expected output values:	off
Randomize data that do not affect the outcome:	off
Additional analysis to reduce instances of rational approximation:	off
Save Data:	on
Save Harness:	off
Save Report:	off

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 blocks in a table, with a hyperlink to each block in the model.

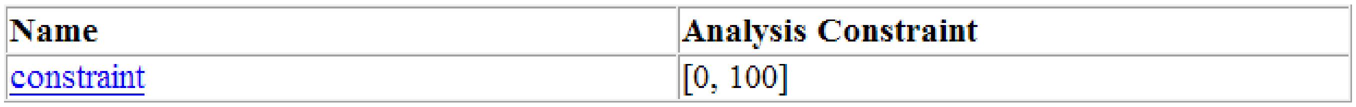


Block	Type
Discrete State-Space	DiscreteStateSpace

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.



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.

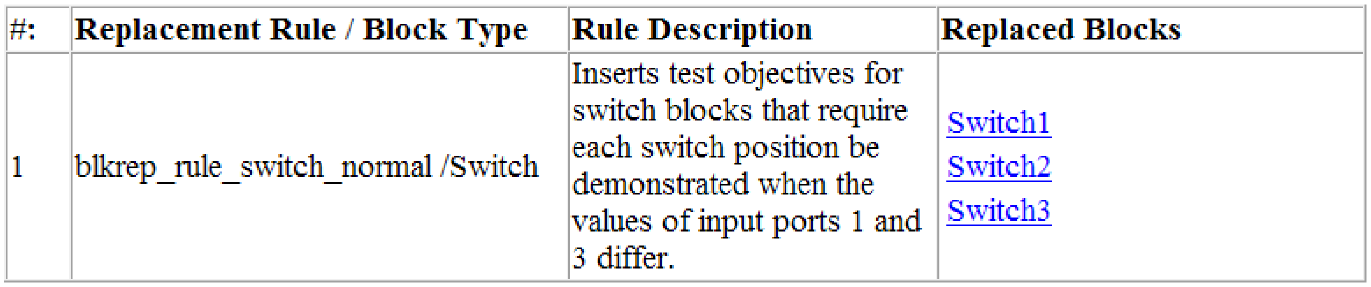
Block Replacements Summary

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

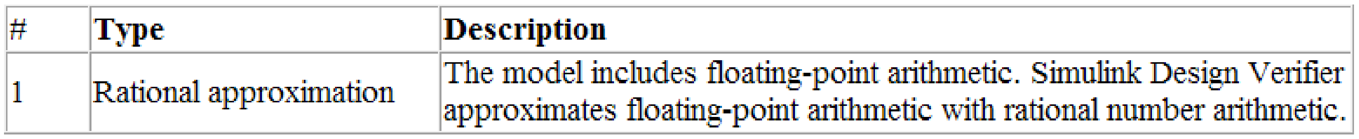
To locate a particular block replacement in your model, click on the name for that replacement in the Replaced Blocks column of the table; the software highlights the affected block in your model window and opens a new window that displays the block in detail.



#:	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.



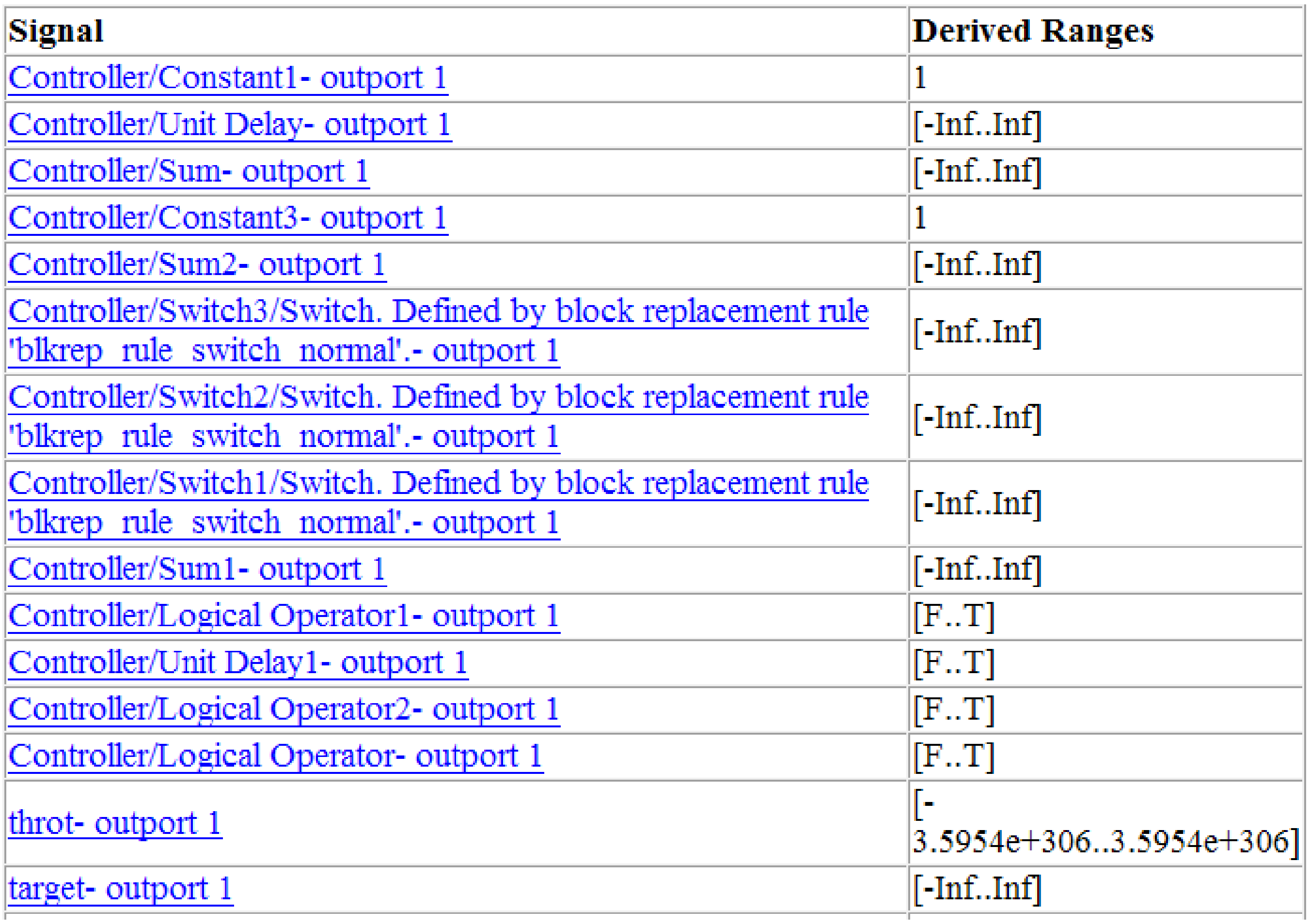
#	Type	Description
1	Rational approximation	The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.

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.



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 the objectives in a model, including the type of the objective, the model item that corresponds to the type, and objective 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 due to Runtime Error” on page 13-39
- “Objectives Undecided Due to Division by Zero” on page 13-39
- “Objectives Undecided Due to Nonlinearities” on page 13-40
- “Objectives Undecided Due to Stubbing” on page 13-40
- “Objectives Undecided Due to Array Out of Bounds” on page 13-40
- “Objectives Undecided” on page 13-40

The software identifies the presence of approximations and reports them at the level of each objective status. For more information, see “Reporting Approximations Through Validation Results” on page 2-22. This table summarizes the objective status for Simulink Design Verifier analysis modes.

Analysis Mode	Objective Status
Design error detection	• “Dead Logic” on page 13-35 • “Dead Logic under Approximation” on page 13-35 • “Active Logic - Needs Simulation” on page 13-35 • “Objectives Valid” on page 13-35 • “Objectives Valid under Approximation” on page 13-36 • “Objectives Falsified - Needs Simulation” on page 13-36 • “Objectives Undecided Due to Division by Zero” on page 13-39 • “Objectives Undecided Due to Nonlinearities” on page 13-40 • “Objectives Undecided Due to Stubbing” on page 13-40 • “Objectives Undecided” on page 13-40 • “Objectives Undecided Due to Array Out of Bounds” on page 13-40
Test generation	• “Objectives Satisfied” on page 13-36 • “Objectives Satisfied - Needs Simulation” on page 13-37 • “Objectives Unsatisfiable” on page 13-37 • “Objectives Unsatisfiable under Approximation” on page 13-37 • “Objectives Undecided with Testcases” on page 13-38 • “Objectives Undecided due to Runtime Error” on page 13-39 • “Objectives Undecided Due to Division by Zero” on page 13-39 • “Objectives Undecided Due to Nonlinearities” on page 13-40 • “Objectives Undecided Due to Stubbing” on page 13-40 • “Objectives Undecided” on page 13-40 • “Objectives Undecided Due to Array Out of Bounds” on page 13-40
Property proving	• “Objectives Valid” on page 13-38 • “Objectives Valid under Approximation” on page 13-38 • “Objectives Falsified with Counterexamples” on page 13-38 • “Objectives Falsified - Needs Simulation” on page 13-39 • “Objectives Undecided with Counterexamples” on page 13-39 • “Objectives Undecided due to Runtime Error” on page 13-39 • “Objectives Undecided Due to Division by Zero” on page 13-39 • “Objectives Undecided Due to Nonlinearities” on page 13-40 • “Objectives Undecided Due to Stubbing” on page 13-40 • “Objectives Undecided” on page 13-40 • “Objectives Undecided Due to Array Out of Bounds” 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 objective statuses:

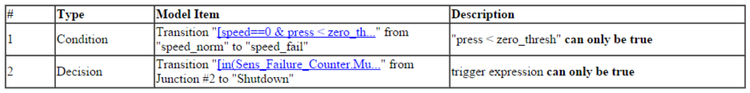
- “Dead Logic” on page 13-35

- “Dead Logic under Approximation” on page 13-35
- “Active Logic - Needs Simulation” on page 13-35
- “Objectives Valid” on page 13-35
- “Objectives Valid under Approximation” on page 13-36
- “Objectives Falsified - Needs Simulation” on page 13-36

Dead Logic

The **Dead Logic** section lists the model items for which the analysis found dead logic.

This image shows the **Dead Logic** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` model.



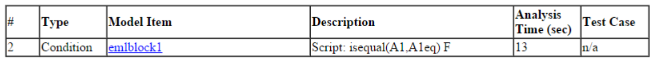
#	Type	Model Item	Description
1	Condition	Transition "[speed==0 & press < zero_th..." from "speed_norm" to "speed_fail"	"press < zero_thresh" **can only be true**
2	Decision	Transition "[in(Sens_Failure_Counter.Mu..." from Junction #2 to "Shutdown"	trigger expression **can only be true**

Dead Logic under Approximation

The **Dead Logic under Approximation** section lists the model items for which the analysis found dead logic under the impact of approximation.

In releases before R2017b, this section can include objectives that were marked as **Dead Logic**.

This image shows the **Dead Logic under Approximation** section of the generated analysis report.



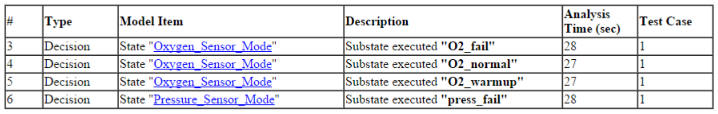
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
2	Condition	emlblock1	Script: isequal(A1,A1eq) F	13	n/a

Active Logic - Needs Simulation

The **Active Logic - Needs Simulation** section lists the model items for which the analysis found active logic. To confirm the active logic status, you must run additional simulations of test cases.

In releases before R2017b, this section can include objectives that were marked as **Active Logic**.

This image shows a portion of the **Active Logic - Needs Simulation** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` model.



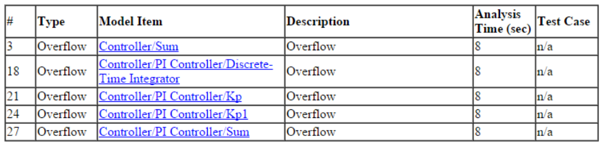
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
3	Decision	State "Oxygen_Sensor_Mode"	Substate executed **"O2_fail"**	28	1
4	Decision	State "Oxygen_Sensor_Mode"	Substate executed **"O2_normal"**	27	1
5	Decision	State "Oxygen_Sensor_Mode"	Substate executed **"O2_warmup"**	27	1
6	Decision	State "Pressure_Sensor_Mode"	Substate executed **"press_fail"**	28	1

Objectives Valid

The **Objectives Valid** section lists the design error detection objectives that the analysis found valid. For these objectives, the analysis determined that the described design errors cannot occur.

In releases before R2017b, this section can include objectives that were marked as **Proven Valid**.

This image shows the **Objectives Valid** section of the generated analysis report for the `sldvdemo_design_error_detection` model.



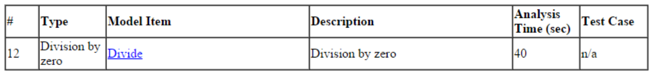
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
3	Overflow	Controller/Sum	Overflow	8	n/a
18	Overflow	Controller/PI Controller/Discrete-Time Integrator	Overflow	8	n/a
21	Overflow	Controller/PI Controller/Kp	Overflow	8	n/a
24	Overflow	Controller/PI Controller/Kp1	Overflow	8	n/a
27	Overflow	Controller/PI Controller/Sum	Overflow	8	n/a

Objectives Valid under Approximation

The **Objectives Valid under Approximation** section lists the design error detection objectives that the analysis found valid under the impact of approximation.

In releases before R2017b, this section can include objectives that were marked as **Proven Valid**.

This image shows the **Objectives Valid under Approximation** section of the generated analysis report.



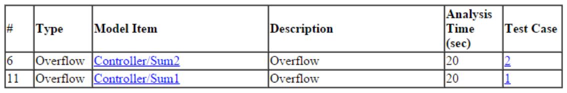
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
12	Division by zero	Divide	Division by zero	40	n/a

Objectives Falsified - Needs Simulation

The **Objectives Falsified - Needs Simulation** section lists the design error detection objectives for which the analysis found test cases that demonstrate design errors. To confirm the falsified status, you must run additional simulations of test cases.

In releases before R2017b, this section can include objectives that were marked as **Falsified**.

This image shows the **Objectives Falsified - Needs Simulation** section of the generated analysis report for the `sldvdemo_design_error_detection` model.



#	Type	Model Item	Description	Analysis Time (sec)	Test Case
6	Overflow	Controller/Sum2	Overflow	20	2
11	Overflow	Controller/Sum1	Overflow	20	1

Test Objectives Status

If you run a test case generation analysis, the **Test Objectives Status** section can include the following objective statuses:

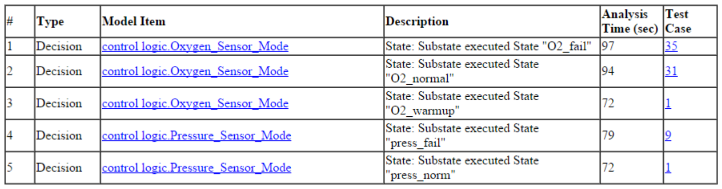
- “Objectives Satisfied” on page 13-36
- “Objectives Satisfied - Needs Simulation” on page 13-37
- “Objectives Unsatisfiable” on page 13-37
- “Objectives Unsatisfiable under Approximation” on page 13-37
- “Objectives Undecided with Testcases” on page 13-38

When you analyze a model with **Model coverage objectives** set to `Enhanced MCDC`, the software reports the detection status of model items. For more information, see “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-35.

Objectives Satisfied

The **Objectives Satisfied** section lists the test objectives that the analysis satisfied. The generated test cases cover the objectives.

This image shows a portion of the **Objectives Satisfied** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` example model.



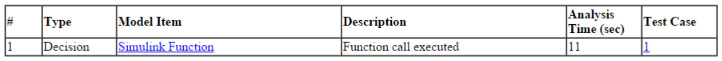
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	control logic.Oxygen_Sensor_Mode	State: Substate executed State "O2_fail"	97	35
2	Decision	control logic.Oxygen_Sensor_Mode	State: Substate executed State "O2_normal"	94	31
3	Decision	control logic.Oxygen_Sensor_Mode	State: Substate executed State "O2_warmup"	72	1
4	Decision	control logic.Pressure_Sensor_Mode	State: Substate executed State "press_fail"	79	9
5	Decision	control logic.Pressure_Sensor_Mode	State: Substate executed State "press_norm"	72	1

Objectives Satisfied - Needs Simulation

The **Objectives Satisfied - Needs Simulation** section lists the test objectives that the analysis satisfied. To confirm the satisfied status, you must run additional simulations of test cases.

In releases before R2017b, this section can include objectives that were marked as **Satisfied**.

This image shows the **Objectives Satisfied - Needs Simulation** section of the generated analysis report.



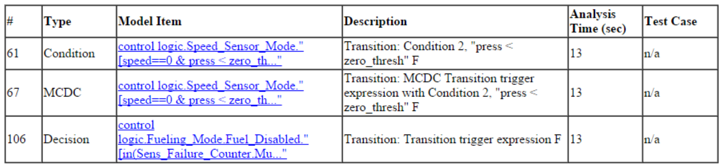
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Simulink Function	Function call executed	11	1

Objectives Unsatisfiable

The **Objectives Unsatisfiable** section lists the test objectives that the analysis determined could not be satisfied.

In releases before R2017b, this section can include objectives that were marked as **Proven Unsatisfiable**.

This image shows the **Objectives Unsatisfiable** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` example model.



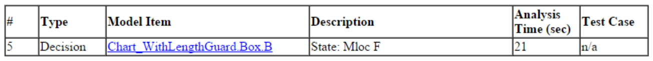
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
61	Condition	control logic.Speed_Sensor_Mode." [speed==0 & press < zero_th..."	Transition: Condition 2, "press < zero_thresh" F	13	n/a
67	MCDC	control logic.Speed_Sensor_Mode." [speed==0 & press < zero_th..."	Transition: MCDC Transition trigger expression with Condition 2, "press < zero_thresh" F	13	n/a
106	Decision	control logic.Fueling_Mode.Fuel_Disabled." [in(Sens_Failure_Counter.Mu..."	Transition: Transition trigger expression F	13	n/a

Objectives Unsatisfiable under Approximation

The **Objectives Unsatisfiable under Approximation** section lists the test objectives that the analysis determined could not be satisfied due to approximation during analysis.

In releases before R2017b, this section can include objectives that were marked as **Proven Unsatisfiable**.

This image shows the **Objectives Unsatisfiable under Approximation** section of the generated analysis report.



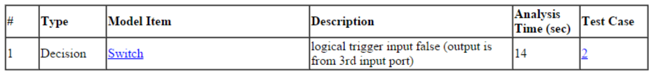
#	Type	Model Item	Description	Analysis Time (sec)	Test Case
5	Decision	Chart_WithLengthGuard.Box.B	State: Mloc F	21	n/a

Objectives Undecided with Testcases

The **Objectives Undecided with Testcases** section lists the test objectives that are undecided due to the impact of approximation during analysis.

In releases before R2017b, this section can include objectives that were marked as **Satisfied**.

This image shows the **Objectives Undecided with Testcases** section of the generated analysis report for the `sldvApproximationsExample` example model.



#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Decision	Switch	logical trigger input false (output is from 3rd input port)	14	2

Proof Objectives Status

If you run a property-proving analysis, the **Proof Objectives Status** section can include:

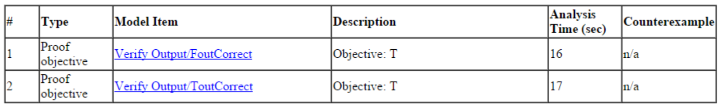
- "Objectives Valid" on page 13-38
- "Objectives Valid under Approximation" on page 13-38
- "Objectives Falsified with Counterexamples" on page 13-38
- "Objectives Falsified - Needs Simulation" on page 13-39
- "Objectives Undecided with Counterexamples" on page 13-39

Objectives Valid

The **Objectives Valid** section lists the proof objectives that the analysis found valid.

In releases before R2017b, this section can include objectives that were marked as **Proven Valid**.

This image shows the **Objectives Valid** section of the generated analysis report for the `sldvdemo_debounce_validprop` example model.



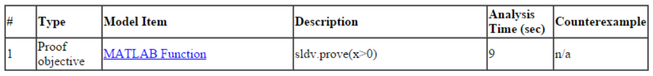
#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Verify Output/FoutCorrect	Objective: T	16	n/a
2	Proof objective	Verify Output/ToutCorrect	Objective: T	17	n/a

Objectives Valid under Approximation

The **Objectives Valid under Approximation** section lists the proof objectives that the analysis found valid under the impact of approximation.

In releases before R2017b, this section can include objectives that were marked as **Objectives Proven Valid**.

This image shows the **Objectives Valid under Approximation** section of the generated analysis report.

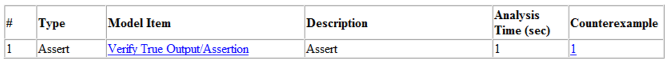


#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	MATLAB Function	sldv.prove(x>0)	9	n/a

Objectives Falsified with Counterexamples

The **Objectives Falsified with Counterexamples** section lists the proof objectives that the analysis disproved. The generated counterexample shows the violation of the proof objective.

This image shows the **Objectives Falsified with Counterexamples** section of the generated analysis report for the `sldvdemo_debounce_falseprop` example model.



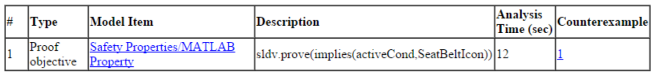
#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Assert	Verify True Output/Assertion	Assert	1	1

Objectives Falsified - Needs Simulation

The **Objectives Falsified - Needs Simulation** section lists the proof objectives that the analysis disproved. To confirm the falsified status, you must run additional simulations of counterexamples.

In releases before R2017b, this section can include objectives that were marked as **Objectives Falsified with Counterexamples**.

This image shows the **Objectives Falsified - Needs Simulation** section of the generated analysis report.



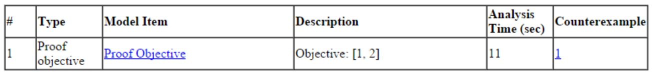
#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Safety Properties/MATLAB Property	sldv.prove(implies(activeCond,SeatBeltIcon))	12	1

Objectives Undecided with Counterexamples

The **Objectives Undecided with Counterexamples** section lists the proof objectives undecided due to the impact of approximation during analysis.

In releases before R2017b, this section can include objectives that were marked as **Falsified**.

This image shows the **Objectives Undecided with Counterexamples** section of the generated analysis report.



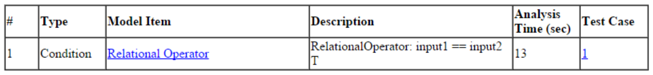
#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Proof Objective	Objective: [1, 2]	11	1

Objectives Undecided due to Runtime Error

For proof objectives and test objectives, the **Objectives Undecided due to Runtime Error** section lists the undecided objectives during analysis due to a run-time error. The run-time error occurred during simulation of a test case or counterexample.

In releases before R2017b, this section can include objectives that were marked as **Falsified** or **Satisfied**.

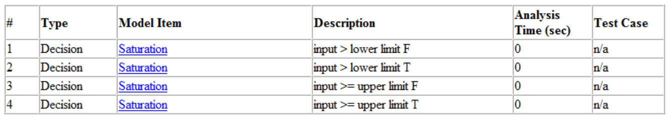
This image shows the **Objectives Undecided due to Runtime Error** section of the generated analysis report.



#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Condition	Relational Operator	RelationalOperator: input1 == input2 T	13	1

Objectives Undecided Due to Division by Zero

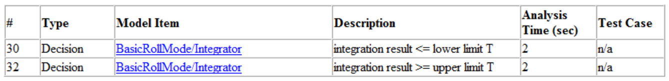
For all types of objectives, the **Objectives Undecided Due to Division by Zero** section lists the undecided objectives during analysis due to division-by-zero errors in the associated model items. To detect division-by-zero errors before running further analysis on your model, follow the procedure in "Detect Integer Overflow and Division-by-Zero Errors" on page 6-25.



#	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

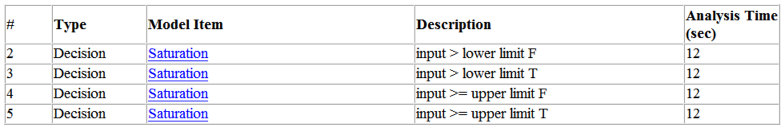
For all types of objectives, the **Objectives Undecided Due to Nonlinearities** section lists the undecided objectives during analysis due to required computation of nonlinear arithmetic. Simulink Design Verifier does not support nonlinear arithmetic or nonlinear logic.



#	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

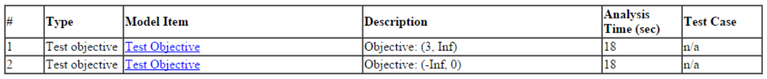
For all types of objectives, the **Objectives Undecided Due to Stubbing** section lists model items with undecided objectives during analysis due to stubbing. In releases before R2013b, these objectives can include objectives that were marked as **Objectives Satisfied – No Test Case** or **Objectives Falsified – No Counterexample**.



#	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

Objectives Undecided Due to Array Out of Bounds

For all types of objectives, the **Objectives Undecided Due to Array Out of Bounds** section lists the undecided objectives during analysis due to array out of bounds errors in the associated model items. To detect out of bounds array errors in your model, see “Detect Out of Bound Array Access Errors” on page 6-34.

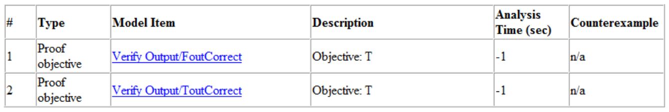


#	Type	Model Item	Description	Analysis Time (sec)	Test Case
1	Test objective	Test Objective	Objective: (3, Inf)	18	n/a
2	Test objective	Test Objective	Objective: (-Inf, 0)	18	n/a

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 this 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.

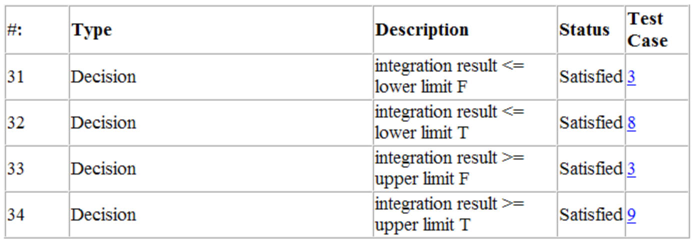


#	Type	Model Item	Description	Analysis Time (sec)	Counterexample
1	Proof objective	Verify Output/FoutCorrect	Objective: T	-1	n/a
2	Proof objective	Verify Output/ToutCorrect	Objective: T	-1	n/a

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 looks similar to this one for the Discrete-Time Integrator in the PI Controller subsystem of the `sldvdemo_cruise_control` example model.



#:	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 perform design error detection analysis and the analysis detects design errors in the model, the report includes a **Design Errors** chapter. This chapter summarizes the design errors that the analysis falsified:

- “Table of Contents” on page 13-41
- “Summary” on page 13-41
- “Test Case” on page 13-42

Table of Contents

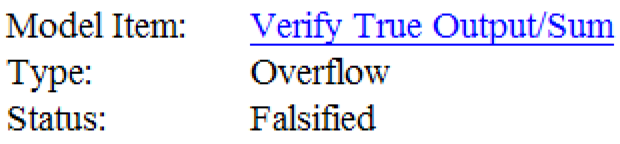
Each Design Errors chapter contains a table of contents. Each item in the table of contents is a hyperlink to results about a specific design error.

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)

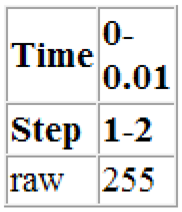
In the following example, the software analyzed the `sldvdemo_debounce_falseprop` model to detect design errors. The analysis detected an overflow error in the Sum block in the Verification Subsystem named Verify True Output.



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.



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 13-42
- "Summary" on page 13-42
- "Objectives" on page 13-42
- "Generated Input Data" on page 13-43
- "Expected Output" on page 13-43
- "Combined Objectives" on page 13-43
- "Long Test Cases" on page 13-44

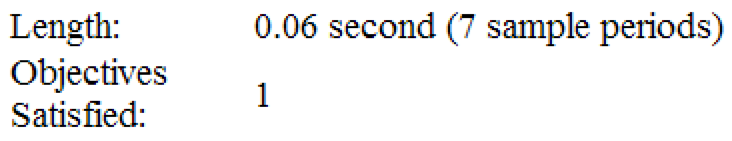
Table of Contents

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



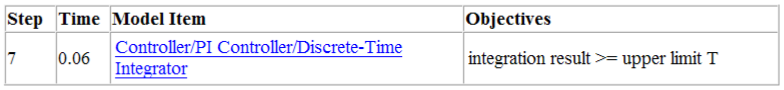
Length:	0.06 second (7 sample periods)
Objectives Satisfied:	1

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.

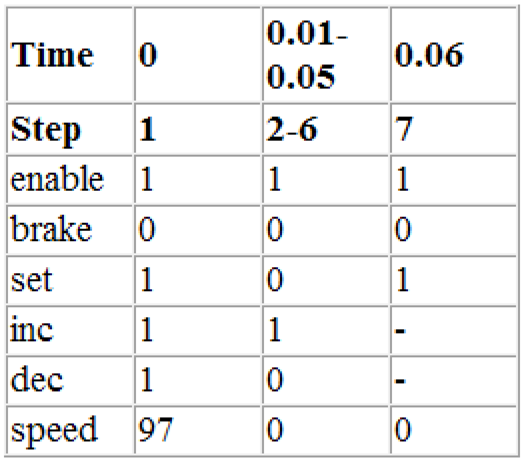
- The objective that was achieved.



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.

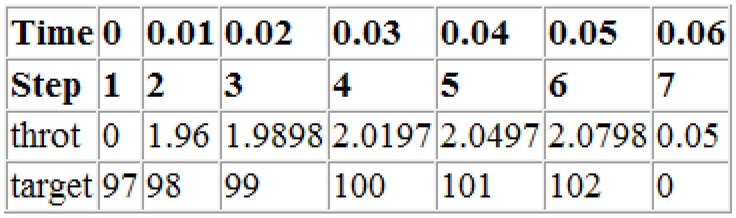


Time	**0**	**0.01-0.05**	**0.06**
Step	**1**	**2-6**	**7**
enable	1	1	1
brake	0	0	0
set	1	0	1
inc	1	1	-
dec	1	0	-
speed	97	0	0

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 15-56).

Expected Output

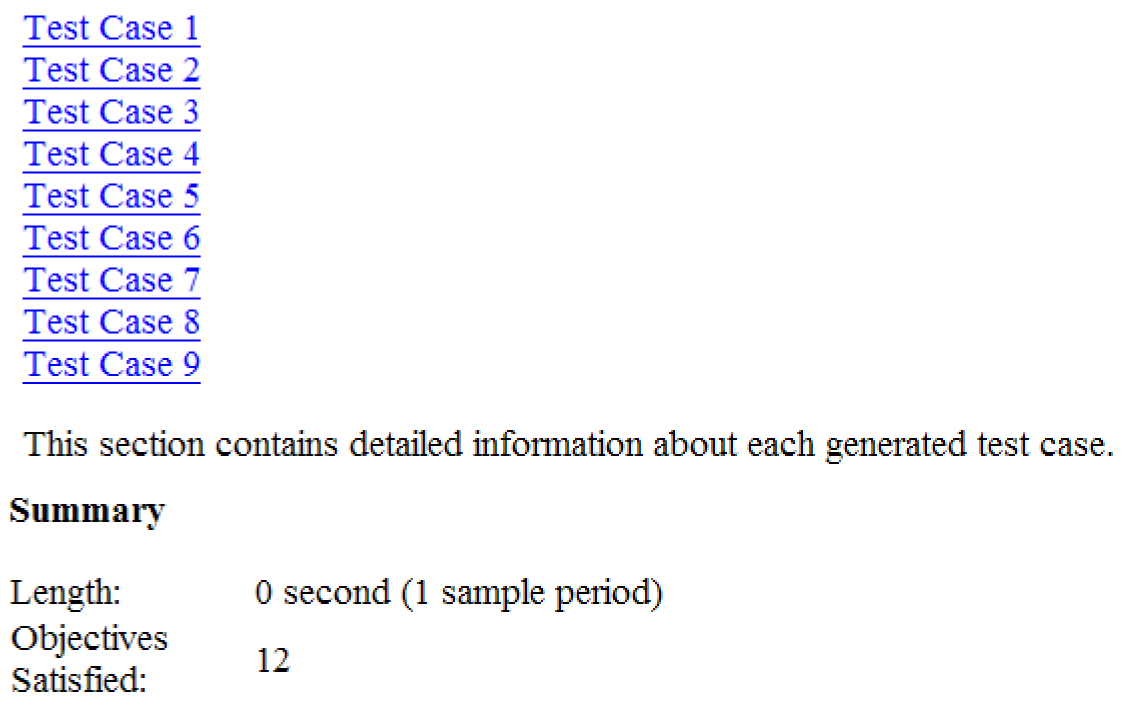
If you select the **Include expected output values** on the **Design Verifier > Results** pane of the Configuration Parameters dialog box, the report includes the Expected Output section for each test case. For each output signal associated with the model item, this table lists the expected output value at each time step.



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.



Test Case 1
Test Case 2
Test Case 3
Test Case 4
Test Case 5
Test Case 6
Test Case 7
Test Case 8
Test Case 9

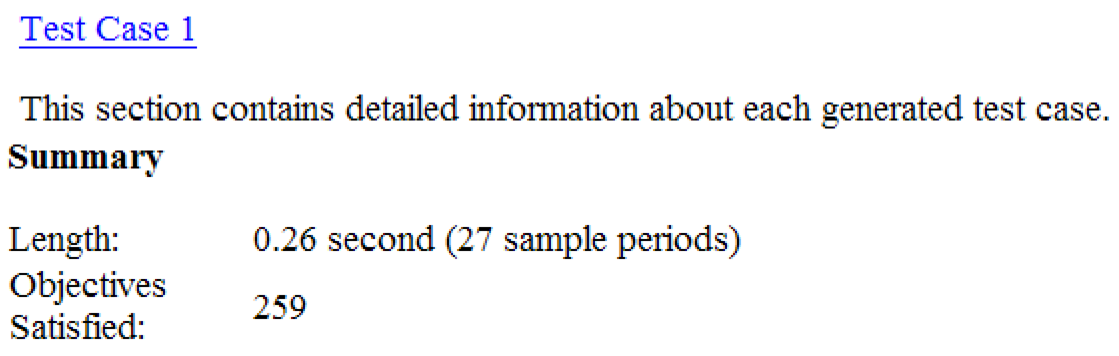
This section contains detailed information about each generated test case.

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.



Test Case 1

This section contains detailed information about each generated test case.

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 13-45
- “Summary” on page 13-45
- “Counterexample” on page 13-45

Table of Contents

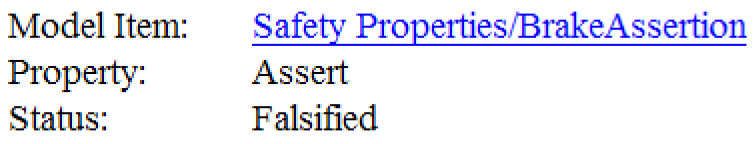
Each Properties chapter contains a table of contents. Each item in the table of contents is a hyperlink to information about a specific property that was falsified.

Summary

The Summary section lists:

- The model item that the software analyzed
- The type of property that was evaluated
- The status of the analysis

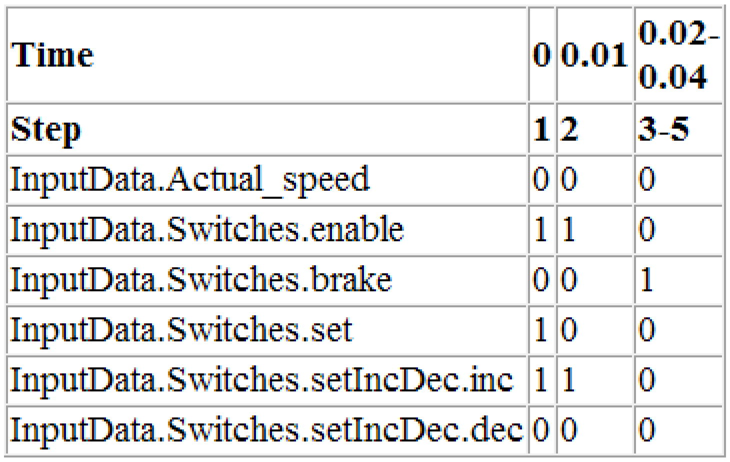
In the following example, the software analyzed the `sldvdemo_cruise_control_verification` model for property proving. The analysis proved that the input to the Assertion block named BrakeAssertion was nonzero.



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.

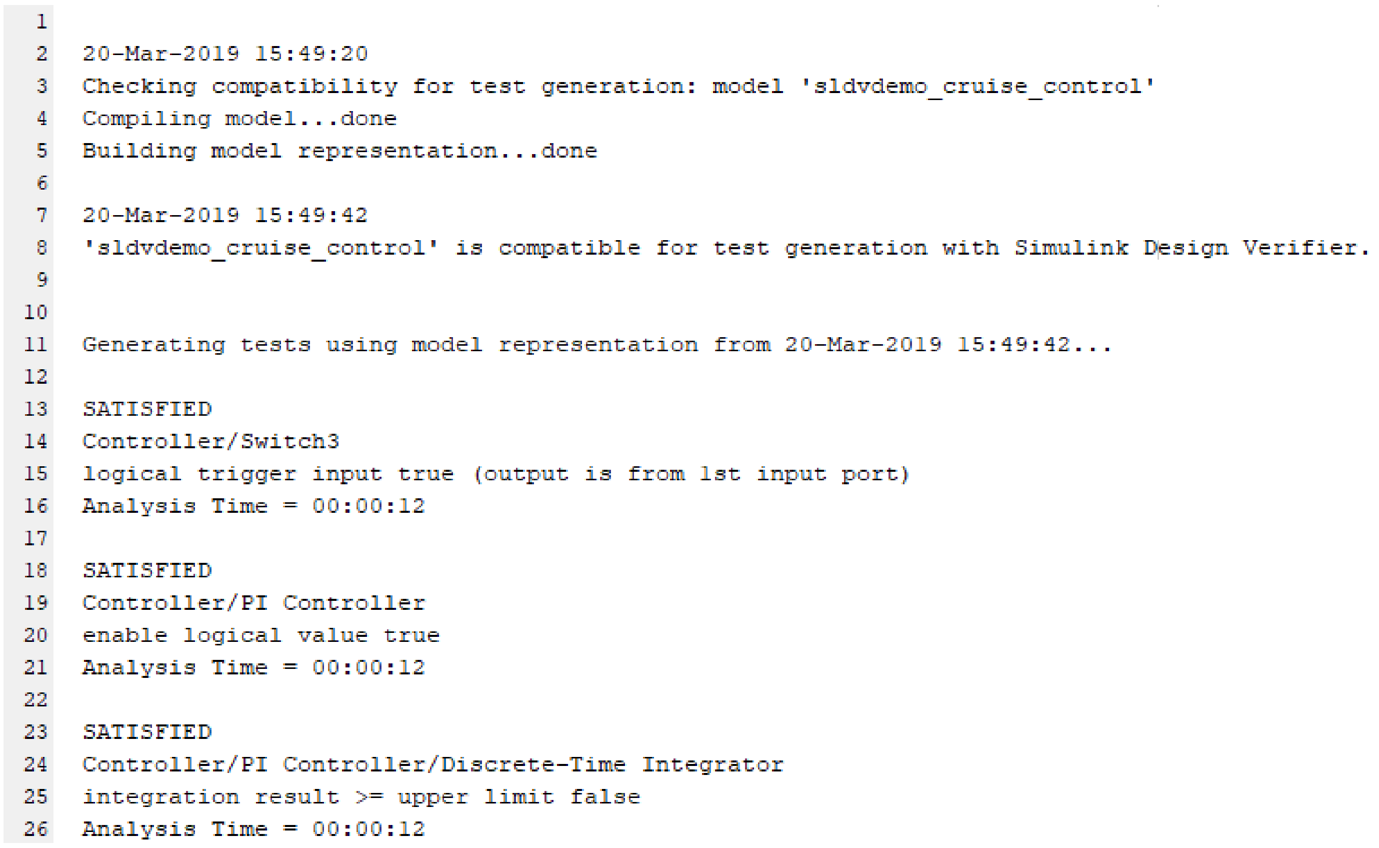


Time	**0**	**0.01**	**0.02-0.04**
Step	**1**	**2**	**3-5**
InputData.Actual_speed	0	0	0
InputData.Switches.enable	1	1	0
InputData.Switches.brake	0	0	1
InputData.Switches.set	1	0	0
InputData.Switches.setIncDec.inc	1	1	0
InputData.Switches.setIncDec.dec	0	0	0

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.



```

20-Mar-2019 15:49:20
Checking compatibility for test generation: model 'sldvdemo_cruise_control'
Compiling model...done
Building model representation...done

20-Mar-2019 15:49:42
'sldvdemo_cruise_control' is compatible for test generation with Simulink Design Verifier.


Generating tests using model representation from 20-Mar-2019 15:49:42...

SATISFIED
Controller/Switch3
logical trigger input true (output is from 1st input port)
Analysis Time = 00:00:12

SATISFIED
Controller/PI Controller
enable logical value true
Analysis Time = 00:00:12

SATISFIED
Controller/PI Controller/Discrete-Time Integrator
integration result >= upper limit false
Analysis Time = 00:00:12
```

Review Analysis Results

In this section...
“View Active Results” on page 13-47
“Load Previous Results” on page 13-47
“Explore Results” on page 13-47

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 13-47.

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.

On the **Design Verifier** tab, click **Results Summary** to view the Results Summary window. The Results Summary window reopens with the latest analysis results for your model.

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. On the **Design Verifier** tab, click **Load Earlier Results** and browse to the data file that corresponds to the analysis you want to review. Click **Results Summary**.

For more information on analysis data files, see “Simulink Design Verifier Data Files” on page 13-7.

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 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 13-28

Task	For more information
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 13-13
View the data file.	“Simulink Design Verifier Data Files” on page 13-7
View the log file.	“Simulink Design Verifier Log Files” on page 13-46

See Also

More About

- “Design Verifier Pane: Results” on page 15-54
- “Simulink Design Verifier Data Files” on page 13-7
- “Simulink Design Verifier Reports” on page 13-28

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 14-22 describes specific techniques for proving properties in a large model.

Analyze a Large Model

In this section...

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.

To access the Test Generation Advisor, on the **Design Verifier** tab, in the **Mode** section, click **Test Generation**. In the **Prepare** section, click **Advisor**. For more information see “Use Test Generation Advisor to Identify Analyzable Components” on page 7-17.

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 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 13-28.
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 5-9.

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 14-8
Nonlinear operations	• "Bottom-Up Approach to Model Analysis" on page 14-13 • "Logical Operations" on page 14-19
Large state spaces	"Models with Large Verification State Space" on page 14-20
Large timers and time delays	"Counters and Timers" on page 14-21

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” (MATLAB Report Generator).

Manage Model Data to Simplify the Analysis

In this section...

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.

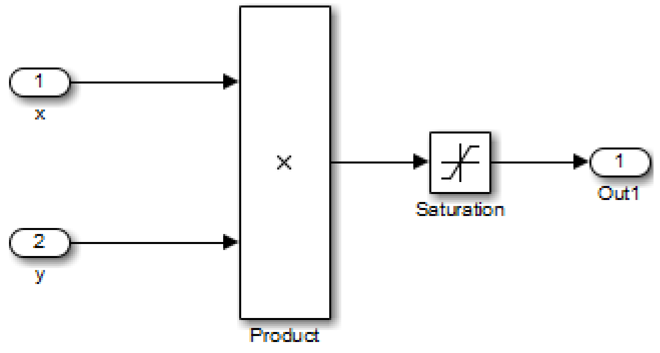
In another example, suppose you have a Sum block with two inputs that are always integers between –10 and 10. Set the **Output data type** parameter to `int8`, rather than `int32` or `double`.

To display the signal data types, on the **Debug** tab, click **Information Overlays** > **Port Data Type**.

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.



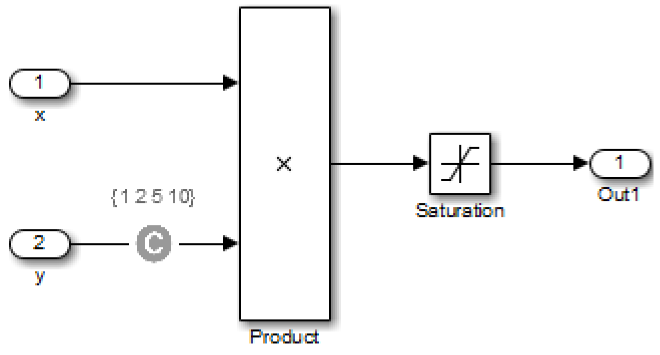
The Simulink Design Verifier software generates errors when attempting to satisfy the upper and lower limits of the Saturation block, because the software does not support nonlinear arithmetic. To work around these errors, restrict one of the inputs to a set of discrete values.

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.



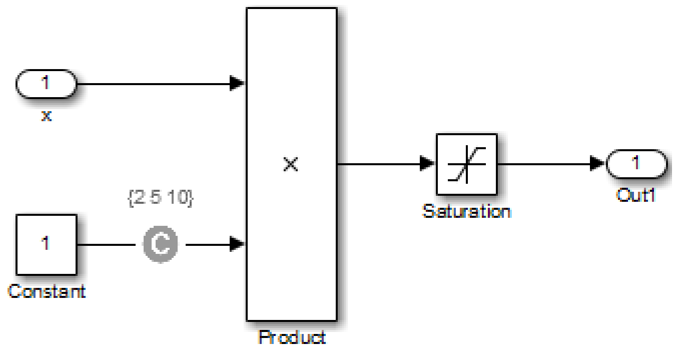
You can also constrain signals that are intermediate or output values of the model. Constraining such signals makes it easier to work around multiplication or division inside lower level subsystems that do not depend on model inputs.

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 14-11.

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 14-8, 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.

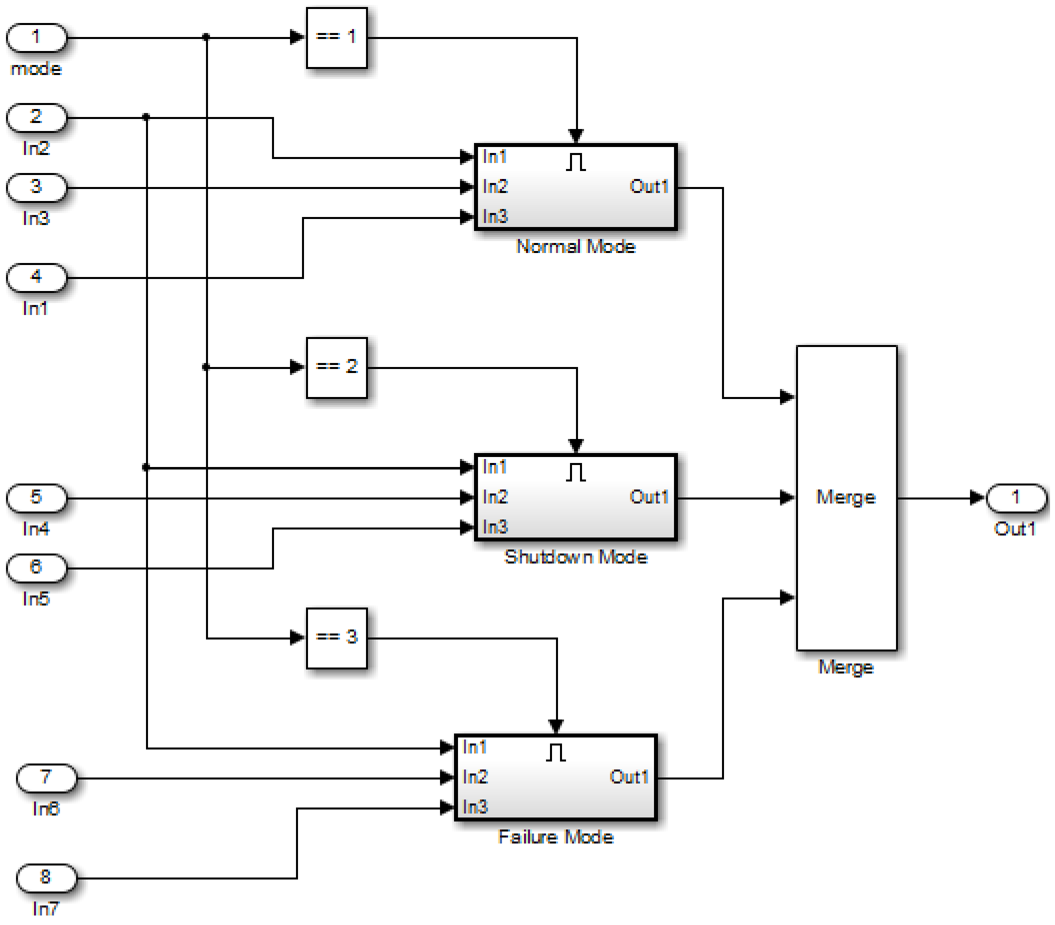
   Note To view an example of extending an existing test suite to achieve missing model coverage, enter the following at the command prompt in the MATLAB Command Window:

   ```
   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 7-17.
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

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 14-13.

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 7-15.

For more information about analyzing atomic subcharts, see "Analyze a Stateflow Atomic Subchart" on page 1-19.

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 10-3.

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.

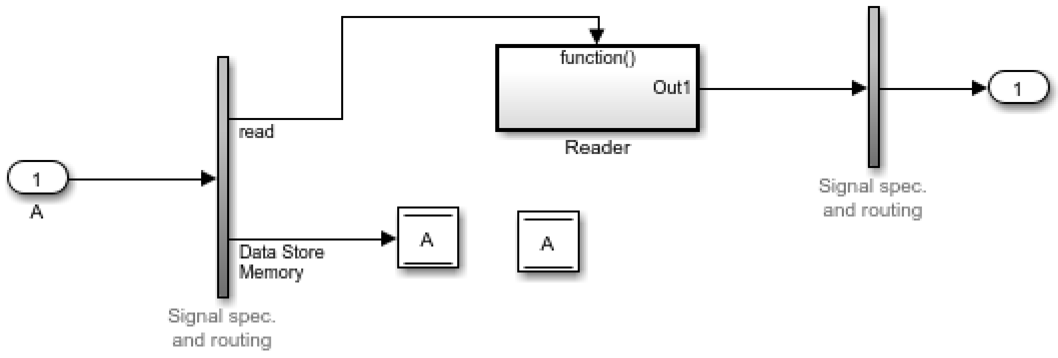
The following example analyzes a subsystem in the `sl_subsys_fcncall8` example model:

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 “Implement Function-Call Subsystems with S-Functions” (Simulink).

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:

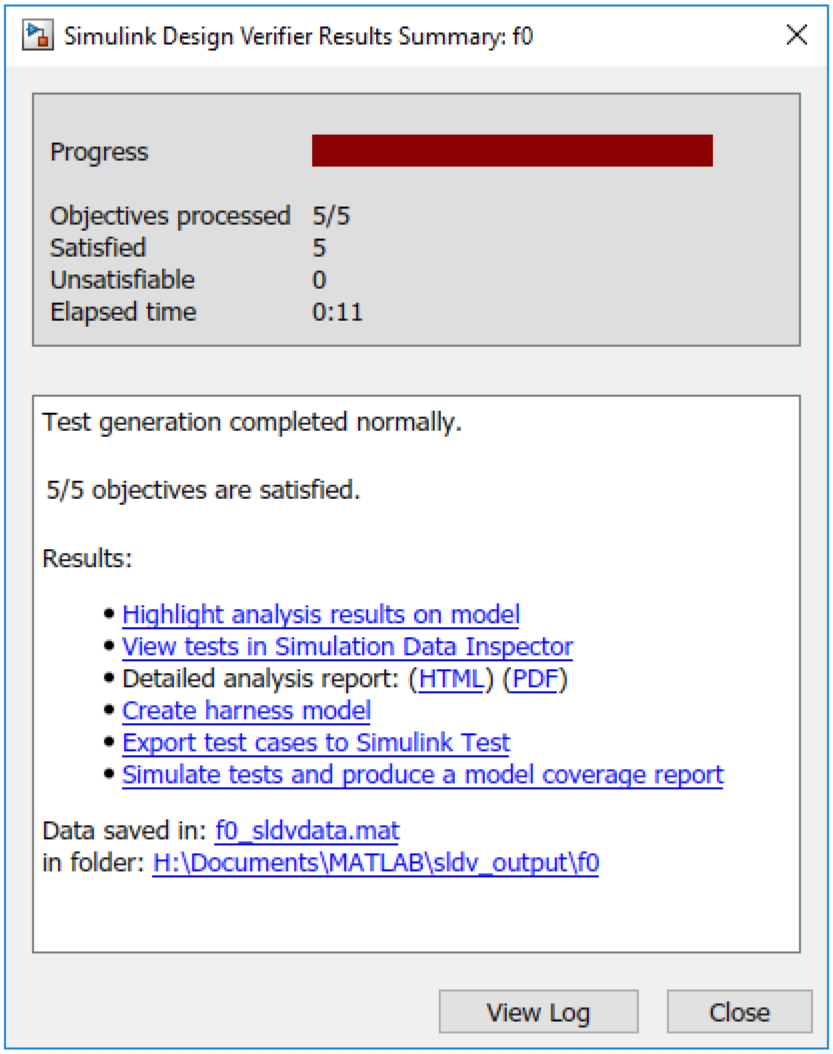
1 Open the `sl_subsys_fcncall2` example model:

```
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 `f0`, analyzes the extracted model, and produces results.


Simulink Design Verifier Results Summary: f0

Progress	
Objectives processed	5/5
Satisfied	5
Unsatisfiable	0
Elapsed time	0:11

Test generation completed normally.

5/5 objectives are satisfied.

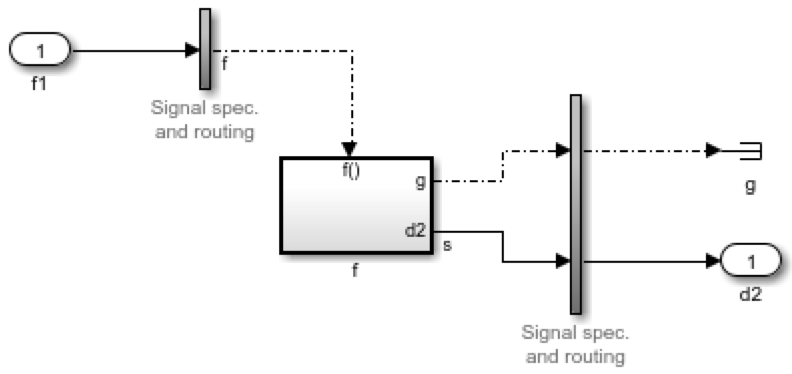
Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML) (PDF)
- Create harness model
- Export test cases to Simulink Test
- Simulate tests and produce a model coverage report

Data saved in: f0_sldvdata.mat
in folder: H:\Documents\MATLAB\sldv_output\f0

3 Open the `f0` model that the software created in `<`*`current_folder`*`>\sldv_output\f0`.

The Inport block and the new subsystem block mimic the trigger for the function-call subsystem f in the new `f0` model.


1
f1
f
Signal spec.
and routing
f()
g
d2
s
f
Signal spec.
and routing
g
1
d2

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 non-linearity in certain situations.

Models with Large Verification State Space

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 14-21.) 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 Extend an Existing Test Suite.

Counters and Timers

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 over complicating 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.
- 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

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 14-22
“Combine Proving Properties and Finding Proof Violations” on page 14-22

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.

During analysis, 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.

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, open the **Design Verifier** tab. In the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**. The **Design Verifier** pane of the model configuration parameters opens.

By default, options for Simulink Design Verifier do not appear in the Configuration Parameters dialog box. When you open the **Design Verifier** tab, Simulink Design Verifier associates its default options with the model. After you save the model, you can access options for Simulink Design Verifier directly from the Configuration Parameters dialog box.

See 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'}`

Parameter	Location	Values
`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>'}`
`DVCodeAnalysisExtraOptions`	Set by the **Additional options for code analysis** parameter on the **Design Verifier** pane.	character array `{''}`
`DVCoverageDataFile`	Set by the **Coverage data file** parameter on the **Design Verifier** > **Test Generation** pane.	character array `{''}`
`DVCovFilter`	Set by the **Ignore objectives based on filter** parameter on the **Design Verifier** pane.	`'on' | {'off'}`
`DVCovFilterFileName`	Set by the **Filter file** parameter on the **Design Verifier** 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 **Specified minimum and maximum value violations** parameter on the **Design Verifier** > **Design Error Detection** pane.	`'on' | {'off'}`
`DVDesignMinMaxConstraints`	Set by the **Use specified input minimum and maximum values** parameter on the **Design Verifier** pane.	`{'on'} | 'off'`
`DVDetectActiveLogic`	Set by **Identify active logic** on the **Design Verifier** > **Design Error Detection** pane.	`'on' | {'off'}`

Parameter	Location	Values
`DVDetectBlockInputRange-Violations`	Set by **Specified block input range violations** on the **Design Verifier** > **Design Error Detection** 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'`
`DVDetectDSM-AccessViolations`	Set by the **Data store access violations** parameter on the **Design Verifier** > **Design Error Detection** pane.	`'on' | {'off'}`
`DVDetectInfNaN`	Set by the **Non-finite and NaN floating-point values** 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'`
`DVDetectOutOfBounds`	Set by the **Out of bound array access** parameter on the **Design Verifier** > **Design Error Detection** pane.	`{'on'} | 'off'`
`DVDetectSubnormal`	Set by the **Subnormal floating-point values** 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'`
`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'}`

Parameter	Location	Values
`DVHarnessSource`	Set by the **Harness source** parameter on the **Design Verifier** > **Results** pane.	`{'Signal Builder'} | 'Signal Editor'`
`DVIgnoreCovSatisfied`	Set by the **Ignore objectives satisfied in existing coverage data** parameter on the **Design Verifier** > **Test Generation** pane.	`'on' | {'off'}`
`DVIgnoreExistTestSatisfied`	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'`
`DVModelCoverageObjectives`	Set by the **Model coverage objectives** parameter on the **Design Verifier** > **Test Generation** pane.	`'None' | 'Decision' | {'ConditionDecision'} | 'MCDC' | 'EnhancedMCDC'`
`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/$ModelName$'}`

Parameter	Location	Values
`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'}`
`DVParametersConfigFileName`	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'}`
`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'}`
`DVRebuildModel-Representation`	Set by the **Rebuild model representation** parameter on the **Design Verifier** pane.	`'Always' | {'If change is detected'}`
`DVReduceRationalApprox`	Set by the **Run additional analysis to reduce instances of rational approximation** parameter on the **Design Verifier** pane.	`{'on'} | 'off'`

Parameter	Location	Values
`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'}`
`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'}`
`DVStrictEnhancedMCDC`	Set by the **Use strict propagation conditions** parameter on the **Design Verifier** > **Test Generation** pane.	`'on' | {'off'}`
`DVTestConditions`	Set by the **Test conditions** parameter on the **Design Verifier** > **Test Generation** pane.	`'EnableAll' | 'DisableAll' | {'UseLocalSettings'}`

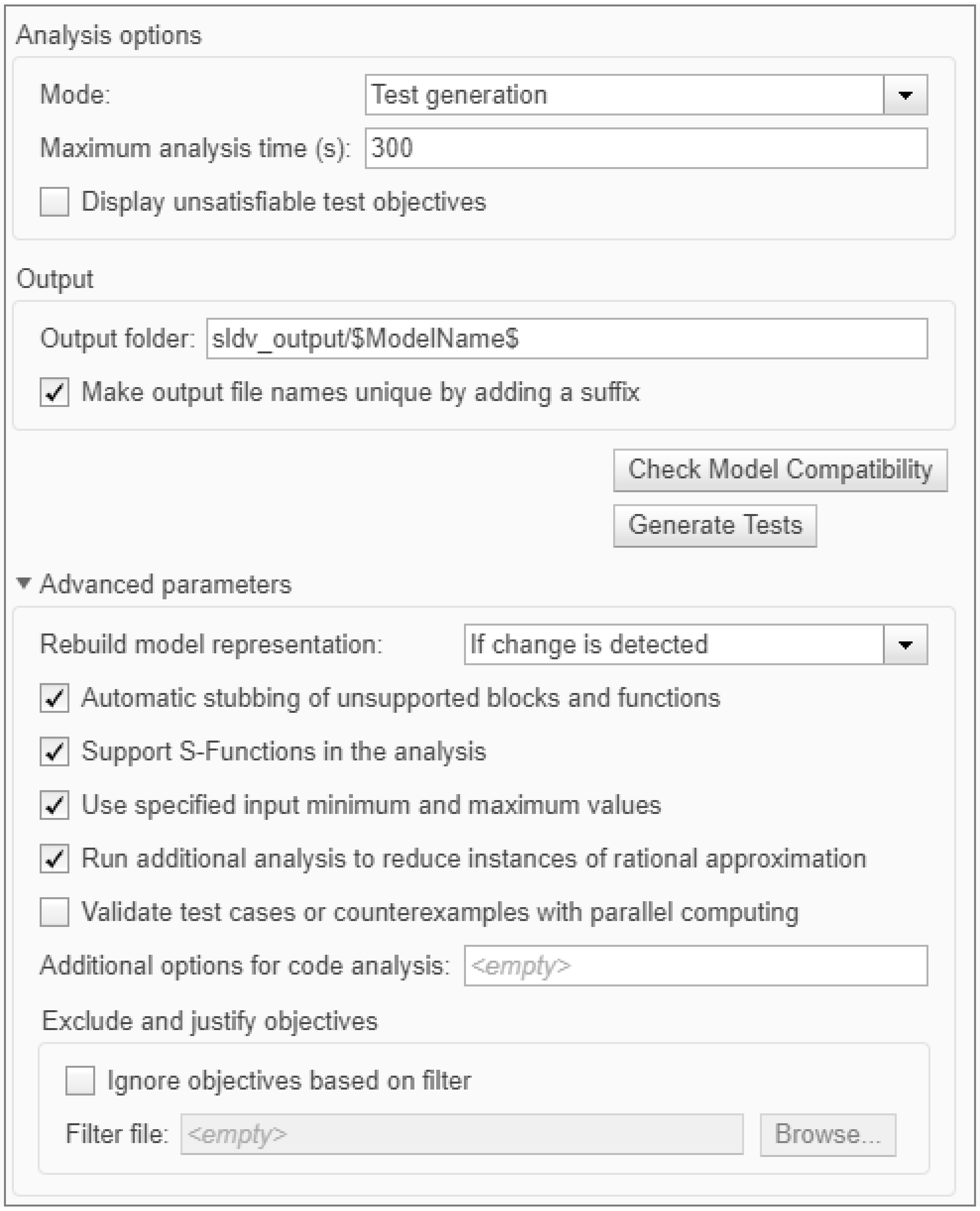
Parameter	Location	Values
`DVTestgenTarget`	Set by the **Test generation target** parameter on the **Design Verifier** > **Test Generation** pane.	`{'Model'} | 'GenCodeTopModel' | 'GenCodeModelRef'`
`DVTestObjectives`	Set by the **Test objectives** parameter on the **Design Verifier** > **Test Generation** pane.	`'EnableAll' | 'DisableAll' | {'UseLocalSettings'}`
`DVTestSuiteOptimization`	Set by the **Test suite optimization** parameter on the **Design Verifier** > **Test Generation** pane. If you analyze your model by using the `Legacy LargeModel (Nonlinear Extended)`, the software displays a warning message that this option has been removed and suggests that you use `Auto` instead.	`{'Auto'} | 'IndividualObjectives' | 'LongTestcases' | 'LargeModel (Nonlinear Extended)'`
`DVUseParallel`	Set by the **Validate test cases or counterexamples with parallel computing** parameter on the **Design Verifier** pane.	`'on' | {'off'}`

See Also

More About

- "Design Verifier Pane" on page 15-9
- `sldvoptions`

Design Verifier Pane


Analysis options

Mode: Test generation

Maximum analysis time (s): 300

☐ Display unsatisfiable test objectives

Output

Output folder: sldv_output/$ModelName$

✓ Make output file names unique by adding a suffix

Advanced parameters

Rebuild model representation: If change is detected

✓ Automatic stubbing of unsupported blocks and functions

✓ Support S-Functions in the analysis

✓ Use specified input minimum and maximum values

✓ Run additional analysis to reduce instances of rational approximation

☐ Validate test cases or counterexamples with parallel computing

Additional options for code analysis: <empty>

Exclude and justify objectives

☐ Ignore objectives based on filter

Filter file: <empty>

In this section...

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

Simulink Design Verifier specifies the value of this option when you select one of these analysis options from the **Design Verifier** tab, in the **Mode** section:

- Select **Design Error Detection**, then click **Detect Design Errors**.
- Select **Test Generation**, then click **Generate Tests**.
- Select **Property Proving**, then click **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 1-21
- “What Is Design Error Detection?” on page 6-2
-
- “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'`

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 .
- **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.

Rebuild model representation

Specify whether to rebuild model representation for Simulink Design Verifier analysis.

Settings

Default: `If change is detected`

`Always`
: Always rebuild the model representation.

`If change is detected`
: Rebuild the model representation only when the software detects any change in the model.

Command-Line Information
Parameter: `DVRebuildModelRepresentation`
Type: `character array`
Value: `'Always'` | `'IfChangeIsDetected'`
Default: `'If change is detected'`

See Also

"Check Model Compatibility" on page 3-2

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 and Considerations for S-Functions and C/C++ Code" on page 3-27

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

Run additional analysis to reduce instances of rational approximation

Specify whether Simulink Design Verifier attempts to reduce the use of rational approximation during analysis.

Settings

Default: On

☑ On

When you use Simulink Design Verifier to analyze models, Simulink Design Verifier attempts to reduce the use of rational approximation if the model. Enabling this setting may increase analysis time.

☐ Off

Simulink Design Verifier does not attempt to reduce the use of rational approximation during analysis.

Command-Line Information
Parameter: `DVReduceRationalApprox`
Type: character array
Value: `'on'` | `'off'`
Default: `'on'`

Validate test cases or counterexamples with parallel computing

Specifies whether to validate test cases or counterexamples with parallel computing. This option requires a Parallel Computing Toolbox™ license.

When to Use Parallel Computing for Validation

In general, parallel execution can help reduce the validation time if:

- You have a complex Simulink model that takes a long time to simulate.
- The Simulink Design Verifier analysis exceeds the maximum analysis time and results in a number of objectives with the Needs Simulation status. For more information, see “Objectives Satisfied - Needs Simulation” on page 13-37 and “Objectives Falsified - Needs Simulation” on page 13-39.
- The test generation analysis generates long test cases. This may be because you have set **Test suite optimization** to `LongTestcases` or the **Maximum test case steps** value is greater than the default value. For more information, see “Test Generation Pane Overview” on page 15-32.

The following points must be considered when using parallel computing for validation:

- Starting a parallel pool can take time, which impacts the overall analysis time. To reduce the analysis time:
  - Make sure that the parallel pool is already running before you run a test generation analysis. By default, the parallel pool shuts down after being idle for a specified number of minutes. To change the setting, see Specify Your Parallel Preferences.
  - Load Simulink on all the parallel pool workers.
- The simulation occurs sequentially when:
  - The cluster is not `local`. Configure parallel preferences to use the `local` cluster only. See Specify Your Parallel Preferences.
  - The model is in `dirty state` prior to launching the SLDV analysis.

  - The model has `ToFile` blocks.
- Cross-product features such as **functional testing and coverage analysis** from Simulink Test Manager do not support parallel computing for validation. For more information, see “Perform Functional Testing and Analyze Test Coverage” (Simulink Test).

Settings

Default: Off

☑ On

If you have a Parallel Computing Toolbox license, then Simulink Design Verifier validates test cases or counterexamples in parallel across multiple workers on the same machine.

☐ Off

Simulink Design Verifier validates test cases or counterexamples in serial.

Command-Line Information
Parameter: `DVUseParallel`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

“Reporting Approximations Through Validation Results” on page 2-22

Additional options for code analysis

Specify additional options for analyzing S-functions that have been compiled to be compatible with Simulink Design Verifier. For more information, see “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-27.

Settings

Default: `''`

Enter additional options for analyzing S-Functions that have been compiled to be compatible with Simulink Design Verifier. For example, to specify the maximum size of arrays, enter `defaultArraySize = 512`.

Command-Line Information
Parameter: `DVCodeAnalysisExtraOptions`
Type: character array
Value: any valid option for analyzing S-Functions
Default: `''`

Ignore objectives based on filter

Specify to analyze the model, ignoring the objectives in the **Filter file**. The **Filter file** contains the model coverage objectives for test generation and design error detection objectives that you want to filter from analysis.

Settings

Default: Off

☑ On

Ignores objectives in the **Filter file** during test generation and design error detection analysis.

☐ Off

Generates results for all the objectives for test generation and design error detection analysis, including those in the **Filter file**.

Dependency

This parameter enables **Filter file**.

Command-Line Information

Parameter: `DVCovFilter`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

“Coverage Filtering” (Simulink Coverage)

Filter file

Specify a folder and file name for the file that contains the model coverage objectives for test generation and design error detection objectives that you want to filter from analysis.

Settings

Default: `''`

- Specify the name of the folder and file name that contains the objectives that you want to ignore from test generation and design error detection analysis.

Click the **Browse** button to select an existing **Filter file**.

Command-Line Information

Parameter: `DVCovFilterFileName`
Type: character array
Value: any valid path and file name
Default: `''`

See Also

“Coverage Filter Rules and Files” (Simulink Coverage)

Filter Objectives by Using Analysis Filter Viewer on page 6-47

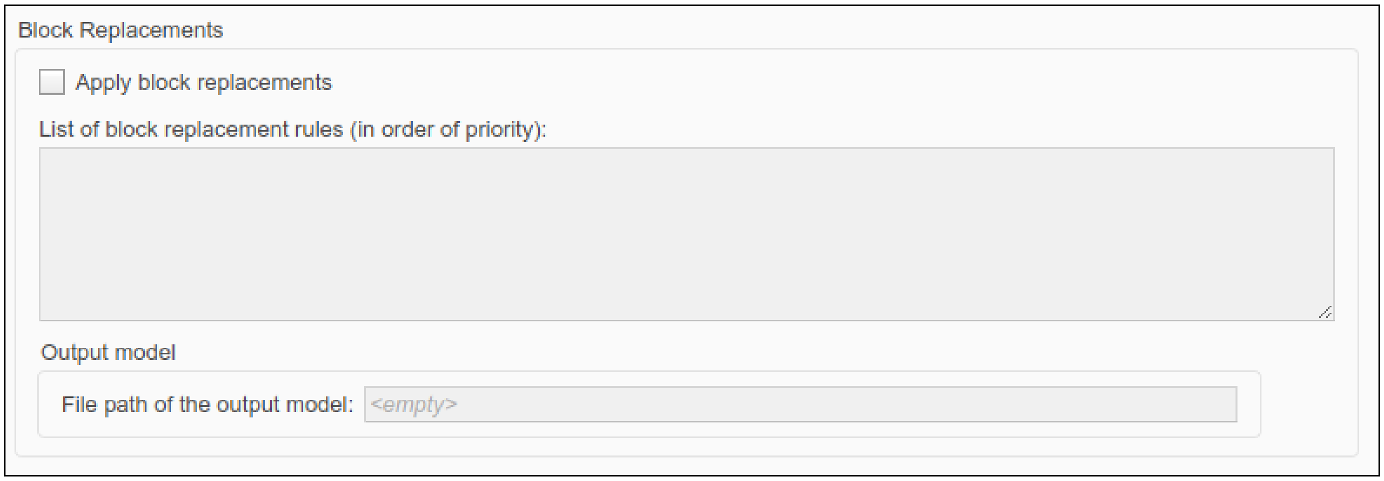
Browse...

Browse to the file that contains the objectives that you want to ignore from design error detection and test generation analysis.

Dependency

This button is enabled by **Ignore objectives based on filter**.

Design Verifier Pane: Block Replacements


Block Replacements

☐ Apply block replacements

List of block replacement rules (in order of priority):

Output model

File path of the output model: <empty>

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**.

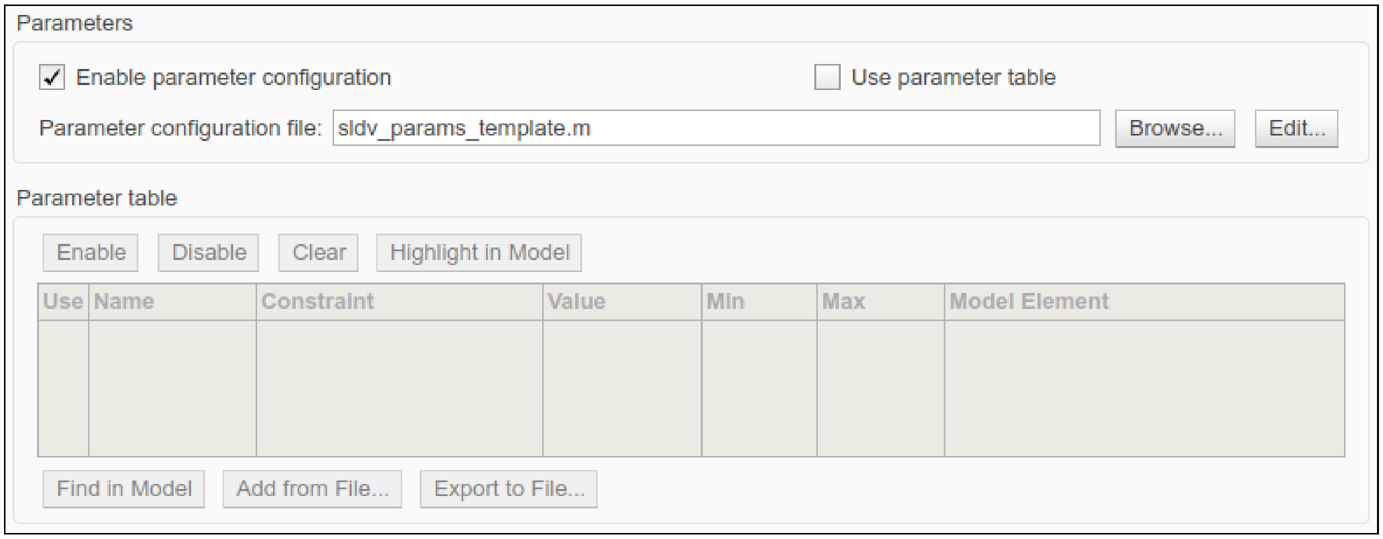
Command-Line Information

Parameter: `DVBlockReplacementModelFileName`
Type: character array
Value: any valid path and file name
Default: `'$ModelName$_replacement'`

See Also

"Block Replacement"

Design Verifier Pane: Parameters



In this section...

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.

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.

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.

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 software searches your model for parameters that you can configure and loads them in the **Parameter Table**. If your model uses a configuration reference, Simulink Design Verifier does not support the search for parameters when using the **Find in Model** button. For more information, see “Share a Configuration with Multiple Models” (Simulink).

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.

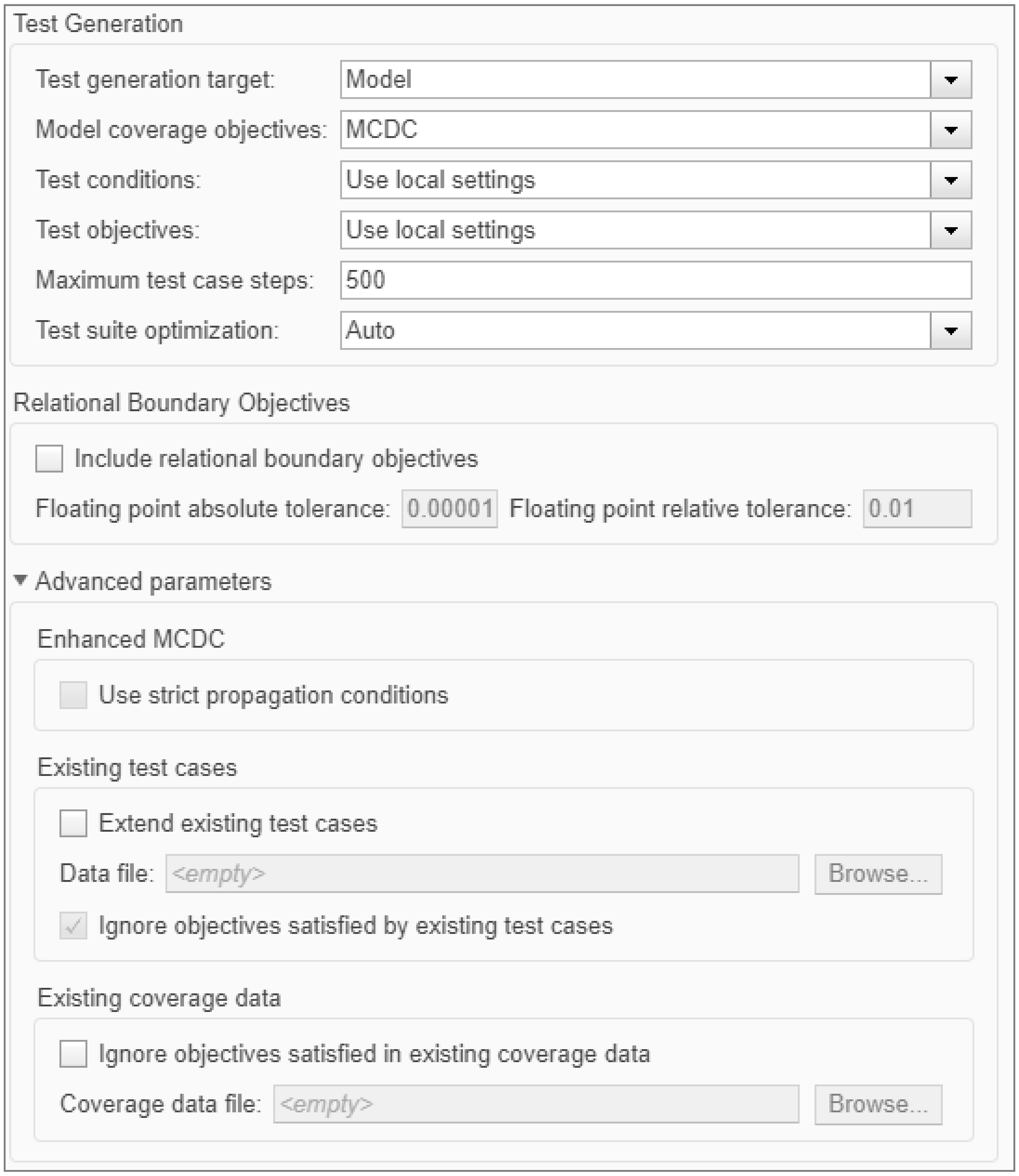
Export to File...

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



Test Generation

Test generation target: Model

Model coverage objectives: MCDC

Test conditions: Use local settings

Test objectives: Use local settings

Maximum test case steps: 500

Test suite optimization: Auto

Relational Boundary Objectives

☐ Include relational boundary objectives

Floating point absolute tolerance: 0.00001 Floating point relative tolerance: 0.01

Advanced parameters

Enhanced MCDC

☐ Use strict propagation conditions

Existing test cases

☐ Extend existing test cases

Data file: <empty> Browse...

✓ Ignore objectives satisfied by existing test cases

Existing coverage data

☐ Ignore objectives satisfied in existing coverage data

Coverage data file: <empty> Browse...

In this section...

In this section...

Test Generation Pane Overview

Specify options that control how Simulink Design Verifier generates tests for the models it analyzes.

See Also

“Workflow for Test Case Generation” on page 7-2

Test generation target

Specify the target for test generation.

- **Default:** `Model` generates test cases for the model.
- `Code Generated as Top Model` generates tests for code generated as top model.
- `Code Generated as Model Reference` generates tests for code generated as model reference.

Command-Line Information
Parameter: `DVTestgenTarget`
Type: character array
Value: `'Model'` | `'GenCodeTopModel'` | `'GenCodeModelRef'` |

See Also

“Code Coverage Test Generation”“Generate Test Cases for Embedded Coder Generated Code” on page 7-21

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 7-23.

`Condition Decision`

Generates test cases that achieve condition and decision coverage. For more information, see “Condition” on page 7-23.

`MCDC`

Generates test cases that achieve modified condition decision coverage (MCDC). When you select `MCDC`, Simulink Design Verifier automatically enables every coverage objective for decision and condition coverage. For more information, see “MCDC” on page 7-23.

`Enhanced MCDC`

Generates test cases that achieve enhanced MCDC coverage. When you select `Enhanced MCDC`, Simulink Design Verifier automatically enables MCDC coverage. For more information, see “Enhanced MCDC” on page 7-24.

Command-Line Information
Parameter: `DVModelCoverageObjectives`
Type: character array
Value: `'None'` | `'Decision'` | `'ConditionDecision'` | `'MCDC'`| `'EnhancedMCDC'`
Default: `'ConditionDecision'`

See Also

“Workflow for Test Case Generation” on page 7-2

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 7-2

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 7-2

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 7-2

Test suite optimization

Specify the optimization strategy to use when generating test cases.

Settings

Default: `Auto`

`Auto`

Analyzes the model by using a strategy that automatically adapts to the model for better analysis performance and precision.

`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, that is, 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.

`Legacy LargeModel (Nonlinear Extended)`

Analyzes the model by using a static strategy that does not adapt to the model. When you analyze a model by using `Legacy LargeModel (Nonlinear Extended)`, Simulink Design Verifier displays a warning message that this option is deprecated and suggests that you use `Auto`. `Auto` is most likely to produce better analysis results than `Legacy LargeModel (Nonlinear Extended)`.

Command-Line Information
Parameter: `DVTestSuiteOptimization`
Type: character array
Value: `'Auto'` | `'IndividualObjectives'` | `'LongTestcases'` | `Legacy LargeModel (Nonlinear Extended)`
Default: `'Auto'`

See Also

"Workflow for Test Case Generation" on page 7-2

Simulink Design Verifier Options on page 15-2

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 15-37
- "Floating point relative tolerance" on page 15-37

Command-Line Information
Parameter: `DVIncludeRelationalBoundary`
Type: character array
Value: `'on'`|`'off'`
Default: `'off'`

See Also

- “Relational Boundary” on page 7-24
- “Model Objects That Receive Coverage” (Simulink Coverage)

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 15-36.

Command-Line Information
Parameter: `DVAbsoluteTolerance`
Type: `double`
Value: Any valid value
Default: 1.0000e-05

See Also

- “Relational Boundary” on page 7-24
- “Model Objects That Receive Coverage” (Simulink Coverage)

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 15-36.

Command-Line Information
Parameter: `DVRelativeTolerance`
Type: `double`
Value: Any valid value
Default: 0.01

See Also

- "Relational Boundary" on page 7-24
- "Model Objects That Receive Coverage" (Simulink Coverage)

Use strict propagation conditions

Specify whether to use strict propagation conditions for enhanced MCDC analysis.

Settings

Default: Off

☑ On

Use strict propagation condition for enhanced MCDC analysis.

☐ Off

Does not use strict propagation conditions for enhanced MCDC analysis.

Dependency

This parameter is enabled when you select `Enhanced MCDC` as **Model coverage objectives**.

Command-Line Information

Parameter: `DVStrictEnhancedMCDC`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

- “Enhanced MCDC” on page 7-24

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.

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 8-2

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.

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 13-7

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 8-10
- "Simulink Design Verifier Data Files" on page 13-7

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 9-11
- “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 9-11
- “Test Case Extension”

Browse...

Browse to the file that contains data about satisfied coverage objectives.

Dependency

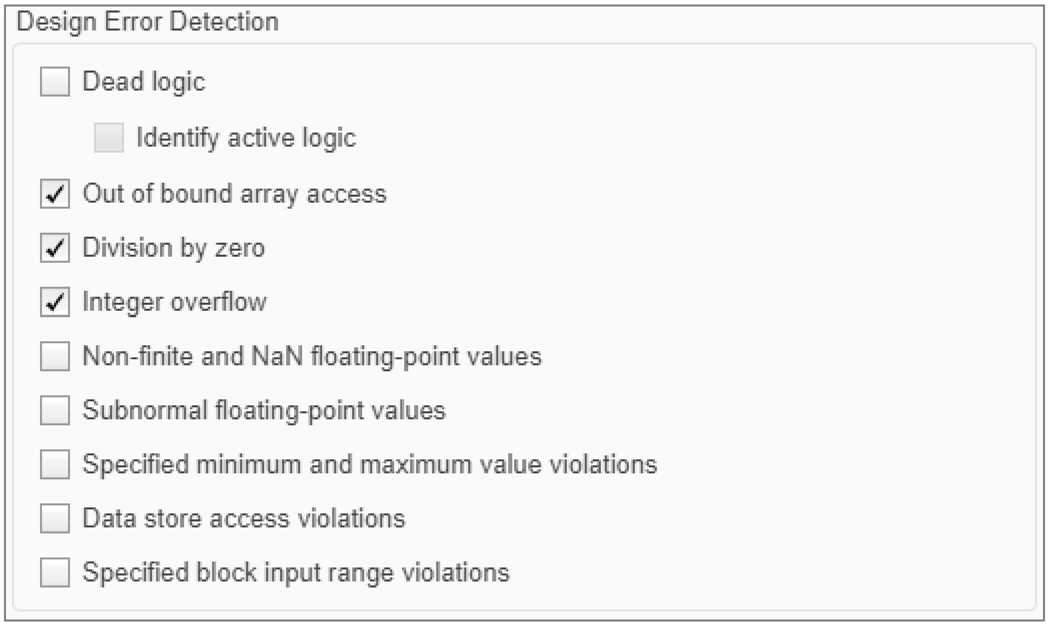
This button is enabled by **Ignore objectives satisfied in existing coverage data**.

See Also

More About

- “Design Verifier Pane” on page 15-9
- “Generate Test Cases for Model Decision Coverage” on page 7-3
- “Workflow for Test Case Generation” on page 7-2

Design Verifier Pane: Design Error Detection


Design Error Detection

- [ ] Dead logic
  - [ ] Identify active logic
- [x] Out of bound array access
- [x] Division by zero
- [x] Integer overflow
- [ ] Non-finite and NaN floating-point values
- [ ] Subnormal floating-point values
- [ ] Specified minimum and maximum value violations
- [ ] Data store access violations
- [ ] Specified block input range violations

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.

Command-Line Information
Parameter: `DVDetectDeadLogic`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Dead Logic Detection" on page 6-8

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**.

Command-Line Information
Parameter: `DVDetectActiveLogic`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Dead Logic Detection" on page 6-8

Out of bound array access

Specify whether to analyze your model for out of bound array access errors.

Settings

Default: On

☑ On

Reports out of bound array access errors in your model.

☐ Off

Does not report out of bound array access errors in your model.

Command-Line Information
Parameter: `DVDetectOutOfBounds`
Type: character array
Value: `'on'` | `'off'`
Default: `'on'`

See Also

"Detect Out of Bound Array Access Errors" on page 6-34

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.

Command-Line Information
Parameter: `DVDetectDivisionByZero`
Type: character array
Value: `'on'` | `'off'`
Default: `'on'`

See Also

"Detect Integer Overflow and Division-by-Zero Errors" on page 6-25

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.

Command-Line Information
Parameter: `DVDetectIntegerOverflow`
Type: character array
Value: `'on'` | `'off'`
Default: `'on'`

See Also

"Detect Integer Overflow and Division-by-Zero Errors" on page 6-25

Non-finite and NaN floating-point values

Specify whether to analyze your model for non-finite and NaN floating-point values.

Settings

Default: Off

☑ On

Reports non-finite and NaN floating-point values in your model.

☐ Off

Does not report non-finite and NaN floating-point values in your model.

Command-Line Information
Parameter: `DVDetectInfNaN`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Detect Non-Finite, NaN, and Subnormal Floating-Point Values" on page 6-39

Subnormal floating-point values

Specify whether to analyze your model for subnormal floating-point values.

Settings

Default: Off

☑ On

Reports subnormal floating-point values in your model.

☐ Off

Does not report subnormal floating-point values in your model.

Command-Line Information
Parameter: `DVDetectSubnormal`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Detect Non-Finite, NaN, and Subnormal Floating-Point Values" on page 6-39

Specified minimum and maximum value violations

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.

Command-Line Information
Parameter: `DVDesignMinMaxCheck`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Check for Specified Minimum and Maximum Value Violations" on page 6-29

Data store access violations

Specify whether to analyze your model for data store access violations. Design error detection checks for these violations related to Data Store Memory blocks:

- Read-before-write
- Write-after-read
- Write-after-write

Settings

Default: Off

☑ On

Reports data store access violations in your model.

☐ Off

Does not report data store access violations in your model.

Command-Line Information
Parameter: `DVDetectDSMAccessViolations`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Detecting Access Order Errors" (Simulink)

Specified block input range violations

Specify whether to analyze your model for block input range violations. The check detects input range violations for blocks with these settings:

- For these blocks, when the **Diagnostic for out-of-range input** parameter is set to `Warning` or `Error`:
  - n-D Lookup Table
  - Interpolation Using Prelookup
  - Prelookup
  - Direct Lookup Table (n-D)
- Multiport Switch blocks, when the **Diagnostic for default case** parameter is set to `Warning` or `Error`.
- Trigonometric Function blocks, when the **Approximation method** parameter is set to `CORDIC`

Note The check does not flag block input range violations for n-D Lookup Table blocks, when the **Interpolation method** is set to `Akima spline` or `Cubic spline`.

Note The check does not flag block input range violations for Trigonometric Function blocks with `CORDIC` **Approximation method**, for which the **Function** parameter is `atan2` and the data types of the input signals are `double`.

Settings

Default: Off

✓ On

Reports block input range violations in your model.

☐ Off

Does not report block input range violations in your model.

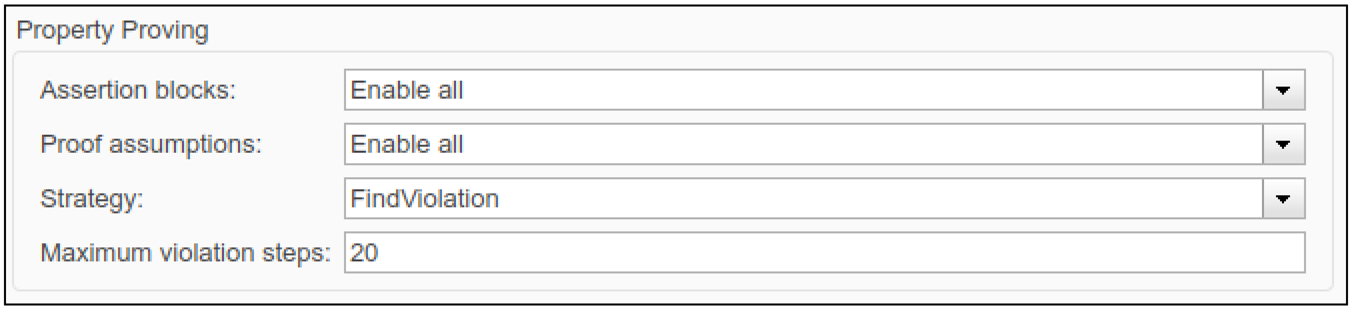
Command-Line Information

Parameter: `DVDetectBlockInputRangeViolations`
Type: character array
Value: `'on'` | `'off'`
Default: `'off'`

See Also

"Detect Block Input Range Violations"

Design Verifier Pane: Property Proving


Property Proving

Assertion blocks:	Enable all
Proof assumptions:	Enable all
Strategy:	FindViolation
Maximum violation steps:	20

In this section...

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

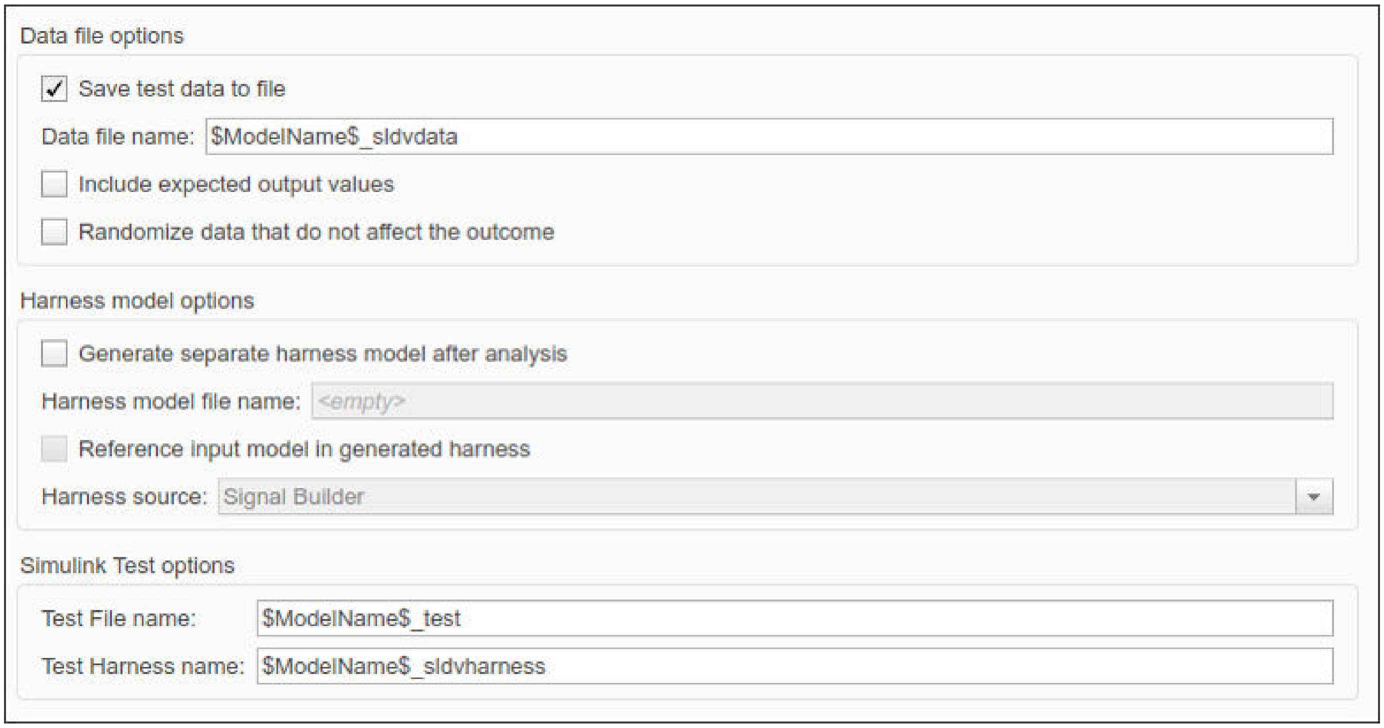
This parameter is enabled when you set **Strategy** to `FindViolation` or `ProveWithViolationDetection`.

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

Harness model options

[ ] Generate separate harness model after analysis

Harness model file name: <empty>

[ ] Reference input model in generated harness

Harness source: Signal Builder

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 13-7
- "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 13-7
- “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 13-7.
- 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 13-7
- “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

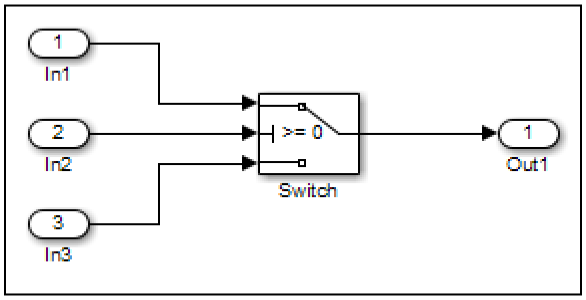
Assigns random values to test case or counterexample signals that do not affect the outcome of test or proof objectives in a model. This option can enhance traceability and improve your regression tests.

☐ 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 13-7
- “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 13-13
- “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 13-13
- “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

- 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 13-13

- “Results Interpretation and Use”

Harness source

Specify the type of Inputs block for the harness model.

Settings

Default: `Signal Builder`

`Signal Builder`
: Generates a separate harness model with the Signal Builder block as the Inputs block.

`Signal Editor`
: Generates a separate harness model with the Signal Editor block as the Inputs block.

Dependency

This parameter is enabled by **Generate separate harness model after analysis**.

Command-Line Information
Parameter: `DVHarnessSource`
Type: character array
Value: `'Signal Builder'` | `'Signal Editor'`
Default: `'Signal Builder'`

See Also

- “Simulink Design Verifier Harness Models” on page 13-13

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 if you have a Simulink Test license.

Command-Line Information
Parameter: `DVSlTestFileName`
Type: character array
Value: any valid path and file name
Default: `'$ModelName$_test'`

See Also

- “Increase Coverage by Generating Test Inputs” (Simulink Test)

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.

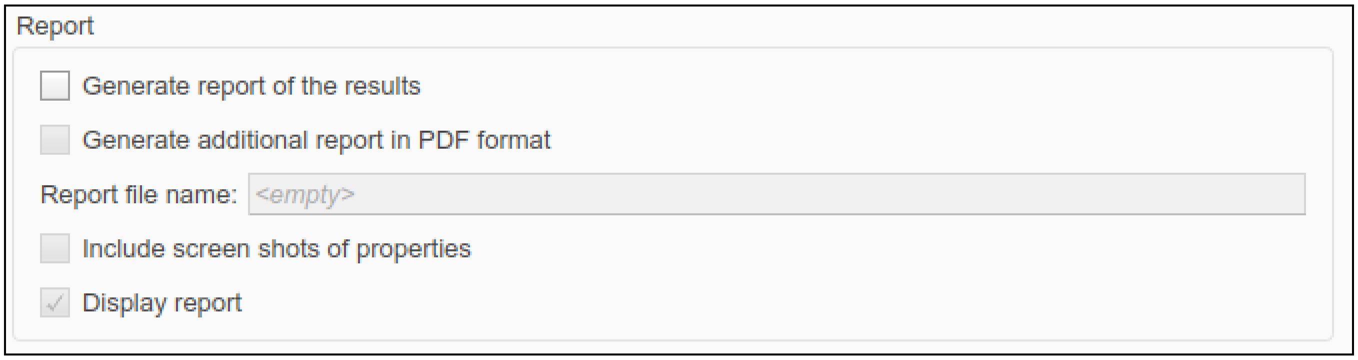
Command-Line Information

Parameter: `DVSlTestHarnessName`
Type: character array
Value: any valid file name
Default: `'$ModelName$_sldvharness'`

See Also

- “Increase Coverage by Generating Test Inputs” (Simulink Test)

Design Verifier Pane: Report



☐ Generate report of the results

☐ Generate additional report in PDF format

Report file name: <empty>

☐ Include screen shots of properties

✓ Display report

In this section...

Report Pane Overview

Specify options that control how Simulink Design Verifier reports its results.

See Also

- “Simulink Design Verifier Reports” on page 13-28
- “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 13-28
- "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 13-28
- "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 13-28
- "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 13-28
- “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 13-28
- “Results Interpretation and Use”

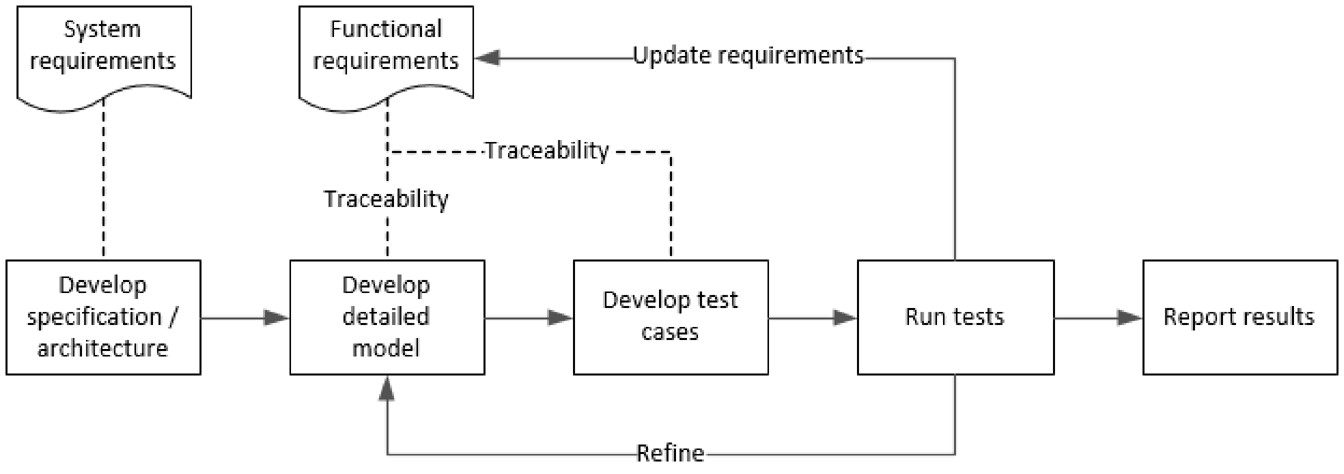
Verification and Validation

Test Model Against Requirements and Report Results

Requirements – Test Traceability Overview

Traceability between requirements and test cases helps you interpret test results and see the extent to which your requirements are verified. You can link a requirement to elements that help verify it, such as test cases in the Test Manager, `verify` statements in a Test Sequence block, or Model Verification blocks in a model. When you run tests, a pass/fail summary appears in your requirements set.

This example demonstrates a common requirements-based testing workflow for a cruise control model. You start with a requirements set, a model, and a test case. You add traceability between the tests and the safety requirements. You run the test, summarize the verification status, and report the results.



In this example, you conduct a simple test of two requirements in the set:

- That the cruise control system transitions to disengaged from engaged when a braking event has occurred
- That the cruise control system transitions to disengaged from engaged when the current vehicle speed is outside the range of 20 mph to 90 mph.

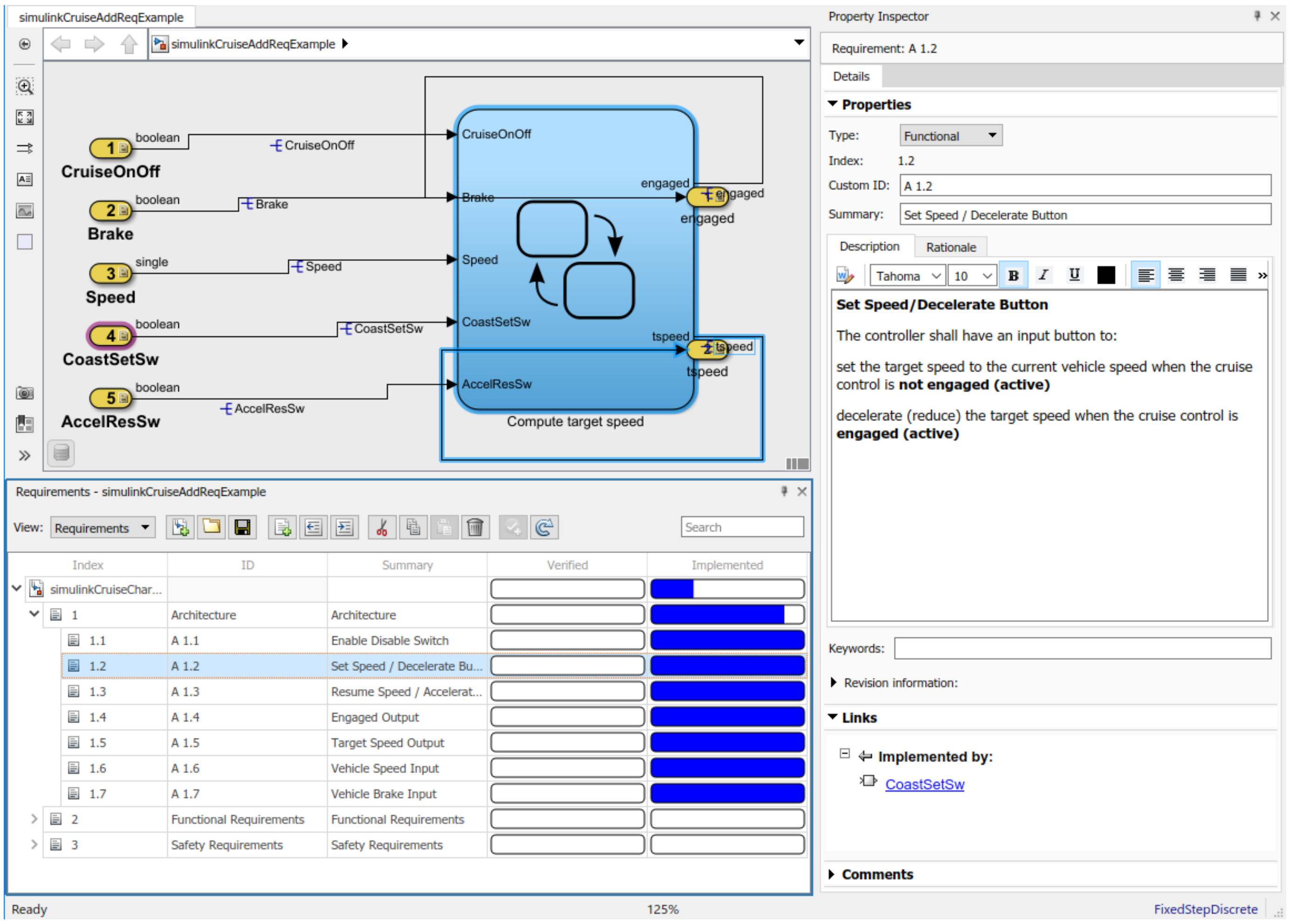
Display the Requirements

1. Create a copy of the project in a working folder. The project contains data, documents, models, and tests. Enter:

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'slVerificationCruiseStart'))
```

2. In the project `models` folder, open the `simulinkCruiseAddReqExample.slx` model.
3. Display the requirements. Click the ▪ icon in the lower-right corner of the model canvas, and select **Requirements**. The requirements appear below the model canvas.

4 Expand the requirements information to include verification and implementation status. Right-click a requirement and select **Verification Status** and **Implementation Status**.



Requirements - simulinkCruiseAddReqExample

Index	ID	Summary	Verified	Implemented
simulinkCruiseChar...				
1	Architecture	Architecture		
1.1	A 1.1	Enable Disable Switch		
1.2	A 1.2	Set Speed / Decelerate Bu...		
1.3	A 1.3	Resume Speed / Accelerat...		
1.4	A 1.4	Engaged Output		
1.5	A 1.5	Target Speed Output		
1.6	A 1.6	Vehicle Speed Input		
1.7	A 1.7	Vehicle Brake Input		
2	Functional Requirements	Functional Requirements		
3	Safety Requirements	Safety Requirements		

Requirement: A 1.2

Type: Functional
Index: 1.2
Custom ID: A 1.2
Summary: Set Speed / Decelerate Button

Set Speed/Decelerate Button

The controller shall have an input button to:

set the target speed to the current vehicle speed when the cruise control is **not engaged (active)**

decelerate (reduce) the target speed when the cruise control is **engaged (active)**

Keywords:

Revision information:

Links

Implemented by:
CoastSetSw

Comments

Ready 125% FixedStepDiscrete

5 In the Project window, open the Simulink Test file `slReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

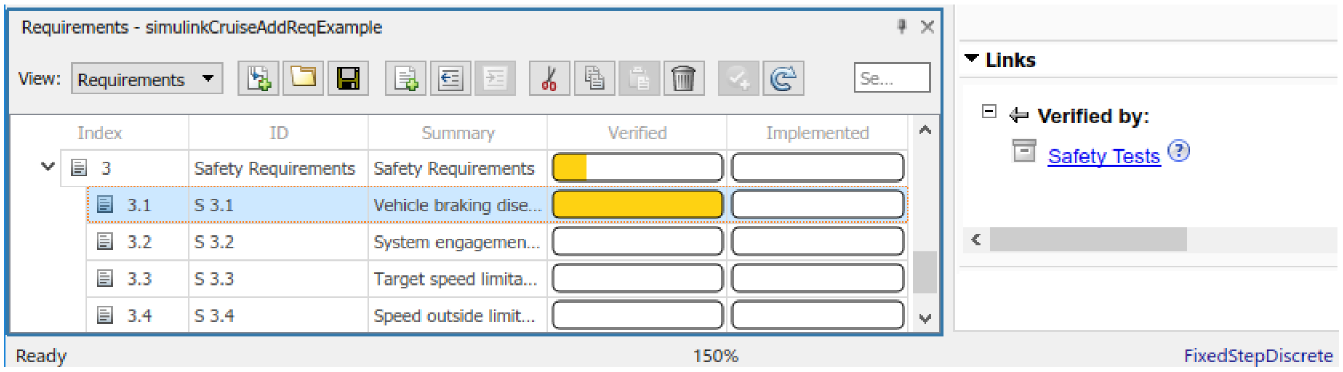
Link Requirements to Tests

Link the requirements to the test case.

1 In the Project window, open the Simulink Test file `slReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager. Explore the test suite and select `Safety Tests`.

Return to the model. Right-click on requirement `S 3.1` and select **Link from Selected Test Case**.

A link to the `Safety Tests` test case is added to **Verified by**. The yellow bars in the **Verified** column indicate that the requirements are not verified.


Requirements - simulinkCruiseAddReqExample

View: Requirements

Index	ID	Summary	Verified	Implemented
3	Safety Requirements	Safety Requirements		
3.1	S 3.1	Vehicle braking dise...		
3.2	S 3.2	System engagemen...		
3.3	S 3.3	Target speed limita...		
3.4	S 3.4	Speed outside limit...		

Links

Verified by:

Safety Tests

Ready 150% FixedStepDiscrete

2 Also add a link for item `S 3.4`.

Run the Test

The test case uses a test harness `SafetyTest_Harness1`. In the test harness, a test sequence sets the input conditions and checks the model behavior:

- The `BrakeTest` sequence engages the cruise control, then applies the brake. It includes the `verify` statement

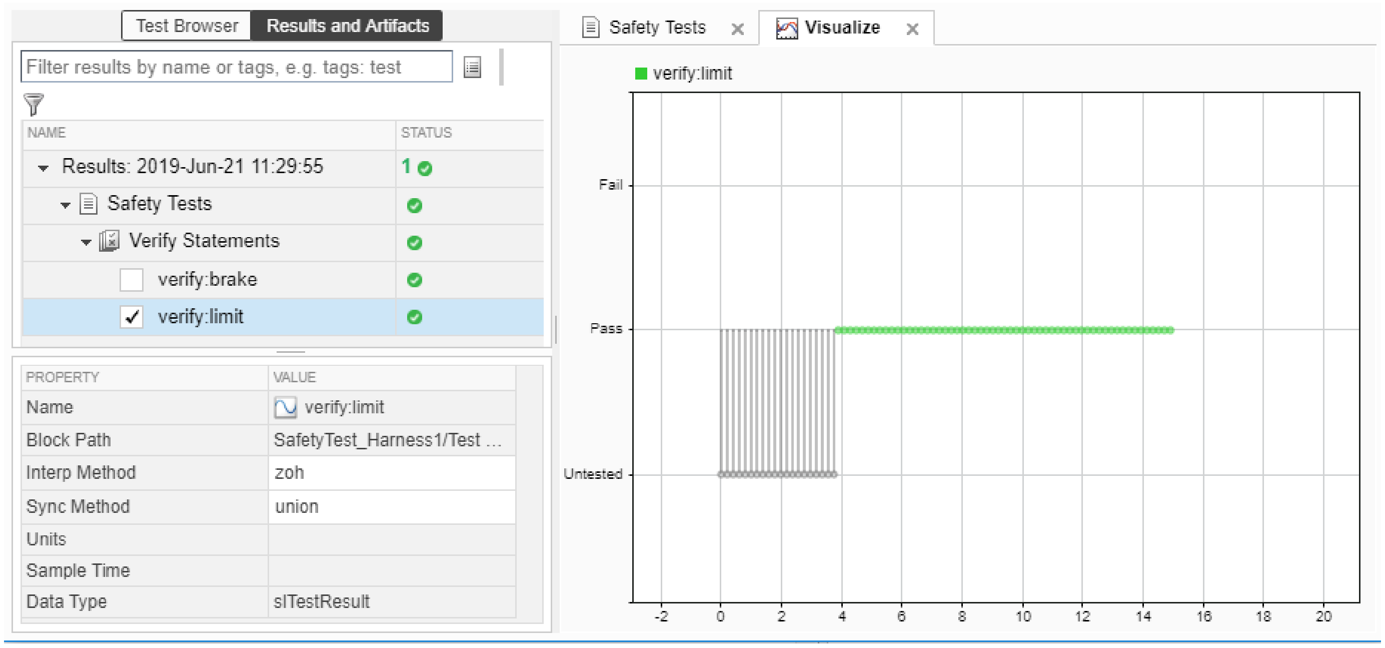
```
verify(engaged == false,...
    'verify:brake',...
    'system must disengage when brake applied')
```

- The `LimitTest` sequence engages the cruise control, then ramps up the vehicle speed until it exceeds the upper limit. It includes the `verify` statement.

```
verify(engaged == false,...
    'verify:limit',...
    'system must disengage when limit exceeded')
```

1 Return to the Test Manager. To run the test case, click **Run**.

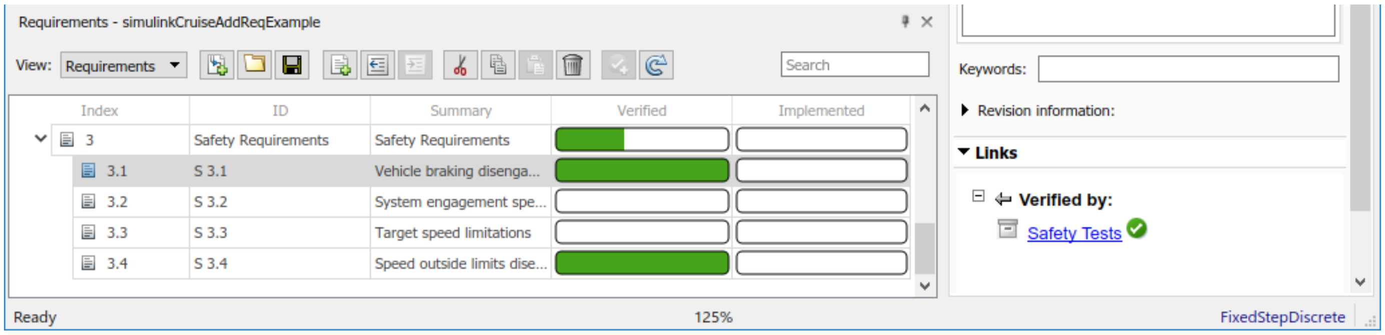
2 When the test finishes, review the results. The Test Manager shows that both assessments pass and the plot provides the detailed results of each `verify` statement.



NAME	STATUS
Results: 2019-Jun-21 11:29:55	1 ✓
Safety Tests	✓
Verify Statements	✓
verify:brake	✓
✓ verify:limit	✓

PROPERTY	VALUE
Name	verify:limit
Block Path	SafetyTest_Harness1/Test ...
Interp Method	zoh
Sync Method	union
Units	
Sample Time	
Data Type	slTestResult

3 Return to the model and refresh the Requirements. The green bar in the **Verified** column indicates that the requirement has been successfully verified.



Index	ID	Summary	Verified	Implemented
3	Safety Requirements	Safety Requirements		
3.1	S 3.1	Vehicle braking disenga...		
3.2	S 3.2	System engagement spe...		
3.3	S 3.3	Target speed limitations		
3.4	S 3.4	Speed outside limits dise...		

Report the Results

1 Create a report using a custom Microsoft Word template.

a From the Test Manager results, right-click the test case name. Select **Create Report**.

b In the Create Test Result Report dialog box, set the options:

- Title — `SafetyTest`
- Results for — `All Tests`
- File Format — `DOCX`
- For the other options, keep the default selections.

c Enter a file name and select a location for the report.

d For the **Template File**, select the `ReportTemplate.dotx` file in the **documents** project folder.

e Click **Create**.

2 Review the report.

   a The **Test Case Requirements** section specifies the associated requirements
   b The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

See Also

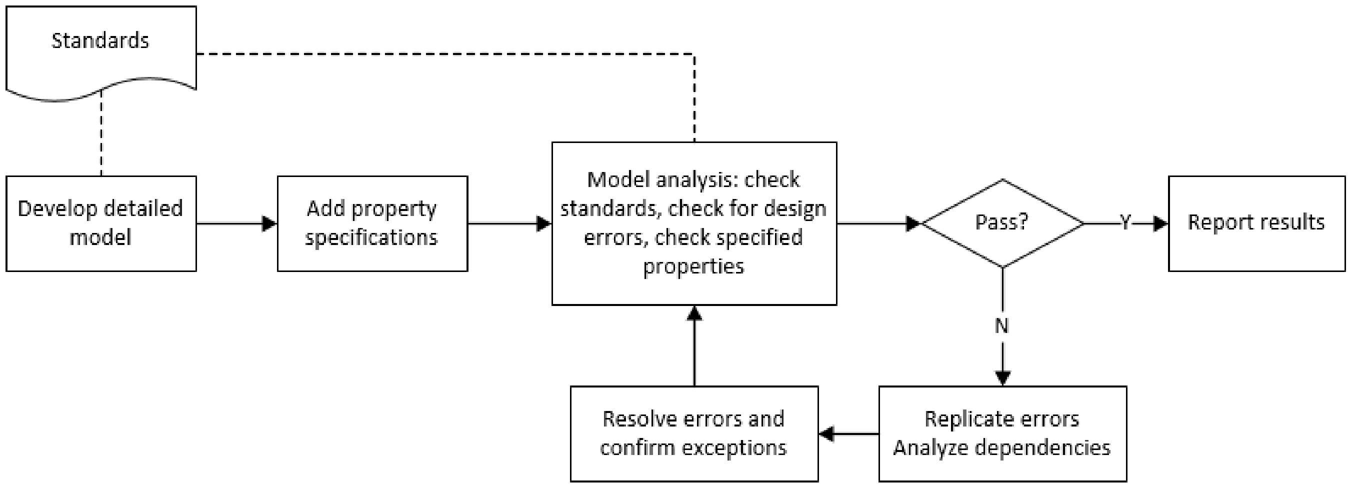
Related Examples

- “Link to Requirements” (Simulink Test)
- “Validate Requirements Links in a Model” (Simulink Requirements)
- “Customize Requirements Traceability Report for Model” (Simulink Requirements)

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 MAB style guidelines and high-integrity system design guidelines such as DO-178 and ISO 26262. Analyze your model for errors, dead logic, and conditions that violate required properties. Using the analysis results, update your model and document exceptions. Report the results using customizable templates.


Standards
Develop detailed model
Add property specifications
Model analysis: check standards, check for design errors, check specified properties
Pass?
Y
Report results
N
Replicate errors Analyze dependencies
Resolve errors and confirm exceptions

Check Model for Style Guideline Violations and Design Errors

This example shows how to use the Model Advisor to check a cruise control model for MathWorks® Advisory Board (MAB) style guideline violations and design errors. Select checks and run the analysis on the model. Iteratively debug issues using the Model Advisor and rerun checks to verify that it is in compliance. After passing your selected checks, report results.

Check Model for MAB Style Guideline Violations

In Model Advisor, you can check that your model complies with MAB modeling guidelines.

1. Create a copy of the project in a working folder. On the command line, enter

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'slVerificationCruiseStart'))
```

2. Open the model. On the command line, enter

```
open_system simulinkCruiseErrorAndStandardsExample
```

3. In the **Modeling** tab, select **Model Advisor**.
4. Click OK to choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
5. Check your model for MAB style guideline violations using Simulink Check.

a. In the left pane, in the **By Product** > **Simulink Check** > **Modeling Standards** > **MAB Checks** folder, select:

   - **Check Indexing Mode**
   - **Check model diagnostic parameters**

b. Right-click on the **MAB Checks** node and select `Run Selected Checks`.
c. Click **Check model diagnostic parameters** to review the configuration parameter settings that violate MAB 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 Check checks, select the **MAB 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**. All the checks in the folder are selected.
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**.

See Also

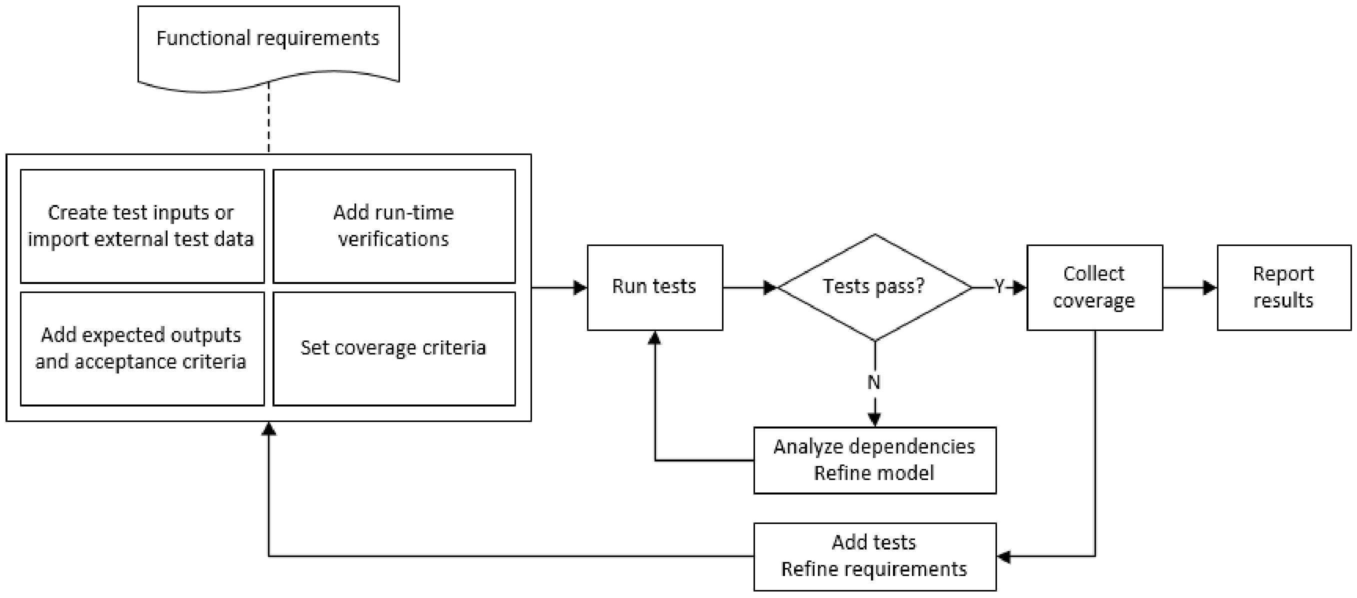
Related Examples

- “Check Model Compliance by Using the Model Advisor” (Simulink Check)
- “Collect Model Metrics Using the Model Advisor” (Simulink Check)
- “Run a Design Error Detection Analysis” on page 6-4
- “Prove Properties in a Model” on page 12-5

Perform Functional Testing and Analyze Test Coverage

Functional testing begins with building test cases based on requirements. These tests can cover key aspects of your design and verify that individual model components meet requirements. Test cases include inputs, expected outputs, and acceptance criteria.

By collecting individual test cases within test suites, you can run functional tests systematically. To check for regression, add baseline criteria to the test cases and test the model iteratively. Coverage measurement reflects the extent to which these tests have fully exercised the model. Coverage measurement also helps you to add tests and requirements to meet coverage targets.


Functional requirements
Create test inputs or import external test data
Add run-time verifications
Add expected outputs and acceptance criteria
Set coverage criteria
Run tests
Tests pass?
Y
N
Collect coverage
Report results
Analyze dependencies Refine model
Add tests Refine requirements

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 Coverage, 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:

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'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” (Simulink Test) and open the Simulink Test Manager. At the command line, enter:

```
sltest.testmanager.load('slReqTests.mldatx')
sltest.testmanager.view
```

4 Open the test sequence block. The sequence tests that the system disengages when the:

- Brake pedal is pressed
- Speed exceeds a limit

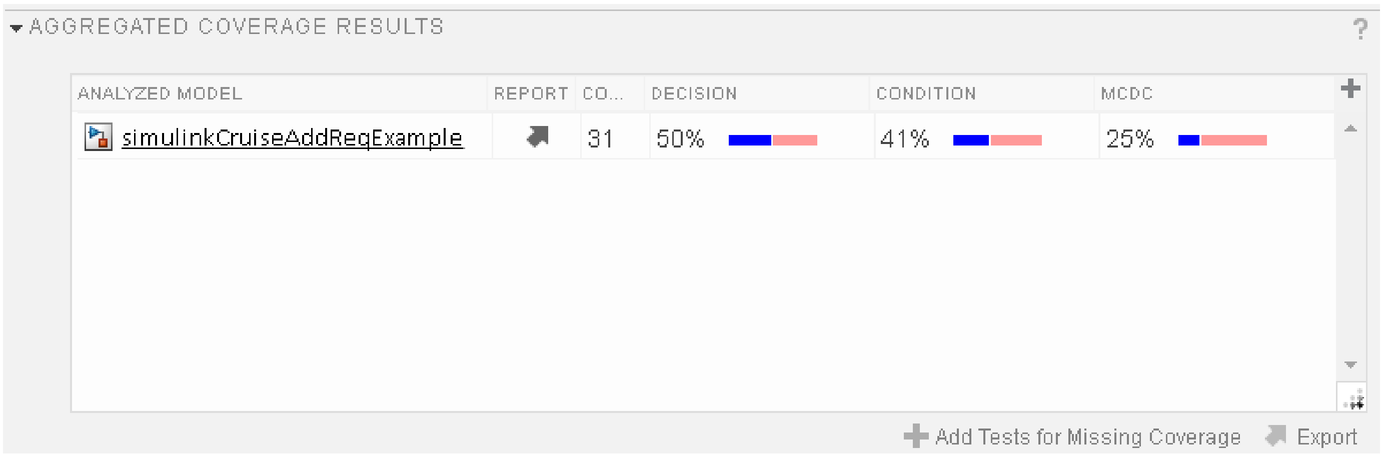
Some test sequence steps are linked to requirements document `simulinkCruiseChartReqs.docx`.

Measure Model Coverage

1 In the Simulink Test Manager, click the `slReqTests` test file.
2 To enable coverage collection for the test file, in the right page under **Coverage Settings**:

- Select **Record coverage for referenced models**
- Use **Coverage filter filename** to specify a coverage filter to use for the coverage analysis. The default setting honors the model configuration parameter settings. Leaving the field empty attaches no coverage filter.
- Select **Decision**, **Condition**, and **MCDC**.

3 To run the tests, on the Test Manager toolstrip, click **Run**.
4 When the test finishes select the Results in the Test Manager. The aggregated coverage results show that the example model achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.


AGGREGATED COVERAGE RESULTS

ANALYZED MODEL	REPORT	CO...	DECISION	CONDITION	MCDC
simulinkCruiseAddReqExample		31	50%	41%	25%

Generate Tests to Increase Model Coverage

1 Use Simulink Design Verifier to generate additional tests to increase model coverage. In **Results and Artifacts**, select the `slReqTests` test file and open the **Aggregated Coverage Results** section located in the right pane.
2 Right-click the test results and select **Add Tests for Missing Coverage**.
3 Under **Harness**, choose `Create a new harness`.
4 Click **OK** to add tests to the test suite using Simulink Design Verifier. The model being tested must either be on the MATLAB path or in the working folder.
5 On the Test Manager toolstrip, click **Run** to execute the updated test suite. The test results include coverage for the combined test case inputs, achieving increased model coverage.

See Also

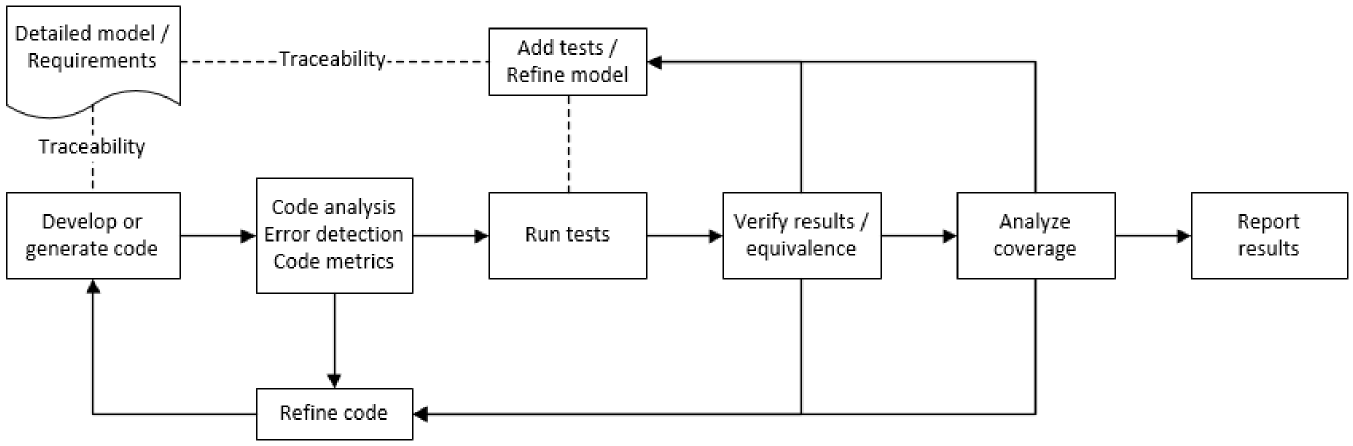
Related Examples

- “Link to Requirements” (Simulink Test)
- “Assess Model Simulation Using verify Statements” (Simulink Test)
- “Compare Model Output To Baseline Data” (Simulink Test)
- “Generate Test Cases for Model Decision Coverage” on page 7-3
- “Increase Test Coverage for a Model” (Simulink Test)

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.



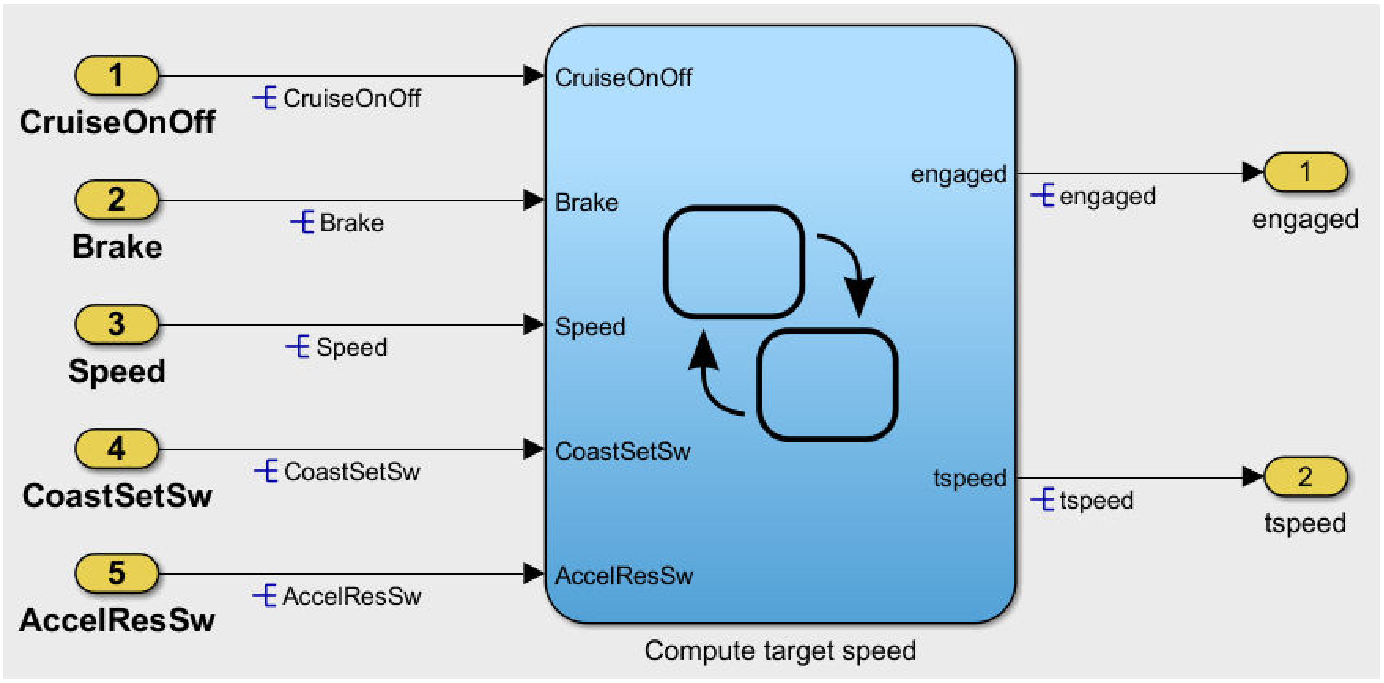
Analyze Code for Defects, Metrics, and MISRA C:2012

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 Advisor. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

1 Open the project.

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'slVerificationCruiseStart'))
```

2 From the project, open the model `simulinkCruiseErrorAndStandardsExample`.

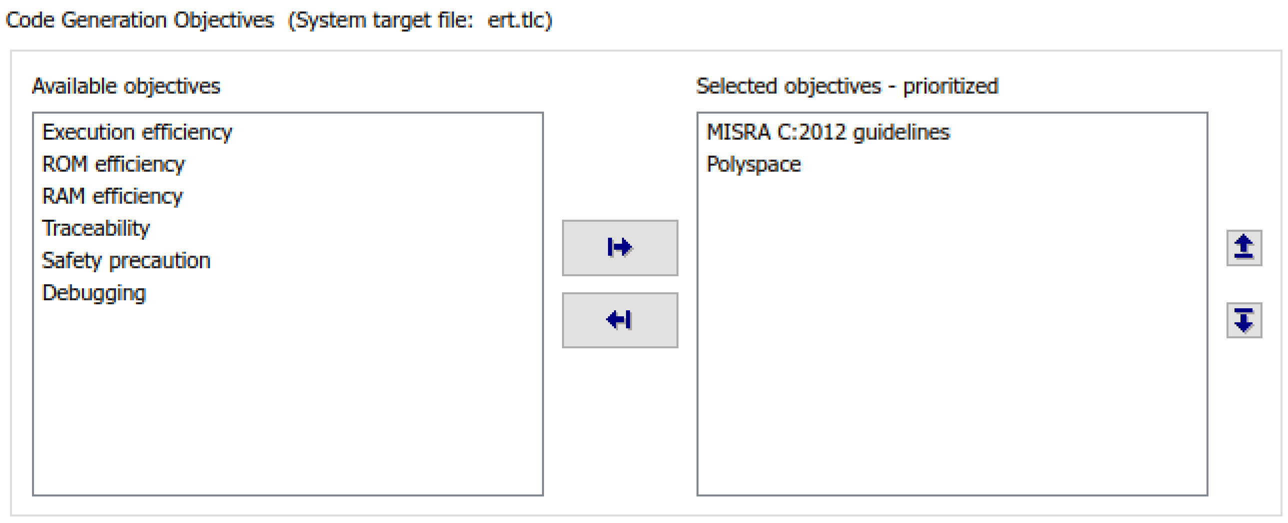


Run Code Generator Checks

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** > **Code Generation Advisor**.

2 Select the Code Generation Advisor folder. In the right pane, move `Polyspace` to **Selected objectives - prioritized** . The `MISRA C:2012 guidelines` objective is already selected.

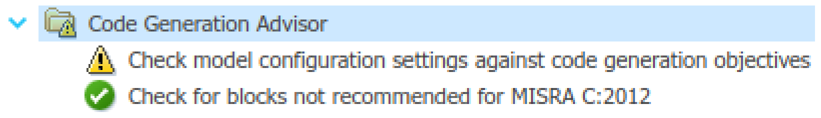


Code Generation Objectives (System target file: ert.tlc)

Available objectives	Selected objectives - prioritized
Execution efficiency	MISRA C:2012 guidelines
ROM efficiency	Polyspace
RAM efficiency	
Traceability	
Safety precaution	
Debugging	

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 model, 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 did not pass. Accept the parameter changes by selecting **Modify Parameters**.
5. Rerun the check by selecting **Run This Check**.

Run Model Advisor Checks

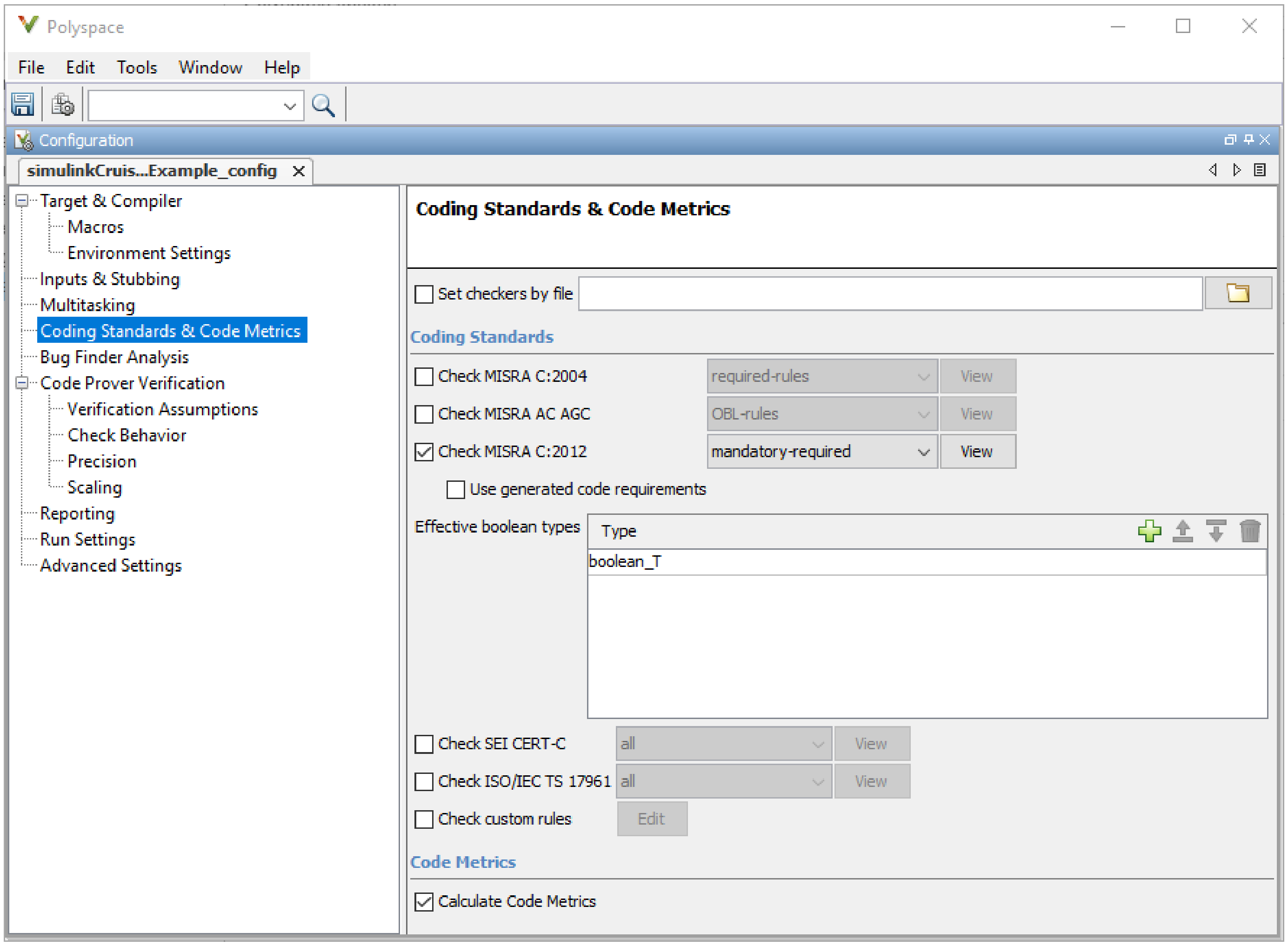
Before you generate code from your model, there are steps you can take to generate code that is more compliant with MISRA C and more compatible with Polyspace. This example shows you how to use the Model Advisor to check your model before generating code.

1. At the bottom of the Code Generation Advisor window, select **Model Advisor**.
2. Under the **By Task** folder, select the **Modeling Standards for MISRA C:2012** advisor checks.
3. Click **Run 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.

Generate and Analyze Code

After you have done the model compliance checking, you can generate the code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

1. In the Simulink editor, right-click Compute target speed and select **C/C++ Code** > **Build This Subsystem**.
2. Use the default settings for the tunable parameters and select **Build**.
3. After the code is generated, right-click Compute target speed and select **Polyspace** > **Options**.
4. Click the **Configure** (Polyspace Bug Finder) button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.



Coding Standards & Code Metrics

☐ Set checkers by file

Coding Standards

☐ Check MISRA C:2004 — required-rules — View
☐ Check MISRA AC AGC — OBL-rules — View
☑ Check MISRA C:2012 — mandatory-required — View
☐ Use generated code requirements

Effective boolean types — Type: boolean_T

☐ Check SEI CERT-C — all — View
☐ Check ISO/IEC TS 17961 — all — View
☐ Check custom rules — Edit

Code Metrics

☑ Calculate Code Metrics

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**.

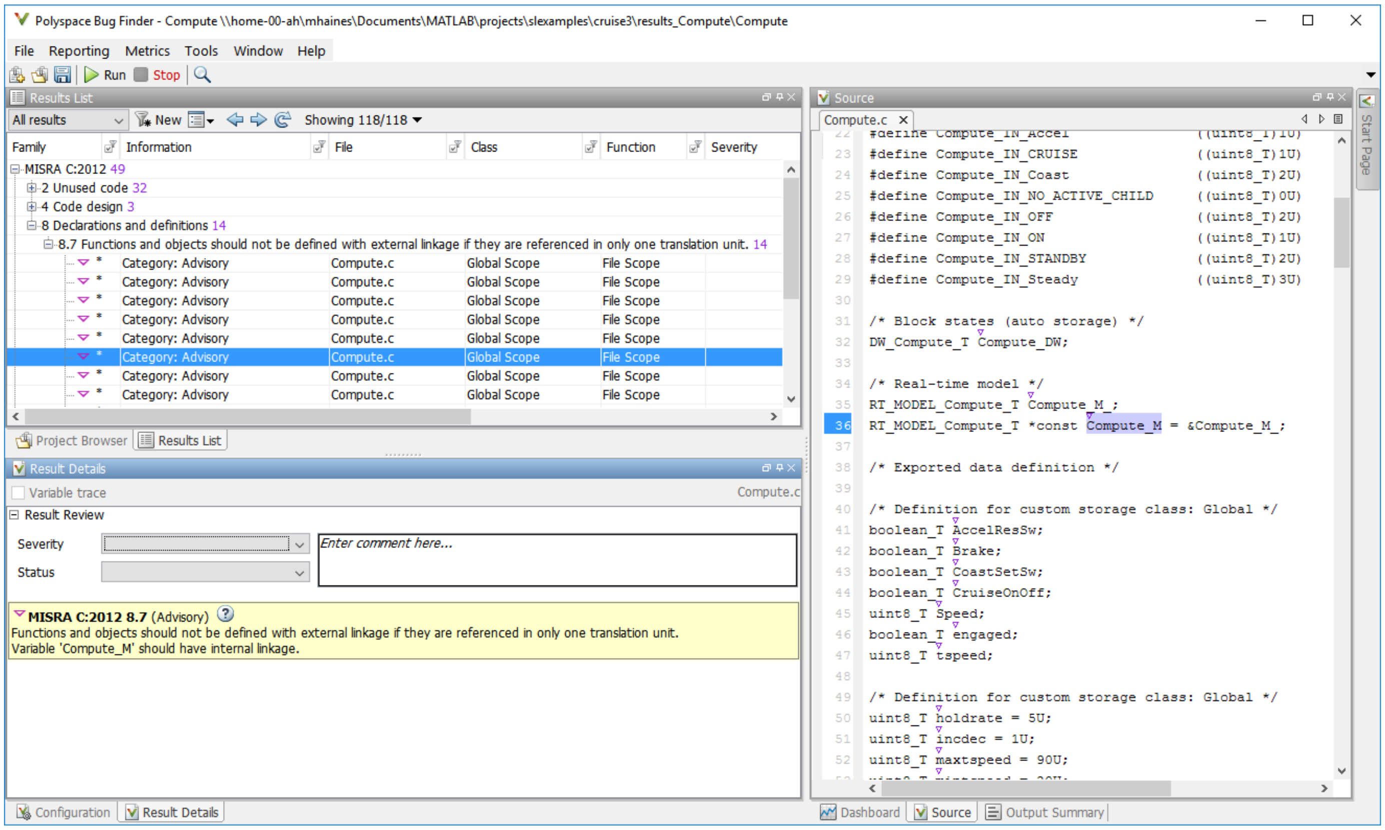
   Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

Review Results

After you run a Polyspace analysis of your generated code, the Polyspace environment shows you the results of the static code analysis.

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.



Polyspace Bug Finder - Compute \\home-00-ah\mhaines\Documents\MATLAB\projects\slexamples\cruise3\results_Compute\Compute

Results List — All results — Showing 118/118

Family	Information	File	Class	Function	Severity
MISRA C:2012 49					
2 Unused code 32					
4 Code design 3					
8 Declarations and definitions 14					
8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit. 14					
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	

Result Details — Compute.c

Result Review

Severity:

Status:

Enter comment here...

MISRA C:2012 8.7 (Advisory)
Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
Variable 'Compute_M' should have internal linkage.

Source — Compute.c

```
#define Compute_IN_Accel                 ((uint8_T)1U)
#define Compute_IN_CRUISE                ((uint8_T)1U)
#define Compute_IN_Coast                 ((uint8_T)2U)
#define Compute_IN_NO_ACTIVE_CHILD       ((uint8_T)0U)
#define Compute_IN_OFF                   ((uint8_T)2U)
#define Compute_IN_ON                    ((uint8_T)1U)
#define Compute_IN_STANDBY               ((uint8_T)2U)
#define Compute_IN_Steady                ((uint8_T)3U)

/* Block states (auto storage) */
DW_Compute_T Compute_DW;

/* Real-time model */
RT_MODEL_Compute_T Compute_M_;
RT_MODEL_Compute_T *const Compute_M = &Compute_M_;

/* Exported data definition */

/* Definition for custom storage class: Global */
boolean_T AccelResSw;
boolean_T Brake;
boolean_T CoastSetSw;
boolean_T CruiseOnOff;
uint8_T Speed;
boolean_T engaged;
uint8_T tspeed;

/* Definition for custom storage class: Global */
uint8_T holdrate = 5U;
uint8_T incdec = 1U;
uint8_T maxtspeed = 90U;
```

2 In your model, right-click Compute target speed and select **Polyspace** > **Options**.

3 Set the **Settings from** (Polyspace Bug Finder) option to `Project configuration`. This option allows you to choose a subset of MISRA rules in the Polyspace configuration.

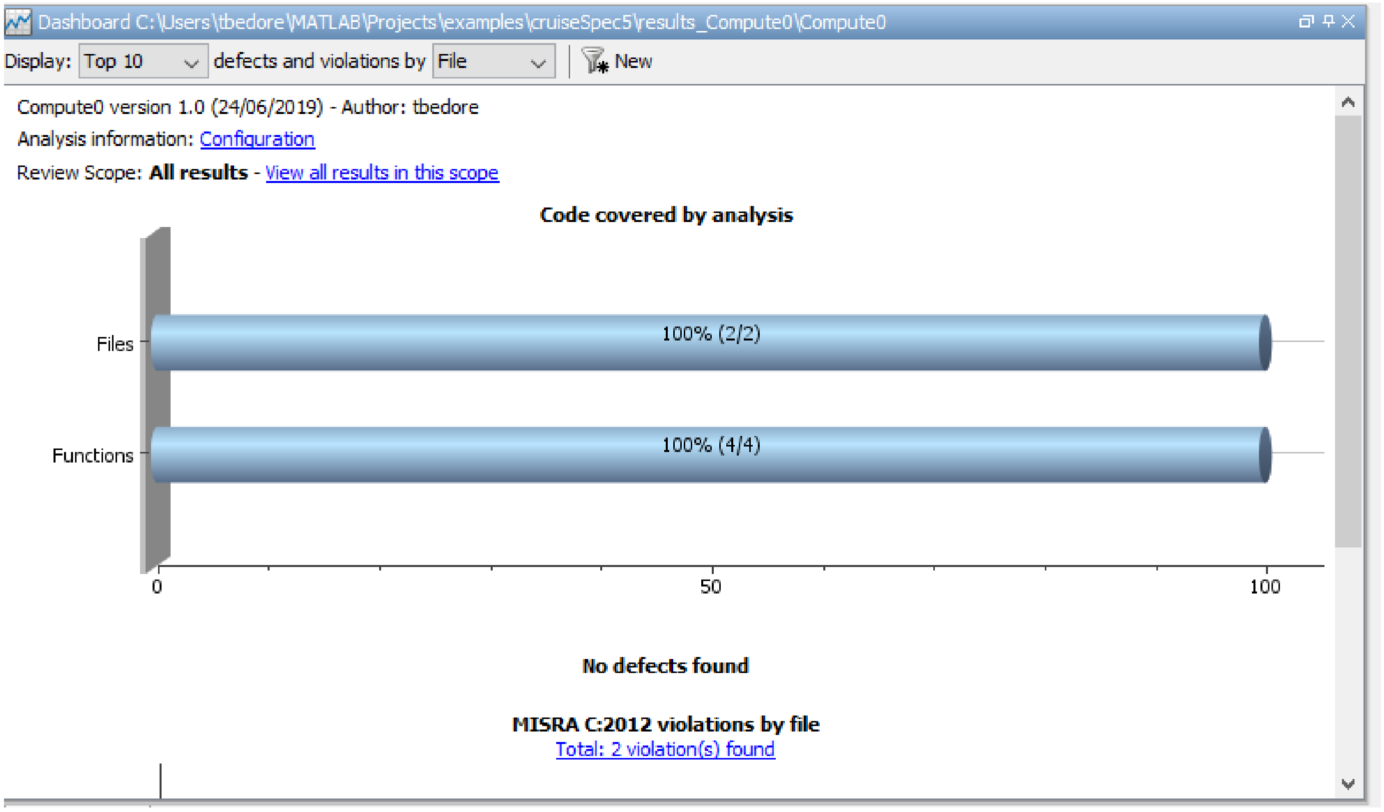
4 Click the **Configure** button.

5 In the Polyspace Configuration window, on the **Coding Standards & 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.

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, only two violations were found.


Dashboard C:\Users\tbedore\MATLAB\Projects\examples\cruiseSpec5\results_Compute0\Compute0
Display: Top 10
defects and violations by File
New

Compute0 version 1.0 (24/06/2019) - Author: tbedore
Analysis information: Configuration
Review Scope: **All results** - View all results in this scope

Code covered by analysis

Files: 100% (2/2)
Functions: 100% (4/4)

No defects found

MISRA C:2012 violations by file
Total: 2 violation(s) 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**.

See Also

Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Bug Finder)
- “Test Two Simulations for Equivalence” (Simulink Test)
- “Export Test Results and Generate Test Results Reports” (Simulink Test)

Glossary

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 “Block Replacements for Unsupported Blocks” on page 4-8).
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” (Simulink Coverage).
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” (Simulink Coverage).
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” (Simulink Coverage).
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.