

# Simulink® Design Verifier™

## User's Guide

**R2012a**

**MATLAB®  
& SIMULINK®**

## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com)  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab@mathworks.com)  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html)

Web  
Newsgroup  
Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com)  
[bugs@mathworks.com](mailto:bugs@mathworks.com)  
[doc@mathworks.com](mailto:doc@mathworks.com)  
[service@mathworks.com](mailto:service@mathworks.com)  
[info@mathworks.com](mailto:info@mathworks.com)

Product enhancement suggestions  
Bug reports  
Documentation error reports  
Order status, license renewals, passcodes  
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

*Simulink® Design Verifier™ User's Guide*

© COPYRIGHT 2007–2012 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### **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)

## Acknowledgment

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



## Acknowledgment

---

## Getting Started

---

# 1

<b>Product Description</b> .....	1-2
Key Features .....	1-2
<b>Before You Begin</b> .....	1-3
What You Need to Know .....	1-3
Required Products .....	1-3
<b>Starting the Simulink® Design Verifier™ Software</b> ...	1-4
<b>Analyzing a Model</b> .....	1-6
About This Demo .....	1-6
Opening the Model .....	1-6
Generating Test Cases .....	1-8
Combining Test Cases .....	1-28
<b>Analyzing a Subsystem</b> .....	1-29
<b>Analyzing a Stateflow Atomic Subchart</b> .....	1-31
Example: Analyzing an Atomic Subchart Using the Simulink® Design Verifier™ Software .....	1-31
<b>Basic Workflow for Using the Simulink® Design     Verifier™ Software</b> .....	1-34
<b>Learning More</b> .....	1-35
Next Step .....	1-35
Product Help .....	1-36

## How the Simulink® Design Verifier™ Software Works

# 2

<b>Analyzing a Model Using the Simulink® Design Verifier™ Software</b> .....	<b>2-2</b>
<b>Analyzing a Simple Model</b> .....	<b>2-3</b>
<b>Analyzing Model Blocks</b> .....	<b>2-6</b>
<b>Block Reduction</b> .....	<b>2-7</b>
<b>Inline Parameters</b> .....	<b>2-9</b>
<b>Analyzing Large Models</b> .....	<b>2-10</b>
<b>Handling Incompatibilities with Automatic Stubbing</b> .....	<b>2-11</b>
What Is Automatic Stubbing? .....	<b>2-11</b>
How Automatic Stubbing Works .....	<b>2-11</b>
Analyzing a Model Using Automatic Stubbing .....	<b>2-14</b>
<b>Handling Nonfinite Data</b> .....	<b>2-19</b>
<b>Approximations</b> .....	<b>2-20</b>
Approximations During Model Analysis .....	<b>2-20</b>
Types of Approximations .....	<b>2-20</b>
Converting Floating-Point Arithmetic to Rational-Number Arithmetic .....	<b>2-21</b>
Linearizing Two-Dimensional Lookup Tables .....	<b>2-21</b>
Unrolling While Loops .....	<b>2-22</b>
Ensuring the Validity of the Analysis .....	<b>2-22</b>

## Ensuring Compatibility with the Simulink® Design Verifier™ Software

# 3

<b>Checking Model Compatibility</b> .....	3-2
Model Is Compatible .....	3-3
Model Is Incompatible .....	3-3
Model is Partially Compatible .....	3-6
<b>Unsupported Simulink Software Features</b> .....	3-9
<b>Simulink Block Support Limitations</b> .....	3-12
<b>Limitations of Support for Model Blocks</b> .....	3-13
<b>Unsupported Stateflow Software Features</b> .....	3-15
ml Namespace Operator, ml Function, ml Expressions ...	3-15
C Math Functions .....	3-15
Atomic Subcharts That Call Exported Graphical Functions	
Outside a Subchart .....	3-16
Recursion and Cyclic Behavior .....	3-16
Custom C or C++ Code .....	3-18
Machine-Parented Data .....	3-18
Textual Functions with Literal String Arguments .....	3-19
<b>Support Limitations for MATLAB for Code</b>	
<b>Generation</b> .....	3-20
Unsupported MATLAB for Code Generation Features ....	3-20
Limitations of MATLAB for Code Generation Library	
Function Support .....	3-21
<b>Fixed-Point Support Limitations</b> .....	3-22

## Working with Block Replacements

### 4

<b>About Block Replacements</b> .....	4-2
<b>Built-In Block Replacements</b> .....	4-4
<b>Template for Block Replacement Rules</b> .....	4-7
<b>Defining Custom Block Replacements</b> .....	4-8
Basic Workflow for Defining Custom Block Replacements .....	4-8
Specifying Replacement Blocks .....	4-8
Writing Block Replacement Rules .....	4-9
Example: Replacing Multiport Switch Blocks .....	4-9
<b>Executing Block Replacements</b> .....	4-18
Configuring Block Replacements .....	4-18
Replacing Blocks in a Model .....	4-19

## Specifying Parameter Configurations

### 5

<b>About Parameter Configurations</b> .....	5-2
<b>Defining Parameter Configurations</b> .....	5-3
Template for Defining Parameters .....	5-3
Syntax for Defining Parameters .....	5-3
Data Types in Parameter Configuration Files .....	5-7
<b>Parameter Configuration Example</b> .....	5-9
About This Example .....	5-9
Constructing the Example Model .....	5-10
Parameterizing the Constant Block .....	5-11
Preloading the Workspace Variable .....	5-12
Specifying a Parameter Configuration .....	5-12
Analyzing the Example Model .....	5-13



## Detecting Design Errors

# 6

<b>What Is Design Error Detection?</b> .....	<b>6-2</b>
<b>Derived Ranges in Design Error Detection</b> .....	<b>6-3</b>
<b>Running a Design Error Detection Analysis</b> .....	<b>6-5</b>
Workflow for Detecting Design Errors .....	<b>6-5</b>
Understanding the Analysis Results .....	<b>6-5</b>
Reviewing the Latest Analysis Results in the Model Explorer .....	<b>6-7</b>
<b>Detecting Dead Logic</b> .....	<b>6-9</b>
Overview of Detecting Dead Logic .....	<b>6-9</b>
Model Objects That Receive Dead Logic Detection .....	<b>6-9</b>
Detecting Dead Logic in Example Model .....	<b>6-25</b>
Reviewing Analysis Results .....	<b>6-26</b>
Reviewing the Analysis Report .....	<b>6-27</b>
<b>Detecting Integer Overflow and Division-by-Zero Errors</b> .....	<b>6-29</b>
About This Example .....	<b>6-29</b>
Analyzing the Model .....	<b>6-29</b>
Reviewing the Analysis Results .....	<b>6-30</b>
<b>Checking for Specified Intermediate Minimum and Maximum Signal Values</b> .....	<b>6-35</b>
Overview of Specified Minimum and Maximum Signal Values .....	<b>6-35</b>
About This Example .....	<b>6-36</b>
Creating the Example Model .....	<b>6-36</b>
Analyzing the Model .....	<b>6-39</b>
Reviewing the Analysis Results .....	<b>6-40</b>

# 7

<b>About Test Case Generation</b> .....	7-2
Test Case Blocks .....	7-2
Test Case Functions .....	7-2
<b>Workflow for Generating Test Cases</b> .....	7-4
<b>Generating Test Cases to Achieve Decision Coverage for a Model</b> .....	7-5
Constructing the Example Model .....	7-5
Checking Compatibility of the Example Model .....	7-7
Configuring Test Generation Options .....	7-8
Analyzing the Example Model .....	7-9
Reviewing the Analysis Results .....	7-10
Customizing Test Generation .....	7-18
Reanalyzing the Example Model .....	7-21
Analyzing Contradictory Models .....	7-22
<b>Generating Test Cases for a Subsystem</b> .....	7-23

## Extending Existing Test Cases

# 8

<b>When to Extend Existing Test Cases</b> .....	8-2
<b>Common Workflow for Extending Existing Test Cases</b> .....	8-3
<b>Example: Extending Existing Test Cases for a Model that Uses Temporal Logic</b> .....	8-4
Creating a Starting Test Case .....	8-4
Logging the Starting Test Case .....	8-7
Extending the Existing Test Cases .....	8-8
Verifying the Analysis Results .....	8-10

<b>Example: Extending Existing Test Cases for a Closed-Loop System</b> .....	8-11
Logging a Starting Test Case .....	8-11
Extending the Existing Test Cases .....	8-12

<b>Example: Extending Existing Test Cases for a Modified Model</b> .....	8-14
Creating Starting Test Cases .....	8-14
Extending the Existing Test Cases .....	8-15

## Achieving Test Cases for Missing Model Coverage

# 9

<b>Generating Test Cases for Missing Coverage Data</b> ....	9-2
---	-----

<b>Example: Achieving Missing Coverage in a Referenced Model</b> .....	9-3
Recording Coverage Data for the Model .....	9-3
Finding Test Cases for the Missing Coverage .....	9-5
Achieving the Missing Coverage .....	9-5
Verifying 100% Model Coverage .....	9-6

<b>Achieving Missing Coverage for Subsystems and Model Blocks</b> .....	9-7
---	-----

<b>Example: Achieving Missing Coverage in a Closed-Loop Simulation Model</b> .....	9-8
Recording Coverage Data for the Model .....	9-8
Finding Test Cases for Missing Coverage .....	9-10

## Verifying Model Components

# 10

<b>What Is Component Verification?</b> .....	10-2
--	------

Component Verification Approaches .....	10-2
Using Simulink® Design Verifier™ Tools for Component Verification .....	10-2
<b>Functions for Component Verification .....</b>	<b>10-4</b>
<b>Example: Verifying a Component for Code</b>	
<b>Generation .....</b>	<b>10-6</b>
About the Example Model .....	10-6
Preparing the Component for Verification .....	10-9
Recording Coverage for the Component .....	10-11
Using the Simulink® Design Verifier™ Software to Record Additional Coverage .....	10-12
Combining the Harness Models .....	10-13
Executing the Component in Simulation Mode .....	10-14
Executing the Component in Software-in-the-Loop (SIL) Mode .....	10-15

## Considering Specified Minimum and Maximum Values for Inputs During Analysis

# 11

<b>Overview .....</b>	<b>11-2</b>
Simulink® Design Verifier™ Support for Specified Input Minimum and Maximum Values .....	11-2
Limitations of Simulink® Design Verifier™ Support for Specified Minimum and Maximum Values .....	11-3
<b>Example: Output Minimum and Maximum Values on     Inport Blocks .....</b>	<b>11-4</b>
<b>sldvData Fields for Minimum and Maximum Input     Values .....</b>	<b>11-6</b>
<b>Example: Minimum and Maximum Values in     Simulink.Signal Objects .....</b>	<b>11-8</b>

<b>Example: Minimum and Maximum Values on Stateflow Data Objects</b> .....	11-10
<b>Example: Minimum and Maximum Values in Subsystems</b> .....	11-13
<b>Example: Minimum and Maximum Values in Global Data Storage</b> .....	11-16

## Proving Properties of a Model

# 12

<b>About Property Proving</b> .....	12-2
Proof Blocks .....	12-2
Proof Functions .....	12-3
 <b>Workflow for Proving Model Properties</b> .....	 12-4
 <b>Proving Properties in a Model</b> .....	 12-5
About This Example .....	12-5
Constructing the Example Model .....	12-6
Checking Compatibility of the Example Model .....	12-7
Instrumenting the Example Model .....	12-9
Configuring Property-Proving Options .....	12-10
Analyzing the Example Model .....	12-11
Reviewing the Analysis Results .....	12-11
Customizing the Example Proof .....	12-21
Reanalyzing the Example Model .....	12-22
Reviewing the Results of the Second Analysis .....	12-23
Analyzing Contradictory Models .....	12-26
Proving Properties in a Large Model .....	12-27
 <b>Using a Verification Model to Prove System-Level Properties</b> .....	 12-28
When to Use a Verification Model for Property Proving ..	12-28
About this Example .....	12-28
Understanding the Verification Model .....	12-29
Proving the Properties of the Design Model .....	12-29

Fixing the Verification Model .....	12-31
<b>Proving Properties in a Subsystem .....</b>	<b>12-32</b>
<b>Property-Proving Examples .....</b>	<b>12-33</b>
Basic Properties .....	12-33
Temporal Properties .....	12-35

## Reviewing the Results

# 13

<b>Highlighted Results on the Model .....</b>	<b>13-2</b>
When to Highlight Results on the Model .....	13-2
Enabling Highlighted Results on a Model .....	13-2
Simulink Design Verifier Results Window .....	13-3
Green Highlighting on Model .....	13-3
Red Highlighting on Model .....	13-3
Orange Highlighting on Model .....	13-4
Gray Highlighting on Model .....	13-6
<b>Simulink® Design Verifier™ Data Files .....</b>	<b>13-7</b>
About Simulink® Design Verifier™ Data Files .....	13-7
Overview of the sldvData Structure .....	13-7
Model Information Fields in sldvData .....	13-8
Simulating Models with Simulink® Design Verifier™ Data Files .....	13-13
<b>Harness Model .....</b>	<b>13-15</b>
About the Harness Model .....	13-15
Creating a Harness Model .....	13-15
Anatomy of a Harness Model .....	13-16
Configuration of the Harness Model .....	13-21
Simulating the Harness Model .....	13-22
<b>SystemTest TEST-Files .....</b>	<b>13-24</b>
<b>Simulink® Design Verifier™ Reports .....</b>	<b>13-27</b>
About Simulink® Design Verifier™ Reports .....	13-27

Creating Analysis Reports .....	13-27
Front Matter .....	13-28
Summary Chapter .....	13-28
Analysis Information Chapter .....	13-28
Derived Ranges Chapter .....	13-34
Objectives Status Chapters .....	13-35
Model Items Chapter .....	13-43
Design Errors Chapter .....	13-44
Test Cases Chapter .....	13-45
Properties Chapter .....	13-50
<b>Simulink® Design Verifier™ Log Files .....</b>	<b>13-52</b>
<b>Reviewing Analysis Results in the Model Explorer ....</b>	<b>13-53</b>

## Analyzing Large Models and Improving Performance

# 14

<b>Sources of Model Complexity .....</b>	<b>14-2</b>
<b>Analyzing a Large Model .....</b>	<b>14-3</b>
Types of Large Model Problems .....	14-3
Using the Default Parameter Values .....	14-4
Modifying the Analysis Parameters .....	14-5
Using the Large Model Optimization .....	14-6
Stopping the Analysis Before Completion .....	14-6
<b>Generating Reports for Large Models .....</b>	<b>14-8</b>
<b>Managing Model Data to Simplify the Analysis .....</b>	<b>14-9</b>
Simplifying Data Types .....	14-9
Constraining Data .....	14-9
<b>Partitioning Model Inputs and Generating Tests     Incrementally .....</b>	<b>14-13</b>

<b>Analyzing the Model Using a Bottom-Up Approach</b> . . .	<b>14-15</b>
<b>Extracting Subsystems for Analysis</b> . . . . .	<b>14-16</b>
Overview of Subsystem Extraction . . . . .	14-16
sldvextract Function . . . . .	14-17
Structure of the Extracted Model . . . . .	14-17
Analyzing Subsystems That Read from Global Data	
Storage . . . . .	14-17
Analyzing Function-Call Subsystems . . . . .	14-19
<b>Analyzing Logical Operations</b> . . . . .	<b>14-23</b>
<b>Handling Models with Large State Spaces</b> . . . . .	<b>14-24</b>
<b>Handling Problems with Counters and Timers</b> . . . . .	<b>14-25</b>
<b>Techniques for Proving Properties of Large Models</b> . .	<b>14-27</b>
Finding Property Violations While Designing Your	
Model . . . . .	14-27
Combining Proving Properties and Finding Proof	
Violations . . . . .	14-28

## Simulink® Design Verifier™ Configuration Parameters

# 15

<b>Overview of Simulink® Design Verifier™ Configuration Parameters</b> . . . . .	<b>15-2</b>
<b>Design Verifier Pane</b> . . . . .	<b>15-3</b>
Design Verifier Pane Overview . . . . .	15-4
Mode . . . . .	15-4
Maximum analysis time . . . . .	15-6
Display unsatisfiable test objectives . . . . .	15-7
Automatic stubbing of unsupported blocks and functions . .	15-8
Use specified input minimum and maximum values . . . . .	15-9
Output directory . . . . .	15-10
Make output file names unique by adding a suffix . . . . .	15-12



<b>Design Verifier Pane: Block Replacements</b> .....	<b>15-13</b>
Block Replacements Pane Overview .....	15-14
Apply block replacements .....	15-15
List of block replacement rules .....	15-16
File path of the output model .....	15-17
<b>Design Verifier Pane: Parameters</b> .....	<b>15-18</b>
Parameters Pane Overview .....	15-19
Apply parameters .....	15-19
Parameter configuration file .....	15-19
<b>Design Verifier Pane: Test Generation</b> .....	<b>15-21</b>
Test Generation Pane Overview .....	15-23
Model coverage objectives .....	15-24
Test conditions .....	15-25
Test objectives .....	15-26
Maximum test case steps .....	15-27
Test suite optimization .....	15-28
Extend existing test cases .....	15-29
Data file .....	15-30
Ignore objectives satisfied by existing test cases .....	15-31
Ignore objectives satisfied in existing coverage data .....	15-31
Coverage data file .....	15-32
Ignore objectives based on filter .....	15-33
Coverage filter file .....	15-34
<b>Design Verifier Pane: Design Error Detection</b> .....	<b>15-35</b>
Design Error Detection Pane Overview .....	15-36
Dead Logic .....	15-36
Integer overflow .....	15-37
Division by zero .....	15-37
Check specified intermediate minimum and maximum values .....	15-38
<b>Design Verifier Pane: Property Proving</b> .....	<b>15-40</b>
Property Proving Pane Overview .....	15-41
Assertion blocks .....	15-42
Proof assumptions .....	15-43
Strategy .....	15-44
Maximum violation steps .....	15-45
<b>Design Verifier Pane: Results</b> .....	<b>15-46</b>

Results Pane Overview .....	15-48
Save test data to file .....	15-49
Data file name .....	15-50
Include expected output values .....	15-51
Randomize data that does not affect outcome .....	15-52
Display results of the analysis on the model .....	15-53
Save test harness as model .....	15-55
Harness model file name .....	15-56
Reference input model in generated harness .....	15-57
Save test harness as SystemTest TEST-file (will reference saved data file) .....	15-59
SystemTest file name .....	15-60
<b>Design Verifier Pane: Report .....</b>	<b>15-61</b>
Report Pane Overview .....	15-62
Generate report of the results .....	15-63
Report file name .....	15-64
Include screen shots of properties .....	15-65
Display report .....	15-66
<b>Parameter Command-Line Information Summary ....</b>	<b>15-67</b>

## Simulink Block Support

# 16

Overview of Simulink Block Support .....	16-2
Additional Math and Discrete Library .....	16-3
Commonly Used Blocks Library .....	16-4
Continuous Library .....	16-5
Discontinuities Library .....	16-6
Discrete Library .....	16-7

<b>Logic and Bit Operations Library</b> .....	<b>16-8</b>
<b>Lookup Tables Library</b> .....	<b>16-9</b>
<b>Math Operations Library</b> .....	<b>16-10</b>
<b>Model Verification Library</b> .....	<b>16-12</b>
<b>Model-Wide Utilities Library</b> .....	<b>16-13</b>
<b>Ports &amp; Subsystems Library</b> .....	<b>16-14</b>
<b>Signal Attributes Library</b> .....	<b>16-16</b>
<b>Signal Routing Library</b> .....	<b>16-17</b>
<b>Sinks Library</b> .....	<b>16-18</b>
<b>Sources Library</b> .....	<b>16-19</b>
<b>User-Defined Functions Library</b> .....	<b>16-20</b>

---

**Glossary**

---

**Examples**

**A**

Generating Test Cases .....	A-2
Automatic Stubbing .....	A-3
Working with Block Replacements .....	A-4
Specifying Parameter Configurations .....	A-5
Component Verification .....	A-6
Considering Specified Minimum and Maximum Inputs .....	A-7
Proving Properties of a Model .....	A-8

---

**Index**

# Getting Started

---

- “Product Description” on page 1-2
- “Before You Begin” on page 1-3
- “Starting the Simulink® Design Verifier™ Software” on page 1-4
- “Analyzing a Model” on page 1-6
- “Analyzing a Subsystem” on page 1-29
- “Analyzing a Stateflow Atomic Subchart” on page 1-31
- “Basic Workflow for Using the Simulink® Design Verifier™ Software” on page 1-34
- “Learning More” on page 1-35

## Product Description

### **Identify design errors, generate test vectors, and verify designs against requirements**

Simulink Design Verifier uses formal methods to identify hard-to-find design errors in models without requiring extensive tests or simulation runs. Design errors detected include dead logic, integer overflow, division by zero, and violations of design properties and assertions.

Simulink Design Verifier highlights blocks in the model containing these errors and blocks proven to be without them. For each block with an error, it calculates signal-range boundaries and generates a test vector that reproduces the error in simulation.

The generated test vectors provide simulation inputs that exercise functionality captured in the model structure and specified by the test objectives. The test vectors, together with the design properties and test objectives, can be used to verify code running in software-in-the-loop (SIL) and processor-in-the-loop (PIL) test configurations.

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

Learn more about verification, validation, and test in Model-Based Design.

## Key Features

- Polyspace® and Prover Plug-In formal analysis engines
- Detection of dead logic, integer and fixed-point overflows, division by zero, and violations of design properties
- Blocks and functions for modeling functional and safety requirements
- Test vector generation from functional requirements and model coverage objectives, including condition, decision, and modified condition/decision (MCDC)
- Property proving, with generation of violation examples for analysis and debugging
- Fixed-point and floating-point model support

## Before You Begin

In this section...
“What You Need to Know” on page 1-3
“Required Products” on page 1-3

### What You Need to Know

Getting started with the Simulink Design Verifier software requires that you have experience using model coverage, as well as building and running Simulink models.

For more information, see:

- “Model Coverage Analysis” in the *Simulink Verification and Validation™ User’s Guide*
- *Simulink Getting Started Guide* and *Simulink User’s Guide*

### Required Products

You must have the following products installed to use the Simulink Design Verifier software:

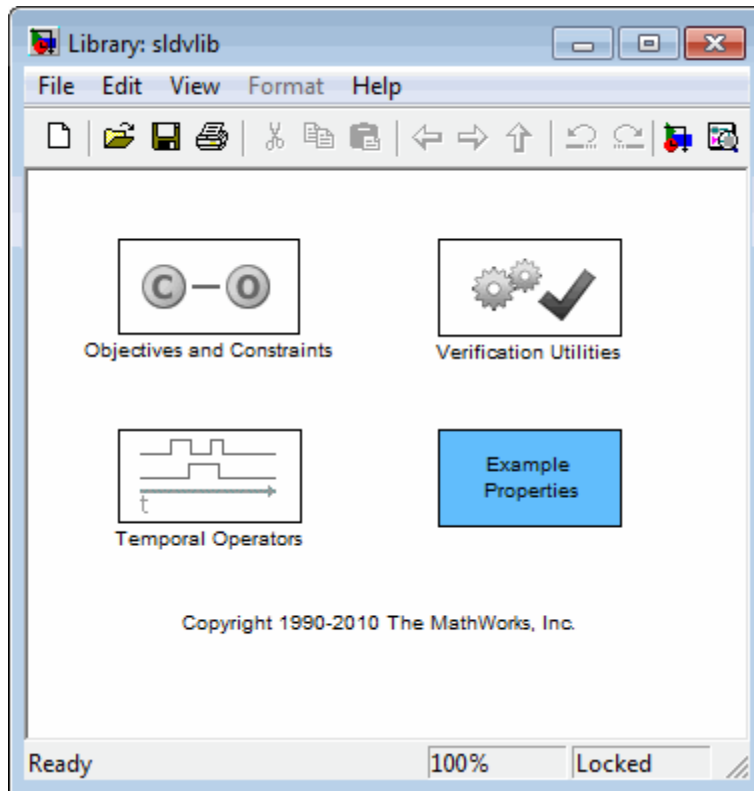
- MATLAB®
- Simulink
- Simulink Verification and Validation

If you want to use the Simulink Design Verifier software to analyze models that contain Stateflow® charts, you must have the Stateflow installed.

## Starting the Simulink Design Verifier Software

The Simulink Design Verifier software is part of your MATLAB installation.

To open the Simulink Design Verifier block library, type at the MATLAB 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.

## Analyzing a Model

In this section...
“About This Demo” on page 1-6
“Opening the Model” on page 1-6
“Generating Test Cases” on page 1-8
“Combining Test Cases” on page 1-28

### About This Demo

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

### Opening the Model

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

```
sldvdemo_cruise_control
```

Simulink Design Verifier  
Cruise Control Test Generation

The diagram shows a central block labeled "Controller". It has six input ports on the left: "enable", "brake", "set", "speed", "inc", and "dec". Each input is connected to a numbered oval: 1 for enable, 2 for brake, 3 for set, 6 for speed, 4 for inc, and 5 for dec. A circular constraint icon with "C" is placed over the speed input, with a text box "[0 100]" next to it. The controller has two output ports on the right: "throt" and "target". "throt" is connected to oval 1, and "target" is connected to oval 2.

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

**Run (double-click)**      **Toggle Speed Constraint (double-click)**      **View Options (double-click)**

Run Simulink Design Verifier      Toggle Constraint      View Simulink Design Verifier Options

Copyright 2006-2010 The MathWorks, Inc.

Ready      100%      FixedStepDiscrete

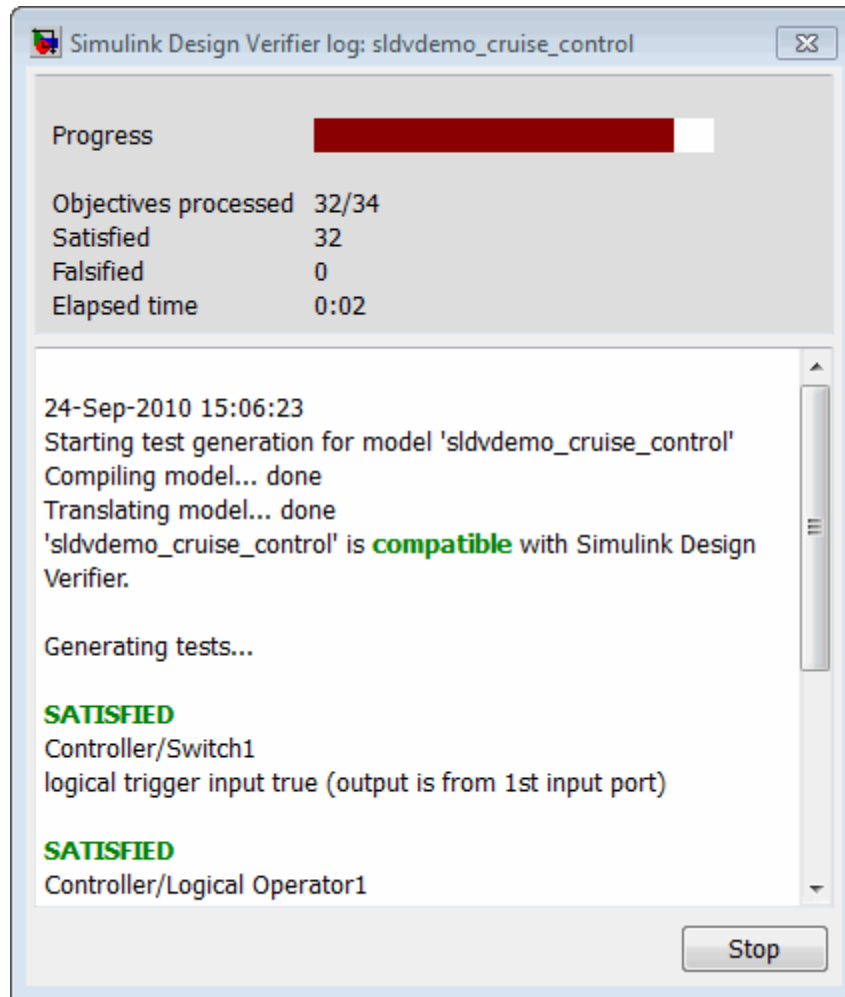
## Generating Test Cases

- “Running the Analysis” on page 1-8
- “Generating Analysis Results” on page 1-10
- “Highlighting Analysis Results on Model” on page 1-11
- “Generating a Detailed Analysis Report” on page 1-14
- “Creating a Harness Model” on page 1-22
- “Simulating Tests and Producing a Model Coverage Report” on page 1-27

## Running the Analysis

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

The Simulink Design Verifier software begins analyzing the model to generate test cases. During its analysis, the software displays a log window.



The log window shows you the progress of the Simulink Design Verifier analysis.

If you need to terminate an analysis while it is running, click **Stop**. The software asks if you want to produce results. If you click **Yes**, the software creates a data file based on the results achieved so far. The path name of the data file appears in the log window.

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

For more information, see “Simulink® Design Verifier™ Data Files” on page 13-7.

## **Generating Analysis Results**

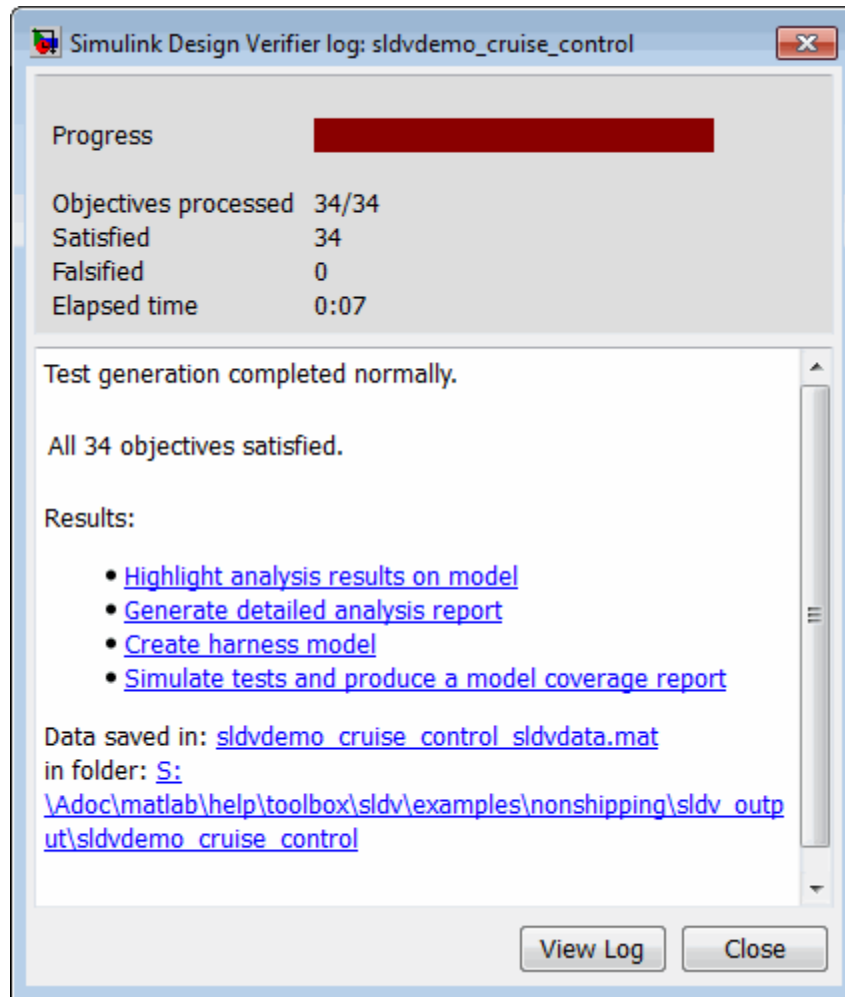
When the Simulink Design Verifier software completes its analysis of the `sldvdemo_cruise_control` model, the log window displays several options:

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

---

**Note** When you analyze other models, depending on the results of the analysis, you may see a subset of these four options.

---



The sections that follow describe these options in detail.

### Highlighting Analysis Results on Model

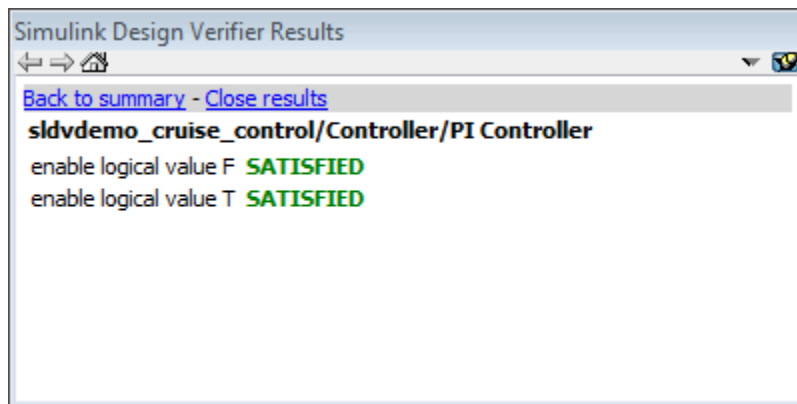
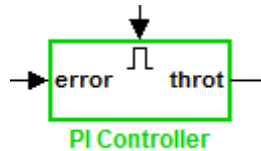
In the Simulink Design Verifier log window, if you click **Highlight analysis results on model**, the software highlights objects in the model in three different colors, depending on the analysis results:

- “Green: Objectives Satisfied” on page 1-12
- “Orange: Objectives Undecided” on page 1-13
- “Red: Objectives Unsatisfiable” on page 1-13

When you highlight the analysis results on a model, the Simulink Design Verifier Results window opens. When you click an object in the model that has analysis results, that window 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 Informer lists the two satisfied test objectives for the PI controller subsystem.

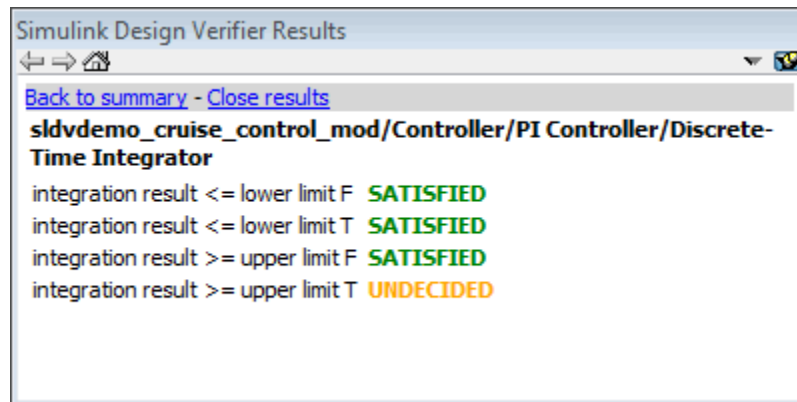




**Orange: Objectives Undecided.** Orange outline indicates that the analysis was not able to determine if an objective was satisfiable or not. This situation might occur when:

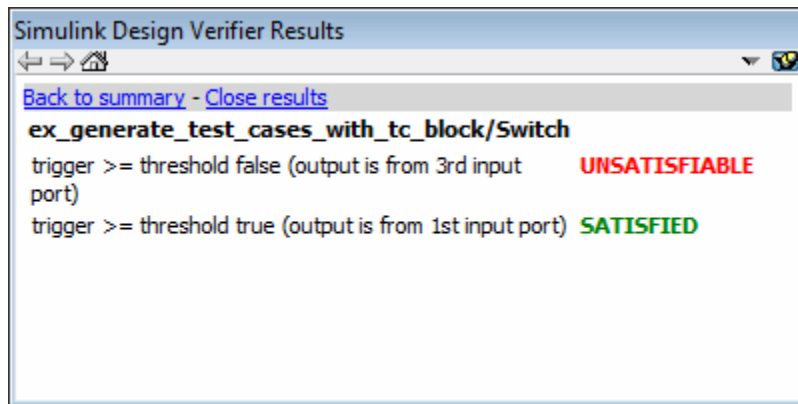
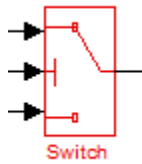
- The analysis times out
- The software satisfies test objectives without generating test cases due to:
  - Automatic stubbing errors
  - Limitations of the analysis engine

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



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

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



## Generating a Detailed Analysis Report

In the Simulink Design Verifier log 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 1-15
- “Analysis Information” on page 1-16
- “Test Objectives Status” on page 1-17
- “Model Items” on page 1-20
- “Test Cases” on page 1-22

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

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

## Chapter 1. Summary

### Analysis Information

Model:	sldvdemo_cruise_control
Mode:	TestGeneration
Status:	Completed normally
Analysis Time:	9s

### Objectives Status

<b>Number of Objectives:</b>	<b>34</b>
Objectives Satisfied:	34

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

## Chapter 2. Analysis Information

### Table of Contents

[Model Information](#)

[Analysis Options](#)

[Constraints](#)

[Approximations](#)

### Model Information

File:	sldvdemo_cruise_control
Version:	1.51
Time Stamp:	Thu Sep 16 17:09:09 2010
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 Processing Time:	60s
Block Replacement:	off
Parameters Analysis:	on
Parameters Configuration File:	sldv_params_template.m
Save Data:	on
Save Harness:	off
Save Report:	off

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

## Chapter 3. Test Objectives Status

### Table of Contents

[Objectives Satisfied](#)

### Objectives Satisfied

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

#	Type	Model Item	Description	Test Case
1	Decision	<a href="#">Controller/Switch1</a>	logical trigger input false (output is from 3rd input port)	<a href="#">3</a>
2	Decision	<a href="#">Controller/Switch1</a>	logical trigger input true (output is from 1st input port)	<a href="#">1</a>
3	Condition	<a href="#">Controller/Logical Operator1</a>	Logic: input port 1 T	<a href="#">1</a>
4	Condition	<a href="#">Controller/Logical Operator1</a>	Logic: input port 1 F	<a href="#">2</a>
5	Condition	<a href="#">Controller/Logical Operator2</a>	Logic: input port 1 T	<a href="#">1</a>
6	Condition	<a href="#">Controller/Logical Operator2</a>	Logic: input port 1 F	<a href="#">3</a>
7	Condition	<a href="#">Controller/Logical Operator2</a>	Logic: input port 2 T	<a href="#">7</a>
8	Condition	<a href="#">Controller/Logical Operator2</a>	Logic: input port 2 F	<a href="#">3</a>
9	Mcdc	<a href="#">Controller/Logical Operator2</a>	Logic: MCDC expression for output with input port 1 T	<a href="#">1</a>

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 30, click the test case number (8) to display more information about test case 8 in the report's **Test Cases** chapter.

## Test Case 8

**Summary**

Length: 0.06 Seconds (7 sample periods)  
Objective Count: 1

**Objectives**

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

**Generated Input Data**

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

In this example, Test Case 8 satisfies one objective, that the integration result be greater than or equal to the upper limit T in the Discrete-Time Integrator block. The table lists the values of the six signals from time 0 through time 0.06.

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



## Chapter 4. Model Items

### Table of Contents

[Controller/Switch1](#)

[Controller/Logical Operator1](#)

[Controller/Logical Operator2](#)

[Controller/Logical Operator](#)

[Controller/PI Controller](#)

[Controller/PI Controller/Discrete-Time Integrator](#)

[Controller/Switch2](#)

[Controller/Switch3](#)

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

### Controller/Switch1

[View](#)

#:	Type	Description	Status	Test Case
1	Decision	logical trigger input false (output is from 3rd input port)	Satisfied	<a href="#">3</a>
2	Decision	logical trigger input true (output is from 1st input port)	Satisfied	<a href="#">1</a>

### Controller/Logical Operator1

[View](#)

#:	Type	Description	Status	Test Case
3	Condition	Logic: input port 1 T	Satisfied	<a href="#">1</a>
4	Condition	Logic: input port 1 F	Satisfied	<a href="#">2</a>

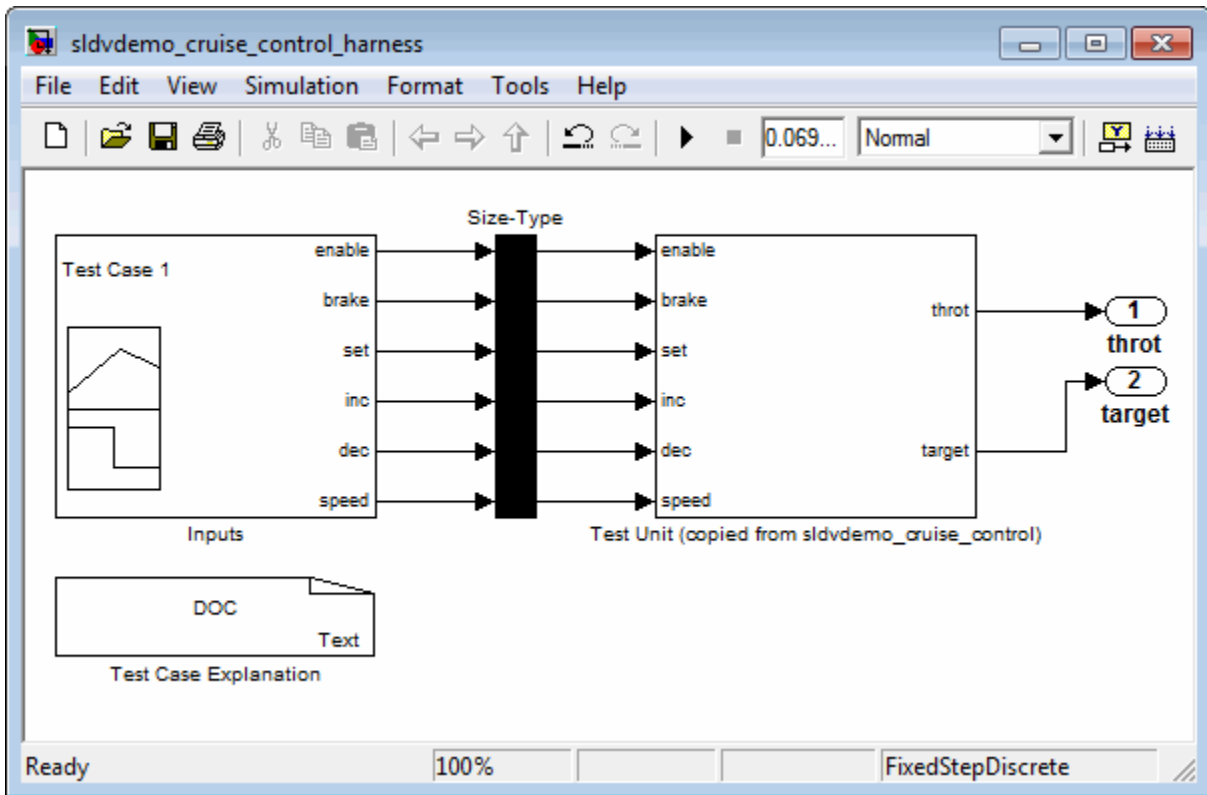
**Test Cases.** In the **Table of Contents**, click **Test Cases** to display detailed information about each generated test case, including:

- 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 8 in “Test Objectives Status” on page 1-17.

## **Creating a Harness Model**

In the Simulink Design Verifier log window, if you click **Create harness model**, the software creates and opens a harness model named `sldvdemo_cruise_control_harness`.



The harness model contains the following blocks:

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

The screenshot shows a text editor window with a menu bar (File, Edit, Text, Go, Tools, Debug, Desktop, Window, Help) and a toolbar. The main text area contains the following content:

```

1 Test Case 1 (8 Objectives)
2   Parameter values:
3
4   1. Controller/Switch1 - logical trigger input t
5   2. Controller/Logical Operator1 - Logic: input p
6   3. Controller/Logical Operator2 - Logic: input p
7   4. Controller/Logical Operator2 - Logic: MCDC es
8   5. Controller/Logical Operator - Logic: input p
9   6. Controller/Logical Operator - Logic: input p
10  7. Controller/Logical Operator - Logic: MCDC exp
11  8. Controller/PI Controller - enable logical val
12
13 Test Case 2 (3 Objectives)
14   Parameter values:
15
16  1. Controller/Logical Operator1 - Logic: input p

```

The status bar at the bottom indicates "plain text file", "Ln 1", "Col 1", and "OVR".

- 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.


---

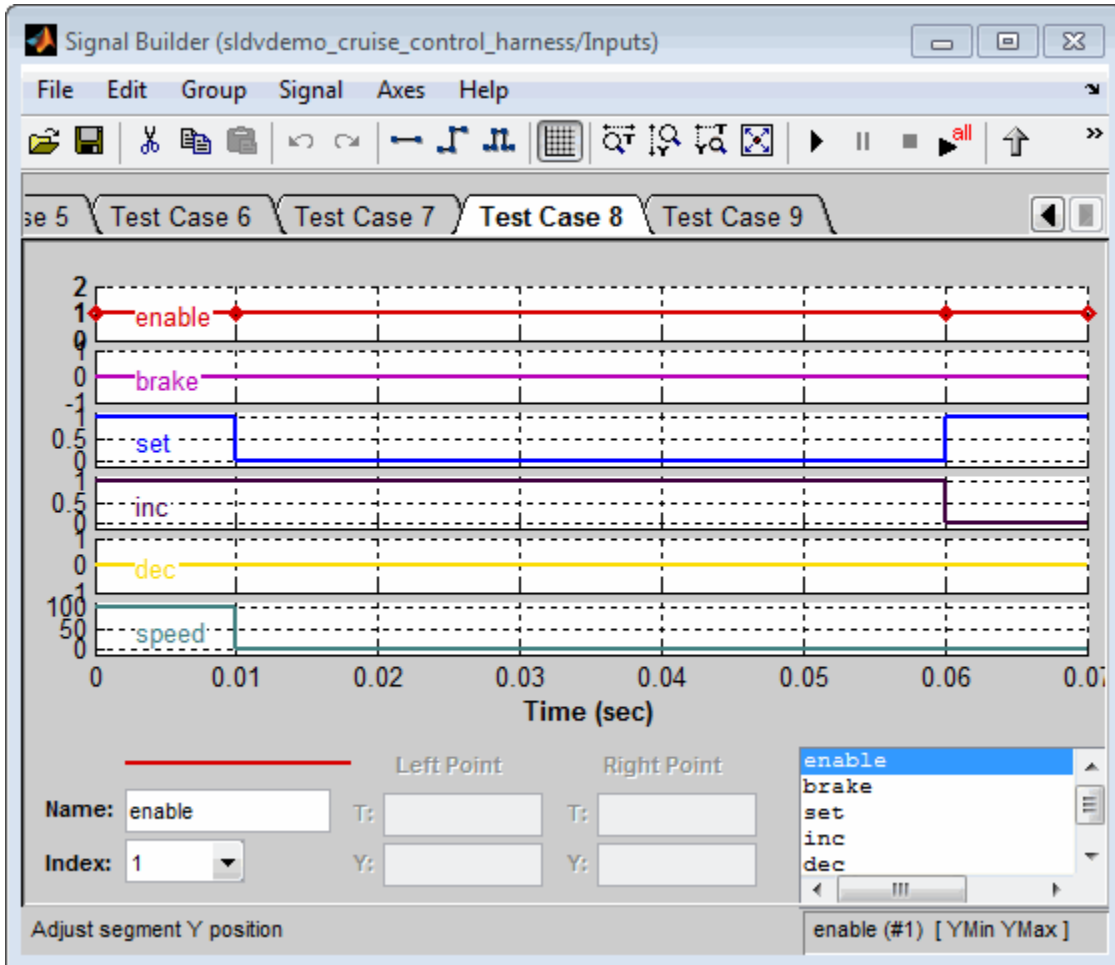
**Note** You can configure the harness model to reference the model that you are analyzing using a Model block instead of using a subsystem. In the Configuration Parameters dialog box, on the **Design Verifier > Results** pane, select **Save test harness as model** and **Reference input model in generated 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 10 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 nine test cases. To look at Test Case 8:


- 1** Click the right-facing arrow next to the test case tabs  to find the **Test Case 8** tab.
- 2** Click the **Test Case 8** tab to display the signal values for Test Case 8.













In Test Case 8 at 0.01 seconds:

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

In the Signal Builder block, the signal group satisfies the test objectives described in the Test Case Explanation block.

- To confirm that the Simulink Design Verifier software achieved complete model coverage, simulate the harness model using all the test cases. In the Signal Builder dialog box, click the **Run all and produce coverage** button .

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

Model Hierarchy/Complexity:		Test 1						
		D1	C1	MCDC				
1.	<a href="#">sldvdemo_cruise_control_harness</a>	8	100%		100%		100%	
2. . . .	<a href="#">Test Unit (copied from sldvdemo_cruise_control)</a>	7	100%		100%		100%	
3. . . . .	<a href="#">Controller</a>	7	100%		100%		100%	
4. . . . . .	<a href="#">PI Controller</a>	4	100%		NA		NA	

## Simulating Tests and Producing a Model Coverage Report

In the Simulink Design Verifier log 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_MATLAB_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.

## Combining Test Cases

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

Open the `sldvdemo_cruise_control` model and rerun the analysis with the Long test cases optimization:

- 1** Select **Tools > Design Verifier > Options**.
- 2** In the Configuration Parameters dialog box, in the **Select** tree on the left side, under the **Design Verifier** category, select **Test Generation**.
- 3** Set the **Test suite optimization** parameter to `LongTestcases`.
- 4** Click **Apply** and **OK** to close the Configuration Parameters dialog box.
- 5** In the `sldvdemo_cruise_control` model, double-click the block labeled **Run**.
- 6** In the log window, click **Create harness model**.

In the harness model, the Signal Builder dialog box now contains two longer test cases instead of the nine shorter test cases created in “Analyzing a Model” on page 1-6.

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

The analysis still satisfies all 34 objectives.



## Analyzing a Subsystem

In addition to analyzing a model, 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 analyzes the Controller subsystem in the `sldvdemo_cruise_control` model.

- 1 Open the demo model:

```
sldvdemo_cruise_control
```

- 2 Right-click the Controller subsystem, and select **Design Verifier > Enable “Treat as atomic unit” to analyze**.

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

- 3 Select **Treat as atomic unit**.

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

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

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

- 4 Click **OK** to close the dialog box.
- 5 Select **File > Save As** and save the Cruise Control Test Generation model under a new name.
- 6 To start the subsystem analysis and generate test cases, right-click the Controller subsystem, and select **Design Verifier > Generate Tests for Subsystem**.

- 7** The Simulink Design Verifier software analyzes the subsystem. When the analysis is complete, view the analysis results for the Controller subsystem by clicking one of the following options:
  - **Highlight analysis results on model**
  - **Generate detailed analysis report**
  - **Create harness model**
  - **Simulate tests and produce a model coverage report**
- 8** Review the results of the subsystem analysis and compare them to the results of the full-model analysis described in “Analyzing a Model” on page 1-6:
  - 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, so both analyses generate the same test cases.

## Analyzing 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 the Simulink Design Verifier software 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 “Analyzing the Model Using a Bottom-Up Approach” on page 14-15, by analyzing atomic subcharts or other components in the model hierarchy individually, you can analyze a model to:

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

---

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

---

### Example: Analyzing an Atomic Subchart Using the Simulink Design Verifier Software

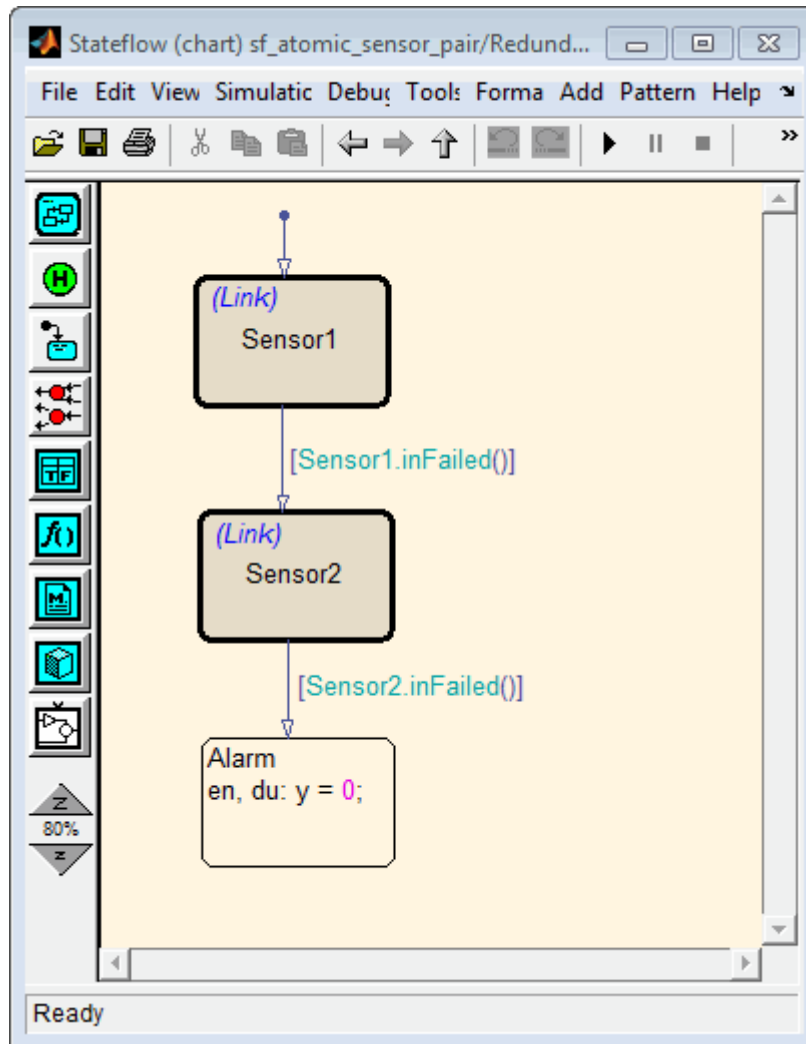
The `sf_atomic_sensor_pair` demo model models a redundant sensor pair using atomic subcharts. This example analyzes the `Sensor1` subchart in the `RedundantSensors` chart.

- 1 Open the `sf_atomic_sensor_pair` demo 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.

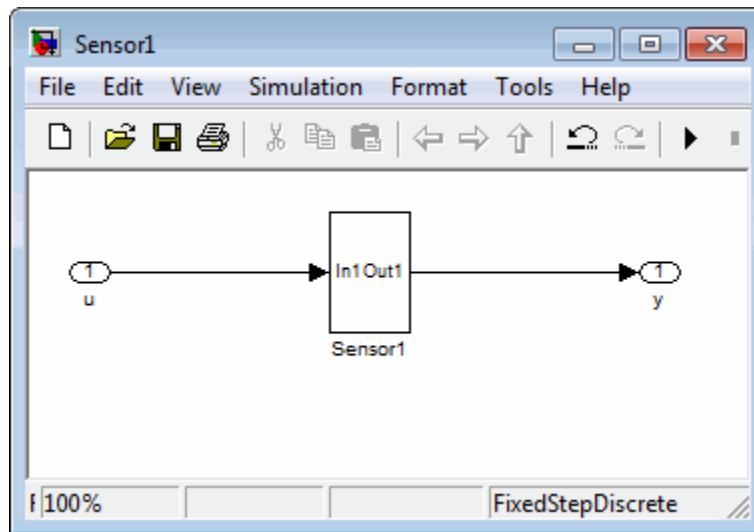


This Stateflow chart has two atomic subcharts:

- Sensor1
- Sensor2

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

During the analysis, the software creates a Simulink model named **Sensor1.mdl** 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:

`matlabroot/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 Using the Simulink Design Verifier Software

The basic workflow for analyzing your model is described in the following steps. For more information, see the cited sections and chapters in the *Simulink Design Verifier User's Guide*.

Step	Action	See...
1	Check the compatibility of your model.	Chapter 3, “Ensuring Compatibility with the Simulink® Design Verifier™ Software”
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.	<ul style="list-style-type: none"> <li>• Chapter 4, “Working with Block Replacements”</li> <li>• Chapter 5, “Specifying Parameter Configurations”</li> </ul>
3	Set Simulink Design Verifier options.	Chapter 15, “Simulink® Design Verifier™ Configuration Parameters”
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.	Chapter 6, “Detecting Design Errors”
5	Analyze your model to: <ul style="list-style-type: none"> <li>• Detect design errors</li> <li>• Generate test cases</li> <li>• Prove properties</li> </ul>	<ul style="list-style-type: none"> <li>• Chapter 6, “Detecting Design Errors”</li> <li>• Chapter 7, “Generating Test Cases”</li> <li>• Chapter 12, “Proving Properties of a Model”</li> </ul>
6	Generate the results.	“Generating Analysis Results” on page 1-10
7	Interpret the results.	Chapter 13, “Reviewing the Results”

## Learning More

In this section...
“Next Step” on page 1-35
“Product Help” on page 1-36
“MathWorks Online” on page 1-36


### Next Step

To begin learning how to use the Simulink Design Verifier software, see Chapter 3, “Ensuring Compatibility with the Simulink® Design Verifier™ Software”. Also see the following topics to continue your exploration of the software:

To...	See...
Detect design errors	Chapter 6, “Detecting Design Errors”
Generate test cases	<ul style="list-style-type: none"> <li>• “Generating Test Cases to Achieve Decision Coverage for a Model” on page 7-5</li> <li>• “Generating Test Cases for a Subsystem” on page 7-23</li> </ul>
Prove properties	<ul style="list-style-type: none"> <li>• “Proving Properties in a Model” on page 12-5</li> <li>• “Proving Properties in a Subsystem” on page 12-32</li> </ul>
Extend existing test cases for a model	<p>“Example: Extending Existing Test Cases for a Model that Uses Temporal Logic” on page 8-4</p> <p>“Example: Extending Existing Test Cases for a Closed-Loop System” on page 8-11</p> <p>“Example: Extending Existing Test Cases for a Modified Model” on page 8-14</p>

To...	See...
Generate test cases for missing coverage	“Example: Achieving Missing Coverage in a Referenced Model” on page 9-3 “Example: Achieving Missing Coverage in a Closed-Loop Simulation Model” on page 9-8
Verify individual components in a model	“Example: Verifying a Component for Code Generation” on page 10-6

## Product Help

In the MATLAB desktop, click  for help. In the **Contents** pane, click the product name.

For...	See...
List of functions	“Functions — Alphabetical List”
List of blocks	Blocks — Alphabetical List
Tutorials	Examples in Documentation
More product demonstrations	Simulink Design Verifier Demos
What’s new in this product	Release Notes

## MathWorks Online

For addition information and support, go to the MathWorks® Web site:

<http://www.mathworks.com/products/sldesignverifier/>



# How the Simulink Design Verifier Software Works

---

- “Analyzing a Model Using the Simulink® Design Verifier™ Software” on page 2-2
- “Analyzing a Simple Model” on page 2-3
- “Analyzing Model Blocks” on page 2-6
- “Block Reduction” on page 2-7
- “Inline Parameters” on page 2-9
- “Analyzing Large Models” on page 2-10
- “Handling Incompatibilities with Automatic Stubbing” on page 2-11
- “Handling Nonfinite Data” on page 2-19
- “Approximations” on page 2-20
- “Short-Circuiting Logic Blocks” on page 2-23

# Analyzing a Model Using the Simulink Design Verifier Software

Simulink Design Verifier software is an efficient analysis tool that explores the simulation behavior of a Simulink model in three ways:

- Identifies design errors that cause data overflow or division-by-zero errors.
- Searches the possible values of model inputs and block parameters to find a simulation that satisfies test objectives.
- Proves model properties and generates examples of property violations.

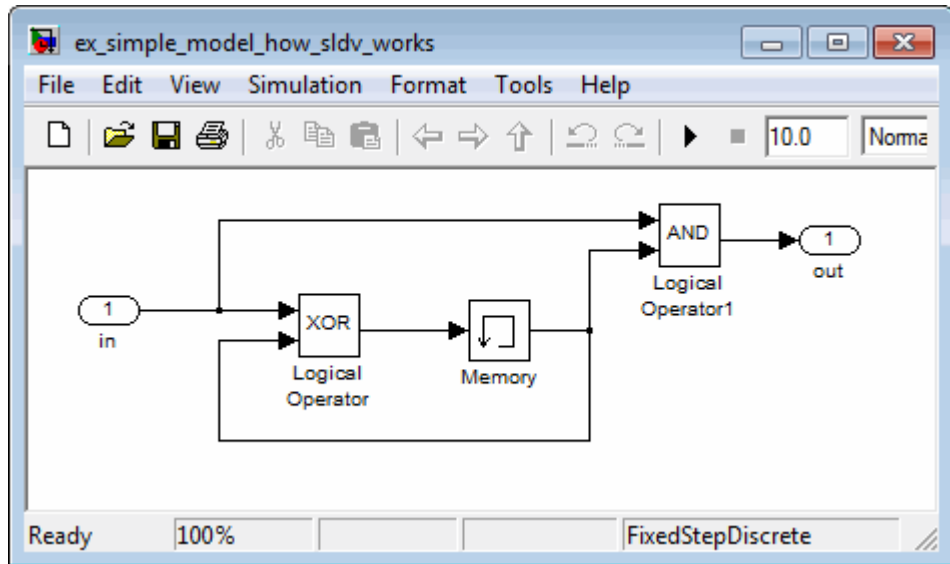
The Simulink Design Verifier analysis always begins with the initial configuration of the model and can span an arbitrary number of time steps. Generally, there is an infinite number of paths through the model because the values of inputs are independent from one time step to the next, and there is no fixed limit to the number of time steps.

If the software cannot find a way to reduce the search space, it might continue its analysis indefinitely. The software limits the analysis by tracking the persistent information in the model such as discrete states, data-store memories, and persistent variables.

After an analysis explores all possible inputs and parameters from all possible configurations, the results are equivalent to those of a complete search of every possible infinite sequence of input parameters.

## Analyzing a Simple Model

This simple Simulink 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 would result 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

Suppose you want to generate test cases that result in a true output; this goal is your *test objective*. If you run the Simulink Design Verifier software to generate test cases that result in a true output, the software searches this table to see if such a scenario is possible.

After the Simulink Design Verifier software discovers a configuration that satisfies the test objective (in this case, when both the input and the Memory block output are true), it needs to find a path to reach this configuration from the initial conditions. If the initial memory value is true, the test case only needs to be a single time step (row 4) where the input was true.

If the initial memory value is false (the default), the test case must force the memory value to be true. In this example, the path requires two 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, so the analysis achieves the specified 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 the Simulink Design Verifier software 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 the Simulink Design Verifier software 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 the Simulink Design Verifier software 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, the Simulink Design Verifier software 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.

### Analyzing Model Blocks

If your model contains Model blocks that reference external models, the Simulink Design Verifier software creates test cases for the top-level model, considering each referenced model in its execution context.

If you have multiple Model blocks that reference the same model, the software analyzes each instance of the referenced model in the context from which it is referenced. The software attempts to satisfy test objectives for each instance within its execution 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 simulate the model using the test cases that the analysis generates, and collect coverage, the Simulink Verification and Validation software combines the coverage data for 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.

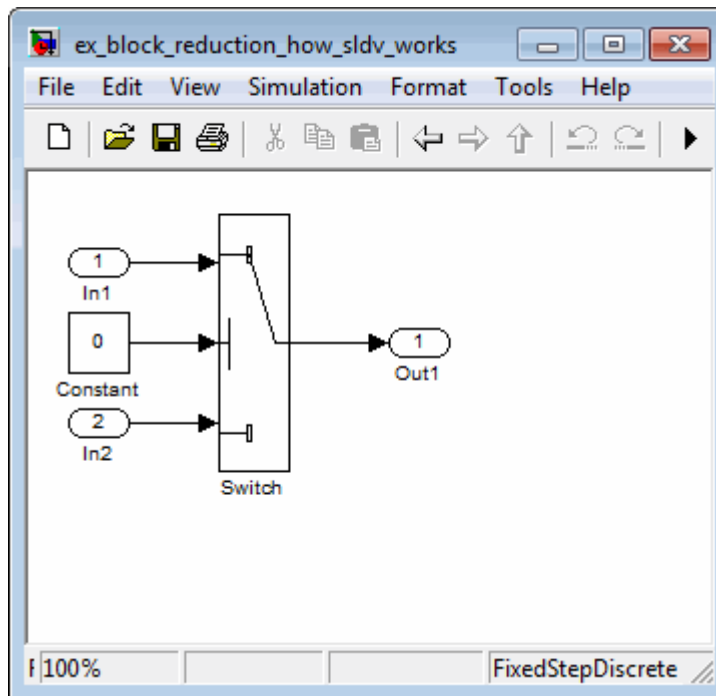
For example, suppose you have three Model blocks that reference the same model. Suppose also that referenced model has three test objectives. When you run a Simulink Design Verifier analysis for the top-level model, the total number of test objectives for the three Model blocks is nine. If you then simulate the model with the test cases generated by the analysis, the coverage results for that referenced model specify three test objectives.

## Block Reduction

Block reduction is a Simulink feature that lets you achieve faster execution during model simulation and in generated code. When block reduction is enabled, the Simulink software collapses certain groups of blocks into a single, more efficient block, or removes them entirely.

When the Simulink Design Verifier software translates a model, block reduction happens automatically, and blocks in unused code paths are eliminated from the model. The 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  $u2 \neq 0$ , the Switch block always passes the third input through to the output port. When you analyze this

model, the Simulink Design Verifier software removes the Switch block from the model and does not report any test objectives for the Switch block.

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



## Inline Parameters

The **Inline parameters** parameter is a Simulink optimization that transforms tunable parameters into constant values. For example, suppose you have a Gain block whose **Gain** parameter is **a**, where **a** is defined in the model workspace. During simulation, Simulink converts that **Gain** parameter to a constant value, as defined in the workspace.

When the Simulink Design Verifier software translates a model, it transforms all tunable parameters in the model into constant values, even if the **Inline parameters** option is off.

To tune parameters during an analysis, define parameter values in a parameter configuration file and use the **Design Verifier > Parameters** pane options to apply those parameters 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 Chapter 5, “Specifying Parameter Configurations”.

## Analyzing Large Models

In larger, more complicated models, the Simulink Design Verifier software 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 through the logical values that describe your model.

For detailed information about analyzing large models, see Chapter 14, “Analyzing Large Models and Improving Performance”.

# Handling Incompatibilities with Automatic Stubbing

## In this section...

“What Is Automatic Stubbing?” on page 2-11

“How Automatic Stubbing Works” on page 2-11

“Analyzing a Model Using Automatic Stubbing” on page 2-14

## What Is Automatic Stubbing?

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

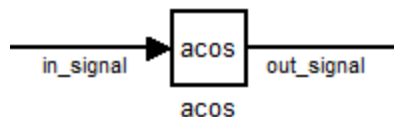
When you enable the automatic stubbing option (it is enabled by default), the software considers only the interface of the unsupported objects, not their actual behavior. This technique allows the software to complete the analysis. However, the analysis may achieve only partial results if any unsupported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

## Stubbing Example: Trigonometric Function Block

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

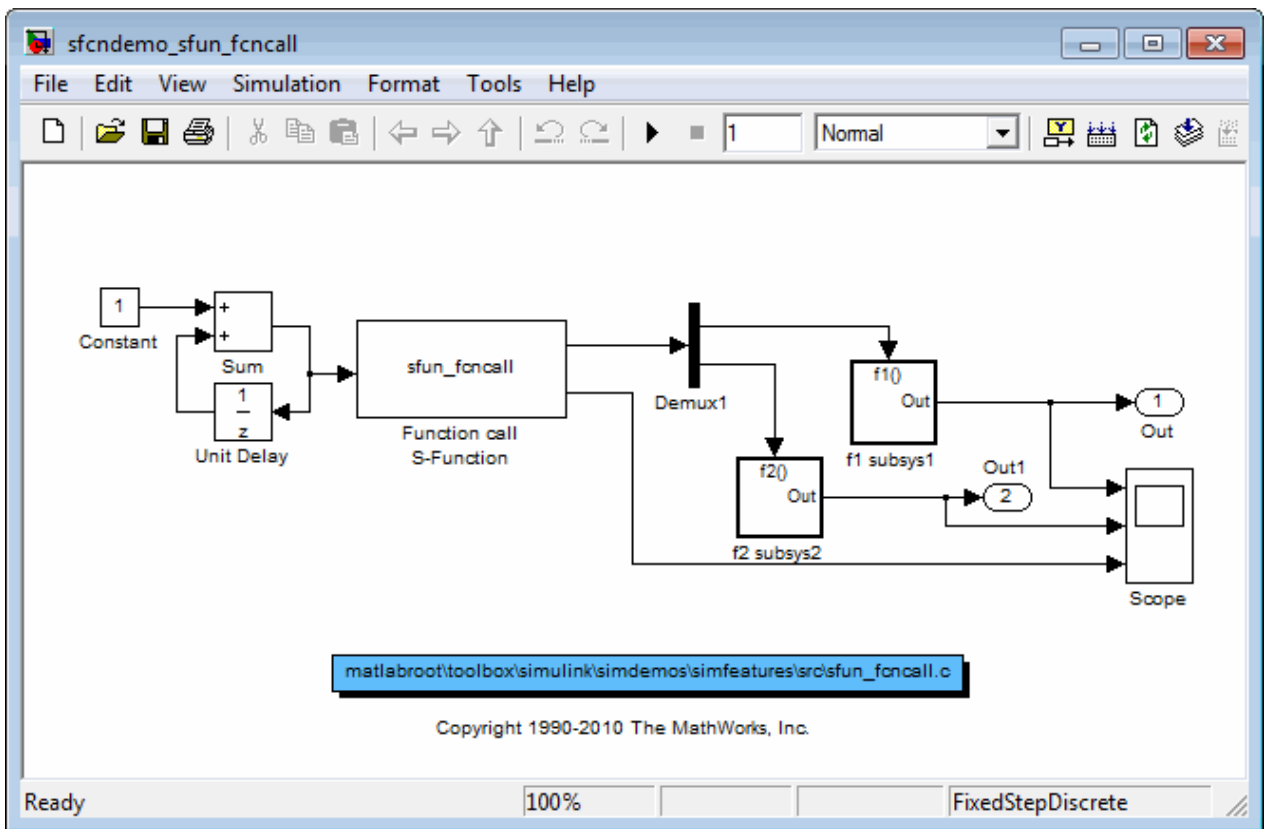


When stubbing this block during analysis, `out_signal` can take any value, with the following results.

Analysis Model	Result of Stubbing <code>out_signal</code>
Design error detection	<ul style="list-style-type: none"><li>• If a design-error objective that depends on <code>out_signal</code> is proven valid, that objective is valid for all simulations. In this case, the stubbing did not affect the results of the analysis.</li><li>• If a design-error objective that depends on <code>out_signal</code> 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.</li></ul>
Test case generation	<ul style="list-style-type: none"><li>• If a test objective that depends on the value of <code>out_signal</code> 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.</li><li>• If a test objective that depends on the value of <code>out_signal</code> is unsatisfiable, there is no simulation that can satisfy that objective. In this case, the stubbing did not affect the results of the analysis.</li></ul>
Property proving	<ul style="list-style-type: none"><li>• If a proof objective that depends on <code>out_signal</code> is proven valid, that objective is valid for all simulations. In this case, the stubbing did not affect the results of the analysis.</li><li>• If a proof objective that depends on <code>out_signal</code> 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.</li></ul>

## Stubbing Example: S-Function Blocks and Function-Call Triggers

The Simulink demo model `sfcndemo_sfun_fcncall` has an S-Function block. The S-function `sfun_fcncall` triggers the execution of the function-call subsystems `f1 subsystem1` and `f2 subsystem2` on the first and second elements of the first output port.



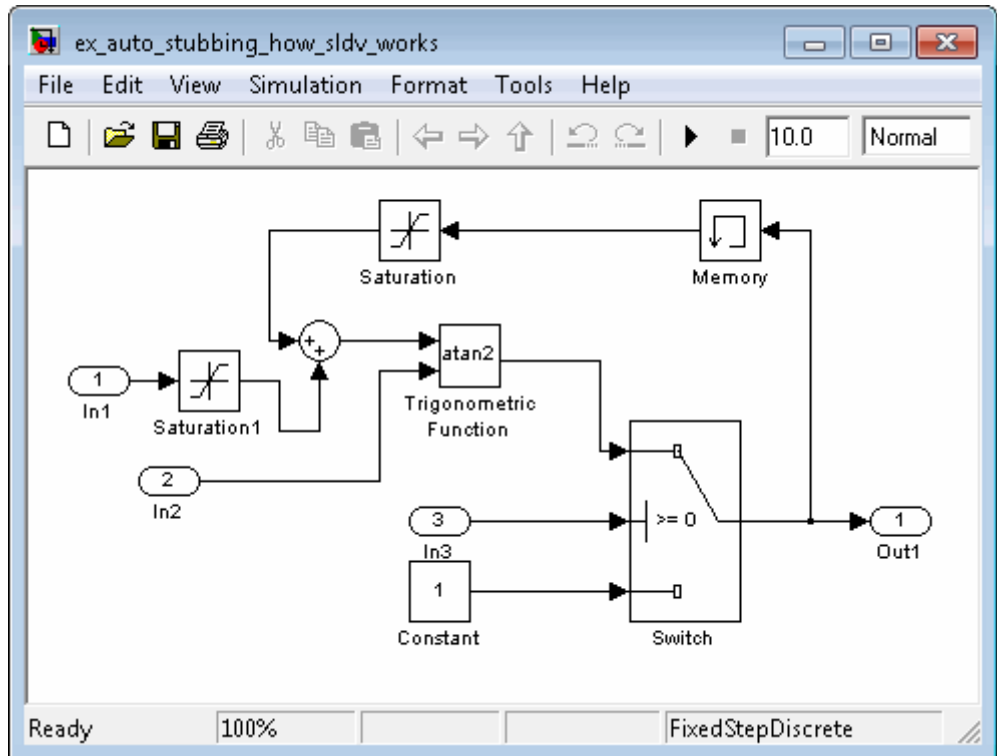
The Simulink Design Verifier software does not support the S-Function block, so if automatic stubbing is enabled, the analysis ignores the behavior of the S-function. As a result, the code that triggers the two function-call subsystems is ignored, resulting in two unsatisfiable objectives. Since the function calls are ignored, the contents of those subsystems are effectively eliminated from the analysis.

## Analyzing a Model Using Automatic Stubbing

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

- “Checking Model Compatibility” on page 2-15
- “Turning On Automatic Stubbing” on page 2-16
- “Reviewing the Results” on page 2-17
- “Achieving Complete Results” on page 2-18

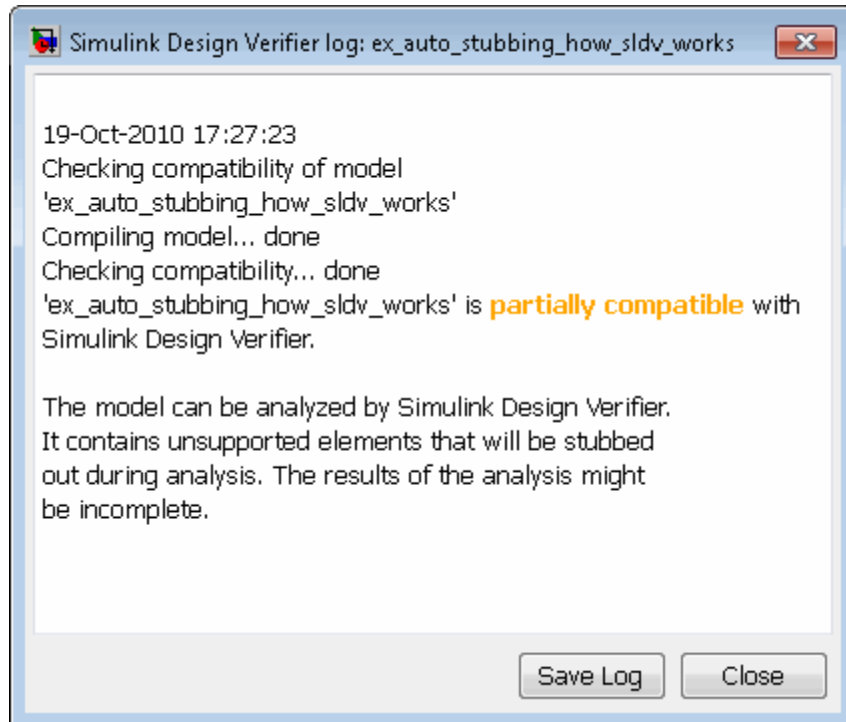
The following model contains a Trigonometric Function block that is not compatible with the Simulink Design Verifier software.



## Checking Model Compatibility

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

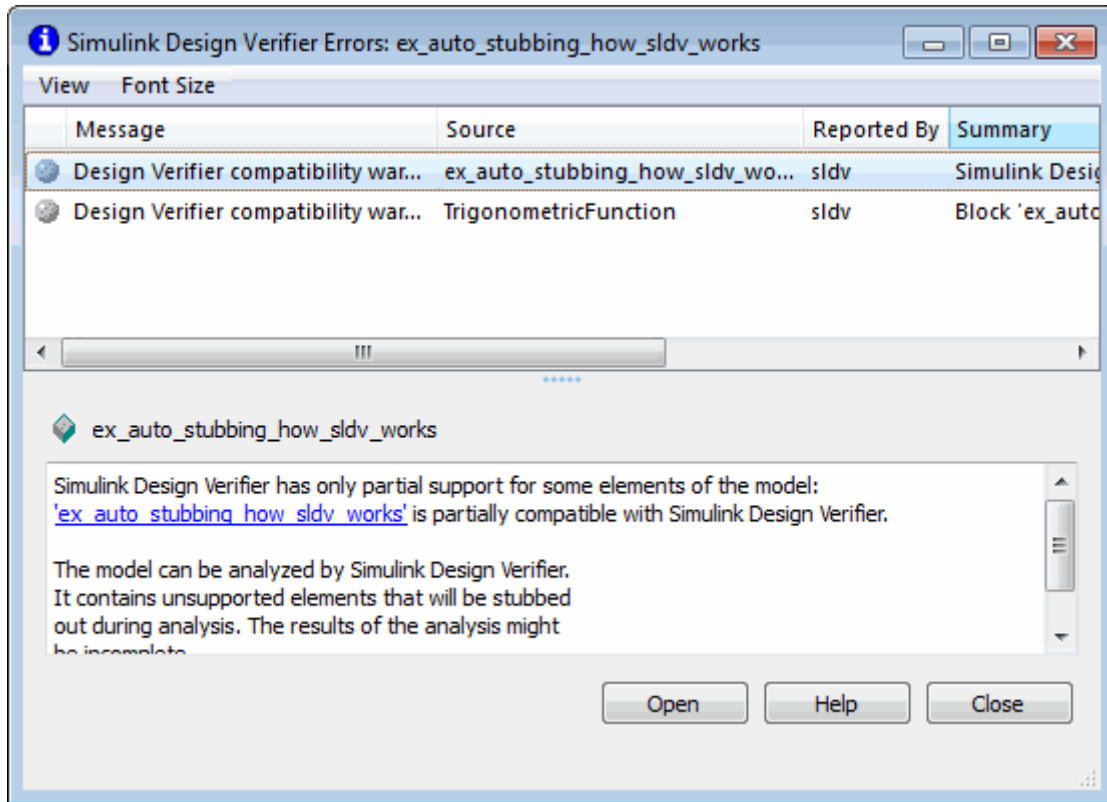
- Run the Simulink Design Verifier compatibility check by selecting **Tools > Design Verifier > Check Model Compatibility**.



- Select the analysis that you want:
  - **Tools > Design Verifier > Detect Design Errors**
  - **Tools > Design Verifier > Generate Tests**
  - **Tools > Design Verifier > 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 Simulation Diagnostics Viewer also opens, listing the incompatibilities.



---

**Note** For more information, see “Simulation Diagnostics Viewer” in the Simulink documentation.

---

### Turning 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.

## Reviewing the Results

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

### Unsupported Blocks

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

Block	Type
<a href="#">Trigonometric Function</a>	Trigonometry

The Summary report for the example model shows that one objective was satisfied without generating a test case. The software cannot generate the test case because it does not understand the operation of the Trigonometric Function block.

### Chapter 1. Summary

#### Analysis Information

Model:	ex_auto_stubbing_how_sldv_works
Mode:	TestGeneration
Status:	Completed normally
Analysis Time:	1s

#### Objectives Status

<b>Number of Objectives:</b>	<b>10</b>
Objectives Satisfied:	9
Objectives Satisfied - No Test Case:	1

### Achieving Complete Results

If your analysis does not achieve complete results because of the stubbing, you can define custom block replacements to give a more precise definition of the unsupported blocks. For more information:

- See “Defining Custom Block Replacements” on page 4-8.
- At the MATLAB command line, enter  

```
echodemo sldvdemo_blockreplacement_unsupportedblocks
```

to step through the “Block Replacements for Unsupported Blocks” demo.

## Handling Nonfinite Data

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

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

- In the Relational Operator block:
  - If the **Operator** parameter is `isFinite`, the output is always 1.
  - If the **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” on page 2-20
“Types of Approximations” on page 2-20
“Converting Floating-Point Arithmetic to Rational-Number Arithmetic” on page 2-21
“Linearizing Two-Dimensional Lookup Tables” on page 2-21
“Unrolling While Loops” on page 2-22
“Ensuring the Validity of the Analysis” on page 2-22

### 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.

Approximations		
<p>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.</p>		
	Type	Description
1	Rational approximation	The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.

### Types of Approximations

The Simulink Design Verifier software performs three types of approximations when it analyzes a model:

- “Converting Floating-Point Arithmetic to Rational-Number Arithmetic” on page 2-21
- “Linearizing Two-Dimensional Lookup Tables” on page 2-21
- “Unrolling While Loops” on page 2-22

## Converting Floating-Point Arithmetic to Rational-Number Arithmetic

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, the Simulink Design Verifier software converts those values to rational numbers before performing its analysis.

---

**Note** As a result of these approximations, Simulink Design Verifier software does not consider the effect of round-off error, or the upper and lower bounds of floating-point numbers.

---

## Linearizing Two-Dimensional Lookup Tables

The Simulink Design Verifier software does not support nonlinear arithmetic. 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, $n = 2$ :	<ul style="list-style-type: none"> <li>• <b>Interpolation method</b> parameter is Linear.</li> <li>• <b>Extrapolation method</b> parameter is Clip or Linear.</li> <li>• The input and output signals both have the floating-point data type.</li> </ul>

### **Unrolling While Loops**

If your model or any Stateflow chart in your model contains a `while` loop, the Simulink Design Verifier software tries to find a bound that allows the `while` loop to exit. To find a bound, it unrolls the `while` loop and executes it three times. If the software does not find a bound for a test case generation analysis, it sets the number of loop iterations to 3 for the purpose of the analysis. If you are performing a design-error detection or property-proving analysis, the analysis terminates.

### **Ensuring the Validity of the Analysis**

The Simulink Design Verifier software records all approximations it performed in the Analysis Information chapter of the HTML report. (For a description of this chapter, see “Analysis Information Chapter” on page 13-28.)

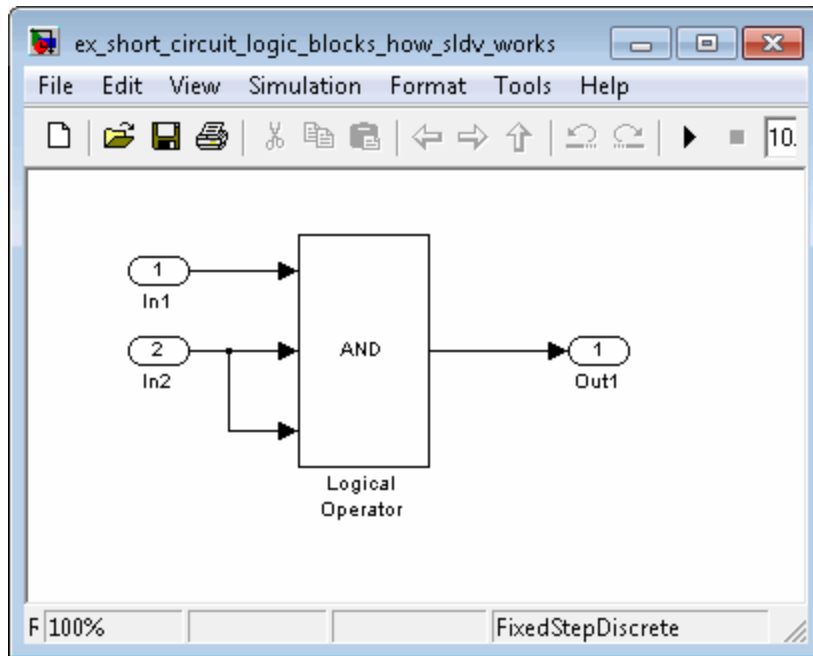
Review the analysis results carefully when the software uses approximations. Evaluate your model to identify which blocks or subsystems caused the software to perform the approximations.

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

## Short-Circuiting Logic Blocks

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

Consider the following example model, with the **Model coverage objectives** parameter set to Condition Decision.



When the Simulink Design Verifier software 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.





# Ensuring Compatibility with the Simulink Design Verifier Software

---

- “Checking Model Compatibility” on page 3-2
- “Unsupported Simulink Software Features” on page 3-9
- “Simulink Block Support Limitations” on page 3-12
- “Limitations of Support for Model Blocks” on page 3-13
- “Unsupported Stateflow Software Features” on page 3-15
- “Support Limitations for MATLAB for Code Generation” on page 3-20
- “Fixed-Point Support Limitations” on page 3-22

# Checking Model Compatibility

The Simulink Design Verifier software analyzes Simulink models in order to:

- Detect design errors that may occur at runtime
- Generate test cases that achieve model coverage
- Prove properties and identify property violations

For these analyses, the models must:

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

The software supports a broad range of Simulink and Stateflow software features in your models. However, there are features that the product does not support, as described in the following sections. Avoid using these particular features in models that you plan to analyze with the Simulink Design Verifier software.

The software automatically checks the compatibility of your model before it begins an analysis.

In addition, you can run a compatibility check before you start the analysis. To run this check, in the model window, select **Tools > Design Verifier > Check Model Compatibility**.

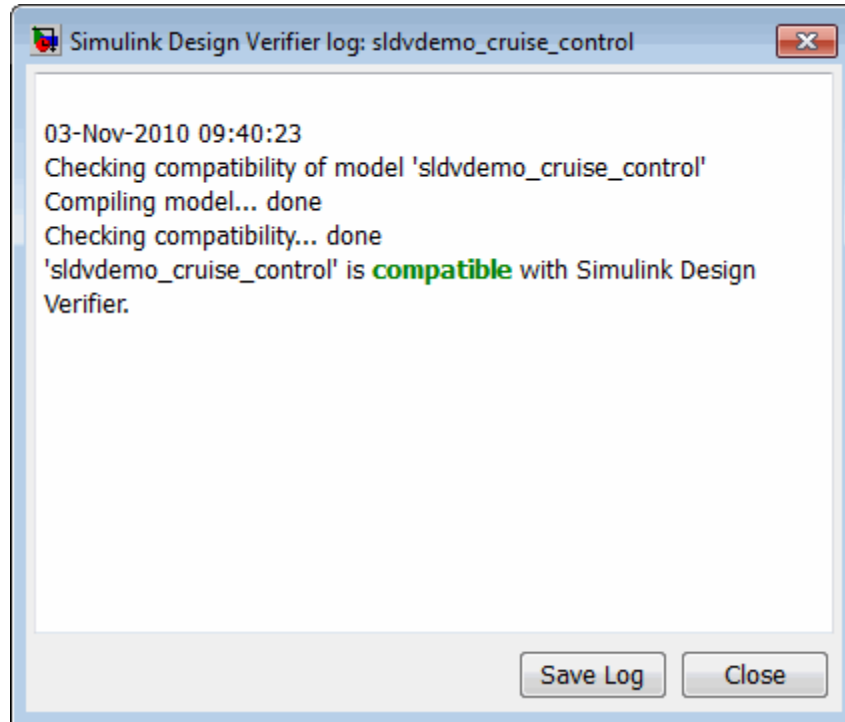
Alternatively, you can use the `sldvcompat` function to run the compatibility checker programmatically at the command line or in a MATLAB program. For more information, see the `sldvcompat` reference page.

There are three outcomes of a compatibility check:

- “Model Is Compatible” on page 3-3
- “Model Is Incompatible” on page 3-3
- “Model is Partially Compatible” on page 3-6

## Model Is Compatible

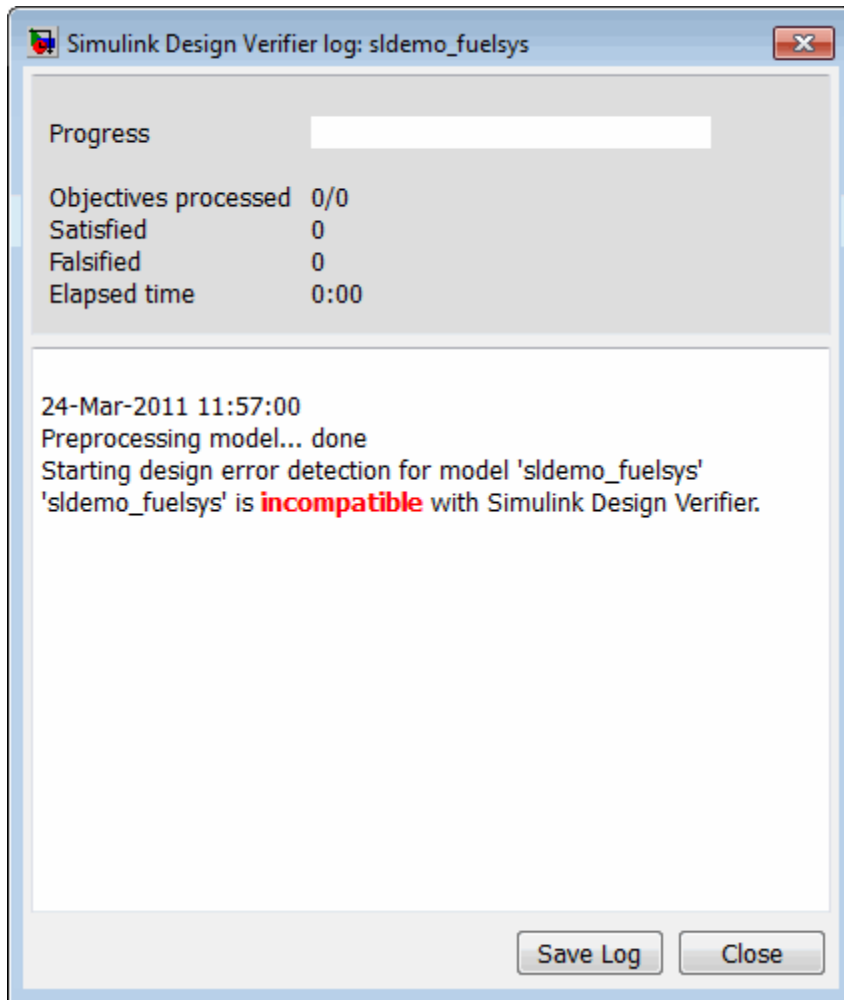
In the log window, you see if your model is compatible with the software.



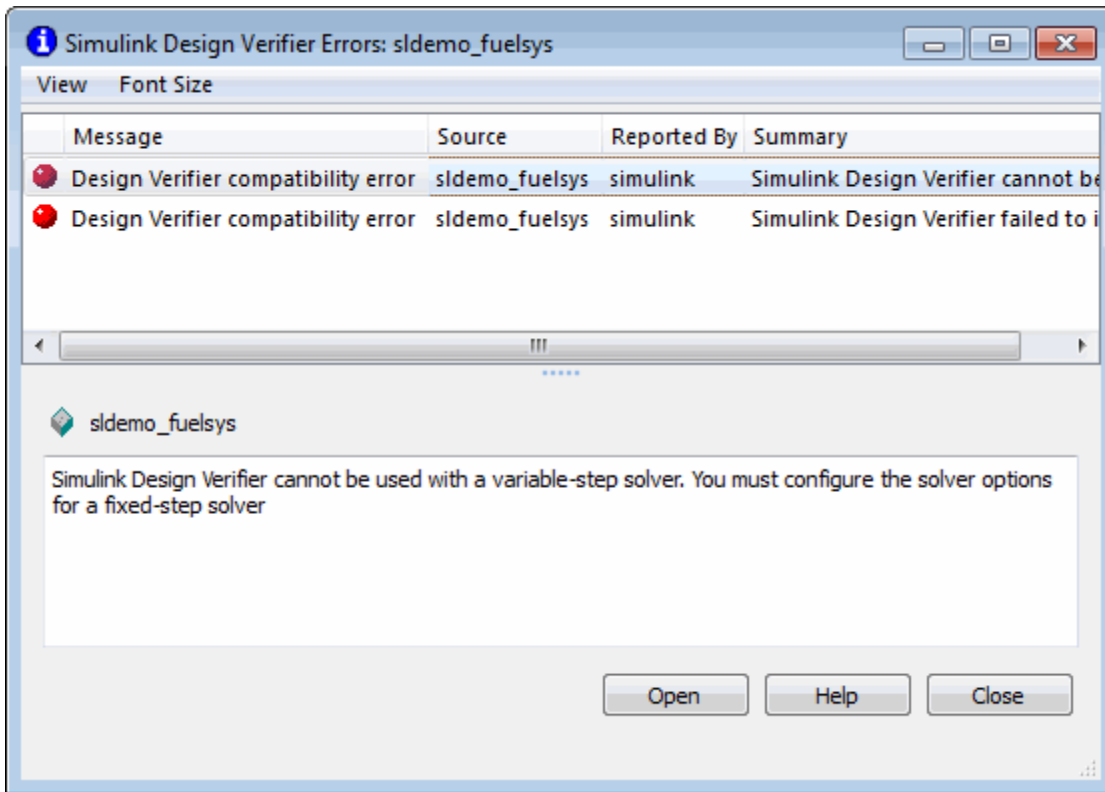
## Model Is Incompatible

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

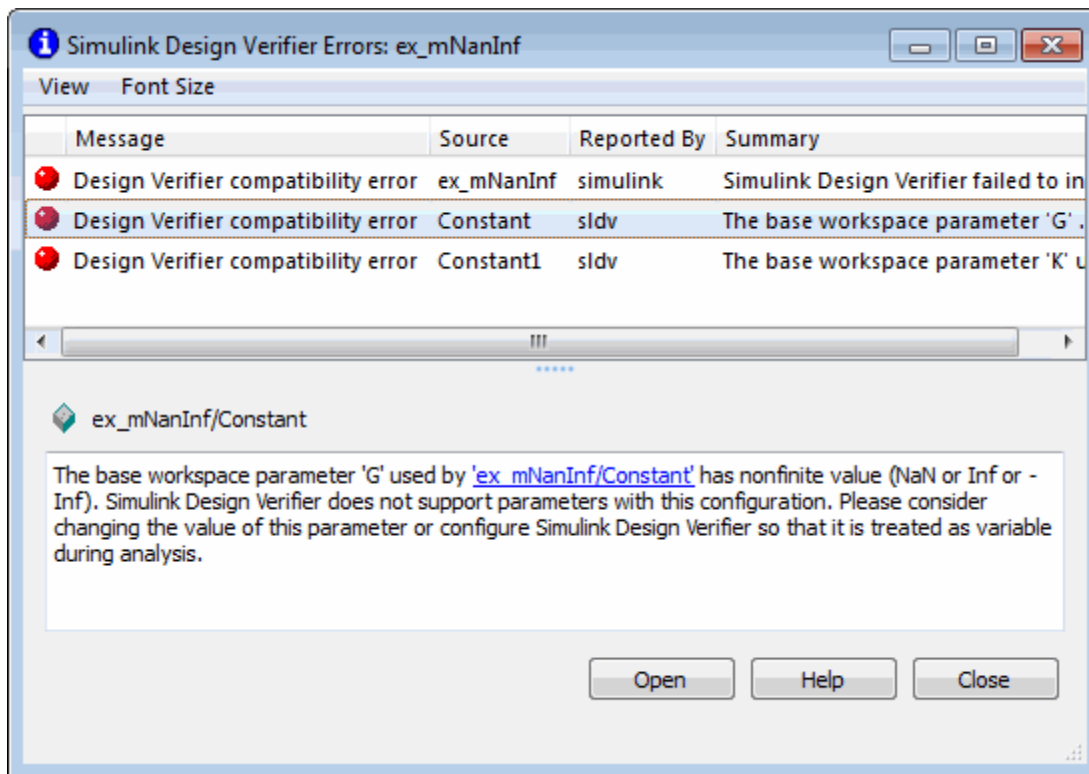
- Simulink Design Verifier log



- Simulation Diagnostics Viewer. Use the information in this dialog box to identify and fix the incompatibility.
  - If your model uses a variable-step solver, configure the solver options to a fixed-step.



- If your model has nonfinite data, change the value of the data or configure the model so that the data is treated as a variable during Simulink Design Verifier analysis.



---

**Note** For more information about this dialog box, see “Simulation Diagnostics Viewer” in the Simulink documentation.

---

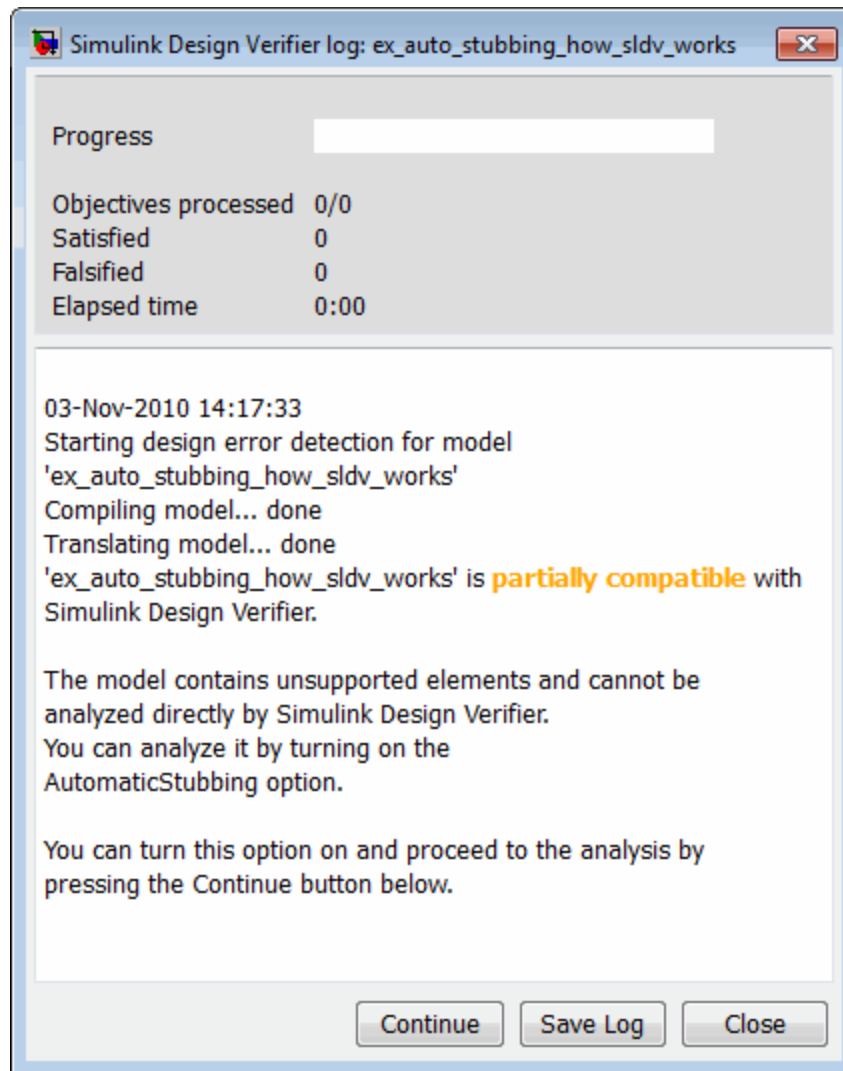
## Model is Partially Compatible

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

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

be incomplete.

If you have disabled automatic stubbing, the analysis stops. A query asks if you want to enable automatic stubbing so that the analysis can proceed.



Click **Continue** to enable automatic stubbing and proceed with the analysis.

---

**Note** For instructions on how to use automatic stubbing, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

---



## Unsupported Simulink Software Features

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

Not Supported	Description
Variable-step solvers	<p>The software supports only fixed-step solvers.</p> <p>For more information, see “Choosing a Fixed-Step Solver” in the Simulink documentation.</p>
Callback functions	<p>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.</p> <ul style="list-style-type: none"> <li>• 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.</li> <li>• Changing the storage class of base workspace variables on model callback functions or mask initializations is not supported.</li> <li>• Callback functions called prior to analysis, such as the <code>PreLoadFcn</code> or <code>PostLoadFcn</code> model callbacks, are fully supported.</li> </ul>
Model callback functions	<p>The software only supports model callback functions if the <code>InitFcn</code> callback of the model is empty.</p>
Algebraic loops	<p>The software does not support models that contain algebraic loops.</p> <p>For more information, see “Algebraic Loops” in the Simulink documentation.</p>
Masked subsystem initialization functions	<p>The software does not support models whose masked subsystem initialization modifies any attribute of any workspace parameter.</p>

Not Supported	Description
Complex signals	<p>The software supports only real signals.</p> <p>For more information, see “Complex Signals” in the Simulink documentation.</p>
Variable-size signals	<p>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.</p> <p>For more information, see “Working with Variable-Size Signals” in the Simulink documentation.</p>
Arrays of buses	<p>The software does not support arrays of buses.</p> <p>For more information, see “Combining Buses into an Array of Buses” in the Simulink documentation.</p>
Multiword fixed-point data types	<p>The software does not support multiword fixed-point data types.</p>
Nonzero start times	<p>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:</p> <ul style="list-style-type: none"> <li>• If you do not select the <b>Reference input model in generated harness</b> 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.</li> <li>• If you select the <b>Reference input model in generated harness</b> 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 <b>Start time</b> parameter to 0.</li> </ul>

Not Supported	Description
Nonfinite data	<p>The software does not support nonfinite data (for example, NaN and Inf) and related operations.</p> <p>In the Relational Operator block, the software assigns the output as follows:</p> <ul style="list-style-type: none"> <li>• If the <b>Operator</b> parameter is <code>isFinite</code>, the output is always 1.</li> <li>• If the <b>Operator</b> parameter is <code>isNan</code> or <code>isInf</code>, the output is always 0.</li> </ul> <p>In the MATLAB Function block, the software assigns the return value as follows:</p> <ul style="list-style-type: none"> <li>• For the <code>isFinite</code> function, the output is always 1.</li> <li>• For the <code>isNan</code> and <code>isInf</code> functions, the output is always 0.</li> </ul>
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.

## Simulink Block Support Limitations

The software 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. However, if any of the unsupported blocks affect the simulation outcome, the analysis may achieve only partial results. For details about automatic stubbing, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

To achieve 100% coverage, avoid using unsupported blocks in models that you analyze.

Similarly, specify only the block parameters that the software recognizes for blocks that it partially supports. See Chapter 16, “Simulink Block Support”.

## Limitations of Support for Model Blocks

The software supports the Model block, but with the following limitations. The software cannot analyze a model that contains one or more Model blocks if:

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

---

**Note** For more information, see “Protecting Referenced Models” in the Simulink documentation.

---

- The parent model or any of the referenced models gives an error when you set one of the following model parameters in the Configuration Parameters dialog box to error:
  - **Diagnostics > Connectivity > Element name mismatch**
  - **Diagnostics > Connectivity > Mux blocks used to create bus signals**

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.

If your model contains Mux blocks that create bus signals, refer to “Tips” in “Mux blocks used to create bus signals” to resolve this problem.

- The Model block uses asynchronous function-call inputs.
- Any of the Model blocks in the model reference hierarchy creates an artificial algebraic loop. If this occurs, take the following steps:
  - 1** On the **Diagnostics** pane of the Configuration Parameters dialog box, set the **Minimize algebraic loop** parameter to **error** so that Simulink reports an algebraic loop error.
  - 2** On the **Model Referencing** Pane of the Configuration Parameters dialog box, select the **Minimize algebraic loop occurrences** parameter.

Simulink tries to eliminate the artificial algebraic loop during simulation.

- 3 Simulate the model.
- 4 If Simulink cannot eliminate the artificial algebraic loop, highlight the location of the algebraic loop by selecting **Edit > Update Diagram**.
- 5 Eliminate the artificial algebraic loop so that the software can analyze the model. Break the loop with Unit Delay blocks so that the execution order is predictable.

---

**Note** For more information, see “Algebraic Loops” in the Simulink documentation.

---

- The parent model and the referenced model have mismatched data type override settings. The data type override setting of the parent model and all of 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 **Tools** menu, in the Fixed Point Tool dialog box under the **Settings for selected system** pane.

## Unsupported Stateflow Software Features

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

In this section...
“ml Namespace Operator, ml Function, ml Expressions” on page 3-15
“C Math Functions” on page 3-15
“Atomic Subcharts That Call Exported Graphical Functions Outside a Subchart” on page 3-16
“Recursion and Cyclic Behavior” on page 3-16
“Custom C or C++ Code” on page 3-18
“Machine-Parented Data” on page 3-18
“Textual Functions with Literal String Arguments” on page 3-19

### 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 “Calling Built-In MATLAB Functions and Accessing Workspace Data” in the Stateflow documentation.)

### C Math Functions

The software supports calls to the following C math functions:

- abs
- ceil
- fabs
- floor
- fmod
- labs
- ldexp

- pow (only for integer exponents)

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

For information about C math functions in Stateflow, see “Calling C Functions in Actions” in the Stateflow documentation.

---

**Note** For details about automatic stubbing, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

---

### **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 how exported graphical functions, see “Exporting Chart-Level Graphical Functions” in the Stateflow documentation.

---

### **Recursion and Cyclic Behavior**

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

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



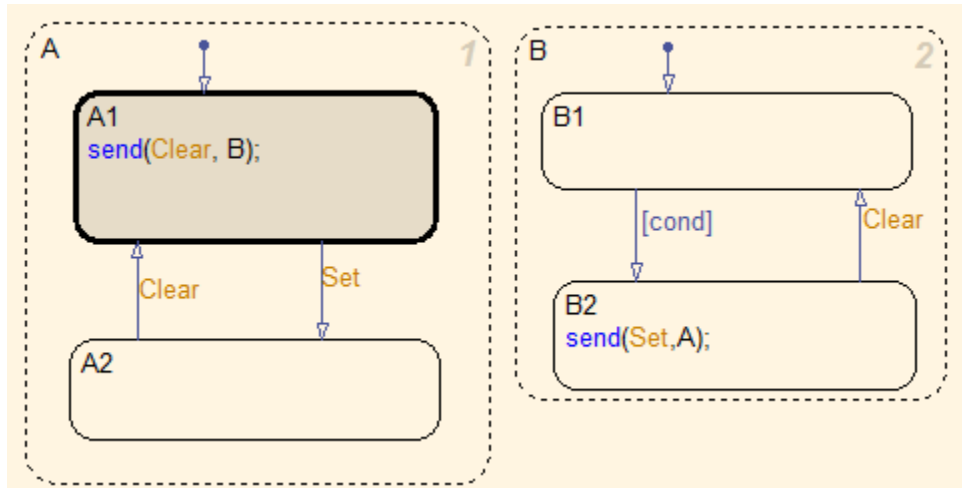
**Note** For information about avoiding recursion in Stateflow charts, see “Guidelines for Avoiding Unwanted Recursion in a Chart” in the Stateflow documentation.

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

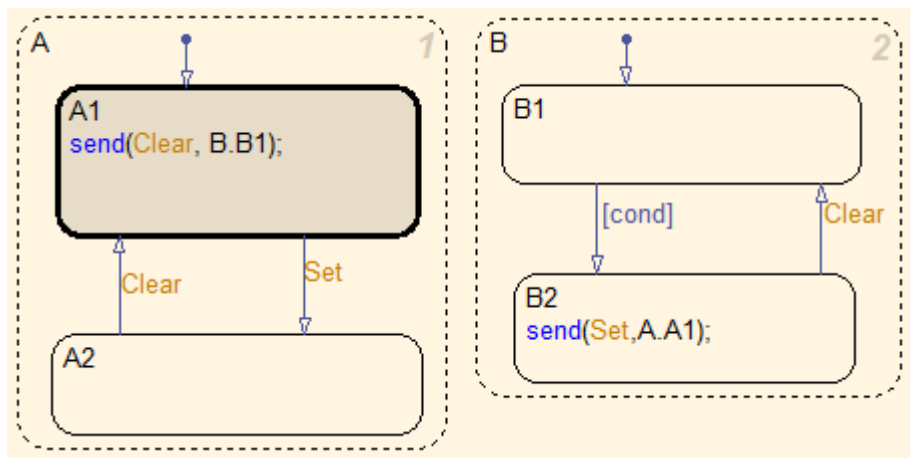
**Note** For information about cyclic behavior in Stateflow charts, see “Cyclic Behavior in a Chart” in the Stateflow documentation.

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

The following chart creates cyclic behavior. State A calls state A1, which broadcasts a Clear event to state B, which calls state B2, which broadcasts a Set event back to state A, causing the cyclic behavior.



If you change the `send` function calls to use directed event broadcasts so that the `Set` and `Clear` events are broadcast directly to the states `B1` and `A1`, respectively, the cyclic behavior disappears and the Simulink Design Verifier software can analyze the model.



---

**Note** For information about the benefits of directed event broadcasts, see “Broadcasting Events to Synchronize States” in the Stateflow documentation.

---

### Custom C or C++ Code

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

For more information about custom code, see “Building Targets” in the Stateflow documentation.

### 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 about machine-parents data, see “Defining Data” in the Stateflow documentation.)

## **Textual Functions with Literal String Arguments**

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

## Support Limitations for MATLAB for Code Generation

In this section...
“Unsupported MATLAB for Code Generation Features” on page 3-20
“Limitations of MATLAB for Code Generation Library Function Support” on page 3-21

### Unsupported MATLAB for Code Generation Features

The software 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 the Simulink Design Verifier software.

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

## **Limitations of MATLAB for Code Generation Library Function Support**

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

If your model contains unsupported functions, you can turn on automatic stubbing, which considers the interface of the unsupported functions, but not their behavior. However, if any of the unsupported functions affect the simulation outcome, the analysis may achieve only partial results. For details about automatic stubbing, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

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

Avoid using unsupported MATLAB library functions in models that you analyze. See Chapter 17, “Support for Code Generation from MATLAB” for a list of the MATLAB library functions for which the software provides limited or no support.

## Fixed-Point Support Limitations

The software supports fixed-point data types in models that it analyzes, with one exception: Parameter configurations do not support fixed-point data types. For more information, see Chapter 5, “Specifying Parameter Configurations”.

For detailed information about these limitations, see “Tunable Expression Limitations” in the *Simulink Coder™ User’s Guide*.

# Working with Block Replacements

---

- “About Block Replacements” on page 4-2
- “Built-In Block Replacements” on page 4-4
- “Template for Block Replacement Rules” on page 4-7
- “Defining Custom Block Replacements” on page 4-8
- “Executing Block Replacements” on page 4-18

# About Block Replacements

Using the Simulink Design Verifier software, 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. Use this capability if you need to:

- 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 in order to resolve incompatibilities. Automatic stubbing replaces unsupported blocks with elements that have the same interface. For more information, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

---

## 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 <b>Vector of input values</b> 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 <b>Number of inputs</b> 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).

File Name	Description
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 <b>Number of inputs</b> 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 <b>Number of inputs</b> 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 <b>Number of inputs</b> parameter is 5. The Test Condition/Proof Assumption block constrains signal values to the interval [1, 5] (or [0, 4] if the block uses zero-based indexing).
blkrep_rule_switch_normal.m blkrep_rule_switch_configss.m	A rule that replaces Switch blocks with an implementation that includes test objectives, requiring that each switch position be exercised when the values of the first and third input ports are different.

File Name	Description
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 <b>Index Option</b> parameter is <b>Index vector (port)</b> . The Test Condition/Proof Assumption block constrains signal values to an interval whose endpoints are derived from the values of the Selector block's <b>Input port size</b> and <b>Index mode</b> 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 <b>Index Option</b> parameter is <b>Starting index (port)</b> . The Test Condition/Proof Assumption block constrains signal values to an interval whose endpoints are derived from the values of the Selector block's <b>Input port size</b> , <b>Output size</b> , and <b>Index mode</b> parameters.

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

`matlabroot/toolbox/sldv/sldv/sldvblockreplacementlib.mdl`

## Template for Block Replacement Rules

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

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

To create a block replacement rule, make a copy of the template and edit the copy to implement the desired behavior for the rule you are creating. The comments in the template provide hints about how to use each section. For a tutorial on using the template to write custom block replacements rules, see “Writing Block Replacement Rules” on page 4-9.

## Defining Custom Block Replacements

**In this section...**

“Basic Workflow for Defining Custom Block Replacements” on page 4-8

“Specifying Replacement Blocks” on page 4-8

“Writing Block Replacement Rules” on page 4-9

“Example: Replacing Multiport Switch Blocks” on page 4-9

### Basic Workflow for Defining Custom Block Replacements

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

- “Specifying Replacement Blocks” on page 4-8
- “Writing Block Replacement Rules” on page 4-9

### Specifying Replacement Blocks

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

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

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

---

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

---

- They must reside in a block library that is available on your MATLAB search path.

- If the replacement block is a subsystem, any Inport and Outport blocks *must* have the default names (In1 and Out1).

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

## Writing Block Replacement Rules

Block replacement rules have the following restrictions:

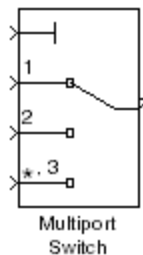
- The function that represents a block replacement rule must include particular callbacks. Use the block replacement rule template as a starting point for writing a custom rule. (See “Template for Block Replacement Rules” on page 4-7.)
- The function that represents a block replacement rule must be on the MATLAB search path.

## Example: Replacing Multiport Switch Blocks

- “Why Replace Multiport Switch Blocks?” on page 4-9
- “Creating the Library and Replacement Block” on page 4-10
- “Writing the Rule for the Replacement Block” on page 4-13

### Why Replace Multiport Switch Blocks?

A Multiport Switch block has one control input port and one or more data input ports; the default number of data inputs is 3.



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

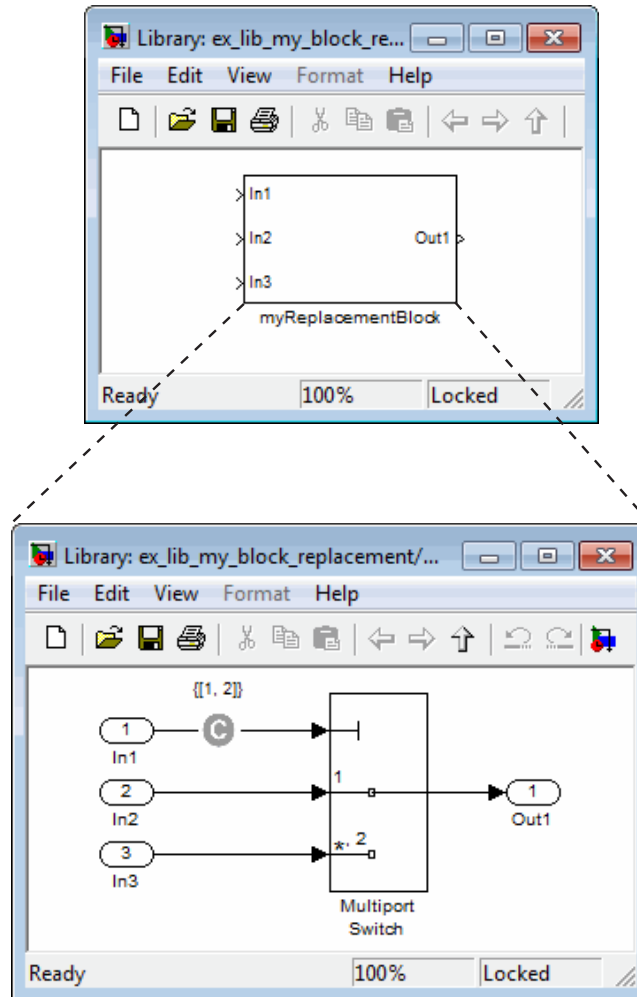
In this example, you create a rule to replace all Multiport Switch blocks that have two data inputs and do not use zero-based indexing. The replacement block is a subsystem that has a Test Condition block that constrains the value of the control input to 1 or 2, so that the analysis does not create out-of-range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

### Creating the Library and Replacement Block

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


- 1 In the Simulink Library Browser, select **File > New > Library**.
- 2 In your library, create a subsystem named `myReplacementBlock` to represent your replacement block. It should look like the following graphic, with several parameters set:
  - In the Multiport Switch block, set the **Number of data ports** parameter to 2.
  - In the Test Condition block, set the **Values** parameter to `{[1, 2]}`.



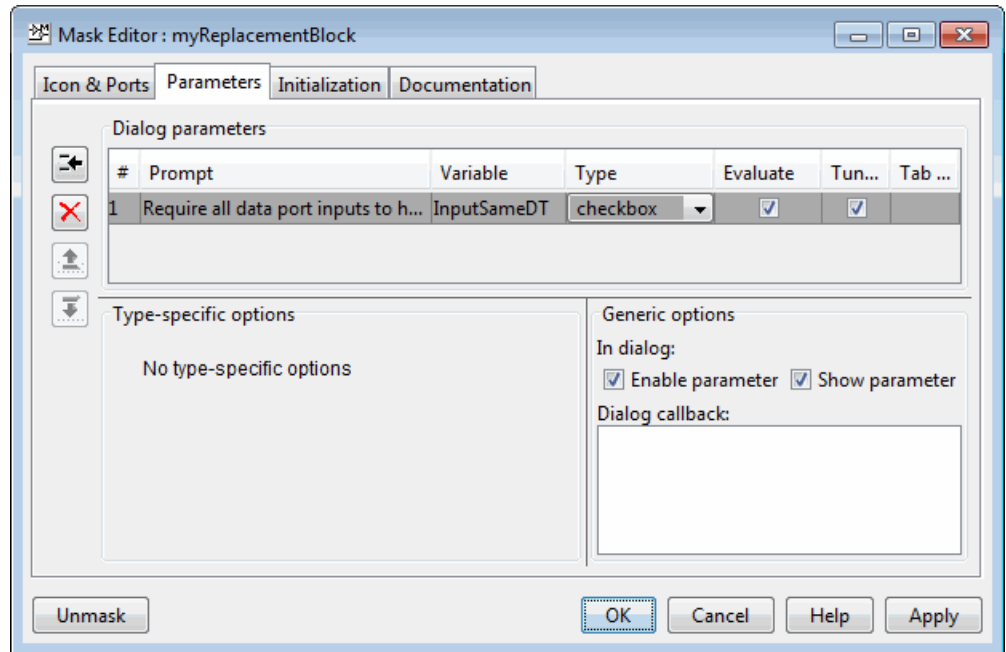


- 3** To create a mask for your subsystem, select the subsystem, right-click, and select **Edit mask** from the context menu.

Specify the following information in the Mask Editor:

- In the **Parameters** pane, click the Add button  to define a mask parameter named InputSameDT as shown.

This parameter replicates the behavior of the **Require all data port inputs to have the same data type** parameter of the underlying Multiport Switch block.




---

**Note** When you create mask parameters that control the behavior of parameters associated with their underlying blocks, specify actual parameter names as dialog box variables in the Mask Editor. For instance, `InputSameDT` is the actual parameter name that controls the **Require all data port inputs to have the same data type** parameter of the Multiport Switch block; therefore, it specifies the name of the dialog box variable in this example.

---

- In the **Initialization** pane, in the **Initialization commands** field, enter commands to specify that the subsystem inherit the `InputSameDT` parameter value of the top-level model:

```
maskInputSameDT = get_param(gcb, 'InputSameDT');
```

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

- 4 Save your block library as `custom_rule.mdl` in a folder on your MATLAB search path.

## Writing the Rule for the Replacement Block

To write a rule for the replacement block:

- 1 Open the block replacement rule template

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

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

---

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

---

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

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

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

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

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

---

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

---

- 5 Specify the full block path name for the replacement block as the `rule.ReplacementPath` object. For this example, to replace Multiport Switch blocks with the replacement block developed in “Specifying Replacement Blocks” on page 4-8, modify the `rule.ReplacementPath` object using the full block path name:

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

---

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

---

- 6 To specify the type of subsystem that the software uses when replacing blocks, specify a value for the `rule.ReplacementMode` object. Valid values are:
  - `Normal` — The software replaces blocks with a copy of the subsystem specified by the `rule.ReplacementPath` object. This is the default.
  - `ConfigurableSubSystem` — The software replaces blocks with a Configurable Subsystem block. With the Configurable Subsystem block, you can choose whether it represents the subsystem specified by the `rule.ReplacementPath` object, or the original block before its replacement.

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

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

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

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

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

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

---

**Note** To determine block parameter names, refer to “Model and Block Parameters” in the *Simulink Reference*.

---

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

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

%% Post Replacement Callback
% Customize the subfunction 'postReplacementFunction' to specify actions
% that will be performed after a block is replaced.
%
```

```
% The usage of this callback in replacement rules is optional. Simulink
% design verifier does not enforce its existence in the rule definition.
%
rule.PostReplacementCallBack = @postReplacementFunction;
```

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

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

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

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

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

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

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

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

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

- 11** Save the edited file and continue to the next section, “Executing Block Replacements” on page 4-18, to execute your replacement rule.

# Executing Block Replacements

In this section...
“Configuring Block Replacements” on page 4-18
“Replacing Blocks in a Model” on page 4-19

## Configuring Block Replacements

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

- 1 Open the `sldvdemo_param_identification` model.
- 2 Rename this model to `my_sldvdemo_param_identification`, and save it in a folder on your MATLAB search path.
- 3 In the Model Editor, select **Tools > Design Verifier > Options**.

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

- 4 In the Configuration Parameters dialog box, select **Design Verifier > Block Replacements**.
- 5 On the **Block Replacements** pane, select **Apply block replacements** to enable block replacements.

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

- 6 To execute your custom block replacement rule, follow these steps:
  - a In the **List of block replacement rules (in order of priority)** box, delete:

```
<FactoryDefaultRules>
```

- b Enter:

```
custom_rule_switch
```



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

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

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

- 8 Click **Apply**.
- 9 Save the `my_sldvdemo_param_identification` model.

## Replacing Blocks in a Model

- “Replacing Blocks and Analyzing the Model with the Block Replacements” on page 4-19
- “Performing the Block Replacements Only” on page 4-20

## Replacing Blocks and Analyzing the Model with the Block Replacements

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

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

---

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

---

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

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

### Performing the Block Replacements Only

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

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

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

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

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

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

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



# Specifying Parameter Configurations

---

- “About Parameter Configurations” on page 5-2
- “Defining Parameter Configurations” on page 5-3
- “Parameter Configuration Example” on page 5-9

### About Parameter Configurations

The Simulink Design Verifier software can treat block 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 the Simulink Design Verifier software to treat that parameter as another input variable in its analysis. This allows you to

- Extend the results of a error detection analysis property proof to consider the impact of additional parameter values.
- Generate comprehensive test cases for situations in which parameter values must vary to achieve more complete coverage results (for an example, see “Parameter Configuration Example” on page 5-9).

## Defining Parameter Configurations

### In this section...

“Template for Defining Parameters” on page 5-3

“Syntax for Defining Parameters” on page 5-3

“Data Types in Parameter Configuration Files” on page 5-7

### Template for Defining Parameters

You define parameter configurations in a MATLAB function. 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
```

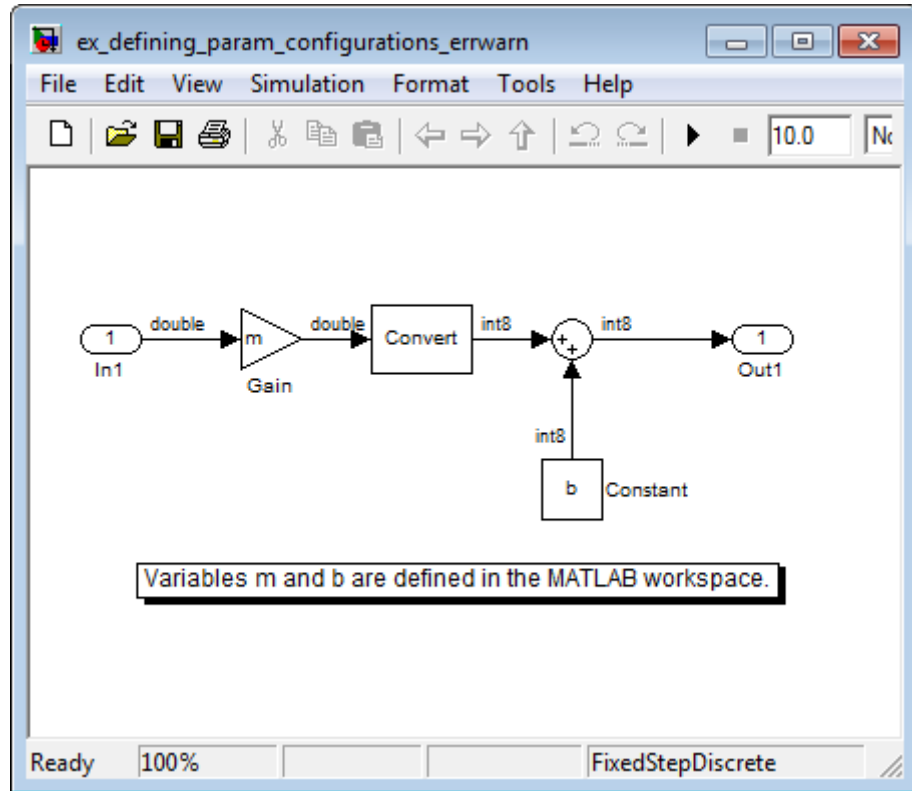
To create a parameter configuration file, make a copy of the template and edit the copy. The comments in the template explain the syntax for defining parameter configurations.

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 for Defining Parameters

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

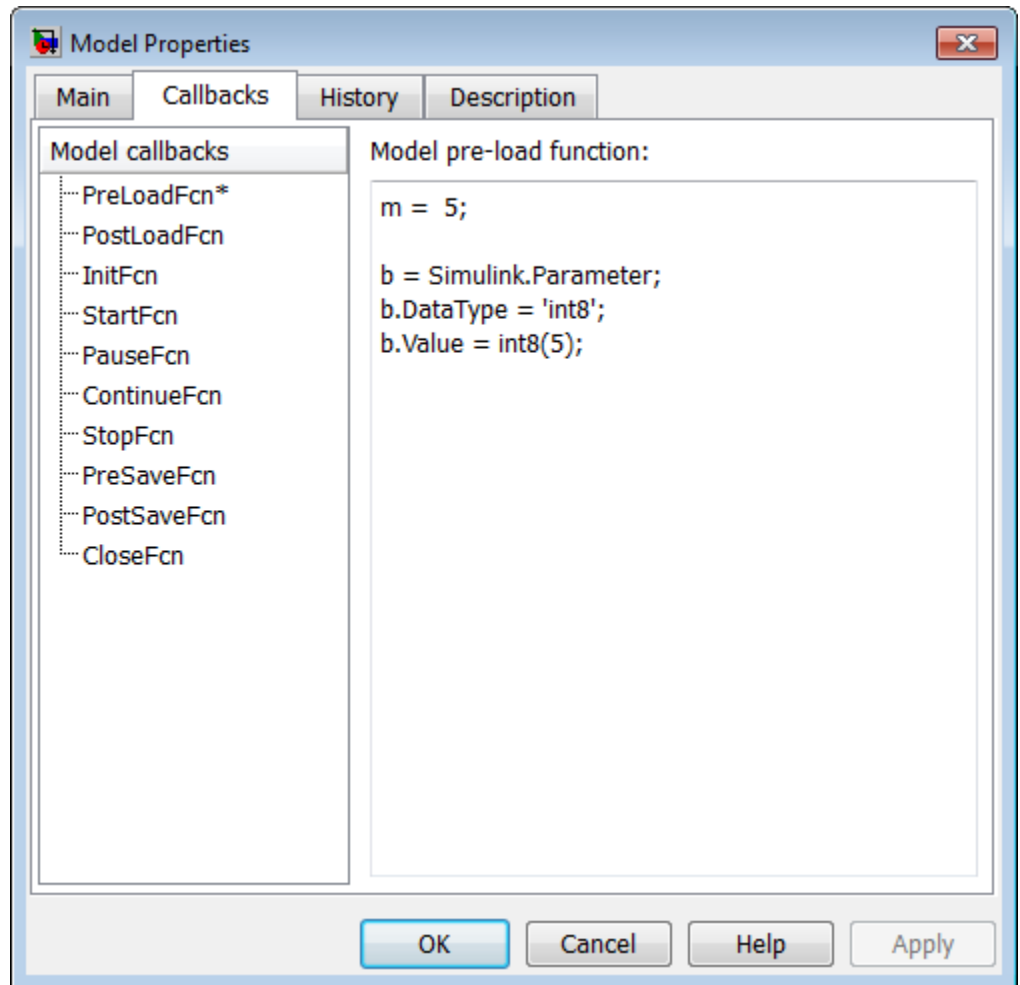
For example, suppose you want to constrain the **Gain** and **Constant value** parameters, *m* and *b*, which appear in the following model:



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.





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 strings 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, the Simulink Design Verifier software considers an interval to be closed if you omit its two-character string.

---

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.

## Data Types in Parameter Configuration Files

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

- “Parameters Cannot Be Structures” on page 5-7
- “Parameters Converted to Fixed Point in the Model” on page 5-8

### Parameters Cannot Be Structures

If the data type of a parameter in the MATLAB workspace is `struct`, Simulink Design Verifier cannot generate values for that parameter during the analysis. However, Simulink Design Verifier *can* generate values for parameters that are not `structs`.

### **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 in the parameter configuration file according to its fixed-point type.

## Parameter Configuration Example

In this section...
“About This Example” on page 5-9
“Constructing the Example Model” on page 5-10
“Parameterizing the Constant Block” on page 5-11
“Preloading the Workspace Variable” on page 5-12
“Specifying a Parameter Configuration” on page 5-12
“Analyzing the Example Model” on page 5-13
“Simulating the Test Cases” on page 5-16

### 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 the Simulink Design Verifier software 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.

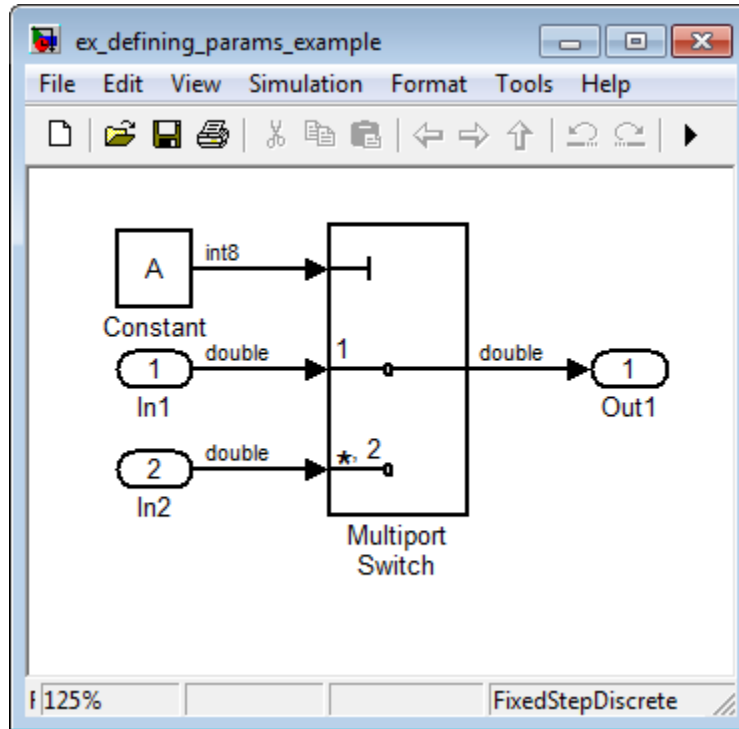
Task	Description	See...
1	Construct the example model.	“Constructing the Example Model” on page 5-10
2	Specify a variable as the value of a Constant block parameter.	“Parameterizing the Constant Block” on page 5-11
3	Constrain the value of the variable that the Constant block specifies.	“Specifying a Parameter Configuration” on page 5-12

<b>Task</b>	<b>Description</b>	<b>See...</b>
4	Generate test cases for your model and interpret the results.	“Analyzing the Example Model” on page 5-13
5	Simulate the test cases and measure the resulting decision coverage.	“Simulating the Test Cases” on page 5-16

### Constructing the Example Model

Construct a simple 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:
    - 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 this:



**5** Select **Simulation > Configuration Parameters**.

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

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

**8** Save your model as `ex_defining_params_example.mdl` for use in the next procedure.

## Parameterizing the 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 for use in the next step.

### Preloading the Workspace Variable

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

- 1 Select **File > Model Properties**.
- 2 Click the **Callbacks** tab.
- 3 In the PreLoadFcn, enter:

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

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

- 4 To close the Model Properties dialog box and save your changes, click **OK**.
- 5 Save your model for use in the next step.

### Specifying a Parameter Configuration

Create the parameter configuration file so that it constrains the variable A, and configure the analysis to use this file:

- 1 In the model window, select **Tools > Design Verifier > Options**.

The Simulink Design Verifier software options appear in the Configuration Parameters dialog box.

- 2 In the **Select** tree on the left side of the Configuration Parameters dialog box, click the **Design Verifier > Parameters** category.



- 3 In the **Parameters** pane on the right side, select the **Apply parameters** parameter.

Enabling the **Apply parameters** option provides access to the **Parameter configuration file** option; the file name `sldv_params_template.m` appears in the text box.

- 4 Click **Edit** next to the **Parameter configuration file** option.

The Simulink Design Verifier software opens `sldv_params_template.m` in an editor.

- 5 Replace the existing content with the following:

```
function params = params_example_function
    % This function defines a parameter configuration for the
    % example model that the documentation discusses.

    params.A = int8([1 2]);
```

This code defines a function `params_example_function` that constrains the parameter `A` to the `int8` values 1 and 2.

- 6 Save your changes to the template as `params_example_function.m` in the same folder as the example model.
- 7 Close the MATLAB Editor.
- 8 In the Configuration Parameters dialog box, click **Browse** next to the **Parameter configuration file** option, and then select your parameter configuration file, `params_example_function.m`.
- 9 Click **OK** to apply your change and close the Configuration Parameters dialog box.
- 10 Save your model for use in the next step.

## Analyzing the Example Model

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

- 1** In the model window, select **Tools > Design Verifier > Generate Tests**.

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

- 2** When the software completes its analysis, in the log window, select **Generate detailed analysis report**.

The Simulink Design Verifier software displays an HTML report named `ex_defining_params_example_report.html`.

Keep the log 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 Seconds (1 sample periods)

Objective Count: 1

### Objectives

Step	Time	Model Item	Objectives
1	0	<a href="#">Multiport Switch</a>	integer input value = 1 (output is from input port 1)

### Generated Parameter Values

Parameter	Value
A	1

### Generated Input Data

Time	0
<b>Step 1</b>	
In1	-
In2	-

This section provides details about Test Case 1 that the Simulink Design Verifier software 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 Seconds (1 sample periods)  
 Objective Count: 1

**Objectives**

Step	Time	Model Item	Objectives
1	0	<a href="#">Multiport Switch</a>	integer input value = *,2 (output is from input port 2)

**Generated Parameter Values**

Parameter	Value
A	2

**Generated Input Data**

Time	0
<b>Step 1</b>	
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.

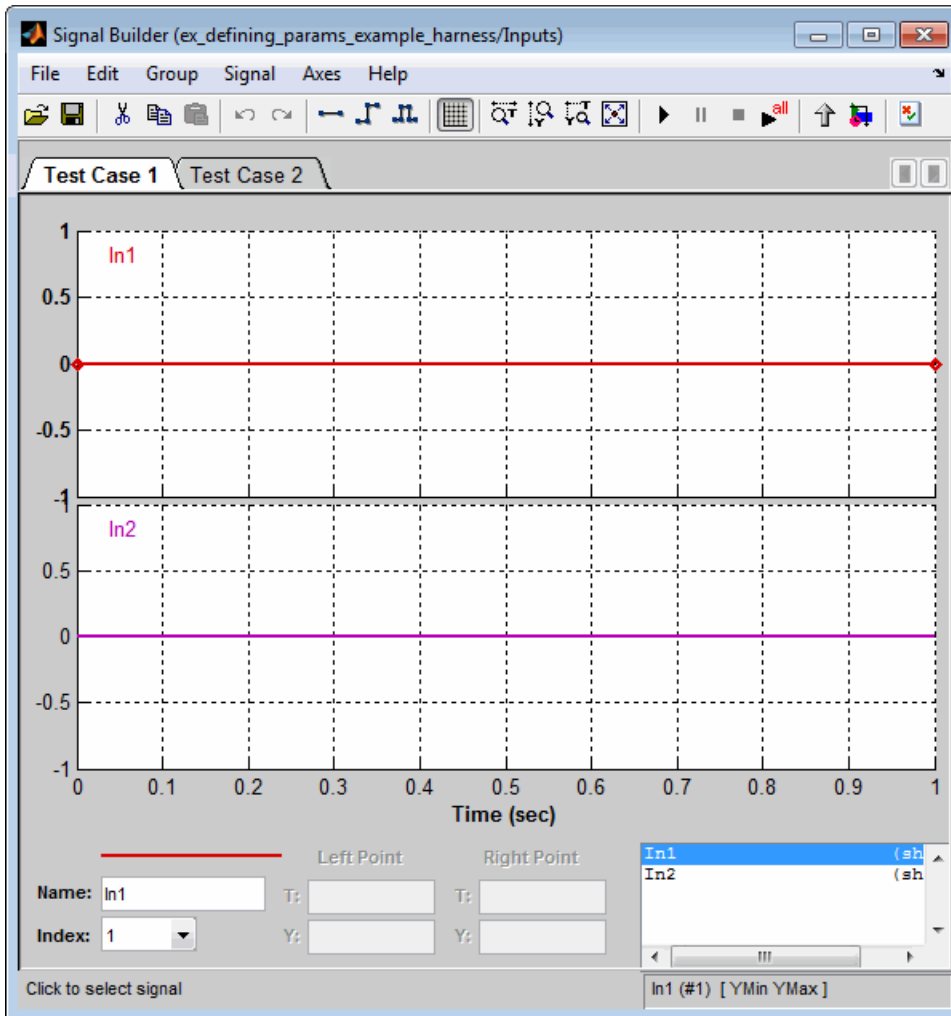
## Simulating the Test Cases


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

- 1 In the Simulink Design Verifier log window, select **Create harness model**.

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



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



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

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

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

<b>Summary</b>	
Model Hierarchy/Complexity:	Test 1
	D1
1. <a href="#">ex_defining_params_example_harness</a>	2 100% 
2. . . . <a href="#">Test Unit (copied from ex_defining_params_example)</a>	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 "[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:

truncated 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.





# Detecting Design Errors

---

- “What Is Design Error Detection?” on page 6-2
- “Derived Ranges in Design Error Detection” on page 6-3
- “Running a Design Error Detection Analysis” on page 6-5
- “Detecting Dead Logic” on page 6-9
- “Detecting Integer Overflow and Division-by-Zero Errors” on page 6-29
- “Checking for Specified Intermediate Minimum and Maximum Signal Values” on page 6-35

# 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

If your model contains arithmetic operations or you suspect dead logic, analyze your model using the design error detection mode. Running this analysis before you simulate your model identifies any design flaws that cause errors. The analysis also identifies the conditions that cause the error and a range of signal values that can occur for all signals in the model.

After the analysis, you can:

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

## Derived Ranges in Design Error Detection

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

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

- Block Outports
- Stateflow local data

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

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

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

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



In this model, the design ranges are:

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

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

However, if you disable the **Use specified input minimum and maximum values** parameter, the analysis calculates the derived ranges based on unrestricted values of the input ports of the model. In the preceding model, the only valid outputs of the Abs block are nonnegative numbers. Consequently, the derived range for the Abs block is [0..Inf].

## Running a Design Error Detection Analysis

### In this section...

“Workflow for Detecting Design Errors” on page 6-5

“Understanding the Analysis Results” on page 6-5

“Reviewing the Latest Analysis Results in the Model Explorer” on page 6-7

### 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 **Transition shadowing** 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.

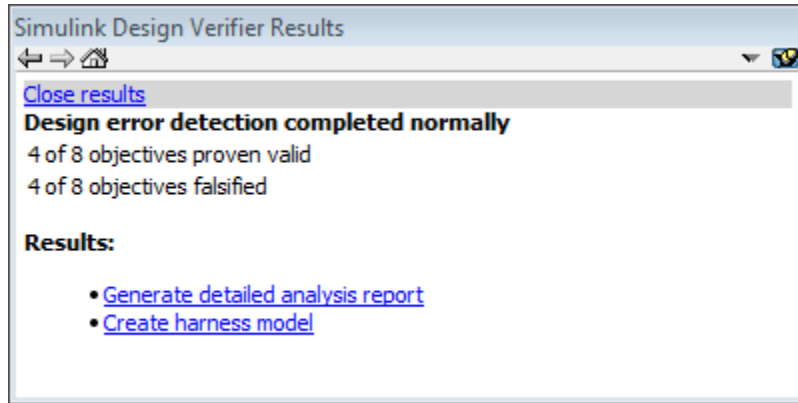
---

### Understanding 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.


<b>Model Object Highlighting Color</b>	<b>Analysis Results</b>
Green	One of the following: <ul style="list-style-type: none"> <li>• The analysis did not find any overflow or division-by-zero errors.</li> <li>• The analysis did not find any dead logic.</li> <li>• The analysis did not find intermediate or output signals outside the range of user-specified minimum and maximum constraints.</li> </ul>
Red	One of the following: <ul style="list-style-type: none"> <li>• The analysis found at least one test case that causes overflow or division-by-zero errors.</li> <li>• The analysis found dead logic.</li> <li>• The analysis found intermediate or output signals outside the range of user-specified minimum and maximum constraints.</li> </ul>
Orange	For at least one objective, the analysis could not determine if there was any dead logic, overflow or division-by-zero errors. This situation can occur when: <ul style="list-style-type: none"> <li>• The analysis times out.</li> <li>• The software cannot determine if an error occurred or not. This result is due to:               <ul style="list-style-type: none"> <li>▪ Automatic stubbing errors; for more information, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.</li> <li>▪ Limitations of the analysis engine</li> </ul> </li> </ul>
Gray	The model object was not part of the analysis.

The Simulink Design Verifier Results window initially displays a summary of the analysis results, as in the following example.



When you click an object in the model, additional details about the results for that object are displayed in the Simulink Design Verifier Results window.

---

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

---

## Reviewing the Latest Analysis Results in the Model Explorer

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

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

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

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

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

---

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

---

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



## Detecting Dead Logic

### In this section...

“Overview of Detecting Dead Logic” on page 6-9

“Model Objects That Receive Dead Logic Detection” on page 6-9

“Detecting Dead Logic in Example Model” on page 6-25

“Reviewing Analysis Results” on page 6-26

“Reviewing the Analysis Report” on page 6-27

### Overview of Detecting Dead Logic

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

To conduct design error detection analysis for dead logic:

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

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

### 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
<a href="#">“Abs” on page 6-12</a>	●	
<a href="#">“Bias” on page 6-12</a>	●	
<a href="#">“Data Type Conversion” on page 6-12</a>	●	

<b>Model Object Receiving Dead Logic Detection</b>	<b>Decision Outcomes</b>	<b>Condition Outcomes</b>
“Dead Zone” on page 6-13	●	
“Discrete Filter” on page 6-13	●	
“Discrete FIR Filter” on page 6-14	●	
“Discrete-Time Integrator” on page 6-14	●	
“Discrete Transfer Fcn” on page 6-15	●	
“Dot Product” on page 6-15	●	
“Enabled Subsystem” on page 6-15	●	●
“Enabled and Triggered Subsystem” on page 6-15	●	●
“Fcn” on page 6-16		●
“For Iterator, For Iterator Subsystem” on page 6-16	●	
“Gain” on page 6-16	●	
“If, If Action Subsystem” on page 6-17	●	●
“Interpolation Using Prelookup” on page 6-17	●	
“Library-Linked Objects” on page 6-17	●	●
“Logical Operator” on page 6-17		●
“1-D Lookup Table” on page 6-17	●	
“2-D Lookup Table” on page 6-18	●	
“n-D Lookup Table” on page 6-18	●	
“Math Function” on page 6-18	●	
“MATLAB Function” on page 6-18	●	●

<b>Model Object Receiving Dead Logic Detection</b>	<b>Decision Outcomes</b>	<b>Condition Outcomes</b>
“MinMax” on page 6-19	●	
“Model” on page 6-19	●	●
“Multiport Switch” on page 6-19	●	
“PID Controller, PID Controller (2 DOF)” on page 6-20	●	
“Product” on page 6-20	●	
“Rate Limiter” on page 6-20	●	
“Relay” on page 6-21	●	
“Saturation” on page 6-21	●	
“Saturation Dynamic” on page 6-22	●	
“Sqrt, Signed Sqrt, Reciprocal Sqrt” on page 6-22	●	
“Stateflow Charts” on page 6-22	●	●
“Sum, Add, Subtract, Sum of Elements” on page 6-23	●	
“Switch” on page 6-23	●	
“SwitchCase, SwitchCase Action Subsystem” on page 6-23	●	
“Triggered Models” on page 6-23	●	●
“Triggered Subsystem” on page 6-24	●	●
“Unary Minus” on page 6-24	●	
“Weighted Sample Time Math” on page 6-24	●	
“While Iterator, While Iterator Subsystem” on page 6-25	●	

### **Abs**

The Abs block has decision outcomes based on:

- Input to the block being less than zero.
- Selection of the **Saturate on integer overflow** parameter.
- 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 **Saturate on integer overflow** parameter is selected, there are two decision outcomes:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, 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 any decision. If the input data type to the Abs block is `Boolean`, an error occurs.

### **Bias**

The Bias block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **Data Type Conversion**

The Data Type Conversion block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

## Dead Zone

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

- **Start of dead zone**
- **End of dead zone**
- **Saturate on integer overflow**

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.

If the **Saturate on integer overflow** parameter is selected, there are two decision outcomes:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

## Discrete Filter

The Discrete Filter block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### Discrete FIR Filter

The Discrete FIR Filter block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### Discrete-Time Integrator

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

- **External reset**
- **Limit output**
- **Saturate on integer overflow**

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.

If the **Saturate on integer overflow** parameter is selected, there are two decision outcomes:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **Discrete Transfer Fcn**

The Discrete Transfer Fcn block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **Dot Product**

The Dot Product block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, 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.

### **Gain**

The Gain block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.



- Block does not saturate on integer overflow, indicating a false decision.

### **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.

### **Interpolation Using Prelookup**

The Interpolation Using Prelookup block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **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.

### **1-D Lookup Table**

The 1-D Lookup Table block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### 2-D Lookup Table

The 2-D Lookup Table block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### n-D Lookup Table

The n-D Lookup Table block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### Math Function

The Math Function block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### 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.
- Selection of the **Saturate on integer overflow** parameter.

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.

If the **Saturate on integer overflow** parameter is selected, there are two decision outcomes:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, 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.
- Selection of the **Saturate on integer overflow** parameter.

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.

If the **Saturate on integer overflow** parameter is selected, there are two decision outcomes:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **PID Controller, PID Controller (2 DOF)**

The PID Controller and PID Controller (2 DOF) blocks have two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **Product**

The Product block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, 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.

### **Saturation Dynamic**

The Saturation Dynamic block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **Sqrt, Signed Sqrt, Reciprocal Sqrt**

The Sqrt, Signed Sqrt, Reciprocal Sqrt blocks have two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **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.

### **Sum, Add, Subtract, Sum of Elements**

The Sum, Add, Subtract, Sum of Elements blocks have two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **Switch**

The Switch block has decision outcomes based on:

- Control input to block
- Selection of the **Saturate on integer overflow** parameter

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.

If the **Saturate on integer overflow** parameter is selected, there are two decision outcomes:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, 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.

### **Unary Minus**

The Unary block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.

### **Weighted Sample Time Math**

The Weighted Sample Time Math block has two decision outcomes if the **Saturate on integer overflow** parameter is selected:

- Block saturates on integer overflow, indicating a true decision.
- Block does not saturate on integer overflow, indicating a false decision.



## 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.

## Detecting Dead Logic in Example Model

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

- 1** Open the `sldvdemo_fuelsys_logic` model.
- 2** Select **Tools > Design Verifier > Options**.
- 3** In the Configuration Parameters dialog box, select **Design Verifier > Design Error Detection**.
- 4** On the **Design Error Detection** pane, enable **Dead Logic**.
- 5** Close the Configuration Parameters dialog box.
- 6** Select **Tools > Design Verifier > Detect Design Errors**.

---

**Note** If your model is incompatible with the Simulink Design Verifier software, see Chapter 3, “Ensuring Compatibility with the Simulink® Design Verifier™ Software” for information about unsupported Simulink and Stateflow software features.

---

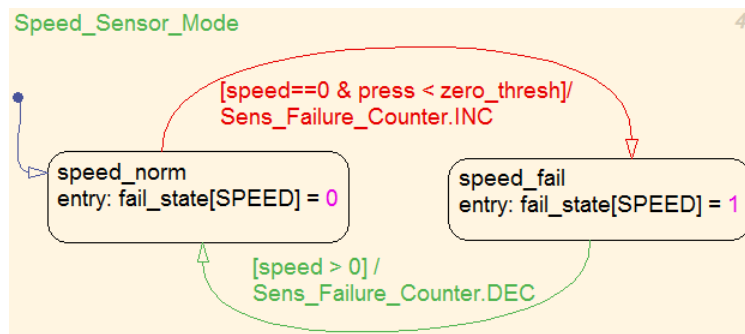
After the analysis:

- The software highlights the model with the analysis results.
- The Simulink Design Verifier Results dialog box opens and displays a summary of the analysis.

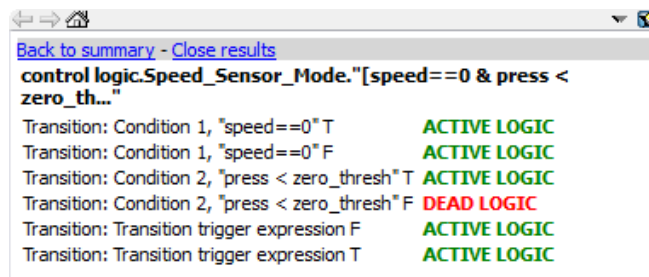
## Reviewing Analysis Results

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

- To review the model analysis results, at the top level of the `sldvdemo_fuelsys_logic` model, open the control logic Stateflow chart.
  - Model objects with active logic are highlighted in green.
  - Model objects with dead logic are highlighted in red.
- In the `sldvdemo_fuelsys_logic/control logic` window, click the `Sens_Failure_Counter.INC` transition in the `Speed_Sensor_Mode` state.



The Simulink Design Verifier Results dialog box displays a summary of the active and dead logic for the transition.



## Reviewing the Analysis Report

You can generate an HTML report containing detailed information about the analysis. This example shows how to generate an analysis report after you detect dead logic in the `sldvdemo_fuelsys_logic` model.

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

**2** Click **Generate detailed analysis report**.

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

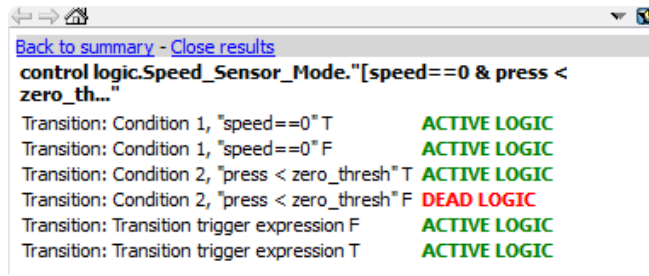
**3** In the analysis report, click **Design Error Detection Objectives Status**.

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

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

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

- In the analysis report, the **Design Min Max Constraints** section lists the constraints for control logic/ Input Data "press". The value of `press` is constrained to values between 0 and 2.
- In the Model Explorer window, select **Base Workspace > zero\_thresh**. The value of `zero_thresh` is 250.
- With the design constraints, `press` is always less than `zero_threshold`. The logic `press < zero_thresh` is never false. The false condition for the transition is never exercised. Therefore, it is dead logic.



5 Close sldvdemo\_fuelsys\_logic.

## Detecting Integer Overflow and Division-by-Zero Errors

### In this section...

“About This Example” on page 6-29

“Analyzing the Model” on page 6-29

“Reviewing the Analysis Results” on page 6-30

### About This Example

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

### Analyzing the Model

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

- 1 Open the `sldvdemo_cruise_control_fxp_fixed` model.
- 2 Select **Tools > Design Verifier > Options**.
- 3 In the Configuration Parameters dialog box, on the **Select** pane, under **Design Verifier**, select **Design Error Detection**.
- 4 On the **Design Error Detection** pane, select:
  - **Integer overflow**
  - **Division by zero**

When detecting integer overflow errors, in the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane, set both **Signals > Detect overflow** and **Parameters > Detect overflow** to error.

- 5 Close the Configuration Parameters dialog box.
- 6 Select **Tools > Design Verifier > Detect Design Errors**.

---

**Note** If your model is incompatible with the Simulink Design Verifier software, see Chapter 3, “Ensuring Compatibility with the Simulink® Design Verifier™ Software” for information about unsupported Simulink and Stateflow software features.

---

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.

### Reviewing the Analysis Results

- “Reviewing the Results on the Model” on page 6-30
- “Reviewing the Harness Model” on page 6-33
- “Reviewing the Analysis Report” on page 6-34

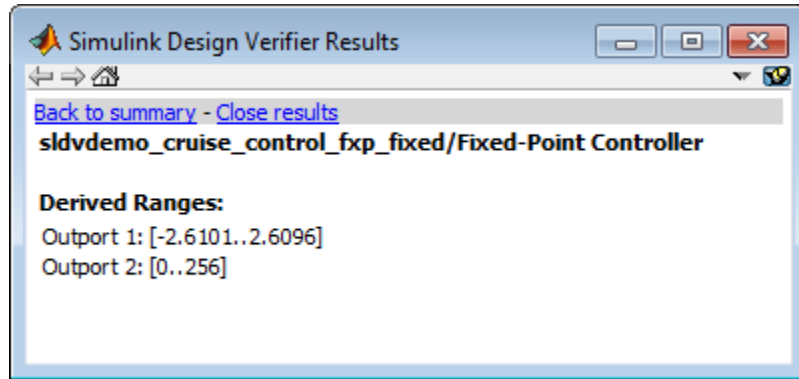
### Reviewing the Results on the Model

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 Output 1 (throt) range from  $-2.6101$  to  $2.6096$ .
- The values of Output 2 (target) range from  $0$  to  $256$ .

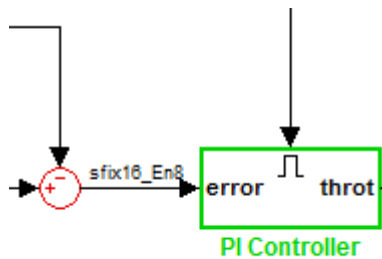


2 Click the Outport blocks of the sldvdemo\_cruise\_control\_fxp\_fixed model to see the same signal bound values.

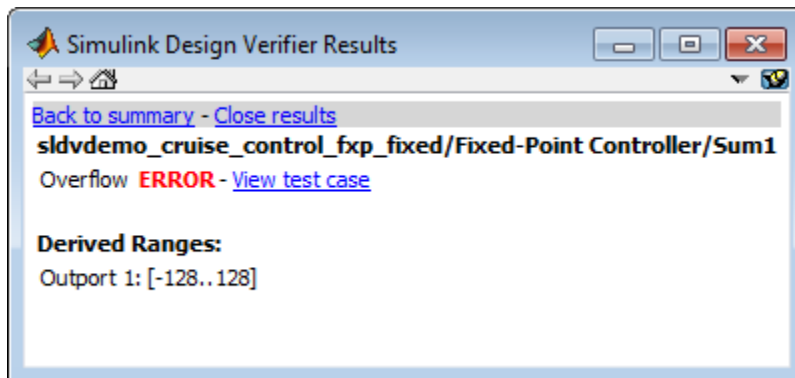
3 Open the Fixed-Point Controller subsystem.

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

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



This Sum block can produce an overflow error. The analysis found a test case that can result in a computation where the output of the Sum block exceeds the range [-128..128].



- 5 To more fully understand this error, click the two blocks that provide the inputs to the Sum block. In the Simulink Design Verifier Results window, view their derived ranges:
- The third Output from the Bus block has a range of [0..256].
  - The Output from the Switch block has a range of [0..256].

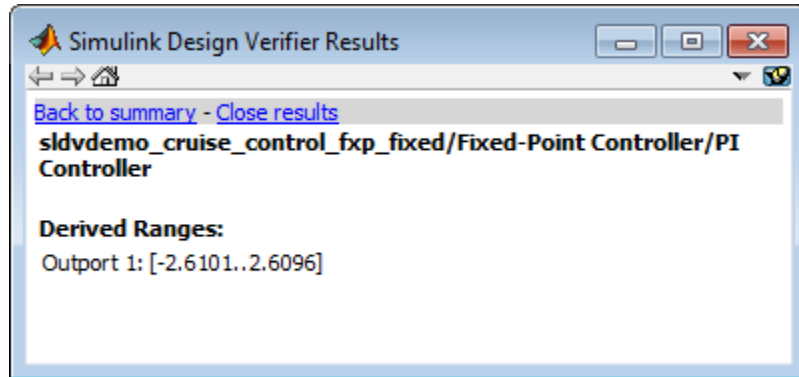
You can see that the sum operation for these signal ranges can compute a value that exceeds the range [-128..128] for the Output of the Sum block.

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. The Simulink Design Verifier Results window displays the derived range for the Output: [-2.6101..2.6096].

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.

## Reviewing 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 **Play** 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 “Harness Model” on page 13-15.

### Reviewing 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 have any errors
- **Objectives Falsified with Test Cases** — Model objects for which test cases generated errors

For more information, see “Simulink® Design Verifier™ Reports” on page 13-27.

## Checking for Specified Intermediate Minimum and Maximum Signal Values

### In this section...

“Overview of Specified Minimum and Maximum Signal Values” on page 6-35

“About This Example” on page 6-36

“Creating the Example Model” on page 6-36

“Analyzing the Model” on page 6-39

“Reviewing the Analysis Results” on page 6-40

### Overview of Specified Minimum and Maximum Signal Values

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

The analysis checks for specified minimum and maximum values on:

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

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

### Limitations of Checking Specified Minimum and Maximum Signal Values

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

- `Simulink.BusElement` objects.
- Any Mux block with an output connected to a Selector block
- Merge block inputs

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

---

**Note** For information about how a Simulink Design Verifier analysis handles specified minimum and maximum values on input ports, see Chapter 11, “Considering Specified Minimum and Maximum Values for Inputs During Analysis”.

---

### 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 any 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.

### Creating the Example Model

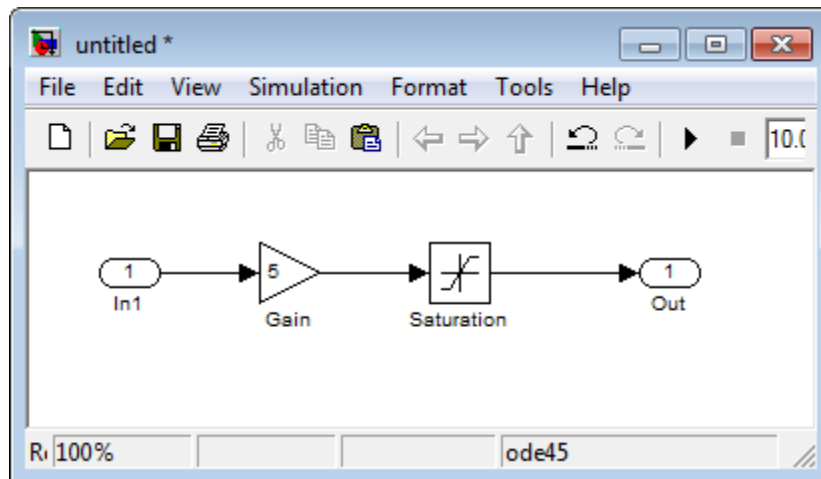
Create the model for this example:

- 1 In the model window, select **File > New > Model**.

- 2** From the Simulink Commonly Used Blocks library, add the following blocks to the model and assign the indicated parameter values.

<b>Block</b>	<b>Tab</b>	<b>Parameter</b>	<b>Value</b>
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 in the model window, select **Format > Port/Signal Displays > Design Ranges**.
- 5 Select **Tools > Design Verifier > Options**.
- 6 In the Configuration Parameters dialog box, on the **Solver** pane, under **Solver options**:
  - a Set **Type** to Fixed-step.  

The Simulink Design Verifier software does not support variable-step solvers.
  - b Set **Solver** to discrete (no continuous states).
- 7 On the **Design Verifier** pane, set **Mode** to Design error detection.
- 8 On the **Design Verifier, Design Error Detection** pane:
  - a Select **Check specified intermediate minimum and maximum values**.
  - b Clear the **Integer overflow** and **Division by zero** parameters.

In this example, you check only for intermediate minimum and maximum violations.

9 To save these settings and exit the Configuration Parameters dialog box, click **OK**.

10 Save the model and name it `ex_interim_minmax.mdl`.

## Analyzing the Model

To analyze the example model to identify any intermediate signals that violate the specified minimum and maximum values, select **Tools > Design Verifier > Detect Design Errors**.

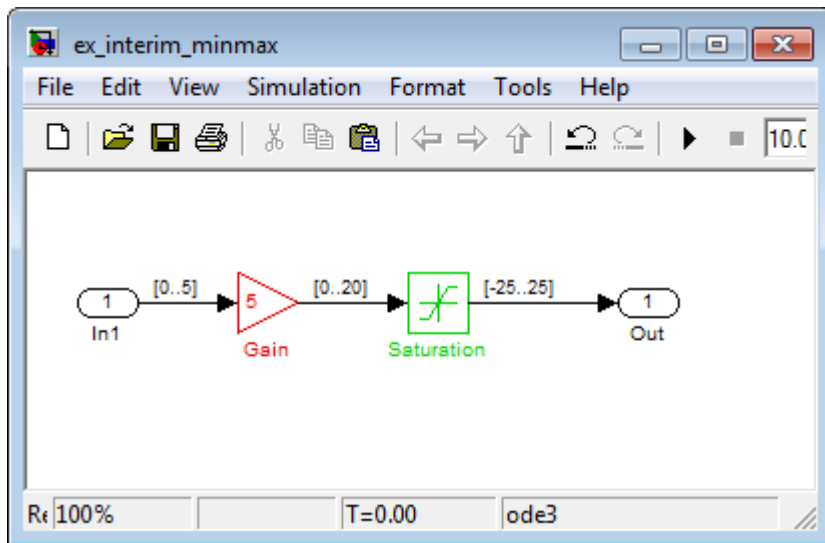
---

**Note** If your model is incompatible with the Simulink Design Verifier software, for information about the Simulink and Stateflow software features that are not supported by Simulink Design Verifier, see Chapter 3, “Ensuring Compatibility with the Simulink® Design Verifier™ Software”.

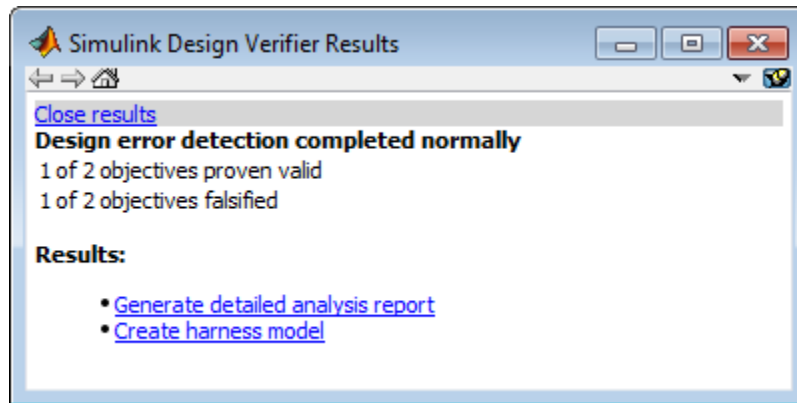
---

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.



### Reviewing the Analysis Results

- “Reviewing Results on the Model” on page 6-40
- “Reviewing the Harness Model” on page 6-42
- “Reviewing the Analysis Report” on page 6-43

### Reviewing Results on the Model

In the model window, the Gain block is colored red and the Saturation block is colored green. This indicates that:

- At least one objective associated with the Gain block was falsified. For this example, the analysis falsified exactly one objective.
- All objectives associated with the Saturation block were satisfied. For this example, the analysis satisfied exactly one objective.

To understand these results:

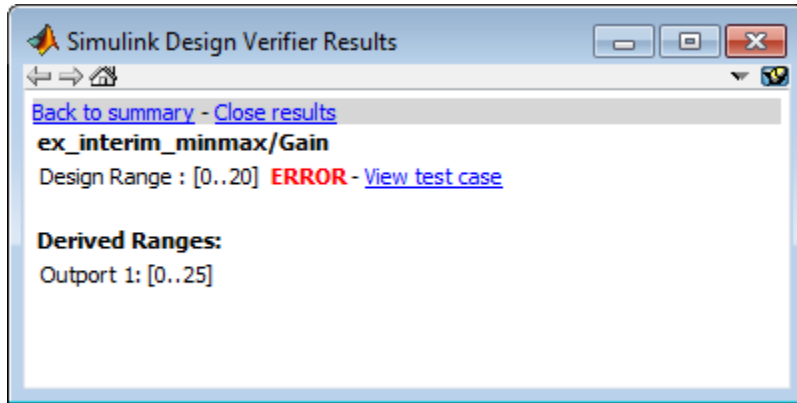
- 1 Click the Gain block.

The Simulink Design Verifier Results window shows that the design range for the output was [0..20], but the analysis detected an error and generated



a test case that demonstrates that error. Because the design range for the input block is  $[0..5]$ , when the input to the Gain block is 5, the output is 25, which exceeds the specified maximum value on that port.

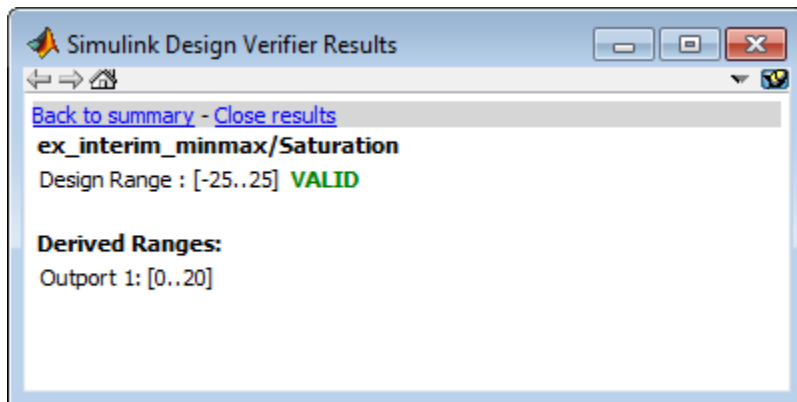
The analysis computes and displays the derived range to help you understand how the design range was exceeded.



When the software detects an error, the analysis proceeds, but constrains the signal to the tightest intersection of the design range and the derived range, which for the Gain block output in this example is  $[0..20]$ .

## 2 Click the Saturation block.

The Simulink Design Verifier Results window shows that the output of the Saturation block never exceeded the design range  $[-25..25]$ . The input to the Saturation block never exceeded  $[0..20]$ , which is the derived range that the analysis propagated from the Gain block.



### Reviewing 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.

## Reviewing 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.



# Generating Test Cases

---

- “About Test Case Generation” on page 7-2
- “Workflow for Generating Test Cases” on page 7-4
- “Generating Test Cases to Achieve Decision Coverage for a Model” on page 7-5
- “Generating Test Cases for a Subsystem” on page 7-23

## About Test Case Generation

The Simulink Design Verifier software can generate test cases that satisfy coverage objectives for your model, including:

- Decision coverage
- Condition coverage
- Modified condition and decision coverage (MC/DC)

Test cases help you confirm model performance by demonstrating how its blocks execute in different modes. When generating test cases, the software performs a formal analysis of your model. After completing the analysis, the software offers several ways for you to review the results.

### Test Case Blocks

The Simulink Design Verifier software provides two blocks for customizing test cases for your Simulink models:

- The Test Objective block defines the values of a signal that a test case must satisfy.
- The Test Condition block constrains the values of a signal during analysis.

### Test Case Functions

The Simulink Design Verifier software provides two MATLAB functions to customize test cases for a Simulink model or Stateflow chart. You can use these functions in a MATLAB Function block. Both functions are active in generated code and in Simulink Design Verifier.

- `sldv.test` — Specifies a test objective
- `sldv.condition` — Specifies a test condition

These functions:

- Identify mathematical relationships for testing in a form that can be more natural than using block parameters.

- Support specifying multiple objectives, assumptions, or conditions without complicating the model.
- Provide access to the power of MATLAB.
- Support separation of verification and model design.

For an example of how to use these functions, see the `sldv.test` or `sldv.condition` reference page.

---

**Note** Simulink Design Verifier blocks and functions are saved with a model. If you open the model on a MATLAB installation that does not have a Simulink Design Verifier license, you can see the blocks and functions, but they do not produce any results.

---

## Workflow for Generating Test Cases

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 the Simulink Design Verifier software.	“Checking Compatibility of the Example Model” on page 7-7
2	If you have Stateflow objects in your model, in the Configuration Parameters dialog box, on the <b>Diagnostics &gt; Stateflow</b> pane, set <b>Transition shadowing</b> to error.	
3	Optionally, instrument your model with blocks or MATLAB functions that specify test objectives and test conditions.	“Customizing Test Generation” on page 7-18
4	Specify options that control how Simulink Design Verifier generates test cases for your model.	“Configuring Test Generation Options” on page 7-8
5	Execute the Simulink Design Verifier analysis.	“Analyzing the Example Model” on page 7-9 and “Reanalyzing the Example Model” on page 7-21
6	Review the analysis results.	“Reviewing the Analysis Results” on page 7-10



# Generating Test Cases to Achieve Decision Coverage for a Model

## In this section...

“Constructing the Example Model” on page 7-5

“Checking Compatibility of the Example Model” on page 7-7

“Configuring Test Generation Options” on page 7-8

“Analyzing the Example Model” on page 7-9

“Reviewing the Analysis Results” on page 7-10

“Customizing Test Generation” on page 7-18

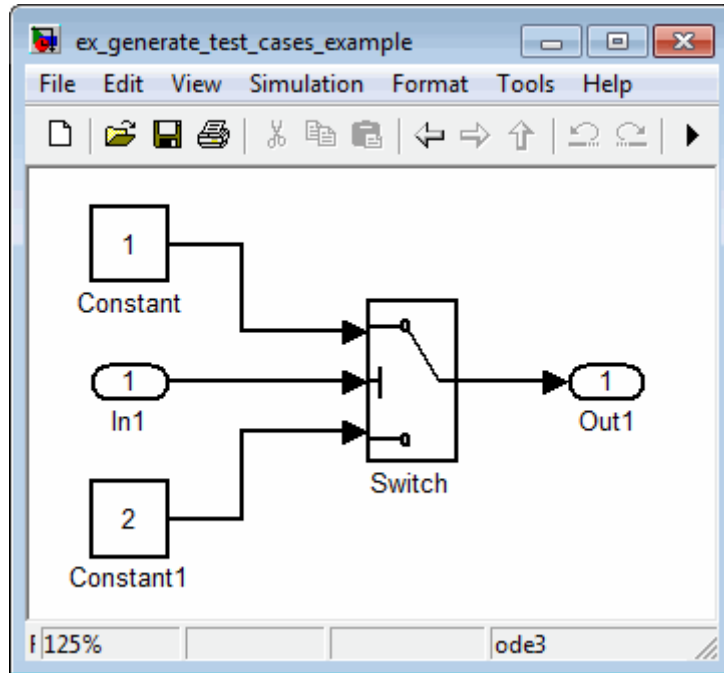
“Reanalyzing the Example Model” on page 7-21

“Analyzing Contradictory Models” on page 7-22

## Constructing the Example Model

Construct a model to use in this example:

- 1 Create a new 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 the Simulink Design Verifier software controls
  - From the Sources library, two Constant blocks to serve as Switch block data inputs
  - From the Signal Routing library, a Switch block to provide simple logic
  - From the Sinks library, an Outport block to receive the output signal
- 3 In your model, double-click one of the Constant blocks and specify its **Constant value** parameter as 2.
- 4 Connect the blocks so that your model appears similar to the following diagram.



**5** In the model window, select **Simulation > Configuration Parameters**.

**6** On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Solver** category. On the right side, under **Solver options**:

- Set the **Type** option to Fixed-step.
- Set the **Solver** option to Discrete (no continuous states).

The Simulink Design Verifier can analyze only models that use a fixed-step solver.

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

**8** Save your model with the name `ex_generate_test_cases_example.mdl`.

## Checking Compatibility of the Example Model

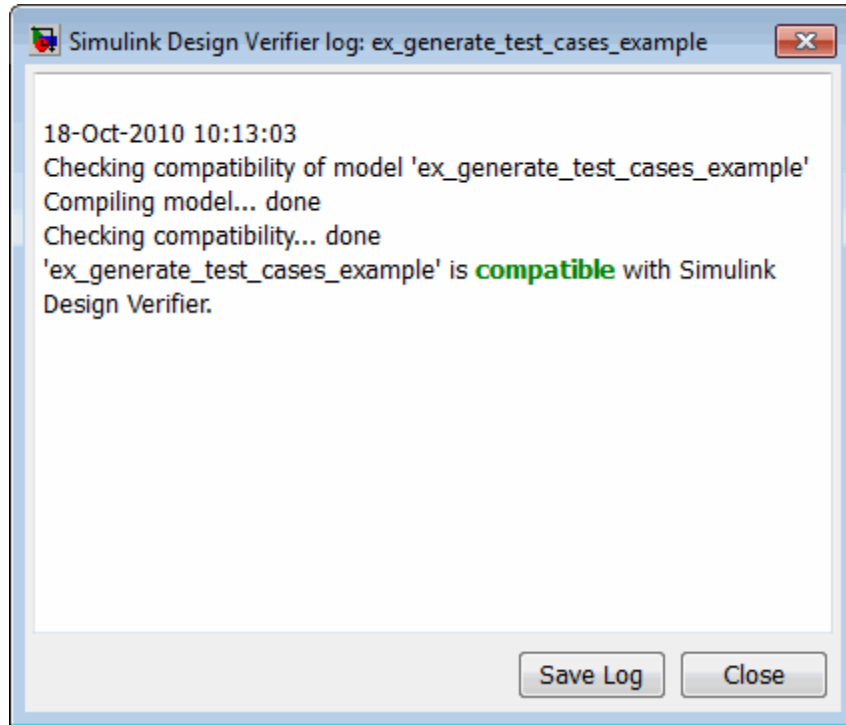
Every time Simulink Design Verifier analyzes a model, before the analysis begins, the software performs a compatibility check. If your model is not compatible, the software cannot analyze it.

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

- 1 Open the `ex_generate_test_cases_example` model.
- 2 In the model window, select **Tools > Design Verifier > Check Model Compatibility**.

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

The model you just created is compatible.



### What If a Model Is Partially Compatible?

If the compatibility check indicates that your model is partially compatible, your model contains at least one object that the Simulink Design Verifier software does not support. You can analyze a partially compatible model, but, by default, the unsupported objects are stubbed out. The results of the analysis might be incomplete.

For detailed information about automatic stubbing, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

### Configuring Test Generation Options

Configure the Simulink Design Verifier software to generate test cases that achieve 100% decision coverage for the `ex_generate_test_cases_example` model:

- 1** Open the `ex_generate_test_cases_example` model.
- 2** In the model window, select **Tools > Design Verifier > Options**.
- 3** On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Design Verifier** category. Under **Analysis options**, set the **Mode** option to **Test generation**.
- 4** On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Test Generation** category.
- 5** On the **Test Generation** pane, set the **Model coverage objectives** parameter to **Decision**.

For this example, the analysis generates test cases that record only decision coverage.

---

**Note** The **Test suite optimization** parameter is set by default to **CombinedObjectives**. If you want to generate fewer but longer test cases, select **LongTestcases** for the **Test suite optimization** parameter.

---

- 6** Click **OK** to save your changes and close the Configuration Parameters dialog box.
- 7** Save the `ex_generate_test_cases_example` model.

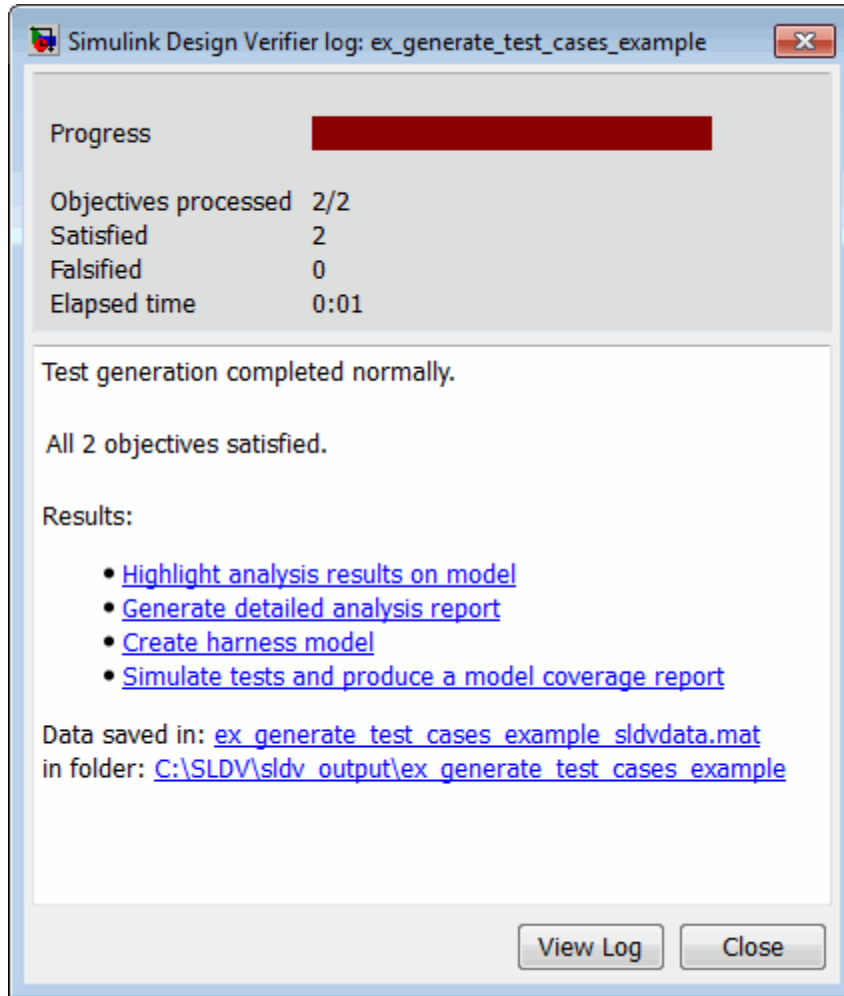
## Analyzing the Example Model

To analyze the `ex_generate_test_cases_example` model, in the model window, select **Tools > Design Verifier > Generate Tests**. The Simulink Design Verifier software begins analyzing your model to generate test cases.

During the analysis, the log window shows the progress of the analysis. It displays information such as the number of test objectives processed and which objectives are satisfied.

## Reviewing the Analysis Results

When the software completes its analysis, the log window displays the following options for reviewing the results:



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

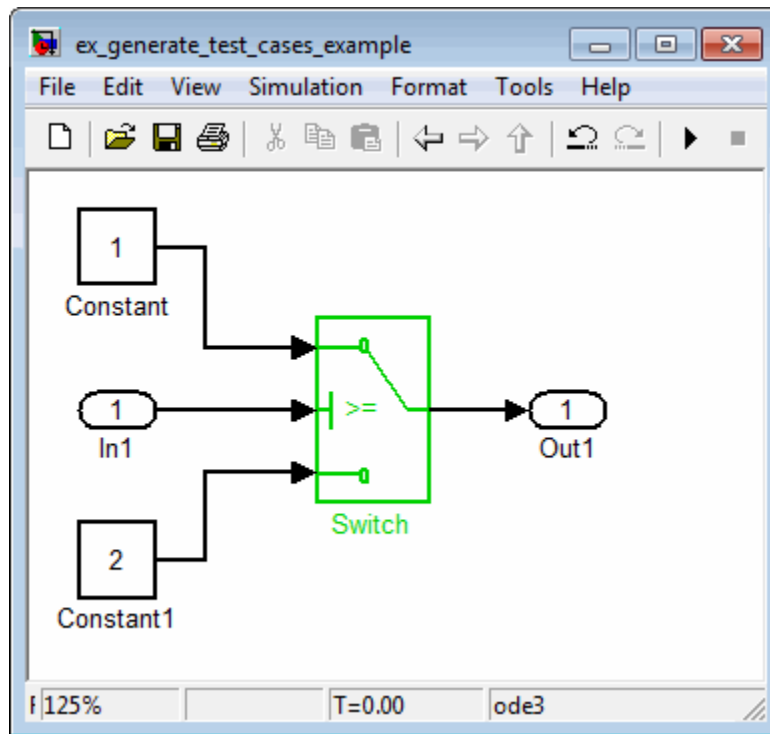
- “Reviewing the Results on the Model” on page 7-11

- “Reviewing the Detailed Analysis Report” on page 7-13
- “Reviewing the Harness Model” on page 7-14
- “Simulating Tests and Producing a Model Coverage Report” on page 7-16
- “Viewing the Data File” on page 7-17
- “Reviewing Analysis Results in the Model Explorer” on page 7-17

## Reviewing the Results on the Model

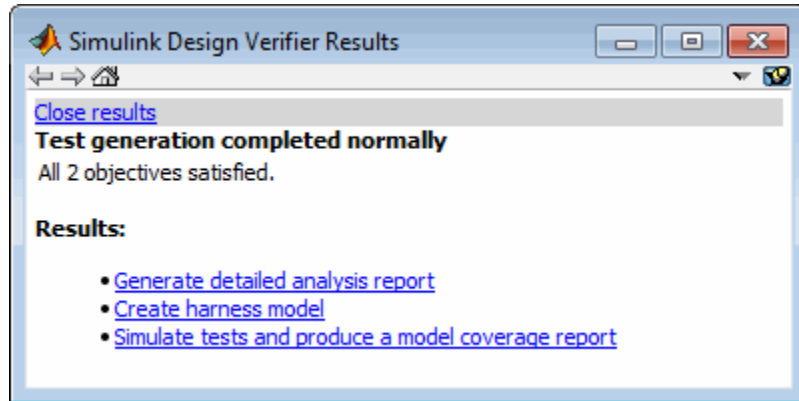
Highlight the analysis results on the example model:

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




The Switch block is outlined in green, which indicates that the Switch block has test cases that satisfy its test objectives.

The Simulink Design Verifier Results window appears. As you click objects in the model, this window changes to display detailed analysis results for that object.



---

**Tip** By default, the Simulink Design Verifier Results window is always the topmost visible window. To allow the window to move behind other window, click  and clear **Always on top**.

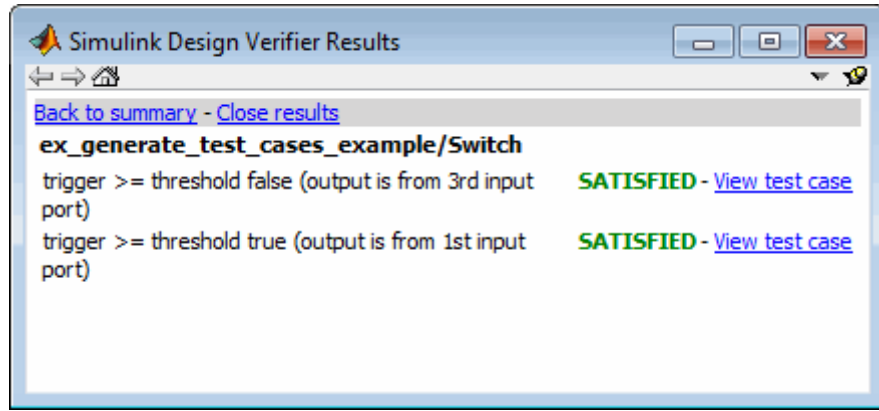
---

### 2 Click the highlighted Switch block.

The Simulink Design Verifier Results window indicates that the analysis generated test cases for both test objectives:

- trigger >= threshold
- trigger < threshold





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

## Reviewing the Detailed Analysis Report

Create a detailed HTML analysis report:

- 1 In the Simulink Design Verifier log window, click **Generate detailed analysis report**.

The HTML report opens in a browser window.

- 2 The report includes the following **Table of Contents**. Click a hyperlink to navigate to a section in the report.

Table of Contents
<a href="#">1. Summary</a>
<a href="#">2. Analysis Information</a>
<a href="#">3. Test Objectives Status</a>
<a href="#">4. Model Items</a>
<a href="#">5. Test Cases</a>

- 3 In the **Table of Contents**, click Summary to display the report’s Summary chapter.

The Summary chapter lists information about the model and the status of the objectives—satisfied or not.

- 4 In the **Table of Contents**, click **Analysis Information** to display the Analysis Information chapter.

The Analysis Information chapter provides information about:

- The model that you analyzed
- The options that you specified for the analysis
- Approximations the software performed during the analysis

- 5 In the **Table of Contents**, click **Test Objectives Status** to display the report's Test Objectives Status chapter.

This table indicates that the analysis satisfied both test objectives associated with the Switch block in the `ex_generate_test_cases_example` model, for which it generated two test cases.

- 6 Under the table **Test Case** column, click 2 to display the Test Case 2 section.

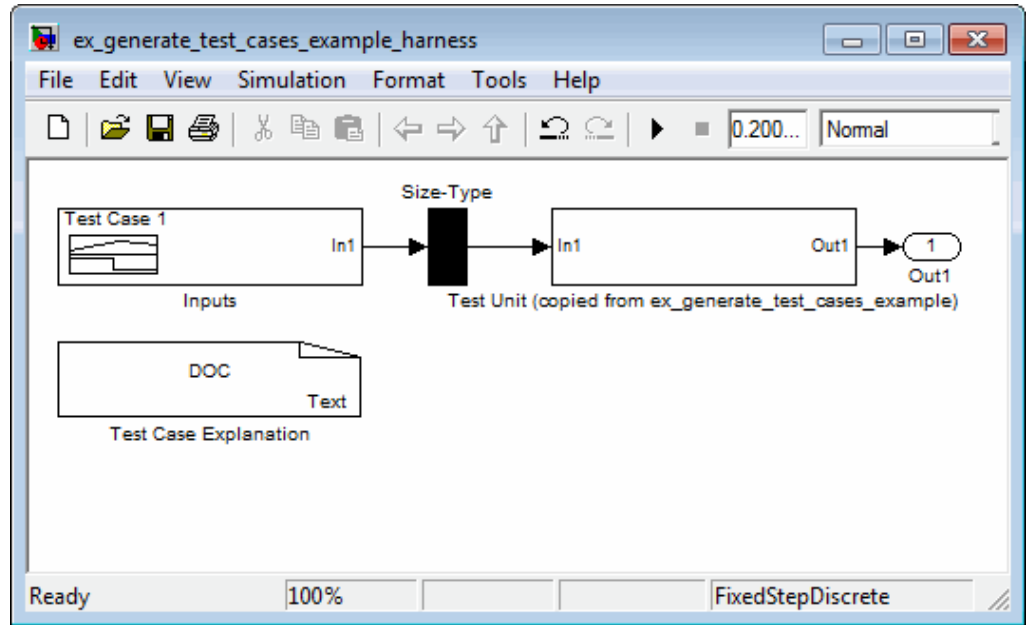
This section provides details about a test case that the analysis generated to achieve an objective in your model. This test case achieves test objective 1, when the Switch block passes its third input to its output port. Specifically, the software determines that a value of  $-1$  for the Switch block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

For more information about the HTML reports, see “Simulink® Design Verifier™ Reports” on page 13-27.

### Reviewing the Harness Model

To create a harness model with test cases that satisfy the test objectives in your model, in the Simulink Design Verifier log window, click **Create harness model**.

The software creates a harness model named `ex_generate_test_cases_example_harness.mdl`.



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 “Simulating Tests and Producing a Model Coverage Report” on page 7-16.

For more information about the harness model, see “Harness Model” on page 13-15.

**If the Analysis Generates Many Test Cases.** If you have a large model, the analysis might produce a harness model that contains a large number of test cases.

To perform a more efficient analysis and create easier-to-review results that are combined into a smaller number of longer test cases:

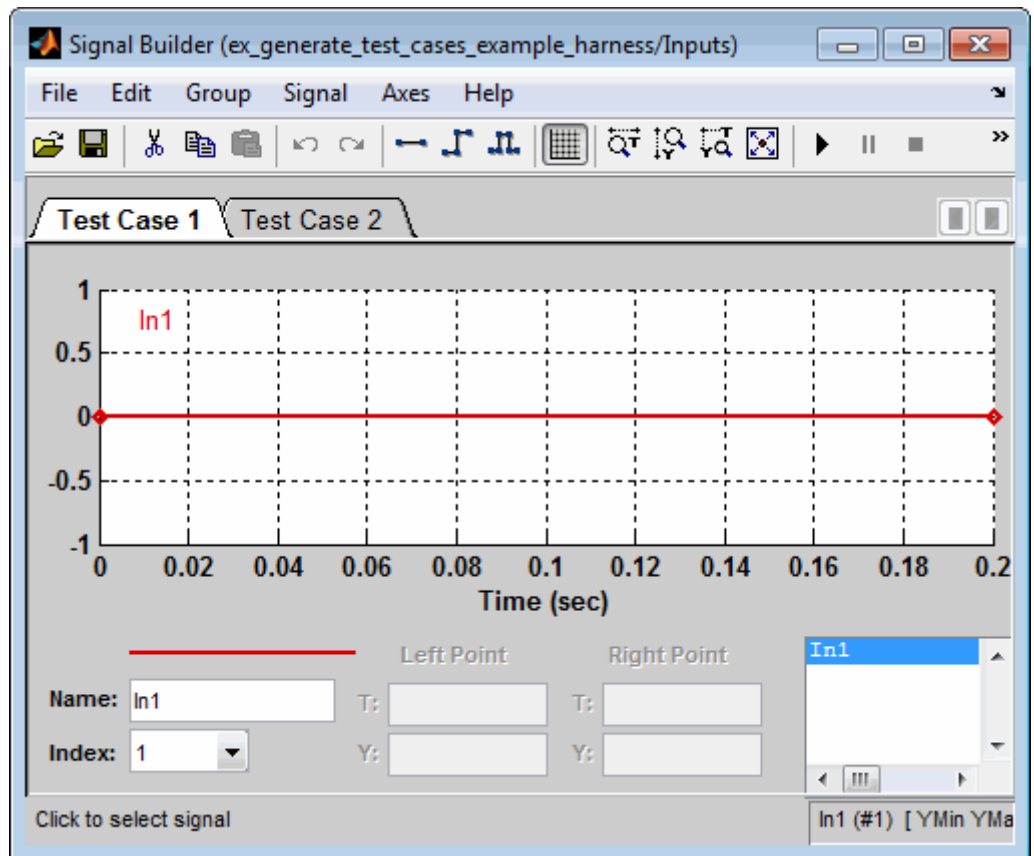
- 1 Set the **Test suite optimization** parameter to LongTestcases.
- 2 Rerun the analysis.


In the LongTestcases optimization, the analysis generates fewer but longer test cases that each satisfy multiple test objectives.

### Simulating Tests and Producing a Model Coverage Report

To simulate the harness model using the generated test cases in the harness model:

- 1 In the harness model, double-click the Inputs block to open the Signal Builder dialog box.



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

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 “Model Coverage Analysis” in the *Simulink Verification and Validation User’s Guide*.

## Viewing the Data 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. 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.

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.

## Reviewing Analysis Results in the Model Explorer

If you close the analysis results so you review any unsatisfiable objectives, you may need to review the analysis results again. As long as your model remains open, you can view the results of your most recent Simulink Design Verifier analysis results in the Model Explorer. After you close your model, you can no longer view any analysis results.

In the model window, select **Tools > Design Verifier > Latest Results**. The Model Explorer opens, and the results of the latest Simulink Design Verifier analysis appear in the right-hand pane.

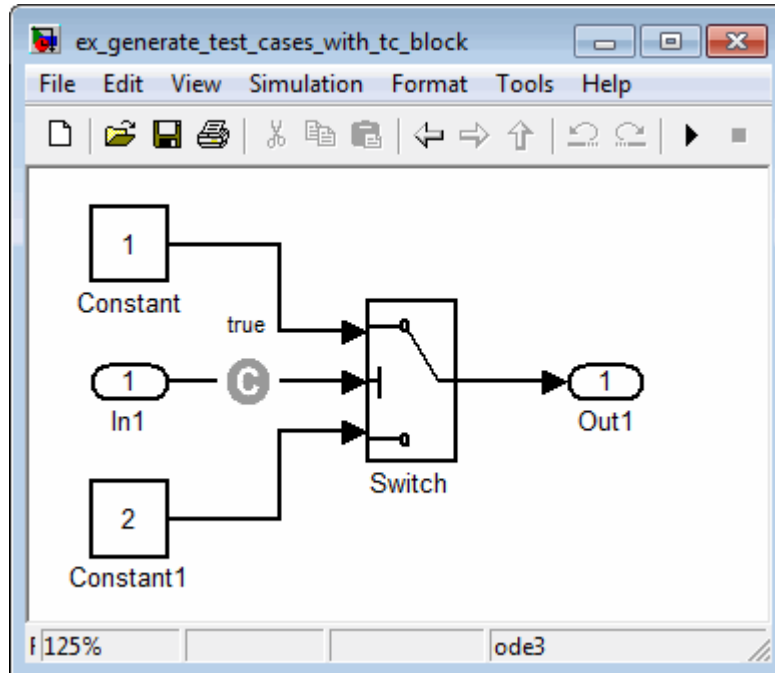
For any Simulink Design Verifier analysis, from the Model Explorer, you can perform any of the following tasks.

<b>Task</b>	<b>For more information</b>
Highlight the analysis results on the model.	“Highlighted Results on the Model” on page 13-2
Generate a detailed analysis report.	“Simulink® Design Verifier™ Reports” on page 13-27
Create the harness model, or if the harness model already exists, open it.  If no test cases were generated during the analysis, this option is not available.	“Harness Model” on page 13-15
View the data file.	“Simulink® Design Verifier™ Data Files” on page 13-7
View the log file.	“Simulink® Design Verifier™ Log Files” on page 13-52

## Customizing Test Generation

Customize the test-generation analysis by using the Test Condition block to constrain signals in your model to certain values during the analysis.

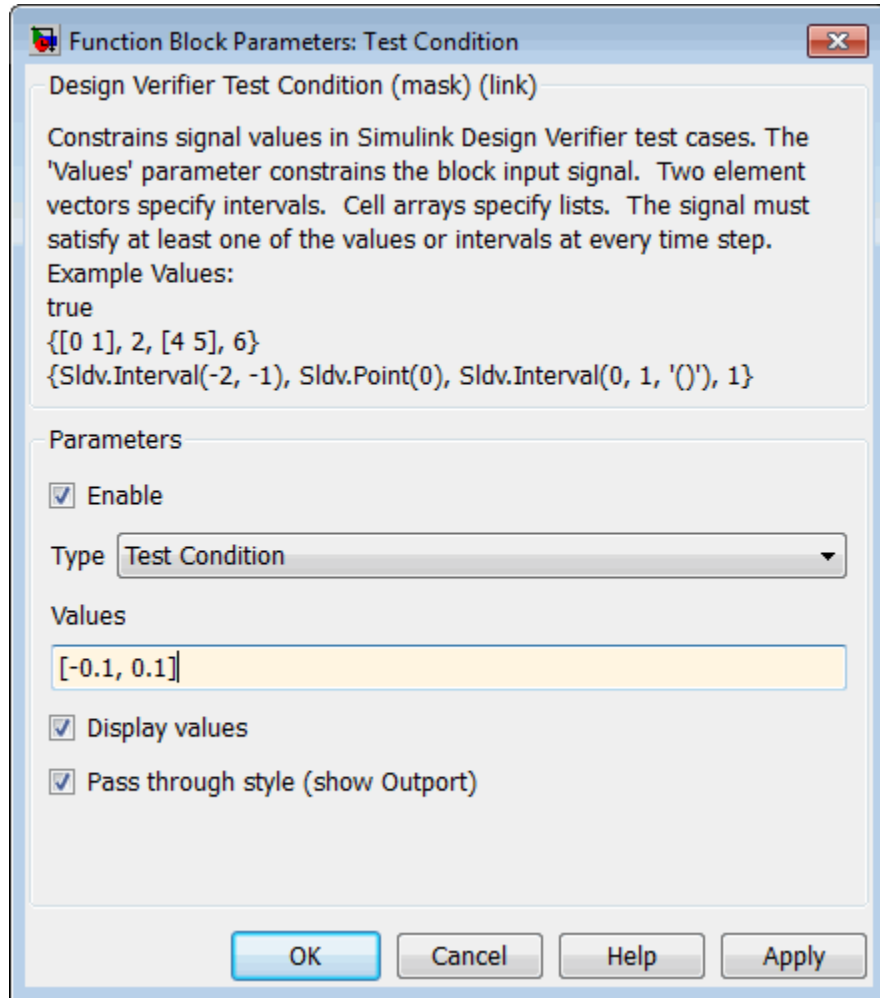
- 1** In the MATLAB Command Window, enter `sldvlib` to display the Simulink Design Verifier library.
- 2** Open the Objectives and Constraints sublibrary.
- 3** Copy the Test Condition block to your model by dragging it from the Simulink Design Verifier library to your model window.
- 4** In the model 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.



- 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.



## Reanalyzing 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 “Analyzing the Example Model” on page 7-9.

- 1** In the model window, select **Tools > Design Verifier > 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 log window displays the options for reviewing the results.

- 2** In the Simulink Design Verifier log window, click **Generate detailed analysis report**.

- 3** To begin reviewing the report, in the **Table of Contents**, click **Summary**.

The Summary chapter indicates that the Simulink Design Verifier software 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 the Simulink Design Verifier software satisfied both test objectives associated with the Switch block in your model, for which it generated two test cases.

- 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 the **Run all** button 

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 the Simulink Design Verifier software generated test cases that achieve complete decision coverage for your example model.

### Analyzing 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 might 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.

## Generating Test Cases for a Subsystem

If you have a large model, you can generate test cases for subsystems in the model and review the analysis in smaller, manageable reports. The workflow for generating test cases for a subsystem is:

- 1** Open the model that contains the subsystem.
- 2** Make the subsystem atomic.
- 3** Run the Simulink Design Verifier software using the **Generate Tests for Subsystem** option.
- 4** Review the results.

The tutorial in “Analyzing a Subsystem” on page 1-29 describes how to analyze the Controller subsystem in the Cruise Control Test Generation model.



# Extending Existing Test Cases

---

- “When to Extend Existing Test Cases” on page 8-2
- “Common Workflow for Extending Existing Test Cases” on page 8-3
- “Example: Extending Existing Test Cases for a Model that Uses Temporal Logic” on page 8-4
- “Example: Extending Existing Test Cases for a Closed-Loop System” on page 8-11
- “Example: Extending Existing Test Cases for a Modified Model” on page 8-14

## When to Extend Existing Test Cases

The Simulink Design Verifier software can analyze your model, using any previously generated test cases that you specify. You can use this feature with the following situations:

- You encounter delays trying to analyze your model, or you see incomplete results. These situations can arise 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 “Example: Extending Existing Test Cases for a Model that Uses 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 “Example: Extending Existing Test Cases for a Closed-Loop System” on page 8-11.

- 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 “Example: Extending Existing Test Cases for a Modified Model” on page 8-14.

## 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 following examples use some or all of these tasks when extending existing test cases during analysis.

## Example: Extending Existing Test Cases for a Model that Uses Temporal Logic

### In this section...

“Creating a Starting Test Case” on page 8-4

“Logging the Starting Test Case” on page 8-7

“Extending the Existing Test Cases” on page 8-8

“Verifying the Analysis Results” on page 8-10

### Creating a 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.

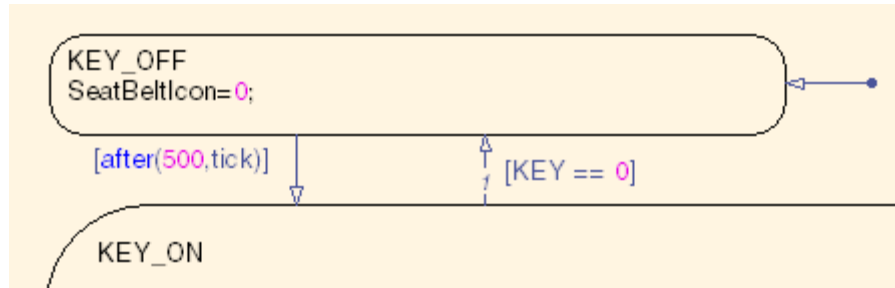
In this example, you create a test case that forces the transition to `KEY_ON` by setting the `KEY` input to 1 for the duration of the test case. You simulate the model using this test case, satisfying the objectives for the `KEY_OFF/KEY_ON` transition. Then you analyze the model, ignoring the objectives already satisfied by the test case you create.

**1** Open the demo model:

```
sldvdemo_sbr_extend_design
```

**2** Open the SBR Stateflow chart to see the `KEY_OFF/KEY_ON` transition.



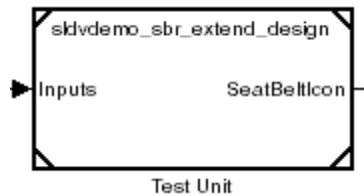


**3** Create a model reference harness model:

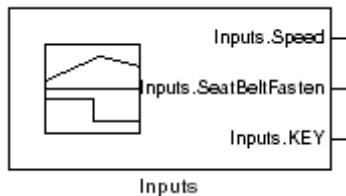
```
[-, harnessModelFilePath] = sldvmakeharness('sldvdemo_sbr_extend_design',[],[],true);
```

The harness model, `sldvdemo_sbr_extend_design_harness`, includes:

- A Model block named Test Unit that references the original model, `sldvdemo_sbr_extend_design`.

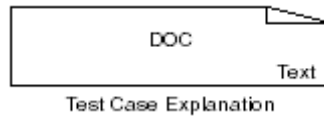


- A Signal Builder block named Inputs that contains the test-case inputs to the model referenced in the Model block.



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.



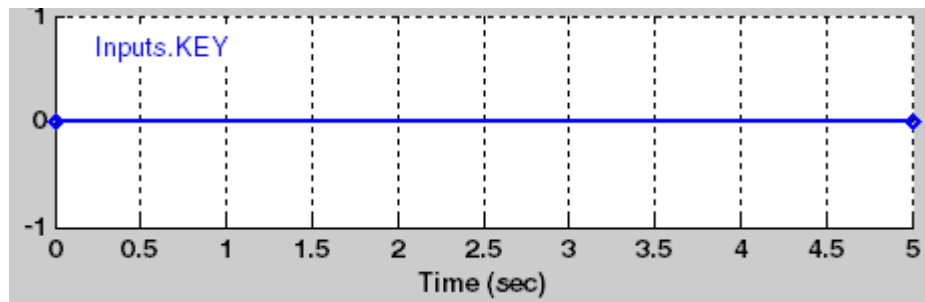
Initially, the Test Case Explanation block does not have any content for the default test case.

- 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:

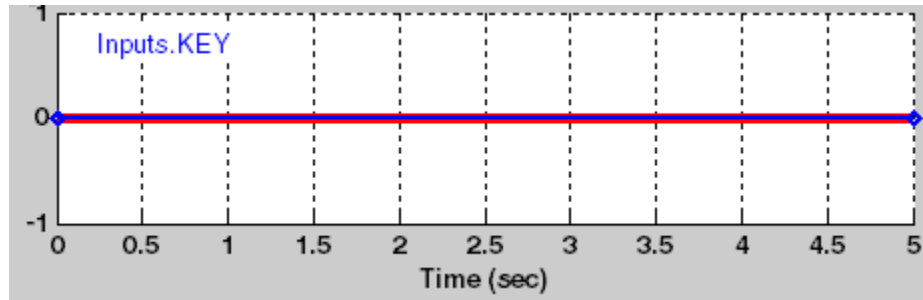
```
[~, harnessModel] = fileparts(harnessModelFilePath);
```

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.

## Logging the Starting Test Case

The next step is to log the starting test case that you created. You can then specify that the Simulink Design Verifier software 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_mdirefs(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 = sldvlogssignals(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.

### Extending the Existing Test Cases

You can now analyze the `sldvdemo_sbr_extend_design` model and specify that the analysis extend the test cases already satisfied. The analysis uses the existing test-case data as a starting point, and does not try to generate test cases for the `KEY_OFF` to `KEY_ON` transition in the SBR Stateflow chart.

Specify the starting test case and analyze the model:

- 1 In the model window for `sldvdemo_sbr_extend_design`, select **Tools > Design Verifier > Options**.
- 2 In the Configuration Parameters dialog box, on the **Select** pane, under **Design Verifier**, select **Test Generation**.
- 3 On the **Test Generation** pane, under **Existing test cases**, select **Extend existing test cases**.
- 4 In the **Data file** field, enter the name of the MAT-file that contains the logged data:

```
existingtestcase.mat
```

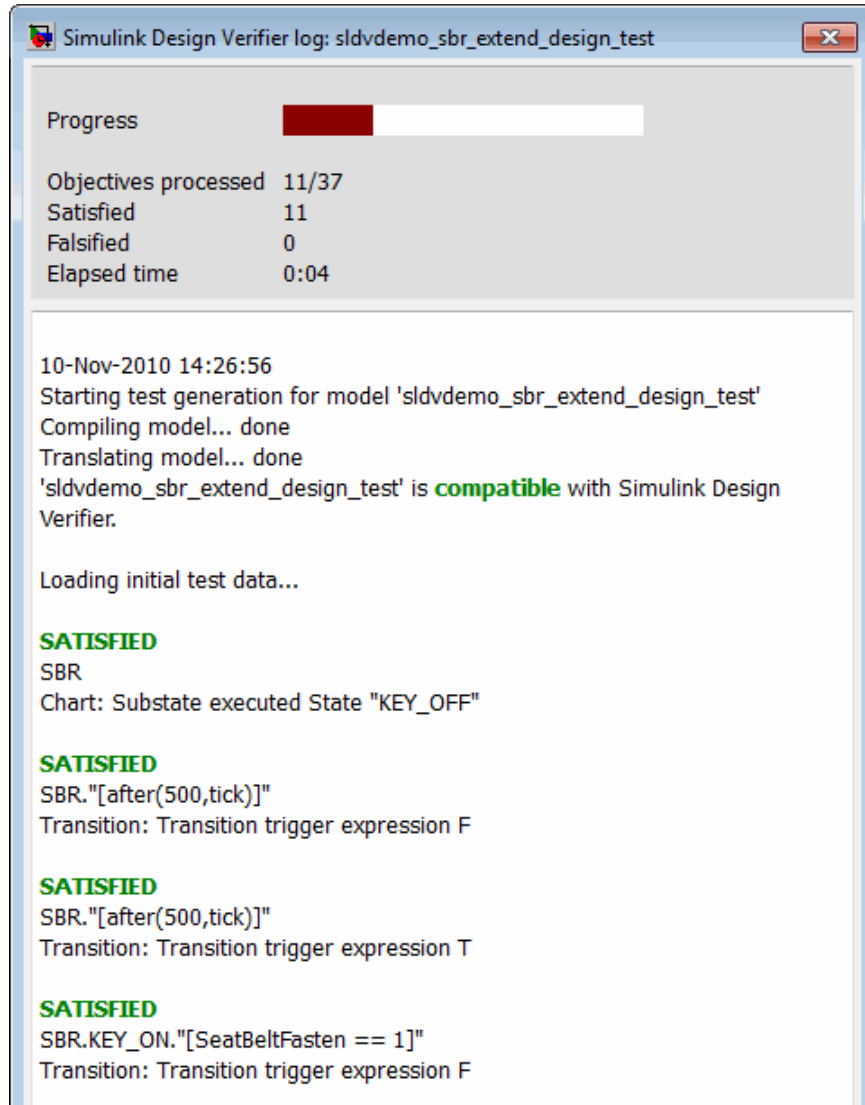
- 5 Clear **Ignore objectives satisfied by existing test cases**.

When you clear this option, the software includes the starting test case in the final test suite. You will see that the complete test suite achieves 100% model coverage.

- 6 To close the Configuration Parameters dialog box, click **OK**.
- 7 Save the `sldvdemo_sbr_extend_design` model on the MATLAB path with the name `sldvdemo_sbr_extend_design_test`.

- 8 In the Model Editor, select **Tools > Design Verifier > Generate Tests**.


The log window first lists the objectives that the starting test case satisfied.



The log window then lists the objectives generated beyond the starting test case.

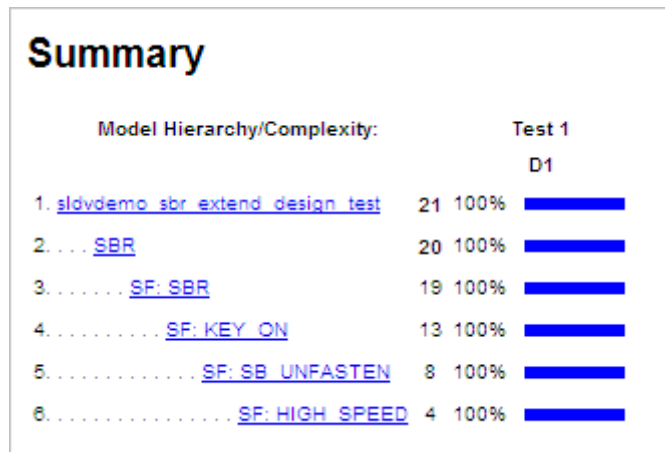
## Verifying the Analysis Results

To make sure that this analysis creates a complete test suite, generate the harness model so you can simulate the model with the generated test cases:

- 1 In the log window, click **Create harness model**.
- 2 In the harness model `sldvdemo_sbr_extend_design_test_harness`, open the Signal Builder block named Inputs.
- 3 To simulate the model using all the test cases, click the **Run all and produce coverage** button .

When the simulation is complete, the model coverage report is displayed.

- 4 View the coverage information for the `sldvdemo_sbr_extend_design_test` model to see that the complete test suite achieves 100% coverage.



## Example: Extending Existing Test Cases for a Closed-Loop System

### In this section...

“Logging a Starting Test Case” on page 8-11

“Extending the Existing Test Cases” on page 8-12

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.

### Logging a Starting Test Case

This example uses the `sldemo_md1ref_bus` model. The CounterA Model block references the model `sldemo_md1ref_counter_bus`. If you simulate the parent model, `sldemo_md1ref_bus`, and collect coverage, you record only 75% coverage for `sldemo_md1ref_counter_bus`. Log the data from the simulation and extend those test cases to achieve 100% coverage for the referenced model.

To create the starting test case, simulate the top-level model and log the input signals to the Model block:

- 1 Open the demo model:

```
sldemo_md1ref_bus
```

- 2 Simulate the `sldemo_md1ref_bus` model and log the input signals for the CounterA Model block:

```
logged_data = sldvlogssignals('sldemo_md1ref_bus/CounterA');
```

- 3 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\_md1ref\_counter\_bus), you specify this MAT-file.

### Extending the Existing Test Cases

Analyze the `sldemo_md1ref_counter_bus` model, specifying that the analysis extend the test cases already satisfied:

- 1 To open the `sldemo_md1ref_counter_bus` model, in the `sldemo_md1ref_bus` model, double-click the CounterA Model block.
- 2 In the Model Editor for `sldemo_md1ref_counter_bus`, select **Tools > Design Verifier > Options**.
- 3 In the Configuration Parameters dialog box, on the **Select** pane, under **Design Verifier**, select **Test Generation**.
- 4 On the **Test Generation** pane, 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 To save these settings, click **Apply**.
- 8 To open the main **Design Verifier** pane, on the **Select** pane, click **Design Verifier**.
- 9 To start the analysis, click **Generate Tests**.

The analysis first loads the nine objectives satisfied by the logged test cases. Then it adds extra time steps to those test cases and tries to satisfy



any missing objectives. In this example, the analysis satisfies three additional objectives.

- 10 To verify the results of the analysis, review the Simulink Design Verifier report. The analysis found test cases that satisfy all 12 test objectives for the referenced model `sldemo_md1ref_counter_bus`.

## Objectives Satisfied

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

#	Type	Model Item	Description	Test Case
1	Decision	<a href="#">Switch</a>	logical trigger input false (output is from 3rd input port)	<a href="#">2</a>
2	Decision	<a href="#">Switch</a>	logical trigger input true (output is from 1st input port)	<a href="#">2</a>
3	Decision	<a href="#">limit</a>	logical trigger input false (output is from 3rd input port)	<a href="#">2</a>
4	Decision	<a href="#">limit</a>	logical trigger input true (output is from 1st input port)	<a href="#">2</a>
5	Condition	<a href="#">And</a>	Logic: input port 1 T	<a href="#">2</a>
6	Condition	<a href="#">And</a>	Logic: input port 1 F	<a href="#">2</a>
7	Condition	<a href="#">And</a>	Logic: input port 2 T	<a href="#">2</a>
8	Condition	<a href="#">And</a>	Logic: input port 2 F	<a href="#">2</a>
9	Mcdc	<a href="#">And</a>	Logic: MDCD expression for output with input port 1 T	<a href="#">2</a>
10	Mcdc	<a href="#">And</a>	Logic: MDCD expression for output with input port 2 T	<a href="#">2</a>
11	Mcdc	<a href="#">And</a>	Logic: MDCD expression for output with input port 1 F	<a href="#">2</a>
12	Mcdc	<a href="#">And</a>	Logic: MDCD expression for output with input port 2 F	<a href="#">2</a>

## Example: Extending Existing Test Cases for a Modified Model

<b>In this section...</b>
“Creating Starting Test Cases” on page 8-14
“Extending the Existing Test Cases” on page 8-15

Suppose that you have a model that you have already analyzed using the Simulink Design Verifier software, 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.

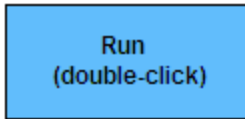
### Creating Starting Test Cases

Analyze the `sldvdemo_cruise_control` model and generate test cases that achieve 100% coverage.

- 1 Open the demo model:

```
sldvdemo_cruise_control
```

- 2 To start a Simulink Design Verifier analysis for the `sldvdemo_cruise_control` model, double-click the Run Simulink Design Verifier block:



### Run Simulink Design Verifier

The analysis satisfies 34 test objectives for the `sldvdemo_cruise_control` model. The software stores the resulting data file in a subfolder of the MATLAB Current Folder:

```
sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_sldvdata.mat
```

In the next section, when you analyze the modified model, this data file specifies the starting test cases that you extend.

- 3 Close the `sldvdemo_cruise_control` model and all the files created by the analysis. If asked, do not save any changes you made to the model.

## Extending the Existing Test Cases

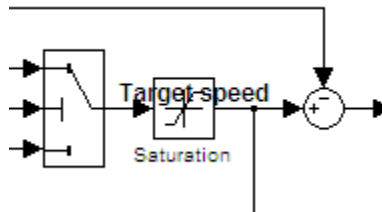
The `sldvdemo_cruise_control_mod` model is a modified version of `sldvdemo_cruise_control`. The Controller subsystem contains a Saturation block that specifies that the target speed cannot exceed 70.

Open the modified model and analyze it, extending the test cases that you generated when analyzing the `sldvdemo_cruise_control` model:

- 1 Open the demo model, the modified version of `sldvdemo_cruise_control`:

```
sldvdemo_cruise_control_mod
```

- 2 Double-click the Controller subsystem to see the change to the original model, a Saturation block that specifies the maximum speed:



- 3** Close the Controller subsystem.
- 4** Select **Tools > Design Verifier > Options**.
- 5** In the Configuration Parameters dialog box, on the **Select** pane, under **Design Verifier**, select **Test Generation**.
- 6** On the **Test Generation** pane, under **Existing test cases**, select **Extend existing test cases**.
- 7** In the **Data file** field, click **Browse** and navigate to the MAT-file created in the MATLAB Current Folder when analyzing the original model:  

```
sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_sldvdata.mat
```
- 8** Clear **Ignore objectives satisfied by existing test cases**.  

When you clear this option, the analysis includes the test cases recorded in the file `sldvdemo_cruise_control_sldvdata.mat` in the final test suite.
- 9** Click **Apply** to save these settings.
- 10** To open the main **Design Verifier** pane, on the **Select** pane, click **Design Verifier**.
- 11** To start the analysis, click **Generate Tests**.

The analysis first loads the 34 objectives satisfied by the initial test cases. Then it adds extra time steps to those test cases and tries to satisfy any missing objectives.

- 12** In the log window, click **Generate detailed analysis report**.

The analysis satisfied a total of 38 satisfied objectives for the `sldvdemo_cruise_control_mod` model. The analysis satisfied four additional objectives that correspond to the Saturation block.

## Objectives Satisfied

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

#	Type	Model Item	Description	Test Case
1	Decision	<a href="#">Controller/Switch1</a>	logical trigger input false (output is from 3rd input port)	<a href="#">3</a>
2	Decision	<a href="#">Controller/Switch1</a>	logical trigger input true (output is from 1st input port)	<a href="#">1</a>
3	Decision	<a href="#">Controller/Saturation</a>	input > lower limit F	<a href="#">1</a>
4	Decision	<a href="#">Controller/Saturation</a>	input > lower limit T	<a href="#">3</a>
5	Decision	<a href="#">Controller/Saturation</a>	input >= upper limit F	<a href="#">1</a>
6	Decision	<a href="#">Controller/Saturation</a>	input >= upper limit T	<a href="#">10</a>



# Achieving Test Cases for Missing Model Coverage

---

- “Generating Test Cases for Missing Coverage Data” on page 9-2
- “Example: Achieving Missing Coverage in a Referenced Model” on page 9-3
- “Achieving Missing Coverage for Subsystems and Model Blocks” on page 9-7
- “Example: Achieving Missing Coverage in a Closed-Loop Simulation Model” on page 9-8

### Generating Test Cases for Missing Coverage Data

If you simulate your model and record coverage data, but your model does not achieve 100% coverage, the Simulink Design Verifier software 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:

- “Example: Achieving Missing Coverage in a Referenced Model” on page 9-3
- “Example: Achieving Missing Coverage in a Closed-Loop Simulation Model” on page 9-8



## Example: Achieving Missing Coverage in a Referenced Model

### In this section...

“Recording Coverage Data for the Model” on page 9-3

“Finding Test Cases for the Missing Coverage” on page 9-5

“Achieving the Missing Coverage” on page 9-5

“Verifying 100% Model Coverage” on page 9-6

This example uses a demo model with a referenced model that does not achieve full coverage. When you run a test-generation analysis on the referenced model, and combine it with the previously recorded coverage data, you can achieve 100% for the referenced model.

### Recording Coverage Data for the Model

Simulate the model, recording Condition, Decision, and MCDC coverage:

- 1 Open the demo model:

```
sldemo_md1ref_basic
```

The Model blocks CounterA, CounterB, and CounterC reference the model `sldemo_md1ref_counter`.

- 2 To specify the coverage options that you want, select **Tools > Coverage Settings**.

The Coverage Settings dialog box opens to the **Coverage** tab.

- 3 On the **Coverage** tab, set the following options:

- To specify that the simulation record coverage for the referenced model and the parent model, select **Coverage for referenced models**.
- Click **Select Models**, then select the check boxes for the referenced model, `sldemo_md1ref_counter`.

Check boxes for `sldemo_md1ref_counter` appear twice, corresponding to CounterA and CounterB. Coverage cannot be enabled for CounterC since it specifies the Accelerator simulation mode in its model reference parameters.

- To specify which types of coverage to record during simulation, under **Coverage metrics**, select:
  - **Decision**
  - **Condition**
  - **MCDC**

**4** Click the **Results** tab.

**5** To specify a unique name for the coverage data workspace variable, in the **cvdata object name** field, enter `covdata_original`.

**6** Click the **Reporting** tab.

**7** To specify that the simulation create a coverage report, select **Generate HTML report**.

**8** To save the settings and close the Coverage Settings dialog box, click **OK**.

**9** Start the simulation of the `sldemo_md1ref_basic` model to record the coverage data.

After the simulation, the coverage report opens. The report indicates that the following coverage is achieved for the referenced model `sldemo_md1ref_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.

**10** Save the coverage data in a file on the MATLAB path:

```
cvsave('existingcov',covdata_original);
```

## Finding Test Cases for the Missing Coverage

To achieve 100% coverage for the `sldemo_md1ref_counter` model, run a test-generation analysis that uses the existing coverage data:

- 1 Open the referenced model using the following command:

```
open_system('sldemo_md1ref_counter');
```

- 2 Create an `sldvoptions` object:

```
opts = sldvoptions;
```

To create the `sldvoptions` object, you need to specify:

- That the analysis ignore satisfied coverage data.
- The file name containing the satisfied coverage data (`existingcov.cvt`)

To specify these options, enter the following commands:

```
opts.IgnoreCovSatisfied = 'on';  
opts.CoverageDataFile = 'existingcov.cvt';
```

- 3 Analyze the referenced model, `sldemo_md1ref_counter`, using the specified options:

```
[status, fileNames] = sldvrun('sldemo_md1ref_counter',opts,true);
```

The Simulink Design Verifier analysis satisfies three additional objectives and creates one test case for the referenced model.

The next section simulates the referenced model, `sldemo_md1ref_counter`, using the test case that the analysis created.

## Achieving the Missing Coverage

To achieve the missing coverage for the referenced model, `sldemo_md1ref_counter`, simulate the model using the test case from the Simulink Design Verifier analysis:

- 1 Create a `cvtest` object for the simulation and specify to record Decision, Condition, and MCDC coverage:

```
cvt = cvtest('sldemo_md1ref_counter');  
cvt.settings.decision = 1;  
cvt.settings.condition = 1;  
cvt.settings.mcdc = 1;
```

- 2 Before you simulate the model using `sldvrntest`, specify to record coverage and set the name of the `cvtest` object:

```
runOpts = sldvrntestopts;  
runOpts.coverageEnabled = true;  
runOpts.coverageSetting = cvt;
```

- 3 Simulate the model using 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] = sldvrntest('sldemo_md1ref_counter', fileNames.DataFile, runOpts);
```

### Verifying 100% Model Coverage

You saved the coverage data from the simulation of the top-level model, `sldemo_md1ref_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_md1ref_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% coverage.

## Achieving Missing Coverage for 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. “Converting a Subsystem to a Referenced Model” describes how to convert a subsystem to a referenced model. You can then follow the steps described in “Example: Achieving Missing Coverage in a Referenced Model” on page 9-3.

However, some subsystems cannot be converted to Model blocks. To test a subsystem to see if it can be converted to a Model block, use the `Simulink.SubSystem.convertToModelReference` function. If that function cannot convert the subsystem, an error message explains why the conversion failed.

In addition, you may have a Stateflow chart or a MATLAB Function block that does not achieve full coverage. Stateflow charts and MATLAB Function blocks cannot be converted to referenced models.

For situations when you cannot use a Model block, follow steps similar to the steps described in the following section, “Example: Achieving Missing Coverage in a Closed-Loop Simulation Model” on page 9-8.

## Example: Achieving Missing Coverage in a Closed-Loop Simulation Model

In this section...
“Recording Coverage Data for the Model” on page 9-8
“Finding Test Cases for Missing Coverage” on page 9-10

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, this example can help you 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.

### Recording Coverage Data for the Model

To simulate the model, recording Condition, Decision, and MCDC coverage for the ShiftLogic controller:

- 1 Open the demo model:

```
sldvdemo_autotrans
```

- 2 To designate the coverage settings that you want, select **Tools > Coverage Settings**.

The Coverage Settings dialog box opens at the **Coverage** tab.

- 3 To specify that the simulation record coverage, select **Coverage for this model: sldvdemo\_autotrans**.

- 4** To specify to record coverage for the ShiftLogic chart, click **Select Subsystem**.
- 5** In the Subsystem Selection dialog box, select ShiftLogic and click **OK**.
- 6** Under **Coverage Metrics**, select the types of coverage that you want the simulation to record:
  - **Decision**
  - **Condition**
  - **MCDC**
- 7** Clear the other coverage metrics if they are checked.
- 8** Click the **Results** tab.
- 9** To specify a unique name for the coverage data workspace variable, in the **cvdata object name** field, enter `covdata_original_controller`.
- 10** Click the **Reporting** tab.
- 11** To specify that the simulation create a coverage report, select **Generate HTML report**.
- 12** To save these settings and close the Coverage Settings dialog box, click **OK**.
- 13** 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%
- Condition: 67%
- MCDC: 33%

The simulation saves the coverage data in the MATLAB workspace variable `covdata_original_controller`, a `cvtest` object that contains the coverage data.

- 14** Save the coverage data in a file on the MATLAB path:

```
cvsave('existingcov',covdata_original_controller);
```

### Finding 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 To analyze the ShiftLogic chart, the model must use a fixed-step solver. Select **Simulation > Configuration Parameters** and on the **Solver** pane:
  - For **Type**, select Fixed-step.
  - For **Solver**, select discrete (no continuous steps).
- 2 To analyze just the ShiftLogic block and specify that the analysis ignore the coverage data already recorded, right-click the ShiftLogic block and select **Design Verifier > Options**.
- 3 In the Configuration Parameters dialog box, on the **Select** pane, under the **Design Verifier** category, select **Test Generation**.
- 4 On the **Test Generation** pane, under **Existing coverage data**, select **Ignore objectives satisfied in existing coverage data**.
- 5 In the **Coverage data file** field, enter the name of the file containing the coverage data that you recorded during simulation:  

```
existingcov.cvt
```
- 6 To save these settings, click **Apply**.
- 7 On the **Select** pane, click **Design Verifier**.
- 8 On the main **Design Verifier** pane, click **Generate Tests for Subsystem**.

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.



**9** When the test-generation analysis is complete, select **View detailed analysis report**.

The Simulink Design Verifier report lists six test cases for the extracted model that satisfy the objectives not covered in the `existingcov.cvt` file.

---

**Note** The Simulink Design Verifier report indicates that two coverage objectives in the Stateflow chart `ShiftLogic` are proven unsatisfiable. Since the `ShiftLogic` chart updates at every time step, the implicit event `tick` is never `false`. 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]
```

the Simulink Design Verifier software 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)`.

---



# Verifying Model Components

---

- “What Is Component Verification?” on page 10-2
- “Functions for Component Verification” on page 10-4
- “Example: Verifying a Component for Code Generation” on page 10-6

## What Is Component Verification?

### In this section...

“Component Verification Approaches” on page 10-2

“Using Simulink® Design Verifier™ Tools for Component Verification” on page 10-2

### 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.

### Using Simulink Design Verifier Tools for Component Verification

By isolating the component to verify, and using tools that the Simulink Design Verifier software 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 “Harness Model” on page 13-15.
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.
sldvruntime	Simulate a model, executing the specified test cases to record model coverage and output values.
sldvruncgvtest	Invoke the Code Generation Verification (CGV) API, and execute the specified test cases on the generated code for the model.  <b>Note</b> To execute a model in different modes of execution, use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see “Programmatic Code Generation Verification”.

Component verification functions do not support the following Simulink features:

- Variable-step solvers for `sldvrntest`
- Component interfaces that contain:
  - Complex signals
  - Variable-size signals
  - Array of buses
  - Multiword fixed-point data types

## Example: Verifying a Component for Code Generation

### In this section...

“About the Example Model” on page 10-6

“Preparing the Component for Verification” on page 10-9

“Recording Coverage for the Component” on page 10-11

“Using the Simulink® Design Verifier™ Software to Record Additional Coverage” on page 10-12

“Combining the Harness Models” on page 10-13

“Executing the Component in Simulation Mode” on page 10-14

“Executing the Component in Software-in-the-Loop (SIL) Mode” on page 10-15

### About the Example Model

This example uses the `slvndemo_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.

---

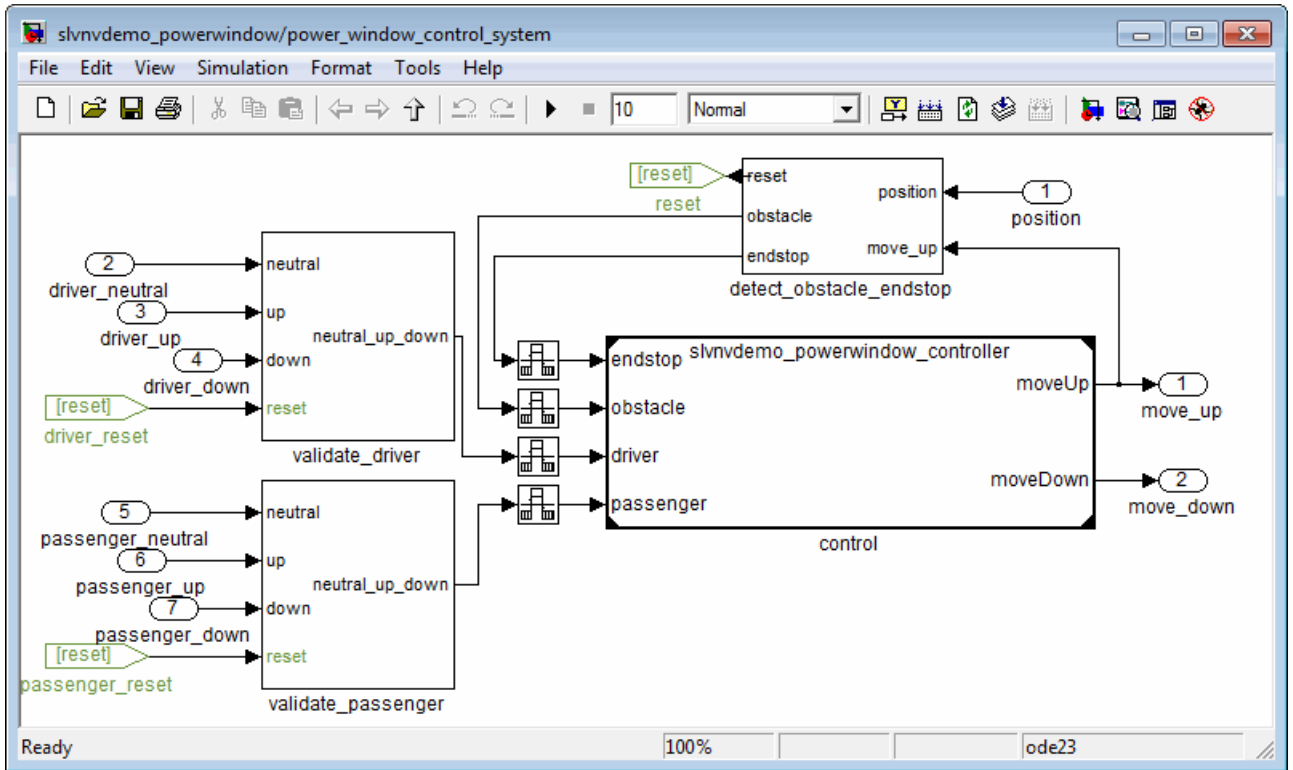
**Note** You must have the following product licenses to run this example:

- Stateflow
- Embedded Coder™
- Simulink Coder

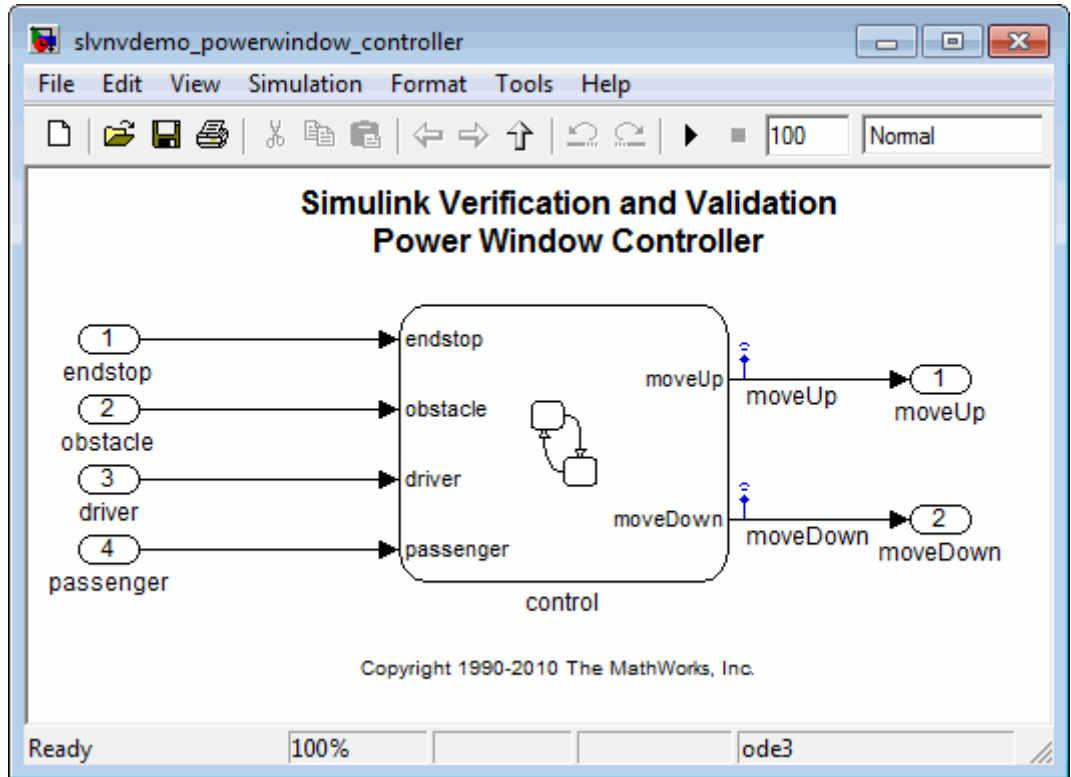
---

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 `slvndemo_powerwindow` model.

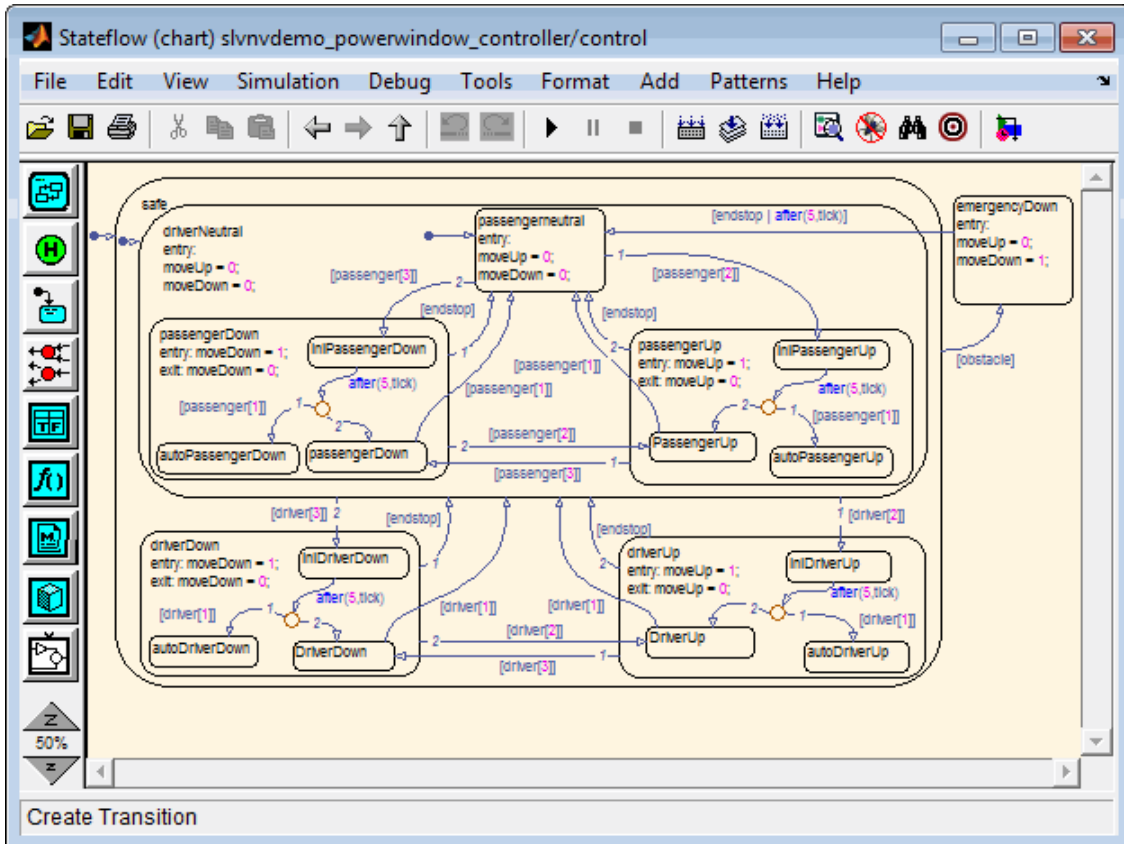




The Model block the top level of the slvndemo\_powerwindow model. references the slvndemo\_powerwindow\_controller model.



The referenced model contains a Stateflow chart `control`, which implements the logic for the power window controller.



## Preparing the Component for Verification

To verify the referenced model `slvndemo_powerwindow_controller`, create a harness model that contains the input signals that simulate the controller in the plant model:

- 1 Open the `slvndemo_powerwindow` demo model and the referenced model:

```
slvndemo_powerwindow;
slvndemo_powerwindow_controller;
```

- 2 Open the `power_window_control_system` subsystem in the demo model.

---

**Note** The Model block named control in the power\_window\_control\_system subsystem references the component that you verify during this example, slvndemo\_powerwindow\_controller.

---

- 3** Simulate the Model block that references the slvndemo\_powerwindow\_controller model and log the input signals to the Model block:

```
loggedSignalsPlant = ...
    sldvlogsignals(...
        'slvndemo_powerwindow/power_window_control_system/control');
```

sldvlogsignals stores the logged signals in loggedSignalsPlant.

- 4** Generate a harness model with the logged signals:

```
harnessModelFilePath = ...
    sldvmakeharness('slvndemo_powerwindow_controller', ...
        loggedSignalsPlant);
```

sldvmakeharness creates and opens a harness model named slvndemo\_powerwindow\_controller\_harness. The Signal Builder block contains one test case containing the logged signals.

---

**Note** For more information about harness models, see “Harness Model” on page 13-15.

---

- 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 slvndemo\_powerwindow\_controller model.

## Recording 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 `slvndemo_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('slvndemo_powerwindow_controller',...  
    loggedSignalsPlant,runOpts);
```

- 4 Display the HTML coverage report:

```
cvhtml('Coverage with Test Cases', covDataFromLoggedSignals);
```

The `slvndemo_powerwindow_controller` model achieved:

- Decision coverage: 40%
- Condition coverage: 35%
- MCDC coverage: 10%

---

**Note** For more information about decision coverage, condition coverage, and MCDC coverage, see “Types of Model Coverage” in the Simulink Verification and Validation documentation.

---

Because you did not achieve 100% coverage for the `slvndemo_powerwindow_controller` model, next, you will analyze the model to record additional coverage and create additional test cases.

## Using the Simulink Design Verifier Software to Record Additional Coverage

You can use the Simulink Design Verifier software to analyze the `slvndemo_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 ignore objectives that you satisfied when you logged the signals to the Model block:

```
opts.IgnoreCovSatisfied = 'on';
```

- 4 Specify the name of the file that contains the satisfied objectives data:

```
opts.CoverageDataFile = 'existingCovFromLoggedSignal.cvt';
```

- 5 Specify that the analysis not display unsatisfiable objectives in the Simulink Diagnostics Viewer:

```
opts.DisplayUnsatisfiableObjectives = 'off';
```

For this example, the focus is on satisfying as many objectives as possible.

- 6 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.

- 7** 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 “Preparing the Component for Verification” on page 10-9 uses a Model block that references the `slvndemo_powerwindow_controller` model. The harness model that the analysis creates must also use a Model block that references `slvndemo_powerwindow_controller`. You can append the test case data to the first harness model, creating a single test suite.

- 8** Analyze the model using the Simulink Design Verifier software:

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

The analysis creates and opens a harness model `slvndemo_powerwindow_controller_harness`. The Signal Builder block contains one long test case that satisfies 82 test objectives.

You can combine this test case with the test case that you created in “Preparing the Component for Verification” on page 10-9, to record additional coverage for the `slvndemo_powerwindow_controller` model.

- 9** 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.

## Combining the Harness Models

You created two harness models when you:

- Logged the signals to the control Model block that references the `slvndemo_powerwindow_controller` model.
- Analyzed the `slvndemo_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 `slvndemo_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 `slvndemo_powerwindow_controller_harness` model. First, configure the options object for `sldvruntime`:

```
runOpts = sldvruntimeopts;  
runOpts.coverageEnabled = true;
```

- 4 Simulate the model and record and display the coverage data:

```
[~, covDataFromMergedSignals] = ...  
    sldvruntime('slvndemo_powerwindow_controller', ...  
               loggedSignalsMergedHarness, runOpts);  
cvhtml('Coverage with Merged Test Cases', ...  
       covDataFromMergedSignals);
```

The `slvndemo_powerwindow_controller` model now achieves:

- Decision coverage: 100%
- Condition coverage: 80%
- MCDC coverage: 60%

## Executing the Component in Simulation Mode

To verify that the generated code for the model produces the same results as simulating the model, use the Code Generation Verification (CGV) API methods.



---

**Note** To execute a model in different modes of execution, use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see “Programmatic Code Generation Verification” in the Embedded Coder documentation.

---

When you perform this procedure, the simulation compiles and executes the model code using both test cases.

- 1 Create a default options object for `sldvruncgvtest`:

```
runcgvopts = sldvruntestopts('cgv');
```

- 2 Specify to execute the model in simulation mode:

```
runcgvopts.cgvConn = 'sim';
```

- 3 Execute the `slvvnv_powerwindow_controller` model using the two test cases and the `runcgvopts` object:

```
cgvSim = sldvruncgvtest('slvvnvdemo_powerwindow_controller', ...  
                      loggedSignalsMergedHarness, ...  
                      runcgvopts);
```

These steps save the results in the workspace variable `cgvSim`.

Next, you will execute the same model with the same test cases in Software-in-the-Loop (SIL) mode and compare the results from both simulations.

For more information about Normal simulation mode, see “Execute the Model” in the Embedded Coder documentation.

## Executing 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 `slvndemo_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('slvndemo_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 `cgv.CGV.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?” in the Embedded Coder documentation.

# Considering Specified Minimum and Maximum Values for Inputs During Analysis

---

- “Overview” on page 11-2
- “Example: Output Minimum and Maximum Values on Inport Blocks” on page 11-4
- “sldvData Fields for Minimum and Maximum Input Values” on page 11-6
- “Example: Minimum and Maximum Values in Simulink.Signal Objects” on page 11-8
- “Example: Minimum and Maximum Values on Stateflow Data Objects” on page 11-10
- “Example: Minimum and Maximum Values in Subsystems” on page 11-13
- “Example: Minimum and Maximum Values in Global Data Storage” on page 11-16

## Overview

### In this section...

“Simulink® Design Verifier™ Support for Specified Input Minimum and Maximum Values” on page 11-2

“Limitations of Simulink® Design Verifier™ Support for Specified Minimum and Maximum Values” on page 11-3

When creating a model, you can specify minimum and maximum values on input ports to mimic environmental constraints as part of your design. The Simulink Design Verifier analysis can automatically consider these values as constraints for:

- Design error detection
- Test case generation
- Property proving

Specifying minimum and maximum input values is similar to the way you use the Test Condition block or the `sldv.condition` function to constrain signals during test case generation or the Proof Assumption block or the `sldv.assume` function to constrain signals during 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 “Working with Signals” in the Simulink documentation.

---

## Simulink Design Verifier Support for Specified Input Minimum and Maximum Values

By default, the Simulink Design Verifier software considers any minimum and maximum input values specified for Inport blocks in your model. To enable this capability:

- 1** In the model window, select **Tools > Design Verifier > Options**.
- 2** On the **Design Verifier** pane, select the **Use specified input minimum and maximum values** parameter.
- 3** After the analysis completes, to view the design minimum and maximum constraints for your model, click **Generate detailed analysis reports**.

The constraints are listed in the **Analysis Information** chapter of the Simulink Design Verifier report.

## **Limitations of Simulink Design Verifier Support for Specified Minimum and Maximum Values**

Simulink Design Verifier support for specified minimum and maximum values has the following limitations:

- The analysis considers specified minimum and maximum values on root-level Inport blocks only. The analysis ignores any minimum and maximum values specified on other Simulink blocks.
- The analysis does not consider specified minimum and maximum values on bus objects. The analysis ignores any minimum and maximum values specified on Simulink bus objects.

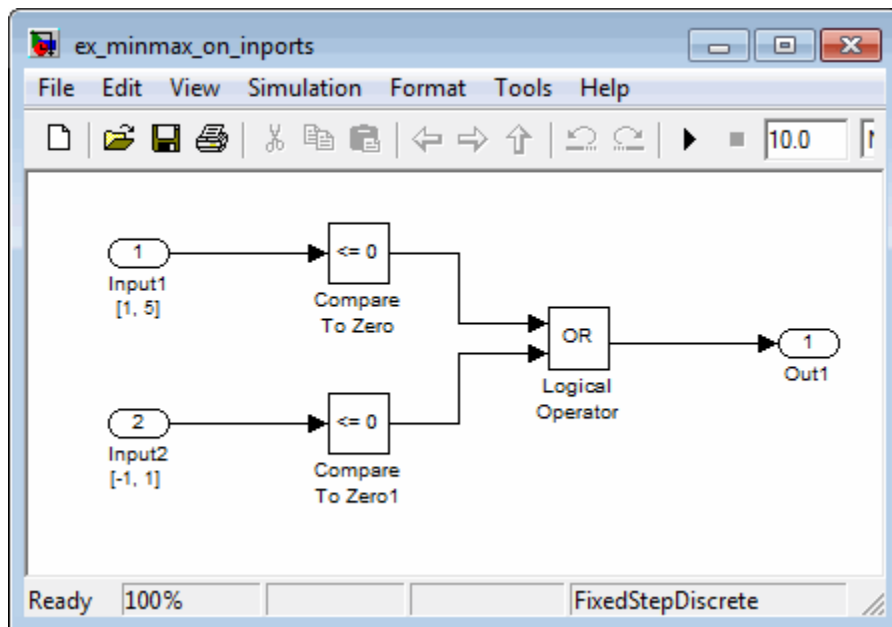
## Example: Output Minimum and Maximum Values on Inport Blocks

You can specify the output minimum and maximum values on Inport blocks in the Block Parameter dialog box, on the **Signal Attributes** tab. Set the following parameters:

- **Minimum**
- **Maximum**

The following example model restricts the signals from two Inport blocks:

- Input1 block: Minimum: 1, Maximum: 5
- Input2 block: Minimum: -1, Maximum: 1



When you analyze this model using the Simulink Design Verifier software, the analysis uses the minimum and maximum values as constraints. The analysis produces the following results:

- The output from Input1 is never less than 0, so the first input to the Logical Operator block is never `false`. The objective that the first input to the Logical Operator equals `false` is therefore unsatisfiable.
- The Logical Operator block cannot achieve 100% MCDC coverage because the condition where the first input is `false` never occurs.

The detailed analysis report indicates the values it used as constraints for Input1 and Input2.

### sldvData Fields for Minimum and Maximum Input Values

When you analyze a model, the Simulink Design Verifier software 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 “Example: Output Minimum and Maximum Values on 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 =
```



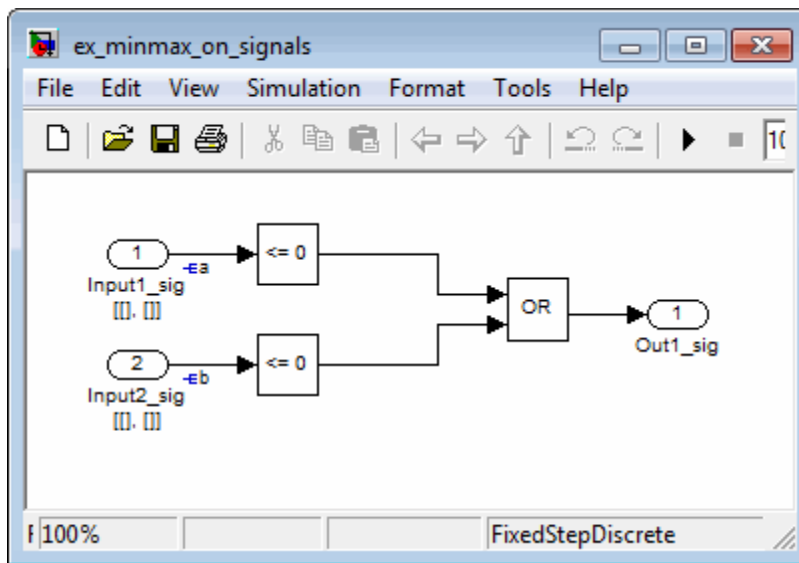
1

## Example: Minimum and Maximum Values in Simulink.Signal Objects

Using the Model Explorer, in the model workspace, you can specify minimum and maximum values on `Simulink.Signal` objects associated with input signals.

The following example model uses the `Simulink.Signal` objects associated with the input signals `a` and `b` to restrict the signal values:

- Signal `a`: Minimum: 1, Maximum: 5
- Signal `b`: Minimum: -1, Maximum: 1



When you analyze this model, the results are the same as if you specified the minimum and maximum values on the input ports.

---

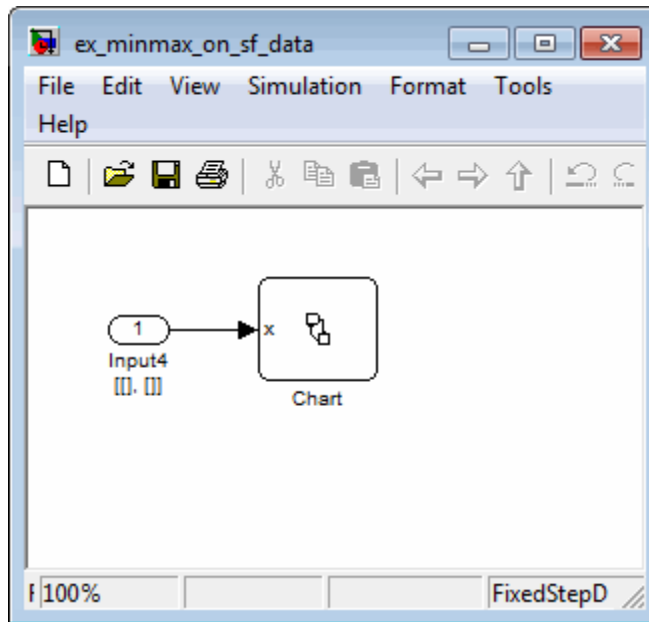
**Specifying Signal Ranges on Inport Blocks and Signals** If you specify ranges on the Inport blocks *and* on the signals, the analysis considers the tightest range. For example, if you specify a range of 4..12 on an input port and a range of 1..8 on the signal from the input port, the analysis considers the range 4..8.

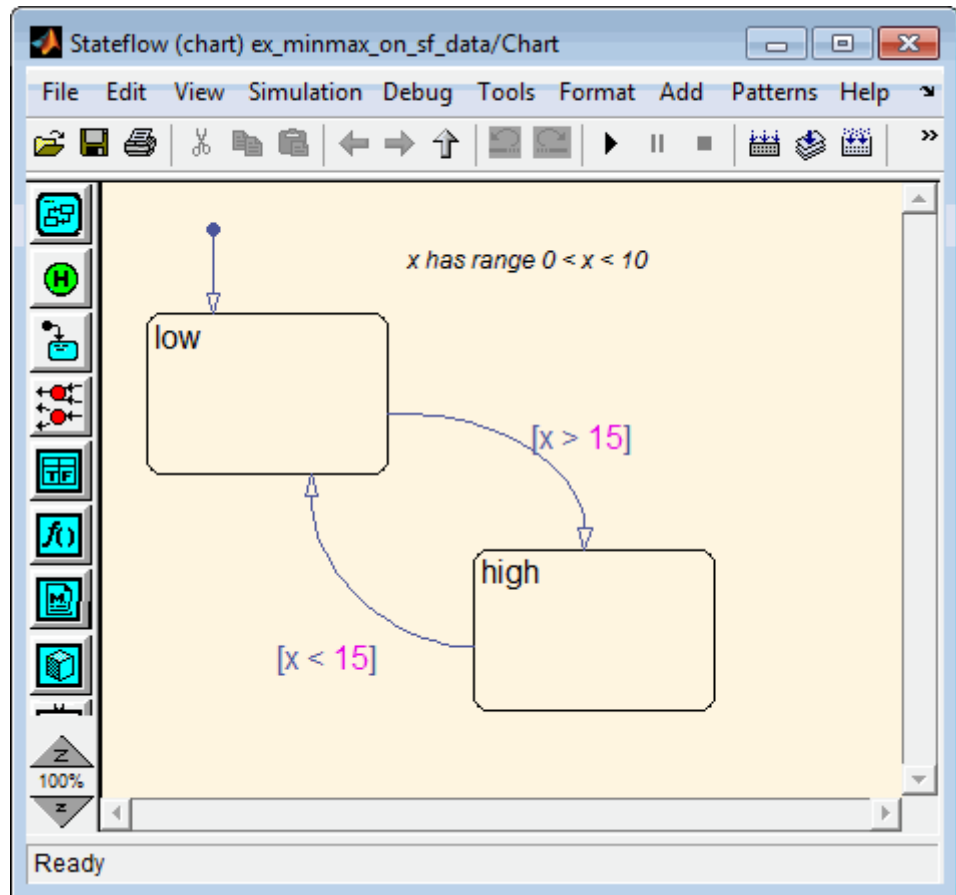
---

## Example: Minimum and Maximum Values on Stateflow Data Objects

Using the Model Explorer, you can specify ranges on data objects that are directly connected to the root-level input ports for a Stateflow chart.

In the following example model, the Stateflow chart named Chart has a data object  $x$  whose range you specified as  $0 < x < 10$ . In this chart,  $x$  must be greater than 15 to trigger the transition from low to high.





The value of  $x$  must fall between 0 and 10, so the transition condition  $[x > 15]$  is never true. The transition from low to high never occurs. Because the high state is never entered, the transition condition  $[x < 15]$  is never tested, and the transition from high to low never occurs; the chart is always in the low state.

When you analyze this model, the following objectives are proven unsatisfiable:

- The high state is never entered.

- The transition condition  $[x > 15]$  is always false, never true.
- The condition  $[x < 15]$  is never tested, so it is never true or false.

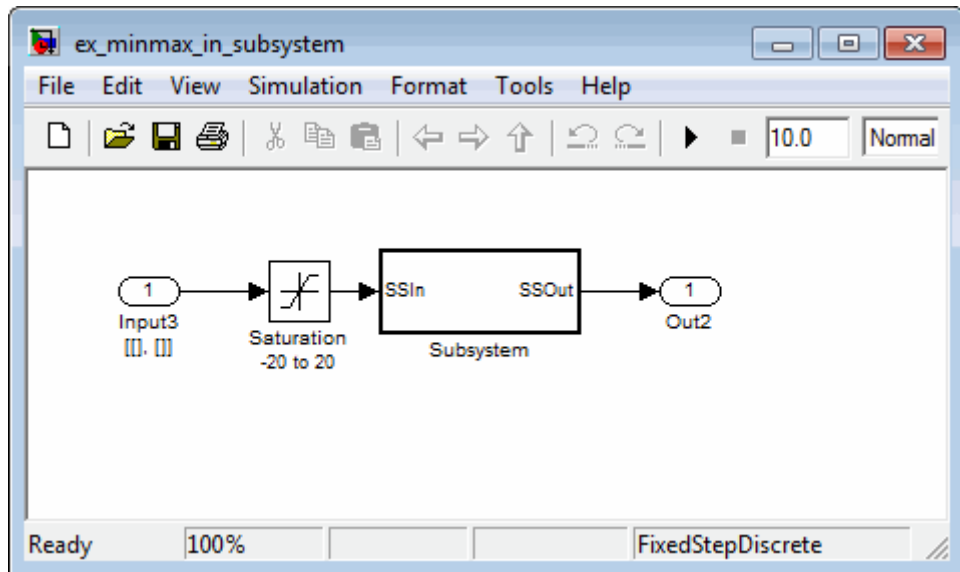
The analysis report indicates the values it used as constraints for  $x$ :  $[0, 10]$ .

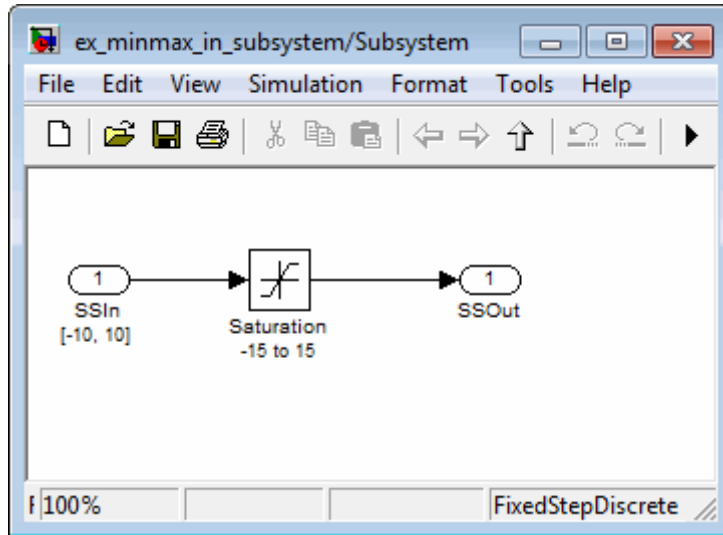
## Example: Minimum and Maximum Values in Subsystems

The Simulink Design Verifier software considers specified input minimum and maximum values as constraints only at the top level of a model. You can specify minimum and maximum values on Input ports on subsystems, but when you analyze the top-level model, the software ignores those values.

However, when you perform the subsystem analysis, the software considers specified minimum and maximum values on the input ports of the subsystem.

For example, consider the following model, and its subsystem.





In Subsystem, the specified minimum and maximum values for input port SSIn are  $-10$  and  $10$ , respectively. The lower and upper limits for the Saturation block are  $-15$  and  $15$ , respectively.

If you right-click Subsystem in the top-level model and select **Generate Tests for Subsystem**, the analysis considers the specified minimum and maximum values as constraints on the SSIn port.

## Constraints

### Design Min Max Constraints

Name	Design Min Max Constraint
<a href="#">SSIn</a>	$[-10, 10]$

The analysis identifies two unsatisfiable objectives:

- input > lower limit F: The input is always greater than the lower limit on the Saturation block ( $-15$ ).



- input  $\geq$  upper limit T: The input is never greater than or equal to the upper limit (15).

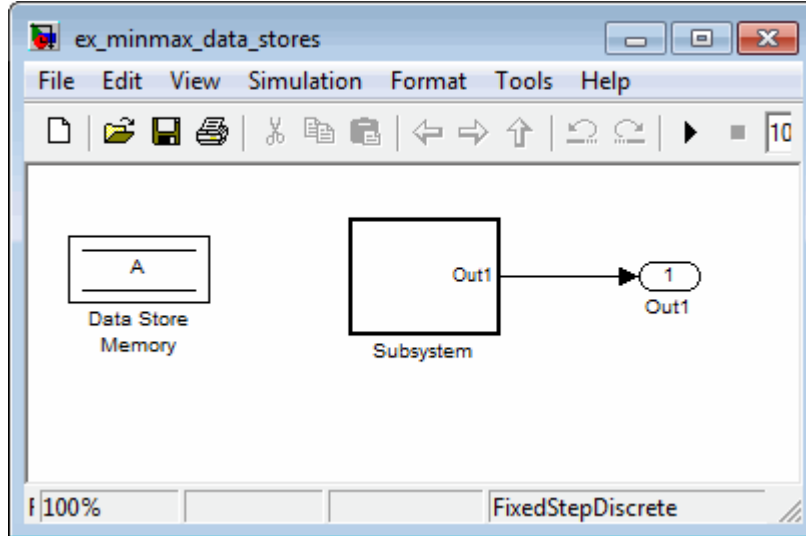
If you analyze the model that contains Subsystem, the analysis does not consider the values specified on the input port SSIn in the subsystem. The analysis considers only the root-level input ports at the respective level of the hierarchy for analysis.

## Example: Minimum and Maximum Values in 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. You can specify minimum and maximum values for any data store.

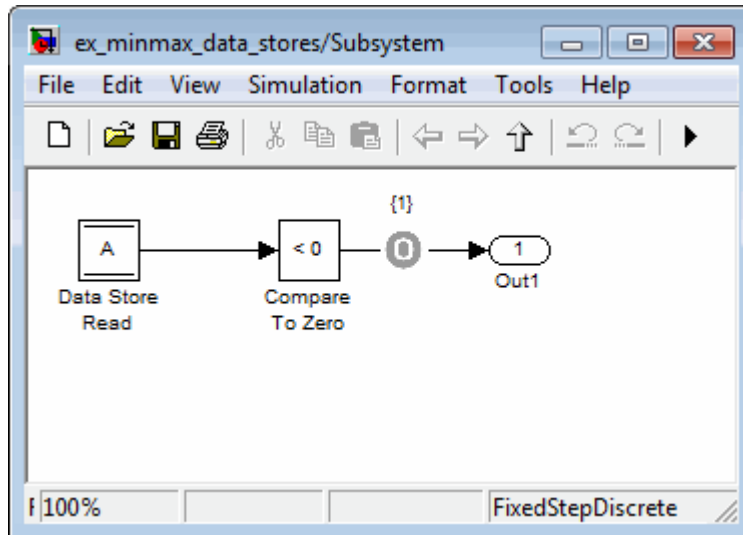
As described in “Extracting Subsystems for Analysis” on page 14-16, during subsystem analysis, the Simulink Design Verifier software creates a new input port to mimic the execution context for a global data store. If the data store has specified minimum and maximum values, those values are assigned as minimum and maximum values on the new input port. The input minimum and maximum values are used as subsystem-level analysis constraints.

In the following example model, data store A has a minimum value of 0 and a maximum value of 10.



The atomic subsystem reads from the data store and checks to see if the input is less than 0. The Compare To Zero block outputs 1 if the input is less than 0,

and outputs 0 if the output is greater than or equal to 0. The Test Objective block checks to see if the output is ever 1.



If you right-click Subsystem in the top-level model and select **Generate Tests for Subsystem**, the analysis considers the constraints for data store A to be  $[0, 10]$ .

The analysis does not satisfy the objective specified in the Test Objective block. The input is always greater than or equal to 0, so the output from the Compare To Zero block is always 0.



# Proving Properties of a Model

---

- “About Property Proving” on page 12-2
- “Workflow for Proving Model Properties” on page 12-4
- “Proving Properties in a Model” on page 12-5
- “Using a Verification Model to Prove System-Level Properties” on page 12-28
- “Proving Properties in a Subsystem” on page 12-32
- “Property-Proving Examples” on page 12-33

### About Property Proving

A *property* is a requirement that you model in Simulink, Stateflow, and using MATLAB Function blocks. A property can be a simple requirement, such as a signal in your model that must attain a particular value or range of values during simulation.

A property can also be a requirement on the model that involves a number of input and output signals modeled as a logical expression that needs to be proved.

The Simulink Design Verifier software performs a formal analysis of your model to prove or disprove the specified properties. After completing the analysis, the software offers several ways for you to review the results:

- Highlighted on the model
- A harness model with test cases
- A detailed HTML report

### Proof Blocks

The Simulink Design Verifier software provides two blocks so you can specify property proofs in your Simulink models:

- Proof Objective — Define the values of a signal to prove
- Proof Assumption — Constrain the values of a signal during a proof

---

**Note** Blocks from the Model Verification library in the Simulink software behave like Proof Objective blocks during Simulink Design Verifier proofs. You can use Assertion blocks and other Model Verification blocks to specify properties of your model. For more information about these blocks, see “Model Verification” in the *Simulink Reference*.

---

## 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 have no functionality.

---

# 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 nullify 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 “Proving Properties in a Model” on page 12-5.



## Proving Properties in a Model

In this section...
“About This Example” on page 12-5
“Constructing the Example Model” on page 12-6
“Checking Compatibility of the Example Model” on page 12-7
“Instrumenting the Example Model” on page 12-9
“Configuring Property-Proving Options” on page 12-10
“Analyzing the Example Model” on page 12-11
“Reviewing the Analysis Results” on page 12-11
“Customizing the Example Proof” on page 12-21
“Reanalyzing the Example Model” on page 12-22
“Reviewing the Results of the Second Analysis” on page 12-23
“Analyzing Contradictory Models” on page 12-26
“Proving Properties in a Large Model” on page 12-27

### 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 the Simulink Design Verifier software.

In this example, you perform the following tasks.

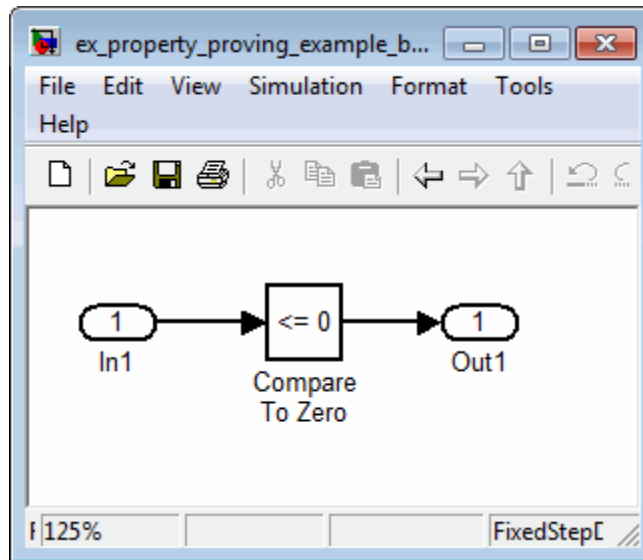
Task	Description	See...
1	Construct the example model.	“Constructing the Example Model” on page 12-6
2	Verify that your model is compatible with the Simulink Design Verifier software.	“Checking Compatibility of the Example Model” on page 12-7

<b>Task</b>	<b>Description</b>	<b>See...</b>
3	Add a Proof Objective block to your model to prepare for its proof.	“Instrumenting the Example Model” on page 12-9
4	Configure the Simulink Design Verifier software to prove properties.	“Configuring Property-Proving Options” on page 12-10
5	Prove a property of your model.	“Analyzing the Example Model” on page 12-11
6	Review the analysis results.	“Reviewing the Analysis Results” on page 12-11
7	Add proof assumptions to specify analysis constraints.	“Customizing the Example Proof” on page 12-21
8	Prove a property of the customized model and interpret the results.	“Reanalyzing the Example Model” on page 12-22

## **Constructing the 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 the Simulink Design Verifier software controls
  - From the Logic and Bit Operations library, a Compare To Zero block to provide simple logic
  - From the Sinks library, an Outport block to receive the output signal
- 3** Connect these blocks such so your model appears similar to the following model:



**4** In the model window, select **Simulation > Configuration Parameters**.

**5** On the left side of the Configuration Parameters dialog box, in the **Select** tree, click the **Solver** category. On the right side, under **Solver options**:

- Set the **Type** option to **Fixed-step**.
- Set the **Solver** option to **Discrete (no continuous states)**.

The Simulink Design Verifier can analyze only models that use a fixed-step solver.

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

**7** Save your model with the name `ex_property_proving_example_basic`.

## Checking Compatibility of the Example Model

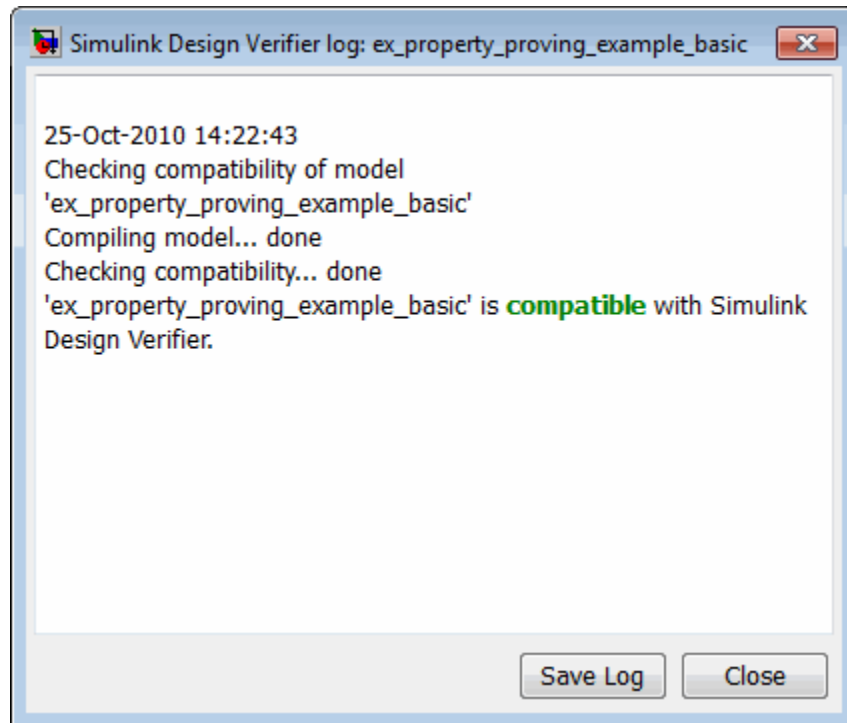
Every time Simulink Design Verifier software analyzes a model, before the analysis begins, the software performs a compatibility check. If your model is not compatible, the software cannot analyze it.

You can also make sure your model is compatible with the Simulink Design Verifier software before you start the analysis:

- 1 Open the `ex_property_proving_example_basic` model.
- 2 In the model window, select **Tools > Design Verifier > Check Model 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.



### 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 the Simulink Design Verifier

software 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 “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

## Instrumenting the Example Model

Prepare your example model so that you can prove its properties with the Simulink Design Verifier software. 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.

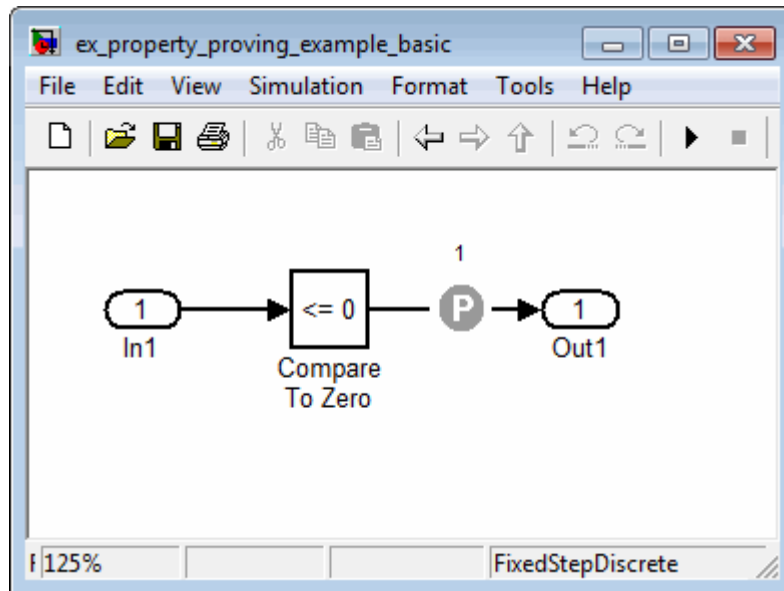
- 4** In your model, double-click the Proof Objective block.

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.

### Configuring Property-Proving Options

Configure the Simulink Design Verifier software to prove properties of the `ex_property_proving_example_basic` model that you instrumented:

- 1 Open the `ex_property_proving_example_basic` model.
- 2 In your Simulink model window, select **Tools > Design Verifier > Options**.
- 3 On the left side of the Configuration Parameters dialog box, in the **Select** tree, select the **Design Verifier** category. Under **Analysis options** on the right side, set the **Mode** parameter to Property proving.
- 4 Click **OK** to apply your changes and close the Configuration Parameters dialog box.

---

**Note** On the **Property Proving** pane, you can optionally specify values for other parameters that control how the Simulink Design Verifier software proves properties of your model. For more information, see “Design Verifier Pane: Property Proving” on page 15-40.

---

5 Save the `ex_property_proving_example_basic` model.

## Analyzing the Example Model

To analyze the `ex_property_proving_example_basic` model, in the model window, select **Tools > Design Verifier > Prove Properties**. The Simulink Design Verifier software begins a property-proving analysis.

During the analysis, the log window shows the progress of the analysis. It displays information such as the number of objectives processed and which objectives were satisfied or falsified.

To terminate the analysis at any time, in the log window, click **Stop**.

## Reviewing the 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:

- “Reviewing the Results on the Model” on page 12-12

- “Reviewing the Detailed Analysis Report” on page 12-14
- “Reviewing the Harness Model” on page 12-16
- “Simulation the Model with the Counterexample” on page 12-18
- “Reviewing Analysis Results in the Model Explorer” on page 12-20

### Reviewing the Results on the Model

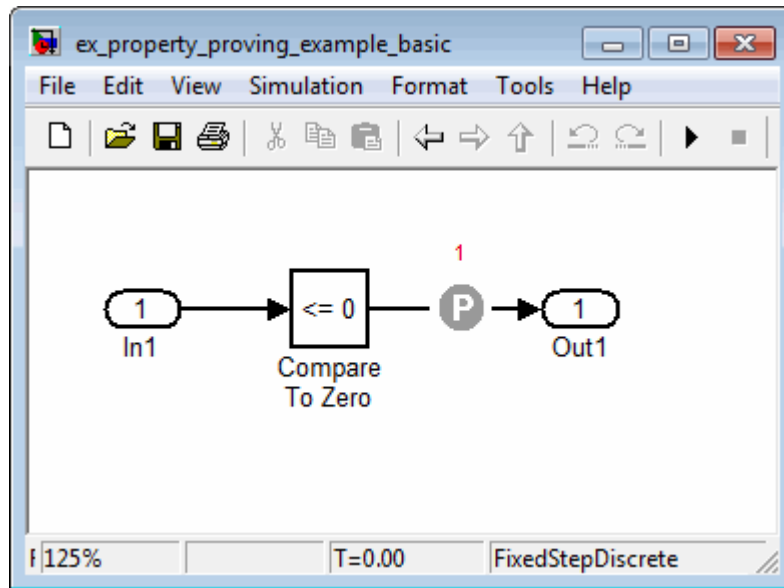
You can review the analysis results at a glance by viewing the blocks that are highlighted in the model window. The highlighting can have four colors:

- Green — The analysis proved all the proof objectives valid.
- Red — The analysis disproved a proof objective and generated a counterexample that falsified that objective.
- Orange — The analysis disproved a proof objective, but it could not generate a counterexample or the proof objective remained undecided. This result occurs due to:
  - A proof objective on a signal whose value the software cannot control, for example, a Constant block
  - A proof objective that depends on nonlinear computation
  - A proof objective that creates an arithmetic error, such as division by zero
  - Automatic stubbing being enabled, and the analysis encountering an unsupported block whose operation it does not understand but that the analysis requires to generate the counterexample
  - The analysis timing out
  - Limitations of the analysis engine
- Gray — The model object was not part of the analysis.

Highlight the analysis results on the example model:

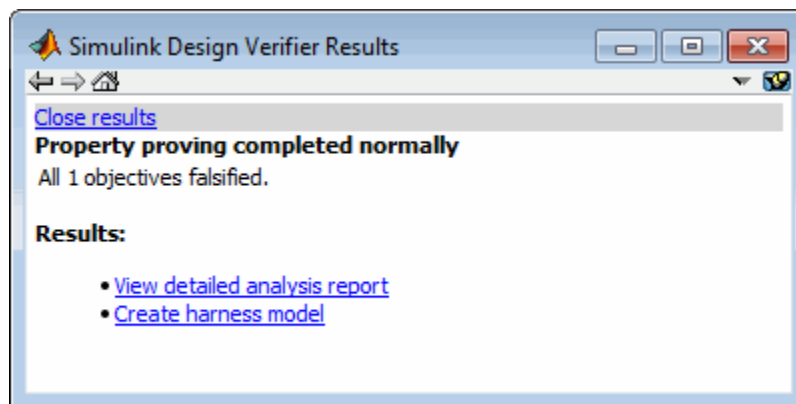
- 1 In the log window for the `ex_property_proving_example_basic` analysis, click **Highlight analysis results on model**.






The Proof Objective block is highlighted in red, which indicates that a proof objective was falsified with a counterexample.

The Simulink Design Verifier Results window appears. As you click objects in the model, this window changes to display detailed analysis results for that object.



---

**Tip** By default, the Simulink Design Verifier Results window is always the topmost visible window. To allow the window to move behind other window, click  and clear **Always on top**.

---

- 2 Click the highlighted Proof Objective block.

The Simulink Design Verifier Results window indicates that the proof objective that the output signal from the Compare to Zero was not 1 was disproved with a counterexample.

### Reviewing the Detailed Analysis Report

To create a detailed HTML analysis report:

- 1 In the Simulink Design Verifier log window, click **Generate detailed analysis report**.

The HTML report opens in a browser window.

- 2 The report includes the following **Table of Contents**. Click a hyperlink to navigate to particular section in the report.

Table of Contents
<a href="#">1. Summary</a>
<a href="#">2. Analysis Information</a>
<a href="#">3. Proof Objectives Status</a>
<a href="#">4. Properties</a>

- 3 In the **Table of Contents**, click Summary.

## Chapter 1. Summary

### Analysis Information

Model: ex\_property\_proving\_example  
 Mode: PropertyProving  
 Status: Completed normally  
 Analysis Time: 0s

### Objectives Status

**Number of Objectives:** 1  
 Objectives Falsified with Counterexamples: 1

The Summary provides an overview of the analysis results, and it indicates that the Simulink Design Verifier software identified a counterexample that falsifies an objective in your model.

- 4 Scroll back to the top of the browser window. In the **Table of Contents**, click **Proof Objectives Status**.

### Objectives Falsified with Counterexamples

#:	Type	Model Item	Description	Counterexample
1	Custom Proof Objective	<a href="#">Proof Objective</a>	Objective: 1	<a href="#">1</a>

The Objectives Falsified with Counterexamples table lists the proof objectives that the Simulink Design Verifier software disproved using a counterexample that it generated. You can locate the objective in your model window by clicking **Proof Objective**; the software highlights the corresponding **Proof Objective** block in your model window.

- 5 In the Objectives Falsified with Counterexamples table, under the **Counterexample** column, click 1.

### Proof Objective

**Summary**

Model Item: [Proof Objective](#)  
Property: Objective: 1  
Status: Falsified

**Counter Example**

<b>Time 0</b>	
<b>Step 1</b>	
In1	1

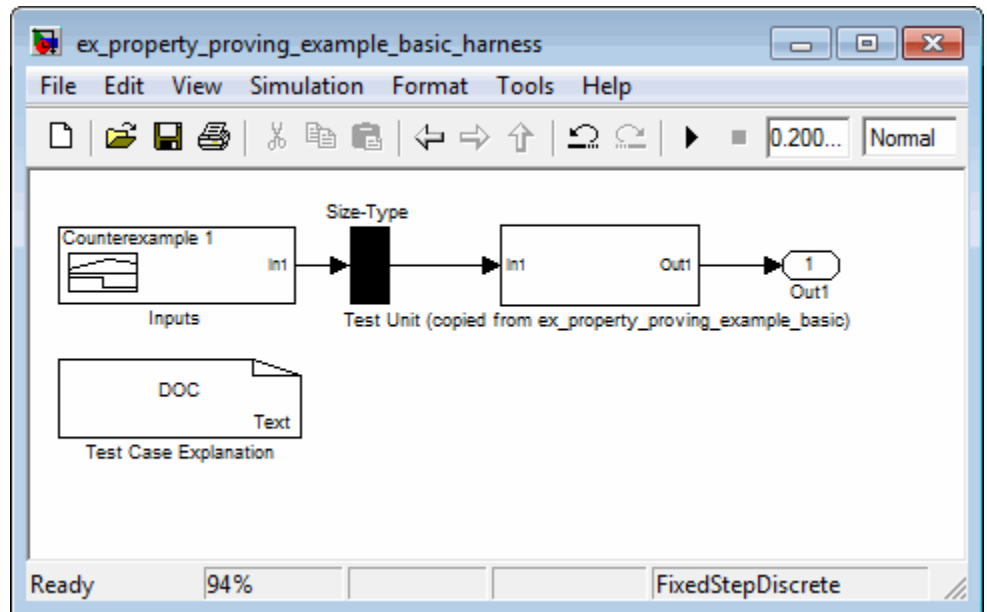
This section displays information about proof objective 1 and provides details about the counterexample that the Simulink Design Verifier software generated to disprove that objective. In this counterexample, a signal value of 99 falsifies the objective that you specified using the Proof Objective block. 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).

### Reviewing the 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**.

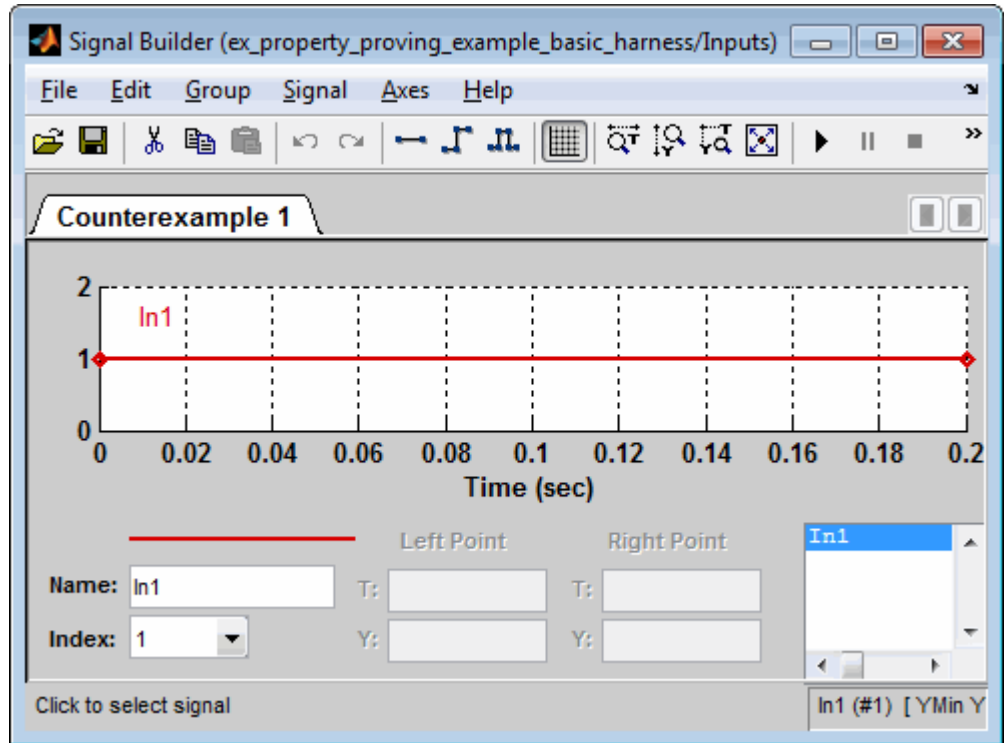
The software creates a harness model named `ex_property_proving_example_basic_harness.mdl`.



The harness model contains the following items:

- Signal Builder block named `Inputs` — A group of signals that falsify proof objectives.
- Subsystem block named `Test Unit` — A copy of your model.
- DocBlock named `Test Case Explanation` — A textual description of the counterexamples that the analysis generates.
- A `Size-Type` block — A subsystem that transmits signals from the `Inputs` block to the `Test Unit` block. This block verifies that the size and data type of the signals are consistent with the `Test Unit` block.

**2** Double-click the `Inputs` block.



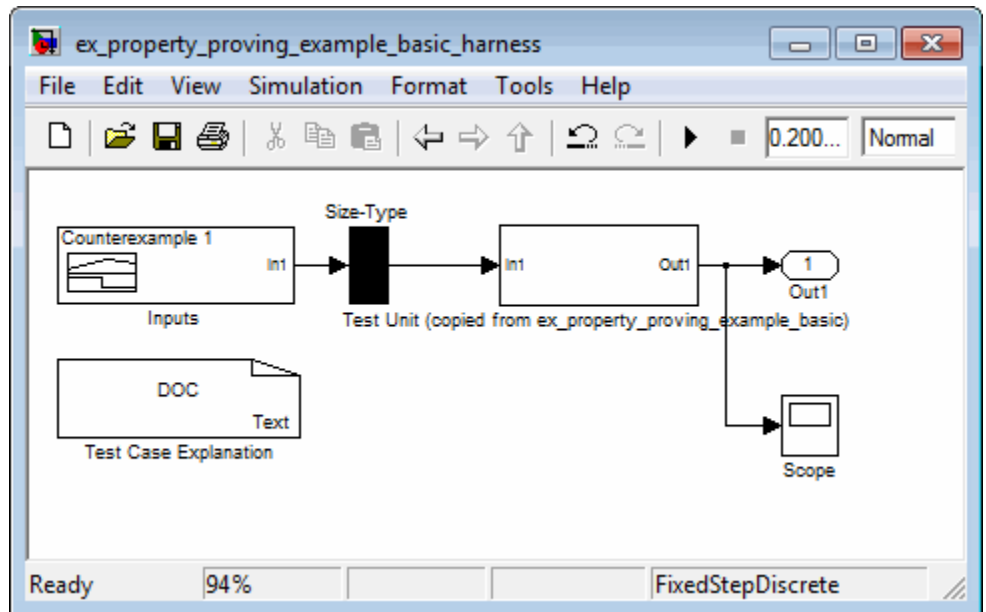
The input signal 1 causes the output of the Compare to Zero block to be 0. This counterexample violates the proof objective that specifies that the output of the Compare to Zero block be 1.

### Simulation the Model with the Counterexample

Simulate the harness model to observe the counterexample that falsifies the proof objective in your model:

- 1 In the `ex_property_proving_example_basic` model window, select **View > Library Browser**
- 2 From the Sinks library, copy a Scope block into your harness model window. The Scope block allows you to see the value of the signal output by the Compare To Zero block in your model.

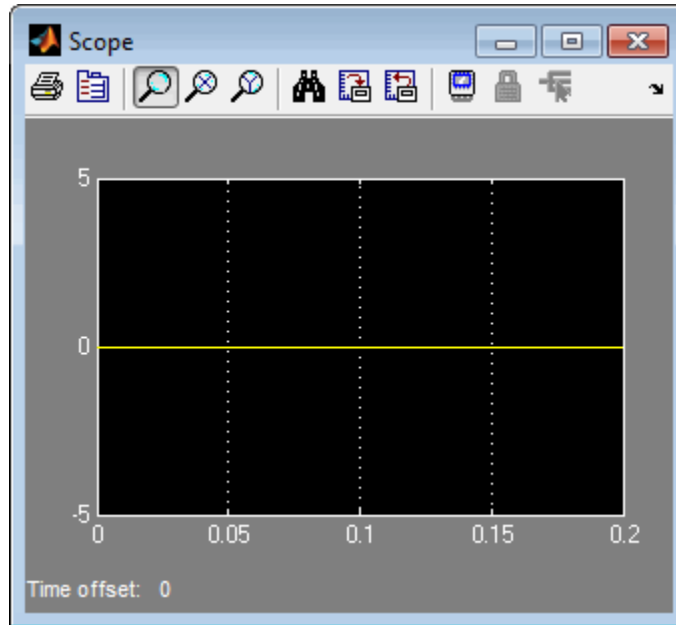
- 3 In your harness model window, connect the output signal of the Test Unit subsystem to the Scope block.



- 4 In your harness model window, select **Simulation > Start** to begin the simulation.

The Simulink software simulates the harness model.

- 5 In your harness model window, double-click the Scope block to open its display window.



The Scope block displays the value of the signal output by the Compare To Zero block in your model. In this example, the Compare To Zero block returns 0 (false) throughout the simulation, which falsifies the proof objective that the output of the Compare to Zero block be 1 (true). The counterexample that the Signal Builder block supplies falsifies the proof objective.

### Reviewing Analysis Results in the Model Explorer

If you close the analysis results so you review any falsified objectives, you may need to review the analysis results again. As long as your model remains open, you can view the results of your most recent Simulink Design Verifier analysis results in the Model Explorer. After you close your model, you can no longer view any analysis results.

In the model window, select **Tools > Design Verifier > Latest Results**. The Model Explorer opens, and the results of the latest Simulink Design Verifier analysis appear in the right-hand pane.



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

<b>Task</b>	<b>For more information</b>
Highlight the analysis results on the model.	“Highlighted Results on the Model” on page 13-2
Generate a detailed analysis report.	“Simulink® Design Verifier™ Reports” on page 13-27
Create the harness model, or if the harness model already exists, open it.  If no counterexamples were created during the analysis, this option is not available.	“Harness Model” on page 13-15
View the data file.	“Simulink® Design Verifier™ Data Files” on page 13-7
View the log file.	“Simulink® Design Verifier™ Log Files” on page 13-52

## Customizing the Example Proof

Modify the simple Simulink model whose proof objective the Simulink Design Verifier software 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.

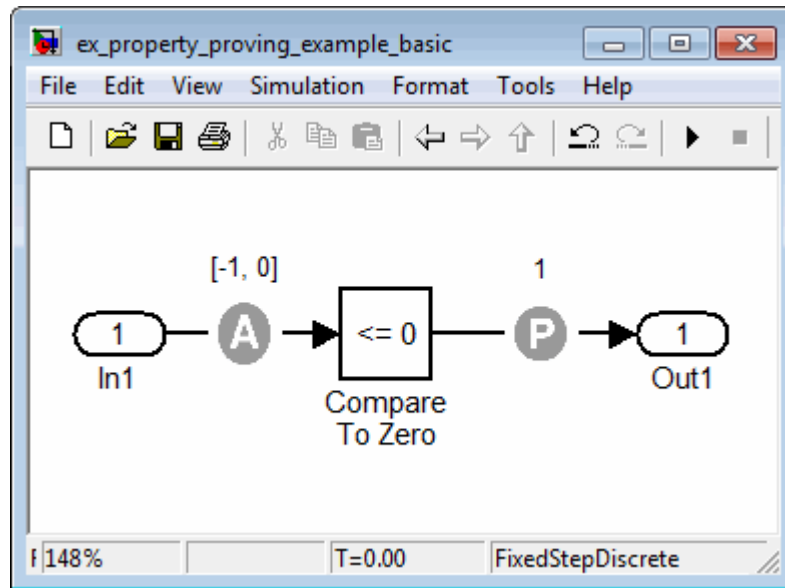
- 3** Copy the Proof Assumption block to your model.

- 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, the Simulink Design Verifier software 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.

### Reanalyzing the Example Model

Analyze the model that you modified to see how the Proof Assumption block affects the property-proving analysis.

In the `ex_property_proving_example_basic` model window, select **Tools > Design Verifier > 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.

## Reviewing the Results of the Second Analysis

Review the results of the second analysis:

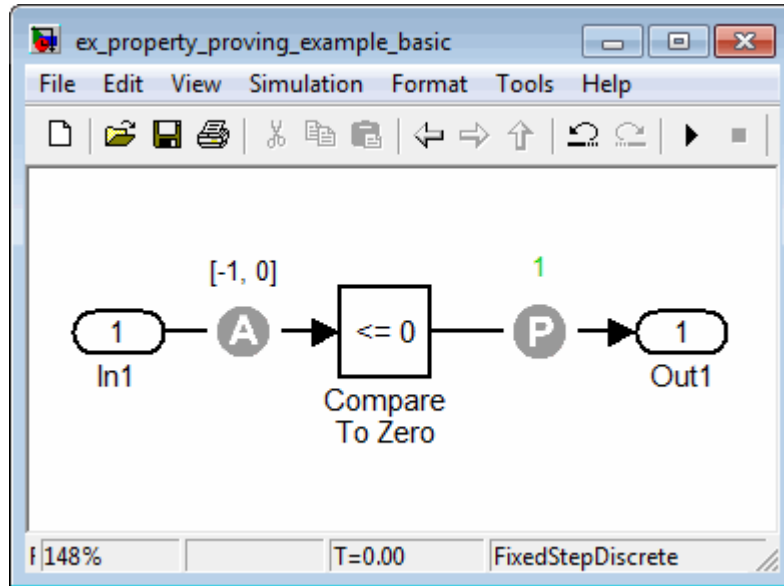
- “Reviewing the Results on the Model” on page 12-23
- “Reviewing the Analysis Report” on page 12-25

## Reviewing the Results on the Model

Highlight the model to see the analysis results:

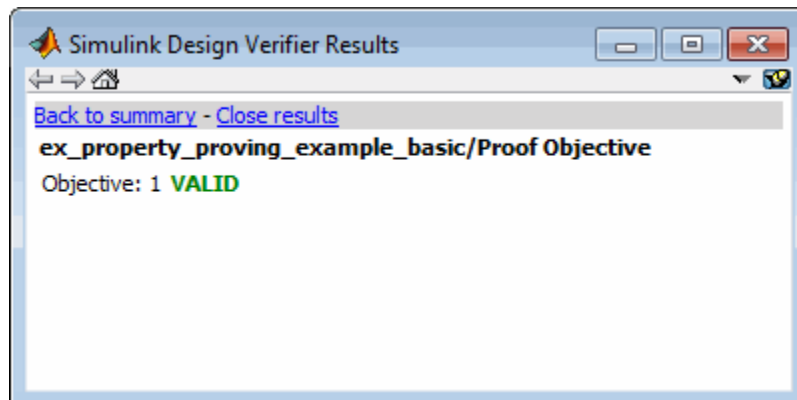
- 1 Click **Highlight analysis results on model**.

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.



## Reviewing the Analysis Report

Review the analysis results in the detailed report:

- 1 Click **Generate detailed analysis report**.
- 2 In the **Table of Contents**, click **Summary**.

### Chapter 1. Summary

**Analysis Information**

Model: ex\_property\_proving\_example\_basic  
 Mode: PropertyProving  
 Status: Completed normally  
 Analysis Time: 0s

**Objectives Status**

<b>Number of Objectives:</b>	<b>1</b>
Objectives Proven Valid:	1

The Summary chapter indicates that the Simulink Design Verifier software 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
<a href="#">Assumption</a>	$[-1, 0]$

- 4 Scroll back to the top of the browser window. In the **Table of Contents**, click **Proof Objectives Status**.

<b>Objectives Proven Valid</b>				
#	Type	Model Item	Description	Counterexample
1	Proof objective	<a href="#">Proof Objective</a>	Objective: 1	n/a

The Objectives Proven Valid table lists the proof objectives that the Simulink Design Verifier software 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**.

<b>Proof Objective</b>	
<b>Summary</b>	
Model Item:	<a href="#">Proof Objective</a>
Property:	Objective: 1
Status:	Proven valid

The Proof Objective summary indicates that the Simulink Design Verifier software 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.

## Analyzing 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 states 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.

## **Proving Properties in a Large Model**

A thorough proof of your model requires that the Simulink Design Verifier software 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.

“Techniques for Proving Properties of Large Models” on page 14-27 gives detailed information about strategies you can use to improve the performance of a property-proving analysis of a large model.

## Using a Verification Model to Prove System-Level Properties

In this section...
“When to Use a Verification Model for Property Proving” on page 12-28
“About this Example” on page 12-28
“Understanding the Verification Model” on page 12-29
“Proving the Properties of the Design Model” on page 12-29
“Fixing the Verification Model” on page 12-31

### 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_sbr_verification_fixed` model, the constraint is enabled and all the properties are proven valid.



## Understanding 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.

## Proving the Properties of the Design Model

Analyze the `sldvdemo_sbr_verification` model to prove the properties:

- 1 In the `sldvdemo_sbr_verification` model window, to start the analysis, double-click the **Run** button to start the analysis.

When the analysis completes, the Simulink Design Verifier log window indicates that one objective was falsified.

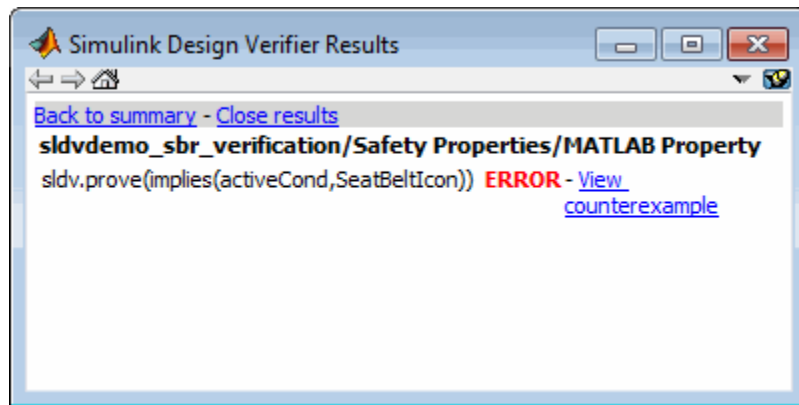
- 2 To see which objective was falsified, click **Highlight analysis results on model**.

The Safety Properties subsystem is highlighted in red.

- 3 Open the Safety Properties subsystem and click the MATLAB Property block.

The Simulink Design Verifier Results window indicates that the statement `sldv.prove(implies(activeCond,SeatBeltIcon))`

was false during at least one time step.



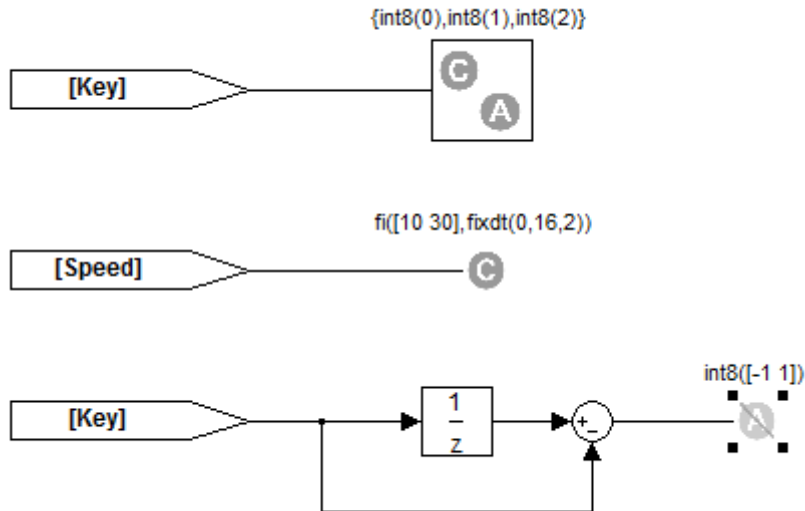
- 4 Click **View counterexample** to see the signal values that violated this property.

The Signal Builder block opens with the counterexample. The KEY input was initially 2, which is invalid.

To validate the property specified in the Safety Properties subsystem, you have to make sure that the initial value of KEY is 0.

## Fixing 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.

### Proving 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 the Simulink Design Verifier software using the **Prove Properties of Subsystem** option.
- 4 Review the results.

The tutorial in “Analyzing a Subsystem” on page 1-29 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.

## Property-Proving Examples

The Simulink Design Verifier block library includes a sublibrary Example Properties. The Example Properties sublibrary includes:

- “Basic Properties” on page 12-33 — Four examples that demonstrate how to prove basic properties.
- “Temporal Properties” on page 12-35 — 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

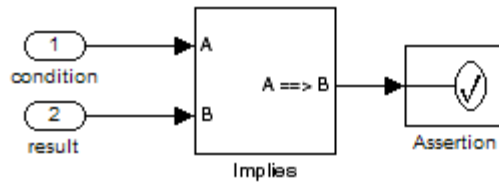
To view the Basic Properties examples:

- 1** Open the Simulink Design Verifier block library. Type:  
`sldvlib`
- 2** Double-click the Example Properties 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.

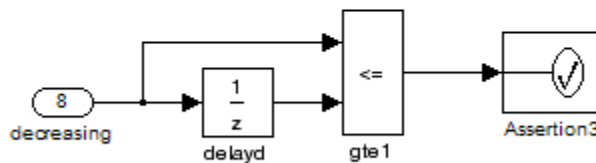
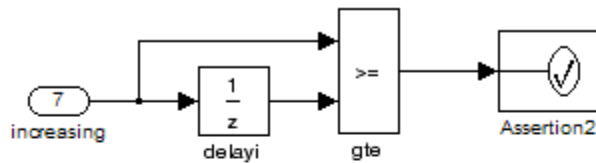


Implies operation describes conditions that should trigger a result.

### Increasing or Decreasing Signals

The two examples in this section specify that a signal is either:

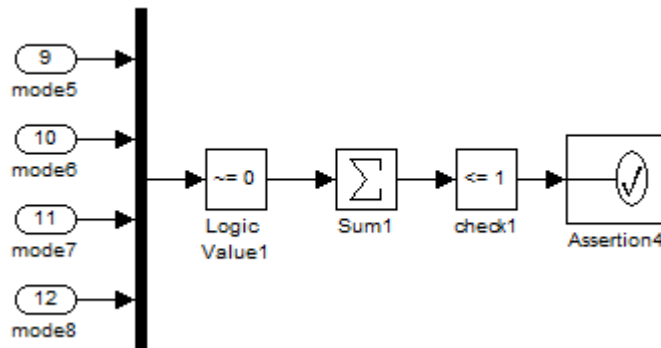
- Always increasing or staying constant
- Always decreasing or staying constant



Increasing and decreasing operations describe signals that should increase or decrease.

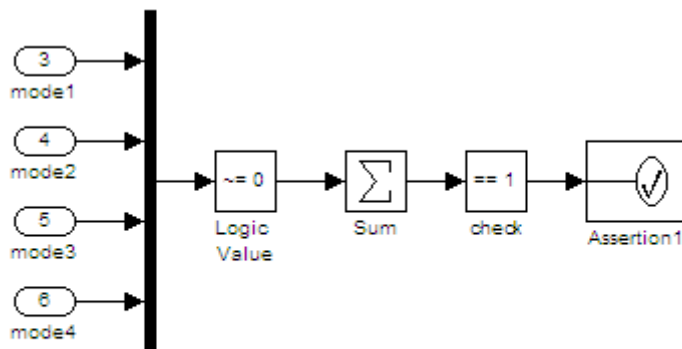
## Exclusivity Operation

This example describes four conditions that should not be true at the same time.



## Conditions with One True Element

This example specifies that only one of the four input signals can be true.



Mutual exclusivity operation describes conditions that should have exactly one true element.

## Temporal Properties

To view the Temporal Properties examples:

1 Open the Simulink Design Verifier block library. Type:

sldvlib

2 Double-click the Temporal Properties sublibrary.

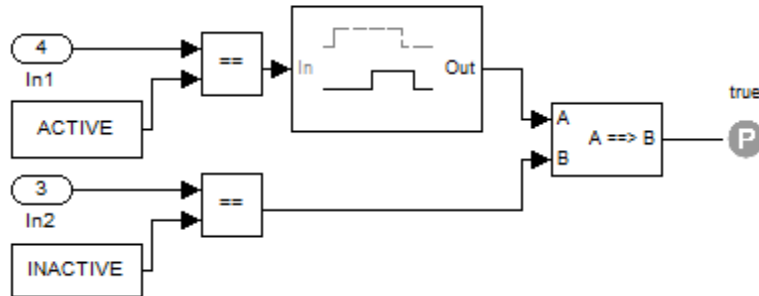
3 Double-click the **Temporal Properties** block that contains the examples.

The sections that follow describe each example in the Temporal Properties sublibrary in detail.

## Synchronizing 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.**

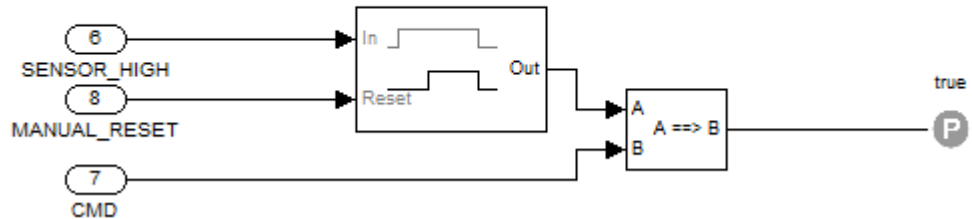


## Making a Signal Inactive After a Delay

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.



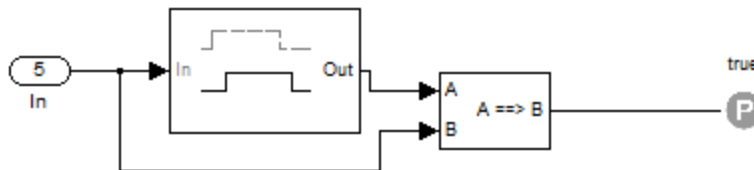
After Sensor is detected at HIGH for 5 consecutive steps, Cmd becomes and stays true for the remaining duration of the Sensor value HIGH unless manual reset is detected.



### Extending 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.

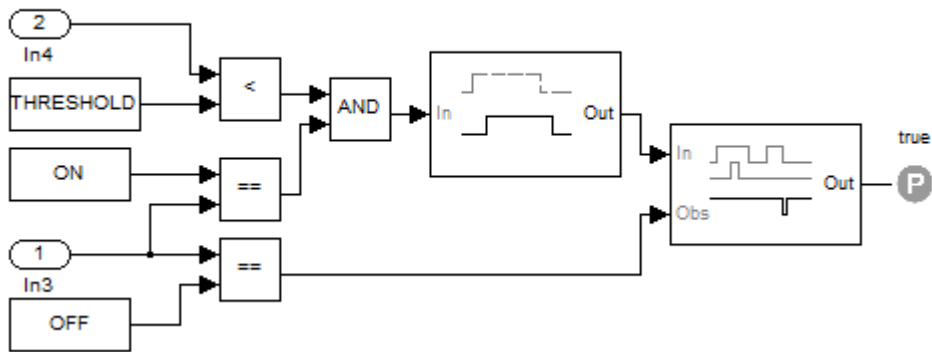
Whenever In becomes true, it shall stay true for the following 5 steps as well.



### Testing 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.



# Reviewing the Results

---

- “Highlighted Results on the Model” on page 13-2
- “Simulink® Design Verifier™ Data Files” on page 13-7
- “Harness Model” on page 13-15
- “SystemTest TEST-Files” on page 13-24
- “Simulink® Design Verifier™ Reports” on page 13-27
- “Simulink® Design Verifier™ Log Files” on page 13-52
- “Reviewing Analysis Results in the Model Explorer” on page 13-53

## Highlighted Results on the Model

In this section...
“When to Highlight Results on the Model” on page 13-2
“Enabling Highlighted Results on a Model” on page 13-2
“Simulink Design Verifier Results Window” on page 13-3
“Green Highlighting on Model” on page 13-3
“Red Highlighting on Model” on page 13-3
“Orange Highlighting on Model” on page 13-4
“Gray Highlighting on Model” on page 13-6

### When to Highlight Results on the Model

When you analyze a model using the Simulink Design Verifier software, you have the option to highlight the analyzed model objects in one of four colors:

- Green
- Red
- Orange
- Gray

Model highlighting allows you to review the analysis results at a glance by viewing the objects that are highlighted in the model window.

### Enabling Highlighted Results on a Model

When you run a design error detection analysis, highlighting results on the model is always enabled.

When you run a test-case generation or property-proving analysis, to enable the highlighting, do one of the following:

- Before the analysis, in the Configuration Parameters dialog box, **Design Verifier > Results** pane, select the **Display the results of the analysis on the model** parameter.
- After the analysis, in the Simulink Design Verifier log window, select **Highlight analysis results on model**.

## Simulink Design Verifier Results Window

When a model is highlighted, you can click an object for which the analysis recorded results. When you click the object, the Simulink Design Verifier Results window displays the detailed analysis results for that object.

## Green Highlighting on Model

Objects that are highlighted in green have the following meaning for each type of analysis.

Analysis mode	Green highlighting means...
Design error detection	One of the following: <ul style="list-style-type: none"> <li>• The analysis did not find any overflow or division-by-zero errors.</li> <li>• The analysis did not find any dead logic</li> <li>• The analysis did not find intermediate or output signals outside the range of user-specified minimum and maximum constraints.</li> </ul>
Test-case generation	The analysis found test cases that satisfy the test objectives.
Property proving	The analysis proved all the proof objectives valid.

## Red Highlighting on Model

Objects that are highlighted in red have the following meaning, depending on the analysis type.

<b>Analysis mode</b>	<b>Red highlighting means...</b>
Design error detection	<p>One of the following:</p> <ul style="list-style-type: none"> <li>• The analysis found at least one test case that causes overflow or division-by-zero errors.</li> <li>• The analysis found dead logic.</li> <li>• The analysis found intermediate or output signals outside the range of user-specified minimum and maximum constraints.</li> </ul>
Test-case generation	The analysis could not satisfy certain test objectives.
Property proving	The analysis disproved a proof objective and generated a counterexample that falsified that objective.

### **Orange Highlighting on Model**

Objects that are highlighted in orange have the following meaning, depending on the analysis type.

<b>Analysis mode</b>	<b>Orange highlighting means...</b>
Design error detection	<p>For at least one objective, the analysis could not determine if there was any dead logic, overflow or division-by-zero errors. This situation can occur when:</p> <ul style="list-style-type: none"> <li>• The analysis times out.</li> <li>• The software cannot determine if an error occurred or not. This result is due to: <ul style="list-style-type: none"> <li>▪ Automatic stubbing errors; for more information, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.</li> </ul> </li> </ul>

Analysis mode	Orange highlighting means...
	<ul style="list-style-type: none"> <li>▪ Limitations of the analysis engine</li> </ul>
Test-case generation	<p>The analysis satisfied a test objective, but could not create a test case. This situation can occur when:</p> <ul style="list-style-type: none"> <li>• A test objective depends on nonlinear computation.</li> <li>• A test objective creates an arithmetic error, for example, division by zero.</li> <li>• The analysis times out.</li> <li>• The software cannot determine if a test objective is satisfiable due to automatic stubbing errors or limitations of the analysis engine.</li> </ul>
Property proving	<p>The analysis disproved a proof objective, but could not generate a counterexample, or the proof was undecided. This situation can occur when:</p> <ul style="list-style-type: none"> <li>• A proof objective exists on a signal whose value the software cannot control, for example, a Constant block.</li> <li>• A proof objective depends on nonlinear computation.</li> <li>• A proof objective creates an arithmetic error, for example, division by zero.</li> <li>• The analysis times out.</li> <li>• The software cannot determine if a proof objective can be validated due to automatic stubbing errors or limitations of the analysis engine.</li> </ul>

## Gray Highlighting on Model

Objects that are highlighted in gray have the following meaning.

<b>Analysis mode</b>	<b>Gray highlighting means...</b>
<ul style="list-style-type: none"><li>• Design error detection</li><li>• Test-case generation</li><li>• Property proving</li></ul>	The model object was not part of the analysis.



# Simulink Design Verifier Data Files

## In this section...

“About Simulink® Design Verifier™ Data Files” on page 13-7

“Overview of the sldvData Structure” on page 13-7

“Model Information Fields in sldvData” on page 13-8

“Simulating Models with Simulink® Design Verifier™ Data Files” on page 13-13

## About Simulink Design Verifier Data Files

When you enable the **Save test data to file** parameter (see “Design Verifier Pane: Results” on page 15-46), the Simulink Design Verifier software 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.

## Overview of the sldvData Structure

When the Simulink Design Verifier software 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 13-9
- “ModelObjects Field” on page 13-10
- “Constraints Field” on page 13-11
- “Objectives Field” on page 13-11
- “TestCases Field / CounterExamples Field” on page 13-12
- “Version Field” on page 13-13

### **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.

<b>Subfield Name</b>	<b>Description</b>
Name	String that specifies the model name.
Version	String that specifies the model number.
Author	String that specifies the user name.
TimeStamp	String that specifies the last date and time the model was updated.
SubsystemPath	String that represents the full path name of the subsystem (if any) that was analyzed.
ExtractedModel	String that represents the name of the model extracted (if any) to analyze the subsystem (if any) specified in SubsystemPath.
ReplacementModel	String that specifies the name of the model (if any) that contains the block replacements.

### **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.

<b>Subfield Name</b>	<b>Description</b>
Status	String that specifies 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.

<b>Subfield Name</b>	<b>Description</b>
AbstractedBlocks	For internal use only.
Approximations	A structure that describes the approximations performed during the analysis. For more information about approximations, see “Approximations” on page 2-20.
ReplacementInfo	For internal use only.

### **ModelObjects Field**

In the `sldvData` structure, the `ModelObjects` field lists the model items and their associated objectives. The following table describes each subfield of the `ModelObjects` field.

<b>Subfield Name</b>	<b>Description</b>
<code>descr</code>	String that specifies the full path to a model object, including objects in a Stateflow chart.
<code>typeDesc</code>	String that specifies the block type of the model object.
<code>s1Path</code>	String that specifies the full path to a Simulink model object.
<code>sfObjType</code>	String that specifies the type of a Stateflow object, e.g., S for state and T for transition.
<code>sfObjNum</code>	Integer that represents the unique identifier of a Stateflow object.
<code>sid</code>	For internal use only.
<code>designSid</code>	For internal use only.
<code>replacementSid</code>	For internal use only.
<code>objectives</code>	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
<code>DesignMinMax</code>	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
<code>type</code>	String that specifies the type of an objective.
<code>status</code>	String that specifies the status of an objective.
<code>descr</code>	String that specifies the description of an objective.
<code>label</code>	String that specifies the label of an objective.
<code>outcomeValue</code>	Integer that specifies an objective's outcome.
<code>coveragePointIdx</code>	Integer that represents the index of a coverage point with which an objective is associated.
<code>linkInfo</code>	For internal use only.
<code>range</code>	For internal use only.
<code>modelObjectIdx</code>	Integer that represents the index of a model object with which an objective is associated.
<code>testCaseIdx</code>	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 cases 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
<code>timeValues</code>	Vector that specifies the time values associated with signals in a test case or counterexample.
<code>dataValues</code>	Cell array that specifies the data values associated with signals in a test case or counterexample.
<code>paramValues</code>	Structure that specifies the parameter values associated with a test case or counterexample. Its fields include:  <code>name</code> — String that specifies the name of a parameter. <code>value</code> — Number that specifies the value of a parameter. <code>noEffect</code> — Logical value that specifies whether a parameter's value affects an objective.
<code>stepValues</code>	Vector that specifies the number of time steps that comprise signals in a test case or counterexample.

Subfield Name	Description
objectives	<p>Structure that specifies objectives that a test case or a counterexample addresses. Its fields include:</p> <p><b>objectiveIdx</b> — Integer that represents the index of an objective that a test case achieves or a counterexample falsifies.</p> <p><b>atTime</b> — Time value at which either a test case achieves an objective or a counterexample falsifies an objective.</p> <p><b>atStep</b> — Time step at which either a test case achieves an objective or a counterexample falsifies an objective.</p>
dataNoEffect	<p>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.</p>
expectedOutput	<p>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 Output block in the top-level system. This subfield is populated if you select <b>Include expected output values</b>.</p>

### Version Field

In the `sldvData` structure, the `Version` field is a string that specifies the version of the Simulink Design Verifier software that analyzed the model.

## Simulating Models with Simulink Design Verifier Data Files

The `sldvruntest` function simulates a model using test cases or counterexamples that reside in a Simulink Design Verifier data file:

- 1 Simulate the `sldvdemo_flipflop` model and generate test cases:

```
sldvdemo_flipflop
```

- 2 Save the location of the data file generated after analyzing the model:

```
sldvDataFile = 'sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat'
```

- 3 Use the `sldvruntime` function to simulate the `sldvdemo_flipflop` model using test case 2 in the data file:

```
[ outdata ] = sldvruntime('sldvdemo_flipflop', sldvDataFile, 2)
```

The output from `sldvruntime` 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_sldvruntime = outData(1).find('tout_sldvruntime');  
xout_sldvruntime = outData(1).find('xout_sldvruntime');  
yout_sldvruntime = outData(1).find('yout_sldvruntime');  
logout_sldvruntime = outData(1).find('logout_sldvruntime');
```



## Harness Model

### In this section...

“About the Harness Model” on page 13-15

“Creating a Harness Model” on page 13-15

“Anatomy of a Harness Model” on page 13-16

“Configuration of the Harness Model” on page 13-21

“Simulating the Harness Model” on page 13-22

### About the Harness Model

During or after a Simulink Design Verifier analysis, you can create a harness model.

The contents of the harness depends on the value of the **Mode** parameter on the Configuration Parameters dialog box Design Verifier pane:

- **Design error detection** — The harness model contains test cases that result in integer-overflow or division-by-zero errors.
- **Test generation** — The harness model contains test cases that achieve test objectives.
- **Property proving** — The harness model contains counterexamples that falsify proof objectives.

By default, the **Save test harness as model** parameter is disabled.

---

**Note** The Simulink Design Verifier software can generate a harness model only when the top level of the system you are analyzing contains an Inport block.

---

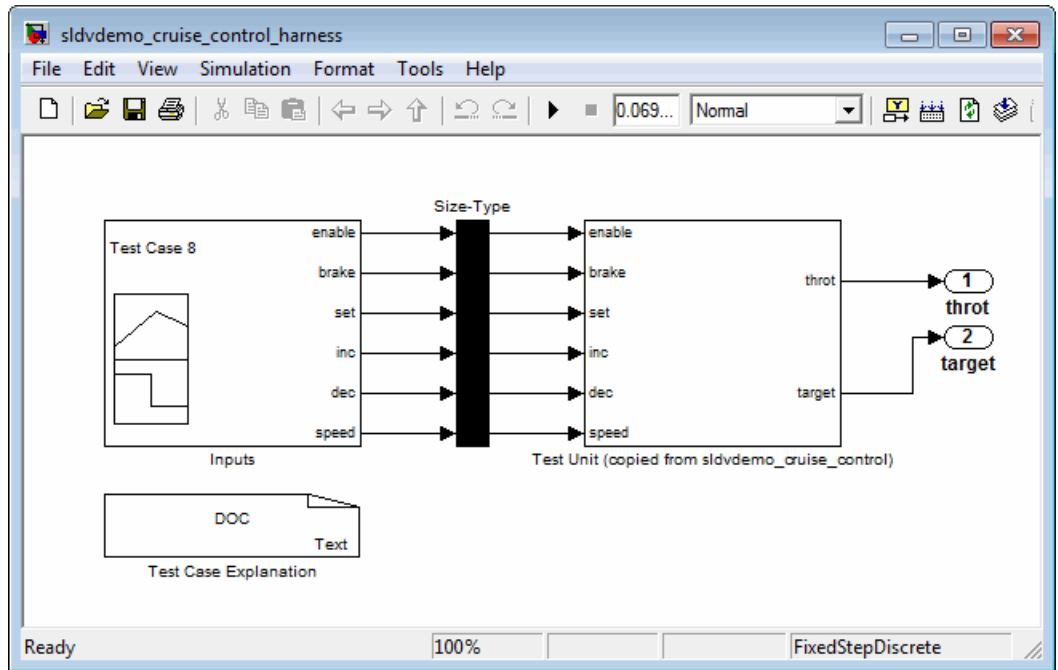
### Creating a Harness Model

To create a harness model before or after the analysis, do one of the following:

- Before the analysis, in the Configuration Parameters dialog box, **Design Verifier > Results** pane, select the **Save test harness as model** parameter.
- After the analysis, in the Simulink Design Verifier log window, select **Create harness model**.

## Anatomy of a Harness Model

The Simulink Design Verifier software produces a harness model that looks like this:



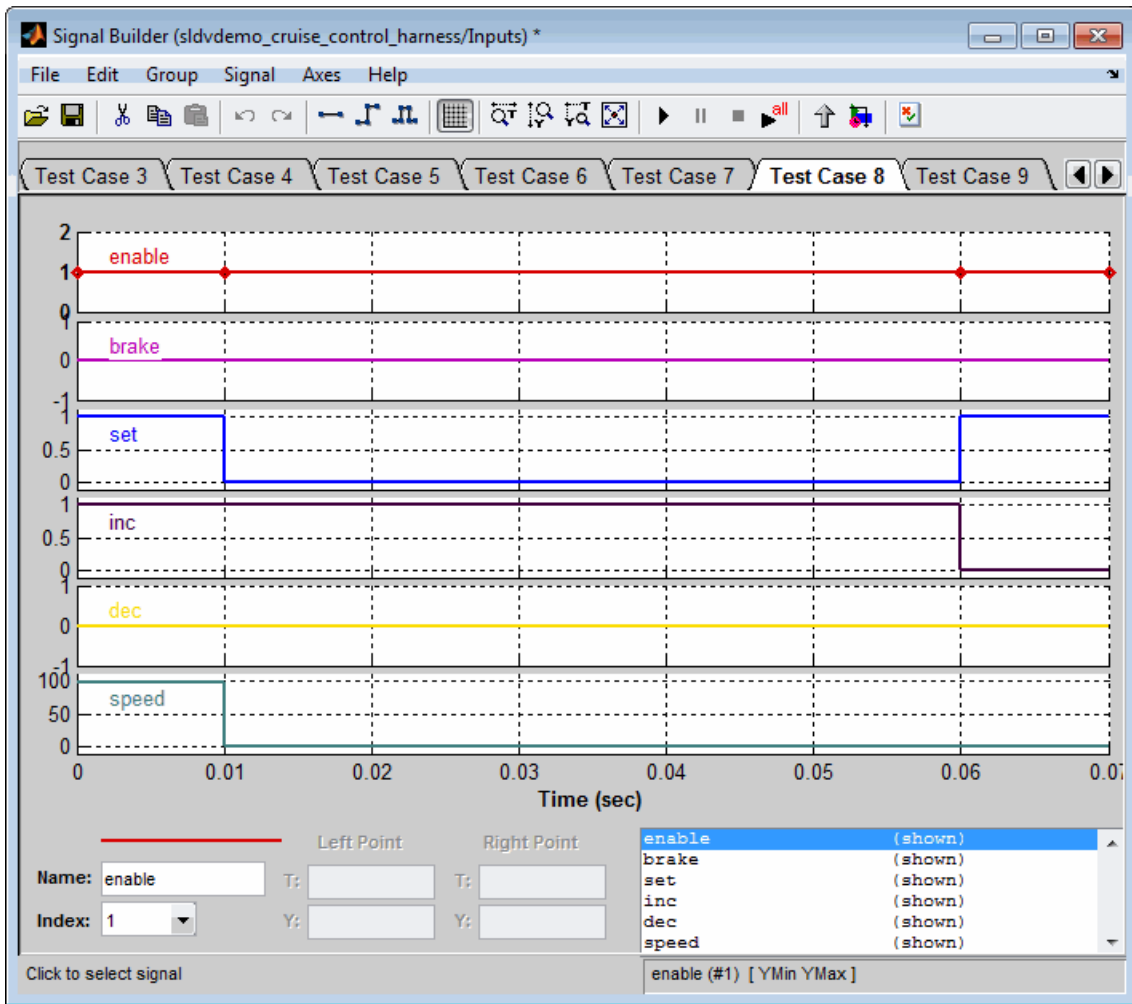
The harness model contains the following items:

- **Inputs** — This Signal Builder block contains signals that comprise the test cases or counterexamples that the Simulink Design Verifier software generated. The Signal Builder block contains signals only for input signals that are used in the model. If an input signal has no effect on the output of the model, that signal is not included in the Signal Builder block.

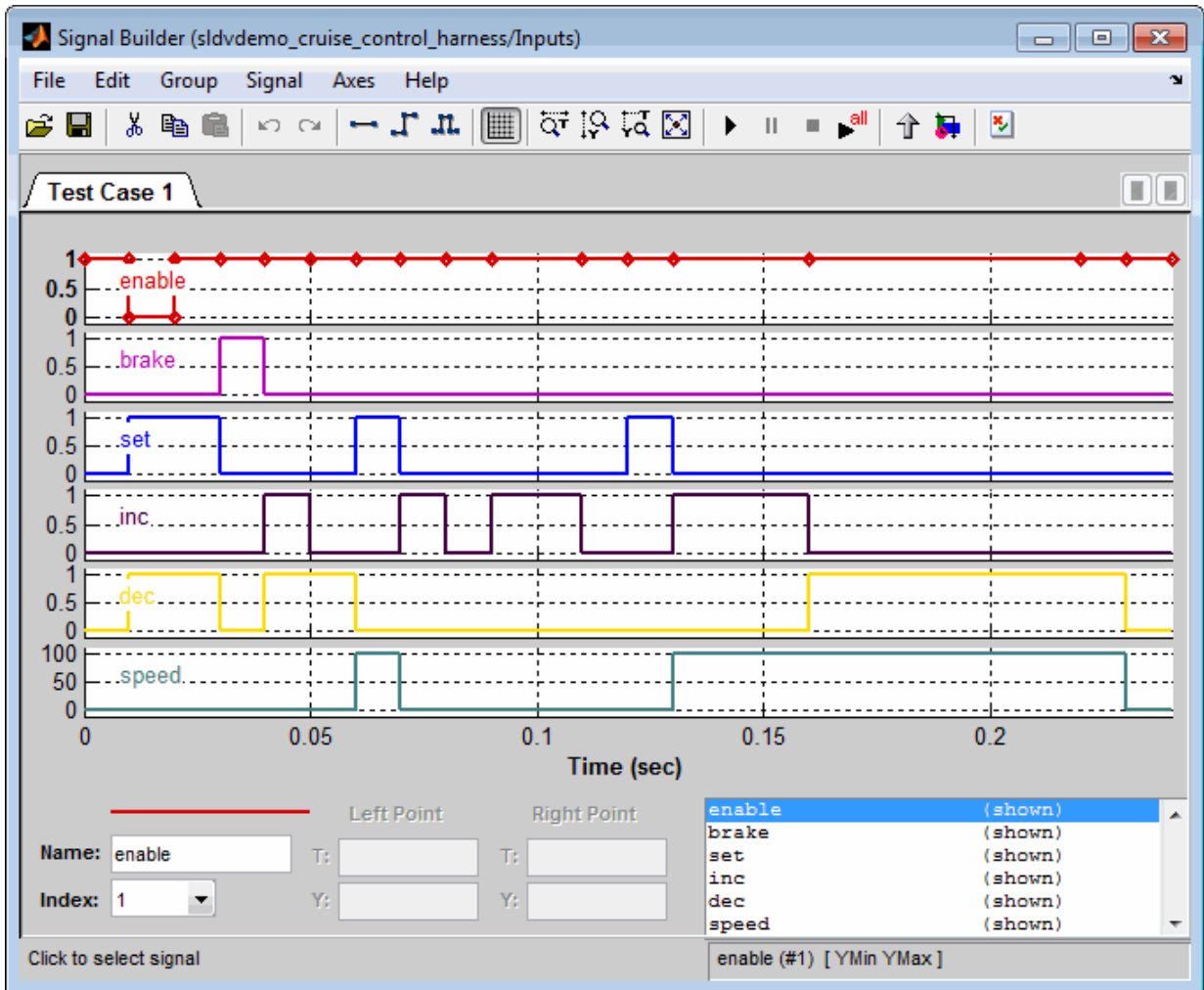
Double-click the Inputs block to open the Signal Builder dialog box and view its signals.

The following Signal Builder block shows the signals for Test Case 8 after analyzing the `sldvdemo_cruise_control` model with the default options.

Each signal group represents a unique test case or counterexample. In the Signal Builder dialog box, select a tab to view the signals associated with a particular test case or counterexample.



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 cases instead of nine shorter test cases. The following Signal Builder dialog box shows the signals for one of those two test cases.



**Note** For more information about the Signal Builder dialog box, see “Working with Signal Groups” in *Simulink User’s Guide*.

- **Size-Type** — This Subsystem block transmits signals from the Inputs block to the Test Unit block. It verifies that the size and data type of the of the signals are consistent with the Test Unit block.
- **Test Unit** — This Subsystem block contains a copy of the original model that the Simulink Design Verifier software analyzed.

If you select the **Reference input model in generated harness** on the **Design Verifier > Results** pane, the Test Unit is a Model block that references the model you are analyzing, not a subsystem.

- **Test Case Explanation** — This DocBlock block documents the test cases or counterexamples that the Simulink Design Verifier software generates. Double-click the Test Case Explanation block to view a description of each test case or counterexample. The block lists either the test objectives that each test case achieves (as in the next graphic) or the proof objectives that each counterexample falsifies.

```

1 | Test Case 1 (8 Objectives)
2 |     Parameter values:
3 |
4 |     1. Controller/Switch1 - logical trigger input t
5 |     2. Controller/Logical Operator1 - Logic: input
6 |     3. Controller/Logical Operator2 - Logic: input
7 |     4. Controller/Logical Operator2 - Logic: MCDC e
8 |     5. Controller/Logical Operator - Logic: input p
9 |     6. Controller/Logical Operator - Logic: input p
10 |    7. Controller/Logical Operator - Logic: MCDC ex
11 |    8. Controller/PI Controller - enable logical va
12 |
13 | Test Case 2 (3 Objectives)
14 |     Parameter values:
15 |
16 |     1. Controller/Logical Operator1 - Logic: input
17 |     2. Controller/Logical Operator - Logic: input p

```

## Configuration of the Harness Model

After the Simulink Design Verifier software generates the harness model, it has the following settings:

- The harness model start time is always 0. If the original model uses a nonzero start time, the software ignores this and always uses 0 for the simulation start time for test cases and counterexamples.
- The harness model stop time always equals the stop time of the longest test case in the Signal Builder dialog box.
- By default, the software enables coverage reporting for harness models that contain test cases. Although it enables coverage reporting with particular

options selected, you can customize the settings to meet your needs. For more information, see “Customizing the Requirements Report” in the *Simulink Verification and Validation User’s Guide*.

- By default, if you select **Ignore objective based on filter** and provide a coverage filter file for the Test Unit, the coverage filter file also applies to the harness model. The coverage objective filter parameters are in the Configuration Parameters dialog box, on the **Test Generation** pane.

## Simulating the Harness Model

The harness model enables you to simulate a copy of your original model using the test cases or counterexamples that the Simulink Design Verifier software generates. Using the harness model, you can simulate:

- A counterexample
- A single test case, for which the Simulink Verification and Validation software collects and displays model coverage information
- All test cases, for which the Simulink Verification and Validation software collects and displays cumulative model coverage information

---

**Note** If you analyze a model that simulates with sample time warnings, when you simulate the harness model, the warnings may be reported as errors, causing the simulation to fail.

---

To simulate a single test case or counterexample:

- 1** In the harness model, double-click the Inputs block.

The Signal Builder dialog box appears.

- 2** In the Signal Builder dialog box, select the tab associated with a particular test case or counterexample.

The Signal Builder dialog displays the signals that comprise the selected test case or counterexample.

- 3** In the Signal Builder dialog box, click the **Start simulation** button .




The Simulink software simulates the harness model using the signals associated with the selected test case or counterexample. When simulating a test case, the Simulink Verification and Validation software collects model coverage information and displays a coverage report.

To simulate all test cases and measure their combined model coverage:

- 1 In the harness model, double-click the Inputs block.

The Signal Builder dialog box appears.

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

The Simulink software simulates the harness model using all test cases, while the Simulink Verification and Validation software collects model coverage information and displays a coverage report.

When you click **Run all**, the software simulates all the test cases using the stop time for the harness model. The stop time equals the stop time for the longest test case, so you may accumulate additional coverage when you simulate the shorter test cases.

If the Test Unit in the harness model is a subsystem, the values of the Simulink simulation optimization parameters on the Configuration Parameters dialog box may impact your coverage results.

---

**Note** The simulation optimization parameters are on the following Configuration Parameters dialog box panes:

- **Optimization** pane
  - **Optimization > Signals and Parameters** pane
  - **Optimization > Stateflow** pane
- 

See “Simulating with Signal Groups” in *Simulink User’s Guide* for more information about simulating models containing Signal Builder blocks.

## SystemTest TEST-Files

If you have installed the SystemTest™ software with your MATLAB application, you can specify that the Simulink Design Verifier software create a SystemTest TEST-file when it analyzes a model. Creating a TEST-file allows you to configure and collect model coverage results and run the test cases from inside the SystemTest environment.

---

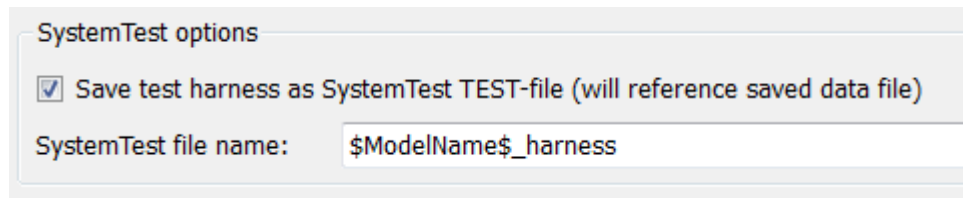
**Note** The option to create a SystemTest TEST-file is only available in test-generation mode; you cannot create this file when running a property-proving analysis.

---

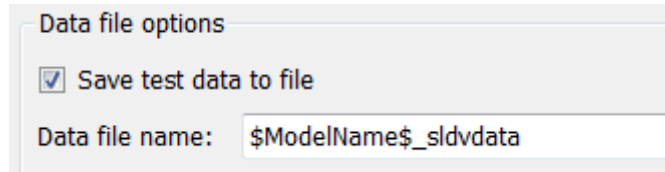
In addition, if you have a model with a large number of inputs, this feature eliminates the overhead of creating the harness model. However, you can create both a harness model and a TEST-file in the same analysis.

To create a TEST-file for the `sldvdemo_cruise_control` model, perform these steps:

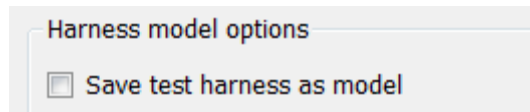
- 1 Type `sldvdemo_cruise_control` at the MATLAB command prompt to open the Cruise Control Test Generation model.
- 2 Select **Tools > Design Verifier > Options** to open the Configuration Parameters dialog box.
- 3 In the **Select** pane, under **Design Verifier**, select **Results**.
- 4 On the **Results** pane, under **SystemTest options**, select **Save test harness as SystemTest TEST-file (will reference saved data file)**.



- 5 If you prefer a file name other than the default, specify the **SystemTest file name**.
- 6 Under **Data File options**, verify that **Save test data to file** is selected. You must select this option to generate a TEST-file.



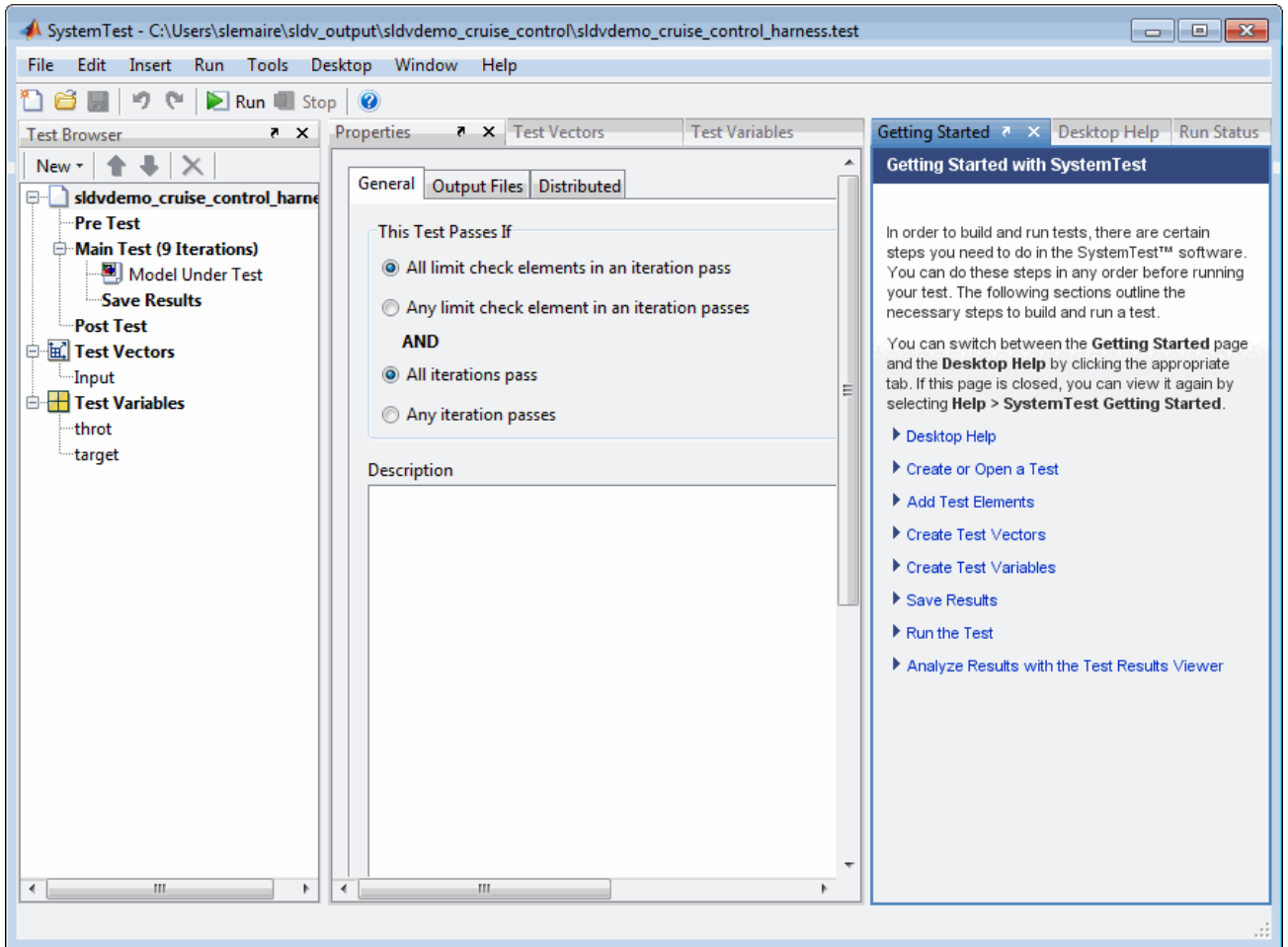
- 7 If you do not need the Simulink Design Verifier harness model in addition to the TEST-file, under **Harness model options**, clear **Save test harness as model**.



- 8 Click **Apply** and **OK** to save the changes and exit the Configuration Parameters dialog box.
- 9 Double-click the Run block in the `sldvdemo_cruise_control` model to start the analysis.

When the analysis completes, the SystemTest desktop opens the TEST-file, which, for this example, is saved as

```
matlabroot\sldvdemo_output\sldv_cruise_control\sldvdemo_cruise_control_harness.test
```



In the **Test Browser** pane, the 9 iterations under Main Test correspond to the 9 test cases the Simulink Design Verifier software generates and describes in the Test Case Explanation block of the harness model.

For information about running the test cases using the SystemTest software, see “Creating a Simulink Design Verifier Data File Test Vector” in the *SystemTest User’s Guide*.

# Simulink Design Verifier Reports

## In this section...

“About Simulink® Design Verifier™ Reports” on page 13-27

“Creating Analysis Reports” on page 13-27

“Front Matter” on page 13-28

“Summary Chapter” on page 13-28

“Analysis Information Chapter” on page 13-28

“Derived Ranges Chapter” on page 13-34

“Objectives Status Chapters” on page 13-35

“Model Items Chapter” on page 13-43

“Design Errors Chapter” on page 13-44

“Test Cases Chapter” on page 13-45

“Properties Chapter” on page 13-50

## About Simulink Design Verifier Reports

After an analysis, the Simulink Design Verifier software can generate an HTML report that contains detailed information about the analysis results.

In addition, 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

## Creating 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, **Design Verifier > Report** pane, select the **Generate report of the results** parameter.

- After the analysis, in the Simulink Design Verifier log window, select **Generate detailed analysis results**.

### **Front Matter**

The report begins with two sections:

- “Title” on page 13-28
- “Table of Contents” on page 13-28

### **Title**

The title section lists the following information:

- Model or subsystem name the Simulink Design Verifier software analyzed
- User name associated with the current MATLAB session
- Date and time that the Simulink Design Verifier software 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 HTML report lists the following information:

- Name of the model
- Analysis mode
- Analysis status
- Status of objectives analyzed

### **Analysis Information Chapter**

The Analysis Information chapter of the HTML report includes the following sections:

- “Model Information” on page 13-29
- “Analysis Options” on page 13-29
- “Unsupported Blocks” on page 13-30
- “Constraints” on page 13-31
- “Block Replacements Summary” on page 13-31
- “Approximations” on page 13-33

### **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 the Simulink Design Verifier software analyzed
- Model version
- Date and time that the model was last saved
- Name of the person who last saved the model

### **Analysis Options**

The Analysis Options section provides information about the Simulink Design Verifier analysis settings.

The Analysis Options section lists the parameters that affected the Simulink Design Verifier analysis. If you enabled coverage filtering, the name of the filter file is included in this section.

## Analysis Options

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

---

**Note** For more information about these parameters, see Chapter 15, “Simulink® Design Verifier™ Configuration Parameters”.

---

## 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 any unsupported blocks; it lists the unsupported block in a table, with a hyperlink to the block in the model.



## Unsupported Blocks

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

Block	Type
<a href="#">Trigonometric Function</a>	Trigonometry

For more information about automatic stubbing, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.

## Constraints

The Constraints section provides information about any test conditions that the Simulink Design Verifier software applied when it analyzed a model.

### Constraints

Name	Constraint
<a href="#">constraint</a>	[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 the Simulink Design Verifier software executed. It appears only if the Simulink Design Verifier software replaced any blocks in a model.

Each row of the table corresponds to a particular block replacement rule that the Simulink Design Verifier software 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 any blocks that the Simulink Design Verifier software replaced in the model

To locate a particular block replacement in your model, click on the name for that replacement in the Replaced Blocks column of the table; the software highlights the affected block in your model window and opens a new window that displays the block in detail.

## Block Replacements Summary

Table 2.1. Block Replacements

#:	Replacement Rule / Block Type	Rule Description	Replaced Blocks
1	blkrep_rule_lookup_normal.m /Lookup	Inserts test objectives for each interval of 1-D lookup table blocks.	<a href="#">Lookup Table</a>
2	blkrep_rule_switch_normal.m /Switch	Inserts test objectives for switch blocks that require each switch position be demonstrated when the values of input ports 1 and 3 differ.	<a href="#">Switch</a>
3	sldvdemo_custom_blkrep_rule_sqrt.m /Math	Approximates the mathematical function sqrt using lookup table. The input range is constrained to [0 10000].	<a href="#">Math Function</a>

See Chapter 4, “Working with Block Replacements” for more information.

### Approximations

Each row of the Approximations table describes a specific type of approximation that the Simulink Design Verifier software used during its analysis of the model.

## Approximations

Simulink Design Verifier performed the following approximations during analysis. These can impact the precision of the results generated by Simulink Design Verifier. Please see the product documentation for further details.

	Type	Description
1	Rational approximation	The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.

---

**Note** Review the analysis results carefully when the software uses approximations. In rare cases, an approximation may result in test cases that fail to achieve test objectives or counterexamples that fail to falsify proof objectives. For example, a floating-point round-off error might prevent a signal from exceeding a designated threshold value.

---

## Derived Ranges Chapter

In a design error detection analysis, the analysis calculates the derived ranges of the signal values for the Outports for each block in the model. This information can help you identify the source of data overflow or division-by-zero errors.

The table in the Derived Ranges chapter of the analysis report lists these bounds.

## Chapter 3. Derived Ranges

Signal	Derived Ranges
<a href="#">Controller/Constant1- output 1</a>	1
<a href="#">Controller/Unit Delay- output 1</a>	[-Inf..Inf]
<a href="#">Controller/Sum- output 1</a>	[-Inf..Inf]
<a href="#">Controller/Constant3- output 1</a>	1
<a href="#">Controller/Sum2- output 1</a>	[-Inf..Inf]
<a href="#">Controller/Switch3- output 1</a>	[-Inf..Inf]
<a href="#">Controller/Switch2- output 1</a>	[-Inf..Inf]
<a href="#">Controller/Switch1- output 1</a>	[-Inf..Inf]
<a href="#">Controller/Sum1- output 1</a>	[-Inf..Inf]
<a href="#">Controller/Logical Operator1- output 1</a>	[F..T]
<a href="#">Controller/Unit Delay1- output 1</a>	[F..T]
<a href="#">Controller/Logical Operator2- output 1</a>	[F..T]
<a href="#">Controller/Logical Operator- output 1</a>	[F..T]
<a href="#">throt- output 1</a>	[-3.5954e+306..Inf]
<a href="#">target- output 1</a>	[-Inf..Inf]
<a href="#">Controller/PI Controller/Discrete-Time Integrator- output 1</a>	[-5..5]
<a href="#">Controller/PI Controller/Kp- output 1</a>	[-3.5954e+306..Inf]
<a href="#">Controller/PI Controller/Kp1- output 1</a>	[-0.05..0.05]
<a href="#">Controller/PI Controller/Sum- output 1</a>	[-3.5954e+306..Inf]

### Objectives Status Chapters

This section of the report provides information about all objectives in a model, including an objective's type, the model item to which it corresponds, and its description.

- “Design Error Detection Objectives Status” on page 13-36
- “Test Objectives Status” on page 13-38
- “Proof Objectives Status” on page 13-41
- “Objectives Undecided” on page 13-42
- “Objectives Producing Errors” on page 13-43

### Design Error Detection Objectives Status

If you run a design error detection analysis, the Design Error Detection Objectives Status section can include the following tables:

- “Active Logic” on page 13-36
- “Dead Logic” on page 13-36
- “Objectives Proven Valid” on page 13-37
- “Objectives Falsified with Test Cases” on page 13-37
- “Undecided Due to Stubbing” on page 13-38

**Active Logic.** The active logic section lists the model items for which the analysis found active logic. Design error detection analysis results for dead logic on `sldvdemo_fuelsys_logic`:

#### Active Logic

Simulink Design Verifier found that these decision and condition outcomes can occur and are active logic in the model.

#	Type	Model Item	Description
5	Decision	<a href="#">control logic.Oxygen_Sensor_Mode</a>	State: Substate executed State "O2_fail"

**Dead Logic.** The dead logic section lists the model items for which the analysis found dead logic. Design error analysis detection results for dead logic on `sldvdemo_fuelsys_logic`:

## Dead Logic

Simulink Design Verifier proved that these decision and condition outcomes cannot occur and are dead-logic in the model. Dead-logic in the model can also be a side-effect of parameter configurations, input specified minimum maximum constraints, or in rare cases, the approximations performed by Simulink Design Verifier.

#	Type	Model Item	Description
57	Condition	<a href="#">control logic.Speed_Sensor_Mode."[speed==0 &amp; press &lt; zero_th..."</a>	Transition: Condition 2, "press < zero_thresh" F
127	Condition	<a href="#">control logic.Fueling_Mode.Fuel_Disabled."[in(MultiFail)]"</a>	Transition: <Unspecified condition> F
128	Decision	<a href="#">control logic.Fueling_Mode.Fuel_Disabled."[in(MultiFail)]"</a>	Transition: Transition trigger expression F

**Objectives Proven Valid.** The Objectives Proven Valid section lists the design error detection objectives that the analysis proved valid; no design errors can occur for these objectives.

### Objectives Proven Valid

#	Type	Model Item	Description	Test Case
20	Overflow	<a href="#">debounce</a>	Overflow	n/a

**Objectives Falsified with Test Cases.** The Objectives Falsified with Test Cases section lists the objectives for which the analysis found test cases that demonstrate design errors.

### Objectives Falsified with Test Cases

#	Type	Model Item	Description	Test Case
12	Overflow	<a href="#">Verify True Output/Sum</a>	Overflow	<a href="#">1</a>

**Undecided Due to Stubbing.** The undecided due to stubbing section lists the model items for which the analysis was unable to decide on the objectives due to stubbing.

### Undecided due to stubbing

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

#	Type	Model Item	Description
4	Decision	<a href="#">Subsystem/Saturation</a>	input(1) > lower limit F
5	Decision	<a href="#">Subsystem/Saturation</a>	input(1) > lower limit T
6	Decision	<a href="#">Subsystem/Saturation</a>	input(1) >= upper limit F
7	Decision	<a href="#">Subsystem/Saturation</a>	input(1) >= upper limit T

### Test Objectives Status

If run a test-case generation analysis, the Test Objectives Status section can include the following tables:

- “Objectives Satisfied” on page 13-38
- “Objectives Satisfied - No Test Case” on page 13-39
- “Objectives Proven Unsatisfiable” on page 13-40

**Objectives Satisfied.** — The Objectives Satisfied section lists test objectives that the analysis satisfied.



## Objectives Satisfied

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

#:	Type	Model Item	Description	Test Case
1	Decision	<a href="#">PI Controller</a>	enable logical value F	<a href="#">2</a>
2	Decision	<a href="#">PI Controller</a>	enable logical value T	<a href="#">1</a>
3	Decision	<a href="#">P Controller</a>	enable logical value F	<a href="#">1</a>
4	Decision	<a href="#">P Controller</a>	enable logical value T	<a href="#">2</a>
5	Decision	<a href="#">mp switch</a>	integer input value = 1 (output is from input port 2)	<a href="#">1</a>
6	Decision	<a href="#">mp switch</a>	integer input value = 2 (output is from input port 3)	<a href="#">2</a>

**Objectives Satisfied - No Test Case.** — The Objectives Satisfied - No Test Case section lists test objectives that the analysis satisfied without generating test cases.

Situations when the software might satisfy an objective but not create a test case are:

- Test objectives that depend on nonlinear computation
- If the test objective creates an arithmetic error, such as division by zero
- If you enable automatic stubbing, and the analysis encounters an unsupported block whose operation it does not understand
- A test case exists that depends on the behavior of a stubbed block

In the following example, nonlinear computation causes the signal to the control input of the Saturation block to be out of range:

### Objectives Satisfied - No Test Case

#:	Type	Model Item	Description	Test Case
2	Decision	<a href="#">Saturation</a>	input > lower limit T	n/a
3	Decision	<a href="#">Saturation</a>	input >= upper limit F	n/a
4	Decision	<a href="#">Saturation</a>	input >= upper limit T	n/a

**Objectives Proven Unsatisfiable.** — The Objectives Proven Unsatisfiable section lists the test objectives that the analysis determined could not be satisfied.

In the following example, the software found that there are no test cases that achieve these objectives because the inputs to the Logical Operator are never false.

### Objectives Proven Unsatisfiable

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

#	Type	Model Item	Description	Test Case
2	Condition	<a href="#">Logical Operator</a>	Logic: input port 1 F	n/a
4	Condition	<a href="#">Logical Operator</a>	Logic: input port 2 F	n/a

In the following example, the Stateflow chart ShiftLogic updates at every time step, so the implicit event tick is never false. The analysis cannot satisfy condition or MCDC coverage for the temporal event `after(TWAIT, tick)`.

### Objectives Proven Unsatisfiable

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

#	Type	Model Item	Description	Test Case
5	Condition	<a href="#">ShiftLogic.selection.state."after(TWAIT,tick) [speed &lt;=..."</a>	Transition: Condition 1, "tick" F	n/a
15	Mcdc	<a href="#">ShiftLogic.selection.state."after(TWAIT,tick) [speed &lt;=..."</a>	Transition: MCDC Transition trigger expression with Condition 1, "tick" F	n/a

### Proof Objectives Status

If you run a property-proving analysis, the Proof Objectives Status section can include:

- “Objectives Proven Valid” on page 13-41
- “Objectives Falsified with Counterexamples” on page 13-42
- “Objectives Falsified - No Counterexample” on page 13-42

**Objectives Proven Valid.** The Objectives Proven Valid section lists the proof objectives that the analysis proved valid.

### Objectives Proven Valid

#:	Type	Model Item	Description	Counterexample
1	Custom Proof Objective	<a href="#">Proof Objective</a>	Objective: 1	n/a

**Objectives Falsified with Counterexamples.** The Objectives Falsified with Counterexamples section lists the proof objectives that the analysis disproved. In this example, the software generated at least one counterexample that falsifies the specified objectives.

#:	Type	Model Item	Description	Counterexample
1	Assert	<a href="#">Verify True Output/Assertion</a>	assert	<a href="#">1</a>

**Objectives Falsified - No Counterexample.** The Objectives Falsified - No Counterexample section lists the proof objectives that the analysis disproved without generating counterexamples. This occurs if, for example, you specified a proof objective on a signal whose value the software cannot control, or the software encountered a divide-by-zero error when instantiating a counterexample.

#:	Type	Model Item	Description	Counterexample
1	Custom Proof Objective	<a href="#">Proof Objective</a>	Objective: F	n/a
2	Custom Proof Objective	<a href="#">Proof Objective1</a>	Objective: T	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 the following property-proving example, either the software exceeded its analysis time limit (which the **Maximum analysis time** parameter specifies), or you aborted the analysis before it completed processing these objectives.

## Objectives Undecided

Simulink Design Verifier was not able to process these objectives with the current options.

#:	Type	Model Item	Description	Counterexample
1	Custom Proof Objective	<a href="#">Verify Output/FoutCorrect</a>	Objective: T	n/a
2	Custom Proof Objective	<a href="#">Verify Output/ToutCorrect</a>	Objective: T	n/a

## Objectives Producing Errors

For all types of objectives, the Objectives Producing Errors table lists the objectives for which the analysis encountered errors during its analysis.

In the following example, analyzing these objectives involves nonlinear arithmetic, which the software does not support.

## Objectives Producing Errors

#:	Type	Model Item	Description	Test Case
4	Decision	<a href="#">Mode switch</a>	logical trigger input true (output is from 1st input port)	n/a
8	Decision	<a href="#">Basic Roll Mode/Integrator</a>	integration result <= lower limit T	n/a
10	Decision	<a href="#">Basic Roll Mode/Integrator</a>	integration result >= upper limit T	n/a

## Model Items Chapter

The Model Items chapter of the HTML report includes a table for each object in the model that defines coverage objectives. The table for a particular object lists all of the associated objectives, the objective types, objective descriptions, and the status of each objective at the end of the analysis.

The table for an individual object in the model will look similar to this one for the TK switch in the Roll Reference subsystem.

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.

<b>Roll Reference/TK switch</b>				
<a href="#">View</a>				
<b>#:</b>	<b>Type</b>	<b>Description</b>	<b>Status</b>	<b>Test Case</b>
27	Decision	logical trigger input false (output is from 3rd input port)	Produced error	n/a
28	Decision	logical trigger input true (output is from 1st input port)	Satisfied	<a href="#">1</a>

## Design Errors Chapter

If you run a design error detection analysis, the report includes a Design Errors chapter. This chapter includes sections that summarize the design errors the analysis validated or falsified:

- “Table of Contents” on page 13-44
- “Summary” on page 13-44
- “Test Case” on page 13-45

### 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.

<b>Summary</b>	
Model Item:	<a href="#">Verify True Output/Sum</a>
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.

<b>Test Case</b>	
Time	0-0.01
Step	1-2
raw	255

### Test Cases Chapter

If you run a test-case 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 13-46
- “Summary” on page 13-46
- “Objectives” on page 13-46
- “Generated Input Data” on page 13-47

- “Expected Output” on page 13-48
- “Combined Objectives” on page 13-48
- “Long Test Cases” on page 13-49

## Table of Contents

Each Test Cases chapter contains a table of contents. Each item in the table of contents is a hyperlink to information about a specific test case.

## Summary

The Summary section lists:

- Length of the signals that comprise the test case
- Total number of test objectives that the test case achieves

Summary	
Length:	0.06 Seconds (7 sample periods)
Objective Count:	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 Model Editor.
- The objective that was achieved.



**Objectives**

Step	Time	Model Item	Objectives
7	0.06	<a href="#">Controller/PI</a> <a href="#">Controller/Discrete-Time Integrator</a>	integration result $\geq$ upper limit T

**Generated Input Data**

For each input signal associated with the model item, the Generated Input Data section lists the time step and corresponding time at which the test case achieves particular test objectives. If the signal value does not change over those time steps, the table lists the time step and time as ranges.

**Generated Input Data**

Time	0	0.01-0.02	0.03	0.04-0.05	0.06
Step	1	2-3	4	5-6	7
enable	1	1	1	1	1
brake	0	0	0	0	0
set	1	0	0	0	1
inc	1	1	1	1	-
dec	0	0	1	0	-
speed	97	0	0	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 does not affect outcome” on page 15-52).

---

**Expected Output**

If you select the **Include expected output values** on the **Design Verifier > Results** pane of the Configuration Parameters dialog box, the report includes the Expected Output section for each test case. For each output signal associated with the model item, this table lists the expected output value at each time step.

**Expected Output** These output values are expected assuming that inputs that do not affect the test objectives (- in the table above) are given a default value - 0 for numeric types, and default value for enumerated types.

Time	0	0.01	0.02	0.03	0.04	0.05	0.06
Step	1	2	3	4	5	6	7
throt	0	1.96	1.9898	2.0197	2.0497	2.0798	0.05
target	97	98	99	100	101	102	0

**Combined Objectives**

If you set the **Test suite optimization** option to CombinedObjectives (the default), the Test Cases chapter may include individual information about many test cases.

## Chapter 5. Test Cases

### Table of Contents

- [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](#)
- [Test Case 10](#)

This section contains detailed information about each generated test case.

### Test Case 1

#### Summary

Length: 0 Seconds (1 sample periods)

Objective Count: 3

### Long Test Cases

If you set the **Test suite optimization** option to LongTestcases, the Test Cases chapter in the report includes fewer sections about longer test cases.

## Chapter 5. Test Cases

### Table of Contents

#### [Test Case 1](#)

This section contains detailed information about each generated test case.

## Test Case 1

### Summary

Length: 0.23 Seconds (24 sample periods)  
Objective Count: 34

## 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 13-50
- “Summary” on page 13-51
- “Counterexample” on page 13-51

## 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 falsified
- The status of the analysis: Falsified

In the following example, the software analyzed the `sldvdemo_cruise_control_verification` model for property proving. The analysis proved that the input to the Assertion block named `BrakeAssertion` was nonzero.

Summary	
Model Item:	<a href="#">Verify True Output/Sum</a>
Type:	Overflow
Status:	Falsified

## Counterexample

The Counterexample section lists the time step and corresponding time at which the counterexample falsified the property. This section also lists the values of the signals at that time step.

Counterexample		
Time	00.010	0.02-0.04
Step	12	3-5
InputData.Actual_speed	00	0
InputData.Switches.enable	11	0
InputData.Switches.brake	00	1
InputData.Switches.set	10	0
InputData.Switches.setIncDec.inc	11	0
InputData.Switches.setIncDec.dec	00	0

## Simulink Design Verifier Log Files

Every time you analyze a model, the Simulink Design Verifier software 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.

```
15-Oct-2010 10:01:11
Starting test generation for model 'sldvdemo_cruise_control'
Compiling model... done
Translating model... done
'sldvdemo_cruise_control' is compatible with Simulink Design Verifier.

Generating tests...

SATISFIED
Controller/Switch1
logical trigger input true (output is from 1st input port)

SATISFIED
Controller/Logical Operator1
Logic: input port 1 T

SATISFIED
Controller/Logical Operator2
Logic: input port 1 T
```

## Reviewing Analysis Results in the Model Explorer

If you close the analysis results so you can fix the cause of any analysis errors in your model, you may need to review the analysis results again. As long as your model remains open, you can view the results of your most recent Simulink Design Verifier analysis results in the Model Explorer. After you close your model, you can no longer view any analysis results.

In the model window, select **Tools > Design Verifier > Latest Results**. The Model Explorer opens, and the results of the latest Simulink Design Verifier analysis appear in the right-hand pane.

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

Task	For more information
Highlight the analysis results on the model.	“Highlighted Results on the Model” on page 13-2
Generate a detailed analysis report.	“Simulink® Design Verifier™ Reports” on page 13-27
<p>Create the harness model, or if the harness model already exists, open it.</p> <p>You will not be able to create the harness model if:</p> <ul style="list-style-type: none"> <li>• No design error objectives were falsified</li> <li>• No test cases were generated</li> <li>• No counterexamples were created</li> </ul>	“Harness Model” on page 13-15
View the data file.	“Simulink® Design Verifier™ Data Files” on page 13-7
View the log file.	“Simulink® Design Verifier™ Log Files” on page 13-52





# Analyzing Large Models and Improving Performance

---

- “Sources of Model Complexity” on page 14-2
- “Analyzing a Large Model” on page 14-3
- “Generating Reports for Large Models” on page 14-8
- “Managing Model Data to Simplify the Analysis” on page 14-9
- “Partitioning Model Inputs and Generating Tests Incrementally” on page 14-13
- “Analyzing the Model Using a Bottom-Up Approach” on page 14-15
- “Extracting Subsystems for Analysis” on page 14-16
- “Analyzing Logical Operations” on page 14-23
- “Handling Models with Large State Spaces” on page 14-24
- “Handling Problems with Counters and Timers” on page 14-25
- “Techniques for Proving Properties of Large Models” on page 14-27

### 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 signal count in the model

The following sections describe techniques designed to reduce the impact of this complexity and achieve the best performance from the Simulink Design Verifier software.

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, “Techniques for Proving Properties of Large Models” on page 14-27 describes specific techniques for proving properties in a large model.

## Analyzing a Large Model

In this section...
“Types of Large Model Problems” on page 14-3
“Using the Default Parameter Values” on page 14-4
“Modifying the Analysis Parameters” on page 14-5
“Using the Large Model Optimization” on page 14-6
“Stopping the Analysis Before Completion” on page 14-6

### Types of Large Model Problems

The Simulink Design Verifier software may encounter some of these problems when analyzing a large model:

- Unsatisfiable objectives — The software proved there are no test cases that exercise these test objectives, and did not generate any test cases.
- Undecided objectives — The software was not able to satisfy or falsify these objectives.
- Objectives with errors — This problem usually occurs when a model component uses nonlinear arithmetic, which can affect a test objective.
- Cannot complete the analysis in the time allotted — This problem may indicate an area of your model where the software encountered problems, or you may need to increase value of the **Maximum analysis time** parameter.
- Analysis hangs — If the number of objectives processed remains constant for a considerable length of time, the software has likely encountered complexity between the model and its objectives.
- Does not achieve a high percentage of model coverage — When you run the test cases on the harness model, the percentage of model coverage is insufficient for your design.

The next few sections describe the initial steps to take when analyzing a large model. Although these steps address test generation, you can use a similar approach when detecting design errors or proving properties in a model.

## Using the Default Parameter Values

When you generate test cases for a model, whether large or small, the first step is to analyze the model using the Simulink Design Verifier default parameter values:

- 1 Check to see if your model is compatible with the Simulink Design Verifier software, as described in Chapter 3, “Ensuring Compatibility with the Simulink® Design Verifier™ Software”.
- 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	600 (seconds)	If the analysis does not finish within the specified time, the analysis times out and terminates.
Test suite optimization	Combined objectives	Generates test cases that address more than one test objective (if possible).
Model coverage objectives	MCDC	Generates test cases that achieve modified condition/decision coverage (MCDC), which includes decision coverage (DC) and condition coverage (CC).  <b>Note</b> MCDC test cases are not generated for XOR configured logic operators. You can achieve MCDC coverage by using the same tests that would be generated from AND configured blocks or OR configured blocks.

- 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, 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.
- 5** For a test-case 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 Chapter 5, “Specifying Parameter Configurations”.

---

## Modifying 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. Such an increase 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.

### Using 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.

### Stopping 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.

<b>Problem in your model</b>	<b>More information</b>
Floating-point inputs	“Managing Model Data to Simplify the Analysis” on page 14-9
Nonlinear operations	<ul style="list-style-type: none"> <li>• “Analyzing the Model Using a Bottom-Up Approach” on page 14-15</li> <li>• “Analyzing Logical Operations” on page 14-23</li> </ul>
Large state spaces	“Handling Models with Large State Spaces” on page 14-24
Large timers and time delays	“Handling Problems with Counters and Timers” on page 14-25

### Generating Reports for Large Models

When you analyze a model with a large root-level input signal count, you may encounter an insufficient memory error when the Simulink Design Verifier software is generating the report.

When this occurs, you need to increase the amount of memory the Sun Java Virtual Machine (JVM™) software can allocate. For steps on how to increase this memory, see “Increase the MATLAB JVM Memory Allocation Limit” in the MATLAB Report Generator™ documentation.



## Managing Model Data to Simplify the Analysis

In this section...
“Simplifying Data Types” on page 14-9
“Constraining Data” on page 14-9

### Simplifying Data Types

One way to simplify your model is to use for the designated signal data type a data type requiring the smallest 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.

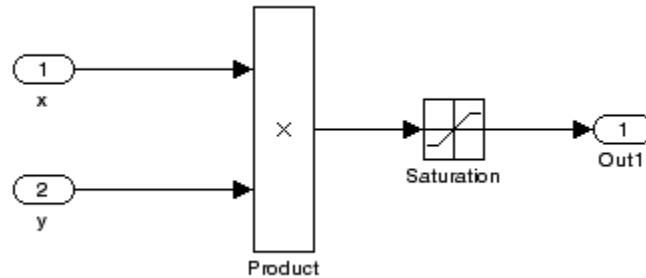
In another example, suppose you have a Sum block with two inputs that are always integers between  $-10$  and  $10$ . Set the **Output data type** parameter to `int8`, rather than `int32` or `double`, or any other data type that requiring the least amount of space. .

To display the signal data types in the model window, select **Format > Port/Signal Displays > Port Data Types**.

### Constraining 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.



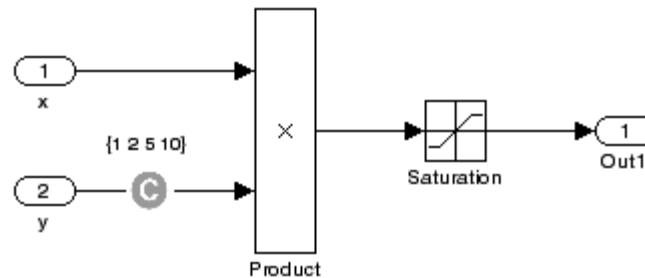
The Simulink Design Verifier software generates errors when attempting to satisfy the upper and lower limits of the Saturation block, because the software does not support nonlinear arithmetic. To work around these errors, restrict one of the inputs to a set of discrete values.

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 “Partitioning Model Inputs and Generating Tests Incrementally” on page 14-13.

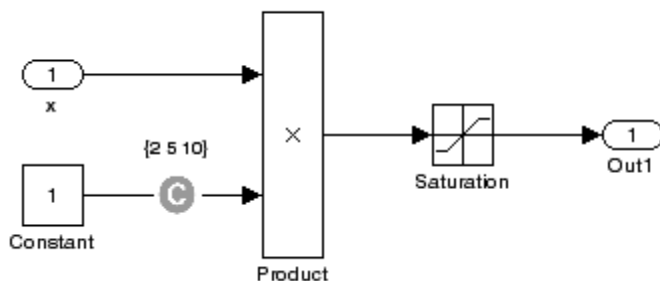
---

Test Condition blocks do not need to be placed exactly on the inputs. In deciding where to place the constraints in your model, consider the following guidelines:

- Favor constraints on the input values because the software can process inputs easier.
- If you need to place constraints on both the input and the output, for example, to avoid nonlinear arithmetic, one of the constraints should be a range such as [minval maxval]. The software first tests the values at both ends of the range and can return a test case, even if the underlying calculations are nonlinear.
- Make sure that constraints at corresponding input and output points are not contradictory. Do not constrain the output signals to values that are not achievable because of the constraints on the input values.

- Avoid creating constraints that contradict the model. Such contradictions occur when a constraint can never be satisfied because it contradicts some aspect of the model or another constraint. Analyzing contradictory models can cause the Simulink Design Verifier software 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.

## Partitioning Model Inputs and Generating Tests Incrementally

As described in “Constraining Data” on page 14-9, you can constrain the values of model inputs using the Simulink Design Verifier Test Condition block.

Like other Simulink parameters, constraint values can be shared across several blocks by referencing a common workspace variable; you can initialize constraint values using MATLAB commands. If you have several inputs related to speed, such as desired speed, measured speed, and average speed, you might choose to constrain all of them to the same set of values.

As an advanced technique for experienced MATLAB programmers, you can use parameterized constraints and successive runs of the Simulink Design Verifier software 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** The Extending an Existing Test Suite demo shows how to extend a test suite so that it satisfies missing model coverage.

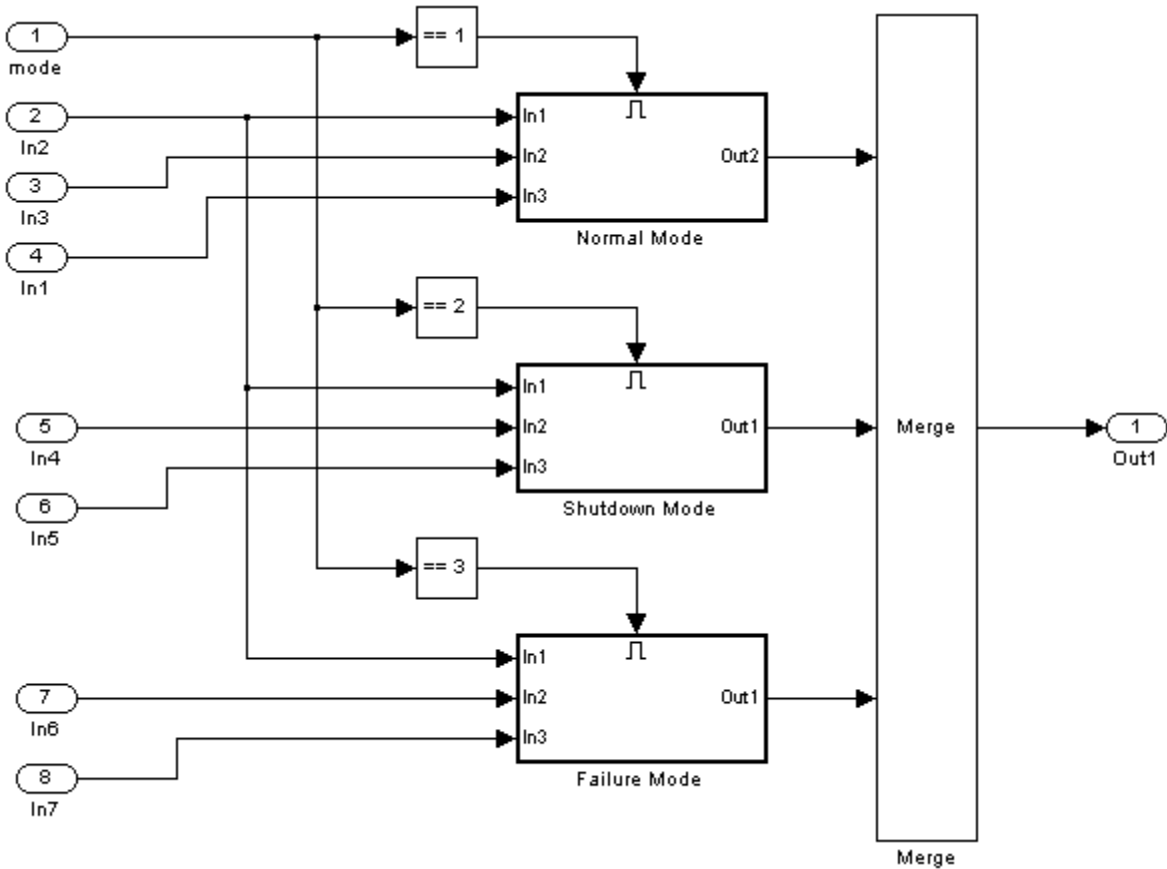
---

- 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.

## Analyzing the Model Using a Bottom-Up Approach

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, it's unreachable, that situation can prevent the software from generating tests for *all* the objectives in a large model.

Try this workflow with your large model:

- 1** Break down the model into components of 100–1000 objectives each. Use the `sldvextract` function to extract components into a new model for analysis purposes.
- 2** Analyze the individual components, starting with the lowest level subsystems.
- 3** Fix any problems by adding constraints or specifying block replacements.
- 4** After you analyze the smaller components, reapply the required constraints and substitutions to the original model and analyze the full model.

When you finish a bottom-up analysis, you should have a top-level model that the Simulink Design Verifier software can analyze quickly.

## Extracting Subsystems for Analysis

In this section...
“Overview of Subsystem Extraction” on page 14-16
“sldvextract Function” on page 14-17
“Structure of the Extracted Model” on page 14-17
“Analyzing Subsystems That Read from Global Data Storage” on page 14-17
“Analyzing Function-Call Subsystems” on page 14-19

### 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 the Simulink Design Verifier software. This technique allows you to implement a bottom-up approach to analyzing a large model, as described in “Analyzing the Model Using a Bottom-Up Approach” on page 14-15.

When you analyze a subsystem or atomic subchart, the software:

- Extracts the subsystem or subchart into a new model.
- If required, adds blocks to the newly created model that replicate the execution context of the subsystem or subchart within its parent model.
- Analyzes the extracted model and produces results.

---

**Note** The Simulink Design Verifier software can only analyze atomic subsystems and atomic subcharts.

For more information about analyzing subsystems, see “Analyzing a Subsystem” on page 1-29.

For more information about analyzing atomic subcharts, see “Analyzing a Stateflow Atomic Subchart” on page 1-31.

---



## sldvextract Function

The `sldvextract` function allows you to extract subsystems and atomic subcharts for component verification, as described in Chapter 10, “Verifying Model Components”. 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.

## Structure of the Extracted Model

When you analyze a subsystem or atomic subchart, the Simulink Design Verifier software 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*.mdl, where name is the *subsystem\_name* of the subsystem.

The next sections provide examples of how the Simulink Design Verifier software extracts and analyzes subsystems.

## Analyzing 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` demo model:

- 1 Open the `sl_subsys_fcncall8` demo model:

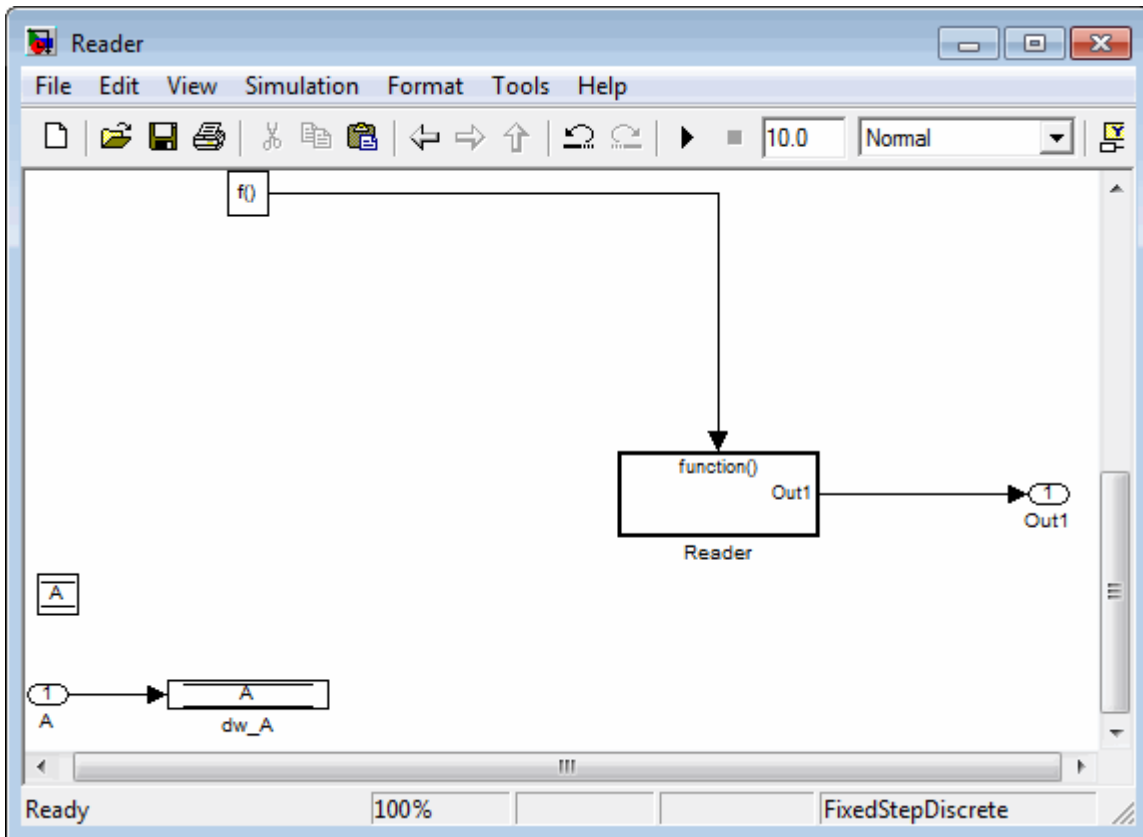
```
sl_subsys_fcncall8
```

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 `matlabroot\sldv_output\Reader`.



The new Inport block A writes into the data store, which is used by the subsystem Reader in the new model.

## Analyzing Function-Call Subsystems

A *function-call subsystem* is a triggered subsystem whose execution is determined by logic internal to a C MEX S-function instead of by the value of a signal. Function-call subsystems are always atomic.

---

**Note** For more information, see “Function-Call Subsystems” in the Simulink documentation.

---

When you analyze a model with a function-call subsystem, the Simulink Design Verifier software 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:

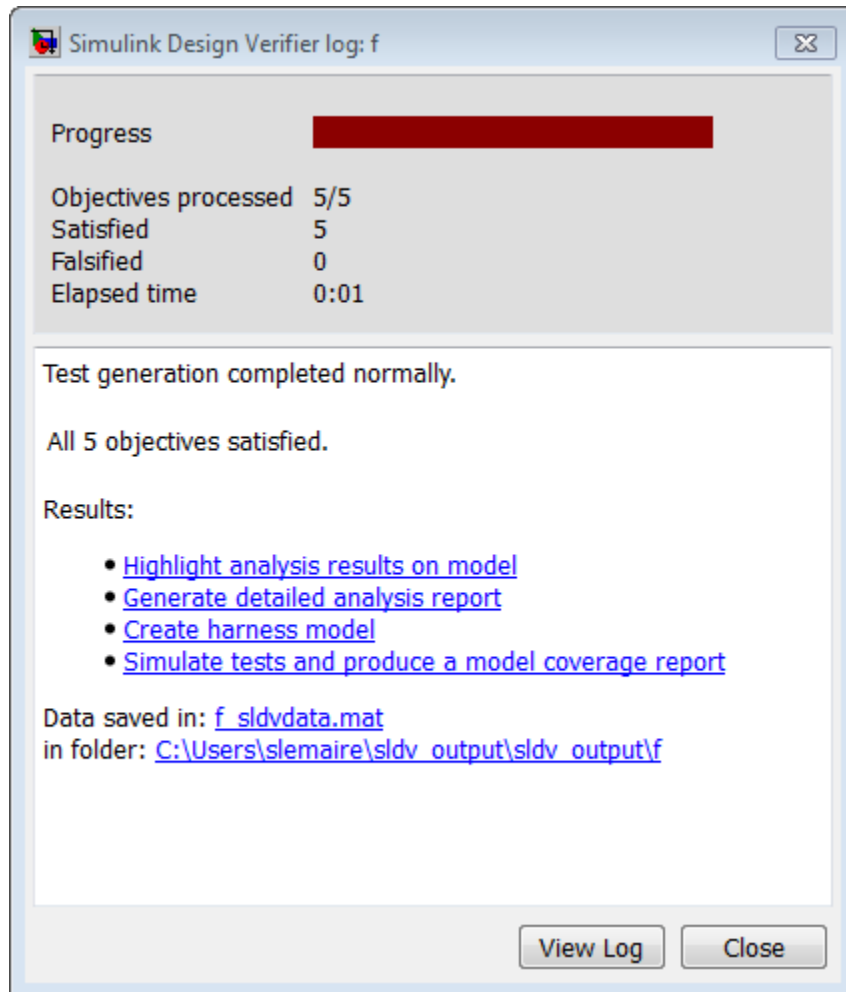
- 1 Open the `sl_subsys_fcncall2` demo model:

```
sl_subsys_fcncall2
```

This model contains a Stateflow chart named Chart that triggers the function-call subsystem `f`.

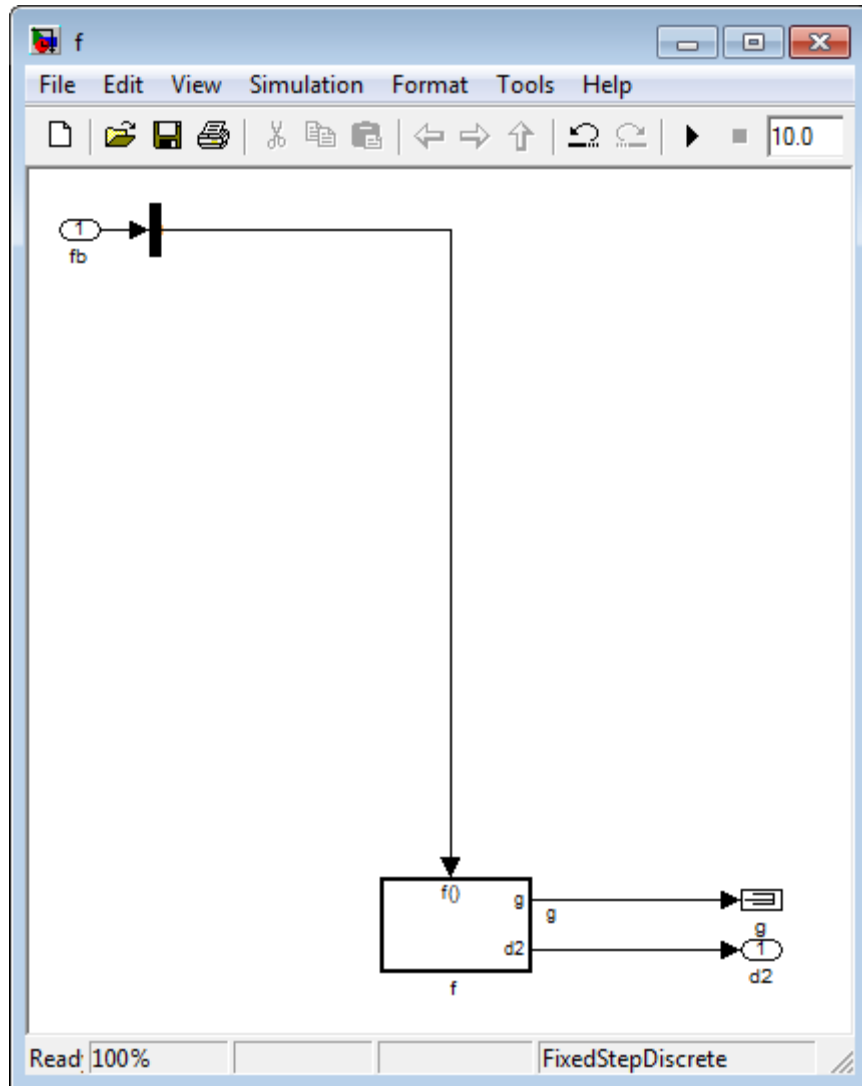
- 2 Right-click the `f` subsystem and select **Design Verifier > Generate Tests for Subsystem**.

The software extracts the subsystem into a new model named `f`, analyzes the extracted model, and produces results.



**3** Open the `f` model that the software created in `matlabroot\sldv_output\f`.

The Inport block and the new subsystem block mimic the trigger for the function-call subsystem `f` in the new `f` model.



## Analyzing Logical Operations

If you have a Simulink model with both logical and arithmetic operations, consider analyzing only the logical operations.

The Simulink Design Verifier software does not support nonlinear arithmetic of floating-point numbers, as occurs with multiplication or division, unless one of the multiply operands or the divisor is a constant.

To simplify models that contain integers or floating-point numbers, the software maps the model computations into expressions of Boolean variables. For example, the software might represent an eight-bit number as a set of eight Boolean values, with one for each digit. It might represent a bit-wise OR operation of two eight-bit integers as eight separate logical OR operations.

Mapping problems of one data type into Boolean variables is complex, and this complexity increases when the software performs such mapping. The software handles models with predominantly logical signals more efficiently than it does those with large integer or floating-point signals.

---

**Note** Simulink Design Verifier software can handle floating-point inputs when their values impact the design through linear inequalities such as  $x < y$  or  $a > 0$ .

In addition, input complexity can result from certain cast operations. For example, casting a `double` to an `int8` can introduce a nonlinearity in certain situations.

---

## Handling Models with Large State Spaces

Persistent design variables (variables that are assigned in one time step and used in a later time step during simulation) affect the complexity of analysis in much the same way as input complexity. You can use one or more of the following techniques to simplify the complexity of the state space you want to search:

- Apply constraints to input signals that are delayed.
- Constraint 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 “Handling Problems with Counters and Timers” on page 14-25.) A test case that you generate at a lower sample rate often has similarities to the test case with a high sample rate that you need to achieve an objective.

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 any existing simulation data to help satisfy your testing needs. If you have existing test data, run it on your model and collect model coverage. Using the `sldvgen cov` function, you can ignore model coverage objectives that have already been satisfied in simulation when you supply a coverage data object.

---

**Note** For more information on satisfying missing model coverage, see the *Extending an Existing Test Suite* demo.

---



## Handling Problems with Counters and Timers

Complexity from states occurs from both the size of the state representation and the number of time steps required to transition from one state to another. The Simulink Design Verifier analysis searches through sequences of time steps, starting from the default configuration, to find input values that reach a state that satisfies an objective.

---

**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 time step before considering any of the configurations that can be reached in two time steps. Likewise, the search investigates all configurations that can be reached in two time steps before it considers any configuration that requires three or more time steps, etc.

Models that contain time delays, such as countdown timers, hinder the analysis by forcing the search to span large numbers of time steps. By design, the value of a counter can reach  $n$  only when its previous value is  $n - 1$ .

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 time steps, resulting in longer test cases that are more difficult to identify.

Some basic techniques you can use to improve performance in models that have delays include the following:

- Make time delays tunable parameters. Choose very small values when running an analysis. 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.
- Choose higher frequency cutoffs for filters and fewer samples to average to minimize filtering delays.

- Make the initial values of counters and timers parameter values that the Simulink Design Verifier software can modify. The software finds initial values that allow shorter test cases to exceed thresholds.

## Techniques for Proving Properties of 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 the Simulink Design Verifier software 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...
“Finding Property Violations While Designing Your Model” on page 14-27
“Combining Proving Properties and Finding Proof Violations” on page 14-28

### Finding Property Violations While Designing Your Model

Simulink Design Verifier software offers a strategy that quickly identifies property violations in larger, more complicated models. While designing your model, analyze your model using this strategy so that you can fix any property violations before finalizing your design.

To identify property violations of a model, on the **Design Verifier > Property Proving** pane of the Configuration Parameters dialog box, specify the value of the **Strategy** parameter as `FindViolation`. When you use this strategy, the **Maximum violation steps** parameter becomes active so that you can specify an upper bound for the number of time steps in the search.

When analyze the model, the software searches only for property violations within the specified number of time steps. By identifying and fixing the

property violations first, you improve the performance of a property-proving analysis that uses the **Prove** strategy.

If a violation is not detected, it is impossible to violate the property with any input sequence having fewer time steps than the specified limit. However, you cannot prove that the property is true because there might be a counterexample within more time steps than the specified limit.

### Combining 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

---

- “Overview of Simulink® Design Verifier™ Configuration Parameters” on page 15-2
- “Design Verifier Pane” on page 15-3
- “Design Verifier Pane: Block Replacements” on page 15-13
- “Design Verifier Pane: Parameters” on page 15-18
- “Design Verifier Pane: Test Generation” on page 15-21
- “Design Verifier Pane: Design Error Detection” on page 15-35
- “Design Verifier Pane: Property Proving” on page 15-40
- “Design Verifier Pane: Results” on page 15-46
- “Design Verifier Pane: Report” on page 15-61
- “Parameter Command-Line Information Summary” on page 15-67

## Overview of Simulink Design Verifier Configuration Parameters

The Simulink Design Verifier software provides numerous options in the Configuration Parameters dialog box that control its behavior when analyzing models. To view the options related to Simulink Design Verifier, in the model window, select **Tools > Design Verifier > Options**.

The Configuration Parameters dialog box opens; the **Design Verifier** panes are listed in the **Select** pane on the left-hand side.

Typically, you specify values for these options using the Configuration Parameters dialog box. See “Configuration Parameters Dialog Box” in *Simulink Graphical User Interface* for more information about working with this interface.

---

**Note** By default, Simulink Design Verifier options do not appear in a model’s Configuration Parameters dialog box. In the model window, if you select **Tools > Design Verifier > Options**, the Simulink Design Verifier software initially associates its default options with that model. After you save the model, you access the Simulink Design Verifier options directly from the Configuration Parameters dialog box or from the Model Explorer.

---

Alternatively, you can use the `sldvoptions` function to view Simulink Design Verifier options at the command line. Use the following syntax to view programmatically the options associated with a Simulink model:

```
opts = sldvoptions('model_name');  
get(opts)
```



## Design Verifier Pane

Analysis options

Mode:

Maximum analysis time (s):

Display unsatisfiable test objectives

Automatic stubbing of unsupported blocks and functions

Use specified input minimum and maximum values

Output

Output directory:

Make output file names unique by adding a suffix

### In this section...

“Design Verifier Pane Overview” on page 15-4

“Mode” on page 15-4

“Maximum analysis time” on page 15-6

“Display unsatisfiable test objectives” on page 15-7

“Automatic stubbing of unsupported blocks and functions” on page 15-8

“Use specified input minimum and maximum values” on page 15-9

“Output directory” on page 15-10

“Make output file names unique by adding a suffix” on page 15-12

## Design Verifier Pane Overview

Specify analysis options and configure Simulink Design Verifier output.

### Mode

Specify the analysis mode for the Simulink Design Verifier software.

### Settings

**Default:** Test generation

Design error detection

Detects integer and fixed-point overflow errors and division-by-zero errors in a model

Test generation

Generates test cases for a model.

Property proving

Proves properties of a model.

### Tip

The Simulink Design Verifier software specifies the value of this option automatically when you select one of the following menu options:

- **Tools > Design Verifier > Generate Tests**
- **Tools > Design Verifier > Detect Design Errors**
- **Tools > Design Verifier > Prove Properties**

### Dependency

Selecting **Test generation** enables the **Display unsatisfiable test objectives** parameter.

When you set the **Mode** parameter, the button below **Check Model Compatibility** changes as follows:

- **Mode:** Test generation, button reads: **Generate Tests**
- **Mode:** Design error detection, button reads: **Detect Errors**

- **Mode:** Property proving, button reads: **Prove Properties**

### **Command-Line Information**

**Parameter:** DVMode

**Type:** string

**Value:** 'TestGeneration' | 'ErrorDetection' | 'PropertyProving'

**Default:** 'TestGeneration'

### **See Also**

- Detecting Design Errors
- Generating Test Cases
- Proving Properties of a Model

### **Maximum analysis time**

Specify the maximum time (in seconds) that the Simulink Design Verifier software spends analyzing a model.

#### **Settings**

**Default:** 300

The value that you enter represents the maximum number of seconds the Simulink Design Verifier software 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 the Simulink Design Verifier software is unable to satisfy a test objective.



Off

Does not display a warning when the Simulink Design Verifier software is unable to satisfy a test objective.

---

**Tip** If you first select **Display unsatisfiable test objectives**, set the **Test suite optimization** option to the **Combined objectives** strategy and analyze the model. If that test returns objectives without outcomes, then select the **Individual objectives** strategy and reanalyze the model. The **Individual objectives** strategy analyzes each objective independently and identifies unsatisfiable objectives.

---

### Command-Line Information

**Parameter:** `DVDisplayUnsatisfiableObjectives`

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

## Automatic stubbing of unsupported blocks and functions

Specify whether or not Simulink Design Verifier software should ignore unsupported blocks and functions and proceed with the analysis.

### Settings

**Default:** On



On

Ignores unsupported blocks and functions and proceeds with the analysis.



Off

Displays a warning when the Simulink Design Verifier software encounters an unsupported block or function and asks if you want to continue the analysis.

### Command-Line Information

**Parameter:** DVAutomaticStubbing

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

Handling Incompatibilities with Automatic Stubbing

## Use specified input minimum and maximum values

Specify whether or not Simulink Design Verifier software should 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:** string

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

Considering Specified Minimum and Maximum Values for Inputs During Analysis

## Output directory

Specify a path name to which the Simulink Design Verifier software 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**
  - **SystemTest 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:** string

**Value:** any valid path

**Default:** 'sldv\_output/\$ModelName\$'



## **See Also**

Reviewing the Results

### Make output file names unique by adding a suffix

Specify whether the Simulink Design Verifier software 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:** string

**Value:** 'on' | 'off'

**Default:** 'on'

#### See Also

Reviewing the Results

## 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:

**In this section...**

“Block Replacements Pane Overview” on page 15-14

“Apply block replacements” on page 15-15

“List of block replacement rules” on page 15-16

“File path of the output model” on page 15-17

### **Block Replacements Pane Overview**

Specify options that control how the Simulink Design Verifier software preprocesses the models it analyzes.

#### **See Also**

Working with Block Replacements

## Apply block replacements

Specify whether the Simulink Design Verifier software replaces blocks in a model before its analysis.

### Settings

**Default:** Off

On

Replaces blocks in a model before the Simulink Design Verifier software analyzes it.

Off

Does not replace blocks in a model before the Simulink Design Verifier software analyzes it.

### Dependencies

This parameter enables **List of block replacement rules** and **File path of the output model**.

### Command-Line Information

**Parameter:** DVBlockReplacement

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

Working with Block Replacements

### List of block replacement rules

Specify a list of block replacement rules that the Simulink Design Verifier software 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, the Simulink Design Verifier software uses its factory default block replacement rules.

#### Dependency

This parameter is enabled when you select **Apply block replacements**.

#### Command-Line Information

**Parameter:** DVBlockReplacementRulesList

**Type:** string

**Value:** any rules

**Default:** '<FactoryDefaultRules>'

#### See Also

Working with Block Replacements

## 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 directory**.
- 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:** string

**Value:** any valid path and file name

**Default:** '\$ModelName\$\_replacement'

### See Also

Working with Block Replacements

## Design Verifier Pane: Parameters

Parameters

Apply parameters

Parameter configuration file:

### In this section...

“Parameters Pane Overview” on page 15-19

“Apply parameters” on page 15-19

“Parameter configuration file” on page 15-19



## Parameters Pane Overview

Specify options that control how the Simulink Design Verifier software uses parameter configurations when analyzing models.

## Apply parameters

Specify whether the Simulink Design Verifier software uses parameter configurations when analyzing a model.

### Settings

**Default:** Off



On

The Simulink Design Verifier software uses 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:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

Specifying Parameter Configurations

## 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 **Apply parameters**.

### Command-Line Information

**Parameter:** `DVParametersConfigFileName`

**Type:** string

**Value:** any valid MATLAB file

**Default:** `'sldv_params_template.m'`

### See Also

Specifying Parameter Configurations

## Design Verifier Pane: Test Generation

Test generation

Model coverage objectives:

Test conditions:

Test objectives:

Maximum test case steps:

Test suite optimization:

Existing test cases

Extend existing test cases:

Data file:

Ignore objectives satisfied by existing test cases

Existing coverage data

Ignore objectives satisfied in existing coverage data:

Coverage data file:

Coverage objective filter

Ignore objectives based on filter:

Coverage filter file:

### In this section...

“Test Generation Pane Overview” on page 15-23

“Model coverage objectives” on page 15-24

“Test conditions” on page 15-25

“Test objectives” on page 15-26

“Maximum test case steps” on page 15-27

“Test suite optimization” on page 15-28

“Extend existing test cases” on page 15-29

“Data file” on page 15-30

“Ignore objectives satisfied by existing test cases” on page 15-31

<b>In this section...</b>
“Ignore objectives satisfied in existing coverage data” on page 15-31
“Coverage data file” on page 15-32
“Ignore objectives based on filter” on page 15-33
“Coverage filter file” on page 15-34

## **Test Generation Pane Overview**

Specify options that control how the Simulink Design Verifier software generates tests for the models it analyzes.

### **See Also**

Generating Test Cases

## Model coverage objectives

Specify the type of model coverage that the Simulink Design Verifier software 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.

Condition Decision

Generates test cases that achieve condition and decision coverage.

MCDC

Generates test cases that achieve modified condition/decision coverage (MCDC).

When you set **Model coverage objectives** to MCDC, the Simulink Design Verifier software automatically enables every coverage objective for decision coverage and condition coverage as well. Similarly, enabling coverage for condition coverage causes every decision and condition coverage outcome to be enabled.

### Command-Line Information

**Parameter:** DVModelCoverageObjectives

**Type:** string

**Value:** 'None' | 'Decision' | 'ConditionDecision' | 'MCDC'

**Default:** 'ConditionDecision'

### See Also

Generating Test Cases

## 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:** string

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Test Condition
- Generating Test Cases

## 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:** string

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Test Objective
- Generating Test Cases



## Maximum test case steps

Specify the maximum number of simulation steps the Simulink Design Verifier software 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:** 500

You can specify a value that represents the maximum number of simulation steps the Simulink Design Verifier software takes when attempting to satisfy a test objective.

### Command-Line Information

**Parameter:** DVMaxTestCaseSteps

**Type:** int32

**Value:** any valid value

**Default:** 500

### See Also

Generating Test Cases

## Test suite optimization

Specify the optimization strategy to use when generating test cases.

### Settings

**Default:** CombinedObjectives (Nonlinear Extended)

CombinedObjectives (Nonlinear Extended)

Analyzes the model using a variation of the CombinedObjectives optimization. This optimization includes improved support for nonlinear arithmetic.

LargeModel (Nonlinear Extended)

Analyzes the model using a variation of the LargeModel optimization. This optimization includes improved support for nonlinear arithmetic.

IndividualObjectives

Maximizes the number of test cases in a suite by generating cases that each address only one test objective. Each test case tends to be short, i.e., it includes only a few time steps.

LongTestcases

Combines test cases to create a smaller number of test cases. This strategy generates fewer, but longer, test cases that each satisfy multiple test objectives and creates a more efficient analysis and easier-to-review results.

CombinedObjectives

Minimizes the number of test cases in a suite by generating cases that address more than one test objective. Each test case tends to be long, i.e., it includes many time steps.

LargeModel

Minimizes the number of test cases in a suite by generating cases that address more than one test objective. This strategy is tailored for large, complex models; consequently, it tends to use all the time that the **Maximum analysis time** option allots.

### Tip

If an analysis using the CombinedObjectives or CombinedObjectives (Nonlinear Extended) strategy returns unsatisfiable objectives, set this option to IndividualObjectives and reanalyze the model. The

`IndividualObjectives` strategy analyzes each objective independently and is better at identifying unsatisfiable objectives.

However, set this option to `LargeModel` or `LargeModel (Nonlinear Extended)` if the model has both of the following characteristics:

- Nonlinearities, such as those that result from multiplying or dividing the model's input signals
- Numerous test objectives, such as those that result when using blocks that receive model coverage

The `LargeModel` and `LargeModel (Nonlinear Extended)` strategies perform an analysis that is tailored to large, complex models. However, these strategies tend to use all the time that the **Maximum analysis time** option allots.

If you have a large number of test objectives, select `LongTestCases` for a more efficient analysis and an easy-to-review report.

## Command-Line Information

**Parameter:** `DVTestSuiteOptimization`

**Type:** string

**Value:** `'CombinedObjectives (Nonlinear Extended)'` | `'LargeModel (Nonlinear Extended)'` | `'IndividualObjectives'` | `'LongTestCases'` | `'CombinedObjectives'` | `'LargeModel'` |

**Default:** `'CombinedObjectives (Nonlinear Extended)'`

## See Also

Generating Test Cases

## 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:** string

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

Extending Existing Test Cases

## 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:** string

**Value:** any valid path and file name

**Default:** ''

## See Also

Simulink Design Verifier Data Files

## 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:** string

**Value:** 'on' | 'off'

**Default:** 'on'

## See Also

- Achieving Test Cases for Missing Model Coverage
- Simulink Design Verifier Data Files

## 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:** string

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

- Achieving Test Cases for Missing Model Coverage
- Simulink Design Verifier Data Files

## Coverage data file

Specify a folder and file name for the file that contains data about any 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:** string  
**Value:** any valid path and file name  
**Default:** ''

### See Also

Achieving Test Cases for Missing Model Coverage

### Ignore objectives based on filter

Specify to analyze the model, ignoring the coverage objectives in the **Coverage filter file**.

### Settings

**Default:** Off



On

Ignores coverage objectives in the **Coverage filter file** during the analysis.



Off

Generates results for all coverage objectives, including those in **Coverage filter file**.

### Dependency

This parameter enables **Coverage filter file**.

### Command-Line Information

**Parameter:** DVCovFilter  
**Type:** string  
**Value:** 'on' | 'off'  
**Default:** 'off'

### See Also

- Achieving Test Cases for Missing Model Coverage
- “Excluding Model Objects From Coverage” in the Simulink Verification and Validation documentation

### Coverage filter file

Specify a folder and file name for the file that contains the coverage objectives you want to ignore. The **Coverage filter file** specifies model objects to exclude from model coverage during test case generation.

### Settings

**Default:** ''

- Specify the name of the folder and file name that contains the coverage objectives you want to ignore.

Click the **Browse** button to select an existing MATLAB file.

### Command-Line Information

**Parameter:** DVCovFilterFileName

**Type:** string

**Value:** any valid path and file name

**Default:** ''

### See Also

- Achieving Test Cases for Missing Model Coverage
- “Excluding Model Objects From Coverage” in the Simulink Verification and Validation documentation



## Design Verifier Pane: Design Error Detection

Design Error Detection

- Dead Logic
- Integer overflow
- Division by zero
- Check specified intermediate minimum and maximum values

### In this section...

“Design Error Detection Pane Overview” on page 15-36

“Dead Logic” on page 15-36

“Integer overflow” on page 15-37

“Division by zero” on page 15-37

“Check specified intermediate minimum and maximum values” on page 15-38

## Design Error Detection Pane Overview

Specify options that control how the Simulink Design Verifier software detects runtime errors in the models it analyzes.

### Dead Logic

Specify whether to analyze your model for dead logic.

#### Settings

**Default:** Off

On

Reports dead logic in your model.

Off

Does not report dead logic in your model.

### Dependency

This parameter disables:

- **Integer overflow**
- **Division by zero**
- **Check specified intermediate minimum and maximum values**

### Command-Line Information

**Parameter:** DVDetectDeadLogic

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

Detecting Design Errors

## 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 overflow errors in your model.

### Dependency

This parameter disables **Dead Logic**.

### Command-Line Information

**Parameter:** DVDetectIntegerOverflow

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

Detecting Design Errors

## Division by zero

Specify whether to analyze your model for division-by-zero errors.

### Settings

**Default:** On



On

Reports division-by-zero errors in your model.

Off

Does not report division-by-zero errors in your model.

### Dependency

This parameter disables **Dead Logic**.

### Command-Line Information

**Parameter:** DVDetectDivisionByZero

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

Detecting Design Errors

## Check specified intermediate minimum and maximum values

Specify whether to check that the intermediate and output signals in your model are within the range of user-specified minimum and maximum constraints.

### Settings

**Default:** Off

On

Check that intermediate and output signals are within the range of user-specified minimum and maximum constraints.

Off

Do not check that intermediate and output signals are within the range of user-specified minimum and maximum constraints.

### Dependency

This parameter disables **Dead Logic**.

## **Command-Line Information**

**Parameter:** DVDesignMinMaxCheck

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

## **See Also**

Detecting Design Errors

## Design Verifier Pane: Property Proving

Property proving

Assertion blocks:

Proof assumptions:

Strategy:

Maximum violation steps:

### In this section...

“Property Proving Pane Overview” on page 15-41

“Assertion blocks” on page 15-42

“Proof assumptions” on page 15-43

“Strategy” on page 15-44

“Maximum violation steps” on page 15-45

## **Property Proving Pane Overview**

Specify options that control how the Simulink Design Verifier software proves properties for the models it analyzes.

### **See Also**

[Proving Properties of a Model](#)

## 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:** string

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Assertion
- Proving Properties of a Model



## 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:** string

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Proof Assumption
- Proving Properties of a Model

## Strategy

Specify the strategy that the Simulink Design Verifier software 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:** string

**Value:** 'Prove' | 'FindViolation' | 'ProveWithViolationDetection'

**Default:** 'Prove'

## See Also

Proving Properties of a Model

## Maximum violation steps

Specify the maximum number of simulation steps over which the Simulink Design Verifier software 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

Proving Properties of a Model

## Design Verifier Pane: Results

Data file options

Save test data to file

Data file name:

Include expected output values

Randomize data that do not affect the outcome

Display results on model

Display results of the analysis on the model

Harness model options

Save test harness as model

Harness model file name:

Reference input model in generated harness

SystemTest options

Save test harness as SystemTest TEST-file (will reference saved data file)

SystemTest file name:

### In this section...

“Results Pane Overview” on page 15-48

“Save test data to file” on page 15-49

“Data file name” on page 15-50

“Include expected output values” on page 15-51

“Randomize data that does not affect outcome” on page 15-52

“Display results of the analysis on the model” on page 15-53

“Save test harness as model” on page 15-55

“Harness model file name” on page 15-56

**In this section...**

“Reference input model in generated harness” on page 15-57

“Save test harness as SystemTest TEST-file (will reference saved data file)”  
on page 15-59

“SystemTest file name” on page 15-60

### **Results Pane Overview**

Specify options that control how the Simulink Design Verifier software handles the results that it generates.

### **See Also**

Reviewing the Results

## 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:** string

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

Simulink Design Verifier Data Files

### 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 directory**.
- 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:** string

**Value:** any valid path and file name

**Default:** `'$modelName$_sldvdata'`

### See Also

Simulink Design Verifier Data Files



## 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 “Overview of the `sldvData` Structure” on page 13-7.
- When **Include expected output values** is enabled, the Simulink Design Verifier software successively simulates the model using each test case that it generates. Enabling this option requires more time for the Simulink Design Verifier software to complete its analysis.

### Dependency

This parameter is enabled by **Save test data to file**.

### Command-Line Information

**Parameter:** DVSaveExpectedOutput

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

Simulink Design Verifier Data Files

## Randomize data that does not affect outcome

Use random values instead of zeros for input signals that have no impact on test or proof objectives.

### Settings

Default: Off

On

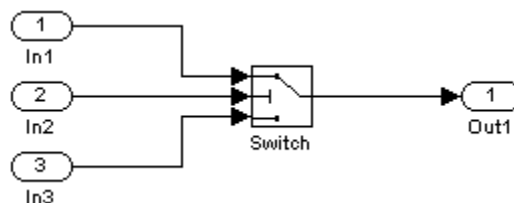
Assigns random values to test case or counterexample signals that do not affect the outcome of test or proof objectives in a model. This option can enhance traceability and improve your regression tests.

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 assigns random values to test case or counterexample signals that otherwise would be zero. In the Simulink Design Verifier report, the Generated Input Data table always displays a dash (–) for such signals.
- 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's control port impacts its decision coverage. If the **Randomize data that does not affect outcome** parameter is off, the Simulink Design Verifier software 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:** string

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

Simulink Design Verifier Data Files

## Display results of the analysis on the model

Display analysis results by highlighting the model and providing context-sensitive details about the results.

## Settings

**Default:** Off



On

Highlight the model with the analysis results and provide context-sensitive details about the results.



Off

Do not display analysis results on the model.

## Command-Line Information

**Parameter:** DVDisplayResultsOnModel

**Type:** string  
**Value:** 'on' | 'off'  
**Default:** 'off'

### **See Also**

Highlighted Results on Model

## Save test harness as model

Create a harness model generated by the Simulink Design Verifier analysis.

### Settings

**Default:** Off

- On  
Saves the harness model that the Simulink Design Verifier software generates as a model file.
- Off  
Does not save the harness model that the Simulink Design Verifier software generates.

### Dependency

This parameter enables **Harness model file name**.

### Command-Line Information

**Parameter:** DVSaveHarnessModel

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

Harness Model

## 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 directory**.
- Enter a file name for the harness model.
- \$modelName\$ is a token that represents the model name.

### Dependency

This parameter is enabled by **Save test harness as model**.

### Command-Line Information

**Parameter:** DVHarnessModelFileName

**Type:** string

**Value:** any valid path and file name

**Default:** '\$modelName\$\_harness'

### See Also

Harness Model

## 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 may impact your coverage results.

---

**Note** The simulation optimization parameters are on the following Configuration Parameters dialog box panes:

- **Optimization** pane
  - **Optimization > Signals and Parameters** pane
  - **Optimization > Stateflow** pane
- 

- If your model contains bus objects and you select **Reference input model in generated harness**, in the **Configuration Parameters > Diagnostics > Connectivity** pane, you must set the **Mux blocks used to create bus signals** parameter to error.
- On the **Design Verifier > Parameters** pane, if you select the **Apply parameters** parameter, the Simulink Design Verifier software always 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:** string

**Value:** 'on' | 'off'

**Default:** 'off'

## **See Also**

Harness Model



## Save test harness as SystemTest TEST-file (will reference saved data file)

Save the analysis results as a SystemTest TEST-file so you can run test cases using the SystemTest capabilities.

---

**Note** The option to create a SystemTest TEST-file is only available in test-generation mode; you cannot create this file when running a property-proving analysis.

---

### Settings

**Default:** Off

On

Saves the analysis results as a SystemTest TEST-file.

Off

Does not save the analysis results as a SystemTest TEST-file.

### Dependency

This parameter enables **SystemTest file name**.

### Command-Line Information

**Parameter:** DVSaveSystemTestHarness

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

SystemTest TEST-Files

## SystemTest file name

Specify a folder and file name for the SystemTest TEST-file.

### Settings

**Default:** \$modelName\$\_harness

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output directory**.
- Enter a file name for the SystemTest TEST-file.
- \$modelName\$ is a token that represents the model name.

### Dependency

This parameter is enabled by **Save test harness as SystemTest TEST-file (will reference saved data file)**.

### Command-Line Information

**Parameter:** DVMSystemTestFileName

**Type:** string

**Value:** any valid path and file name

**Default:** '\$modelName\$\_harness'

### See Also

SystemTest TEST-Files

## Design Verifier Pane: Report

Report

Generate report of the results

Report file name:

Include screen shots of properties

Display report

### In this section...

“Report Pane Overview” on page 15-62

“Generate report of the results” on page 15-63

“Report file name” on page 15-64

“Include screen shots of properties” on page 15-65

“Display report” on page 15-66

### **Report Pane Overview**

Specify options that control how the Simulink Design Verifier software reports its results.

### **See Also**

Simulink Design Verifier Reports

## Generate report of the results

Generate and save a Simulink Design Verifier report.

### Settings

**Default:** Off

- On  
Saves the HTML report that the Simulink Design Verifier software generates.
- Off  
Does not generate a Simulink Design Verifier report.

### Dependencies

When this parameter is enabled, you must enable **Save test harness as model**.

This parameter enables the following parameters:

- **Report file name**
- **Include screen shots of properties**
- **Display report**

### Command-Line Information

**Parameter:** DVSaveReport

**Type:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

Simulink Design Verifier Reports

### 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 directory**.
- 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:** string

**Value:** any valid path and file name

**Default:** '\$ModelName\$\_report'

### See Also

Simulink Design Verifier Reports

## 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:** string

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

Simulink Design Verifier Reports

### 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:** string

**Value:** 'on' | 'off'

**Default:** 'on'

#### See Also

Simulink Design Verifier Reports



## Parameter Command-Line Information Summary

The following table lists parameters that you can use to configure the behavior of the Simulink Design Verifier software. Use the `get_param` and `set_param` functions to retrieve and specify values for these parameters programmatically.

For each parameter listed in the table, the **Description** column indicates where you can set its value on 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	Description	Values
DVAssertions	Set by the <b>Assertion blocks</b> parameter on the <b>Design Verifier &gt; Property Proving</b> pane.	'EnableAll'   'DisableAll'   {'UseLocalSettings'}
DVAutomaticStubbing	Set by the <b>Automatic stubbing of unsupported blocks and functions</b> parameter on the <b>Design Verifier</b> pane.	{'on'}   'off'
DVBlockReplacement	Set by the <b>Apply block replacements</b> parameter on the <b>Design Verifier &gt; Block Replacements</b> pane.	'on'   {'off'}
DVBlockReplacement-ModelFileName	Set by the <b>File path of the output model</b> parameter on the <b>Design Verifier &gt; Block Replacements</b> pane.	string {'\$modelName\$_replacement'}
DVBlockReplacement-RulesList	Set by the <b>List of block replacement rules</b> parameter on the <b>Design Verifier &gt; Block Replacements</b> pane.	string {'<FactoryDefaultRules>'}

<b>Parameter</b>	<b>Description</b>	<b>Values</b>
DVCoverageDataFile	Set by the <b>Coverage data file</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	string { ' ' }
DVCovFilter	Set by the <b>Ignore objectives based on filter</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	'on'   {'off'}
DVCovFilterFileName	Set by the <b>Coverage filter file</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	string { ' ' }
DVDataFileName	Set by the <b>Data file name</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	string { '\$modelName\$_sldvdata' }
DVDesignMinMax-Constraints	Set by the <b>Use specified input minimum and maximum values</b> parameter on the <b>Design Verifier</b> pane.	{ 'on' }   'off'
DVDetectIntegerOverflow	Set by the <b>Integer overflow</b> parameter on the <b>Design Verifier &gt; Design Error Detection</b> pane.	{ 'on' }   'off'
DVDetectDivisionByZero	Set by the <b>Division by zero</b> parameter on the <b>Design Verifier &gt; Design Error Detection</b> pane.	{ 'on' }   'off'
DVDisplayReport	Set by the <b>Display report</b> parameter on the <b>Design Verifier &gt; Report</b> pane.	{ 'on' }   'off'

Parameter	Description	Values
DVDisplayResultsOnModel	Set by the <b>Display results of the analysis on the model</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	'on'   {'off'}
DVDisplayUnsatisfiable-Objectives	Set by the <b>Display unsatisfiable test objectives</b> parameter on the <b>Design Verifier</b> pane.	'on'   {'off'}
DVExtendExistingTests	Set by the <b>Extend existing test cases</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	'on'   {'off'}
DVExistingTestFile	Set by the <b>Data file</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	string {''}
DVHarnessModelFileName	Set by the <b>Harness model file name</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	string { '\$modelName\$_harness' }
DVIgnoreCovSatisfied	Set by the <b>Ignore objectives satisfied in existing coverage data</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	'on'   {'off'}
DVIgnoreExistTest-Satisfied	Set by the <b>Ignore objectives satisfied by existing test cases</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	{'on'}   'off'
DVMakeOutputFilesUnique	Set by the <b>Make output file names unique by adding a suffix</b> check box on the <b>Design Verifier</b> pane.	{'on'}   'off'

<b>Parameter</b>	<b>Description</b>	<b>Values</b>
DVMaxProcessTime	Set by the <b>Maximum analysis time</b> parameter on the <b>Design Verifier</b> pane.	double {'300'}
DVMaxTestCaseSteps	Set by the <b>Maximum test case steps</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	int32 {'500'}
DVMaxViolationSteps	Set by the <b>Maximum violation steps</b> parameter on the <b>Design Verifier &gt; Property Proving</b> pane.	int32 {'20'}
DVMode	Set by the <b>Mode</b> parameter on the <b>Design Verifier</b> pane.	{'TestGeneration'}   'ErrorDetection'   'PropertyProving'
DVModelCoverage-Objectives	Set by the <b>Model coverage objectives</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	'None'   'Decision'   {'ConditionDecision'}   'MCDC'
DVModelReferenceHarness	Set by the <b>Reference input model in generated harness</b> parameter on the <b>Design Verifier &gt; Results</b> pane of the Configuration Parameters dialog box.	'on'   {'off'})
DVOutputDir	Set by the <b>Output directory</b> parameter on the <b>Design Verifier</b> pane.	string {'sldv_output/\$modelName\$'}
DVParameters	Set by the <b>Apply parameters</b> parameter on the <b>Design Verifier &gt; Parameters</b> pane.	'on'   {'off'}

Parameter	Description	Values
DVParametersConfigFileName	Set by the <b>Parameter configuration file</b> parameter on the <b>Design Verifier &gt; Parameters</b> pane.	string {'sldv_params_template.m'}
DVProofAssumptions	Set by the <b>Proof assumptions</b> parameter on the <b>Design Verifier &gt; Property Proving</b> pane.	'EnableAll'   'DisableAll'   {'UseLocalSettings'}
DVProvingStrategy	Set by the <b>Strategy</b> parameter on the <b>Design Verifier &gt; Property Proving</b> pane.	'FindViolation'   {'Prove'}   'ProveWithViolationDetection'
DVRandomizeNoEffectData	Set by the <b>Randomize data that does not affect outcome</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	'on'   {'off'}
DVReportFileName	Set by the <b>Report file name</b> parameter on the <b>Design Verifier &gt; Report</b> pane.	string {'\$ModelName\$_report'}
DVReportIncludeGraphics	Set by the <b>Include screen shots of properties</b> parameter on the <b>Design Verifier &gt; Report</b> pane.	'on'   {'off'}
DVSaveDataFile	Set by the <b>Save test data to file</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	{'on'}   'off'
DVSaveExpectedOutput	Set by the <b>Include expected output values</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	'on'   {'off'}

<b>Parameter</b>	<b>Description</b>	<b>Values</b>
DVSaveHarnessModel	Set by the <b>Save test harness as model</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	'on'   {off'}
DVSaveReport	Set by the <b>Generate report of the results</b> parameter on the <b>Design Verifier &gt; Report</b> pane.	'on'   {off'}
DVSaveSystemTestHarness	Set by the <b>Save text harness as SystemTest TEST-file (will reference saved data file)</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	'on'   {off'}
DVSystemTestFileName	Set by the <b>SystemTest file name</b> parameter on the <b>Design Verifier &gt; Results</b> pane.	string {'\$modelName\$_harness'}
DVTestConditions	Set by the <b>Test conditions</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	'EnableAll'   'DisableAll'   {'UseLocalSettings'}
DVTestObjectives	Set by the <b>Test objectives</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	'EnableAll'   'DisableAll'   {'UseLocalSettings'}
DVTestSuiteOptimization	Set by the <b>Test suite optimization</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.	{'CombinedObjectives'}   'IndividualObjectives'   'LargeModel'   'LongTestCases'   'CombinedObjectives (Nonlinear Extended)'   'LargeModel (Nonlinear Extended)'

# Simulink Block Support

---

- “Overview of Simulink Block Support” on page 16-2
- “Additional Math and Discrete Library” on page 16-3
- “Commonly Used Blocks Library” on page 16-4
- “Continuous Library” on page 16-5
- “Discontinuities Library” on page 16-6
- “Discrete Library” on page 16-7
- “Logic and Bit Operations Library” on page 16-8
- “Lookup Tables Library” on page 16-9
- “Math Operations Library” on page 16-10
- “Model Verification Library” on page 16-12
- “Model-Wide Utilities Library” on page 16-13
- “Ports & Subsystems Library” on page 16-14
- “Signal Attributes Library” on page 16-16
- “Signal Routing Library” on page 16-17
- “Sinks Library” on page 16-18
- “Sources Library” on page 16-19
- “User-Defined Functions Library” on page 16-20

## Overview of Simulink Block Support

The following tables summarize the analysis support for Simulink blocks. Each table lists all the blocks in each Simulink library and describes support information for that particular block. A dash (—) indicates that the software supports that block under all conditions.

If the software does not support a given block, automatic stubbing, which is enabled by default, considers the interface of the unsupported blocks, but not their behavior, during the analysis. However, if any of the unsupported blocks affect the simulation outcome, the analysis may achieve only partial results.

For details about automatic stubbing, see “Handling Incompatibilities with Automatic Stubbing” on page 2-11.



## **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

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

## **Discontinuities Library**

The software supports all blocks in the Discontinuities library.

## Discrete Library

<b>Block</b>	<b>Support Notes</b>
Delay	—
Difference	—
Discrete Derivative	—
Discrete Filter	—
Discrete FIR Filter	—
Discrete PID Controller	—
Discrete PID Controller (2 DOF)	—
Discrete State-Space	Not supported
Discrete Transfer Fcn	—
Discrete Zero-Pole	Not supported
Discrete-Time Integrator	—
First-Order Hold	—
Memory	—
Tapped Delay	—
Transfer Fcn First Order	—
Transfer Fcn Lead or Lag	—
Transfer Fcn Real Zero	—
Unit Delay	—
Zero-Order Hold	—

## **Logic and Bit Operations Library**

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

## Lookup Tables Library

Block	Support Notes
Cosine	—
Direct Lookup Table (n-D)	—
Interpolation Using Prelookup	<p>Not supported when:</p> <ul style="list-style-type: none"> <li>The <b>Interpolation method</b> parameter is <b>Linear</b> and the <b>Number of table dimensions</b> parameter is greater than 4.</li> </ul> <p>or</p> <ul style="list-style-type: none"> <li>The <b>Interpolation method</b> parameter is <b>Linear</b> and the <b>Number of sub-table selection dimensions</b> parameter is not 0.</li> </ul>
1-D Lookup Table	Not supported when the <b>Interpolation method</b> or the <b>Extrapolation method</b> parameter is <b>Cubic Spline</b> .
2-D Lookup Table	Not supported when the <b>Interpolation method</b> or the <b>Extrapolation method</b> parameter is <b>Cubic Spline</b> .
n-D Lookup Table	<p>Not supported when:</p> <ul style="list-style-type: none"> <li>The <b>Interpolation method</b> or the <b>Extrapolation method</b> parameter is <b>Cubic Spline</b>.</li> </ul> <p>or</p> <ul style="list-style-type: none"> <li>The <b>Interpolation method</b> parameter is <b>Linear</b> and the <b>Number of table dimensions</b> parameter is greater than 5.</li> </ul>
Lookup Table Dynamic	—
Prelookup	—
Sine	—

## Math Operations Library

Block	Support Notes														
Abs	—														
Add	—														
Algebraic Constraint	—														
Assignment	—														
Bias	—														
Complex to Magnitude-Angle	Not supported														
Complex to Real-Imag	Not supported														
Divide	—														
Dot Product	—														
Find Nonzero Elements	—														
Gain	—														
Magnitude-Angle to Complex	Not supported														
Math Function	<p>All signal types support the following <b>Function</b> parameter settings.</p> <table border="1"> <tbody> <tr> <td>conj</td> <td>hermitian</td> <td>magnitude^2</td> <td>mod</td> </tr> <tr> <td>rem</td> <td>reciprocal</td> <td>square</td> <td>transpose</td> </tr> </tbody> </table> <p>The software does not support the following <b>Function</b> parameter settings.</p> <table border="1"> <tbody> <tr> <td>10^u</td> <td>exp</td> <td>hypot</td> </tr> <tr> <td>log</td> <td>log10</td> <td>pow</td> </tr> </tbody> </table>	conj	hermitian	magnitude^2	mod	rem	reciprocal	square	transpose	10^u	exp	hypot	log	log10	pow
conj	hermitian	magnitude^2	mod												
rem	reciprocal	square	transpose												
10^u	exp	hypot													
log	log10	pow													
Matrix Concatenate	—														
MinMax	—														
MinMax Running Resettable	—														
Permute Dimensions	—														



<b>Block</b>	<b>Support Notes</b>
Polynomial	—
Product	—
Product of Elements	—
Real-Imag to Complex	Not supported
Reciprocal Sqrt	Not supported
Reshape	—
Rounding Function	—
Sign	—
Signed Sqrt	Not supported
Sine Wave Function	Not supported
Slider Gain	—
Sqrt	Not supported
Squeeze	—
Subtract	—
Sum	—
Sum of Elements	—
Trigonometric Function	Supported when <b>Function</b> is sin, cos, or sincos and <b>Approximation method</b> is CORDIC.
Unary Minus	—
Vector Concatenate	—
Weighted Sample Time Math	—

## **Model Verification Library**

The software supports all blocks in the Model Verification library.

## Model-Wide Utilities Library

<b>Block</b>	<b>Support Notes</b>
Block Support Table	—
DocBlock	—
Model Info	—
Timed-Based Linearization	Not supported
Trigger-Based Linearization	Not supported

## Ports & Subsystems Library

Block	Support Notes
Atomic Subsystem	—
Code Reuse Subsystem	—
Configurable Subsystem	—
Enable	—
Enabled Subsystem	—
Enabled and Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type.
For Each	Not supported and not stubbable
For Each Subsystem	Not supported
For Iterator Subsystem	—
Function-Call Feedback Latch	—
Function-Call Generator	—
Function-Call Split	—
Function-Call Subsystem	—
If	<p>When running test case generation for coverage, the If block records condition, decision, and MCDC coverage. However, the Simulink Design Verifier software generates test cases only for condition and decision coverage.</p> <p>Parameter configurations are not supported. The analysis ignores any parameter configurations that you specify for an If block.</p>
If Action Subsystem	—
Inport	—
Model	Supported except for the limitations described in “Limitations of Support for Model Blocks” on page 3-13
Model Variants	Supported except for the limitations described in “Limitations of Support for Model Blocks” on page 3-13

<b>Block</b>	<b>Support Notes</b>
Outport	—
Subsystem	—
Switch Case	—
Switch Case Action Subsystem	—
Trigger	—
Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type.
Variant Subsystem	Not supported when the <b>Generate preprocessor conditionals</b> parameter is enabled.
While Iterator Subsystem	—

## **Signal Attributes Library**

The software supports all blocks in the Signal Attributes library.

## Signal Routing Library

Block	Support Notes
Bus Assignment	—
Bus Creator	—
Bus Selector	—
Data Store Memory	—
Data Store Read	—
Data Store Write	—
Demux	—
Environment Controller	—
From	—
Goto	—
Goto Tag Visibility	—
Index Vector	—
Manual Switch	<p>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.</p> <p>Model coverage data is collected for the Manual Switch block.</p>
Merge	—
Multiport Switch	—
Mux	—
Selector	—
Switch	—
Vector Concatenate	—

## Sinks Library

Block	Support Notes
Display	—
Floating Scope	—
Outport (Out1)	—
Scope	—
Stop Simulation	Not supported and not stubbable
Terminator	—
To File	—
To Workspace	—
XY Graph	—



## Sources Library

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

## User-Defined Functions Library

Block	Support Notes
Fcn	<p>Supports all operators except ^, and supports only the mathematical functions abs, ceil, fabs, floor, rem, and sgn.</p> <p>Parameter configurations are not supported. The analysis ignores any parameter configurations that you specify for these blocks.</p>
Interpreted MATLAB Function	Not supported
Level-2 MATLAB S-Function	Not supported
MATLAB Function	For limitations, see “Support Limitations for MATLAB for Code Generation” on page 3-20 for more information.
S-Function	Not supported
S-Function Builder	Not supported

# Support for Code Generation from MATLAB

---

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 Supported for Code Generation”.

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

<b>Function Category</b>	<b>Support Notes</b>	
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 with the following limitations:	
	exp	Not supported.
	expm	Not supported.
	expm1	Not supported.
	log	Not supported.
	log2	Not supported.
	log10	Not supported.
	log1p	Not supported.
	nextpow2	Not supported.
	nthroot	Not supported.
	reallog	Not supported.
	realpow	Not supported.
	realsqrt	Not supported.
	sqrt	Not supported.
Filtering and convolution functions	Supported with the following limitations:	
	detrend	Not supported.
Fixed-Point Toolbox™ functions	Supported with the following limitations:	
	complex	Not supported.
Histogram functions	Not supported.	
Image Processing Toolbox™ functions	Not supported.	
Input and output functions	—	

<b>Function Category</b>	<b>Support Notes</b>	
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.
	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.

<b>Function Category</b>	<b>Support Notes</b>	
	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.	





**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 the Simulink Design Verifier software 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 Chapter 4, “Working with Block Replacements”).

**component verification**

The process of verifying an individual components in a model. You can verify a component within the execution context of the model, or independently of its parent model.

**condition coverage**

Measures the percentage of the total number of logic conditions associated with logical model objects that the simulation actually exercised. Enabling condition coverage causes every decision and condition coverage outcome to be enabled. See “Types of Model Coverage” in the *Simulink Verification and Validation User’s Guide*.

**constraint**

A property that is forced to be true during test case generation.

**counterexample**

A test case that demonstrates a property violation.

**coverage objective**

A test objective that defines when a coverage point results in a particular outcome.

**coverage point**

A decision, condition, or MCDC expression associated with a model object. Each coverage point has a fixed number of mutually exclusive outcomes.

**decision coverage**

Measures the percentage of the total number of simulation paths through model objects that the simulation actually traversed. Decision coverage is a subset of modified decision/condition coverage. See “Types of Model Coverage” in the *Simulink Verification and Validation User’s Guide*.

**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  $x == (x+1)$  is true; however, this expression never holds if  $x$  is a rational number.

**invalid test case**

A test case that does not satisfy its objectives.

**modified condition/decision coverage (MCDC)**

Measures the independence of logical block inputs and transition conditions associated with logical model objects during the simulation. When you set the coverage objective to MCDC, Simulink Design Verifier automatically enables every coverage objective for decision coverage and condition coverage as well.

Note that MCDC test cases are not generated for XOR configured logic operators. You can achieve MCDC by using the same tests that would be generated from AND configured blocks or OR configured blocks.

See “Types of Model Coverage” in the *Simulink Verification and Validation User’s Guide*.

**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.

# Examples

---

Use this list to find examples in the documentation.

## **Generating Test Cases**

- “Analyzing a Model” on page 1-6
- “Analyzing a Subsystem” on page 1-29
- “Analyzing a Stateflow Atomic Subchart” on page 1-31
- “Constructing the Example Model” on page 7-5
- “Checking Compatibility of the Example Model” on page 7-7
- “Configuring Test Generation Options” on page 7-8
- “Analyzing the Example Model” on page 7-9
- “Customizing Test Generation” on page 7-18
- “Reanalyzing the Example Model” on page 7-21
- “Example: Extending Existing Test Cases for a Model that Uses Temporal Logic” on page 8-4
- “Example: Extending Existing Test Cases for a Closed-Loop System” on page 8-11
- “Example: Extending Existing Test Cases for a Modified Model” on page 8-14
- “Example: Achieving Missing Coverage in a Referenced Model” on page 9-3
- “Example: Achieving Missing Coverage in a Closed-Loop Simulation Model” on page 9-8

## **Automatic Stubbing**

“Analyzing a Model Using Automatic Stubbing” on page 2-14

## **Working with Block Replacements**

“Example: Replacing Multiport Switch Blocks” on page 4-9

“Configuring Block Replacements” on page 4-18



## **Specifying Parameter Configurations**

“Constructing the Example Model” on page 5-10

“Parameterizing the Constant Block” on page 5-11

“Specifying a Parameter Configuration” on page 5-12

“Analyzing the Example Model” on page 5-13

“Simulating the Test Cases” on page 5-16

## **Component Verification**

“Example: Verifying a Component for Code Generation” on page 10-6

## **Considering Specified Minimum and Maximum Inputs**

“Example: Output Minimum and Maximum Values on Inport Blocks” on page 11-4

“Example: Minimum and Maximum Values in Simulink.Signal Objects” on page 11-8

“Example: Minimum and Maximum Values on Stateflow Data Objects” on page 11-10

“Example: Minimum and Maximum Values in Subsystems” on page 11-13

“Example: Minimum and Maximum Values in Global Data Storage” on page 11-16

## **Proving Properties of a Model**

“Constructing the Example Model” on page 12-6

“Instrumenting the Example Model” on page 12-9

“Configuring Property-Proving Options” on page 12-10

“Analyzing the Example Model” on page 12-11

“Customizing the Example Proof” on page 12-21

“Reanalyzing the Example Model” on page 12-22

“Using a Verification Model to Prove System-Level Properties” on page 12-28

“Property-Proving Examples” on page 12-33

## A

- Additional Math and Discrete Library
  - support for blocks in 16-3
- AnalysisInformation field 13-9
- analyzing
  - Model blocks
    - Simulink Design Verifier analysis 2-6
- analyzing large models
  - bottom-up approach to 14-15
  - characteristics that cause problems
    - with 14-2
  - generating reports 14-8
  - generating tests incrementally 14-13
  - initial steps 14-4
  - mathematical techniques for 2-10
  - optimization strategy for 14-6
  - partitioning model inputs 14-13
  - property-proving techniques 14-27
  - simplifying 14-9
  - types of problems 14-3
- analyzing models
  - overview 2-2
  - simple example 2-3
  - with counters and timers 14-25
  - with large state spaces 14-24
- approximations
  - converting floating-point arithmetic 2-21
  - during model analysis 2-20
  - ensuring validity of analysis when using 2-22
  - linearizing two-dimensional lookup
    - tables 2-21
  - types 2-20
  - unrolling while loops 2-22
- atomic subcharts
  - analyzing 1-31
- automatic stubbing
  - achieving complete results after 2-18
  - definition 2-11
  - enabling 2-16
  - enabling after compatibility check 3-7

- function-call triggers and 2-13
- reviewing results after 2-17
- S-Function blocks and 2-13
- Trigonometric Function blocks and 2-11
- workflow 2-14

## B

- block reduction
  - how analysis handles 2-7
- block replacements
  - configuration 4-18
  - example 4-8
  - execution 4-19
  - factory defaults 4-4
  - for unsupported blocks after automatic stubbing 2-18
  - introduction 4-2
  - template 4-7
- block support
  - limitations 3-12
  - summary 16-1
- blocks
  - Model
    - Simulink Design Verifier analysis 2-6

## C

- closed-loop controllers
  - achieving missing coverage for 9-8
  - extending existing test cases for 8-11
- code generation from MATLAB
  - support for 17-2
- combining
  - test cases 1-28
- Commonly Used Blocks Library
  - support for blocks in 16-4
- component verification
  - approaches 10-2
  - example 10-6

- functions for 10-4
- tools for 10-2
- components
  - verifying. *See* component verification
- configuration parameters
  - Block Replacements pane 15-14
    - Apply block replacements 15-15
    - File path of the output model 15-17
    - List of block replacement rules 15-16
  - Design Error Detection pane 15-36
    - Check specified intermediate minimum and maximum values 15-38
    - Dead logic 15-36
    - Division by zero 15-37
    - Integer overflow 15-37
  - Design Verifier pane 15-4
    - Automatic stubbing of unsupported blocks and functions 15-8
    - Display unsatisfiable test objectives 15-7
    - Make output file names unique by adding a suffix 15-12
    - Maximum analysis time 15-6
    - Mode 15-4
    - Output directory 15-10
    - Use specified input minimum and maximum values 15-9
  - Parameters pane 15-19
    - Apply parameters 15-19
    - Parameter configuration file 15-19
  - Property Proving pane 15-41
    - Assertion blocks 15-42
    - Maximum violation steps 15-45
    - Proof assumptions 15-43
    - Strategy 15-44
  - Report pane 15-62
    - Display report 15-66
    - Generate report of the results 15-63
    - Include screen shots of properties 15-65
    - Report file name 15-64
- Results pane 15-48
  - Data file name 15-50
  - Display results of the analysis on the model 15-53
  - Harness model file name 15-56
  - Include expected output values 15-51
  - Randomize data that does not affect outcome 15-52
  - Reference input model in generated harness 15-57
  - Save test data to file 15-49
  - Save test harness as model 15-55
  - Save test harness as SystemTest TEST-file (will reference saved data file) 15-59
  - SystemTest file name: 15-60
- summary 15-67
- Test Generation pane 15-23
  - Coverage data file 15-32
  - Coverage filter file 15-34
  - Data file 15-30
  - Extend existing test cases 15-29
  - Ignore objectives based on filter 15-33
  - Ignore objectives satisfied by existing test cases 15-31
  - Ignore objectives satisfied in existing coverage data 15-31
  - Maximum test case steps 15-27
  - Model coverage objectives 15-24
  - Test conditions 15-25
  - Test objectives 15-26
  - Test suite optimization 15-28
- Constraints field 13-11
- Continuous Library
  - support for blocks in 16-5
- CounterExamples field 13-12
- counters
  - analyzing models with 14-25

**D**

- data stores
  - analyzing subsystems that read from 14-17
  - specified input minimum and maximum values in 11-16
- dead logic
  - detecting, example 6-9
- derived range
  - definition 6-3
- design error detection
  - analysis report 6-34
  - analyzing model for 6-29
  - harness model 6-33
  - introduction 6-2
  - model highlighting 6-5
  - reviewing results on model 6-30
  - workflow 6-5
- design error detection objectives
  - Simulink Design Verifier reports 13-36
- design range
  - definition 6-3
- Discontinuities Library
  - support for blocks in 16-6
- Discrete Library
  - support for blocks in 16-7
- discretization
  - constraining data 14-9
- division-by-zero errors
  - detecting, example 6-29

**E**

- extending test cases
  - common workflow 8-3
  - for closed-loop systems 8-11
  - for models with temporal logic 8-4
  - for modified models 8-14
  - when to use 8-2

**F**

- fixed-point data overflow errors
  - detecting, example 6-29
- floating-point data
  - constraining for model analysis 14-9
  - converting to rational 2-21
- function-call subsystems
  - analyzing 14-19

**G**

- generating test cases 1-8
  - incrementally 14-13

**H**

- harness model
  - configuration 13-21
  - contents 1-22
  - simulating 13-22
- harness models
  - contents 13-16
- highlighted results on model 13-2

**I**

- incompatibilities
  - automatic stubbing and 2-11
- inline parameters
  - how analysis handles 2-9
- input ports
  - Simulink Design Verifier support for 11-2
    - enabling 11-2
    - in subsystems 11-13
    - limitations 11-3
    - on `Simulink.Signal` objects 11-8
    - on Stateflow data objects 11-10
    - parameters 11-4
- integer overflow errors
  - detecting, example 6-29

**L**

## large models

- analyzing, initial steps 14-4
- bottom-up approach to analyzing 14-15
- complexity of 14-2
- generating analysis reports for 14-8
- mathematical techniques for analyzing 2-10
- optimization strategy 14-6
- simplifying analysis of 14-9
- techniques for proving properties of 14-27
- type of problems analyzing 14-3

## linearizing

- two-dimensional lookup tables 2-21

## log files 13-52

## Logic and Bit Operations Library

- support for blocks in 16-8

## logic blocks

- short-circuiting 2-23

## logical operations

- analyzing 14-23

## Logical Operator blocks

- short-circuiting 2-23

## Lookup Table Library

- support for blocks in 16-9

## lookup tables

- linearizing 2-21

**M**

## Math Operations Library

- support for blocks in 16-10

## MATLAB for code generation

- features Simulink Design Verifier does not support 3-20

## MATLAB functions

- for property proofs 12-3
- for test cases 7-2
- limitations for Simulink Design Verifier analysis 3-21

## minimum and maximum values

## Simulink Design Verifier support for 11-2

- enabling 11-2
- in subsystems 11-13
- limitations 11-3
- on `Simulink.Signal` objects 11-8
- on Stateflow data objects 11-10
- parameters 11-4

## model compatibility

- checking 2-15 3-2
- enabling automatic stubbing after 3-7

## model coverage

- using Simulink Design Verifier analysis to achieve missing 9-2

## Model Explorer

- reviewing latest analysis results in 6-7

## Model Verification Library

- support for blocks in 16-12

## Model-Wide Utilities Library

- support for blocks in 16-13

## ModelInformation field 13-8

## ModelObjects field 13-10

## models 14-2

- analyzing, overview 2-2
  - complexity of 14-2
  - mathematical techniques for simplifying analysis 2-10
- See also* large models

**N**

## nonfinite data

- Simulink Design Verifier handling of 2-19

**O**

## Objectives field 13-11

**P**

## parameter configurations

- example 5-9



- introduction 5-2
- parameters
  - configuring for analysis 5-3
  - data type issues for defining constraints in parameter configuration file 5-7
  - syntax for defining constraints in parameter configuration file 5-3
  - template for defining constraints in parameter configuration file 5-3
- Ports & Subsystems Library
  - support for blocks in 16-14
- proof objectives
  - Simulink Design Verifier reports 13-41
- property proofs
  - blocks 12-2
  - example 12-5
  - introduction 12-2
  - MATLAB functions 12-3
  - Stateflow actions 12-2
  - subsystems 12-32
  - techniques for large models 14-27
  - workflow 12-4

## R

- rational data
  - converting floating-point data to 2-21
- referenced models
  - achieving missing coverage for 9-3

## S

- S-Function blocks
  - automatic stubbing of 2-13
- short-circuiting
  - logic blocks 2-23
- Signal Attributes Library
  - support for blocks in 16-16
- Signal Routing Library
  - support for blocks in 16-17

- Simulation range checking** parameter
  - importance for specified minimum and maximum violations 6-43
- Simulink Design Verifier data files
  - fields 13-8
  - overview 13-7
  - simulation 13-13
- Simulink Design Verifier reports
  - analysis information 13-28
  - approximations 13-33
  - block replacements summary 13-31
  - Constraints 13-31
  - derived ranges 13-34
  - design error detection objectives 13-36
  - design errors 13-44
  - model items 13-43
  - proof objectives 13-41
  - properties 13-50
  - summary 13-28
  - table of contents 13-28
  - test cases 13-45
  - test objectives 13-38
  - test/proof objectives 13-35
  - title 13-28
  - Unsupported Blocks 13-30
- Simulink Design Verifier Results window
  - clicking objects on highlighted model displays 13-3
- Simulink® Design Verifier™ software
  - analyzing demo model 1-6
  - block library 1-4
  - model parameters 15-2 15-67
  - starting 1-4
  - workflow 1-34
- Sinks Library
  - support for blocks in 16-18
- sldvData structure
  - fields for minimum and maximum inputs 11-6
- Sources Library

- support for blocks in 16-19
- specified input minimum and maximum values
  - in data stores 11-16
  - Simulink Design Verifier support for 11-2
    - enabling 11-2
    - in subsystems 11-13
    - limitations 11-3
    - on Simulink.Signal objects 11-8
    - on Stateflow data objects 11-10
    - parameters 11-4
  - sldvData fields for 11-6
- specified minimum and maximum violations
  - analysis report 6-43
  - analyzing model for 6-39
  - example model for 6-36
  - harness model 6-42
  - overview 6-35
  - reviewing results on model 6-40
  - Simulation range checking** parameter
    - importance for 6-43
- state spaces
  - analyzing models with large 14-24
- stubbing. *See* automatic stubbing
- subsystems
  - achieving missing coverage for 9-8
  - analyzing 1-29
  - extracting, for analysis 14-16
  - function-call, analyzing 14-19
  - generating test cases for 7-23
  - proving properties of 12-32
  - that read from data stores, analyzing 14-17
- system requirements 1-3
- SystemTest TEST-files
  - creating 13-24

## T

- temporal logic
  - extending test cases for models with 8-4 8-11
- test case generation

- blocks 7-2
- example 7-5
- introduction 7-2
- MATLAB functions 7-2
- subsystems 7-23
- test objectives 2-4
- workflow 7-4

- test cases
  - combining 1-28
  - extending existing
    - common workflow 8-3
    - for closed-loop system 8-11
    - for models with temporal logic 8-4
    - for modified models 8-14
    - when to use 8-2
  - for missing model coverage 9-2
  - generating 1-8
  - generating, incrementally 14-13
- test objectives
  - generating test cases 2-4
  - Simulink Design Verifier reports 13-38
- test suite optimization
  - large model option 14-6
- TestCases field 13-12
- timers
  - analyzing models with 14-25
- Trigonometric Function block
  - Simulink Design Verifier does not support 2-14
- Trigonometric Function blocks
  - automatic stubbing of 2-11
- two-dimensional lookup tables
  - linearizing 2-21

## U

- unrolling
  - while loops 2-22
- unsupported features
  - MATLAB for code generation 3-20

Simulink 3-9  
Stateflow 3-15  
User-Defined Functions Library  
support for blocks in 16-20

**V**

Version field 13-13

**W**

while loops  
unrolling 2-22