

# Simulink® Design Verifier™

User's Guide



# MATLAB® & SIMULINK®

R2023a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Simulink® Design Verifier™ User's Guide*

© COPYRIGHT 2007–2023 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

Prover, Prover Technology, Prover Plug-In and the Prover logo are trademarks or registered trademarks of Prover Technology AB in Sweden, the United States and in other countries. 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)         |
| September 2012 | Online only | Revised for Version 2.3 (Release 2012b)         |
| March 2013     | Online only | Revised for Version 2.4 (Release 2013a)         |
| September 2013 | Online only | Revised for Version 2.5 (Release 2013b)         |
| March 2014     | Online only | Revised for Version 2.6 (Release 2014a)         |
| October 2014   | Online only | Revised for Version 2.7 (Release 2014b)         |
| March 2015     | Online only | Revised for Version 2.8 (Release 2015a)         |
| September 2015 | Online only | Revised for Version 3.0 (Release 2015b)         |
| October 2015   | Online only | Rereleased for Version 2.8.1 (Release 2015aSP1) |
| March 2016     | Online only | Revised for Version 3.1 (Release 2016a)         |
| September 2016 | Online only | Revised for Version 3.2 (Release 2016b)         |
| March 2017     | Online only | Revised for Version 3.3 (Release 2017a)         |
| September 2017 | Online only | Revised for Version 3.4 (Release 2017b)         |
| March 2018     | Online only | Revised for Version 3.5 (Release 2018a)         |
| September 2018 | Online only | Revised for Version 4.0 (Release 2018b)         |
| March 2019     | Online only | Revised for Version 4.1 (Release 2019a)         |
| September 2019 | Online only | Revised for Version 4.2 (Release 2019b)         |
| March 2020     | Online only | Revised for Version 4.3 (Release 2020a)         |
| September 2020 | Online only | Revised for Version 4.4 (Release 2020b)         |
| March 2021     | Online only | Revised for Version 4.5 (Release 2021a)         |
| September 2021 | Online only | Revised for Version 4.6 (Release 2021b)         |
| March 2022     | Online only | Revised for Version 4.7 (Release 2022a)         |
| September 2022 | Online only | Revised for Version 4.8 (Release 2022b)         |
| March 2023     | Online only | Revised for Version 4.9 (Release 2023a)         |



## Acknowledgments

## Getting Started

### 1

|  |             |
|--|-------------|
| <b>Simulink Design Verifier Product Description</b> .....          | <b>1-2</b>  |
| <b>Simulink Design Verifier Block Library</b> .....                | <b>1-3</b>  |
| <b>Analyze a Model</b> .....                                       | <b>1-4</b>  |
| About This Example .....   | 1-4         |
| Open the Model .....   | 1-4         |
| Generate Test Cases .....  | 1-5         |
| Combine Test Cases .....   | 1-15        |
| <b>Analyze a Stateflow Atomic Subchart</b> .....                   | <b>1-17</b> |
| Analyze an Atomic Subchart by Using Simulink Design Verifier ..... | 1-17        |
| <b>Overview of the Simulink Design Verifier Workflow</b> .....     | <b>1-19</b> |
| Check Model Compatibility .....                                    | 1-19        |
| Apply Block Replacement Rules .....                                | 1-19        |
| Set Simulink Design Verifier Options .....                         | 1-20        |
| Perform Analysis on Model .....                                    | 1-20        |
| Generate Analysis Results .....                                    | 1-20        |
| Interpret Analysis Results .....                                   | 1-20        |

## How the Simulink Design Verifier Software Works

### 2

|   |            |
|---|------------|
| <b>Analyze a Simple Model</b> .....                           | <b>2-2</b> |
| <b>Model Blocks</b> .....                                     | <b>2-4</b> |
| <b>Block Reduction</b> .....                                  | <b>2-5</b> |
| <b>Large Models</b> .....                                     | <b>2-6</b> |
| <b>Handle Incompatibilities with Automatic Stubbing</b> ..... | <b>2-7</b> |
| What Is Automatic Stubbing? .....                             | 2-7        |

|   |             |
|---|-------------|
| How Automatic Stubbing Works .....  | 2-7         |
| Analyze a Model Using Automatic Stubbing .....  | 2-9         |
| <b>Analyze Export-Function Models .....</b>   | <b>2-12</b> |
| Limitations .....   | 2-12        |
| <b>Analyze Export-Function Model with Function-Call Subsystems .....</b>                                | <b>2-13</b> |
| <b>Analyze Export-Function Model with Global Simulink Function .....</b>                                | <b>2-16</b> |
| <b>Nonfinite Data .....</b>   | <b>2-19</b> |
| <b>Role of Approximations During Model Analysis .....</b>   | <b>2-20</b> |
| Types of Approximations .....   | 2-20        |
| Floating-Point to Rational Number Conversion .....  | 2-20        |
| Linearization of Two-Dimensional Lookup Tables for Floating-Point Data<br>Types .....                   | 2-21        |
| Approximation of One- and Two-Dimensional Lookup Tables for Integer and<br>Fixed-Point Data Types ..... | 2-21        |
| While Loops .....   | 2-22        |
| <b>How Simulink Design Verifier Reports Approximations Through<br/>Validation Results .....</b>         | <b>2-23</b> |
| Impact of Approximations on Objectives Status .....   | 2-23        |
| Identify the Effect of Approximations Through Validation Results .....                                  | 2-24        |
| <b>Logic Operations Short-Circuiting .....</b>  | <b>2-26</b> |
| <b>Model Representation for Analysis .....</b>  | <b>2-28</b> |
| Reuse Model Representation for Analysis .....   | 2-28        |
| Limitations .....   | 2-30        |
| <b>Share Simulink Cache File for Faster Analysis .....</b>  | <b>2-31</b> |
| Store the Simulink Cache File .....   | 2-31        |
| Reuse the Simulink Cache File .....   | 2-31        |
| <b>Analyze AUTOSAR Component Models .....</b>   | <b>2-33</b> |
| Limitations .....   | 2-33        |
| <b>Extend Existing Test Cases by Reusing Model Representation .....</b>                                 | <b>2-35</b> |
| <b>Configure Model Representation Options .....</b>   | <b>2-39</b> |
| <b>Run Additional Analysis to Reduce Instances of Rational Approximation<br/>.....</b>                  | <b>2-42</b> |
| <b>Detect Design Errors in AUTOSAR Software Component Model .....</b>                                   | <b>2-47</b> |

# Checking Compatibility with the Simulink Design Verifier Software

## 3

|   |             |
|---|-------------|
| <b>Check Model Compatibility</b> .....  | <b>3-2</b>  |
| Run Compatibility Check .....   | <b>3-2</b>  |
| Compatibility Check Results .....   | <b>3-3</b>  |
| <b>Supported and Unsupported Simulink Blocks in Simulink Design Verifier</b><br>.....       | <b>3-7</b>  |
| <b>Support Limitations for Simulink Software Features</b> .....                             | <b>3-16</b> |
| <b>Support Limitations for Model Blocks</b> .....   | <b>3-19</b> |
| <b>Support Limitations for Stateflow Software Features</b> .....                            | <b>3-21</b> |
| ml Namespace Operator, ml Function, ml Expressions .....                                    | <b>3-21</b> |
| C or C++ Operators .....  | <b>3-21</b> |
| C Math Functions .....  | <b>3-21</b> |
| Atomic Subcharts That Call Exported Graphical Functions Outside a<br>Subchart .....         | <b>3-22</b> |
| Atomic Subchart Input and Output Mapping .....  | <b>3-22</b> |
| Recursion and Cyclic Behavior .....   | <b>3-22</b> |
| Custom C/C++ Code .....   | <b>3-23</b> |
| Textual Functions with Literal String Arguments .....                                       | <b>3-24</b> |
| <b>Support Limitations for MATLAB for Code Generation</b> .....                             | <b>3-25</b> |
| Unsupported MATLAB for Code Generation Features .....                                       | <b>3-25</b> |
| Support Limitations for MATLAB for Code Generation Library Functions<br>.....               | <b>3-25</b> |
| <b>Support Limitations and Considerations for S-Functions and C/C++ Code</b><br>.....       | <b>3-28</b> |
| Enabling S-Functions in Simulink Design Verifier .....                                      | <b>3-28</b> |
| Support Limitations for S-Functions and C/C++ Code .....                                    | <b>3-28</b> |
| Handle Volatile Variables as Normal Variables .....   | <b>3-29</b> |
| Considerations for Enabling S-Functions and C/C++ Code in Simulink<br>Design Verifier ..... | <b>3-29</b> |
| Source Code Protection .....  | <b>3-29</b> |

## Working with Block Replacements

## 4

|  |            |
|--|------------|
| <b>What Is Block Replacement?</b> .....                | <b>4-2</b> |
| Block Replacement Effects on Test Generation .....     | <b>4-2</b> |
| <b>Built-In Block Replacements</b> .....               | <b>4-4</b> |
| <b>Template for Block Replacement Rules</b> .....      | <b>4-6</b> |
| <b>Block Replacements for Unsupported Blocks</b> ..... | <b>4-7</b> |

|   |      |
|---|------|
| <b>Parameter Configuration for Analysis</b> .....                                 | 5-2  |
| What is Parameter Configuration for Analysis? .....                               | 5-2  |
| Specify Parameter Constraints for Models Using Referenced Configuration Set ..... | 5-3  |
| Data Types in Parameter Configurations .....                                      | 5-4  |
| Parameters in Variant Blocks .....  | 5-5  |
| <b>Use Parameter Table</b> .....  | 5-7  |
| Find Parameters .....   | 5-8  |
| Edit Parameter Constraints .....  | 5-10 |
| Highlight Constrained Parameters in Model .....                                   | 5-11 |
| <b>Specify Parameter Configuration for Structure or Bus Parameters</b> ....       | 5-12 |
| About This Example Model .....  | 5-12 |
| Preload Workspace Variable for Structure Parameter .....                          | 5-12 |
| Define Parameter Constraint Values .....  | 5-13 |
| Define Parameter Constraint Values using Parameter Table .....                    | 5-13 |
| Define Constraint Values using Parameter Configuration File .....                 | 5-14 |
| Analyze Example Model .....   | 5-15 |
| <b>Specify Parameter Configuration for Full Coverage</b> .....                    | 5-17 |
| About This Example .....  | 5-17 |
| Construct Example Model .....   | 5-17 |
| Parameterize Constant Block .....   | 5-18 |
| Preload Workspace Variable .....  | 5-18 |
| Autogenerate Parameter Constraint .....   | 5-19 |
| Analyze Example Model .....   | 5-20 |
| Simulate Test Cases .....   | 5-22 |
| <b>Store Parameter Constraints in MATLAB Code Files</b> .....                     | 5-26 |
| Export Parameter Constraints to File .....  | 5-26 |
| Import Parameter Constraints from File .....                                      | 5-27 |
| <b>Use Parameter Configuration File</b> .....                                     | 5-29 |
| Template Parameter Configuration File .....                                       | 5-29 |
| Syntax in Parameter Configuration Files .....                                     | 5-29 |
| <b>Automatically Infer Parameter Specification</b> .....                          | 5-32 |
| Configuring Parameters by Using Automatically infer parameter specification ..... | 5-33 |
| <b>Determine from Generated Code</b> .....  | 5-36 |
| Configuring Parameters by Using Determine from generated code .....               | 5-37 |
| <b>Using Command Line Functions to Support Changing Parameters</b> ....           | 5-39 |
| <b>Generate Parameters Values</b> .....   | 5-45 |
| <b>Extend Existing Test Cases After Applying Parameter Configurations</b> ..      | 5-46 |



|  |             |
|--|-------------|
| <b>What Is Design Error Detection?</b> .....                                 | <b>6-2</b>  |
| <b>Derived Ranges in Design Error Detection</b> .....                        | <b>6-3</b>  |
| <b>Analyze Models for Design Errors</b> .....                                | <b>6-4</b>  |
| Workflow for Detecting Design Errors .....                                   | <b>6-4</b>  |
| Understand the Analysis Results .....  | <b>6-4</b>  |
| Review the Latest Analysis Results in the Results Summary Window .....       | <b>6-5</b>  |
| Check For Design Errors using the Model Advisor .....                        | <b>6-6</b>  |
| <b>Dead Logic Detection</b> .....  | <b>6-7</b>  |
| Run a Partial Check for Dead Logic .....                                     | <b>6-7</b>  |
| Run an Exhaustive Analysis for Dead Logic .....                              | <b>6-7</b>  |
| Run a Dead Logic Analysis and Review Results .....                           | <b>6-8</b>  |
| <b>Detect Dead Logic Caused by an Incorrect Value</b> .....                  | <b>6-12</b> |
| Analyze the Fuel System Model .....  | <b>6-12</b> |
| Review the Results and Trace to the Model .....                              | <b>6-13</b> |
| Investigate the Cause of the Dead Logic .....                                | <b>6-13</b> |
| Update the Input Constraint and Reanalyze the Model .....                    | <b>6-14</b> |
| <b>Common Causes for Dead Logic</b> .....                                    | <b>6-15</b> |
| Short-Circuiting of a Logical Operator Block During Analysis .....           | <b>6-15</b> |
| Conditional Execution of a Block .....                                       | <b>6-15</b> |
| Parameter Values Treated as Constants .....                                  | <b>6-16</b> |
| Upstream Blocks .....  | <b>6-17</b> |
| Library-Linked Blocks .....  | <b>6-17</b> |
| Restrictions on Signal Ranges .....  | <b>6-17</b> |
| <b>Detect Integer Overflow and Division-by-Zero Errors</b> .....             | <b>6-19</b> |
| About This Example .....   | <b>6-19</b> |
| Analyze the Model .....  | <b>6-19</b> |
| Review the Analysis Results .....  | <b>6-19</b> |
| <b>Check for Specified Minimum and Maximum Value Violations</b> .....        | <b>6-23</b> |
| Limitations of Checking Specified Minimum and Maximum Value Violations ..... | <b>6-23</b> |
| About This Example .....   | <b>6-23</b> |
| Create the Example Model .....   | <b>6-24</b> |
| Analyze the Model .....  | <b>6-25</b> |
| Review the Analysis Results .....  | <b>6-25</b> |
| <b>Detect Out of Bound Array Access Errors</b> .....                         | <b>6-28</b> |
| Design Error Detection for Out of Bound Array Access .....                   | <b>6-28</b> |
| Detect Out of Bound Array Access Example Model .....                         | <b>6-28</b> |
| Limitations of Support for Out of Bound Array Access Design Error .....      | <b>6-31</b> |
| Detection .....  | <b>6-31</b> |
| <b>Detect Non-Finite, NaN, and Subnormal Floating-Point Values</b> .....     | <b>6-33</b> |
| Assumptions and Limitations .....  | <b>6-33</b> |
| Run Design Error Detection Analysis to Detect Floating-Point Errors .....    | <b>6-33</b> |

|  |             |
|--|-------------|
| <b>Detect Data Store Access Violations</b> .....                                 | <b>6-37</b> |
| Detect Data Store Access Violations in a Model .....                             | <b>6-37</b> |
| <b>Detect Violations of High-Integrity Systems Modeling Guidelines</b> .....     | <b>6-41</b> |
| Usage of rem and reciprocal operations - hisl_0002 .....                         | <b>6-41</b> |
| Usage of square root operations - hisl_0003 .....                                | <b>6-41</b> |
| Usage of log and log10 operations - hisl_0004 .....                              | <b>6-41</b> |
| Usage of Reciprocal Square Root blocks - hisl_0028 .....                         | <b>6-41</b> |
| Detect Violations of High-Integrity Systems Modeling Guidelines .....            | <b>6-41</b> |
| <b>Filter Objectives by Using Simulink Design Verifier Filter Explorer</b> ..... | <b>6-46</b> |
| Use the Simulink Design Verifier Filter Explorer to Edit Filter Files .....      | <b>6-46</b> |
| Limitations .....  | <b>6-49</b> |
| <b>Detect Integer Overflow Errors</b> .....                                      | <b>6-51</b> |
| <b>Detect Out of Bound Array Access Example Model</b> .....                      | <b>6-54</b> |
| <b>Detect Design Errors in C/C++ Custom Code</b> .....                           | <b>6-57</b> |
| <b>Exclude and Justify Objectives for Design Error Detection</b> .....           | <b>6-59</b> |
| <b>Detect Integer Overflow in a Model with Complex Inputs</b> .....              | <b>6-65</b> |
| <b>Debug Integer Overflow Design Error Detection Using Model Slicer</b> ...      | <b>6-68</b> |
| <b>Analyzing the Results for a Dead Logic Analysis</b> .....                     | <b>6-73</b> |

## Generating Test Cases

# 7

|  |             |
|--|-------------|
| <b>What Is Test Case Generation?</b> .....                             | <b>7-3</b>  |
| Test Case Blocks .....   | <b>7-3</b>  |
| Test Case Functions .....  | <b>7-3</b>  |
| <b>Workflow for Test Case Generation</b> .....                         | <b>7-5</b>  |
| <b>Generate Test Cases for Model Decision Coverage</b> .....           | <b>7-6</b>  |
| Construct the Example Model .....                                      | <b>7-6</b>  |
| Check Compatibility of the Example Model .....                         | <b>7-7</b>  |
| Configure Test Generation Options .....                                | <b>7-8</b>  |
| Analyze the Example Model .....  | <b>7-8</b>  |
| Review Analysis Results .....  | <b>7-8</b>  |
| Customize Test Generation .....  | <b>7-14</b> |
| Reanalyze the Example Model .....                                      | <b>7-16</b> |
| Analyze Contradictory Models .....                                     | <b>7-16</b> |
| <b>Generate Test Cases for a Subsystem</b> .....                       | <b>7-18</b> |
| Generate Test Cases for Subsystems for Normal Mode .....               | <b>7-18</b> |
| Generate Test Cases for Subsystems for Software-in-the-Loop Mode ..... | <b>7-19</b> |

|   |             |
|---|-------------|
| <b>Generate Test Cases for a Reusable Library Subsystem</b> .....                         | <b>7-21</b> |
| Generate Test Cases for RLS in Software-in-the-Loop Mode .....                            | <b>7-21</b> |
| <b>Use Test Generation Advisor to Identify Analyzable Components</b> .....                | <b>7-24</b> |
| Test Generation Advisor .....   | <b>7-24</b> |
| Test Generation Advisor Requirements .....  | <b>7-25</b> |
| Identify Analyzable Components .....  | <b>7-25</b> |
| Analyze and Generate Tests for Model Components .....                                     | <b>7-25</b> |
| Manually Select Components for Testing .....  | <b>7-27</b> |
| <b>Generate Test Cases for Embedded Coder Generated Code</b> .....                        | <b>7-28</b> |
| Generate Test Cases for Generated Code from the Simulink Model Toolstrip<br>.....         | <b>7-28</b> |
| Generate Test Cases for Generated Code by Using the Simulink Design<br>Verifier API ..... | <b>7-29</b> |
| Generate Test Cases for Generated Code from the Simulink Test Test<br>Manager .....       | <b>7-29</b> |
| <b>Model Coverage Objectives for Test Generation</b> .....                                | <b>7-30</b> |
| Decision .....  | <b>7-30</b> |
| Condition .....   | <b>7-30</b> |
| MCDC .....  | <b>7-31</b> |
| Enhanced MCDC .....   | <b>7-31</b> |
| Relational Boundary .....   | <b>7-31</b> |
| <b>Enhance Model Coverage of Older Release Models</b> .....                               | <b>7-32</b> |
| Enhance Model Coverage by Generating Test Cases for Older Release<br>Model .....          | <b>7-33</b> |
| Enhance Model Coverage by Using Generated Code from Older Release<br>.....                | <b>7-37</b> |
| <b>Enhanced MCDC Coverage in Simulink Design Verifier</b> .....                           | <b>7-42</b> |
| Use Model Coverage Objectives for Enhanced MCDC Coverage .....                            | <b>7-42</b> |
| Author Custom Test Objectives for Enhanced MCDC Coverage .....                            | <b>7-43</b> |
| <b>Analyze Model for Enhanced MCDC Analysis</b> .....                                     | <b>7-44</b> |
| <b>Basic Workflow for Enhanced MCDC Analysis</b> .....                                    | <b>7-47</b> |
| Configure Detection Sites using Test-pointed Logged Signals .....                         | <b>7-48</b> |
| Configure Advanced Options for Enhanced MCDC Analysis .....                               | <b>7-49</b> |
| Inspect Enhanced MCDC Objectives using Model Slicer .....                                 | <b>7-50</b> |
| <b>Author Custom Test Objective Workflow</b> .....  | <b>7-52</b> |
| Steps for Authoring Custom Test Objectives .....  | <b>7-52</b> |
| Analyze Custom Test Objectives in Model for Enhanced MCDC .....                           | <b>7-53</b> |
| <b>What Is a Specification Model?</b> .....   | <b>7-60</b> |
| Use Specification Models in Requirements-Based Testing .....                              | <b>7-60</b> |
| Construct a Specification Model .....   | <b>7-61</b> |
| Iterate Through the Steps .....   | <b>7-65</b> |
| <b>Test Generation Examples</b> .....   | <b>7-66</b> |
| <b>Test Generation for Custom Code in MATLAB Function Block</b> .....                     | <b>7-67</b> |
| Generating Tests for Custom code in MATLAB function block .....                           | <b>7-67</b> |

|  |              |
|--|--------------|
| <b>Use Specification Models for Requirements-Based Testing</b> .....                 | <b>7-69</b>  |
| <b>Flip Flop Test Generation</b> .....   | <b>7-80</b>  |
| <b>Model Coverage Test Generation</b> .....  | <b>7-81</b>  |
| <b>Test Objective Block</b> .....  | <b>7-82</b>  |
| <b>Test Condition Block</b> .....  | <b>7-83</b>  |
| <b>Cruise Control Test Generation</b> .....  | <b>7-84</b>  |
| <b>Fuel Rate Controller Logic</b> .....  | <b>7-85</b>  |
| <b>Extend an Existing Test Suite</b> .....   | <b>7-86</b>  |
| <b>Defining and Extending Existing Tests Cases</b> .....                             | <b>7-91</b>  |
| <b>Using Existing Coverage Data During Subsystem Analysis</b> .....                  | <b>7-97</b>  |
| <b>Creating and Executing Test Cases</b> .....                                       | <b>7-100</b> |
| <b>Using Specified Input Minimum and Maximum Values as Constraints</b>               | <b>7-107</b> |
| <b>Configuring S-Function for Test Case Generation</b> .....                         | <b>7-109</b> |
| <b>Code Coverage Test Generation</b> .....   | <b>7-111</b> |
| <b>Test Generation on Model with C Caller Block</b> .....                            | <b>7-119</b> |
| <b>Debug Enhanced Modified Condition Decision Coverage Using Model Slicer</b> .....  | <b>7-121</b> |
| <b>Test Generation for Custom Code in a Stateflow Chart</b> .....                    | <b>7-124</b> |
| <b>Generate Test Cases for Model Blocks</b> .....                                    | <b>7-126</b> |
| <b>Use Observer Reference Block for Test Case Generation</b> .....                   | <b>7-130</b> |
| <b>Inspect Test Generation Objectives by Using Model Slicer</b> .....                | <b>7-135</b> |
| <b>Generate Tests for Model Block Component by Using Default Simulation</b><br>..... | <b>7-138</b> |
| <b>Add Test Cases Using Excel File</b> .....   | <b>7-142</b> |
| <b>Achieve Missing Coverage in Custom Code</b> .....                                 | <b>7-146</b> |
| <b>Achieve Missing Coverage in Generated Code of RLS</b> .....                       | <b>7-149</b> |

|  |             |
|--|-------------|
| <b>When to Extend Existing Test Cases</b> .....                    | <b>8-2</b>  |
| Common Workflow for Extending Existing Test Cases .....            | <b>8-2</b>  |
| Considerations for Starting Test Cases .....                       | <b>8-3</b>  |
| <b>Extend Test Cases for Model with Temporal Logic</b> .....       | <b>8-4</b>  |
| Create Starting Test Case .....                                    | <b>8-4</b>  |
| Log Starting Test Case .....                                       | <b>8-6</b>  |
| Extend Existing Test Cases .....                                   | <b>8-7</b>  |
| Verify Analysis Results .....                                      | <b>8-8</b>  |
| <b>Extend Test Cases for Closed-Loop System</b> .....              | <b>8-10</b> |
| Log Starting Test Case .....                                       | <b>8-10</b> |
| Extend Existing Test Cases .....                                   | <b>8-12</b> |
| <b>Extend Test Cases for Modified Model</b> .....                  | <b>8-15</b> |
| Create Starting Test Cases .....                                   | <b>8-15</b> |
| Extend Existing Test Cases .....                                   | <b>8-15</b> |
| <b>Create and Run Back-to-Back Tests Using Enhanced MCDC</b> ..... | <b>8-18</b> |

**Achieving Test Cases for Missing Model Coverage**

|  |             |
|--|-------------|
| <b>Generate Test Cases for Missing Coverage Data</b> .....                           | <b>9-2</b>  |
| <b>Achieve Missing Coverage in Referenced Model</b> .....                            | <b>9-3</b>  |
| Programmatically Achieve Missing Coverage in Referenced Model .....                  | <b>9-3</b>  |
| Increase Coverage for Referenced Models in a Test Harness .....                      | <b>9-5</b>  |
| <b>Achieve Missing Coverage in Subsystems and Model Blocks</b> .....                 | <b>9-10</b> |
| <b>Achieve Missing Coverage in Closed-Loop Simulation Model</b> .....                | <b>9-11</b> |
| Record Coverage Data for the Model .....   | <b>9-11</b> |
| Find Test Cases for Missing Coverage .....   | <b>9-12</b> |
| <b>Analyze Coverage for Lookup Table Boundary Values</b> .....                       | <b>9-14</b> |
| Generate Tests for Lookup Table Boundary Values .....                                | <b>9-16</b> |
| <b>Modified Condition and Decision Coverage in Simulink Design Verifier</b><br>..... | <b>9-21</b> |
| MCDC Definitions for Simulink Coverage and Simulink Design Verifier ..               | <b>9-21</b> |
| <b>Achieve Coverage in Models with Variable-Size Inputs</b> .....                    | <b>9-24</b> |

10

|  |              |
|--|--------------|
| <b>What Is Component Verification?</b> .....                           | <b>10-2</b>  |
| Component Verification Approaches .....                                | <b>10-2</b>  |
| Simulink Design Verifier Tools for Component Verification .....        | <b>10-2</b>  |
| <b>Functions for Component Verification</b> .....                      | <b>10-3</b>  |
| <b>Verify a Component for Code Generation</b> .....                    | <b>10-4</b>  |
| About the Example Model .....  | <b>10-4</b>  |
| Prepare the Component for Verification .....                           | <b>10-6</b>  |
| Record Coverage for the Component .....                                | <b>10-7</b>  |
| Use Simulink Design Verifier Software to Record Additional Coverage .. | <b>10-7</b>  |
| Combine the Harness Models .....                                       | <b>10-8</b>  |
| Execute the Component in Simulation Mode .....                         | <b>10-9</b>  |
| Execute the Component in Software-in-the-Loop (SIL) Mode .....         | <b>10-10</b> |

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

11

|   |              |
|---|--------------|
| <b>Minimum and Maximum Input Constraints</b> .....  | <b>11-2</b>  |
| Simulink Design Verifier Support for Specified Input Minimum and<br>Maximum Values .....          | <b>11-2</b>  |
| Limitations of Simulink Design Verifier Support for Specified Minimum and<br>Maximum Values ..... | <b>11-2</b>  |
| <b>Specify Input Ranges on Simulink and Stateflow Elements</b> .....                              | <b>11-4</b>  |
| Specify Input Ranges for Inport Blocks .....  | <b>11-4</b>  |
| Specify Input Ranges for Simulink.Signal Objects .....  | <b>11-5</b>  |
| Specify Input Ranges for Stateflow Data Objects .....   | <b>11-5</b>  |
| Specify Input Ranges for Subsystems .....   | <b>11-6</b>  |
| Specify Input Ranges for Global Data Stores .....   | <b>11-7</b>  |
| Specify Input Ranges for Bus Elements .....   | <b>11-8</b>  |
| <b>Specification of Input Ranges in sldvData Fields</b> .....                                     | <b>11-10</b> |

**Proving Properties of a Model**

12

|  |             |
|--|-------------|
| <b>What Is Property Proving?</b> .....             | <b>12-2</b> |
| Proof Blocks .....                                 | <b>12-2</b> |
| Proof Functions .....                              | <b>12-2</b> |
| <b>Workflow for Proving Model Properties</b> ..... | <b>12-4</b> |

|   |              |
|---|--------------|
| <b>Prove Properties in a Model</b> .....  | <b>12-5</b>  |
| About This Example .....  | 12-5         |
| Construct Example Model .....   | 12-5         |
| Check Compatibility of Example Model .....  | 12-6         |
| Instrument Example Model .....  | 12-7         |
| Configure Property-Proving Options .....  | 12-8         |
| Analyze Example Model .....   | 12-8         |
| Review Analysis Results .....   | 12-8         |
| Customize Example Proof .....   | 12-15        |
| Reanalyze Example Model .....   | 12-16        |
| Review Results of Second Analysis .....   | 12-16        |
| Analyze Contradictory Models .....  | 12-18        |
| Prove Properties in a Large Model .....   | 12-19        |
| <br>  |              |
| <b>Prove System-Level Properties Using Verification Model</b> .....                                   | <b>12-20</b> |
| When to Use a Verification Model for Property Proving .....   | 12-20        |
| About This Example .....  | 12-20        |
| Understand the Verification Model .....   | 12-20        |
| Prove the Properties of the Design Model .....  | 12-21        |
| Fix the Verification Model .....  | 12-22        |
| <br>  |              |
| <b>Prove Properties in a Subsystem</b> .....  | <b>12-23</b> |
| <br>  |              |
| <b>Model Requirements</b> .....   | <b>12-24</b> |
| Basic Properties .....  | 12-24        |
| Temporal Properties .....   | 12-26        |
| <br>  |              |
| <b>Isolate Verification Logic with Observers</b> .....  | <b>12-29</b> |
| Replace a Verification Subsystem with an Observer Reference Block ..                                  | 12-29        |
| Report on Observer Reference Blocks .....   | 12-31        |
| Limitations .....   | 12-31        |
| <br>  |              |
| <b>Property Proving with an Invalid Property</b> .....  | <b>12-32</b> |
| <br>  |              |
| <b>Property Proving with Multiple Properties</b> .....  | <b>12-33</b> |
| <br>  |              |
| <b>Property Proving with an Assumption Block</b> .....  | <b>12-34</b> |
| <br>  |              |
| <b>Property Proving Workflow for Cruise Control</b> .....   | <b>12-35</b> |
| <br>  |              |
| <b>Property Proving Workflow for Fixed-Point Cruise Control with Block<br/>    Replacements</b> ..... | <b>12-39</b> |
| <br>  |              |
| <b>Property Proving Using MATLAB Function Block</b> .....   | <b>12-40</b> |
| <br>  |              |
| <b>Property Proving Using MATLAB Truth Table Block</b> .....  | <b>12-41</b> |
| <br>  |              |
| <b>Property Proving Workflow for Thrust Reverser</b> .....  | <b>12-42</b> |
| <br>  |              |
| <b>Debounce Temporal Properties</b> .....   | <b>12-43</b> |
| <br>  |              |
| <b>Power Window Controller Temporal Properties</b> .....  | <b>12-46</b> |
| <br>  |              |
| <b>Debug Property Proving Violations by Using Model Slicer</b> .....                                  | <b>12-55</b> |

|  |              |
|--|--------------|
| <b>Design and Verify Properties in a Model</b> .....                     | <b>12-60</b> |
| <b>Validate Requirements by Analyzing Model Properties</b> .....         | <b>12-63</b> |
| <b>Use Observer Reference Blocks for Property Proving Analysis</b> ..... | <b>12-70</b> |
| <b>Prove Properties with Requirements Table Blocks</b> .....             | <b>12-73</b> |

## Reviewing the Results

# 13

|  |              |
|--|--------------|
| <b>Highlight Results on the Model</b> .....  | <b>13-2</b>  |
| Results Review with Model Highlighting .....   | <b>13-2</b>  |
| Simulink Design Verifier Results Inspector .....   | <b>13-2</b>  |
| Highlight Results on Model Automatically .....   | <b>13-2</b>  |
| Green Highlighting on Model .....  | <b>13-4</b>  |
| Red Highlighting on Model .....  | <b>13-4</b>  |
| Orange Highlighting on Model .....   | <b>13-4</b>  |
| Gray Highlighting on Model .....   | <b>13-6</b>  |
| <b>Manage Simulink Design Verifier Data Files</b> .....  | <b>13-7</b>  |
| Generate sldvData Structure .....  | <b>13-7</b>  |
| Model Information Fields in sldvData .....   | <b>13-7</b>  |
| Simulate Models Using Data Files .....   | <b>13-11</b> |
| Load Results from Data Files .....   | <b>13-11</b> |
| <b>Manage Simulink Design Verifier Harness Models</b> .....  | <b>13-13</b> |
| Harness Model Generation .....   | <b>13-13</b> |
| Create a Harness Model .....   | <b>13-13</b> |
| Contents of a Harness Model .....  | <b>13-13</b> |
| Configuration of the Harness Model .....   | <b>13-19</b> |
| Simulate the Harness Model .....   | <b>13-19</b> |
| <b>Simulate Harness Model with Signal Editor Inputs Block</b> .....  | <b>13-22</b> |
| <b>Export Test Cases to Simulink Test</b> .....  | <b>13-27</b> |
| Generate and Export Test Cases to Simulink Test .....  | <b>13-27</b> |
| <b>Export Tests from Models That Contain Requirements Table Blocks with Simulink Design Verifier</b> ..... | <b>13-30</b> |
| Construct the Model and Generate Tests .....   | <b>13-30</b> |
| Export the Tests to the Test Manager .....   | <b>13-31</b> |
| Run the Tests .....  | <b>13-33</b> |
| Inspect Test Failures .....  | <b>13-33</b> |
| <b>Review Results</b> .....  | <b>13-35</b> |
| Simulink Design Verifier Report Generation .....   | <b>13-35</b> |
| Create Analysis Reports .....  | <b>13-35</b> |
| Front Matter .....   | <b>13-35</b> |
| Summary Chapter .....  | <b>13-36</b> |
| Analysis Information Chapter .....   | <b>13-36</b> |
| Derived Ranges Chapter .....   | <b>13-40</b> |



|                                      |              |
|--------------------------------------|--------------|
| Objectives Status Chapters .....     | 13-42        |
| Model Items Chapter .....            | 13-50        |
| Design Errors Chapter .....          | 13-51        |
| Test Cases Chapter .....             | 13-52        |
| Properties Chapter .....             | 13-54        |
| <b>View Log Files .....</b>          | <b>13-56</b> |
| <b>Review Analysis Results .....</b> | <b>13-57</b> |
| View Active Results .....            | 13-57        |
| Load Previous Results .....          | 13-57        |
| Explore Results .....                | 13-57        |

# 14

## Analyzing Large Models and Improving Performance

|   |              |
|---|--------------|
| <b>Sources of Model Complexity .....</b>                              | <b>14-2</b>  |
| <b>Analyze a Large Model .....</b>                                    | <b>14-3</b>  |
| Types of Large Model Problems .....                                   | 14-3         |
| Summarize Model Hierarchy and Compatibility .....                     | 14-3         |
| Use the Default Parameter Values .....                                | 14-4         |
| Modify the Analysis Parameters .....                                  | 14-5         |
| Stop the Analysis Before Completion .....                             | 14-5         |
| <b>Increase Allocated Memory for Analysis Report Generation .....</b> | <b>14-7</b>  |
| <b>Manage Model Data to Simplify the Analysis .....</b>               | <b>14-8</b>  |
| Simplify Data Types .....   | 14-8         |
| Constrain Data .....  | 14-8         |
| <b>Partition Model Inputs for Incremental Test Generation .....</b>   | <b>14-11</b> |
| <b>Bottom-Up Approach to Model Analysis .....</b>                     | <b>14-13</b> |
| Reuse of Analysis Results from Subsystems at the System level .....   | 14-13        |
| Limitations .....   | 14-14        |
| <b>Extract Subsystems for Analysis .....</b>                          | <b>14-15</b> |
| Overview of Subsystem Extraction .....                                | 14-15        |
| sldvextract Function .....  | 14-15        |
| Structure of the Extracted Model .....                                | 14-15        |
| Analyze Subsystems That Read from Global Data Storage .....           | 14-16        |
| Analyze Function-Call Subsystems .....                                | 14-17        |
| Analyze Global Simulink Function .....                                | 14-19        |
| <b>Logical Operations .....</b>                                       | <b>14-21</b> |
| <b>Analyzing Models with Large Verification State Space .....</b>     | <b>14-22</b> |
| <b>Counters and Timers .....</b>                                      | <b>14-23</b> |

|   |              |
|---|--------------|
| <b>Prove Properties in Large Models</b> .....                 | <b>14-24</b> |
| Find Property Violations While Designing Your Model .....     | <b>14-24</b> |
| Combine Proving Properties and Finding Proof Violations ..... | <b>14-24</b> |

# 15

## Simulink Design Verifier Configuration Parameters

|   |              |
|---|--------------|
| <b>Simulink Design Verifier Options</b> .....                           | <b>15-2</b>  |
| Options in Configuration Parameters Dialog Box .....                    | <b>15-2</b>  |
| Design Verification Options Objects .....                               | <b>15-2</b>  |
| Command-Line Parameters for Design Verification Options .....           | <b>15-2</b>  |
| <br>  |              |
| <b>Design Verifier Pane</b> .....                                       | <b>15-9</b>  |
| Design Verifier Pane Overview .....                                     | <b>15-10</b> |
| Mode .....  | <b>15-10</b> |
| Maximum analysis time .....   | <b>15-11</b> |
| Output folder .....   | <b>15-11</b> |
| Make output file names unique by adding a suffix .....                  | <b>15-12</b> |
| Check Model Compatibility .....   | <b>15-13</b> |
| Generate Tests/Detect Errors/Prove Properties .....                     | <b>15-13</b> |
| Rebuild model representation .....                                      | <b>15-13</b> |
| Automatic stubbing of unsupported blocks and functions .....            | <b>15-13</b> |
| Support S-Functions in the analysis .....                               | <b>15-14</b> |
| Use specified input minimum and maximum values .....                    | <b>15-15</b> |
| Run additional analysis to reduce instances of rational approximation . | <b>15-15</b> |
| Validate test cases or counterexamples with parallel computing .....    | <b>15-16</b> |
| Additional options for code analysis .....                              | <b>15-17</b> |
| Ignore objectives based on filter .....                                 | <b>15-17</b> |
| Filter file(s) .....  | <b>15-18</b> |
| Browse... .....   | <b>15-18</b> |
| <br>  |              |
| <b>Design Verifier Pane: Block Replacements</b> .....                   | <b>15-19</b> |
| Block Replacements Pane Overview .....                                  | <b>15-19</b> |
| Apply block replacements .....  | <b>15-19</b> |
| List of block replacement rules .....                                   | <b>15-20</b> |
| File path of the output model .....                                     | <b>15-20</b> |
| <br>  |              |
| <b>Design Verifier Pane: Parameters and Variants</b> .....              | <b>15-22</b> |
| Parameters Pane Overview .....  | <b>15-23</b> |
| Parameter configuration .....   | <b>15-23</b> |
| Enable .....  | <b>15-23</b> |
| Disable .....   | <b>15-23</b> |
| Clear .....   | <b>15-23</b> |
| Highlight in Model .....  | <b>15-24</b> |
| Use .....   | <b>15-24</b> |
| Name .....  | <b>15-24</b> |
| Constraint .....  | <b>15-25</b> |
| Value .....   | <b>15-25</b> |
| Min .....   | <b>15-26</b> |
| Max .....   | <b>15-26</b> |
| Model Element .....   | <b>15-26</b> |
| Find parameters .....   | <b>15-27</b> |

|  |              |
|--|--------------|
| Import .....   | 15-27        |
| Export .....   | 15-27        |
| Parameter configuration file .....   | 15-27        |
| Browse... .....  | 15-28        |
| Edit... .....  | 15-28        |
| Analyze all Startup Variants .....   | 15-28        |
| Launch Variant Manager... .....  | 15-29        |
| <b>Design Verifier Pane: Test Generation .....</b>   | <b>15-30</b> |
| Test Generation Pane Overview .....  | 15-31        |
| Test generation target .....   | 15-31        |
| Model coverage objectives .....  | 15-31        |
| Test conditions .....  | 15-32        |
| Test objectives .....  | 15-33        |
| Maximum test case steps .....  | 15-33        |
| Test suite optimization .....  | 15-34        |
| Include relational boundary objectives .....   | 15-35        |
| Floating point absolute tolerance .....  | 15-36        |
| Floating point relative tolerance .....  | 15-36        |
| Use strict propagation conditions .....  | 15-37        |
| Extend using existing coverage data .....  | 15-38        |
| Coverage data .....  | 15-38        |
| Browse .....   | 15-39        |
| Extend using existing test data .....  | 15-39        |
| Test data .....  | 15-39        |
| Browse .....   | 15-40        |
| Separate objectives satisfied with the existing tests/coverage data in the<br>report ..... | 15-40        |
| <b>Design Verifier Pane: Design Error Detection .....</b>                                  | <b>15-42</b> |
| Design Error Detection Pane Overview .....   | 15-43        |
| Dead logic (partial) .....   | 15-43        |
| Run exhaustive analysis .....  | 15-43        |
| Coverage objectives to be analyzed .....   | 15-44        |
| Out of bound array access .....  | 15-45        |
| Data store access violations .....   | 15-45        |
| Division by zero .....   | 15-46        |
| Integer overflow .....   | 15-46        |
| Non-finite and NaN floating-point values .....   | 15-47        |
| Subnormal floating-point values .....  | 15-47        |
| Specified minimum and maximum value violations .....                                       | 15-48        |
| Specified block input range violations .....   | 15-48        |
| Usage of rem and reciprocal operations - hisl_0002 .....                                   | 15-49        |
| Usage of Square Root operations - hisl_0003 .....  | 15-50        |
| Usage of log and log10 operations - hisl_0004 .....  | 15-50        |
| Usage of Reciprocal Square Roots blocks - hisl_0028 .....                                  | 15-51        |
| <b>Design Verifier Pane: Property Proving .....</b>  | <b>15-52</b> |
| Property Proving Pane Overview .....   | 15-52        |
| Assertion blocks .....   | 15-52        |
| Proof assumptions .....  | 15-53        |
| Strategy .....   | 15-53        |
| Maximum violation steps .....  | 15-54        |
| <b>Design Verifier Pane: Results .....</b>   | <b>15-56</b> |
| Results Pane Overview .....  | 15-56        |

|  |              |
|--|--------------|
| Data file name .....                                 | 15-57        |
| Include expected output values .....                 | 15-57        |
| Randomize data that do not affect the outcome .....  | 15-58        |
| Generate separate harness model after analysis ..... | 15-59        |
| Harness model file name .....                        | 15-59        |
| Reference input model in generated harness .....     | 15-60        |
| Harness source .....                                 | 15-61        |
| Test File Name .....                                 | 15-61        |
| Test Harness Name .....                              | 15-62        |
| <b>Design Verifier Pane: Report .....</b>            | <b>15-63</b> |
| Report Pane Overview .....                           | 15-63        |
| Generate report of the results .....                 | 15-63        |
| Generate additional report in PDF format .....       | 15-64        |
| Report file name .....                               | 15-64        |
| Include screen shots of properties .....             | 15-65        |
| Display report .....                                 | 15-66        |

## Verification and Validation

# 16

|  |              |
|--|--------------|
| <b>Test Model Against Requirements and Report Results .....</b>        | <b>16-2</b>  |
| Requirements - Test Traceability Overview .....                        | 16-2         |
| Display the Requirements .....   | 16-2         |
| Link Requirements to Tests .....                                       | 16-3         |
| Run the Test .....   | 16-4         |
| Report the Results .....   | 16-5         |
| <b>Analyze Models for Standards Compliance and Design Errors .....</b> | <b>16-7</b>  |
| Standards and Analysis Overview .....                                  | 16-7         |
| Check Model for Style Guideline Violations and Design Errors .....     | 16-7         |
| <b>Perform Functional Testing and Analyze Test Coverage .....</b>      | <b>16-9</b>  |
| Incrementally Increase Test Coverage Using Test Case Generation .....  | 16-9         |
| <b>Analyze Code and Test Software-in-the-Loop .....</b>                | <b>16-12</b> |
| Code Analysis and Testing Software-in-the-Loop Overview .....          | 16-12        |
| Analyze Code for Defects, Metrics, and MISRA C:2012 .....              | 16-12        |
| Test Code Against Model Using Software-in-the-Loop Testing .....       | 16-17        |
| <b>Create Back-to-Back Tests Using Enhanced MCDC .....</b>             | <b>16-20</b> |
| Set Up Test Inputs and Verification Strategy .....                     | 16-20        |

## Glossary

# Acknowledgments

The Simulink Design Verifier software uses Prover Plug-In<sup>®</sup> product Prover<sup>®</sup> PSL from Prover Technology to generate test cases and prove model properties.





# Getting Started

---

- “Simulink Design Verifier Product Description” on page 1-2
- “Simulink Design Verifier Block Library” on page 1-3
- “Analyze a Model” on page 1-4
- “Analyze a Stateflow Atomic Subchart” on page 1-17
- “Overview of the Simulink Design Verifier Workflow” on page 1-19

## **Simulink Design Verifier Product Description**

### **Identify design errors, prove requirements compliance, and generate tests**

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

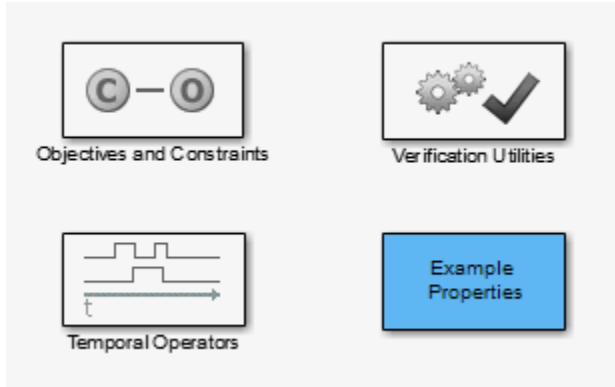
Simulink Design Verifier generates test cases for model coverage and custom objectives to extend existing requirements-based test cases. These test cases drive your model to satisfy condition, decision, modified condition/decision (MCDC), and custom coverage objectives. In addition to coverage objectives,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

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



## Simulink Design Verifier Block Library

To open the Simulink Design Verifier block library, at the MATLAB® command prompt, type `sldvlib`.



The Simulink Design Verifier block library has three categories of blocks:

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

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

## Analyze a Model

### In this section...

“About This Example” on page 1-4

“Open the Model” on page 1-4

“Generate Test Cases” on page 1-5

“Combine Test Cases” on page 1-15

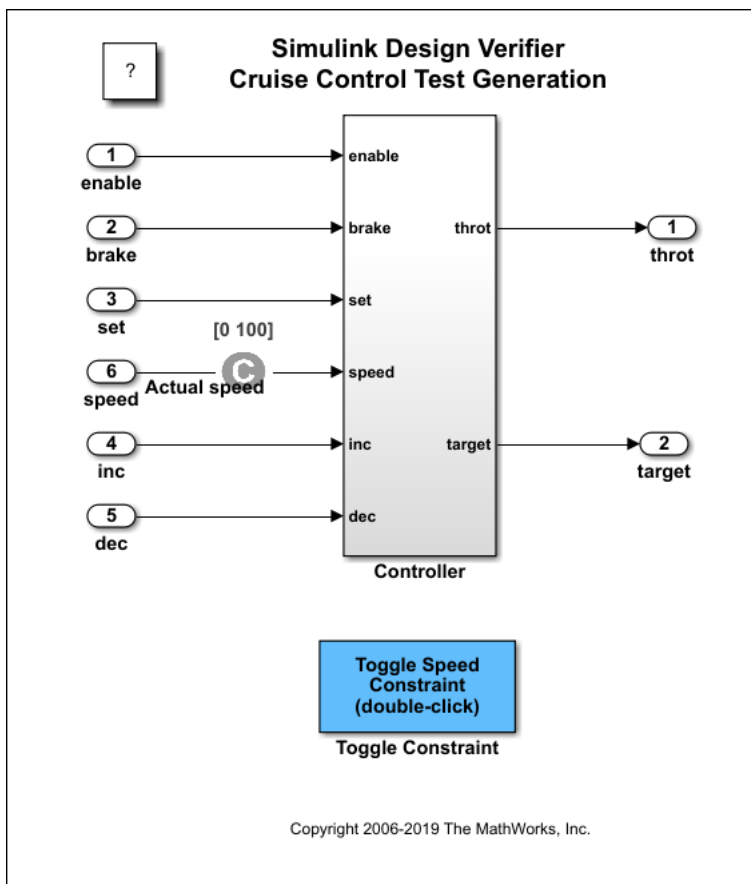
### About This Example

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

### Open the Model

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

```
sldvdemo_cruise_control
```



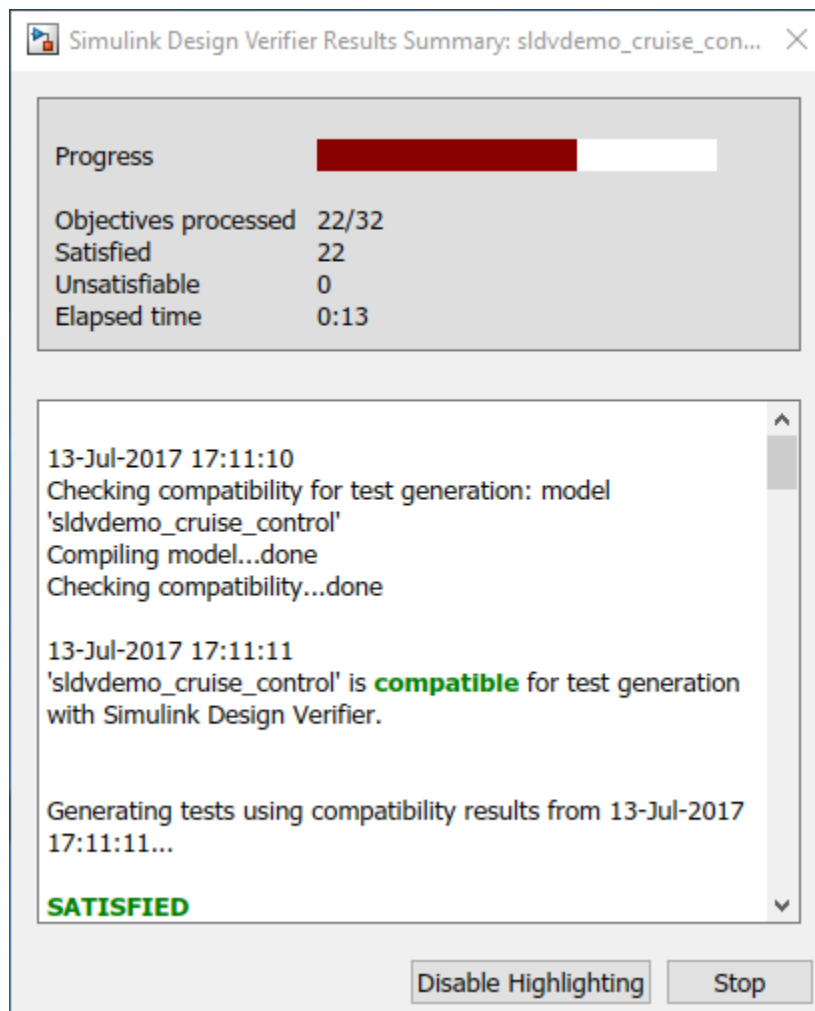
## Generate Test Cases

- “Run Analysis” on page 1-5
- “Generate Analysis Results” on page 1-6
- “Highlight Analysis Results on Model” on page 1-7
- “Detailed analysis report: (HTML) (PDF)” on page 1-8
- “Create Harness Model” on page 1-12
- “Simulate Tests and Produce Model Coverage Report” on page 1-15

### Run Analysis

To generate test cases for the Cruise Control Test Generation model, click on **Generate Tests**.

Simulink Design Verifier begins analyzing the model to generate test cases, and the Simulink Design Verifier Results Summary window opens. The Results Summary window displays a running log showing the progress of the analysis.



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

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

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

## Generate Analysis Results

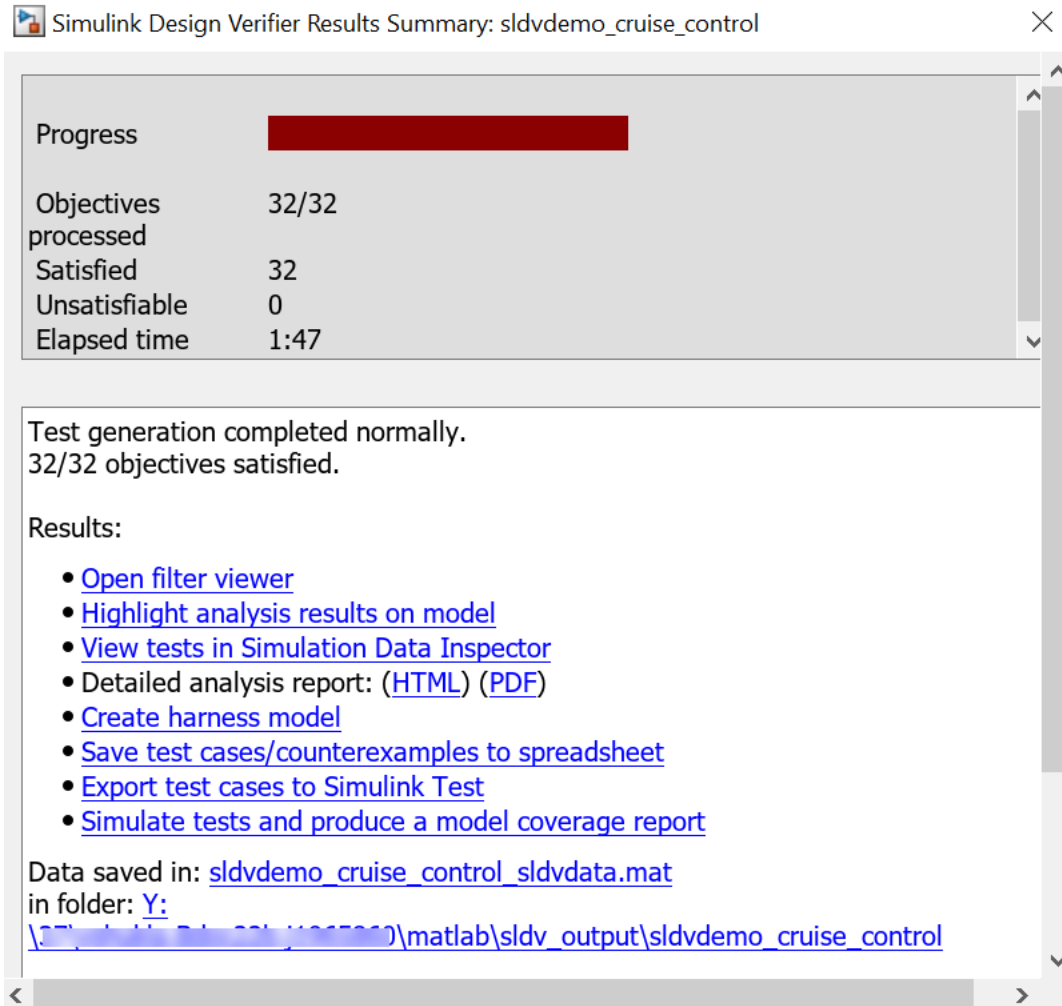
When Simulink Design Verifier completes its analysis of the `sldvdemo_cruise_control` model, the Results Summary window displays several options. Some of them are:

- **Highlight analysis results on model**
- **Detailed analysis report: (HTML) (PDF)**
- **Create harness model**
- **Simulate tests and produce a model coverage report**
- **Save test cases/counterexamples to spreadsheet**

---

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

---



The sections that follow describe these options in detail.

### Highlight Analysis Results on Model

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

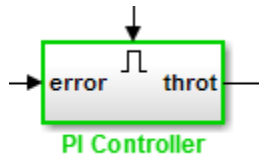
- “Green: Objectives Satisfied” on page 1-8
- “Orange: Objectives Undecided” on page 1-8
- “Red: Objectives Unsatisfiable” on page 1-8

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

### Green: Objectives Satisfied

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

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



### Orange: Objectives Undecided

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

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

### Red: Objectives Unsatisfiable

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

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

### Detailed analysis report: (HTML) (PDF)

In the Simulink Design Verifier Results Summary window, if you click **HTML** on **Detailed analysis report: (HTML) (PDF)**, 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 1-9
- “Analysis Information” on page 1-10
- “Test Objectives Status” on page 1-10
- “Model Items” on page 1-11
- “Test Cases” on page 1-11

### Summary

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

- Name of the model
- Release and Checksum information
- Mode of the analysis (test generation, property proving, design error detection)
- Status of the analysis
- Length of the analysis in seconds

The **Objective Status** sub-section under **Summary** shows number of objectives satisfied.

#### Analysis Information

```
Model:          sldvdemo_cruise_control
Release:        R2022b Prerelease
Checksum:       3016843220 2232669898 18135123 2081307571
Mode:           Test generation
Model Representation: Built on 25-Jun-2022 22:19:16
Test Generation Target: Model
Status:         Completed normally
PreProcessing Time: 106s
Analysis Time:  108s
```

#### Objectives Status

```
Number of Objectives: 32
Objectives Satisfied: 32 ( 100%)
```

## Analysis Information

In the **Table of Contents**, click **Analysis Information** to display information about the analyzed model and the analysis options. You can click on any of these options to know more about the model analysis.

## Chapter 2. Analysis Information

### Table of Contents

- [2.1. Model Information](#)
- [2.2. Analysis Options](#)
- [2.3. User Artifacts](#)
- [2.4. Constraints](#)

## Test Objectives Status

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

| #  | Type      | Model Item                                   | Description   | Analysis Time (sec) | Test Case         |
|----|-----------|--|---|---------------------|-------------------|
| 1  | Decision  | <a href="#">Controller/Switch3</a>           | logical trigger input <b>false (output is from 3rd input port)</b>    | 38                  | <a href="#">1</a> |
| 2  | Decision  | <a href="#">Controller/Switch3</a>           | logical trigger input <b>true (output is from 1st input port)</b>     | 38                  | <a href="#">1</a> |
| 3  | Decision  | <a href="#">Controller/Switch2</a>           | logical trigger input <b>false (output is from 3rd input port)</b>    | 38                  | <a href="#">1</a> |
| 4  | Decision  | <a href="#">Controller/Switch2</a>           | logical trigger input <b>true (output is from 1st input port)</b>     | 38                  | <a href="#">1</a> |
| 5  | Decision  | <a href="#">Controller/Switch1</a>           | logical trigger input <b>false (output is from 3rd input port)</b>    | 38                  | <a href="#">1</a> |
| 6  | Decision  | <a href="#">Controller/Switch1</a>           | logical trigger input <b>true (output is from 1st input port)</b>     | 38                  | <a href="#">1</a> |
| 7  | Condition | <a href="#">Controller/Logical Operator1</a> | Logic: input port 1 <b>true</b>                                       | 38                  | <a href="#">1</a> |
| 8  | Condition | <a href="#">Controller/Logical Operator1</a> | Logic: input port 1 <b>false</b>                                      | 38                  | <a href="#">1</a> |
| 9  | Condition | <a href="#">Controller/Logical Operator2</a> | Logic: input port 1 <b>true</b>                                       | 38                  | <a href="#">1</a> |
| 10 | Condition | <a href="#">Controller/Logical Operator2</a> | Logic: input port 1 <b>false</b>                                      | 38                  | <a href="#">1</a> |
| 11 | Condition | <a href="#">Controller/Logical Operator2</a> | Logic: input port 2 <b>true</b>                                       | 101                 | <a href="#">2</a> |
| 12 | Condition | <a href="#">Controller/Logical Operator2</a> | Logic: input port 2 <b>false</b>                                      | 38                  | <a href="#">1</a> |
| 13 | Condition | <a href="#">Controller/Logical Operator</a>  | Logic: input port 1 <b>true</b>                                       | 38                  | <a href="#">1</a> |
| 14 | Condition | <a href="#">Controller/Logical Operator</a>  | Logic: input port 1 <b>false</b>                                      | 38                  | <a href="#">1</a> |
| 15 | Condition | <a href="#">Controller/Logical Operator</a>  | Logic: input port 2 <b>true</b>                                       | 38                  | <a href="#">1</a> |
| 16 | Condition | <a href="#">Controller/Logical Operator</a>  | Logic: input port 2 <b>false</b>                                      | 38                  | <a href="#">1</a> |
| 17 | Condition | <a href="#">Controller/Logical Operator</a>  | Logic: input port 3 <b>true</b>                                       | 38                  | <a href="#">1</a> |
| 18 | Condition | <a href="#">Controller/Logical Operator</a>  | Logic: input port 3 <b>false</b>                                      | 38                  | <a href="#">1</a> |
| 19 | MCDC      | <a href="#">Controller/Logical Operator</a>  | (C1 && ~C2) && (C3    C4) with C1 (Logical Operator In1) <b>true</b>  | 38                  | <a href="#">1</a> |
| 20 | MCDC      | <a href="#">Controller/Logical Operator</a>  | (C1 && ~C2) && (C3    C4) with C1 (Logical Operator In1) <b>false</b> | 38                  | <a href="#">1</a> |

## Objectives Status

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

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

In the row for objective 32, click the test case number (**5**) to display more information about Test Case 5 in the report's **Test Cases** chapter.



**Summary**

Length: 0.06 second (7 sample periods)

Objectives Satisfied: 1

**Objectives**

| Step | Time | Model Item  | Objectives  |
|------|------|---|---|
| 7    | 0.06 | <a href="#">Controller/PI Controller/Discrete-Time Integrator</a> | <a href="#">31. integration result &gt;= upper limit true</a> |

**Generated Input Data**

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

**Test Case 5**

In this example, Test Case 5 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.

[View](#)

| #: | Type     | Description  | Status    | Test Case         |
|----|----------|--|-----------|-------------------|
| 1  | Decision | <a href="#">4.1. Controller/Switch3</a><br>logical trigger input false (output is from 3rd input port) | Satisfied | <a href="#">1</a> |
| 2  | Decision | logical trigger input true (output is from 1st input port)   | Satisfied | <a href="#">1</a> |

**Model Items - Controller/Switch3**[View](#)

| #: | Type     | Description   | Status    | Test Case         |
|----|----------|---|-----------|-------------------|
| 3  | Decision | logical trigger input false (output is from 3rd input port) | Satisfied | <a href="#">1</a> |
| 4  | Decision | logical trigger input true (output is from 1st input port)  | Satisfied | <a href="#">1</a> |

**Model Items - Controller/Switch2****Test Cases**

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

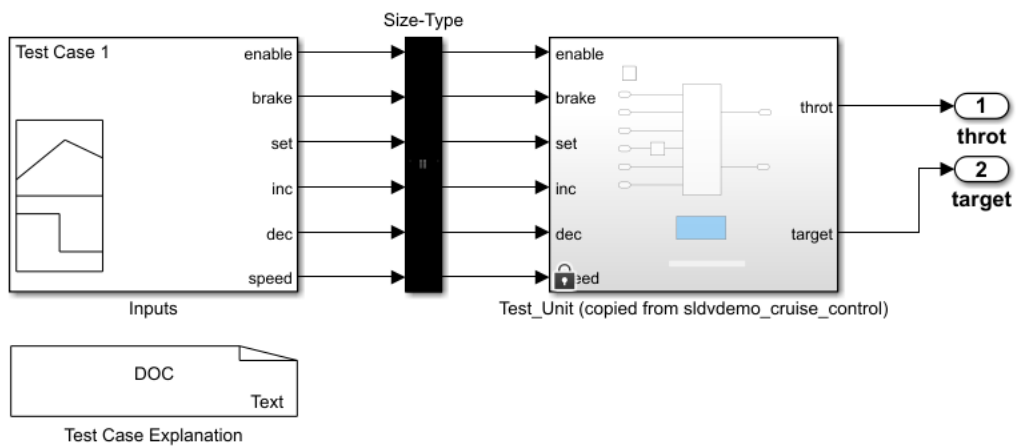
- Length of time to execute the test case
- Number of objectives satisfied
- Detailed information about the satisfied objectives

- Input data

For an example, see the section for Test Case 5 in “Test Objectives Status” on page 1-10.

## Create Harness Model

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



The harness model contains the following blocks:

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

```

tp7469ce30_944a_4887_840e_c8d2d392a9ee.txt +
22 19. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C1 (Logical Operator In1) false @ T=0.03
23 20. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C2 (Logical Operator1 In1) true @ T=0.04
24 21. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C2 (Logical Operator1 In1) false @ T=0.05
25 22. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C3 (Logical Operator2 In1) true @ T=0.04
26 23. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C3 (Logical Operator2 In1) false @ T=0.01
27 24. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C4 (Logical Operator2 In2) false @ T=0.01
28 25. Controller/PI Controller - Enable control activated true @ T=0.04
29 26. Controller/PI Controller - Enable control activated false @ T=0.00
30 27. Controller/PI Controller/Discrete-Time Integrator - integration result <= lower limit false @ T=0.04
31 28. Controller/PI Controller/Discrete-Time Integrator - integration result >= upper limit false @ T=0.04
32
33 Test Case 2 (1 Objectives)
34 Parameter values:
35
36 1. Controller/Logical Operator2 - Logic: input port 2 true @ T=0.01
37
38 Test Case 3 (1 Objectives)
39 Parameter values:
40
41 1. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C4 (Logical Operator2 In2) true @ T=0.01
42
43 Test Case 4 (1 Objectives)
44 Parameter values:
45
46 1. Controller/PI Controller/Discrete-Time Integrator - integration result <= lower limit true @ T=0.06
47
48 Test Case 5 (1 Objectives)
49 Parameter values:
50
51 1. Controller/PI Controller/Discrete-Time Integrator - integration result >= upper limit true @ T=0.06

```

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

---

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

---

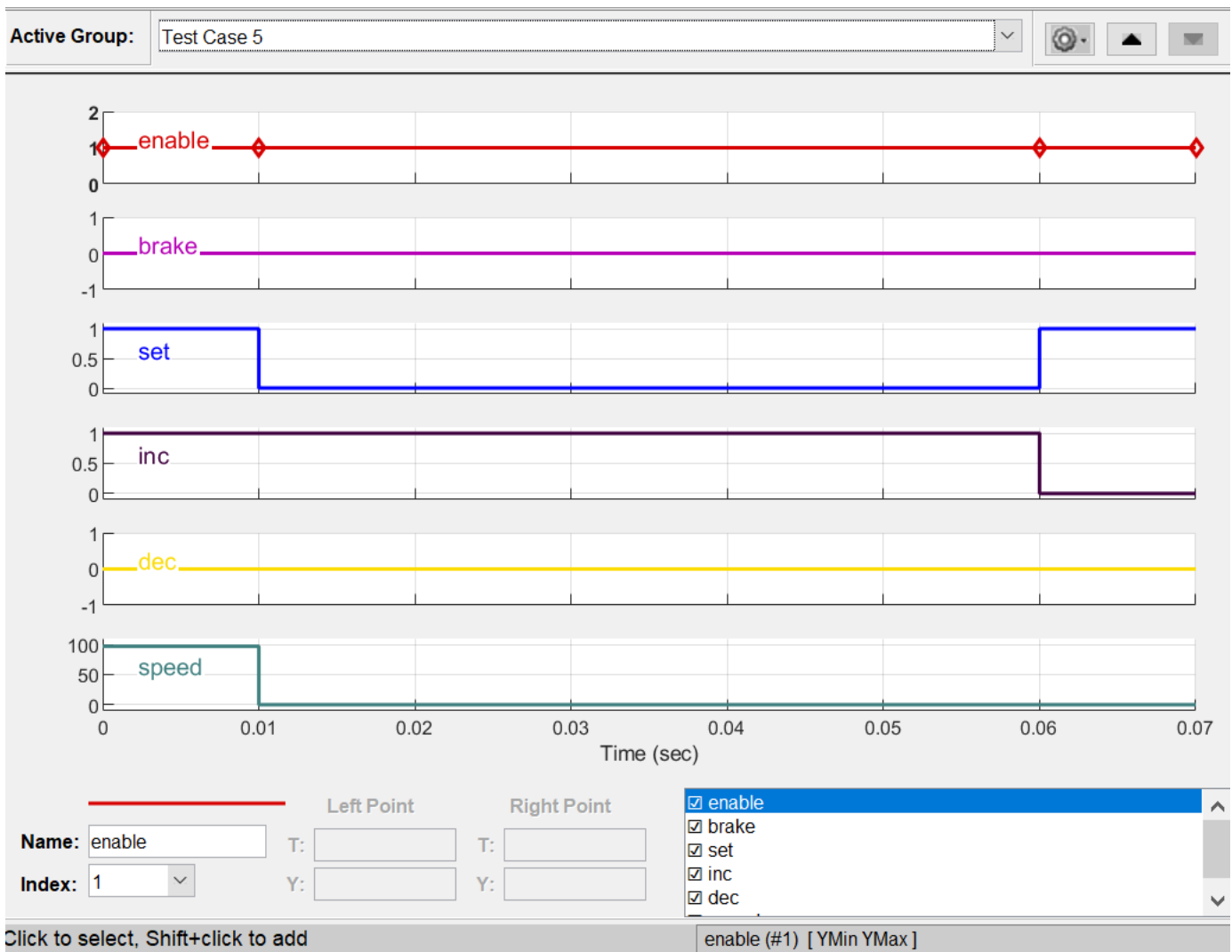
- The Inputs block is a Signal Builder block that contains the generated test case signals. Double-click the Inputs block to open the Signal Builder dialog box and view the eight test case signals.
- The Size-Type block is a subsystem that transmits signals from the Inputs block to the Test Unit block. This block verifies that the size and data type of the signals are consistent with the Test Unit block.

The Signal Builder dialog box contains eight test cases.


- 1 To view Test Case 5, from the **Active Group** list, select Test Case 5.

In Test Case 7 at 0.01 seconds:

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













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

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

The Simulink software simulates all the test cases. The Simulink Coverage™ software collects coverage data for the harness model and displays a coverage report. The report summary shows that the sldvdemo\_cruise\_control\_harness model achieves 100% coverage.

**Model Hierarchy/Complexity:**

|  |   | <b>D1</b>   | <b>C1</b>  | <b>MCDC</b>  |
|--|---|---|--|--|
| 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   |

**Summary****Simulate Tests and Produce Model Coverage Report**

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

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

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

- The harness model coverage report might contain additional time steps. When you collect coverage for the harness model, the model stop time equals the stop time for the longest test case. As a result, you might achieve additional coverage when you simulate the shorter test cases.
- The cyclomatic complexity coverage for the Test Unit subsystem in the harness model might be different than the coverage for the model itself due to the structure of the harness model.

**Combine Test Cases**

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

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

- 1 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
- 2 In the Configuration Parameters dialog box, in the **Select** tree on the left side, under the **Design Verifier** category, select **Test Generation**.
- 3 Set the **Test suite optimization** parameter to `LongTestcases`.
- 4 Click **Apply** and **OK** to close the Configuration Parameters dialog box.
- 5 In the `sldvdemo_cruise_control` model, double-click the block labeled **Run**.
- 6 In the Results Summary window, click **Create harness model**.

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

The screenshot shows an IDE window titled "Editor - S:\sca\_sldv\sldvdemo\_cruise\_control\_harness\_testcase\_long.txt". The window contains a list of 34 test case objectives, numbered 1 through 37. The objectives are as follows:

```

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

```

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

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

The analysis still satisfies all 34 objectives.

## Analyze a Stateflow Atomic Subchart

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

If you are having problems analyzing a large model, analyzing an atomic subchart in a controlled environment is helpful. As described in “Bottom-Up Approach to Model Analysis” on page 14-13, by analyzing atomic subcharts or other components in the model hierarchy individually, you can analyze a model to:

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

---

**Note** For more information about atomic subcharts, see “Create Reusable Subcomponents by Using Atomic Subcharts” (Stateflow).

---

### Analyze an Atomic Subchart by Using Simulink Design Verifier

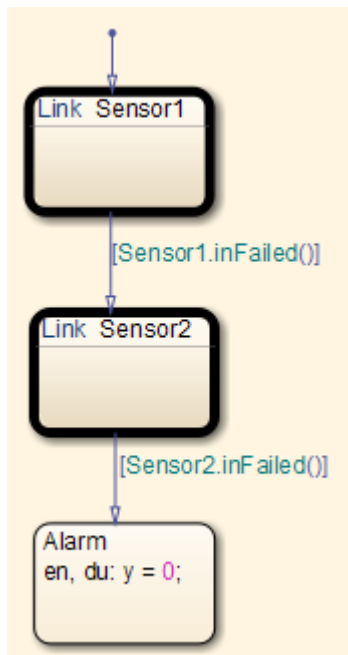
The `sf_atomic_sensor_pair` example model models a redundant sensor pair using atomic subcharts. This example analyzes the `Sensor1` subchart in the `RedundantSensors` chart.

- 1 Open the `sf_atomic_sensor_pair` example model:

```
sf_atomic_sensor_pair
```

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

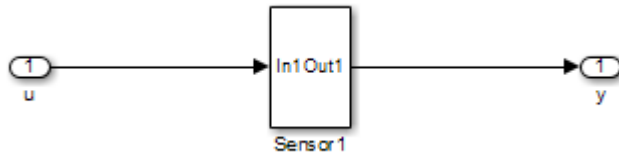
- 2 Double-click the `RedundantSensors` chart to open it.



This Stateflow chart has two atomic subcharts:

- Sensor1
  - Sensor2
- 3** To analyze the Sensor1 subchart using Simulink Design Verifier, right-click the subchart and select **Design Verifier > Generate Tests for Subchart**.

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



The software saves the new model and other files generated by the analysis in:

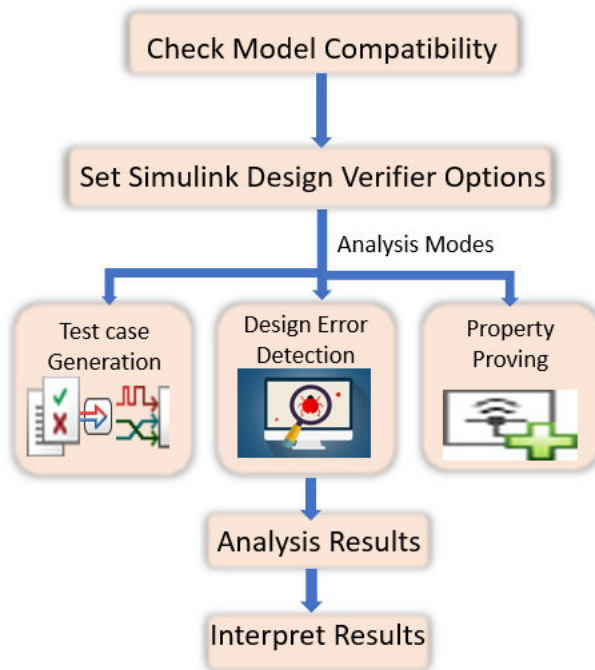
`<current_folder>/sldv_output/Sensor1`

- 4** When the analysis is complete, view the analysis results for the `Sensor1` subchart by clicking one of the following options:
- **Highlight analysis results on model**
  - **Generate detailed analysis report**
  - **Create harness model**
  - **Simulate tests and produce a model coverage report**



## Overview of the Simulink Design Verifier Workflow

Before you analyze a model for design error detection, test case generation, and property proving, you must complete a few as shown in this diagram:



The following sections provide a brief overview of the Simulink Design Verifier workflow and include with links to related documentation in Simulink Design Verifier.

### Check Model Compatibility

Before Simulink Design Verifier analyzes a model, the software checks whether the model is compatible for analysis. For more information on model compatibility, see “Check Model Compatibility” on page 3-2. The software runs a compatibility check on your model, and then creates a model representation. The model representation includes the model artifacts that you can use during analysis. The compatibility check tells you if your model is fully compatible, partially compatible, or not compatible.

Simulink supports a broad range of and software capabilities in your models but there are some capabilities that Simulink Design Verifier does not support. For more information, see “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” on page 3-7 and “Support Limitations for Simulink Software Features” on page 3-16.

### Apply Block Replacement Rules

If you want to work around the compatibility limitations in your model or customize model elements for analysis, you can use the Simulink Design Verifier block replacement rules. For more information,

see “What Is Block Replacement?” on page 4-2 and “Block Replacements for Unsupported Blocks” on page 4-7.

If you want to generate additional values for parameters in your model during analysis, use Simulink Design Verifier parameter configurations. See “Parameter Configuration for Analysis” on page 5-2 for more information.

## Set Simulink Design Verifier Options

You can set the Simulink Design Verifier analysis options in the Configuration Parameters dialog box. Alternatively, you can use the `sldvoptions` function to specify the Simulink Design Verifier options at the command line. For more information, see “Simulink Design Verifier Options” on page 15-2.

## Perform Analysis on Model

You can analyze your model for:

- Design Error Detection: Detect design errors that can occur at run time. For more information, see “Analyze Models for Design Errors” on page 6-4.
- Test Case Generation: Generate test cases that achieve model coverage. For more information, see “Workflow for Test Case Generation” on page 7-5
- Property Proving Analysis: Prove properties and identify property violations. For more information, see “Workflow for Proving Model Properties” on page 12-4.

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. Refer to these topics for more information:

- “What Is Design Error Detection?” on page 6-2
- “Detect Integer Overflow and Division-by-Zero Errors” on page 6-19
- “Debug Integer Overflow Design Error Detection Using Model Slicer” on page 6-68

## Generate Analysis Results

Once Simulink Design Verifier finishes analyzing the model, it displays the analysis highlights and the results options in the Results Summary window. For more information, see “Generate Analysis Results” on page 1-6.

## Interpret Analysis Results

You can review analysis results and generate analysis reports in the HTML, DOCX, or PDF format. For more information, see “Review Analysis Results”

## See Also

### More About

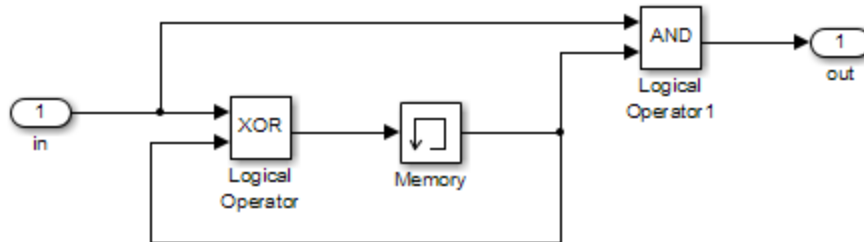
- Systematic Model Verification using Simulink Design Verifier
- “Analyze a Model” on page 1-4

# How the Simulink Design Verifier Software Works

---

- “Analyze a Simple Model” on page 2-2
- “Model Blocks” on page 2-4
- “Block Reduction” on page 2-5
- “Large Models” on page 2-6
- “Handle Incompatibilities with Automatic Stubbing” on page 2-7
- “Analyze Export-Function Models” on page 2-12
- “Analyze Export-Function Model with Function-Call Subsystems” on page 2-13
- “Analyze Export-Function Model with Global Simulink Function” on page 2-16
- “Nonfinite Data” on page 2-19
- “Role of Approximations During Model Analysis” on page 2-20
- “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23
- “Logic Operations Short-Circuiting” on page 2-26
- “Model Representation for Analysis” on page 2-28
- “Share Simulink Cache File for Faster Analysis” on page 2-31
- “Analyze AUTOSAR Component Models” on page 2-33
- “Extend Existing Test Cases by Reusing Model Representation” on page 2-35
- “Configure Model Representation Options” on page 2-39
- “Run Additional Analysis to Reduce Instances of Rational Approximation” on page 2-42
- “Detect Design Errors in AUTOSAR Software Component Model” on page 2-47

## Analyze a Simple Model



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

| # | Input | Memory Value | Output of XOR Block =<br>Next Memory Value | Output of AND Block |
|---|-------|--------------|--|---------------------|
| 1 | false | false        | false                                      | false               |
| 2 | true  | false        | true                                       | false               |
| 3 | false | true         | true                                       | false               |
| 4 | true  | true         | false                                      | true                |

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

- If the initial memory value is `true`, the test case is a single time step where the input is `true`.
- If the initial memory value is `false`, the test case is two time steps:
  - 1 The input value is `true` and the memory value is `false` (row 2). Thus, the output of the XOR block is `true`, making the memory value `true`.
  - 2 Now that the input value and memory value are both `true` (row 4), the output is `true`, and the analysis achieves the test objective.

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

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

During a property proving analysis, you specify a functional requirement, or property, that you want Simulink Design Verifier to prove, for example, that the output is always `true`. If the search completes

without finding a path that violates the property, the property is proven. If the software finds a path where the output is false, it creates a counterexample that causes the output to be false.

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

## Model Blocks

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

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

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

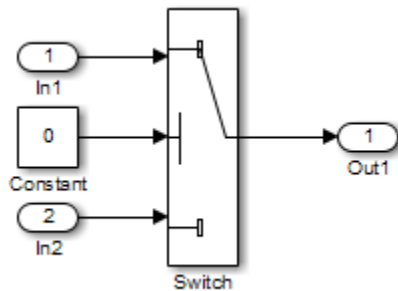
For example, consider a top-level model with three Model blocks referencing the same model. The referenced model has three test objectives. Analyzing the top-level model produces nine test objectives. If you simulate the model with the nine test cases, the coverage results for that referenced model specify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

## Block Reduction

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

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

Consider the Switch block in the following model.



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

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

## Large Models

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

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

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

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



## Handle Incompatibilities with Automatic Stubbing

### In this section...

“What Is Automatic Stubbing?” on page 2-7

“How Automatic Stubbing Works” on page 2-7

“Analyze a Model Using Automatic Stubbing” on page 2-9

### What Is Automatic Stubbing?

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

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

### How Automatic Stubbing Works

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

#### Stub Trigonometric Function Block

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



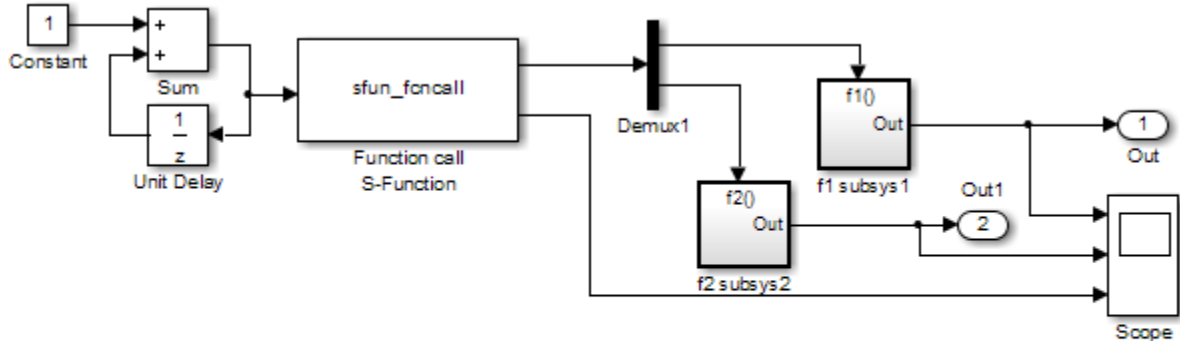
When stubbing this block during analysis, `out_signal` can take any value, with the following results.

| Analysis Model         | Result of Stubbing <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> |

| Analysis Model       | Result of Stubbing out_signal  |
|----------------------|--|
| 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>                                |

### Stub S-Function Block Containing Function-Call Triggers

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



```
matlabroot\toolbox\simulink\simdemos\simfeatures\src\sfun_fncall.c
```

If you do not enable support for an S-function in Simulink Design Verifier and automatic stubbing is enabled, the analysis ignores the behavior of the S-function. As a result, the code that triggers the two function-call subsystems is ignored, resulting in two unsatisfiable objectives. Since the function calls are ignored, the contents of those subsystems are effectively eliminated from the analysis.

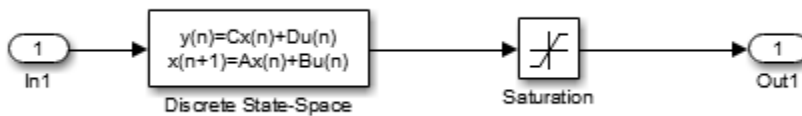
To enable support for an S-function in Simulink Design Verifier, see “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-28

## Analyze a Model Using Automatic Stubbing

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

- “Check Model Compatibility” on page 2-9
- “Turn On Automatic Stubbing” on page 2-10
- “Review Results” on page 2-11
- “Achieve Complete Results” on page 2-11

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

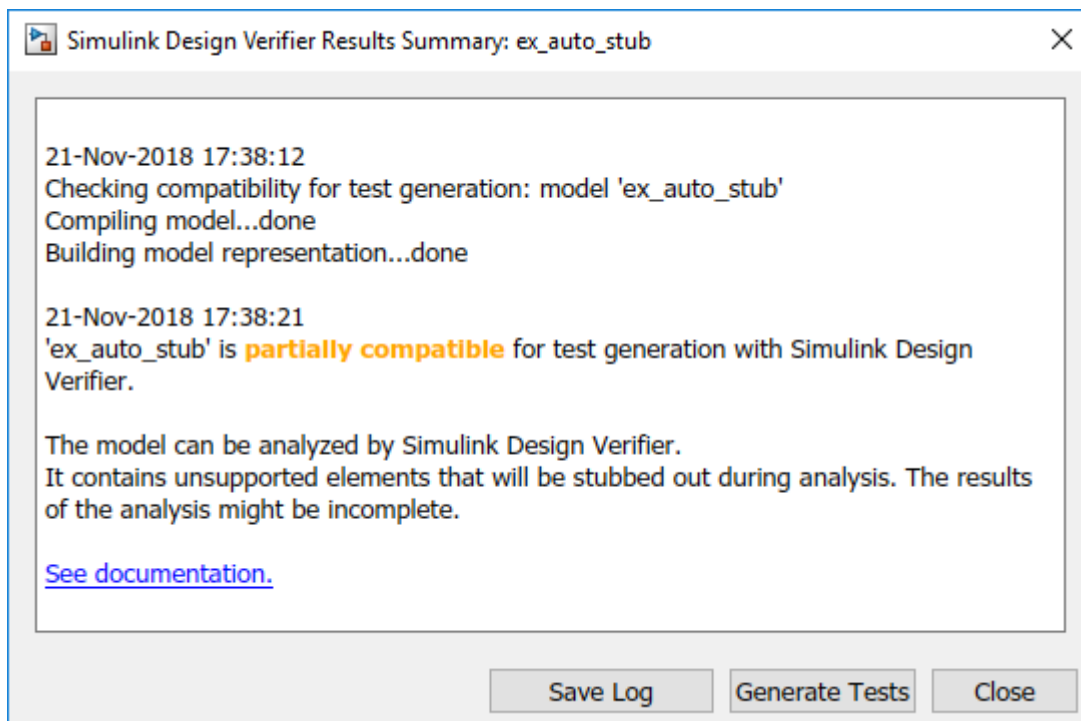


### Check Model Compatibility

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

To run the Simulink Design Verifier compatibility check:

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



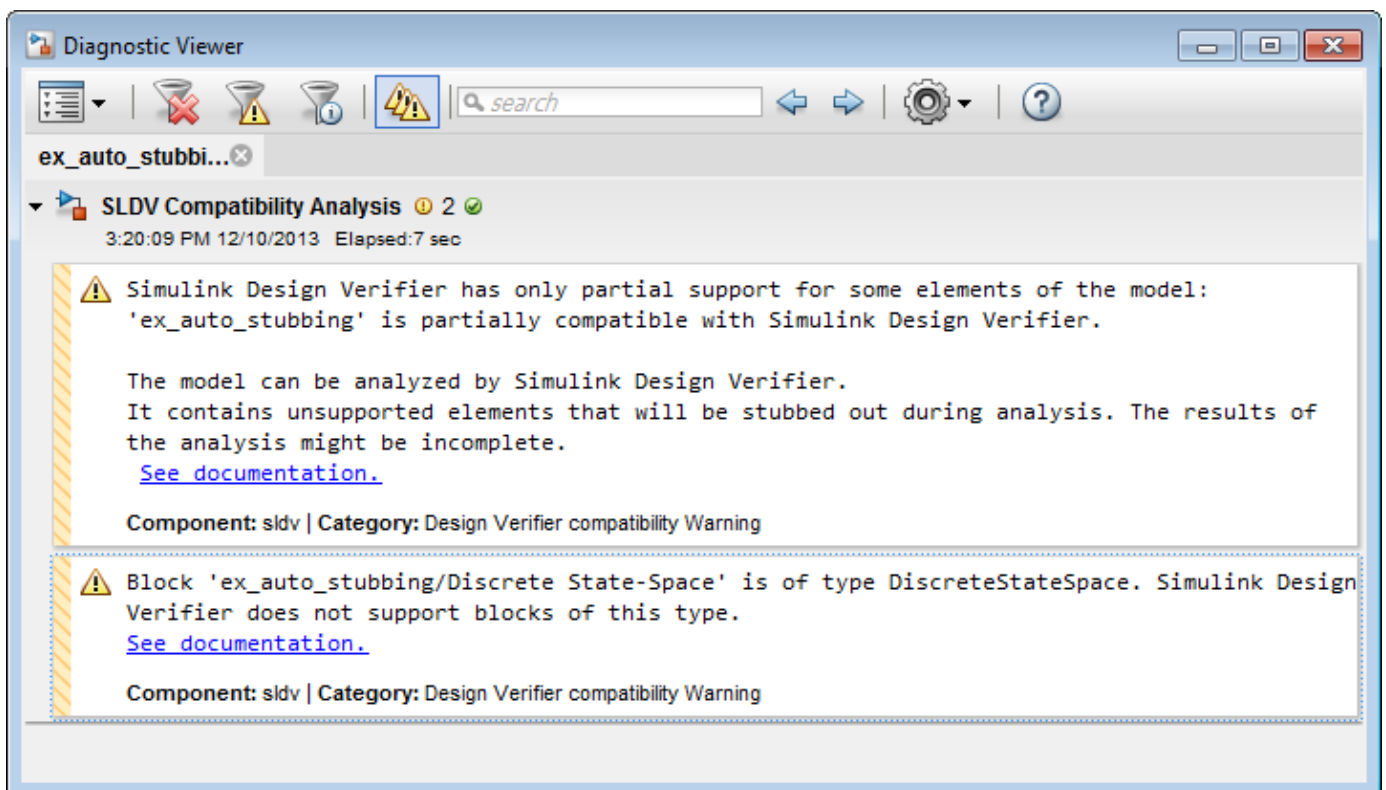
- Select the analysis that you want to perform.

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

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

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

If it finds incompatible elements in the model, the software analyzes the model and, by default, stubs out the incompatible elements. The Diagnostic Viewer also opens, listing the incompatibilities.



---

**Note** For more information, see “View Diagnostics”.

---

### Turn On Automatic Stubbing

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

## Review Results

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

| Block                                | Type               |
|--------------------------------------|--------------------|
| <a href="#">Discrete State-Space</a> | DiscreteStateSpace |

## Unsupported Blocks

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

| # | Type     | Model Item                 | Description            | Analysis Time (sec) |
|---|----------|----------------------------|------------------------|---------------------|
| 2 | Decision | <a href="#">Saturation</a> | input > lower limit F  | 12                  |
| 3 | Decision | <a href="#">Saturation</a> | input > lower limit T  | 12                  |
| 4 | Decision | <a href="#">Saturation</a> | input >= upper limit F | 12                  |
| 5 | Decision | <a href="#">Saturation</a> | input >= upper limit T | 12                  |

## Objective Undecided Due to Stubbing

### Achieve Complete Results

If your analysis does not achieve complete results because of the stubbing, you can define custom block replacements to give a more precise definition of the unsupported blocks. For more information, follow the steps in “Block Replacements for Unsupported Blocks” on page 4-7.

# Analyze Export-Function Models

Simulink Design Verifier supports design error detection, test generation, and property proving for export-function models. The software creates a scheduler model that invokes the export-function models, and then performs the analysis on the scheduler model. The scheduler model invokes the function calls based on the sample times and priorities set in the top model. By default, the software saves the scheduler model in `<current_folder>\sldv_output\<>model_name>\<model_name>_SldvScheduler.slx`. You can analyze export-function models with periodic and aperiodic function-call groups. If the model consists of aperiodic function-call or global Simulink Function call, the scheduler has an additional port called the `FcnTriggerPort`. For more information, see “Export-Function Models Overview”.

These topics cover examples that explain a periodic function-call subsystem and global Simulink Function that you can use as an AUTOSAR server runnable.

- “Analyze Export-Function Model with Function-Call Subsystems” on page 2-13
- “Analyze Export-Function Model with Global Simulink Function” on page 2-16

## Limitations

Simulink Design Verifier does not support:

- Models that include export functions with multiple function-call initiators.
- Masked model blocks that export Simulink Function blocks.
- Scoped Simulink functions in export-function models.

## See Also

### More About

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

## Analyze Export-Function Model with Function-Call Subsystems

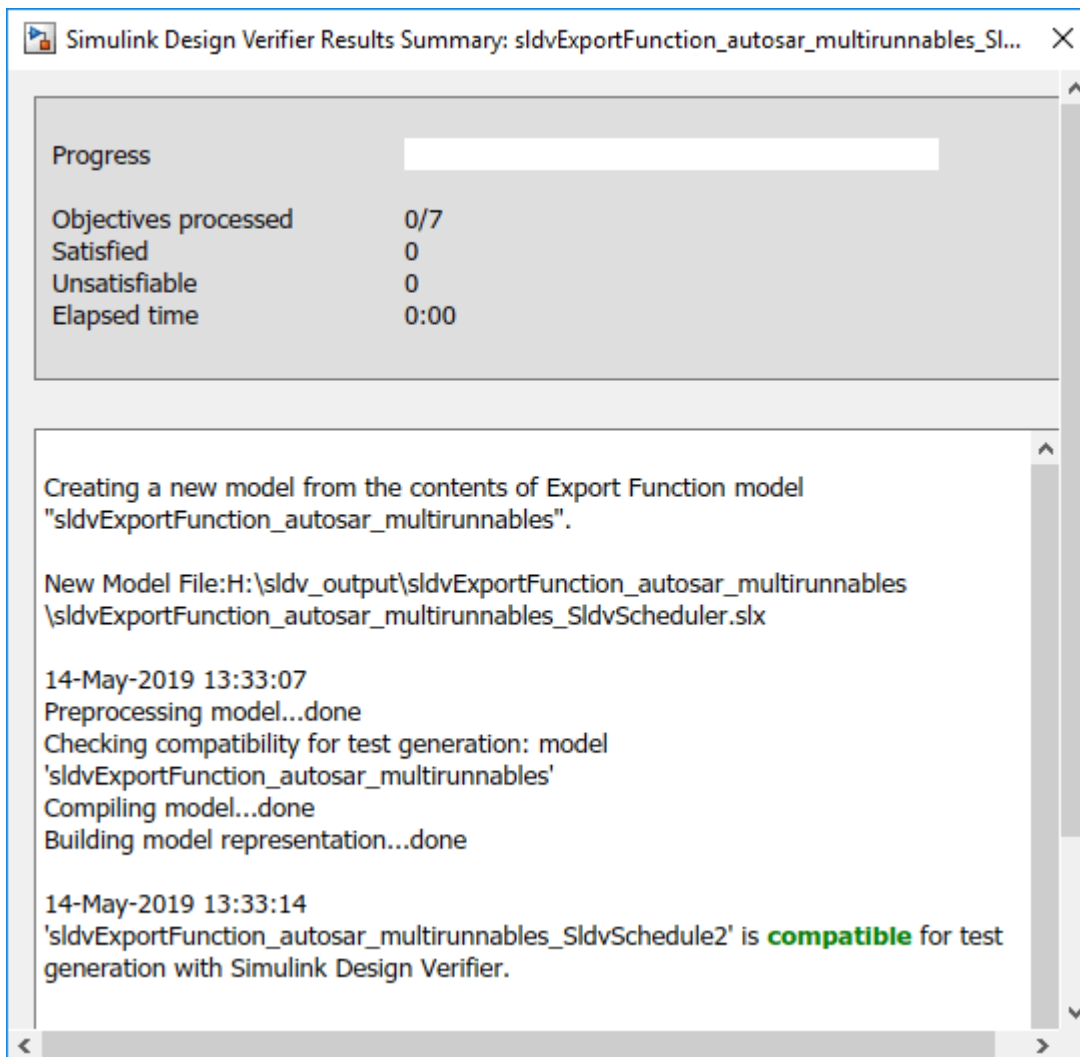
This example shows how you can analyze a model which consists of periodic function-call subsystems. This example uses the AUTOSAR example model `sldvExportFunction_autosar_multirunnables`.

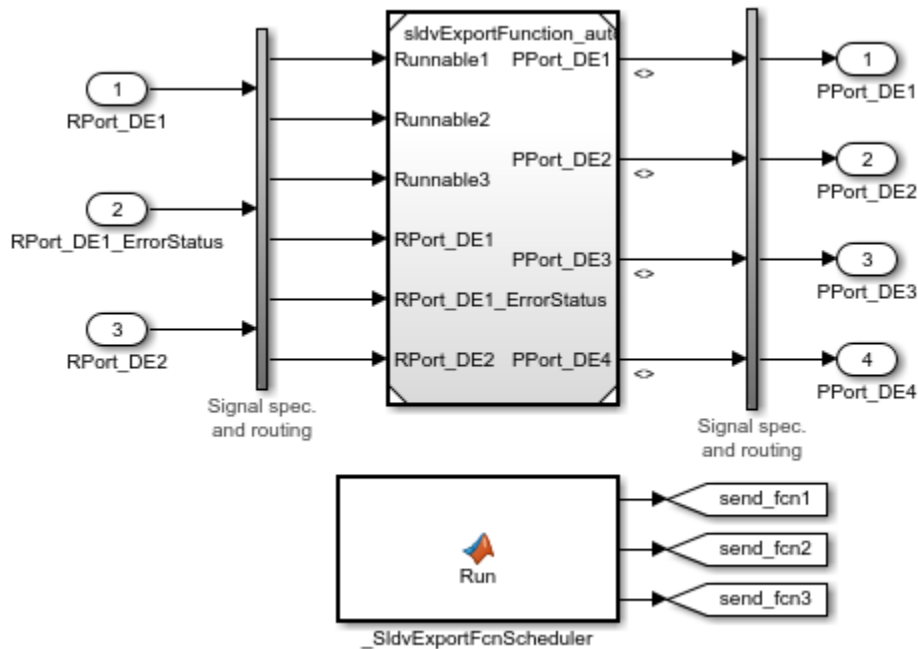
1. Open the `sldvExportFunction_autosar_multirunnables` model.

```
open_system('sldvExportFunction_autosar_multirunnables');
```

2. To run the test generation analysis, on the **Design Verifier** tab, click **Generate Tests**.

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





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

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

| Timing Legend                          |                          |
|--|--------------------------|
| Discrete                               | Period                   |
| <span style="color: red;">■</span>     | F0 1                     |
| <span style="color: red;">■</span>     | F1 1                     |
| <span style="color: green;">■</span>   | F2 10                    |
| Other                                  |                          |
| <span style="color: magenta;">■</span> | Inf Constant             |
| <span style="color: cyan;">■</span>    | T1 Triggered, Source: F0 |

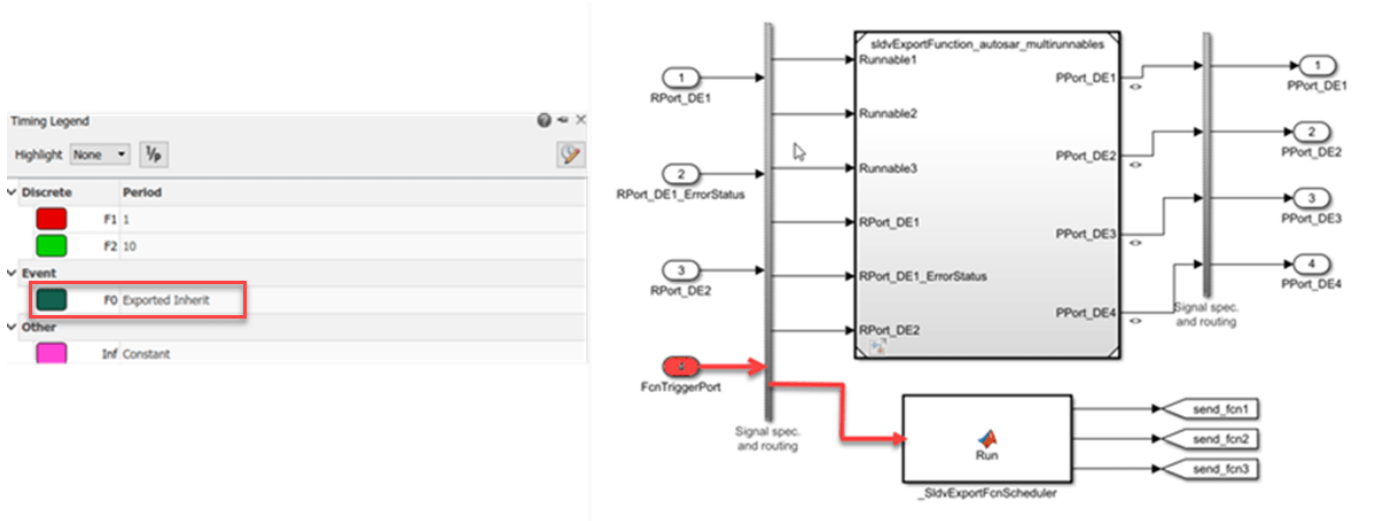
```

sldvExportFunction_autosar_multirunnables_SldvScheduler ▶ _SldvExportFcnScheduler
1  function Run()
2
3  persistent t;
4  if isempty(t)
5      t = int32(0);
6  end
7
8  Runnable1();
9
10 Runnable2();
11
12 if mod(t, int32(10)) == 0
13     Runnable3();
14 end
15
16 t = t + int32(1);
17
18 end
    
```

If the model consists of aperiodic function-call subsystems, the scheduler consists of an additional inport `FcnTriggerPort`. The value of `FcnTriggerPort` indicates whether to invoke the function-call in a time step.



For example, if Runnable1 is an aperiodic function-call subsystem, the FcnTriggerPort Inport block invokes the scheduler model. This graphic shows the Timing Legend window and the scheduler model for an aperiodic function-call.



After the test generation analysis, in the Simulink Design Verifier Results Summary window, you see the results that 7/7 objectives are Satisfied.

3. To simulate the test cases and generate a coverage report, click **Simulate tests** and produce a model coverage report in the Simulink Design Verifier Results Summary window. The software simulates the test cases, collects model coverage information, and displays a coverage report.

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

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

| Order | Function-Call Inport      | Sample Time(sec) | Number of times invoked per sample hit |
|-------|---------------------------|------------------|--|
| 1     | <a href="#">Runnable1</a> | 1                | 1                                      |
| 2     | <a href="#">Runnable2</a> | 1                | 1                                      |
| 3     | <a href="#">Runnable3</a> | 10               | 1                                      |

**See Also**

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

# Analyze Export-Function Model with Global Simulink Function

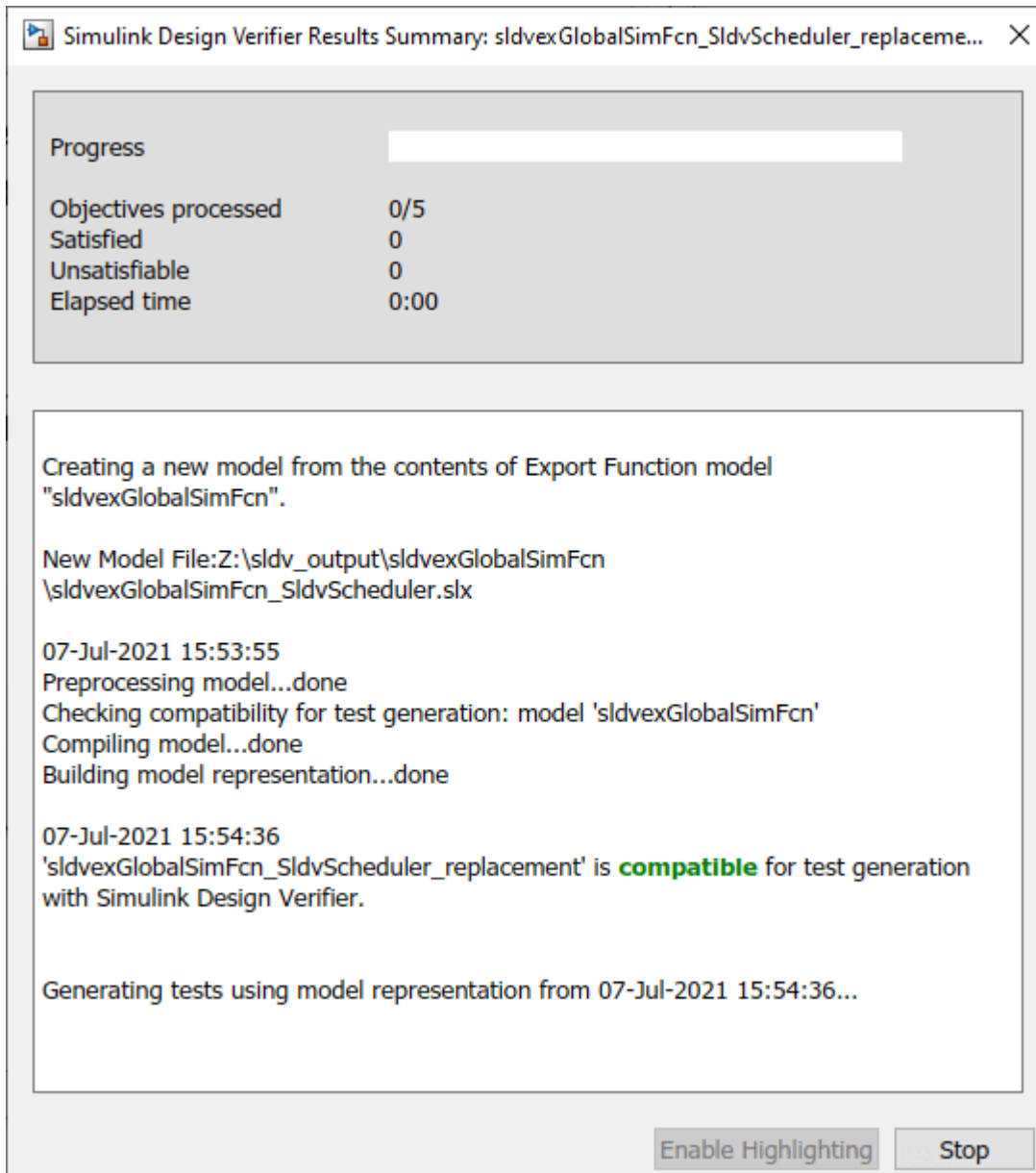
This example shows how you can analyze an export-function model `sldvexGlobalSimFcn` that consists of a global Simulink Function to be used as an AUTOSAR server runnable.

1. Open the `sldvexGlobalSimFcn` model.


```
open_system('sldvexGlobalSimFcn');
```

2. To run the test generation analysis, on the **Design Verifier** tab, click **Generate Tests**.

The Simulink Design Verifier Results Summary window indicates that a scheduler model `sldvexGlobalSimFcn_sldvScheduler.slx` is created. You can also generate a scheduler model by using `sldvextract`.



Simulink Design Verifier Results Summary: sldvexGlobalSimFcn\_SldvScheduler\_replaceme... X

Progress 

|                      |      |
|----------------------|------|
| Objectives processed | 0/5  |
| Satisfied            | 0    |
| Unsatisfiable        | 0    |
| Elapsed time         | 0:00 |

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

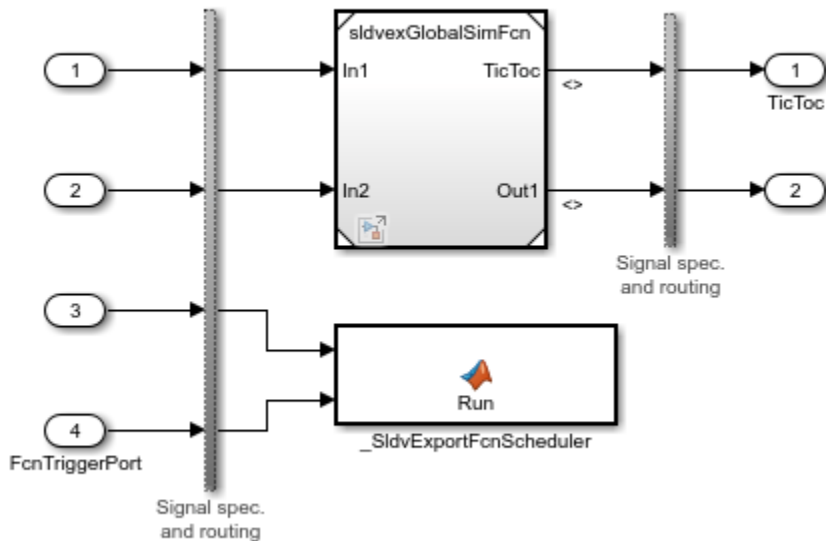
New Model File:Z:\sldv\_output\sldvexGlobalSimFcn\sldvexGlobalSimFcn\_SldvScheduler.slx

07-Jul-2021 15:53:55  
Preprocessing model...done  
Checking compatibility for test generation: model 'sldvexGlobalSimFcn'  
Compiling model...done  
Building model representation...done

07-Jul-2021 15:54:36  
'sldvexGlobalSimFcn\_SldvScheduler\_replacement' is **compatible** for test generation with Simulink Design Verifier.

Generating tests using model representation from 07-Jul-2021 15:54:36...

Enable Highlighting Stop



The scheduler model consists of a MATLAB function block `_SldvExportFcnScheduler` and a function-call subsystem that calls the function calls periodically. This MATLAB function block is driven by inports which represent the input arguments of the Simulink Function. An additional Inport block called `FcnTriggerPort`, the value of which indicates whether to invoke a particular function in a time step.

3. After the test generation analysis, in the Simulink Design Verifier Results Summary window, you see the results that 5/5 objectives are Satisfied.

### See Also

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

## Nonfinite Data

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

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

- In the Relational Operator block:
  - If the **Relational operator** parameter is `isFinite`, the output is always 1.
  - If the **Relational operator** parameter is `isNan` or `isInf`, the output is always 0.
- In the MATLAB Function block:
  - For the `isFinite` function, the output is always 1.
  - For the `isNan` and `isInf` functions, the output is always 0.

## Role of Approximations During Model Analysis

| In this section...  |
|---|
| “Types of Approximations” on page 2-20  |
| “Floating-Point to Rational Number Conversion” on page 2-20   |
| “Linearization of Two-Dimensional Lookup Tables for Floating-Point Data Types” on page 2-21                   |
| “Approximation of One- and Two-Dimensional Lookup Tables for Integer and Fixed-Point Data Types” on page 2-21 |
| “While Loops” on page 2-22  |

The Simulink Design Verifier software generates inputs and parameters to achieve objectives. However, there can 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 all the approximations it performed in the Analysis Information chapter of the Simulink Design Verifier HTML report. For a description of this chapter, see “Analysis Information Chapter” on page 13-36.

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.

In rare cases, 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. For more information, see “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23.

### Types of Approximations

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

- “Floating-Point to Rational Number Conversion” on page 2-20
- “Linearization of Two-Dimensional Lookup Tables for Floating-Point Data Types” on page 2-21
- “Approximation of One- and Two-Dimensional Lookup Tables for Integer and Fixed-Point Data Types” on page 2-21
- “While Loops” on page 2-22

### Floating-Point to Rational Number Conversion

In some cases, the Simulink Design Verifier software simplifies the linear arithmetic of floating-point numbers by approximating them with infinite-precision rational numbers. The software discovers how the logical relationships between these values affect the objectives. This analysis enables the software to support supervisory logic that is commonly found in embedded controller designs. For an example, see “Identify the Effect of Approximations Through Validation Results” on page 2-24.

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

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

---

**Note** You can use the **Run additional analysis to reduce instances of rational approximation** option in the Configuration parameters window to reduce instances of approximation. For more information, see “Run Additional Analysis to Reduce Instances of Rational Approximation” on page 2-42.

---

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

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

| Block                             | Characteristics  |
|-----------------------------------|--|
| n-D Lookup Table block, $n = 2$ : | <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> |

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

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

| Block  | Characteristics   |
|--|---|
| n-D Lookup Table block, $n = 1$ or $n = 2$ : | <ul style="list-style-type: none"> <li>• <b>Interpolation method</b> parameter is Linear.</li> <li>• <b>Extrapolation method</b> parameter is Clip .</li> <li>• <b>Index search method</b> parameter is Linear search or Binary search.</li> <li>• The input and output signals are both of the same type and are both integer type or fixed-point type.</li> </ul> |

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

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

## While Loops

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

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

| <b>Analysis Mode</b>   | <b>While Loop Approximation</b>   |
|------------------------|---|
| Design Error Detection | Sets number of <code>while</code> loop iterations to 3. Does not report dead logic or valid objectives. |
| Test Case Generation   | Sets number of <code>while</code> loop iterations to 3. Does not report unsatisfiable objectives.       |
| Property Proving       | Sets number of <code>while</code> loop iterations to 3. Does not report valid objectives.               |

## See Also

“How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23 |  
“Review Analysis Results” on page 7-8



## How Simulink Design Verifier Reports Approximations Through Validation Results

Simulink Design Verifier performs approximations during analysis. The software identifies the presence of approximations and reports them at the level of each objective status in the Objective Status Chapter of the Simulink Design Verifier HTML report. For more information, see “Role of Approximations During Model Analysis” on page 2-20 and “Objectives Status Chapters” on page 13-42.

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

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

### Impact of Approximations on Objectives Status

The software provides the test cases or counterexamples for the objectives that are impacted due to approximations during analysis. These objectives are reported as “Objectives Undecided with Testcases” on page 13-47 for test generation analysis and “Objectives Undecided with Counterexamples” on page 13-49 for property-proving analysis.

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

| Analysis Mode          | Objectives Status  |
|------------------------|--|
| Design error detection | <ul style="list-style-type: none"> <li>“Dead Logic under Approximation” on page 13-44</li> <li>“Objectives Valid under Approximation” on page 13-45</li> </ul> |
| Test generation        | “Objectives Unsatisfiable under Approximation” on page 13-47   |
| Property proving       | “Objectives Valid under Approximation” on page 13-48   |

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

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

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

| Analysis Mode          | Objectives Status  |
|------------------------|--|
| Design error detection | <ul style="list-style-type: none"> <li>• “Active Logic - Needs Simulation” on page 13-44</li> <li>• “Objectives Error - Needs Simulation” on page 13-45</li> </ul> |
| Test generation        | “Objectives Satisfied - Needs Simulation” on page 13-46  |
| Property proving       | “Objectives Falsified - Needs Simulation” on page 13-49  |

## Identify the Effect of Approximations Through Validation Results

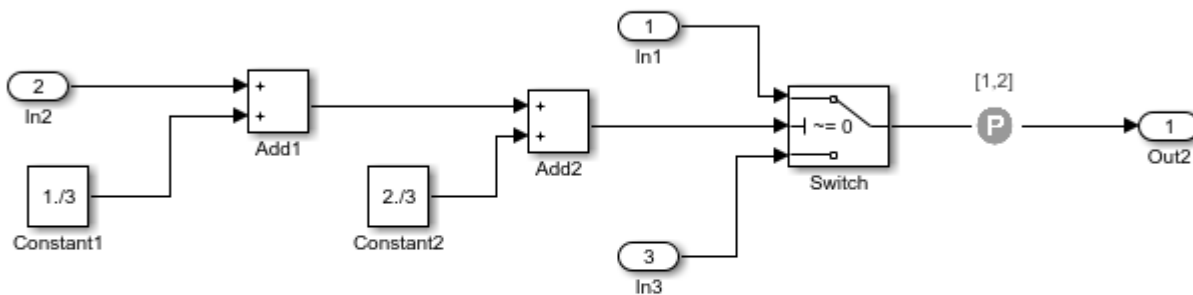
This example shows how approximations affect the objectives status of the Switch block. In the `sldvApproximationsExample` model, the calculations  $1./3$  and  $2./3$  in the Constant block result in “Floating-Point to Rational Number Conversion” on page 2-20 during analysis.

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

1. Open the model `sldvApproximationsExample`:

```
open_system('sldvApproximationsExample');
```

### Reporting Approximations Through Validation Results



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

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

Copyright 2017 The MathWorks, Inc.

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

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

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

#### Test Objectives Status - Objective Satisfied

| # | Type     | Model Item             | Description  | Analysis Time (sec) | Test Case         |
|---|----------|------------------------|--|---------------------|-------------------|
| 2 | Decision | <a href="#">Switch</a> | logical trigger input true (output is from 1st input port) | 14                  | <a href="#">1</a> |

#### Test Objectives Status - Objective Undecided with Testcases

| # | Type     | Model Item             | Description   | Analysis Time (sec) | Test Case         |
|---|----------|------------------------|---|---------------------|-------------------|
| 1 | Decision | <a href="#">Switch</a> | logical trigger input false (output is from 3rd input port) | 14                  | <a href="#">2</a> |

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

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

#### Proof Objectives Status - Objective Undecided with Counterexamples

| # | Type            | Model Item                      | Description       | Analysis Time (sec) | Counterexample    |
|---|-----------------|---------------------------------|-------------------|---------------------|-------------------|
| 1 | Proof objective | <a href="#">Proof Objective</a> | Objective: [1, 2] | 11                  | <a href="#">1</a> |

The software provides one counterexample that is impacted by approximations.

Note: The `sldvApproximationsExample` example model is preconfigured with the Run additional analysis to reduce the instances of approximations option set to `Off`. If you enable this option and run the analysis, the Undecided with Testcases test objective is reported as `Unsatisfiable` and the proof objective is reported as `Valid`.

## See Also

### More About

- “Review Results” on page 13-35
- “Role of Approximations During Model Analysis” on page 2-20

## Logic Operations Short-Circuiting

Simulink Design Verifier considers logical operations and logical expressions as short-circuiting when analyzing for dead logic and when generating tests.

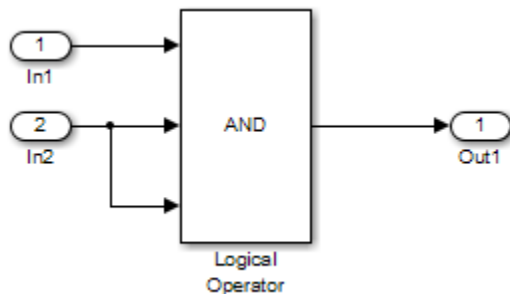
Logical Operators and Logical Expressions for Condition and MCDC objectives can be considered short-circuiting or not when you analyze for dead logic or generate tests. The table summarizes different considerations:

### Short-Circuit consideration for Condition or MCDC Objectives

| Modeling element   | Short-Circuit consideration for Condition or MCDC Objectives |
|--|--|
| MATLAB, Stateflow (C/MATLAB) and other Simulink Blocks (Fcn, If) | Always short-circuited                                       |
| Logic blocks (standalone/cascaded)                               | Short-circuited only when CovLogicBlockShortCircuit is ON    |

For Condition objective, consider the following simple logical operator example model. When CovLogicBlockShortCircuit parameter is ON, a previous input alone determines the block output, the analysis ignores any remaining block inputs. If the first input to a Logical Operator block whose **Operator** parameter specifies AND is false, the analysis ignores the values of the other inputs.

When CovLogicBlockShortCircuit parameter is OFF, all the inputs are considered.



The tables summarizes the difference in objectives for short-circuit and non short-circuit case for Condition objective by using a similar logical expression for MATLAB function block:

### Short-Circuit considerations for Condition objective

| Condition 'F' Port 3  | CovLogicBlockShortCircuit: ON         | CovLogicBlockShortCircuit: OFF        |
|---|---------------------------------------|---------------------------------------|
| Logical operator  | (in1) && (in2) && (~in2) (dead logic) | ~ in2 (active logic)                  |
| MATLAB Function with logical expression (in1 && in2 && in2) | (in1) && (in2) && (~in2) (dead logic) | (in1) && (in2) && (~in2) (dead logic) |

For MCDC objective, along with the short-circuit mode, the CovMCDCMode or the coverage MCDC mode is set as a parameter.

The tables summarizes the difference in objectives for short-circuit and non short-circuit case for MCDC objective.

**Short-Circuit considerations for MCDC objective**

| <b>CovLogicBlockShortCircuit</b> | <b>CovMCDCMode</b> | <b>Standalone block</b>  | <b>Cascaded Network</b>  |
|----------------------------------|--------------------|--------------------------|--|
| ON                               | Masking            | Short circuited MCDC     | MCDC for network (Short-circuited)   |
| OFF                              | Masking            | Non short-circuited MCDC | MCDC for network (Non short-circuited)   |
| ON                               | Unique cause       | Short-circuited MCDC     | MCDC (Short-circuited) coverage result can differ from Simulink Design Verifier. |
| OFF                              | Unique cause       | Non short-circuited MCDC | NA   |

---

**Note** If covMCDCMode is Unique cause, then MCDC definition differs between coverage and MCDC.

---

For more information, see "Short-Circuiting of Boolean Expressions for MCDC" in "Analyzing MCDC for Cascaded Logic Blocks" (Simulink Coverage).

## Model Representation for Analysis

### In this section...

“Reuse Model Representation for Analysis” on page 2-28

“Limitations” on page 2-30

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

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

### Reuse Model Representation for Analysis

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

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

- Simulink Design Verifier Options on page 2-28
- “Structural Checksum of a Model” on page 2-29
- “Additional Dependencies” on page 2-30

### Simulink Design Verifier Options

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

|                         |  |
|-------------------------|--|
| Design Verifier Options | <ul style="list-style-type: none"> <li>• “Maximum analysis time” on page 15-11</li> <li>• “Output folder” on page 15-11</li> <li>• “Make output file names unique by adding a suffix” on page 15-12</li> <li>• “Run additional analysis to reduce instances of rational approximation” on page 15-15</li> <li>• “Ignore objectives based on filter” on page 15-17</li> <li>• “Filter file(s)” on page 15-18</li> </ul> |
|-------------------------|--|

|                            |   |
|----------------------------|---|
| Test Generation options    | <ul style="list-style-type: none"> <li>• “Test conditions” on page 15-32</li> <li>• “Test objectives” on page 15-33</li> <li>• “Maximum test case steps” on page 15-33</li> <li>• “Test suite optimization” on page 15-34</li> <li>• “Extend using existing coverage data” on page 15-38</li> <li>• “Extend using existing coverage data” on page 15-38</li> <li>• “Extend using existing test data” on page 15-39</li> <li>• “Separate objectives satisfied with the existing tests/coverage data in the report” on page 15-40</li> </ul>                |
| Property Proving options   | <ul style="list-style-type: none"> <li>• “Assertion blocks” on page 15-52</li> <li>• “Proof assumptions” on page 15-53</li> <li>• “Strategy” on page 15-53</li> <li>• “Maximum violation steps” on page 15-54</li> </ul>  |
| Results generation options | <ul style="list-style-type: none"> <li>• “Data file name” on page 15-57</li> <li>• “Include expected output values” on page 15-57</li> <li>• “Randomize data that do not affect the outcome” on page 15-58</li> <li>• “Generate separate harness model after analysis” on page 15-59</li> <li>• “Harness model file name” on page 15-59</li> <li>• “Reference input model in generated harness” on page 15-60</li> <li>• “Harness source” on page 15-61</li> <li>• “Test File Name” on page 15-61</li> <li>• “Test Harness Name” on page 15-62</li> </ul> |
| Report generation options  | <ul style="list-style-type: none"> <li>• “Generate report of the results” on page 15-63</li> <li>• “Generate additional report in PDF format” on page 15-64</li> <li>• “Report file name” on page 15-64</li> <li>• “Include screen shots of properties” on page 15-65</li> <li>• “Display report” on page 15-66</li> </ul>  |

### Structural Checksum of a Model

The Simulink Design Verifier uses both structural checksum and code checksum. A structural checksum is a computation that detects changes in the model that can affect simulation results. For more information about the kinds of changes that affect the model, see Rebuild.

---

**Note** When you “Generate Test Cases for Embedded Coder Generated Code” on page 7-28, Simulink Design Verifier also considers checksum of the generated code.

---

### Additional Dependencies

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

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

### Limitations

- The model representation is always rebuilt:
  - When Simulink Design Verifier analysis is started from other products such as Simulink Test™, Simulink Coverage, Simulink Check™, and Requirements Toolbox™.
  - When the model contains MATLAB System blocks.
- Simulink Design Verifier does not detect changes in the custom block replacement rules that you apply, even if the **Rebuild model representation** option is set to **If change is detected**. In such cases, the Simulink cache is reused for analysis and a warning message is displayed in the Diagnostic Viewer that suggests you to set the **Rebuild model representation** option to **Always**, if you want to rebuild the model representation.

### See Also

“Extend Existing Test Cases by Reusing Model Representation” on page 2-35

### More About

- Configure Model Representation Options on page 2-39
- “Check Model Compatibility” on page 3-2
- “Simulink Design Verifier Options” on page 15-2



## Share Simulink Cache File for Faster Analysis

### In this section...

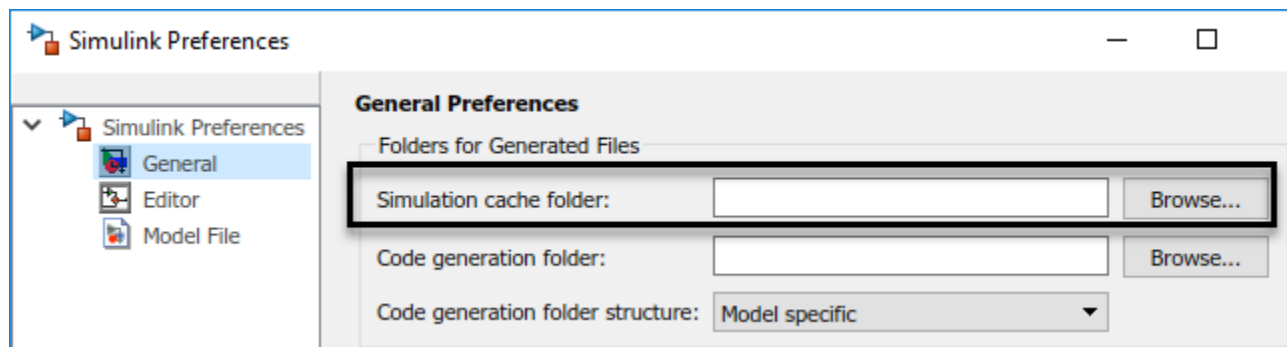
“Store the Simulink Cache File” on page 2-31

“Reuse the Simulink Cache File” on page 2-31

You can share the Simulink cache file for faster Simulink Design Verifier analysis. When you analyze a model, Simulink Design Verifier performs a compatibility check and creates a Simulink cache file that contains the model representation information. If there is no change in the model, Simulink Design Verifier reuses the model representation from the Simulink cache file without performing the compatibility check again. For more information, see “Share Simulink Cache Files for Faster Simulation” and “Model Representation for Analysis” on page 2-28.

### Store the Simulink Cache File

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



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

### Reuse the Simulink Cache File

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

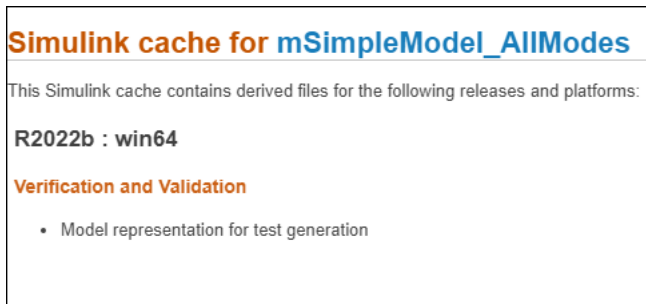
If **Rebuild model representation** is set to **Always** or if the software detects any change in the model during analysis, the software rebuilds the model representation and updates the Simulink cache file.

---

**Note** The Simulink cache file accumulates model representation build artifacts for the release in which it was created and is platform dependent. This cache file does not support cross-release compatibility.

---

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



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

- 1 A developer pulls the latest version of the Simulink model from the source control system.
- 2 Performs Simulink Design Verifier test case generation analysis and shares the latest version of Simulink cache file to the source control system and the generated test cases to the build archive.
- 3 Test engineer pulls the latest version of the model and the Simulink cache file from the source control systems. Also, pulls the existing test cases from the build archive.
- 4 Performs test case extension on the same model by using the existing test cases. If no changes are detected in the model, the model representation from the Simulink cache file is reused for analysis. For a detailed example, see "Extend Existing Test Cases by Reusing Model Representation" on page 2-35.

If the test engineer, changes the model or Simulink Design Verifier options that affects the compatibility results, the model representation is rebuilt and the Simulink cache file is updated. For more information on Simulink Design Verifier options that leverage the reuse of model representation, see "Reuse Model Representation for Analysis" on page 2-28.

## See Also

### More About

- "Model Representation for Analysis" on page 2-28
- Configure Model Representation Options on page 2-39

### External Websites

- Simulink Cache (1 min, 27 sec)

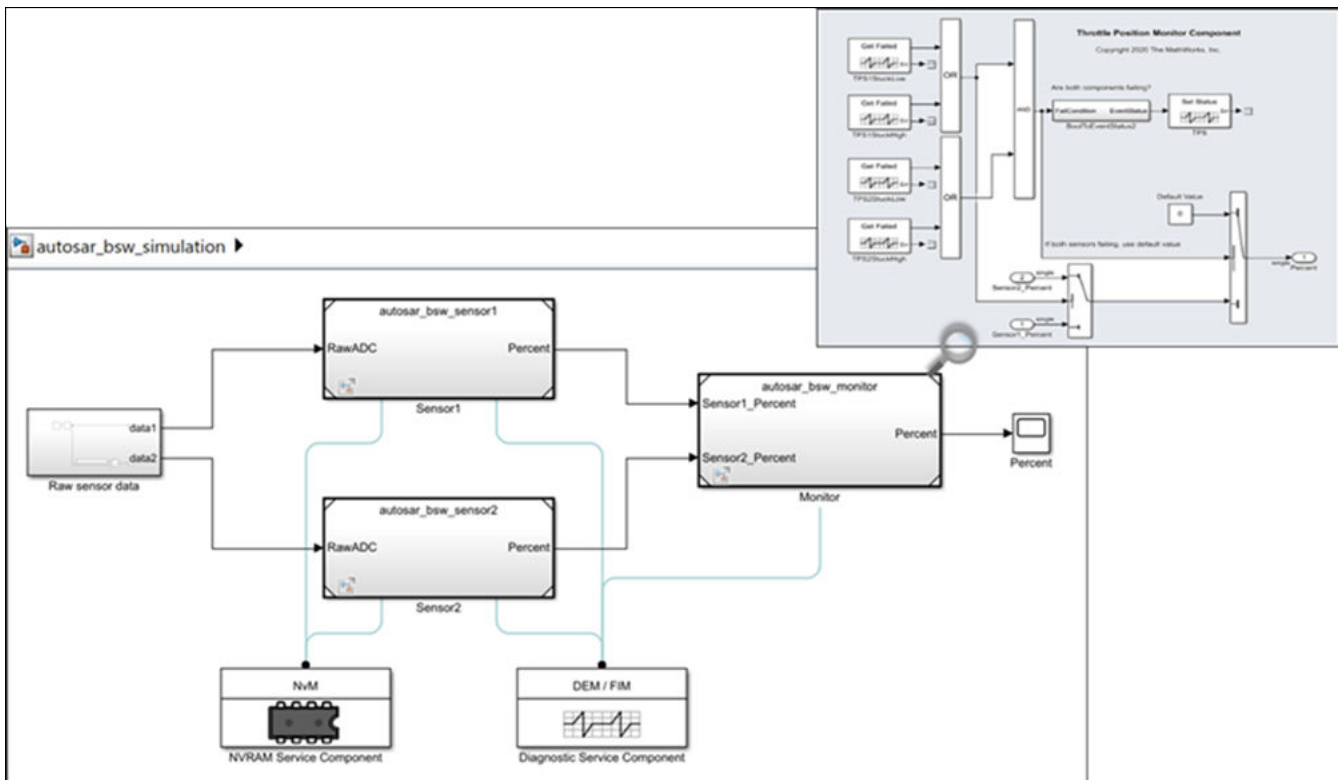
## Analyze AUTOSAR Component Models

Simulink Design Verifier supports design error detection, test generation, and property proving analysis for AUTOSAR software components (SWC) at the unit level. You can analyze an AUTOSAR component that contains blocks from the AUTOSAR Blockset Basic Software block library, which model component calls to AUTOSAR Basic Software (BSW) services, including:

- Diagnostic Event Manager (Dem)
- Function Inhibition Manager (FiM)
- NVRAM Manager (NvM)

Additionally, you can analyze a Simulink model generated by importing descriptions of AUTOSAR software components from AUTOSAR XML (ARXML) files. See, “Create and Configure AUTOSAR Software Component” (AUTOSAR Blockset).

The software creates an analysis harness that provides stub implementations of the Basic Software service operations called by the component, and then performs the analysis on the harness model. By default, the software saves the harness model in `<current_folder>\sldv_output \<model_name>\<model_name>_SldvStub.slx`.



### AUTOSAR Model at Component Level

### Limitations

The Simulink Design Verifier analysis reports an incompatibility if:

- You use Simulink Design Verifier to generate tests in the Simulink Test, and the harness parameter is set to `Signal Editor`.
- The component model contains service component blocks, such as the Diagnostic Service Component or NVRAM Service Component blocks.
- The component model contains Initialize Function, Reinitialize Function, Reset Function, or Terminate Function blocks that call a Simulink functions that is not defined in the same component.
- If you perform Software-in-the-Loop (SIL) code analysis on an AUTOSAR component model
- You export test cases generated by Simulink Design Verifier and run software-in-the-loop (SIL) simulation on those test cases in Simulink Test Manager. The recommended approach is to perform back-to-back testing using Simulink Test.

### See Also

“Configure Elements of AUTOSAR Software Component for Simulink Modeling Environment” (AUTOSAR Blockset) | “Import Test Cases for Equivalence Testing” (Simulink Test)

### Related Examples

- “Detect Design Errors in AUTOSAR Software Component Model” on page 2-47

## Extend Existing Test Cases by Reusing Model Representation

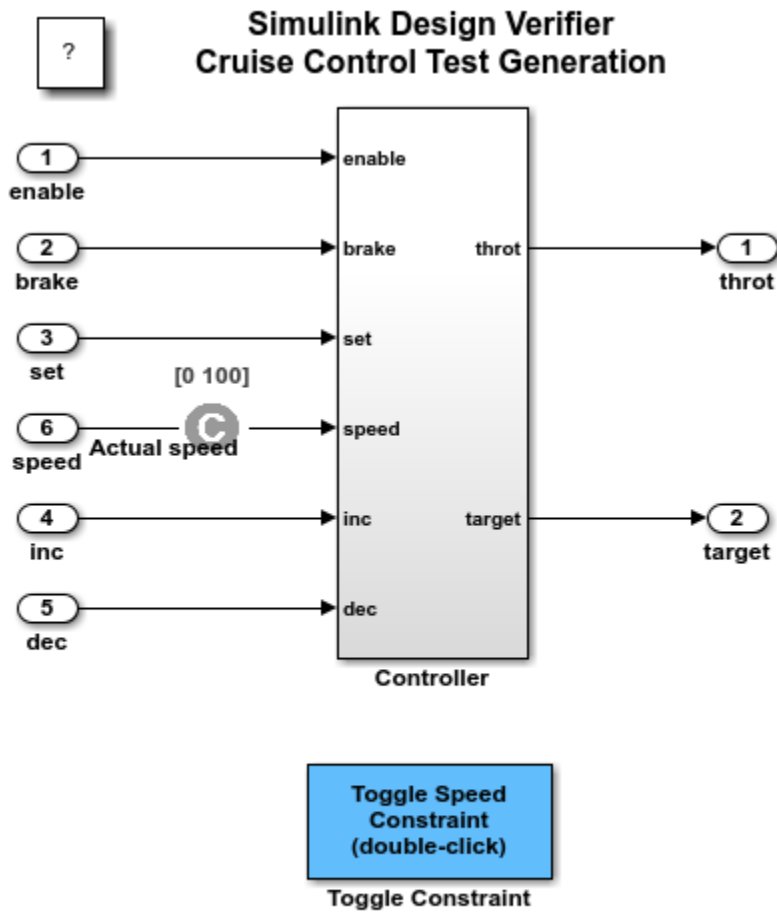
This example shows how to avoid unneeded model representation builds when reanalyzing a model. Consider a case where you perform test generation and the analysis exceeds maximum analysis time. In the specified analysis time, Simulink Design Verifier analyzes some objectives and saves the generated test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

To reanalyze the model, you update the maximum analysis time and select the extend existing test cases option. To speed up the analysis, set the **Rebuild model representation** option to `If change is detected`. Simulink Design Verifier reanalyzes the model by reusing the model representation. For more information, see “Model Representation for Analysis” on page 2-28.

### Step 1. Open the model and specify analysis options

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

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



Copyright 2006-2023 The MathWorks, Inc.

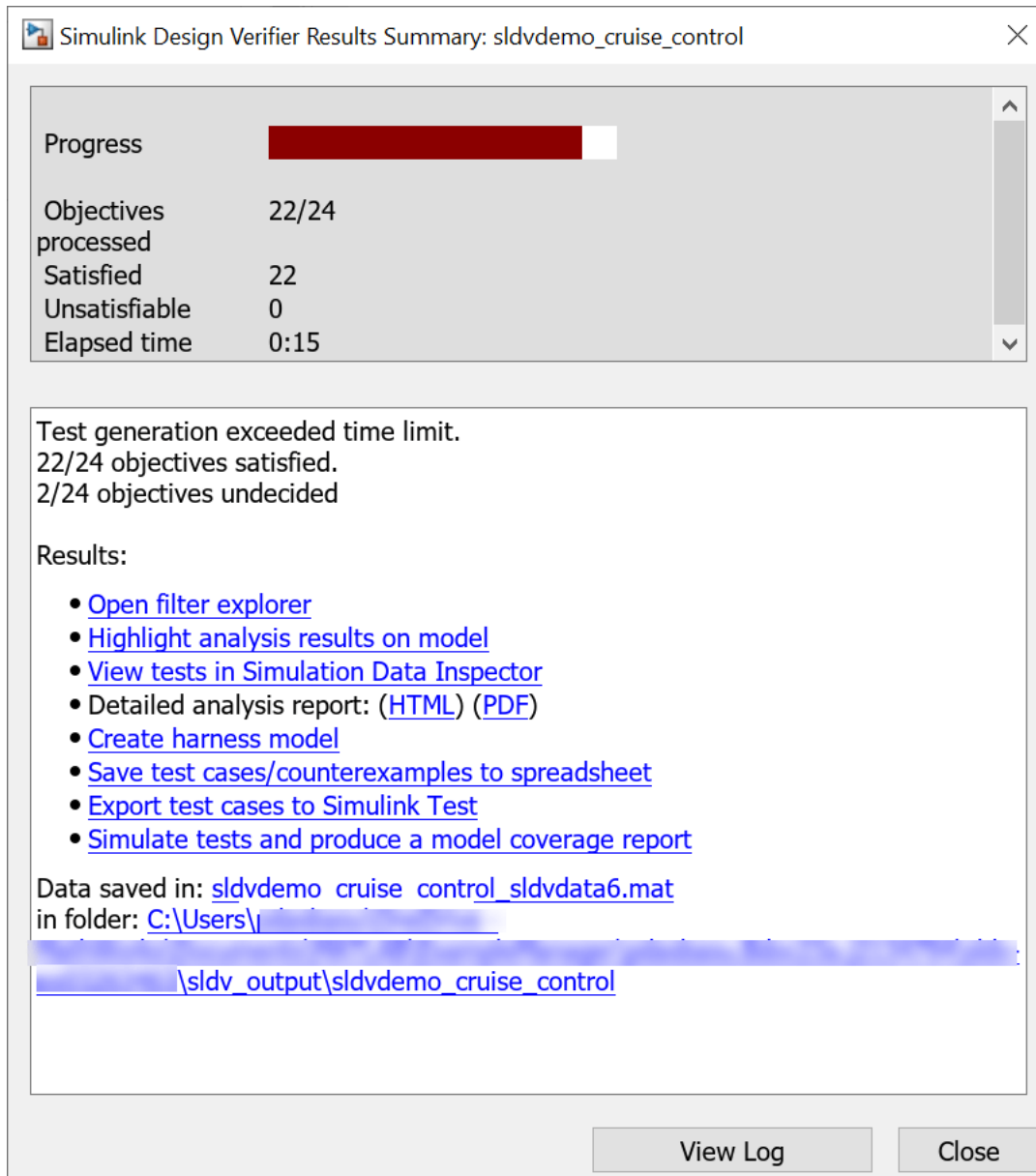
Analyze the model by using this command.

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

The Diagnostic Viewer window displays the Test Generation analysis error.

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

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

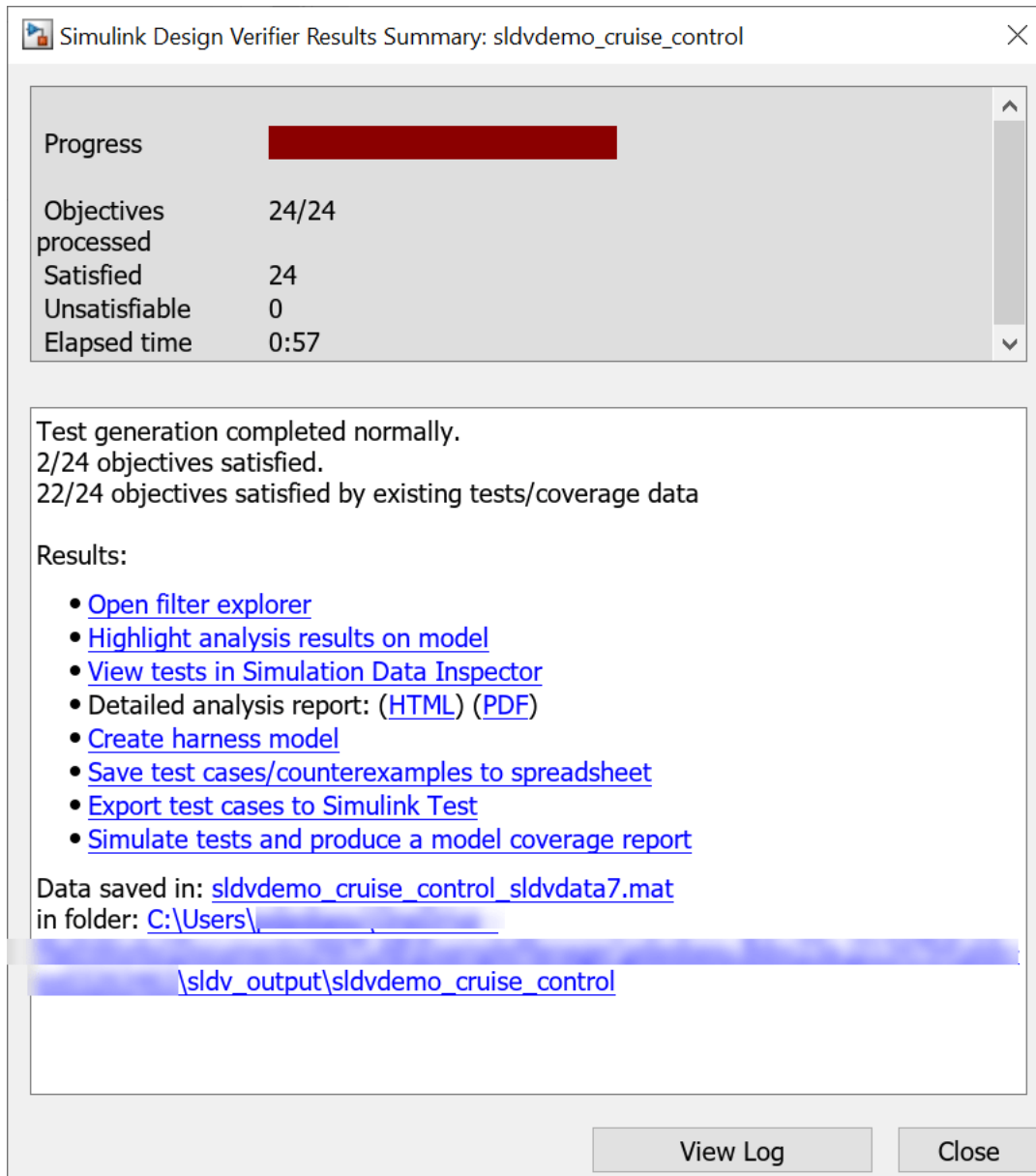


## Step 2. Reanalyze the model by modifying the sldvoptions

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

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

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



Close the model.

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

### Related Topics

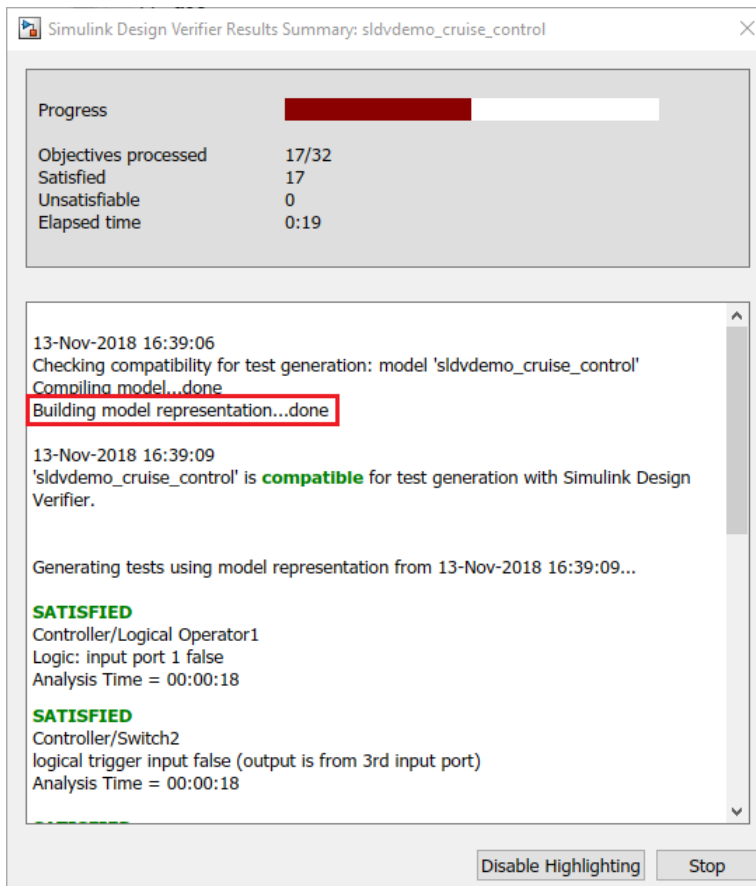
- “Model Representation for Analysis” on page 2-28
- “Extend an Existing Test Suite” on page 7-86



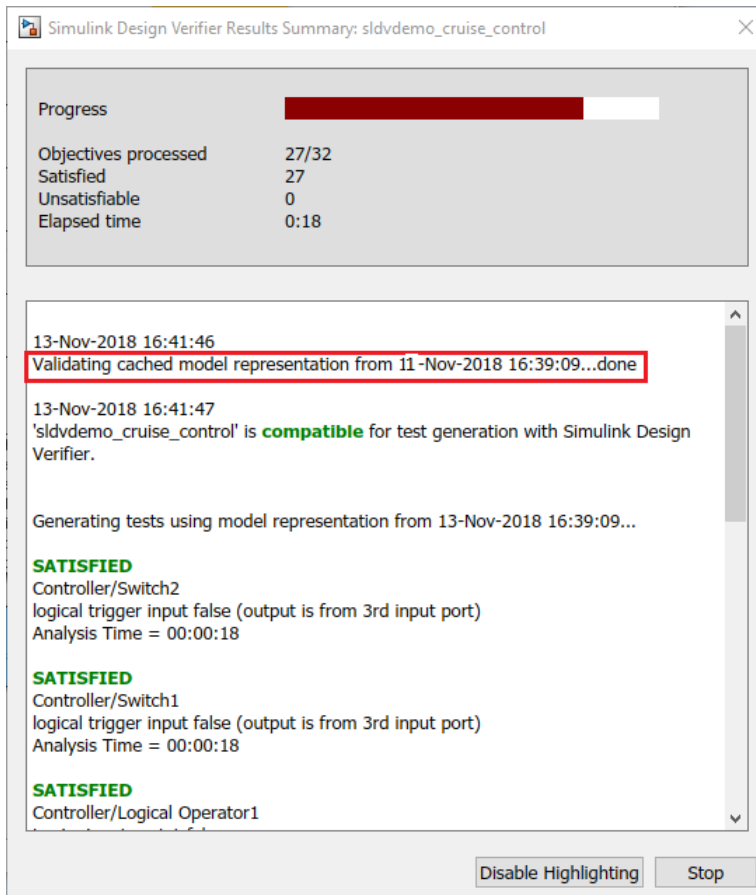
## Configure Model Representation Options

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

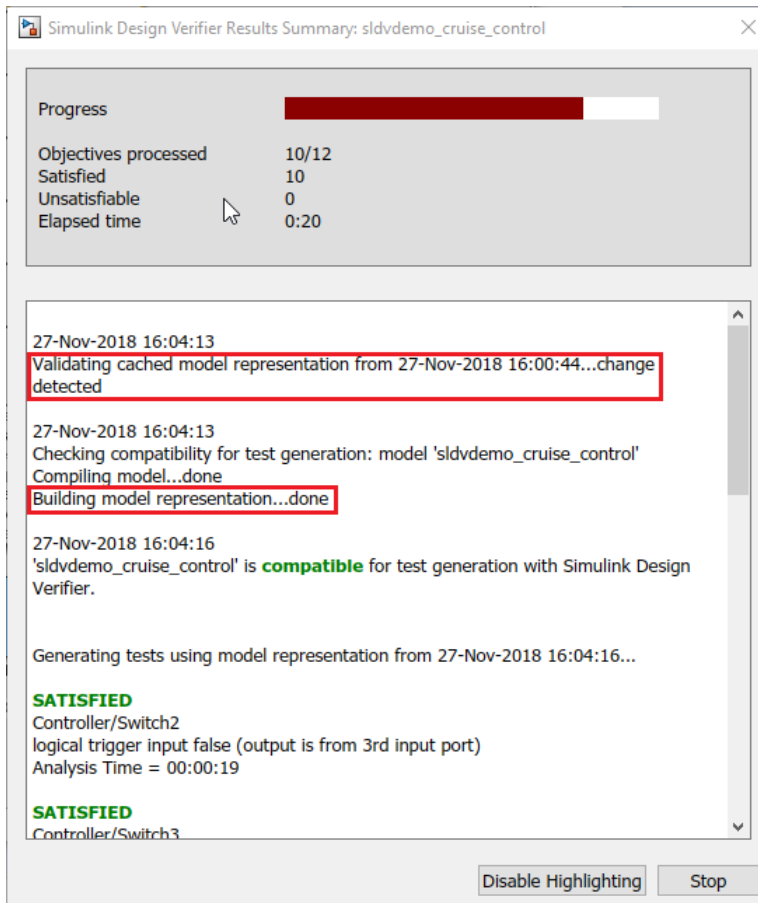
When you perform analysis, the Results Summary window displays the information regarding the model representation. If you select `Always` for the **Rebuild model representation** option, the software rebuilds the model representation during analysis.



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



If change is detected in the model, the model representation is rebuilt. For more information, see [Changes That Affect the Model Representation Rebuild](#) on page 2-28.



## See Also

### More About

- “Model Representation for Analysis” on page 2-28
- “Check Model Compatibility” on page 3-2

## Run Additional Analysis to Reduce Instances of Rational Approximation

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

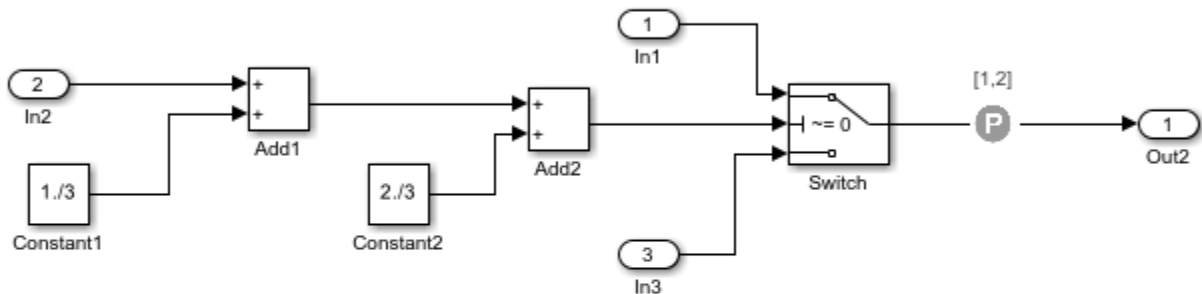
You enable the **Run additional analysis to reduce instances of rational approximation** option to perform additional analysis to confirm the undecided objectives. When you rerun the analysis, Simulink cache that contains the model representation information is reused to perform faster analysis. For more information see “Reuse Model Representation for Analysis” on page 2-28.

### Open the Model

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

```
open_system('sldvApproximationsExample')
```

### Reporting Approximations Through Validation Results

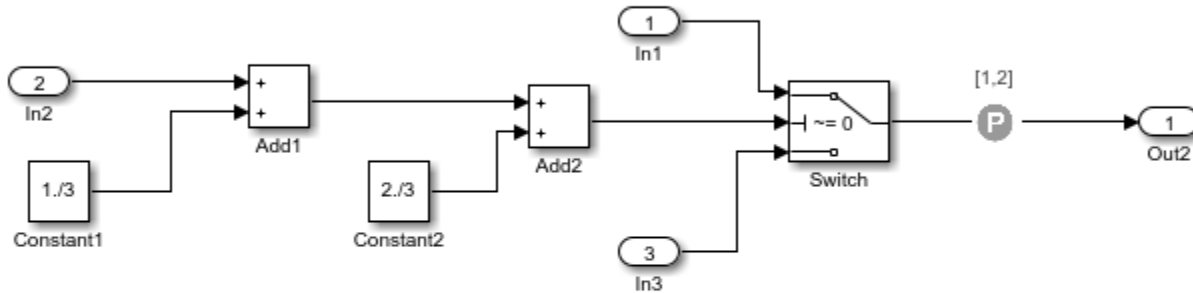


This example shows how Simulink Design Verifier reports the impact of approximations through validation results.

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

Copyright 2017-2019 The MathWorks, Inc.

## Reporting Approximations Through Validation Results



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


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

Copyright 2017-2019 The MathWorks, Inc.

### Perform Test Case Generation Analysis and Review Results

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

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

Progress 

Objectives processed 2/2

Satisfied 1

Unsatisfiable 0

Elapsed time 0:20

---

Test generation completed normally.

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

Results:

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

Data saved in: [sldvApproximationsExample\\_sldvdata.mat](#)  
in folder: [H:\sldv\\_output\sldvApproximationsExample](#)

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

### Approximations

Simulink Design Verifier performed the following approximations during analysis. These can impact the precision of the results generated by Simulink Design Verifier. Please see the product documentation for further details.

| # | Type                   | Description   |
|---|------------------------|---|
| 1 | Rational approximation | The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic. Specifying minimum and maximum values that mimic environmental constraints on root-level Inport blocks may reduce instances of rational approximation. |

The Objective Status chapter gives detailed description of the objectives.

| <b>Objectives Satisfied</b>   |          |                        |   |                     |                   |
|---|----------|------------------------|---|---------------------|-------------------|
| Simulink Design Verifier found test cases that exercise these test objectives.  |          |                        |   |                     |                   |
| #   | Type     | Model Item             | Description   | Analysis Time (sec) | Test Case         |
| 2   | Decision | <a href="#">Switch</a> | logical trigger input true (output is from 1st input port)  | 8                   | <a href="#">1</a> |
| <b>Objectives Undecided with Testcases</b>  |          |                        |   |                     |                   |
| Simulink Design Verifier was not able to decide these objectives due to the impact of approximations during analysis. |          |                        |   |                     |                   |
| #   | Type     | Model Item             | Description   | Analysis Time (sec) | Test Case         |
| 1   | Decision | <a href="#">Switch</a> | logical trigger input false (output is from 3rd input port) | 22                  | <a href="#">2</a> |

### Run Additional Analysis by Reusing Cache

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

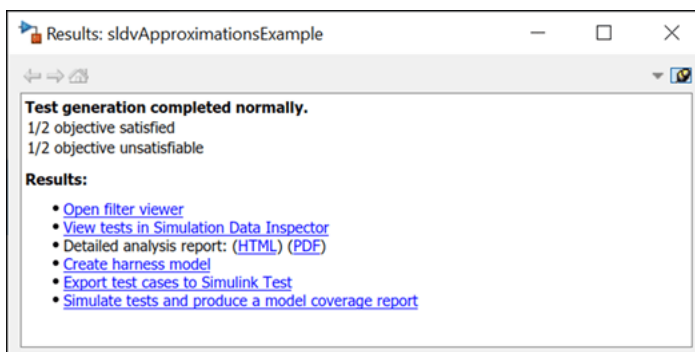
(a) On the **Design Verifier** tab, click **Test Generation Settings > Settings**.

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

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

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

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



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

**Review Analysis Results**

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

| <b>Objectives Satisfied</b>   |          |                        |   |                     |                   |
|---|----------|------------------------|---|---------------------|-------------------|
| Simulink Design Verifier found test cases that exercise these test objectives.  |          |                        |   |                     |                   |
| #   | Type     | Model Item             | Description   | Analysis Time (sec) | Test Case         |
| 2   | Decision | <a href="#">Switch</a> | logical trigger input true (output is from 1st input port)  | 2                   | <a href="#">1</a> |
| <b>Objectives Unsatisfiable</b>   |          |                        |   |                     |                   |
| Simulink Design Verifier found that there does not exist any test case exercising these test objectives. This often indicates the presence of dead logic in the model. Other possible reasons can be inactive blocks in the model due to parameter configuration or test constraints such as given using Test Condition blocks. |          |                        |   |                     |                   |
| #   | Type     | Model Item             | Description   | Analysis Time (sec) | Test Case         |
| 1   | Decision | <a href="#">Switch</a> | logical trigger input false (output is from 3rd input port) | 264                 | n/a               |

**Related Topics**

- “Model Representation for Analysis” on page 2-28
- “Run additional analysis to reduce instances of rational approximation” on page 15-15
- “Rebuild model representation” on page 15-13



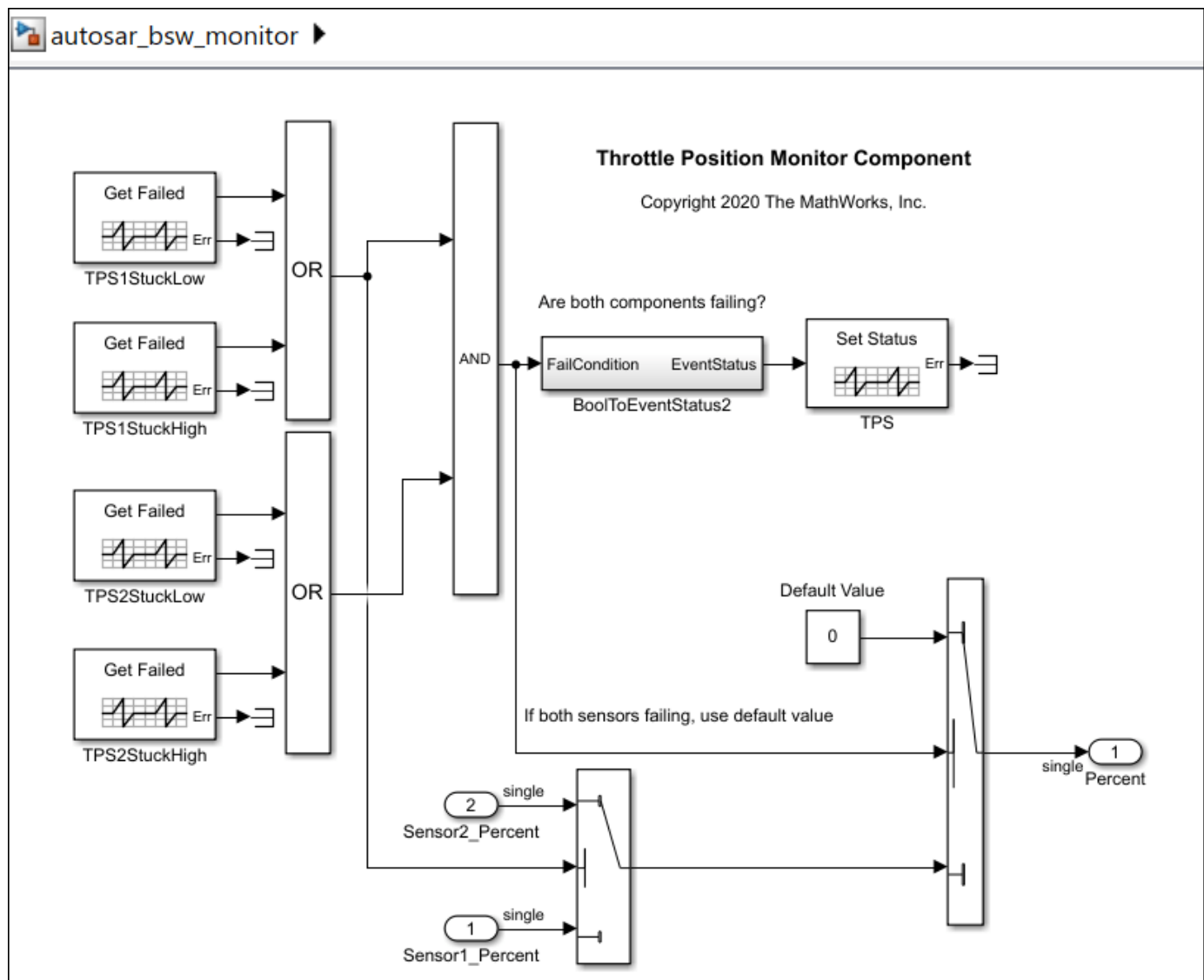
## Detect Design Errors in AUTOSAR Software Component Model

The AUTOSAR standard defines Basic Software (BSW) services that run in the AUTOSAR run-time environment. The services include NVRAM Manager (NvM) Diagnostic Event Manager (Dem), and Function Inhibition Manager (FiM) services. The following example shows how to use Simulink Design Verifier to run design error checks on the AUTOSAR component model.

### Prepare the Model

Open the AUTOSAR software component. This example uses AUTOSAR simulink model `autosar_bsw_monitor`.

```
model = 'autosar_bsw_monitor';
open_system(model);
```



Monitor component `autosar_bsw_monitor` contains a call to the Dem service interface `DiagnosticMonitor` and four calls to the Dem service interface `DiagnosticInfo`. The four `DiagnosticInfo` calls are implemented using the Basic Software library block `DiagnosticInfoCaller` (AUTOSAR Blockset). Each block is configured to call the `DiagnosticInfo` operation `GetEventFailed`. The `GetEventFailed` calls use client ports `TPS1StuckLow`, `TPS1StuckHigh`, `TPS2StuckLow`, and `TPS2StuckHigh`.

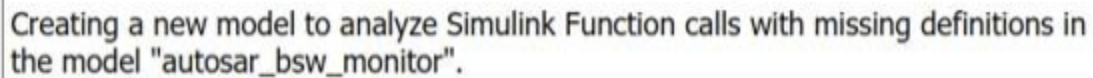
### Perform Design Error Detection Analysis

To detect the design errors in the above component model, configure the Design Verifier options as follows:

```
opts = sldvoptions;  
opts.Mode = "DesignErrorDetection";  
opts.DetectDeadLogic = 'on';  
opts.DetectActiveLogic = 'on';
```

Analyze the model.

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



Creating a new model to analyze Simulink Function calls with missing definitions in the model "autosar\_bsw\_monitor".

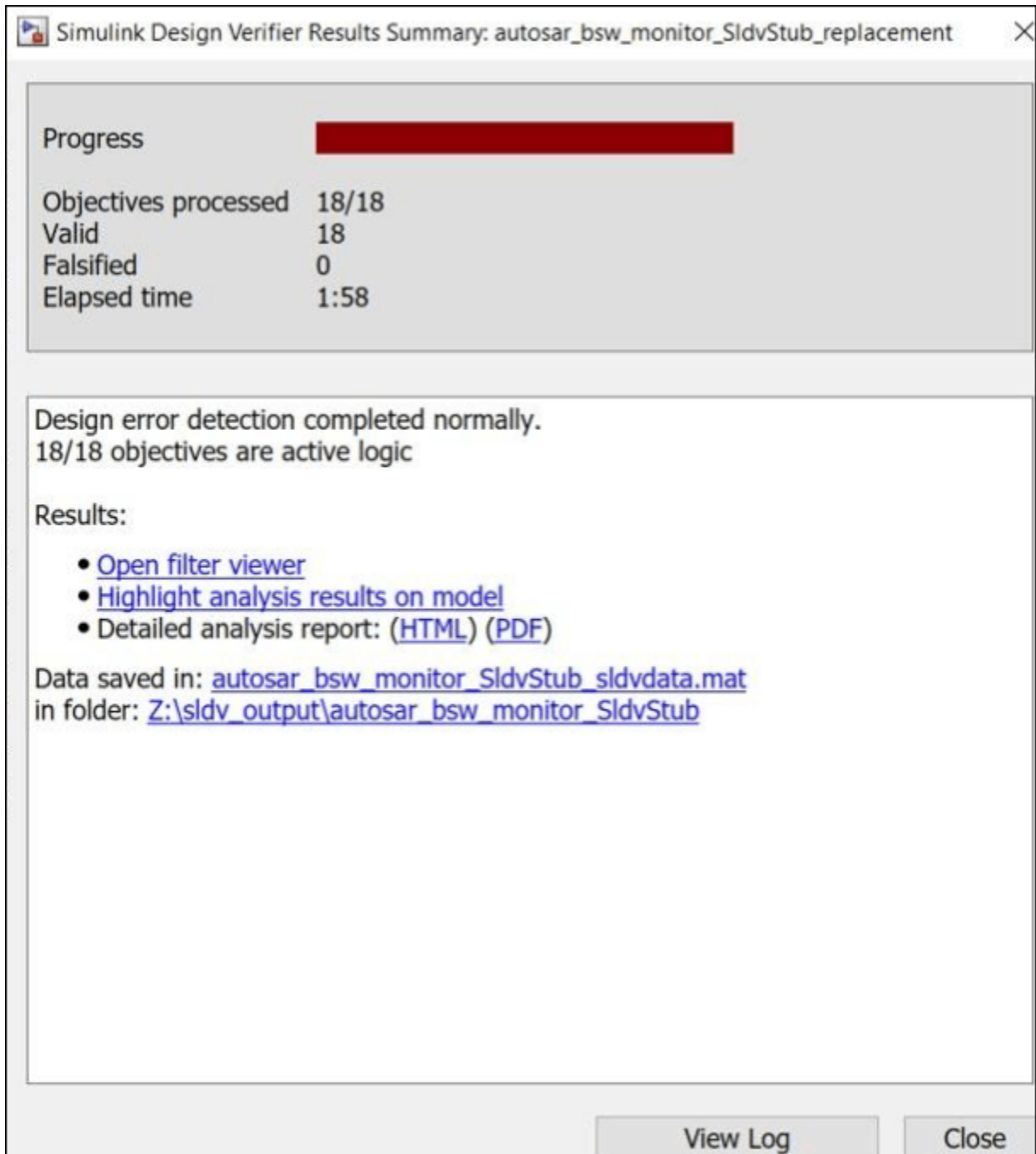


New Model File:Z:\sldv\_output\autosar\_bsw\_monitor  
\autosar\_bsw\_monitor\_SldvStub.slx

The Simulink® Design Verifier™ Results Summary window indicates that an analysis harness model `autosar_bsw_monitor_SldvStub` is created. You can also generate the same analysis harness model using `sldvextract` function.

### Review the Analysis Results

The Simulink Design Verifier Results Summary window shows that 18 of 18 objectives are active logic in the model.



To view the detailed analysis report, click the HTML link in the Results Summary window. The **Design Error Detection Objectives Status** section includes the **Active Logic** objectives statuses for the model.

| #  | Type      | Model Item  | Description  | Analysis Time (sec) |
|----|-----------|---|--|---------------------|
| 6  | Condition | <a href="#">autosar_bsw_monitor/Logical Operator</a>          | Logic: input port 1 <b>true</b>                                    | 38                  |
| 7  | Condition | <a href="#">autosar_bsw_monitor/Logical Operator</a>          | Logic: input port 1 <b>false</b>                                   | 38                  |
| 8  | Condition | <a href="#">autosar_bsw_monitor/Logical Operator</a>          | Logic: input port 2 <b>true</b>                                    | 38                  |
| 9  | Condition | <a href="#">autosar_bsw_monitor/Logical Operator</a>          | Logic: input port 2 <b>false</b>                                   | 38                  |
| 15 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator2</a>         | Logic: input port 1 <b>true</b>                                    | 38                  |
| 16 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator2</a>         | Logic: input port 1 <b>false</b>                                   | 38                  |
| 17 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator2</a>         | Logic: input port 2 <b>true</b>                                    | 38                  |
| 18 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator2</a>         | Logic: input port 2 <b>false</b>                                   | 38                  |
| 20 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator1</a>         | Logic: input port 1 <b>true</b>                                    | 38                  |
| 21 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator1</a>         | Logic: input port 1 <b>false</b>                                   | 38                  |
| 22 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator1</a>         | Logic: input port 2 <b>true</b>                                    | 38                  |
| 23 | Condition | <a href="#">autosar_bsw_monitor/Logical Operator1</a>         | Logic: input port 2 <b>false</b>                                   | 38                  |
| 25 | Decision  | <a href="#">autosar_bsw_monitor/BoolToEventStatus2/Switch</a> | logical trigger input <b>false</b> (output is from 3rd input port) | 38                  |
| 26 | Decision  | <a href="#">autosar_bsw_monitor/BoolToEventStatus2/Switch</a> | logical trigger input <b>true</b> (output is from 1st input port)  | 38                  |
| 27 | Decision  | <a href="#">autosar_bsw_monitor/Switch1</a>                   | logical trigger input <b>false</b> (output is from 3rd input port) | 38                  |
| 28 | Decision  | <a href="#">autosar_bsw_monitor/Switch1</a>                   | logical trigger input <b>true</b> (output is from 1st input port)  | 38                  |
| 30 | Decision  | <a href="#">autosar_bsw_monitor/Switch</a>                    | logical trigger input <b>false</b> (output is from 3rd input port) | 38                  |
| 31 | Decision  | <a href="#">autosar_bsw_monitor/Switch</a>                    | logical trigger input <b>true</b> (output is from 1st input port)  | 38                  |

The analysis report also captures information about the analysis harness for analyzing the model in the **Analysis Harness Information** section. The **Stubbed Simulink Functions for Analysis** section in the **Analysis Harness Information** section lists the stubbed Simulink functions.

| Function Prototype   |
|--|
| <a href="#">[EventFailed,ERR] = TPS2StuckLow_GetEventFailed()</a>  |
| <a href="#">[EventFailed,ERR] = TPS2StuckHigh_GetEventFailed()</a> |
| <a href="#">[EventFailed,ERR] = TPS1StuckLow_GetEventFailed()</a>  |
| <a href="#">[EventFailed,ERR] = TPS1StuckHigh_GetEventFailed()</a> |
| <a href="#">ERR = TPS_SetEventStatus(EventStatus)</a>              |

Note that Simulink Design Verifier assumes that the output of stubbed Simulink Functions is held when the functions are invoked multiple times in a single step.

**Related Links**

- “Analyze AUTOSAR Component Models” on page 2-33

# Checking Compatibility with the Simulink Design Verifier Software

---

- “Check Model Compatibility” on page 3-2
- “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” on page 3-7
- “Support Limitations for Simulink Software Features” on page 3-16
- “Support Limitations for Model Blocks” on page 3-19
- “Support Limitations for Stateflow Software Features” on page 3-21
- “Support Limitations for MATLAB for Code Generation” on page 3-25
- “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-28

## Check Model Compatibility

| In this section...                        |
|---|
| “Run Compatibility Check” on page 3-2     |
| “Compatibility Check Results” on page 3-3 |

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

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

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

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

The software supports a broad range of Simulink and Stateflow software capabilities in your models. However, there are capabilities that the product does not support, described in “Support Limitations for Simulink Software Features” on page 3-16 and “Support Limitations for Stateflow Software Features” on page 3-21.

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

### Run Compatibility Check

Before the software begins an analysis, it checks the compatibility of your model, and then creates a model representation. The model representation includes the model artifacts that are used during analysis. For more information, see “Model Representation for Analysis” on page 2-28.

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

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

For more information, see “Simulink Design Verifier Checks”.

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

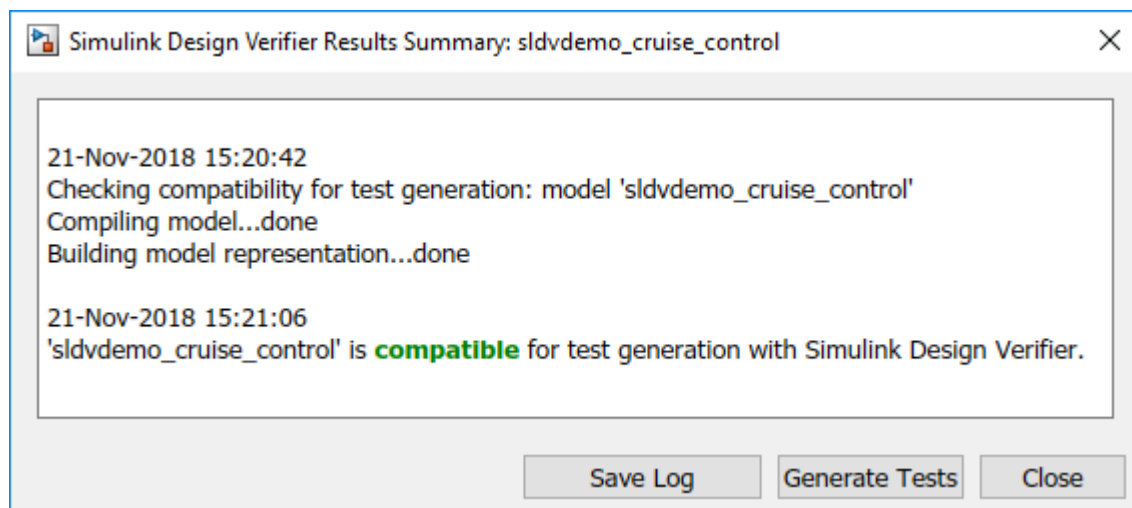
## Compatibility Check Results

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

- “Model Is Compatible” on page 3-3
- “Model Is Incompatible” on page 3-3
- “Model Is Partially Compatible” on page 3-5

### Model Is Compatible

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



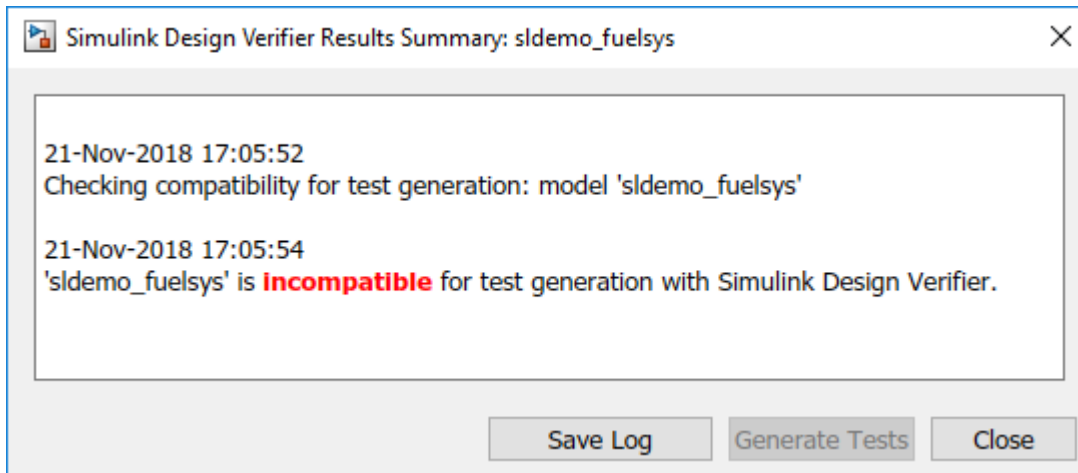
---

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

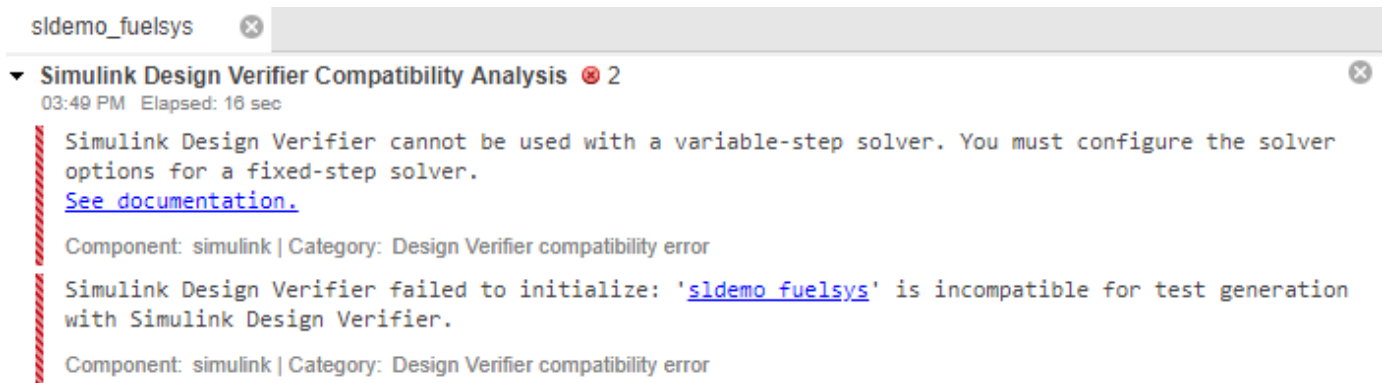
---

### Model Is Incompatible

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

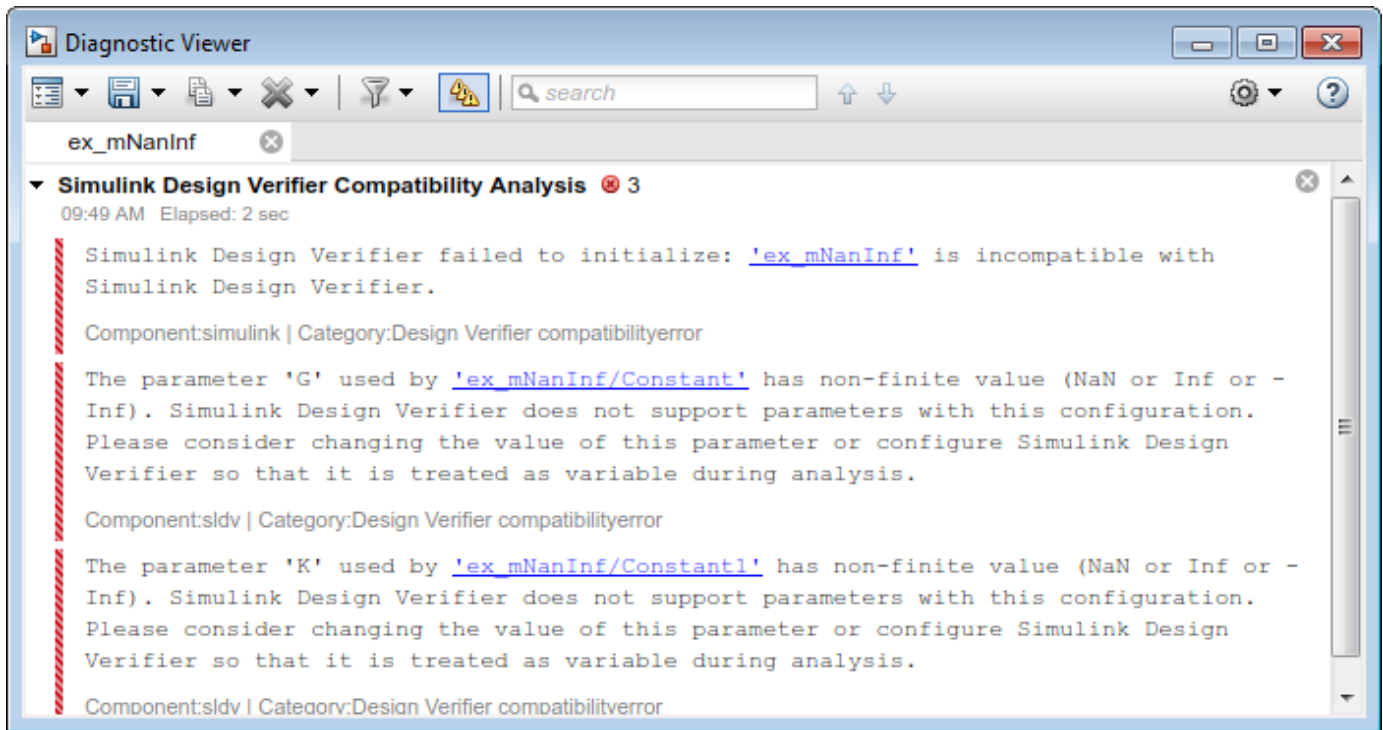


- If your model uses a variable-step solver, configure the solver Type to Fixed-step.



- 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. For more information, see "Nonfinite Data" on page 2-19.

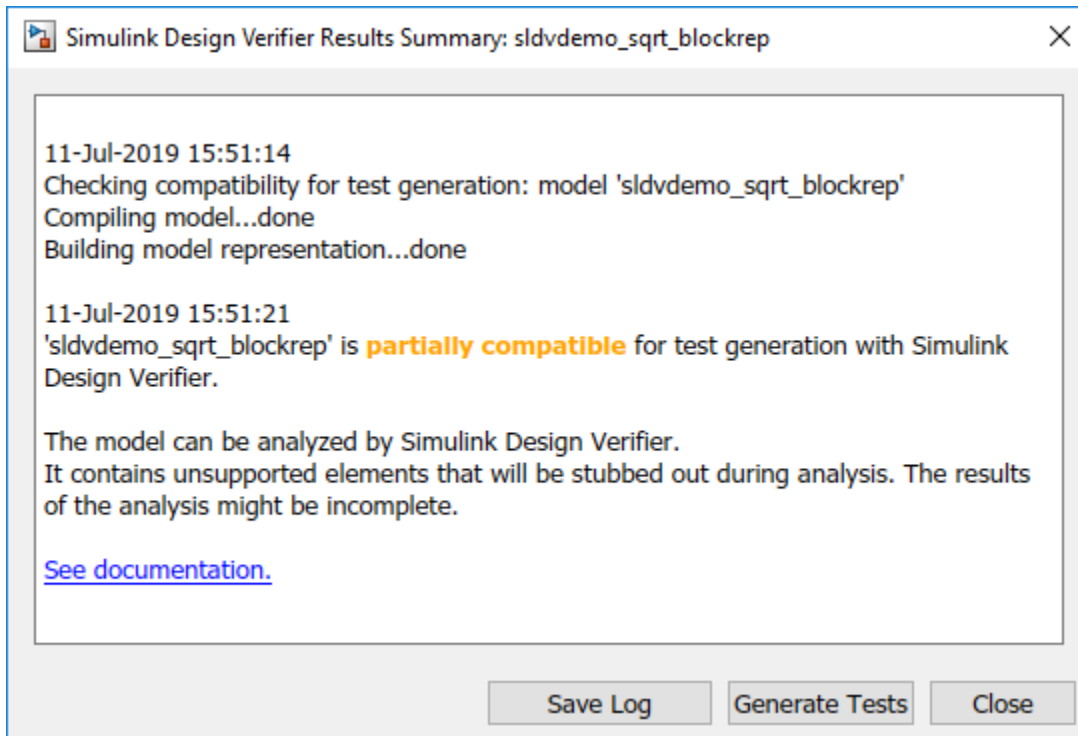




If your model is large and contains many subsystems, you can use the Test Generation Advisor to determine whether certain subsystems cause the incompatibility. For more information, see “Use Test Generation Advisor to Identify Analyzable Components” on page 7-24.

### Model Is Partially Compatible

A model is partially compatible if at least one model object in the model is incompatible. Simulink Design Verifier continues the analysis for partially compatible model by stubbing out the unsupported elements. By default, the “Automatic stubbing of unsupported blocks and functions” on page 15-13 option is set to On. For more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.



### See Also

“Overview of the Simulink Design Verifier Workflow” on page 1-19 | “Block Replacements for Unsupported Blocks” on page 4-7 | “Model Representation for Analysis” on page 2-28

# Supported and Unsupported Simulink Blocks in Simulink Design Verifier

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

- Supported
- Partially supported
- Not supported

If your model contains partially supported blocks, you can enable automatic stubbing. In order to improve the scalability of the analysis, automatic stubbing conservatively abstracts the block behavior. As a result, the analysis may not successfully analyze all the objectives. For more details about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.

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

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

## Additional Math and Discrete Library

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

## Commonly Used Blocks Library

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

## Continuous Library

| Block                           | Support Notes |
|---------------------------------|---------------|
| Derivative                      | Not supported |
| Integrator                      | Not supported |
| Integrator Limited              | Not supported |
| PID Controller                  | Not supported |
| PID Controller (2DOF)           | Not supported |
| Second-Order Integrator         | Not supported |
| Second-Order Integrator Limited | Not supported |
| 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**

| Block                          | Support Notes |
|--------------------------------|---------------|
| Delay                          | Supported     |
| Difference                     | Supported     |
| Discrete Derivative            | Supported     |
| Discrete Filter                | Supported     |
| Discrete FIR Filter            | Supported     |
| Discrete PID Controller        | Supported     |
| Discrete PID Controller (2DOF) | Supported     |
| Discrete State-Space           | Not supported |
| Discrete Transfer Fcn          | Supported     |
| Discrete Zero-Pole             | Not supported |
| Discrete-Time Integrator       | Supported     |
| Memory                         | Supported     |
| Tapped Delay                   | Supported     |
| Transfer Fcn First Order       | Supported     |
| Transfer Fcn Lead or Lag       | Supported     |
| Transfer Fcn Real Zero         | Supported     |
| Unit Delay                     | Supported     |
| Zero-Order Hold                | Supported     |

**Logic and Bit Operations Library**

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

**Lookup Tables Library**

| Block                         | Support Notes   |
|-------------------------------|---|
| Cosine                        | Supported   |
| Direct Lookup Table (n-D)     | Supported   |
| Interpolation Using Prelookup | Partially supported when: <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> or <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> |

| Block                | Support Notes  |
|----------------------|--|
| 1-D Lookup Table     | Partially supported when the <b>Interpolation method</b> or the <b>Extrapolation method</b> parameter is Cubic Spline.   |
| 2-D Lookup Table     | Not supported when the <b>Interpolation method</b> or the <b>Extrapolation method</b> parameter is Akima Spline.   |
| n-D Lookup Table     | Partially supported when: <ul style="list-style-type: none"> <li>The <b>Interpolation method</b> or the <b>Extrapolation method</b> parameter is Cubic Spline.</li> </ul> or <ul style="list-style-type: none"> <li>The <b>Interpolation method</b> parameter is Linear and the <b>Number of table dimensions</b> parameter is greater than 5.</li> </ul> Not supported when the <b>Interpolation method</b> or the <b>Extrapolation method</b> parameter is Akima Spline. |
| Lookup Table Dynamic | Supported  |
| Prelookup            | Supported  |
| Sine                 | Supported  |

### Math Operations Library

| Block                      | Support Notes   |
|----------------------------|---|
| Abs                        | Supported   |
| Add                        | Supported   |
| Algebraic Constraint       | Supported   |
| Assignment                 | Supported   |
| Bias                       | Supported   |
| Complex to Magnitude-Angle | Supported   |
| Complex to Real-Imag       | Supported   |
| Divide                     | Supported   |
| Dot Product                | Supported   |
| Find Nonzero Elements      | Not supported   |
| Gain                       | Supported   |
| Magnitude-Angle to Complex | Supported   |
| Math Function              | Supported. Support for pow function is limited to integer exponents only. |
| Matrix Concatenate         | Supported   |
| MinMax                     | Supported   |
| MinMax Running Resettable  | Supported   |
| Permute Dimensions         | Supported   |
| Polynomial                 | Supported   |

| <b>Block</b>              | <b>Support Notes</b>   |
|---------------------------|--|
| Product                   | Supported  |
| Product of Elements       | Supported  |
| Real-Imag to Complex      | Supported  |
| Reciprocal Sqrt           | Partially supported  |
| Reshape                   | Supported  |
| Rounding Function         | Supported  |
| Sign                      | Supported  |
| Signed Sqrt               | Partially supported  |
| Sine Wave Function        | Partially supported  |
| Slider Gain               | Supported  |
| Sqrt                      | Partially supported  |
| Squeeze                   | Supported  |
| Subtract                  | Supported  |
| Sum                       | Supported  |
| Sum of Elements           | Supported  |
| Trigonometric Function    | Supported if <b>Function</b> is sin, cos, or sincos, and <b>Approximation method</b> is CORDIC. Partially supported otherwise. |
| Unary Minus               | Supported  |
| Vector Concatenate        | Supported  |
| Weighted Sample Time Math | Supported  |

### **Model Verification Library**

The software supports all blocks in the Model Verification library.

### **Model-Wide Utilities Library**

| <b>Block</b>                | <b>Support Notes</b> |
|-----------------------------|----------------------|
| Block Support Table         | Supported            |
| DocBlock                    | Supported            |
| Model Info                  | Supported            |
| Timed-Based Linearization   | Not supported        |
| Trigger-Based Linearization | Not supported        |

### **Ports & Subsystems Library**

| <b>Block</b>         | <b>Support Notes</b> |
|----------------------|----------------------|
| Atomic Subsystem     | Supported            |
| Code Reuse Subsystem | Supported            |

| Block                           | Support Notes  |
|---------------------------------|--|
| Configurable Subsystem          | Supported  |
| Enable                          | Supported  |
| Enabled Subsystem               | <p>Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-23.</p> <p>Simulink Design Verifier treats Enabled Subsystems as short-circuited during test generation.</p>   |
| Enabled and Triggered Subsystem | <p>Not supported when the trigger control signal specifies a fixed-point data type.</p> <p>Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-23.</p> <p>Simulink Design Verifier treats Enabled and Triggered Subsystems as short-circuited during test generation.</p> |
| For Each                        | <p>Supported with the following limitations:</p> <ul style="list-style-type: none"> <li>• When For Each Subsystem contains one or more Simulink Design Verifier Test Condition, Test Objective, Proof Assumption, or Proof Objective blocks, not supported.</li> <li>• When the mask parameters of the For Each Subsystem are partitioned, not supported.</li> </ul>   |
| For Each Subsystem              | <p>Supported with the following limitations:</p> <ul style="list-style-type: none"> <li>• When For Each Subsystem contains one or more Simulink Design Verifier Test Condition, Test Objective, Proof Assumption, or Proof Objective blocks, not supported.</li> <li>• When the mask parameters of the For Each Subsystem are partitioned, not supported.</li> </ul>   |
| For Iterator Subsystem          | Supported  |
| Function-Call Feedback Latch    | Supported  |
| Function-Call Generator         | Supported  |
| Function-Call Split             | Supported  |
| Function-Call Subsystem         | <p>Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-23.</p>  |
| If                              | Parameter configurations are not supported. The analysis ignores parameter configurations that you specify for an If block.  |

| Block                            | Support Notes   |
|----------------------------------|---|
| If Action Subsystem              | Supported   |
| In Bus Element                   | Supported   |
| Inport                           | Supported   |
| Model                            | Supported except for the limitations described in “Support Limitations for Model Blocks” on page 3-19.  |
| Out Bus Element                  | Supported   |
| Outport                          | Supported   |
| Resettable Subsystem             | Supported   |
| Subsystem                        | Supported   |
| Variant Transitions in Stateflow | Supported.<br>Only the active variant is analyzed.  |
| Switch Case                      | Supported   |
| Switch Case Action Subsystem     | Supported   |
| Trigger                          | Supported   |
| Triggered Subsystem              | Not supported when the trigger control signal specifies a fixed-point data type.<br><br>Design range checks do not consider specified minimum and maximum values for blocks connected to the output port of the subsystem. For more information on design range checks, see “Check for Specified Minimum and Maximum Value Violations” on page 6-23.<br><br>Simulink Design Verifier treats Enabled Subsystems as short-circuited during test generation. |
| Variant Subsystem                | Not supported when the <b>Generate preprocessor conditionals</b> parameter is enabled.<br><br>Only the active variant is analyzed.  |
| While Iterator Subsystem         | Supported   |

#### Signal Attributes Library

The software supports all blocks in the Signal Attributes library.

#### Signal Routing Library

| Block             | Support Notes |
|-------------------|---------------|
| Bus Assignment    | Supported     |
| Bus Creator       | Supported     |
| Bus Selector      | Supported     |
| Data Store Memory | Supported     |
| Data Store Read   | Supported     |



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

### Sinks Library

| Block           | Support Notes |
|-----------------|---------------|
| Display         | Supported     |
| Floating Scope  | Supported     |
| Outport (Out1)  | Supported     |
| Out Bus Element | Supported     |
| Scope           | Supported     |
| Stop Simulation | Not supported |
| Terminator      | Supported     |
| To File         | Supported     |
| To Workspace    | Supported     |

### Sources Library

| Block                    | Support Notes       |
|--------------------------|---------------------|
| Band-Limited White Noise | Not supported       |
| Chirp Signal             | Partially supported |
| Clock                    | Supported           |

| Block                           | Support Notes   |
|---------------------------------|---|
| Constant                        | Supported unless <b>Constant value</b> is <code>inf</code> or <code>nan</code> (in which case, it is not supported).  |
| Counter Free-Running            | Supported   |
| Counter Limited                 | Supported   |
| Digital Clock                   | Supported   |
| Enumerated Constant             | Supported   |
| From File                       | Partially supported. When MAT-file data is stored in MATLAB <code>timeseries</code> format, not supported.  |
| From Workspace                  | Partially supported   |
| Ground                          | Supported   |
| Inport (In1)                    | Supported   |
| In Bus Element                  | Supported if <code>Simulink.Bus</code> type is defined for the In Bus Element.  |
| Pulse Generator                 | Supported   |
| Ramp                            | Supported   |
| Random Number                   | Not supported   |
| Repeating Sequence              | Partially supported   |
| Repeating Sequence Interpolated | Partially supported   |
| Repeating Sequence Stair        | Supported   |
| Signal Editor                   | Not supported   |
| Signal Generator                | Partially supported if wave form is <code>sine</code> . Supported if wave form is <code>square</code> . Not supported if wave form is <code>random</code> . |
| Sine Wave                       | Partially supported   |
| Step                            | Supported   |
| Uniform Random Number           | Not supported   |

### User-Defined Functions Library

| Block                       | Support Notes  |
|-----------------------------|--|
| C Function                  | Partially supported. The C Function block is stubbed out during the Simulink Design Verifier analysis.   |
| C Caller                    | Supported.   |
| Initialize Function         | <ul style="list-style-type: none"> <li>Not Supported for Initialize function containing Parameter Writer blocks.</li> <li>Not supported as a target for subsystem analysis.</li> </ul> |
| Interpreted MATLAB Function | Partially supported  |
| Level-2 MATLAB S-Function   | For limitations, see “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-28.   |
| MATLAB Function             | For limitations, see “Support Limitations for MATLAB for Code Generation” on page 3-25.  |

| Block              | Support Notes  |
|--------------------|--|
| MATLAB System      | <ul style="list-style-type: none"> <li>• Decision, Condition and MCDC Coverage objectives are supported in Test Generation. Enhanced MCDC, Relational Boundary and Custom Test objectives are not supported.</li> <li>• Custom Proof objectives are not supported in Property Proving.</li> <li>• For further limitations, see “Support Limitations for MATLAB for Code Generation” on page 3-25.</li> </ul> <p>Logical expressions within assignment statements are not analyzed for coverage objectives.</p> |
| Reset Function     | Not supported  |
| S-Function Builder | For limitations, see “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-28.   |
| Simulink Function  | <ul style="list-style-type: none"> <li>• For export-function models, see “Analyze Export-Function Models” on page 2-12.</li> <li>• Global Simulink functions within a non export-function model reference are not supported.</li> </ul>  |
| Terminate Function | <p>Partially supported.</p> <ul style="list-style-type: none"> <li>• The behaviour of Terminate function is ignored and is replaced by an empty function during the analysis.</li> <li>• Not supported as a target for subsystem analysis.</li> </ul>  |
| Observer Reference | Supported with limitations. See “Isolate Verification Logic with Observers” on page 12-29.   |
| Simscape Library   | Not supported  |

## Support Limitations for Simulink Software Features

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

| Not Supported                             | Description   |
|---|---|
| Variable-step solvers                     | <p>The software supports only fixed-step solvers.</p> <p>For more information, see “Fixed Step Solvers in Simulink”.</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 supports model callback functions only 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 Loop Concepts”.</p>   |
| Masked subsystem initialization functions | <p>The software does not support models whose masked subsystem initialization:</p> <ul style="list-style-type: none"> <li>• Modifies any attribute of any workspace parameter.</li> <li>• Deletes or creates blocks.</li> </ul>   |

| Not Supported                    | Description  |
|----------------------------------|--|
| Variable-size signals            | <p>The software supports test generation for models with bounded variable-size signals. For more information on how to generate test cases when input signals are of variable-size, see “Achieve Coverage in Models with Variable-Size Inputs” on page 9-24.</p> <p>In addition, the following are the limitations for analysis:</p> <ol style="list-style-type: none"> <li><b>1</b> Relational boundary coverage objectives</li> <li><b>2</b> Enhanced MCDC coverage objectives</li> <li><b>3</b> Models with variable-size signals at root level input port</li> <li><b>4</b> Models with variable-size signals with maximum size 1</li> </ol> <hr/> <p><b>Note</b></p> <ul style="list-style-type: none"> <li>• Coverage objectives of single port logical and min-max blocks with variable size signals are not considered.</li> <li>• The analysis is performed under the assumptions that at any step, all the variable-size inputs of a block will have same size.</li> </ul>   |
| Multiword fixed-point data types | <p>The software does not support multiword fixed-point data types larger than 128 bits.</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> <li>• Simulink Design Verifier assumes zero start time for analysis and generates signal data that begins at zero. Zero start time might impact the reporting of the objective status. For example, in the test generation analysis, the software might report some objectives as <b>Undecided with Testcases</b>. For more information, see “Simulation Basics”.</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>Relational operator</b> parameter is <code>isFinite</code>, the output is always 1.</li> <li>• If the <b>Relational 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. If a model contains test condition or test objective blocks and no output ports are present in the model, then nominal test cases will be generated.   |
| Large floating-point constants outside the range $[-\text{realmax}/2, \text{realmax}/2]$ | The use of large floating-point constants can cause out of memory errors or substantial loss of precision. Avoid using such constants if possible.   |
| Symbolic Dimensions  | The software does not support symbolic dimensions for test generation, property proving, or design error detection.  |
| Simulink Strings   | Models that contain blocks with string data types as block parameters are not supported. For more information, see “Simulink Strings”.   |
| Parameter Tuning   | The software does not support parameter tuning for the parameters that are defined in the Model Workspace.   |
| Row-major Algorithms   | The software does not support models that contain MATLAB System blocks that use <code>coder.rowMajor</code> directive. For more information see, “Use algorithms optimized for row-major array layout”.  |

## Support Limitations for Model Blocks

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

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

For more information, see “Reference Protected Models from Third Parties”.

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

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

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

Simulink tries to eliminate the artificial algebraic loop during simulation.

- 3 Simulate the model.
- 4 Simulink will remove the algebraic loop if possible. If Simulink cannot eliminate the artificial algebraic loop, highlight the location of the algebraic loop by opening the **Modeling** tab and, in the **Compile** section, clicking **Update Model**.
- 5 Eliminate the artificial algebraic loop so that the software can analyze the model. Break the loop with Unit Delay blocks so that the execution order is predictable.

---

**Note** For more information, see “Algebraic Loop Concepts”.

---

- The parent model and the referenced model have mismatched data type override settings. The data type override setting of the parent model and its referenced models must be the same, unless the data type override setting of the parent model is `Use local settings`. You can configure data type override settings to simulate a model that specifies fixed-point data types. Using this setting, the software temporarily overrides data types with floating-point data types during simulation.

```
set_param('MyModel','DataTypeOverride','Double')
```

For more information, see `set_param`.

To observe the true behavior of your model, set the data type override parameter to `UseLocalSettings` or `Off`.

```
set_param('MyModel','DataTypeOverride','Off')
```

- The referenced model is a Model block with virtual buses at input ports, and the signals in the bus do not all have the same sample time at compilation. To make the model compatible with Simulink

Design Verifier analysis, convert the virtual bus to a nonvirtual bus, or specify an explicit sample time for the port.

- When you run the analysis on Model block, then the code generated as a top model is not supported.



## Support Limitations for Stateflow Software Features

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

### In this section...

“ml Namespace Operator, ml Function, ml Expressions” on page 3-21

“C or C++ Operators” on page 3-21

“C Math Functions” on page 3-21

“Atomic Subcharts That Call Exported Graphical Functions Outside a Subchart” on page 3-22

“Atomic Subchart Input and Output Mapping” on page 3-22

“Recursion and Cyclic Behavior” on page 3-22

“Custom C/C++ Code” on page 3-23

“Textual Functions with Literal String Arguments” on page 3-24

### ml Namespace Operator, ml Function, ml Expressions

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

### C or C++ Operators

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

### C Math Functions

The software supports calls to the following C math functions:

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

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

For information about C math functions in Stateflow, see “Call C Library Functions in C Charts” (Stateflow).

---

**Note** For details about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.

---

## Atomic Subcharts That Call Exported Graphical Functions Outside a Subchart

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

---

**Note** For information about exported functions, see “Export Stateflow Functions for Reuse” (Stateflow).

---

## Atomic Subchart Input and Output Mapping

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

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

## Recursion and Cyclic Behavior

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

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

---

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

---

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

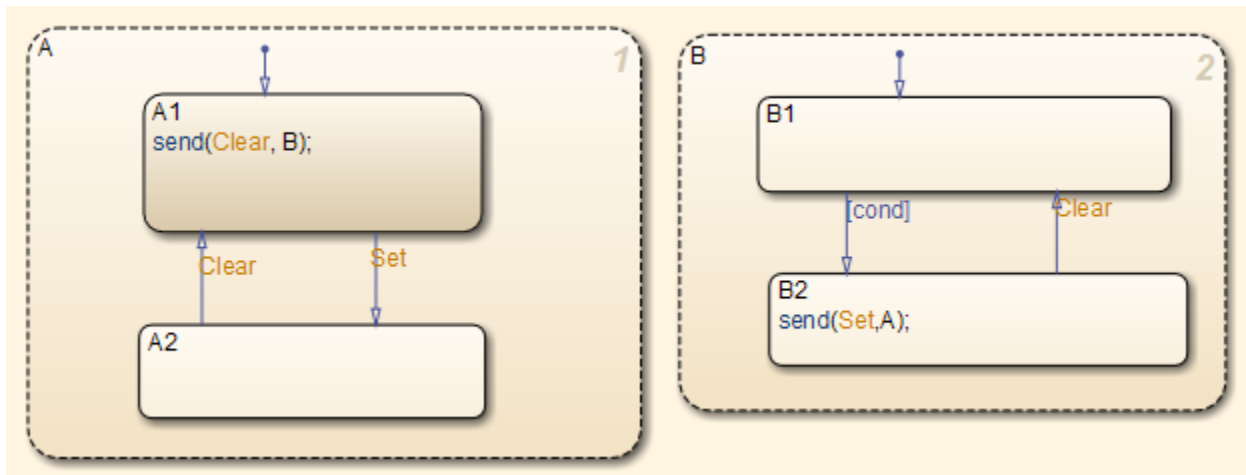
---

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

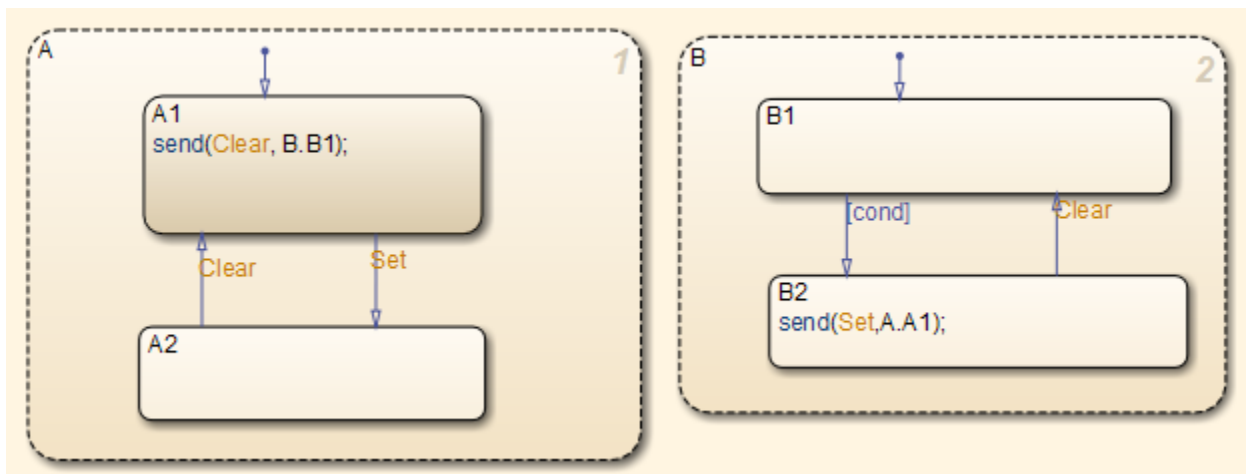
---

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

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



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



**Note** For information about the benefits of directed event broadcasts, see “Broadcast Local Events to Synchronize Parallel States” (Stateflow).

## Custom C/C++ Code

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

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

## **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

|   |
|---|
| <b>In this section...</b>   |
| “Unsupported MATLAB for Code Generation Features” on page 3-25                      |
| “Support Limitations for MATLAB for Code Generation Library Functions” on page 3-25 |

### Unsupported MATLAB for Code Generation Features

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

| Not Supported       | Description   |
|---------------------|---|
| Characters          | The software does not support characters, which MATLAB for code generation allows.                      |
| C functions         | The software does not support calls to external C functions, which MATLAB for code generation allows.   |
| Extrinsic functions | The software supports extrinsic functions only when they do not affect the output of a MATLAB function. |

### Support Limitations for MATLAB for Code Generation Library Functions

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

If your model contains unsupported functions, you can turn on automatic stubbing, which considers the interface of the unsupported functions, but not their behavior. However, if any of the unsupported functions affect the simulation outcome, the analysis might achieve only partial results. For details about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.

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

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

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

For the complete listing of available functions, see “Functions and Objects Supported for C/C++ Code Generation”.

| Function Category           | Support Notes  |
|-----------------------------|----------------|
| Aerospace Toolbox functions | Not supported. |

| <b>Function Category</b>               | <b>Support Notes</b>                      |  |
|--|---|--|
| Arithmetic operator functions          | Supported with the following limitations: |  |
|  | <code>mldivide (\)</code>                 | Supported.   |
|  | <code>mpower (^)</code>                   | Supports only integer exponents. Otherwise partially supported.    |
|  | <code>mrdivide (/)</code>                 | Supported.   |
|  | <code>power (.^)</code>                   | Supports integer exponents. Float exponents partially supported.   |
| Bit-wise operation functions           | —   |  |
| Casting functions                      | Supported with the following limitations: |  |
|  | <code>char</code>                         | Not supported.   |
|  | <code>typecast</code>                     | Not supported.   |
| Communications Toolbox™ functions      | Not supported.                            |  |
| Complex number functions               | Partially supported.                      |  |
| Computer Vision Toolbox™ functions     | Not supported.                            |  |
| Data type functions                    | —   |  |
| Derivative and Integral functions      | Not supported.                            |  |
| Discrete math functions                | —   |  |
| Error handling functions               | Supported with the following limitations: |  |
|  | <code>assert</code>                       | Supported, but does not behave like a Proof Objective block.       |
| Exponential functions                  | Supported.                                |  |
| Filtering and convolution functions    | Supported with the following limitations: |  |
|  | <code>detrend</code>                      | Supported if argument is a scalar. Otherwise, partially supported. |
| Fixed-Point Designer functions         | Supported.                                |  |
| Histogram functions                    | Not supported.                            |  |
| Image Processing Toolbox™ functions    | Not supported.                            |  |
| Input and output functions             | —   |  |
| Interpolation and computation geometry | Supported with the following limitations: |  |
|  | <code>cart2pol</code>                     | Partially supported.   |
|  | <code>cart2sph</code>                     | Partially supported.   |
|  | <code>pol2cart</code>                     | Partially supported.   |
|  | <code>sph2cart</code>                     | Partially supported.   |
| Linear algebra                         | Not supported.                            |  |
| Logical operator functions             | —   |  |
| MATLAB Compiler™ functions             | Not supported.                            |  |
| Matrix and array functions             | Supported with the following limitations: |  |
|  | <code>angle</code>                        | Partially supported.   |

| Function Category                     | Support Notes                             |   |
|---------------------------------------|---|---|
|                                       | cond                                      | Partially supported.  |
|                                       | det                                       | Supported.  |
|                                       | eig                                       | Partially supported.  |
|                                       | inv                                       | Supported.  |
|                                       | invhilb                                   | Not supported.  |
|                                       | logspace                                  | Partially supported.  |
|                                       | lu  | Supported.  |
|                                       | norm                                      | Supported when invoked using the syntax <code>norm(A,p)</code> where <code>p</code> is either <code>1</code> or <code>inf</code> . Otherwise partially supported. |
|                                       | normest                                   | Partially supported.  |
|                                       | pinv                                      | Partially supported.  |
|                                       | planerot                                  | Partially supported.  |
|                                       | qr  | Partially supported.  |
|                                       | rank                                      | Partially supported.  |
|                                       | rcond                                     | Supported.  |
|                                       | subspace                                  | Partially supported.  |
| Nested functions                      | 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                                      | Partially supported.  |
|                                       | randn                                     | Partially supported.  |
| Specialized math                      | Not supported.                            |   |
| Statistical functions                 | —   |   |
| String functions                      | Supported with the following limitations: |   |
|                                       | char                                      | Not supported.  |
|                                       | ischar                                    | Not supported.  |
| Trigonometric functions               | Not supported.                            |   |

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

### In this section...

“Enabling S-Functions in Simulink Design Verifier” on page 3-28

“Support Limitations for S-Functions and C/C++ Code” on page 3-28

“Handle Volatile Variables as Normal Variables” on page 3-29

“Considerations for Enabling S-Functions and C/C++ Code in Simulink Design Verifier” on page 3-29

“Source Code Protection” on page 3-29

### Enabling S-Functions in Simulink Design Verifier

Simulink Design Verifier supports test case generation for code generated with Embedded Coder®. Simulink Design Verifier also supports error detection, test case generation, and property proving for S-Functions that:

- The Legacy Code Tool generates, with `def.Options.supportCoverageAndDesignVerifier` set to true.
- The S-Function Builder generates, with **Enable support for Design Verifier** selected on the **Build Info** tab of the S-Function Builder dialog box.
- The function `slcovmex` compiles, with the option `-sldv` passed to the function when compiling the S-function.

For more information on the three approaches, see “About C MEX S-Functions”.

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

- Simulink Design Verifier does not support S-Functions or C/C++ code containing:
  - Continuous states. Simulink Design Verifier does not analyze such code.
  - Zero-crossing functions. Simulink Design Verifier ignores such code during analysis.
  - Constants that describe INF or NaN objects. Simulink Design Verifier considers such code as containing floating-point overflow errors. Although Simulink Design Verifier analysis cannot determine the type of overflow error for such cases, the analysis can determine which lines of code introduce the incompatibility. Polyspace® can provide more information on why your code contains floating-point overflow errors.
- You must specify that the signal elements entering the ports of S-Functions compiled with `slcovmex` are contiguous. Use the `SimStruct` function `ssSetInputPortRequiredContiguous`.

Simulink Design Verifier supports the following design errors for S-Function and C/C++ code:

- Dead logic including active logic.
- Array out of bounds. This includes pointer out of bounds in case of C/C++.
- Division-by-Zero.



## Handle Volatile Variables as Normal Variables

Simulink Design Verifier allows the option for volatile variables to be stubbed or handled as normal variables. When you select the **Ignore the volatile qualifier** parameter, volatile elements will be treated in the same as the non-volatile elements. Deselecting the **Ignore the volatile qualifier** will revert to the previous behavior of stubbing access to volatile elements.

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

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

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

- If Simulink Design Verifier cannot determine the size of arrays in your code (for instance for arrays that are dynamically allocated with non-constant size), Simulink Design Verifier assumes an upper bound for the array. Ensure that the given upper bound is appropriate.
- If you do not enable Simulink Design Verifier support for an S-function, Simulink Design Verifier stubs the S-function. With S-function support enabled, Simulink Design Verifier analyzed the content of the S-function to get more detailed information. Sometimes, Simulink Design Verifier internally stubs the S-function. Internal stubs can be the result of different C/C++ constructs, such as:
  - Calls to library functions (the library function is replaced by a stub).
  - Complex pointer operations.
  - Casts to or from incompatible or unknown pointer types.

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

## Source Code Protection

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

## See Also

“Configuring S-Function for Test Case Generation” on page 7-109 | “Generate Test Cases for Embedded Coder Generated Code” on page 7-28



# Working with Block Replacements

---

- “What Is Block Replacement?” on page 4-2
- “Built-In Block Replacements” on page 4-4
- “Template for Block Replacement Rules” on page 4-6
- “Block Replacements for Unsupported Blocks” on page 4-7

## What Is Block Replacement?

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

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

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

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

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

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

- Work around an incompatibility, such as the presence of unsupported blocks in your model.
- Customize a block for analysis, such as:
  - Adding constraints to its input signals
  - Adding objectives to its output signals
  - Eliminating the contents of a subsystem or Model block to simplify your analysis

---

**Note** You can use automatic stubbing as an alternative to block replacements to resolve incompatibilities. Automatic stubbing replaces unsupported blocks with elements that have the same interface. For more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.

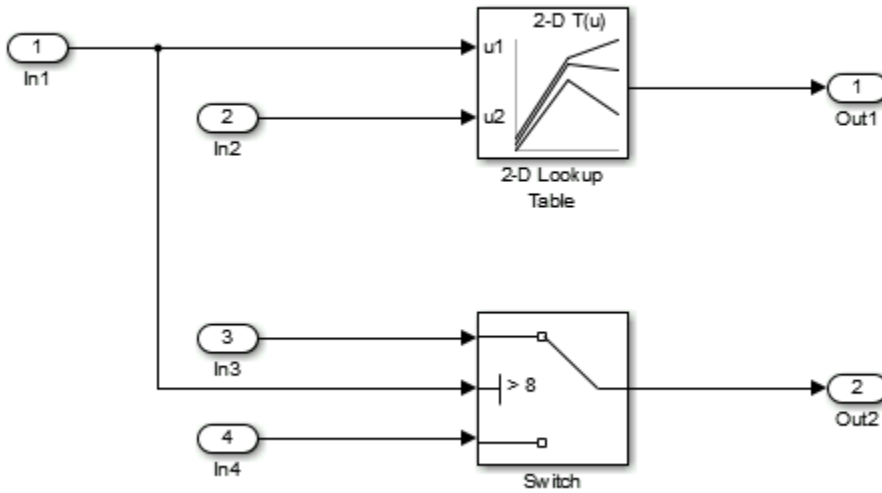
---

## Block Replacement Effects on Test Generation

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

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

In this example, the breakpoint range of u1 in the 2-D Lookup Table is 5–7. The switch threshold 8 falls outside the u1 lookup table range.

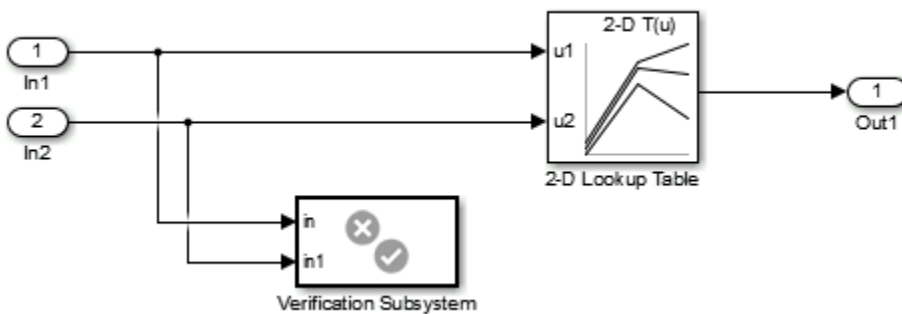


Tests generated without replacing the 2D Lookup Table satisfy two objectives: that the trigger is not greater than the Switch block threshold 8, and that the trigger is greater than the Switch block threshold 8.

| # | Type     | Model Item             | Description   | Analysis Time (sec) | Test Case         |
|---|----------|------------------------|---|---------------------|-------------------|
| 1 | Decision | <a href="#">Switch</a> | trigger > threshold false (output is from 3rd input port) | 0                   | <a href="#">1</a> |

### Objectives Satisfied

The blkrep\_rule\_lookup2D\_normal.m block replacement rule replaces the 2D Lookup Table with a masked subsystem containing the 2D Lookup Table and a MATLAB Function block.



The MATLAB Function block constrains the analysis within the breakpoint bounds of the table.

## 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.

| File Name                                   | Description  |
|---|--|
| <code>blkrep_rule_lookup_normal.m</code>    | 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>Breakpoints</b> parameter.  |
| <code>blkrep_rule_lookup2D_normal.m</code>  | 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.   |
| <code>blkrep_rule_mpswitch2_normal.m</code> | A rule that adds a Test Condition/Proof Assumption block to the control input port of Multiport Switch blocks whose <b>Number of data ports</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). |
| <code>blkrep_rule_mpswitch3_normal.m</code> | A rule that adds a Test Condition/Proof Assumption block to the control input port of Multiport Switch blocks whose <b>Number of data ports</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). |
| <code>blkrep_rule_mpswitch4_normal.m</code> | A rule that adds a Test Condition/Proof Assumption block to the control input port of Multiport Switch blocks whose <b>Number of data ports</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). |
| <code>blkrep_rule_mpswitch5_normal.m</code> | A rule that adds a Test Condition/Proof Assumption block to the control input port of Multiport Switch blocks whose <b>Number of data ports</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). |
| <code>blkrep_rule_switch_normal.m</code>    | 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_switch_nonvir_normal.m               | A rule that replaces Switch blocks having non-virtual bus inputs with an implementation that converts non-virtual bus inputs to virtual bus inputs. This implementation includes test objectives and requires that each switch position be exercised when the values of the first and third input ports are different.  |
| blkrep_rule_selector<br>IndexVecPort_normal.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 <code>Index vector (port)</code> . 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<br>StartingIdxPort_normal.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 <code>Starting index (port)</code> . 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`

### Template for Block Replacement Rules

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

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

To create a block replacement rule, make a copy of the template and edit the copy to implement the desired behavior for the rule you are creating. The comments in the template provide hints about how to use each section.

Block replacement rules have the following restrictions:

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



## Block Replacements for Unsupported Blocks

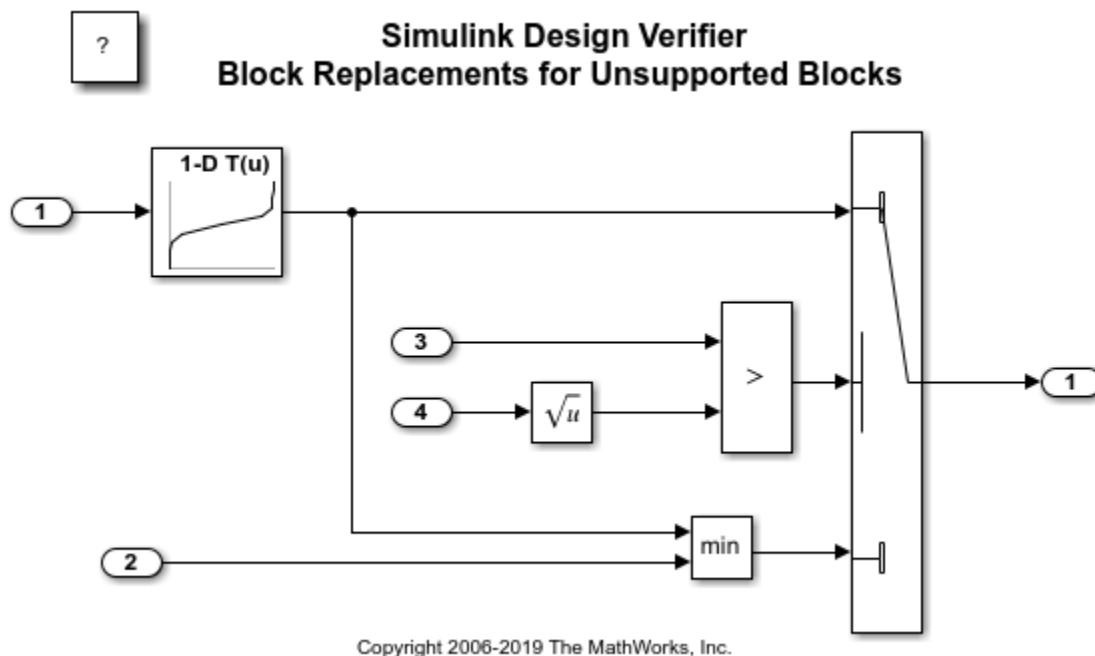
This example shows how to use Simulink® Design Verifier™ functions to replace unsupported blocks and how to customize test vector generation for specific requirements.

### Model with an Unsupported Block

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

1. Achieve 100% lookup table coverage.
2. Test vectors demonstrate each Switch block position when the values of its first and third input ports differ.

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



### Checking Model Compatibility

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

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

```
04-Mar-2023 00:17:36
Checking compatibility for test generation: model 'sldvdemo_sqrt_blockrep'
Compiling model...done
Building model representation...done
```

04-Mar-2023 00:17:41

'sldvdemo\_sqrt\_blockrep' is partially compatible for test generation with Simulink Design Verifier.

The model can be analyzed by Simulink Design Verifier.

It contains unsupported elements that will be stubbed out during analysis. The results of the analysis are shown in the Diagnostic Viewer. See the Diagnostic Viewer for more details on the unsupported elements.

### Creating a Custom Block Replacement Rule to Work Around the Incompatibility

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

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

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

```
function rule = sldvdemo_custom_blkrep_rule_sqrt

    rule = SldvBlockReplacement.blockrepreule;
    rule.fileName = mfilename;

    rule.blockType = 'Sqrt';

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

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

    rule.isReplaceableCallback = @replacementTestFunction;
end

function out = replacementTestFunction(blockH)

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

        portDataTypes = get_param(blockH,'CompiledPortDataTypes');

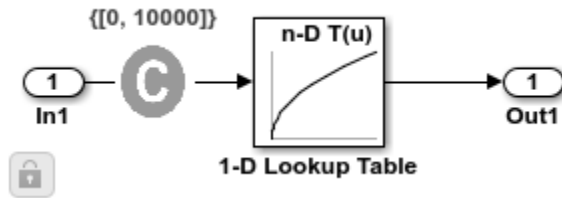
        out = any(strcmp(portDataTypes.Inport,{'double','single'})) && ...
            strcmp(portDataTypes.Inport,portDataTypes.Outport);
    end
end
```

```

end
end

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

```



### Configuring Simulink® Design Verifier™ Options for Block Replacement

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

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

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

```

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

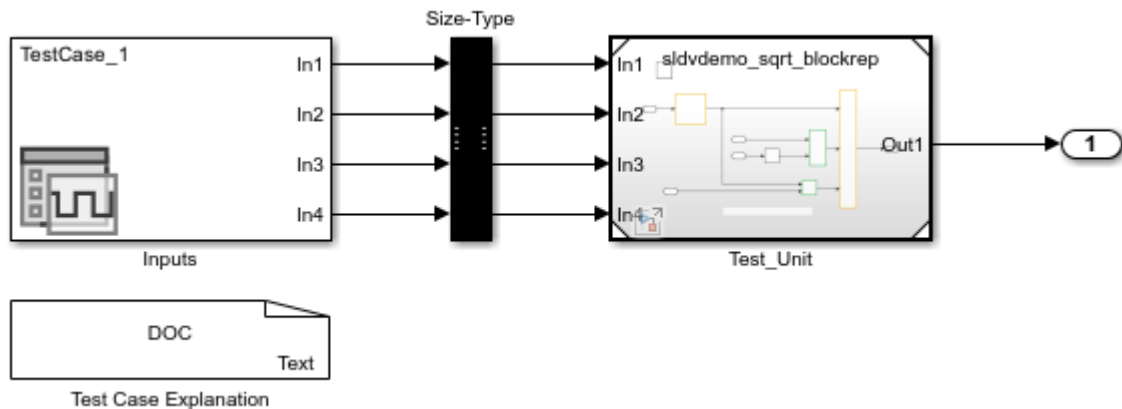
opts.SaveHarnessModel = 'on';
opts.ModelReferenceHarness = 'on';
opts.SaveReport = 'on';

```

### Executing Test Generation with Block Replacements

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

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



### Executing Tests in the Harness Model

Enable the lookup table coverage metric and then run the test cases using the harness model. You can also execute the suite of tests by clicking the "Run all" button on the Signal Builder dialog box after enabling lookup table coverage from the Configuration Parameters dialog. In the **Coverage** tab, select **Enable coverage analysis** and then select **Coverage metrics > Other metrics > Lookup table**.

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

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

### Clean Up

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

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

# Specifying Parameter Configurations

---

- “Parameter Configuration for Analysis” on page 5-2
- “Use Parameter Table” on page 5-7
- “Specify Parameter Configuration for Structure or Bus Parameters” on page 5-12
- “Specify Parameter Configuration for Full Coverage” on page 5-17
- “Store Parameter Constraints in MATLAB Code Files” on page 5-26
- “Use Parameter Configuration File” on page 5-29
- “Automatically Infer Parameter Specification” on page 5-32
- “Determine from Generated Code” on page 5-36
- “Using Command Line Functions to Support Changing Parameters” on page 5-39
- “Generate Parameters Values” on page 5-45
- “Extend Existing Test Cases After Applying Parameter Configurations” on page 5-46

## Parameter Configuration for Analysis

| In this section...  |
|---|
| “What is Parameter Configuration for Analysis?” on page 5-2                               |
| “Specify Parameter Constraints for Models Using Referenced Configuration Set” on page 5-3 |
| “Data Types in Parameter Configurations” on page 5-4                                      |
| “Parameters in Variant Blocks” on page 5-5  |

### What is Parameter Configuration for Analysis?

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

You can achieve this by placing a constraint on a parameter in your model, during analysis that parameter takes only your specified constraint value or values. A group of constraints on parameters in the same model is also called a parameter configuration.

This allows you to, for example:

- Extend the results of design error detection or property proving analysis to consider the impact of additional parameter values.
- Generate comprehensive test cases for situations in which parameter values must vary to achieve more complete coverage results. For more information, see “Specify Parameter Configuration for Full Coverage” on page 5-17.

Simulink Design Verifier provides the following workflows to specify parameter configuration:

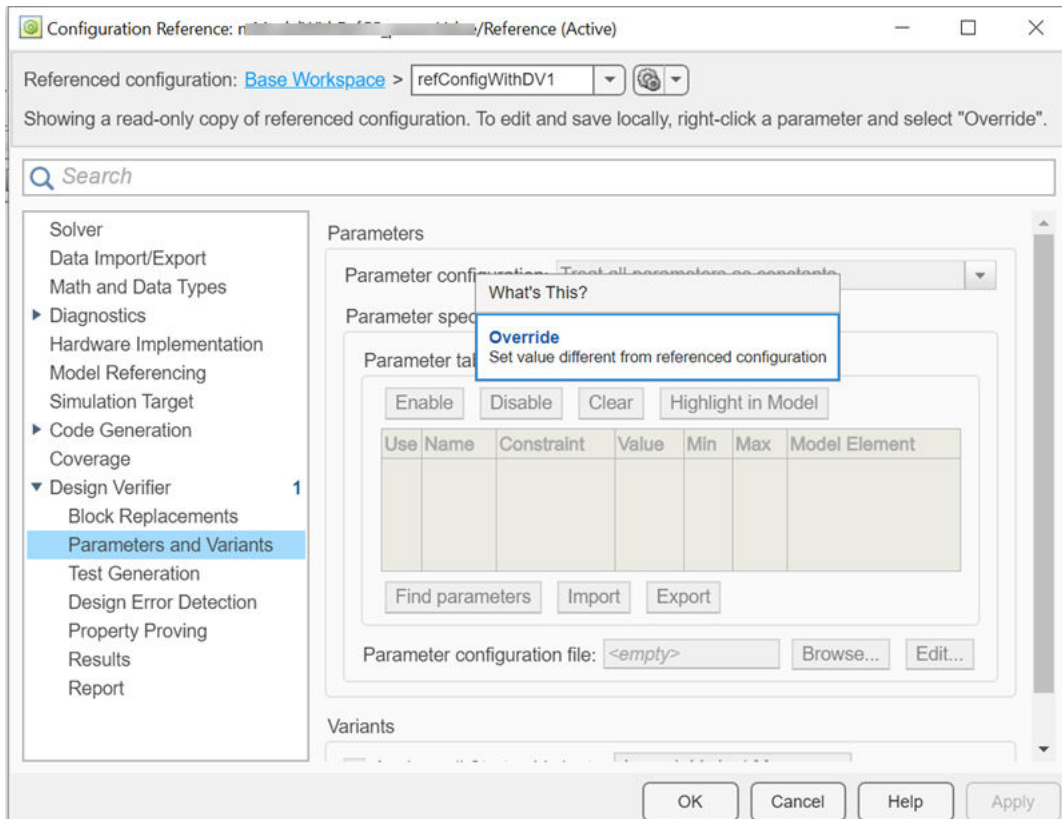
## Parameter Configuration Workflows

| Parameter Configuration                     | How to Select Parameters Constraints?  |
|---|--|
| Treat all parameters as constants           | Retains the initial value for all parameters during the analysis. Thus, analysis considers all parameters as constants.  |
| Automatically infer parameter specification | <p>For each parameter, the minimum or maximum value configured in <code>Simulink.Parameter</code> object is used as the parameter configuration for analysis.</p> <p>When test generation target is <b>Model</b>, Simulink Design Verifier selects as many parameters as possible for parameter configuration.</p> <p>When test generation target is <b>Code Generated as Top Model</b> or <b>Code Generated as Model Reference</b>, parameters whose value can be changed in the generated code are selected for parameter configuration. See “Automatically Infer Parameter Specification” on page 5-32.</p> |
| Determine from generated code               | <p>Parameters whose value can be changed in the generated code are selected for parameter configuration during the analysis.</p> <p>For such parameters, the minimum or maximum value from <code>Simulink.Parameter</code> object is used as the parameter configuration for analysis. See “Determine from Generated Code” on page 5-36.</p>   |
| Use parameter table                         | Parameters and constraints in the parameter table must be specified. See “Use Parameter Table” on page 5-7   |
| Use parameter configuration files           | Parameters and constraints in the input file must be specified. See “Use Parameter Configuration File” on page 5-29  |

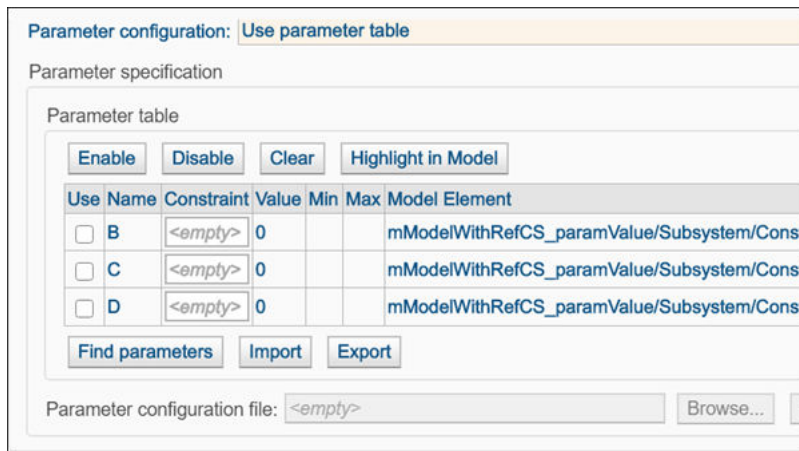
## Specify Parameter Constraints for Models Using Referenced Configuration Set

If your model uses reference configuration set, you can use **Override** capability to specify parameter constraints. Before you work with parameter table in a referenced configuration set, follow these steps:

- 1 Open the model.
- 2 On the **Design Verifier** tab, click **Settings** to open the Configuration parameters window. The Configuration parameters window shows the Configuration reference for the model.
- 3 Click on **Parameters and Variants** from **Design Verifier** pane.
- 4 To edit and save the constraints locally, right-click on the **Parameters configuration** and select **Override**.



- Similarly, override the values in **Parameter table**. Right-click in the Parameter table area and select **Override** and specify the values for the model by clicking on **Find parameters**.



- The Parameter table area highlights the override settings for the model.

You can perform the analysis after specifying the values for the parameter table. For more information on how to specify constraint values, see “Use Parameter Table” on page 5-7.

### Data Types in Parameter Configurations

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



- “Parameters Converted to Fixed Point in the Model” on page 5-5
- “Parameters Defined as Simulink.Parameter and Referenced by Multiple Locations” on page 5-5
- “Complex Data as Parameters not Supported” on page 5-5
- “Tuning Array of Structure or Bus Data types are not supported” on page 5-5

### **Parameters Converted to Fixed Point in the Model**

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

### **Parameters Defined as Simulink.Parameter and Referenced by Multiple Locations**

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

### **Complex Data as Parameters not Supported**

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

### **Tuning Array of Structure or Bus Data types are not supported**

Simulink Design Verifier does not support tuning array of structure or bus data types during the analysis.

## **Parameters in Variant Blocks**

Parameters can be used to select variants in the model using variant blocks such as Variant Subsystem, Variant Source and Variant Sinks.

Simulink Design Verifier supports only active variant for blocks where **Variant activation time** parameter is not set to `startup`. For blocks where **Variant activation time** is `startup`, Simulink Design Verifier analyzes all variants when you select **Analyze all Startup Variants** under **Design Verifier > Parameters and Variants** in Configuration Parameters dialog box.

To analyze a model that contains variant constraints, open the **Launch Variant Manager**. Use the Variant Manager to run predefined configurations for a model, and use the model under any of the configurations. The Simulink Design Verifier analysis report includes the results information about the variants blocks.

Simulink Design Verifier does not support block replacement in models that contain model reference with `startup` variants.

To perform the Simulink Design Verifier analysis on variant blocks with **Variant activation time** set to `startup`, see “Verify and Validate Variant Models with Startup Activation Time”.

## **See Also**

“Variant Manager for Simulink” | “Variant Activation Time for Variant Blocks”

### **More About**

- “Specify Parameter Configuration for Full Coverage” on page 5-17
- “Extend Existing Test Cases After Applying Parameter Configurations” on page 5-46

## Use Parameter Table

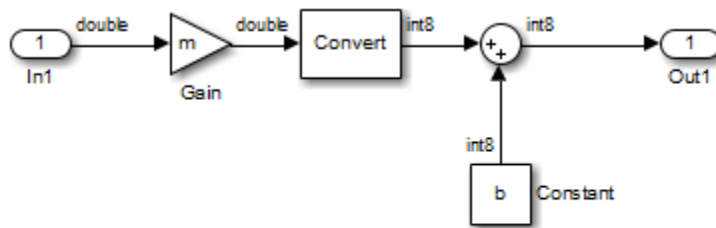
### In this section...

“Find Parameters” on page 5-8

“Edit Parameter Constraints” on page 5-10

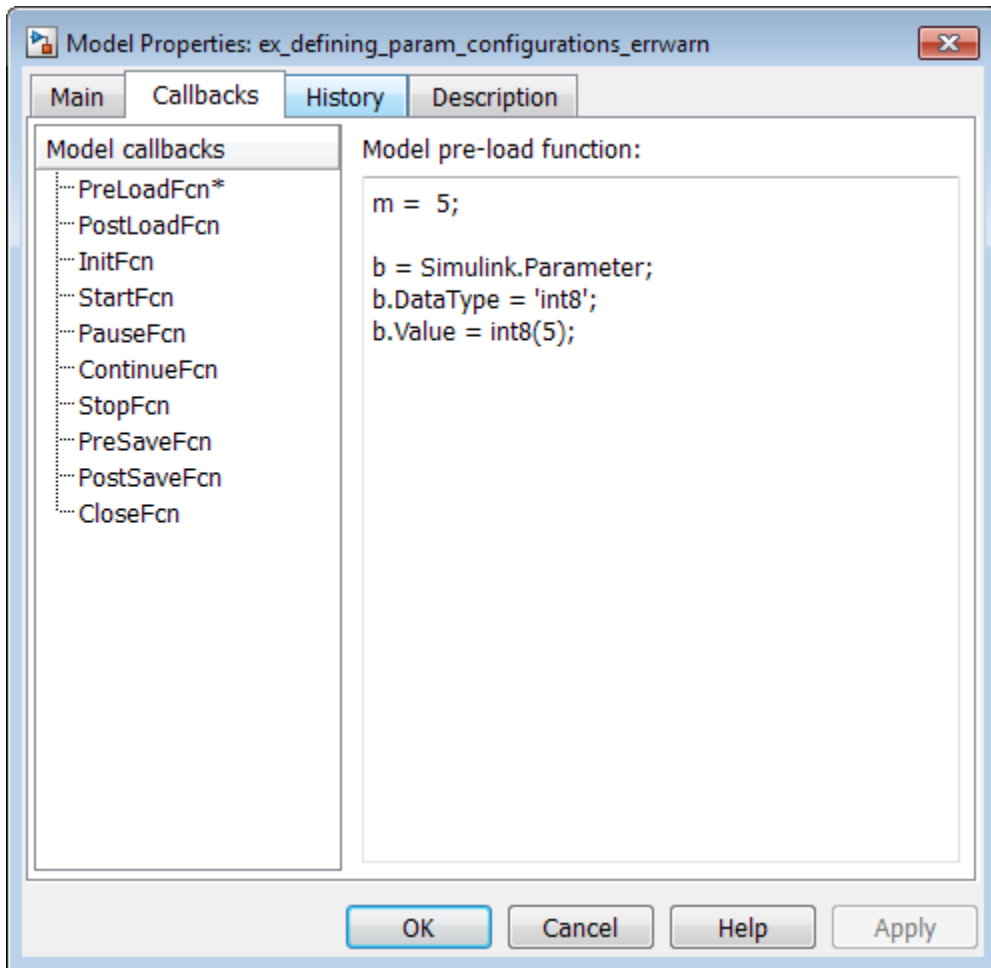
“Highlight Constrained Parameters in Model” on page 5-11

Using the Parameter Table, you can find and autogenerate constraints for parameters in your model. This example uses the following model, which contains **Gain** and **Constant** parameters defined as *m* and *b*, respectively.



Variables *m* and *b* are defined in the MATLAB workspace.

The model callback function `PreLoadFcn` defines *m* and *b* in the MATLAB workspace.



When the model opens:

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

## Find Parameters

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

Open the Parameter Table.

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

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

Find parameters that can be constrained for analysis.

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

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

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

Parameter table

Enable Disable Clear Highlight in Model

| Use                      | Name | Constraint | Value | Min | Max | Model Element                                     |
|--------------------------|------|------------|-------|-----|-----|---|
| <input type="checkbox"/> | b    |            | 5     |     |     | ex_defining_param_configurations_errwarn/Constant |
| <input type="checkbox"/> | m    |            | 5     |     |     | ex_defining_param_configurations_errwarn/Gain     |

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

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

**Note** If you use a MATLAB variable from a data dictionary as a model parameter, SLDV analysis does not consider the parameter as tunable. If you want the parameter to be tunable for the analysis, use a

Simulink.Parameter object for the parameter. To create a Simulink.Parameter object in the data dictionary:

- 1 In the Model Explorer, on the **Model Hierarchy** pane, select the workspace under the data dictionary that contains your MATLAB variable.
- 2 Select **Add > Simulink Parameter**. You see a new variable titled Param in the workspace.
- 3 Rename the variable. Assign the same data type as the original MATLAB variable.
- 4 In your model, use the variable that you just created as your parameter.

## Edit Parameter Constraints

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

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

For this example:

- For parameter b, specify the value range [4, 10].
- For parameter m, specify the value 5.

| Parameter table   |      |            |       |     |     |   |
|---|------|------------|-------|-----|-----|---|
| <input type="button" value="Enable"/> <input type="button" value="Disable"/> <input type="button" value="Clear"/> <input type="button" value="Highlight in Model"/> |      |            |       |     |     |   |
| Use   | Name | Constraint | Value | Min | Max | Model Element                                     |
| <input checked="" type="checkbox"/>   | b    | [4,10]     | 5     |     |     | ex_defining_param_configurations_errwarn/Constant |
| <input checked="" type="checkbox"/>   | m    | 5          | 5     |     |     | ex_defining_param_configurations_errwarn/Gain     |

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

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

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

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

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

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

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

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

## Highlight Constrained Parameters in Model

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

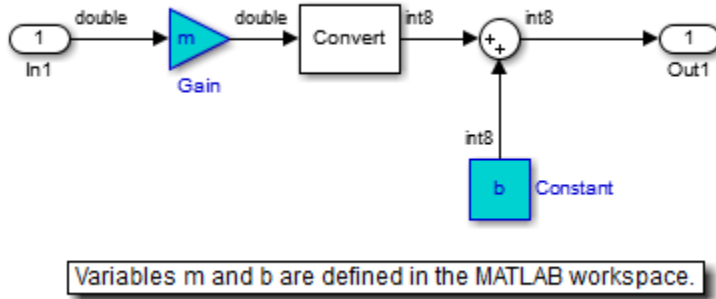
Select the parameter(s) you want to highlight in the model.

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

| Parameter table   |      |            |       |     |     |   |
|---|------|------------|-------|-----|-----|---|
| <input type="button" value="Enable"/> <input type="button" value="Disable"/> <input type="button" value="Clear"/> <input type="button" value="Highlight in Model"/> |      |            |       |     |     |   |
| Use   | Name | Constraint | Value | Min | Max | Model Element                                     |
| <input checked="" type="checkbox"/>   | b    | [4,10]     | 5     |     |     | ex_defining_param_configurations_errwarn/Constant |
| <input checked="" type="checkbox"/>   | m    | 5          | 5     |     |     | ex_defining_param_configurations_errwarn/Gain     |

Click **Highlight in Model**.

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



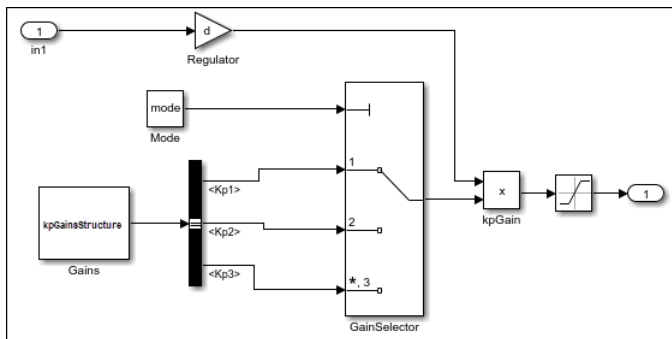
You can also define constraints for parameters using Parameter Configuration File. For more information, see “Template Parameter Configuration File” on page 5-29 in “Use Parameter Configuration File” on page 5-29.

To define constraints for structure or bus parameter, see “Specify Parameter Configuration for Structure or Bus Parameters” on page 5-12.

## Specify Parameter Configuration for Structure or Bus Parameters

### About This Example Model

This example describes how to generate tests that constrain the values for the structures and bus signals in a model. Suppose that your model includes a variable called `kpGainsStructure`, which is a structure in the MATLAB workspace. The model uses a Bus Selector block to separate the structure fields into individual bus signals. You can constrain the values of the structure or the values of the bus signals to ensure that they stay within the specified range during simulation.



This example describes how to create and analyze a simple Simulink model, then use Simulink Design Verifier to generate test cases for the model. The model contains an input signal `In1` whose value is set between -1 to 1. `kpGainsStructure` is a structure that contains three fields, `Kp1`, `Kp2`, and `Kp3`, and outputs them to a Bus Selector block that separates the fields into individual bus signals. The block called `Mode` has a constant value parameter, which is set to `mode` determines the three bus signals as an input to the `kpGain` block.

The value of `In1` is multiplied by `d`, then multiplied by the selected bus signal. The result passes to a Saturation block whose limit is defined between -0.5 to 0.5.

Based on the `mode` value, Simulink selects one of the three `kpGainsStructure` fields and specifies the constraints. The input signal to the Saturation block must be below the lower limit or fall above the upper limit to satisfy the decision objective of the Saturation block. Simulink Design Verifier then tunes these parameters to achieve this limit. The following workflow guides you through the process of completing this example.

### Preload Workspace Variable for Structure Parameter

Preload the value of the MATLAB workspace variable `kpGainsStructure`. The structure contains the fields `Kp1`, `Kp2`, and `Kp3`.

- 1 On the **Modeling** tab, select **Model Settings > Model Properties**.
- 2 Click the **Callbacks** tab.
- 3 Click **PreLoadFcn**, then load the `Kp1`, `Kp2`, and `Kp3` fields of `myStruct`:

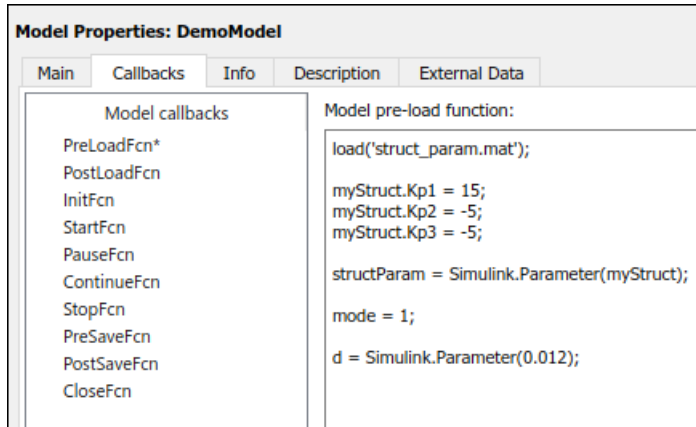
```
load('struct_param.mat');
myStruct.Kp1 = 15;
myStruct.Kp2 = -5;
```



```

myStruct.Kp3 = -5;
gainsParam = Simulink.Parameter(myStruct);
mode = 1;
d = Simulink.Parameter(0.012);

```



- 4 Click **OK** to close the Model Properties dialog box and save the model.

Because the structure parameter is called by the Constant block, you need to define the output of the Constant block as a bus. Follow these steps:

- 1 Double-click the Gains block to open Block Parameters dialog box.
- 2 Under **Signal Attributes**, select **Output data type** as Bus:Bus0.
- 3 Click **OK**.

## Define Parameter Constraint Values

There are two ways to constrain the values of structure or bus signals in the Configuration Parameter window: by using the parameter table or the parameter configuration file.

- **Parameter Table**
- **Parameter configuration file**

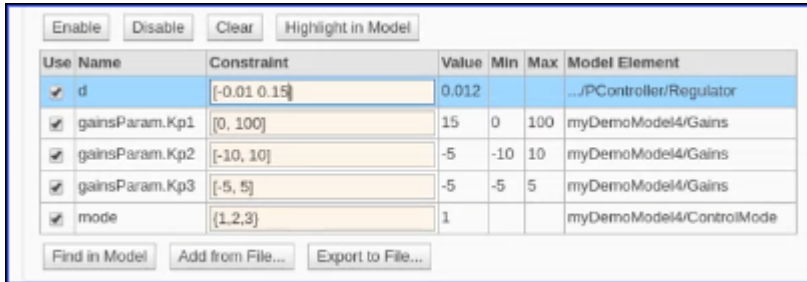
## Define Parameter Constraint Values using Parameter Table

When possible, parameter table automatically generates constraint values for each parameter, depending on the data type and location of the parameter in the model. For more information, see "Use Parameter Table" on page 5-7.

Follow these steps to generate the constraint value for each parameter:

- 1 On the **Apps** tab, under **Model Verification, Validation, and Test**, click **Design Verifier**.
- 2 On the **Design Verifier** tab, click **Test Generation Settings**.
- 3 In Configuration Parameters dialog box, select **Design Verifier > Parameters and Variants**.
- 4 Select **Use parameter table**.
- 5 Click **Find parameters**.
- 6 The parameter table populates with the parameters from your model.

- 7 In the parameter table, in the **Constraint** column,
- {1, 2, 3} for mode
  - [-0.01 0.15] for d



| Use Name   | Constraint   | Value | Min | Max | Model Element             |
|--|--------------|-------|-----|-----|---------------------------|
| <input checked="" type="checkbox"/> d              | [-0.01 0.15] | 0.012 |     |     | .../PController/Regulator |
| <input checked="" type="checkbox"/> gainsParam.Kp1 | [0, 100]     | 15    | 0   | 100 | myDemoModel4/Gains        |
| <input checked="" type="checkbox"/> gainsParam.Kp2 | [-10, 10]    | -5    | -10 | 10  | myDemoModel4/Gains        |
| <input checked="" type="checkbox"/> gainsParam.Kp3 | [-5, 5]      | -5    | -5  | 5   | myDemoModel4/Gains        |
| <input checked="" type="checkbox"/> mode           | {1,2,3}      | 1     |     |     | myDemoModel4/ControlMode  |

- 8 Click **OK**.

## Define Constraint Values using Parameter Configuration File

This is an alternative approach that you can use to define the values of constraints instead of using the Parameter Table. The Simulink Design Verifier software provides a template that you can make a copy and edit it. For more information, see “Template Parameter Configuration File” on page 5-29 in “Use Parameter Configuration File” on page 5-29. By default, the path to the parameter configuration file is:

```
matlabroot/toolbox/sldv/sldv/sldv_params_template.m
```

To associate the parameter configuration file with your model before analyzing the model, in the Configuration Parameters dialog box, on the **Design Verifier > Parameters and Variants** pane, ensure that **Use parameter table** is cleared and enter the file name of the configuration file in the **Parameter configuration file** field.

Follow these steps to define the constraint values in Parameter configuration file:

- 1 In `sldv_params_template.m`, enter:

```
function params = params_config
params.mode = {1, 2, 3};
params.d = [-.001 0.15];
params.gainsParam.Kp1 = Sldv.Interval(0, 50);
params.gainsParam.Kp2 = Sldv.Interval(-10, 10);
params.gainsParam.Kp3 = [-5, 5];
```

- 2 Save the file with the name `params_config.m`.
- 3 Open the model `DemoModel`.
- 4 On the **Apps** pane, under **Model Verification, Validation, and Test**, click **Design Verifier**.
- 5 On the **Design Verifier** tab, click **Test Generation Settings**.
- 6 In Configuration Parameters dialog box, select **Design Verifier > Parameters and Variants**.
- 7 Click **Browse**, then select `params_config.m` parameter configuration file created saved in step 2.

## Analyze Example Model

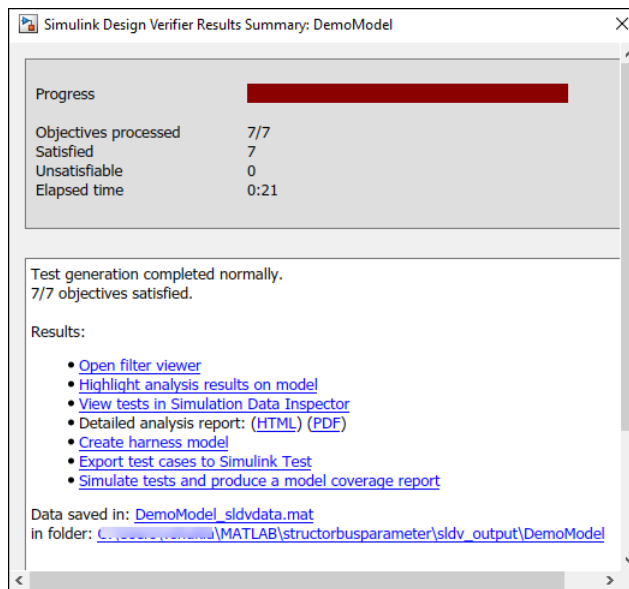
Analyze the model with the parameter constraints enabled and generate the analysis report:

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

Simulink Design Verifier analyzes your model to generate test cases.

- 2 When the software completes its analysis, in the Simulink Design Verifier Results Summary window, next to Detailed analysis report, select HTML.

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



- 3 In the table of contents of the Simulink Design Verifier report, click **Test Cases**.
- 4 Click **Test Case 1** to display the subsection for that test case.

Test Case 1 shows that Simulink Design Verifier tuned all the parameters in such a way that all the inputs coming from the In1 input signal, the Gain block and the mode variable will either fall below -0.5 or above 0.5. While generating test cases, all the constraints satisfy the objectives.

| Generated Parameter Values |          |
|----------------------------|----------|
| Parameter                  | Value    |
| d                          | 0.11128  |
| mode                       | 1        |
| structParam.Kp1            | 50.464   |
| structParam.Kp2            | 1.9482   |
| structParam.Kp3            | -0.11487 |

| Generated Input Data |         |     |     |
|----------------------|---------|-----|-----|
| Time                 | 0       | 0.2 | 0.4 |
| Step                 | 1       | 2   | 3   |
| in1                  | 0.70985 | -1  | 0   |

Similarly, the parameters for Test Case 2 and Test Case 3 are tuned and satisfy the objectives.

**See Also**

“Use Parameter Table” on page 5-7

## Specify Parameter Configuration for Full Coverage

| In this section...                               |
|--|
| “About This Example” on page 5-17                |
| “Construct Example Model” on page 5-17           |
| “Parameterize Constant Block” on page 5-18       |
| “Preload Workspace Variable” on page 5-18        |
| “Autogenerate Parameter Constraint” on page 5-19 |
| “Analyze Example Model” on page 5-20             |
| “Simulate Test Cases” on page 5-22               |

### About This Example

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

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

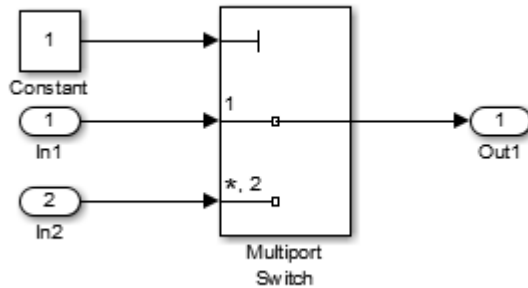
| Task | Description  | See...   |
|------|--|--|
| 1    | Construct the example model.   | “Construct Example Model” on page 5-17           |
| 2    | Specify a variable as the value of a Constant block parameter.         | “Parameterize Constant Block” on page 5-18       |
| 3    | Constrain the value of the variable that the Constant block specifies. | “Autogenerate Parameter Constraint” on page 5-19 |
| 4    | Generate test cases for your model and interpret the results.          | “Analyze Example Model” on page 5-20             |
| 5    | Simulate the test cases and measure the resulting decision coverage.   | “Simulate Test Cases” on page 5-22               |

### Construct Example Model

Construct a simple Simulink model to use in this example:

- 1 Create an empty Simulink model.
- 2 Copy the following blocks into the empty Simulink Editor:
  - From the Sources library:
    - Two Inport blocks to initiate the input signals
    - A Constant block to control the switch
  - From the Signal Routing library: A Multiport Switch block to provide simple logic
  - From the Sinks library: An Outport block to receive the output signal

- 3 Double-click the Multiport Switch block to access its dialog box and specify its **Number of data ports** option as 2.
- 4 Connect the blocks so that your model looks like the following.



- 5 On the **Simulation** tab, click the arrow on the right of the **Prepare** section and click **Model Settings**.
- 6 In the Configuration Parameters dialog box, select the **Solver**. Under **Solver selection**, set the **Type** option to Fixed-step, and then set the **Solver** option to discrete (no continuous states).
- 7 In the **Diagnostics** pane, set **Automatic solver parameter selection** to none.
- 8 Click **OK** to apply your changes and close the Configuration Parameters dialog box.
- 9 Save your model as `ex_defining_params_example` for use in the next procedure.

## Parameterize Constant Block

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

- 1 Double-click the Constant block.
- 2 In the **Constant value** box, enter A.
- 3 Click **OK** to apply your change and close the Constant block parameter dialog box.
- 4 Save your model.

## Preload Workspace Variable

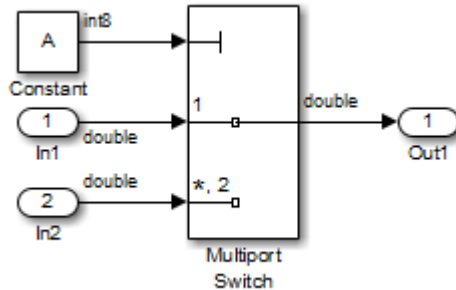
Preload the value of the MATLAB workspace variable A referenced by the Constant block:

- 1 On the **Modeling** tab, select **Model Settings > Model Properties**.
- 2 Click the **Callbacks** tab.
- 3 In the PreLoadFcn, enter:
 

```
A = Simulink.Parameter(int8(1));
A.Min = 1;
A.Max = 2;
```
- 4 Click **OK** to close the Model Properties dialog box and save your changes.
- 5 Close your model.

## 6 Open your model.

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



## Autogenerate Parameter Constraint

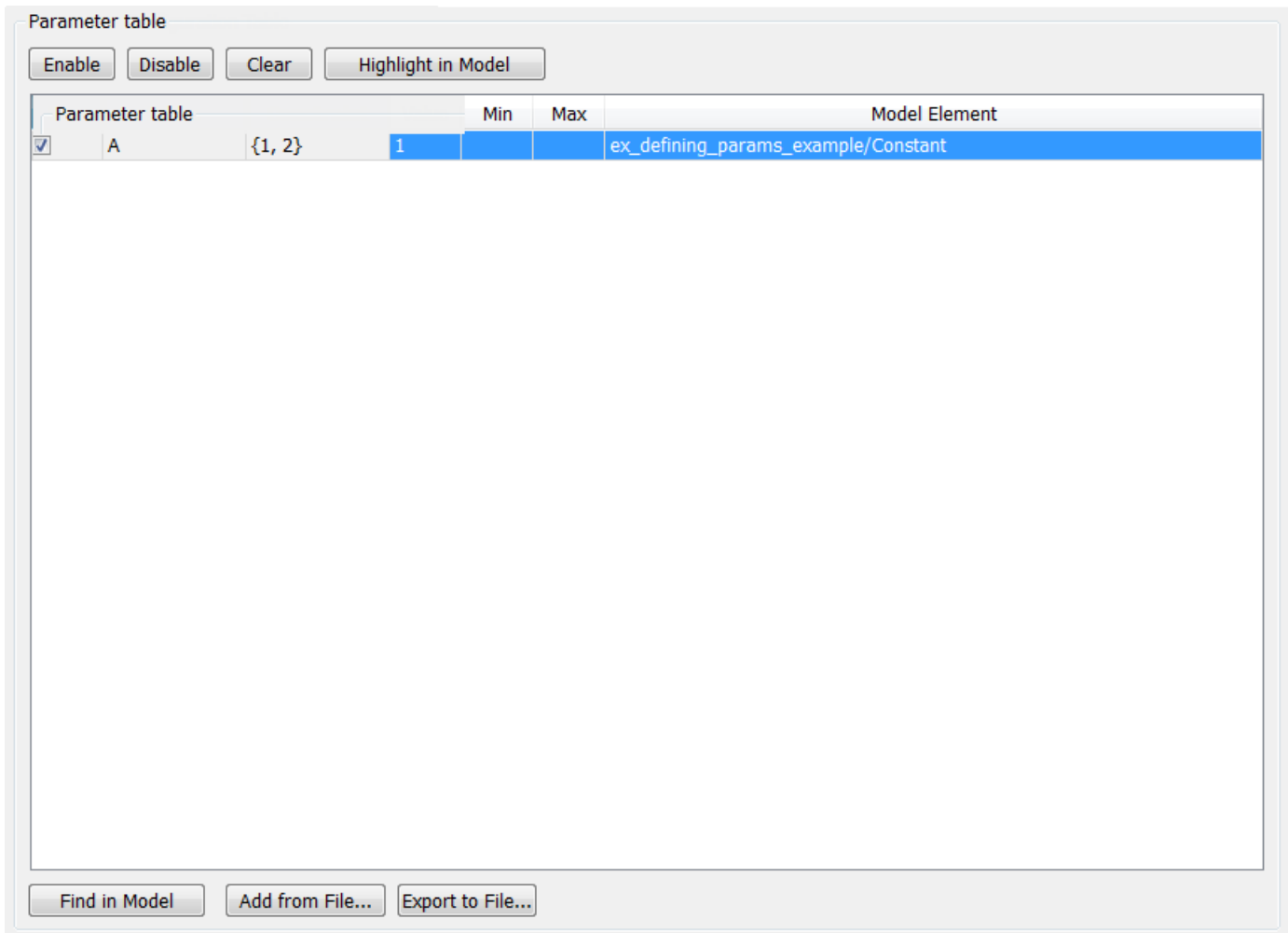
Use the Parameter Table to constrain variable A to specified values.

- 1 On the **Apps** tab, click the arrow on the right of the **Apps** section.  
Under **Model Verification, Validation, and Test**, click **Design Verifier**.
- 2 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
- 3 In Configuration Parameters dialog box, select **Design Verifier > Parameters and Variants**.
- 4 Select **Use parameter table**.
- 5 Click **Find parameters**.

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

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

- In the **Name** column, the parameter name (A).
- In the **Constraint** column, the constraint specified on parameter A. The Parameter Table autogenerates the constraint values [1, 2].
- In the **Value** column, the value of A in the base workspace. This value is 1.
- In the **Model Element** column, the model component in which A resides (ex\_defining\_params\_example/Constant).
- In the **Use** column, a check box indicating whether the specified constraint values in the table are configured for analysis.



- 6 In the Parameter Table, in the row for parameter A, make sure that you select the **Use** check box. When you enable this parameter configuration, during Simulink Design Verifier analysis, the parameter A takes only the `int8` values 1 and 2.
- 7 In the Configuration Parameters dialog box, click **OK**.
- 8 Save your model.

## Analyze Example Model

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

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

Simulink Design Verifier analyzes your model to generate test cases.

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

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



Keep the Results Summary window open for the next procedure.

- 3 In the Simulink Design Verifier report **Table of Contents**, click **Test Cases**.
- 4 Click **Test Case 1** to display the subsection for that test case.

## Test Case 1

### Summary

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

### 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 |
| Step | 1 |
| In1  | - |
| In2  | - |

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

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

## Test Case 2

### Summary

Length: 0 second (1 sample period)  
 Objectives: 1  
 Satisfied: 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 |
| Step | 1 |
| In1  | - |
| In2  | - |

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

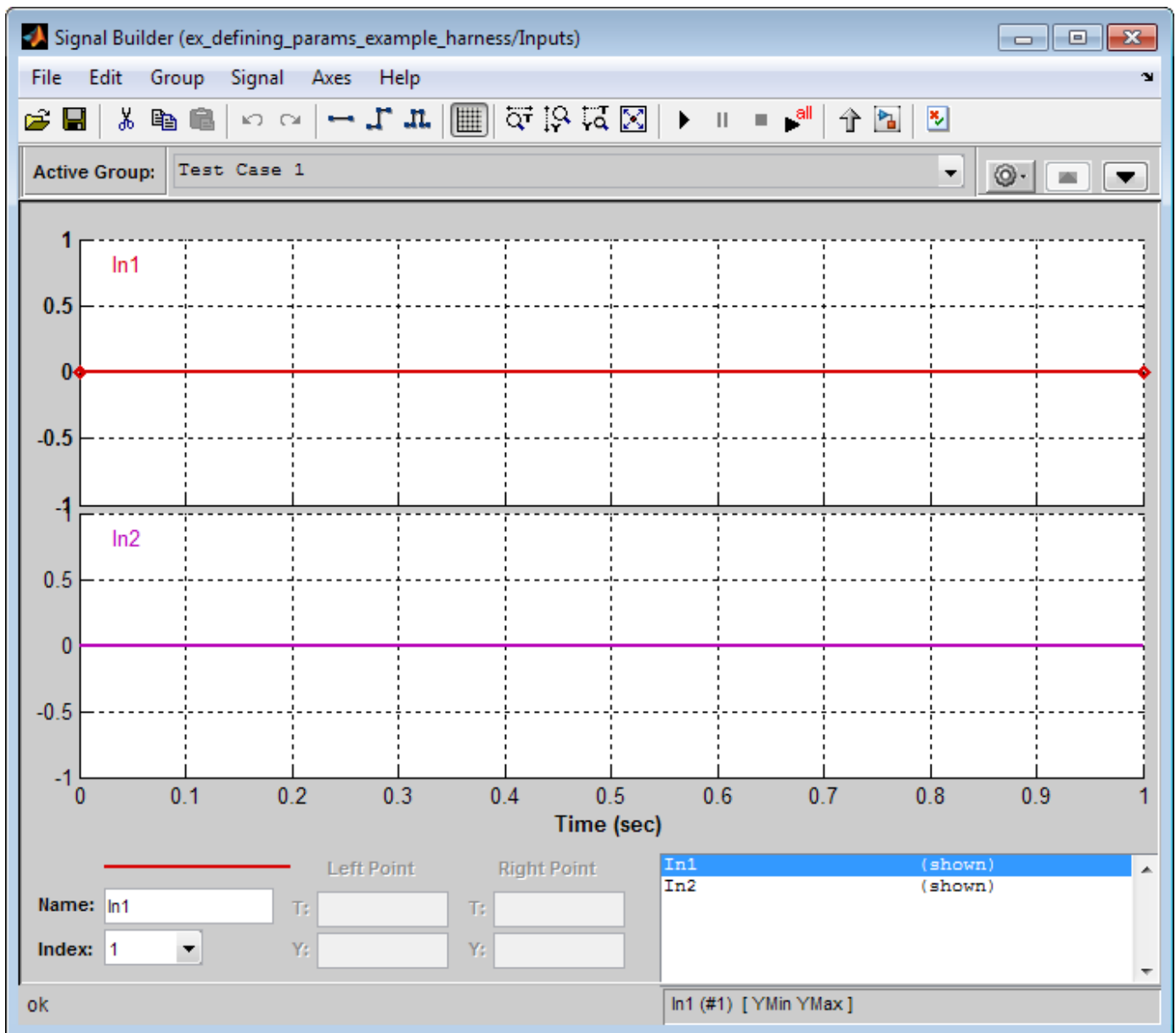
## Simulate Test Cases


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

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

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

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





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

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

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

## Summary

### Model Hierarchy/Complexity:

|   |   | D1   |
|---|---|--|
| 1. <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 block "[Test Unit \(copied from ex\\_defining\\_param...](#)"

**Parent:** [/ex\\_defining\\_params\\_example\\_harness](#)

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

### MultiPortSwitch block "[Multiport Switch](#)"

**Parent:** [ex\\_defining\\_params\\_example\\_harness/Test Unit \(copied from ex\\_defining\\_params\\_example\)](#)

| <b>Metric</b>         | <b>Coverage</b>              |
|-----------------------|------------------------------|
| Cyclomatic Complexity | 1                            |
| Decision (D1)         | 100% (2/2) decision outcomes |

#### **Decisions analyzed:**

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

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

#### **See Also**

"Extend Existing Test Cases After Applying Parameter Configurations" on page 5-46

## Store Parameter Constraints in MATLAB Code Files

|   |
|---|
| <b>In this section...</b>                             |
| “Export Parameter Constraints to File” on page 5-26   |
| “Import Parameter Constraints from File” on page 5-27 |

You can use the Parameter Table to manage constraints on your model parameters for analysis. If you place a constraint on a parameter in your model, during analysis that parameter takes only your specified constraint value or values. A group of constraints on parameters in the same model is also called a parameter configuration. You can store groups of parameter constraints in a MATLAB code file called a parameter configuration file. For more information on configuring parameters for Simulink Design Verifier, see “Use Parameter Table” on page 5-7.

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

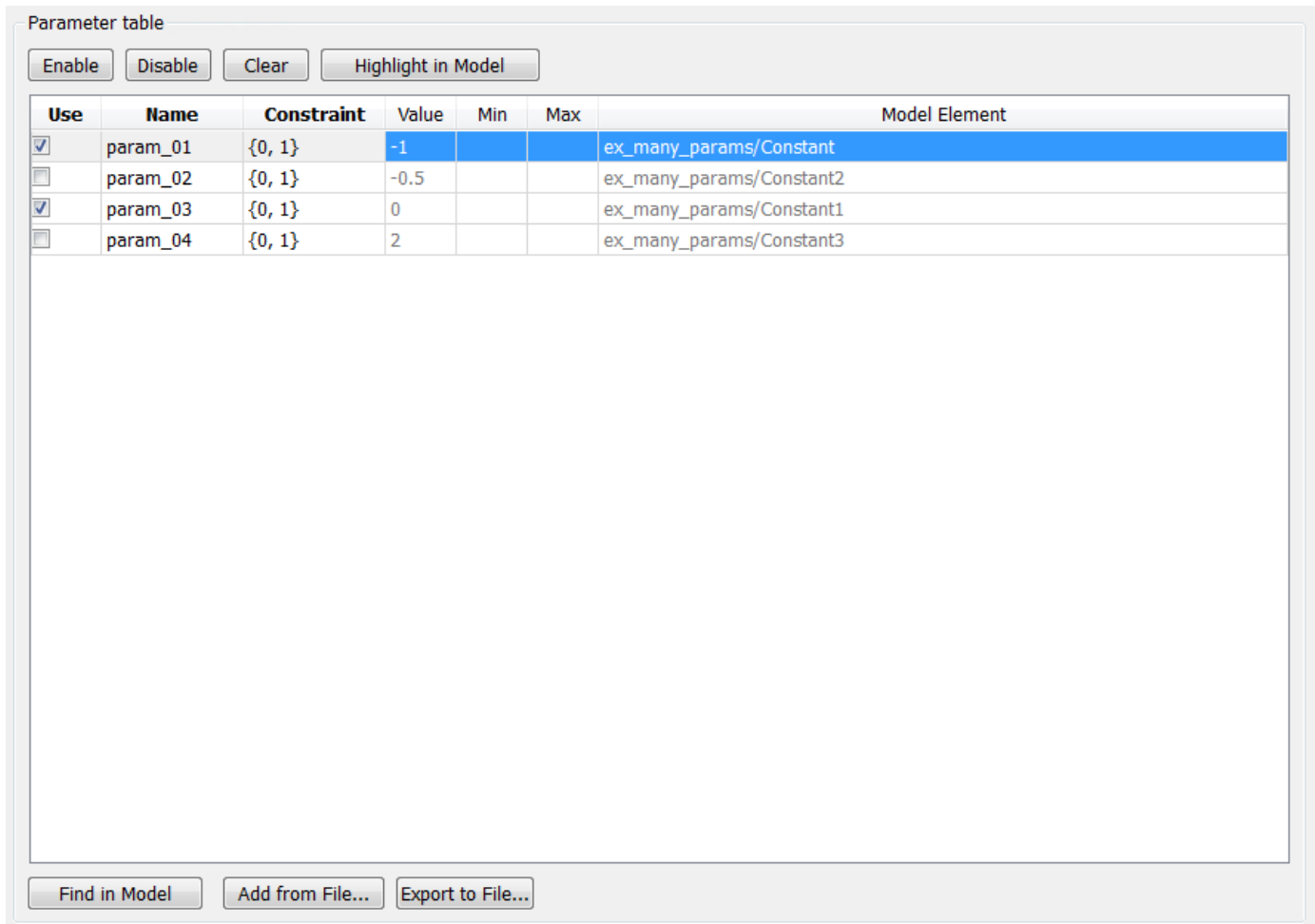
### Export Parameter Constraints to File

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

To export parameter constraint values to a file:

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

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



## 2 Click **Export to File**.

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

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

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

## Import Parameter Constraints from File

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

To import parameter constraints from a MATLAB code file:

- 1 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**. In the Configuration Parameters dialog box, select **Design Verifier > Parameters and Variants**.
- 2 Click **Add from File**. Choose a parameter configuration file.

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

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

```
matlabroot/toolbox/sldv/sldvdemos/sldvdemo_param_ident_config.m
```

### See Also

### More About

- “Generate Parameters Values” on page 5-45



## Use Parameter Configuration File

### In this section...

“Template Parameter Configuration File” on page 5-29

“Syntax in Parameter Configuration Files” on page 5-29

To specify parameters as variables for analysis, you can use the Parameter Table or define parameter configurations in a MATLAB code file. You can also export parameter configuration files from the Parameter Table. For more information, see “Store Parameter Constraints in MATLAB Code Files” on page 5-26.

This example shows how to define parameter configurations in a MATLAB code file. For an example that shows how to define these parameter configurations using the Parameter Table, see “Use Parameter Table” on page 5-7.

### Template Parameter Configuration File

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

```
matlabroot/toolbox/sldv/sldv/sldv_params_template.m
```

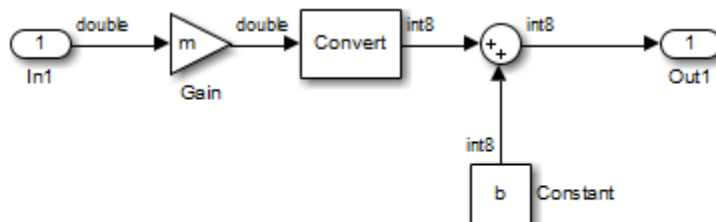
To create a parameter configuration file, make a copy of the template and edit the copy. The comments in the template explain the syntax for defining parameter configurations.

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

### Syntax in Parameter Configuration Files

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

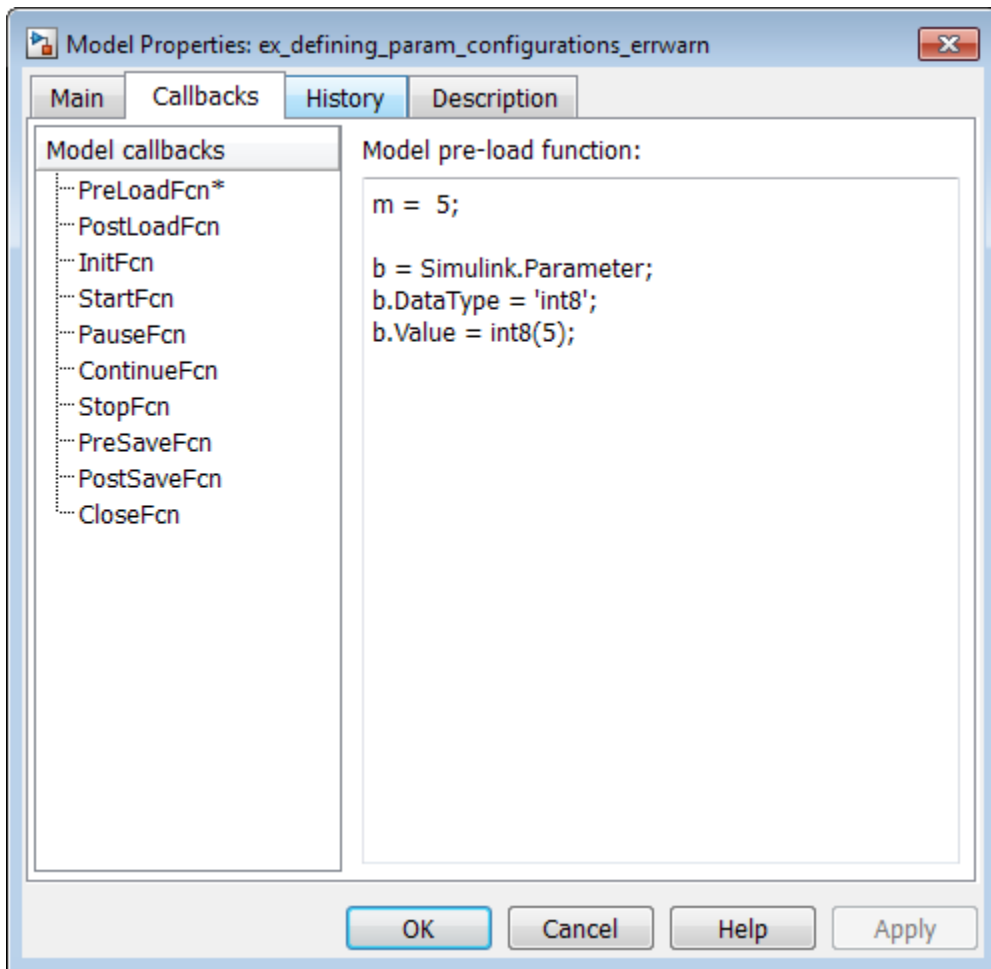
For example, suppose you want to constrain the **Gain** and **Constant value** parameters, **m** and **b**, which appear in the following model:



Variables **m** and **b** are defined in the MATLAB workspace.

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

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



In your parameter configuration file, specify constraints for `m` and `b`:

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

This file specifies:

- `b` is an 8-bit signed integer from 4 to 10. The constraint type must match the type of the parameter `b` in the MATLAB workspace, `int8`, in this example.
- `m` is not constrained to any values.

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

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

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

---

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

---

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

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

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

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

---

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

---

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

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

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

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

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

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

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

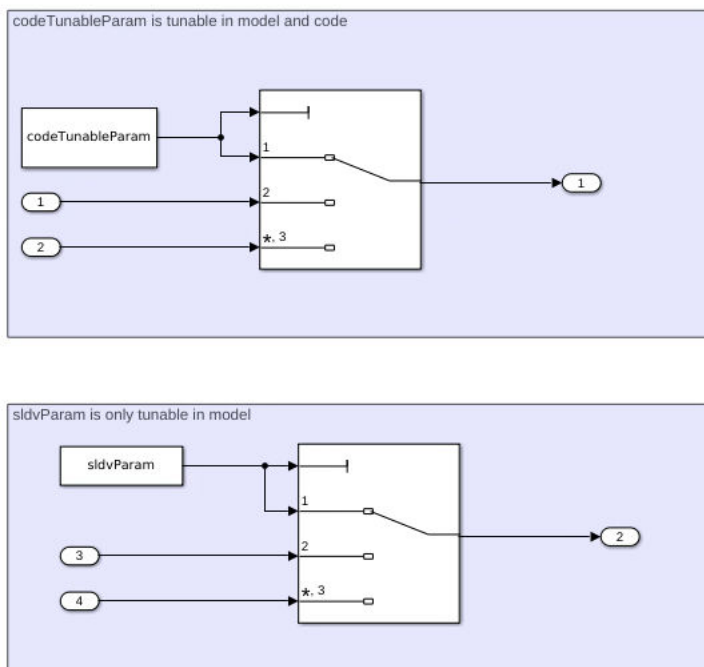
## Automatically Infer Parameter Specification

Simulink Design Verifier automates the process of selecting parameters that is a part of parameter configuration and determines minimum and maximum values of such parameters configured in the `Simulink.Parameter` object.

When test generation target is **Model**, Simulink Design Verifier selects as many parameters as possible for parameter configuration.

When test generation target is **Code Generated as Top Model** or **Code Generated as Model Reference**, parameters whose value can be changed in the generated code are selected for parameter configuration.

The `PreLoadFcn` callback function model, defines `codeTunableParam` and `constParam` in the MATLAB workspace.

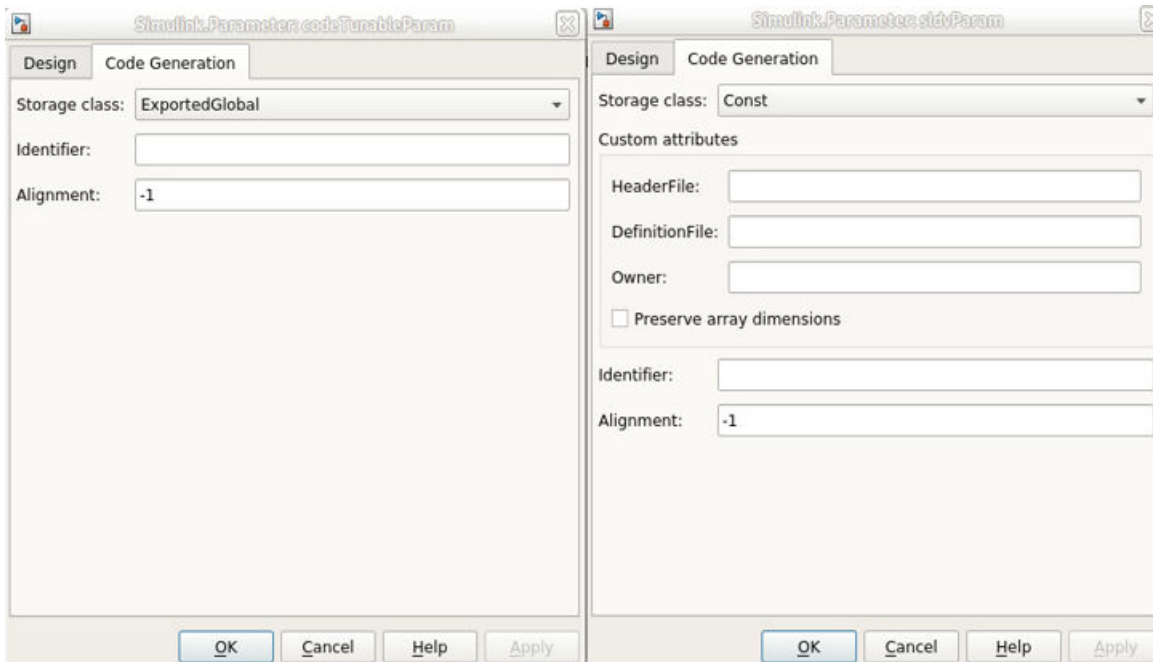


The code generation settings for the model:

**Model Properties: mTunability**

| Main  | Callbacks | Info | Description | External Data |
|---|-----------|------|-------------|---------------|
| Model callbacks   |           |      |             |               |
| <ul style="list-style-type: none"> <li>PreLoadFcn*</li> <li>PostLoadFcn</li> <li>InitFcn</li> <li>StartFcn</li> <li>PauseFcn</li> <li>ContinueFcn</li> <li>StopFcn</li> <li>PreSaveFcn</li> <li>PostSaveFcn</li> <li>CloseFcn*</li> </ul> |           |      |             |               |
| Model pre-load function:  |           |      |             |               |
| <pre>sldvParam = Simulink.Parameter(1); sldvParam.Min = -10; sldvParam.Max = 10;  codeTunableParam = Simulink.Parameter(2); codeTunableParam.Max = 10; codeTunableParam.Min = 0;</pre>  |           |      |             |               |

Set storage class of `constParam` to **Const** and `codeTunableParam` to **ExportedGlobal**.



## Configuring Parameters by Using Automatically infer parameter specification

This example shows how to automatically infer constraint values used for model parameters during Simulink Design Verifier analysis.

- 1 Open **Model Settings > Design Verifier > Parameters and Variants**.
- 2 Click on the drop down for **Parameter Configuration** and select **Automatically infer parameter specification**.

This automatically infers the parameters that will be selected based on the test generation target and the parameter settings based on their definition.

When the test generation target is **Model**, Simulink Design Verifier analysis selects all the supported parameters.

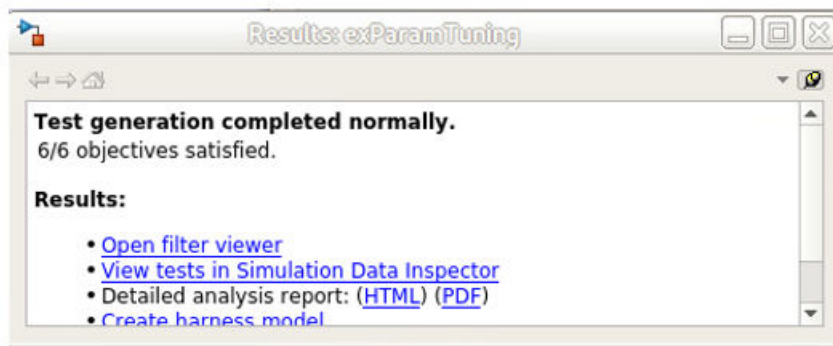
In the above example, both the parameters `constParam` and `codeTunableParam`, are configured during the analysis.

## 2.4.1. Parameter Constraints

### Constraint 1

| Parameter        | Constraint |
|------------------|------------|
| codeTunableParam | [0, 10]    |
| sldvParam        | [-10, 10]  |

The results window shows that all objectives for both the Multiport switch blocks are satisfied.

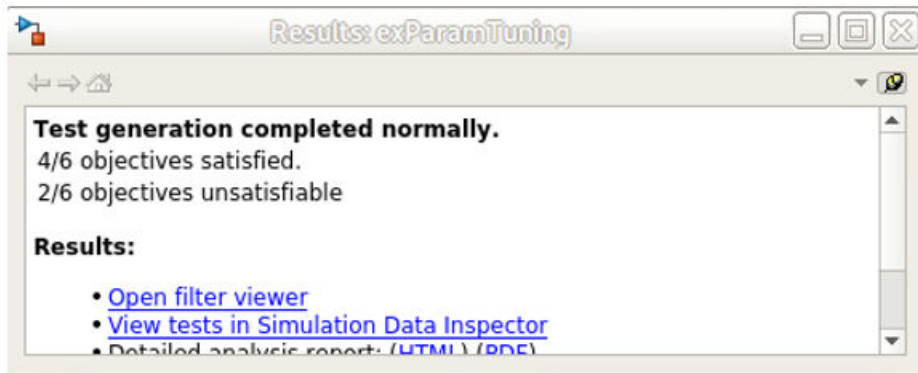


When the test generation target is set to **Code Generated as Top Model**, parameter constParam cannot be changed in the generated code. So, Simulink Design Verifier selects codeTunableParam for parameter configuration.

## 2.4.1. Parameter Constraints

### Constraint 1

| Parameter        | Constraint |
|------------------|------------|
| codeTunableParam | [0, 10]    |



The Undecided objectives are related to the code corresponding to Multiport Switch1.

## Determine from Generated Code

Simulink Design Verifier selects the parameters whose value can be changed in the generated code for parameter configuration.

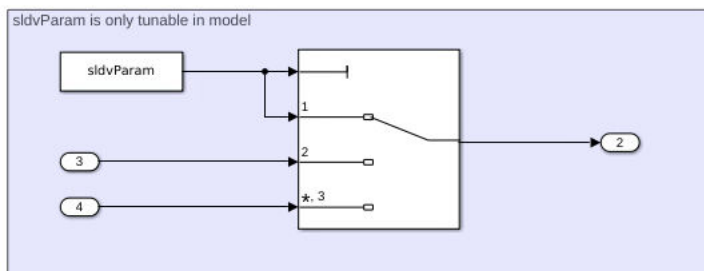
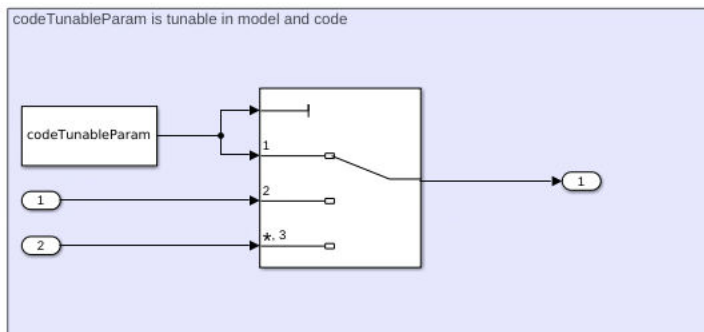
For such parameters, the minimum or maximum value from `Simulink.Parameter` object is used as parameter configuration for analysis.

---

### Note

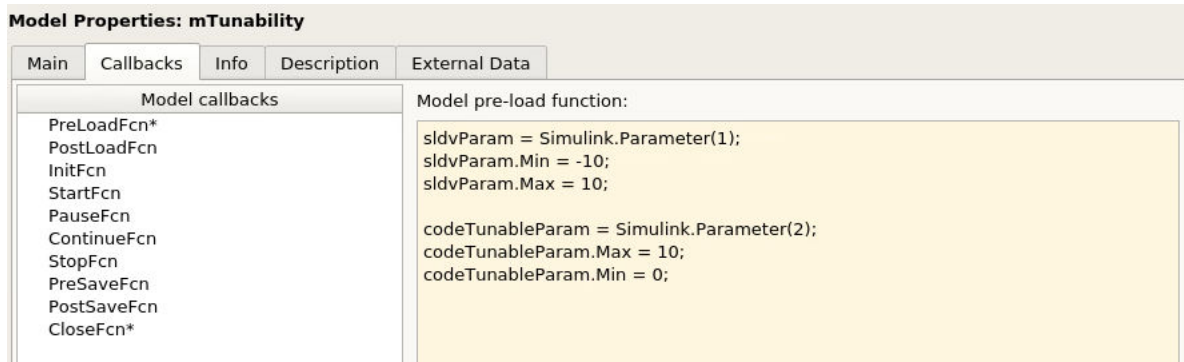
- This workflow is recommended when you have generated the code before the analysis is run.
  - This parameter configuration can be used for both Model and Code workflows.
- 

The `PreLoadFcn` callback function model, defines `codeTunableParam` and `constParam` in the MATLAB workspace.

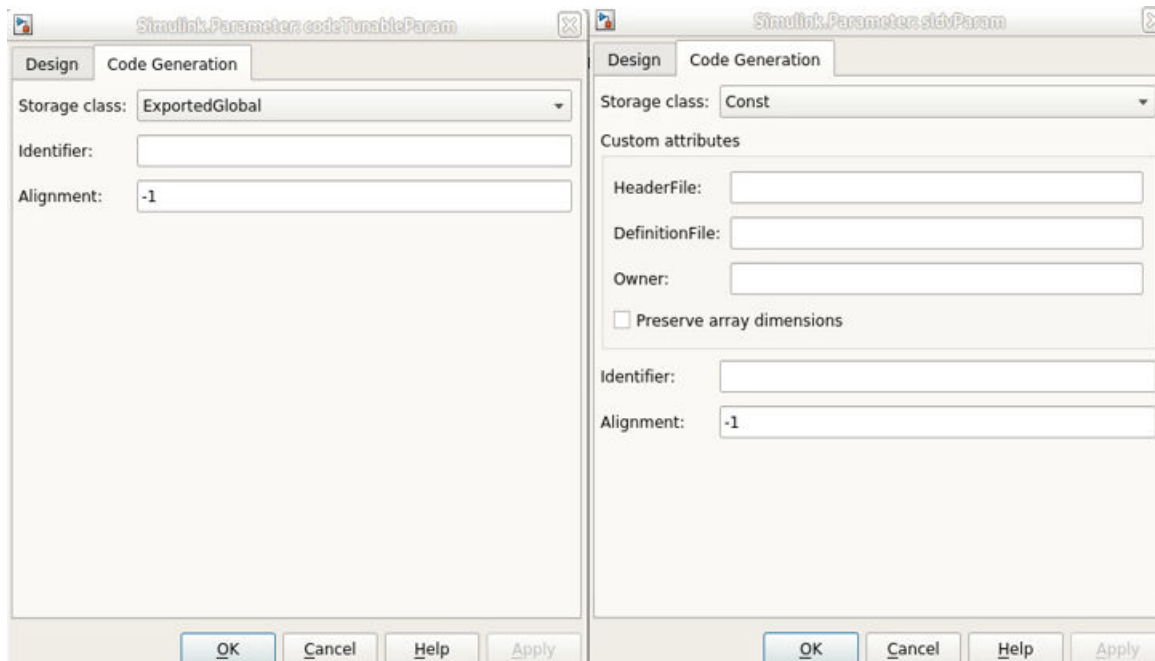


The code generation settings for the model:





Set storage class of constParam to **Const** and codeTunableParam to **ExportedGlobal**.



## Configuring Parameters by Using Determine from generated code

This example shows how to configure parameters by using **Determine from generated code** workflow during the Simulink Design Verifier analysis.

- 1 Open **Model Settings > Design Verifier > Parameters and Variants**.
- 2 Click on the drop down for **Parameter Configuration** and select **Determine from generated code**.

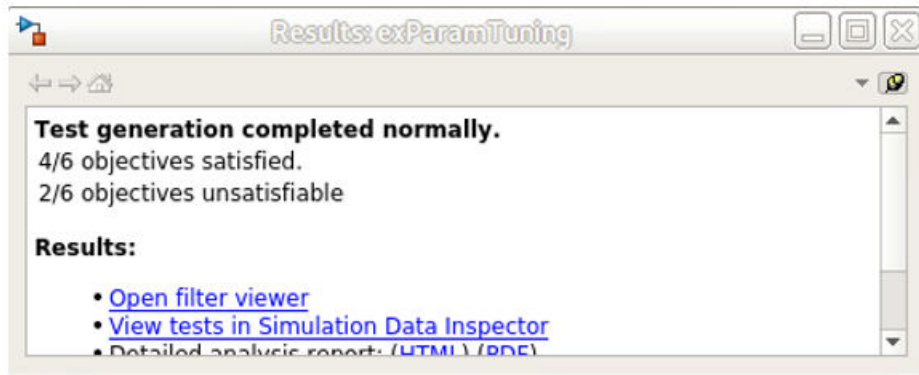
This automatically infers the parameters that will be selected based on the code generated and the parameter settings based on their definition.

In the above example, the parameter constParam cannot be changed in the generated code. So Simulink Design Verifier selects codeTunableParam for parameter configuration.

## 2.4.1. Parameter Constraints

### Constraint 1

| Parameter        | Constraint |
|------------------|------------|
| codeTunableParam | [0, 10]    |



The Undecided objectives are related to the code corresponding to Multiport Switch1.

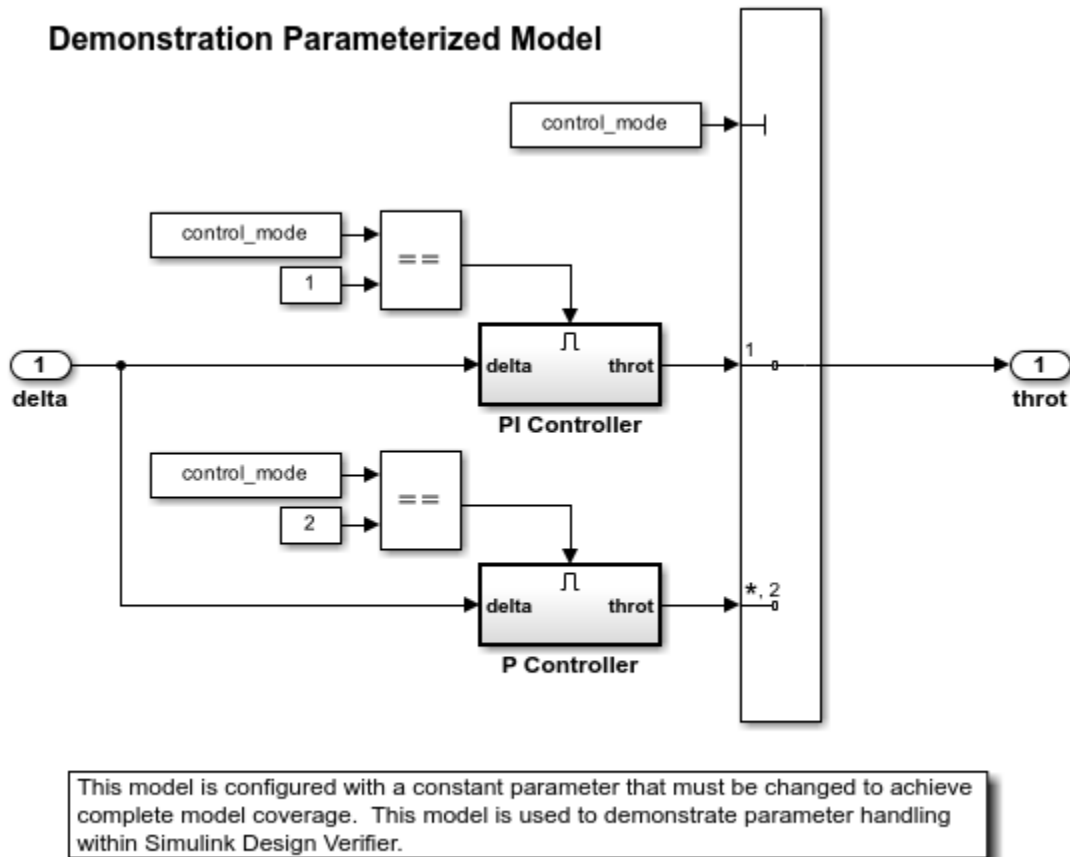
## Using Command Line Functions to Support Changing Parameters

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

### Controller Model with an Adjustable Parameter

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

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



Copyright 2006-2023 The MathWorks, Inc.

### Specifying Parameter Values for Analysis

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

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

```
control_mode = 1;
```

### Simulink® Design Verifier™ Options

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

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

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

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

get(opts)

                Mode: 'TestGeneration'
                MaxProcessTime: 300
                AutomaticStubbing: 'on'
                UseParallel: 'off'
                DesignMinMaxConstraints: 'on'
                OutputDir: 'sldv_output/$modelName$'
                MakeOutputFilesUnique: 'on'
                BlockReplacement: 'off'
                BlockReplacementRulesList: '<FactoryDefaultRules>'
                BlockReplacementModelFileName: '$modelName$_replacement'
                ParameterConfiguration: 'None'
                ParametersConfigFileName: 'sldv_params_template.m'
                ParameterNames: []
                ParameterConstraints: []
                ParameterUseInAnalysis: []
                TestgenTarget: 'Model'
                ModelCoverageObjectives: 'ConditionDecision'
                TestConditions: 'UseLocalSettings'
                TestObjectives: 'UseLocalSettings'
                MaxTestCaseSteps: 10000
                TestSuiteOptimization: 'Auto'
                Assertions: 'UseLocalSettings'
                ProofAssumptions: 'UseLocalSettings'
                ExtendExistingTests: 'off'
                ExistingTestFile: ''
                IgnoreExistTestSatisfied: 'on'
                IgnoreCovSatisfied: 'off'
                CoverageDataFile: ''
                CovFilter: 'off'
                CovFilterFileName: ''
                IncludeRelationalBoundary: 'off'
                RelativeTolerance: 0.0100
                AbsoluteTolerance: 1.0000e-05
                DetectDeadLogic: 'off'
```

```

    DetectActiveLogic: 'off'
    DeadLogicObjectives: 'ConditionDecision'
    DetectOutOfBounds: 'on'
    DetectDivisionByZero: 'on'
    DetectIntegerOverflow: 'on'
        DetectInfNaN: 'off'
        DetectSubnormal: 'off'
        DesignMinMaxCheck: 'off'
    DetectDSMAccessViolations: 'off'
    DetectHISMViolationsHisl_0002: 'off'
    DetectHISMViolationsHisl_0003: 'off'
    DetectHISMViolationsHisl_0004: 'off'
    DetectHISMViolationsHisl_0028: 'off'
    DetectBlockInputRangeViolations: 'off'
        ProvingStrategy: 'Prove'
    MaxViolationSteps: 20
        DataFileName: '$ModelName$_sldvdata'
    SaveExpectedOutput: 'off'
    RandomizeNoEffectData: 'off'
        SaveHarnessModel: 'on'
        HarnessModelFileName: '$ModelName$_harness.slx'
    ModelReferenceHarness: 'on'
        HarnessSource: 'Signal Editor'
        SaveReport: 'off'
        ReportPDFFormat: 'off'
        ReportFileName: '$ModelName$_report'
    ReportIncludeGraphics: 'off'
        DisplayReport: 'on'
        SFCnSupport: 'on'
    CodeAnalysisExtraOptions: ''
    CodeAnalysisIgnoreVolatile: 'on'
        ReduceRationalApprox: 'on'
        SLTestFileName: '$ModelName$_test'
        SLTestHarnessName: '$ModelName$_sldvharness'
        SLTestHarnessSource: 'Inport'
        StrictEnhancedMCDC: 'off'
    RebuildModelRepresentation: 'IfChangeIsDetected'
    AnalyzeAllStartupVariants: 'on'

```

## Generating Tests and Collecting Coverage

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

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

```
04-Mar-2023 00:18:27
Checking compatibility for test generation: model 'sldvdemo_param_controller'
Compiling model...done
Building model representation...done
```

```
04-Mar-2023 00:18:32
```

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

Generating tests using model representation from 04-Mar-2023 00:18:32...

.....  
04-Mar-2023 00:18:41

Completed normally.

Generating output files:

Harness model:

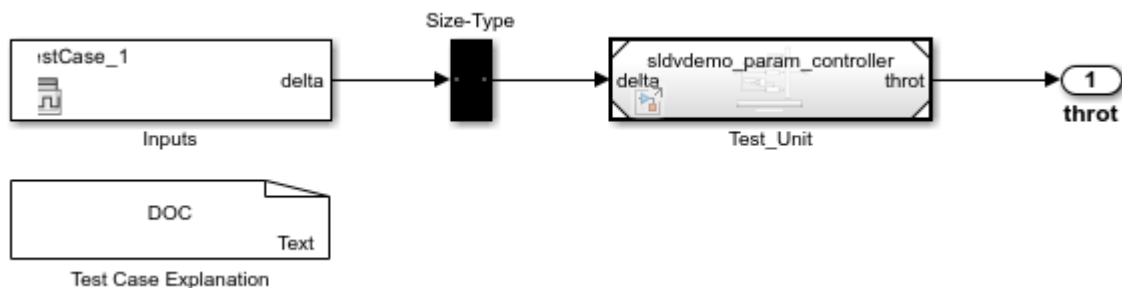
C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\28\tp27a1e6fc\sldv-ex05697027\sldv\_output\sldvdemo\_param

04-Mar-2023 00:18:44

Results generation completed.

Data file:

C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\28\tp27a1e6fc\sldv-ex05697027\sldv\_output\sldvdemo\_param



### Integrating Parameter Initialization Into a Test Harness

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

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

```
initCmdStr =
```

```
    'control_mode=1;'
```

### Modifying Parameters and Repeating Test Generation

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

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

04-Mar-2023 00:18:48

Validating cached model representation from 04-Mar-2023 00:18:32...change detected

04-Mar-2023 00:18:48

Checking compatibility for test generation: model 'sldvdemo\_param\_controller'

Compiling model...done

Building model representation...done

04-Mar-2023 00:18:52

'sldvdemo\_param\_controller' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 04-Mar-2023 00:18:52...

.....

04-Mar-2023 00:18:56

Completed normally.

Generating output files:

Harness model:

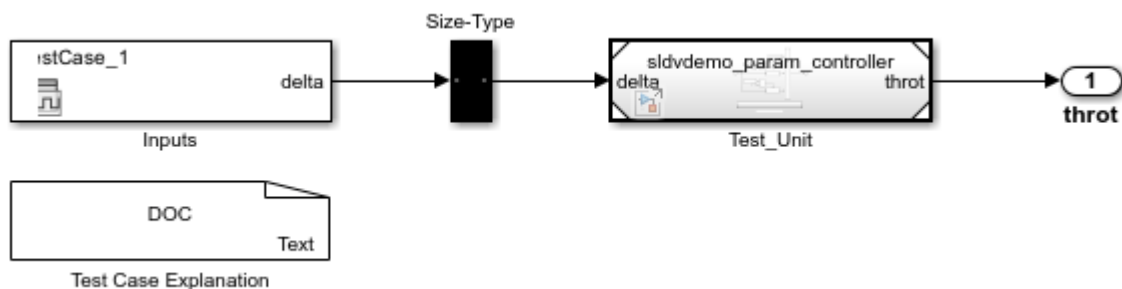
C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\28\tp27a1e6fc\sldv-ex05697027\sldv\_output\sldvdemo\_param

04-Mar-2023 00:18:58

Results generation completed.

Data file:

C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\28\tp27a1e6fc\sldv-ex05697027\sldv\_output\sldvdemo\_param



### Merging Test Harnesses Into a Single Model

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

```

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

```

newInitCmd =
  
```

```

    'control_mode=2;';
  
```

### Executing the Tests in the Harness Model

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

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

### Clean Up

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

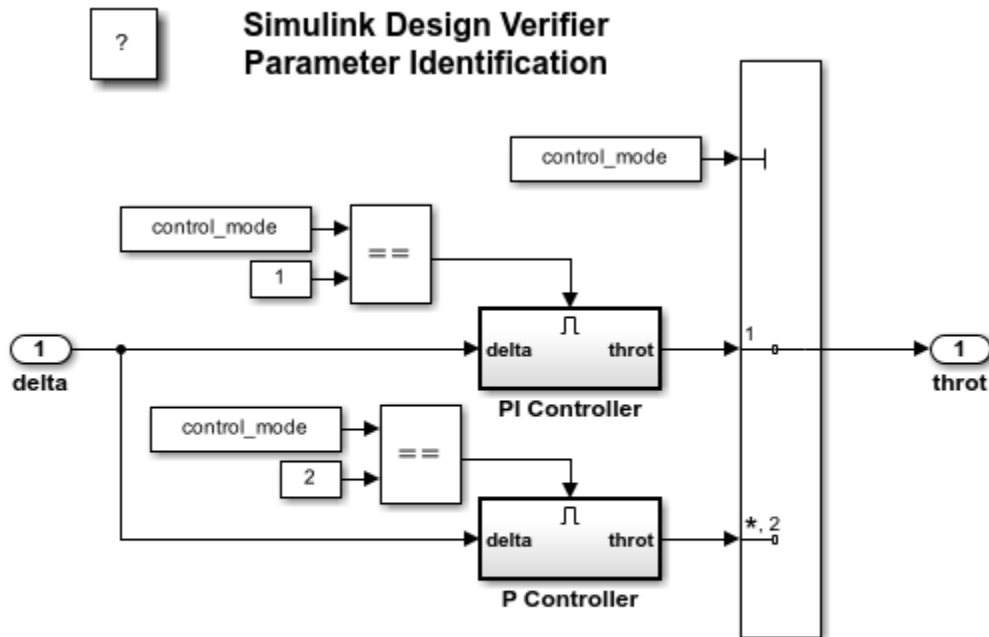
```
close_system(modelName,0);  
close_system('sldvdemo_param_controller',0);  
delete(files.HarnessModel);  
delete(newFiles.HarnessModel);
```



## Generate Parameters Values

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

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



Copyright 2006-2019 The MathWorks, Inc.

## Extend Existing Test Cases After Applying Parameter Configurations

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

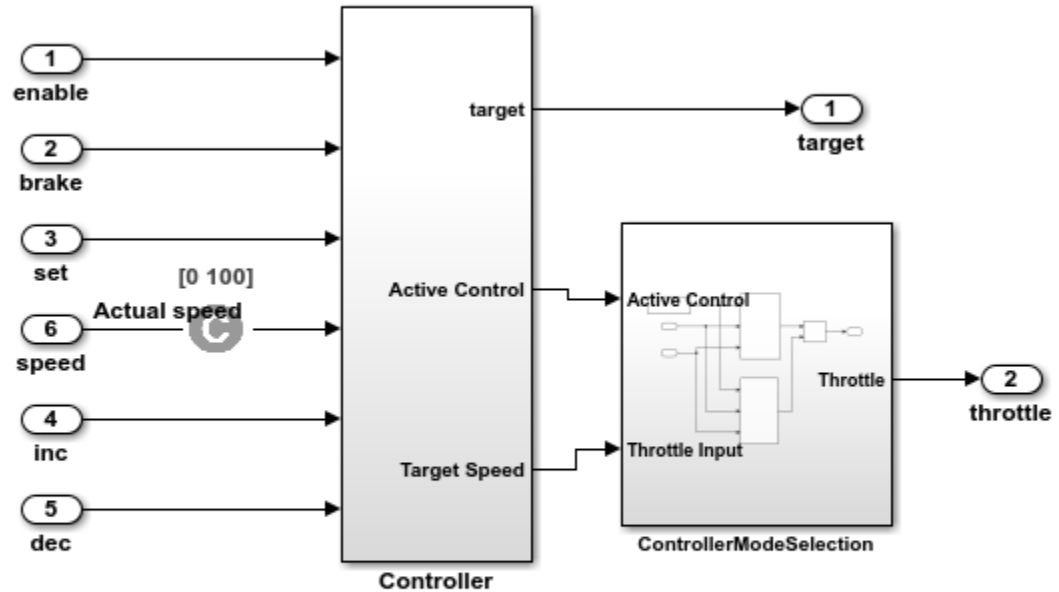
In this example, you generate test cases for a model and review the analysis results. The results show that the model consists of unsatisfiable objectives and does not achieve full coverage. Then, you apply parameter configurations in the model and reuse the previously generated test cases to achieve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

### Step 1: Generate Initial Test Cases and Review Results

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

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

## Simulink Design Verifier Extend Test Cases in Presence of Parameter Configurations



This example shows how to extend existing test cases in presence of parameter configurations. The ControllerModeSelection selects the mode of the controller based on the parameter value.

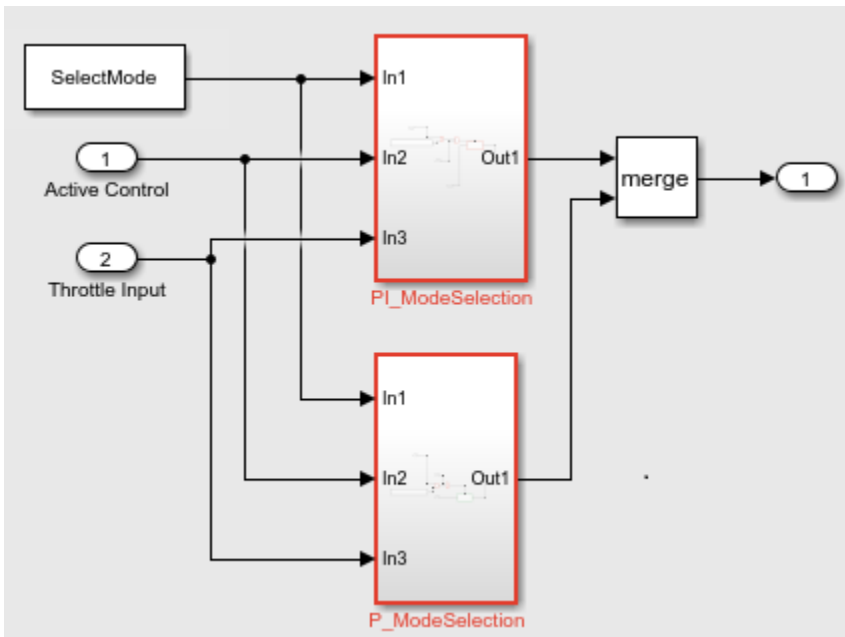
Copyright 2019 The MathWorks, Inc.

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

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

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

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



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

## Summary

### Model Hierarchy/Complexity

|   | Decision | Condition | MCDC | Test Condition | Execution |
|---|----------|-----------|------|----------------|-----------|
| 1. <a href="#">sldvexParameterController</a>    | 10 64%   | 83%       | 63%  | 100%           | 84%       |
| 2. .... <a href="#">Controller</a>              | 9 64%    | 83%       | 63%  | NA             | 84%       |
| 3. .... <a href="#">ControllerModeSelection</a> | 6 38%    | 67%       | 25%  | NA             | 67%       |
| 4. .... <a href="#">P_ModeSelection</a>         | 2 100%   | 67%       | 50%  | NA             | 100%      |
| 5. .... <a href="#">P_Controller2</a>           | 2 100%   | NA        | NA   | NA             | 100%      |
| 6. .... <a href="#">PI_ModeSelection</a>        | 4 17%    | 67%       | 0%   | NA             | 43%       |
| 7. .... <a href="#">PI_Controller1</a>          | 4 17%    | NA        | NA   | NA             | 0%        |

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

### Step 2: Configure Parameter Configurations and Extend Existing Test Cases

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

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

```

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

```

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

```

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

```

### Step 3: Perform Analysis and Review Coverage Report

Analyze the model by using the specified options.







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

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

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

## Summary

### Model Hierarchy/Complexity Test 1

|   |   | Decision   | Execution  |
|---|---|--|--|
| 1. <a href="#">sldvexRollApController</a> | 8 | 100%  | 100%  |
| 2. ... <a href="#">Roll Reference</a>     | 5 | 100%  | 100%  |
| 3. .... <a href="#">Latch Phi</a>         | 1 | 100%  | 100%  |

To complete this example, close the model.

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

### See also

- “Parameter Configuration for Analysis” on page 5-2
- “When to Extend Existing Test Cases” on page 8-2



# Detecting Design Errors

---

- “What Is Design Error Detection?” on page 6-2
- “Derived Ranges in Design Error Detection” on page 6-3
- “Analyze Models for Design Errors” on page 6-4
- “Dead Logic Detection” on page 6-7
- “Detect Dead Logic Caused by an Incorrect Value” on page 6-12
- “Common Causes for Dead Logic” on page 6-15
- “Detect Integer Overflow and Division-by-Zero Errors” on page 6-19
- “Check for Specified Minimum and Maximum Value Violations” on page 6-23
- “Detect Out of Bound Array Access Errors” on page 6-28
- “Detect Non-Finite, NaN, and Subnormal Floating-Point Values” on page 6-33
- “Detect Data Store Access Violations” on page 6-37
- “Detect Violations of High-Integrity Systems Modeling Guidelines” on page 6-41
- “Filter Objectives by Using Simulink Design Verifier Filter Explorer” on page 6-46
- “Detect Integer Overflow Errors” on page 6-51
- “Detect Out of Bound Array Access Example Model” on page 6-54
- “Detect Design Errors in C/C++ Custom Code” on page 6-57
- “Exclude and Justify Objectives for Design Error Detection” on page 6-59
- “Detect Integer Overflow in a Model with Complex Inputs” on page 6-65
- “Debug Integer Overflow Design Error Detection Using Model Slicer” on page 6-68
- “Analyzing the Results for a Dead Logic Analysis” on page 6-73

Analyzing the Results for a Dead Logic Analysis

## What Is Design Error Detection?

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

- Dead logic
- Out of bound array access
- Integer or fixed-point data overflow
- Division by zero
- Errors in floating-point usage (Inf/NaN and subnormal)
- Intermediate signal values that are outside the specified minimum and maximum values
- Data store access violations
- Specified block input range violations
- High-Integrity Systems Modeling checks

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

Model objects that have decision or condition outcomes receive dead logic detection.

After the analysis, you can:

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

### See Also

“Analyze Models for Design Errors” on page 6-4 | “Design Verifier Pane: Design Error Detection” on page 15-42



## 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:

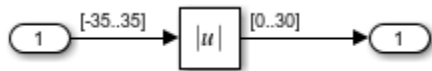
- Block Outports
- Stateflow local data

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

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

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

## Analyze Models for Design Errors

### In this section...

“Workflow for Detecting Design Errors” on page 6-4

“Understand the Analysis Results” on page 6-4

“Review the Latest Analysis Results in the Results Summary Window” on page 6-5

“Check For Design Errors using the Model Advisor” on page 6-6

### Workflow for Detecting Design Errors

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

- 1 Verify that your model is compatible with Simulink Design Verifier software.
- 2 If you have Stateflow objects in your model, in the Configuration Parameters dialog box, on the **Diagnostics > Stateflow** pane, set **Unreachable execution path** to error.
- 3 Specify options that control how Simulink Design Verifier detects design errors in your model.
- 4 Execute the Simulink Design Verifier analysis.
- 5 Review the analysis results.

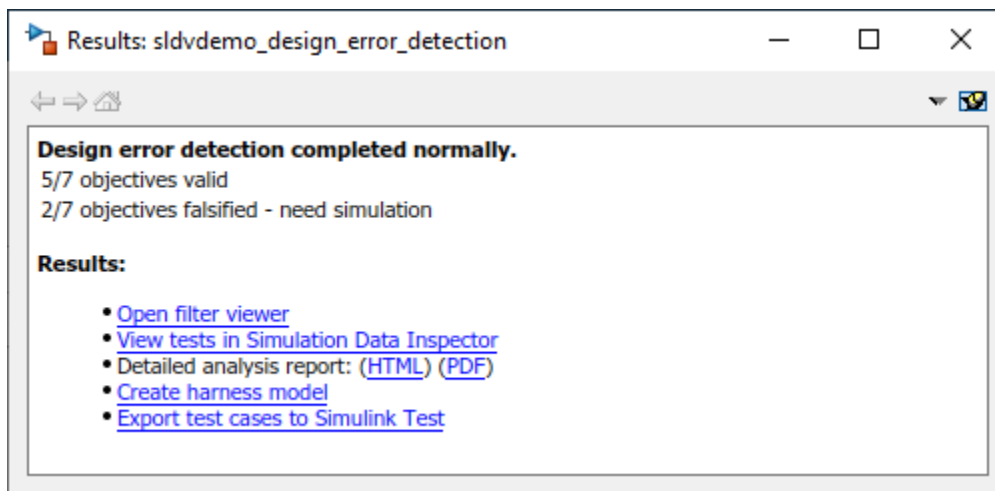
### Understand the Analysis Results

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


| Model Object Highlighting Color | Analysis Results  |
|---------------------------------|---|
| Green                           | Both of the following: <ul style="list-style-type: none"> <li>• The analysis proved the absence of dead logic.</li> <li>• The analysis proved the absence of errors for the other design error detection checks.</li> </ul> |
| Red                             | At least one of the following: <ul style="list-style-type: none"> <li>• The analysis found dead logic.</li> <li>• The analysis found an error for one of the other design error detection checks.</li> </ul>                |

| Model Object Highlighting Color | Analysis Results  |
|---------------------------------|---|
| Orange                          | <p>For at least one objective, the analysis could not determine if the model object has dead logic or one of the other design error detection 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; for more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.</li> <li>• Limitations of the analysis engine.</li> </ul> </li> </ul> |
| Gray                            | The model object was not part of the analysis.  |
| Steel blue                      | All objectives from this model object were excluded or justified using a filter files provided during the analysis.   |

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



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

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

## Review the Latest Analysis Results in the Results Summary Window

If you close the analysis results to fix the cause of the errors in your model, you might need to review the analysis results again. As long as your model remains unchanged, you can view the results of your most recent analysis results in the Results Summary Window.

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

For any Simulink Design Verifier analysis, from the Results Summary Window, you can perform the following tasks:

- Open filter explorer.
- Highlight the analysis results on the model.
- View tests in Simulation Data Inspector.
- Generate a detailed analysis report.
- Create the harness model, or if the harness model already exists, open it.

---

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

---

- Export test cases to Simulink Test.
- View the data file.
- View the log file.

## Check For Design Errors using the Model Advisor

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

- 1 Specify options that control how Simulink Design Verifier detects design errors in your model.
- 2 Open the Model Advisor.
- 3 From the system hierarchy, select the model or model component you want to analyze
- 4 Expand the design error detection analysis items. Look for Simulink Design Verifier under either **By Product** or **By Task**.
- 5 If you have not checked your model for compatibility, enable the compatibility check for Simulink Design Verifier.
- 6 Select the design error detection checks you want to run.
- 7 Run the selected checks.
- 8 Review the analysis results.

## See Also

### More About

- “Check Your Model Using the Model Advisor”

## Dead Logic Detection

### In this section...

“Run a Partial Check for Dead Logic” on page 6-7

“Run an Exhaustive Analysis for Dead Logic” on page 6-7

“Run a Dead Logic Analysis and Review Results” on page 6-8

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

- **Dead logic (partial):** If you select this option, Simulink Design Verifier analyzes your model without making any approximations, such as rational approximation for floating points, or while loop approximation. For more information, see “Role of Approximations During Model Analysis” on page 2-20. With this option, Simulink Design Verifier does not report active logic or undecided objectives and it may not identify some dead logic in your model.

This option is available in:

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

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

### Run a Partial Check for Dead Logic

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

- 1 On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
- 2 Click **Error Detection Settings**.
- 3 In the Configuration Parameters dialog box, on the **Design Verifier > Design Error Detection** pane:
  - a Enable the “Dead logic (partial)” on page 15-43 option.
  - b Clear the “Run exhaustive analysis” on page 15-43 option, if it is selected.
  - c Set “Coverage objectives to be analyzed” on page 15-44 to MCDC. The available options from the drop-down menu are **Decision**, **Condition Decision**, and **MCDC**.
- 4 To apply these settings, click **OK** and close the Configuration Parameters dialog box.
- 5 Click **Detect Design Errors**.

### Run an Exhaustive Analysis for Dead Logic

- 1 On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
- 2 Click **Error Detection Settings**.

- 3** In the Configuration Parameters dialog box, on the **Design Verifier > Design Error Detection** pane:
  - a** Enable the “Dead logic (partial)” on page 15-43 option.
  - b** Clear the “Run exhaustive analysis” on page 15-43 option, if it is selected.
  - c** Set “Coverage objectives to be analyzed” on page 15-44 to MCDC. The available options from the drop-down menu are **Decision**, **Condition Decision**, and **MCDC**.
- 4** To apply these settings, click **OK** and close the Configuration Parameters dialog box.
- 5** Click **Detect Design Errors**.

## Run a Dead Logic Analysis and Review Results

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

- 1** Open the `sldvSlicerdemo_dead_logic` model.

```
open_system('sldvSlicerdemo_dead_logic');
```
- 2** On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
- 3** Click **Error Detection Settings**.
- 4** In the Configuration Parameters dialog box, on the **Design Verifier > Design Error Detection** pane:
  - a** Enable the “Dead logic (partial)” on page 15-43 option.
  - b** Clear the “Run exhaustive analysis” on page 15-43 option, if it is selected.
  - c** Set “Coverage objectives to be analyzed” on page 15-44 to MCDC. The available options from the drop-down menu are **Decision**, **Condition Decision**, and **MCDC**.
- 5** Click **Detect Design Errors**.

The software analyzes the model for dead logic and displays the results in the Results Summary window. The result indicates that 10 of the 32 objectives were found to be dead logic.

Simulink Design Verifier Results Summary: sldvSlicerdemo\_dead\_logic

|                      |   |
|----------------------|---|
| Progress             | <div style="width: 100%; height: 10px; background-color: red;"></div> |
| Objectives processed | 24/24   |
| Valid                | 0   |
| Falsified            | 7   |
| Elapsed time         | 0:38  |

Design error detection completed normally.

Simulink Design Verifier ran a partial check for dead logic. Consider enabling the 'Dead logic > Run exhaustive analysis' configuration option in order to perform an exhaustive analysis.

7/24 objectives are dead logic

Results:

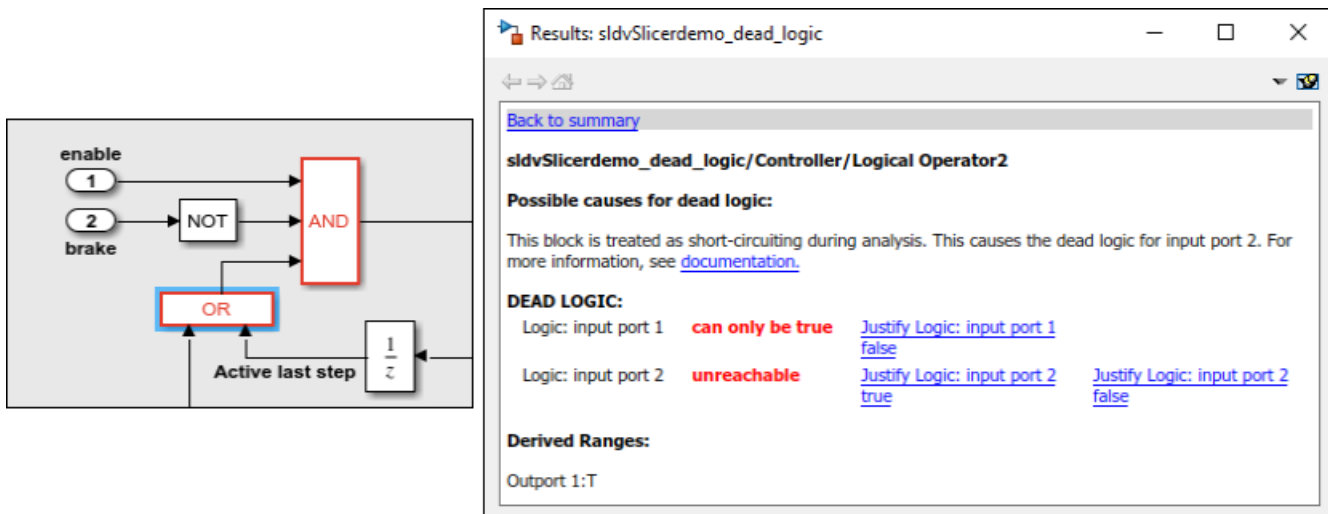
- [Open filter viewer](#)
- [Highlight analysis results on model](#)
- Detailed analysis report: ([HTML](#)) ([PDF](#))

Data saved in: [sldvSlicerdemo\\_dead\\_logic\\_sldvdata.mat](#)  
 in folder: [.matlab\sldv\\_output\sldvSlicerdemo\\_dead\\_logic](#)

[View Log](#) [Close](#)

- 6 Click **Highlight analysis results on model**. The dead logic model elements are highlighted in red.
- 7 Open the Controller subsystem, and click the OR block highlighted in red. The Result Inspector displays the summary of the dead logic.

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



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

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

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

### Dead Logic

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

### Suggestion:

You can use Model Slicer to find the parameters which could have an impact on a particular block by following these steps:

- a. Create an object of `SLSlicerAPI.ParameterDependence` using Model Slicer.

```
slicerObj = slslicer('sldvSlicerdemo_dead_logic');
pd = slicerObj.parameterDependence;
```

- b. Find the parameters affecting the **Discrete-Time Integrator** block.

```
param = parametersAffectingBlock(pd, 'sldvSlicerdemo_dead_logic/Controller/PI Controller/Discrete-Time Integrator');
```



```
param =
```

```
VariableUsage with properties:
```

```
    Name: 'c'  
    Source: 'base workspace'  
    SourceType: 'base workspace'  
    Users: {'sldvSlicerdemo_dead_logic/Constant'}
```

The image above displays the parameters returned by the function **parametersAffectingBlock** which have an impact on the **Discrete-Time Integrator** block. The parameters returned by the function can be considered for tuning.

c. Perform clean-up to exit compile state of the model.

```
slicerObj.terminate;
```

## See Also

## More About

- “Design Verifier Pane: Design Error Detection” on page 15-42

## Detect Dead Logic Caused by an Incorrect Value

### In this section...

“Analyze the Fuel System Model” on page 6-12

“Review the Results and Trace to the Model” on page 6-13

“Investigate the Cause of the Dead Logic” on page 6-13

“Update the Input Constraint and Reanalyze the Model” on page 6-14

Dead logic detection helps you to identify:

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

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

### Analyze the Fuel System Model

- 1 Open the model.

```
sldvdemo_fuelsys_logic_simple
```

Ensure that the current folder is writable.

- 2 Configure dead logic detection.

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

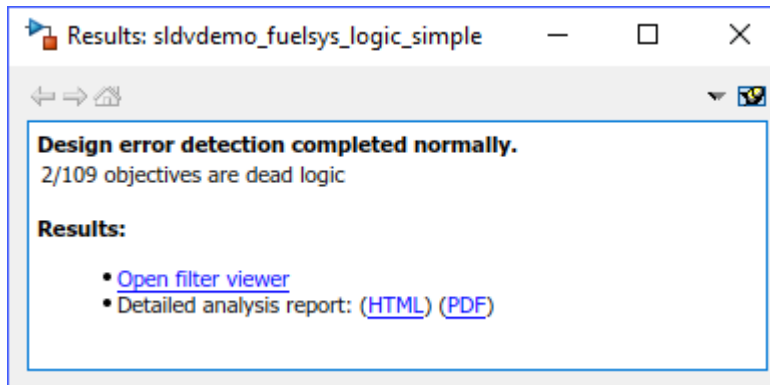
- 3 Select **Error Detection Settings**.

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

- a Enable the “Dead logic (partial)” on page 15-43 option.
- b Clear the “Run exhaustive analysis” on page 15-43 option, if it is selected.
- c Set **Coverage objectives to be analyzed** to **Condition Decision**. The available options from the drop-down menu are **Decision**, **Condition Decision**, and **MCDC**.

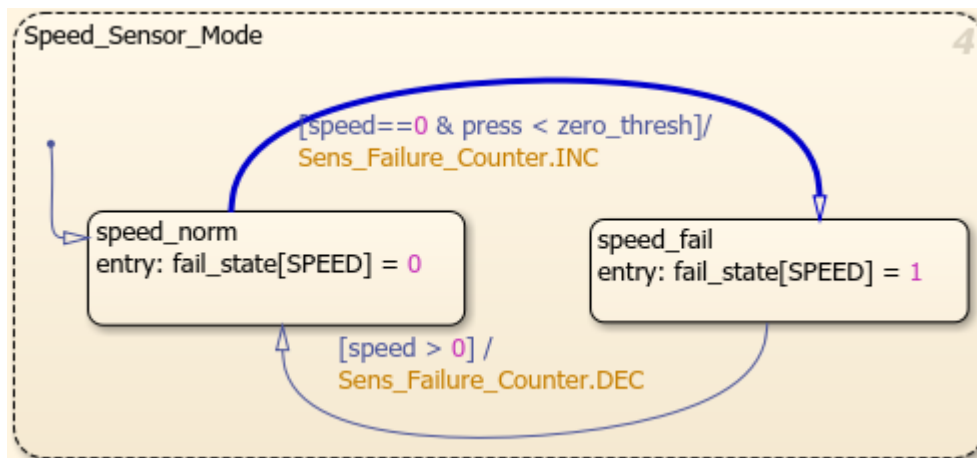
- 5 Click **Detect Design Errors**.

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



## Review the Results and Trace to the Model

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



## Investigate the Cause of the Dead Logic

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

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

## **Update the Input Constraint and Reanalyze the Model**

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

## **See Also**

### **Related Examples**

- “Dead Logic Detection” on page 6-7

## Common Causes for Dead Logic

Common modeling patterns that lead to dead logic in a model include:

### In this section...

“Short-Circuiting of a Logical Operator Block During Analysis” on page 6-15

“Conditional Execution of a Block” on page 6-15

“Parameter Values Treated as Constants” on page 6-16

“Upstream Blocks” on page 6-17

“Library-Linked Blocks” on page 6-17

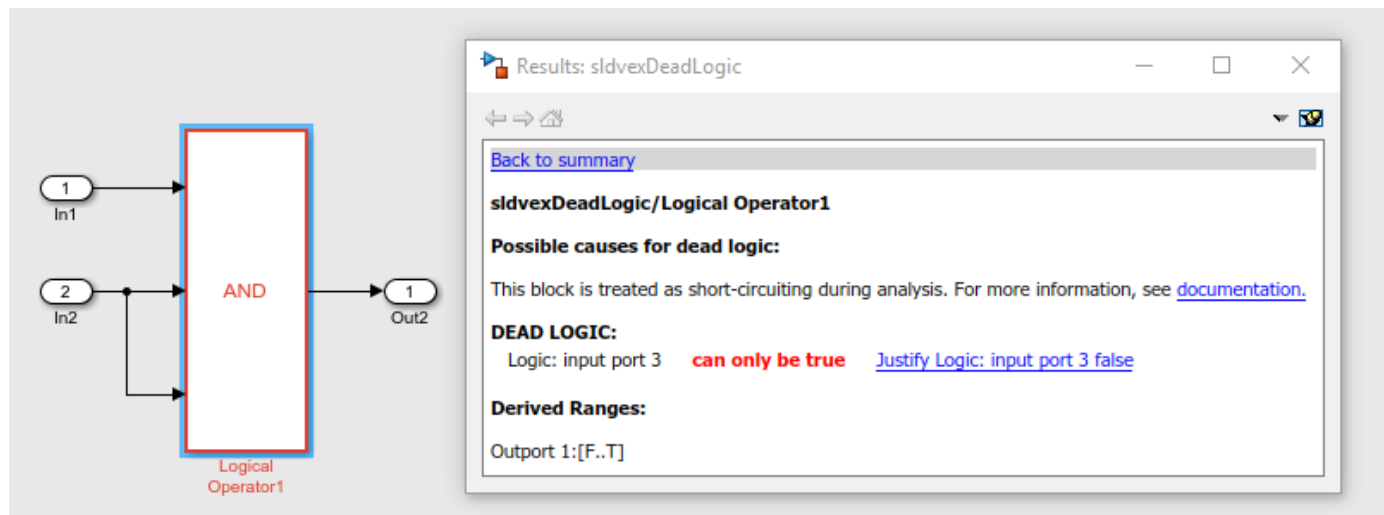
“Restrictions on Signal Ranges” on page 6-17

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

### Short-Circuiting of a Logical Operator Block During Analysis

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

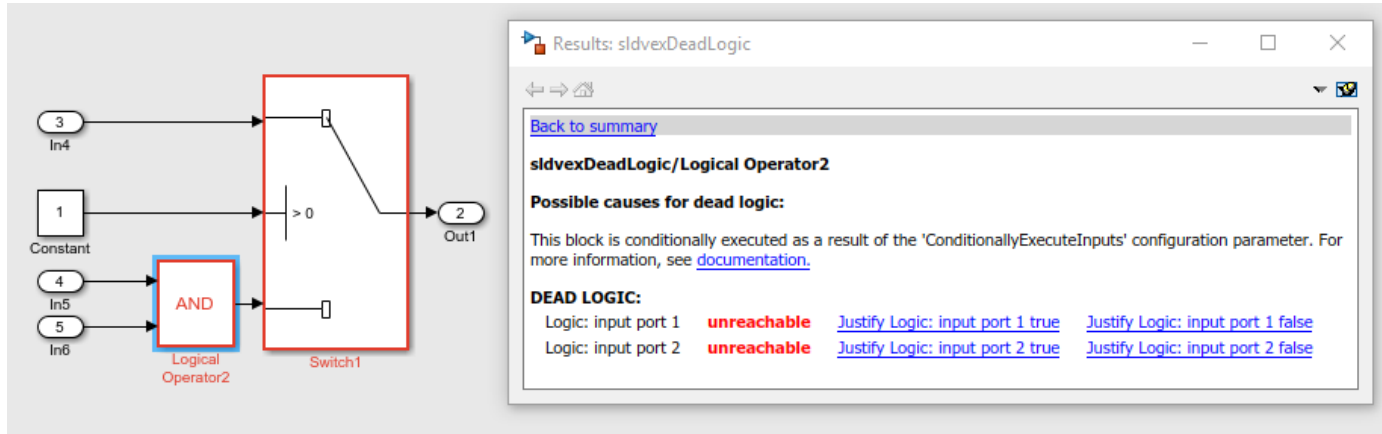
For example, in this model, if In2 is false, the software ignores the third input due to the short-circuiting. The Results window lists this port as dead logic. See “Logic Operations Short-Circuiting” on page 2-26.



### Conditional Execution of a Block

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

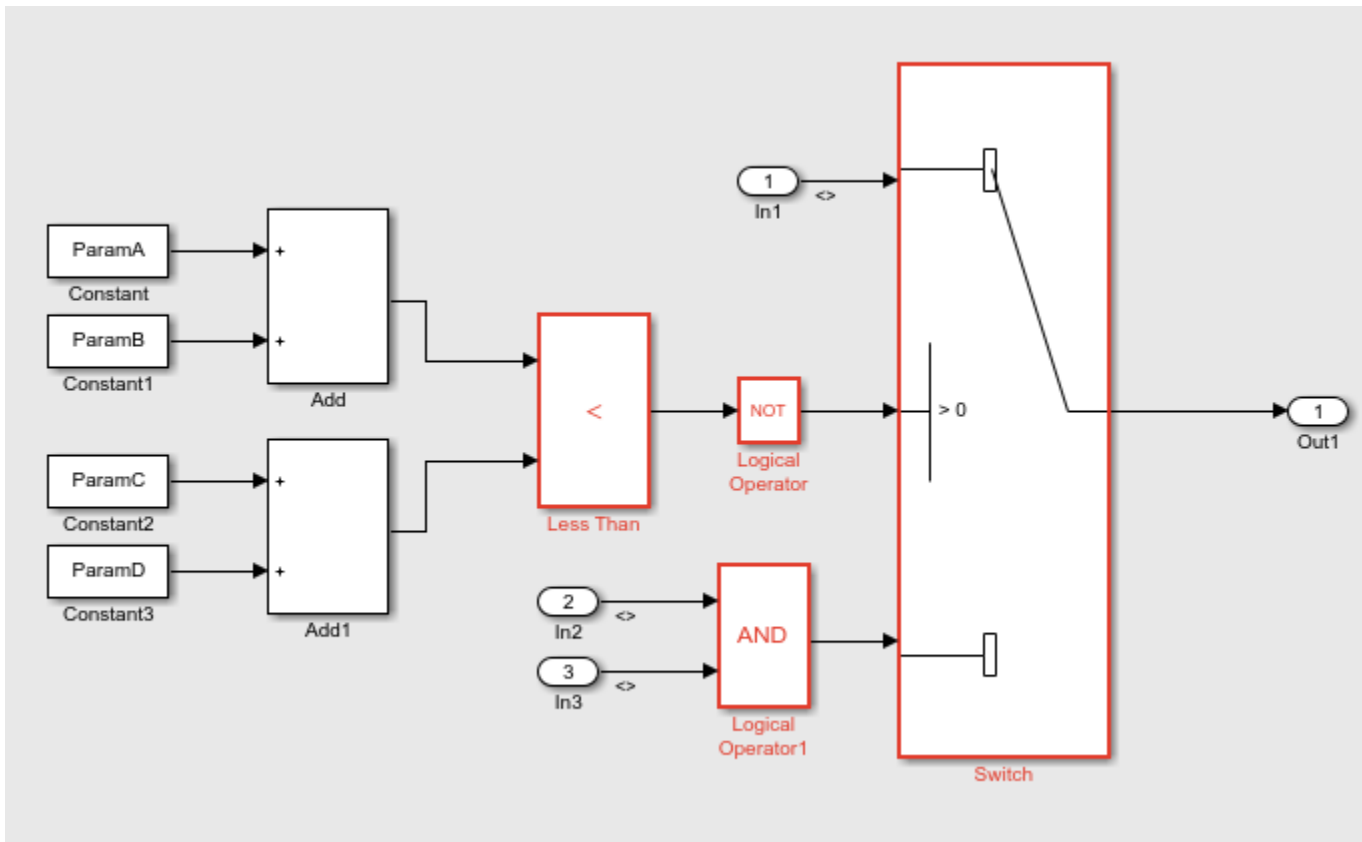
Consider this example model where the **Conditional input branch execution** parameter is set to On. The AND Logical Operator block is conditionally executed, which causes the dead logic for the block. For more information, see “Conditional input branch execution”.



## Parameter Values Treated as Constants

If your model contains parameters, Simulink Design Verifier treats the values as constants by default. This might cause dead logic in the model. In these cases, consider configuring these parameters to be tuned during analysis.

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



## Upstream Blocks

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

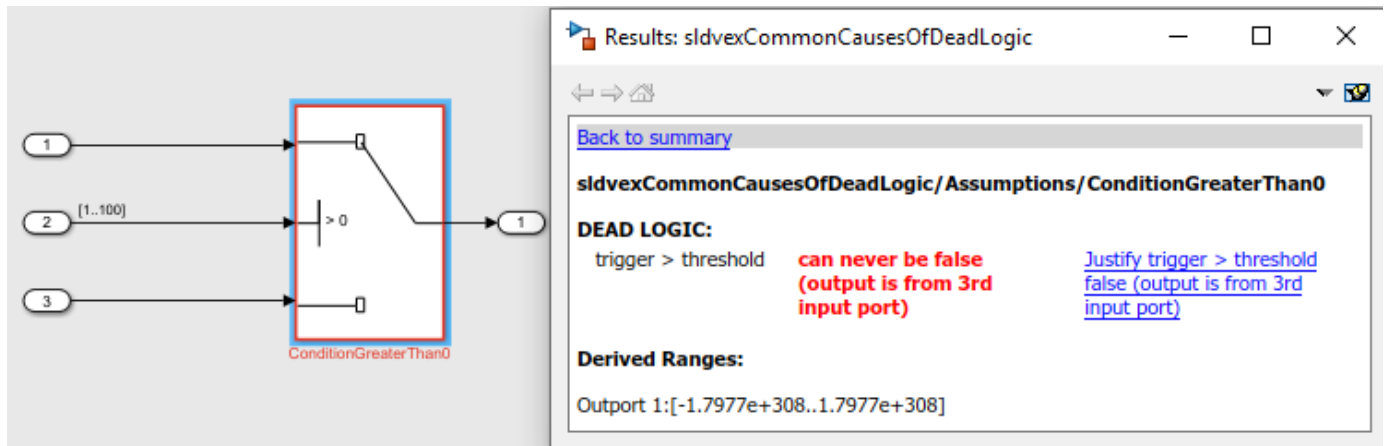
Consider the above example model. The dead logic in the Less Than block causes the dead logic in the corresponding downstream blocks. It is therefore often helpful to review the upstream dead logic before reviewing any downstream dead logic.

## Library-Linked Blocks

Library blocks may be written with defensive conditions that are redundant in some of the locations where they are used. In some cases, this may cause dead logic. See “Exclude and Justify Objectives for Design Error Detection” on page 6-59.

## Restrictions on Signal Ranges

Root-level Inport blocks with minimum and maximum values as constraints and Test Condition blocks in the test generation may cause dead logic. For example, consider ConditionGreaterThan0 Switch block, where the second Inport block has a minimum and maximum range of 1 and 100, respectively. This causes the Switch block in this subsystem to have dead logic.



The image shows a logic analyzer interface. On the left, a block labeled 'ConditionGreaterThan0' is highlighted with a red border. It has three input ports labeled 1, 2, and 3. Input 2 has a range of [1..100]. The block contains a logic symbol for a greater-than comparison (> 0). The output of the block is labeled 1. On the right, a window titled 'Results: sldvexCommonCausesOfDeadLogic' displays the analysis results. It includes a 'Back to summary' link, the path 'sldvexCommonCausesOfDeadLogic/ Assumptions/ ConditionGreaterThan0', and a 'DEAD LOGIC' section. The 'DEAD LOGIC' section shows the condition 'trigger > threshold' with a red note stating 'can never be false (output is from 3rd input port)' and a blue link 'Justify trigger > threshold false (output is from 3rd input port)'. Below this, a 'Derived Ranges' section shows 'Output 1: [-1.7977e+308..1.7977e+308]'.

## See Also

### More About

- “Run a Dead Logic Analysis and Review Results” on page 6-8
- “Analyzing the Results for a Dead Logic Analysis” on page 6-73



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

### In this section...

“About This Example” on page 6-19

“Analyze the Model” on page 6-19

“Review the Analysis Results” on page 6-19

### About This Example

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

### Analyze the Model

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

- 1 Open the `sldvdemo_cruise_control_fxp_fixed` model.
- 2 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
- 3 In the Configuration Parameters dialog box, select **Design Verifier > Design Error Detection**.
- 4 On the **Design Error Detection** pane, select:
  - **Integer overflow**
  - **Division by zero**
- 5 In the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane, set **Signals > Wrap on overflow**, **Signals > Saturate on overflow** and **Parameters > Detect overflow** to error.
- 6 Click **OK** to save these settings and close the Configuration Parameters dialog box.
- 7 In the **Mode** section, select **Design Error Detection**.
- 8 Click **Detect Design Errors**.

When the analysis is complete:

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

### Review the Analysis Results

- “Review the Results on the Model” on page 6-19
- “Review the Harness Model” on page 6-21
- “Review the Analysis Report” on page 6-22

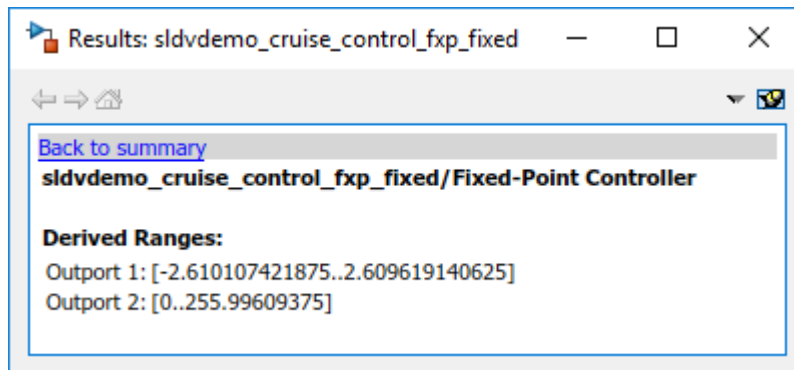
### Review the Results on the Model

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

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

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

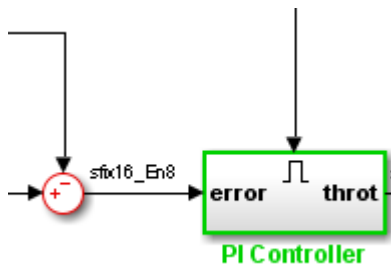
- The values of Output 1 (throt) range from  $-2.6101$  to  $2.6096$ .
- The values of Output 2 (target) range from  $0$  to  $255.9960$ .



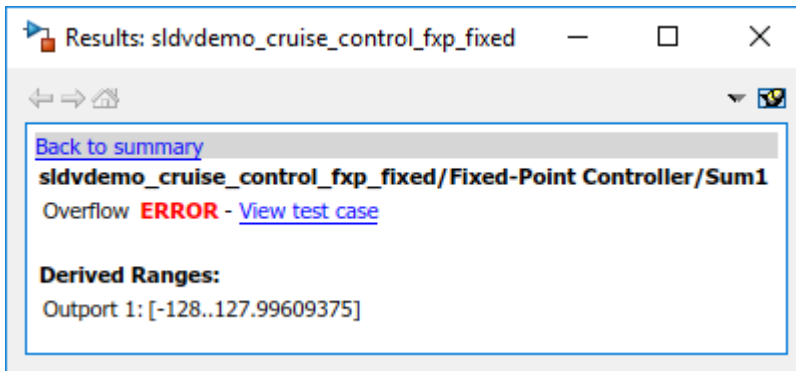
- 2 Click the Output blocks of the `sldvdemo_cruise_control_fxp_fixed` model to see the same signal bound values.
- 3 Open the Fixed-Point Controller subsystem.

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

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



This Sum block can produce an overflow error. The analysis found a test case that can result in a computation where the output of the Sum block exceeds the range  $[-128..127.9960]$ .



- 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 Outputport from the Bus block has a range of [0..256].
  - The Outputport 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 Outputport of the Sum block.

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

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

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

### Review the Harness Model

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

- 1 In the `sldvdemo_cruise_control_fxp_fixed` model, open the Fixed-Point Controller subsystem.
- 2 Click the Sum block, outlined in red, that provides the error input to the PI Controller subsystem.

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

- 3 In the Simulink Design Verifier Results window, click **View counterexamples**.

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

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

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

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

For more information, see “Manage Simulink Design Verifier Harness Models” on page 13-13.

### Review the Analysis Report

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

- 1 In the Simulink Design Verifier Results window, to redisplay the results summary, click **Back to summary**.
- 2 Click **Generate detailed analysis report**.

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

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

- **Objectives Valid** — Model objects that did not produce errors
- **Objectives Falsified with Counterexamples** — Model objects for which test cases generated errors

Model objects that have decision or condition outcomes receive dead logic detection. For more information on the complete list of model objects that have decision or condition objectives, see “Model Objects That Receive Coverage” (Simulink Coverage).

For more information, see “Review Results” on page 13-35.

### See Also

#### More About

- “Detect Integer Overflow Errors” on page 6-51
- “Detect Integer Overflow in a Model with Complex Inputs” on page 6-65

## Check for Specified Minimum and Maximum Value Violations

### In this section...

“Limitations of Checking Specified Minimum and Maximum Value Violations” on page 6-23  
“About This Example” on page 6-23  
“Create the Example Model” on page 6-24  
“Analyze the Model” on page 6-25  
“Review the Analysis Results” on page 6-25

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

The analysis checks for specified minimum and maximum values on:

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

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

### Limitations of Checking Specified Minimum and Maximum Value Violations

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

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

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

---

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

---

### About This Example

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

- The input ports

- The output ports of two of the intermediate blocks

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

## Create the Example Model

Create the model for this example:

- 1 In the **MATLAB** toolstrip, on the **Home** tab, select **New > Simulink Model**.
- 2 From the Simulink Commonly Used Blocks library, add the following blocks to the model and assign the indicated parameter values.

| Block      | Tab                      | Parameter               | Value |
|------------|--------------------------|-------------------------|-------|
| Inport     | <b>Signal Attributes</b> | <b>Minimum</b>          | 0     |
| Inport     | <b>Signal Attributes</b> | <b>Maximum</b>          | 5     |
| Gain       | <b>Main</b>              | <b>Gain</b>             | 5     |
| Gain       | <b>Signal Attributes</b> | <b>Output minimum</b>   | 0     |
| Gain       | <b>Signal Attributes</b> | <b>Output maximum</b>   | 20    |
| Gain       | <b>Signal Attributes</b> | <b>Output data type</b> | int16 |
| Saturation | <b>Main</b>              | <b>Upper limit</b>      | 25    |
| Saturation | <b>Main</b>              | <b>Lower limit</b>      | -25   |
| Saturation | <b>Signal Attributes</b> | <b>Output minimum</b>   | -25   |
| Saturation | <b>Signal Attributes</b> | <b>Output maximum</b>   | 25    |
| Output     | No changes               |                         |       |

- 3 Connect the four blocks as shown.



- 4 To display the specified minimum and maximum values, on the **Debug** tab, select **Information Overlays > Signal Data Ranges**.
- 5 On the **Modeling** tab, click **Model Settings**.
- 6 In the Configuration Parameters dialog box, on the **Solver** pane, under **Solver selection**:
  - a Set **Type** to Fixed-step.  
The Simulink Design Verifier software does not support variable-step solvers.
  - b Set **Solver** to discrete (no continuous states).
- 7 On the **Design Verifier** pane, set **Mode** to Design error detection.
- 8 On the **Design Verifier > Design Error Detection** pane:
  - a Select **Specified minimum and maximum value violations**.

- b** Clear the **Integer overflow** and **Division by zero** parameters.

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

- 9** To save these settings and exit the Configuration Parameters dialog box, click **OK**.
- 10** Save the model and name it `ex_interim_minmax`.

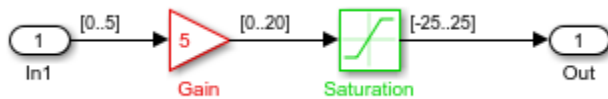
## Analyze the Model

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

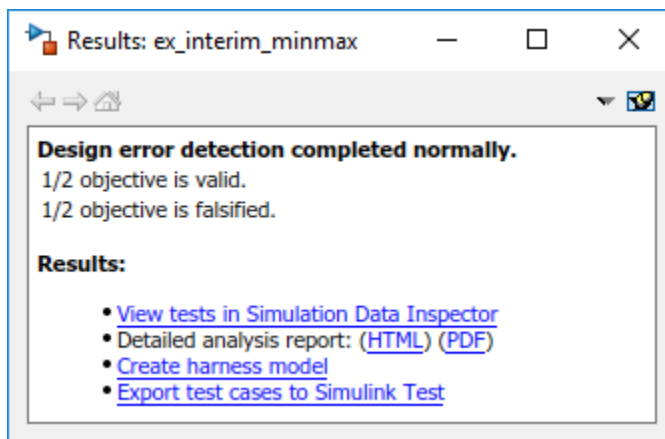
On the **Design Verifier** tab, click **Detect Design Errors**.

After the analysis is complete:

- The software highlights the model with the analysis results.



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



## Review the Analysis Results

- “Review Results on the Model” on page 6-25
- “Review the Harness Model” on page 6-26
- “Review the Analysis Report” on page 6-27

### Review Results on the Model

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

- At least one objective associated with the Gain block was falsified. For this example, the analysis falsified exactly one objective.

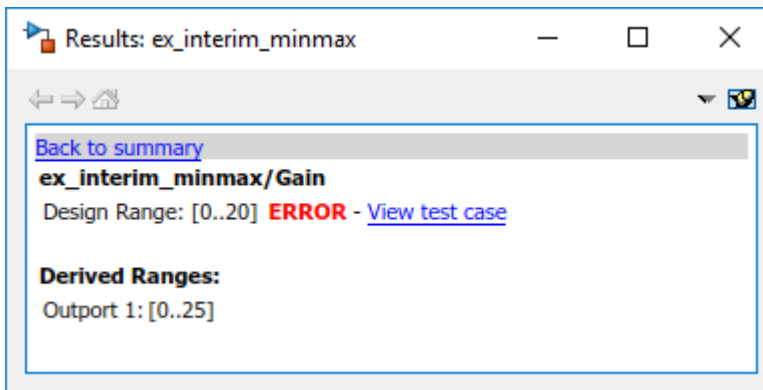
- All objectives associated with the Saturation block were satisfied. For this example, the analysis satisfied exactly one objective.

To understand these results:

- 1 Click the Gain block.

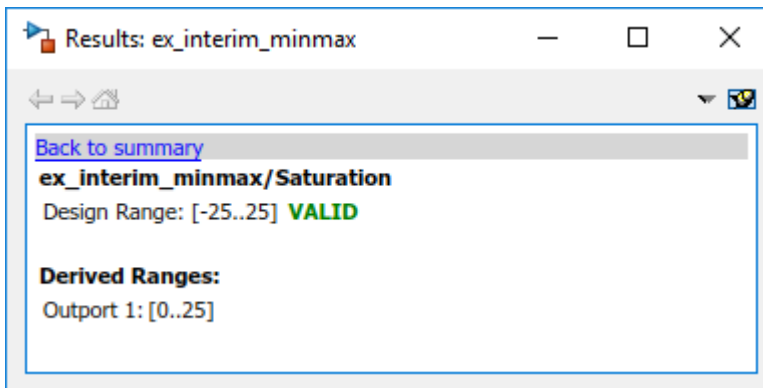
The Simulink Design Verifier Results window shows that the design range for the output was [0..20], but the analysis detected an error and generated a test case that demonstrates that error. Because the design range for the input block is [0..5], when the input to the Gain block is 5, the output is 25, which exceeds the specified maximum value on that port.

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



- 2 Click the Saturation block.

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



## Review the Harness Model

When the analysis completes, you can create a harness model contains the test cases that result in errors.

For the example model, view the test case that caused the design range error in the Gain block:



- 1 After the analysis completes and the model is highlighted, click the Gain block.
- 2 In the Simulink Design Verifier Results window, click **View test case**.

The software creates a harness model named `ex_interim_minmax_harness` and opens the Signal Builder block in the harness model that contains the test case.

In the Signal Builder block, one test case, whose signal value is 5, caused the output of the Gain block to be 25, which exceeds the specified maximum of 20.

- 3 Before you simulate this test case, in the Configuration Parameters dialog box, on the **Diagnostics > Data Validity** pane, set **Simulation range checking** to warning or error.

Setting this parameter specifies the diagnostic action to take if Simulink detects signals that exceed specified minimum or maximum values during simulation.

- If you specify `warning`, the simulation displays a warning message and continues.
- If you specify `error`, the simulation displays an error message and stops.

- 4 Click **OK** to save your change and close the Configuration Parameters dialog box.
- 5 In the Signal Builder block window, click **Start simulation** to simulate the model with this test case.

As expected, in the MATLAB window, the simulation displays a warning or error that the output value of the Gain block exceeds the specified maximum.

### Review the Analysis Report

You can also generate an HTML report containing detailed information about the analysis report for the `ex_interim_minmax` model. To create this report, in the Simulink Design Verifier Results window, click **Generate detailed analysis report**. The analysis report opens in a browser.

In the analysis report, the **Design Error Detection Objectives Status** chapter of the report provides detailed results in two categories:

- **Objectives Proven Valid** — The output values for the Saturation block are always within the design range.
- **Objectives Falsified with Test Cases** — The output values for the Gain block violated the design range.

## Detect Out of Bound Array Access Errors

### In this section...

“Design Error Detection for Out of Bound Array Access” on page 6-28

“Detect Out of Bound Array Access Example Model” on page 6-28

“Limitations of Support for Out of Bound Array Access Design Error Detection” on page 6-31

### Design Error Detection for Out of Bound Array Access

Simulink Design Verifier design error detection analysis detects out of bound array access errors in your model. In simulation, when your model attempts to access an array element using an invalid index, an out of bound array access error occurs.

To detect out of bound array access errors in your model:

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

The Simulink Design Verifier log window opens, showing the progress of the analysis.

When the analysis is complete:

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

---

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

---

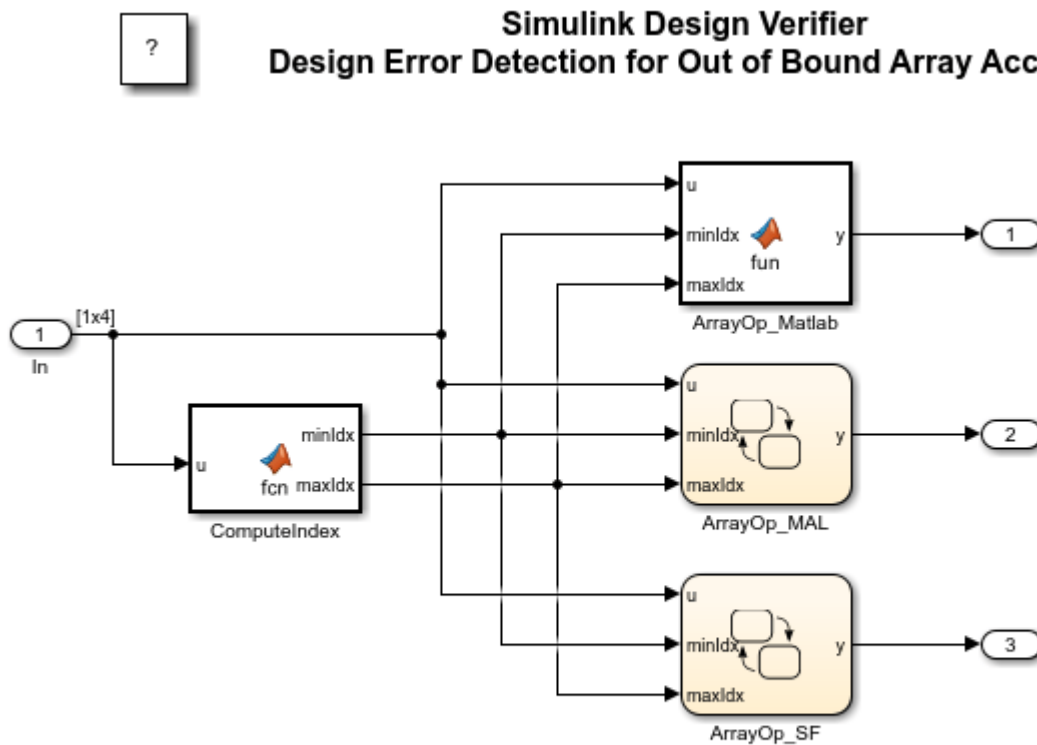
### Detect Out of Bound Array Access Example Model

This example shows how to detect out of bound array access errors and review the analysis results. In the `sldvdemo_array_bounds` example model, the ComputeIndex MATLAB Function block uses the input signal values to determine range of indices with minimum `minIdx` and maximum `maxIdx`. The `ArrayOp_Matlab`, `ArrayOp_MAL`, and `ArrayOp_SF` blocks use the set of integer indices between `minIdx` and `maxIdx` to access array elements and perform array operations.

#### Step 1: Open the Model

At the command prompt, enter:

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

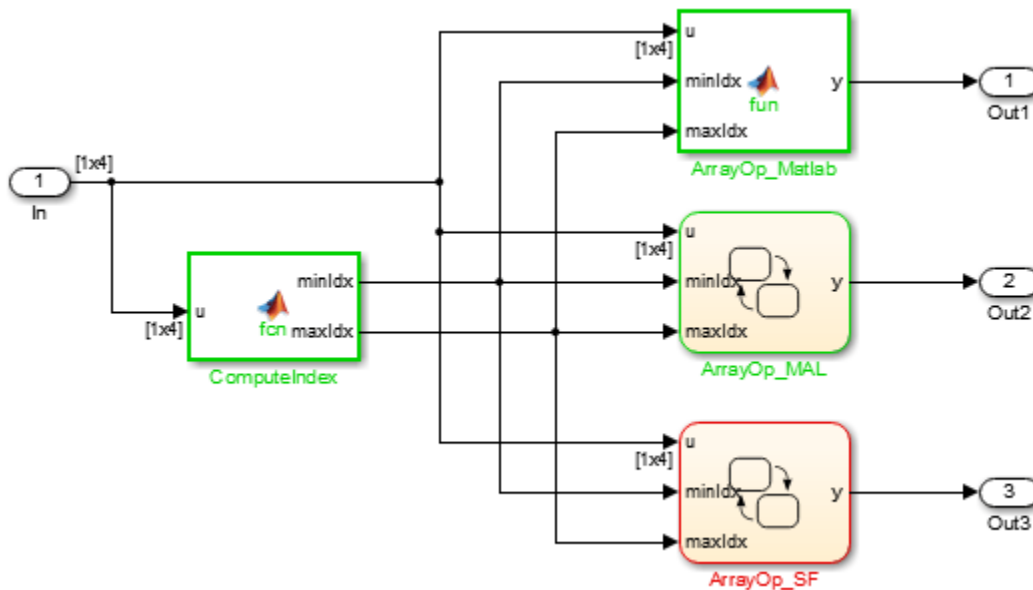


Copyright 2010-2019 The MathWorks, Inc.

## Step 2: Perform Design Error Detection Analysis

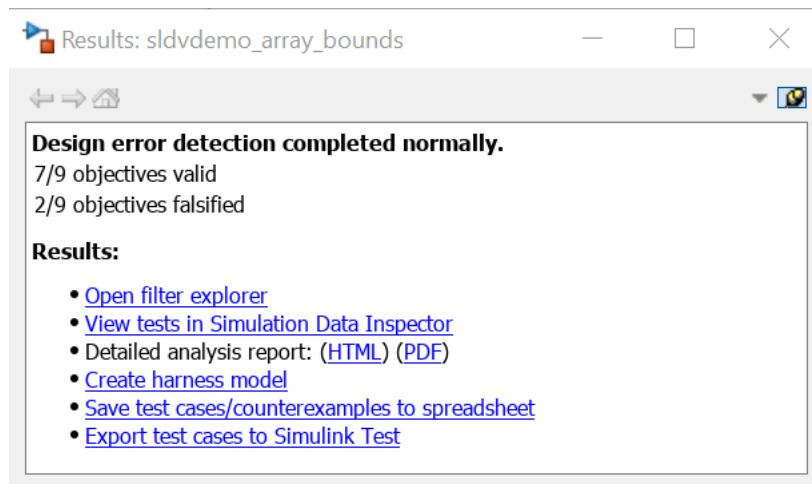
The analysis options in the model are preconfigured for out of bound array access error detection. To view these options, in the Simulink Editor, double-click the **View Options** button.

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



### Step 3: Review Analysis Results

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



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

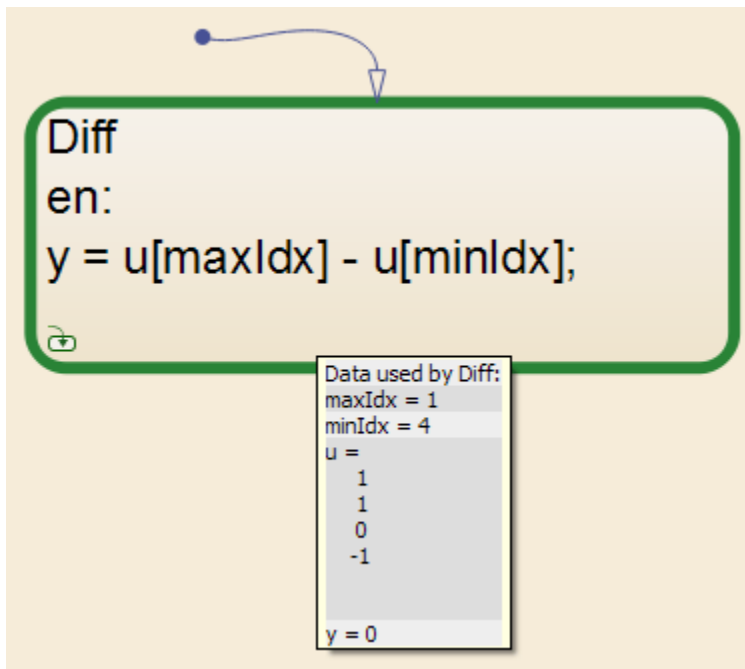
### Step 4: Create Harness and Simulate Test Cases

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

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

Attempted to access index 4 of `u` with smaller dimension sizes. The valid index range is 0 to 3. This error will stop the simulation. State 'Diff' in Chart 'sldvdemo\_array\_bounds\_harness/Test Unit (copied from sldvdemo\_array\_bounds)/ArrayOp\_SF': `y = u[maxIdx] - u[minIdx];`

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



Using Test Case 2 input signal values, the ComputeIndex MATLAB Function block determines the range of array indices to be 1:4. One-based indexing is consistent with MATLAB syntax, so these indices are valid for the ArrayOp\_Matlab MATLAB Function block and the ArrayOp\_MAL Stateflow® chart.

The ArrayOp\_SF Stateflow® chart uses C as the action language, which does not support one-based indexing. Thus, 1:4 is not a valid index range for array access in the chart. The valid index range for array access in the chart is 0:3, as reported by the error message. When either `maxIdx` or `minIdx` evaluates to 4, an out of bound array access error occurs in the ArrayOp\_SF Chart block. For more information on zero-based indexing support, see “Differences Between MATLAB and C as Action Language Syntax” (Stateflow).

## Limitations of Support for Out of Bound Array Access Design Error Detection

### Inf Index Values

Design error detection does not support indexing by `Inf`. If your model attempts to access an array using an index value that evaluates to `Inf`, design error detection does not report an out of bound array access error, but in simulation, an out of bound array access error occurs.

**Index Vector Block with Scalar Data Input**

Out of bound array access design error detection does not support Index Vector blocks with scalar data inputs. If your model includes an Index Vector block that specifies a scalar data input instead of a vector data input and the control input causes an out of bounds array access, design error detection does not report an error, but an error occurs in simulation.

**See Also****More About**

- “Detect Out of Bound Array Access Example Model” on page 6-54

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

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

- 1 On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
- 2 Click **Error Detection Settings**.
- 3 In the Configuration Parameters dialog box, in **Design Error Detection** pane:
  - a Select the check box for “Non-finite and NaN floating-point values” on page 15-47.
  - b Select the check box for “Subnormal floating-point values” on page 15-47.
  - c To apply these settings, click **OK** and close the Configuration Parameters dialog box.
- 4 Click **Detect Design Errors**.

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

After the analysis is complete:

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

### Assumptions and Limitations

When you analyze a model and select “Non-finite and NaN floating-point values” on page 15-47, the software assumes that the floating-point input values and the tunable parameter values are finite.

When you analyze a model and select “Subnormal floating-point values” on page 15-47, the software assumes that the floating-point input values and the tunable parameter values are normal.

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

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

### Run Design Error Detection Analysis to Detect Floating-Point Errors

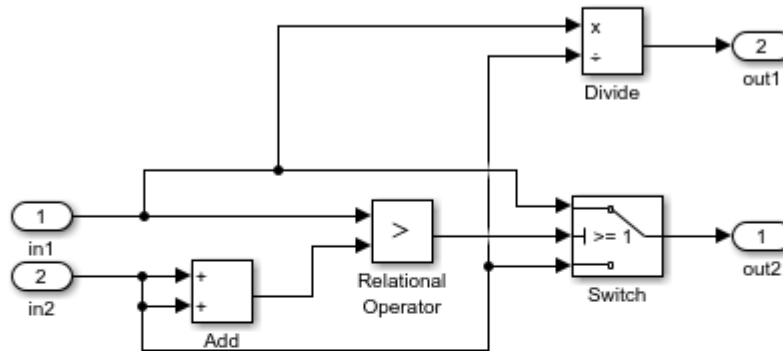
This example shows how to detect nonfinite, NaN, and subnormal floating-point values in the `sldvexFloatingPointErrorChecks` example model. The model consists of floating-point arithmetic operations that result in an error. Perform design error detection analysis to detect these errors in the model.

#### 1. Open the Model

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

```
open_system('sldvxFloatingPointErrorChecks');
```

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



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

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



Copyright 2018 The MathWorks, Inc.

### 2. Perform Design Error Detection Analysis

The model is preconfigured with **Non-finite and NaN floating-point values** and **Subnormal floating-point values** options set to **On**. For more information see “Design Verifier Pane: Design Error Detection” on page 15-42.

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

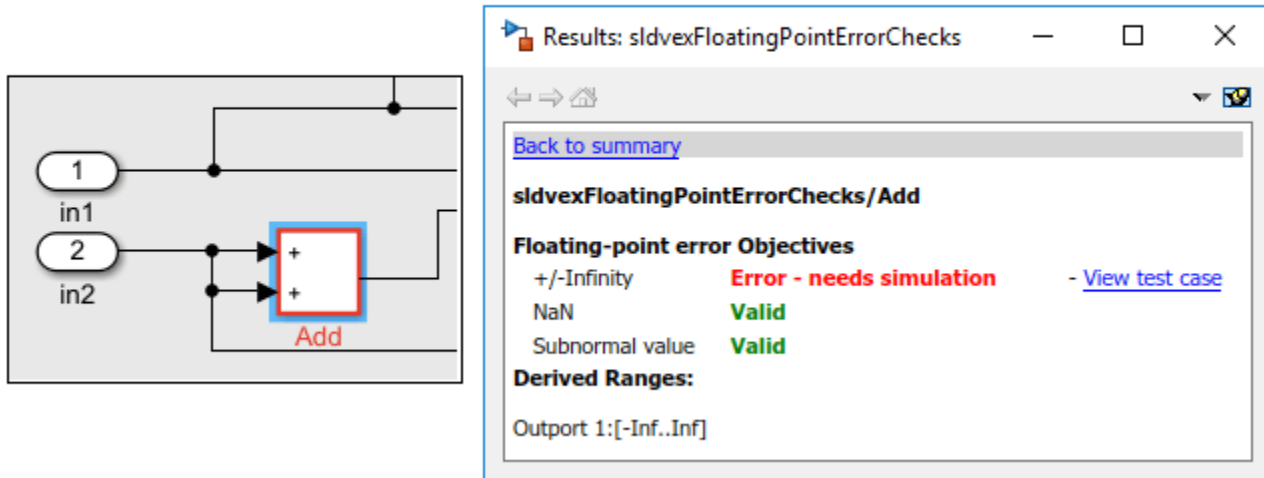
The software analyzes the model for floating-point errors and displays the results in the Results Summary window. The result indicates that 4 out of 6 objectives are falsified.

### 3. Review Analysis Results

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



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

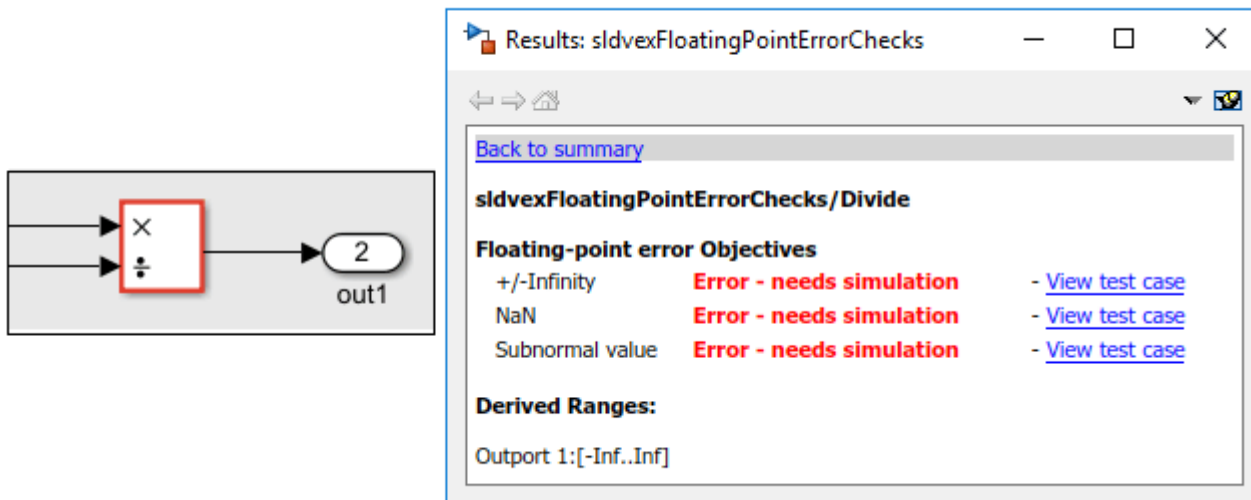


The diagram shows a block with two input ports labeled '1 in1' and '2 in2', and one output port. The block is labeled 'Add' and contains a '+' symbol. The block is highlighted with a red border. To the right is a window titled 'Results: sldvexFloatingPointErrorChecks' showing the following summary:

```

Back to summary
sldvexFloatingPointErrorChecks/Add
Floating-point error Objectives
+/-Infinity      Error - needs simulation  - View test case
NaN              Valid
Subnormal value  Valid
Derived Ranges:
Output 1:[-Inf..Inf]
  
```

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



The diagram shows a block with two input ports and one output port labeled '2 out1'. The block is labeled 'Divide' and contains a '÷' symbol. The block is highlighted with a red border. To the right is a window titled 'Results: sldvexFloatingPointErrorChecks' showing the following summary:

```

Back to summary
sldvexFloatingPointErrorChecks/Divide
Floating-point error Objectives
+/-Infinity      Error - needs simulation  - View test case
NaN              Error - needs simulation  - View test case
Subnormal value  Error - needs simulation  - View test case
Derived Ranges:
Output 1:[-Inf..Inf]
  
```

#### 4. View Detailed Analysis Report

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

## Chapter 3. Design Error Detection Objectives Status

### Table of Contents

[Objectives Valid](#)

[Objectives Falsified - Needs Simulation](#)

### Objectives Valid

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

### Objectives Falsified - Needs Simulation

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

### 5. Clean Up

To complete this example, close the model.

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

### See Also

### More About

- “Design Verifier Pane: Design Error Detection” on page 15-42
- “Simulink Design Verifier Options” on page 15-2

## Detect Data Store Access Violations

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

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

To detect data store access violations in your model:

- 1 On the **Design Verifier** tab, in the **Mode** section, select **Design Error Detection**.
- 2 Click **Error Detection Settings**.
- 3 In the Configuration Parameters dialog box, in the **Design Error Detection** pane, select “Data store access violations” on page 15-45. Click **OK**.
- 4 Click **Detect Design Errors**.

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

### Detect Data Store Access Violations in a Model

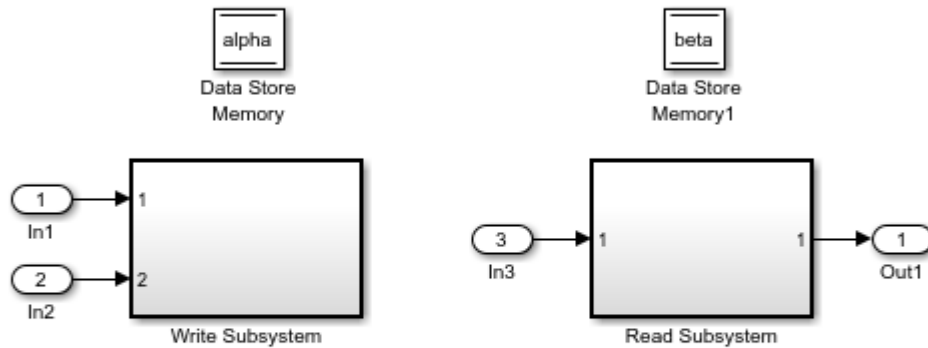
This example shows how to detect data store access violations and review the analysis results. The `sldvexDataStoreAccessViolations` example model consists of Data Store Memory blocks that define the alpha and beta data stores. In the example model, the `Write Subsystem` writes the data to the data store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

#### Step 1: Open the Model

At the command prompt, enter:

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

## Simulink Design Verifier Detect Design Error for Data Store Access Violations



This example shows how to detect data store access violations using Simulink Design Verifier. This model contains a read-before-write violation that results from the "beta" data store not being written on certain execution paths.

Copyright 2019 The MathWorks, Inc.

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

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

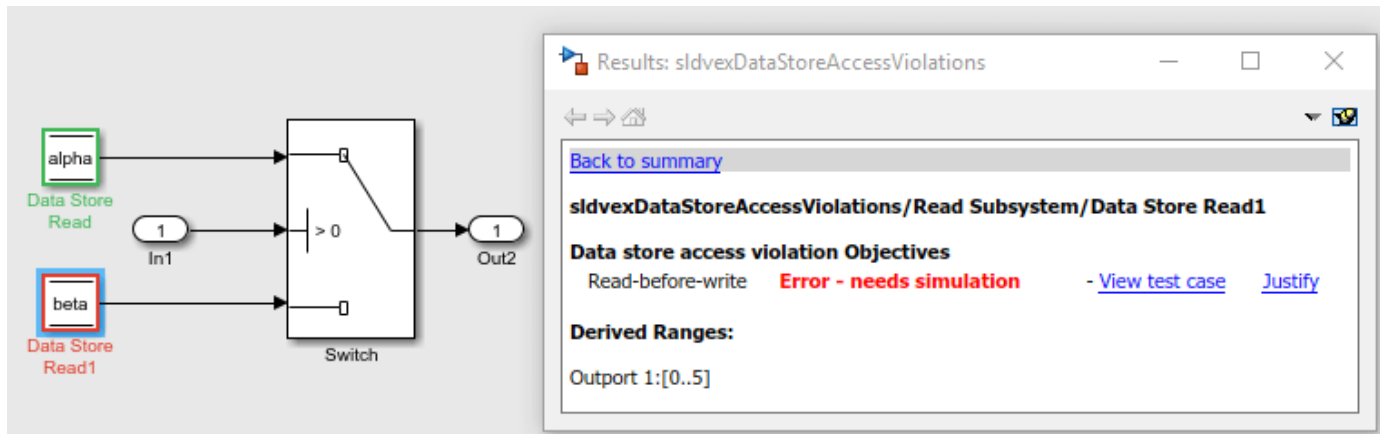
### Step 3: Perform Design Error Detection Analysis

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

### Step 4: Review Analysis Results

The model is highlighted with the analysis results.

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



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

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

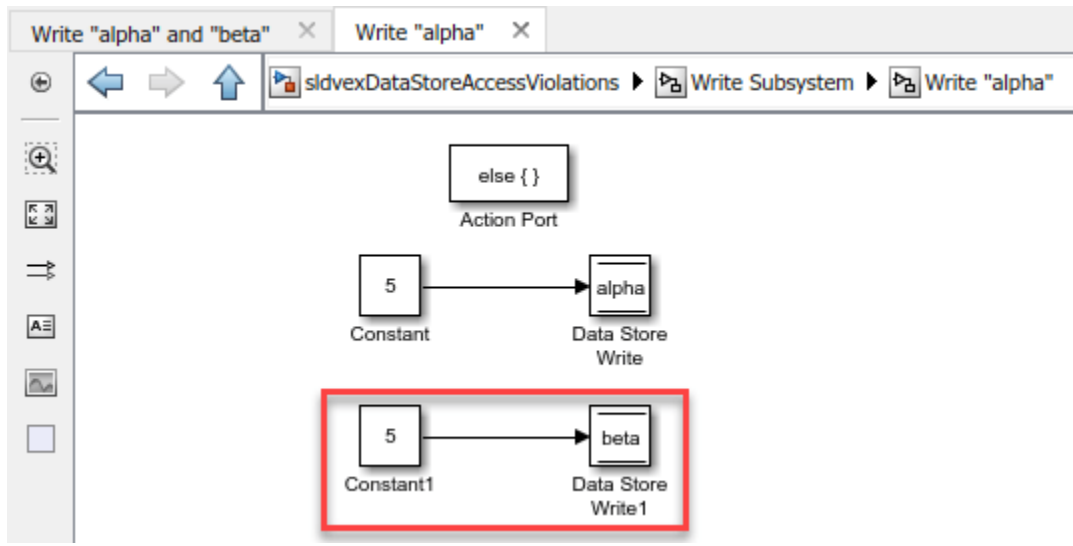
The block 'sldvexDataStoreAccessViolations\_harness/Test Unit (copied from sldvexDataStoreAccessViolations)/Read Subsystem/Data Store Read1' is reading from the data store 'sldvexDataStoreAccessViolations\_harness/Test Unit (copied from sldvexDataStoreAccessViolations)/Data Store Memory1' before any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

### Step 5: Fix the Data Store Access Violation Error

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

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

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



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

### See Also

- “Data Store Basics”
- “Detect Data Store Access Violations”

### See Also

### More About

- “Design Verifier Pane: Design Error Detection” on page 15-42

## Detect Violations of High-Integrity Systems Modeling Guidelines

Simulink Design Verifier design error detection analysis detects violations of the following High-Integrity Systems Modeling Guidelines:

- Usage of rem and reciprocal operations - hisl\_0002
- Usage of square root operations - hisl\_0003
- Usage of log and log10 operations - hisl\_0004
- Usage of Reciprocal Square Root blocks - hisl\_0028

### Usage of rem and reciprocal operations - hisl\_0002

Specify whether to check the usage of `rem` and `reciprocal` operations that cause non-finite results.

This corresponds to the hisl\_0002 check for High-Integrity Systems Modeling. For more information, see hisl\_0002: Usage of Math Function blocks (rem and reciprocal).

### Usage of square root operations - hisl\_0003

Specify whether to check the usage of Square Root operations with inputs that can be negative.

This corresponds to the hisl\_0003 check for High-Integrity Systems Modeling. For more information, see hisl\_0003: Usage of Square Root blocks.

### Usage of log and log10 operations - hisl\_0004

Specify whether to check the usage of `log` and `log10` operations that cause non-finite results.

This corresponds to the hisl\_0004 check for High-Integrity Systems Modeling. For more information, see hisl\_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm).

### Usage of Reciprocal Square Root blocks - hisl\_0028

Specify whether to check the usage of Reciprocal Square Root blocks with inputs that can go zero or negative.

This corresponds to the hisl\_0028 check for High Integrity Systems Modeling. For more information, see hisl\_0028: Usage of Reciprocal Square Root blocks.

## Detect Violations of High-Integrity Systems Modeling Guidelines

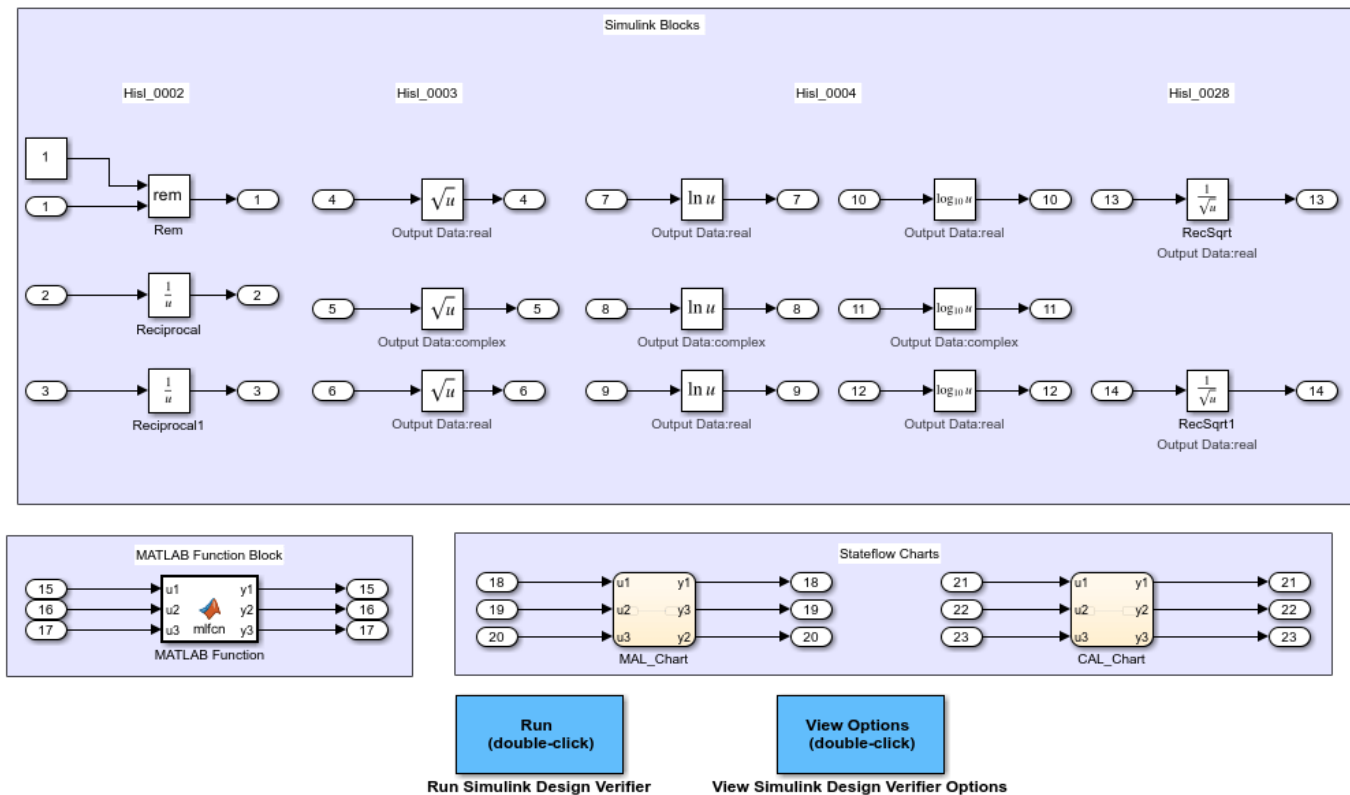
This example shows how to detect violations of High-Integrity Systems Modeling guidelines.

### 1. Open the Model

This example model explains about usage of remainder and reciprocal operations, square root operations, log and log10 operations, and Reciprocal Square Root blocks.

```
open_system('sldvexHislChecks');
```

### Simulink Design Verifier Design Error Detection for High-Integrity Systems Modeling Guidelines



Copyright 2021 The MathWorks, Inc.

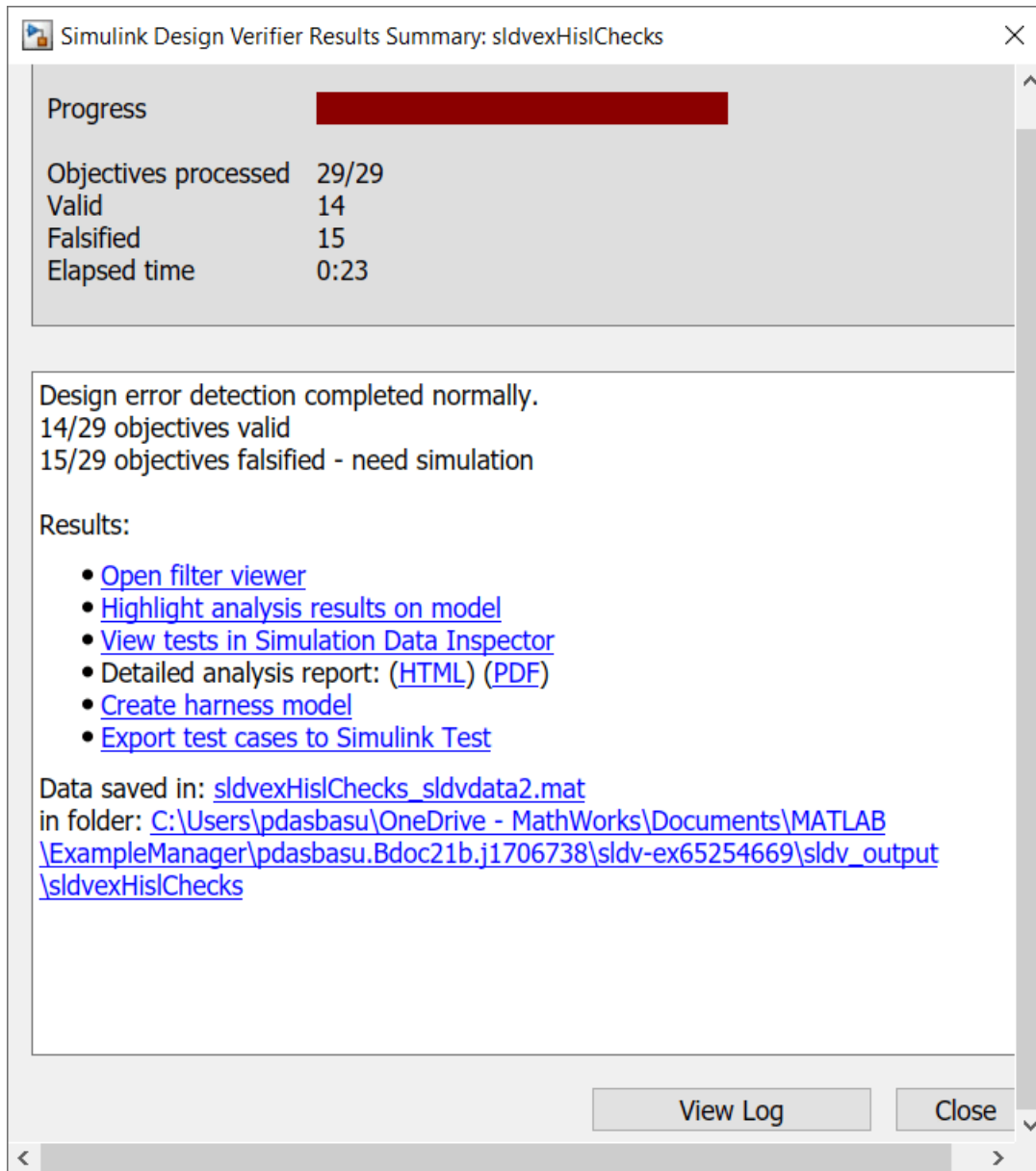
## 2. Perform Design Error Detection Analysis

The model is preconfigured with High-Integrity Systems Modeling checks, **Usage of remainder and reciprocal operations-hisl\_0002**, **Usage of square root operations-hisl\_0003**, **Usage of log and log<sub>10</sub> operations-hisl\_0004**, and **Usage of Reciprocal Square Root blocks-hisl\_0028**. For more information see "Design Verifier Pane: Design Error Detection" on page 15-42.

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

The software analyzes the model for violations of the High-Integrity Systems Modeling guidelines and displays the results in the Results Summary window. The results indicate that 15 out of 29 objectives are falsified.

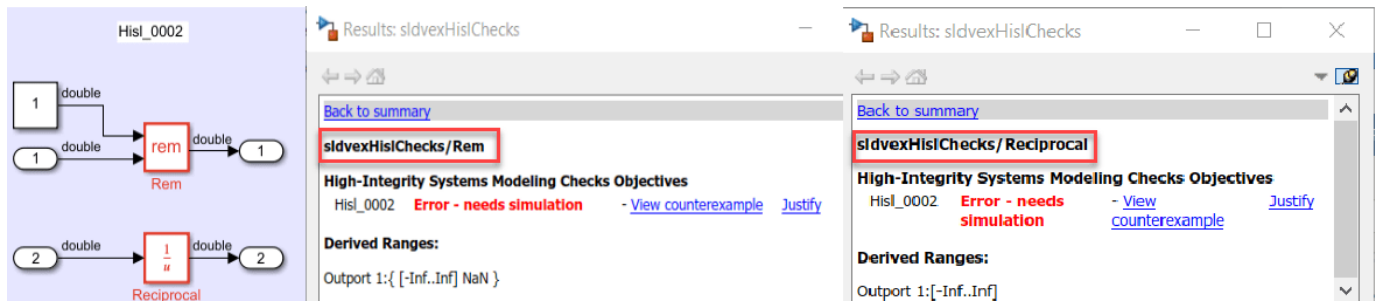




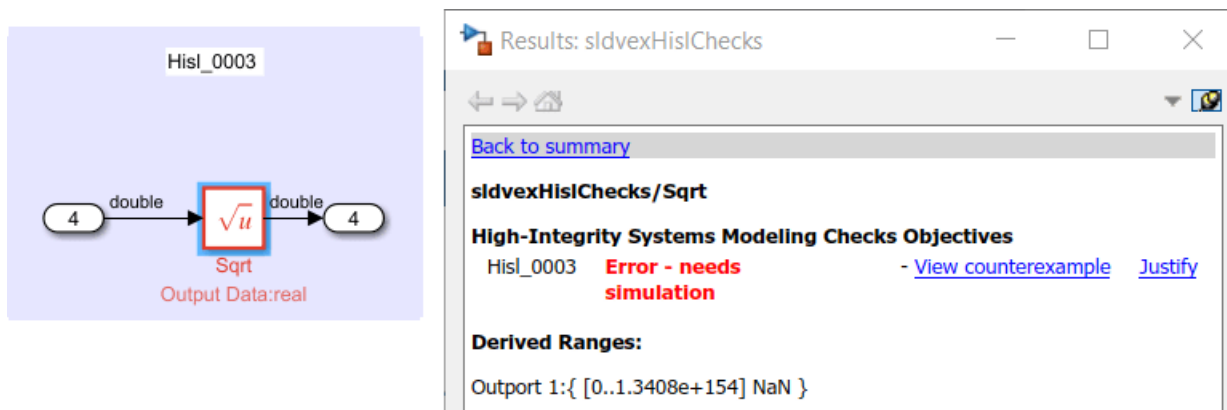
### 3. Review Analysis Results

Click **Highlight analysis results on model**. The blocks that result in violations of High-Integrity Systems Modeling guidelines are highlighted in red.

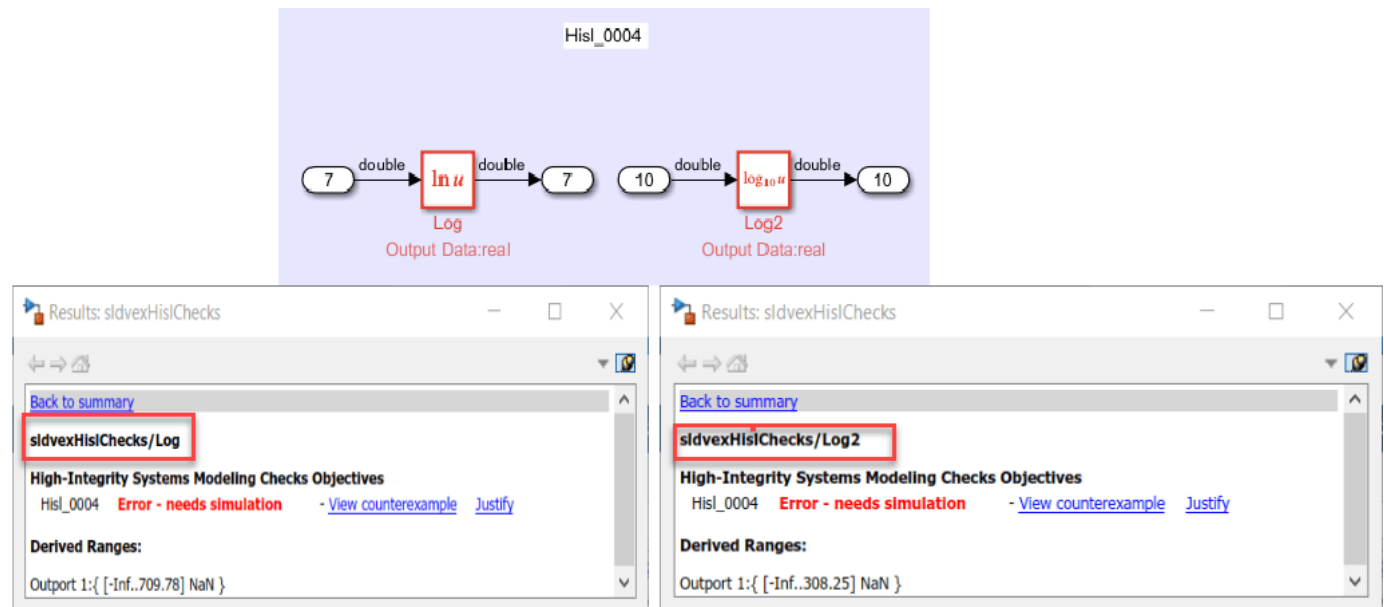
a. Click the **Rem** and **Reciprocal** blocks highlighted in red. The Result Inspector displays the summary of the violation of hisl\_0002 guideline.



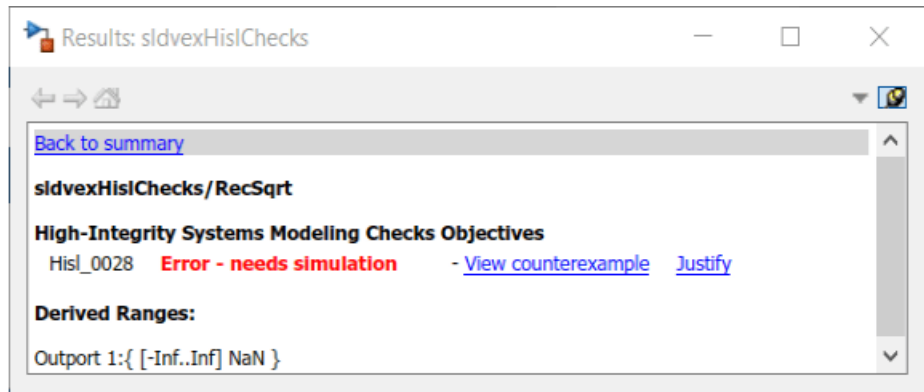
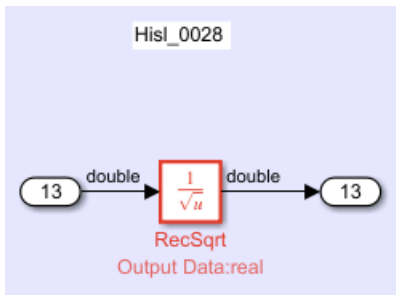
b. Click the **Sqrt** block highlighted in red. The Result Inspector displays the summary of the violation of hisl\_0003 guideline.



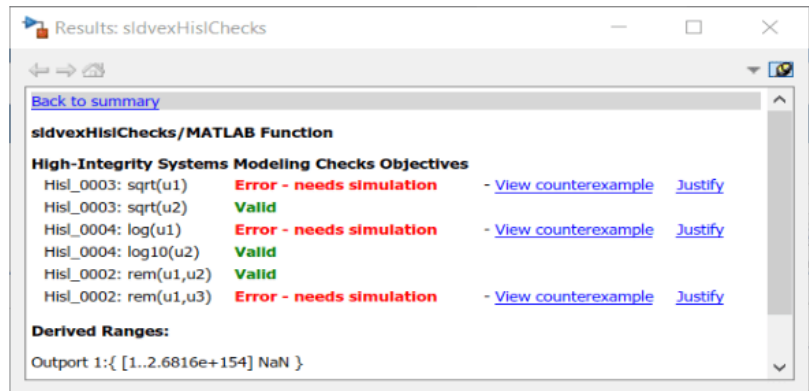
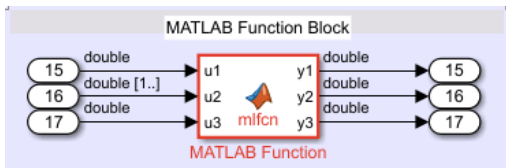
c. Click the **Log** and **Log10** blocks highlighted in red. The Result Inspector displays the summary of the violation of hisl\_0004 guideline.



d. Click the **Reciprocal Square Root** block highlighted in red. The Result Inspector displays the summary of the violation of hisl\_0028 guideline.



e. Click the **MATLAB Function** block highlighted in red. The Result Inspector displays the summary of hisl\_0002, hisl\_0003, and hisl\_0004 checks.



#### 4. View Detailed Analysis Report

To view the detailed analysis report, in the Results Summary window, click **HTML**. The report displays the summary of all occurrences of High-Integrity Systems Modeling violations in the model.

#### 5. Clean Up

To complete this example, close the model.

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

#### See Also

#### More About

- “Simulink Design Verifier Options” on page 15-2
- “Design Verifier Pane: Design Error Detection” on page 15-42

## Filter Objectives by Using Simulink Design Verifier Filter Explorer

Filtering model objects and code expressions from design error detection or test generation analysis allows you to focus on a subset of objects for Simulink Design Verifier analysis. Use filters when you have model objects that take a long time to analyze or when you want to focus on specific objectives for analysis.

You can add one or more filter files by opening the **Configuration Parameters** window, clicking **Design Verifier** and, under **Advanced parameters**, selecting “Ignore objectives based on filter” on page 15-17. Enter your filter files in the **Filter file(s)** parameter. For more information about coverage filter files, see “Creating and Using Coverage Filters” (Simulink Coverage). You can also filter the **Design Verifier** objectives for code-based analysis to align code-based results to model-based results.

After you perform design error detection or test generation analysis, you can justify `unsatisfiable`, `dead logic`, `undecided`, and `falsified` objectives by using the Simulink Design Verifier Filter Explorer. When you edit filters by using Simulink Design Verifier Filter Explorer, you can update the Simulink Design Verifier report and highlight the analysis results on the model without reanalyzing the model. For detailed example on how to filter objectives, see “Exclude and Justify Objectives for Design Error Detection” on page 6-59.

### Use the Simulink Design Verifier Filter Explorer to Edit Filter Files

After analyzing your model, you can use Simulink Design Verifier Filter Explorer to justify the `falsified`, `unsatisfiable`, `undecided`, and `dead logic` objectives and update the filter files.

You can open the filter explorer from the Results Summary window or from the Results Inspector window.

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

```

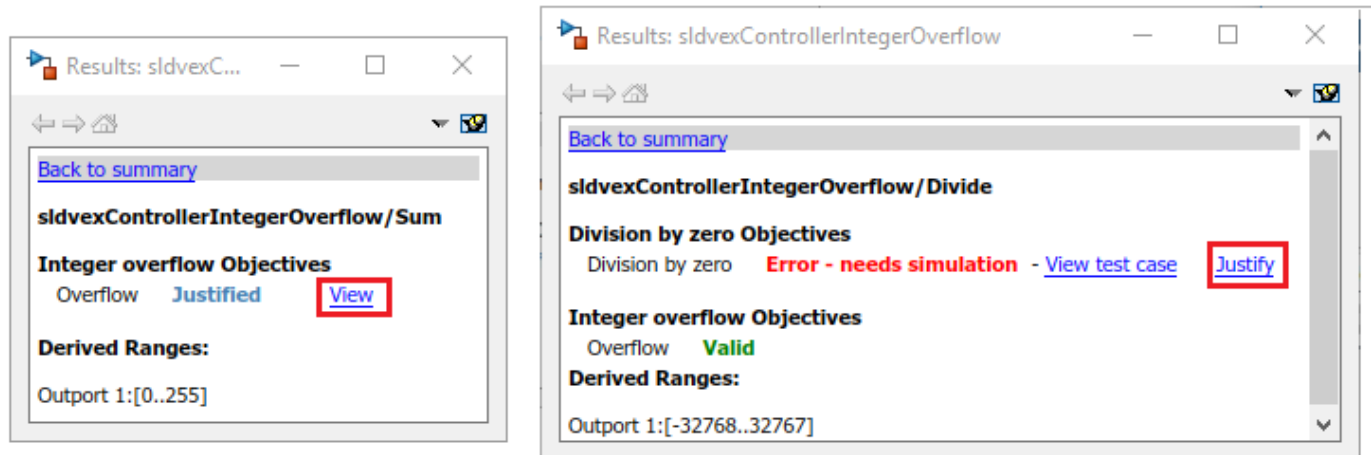
Design error detection completed normally.
3/6 objectives valid
1/6 objective falsified
1/6 objective excluded
1/6 objective justified

Results:


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

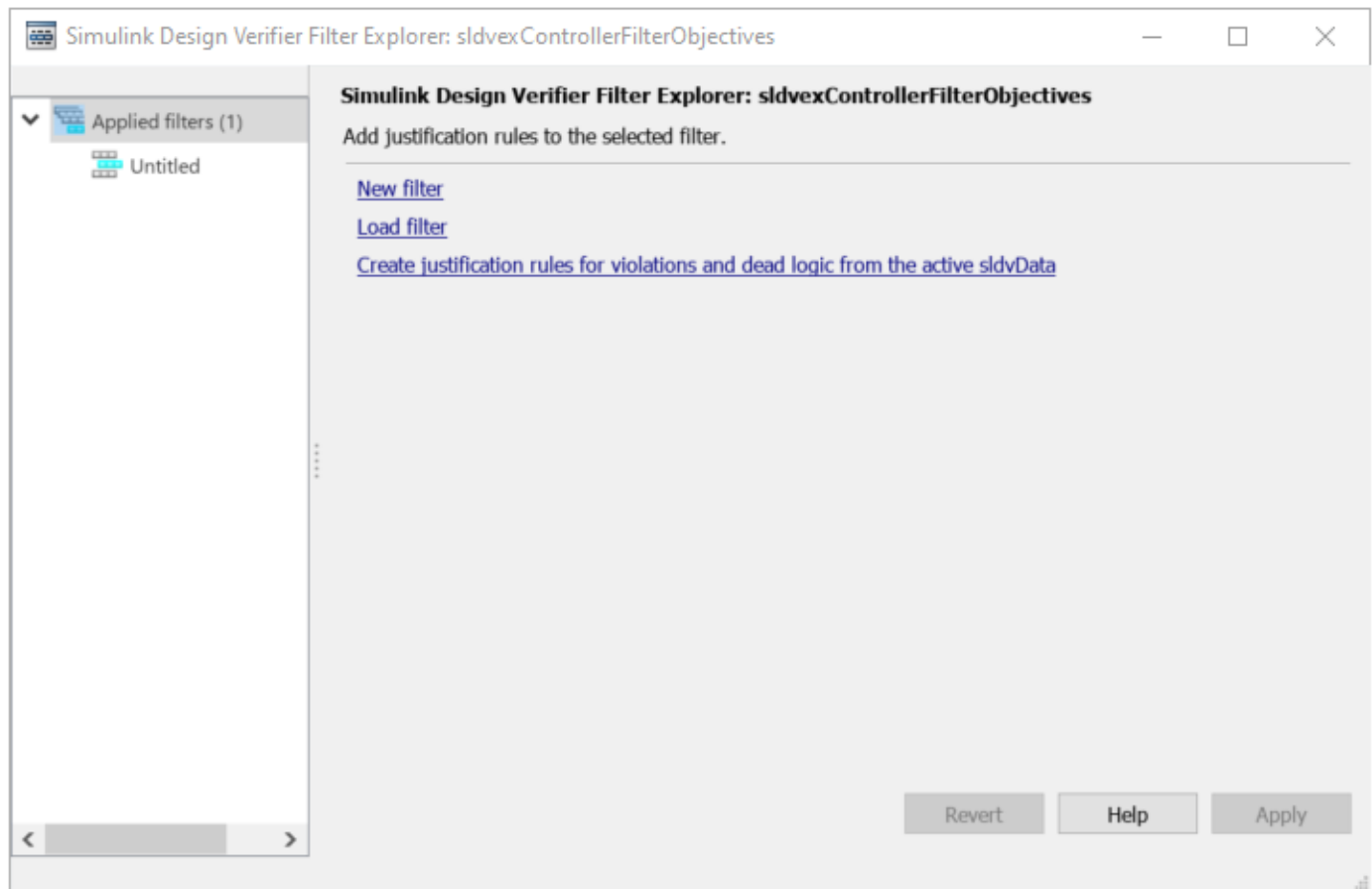
```

- In the Results Inspector window,
  - To see the filter rule for a justified objective, click **View**.
  - To justify an objective, click **Justify**.



In the Simulink Design Verifier Filter Explorer, you can:

- Create, load, edit, or save filter files.
- Create a filter file to justify all the Unsatisfiable, Falsified and Dead Logic objectives from the active sldvData.
- Navigate to the model to inspect the model objects associated with a filter rule.
- Add rationale description about why the objective or model object or code expression is excluded or justified.



Design error detection completed normally.

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

Results:

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

| Task  | Action  |
|---|---|
| Navigate to a model object associated with a rule.<br><hr/> <b>Note</b> This step is valid only for model objective analysis. | <ol style="list-style-type: none"> <li><b>1</b> Select the rule.</li> <li><b>2</b> Click <b>View in model</b>. The model object is highlighted in blue.</li> </ol>  |
| Delete a rule.  | <ol style="list-style-type: none"> <li><b>1</b> Select the rule.</li> <li><b>2</b> Click <b>Remove rule</b>.</li> </ol>   |
| Save the current rules to a file.   | <ol style="list-style-type: none"> <li><b>1</b> Click <b>Apply</b>.</li> <li><b>2</b> Specify a file name and folder for the filter file and click <b>Save</b>.</li> </ol>  |
| Rename a filter file  | <ol style="list-style-type: none"> <li><b>1</b> Click <b>Save as</b>.</li> <li><b>2</b> Specify a file name and folder for the filter file and click <b>Save</b>.</li> </ol>  |
| Load an existing filter file.   | <ol style="list-style-type: none"> <li><b>1</b> Click <b>Load filter</b>.</li> <li><b>2</b> Navigate to the filter file and click <b>Open</b>.</li> </ol>   |
| Highlight the model and update the current analysis report with the current filter files.                                     | <ol style="list-style-type: none"> <li><b>1</b> <b>Apply</b> or <b>Revert</b> any changes you have made.<br/><br/>The model is highlighted with the updated filter rules.</li> <li><b>2</b> In the Results Summary window or in the Results inspector window, click <b>HTML</b> or <b>PDF</b>.</li> </ol> |
| Create an empty filter file.  | Click <b>New filter</b><br>.  |
| Remove a filter from Filter Explorer.   | Right-click the corresponding node under <b>Applied filters</b> and select <b>Remove</b><br>.   |
| Create a filter file to justify all Unsatisfiable, Falsified, and Dead Logic objectives in the active sldvData                | <ol style="list-style-type: none"> <li><b>1</b> Click <b>Create justification rules for violations and dead logic from the active sldvData</b></li> <li><b>2</b> Click <b>Save as</b></li> <li><b>3</b> Specify a file name and folder for the filter file and click <b>Save</b></li> </ol>               |

## Limitations

Simulink Design Verifier does not support filtering objectives associated with property proving analysis.

## **See Also**

### **More About**

- “Design Verifier Pane” on page 15-9
- “Create, Edit, and View Coverage Filter Rules” (Simulink Coverage)
- “Review Results” on page 13-35



## Detect Integer Overflow Errors

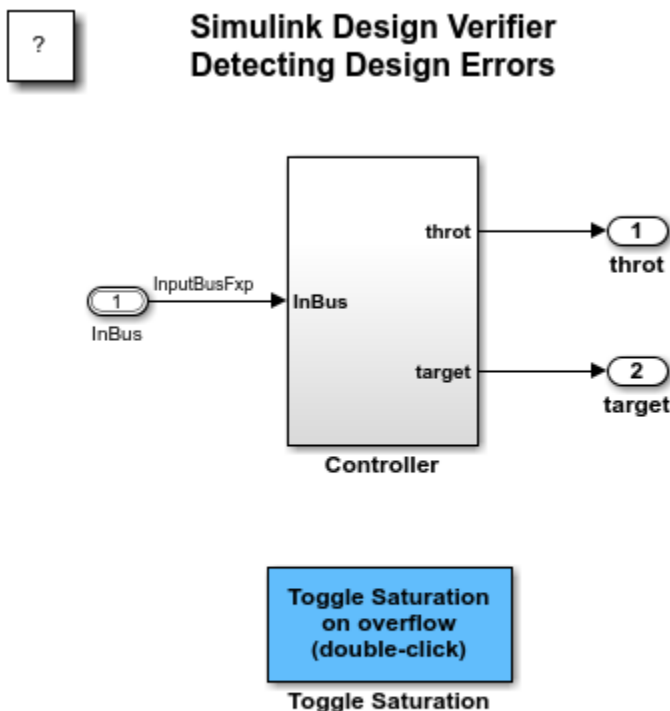
This example shows how to detect integer overflow errors in a model by using design error detection analysis. Simulink® Design Verifier™ identifies the model constructs that may result in integer overflows and then either proves that the integer overflow cannot occur during simulation or generates test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

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

### Step 1: Open the Model

At the command prompt, enter:

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



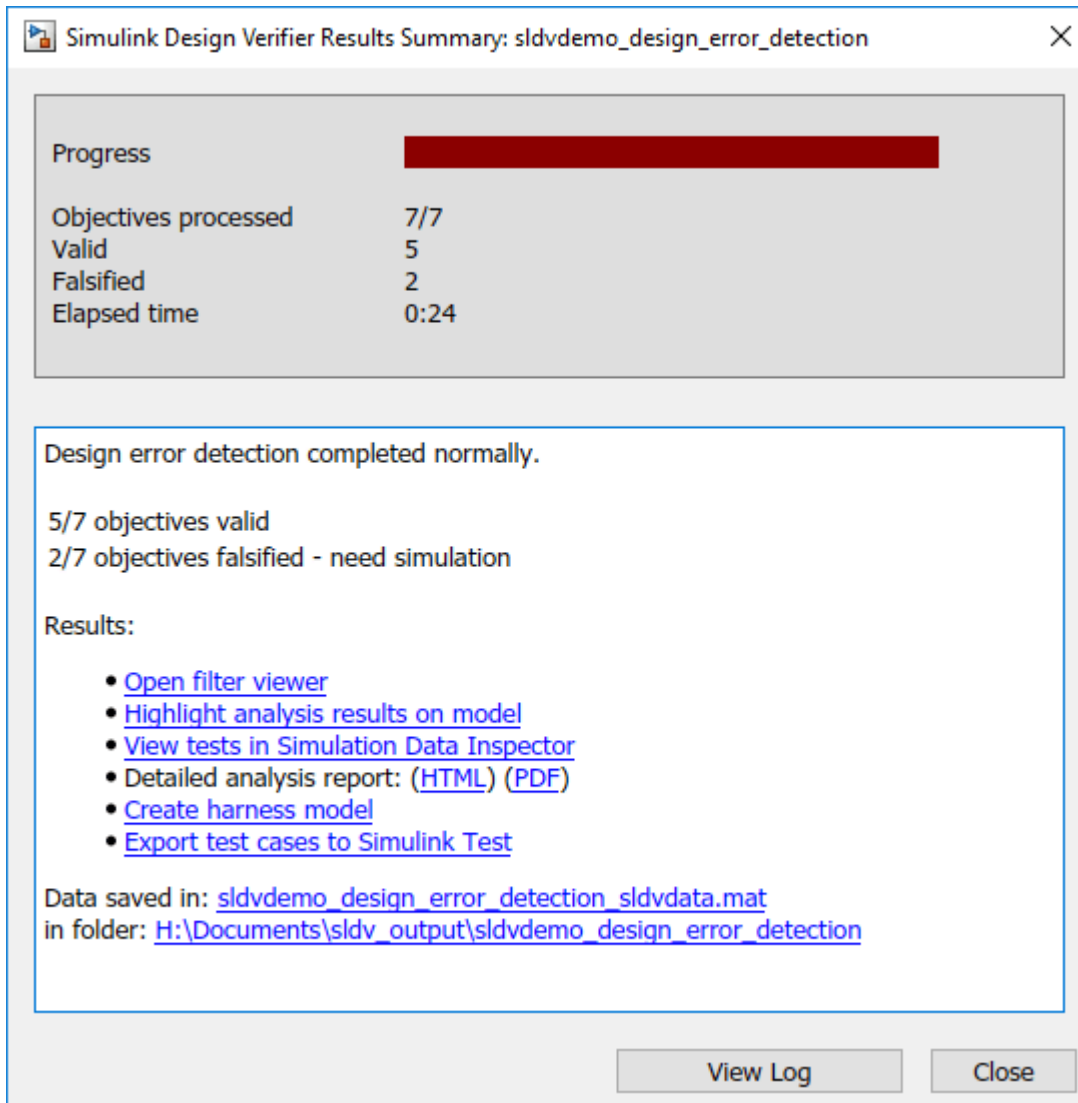
Copyright 2006-2023 The MathWorks, Inc.

### Step 2: Perform Design Error Detection Analysis

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

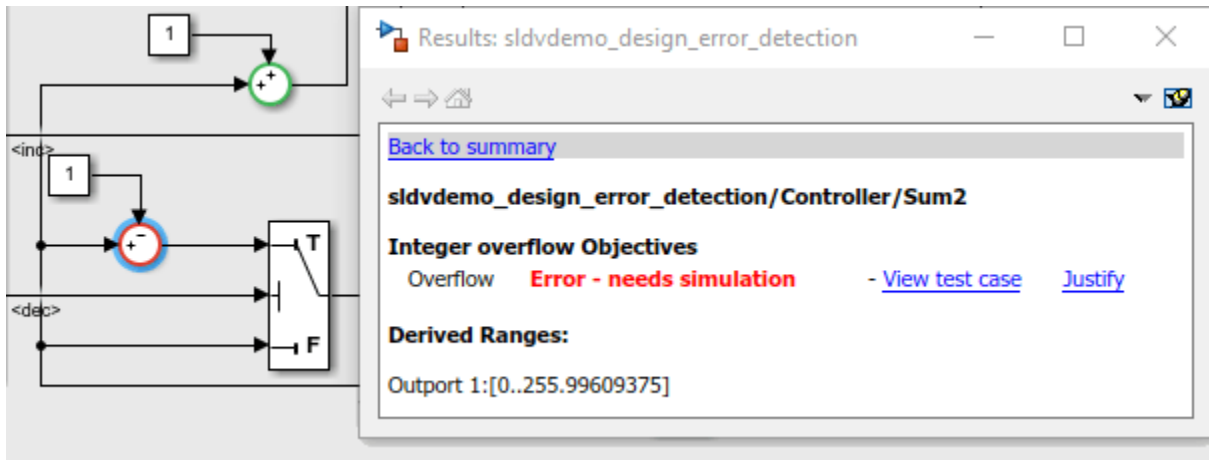
On the **Design Verifier** tab, click **Detect Design Errors**.

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



### Step 3: Review Analysis Results

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



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

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

#### Step 4: Fix the Integer Overflow Error

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

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

#### Related Topics

- “Detect Integer Overflow and Division-by-Zero Errors” on page 6-19
- “Understand the Analysis Results” on page 6-4

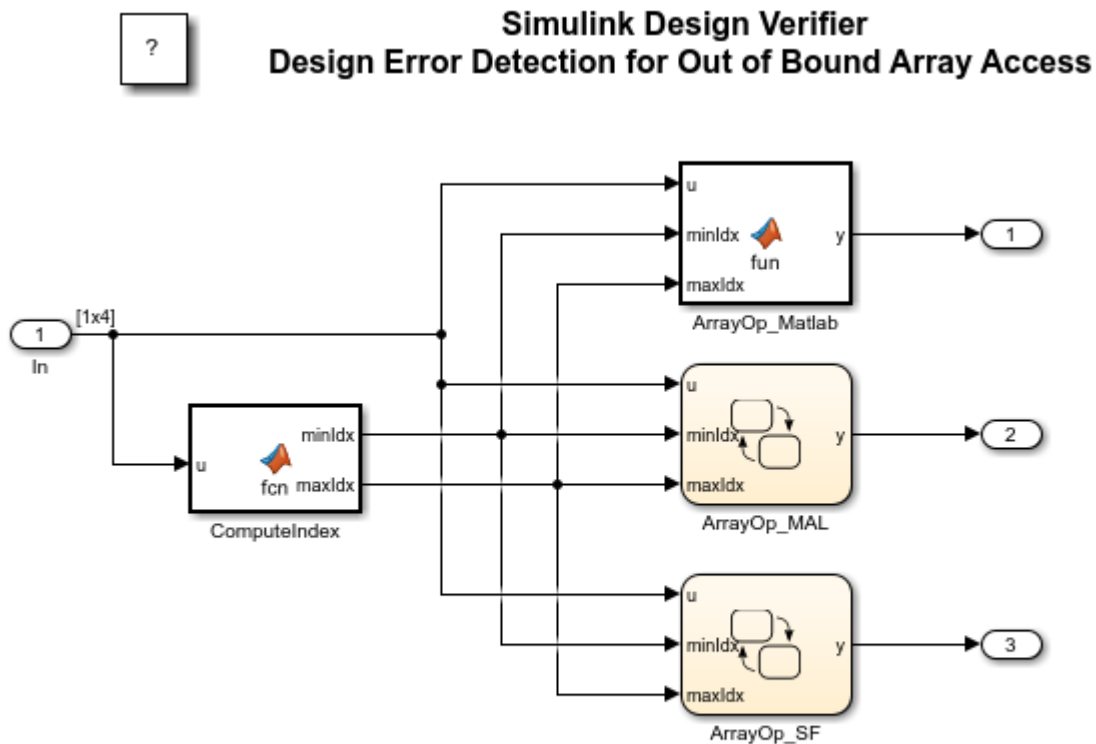
## Detect Out of Bound Array Access Example Model

This example shows how to detect out of bound array access errors and review the analysis results. In the `sldvdemo_array_bounds` example model, the ComputeIndex MATLAB Function block uses the input signal values to determine range of indices with minimum `minIdx` and maximum `maxIdx`. The `ArrayOp_Matlab`, `ArrayOp_MAL`, and `ArrayOp_SF` blocks use the set of integer indices between `minIdx` and `maxIdx` to access array elements and perform array operations.

### Step 1: Open the Model

At the command prompt, enter:

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

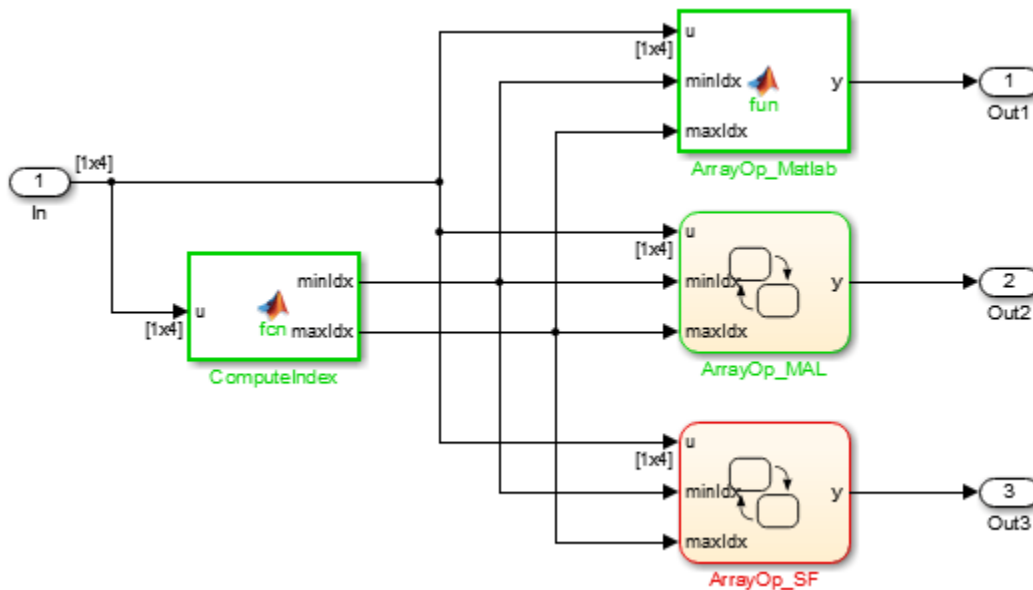


Copyright 2010-2019 The MathWorks, Inc.

### Step 2: Perform Design Error Detection Analysis

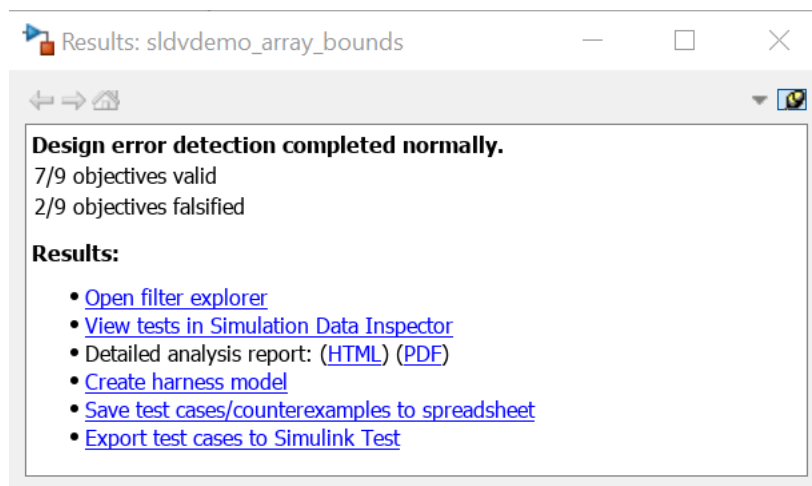
The analysis options in the model are preconfigured for out of bound array access error detection. To view these options, in the Simulink Editor, double-click the **View Options** button.

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



### Step 3: Review Analysis Results

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



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

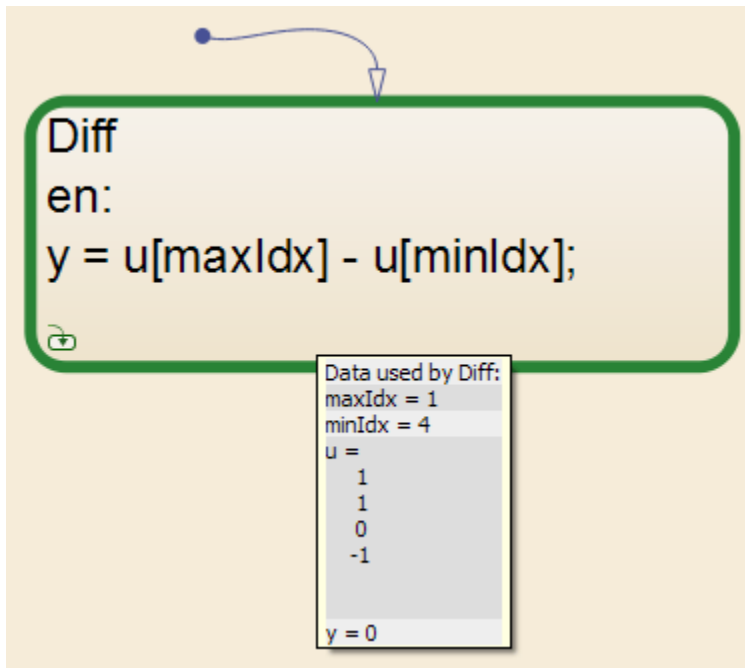
### Step 4: Create Harness and Simulate Test Cases

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

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

Attempted to access index 4 of u with smaller dimension sizes. The valid index range is 0 to 3. This error will stop the simulation. State 'Diff' in Chart 'sldvdemo\_array\_bounds\_harness/Test Unit (copied from sldvdemo\_array\_bounds)/ArrayOp\_SF':  $y = u[\text{maxIdx}] - u[\text{minIdx}]$ ;

Keep the Stateflow® Debugger open at this breakpoint. In the sldvdemo\_array\_bounds\_harness model, hold your cursor over the Diff state to see the data values at this simulation breakpoint.



Using Test Case 2 input signal values, the ComputeIndex MATLAB Function block determines the range of array indices to be 1:4. One-based indexing is consistent with MATLAB syntax, so these indices are valid for the ArrayOp\_Matlab MATLAB Function block and the ArrayOp\_MAL Stateflow® chart.

The ArrayOp\_SF Stateflow® chart uses C as the action language, which does not support one-based indexing. Thus, 1:4 is not a valid index range for array access in the chart. The valid index range for array access in the chart is 0:3, as reported by the error message. When either maxIdx or minIdx evaluates to 4, an out of bound array access error occurs in the ArrayOp\_SF Chart block. For more information on zero-based indexing support, see "Differences Between MATLAB and C as Action Language Syntax" (Stateflow).

## Detect Design Errors in C/C++ Custom Code

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

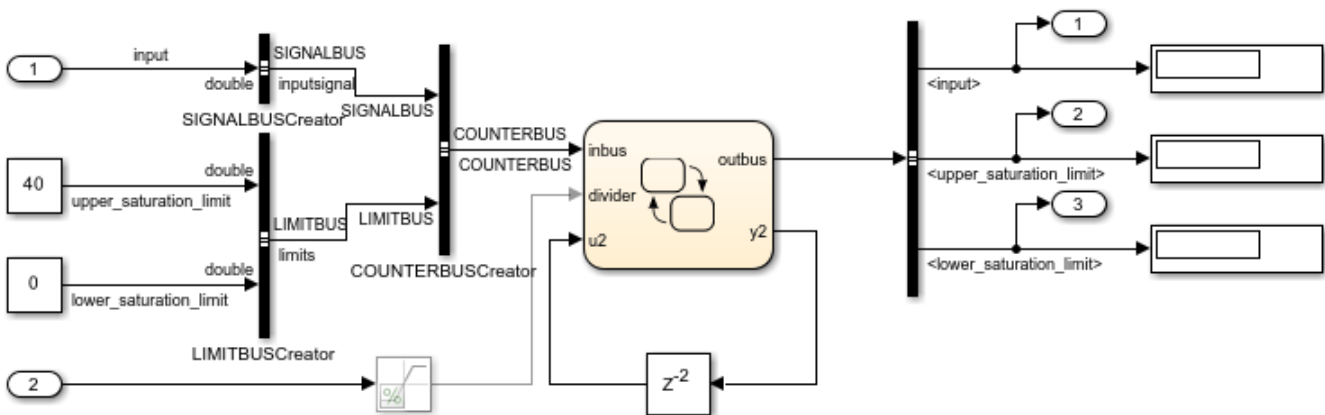
This example shows how to detect division by zero errors in a model that consists of C/C++ code in a Stateflow® chart.

### Step 1: Open the Model

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

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

### Simulink Design Verifier Detect Design Errors in C/C++ Custom Code



This model contains a stateflow chart which is calling C custom-code with buses input and output.

**Open**  
Custom code sources  
(double-click)

Open Source Files

**View**  
Custom code settings  
(double-click)

View Custom code settings

**Run**  
(double-click)

**View Options**  
(double-click)

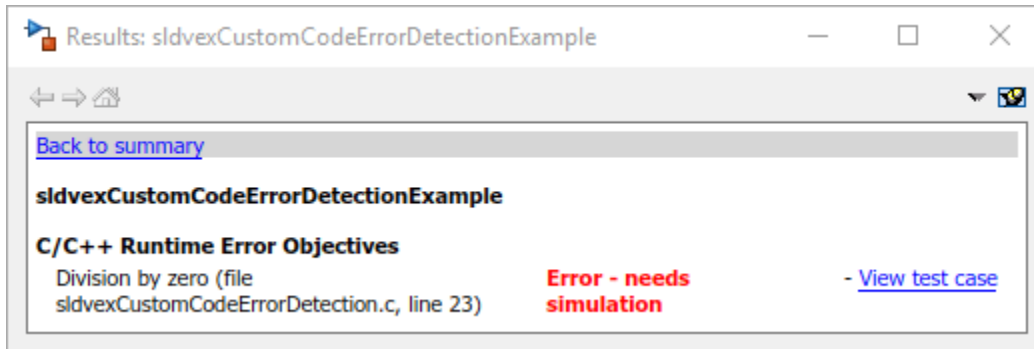
Copyright 2019 The MathWorks, Inc.

### Step 2: Perform Design Error Detection Analysis

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

### Step 3: Review the Analysis Results

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



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

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

## Objectives Falsified - Needs Simulation

| #  | Type                | Model Item  | Description   | Analysis Time (sec) | Test Case |
|----|---------------------|---|---|---------------------|-----------|
| 20 | C/C++ Runtime Error | <a href="#">sldvexCustomCodeErrorDetectionExample</a> | Division by zero (file sldvexCustomCodeErrorDetection.c, line 23) | 21                  | 1         |

### Step 4: Fix Design Errors

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

### Step 5: Clean Up

To complete the example, close the model.

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

### Related Topics

- “Design Error Detection Objectives Status” on page 13-43
- “Design Verifier Pane: Design Error Detection” on page 15-42



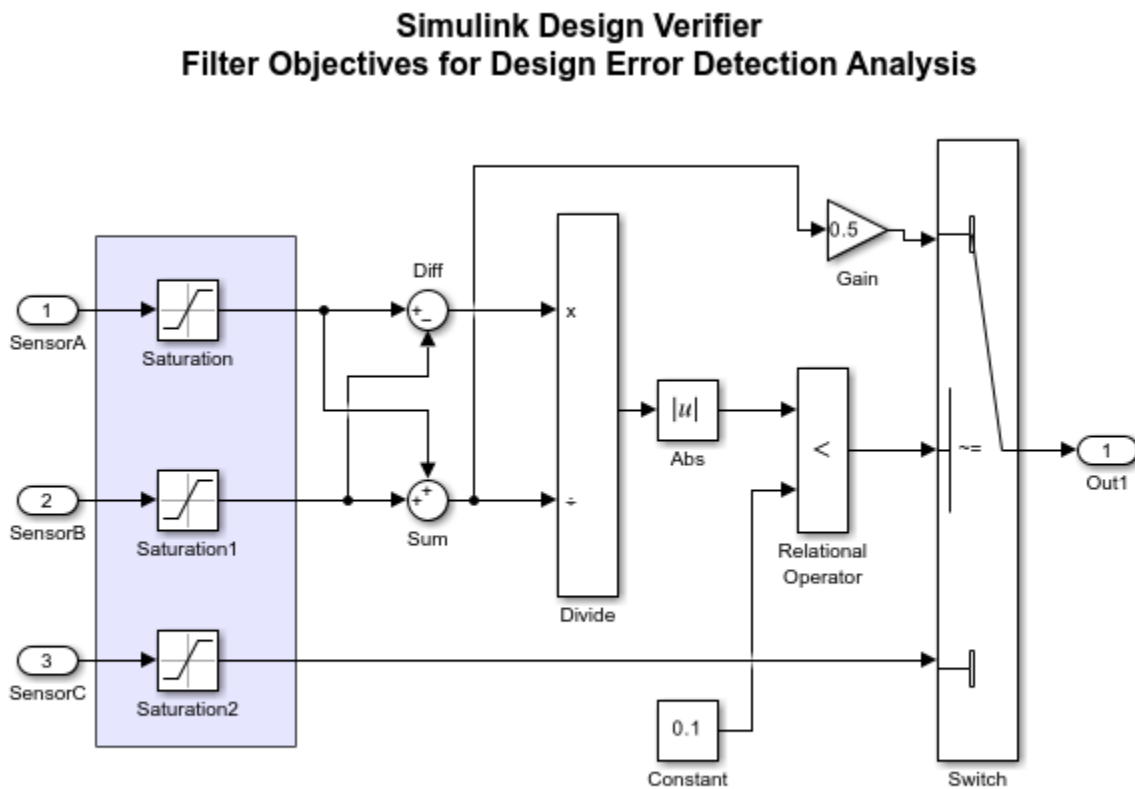
## Exclude and Justify Objectives for Design Error Detection

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

### Step 1: Open the Model

The example model `sldvexControllerFilterObjectives` is a controller model that operates according to the controller algorithm.

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

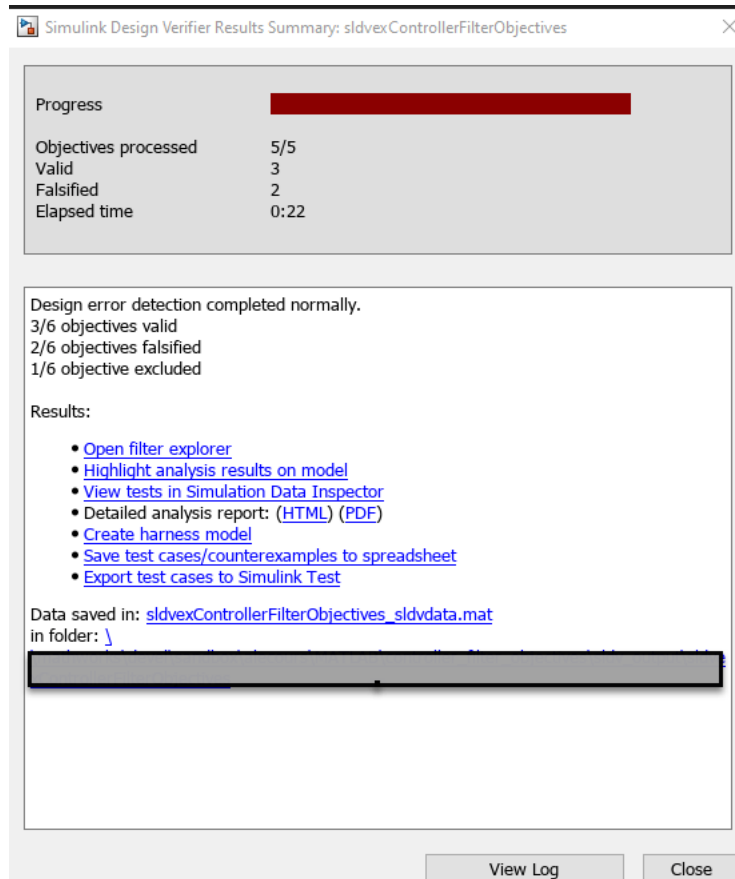


Copyright 2019 The MathWorks, Inc.

### Step 2: Exclude a Model Object from Analysis

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

On the **Apps** tab, under **Model Verification, Validation, and Test**, click **Design Verifier**. Then, click **Detect Design Errors**. After the analysis completes, the Results Summary window reports that 5 objectives were processed, out of which, 3 were valid and 2 were falsified. The summary shows that 1 objective was excluded from analysis.



### Step 3: Open the Analysis Filter Viewer

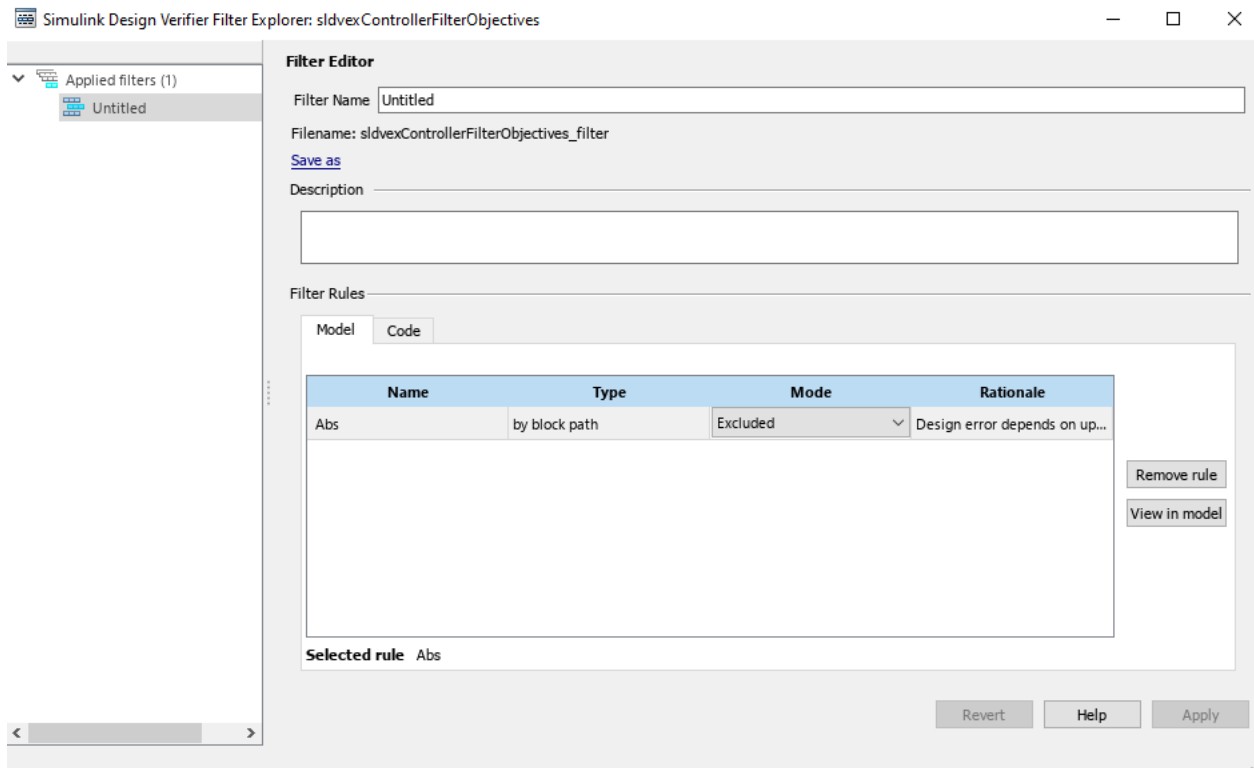
On the Results Summary window, click **Open filter viewer**. The **Analysis Filter** viewer opens that displays the name, type, and rationale for the excluded

### Step 3: Open the Analysis Filter Explorer

On the Results Summary window, click **Open filter explorer**. The **Analysis Filter** explorer opens that displays the name, type, and rationale for the excluded

### Step 3: Open the Simulink Design Verifier Filter Explorer

On the Results Summary window, click **Open filter explorer**. The Filter Explorer opens.



Click on the applied filter's node to view the names, type, and rationale for the excluded objectives specified in the coverage filter file.

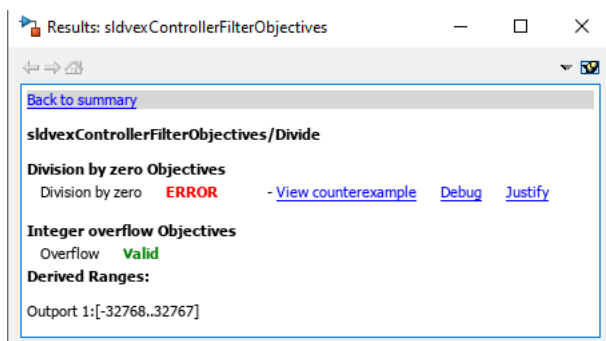
#### Step 4: Justify Objectives

(a) Close the Filter Explorer.

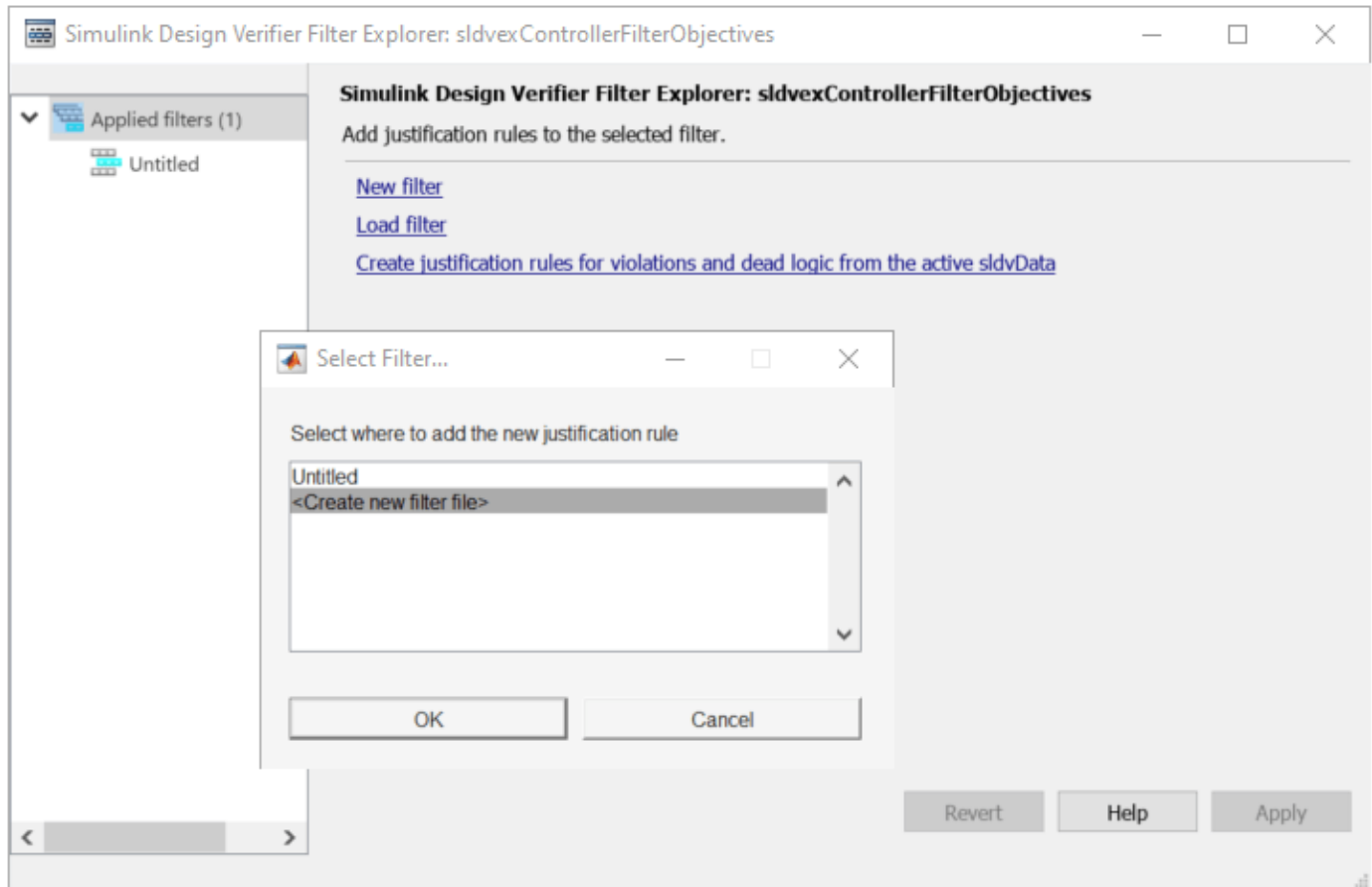
(b) On the Results Summary window, click **Highlight analysis results on model**. The model is highlighted with the analysis results. The excluded model objects are highlighted in steel blue and the model objects that result in errors are highlighted in red.

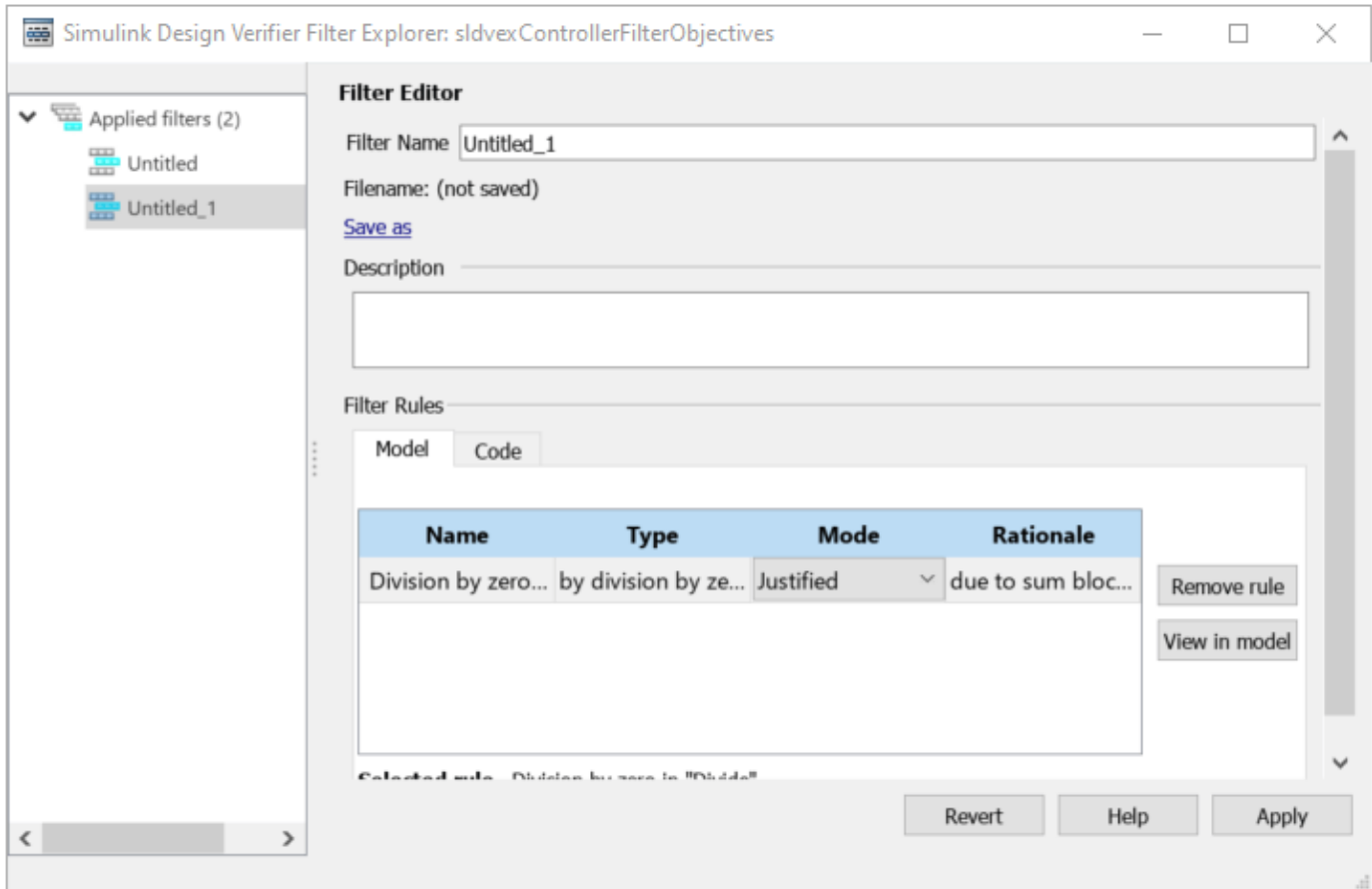
(b) To view the excluded objectives, click Abs block and click **View**. The **Analysis Filter** viewer opens. The **Analysis Filter** explorer opens. The Filter Explorer opens and displays the relevant filter rule.

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



(d) To justify the division by zero objective, click **Justify**. The **Analysis Filter** viewer is updated with a rule that justifies this objective. Optionally, you can update the **Mode** or **Rationale** for the objectives. (d) To justify the division by zero objective, click **Justify**. The **Analysis Filter** explorer is updated with a rule that justifies this objective. Optionally, you can update the **Mode** or **Rationale** for the objectives. (d) To justify the division by zero objective, click on the **Applied filters** node in Filter Explorer and click **Justify** in the **Results Inspector** window. The Filter Explorer opens and queries about where to add the justification rule. You may choose to add it to the existing filter file or create a new filter file. Create a new file.





### Step 5: Apply the Filter File and View Results

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

To save the updated filter file, in the **Analysis Filter** viewer, click **Save Filter**, enter the name of file, and click **OK**. On the **Analysis Filter** explorer, click **Apply**. The model is highlighted with the updated filter. The Divide block is highlighted in green because all the objectives of the block are valid.

To save the updated filter file, in the **Analysis Filter** explorer, click **Save Filter**, enter the name of file, and click **OK**. On the Filter Explorer, click **Apply**. You will be prompted to provide a file name for the new filter. Enter the desired name and click **Save**. The model is highlighted with the applied filters. The Divide block is highlighted in green because all the objectives of the block are valid or justified.

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

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

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

## Objectives Excluded

| #  | Type             | Model Item          | Description | Rationale                               |
|----|------------------|---------------------|-------------|---|
| 11 | Integer overflow | <a href="#">Abs</a> | Overflow    | Design error depends on upstream blocks |

## Objectives Justified

| # | Type             | Model Item             | Description      | Rationale                         |
|---|------------------|------------------------|------------------|-----------------------------------|
| 8 | Division by zero | <a href="#">Divide</a> | Division by zero | due to sum block integer overflow |

### Related Topics

- “Filter Objectives by Using Simulink Design Verifier Filter Explorer” on page 6-46
- “Detect Integer Overflow and Division-by-Zero Errors” on page 6-19

## Detect Integer Overflow in a Model with Complex Inputs

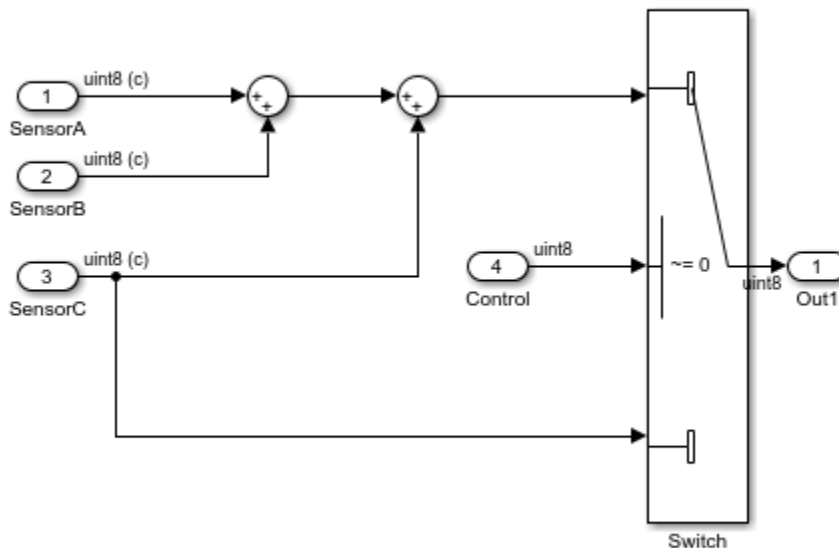
This example shows how to detect integer overflow errors in a model that consists of complex type inputs.

### Step 1: Open the Model

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

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

### Simulink Design Verifier Detect Integer Overflow Errors in Model with Complex Inputs

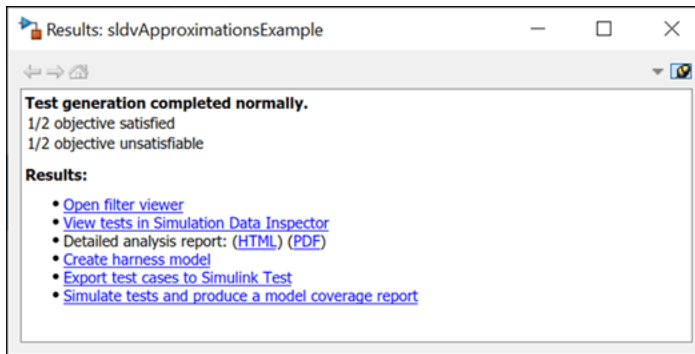


Copyright 2019 The MathWorks, Inc.

### Step2: Perform Design Error Detection Analysis

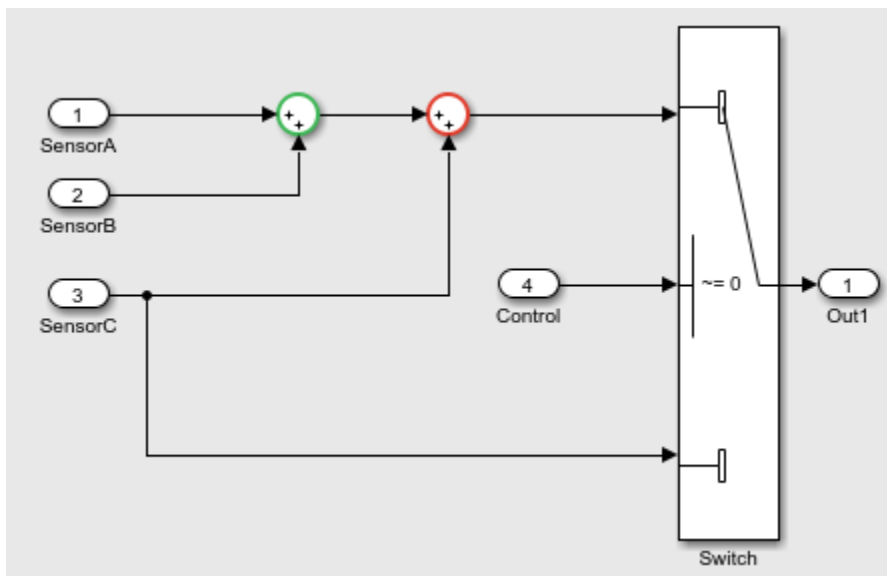
On the **Apps** tab, in the **Model Verification, Validation, and Test** group, select **Design Verifier**.

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



### Step 3: Review Analysis Results

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



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



## Objectives Excluded

| #  | Type             | Model Item          | Description | Rationale                               |
|----|------------------|---------------------|-------------|---|
| 11 | Integer overflow | <a href="#">Abs</a> | Overflow    | Design error depends on upstream blocks |

## Objectives Justified

| # | Type             | Model Item             | Description      | Rationale                         |
|---|------------------|------------------------|------------------|-----------------------------------|
| 8 | Division by zero | <a href="#">Divide</a> | Division by zero | due to sum block integer overflow |

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

|             |        |
|-------------|--------|
| <b>Time</b> | 0      |
| <b>Step</b> | 1      |
| SensorA     | 16+93i |
| SensorB     | 26+78i |
| SensorC     | 94+93i |
| Control     | 1      |

### See also

- “Detect Integer Overflow Errors” on page 6-51
- “Understand the Analysis Results” on page 6-4

## Debug Integer Overflow Design Error Detection Using Model Slicer

This example shows how to use Model Slicer to debug integer overflow design errors in a Simulink® model.

### Prerequisites

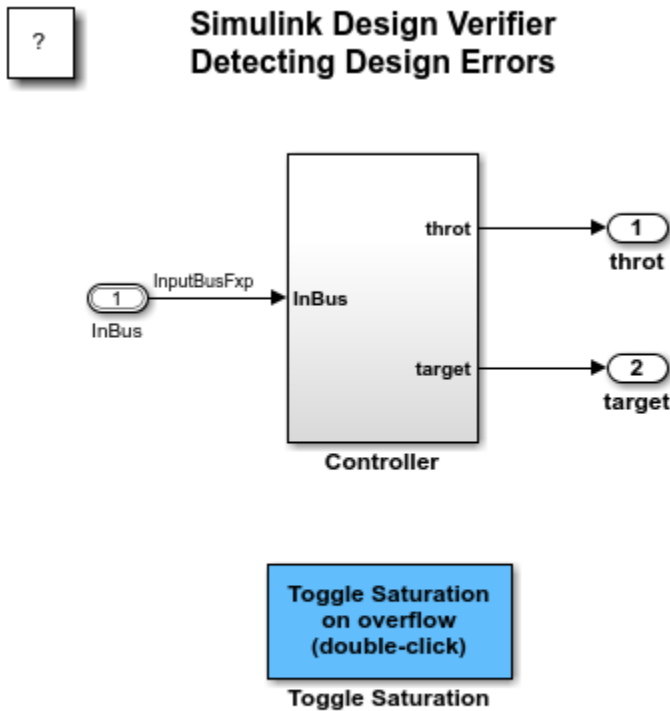
This example uses the following products to demonstrate debugging the Design Error Detection violations:

- Simulink Design Verifier™
- Simulink Check™ (Model Slicer)

### Example

1. Open model `sldvdemo_design_error_detection`.

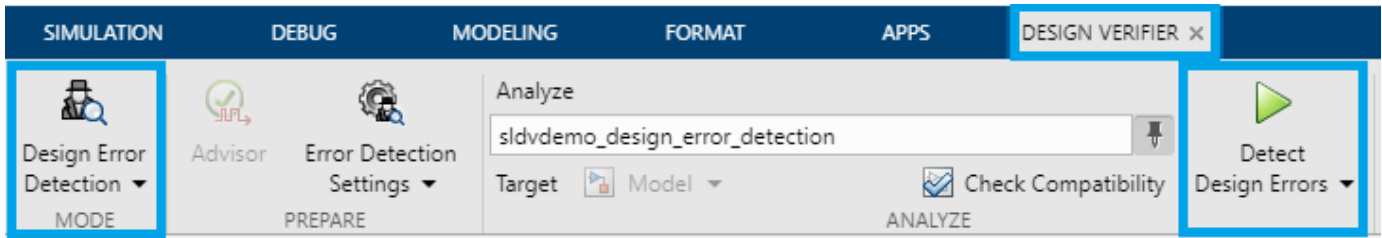
```
open_system('sldvdemo_design_error_detection');
```



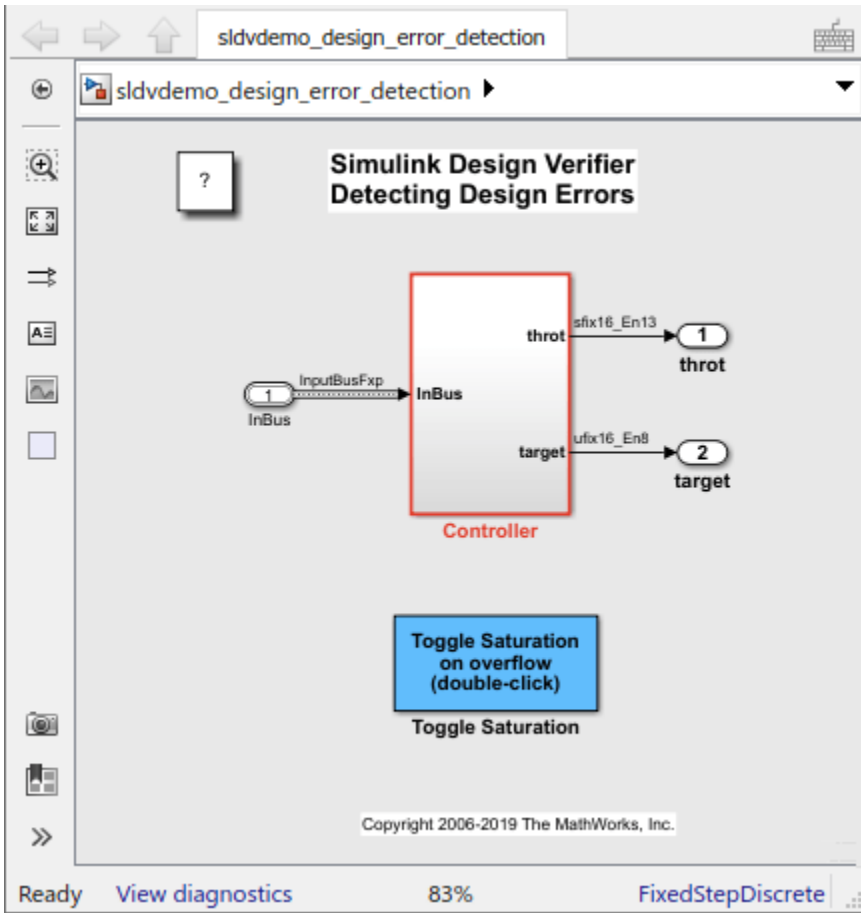
Copyright 2006-2023 The MathWorks, Inc.

2. Open **Simulink Design Verifier** by clicking on **Apps > Design Verifier**.

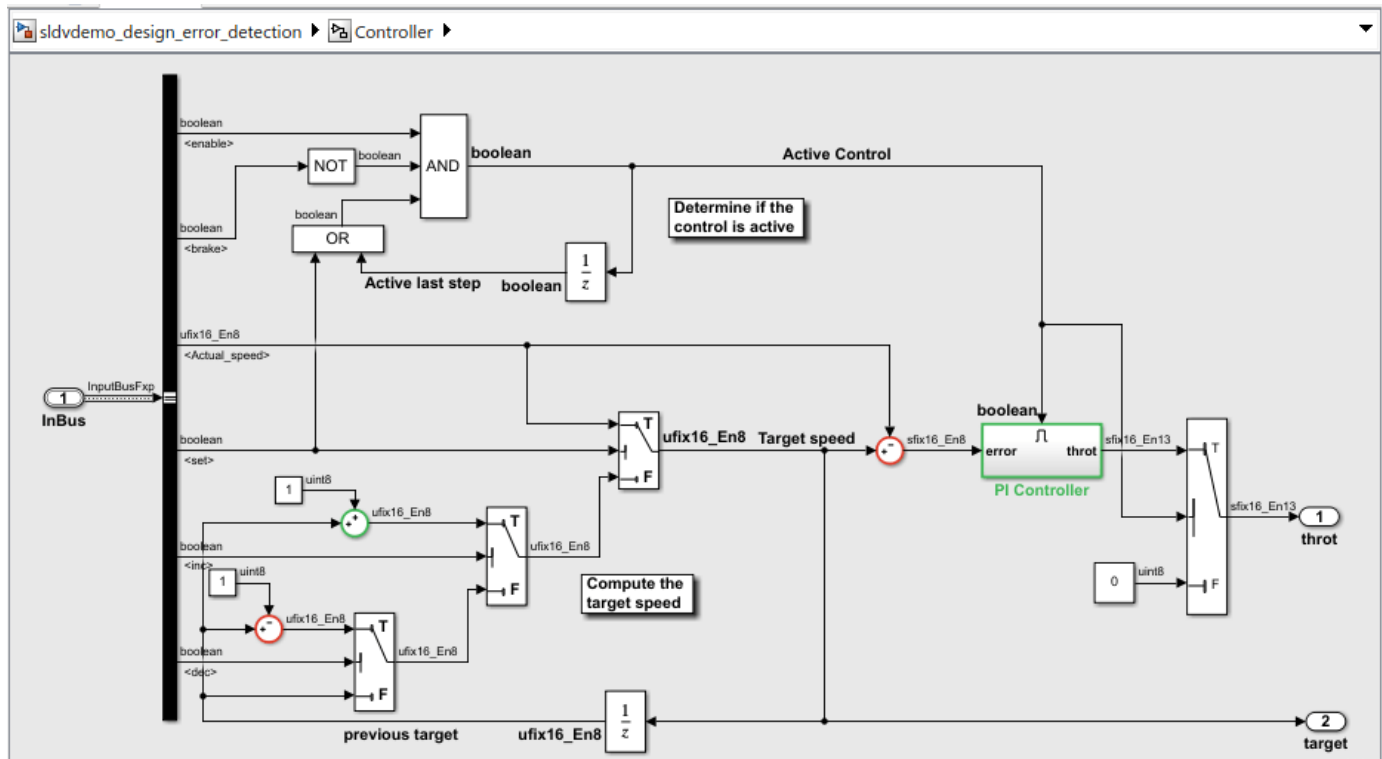
3. In the Design Verifier tab, click **Detect Design Errors**. Simulink Design Verifier analyzes the model and displays the results in **Results Summary** window.



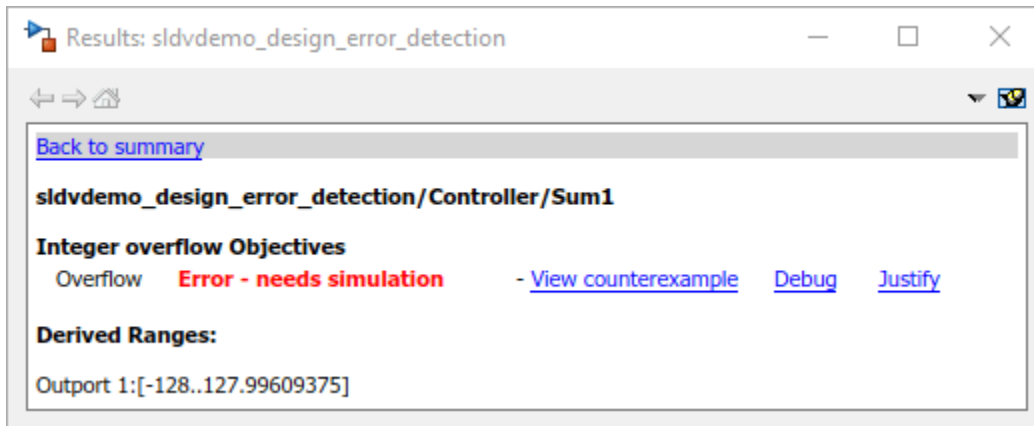
The model highlights the subsystem where the failed objectives are located.



4. Open Controller subsystem and select either of the blocks that are highlighted in red.



5. In the Results window, click **Debug** to debug the violation using Model Slicer. Alternatively, in the Design Verifier tab, click **Review Results > Debug using Slicer** to debug the violation using Model Slicer.

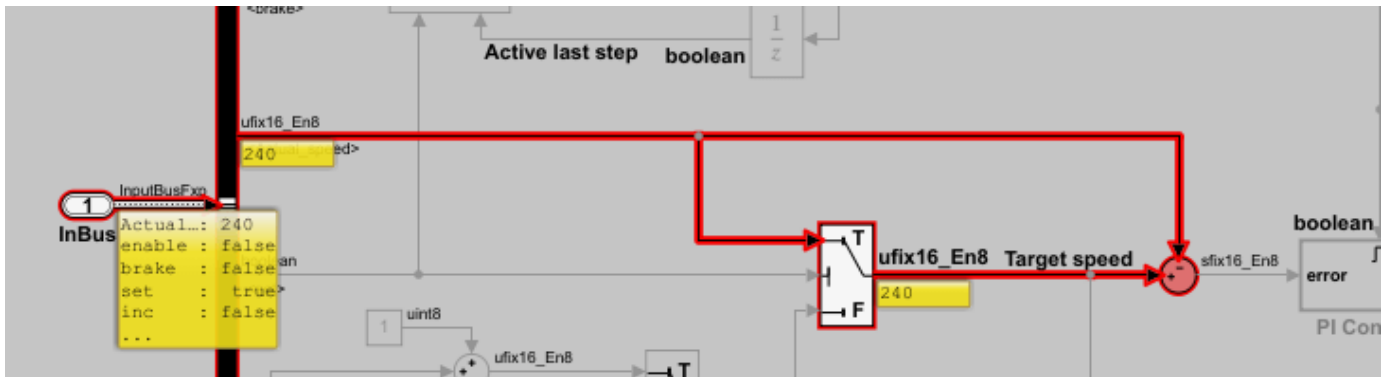


On clicking either of the entry points for debugging, the following setup is done on the model:

- The selected block with a failed objective is added as a starting point for Model Slicer.
- The model is highlighted with the slice responsible for the failing objective.
- The design model is simulated and paused at the time of violation.

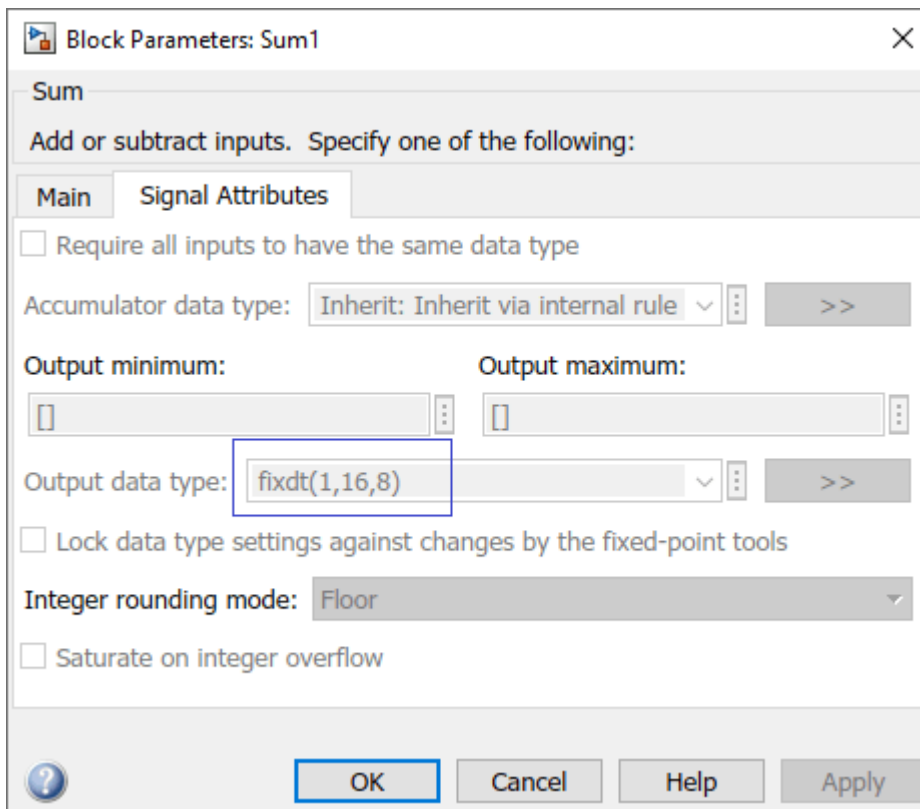
6. Debug and analyze the model by inspecting the port labels.

**Tip:** Click on the output signal line of the Sum block to enable the port value label for the block.



You can observe that the sum of the input variables should result in a non-zero number.

7. Investigate the input and output data types of the sum block.



Here, the datatype conversion results in the integer overflow. The datatype for inputs is `ufix16_En8`, which have a maximum value of 255.9961, whereas the datatype for output block is `sfix16_En8`, which has a maximum of 127.9961. In the counterexample the value is between these two values. The overflow happens when the sum block (without saturation) first casts the input values down to its output type and then does the arithmetic operation.

### Verification

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

### **Additional Capabilities**

You can use the workflow demonstrated in this example to debug the other Design Error Detection violations using Model Slicer. Following are the design errors supported:

- Division by zero
- Integer Overflow
- Non-Finite and NaN (Not a Number) floating-point values
- Specified minimum and maximum value violations
- Datastore access violations
- Specified block input range violations

## Analyzing the Results for a Dead Logic Analysis

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

### Workflow

The `sldvexCommonCausesOfDeadLogic` model demonstrates some of the common patterns that often lead to dead logic in a model. The six subsystems in the model represent a different pattern. These subsystems are:

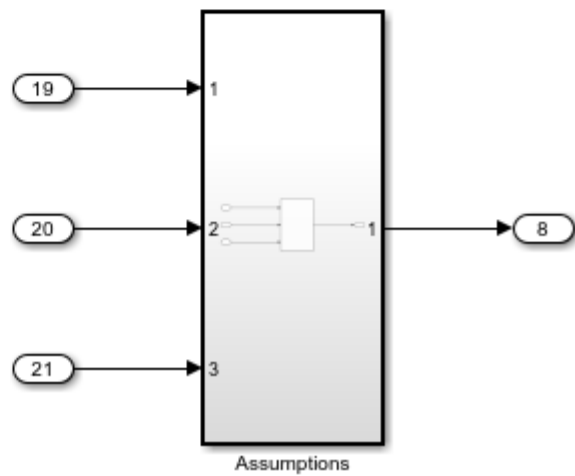
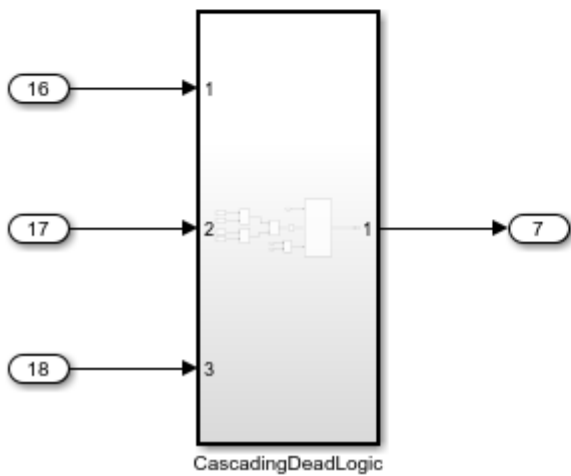
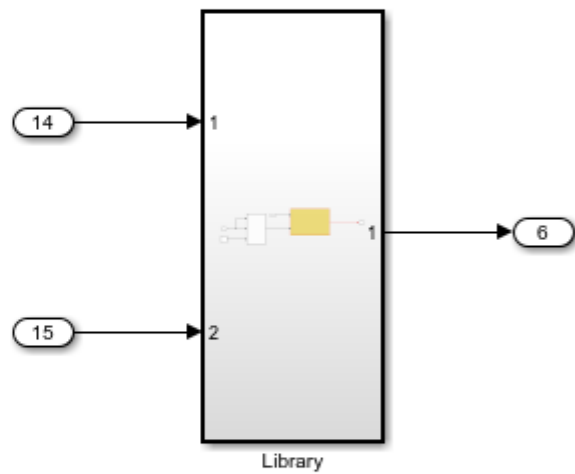
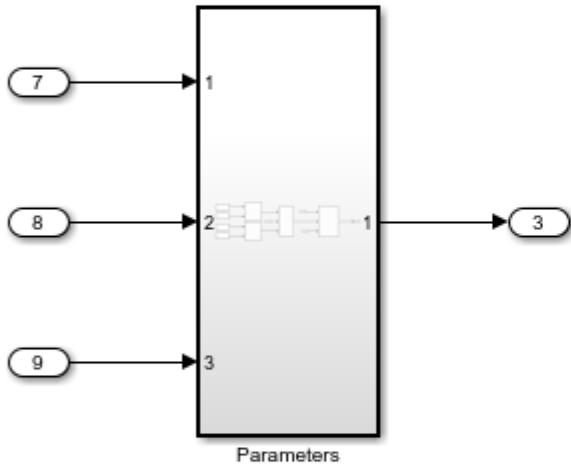
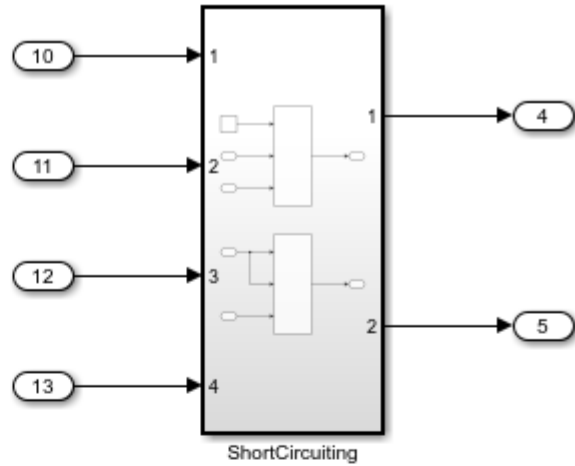
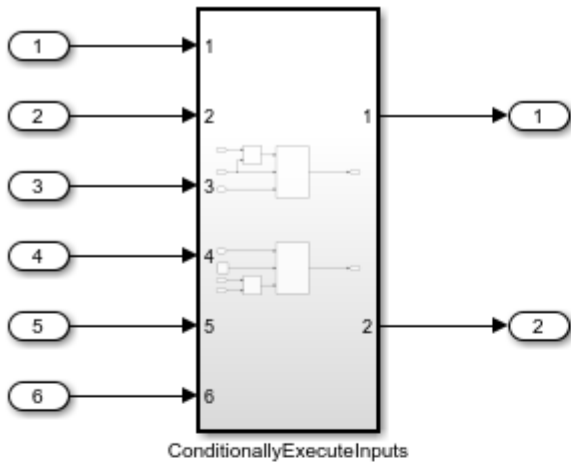
- 1 Conditional execution of a subsystem
- 2 Short-circuiting of a logical operator block during analysis
- 3 Parameter values treated as constants
- 4 Library-linked blocks
- 5 Upstream blocks
- 6 Restrictions on signal ranges

### Section 1 : Run a Dead Logic Analysis

Follow these steps to run the dead logic analysis:

1: Open the model `sldvexCommonCausesOfDeadLogic`.

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





2: In the **Apps** pane, open **Design Verifier**.

3: On the **Design Verifier** tab, click **Error Detection Settings**.

4: In the **Configuration Parameters** dialog box:

a. Enable the **Dead logic (partial)** option.

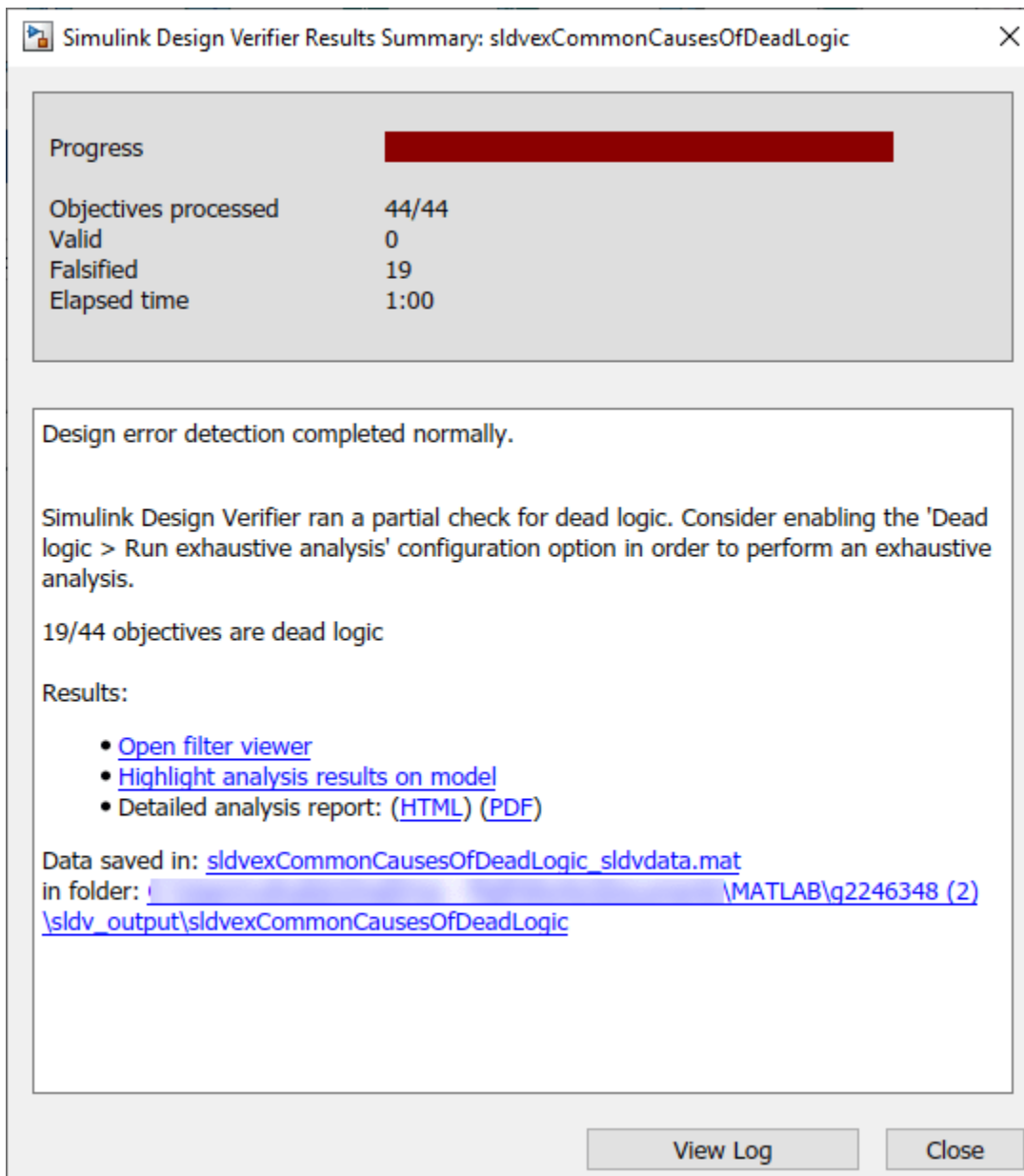
b. Clear the **Run exhaustive analysis** option, if it is selected.

c. Set **Coverage objectives to be analyzed** to **Condition Decision** option. The available options from the drop-down menu are **Decision**, **Condition Decision**, and **MCDC**.

5: In the **Design Verifier** tab, Click **Detect Design Errors**.

## **Section 2: Analyze and Review the Results**

The software analyzes the model for dead logic and displays the results in the Results Summary window. The results indicate that 19 of the 44 objectives are dead logic.



### Section 3: Highlight Analysis Results in the Subsystem Blocks

This section explains the common patterns that lead to dead logic in the `sldvexCommonCausesOfDeadLogic` model. In the Results Summary window, click on **Highlight analysis results on model**. The subsystems with dead logic are highlighted in red. These subsystems are:

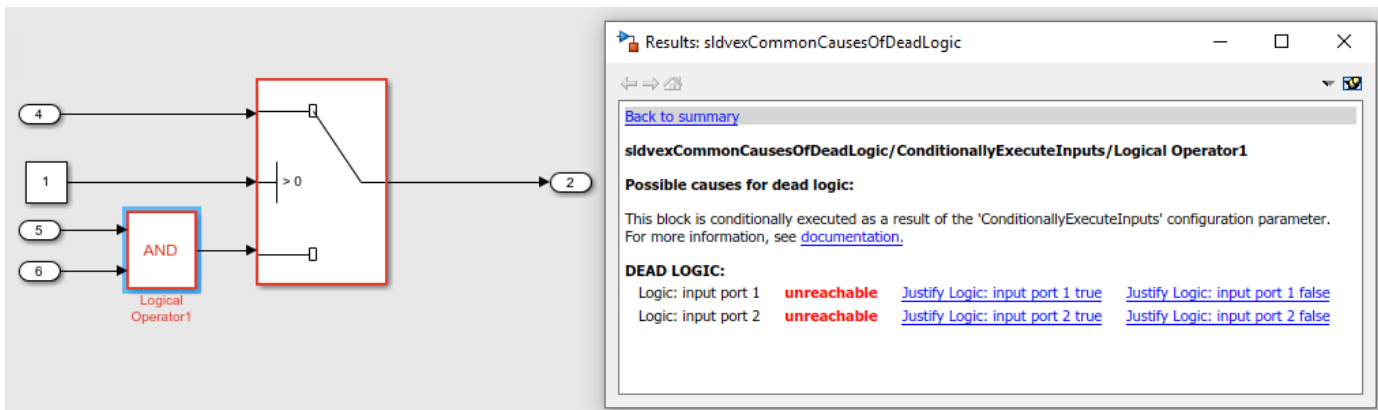
- 1 ConditionallyExecuteInputs
- 2 ShortCircuiting
- 3 Parameters
- 4 Library

- 5 CascadingDeadLogic
- 6 ConditionGreaterThan0

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

### 1. Conditional Execution of a Subsystem

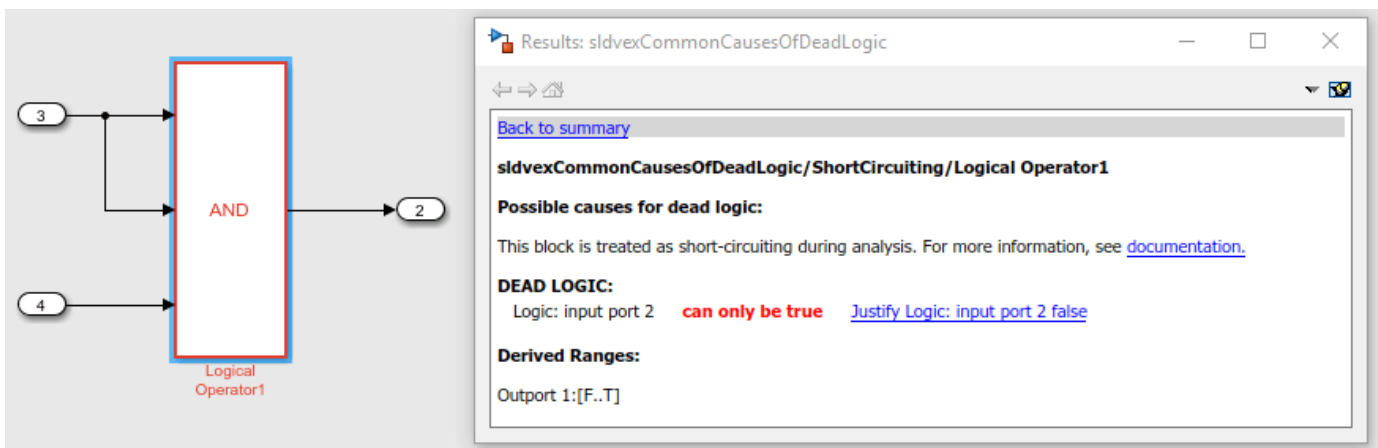
If your model includes **Switch** or **Multiport Switch** blocks and the conditional input branch execution parameter is set to `On`, the conditional execution can often cause unexpected dead logic. Open the `ConditionallyExecuteInputs` subsystem and click the **AND** block highlighted in red. The Results window summarizes the dead logic.



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

### 2. Short-Circuiting of a Logical Operator Block During Analysis

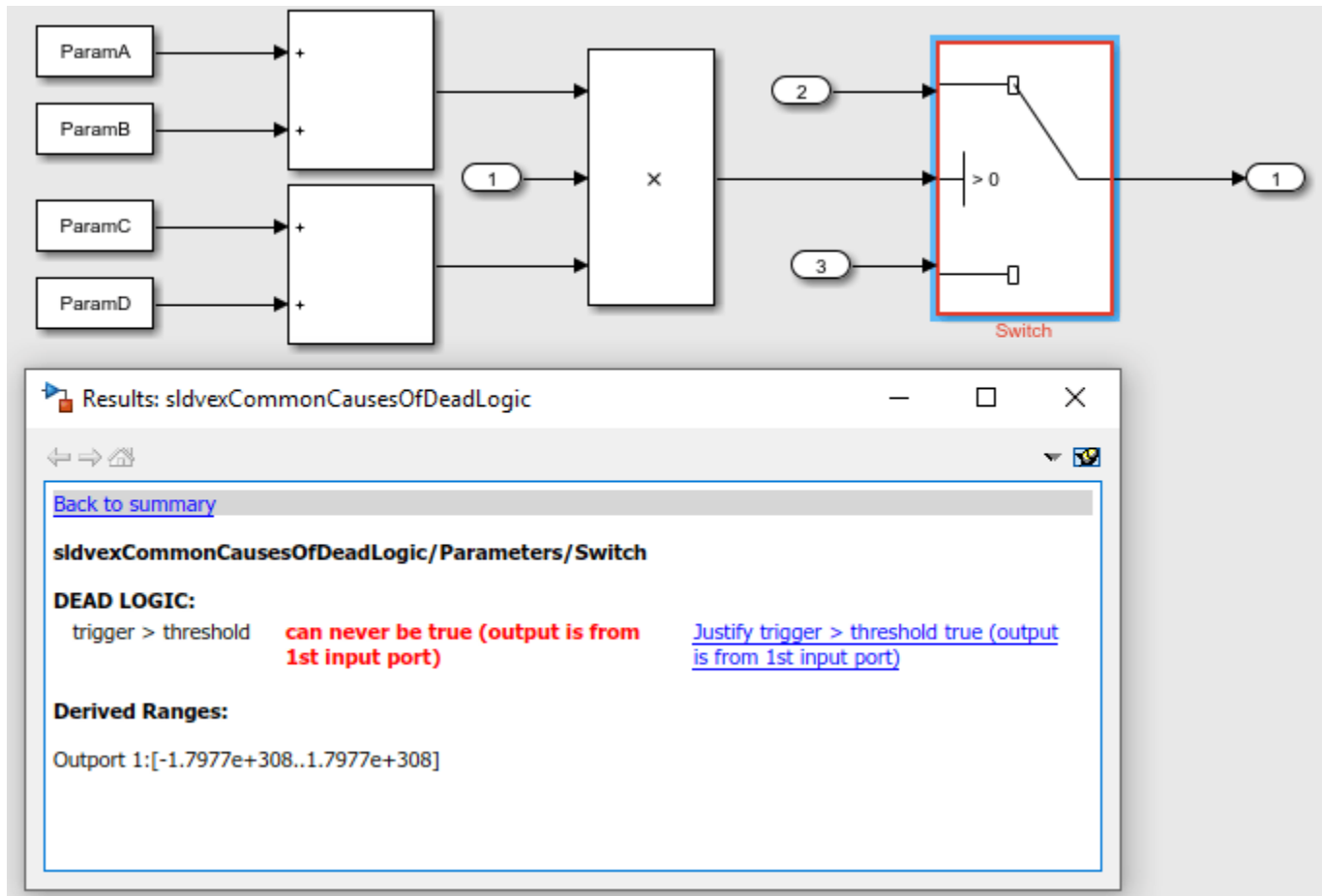
Simulink Design Verifier treats logic blocks as if they are short-circuiting when analyzing for dead logic. Open the `ShortCircuiting` subsystem, and click the **AND** block highlighted in red. The Results window summarizes the dead logic.



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

### 3. Parameter Values Treated as Constants

If your model contains parameters, Simulink Design Verifier treats the values as constants by default, which might cause dead logic in the model. In these cases, consider configuring these parameters to be tuned during analysis. Open the ShortCircuiting subsystem and click the **Switch** block highlighted in red. The Results window summarizes the dead logic.



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

#### Suggestion

You can use Model Slicer to find the parameters which could have an impact on a particular block by following these steps:

- a. Create an object of `SLSlicerAPI.ParameterDependence` using Model Slicer.

```
slicerObj = slslicer('sldvexCommonCausesOfDeadLogic');
pd = slicerObj.parameterDependence;
```

- b. Find the parameters affecting the **Product** block.

```
params = parametersAffectingBlock(pd, 'sldvexCommonCausesOfDeadLogic/Parameters/Product');
```

|   |   |
|---|---|
| <pre>&gt;&gt; params (1)  ans =  VariableUsage with properties:      Name: 'ParamA'     Source: 'base workspace'     SourceType: 'base workspace'     Users: {2x1 cell}</pre> | <pre>&gt;&gt; params (2)  ans =  VariableUsage with properties:      Name: 'ParamB'     Source: 'base workspace'     SourceType: 'base workspace'     Users: {2x1 cell}</pre> |
| <pre>&gt;&gt; params (3)  ans =  VariableUsage with properties:      Name: 'ParamC'     Source: 'base workspace'     SourceType: 'base workspace'     Users: {2x1 cell}</pre> | <pre>&gt;&gt; params (4)  ans =  VariableUsage with properties:      Name: 'ParamD'     Source: 'base workspace'     SourceType: 'base workspace'     Users: {2x1 cell}</pre> |

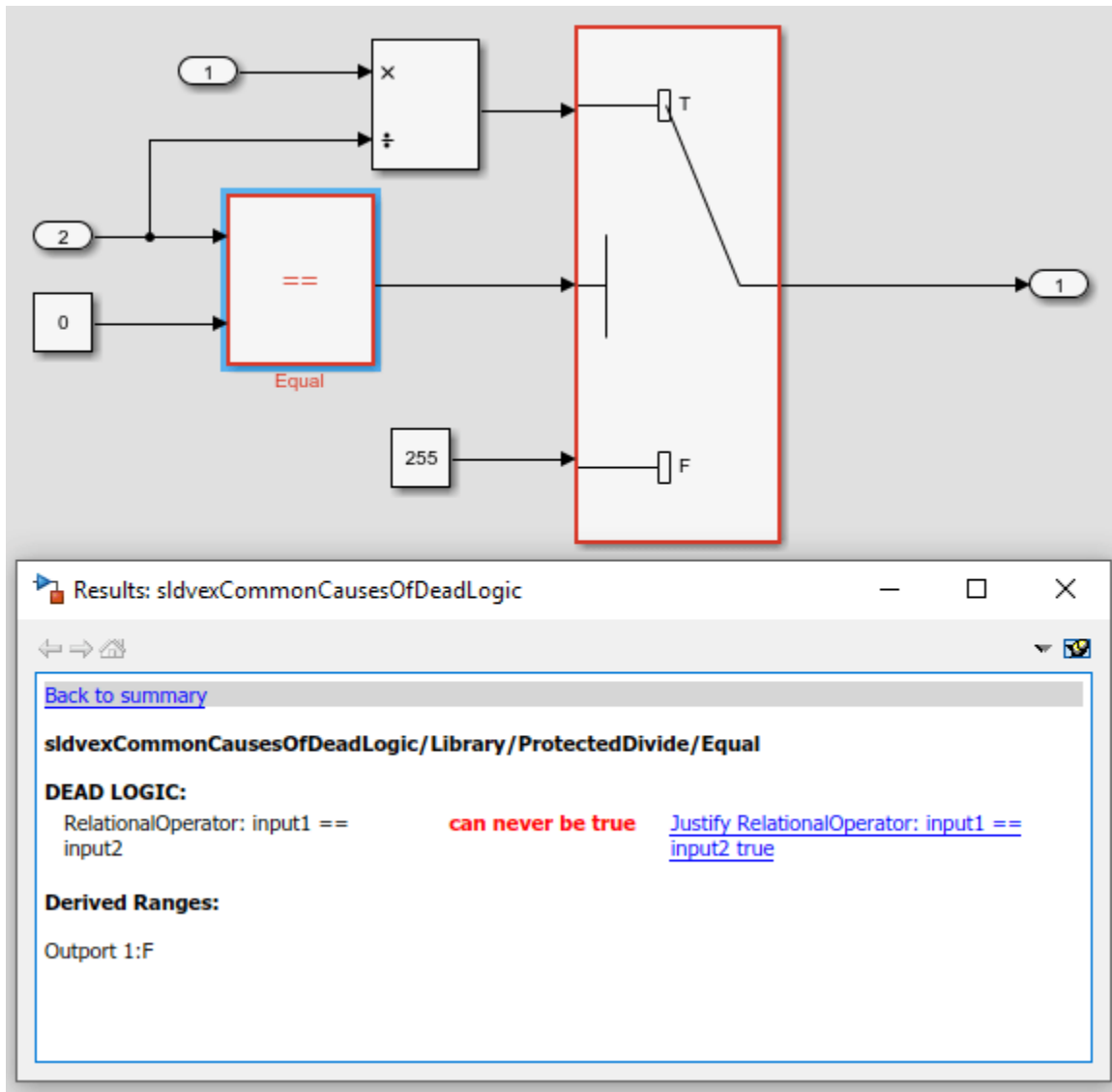
The image above displays the parameters returned by the function **parametersAffectingBlock** which have an impact on the Product block. The list of parameters returned by the function can be considered for tuning.

c. Perform clean-up to exit compile state of the model.

```
slicerObj.terminate;
```

#### 4. Library-Linked Blocks

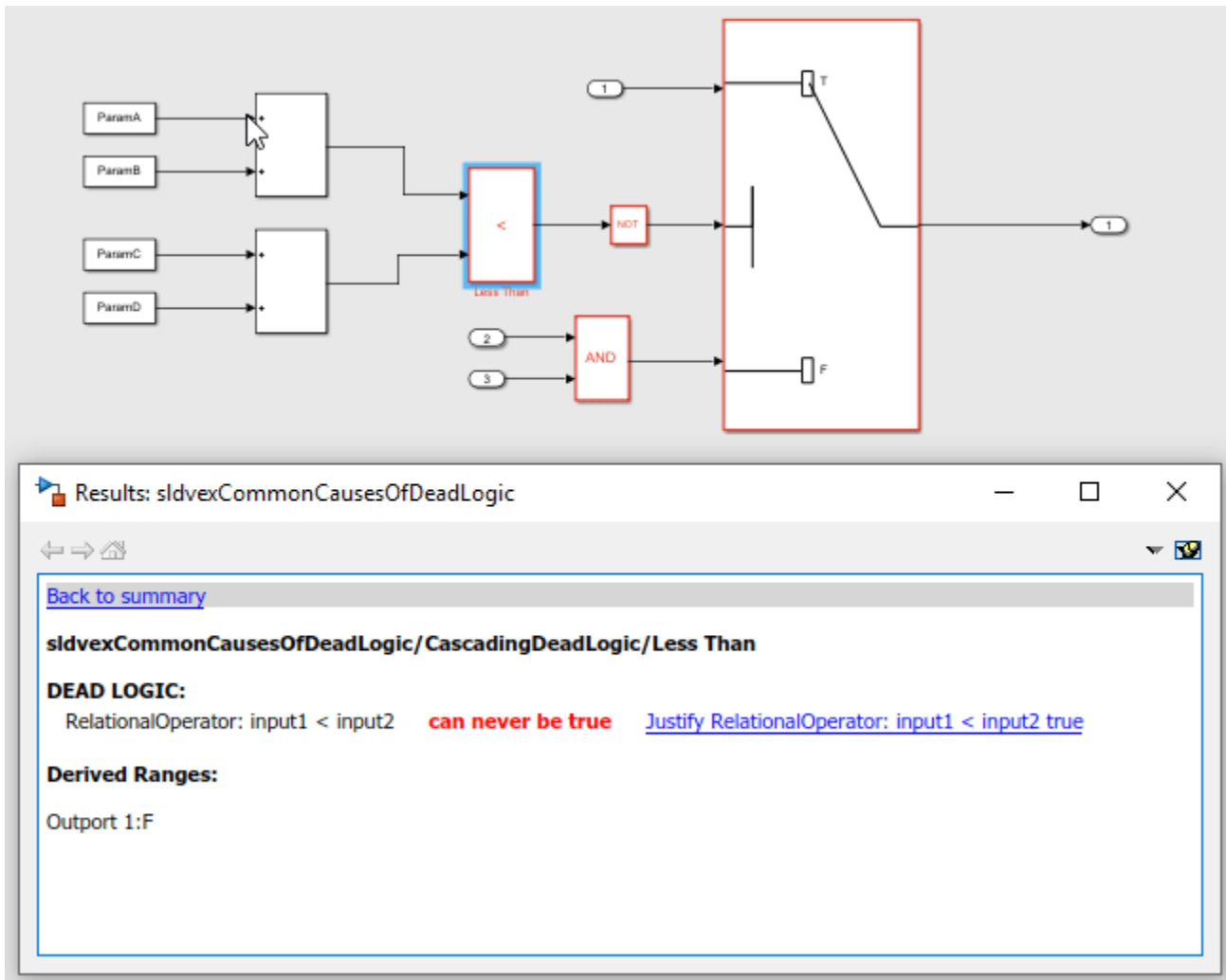
The ProtectedDivide library subsystem has protection for division by zero. **Library** blocks may be written with defensive conditions that are redundant in some of the locations where they are used. In some cases, this may cause dead logic. Open the **Library** block, and click the ProtectedDivide subsystem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



Consider justifying the dead logic that arises from those library blocks.

### 5. Upstream Blocks

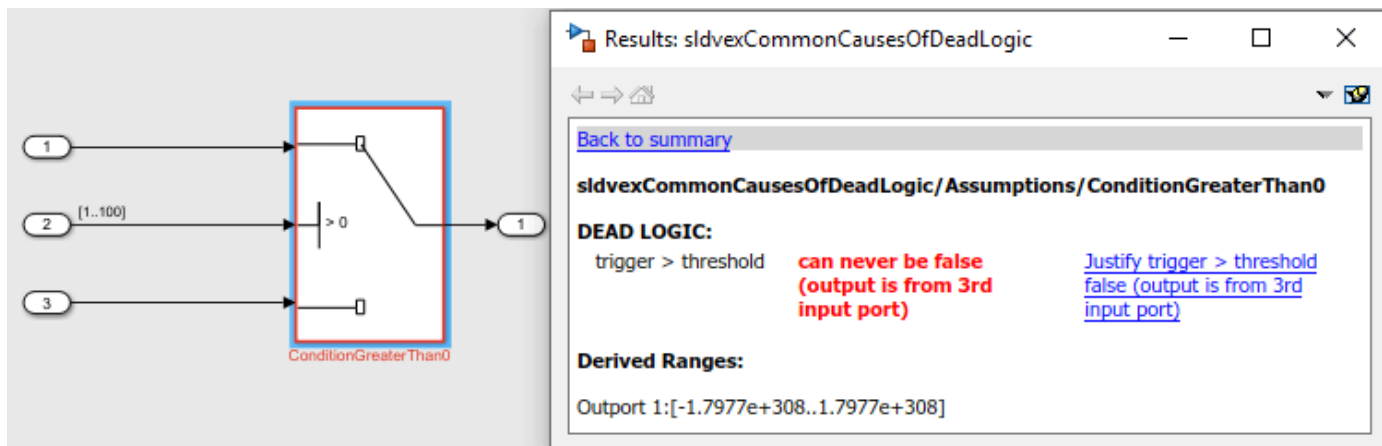
When a particular block has dead logic, this often leads to a cascading effect that causes downstream blocks to also have dead logic. Open the CascadingDeadLogic subsystem and click the **Less Than** block highlighted in red. The Results window summarizes the dead logic.



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

## 6. Restrictions on Signal Ranges

Root-level Inport blocks with minimum and maximum values as constraints and **Test Condition** blocks in the test generation may cause dead logic. For example, consider the ConditionGreaterThan0 **Switch** block, where the second Inport block has a minimum and maximum range of 1 and 100, respectively. This causes the **Switch** block in this subsystem to have dead logic, because the constrained range means the signal will always be greater than 0.



### Section 4: View the Analysis Report

In the Results summary window, click **HTML** to view the detailed analysis report. The report summarizes all of the dead logic results in the model.

## Chapter 3. Dead Logic

Simulink Design Verifier proved these decisions and conditions to be unreachable or dead logic. Dead logic can be a side effect of parameter configurations or minimum and maximum constraints specified on inputs. Simulink Design Verifier ran a partial check for dead logic. Consider enabling the 'Dead logic > Run exhaustive analysis' configuration option in order to perform an exhaustive analysis.

| #  | Type      | Model Item   | Description   |
|----|-----------|--|---|
| 1  | Condition | <a href="#">ConditionallyExecuteInputs/Logical Operator</a>  | Logic: input port 2 <b>can only be true</b>                                     |
| 2  | Decision  | <a href="#">ConditionallyExecuteInputs/Switch1</a>           | trigger > threshold <b>can never be false (output is from 3rd input port)</b>   |
| 3  | Condition | <a href="#">ConditionallyExecuteInputs/Logical Operator1</a> | Logic: input port 1 <b>unreachable</b>  |
| 4  | Condition | <a href="#">ConditionallyExecuteInputs/Logical Operator1</a> | Logic: input port 2 <b>unreachable</b>  |
| 5  | Decision  | <a href="#">Parameters/Switch</a>                            | trigger > threshold <b>can never be true (output is from 1st input port)</b>    |
| 6  | Condition | <a href="#">ShortCircuiting/Logical Operator</a>             | Logic: input port 1 <b>can only be true</b>                                     |
| 7  | Condition | <a href="#">ShortCircuiting/Logical Operator1</a>            | Logic: input port 2 <b>can only be true</b>                                     |
| 8  | Condition | <a href="#">Library/ProtectedDivide/Equal</a>                | RelationalOperator: input1 == input2 <b>can never be true</b>                   |
| 9  | Decision  | <a href="#">Library/ProtectedDivide/Switch</a>               | logical trigger input <b>can never be true (output is from 1st input port)</b>  |
| 10 | Condition | <a href="#">CascadingDeadLogic/Less Than</a>                 | RelationalOperator: input1 < input2 <b>can never be true</b>                    |
| 11 | Condition | <a href="#">CascadingDeadLogic/Logical Operator</a>          | Logic: input port 1 <b>can never be true</b>                                    |
| 12 | Decision  | <a href="#">CascadingDeadLogic/Switch</a>                    | logical trigger input <b>can never be false (output is from 3rd input port)</b> |
| 13 | Condition | <a href="#">CascadingDeadLogic/Logical Operator1</a>         | Logic: input port 1 <b>unreachable</b>  |
| 14 | Condition | <a href="#">CascadingDeadLogic/Logical Operator1</a>         | Logic: input port 2 <b>unreachable</b>  |
| 15 | Decision  | <a href="#">Assumptions/ConditionGreaterThan0</a>            | trigger > threshold <b>can never be false (output is from 3rd input port)</b>   |



To perform an exhaustive analysis for dead logic, in the **Configuration Parameters Window** in the **Design Error Detection** pane, select **Run exhaustive analysis**. The software stores the detailed analysis results in the **DeadLogic** field in the Simulink Design Verifier data files. You can use the data file to further analyze the results.

**Related Topics**

- “Common Causes for Dead Logic” on page 6-15



# Generating Test Cases

---

- “What Is Test Case Generation?” on page 7-3
- “Workflow for Test Case Generation” on page 7-5
- “Generate Test Cases for Model Decision Coverage” on page 7-6
- “Generate Test Cases for a Subsystem” on page 7-18
- “Generate Test Cases for a Reusable Library Subsystem” on page 7-21
- “Use Test Generation Advisor to Identify Analyzable Components” on page 7-24
- “Generate Test Cases for Embedded Coder Generated Code” on page 7-28
- “Model Coverage Objectives for Test Generation” on page 7-30
- “Enhance Model Coverage of Older Release Models” on page 7-32
- “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42
- “Analyze Model for Enhanced MCDC Analysis” on page 7-44
- “Basic Workflow for Enhanced MCDC Analysis” on page 7-47
- “Author Custom Test Objective Workflow” on page 7-52
- “What Is a Specification Model?” on page 7-60
- “Test Generation Examples” on page 7-66
- “Test Generation for Custom Code in MATLAB Function Block” on page 7-67
- “Use Specification Models for Requirements-Based Testing” on page 7-69
- “Flip Flop Test Generation” on page 7-80
- “Model Coverage Test Generation” on page 7-81
- “Test Objective Block” on page 7-82
- “Test Condition Block” on page 7-83
- “Cruise Control Test Generation” on page 7-84
- “Fuel Rate Controller Logic” on page 7-85
- “Extend an Existing Test Suite” on page 7-86
- “Defining and Extending Existing Tests Cases” on page 7-91
- “Using Existing Coverage Data During Subsystem Analysis” on page 7-97
- “Creating and Executing Test Cases” on page 7-100
- “Using Specified Input Minimum and Maximum Values as Constraints” on page 7-107
- “Configuring S-Function for Test Case Generation” on page 7-109
- “Code Coverage Test Generation” on page 7-111
- “Test Generation on Model with C Caller Block” on page 7-119
- “Debug Enhanced Modified Condition Decision Coverage Using Model Slicer” on page 7-121
- “Test Generation for Custom Code in a Stateflow Chart” on page 7-124
- “Generate Test Cases for Model Blocks” on page 7-126
- “Use Observer Reference Block for Test Case Generation” on page 7-130

- “Inspect Test Generation Objectives by Using Model Slicer” on page 7-135
- “Generate Tests for Model Block Component by Using Default Simulation” on page 7-138
- “Add Test Cases Using Excel File” on page 7-142
- “Achieve Missing Coverage in Custom Code” on page 7-146
- “Achieve Missing Coverage in Generated Code of RLS” on page 7-149

## What Is Test Case Generation?

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

- “Decision” on page 7-30
- “Condition” on page 7-30
- “MCDC” on page 7-31
- “Enhanced MCDC” on page 7-31

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

---

**Note** If your model does not have conditions, decisions, or custom test objectives, then Simulink Design Verifier generates a test case that represents a basic simulation of your model. The test inputs satisfy minimum or maximum constraints on input ports and intermediate signal values satisfy constraints specified by the Test Condition blocks in the model.

---

### Test Case Blocks

For customizing test cases for your Simulink models, Simulink Design Verifier provides two blocks:

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

### Test Case Functions

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

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

These functions:

- Identify mathematical relationships for testing in a form that can be more natural than using block parameters.
- Support specifying multiple objectives, assumptions, or conditions without complicating the model.
- Provide access to the power of MATLAB.
- Support separation of verification and model design.

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

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

---

### See Also

### More About

- “Workflow for Test Case Generation” on page 7-5

## Workflow for Test Case Generation

To generate test cases for your model, use the following workflow.

| Task | Description  | For an example, see  |
|------|--|--|
| 1    | Verify that your model is compatible for use with Simulink Design Verifier.  | “Check Compatibility of the Example Model” on page 7-7                                 |
| 2    | Optionally, use the Test Generation Advisor to select model components (atomic subsystems and model blocks) for test generation. Before test generation, you can use the results to better understand your model, particularly large models, complex models, or models for which you are uncertain of the test generation compatibility. | “Use Test Generation Advisor to Identify Analyzable Components” on page 7-24           |
| 3    | If you have Stateflow objects in your model, in the Configuration Parameters dialog box, on the <b>Diagnostics &gt; Stateflow</b> pane, set <b>Unreachable execution path</b> to <b>error</b> .  |  |
| 4    | Optionally, instrument your model with blocks or MATLAB functions that specify test objectives and test conditions.  | “Customize Test Generation” on page 7-14   |
| 5    | Specify options that control how Simulink Design Verifier generates test cases for your model.   | “Configure Test Generation Options” on page 7-8  |
| 6    | Execute the Simulink Design Verifier analysis.   | “Analyze the Example Model” on page 7-8 and “Reanalyze the Example Model” on page 7-16 |
| 7    | Review the analysis results.   | “Review Analysis Results” on page 7-8  |

### See Also

#### More About

- “Flip Flop Test Generation” on page 7-80
- “Cruise Control Test Generation” on page 7-84
- “Fuel Rate Controller Logic” on page 7-85

## Generate Test Cases for Model Decision Coverage

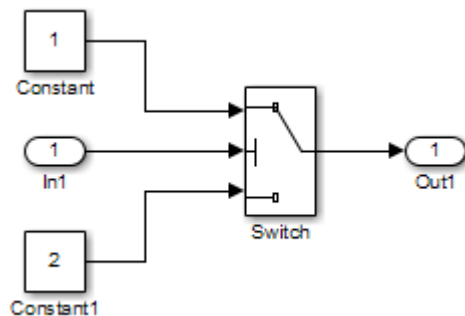
### In this section...

“Construct the Example Model” on page 7-6  
 “Check Compatibility of the Example Model” on page 7-7  
 “Configure Test Generation Options” on page 7-8  
 “Analyze the Example Model” on page 7-8  
 “Review Analysis Results” on page 7-8  
 “Customize Test Generation” on page 7-14  
 “Reanalyze the Example Model” on page 7-16  
 “Analyze Contradictory Models” on page 7-16

### Construct the Example Model

Construct a model for this example:

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



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



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

Simulink Design Verifier analyzes only models that use a fixed-step solver.

- 8 Click **OK** to save your changes and close the Configuration Parameters dialog box.
- 9 Save your model with the name `ex_generate_test_cases_example`.

## Check Compatibility of the Example Model

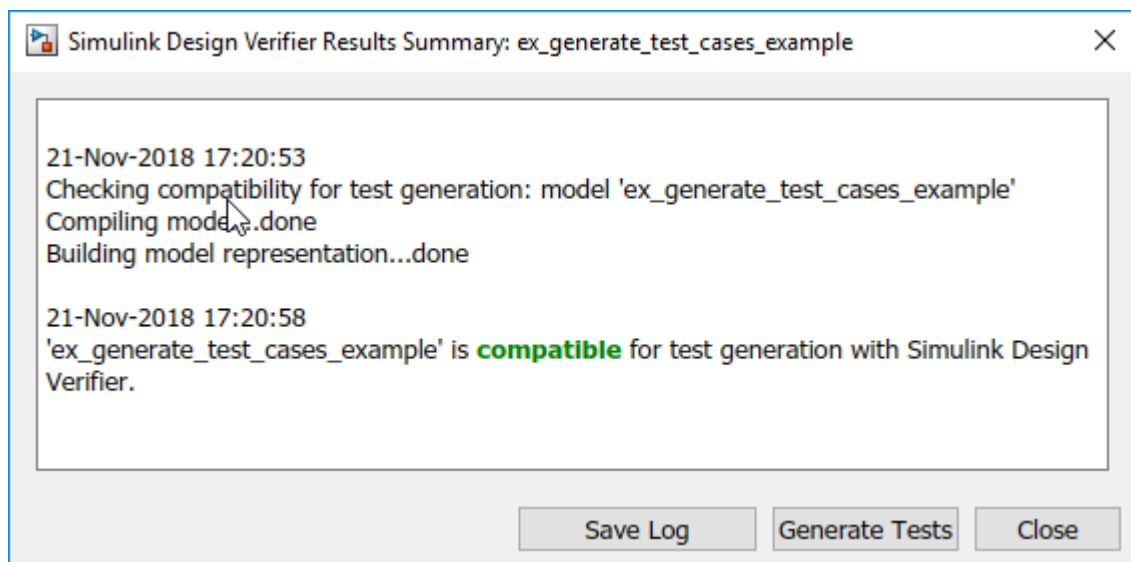
Every time Simulink Design Verifier analyzes a model, before the analysis begins, the software performs a compatibility check. If your model is not compatible, the software cannot analyze it.

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

- 1 Open the `ex_generate_test_cases_example` model.
- 2 On the **Design Verifier** tab, click **Check Compatibility**.

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

The model you just created is compatible.



### What If a Model Is Partially Compatible?

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

For detailed information about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.

## Configure Test Generation Options

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

- 1 Open the `ex_generate_test_cases_example` model.
- 2 On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**.
- 3 Click **Test Generation Settings**.
- 4 In the Configuration Parameters dialog box, on the **Test Generation** pane, set the **Model coverage objectives** parameter to **Decision**.

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

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

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

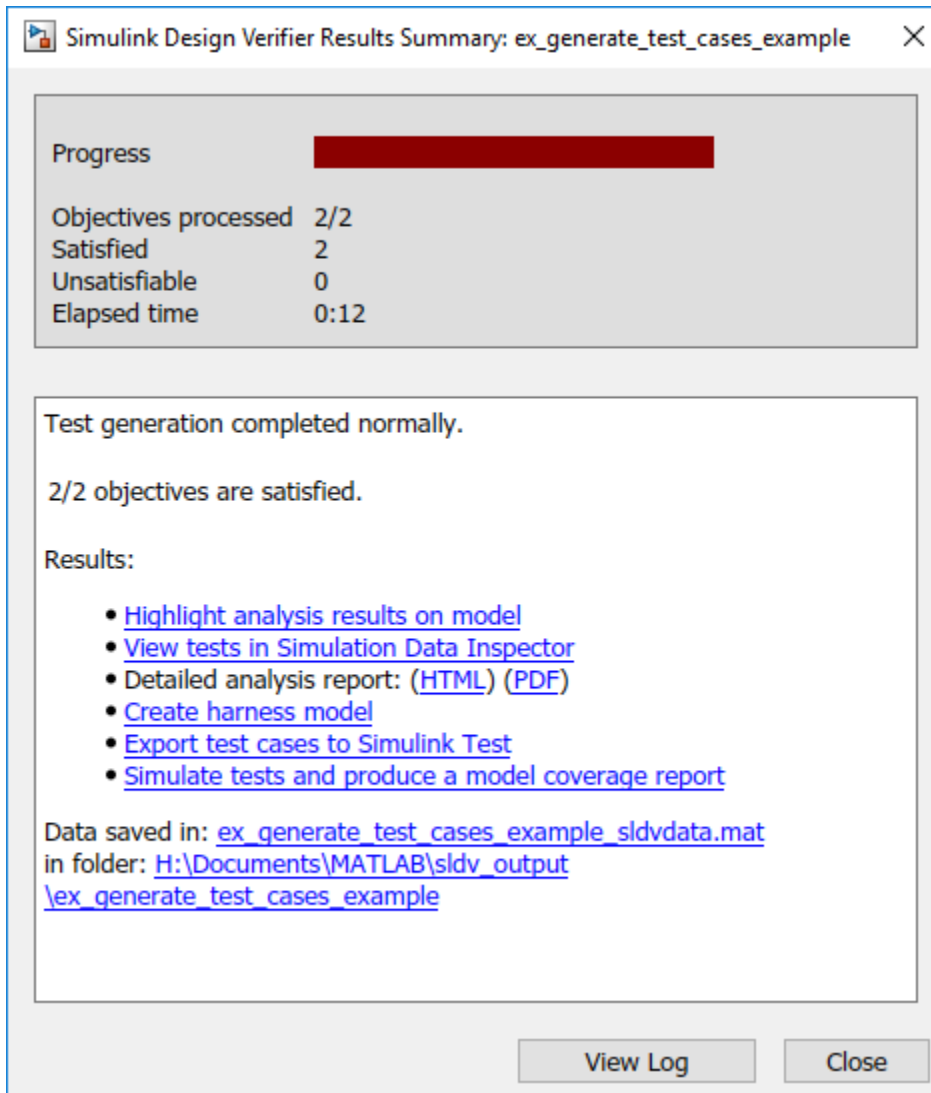
## Analyze the Example Model

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

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

## Review Analysis Results

When the software completes its analysis, the Results Summary window displays these options for reviewing the results.



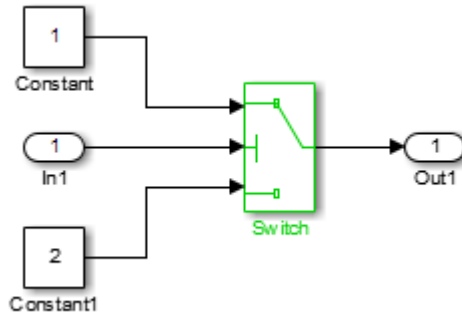
The following sections describe how you can review the analysis results:

- "Review Analysis Results on the Model" on page 7-9
- "Review Detailed Analysis Report" on page 7-11
- "Review Harness Model" on page 7-12
- "Simulate Tests and Produce a Model Coverage Report" on page 7-12
- "View sldvData File" on page 7-14
- "Review Analysis Results in the Results Summary Window" on page 7-14


### Review Analysis Results on the Model

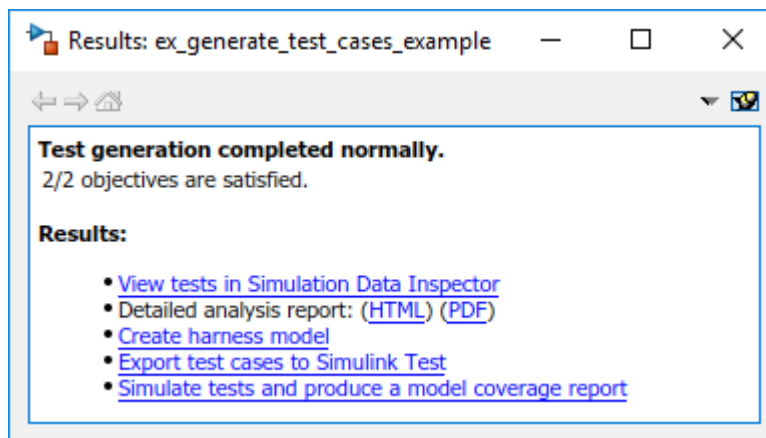
Highlight the analysis results on the example model:

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



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

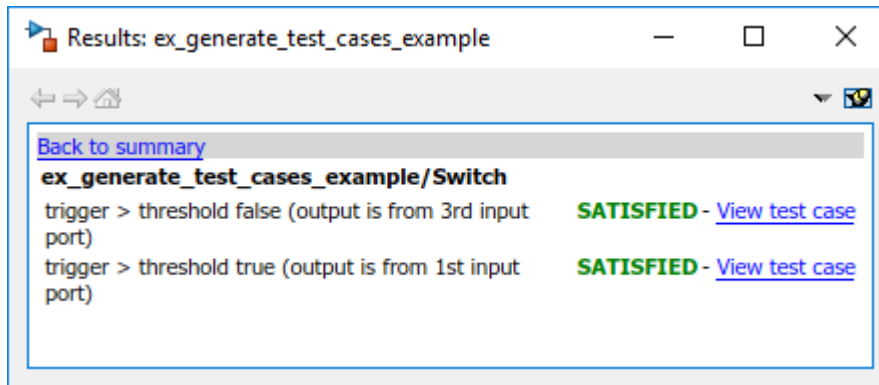
The Simulink Design Verifier Results window opens. As you click objects in the model, this window changes to display detailed analysis results for that object. By default, the Simulink Design Verifier Results window is always the topmost visible window. To allow the window to move behind other window, click  and clear **Always on top**.



- 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 “Highlight Results on the Model” on page 13-2.

### Review Detailed Analysis Report

Create a detailed HTML analysis report:

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

The HTML report opens in a browser window.

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

| Table of Contents                         |
|---|
| <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.

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

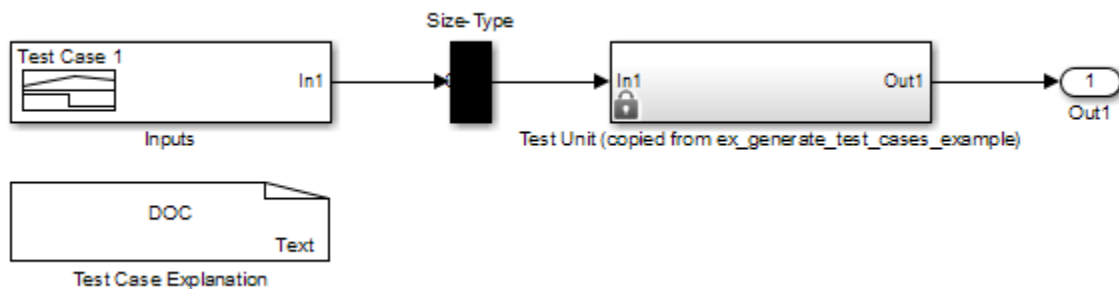
This section provides details about a test case that the analysis generated to achieve an objective in your model. This test case achieves test objective 1, when the Switch block passes its third input to its output port. Specifically, the software determines that a value of -1 for the Switch block control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

For more information about the HTML reports, see “Review Results” on page 13-35.

### Review Harness Model

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

The software creates a harness model named `ex_generate_test_cases_example_harness`.



The Signal Builder block named Inputs contains the test cases. Double-click the Inputs block to see the test cases. From the Signal Builder block, you can simulate the model using the test cases and produce a model coverage report, as described in “Simulate Tests and Produce a Model Coverage Report” on page 7-12.

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

### If Analysis Generates Many Test Cases

If you have a large model, the analysis might produce a harness model that contains a large number of test cases.

To generate fewer test cases:

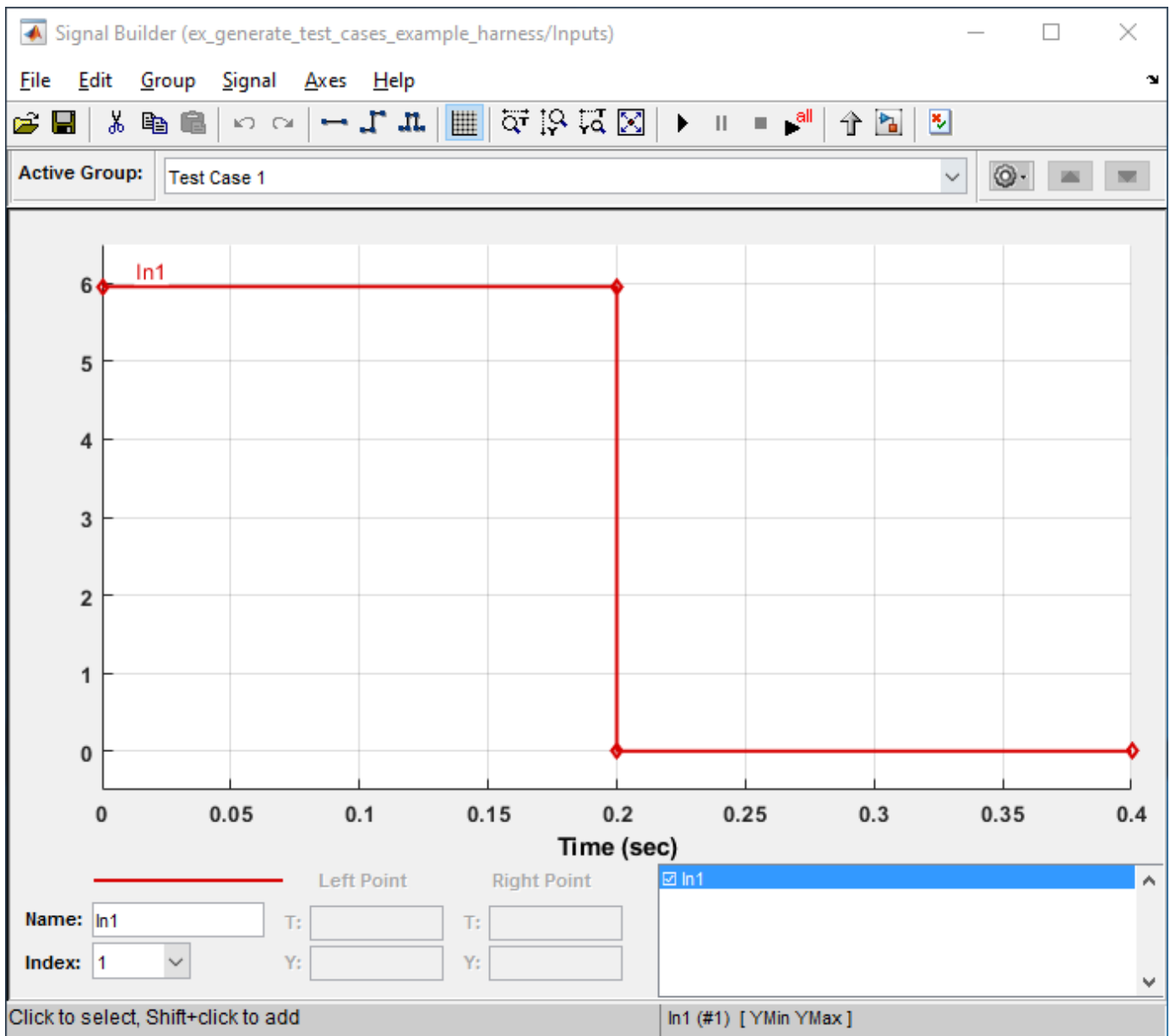
- Set the **Test suite optimization** parameter to `LongTestcases`.
- Rerun the analysis.

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


### Simulate Tests and Produce a Model Coverage Report

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

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



2

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

The software simulates the harness model using both test cases, collects model coverage information, and displays a coverage report. The coverage report indicates that the test cases record 100% decision coverage for the `ex_generate_test_cases_example` model.

You can also simulate the model without creating a harness model. In the Simulink Design Verifier log window, click **Simulate tests and produce a model coverage report**.

For more information about model coverage, see “Top-Level Model Coverage Report” (Simulink Coverage).

## View sldvData File

The Simulink Design Verifier data file is a MAT-file that contains a structure named `sldvData`. This structure stores all the data that the analysis gathers and produces during the analysis. You can use the data file to conduct your own analysis or to generate a custom report.

To view the data file, click the data file name in the log window, in this example, `ex_generate_test_cases_example_sldvdata.mat`. When you click the file name, a copy of the `sldvData` object is instantiated in the MATLAB workspace so that you can review and manipulate the data.

For more information about Simulink Design Verifier data files, see “Manage Simulink Design Verifier Data Files” on page 13-7.

## Review Analysis Results in the Results Summary Window

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

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

For any Simulink Design Verifier analysis, from the Results Summary window, you can perform these tasks.

| Task   | For more information   |
|--|--|
| Highlight the analysis results on the model.   | “Highlight Results on the Model” on page 13-2                  |
| Generate a detailed analysis report.   | “Review Results” on page 13-35                                 |
| Create the harness model, or if the harness model already exists, open it.<br>If no test cases were generated during the analysis, this option is not available. | “Manage Simulink Design Verifier Harness Models” on page 13-13 |
| View the data file.  | “Manage Simulink Design Verifier Data Files” on page 13-7      |
| View the log file.   | “View Log Files” on page 13-56                                 |

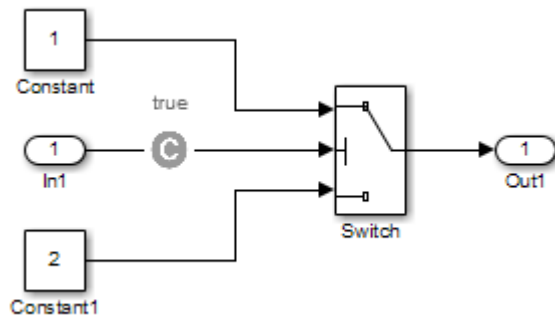
After you close your model, you can no longer view analysis results.

## Customize Test Generation

You can use the Test Condition block to constrain signals in your model to certain values during the analysis.

- 1 At the MATLAB command prompt, enter `sldvlib` to display the Simulink Design Verifier library.
- 2 Open the Objectives and Constraints sublibrary.
- 3 Copy the Test Condition block to your model by dragging it from the Simulink Design Verifier library to your model window.
- 4 In the model window, insert the Test Condition block between the Inport and Switch blocks.

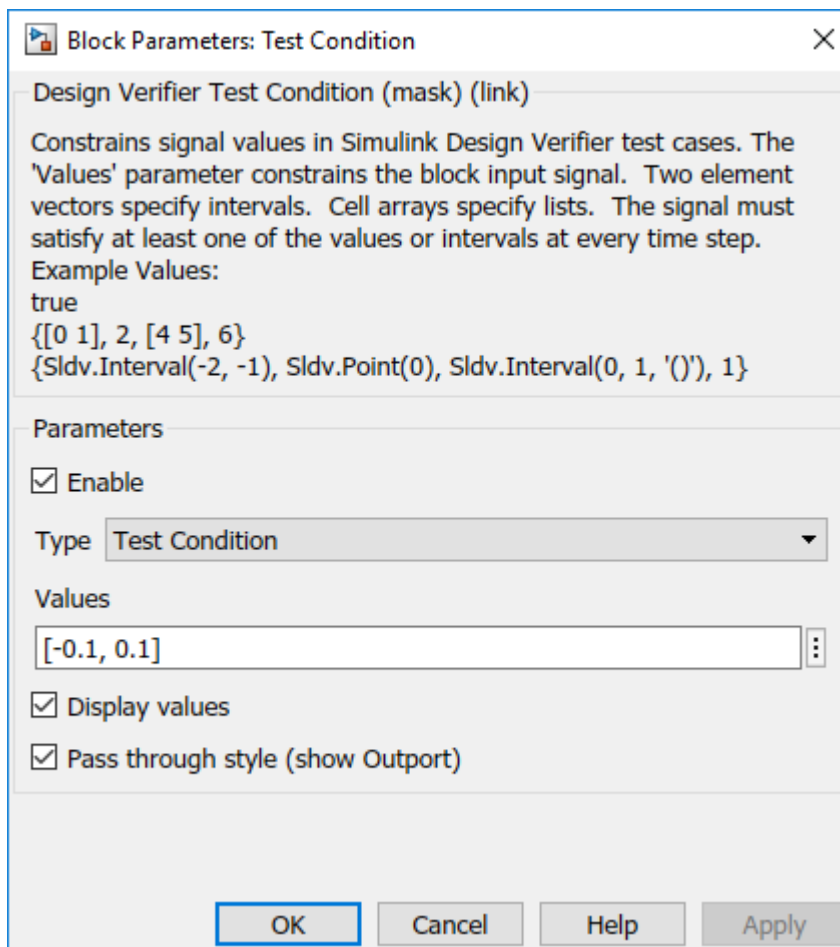




- 5 Double-click the Test Condition block to access its attributes.

The Test Condition block parameters dialog box opens.

- 6 In the **Values** box, enter  $[-0.1, 0.1]$ . When generating test cases for this model, the analysis constrains the signal values, entering the Switch block control port to the specified range.



- 7 Click **OK** to save your changes and close the Test Condition block parameters dialog box.
- 8 Save your model as `ex_generate_test_cases_with_tc_block` and keep it open.

## Reanalyze the Example Model

Analyze the `ex_generate_test_cases_with_tc_block` model with the Test Condition block. To observe how the Test Condition block affects test generation, compare the result of this analysis to the result that you obtained in “Analyze Example Model” on page 5-20.

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

The Simulink Design Verifier software displays a log window and begins analyzing your model to generate test cases. When the software completes the analysis, the Results Summary window displays the options for reviewing the results.

- 2 In the Results Summary window, click **HTML Report**.
- 3 To begin reviewing the report, in the **Table of Contents**, click Summary.

The Summary chapter indicates that Simulink Design Verifier satisfied two test objectives in the model.

- 4 In the **Table of Contents**, click Analysis Information. Scroll to the bottom of this chapter, to the Constraints section.

This section lists the Test Condition block that you added to constrain the value of the Switch block control signal to the interval  $[-0.1, 0.1]$ .


- 5 In the **Table of Contents**, click Test Objectives Status.

This table indicates that Simulink Design Verifier satisfied both test objectives for the Switch block through the two test cases generated.

- 6 Under the table **Test Case** column, click 1.

This section provides details about a test case that the software generated to achieve an objective in your model. This test case achieves test objective 1, when the Switch block passes its third input to its output port. Although the Test Condition block restricts the domain of input signals to the interval  $[-0.1, 0.1]$ , the software determines that a value of  $-0.1$  for the Switch block control signal satisfies this objective.

- 7 To confirm that the test case achieves 100% decision coverage, open the harness model.
- 8 Double-click the Inputs block to open the Signal Builder dialog box.

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

The Simulink software simulates the harness model using both test cases, collects model coverage information, and displays a coverage report. The Summary section of the report indicates that Simulink Design Verifier generated test cases that achieve complete decision coverage for your example model.

## Analyze Contradictory Models

If the analysis produces the error `The model is contradictory in its current configuration`, the software detected a contradiction in your model and cannot analyze the model.

You can have a contradiction if your model has Test Objective blocks with incorrect parameters. For example, a contradiction can be an objective that states that a signal must be between 0 and 5 when the signal is the constant 10.

If the software detects a contradiction, all previous results are invalidated and the software reports that some of the objectives cannot be satisfied.

## **See Also**

## **More About**

- Model Coverage Test Generation on page 7-81

## Generate Test Cases for a Subsystem

You can analyze a subsystem within a model. This technique is good for large models, where you want to review the analysis in smaller, manageable reports. Following two methods help you to generate test cases for subsystem in different modes:

- “Generate Test Cases for Subsystems for Normal Mode” on page 7-18
- “Generate Test Cases for Subsystems for Software-in-the-Loop Mode” on page 7-19

### Generate Test Cases for Subsystems for Normal Mode

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

- 1 Open the example model:

```
sldvdemo_cruise_control
```

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

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

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

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

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

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

- 4 To close the dialog box, click **OK**.
- 5 On the **Simulation** tab, in the **File** section, select **Save > Save As** and save the Cruise Control Test Generation model with a new name.
- 6 To start the subsystem analysis and generate test cases, right-click the Controller subsystem, and select **Design Verifier > Generate Tests for Subsystem**.
- 7 The Simulink Design Verifier software analyzes the subsystem. When the analysis is complete, view the analysis results for the Controller subsystem by clicking one of the following options:

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

---

**Note** After processing a certain number of objectives, if the analysis stops, or if the analysis times out, you can use the Test Generation Advisor to better understand which subsystems are

causing the problem. For more information, see “Use Test Generation Advisor to Identify Analyzable Components” on page 7-24.

- 8 Review the results of the subsystem analysis and compare the results to the results of the full-model analysis as described in “Analyze a Model” on page 1-4:
  - The subsystem analysis analyzes the Controller as a standalone model.
  - The Controller subsystem contains all the test objectives in the Cruise Control Test Generation model. Both the analyses generate the same test cases.

## Generate Test Cases for Subsystems for Software-in-the-Loop Mode

This example shows how to generate test cases for atomic subsystems in software-in-the-loop (SIL) mode by using the `sldvdemo_cruise_control_ATS` model.

- 1 Open the example model: `sldvdemo_cruise_control_ATS`

```
model = 'sldvdemo_cruise_control_ATS';
open_system(model);
```

- 2 In the **Configuration Parameters** window, click **Code Generation** and set **System Target File** to `ert.tlc`. Alternatively, enter:

```
set_param(model, 'SystemTargetFile', 'ert.tlc');
```

- 3 Click **Hardware Implementation**, then set **Device vendor** and **Device type** to the vendor and type of your SIL system. For example, for a 64-bit Linux machine, set **Device vendor** to Intel and **Device type** to `x-86-64 (Linux)`. Alternatively, enter:

```
if ismac
    lProdHWDeviceType = 'Intel->x86-64 (Mac OS X)';
elseif isunix
    lProdHWDeviceType = 'Intel->x86-64 (Linux 64)';
else
    lProdHWDeviceType = 'Intel->x86-64 (Windows64)';
end
set_param(model, 'ProdHWDeviceType', lProdHWDeviceType);
```

- 4 Generate the code for the target. For subsystem analysis in SIL mode, code needs to be generated before invoking test generation.

- a If the test generation target is **Code Generated as Top model**, generate the code for the target by entering:

```
slbuild(model, 'StandaloneCoderTarget');
```

- b If the test generation target is **Code Generated as Model Reference**, generate the code for the target by entering:

```
slbuild(model, 'ModelReferenceCoderTargetOnly');
```

---

### Note

- If there is a mismatch of the test generation target and the generated code interface target, then test generation returns an error.
  - If you generate a code for both targets, the test generation returns an error.
-

- 5 Set up the function packaging of the subsystem by right-clicking **PI Controller > Block Parameters (Subsystem) > Code Generation > Function Packaging** and set as **Reusable function** or **Nonreusable function**.

Alternatively enter:

```
ssPath = [model '/PI Controller'];  
set_param(ssPath, 'RTWSystemCode', 'Reusable function'); % For Resuable function  
set_param(ssPath, 'RTWSystemCode', 'Nonreusable function'); % For Nonresuable function
```

- 6 In the **Apps** tab, click **Design Verifier**. Then, in the **Design Verifier** tab, set **Target** to **Code Generated as Top Model**. Generate tests by using one of these methods:
  - Right click the **PI Controller** block, then click **Design Verifier > Generate Tests for Subsystems**.
  - Select the **PI Controller** block by unpinning it from the toolbar. Then click **Generate Tests**.
  - Create a harness for the subsystem and then invoke test generation by right-clicking the **PI Controller** block, then clicking **Test Harness > Create for PI Controller**.

Select the harness name and click **OK**.

Open the new harness. Then click **Design Verifier** and click **Generate Tests**.

Alternatively, you can use the API to generate the tests by entering:

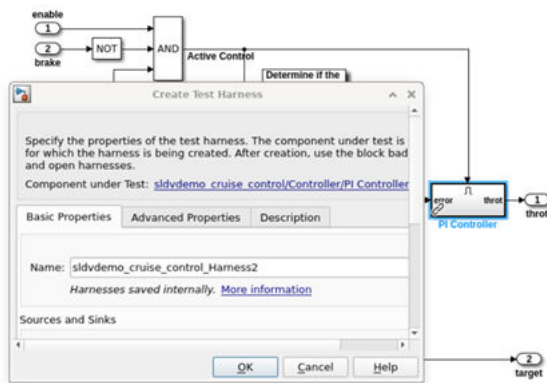
```
opts = sldvoptions;  
opts.TestgenTarget = Sldv.utils.Options.TestgenTargetGeneratedCodeStr;  
[status, fileNames] = sldvrun(ssPath,opts,true);
```

- 7 Review the results of the subsystem analysis and compare the results to the results of the full-model analysis as described in “Generate Test Cases for Subsystems for Normal Mode” on page 7-18.

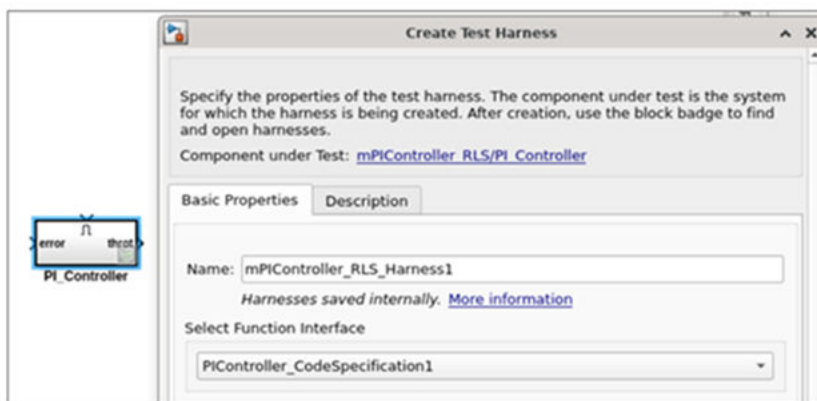
## Generate Test Cases for a Reusable Library Subsystem

A reusable library subsystem (RLS) is a subsystem that you define and include in a library and configure for reuse across models. For more information on how to configure an RLS for analysis, see “Generate Reusable Code for Subsystems Shared Across Models” (Embedded Coder). You must test the configured RLS by creating a harness from the library and not from an instance in a design model.

This example uses `sldvdemo_cruisecontrol` model, where `PI controller` is the RLS block. You can create a harness from the instance of this RLS block as shown. Test generation of RLS can be invoked on a harness RLS block created from the library and not from its instance.



When you create the test harness from the library as shown, the test generation for the RLS code from this harness is supported by the design model.



This example shows how to analyse RLS code in the Software-in-the-Loop mode.

## Generate Test Cases for RLS in Software-in-the-Loop Mode

This example shows how to generate test cases for RLS in the software-in-the-loop (SIL) mode.

1. Open the example model: 'mRLS'

```
model = 'mRLS';
open_system(model);
```

2. Unlock the library model. In the Configuration Parameters window, click **Code Generation** and set **System Target File** to `ert.tlc`. Alternatively, enter the following command:

```
set_param(model, 'Lock', 'off');
set_param(model, 'SystemTargetFile', 'ert.tlc');
```

3. Click **Hardware Implementation**, then set **Device vendor** and **Device type** to the vendor and type of your SIL system. For example, for a 64-bit Linux machine, set **Device vendor** to Intel and **Device type** to x-86-64 (Linux). Alternatively, enter the following code:

```
if ismac
    lProdHWDeviceType = 'Intel->x86-64 (Mac OS X)';
elseif isunix
    lProdHWDeviceType = 'Intel->x86-64 (Linux 64)';
else
    lProdHWDeviceType = 'Intel->x86-64 (Windows64)';
end
set_param(model, 'ProdHWDeviceType', lProdHWDeviceType);
```

4. Use the device settings to set up the function interface. For more information on how to set the function interface from within a library, see [Configure Function Interfaces from Within a Library](#).

5. Generate the top-model code before generating tests for the RLS. Before you generate the code, set up the code generation target environment. For more information on setting up the target environment, see [SIL Testing a Reusable Library Subsystem](#).

```
orig = Simulink.fileGenControl('get', 'CodeGenFolderStructure');
Simulink.fileGenControl('set', 'CodeGenFolderStructure', ...
    Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);
```

```
slbuild('mRLS');
```

```
### Starting build procedure for: Controller_CodeSpecification1
### Generating code and artifacts to 'Target environment subfolder' folder structure
### Generating code into build folder: C:\TEMP\Bdoc23a_2213998_3568\ib570499\0\tp65d2e03e\slldv-ex
### Invoking Target Language Compiler on Controller_CodeSpecification1.rtw
### Using System Target File: B:\matlab\rtw\c\ert\ert.tlc
### Loading TLC function libraries
.....
### Initial pass through model to cache user defined code
.
### Caching model source code
.....
### Writing header file Controller_Lp0dbbft.c
### Writing header file Controller_CodeSpecification1_types.h
### Writing header file Controller_CodeSpecification1.h
### Writing header file rtwtypes.h
.
### Writing header file Controller_Lp0dbbft.h
### Writing source file Controller_CodeSpecification1.c
### Writing header file Controller_CodeSpecification1_private.h
### Writing source file ert_main.c
### TLC code generation complete (took 8.15s).
### Saving binary information cache.
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
```



```
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\0\tp65d2e03e\sldv-ex41550386\IntelWin64\_sha
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\0\tp65d2e03e\sldv-ex41550386\IntelWin64\Cont
### Successful completion of code generation for: Controller_CodeSpecification1
```

The following files will be copied from IntelWin64\\_shared to C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\0\tp65d2e03e\sldv-ex41550386\IntelWin64\\_shared:

```
Controller_Lp0dbbft.c
Controller_Lp0dbbft.h
shared_file.dmr
```

Files copied from IntelWin64\\_shared to C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\0\tp65d2e03e\sldv-ex41550386\IntelWin64\\_shared:

6. If the library model is locked, unlock the library model to create a Simulink test harness for the subsystem block.

Create the harness for the subsystem block for a particular function interface. In this example, create the harness for the function interface `Double`.

7. Open the harness model and select the appropriate target and then start test generation.

**Note:** For RLS you can generate subsystem code from the library that gets compiled into a static library and can be reused by components. Test generation on the harness, created from the library and if you set the target as **Code Generated as Model Reference** you will receive an error message as this is not supported.

## See Also

### More About

- “Generate Reusable Code for Subsystems Shared Across Models” (Embedded Coder)
- “Test Library Blocks” (Simulink Test)

## Use Test Generation Advisor to Identify Analyzable Components

### In this section...

“Test Generation Advisor” on page 7-24

“Test Generation Advisor Requirements” on page 7-25

“Identify Analyzable Components” on page 7-25

“Analyze and Generate Tests for Model Components” on page 7-25

“Manually Select Components for Testing” on page 7-27

### Test Generation Advisor

You can use the Test Generation Advisor to select model components (atomic subsystems and model blocks) for test generation. The Test Generation Advisor summarizes test generation compatibility, condition and decision objectives, and dead logic for the model and model components.

The Test Generation Advisor performs a high-level analysis and fast dead logic detection. You can use the results to better understand your model before test generation, particularly for large models, complex models, or models for which you are uncertain of the test generation compatibility. For example, you can:

- Identify components that are incompatible with test case generation.
- Identify complex components that may be time-consuming to analyze.
- Determine instances of dead logic.
- Get a snapshot of the component hierarchy.
- Get recommended test generation parameters.

The screenshot shows the Test Generation Advisor interface. On the left, the Component Hierarchy tree is expanded to show the 'sldv\_testgen\_advisor' component, which contains subcomponents: 'Subsys\_Analysable', 'PI Controller', 'Subsys\_Complex', and 'Subsys\_Incompatible'. The main panel displays the 'Component Name: sldv\_testgen\_advisor' and an overall progress bar. Below the progress bar, it shows 'Components processed: 5/5'. A summary of subcomponents is provided in a table:

| Component Name   | Status | Objectives (Condition Decision) | Dead Logic Detected | Objectives Decided (%) |
|--|--------|---------------------------------|---------------------|------------------------|
| <a href="#">sldv_testgen_advisor</a>                                 | ✘      | 43                              | 11                  | NA                     |
| <a href="#">sldv_testgen_advisor/Subsys_Analysable</a>               | ✔      | 26                              | 11                  | 100%                   |
| <a href="#">sldv_testgen_advisor/Subsys_Analysable/PI Controller</a> | ✔      | 6                               | NA                  | NA                     |
| <a href="#">sldv_testgen_advisor/Subsys_Complex</a>                  | ⚠      | 15                              | 0                   | 40%                    |
| <a href="#">sldv_testgen_advisor/Subsys_Incompatible</a>             | ✘      | 2                               | NA                  | NA                     |

Below the table, it indicates: Incompatible: 2, Analyzable: 2, Complex: 1. A section titled 'Model items that are incompatible:' lists two messages:

- [sldv\\_testgen\\_advisor](#): Translation failed: Algebraic loops are not supported in generated code. Use the 'ashow' command in the Simulink Debugger to see the algebraic loops
- [sldv\\_testgen\\_advisor](#): Simulink Design Verifier failed to initialize: 'sldv\_testgen\_advisor/Subsys\_Incompatible' is incompatible for design error detection with Simulink Design Verifier.

The Test Generation Advisor classifies components as analyzable, complex, or incompatible.

- *Analyzable* components are compatible with Simulink Design Verifier. The preliminary analysis indicates that Simulink Design Verifier might achieve high component coverage.
- *Complex* components are also compatible with Simulink Design Verifier. However, the preliminary analysis indicates that Simulink Design Verifier might require more time and resources to achieve high component coverage due to component complexity or other factors. For more information, see “Sources of Model Complexity” on page 14-2.
- You cannot generate tests for *incompatible* components. For more information, see “Check Model Compatibility” on page 3-2.

The results summary displays specific information about the model and each component:


- **Status:** The compatibility or complexity
- **Objectives:** The number of condition and decision objectives
- **Dead Logic Detected:** The number of instances of dead logic decided during the analysis. This might not include every instance of dead logic.
- **Objectives Decided:** The percentage of condition and decision objectives determined by test cases and dead logic.

## Test Generation Advisor Requirements

For analysis, your model must compile. Also, if you change the model name, you must reload the model and reopen the Test Generation Advisor.

## Identify Analyzable Components

To analyze your model using the Test Generation Advisor, follow this high-level workflow:

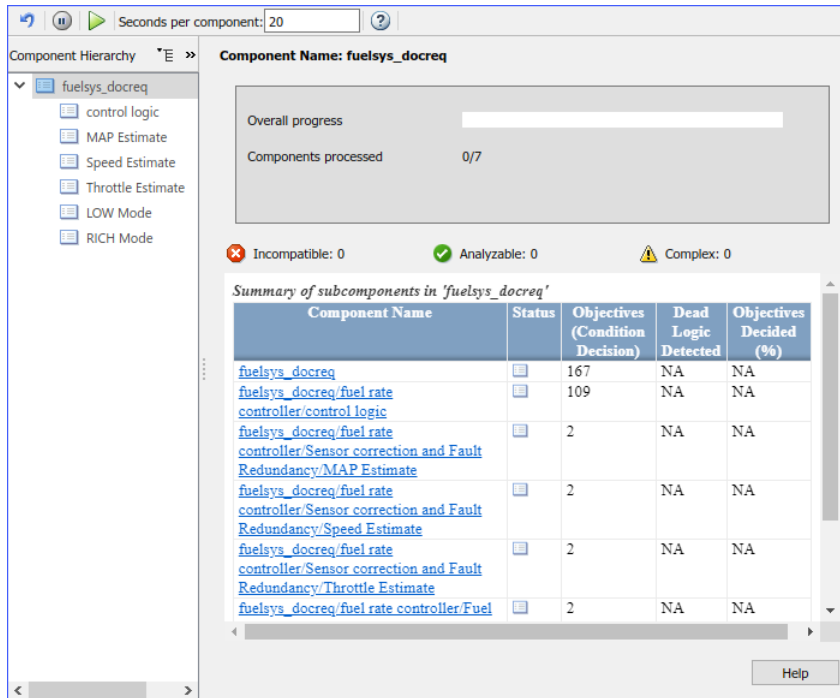
- 1 Open your model.
- 2 On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**, then click **Advisor**.
- 3 Your model compiles, and the Test Generation Advisor opens. It displays the model hierarchy and summary table.
- 4 Enter a time value for **Seconds per component**, which limits the analysis time per component. This value does not include time for other operations such as compilation.
- 5 Run the analysis by clicking the Start Analysis button . Track the analysis using the progress indicator.
- 6 Determine incompatibilities, complexities and characteristics from the component hierarchy tree and the results summary.
- 7 Trace from the summary to the model using the component hyperlinks.


## Analyze and Generate Tests for Model Components

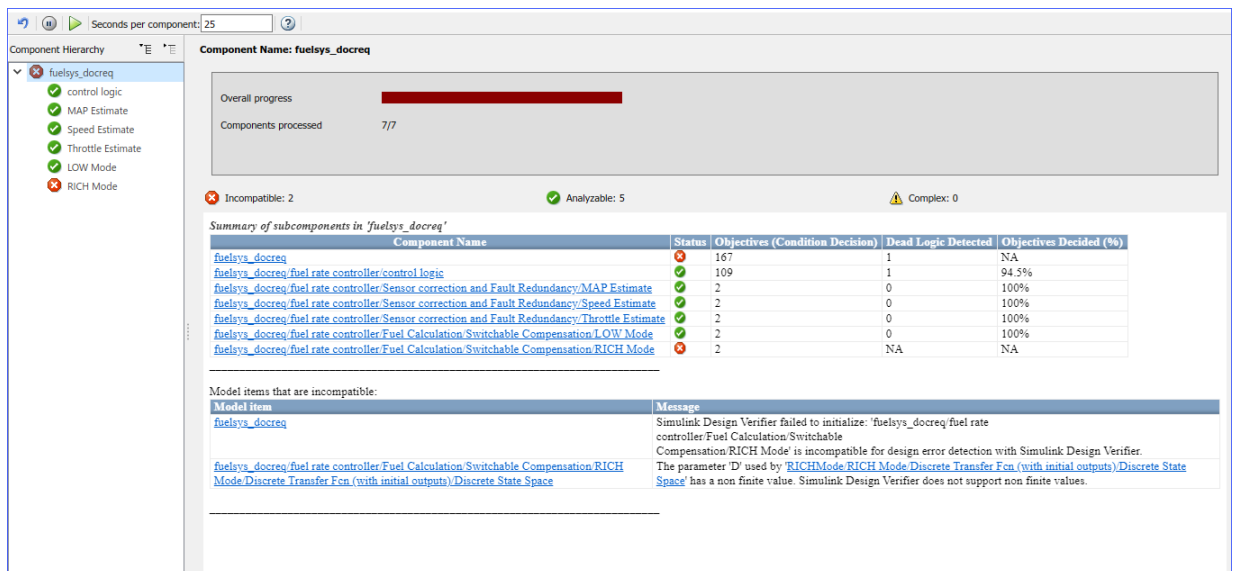
This example demonstrates analysis and test generation using the Test Generation Advisor. The example model has analyzable and incompatible subsystems.

- 1 At the command line, enter `fuelsys_docreq` to open the `fuelsys_docreq` model.

- 2 Save a copy of the model in a writable location on the MATLAB path.
- 3 On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**, then click **Advisor**.



- 4 In the **Seconds per component** text box, enter 25.
- 5 Click the Start Analysis button  to begin the model analysis.
- 6 After the analysis is complete, the component tree displays results for the overall model and each component.



- 7 Highlight the `control logic` subsystem in the component hierarchy. The analysis was partial, in that it determined 87% of the objectives for `control logic` by test cases and dead logic. To load the test generation summary, click the **Show test generation results summary** link.

At the bottom of the summary, the table lists recommended test generation parameters.

The screenshot displays the Test Generation Advisor interface for the component 'control logic'. The component hierarchy on the left shows 'fuelsys\_docreq' expanded to 'control logic', which is highlighted. The main panel shows the component name 'control logic' and an overall progress bar. Below this, it indicates 'Components processed: 7/7', 'Incompatible: 2', 'Analyzable: 5', and 'Complex: 0'. A table titled 'Summary of subcomponents in 'control logic'' shows one subcomponent: 'fuelsys\_docreq/fuel\_rate\_controller/control logic' with a status of 'Analyzable', 109 objectives, 1 dead logic detected, and 94.5% objectives decided. The 'Preliminary Test Generation Results' section states that 103 out of 109 objectives were decided. The 'Preliminary Dead Logic Detection' section notes that 1 objective is dead logic. A table lists a decision: 'fuelsys\_docreq/fuel\_rate\_controller/control logic/Fueling\_Mode/Fuel\_Disabled\_transition(#784)' with the description 'trigger expression false'. The 'Recommendations' table lists three options: 'Maximum analysis time(seconds)' at 300, 'Automatic stubbing of unsupported atomic blocks' set to 'on', and 'Testsuite generation strategy' set to 'Auto'. At the bottom right, there is a button labeled 'Extract this component and generate tests' and a 'Help' button.

- 8 Click the **Component name** hyperlink. Simulink traces to the `control logic` Stateflow chart.
- 9 Generate the full set of tests for the subsystem. In the Test Generation Advisor summary for `control logic`, click **Extract this component and generate tests**.

## Manually Select Components for Testing

If you know which model components that you want to test, you can manually select these components. Break down the model into components of 100–1000 objectives each. Use the `sldvextract` function to extract components into a new model. You can then analyze the individual components, starting with the lowest-level subsystems.

## See Also

### More About

- “Model Coverage Objectives for Test Generation” on page 7-30
- “Generate Test Cases for Model Decision Coverage” on page 7-6

## Generate Test Cases for Embedded Coder Generated Code

### In this section...

“Generate Test Cases for Generated Code from the Simulink Model Toolstrip” on page 7-28

“Generate Test Cases for Generated Code by Using the Simulink Design Verifier API” on page 7-29

“Generate Test Cases for Generated Code from the Simulink Test Test Manager” on page 7-29

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

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

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

### Generate Test Cases for Generated Code from the Simulink Model Toolstrip

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

- 1 On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**.
  - To generate tests for code generated as top model, select **Target > Code Generated as Top Model**, then click **Generate Tests**.
  - To generate tests for code generated as model reference, select **Target > Code Generated as Model Reference**, then click **Generate Tests**.

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

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

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

- “Choose a SIL or PIL Approach” (Embedded Coder)

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

For an example of how to programmatically generate test cases for generated code, see “Code Coverage Test Generation” on page 7-111.

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

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

## **See Also**

### **More About**

- “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-28

## Model Coverage Objectives for Test Generation

| In this section...                 |
|------------------------------------|
| “Decision” on page 7-30            |
| “Condition” on page 7-30           |
| “MCDC” on page 7-31                |
| “Enhanced MCDC” on page 7-31       |
| “Relational Boundary” on page 7-31 |

Test cases are generated to drive your model to satisfy condition, decision, modified condition/decision (MCDC), and custom coverage objectives. But, if your model does not have any of these objectives, then Simulink Design Verifier generates a test case that represents a basic simulation of your model. The test inputs satisfy minimum or maximum constraints on input ports and intermediate signal values satisfy constraints specified by the Test Condition blocks in the model.

### Decision

Decision coverage in Simulink Design Verifier examines blocks and Stateflow states that represent decision points in a model. For instance, the Switch block involves the decision about whether the control input is greater than a threshold value. For more information, see “Model Objects That Receive Coverage” (Simulink Coverage).

To enable decision coverage, under **Design Verifier > Test Generation**, for **Model coverage objectives**, select one of the following:

- Decision
- Condition Decision
- MCDC

For each decision in your model, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see “Decision Coverage (DC)” (Simulink Coverage).

### Condition

Condition coverage examines blocks that output the logical combination of their inputs and Stateflow transitions. For more information, see “Model Objects That Receive Coverage” (Simulink Coverage).

To enable condition coverage, under **Design Verifier > Test Generation**, for **Model coverage objectives**, select one of the following:

- Condition Decision
- MCDC

For each input to a logical block and each condition in a transition, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see “Condition Coverage (CC)” (Simulink Coverage).



## MCDC

Modified condition decision coverage examines blocks that output the logical combination of their inputs and Stateflow transitions. For more information, see “Model Objects That Receive Coverage” (Simulink Coverage).

To enable MCDC coverage, under **Design Verifier > Test Generation**, for **Model coverage objectives**, select MCDC.

For each input to a logical block and each condition in a transition, Simulink Design Verifier generates test cases that satisfy the coverage objective. For more information, see “MCDC Coverage for Stateflow Charts” (Simulink Coverage).

For information on how MCDC test generation in Simulink Design Verifier can deviate from MCDC coverage recorded by Simulink Coverage, see “Modified Condition and Decision Coverage in Simulink Design Verifier” on page 9-21.

## Enhanced MCDC

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

To enable enhanced MCDC coverage, under **Design Verifier > Test Generation**, for **Model coverage objectives**, select Enhanced MCDC. For more information, see “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42.

## Relational Boundary

Relational boundary coverage examines blocks that have an explicit or implicit relational operation and Stateflow transitions. For more information, see “Model Objects That Receive Coverage” (Simulink Coverage). Test generation for relational boundary coverage is not supported for If and Fcn blocks.

To enable relational boundary coverage, under **Design Verifier > Test Generation**, select **Include relational boundary objectives**.

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

---

**Note** In case your model does not have conditions, decisions, or custom test objectives, then Simulink Design Verifier will generate a test case that represents a basic simulation of your model. The test inputs will satisfy min/max constraints on input ports and intermediate signal values will satisfy constraints specified by the Test Condition blocks in the model.

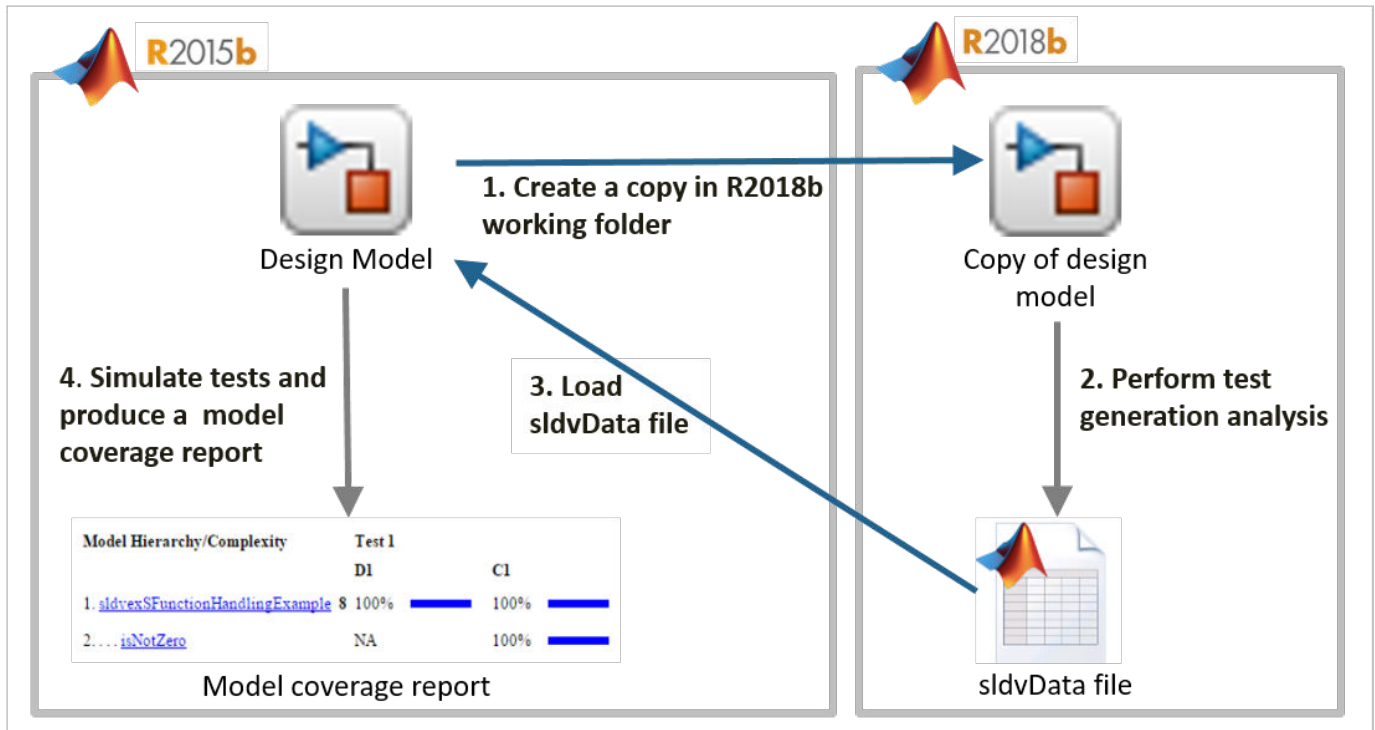
---

## Enhance Model Coverage of Older Release Models

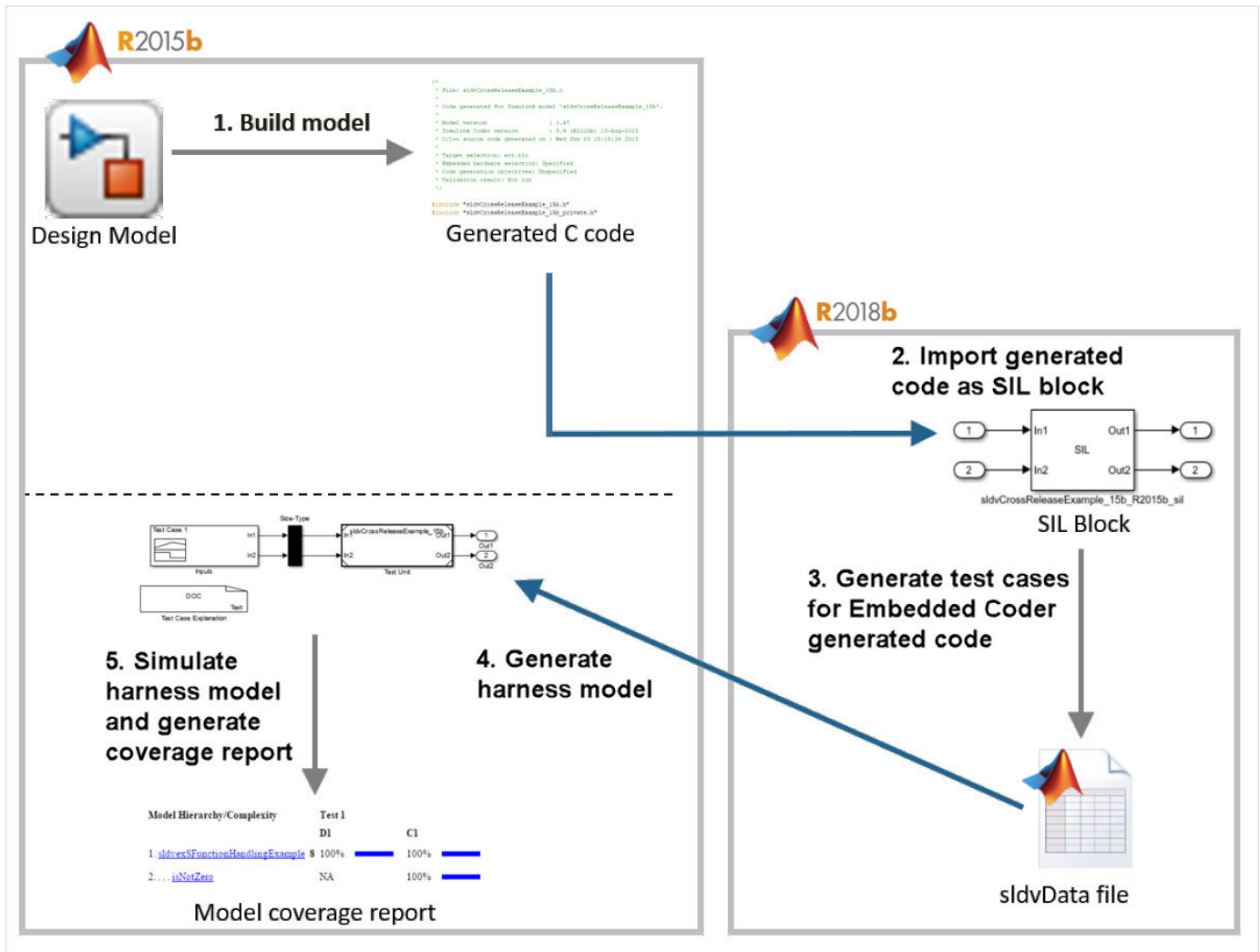
To enhance the model coverage of a model that you created in an older release, use a test generation workflow or a code generation workflow. You can leverage the latest release capabilities of Simulink Design Verifier to generate the test cases for a Model-Based Design.

These workflows enhance model coverage.

- “Enhance Model Coverage by Generating Test Cases for Older Release Model” on page 7-33



- “Enhance Model Coverage by Using Generated Code from Older Release” on page 7-37



## Enhance Model Coverage by Generating Test Cases for Older Release Model

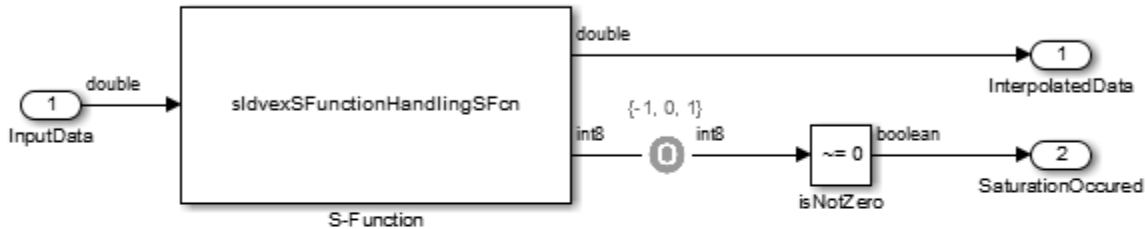
This example shows how to upgrade model coverage of a model created in R2015b. You use test generation for supported S-functions available in the latest release.

The example model `sldvexSFunctionHandlingExample` contains the handwritten S-Function, which implements a lookup table algorithm. The handwritten S-Function is in the file `sldvexSFunctionHandlingSFcn.c`. The user source code for the lookup table is in the file `sldvexSFunctionHandlingSource.c`.

1. In MATLAB R2015b, open the `sldvexSFunctionHandlingExample` model.

```
open_system('sldvexSFunctionHandlingExample');
```

## Simulink Design Verifier S-Function Handling for Test Generation



This model contains a handwritten S-Function which implements a lookup table algorithm. The S-Function block returns the interpolated value at the first output port and returns the status of the interpolation at the second output port. The second output port returns the value -1 if a lower saturation occurs, 1 if a upper saturation occurs, and 0 otherwise.

**Open  
S-Function sources  
(double-click)**

Open Source Files

**Run  
(double-click)**

Run Simulink Design Verifier

**View Options  
(double-click)**

View Simulink Design Verifier Options

2. To simulate the model and generate the coverage report, in the Simulink Editor, click the Run button. See “View Coverage Results in Simulink Canvas” (Simulink Coverage) .

After the simulation, the coverage report indicates that full coverage is not achieved for sldvexSFunctionHandlingExample model.

### Summary

| Model Hierarchy/Complexity                        | Test 1 |     |           |
|---|--------|-----|-----------|
|   | DI     | CI  | Execution |
| 1. <a href="#">sldvexSFunctionHandlingExample</a> | 8 13%  | 50% | 100%      |
| 2. ... <a href="#">isNotZero</a>                  | NA     | 50% | 100%      |

3. In MATLAB R2018b or later releases, open the sldvexSFunctionHandlingExample model. The example model sldvexSFunctionHandlingExample is available in R2015b and later releases, so you can use the same model for test generation workflow.

```
open_system('sldvexSFunctionHandlingExample');
```

To avoid any potential changes in the model, create a copy of the older release model in the current working folder, and then open the model in R2018b or later releases. To upgrade and improve models that you use in the current release, you can use the `upgradeadvisor` function. See “Programmatically Analyze and Upgrade Model”.

4. Compile the S-function to be compatible with Simulink Design Verifier for test case generation by using `sldcovmex` (Simulink Coverage). For more information, see “Configuring S-Function for Test Case Generation” on page 7-109.

```
sldcovmex('-sldv', ...
          '-output', 'sldvexSFunctionHandlingSFcn',...
          'sldvexSFunctionHandlingSource.c','sldvexSFunctionHandlingSFcn.c');
```

```
mex C:\TEMP\Bdoc23a_2213998_3568\ib570499\0\tp5f4630b4_5f10_43a2_a702_c33a560effc4\tpd5aa63d5_e5
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
mex sldvexSFunctionHandlingSource.c C:\TEMP\Bdoc23a_2213998_3568\ib570499\0\tp5f4630b4_5f10_43a2
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
```

5. Create an `opts` option for the `sldvexSFunctionHandlingExample` model.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'Condition';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
opts.SFcnSupport = 'on';
```

6. To generate test cases by using the specified `opts` options, use `sldvrun` to analyze the model.

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

```
03-Mar-2023 23:40:52
Checking compatibility for test generation: model 'sldvexSFunctionHandlingExample'
Compiling model...done
Building model representation...done
```

```
03-Mar-2023 23:41:24
```

```
'sldvexSFunctionHandlingExample' is compatible for test generation with Simulink Design Verifier
```

```
Generating tests using model representation from 03-Mar-2023 23:41:24...
```

```
Generating output files:
```

```
03-Mar-2023 23:41:47
Results generation completed.
```

```
Data file:
```

```
C:\TEMP\Bdoc23a_2213998_3568\ib570499\0\tp65d2e03e\sldv-ex67693772\sldv_output\sldvexSFunction
```

After analysis, the software generates a Simulink Design Verifier data file and stores it in the default location `<current_folder>\sldv_output\sldvexSFunctionHandlingExample_sldvdata.mat`.

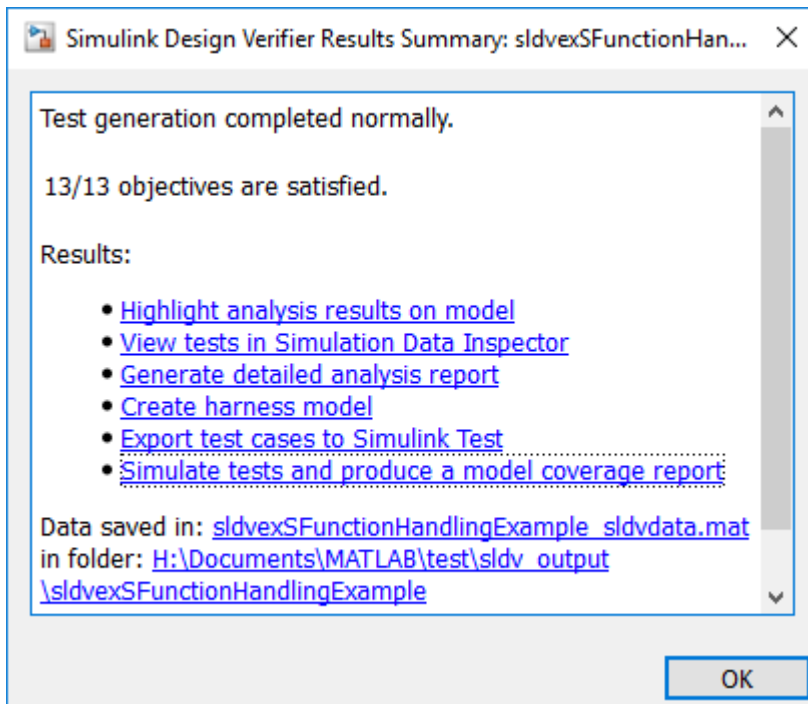
7. In R2015b, open the model.

```
open_system('sldvexSFunctionHandlingExample');
```

8. Load the sldvData file created in R2018b or later releases.





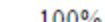

a. On the **Design Verifier** tab, click **Load Earlier Results** and browse to the sldvData MAT-file generated in R2018b or later releases.

b. Click **Open**.



9. In the Simulink Design Verifier Results Summary window, click **Simulate tests** and produce a model coverage report. The report indicates that 100% coverage is achieved for sldvexSFunctionHandlingExample model.

## Summary

| Model Hierarchy/Complexity                        | Test 1 |  | C1   | Test Objective   | Execution  |
|---|--------|--|--|--|--|
|   | DI     |  |  |  |  |
| 1. <a href="#">sldvexSFunctionHandlingExample</a> | 8      | 100%  | 100%    | 100%  | 100%  |
| 2. ... <a href="#">isNotZero</a>                  |        | NA   | 100%  | NA   | 100%  |

For more information, see “Manage Simulink Design Verifier Data Files” on page 13-7 and “Simulate Tests and Produce Model Coverage Report” on page 1-15.

## Enhance Model Coverage by Using Generated Code from Older Release

This example shows how to upgrade the model coverage of a model created in R2015b by using code generation workflow.

For this workflow, you must have Simulink Coder™ and Embedded Coder.

The example model `sldvCrossReleaseExample` contains the handwritten S-Function, which implements a relational boundary algorithm. The handwritten S-Function is in the file `rel_sfcn.c`. The user source code is in the file `rel_comp.c`.

To inline the S-function, use the `rel_sfcn.tlcfile`. For more information, see “Inline S-Functions with TLC” (Embedded Coder).

- 1 Copy the example model `sldvCrossReleaseExample` and S-Function files, `rel_sfcn.c`, `rel_comp.c`, and `rel_sfcn.tlc` in the current working folder. Copy the header files `rel_comp.h` into the current working folder. You use the example model and supporting files in R2015b for a “Cross-Release Code Integration” (Embedded Coder) workflow.

---

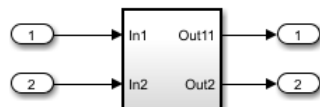
**Note** The example model `sldvCrossReleaseExample` is created for example purpose. To perform code generation workflow by using the example model, export `sldvCrossReleaseExample` model to 15b. Save the model as `sldvCrossReleaseExample_15b` in the current working folder. For more information, see “Export Model to Previous Version of Simulink”.

---

- 2 In MATLAB R2015b, open `sldvCrossReleaseExample_15b` model from the current working folder.

```
open_system('sldvCrossReleaseExample_15b');
```

### Simulink Design Verifier Enhance Model Coverage by Using Code Generation Workflow



The Subsystem block contains a handwritten S-Function which implements a relational boundary algorithm. The S-function block returns an output value in 100-200 range.

- 3 Compile the S-function by using the function `legacy_code`.

```
def = legacy_code('initialize');
def.SFunctionName = 'rel_sfcn';
def.OutputFcnSpec = 'uint8 y1 = relational_bound(uint8 u1)';
def.HeaderFiles = {'rel_comp.h'};
def.SourceFiles = {'rel_comp.c'};
def.IncPaths = {pwd};
def.SrcPaths = {pwd};
def.Options.supportCoverageAndDesignVerifier = true;
```

```
legacy_code('sfcn_cmex_generate', def);
legacy_code('compile', def);
```

- 4 To simulate the model and generate the coverage report, in the Simulink Editor, click the **Run** button. See “View Coverage Results in Simulink Canvas” (Simulink Coverage).

After the simulation, the coverage report indicates that 50% coverage is achieved for sldvCrossReleaseExample\_15b model.

## Summary

| Model Hierarchy/Complexity                     | Test 1 |           |
|--|--------|-----------|
|  | DI     | Execution |
| 1. <a href="#">sldvCrossReleaseExample_15b</a> | 6 50%  | 100%      |
| 2. ... <a href="#">Subsystem</a>               | 5 50%  | 100%      |

- 5 To generate code using Embedded Coder, from the **Apps** tab, select **Embedded Coder**. For more information, see “Generate Code Using Embedded Coder” (Embedded Coder).

In the **C Code** tab, click **Generate Code**.

The model is preconfigured with these code generation settings.

```
set_param(sldvCrossReleaseExample_15b, 'SystemTargetFile', 'ert.tlc');
set_param(sldvCrossReleaseExample_15b, 'PortableWordSizes', 'on');
set_param(sldvCrossReleaseExample_15b, 'SupportNonFinite', 'off');
set_param(sldvCrossReleaseExample_15b, 'GenCodeOnly', 'on');
set_param(sldvCrossReleaseExample_15b, 'SolverMode', 'SingleTasking');
set_param(sldvCrossReleaseExample_15b, 'ProdEqTarget', 'on');
```

The software generates C code for the model and saves the files in the default folder location <current\_folder>\sldvCrossReleaseExample\_15b\_ert\_rtw.

- 6 Save the configuration set of the model sldvCrossReleaseExample\_15b to a MAT-file. This ConfigSet is used to set the configuration set of the model in R2018b and later releases.

```
config_set = getActiveConfigSet('sldvCrossReleaseExample_15b');
copiedConfig = config_set.copy;
save('copiedConfig.mat', 'copiedConfig');
```

- 7 In MATLAB R2018b or later releases, import the components exported from R2015b.

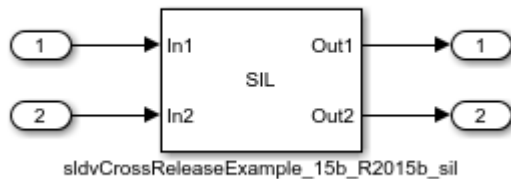
- a Before you import components in current release, rename or delete `rtwtypes.h` file available in the folder <current\_folder>\sldvCrossReleaseExample\_15b\_ert\_rtw. During cross-release import, MATLAB tries to regenerate a file with same name. If you do not delete or rename the file `rtwtypes.h`, MATLAB displays an error.
- b Import the generated component code from R2015b as software-in-the-loop (SIL) block.

```
crossReleaseImport('sldvCrossReleaseExample_15b_ert_rtw', ...
'sldvCrossReleaseExample_15b', 'SimulationMode', 'SIL');
```

The `crossReleaseImport` function creates an untitled model that contains software-in-the-loop (SIL) block `sldvCrossReleaseExample_15b_R2015b_sil`.



- 8 Add Inport and Outport ports to the `sldvCrossReleaseExample_15b_R2015b_sil` block and save the model as `sldvCrossReleaseExample_sil_18b`.



- 9 Apply the model configuration set similar to R2015b model.

```
load('copiedConfig.mat');
attachConfigSet('sldvCrossReleaseExample_sil_18b', copiedConfig, true);
setActiveConfigSet('sldvCrossReleaseExample_sil_18b', copiedConfig.Name);
```

- 10 Set the simulation mode to Software-in-the-Loop (SIL). To simulate the model, in the Simulink Editor, click the **Run** button.
- 11 To generate test cases for Embedded Coder generated code, on the **Design Verifier** tab, select **Target > Code Generated as Top Model** and click **Generate Tests**. For more information, see “Generate Test Cases for Embedded Coder Generated Code” on page 7-28.

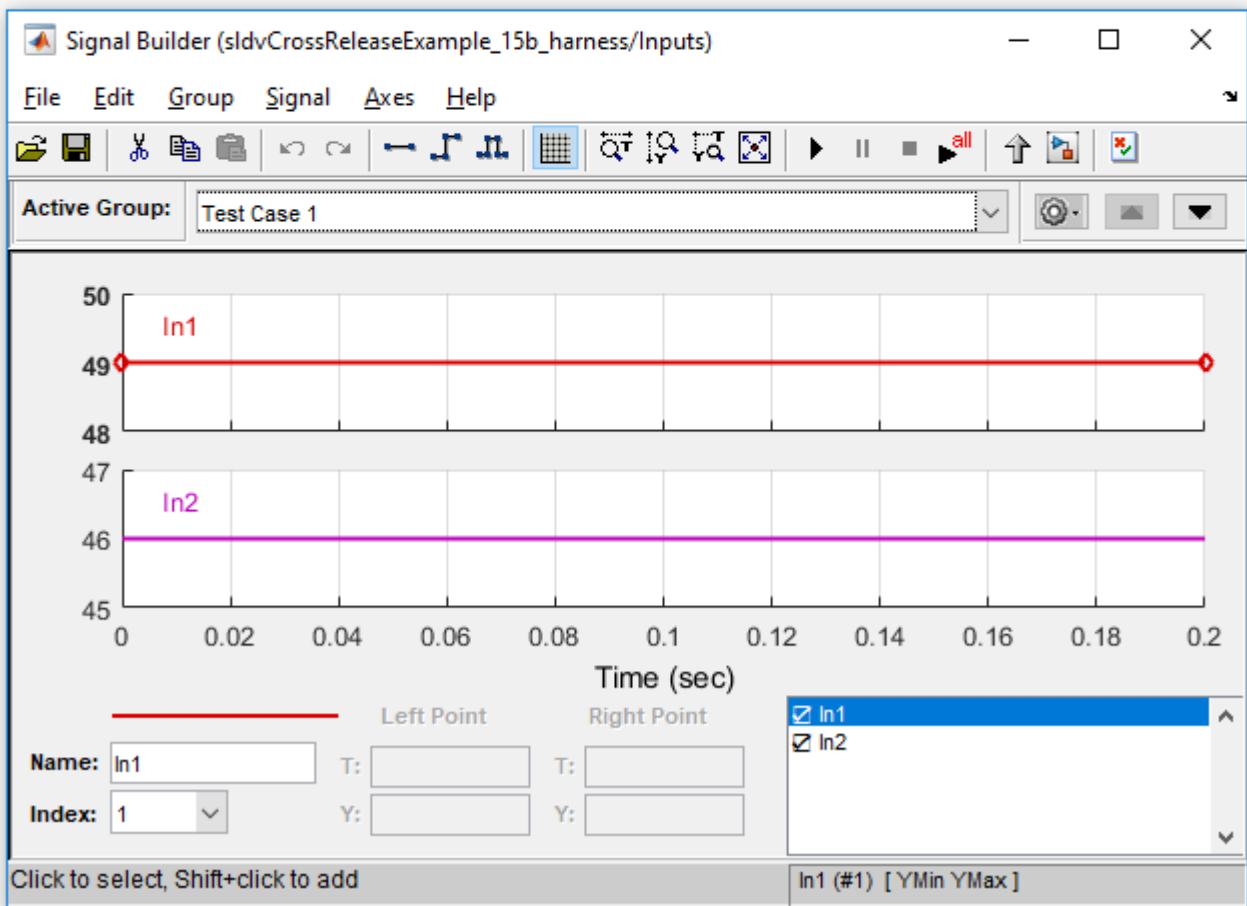
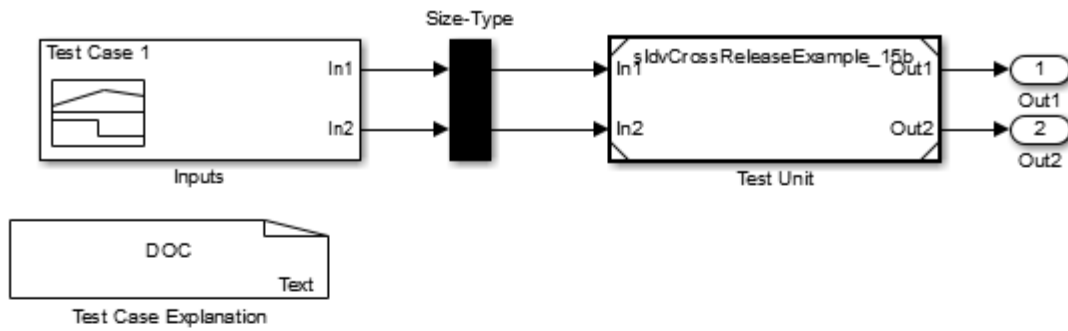
After Simulink Design Verifier analysis, the software generates the test cases and saves the `sldvData` in folder at default location `<current_folder>\sldv_output\sldvCrossReleaseExample_sil_18b`.


- 12 In R2015b, open the model.

```
open_system('sldvCrossReleaseExample_15b');
```

- 13 Update the `sldvData.ModelInformation.Name` field in `sldvData` same as the model name in older release. For example, `sldvCrossReleaseExample_15b.slx`.
- 14 Create a harness model by using the `sldvData` created in R2018b or later releases. This data consists of test cases generated from Embedded Coder generated code. In the `dataFile`, type the location of the `sldvData` generated for `sldvCrossReleaseExample_sil_18b` model.





```
sldvmakeharness('sldvCrossReleaseExample_15b.slx', 'dataFile')
```



15 To simulate the model by using all the test cases, click the **Run all** button .

The software simulates all the test cases and generates a coverage report. The report indicates that 100% coverage is achieved for sldvCrossReleaseExample\_15b model.

## Summary

| Model Hierarchy/Complexity                     | Test 1   |  |
|--|--|--|
|  | DI   | Execution  |
| 1. <a href="#">sldvCrossReleaseExample_15b</a> | 6 100%  | 100%  |
| 2... <a href="#">Subsystem</a>                 | 5 100%  | 100%  |

## See Also

### More About

- “Generate Test Cases for Embedded Coder Generated Code” on page 7-28
- “Cross-Release Code Integration” (Embedded Coder)
- “Manage Simulink Design Verifier Data Files” on page 13-7
- “Manage Simulink Design Verifier Harness Models” on page 13-13

## Enhanced MCDC Coverage in Simulink Design Verifier

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

To detect the effect of a test block by using the enhanced MCDC coverage objective, you can consider a standard model coverage objective of a test block or you can author your own custom test objectives for analysis. For more information, see:

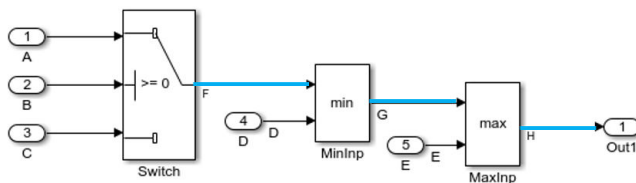
- Use Model Coverage Objectives for Enhanced MCDC Coverage on page 7-42
- Author Custom Test Objectives for Enhanced MCDC Coverage on page 7-43

To generate test cases by using enhanced MCDC model coverage objectives, and then analyzing the results, see Basic Workflow for Enhanced MCDC Analysis on page 7-47.

### Use Model Coverage Objectives for Enhanced MCDC Coverage

For a given test block, you can detect the effect on a model coverage objective from the downstream blocks. When you generate test cases by using enhanced MCDC model coverage objectives, the generated test cases avoid the masking effect from the downstream blocks. The model coverage objective is detectable at a detection site.

Consider this model that consists of a cascade of Switch, Min, and Max blocks.



The test cases generated for enhanced MCDC coverage ensure that the decision objective of the “Switch” (Simulink Coverage) test block is not masked by the downstream Min and Max blocks. The generated test cases ensure that these nonmasking conditions for Min and Max blocks are satisfied:

- 1  $F < D$ , ensures that the Min block does not mask the Switch output.
- 2  $G > E$ , ensures that the Max block does not mask the Min output.

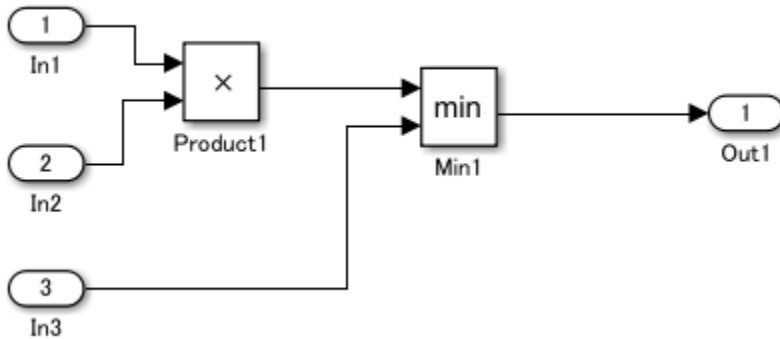
The decision objective of the Switch block and the nonmasking conditions of the Min and Max blocks are satisfied along the path and are detected at the detection site (Out1). For example, the path starts from the output signal of the Switch block, propagates along the Min block, and ends at the output signal of the Max block (highlighted in the example model).

Enhanced MCDC criteria ensure better quality test cases because the test case detects the effect of a model coverage objective of the test block at the detection site. To analyze a model for enhanced MCDC analysis, see example “Analyze Model for Enhanced MCDC Analysis” on page 7-44.

## Author Custom Test Objectives for Enhanced MCDC Coverage

Enhanced MCDC considers the default coverage objectives of a test block that are detectable at the detection site. To check the detectability status of a custom test objective, you can author the test objective of a model object, and then perform enhanced MCDC analysis.

Consider this model that consists of a Product block and a Min block. The Product block does not have a coverage objective.



You can author a custom test objective for the Product block to render the output value less than 0 and detect the effect of the custom test objective at a detection site.

For more information, see Author Custom Test Objective Workflow on page 7-52.

### See Also

### More About

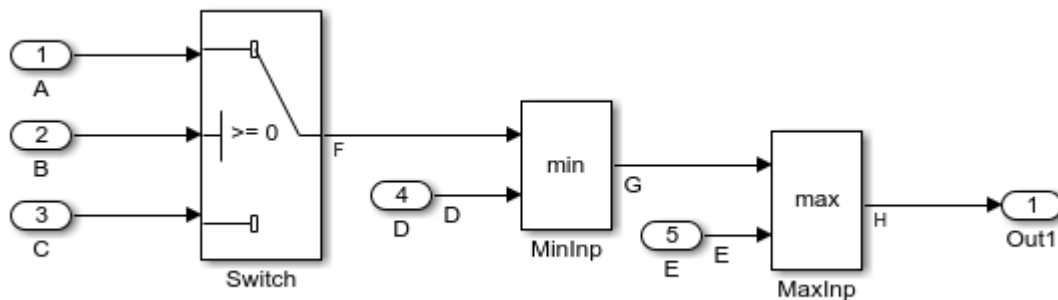
- “Model Coverage Objectives for Test Generation” on page 7-30
- “Design Verifier Pane: Test Generation” on page 15-30

## Analyze Model for Enhanced MCDCC Analysis

This example shows how to generate test cases for enhanced Modified Condition Decision Coverage (MCDCC) objectives. You generate test cases for enhanced MCDCC coverage objectives and review analysis results. The `sldvEnhancedMCDCCExample` model consists of Switch, Min, and Max blocks.

1. Open the model `sldvEnhancedMCDCCExample`:

```
sldvEnhancedMCDCCExample;
```



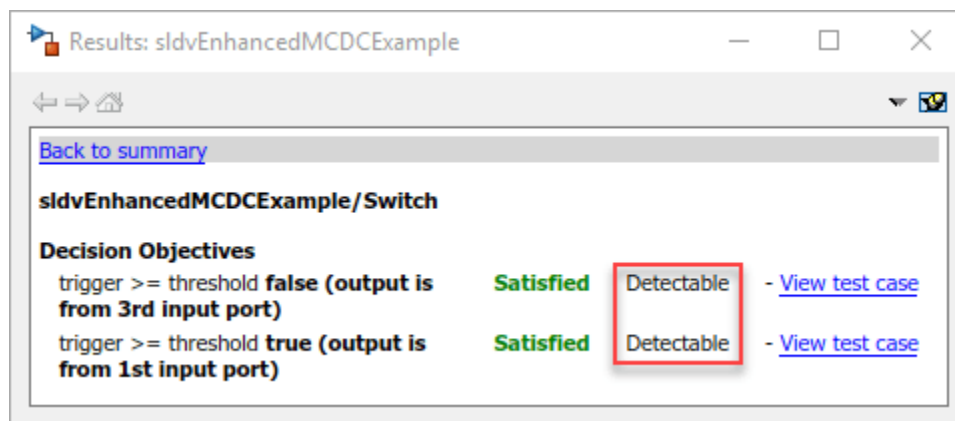
2. To configure the model for Enhanced MCDCC objectives, in the Configuration Parameters dialog box, on the **Design Verifier > Test generation pane**, set **Model coverage objectives** to Enhanced MCDCC. Click **OK**.

3. To generate test cases, on the **Design Verifier** tab, click **Generate Tests**.

After the analysis is completed, the Results Summary window displays the processed objectives and options to review the results.

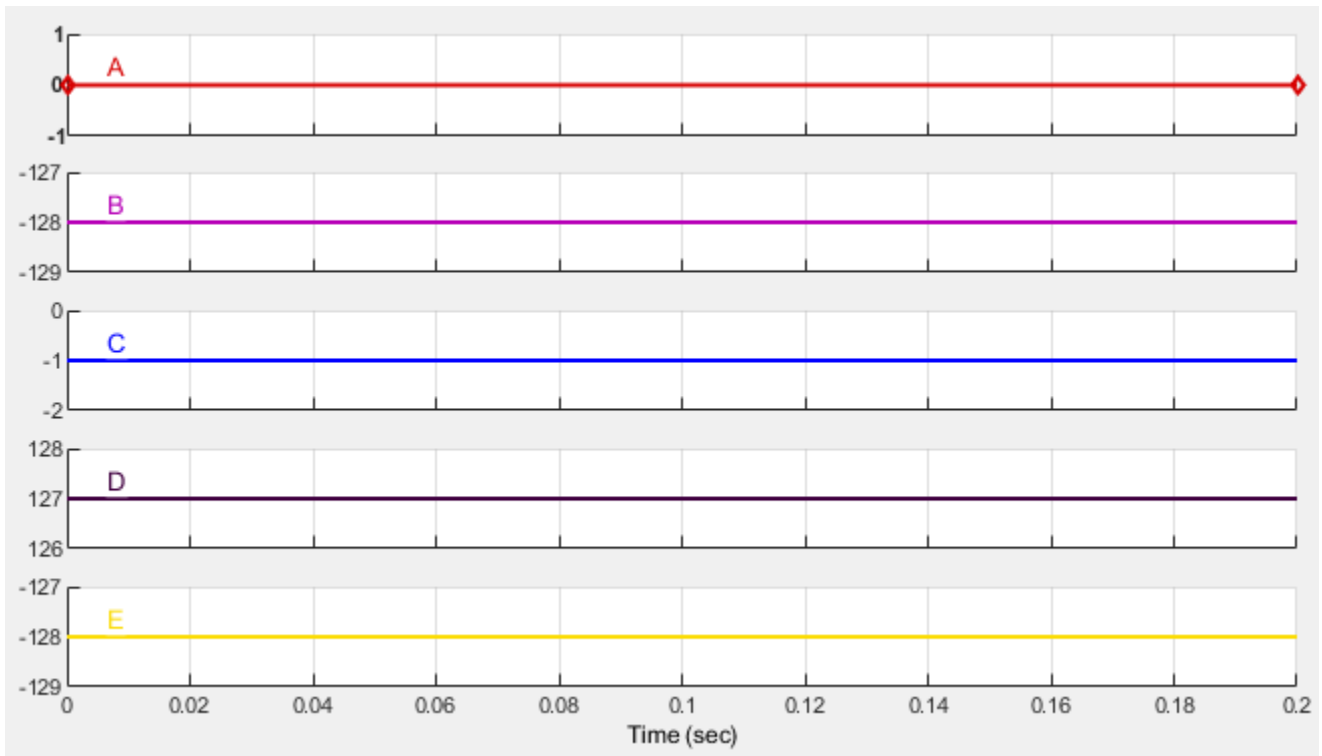
4. To highlight the analysis results, click **Highlight analysis results on model**.

To analyze whether the model coverage objectives of the Switch test block are detectable, click the Switch block.



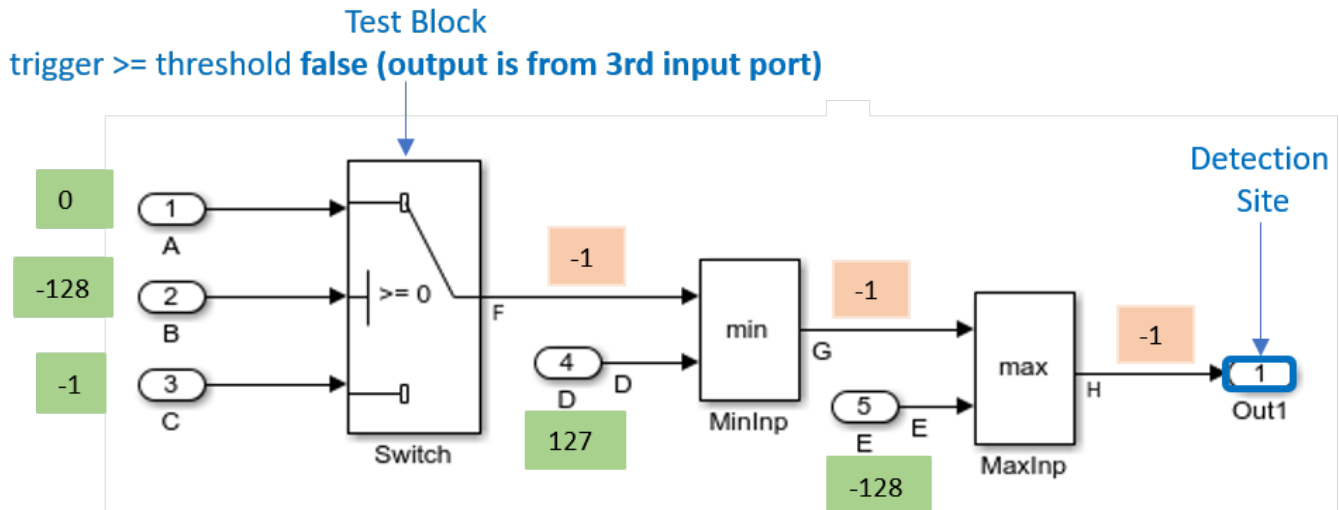
The results show that the decision objectives of the Switch block are detectable.

5. Click **View test case**. The harness model opens and the Signal Builder block displays Test case 4.



You can also view the test cases from the detailed analysis report.

|               |      |
|---------------|------|
| <b>Time 0</b> |      |
| <b>Step 1</b> |      |
| A             | 0    |
| B             | -128 |
| C             | -1   |
| D             | 127  |
| E             | -128 |



The test case inputs A, B, and C result in  $F = -1$  and  $G = -1$ . The value of  $E = -128$  results in  $H = -1$ , so the impact of the test objective is detected at the detection site Out1. The impact of the model coverage objective of the test block is not masked along the path and is detectable at Out1.

6. To view the detailed analysis report, click **HTML** in the Results Summary. The Test Objectives Status section lists the satisfied objectives. The coverage objective that is detectable at the detection site is summarized in the table.

## Objectives Satisfied

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

| # | Type     | Model Item             | Description  | Detection Status | Analysis Time (sec) | Test Case         |
|---|----------|------------------------|--|------------------|---------------------|-------------------|
| 1 | Decision | <a href="#">Switch</a> | trigger $\geq$ threshold false (output is from 3rd input port) | Detectable       | 32                  | <a href="#">4</a> |
| 2 | Decision | <a href="#">Switch</a> | trigger $\geq$ threshold true (output is from 1st input port)  | Detectable       | 33                  | <a href="#">5</a> |
| 3 | Decision | <a href="#">MinInp</a> | Logic to determine output input 1 is the minimum               | Detectable       | 31                  | <a href="#">2</a> |
| 4 | Decision | <a href="#">MinInp</a> | Logic to determine output input 2 is the minimum               | Detectable       | 32                  | <a href="#">3</a> |
| 5 | Decision | <a href="#">MaxInp</a> | Logic to determine output input 1 is the maximum               | Detectable       | 31                  | <a href="#">2</a> |
| 6 | Decision | <a href="#">MaxInp</a> | Logic to determine output input 2 is the maximum               | Detectable       | 2                   | <a href="#">1</a> |

The Objectives field in the Simulink Design Verifier data files lists the detectability status and the detection sites for the model coverage objectives. For more information, see “Manage Simulink Design Verifier Data Files” on page 13-7.

### See Also

- “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42



# Basic Workflow for Enhanced MCDC Analysis

To generate test cases for enhanced Modified Condition Decision Coverage (MCDC) coverage objectives:

- 1 On the **Design Verifier** tab, in the **Mode** section, select **Test Generation**.
- 2 Click **Test Generation Settings**.
- 3 In the Configuration Parameters dialog box, on the **Design Verifier > Test Generation** pane, set **Model coverage objectives** to Enhanced MCDC. Click **OK**.
- 4 Click **Generate Tests**.

---

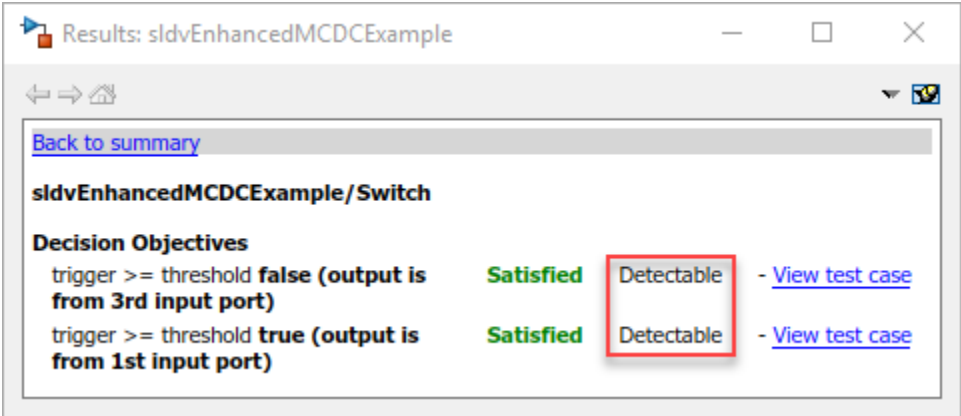
**Note** Enhanced MCDC analysis is not supported when you “Generate Test Cases for Embedded Coder Generated Code” on page 7-28. The software considers MCDC coverage objectives for test generation analysis.

---

Simulink Design Verifier analyzes the model for Enhanced MCDC coverage objectives.

After the analysis is complete:

- The software highlights the model with the analysis results.
- The Results Inspector window displays the summary of the model coverage objectives including the detectability status.



The Results Inspector window displays these detectability statuses for a model coverage objective:

- Detectable
- Not Detectable
- Undecided

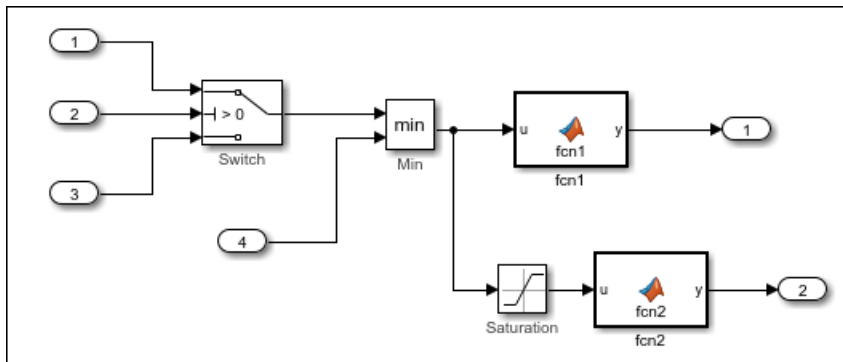
The table lists the possible combinations of the objective status and the detectability statuses.

| Objective Status             | Detectability Status | Test Case Description   |
|------------------------------|----------------------|---|
| Satisfied                    | Detectable           | The test case satisfies the model coverage objective and is detectable at the detection site.   |
| Satisfied - Needs Simulation | Detectable           | The test case satisfies the model coverage objective and is detectable at the detection site.<br><br>To confirm the satisfied status, you must run additional simulations of test cases. For more information, see “Objectives Satisfied - Needs Simulation” on page 13-46. |
| Satisfied                    | Not detectable       | The test case satisfies the model coverage objective. However, the test objective is not detectable at any detection site.  |
| Satisfied                    | Undecided            | The test case satisfies the model coverage objective. The software is unable to show the effect of model coverage objective on the downstream blocks.   |
| Unsatisfiable                | Not Detectable       | The test objective is unsatisfiable and not detectable at any detection site.   |
| Undecided                    | Undecided            | The test objective is undecided and the software is unable to show its effect on the downstream blocks.   |

- The Simulink Design Verifier data file stores the detectability status and detection site for model coverage objectives. For more information see, “Manage Simulink Design Verifier Data Files” on page 13-7.

## Configure Detection Sites using Test-pointed Logged Signals

If you mark any signal as test-pointed logged signal, Enhanced MCDC analysis will prioritize such signals as detection sites for test blocks wherever possible. For example, consider the model shown below:



If you make the output of Min block as the test-pointed logged signal, the detection site for the switch block is min block's output. Otherwise, it would be saturation block's output.

```
portHandle_MinBlk = get_param('model/Min', 'PortHandles');
set_param(portHandle_MinBlk.Output, 'TestPoint', 'on');
set_param(portHandle_MinBlk.Output, 'DataLogging', 'on');
```

For more information on test points, see “Configure Signals as Test Points”. For signal logging, refer to “Configure Signals for Logging”.

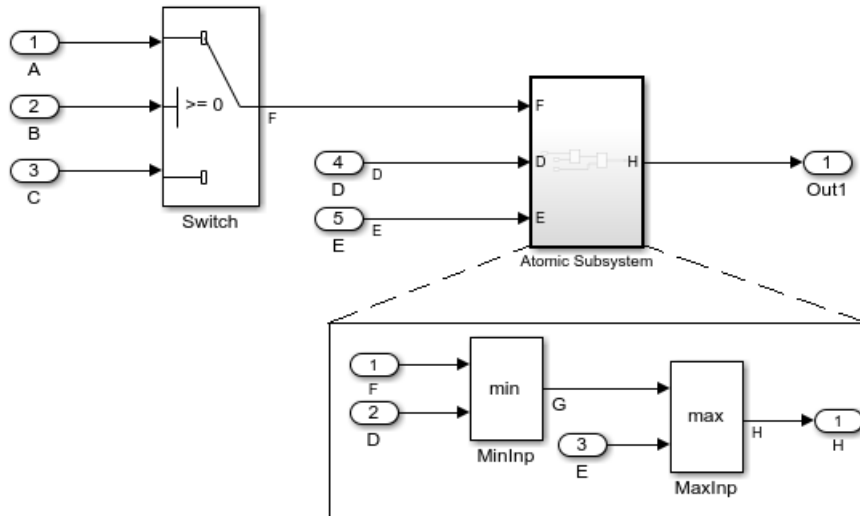
## Configure Advanced Options for Enhanced MDC Analysis

To analyze a model with stricter nonmasking conditions, enable the “Use strict propagation conditions” on page 15-37 option. This option is available in the Configuration Parameters dialog box, on the **Design Verifier > Test Generation** pane, in **Advanced parameters**.

The software evaluates stricter nonmasking conditions to analyze the effect on the test block from the downstream blocks. For example:

- If your model consists of Atomic Subsystem with the Function packaging option set to **Auto** or **Inline**.

Consider a model that consists of Switch and Atomic Subsystem blocks. The Function packaging option is set to **Auto** and you enable the “Use strict propagation conditions” on page 15-37 option. The effect of the Switch test block is detectable at the detection point Out1.



When you analyze the model with the “Use strict propagation conditions” on page 15-37 option set to Off, the software analyzes the model until the effect of the Switch test block reaches the Atomic Subsystem. The Atomic Subsystem is the detection point.

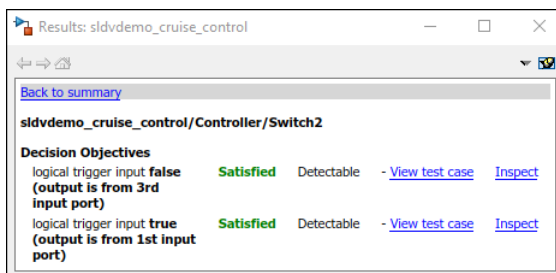
- If your model consists of blocks such as Gain or Product with the **Saturate on integer overflow** option set to 0n.

## Inspect Enhanced MCDC Objectives using Model Slicer

Model Slicer supports the following objective statuses for test case generation:

- Satisfied
- Satisfied - needs simulation
- Satisfied by existing test cases
- Undecided with test case
- Undecided due to the runtime error

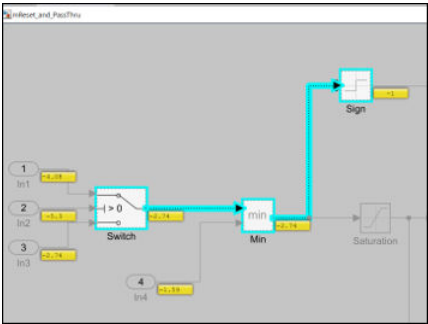
You can analyze enhanced MCDC objectives and their impact on the model by using Model Slicer. In the Results window, use the **Inspect** link to the right of the satisfied and detectable objectives.



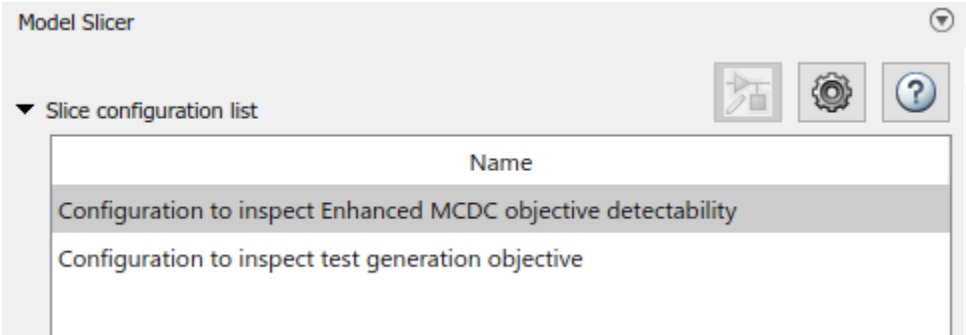
Alternatively, you can click on the **Inspect Using Slicer** button in the **Design Verifier** tab.

After launching Model Slicer, the tool sets the input based on the test case values that are relevant to the objective generated by Simulink Design Verifier and steps to the time of observation logged in

sldvData. Model Slicer then adds the model object being observed as the starting point and shows its impact on the detection point by highlighting the slice.



When you set the model coverage objective to enhanced MCDC in the Configuration parameter window, you can analyze its detectability along with inspecting the objective. In this case, the Slicer Configuration window allows you to switch to different modes using the slicer Configuration list.



**See Also**

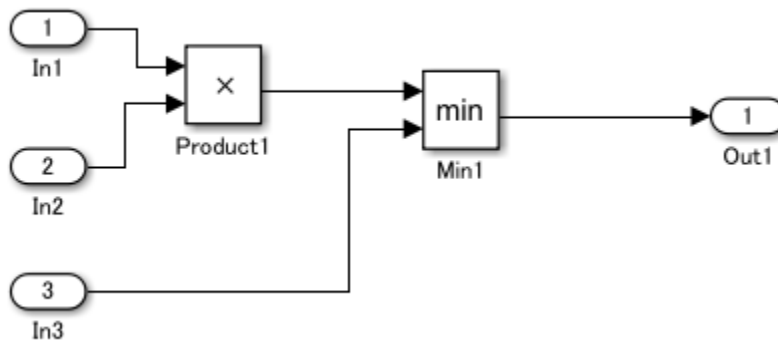
**More About**

- “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42
- “Debug Enhanced Modified Condition Decision Coverage Using Model Slicer” on page 7-121
- “Create and Run Back-to-Back Tests Using Enhanced MCDC” on page 8-18

## Author Custom Test Objective Workflow

Enhanced Modified Condition Decision Coverage (MCDC) considers the default coverage objectives of a test block that are detectable at the detection site. To check the detectability status of a custom test objective, you can author the test objective of a model object, and perform Enhanced MCDC analysis.

Consider this model that consists of a Product block and a Min block. You can author a custom test objective for the Product block to render the output value less than 0 and detect the effect of the custom test objective at a detection site.

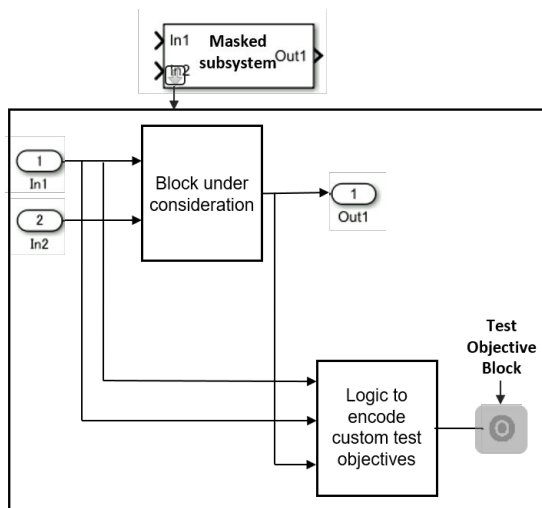


### Steps for Authoring Custom Test Objectives

This workflow describes the steps for authoring custom test objectives for a block.

**Step 1:** Create a library of atomic masked subsystem to author the custom test objectives. The masked subsystem consists of these blocks:

- Block under consideration, for example, a Product block.
- Logic to encode the custom test objective, for example, a MATLAB Function block.
- Simulink Design Verifier Test Objective blocks.



**Step 2:** In the masked subsystem:

- Add `isEnabledForDetectability` parameter and set the parameter to `On`.
- Add the `detectBlock` parameter with the name of the block under consideration.
- Set the `Evaluate` attribute of the `detectBlock` parameter to `Off`.

**Step 3:** Define the block replacement rule to replace the block under consideration with a masked subsystem.

To author custom test objectives, use `blkrep_rule_product_customTestObjective.m` block replacement rule example file. In the block replacement file, you update the `rule.BlockType` and `rule.ReplacementPath` based on your model blocks. For more information, see “Block Replacements for Unsupported Blocks” on page 4-7.

**Step 4:** Configure your model with the block replacement rule. In the Configuration Parameters dialog box, in **Design Verifier > Block Replacements** pane, enter the **List of block replacement rules**.

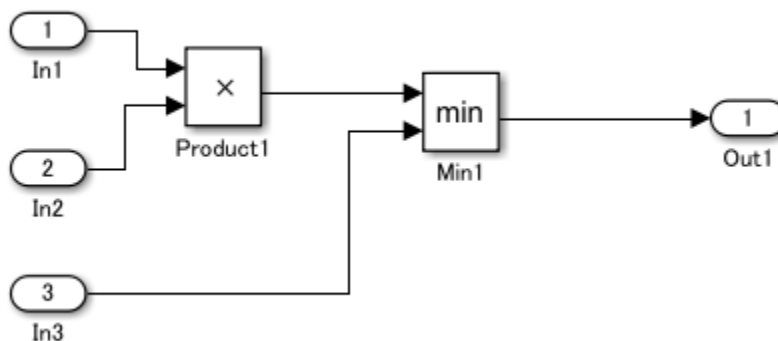
**Step 5:** Select Enhanced MCDC for **Model coverage objectives** and perform test generation analysis.

## Analyze Custom Test Objectives in Model for Enhanced MCDC

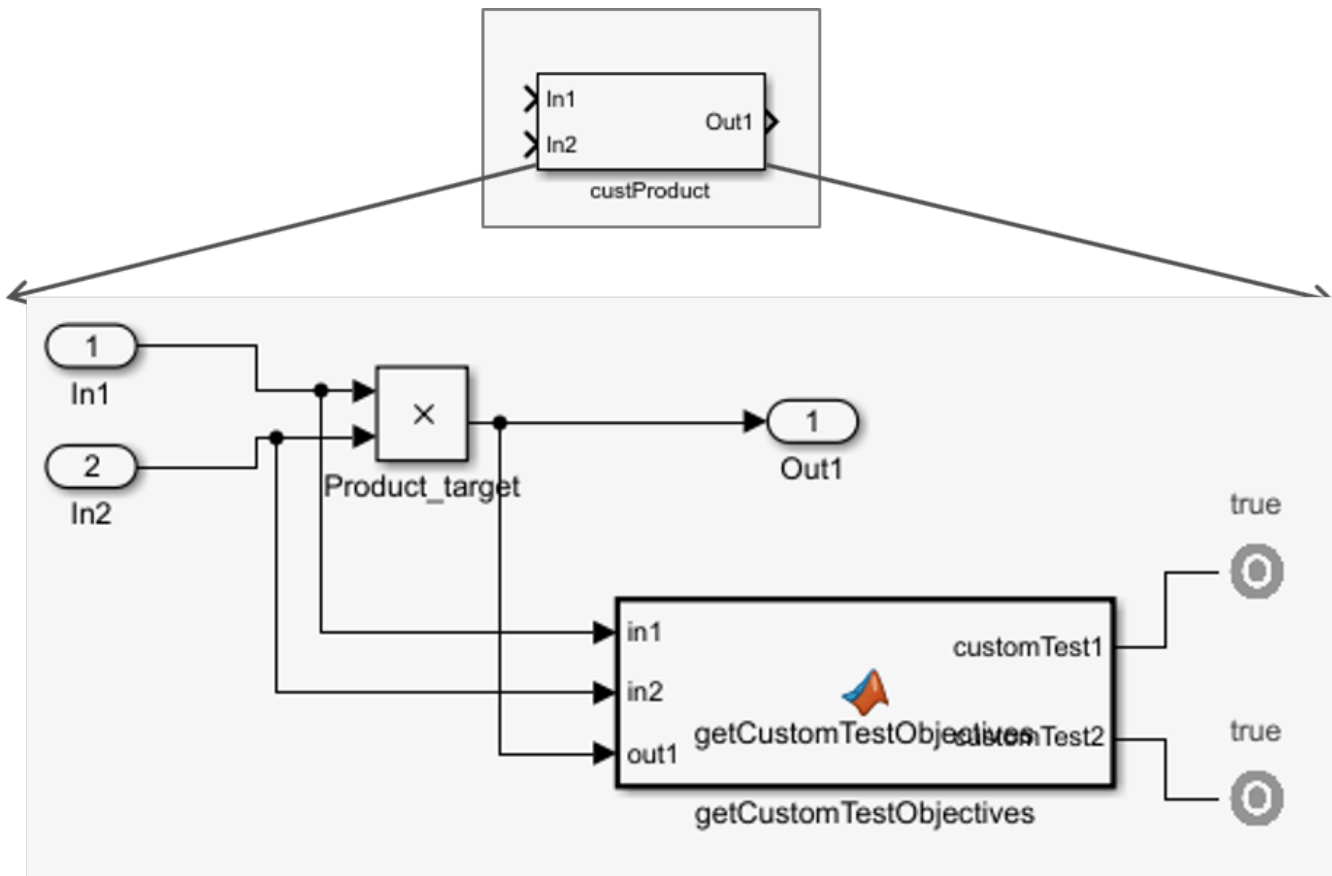
This example shows how to author custom test objectives for the Product block in the `sldvCustomTestObjectiveExample` model. Then, it shows how you can detect the effect of the test objective at a detection site.

1. Open the `sldvCustomTestObjectiveExample` model:

```
open_system('sldvCustomTestObjectiveExample');
```

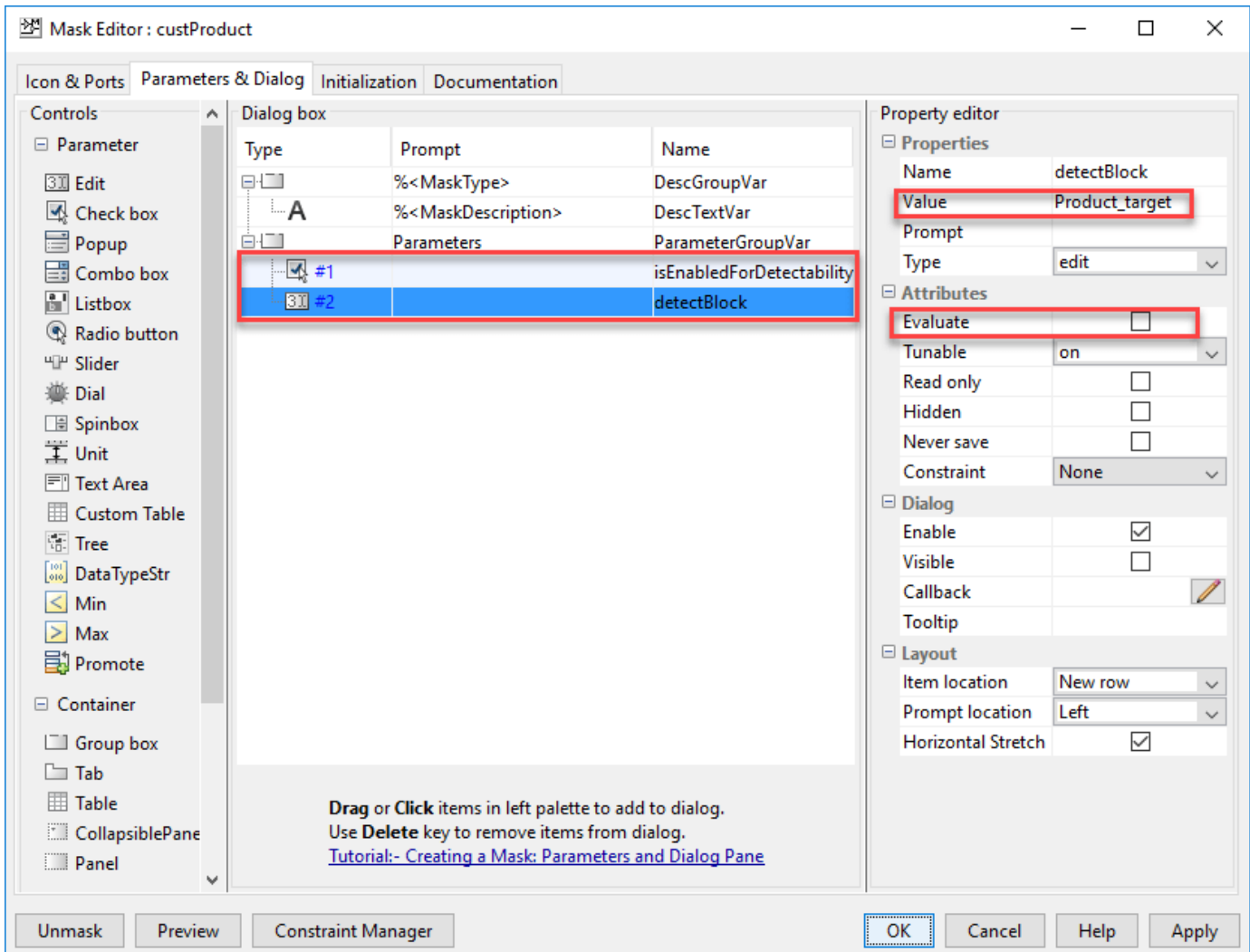


**Library of atomic masked subsystem:** The `blkReplacementlib_customTestObjective` library consists of the `custProduct` masked subsystem. The logic to encode the custom test objective is defined in the MATLAB Function block. The `getCustomTestObjectives` MATLAB Function block consists of two custom conditions for the Test Objective blocks.



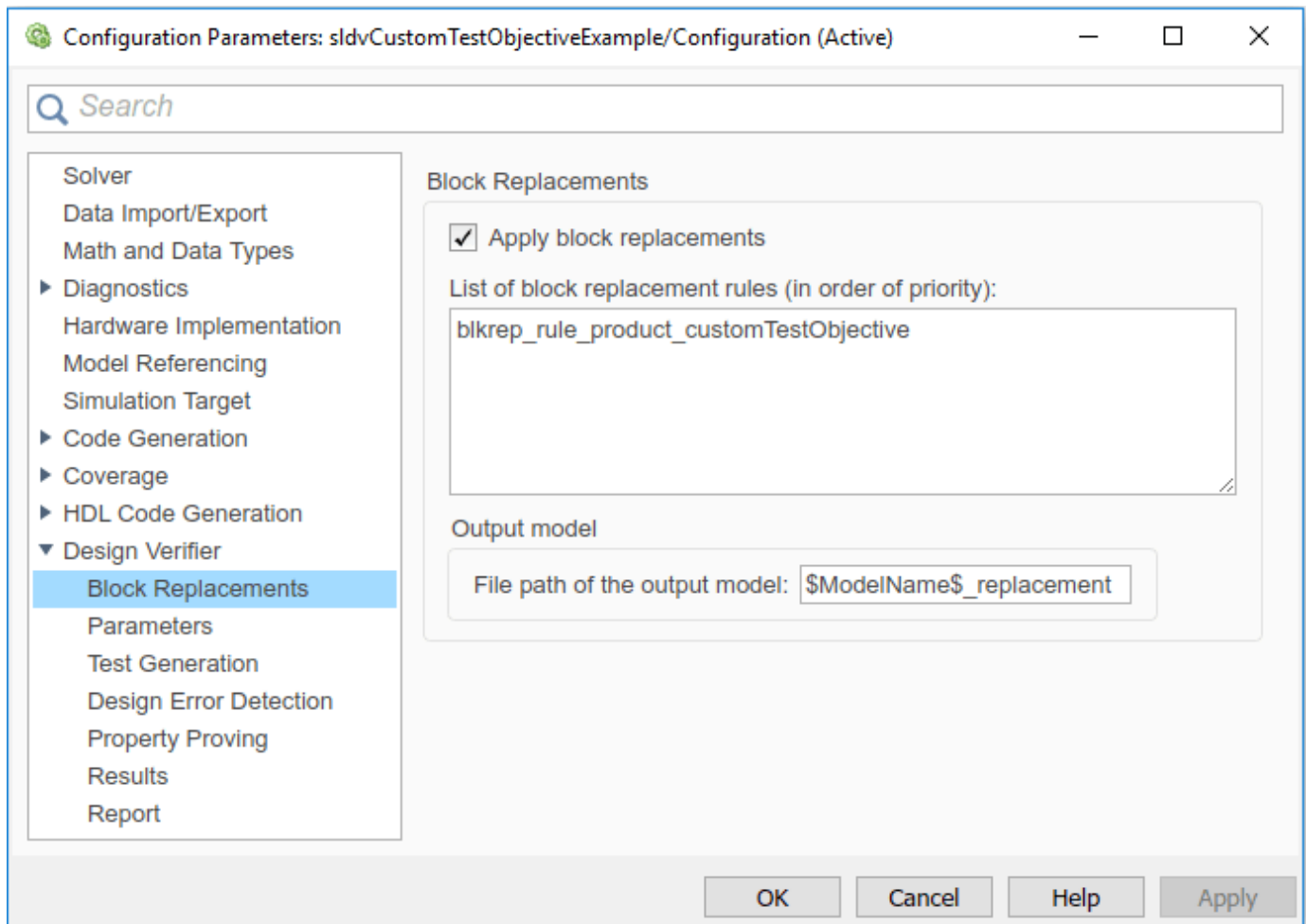
The custProduct masked subsystem is preconfigured with these parameters. For more information, see “Mask Editor Overview”.





**Block replacement rule to replace the block under consideration with a masked subsystem:**

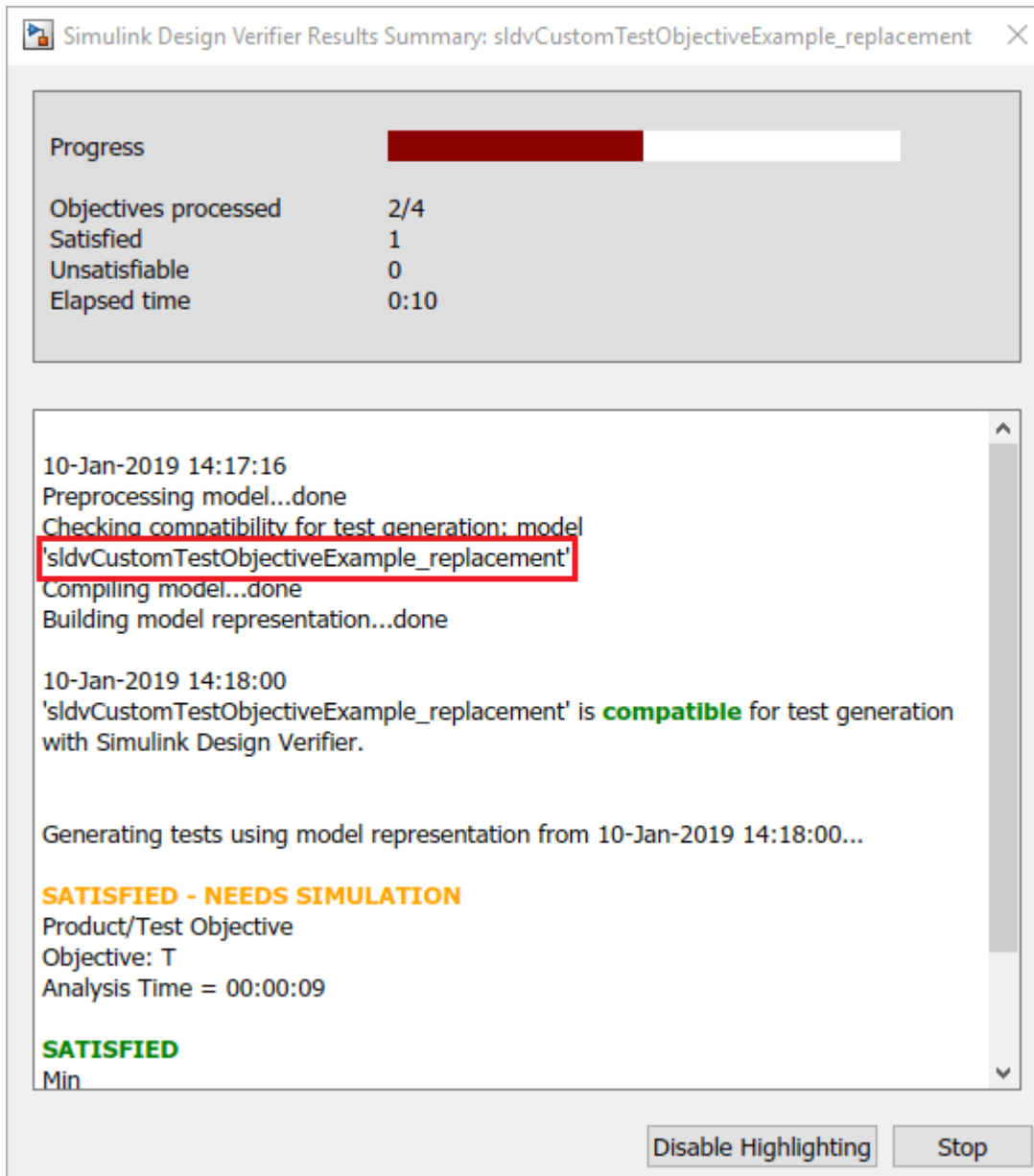
The `sldvCustomTestObjectiveExample` model is preconfigured with the block replacement options. The block replacement rule is defined in the `blkrep_rule_product_customTestObjective` file that replaces the Product block with the `custProduct` masked subsystem.



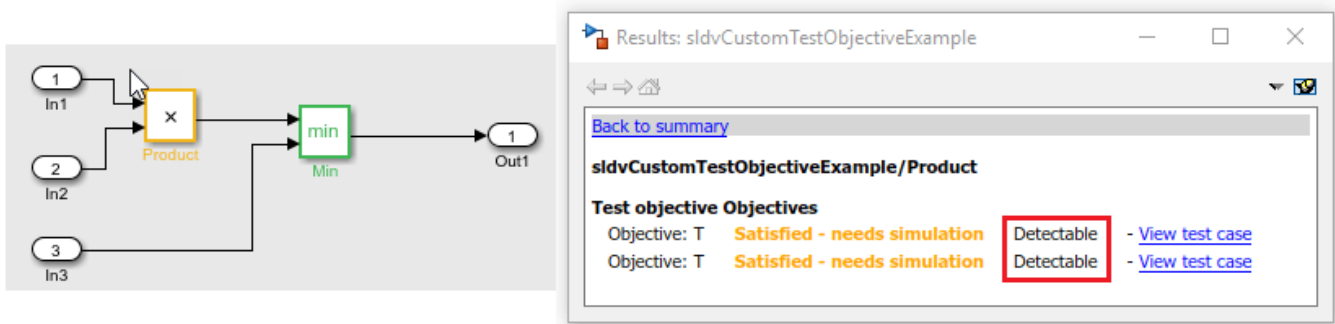
2. To configure the model for enhanced MCDC objectives, on the **Design Verifier** tab, click **Test Generation Settings**. In the Configuration Parameters dialog box, in **Design Verifier > Test Generation** pane, for Model coverage objectives, select Enhanced MCDC. Click **OK**.

3. To generate test cases, click **Generate Tests**.

The software analyzes the replacement model for test generation.



4. Click **Highlight analysis results on model**. To analyze the detectability of the Product block, click the Product block.



The results show that the test objectives of the Product block are detectable. The test case is generated.

**Note:** The software is unable to confirm the objectives status through validation results for the objectives introduced by block replacement. Therefore, the test objective status is reported as satisfied - needs simulation. For more information on validation, see “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23.

5. Click **View test case**. The harness model opens and the Signal Builder block displays the test case.
6. To view the detailed analysis report, click **HTML** in the Results Summary. The Block Replacement Summary provides details about the replaced blocks.

## Block Replacements Summary

Table 2.1. Block Replacements

| #: | Replacement Rule / Block Type             | Rule Description                  | Replaced Blocks          |
|----|---|-----------------------------------|--------------------------|
| 1  | blkrep_rule_product_customTestObj/Product | blkrep_rule_product_customTestObj | <a href="#">Product1</a> |

The Test Objectives Status section lists the objectives. The test objective that is detectable at the detection site is summarized in the table.

## Objectives Satisfied

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

| # | Type     | Model Item          | Description   | Detection Status | Analysis Time (sec) | Test Case         |
|---|----------|---------------------|---|------------------|---------------------|-------------------|
| 3 | Decision | <a href="#">Min</a> | Logic to determine output <b>input 1 is the minimum</b> | Detectable       | 13                  | <a href="#">3</a> |
| 4 | Decision | <a href="#">Min</a> | Logic to determine output <b>input 2 is the minimum</b> | Detectable       | 12                  | <a href="#">1</a> |

## Objectives Satisfied - Needs Simulation

Simulink Design Verifier found test cases that exercise these test objectives. However, further simulation is needed to confirm the Satisfied status.

| # | Type           | Model Item  | Description  | Detection Status | Analysis Time (sec) | Test Case         |
|---|----------------|---|--------------|------------------|---------------------|-------------------|
| 1 | Test objective | <a href="#">Product/Test Objective. Defined by block replacement rule 'blkrep_rule_product_customTestObjective'.</a>  | Objective: T | Detectable       | 12                  | <a href="#">4</a> |
| 2 | Test objective | <a href="#">Product/Test Objective1. Defined by block replacement rule 'blkrep_rule_product_customTestObjective'.</a> | Objective: T | Detectable       | 14                  | <a href="#">3</a> |

## See Also

### More About

- “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42
- “Block Replacements for Unsupported Blocks” on page 4-7

## What Is a Specification Model?

When you systematically verify a model against requirements, you generate test cases for each requirement. These tests validate the model, which you can use to generate production code and build confidence that your model satisfies requirements. To create tests that satisfy your requirements, you can construct a *specification model*. A specification model is an executable entity that you can use to perform requirements-based testing by using Simulink Design Verifier and Requirements Toolbox.

If you have a set of requirements written in natural language text, you can express them as formal requirements by using a Requirements Table block. After defining the requirements in one or more blocks, the blocks and the signals become the specification model. Unlike the model that you want to test, known as the *design model*, the specification model only specifies what to do, not how to do it.

You can use a specification model to:

- Validate the set of requirements in a systematic and quantitative manner.
- Automate requirements-based testing.
- Identify issues with your design model and requirements.

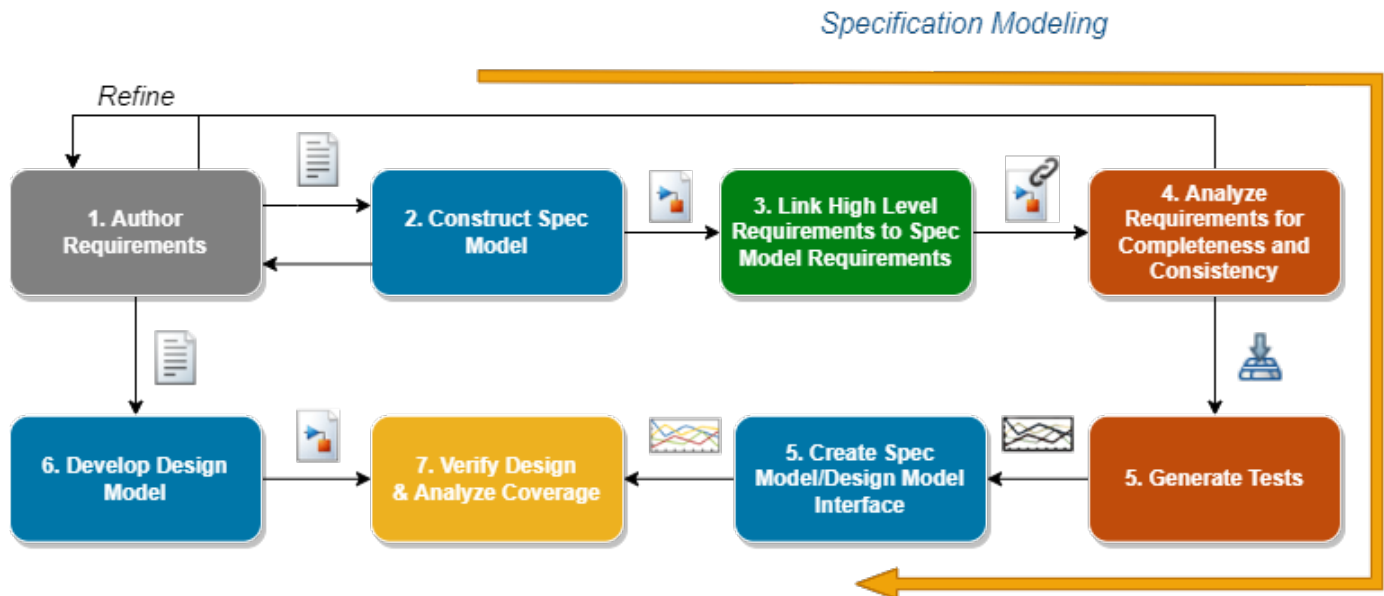
## Use Specification Models in Requirements-Based Testing

To create and deploy a specification model, follow these steps:

- 1** Author the requirements — Write your requirements in natural language text that describes the behavior of the system under design. Author them directly in the **Requirements Editor** or import them. For more information on importing requirements, see “Import Requirements from Third-Party Applications” (Requirements Toolbox).
- 2** Construct the specification model — Design the specification model as an formal representation of the requirements by using at least one Requirements Table block.
- 3** Link the requirements — Each requirement that you create in the Requirements Table block creates an equivalent requirement in the **Requirements Editor**. See “Configure Properties of Formal Requirements” (Requirements Toolbox). Link the high-level requirements to the formal requirements from the specification model.
- 4** Analyze the formal requirements for completeness and consistency — Identifying incomplete and inconsistent requirement sets can be difficult to do manually. The Requirements Table block allows you to automatically analyze the requirements for these issues. See “Identify Inconsistent and Incomplete Formal Requirement Sets” (Requirements Toolbox).
- 5** Generate tests for the specification model — Generate at least one test per requirement that demonstrates its conformance to that requirement. For more information on generating tests, see “Generate Test Cases for a Subsystem” on page 7-18. Simulink Design Verifier automatically creates test objectives from the requirements defined in Requirements Table blocks.
- 6** Interface the specification model with the design model — The specification and design models often do not use identical input and output signals. Convert the test cases that you generate in step 5 by developing an interface between both models.
- 7** Develop the design model — Develop the design model by using the requirements. Link the requirements to the design model.
- 8** Verify the design and analyze the coverage — Run the tests generated in step 5 on the design model and verify whether the results agree with the specification model and requirements.

Generate a coverage report to identify the missing coverage and refine the requirements, if required.

This flow chart illustrates this process.



## Construct a Specification Model

Consider the autopilot controller model described in “Use Specification Models for Requirements-Based Testing” on page 7-69. In this example, you develop requirements that contain logical and temporal conditions that define outputs.

### Identify the Specification Model Interface

List the input and output signals for the specification model that are related to the requirements that you want to test. Ignore the signals that the requirements do not specify and that do not affect the tested outputs. In this example, the requirements specify five inputs and two outputs. The specification model input signals are:

- 1 Autopilot Engage Switch — A switch that enables or disables the autopilot controller
- 2 Heading Engage Switch — A switch that specifies the mode of the autopilot controller when the autopilot switch is engaged
- 3 Roll Reference Target Turn Knob — A knob that feeds the desired roll angle value to the autopilot controller
- 4 Heading Reference Turn Knob — A knob that gives the set-point value for heading mode
- 5 Aircraft Roll Angle — The current roll angle of the aircraft

The output signals are:

- 1 Aileron Command — The output to the aileron actuator
- 2 Roll Reference Command — The output on the display window that indicates the set-point value for the aileron actuator

## Identify Preconditions, Postconditions, and Actions for Each Requirement

For the requirements that you want to verify, transform the textual requirements into logical expressions that can be represented as preconditions, postconditions, and actions. You define formal requirements as a combination of Preconditions, Postconditions, and Actions:

- **Precondition** — A condition that must be true for a specified duration before evaluating the rest of the requirement
- **Postcondition** — A condition that must be true if the associated precondition is true for the specified duration
- **Action** — A behavior that must be performed if the associated precondition is true for the specified duration

You may find that some requirements can use a postcondition or an action interchangeably, or both postconditions and actions. Specify which you want to use based on the configuration of your design model.

For example consider this high level requirement that specifies the modes of the autopilot controller:

The autopilot controller mode is determined by the following:

- The autopilot controller is OFF when the autopilot engage switch is not engaged.
- The autopilot controller is ROLL\_HOLD\_MODE when the autopilot engage switch is engaged and the heading engage switch is not engaged.
- The autopilot controller is HDG\_HOLD\_MODE when the autopilot engage switch and the heading engage switch are both engaged.

You can write these requirements as these logical expressions:

| Requirement | Precondition   | Action                |
|-------------|--|-----------------------|
| 1           | AP_Engage_Switch == false                              | Mode = Off            |
| 2           | AP_Engage_Switch == true && HDG_Engage_Switch == false | Mode = ROLL_HOLD_MODE |
| 3           | AP_Eng_Switch == true && HDG_Engage_Switch == true     | Mode = HDG_Hold_Mode  |

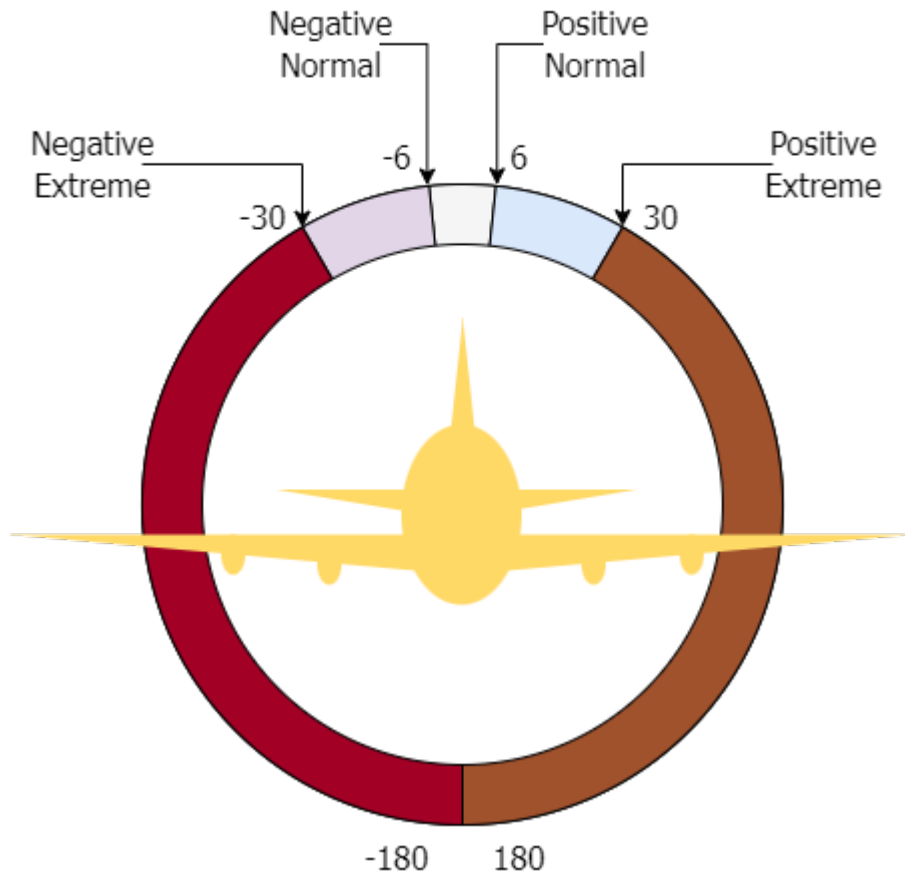
Repeat this process for the remaining requirements.

## Identify Design Values Representations in Requirements

Your requirements may specify ranges of values that your design model must satisfy, or you may want to parameterize the values that you evaluate in each requirement. These values cannot always be described easily with literal values. You can use the Requirements Table block to represent values in the expressions as constant or parameter data. See “Define Data in Requirements Table Blocks” (Requirements Toolbox). You can change data throughout simulation. In addition to assigning numerical values to data, the block supports other data types, such as strings, enumerations, or ranges. Use the representation of values that fits your needs.



In the autopilot controller model, the requirements specify threshold values for the aircraft roll angle. This graphic illustrates the numerical and verbal equivalents of the thresholds.



### Create the Requirements Table Blocks

After identifying the signal representations, values, and the expressions that you want to use in the formal requirements, write the logical expression of the precondition, postconditions, and actions in the **Precondition Postcondition**, and **Action** columns for each requirement respectively. If your requirements have children or dependencies, you can include those relationships in the block. See “Establish Hierarchy in Requirements Table Blocks” (Requirements Toolbox).

Each requirement that you create in the Requirements Table block creates an equivalent requirement in the **Requirements Editor**. Update additional textual properties of the requirements, such as the description, in the editor. See “Configure Properties of Formal Requirements” (Requirements Toolbox).

In the autopilot controller model, the specification model includes two Requirements Table blocks. `AP_Mode_Determination` defines the formal requirements for the autopilot controller mode.

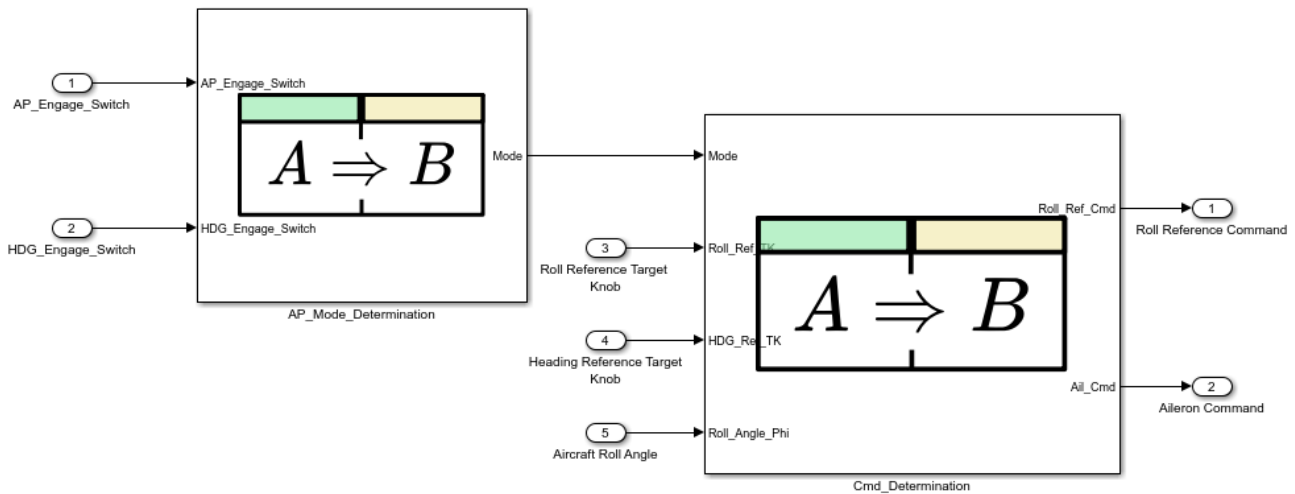
| Requirements |  | Assumptions      |                   |                |
|--------------|--|------------------|-------------------|----------------|
| Index        | Summary  | Precondition     |                   | Action         |
|              |  | AP_Engage_Switch | HDG_Engage_Switch | Mode           |
| 1            | AP_Engage_Switch and HDG_Engage_Switch both engaged              | true             | true              | HDG_HOLD_MODE  |
| 2            | AP_Engage_Switch is engaged and HDG_Engage_Switch is not engaged | true             | false             | ROLL_HOLD_MODE |
| 3            | AP_Engage_Switch is not engaged                                  | false            |                   | OFF            |

The other Requirements Table block, *Cmd\_Determination*, describes the desired output of the aileron command and the roll reference command.

| Requirements |   | Assumptions                    |  |                              |                    |                |     |
|--------------|---|--------------------------------|--|------------------------------|--------------------|----------------|-----|
| Index        | Summary   | Precondition                   |  |                              | Action             |                |     |
|              |   | Mode                           | Roll_Ref_TK  | prev(Roll_Angle_Phi)         | Roll_Ref_Cmd       | Ail_Cmd        |     |
| 1            | Autopilot mode is OFF   | OFF                            |  |                              | 0                  | Zero           |     |
| 2            | ROLL_HOLD_MODE becomes active mode                                    | hasChangedTo(X,ROLL_HOLD_MODE) |  |                              |                    | All            |     |
| 2.1          | Roll_Ref_TK between [-30, -3] or [+3, +30] degrees                    |                                | [TK_neg_extreme, TK_neg_norm]    [TK_pos_norm, TK_pos_extreme] |                              |                    | Roll_Ref_TK    |     |
| 2.2          | Roll_Ref_TK greater than -3 and less than +3:                         |                                | (TK_neg_norm, TK_pos_norm)                                     |                              |                    |                |     |
| 2.2.1        | Roll_Angle_Phi greater than neg_norm and less than pos_norm           |                                |  | [phi_neg_norm, phi_pos_norm] | 0                  |                |     |
| 2.2.2        | Roll_Angle_Phi greater than +30                                       |                                |  | > phi_pos_extreme            | TK_pos_extreme     |                |     |
| 2.2.3        | Roll_Angle_Phi less than -30  |                                |  | < phi_neg_extreme            | TK_neg_extreme     |                |     |
| 2.2.4        | Otherwise, Roll_Ref_Cmd default setting                               | Else                           |  |                              |                    | Roll_Angle_Phi |     |
| 3            | HDG_HOLD_MODE becomes active mode                                     | hasChangedTo(X,HDG_HOLD_MODE)  |  |                              |                    | HDG_Ref_TK     | All |
| 4            | Otherwise, Roll_Ref_Cmd shall hold the previous value of Roll_Ref_Cmd | Else                           |  |                              | prev(Roll_Ref_Cmd) | All            |     |

### Final Specification Model

After connecting the Requirements Table blocks to the inputs, outputs, and each other, the final specification model is:



## Prepare the Specification Model for Test Generation

Simulink Design Verifier automatically creates test objectives from the requirements defined in Requirements Table blocks. If you need to constrain the values of the test objectives, you can specify them either in the signal source, or by including them in the **Assumptions** table of the block. See “Add Assumptions to Requirements” (Requirements Toolbox). To prepare the specification model for test generation, set the model coverage objectives. In the **Design Verifier** tab, in the **Prepare** section, click **Test Generation Settings**. In the Configuration Parameters window, expand the **Design Verifier** list and click **Test Generation**. Set **Model coverage objectives** to the option that best captures the desired coverage.

## Iterate Through the Steps

As you develop the specification model and test your design model, you typically need to update the requirements, specification model, and design model. This process is iterative. Continue iterating until you reach the desired test outcomes, such as desired model outputs and test coverage.

## See Also

Requirements Table

## Related Examples

- “Use a Requirements Table Block to Create Formal Requirements” (Requirements Toolbox)
- “Use Specification Models for Requirements-Based Testing” on page 7-69
- “Export Tests from Models That Contain Requirements Table Blocks with Simulink Design Verifier” on page 13-30

## Test Generation Examples

These test generation examples help you understand and use the test generation capabilities.

| Test Generation Capabilities               | Related Examples  |
|--|---|
| Generate tests for model coverage analysis | "Cruise Control Test Generation" on page 7-84                                     |
|  | "Fuel Rate Controller Logic" on page 7-85   |
|  | "Flip Flop Test Generation" on page 7-80  |
|  | "Model Coverage Test Generation" on page 7-81                                     |
| Functional Requirements Testing            | "Test Condition Block" on page 7-83   |
|  | "Test Objective Block" on page 7-82   |
| Generate tests for code coverage analysis  | "Configuring S-Function for Test Case Generation" on page 7-109                   |
|  | "Code Coverage Test Generation" on page 7-111                                     |
|  | "Test Generation on Model with C Caller Block" on page 7-119                      |
|  | "Test Generation for Custom Code in a Stateflow Chart" on page 7-124              |
| Extend existing test cases                 | "Defining and Extending Existing Tests Cases" on page 7-91                        |
|  | "Extend an Existing Test Suite" on page 7-86                                      |
|  | "Creating and Executing Test Cases" on page 7-100                                 |
|  | "Extend Existing Test Cases After Applying Parameter Configurations" on page 5-46 |
| Achieve missing coverage                   | "Achieve Missing Coverage in Referenced Model" on page 9-3                        |
|  | "Achieve Missing Coverage in Closed-Loop Simulation Model" on page 9-11           |
|  | "Using Existing Coverage Data During Subsystem Analysis" on page 7-97             |
| Integrate with other products              | "Export Test Cases to Simulink Test" on page 13-27                                |
|  |   |

## Test Generation for Custom Code in MATLAB Function Block

Simulink Design Verifier analysis supports models that call custom code from MATLAB function blocks by using `coder.ceval`. For such design models, you can generate test cases for model coverage or perform design error detection to find dead logic or detect design errors.

The table summarizes various `coder.ceval` use-cases that Simulink Design Verifier supports:

### Supported `coder.ceval` use-cases:

| <code>coder.ceval</code> usage  | Custom code sources  | Analysis    |
|---|--|-------------|
| Basic calls - with or without arguments   | Source files mentioned in Simulink target in <b>Configuration Parameters</b> . | Supported   |
| Layout - rowMajor, columnMajor  |  |             |
| Passing references using <code>coder.ref</code> , <code>coder.wref</code> , <code>coder.rref</code> |  |             |
| Any layout -global  | -  | Unsupported |
| -   | Source file mentioned by using <code>coder.updateBuildInfo</code>              | Unsupported |

## Generating Tests for Custom code in MATLAB function block

This example demonstrates test generation workflow for model by using `coder.ceval`.

Consider a model with MATLAB function block calling the custom code by using `coder.ceval`.

- 1 Create the required source files as mentioned in `coder.ceval`.

The C-function `checkIfSignalsInRange`, represents custom code. The function returns 1 if both the signals are in acceptable range else, the function returns 0. The MATLAB function block `checkIfSignalsINRangeWrapper`, receives sensor inputs and invokes the C-function.

C file:

```
#include <stdio.h>
#include <stdlib.h>
#include "checkIfSignalsInRange.h"
int checkIfSignalsInRange(double sig1, double sig2) {
    double acceptableMin = 15;
    double acceptableMax = 150;
    if ( ((sig1 > acceptableMin) && (sig1 < acceptableMax)) && ((sig2 > acceptableMin) && (sig2 < acceptableMax)) )
        return 1;
    }
    return 0;
}
```

Header file:

```
int checkIfSignalsInRange(double sig1, double sig2);
```

MATLAB function Block:

```
function result = checkIfSignalsINRangeWrapper(sig1,sig2)
result = 0;
% Check if both the signals are within acceptable range
result = coder.ceval('checkIfSignalsInRange',sig1,sig2);
```

- 2 Navigate to **Simulation Target** in **Configuration Parameters**. In the **Code information** tab add the required files.
- 3 Set the **Enable custom code analysis** option in the **Import settings** tab.
- 4 Set the model coverage objectives to **Decision** and invoke test generation analysis in **Configuration Parameters**.

## Results

The model has three decision objectives, one for MATLAB function block execution and two for custom code. The two decision objectives correspond in making the outcome of `if` condition once true and once false. The generated test and report is as follows:

### 4.2. CoderCevalExample

[View](#)

| #: | Type     | Description  | Status    | Test Case         |
|----|----------|--|-----------|-------------------|
| 2  | Decision | decision ((sig1 > acceptableMin) && (sig1 < acceptableMax)) && ((sig2 > acceptableMin) && (sig2 < acceptableMax)) T (file checkIfSignalsInRange.c, function checkIfSignalsInRange, line 9) | Satisfied | <a href="#">1</a> |
| 3  | Decision | decision ((sig1 > acceptableMin) && (sig1 < acceptableMax)) && ((sig2 > acceptableMin) && (sig2 < acceptableMax)) F (file checkIfSignalsInRange.c, function checkIfSignalsInRange, line 9) | Satisfied | <a href="#">1</a> |

#### Generated Input Data

|             |          |            |
|-------------|----------|------------|
| <b>Time</b> | <b>0</b> | <b>0.2</b> |
| <b>Step</b> | <b>1</b> | <b>2</b>   |
| u           | 16       | 0          |
| u1          | 16       | 0          |

From the report it is inferred that in Step 1 both signals are in acceptable range and in Step 2 the signals are out of range.

## Use Specification Models for Requirements-Based Testing

This example shows how to use a specification model to model and test formal requirements on a model of an aircraft autopilot controller. The specification model uses two Requirements Table blocks to model the required inputs and outputs of the aircraft autopilot controller model. You generate tests from the specification model, and then run those tests on the aircraft autopilot controller model. The model that you test is the *design model*.

For more information on how to define and configure Requirements Table blocks, see “Use a Requirements Table Block to Create Formal Requirements” (Requirements Toolbox) and “Configure Properties of Formal Requirements” (Requirements Toolbox).

### View the High-Level Requirements

Open the requirements set, AP\_Controller\_Reqs, in the **Requirements Editor**.

```
slreq.open("AP_Controller_Reqs");
```

The high-level requirements specify the outputs of the model and the autopilot controller mode. Each requirement description uses high-level language that you can use to explicitly define the logic needed in the formal requirements.

The screenshot shows the Requirements Editor interface. On the left, a tree view shows the 'AP\_Controller\_Reqs' set containing five requirements. The main pane displays the details for 'Requirement: 1'.

| Index | ID | Summary                            |
|-------|----|------------------------------------|
| 1     | 1  | High Level: Autopilot Controlle... |
| 2     | 2  | High Level: Off Reference          |
| 3     | 3  | High Level: Roll Hold Reference    |
| 4     | 4  | High Level: Heading hold mode...   |
| 5     | 5  | High Level: Default Behavior       |

**Requirement: 1**

**Properties**

- Type: Functional
- Index: 1
- Custom ID: 1
- Summary: High Level: Autopilot Controller Modes

**Description**

Times New Roman 11

The autopilot controller has the following high level system modes:

- OFF: The autopilot controller is off.
- ROLL\_HOLD\_MODE: The autopilot controller is in roll hold mode.
- HDG\_HOLD\_MODE: The autopilot controller is in heading hold mode.

The autopilot controller mode is determined by the following:

- The autopilot controller is OFF when the autopilot engage switch is not engaged.
- The autopilot controller is ROLL\_HOLD\_MODE when the autopilot engage switch is engaged and the heading engage switch is not engaged.
- The autopilot controller is HDG\_HOLD\_MODE when the autopilot engage switch and the heading engage switch are both engaged.

Keywords:

Revision information:

Links

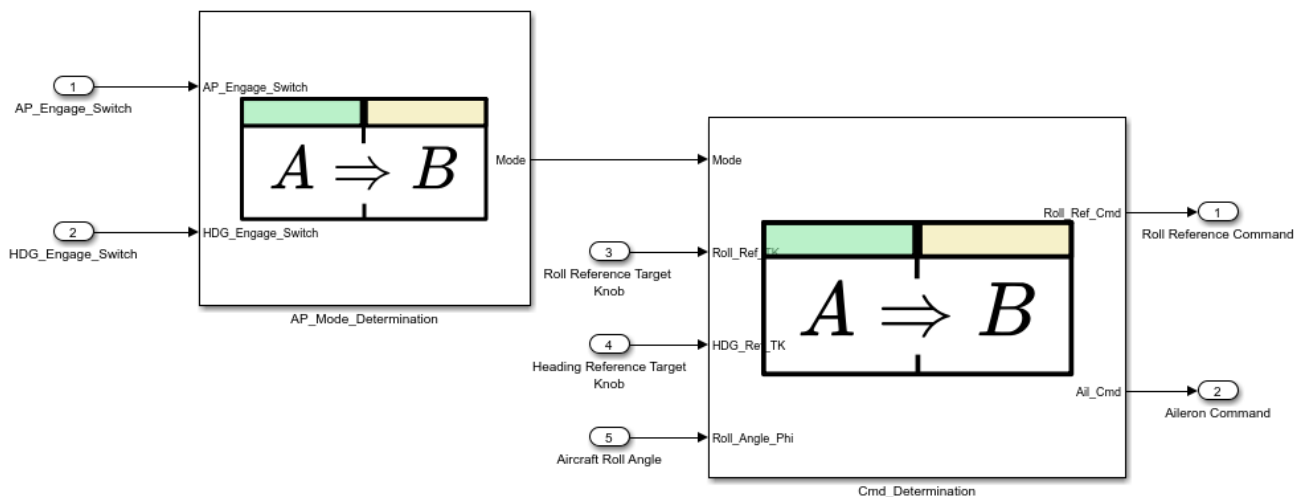
Comments

### View the First Iteration of the Specification Model

Open the specification model, `spec_model_partial`.

```
spec_model = "spec_model_partial";
open_system(spec_model);
```

The model contains two Requirements Table blocks that define the formal requirements that translate the high-level requirements into testable logical expressions. The block `AP_Mode_Determination` specifies the formal requirements for the autopilot controller mode, and the block `Cmd_Determination` specifies the outputs of the controller.



To view the formal requirements, inspect each Requirements Table block.

#### Requirements Table Block for Controller Mode

Open `AP_Mode_Determination`. The block specifies the formal requirements for the autopilot controller mode. To determine the output data `Mode`, `AP_Mode_Determination` specifies three requirements by using two input data:

- `AP_Engage_Switch` — The autopilot engage switch
- `HDG_Engage_Switch` — The heading engage switch

Each requirement uses a combination of the inputs to specify a unique output value for `Mode`.



| Requirements |  | Assumptions      |                   |                |
|--------------|--|------------------|-------------------|----------------|
| Index        | Summary  | Precondition     |                   | Action         |
|              |  | AP_Engage_Switch | HDG_Engage_Switch | Mode           |
| 1            | AP_Engage_Switch and HDG_Engage_Switch both engaged              | true             | true              | HDG_HOLD_MODE  |
| 2            | AP_Engage_Switch is engaged and HDG_Engage_Switch is not engaged | true             | false             | ROLL_HOLD_MODE |
| 3            | AP_Engage_Switch is not engaged                                  | false            |                   | OFF            |

### Requirements Table Block for Controller Commands

Open `Cmd_Determination`. `Cmd_Determination` specifies the requirements for the aileron command and roll reference command. `Cmd_Determination` uses four input data:

- `Mode` — The `AP_Mode_Determination` output, `Mode`
- `Roll_Ref_TK` — The setting of the roll reference target knob
- `Roll_Angle_Phi` — The actual aircraft roll angle
- `HDG_Ref_TK` — The setting of the heading reference target knob

The block uses these input data to determine the controller output data:

- `Roll_Ref_Cmd` — Roll reference command
- `Ail_Cmd` — Aileron command

| Requirements |   | Assumptions                                 |   |   |                                 |                |     |
|--------------|---|---|---|---|---------------------------------|----------------|-----|
| Index        | Summary   | Precondition                                |   | Action                                    |                                 |                |     |
|              |   | Mode  | Roll_Ref_TK   | prev(Roll_Angle_Phi)                      | Roll_Ref_Cmd                    | Ail_Cmd        |     |
| 1            | Autopilot mode is OFF   | OFF   |   |   | 0                               | Zero           |     |
| 2            | ROLL_HOLD_MODE becomes active mode  | <code>hasChangedTo(X,ROLL_HOLD_MODE)</code> |   |   |                                 | All            |     |
| 2.1          | Roll_Ref_TK between [-30, -3] or [+3, +30] degrees                                    |   | <code>[TK_neg_extreme, TK_neg_norm]    [TK_pos_norm, TK_pos_extreme]</code> |   |                                 | Roll_Ref_TK    |     |
| 2.2          | Roll_Ref_TK greater than -3 and less than +3  |   | <code>(TK_neg_norm, TK_pos_norm)</code>                                     |   |                                 |                |     |
| 2.2.1        | Roll_Angle_Phi greater than <code>neg_norm</code> and less than <code>pos_norm</code> |   |   | <code>[phi_neg_norm, phi_pos_norm]</code> | 0                               |                |     |
| 2.2.2        | Roll_Angle_Phi greater than +30   |   |   | <code>&gt; phi_pos_extreme</code>         |                                 | TK_pos_extreme |     |
| 2.2.3        | Roll_Angle_Phi less than -30  |   |   | <code>&lt; phi_neg_extreme</code>         |                                 | TK_neg_extreme |     |
| 3            | HDG_HOLD_MODE becomes active mode   | <code>hasChangedTo(X,HDG_HOLD_MODE)</code>  |   |   |                                 | HDG_Ref_TK     | All |
| 4            | Otherwise, Roll_Ref_Cmd shall hold the previous value of Roll_Ref_Cmd                 | Else  |   |   | <code>prev(Roll_Ref_Cmd)</code> | All            |     |

In this example, the expressions use constant data to define the ranges of values for `Roll_Ref_TK` and `Roll_Angle_Phi`. You can also parameterize the values or use literal values. See “Define Data in Requirements Table Blocks” (Requirements Toolbox). To view these values, open the **Symbols** pane. In the **Modeling** tab, in the **Design Data** section, click **Symbols Pane**.

In addition to requirements, `Cmd_Determination` also defines the assumptions for the design. See “Add Assumptions to Requirements” (Requirements Toolbox). In this example, the assumptions constrain the values for the roll angle and the roll reference target knob based on their physical limitations. The roll angle cannot exceed 180 or fall below -180 degrees, and the roll reference target knob cannot exceed 30 or fall below -30. In the table, click the **Assumptions** tab.

| Requirements |   | Assumptions                                     |
|--------------|---|---|
| Index        | Summary                                     | Postcondition                                   |
| 1            | Roll angle geometric limitations            | Roll_Angle_Phi >= -180 && Roll_Angle_Phi <= 180 |
| 2            | Reference knob minimum and maximum settings | Roll_Ref_TK <= 30 && Roll_Ref_TK >= -30         |

You can also specify data range limitations in the **Minimum** and **Maximum** properties of the data or explicitly specify the range from the signal with blocks.

### Generate Tests

Simulink® Design Verifier™ automatically creates test objectives from the requirements defined in Requirements Table blocks. To generate tests, use the Configuration Parameter window or specify the tests programmatically. See “Model Coverage Objectives for Test Generation” on page 7-30. Select different coverage objectives to determine if you want to minimize the number of tests generated, or if you want to improve test granularity and traceability.

In this example, generate tests with decision coverage and save the output to a MAT-file.

```
opts = sldvoptions;
opts.Mode = "TestGeneration";
opts.ModelCoverageObjectives = "Decision";
[~,files] = sldvrun(spec_model,opts,true);
```

Simulink Design Verifier generates the test objectives and the tests from the requirements, however the requirements satisfy only seven of the test objectives.

Simulink Design Verifier Results Summary: spec\_model\_partial

|                      |       |
|----------------------|-------|
| Progress             |       |
| Objectives processed | 24/24 |
| Satisfied            | 7     |
| Unsatisfiable        | 0     |
| Elapsed time         | 0:26  |

Test generation completed normally.  
7/24 objectives satisfied.  
17/24 objectives undecided due to runtime error

Results:

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

To satisfy the test objectives, you must revise the specification model. In general, avoid generating tests from a specification model without confirming that the formal requirements are complete, consistent, and correspond to the high-level requirements. Otherwise, the tests that you generate are less likely to satisfy the test objectives.

```
clear("files")
```

### Investigate and Update the Specification Model

Investigate the specification model and update the formal requirements. In this example, the requirement set in `Cmd_Determination` is missing the formal requirement that corresponds to the third bullet of requirement 3.

Requirement: 3

**▼ Properties**

Type:

Index: 3

Custom ID:

Summary:

Description Rationale

Times New Roman 11 **B** *I* U █

When roll hold mode becomes the active mode of the autopilot controller, the roll reference command shall be set to the cockpit turn knob command, up to a 30 degree limit, if the turn knob is commanding 3 degrees or more in either direction.

If the turn knob is commanding between -3 and 3 degrees in either direction:

- The roll reference command shall be set to zero if the actual roll angle is less than 6 degrees, in either direction, just before the roll hold mode is received.
- The roll reference command shall be set to 30 degrees in the same direction as the actual roll angle if the actual roll angle is greater than 30 degrees just before the roll hold mode is received.
- The roll reference command shall be set to the actual roll angle when the actual roll angle equals other angles.

Open `Cmd_Determination` in the model `spec_model_final` to view the updated requirement set. The additional requirement has the index 2.2.4.

```
spec_model = "spec_model_final";
load_system(spec_model);
open_system(spec_model + "/Cmd_Determination");
```

|       |   |                               |  |                   |                |
|-------|---|-------------------------------|--|-------------------|----------------|
| 2.2.3 | Roll_Angle_Phi less than -30            |                               |  | < phi_neg_extreme | TK_neg_extreme |
| 2.2.4 | Otherwise, Roll_Ref_Cmd default setting | Else                          |  |                   | Roll_Angle_Phi |
| 3     | HDG_HOLD_MODE becomes active mode       | hasChangedTo(X,HDG_HOLD_MODE) |  |                   | HDG_Ref_TK     |

Finding issues in your requirement set can be challenging to do manually. You can use Simulink Design Verifier to analyze the requirement set and identify inconsistencies and incompletenesses. For more information, see “Analyze the Block” (Requirements Toolbox).

### Link High-Level and Formal Requirements

Loading the specification model loads the formal requirements in the **Requirements Editor**. Closing the specification model also closes the associated requirement set. When developing your formal requirements, link formal requirements to the corresponding high-level requirement to track the

requirements in the specification model. In this example, linking the requirements does not affect test generation or test results.

To link the first formal requirement to the corresponding high-level requirement:

- 1 In `spec_model_final`, expand the requirement set named `Table1`.
- 2 Right-click the formal requirement with the **Index** of 1 and select **Select for Linking with Requirement**.
- 3 Expand the `AP_Controller_Reqs` requirement set.
- 4 Right-click the requirement with an **ID** of 1 and click **Create a link from "1: Autopilot mode is OFF" to "1: High Level: Autopilot Con..."**.

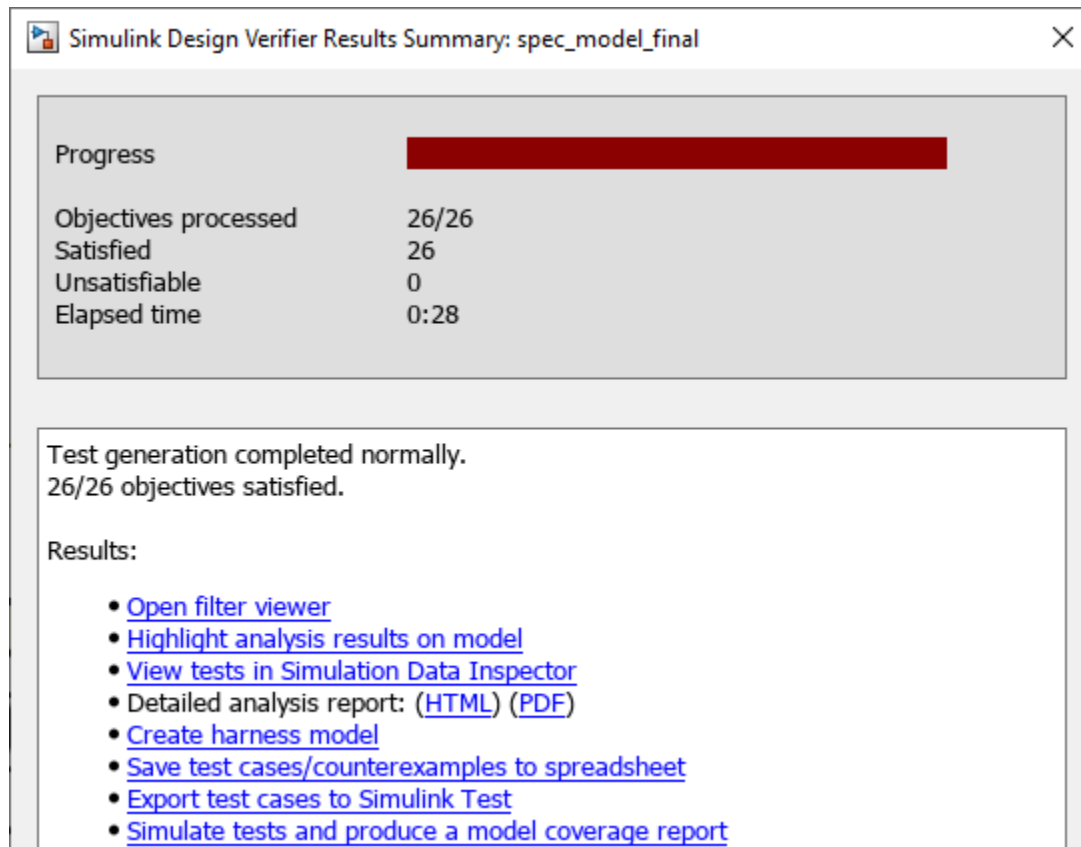
The link type defaults to `Related` to. For more information on link types, see "Link Types" (Requirements Toolbox).

### Generate Tests from the Updated Model

Generate the tests from the updated specification model by using the options defined previously.

```
opts = sldvoptions;
opts.Mode = "TestGeneration";
opts.ModelCoverageObjectives = "Decision";
[~, files] = sldvrun(spec_model,opts,true);
```

In this version of the specification model, the test objectives are satisfied.



### **Run the Tests on the Design Model**

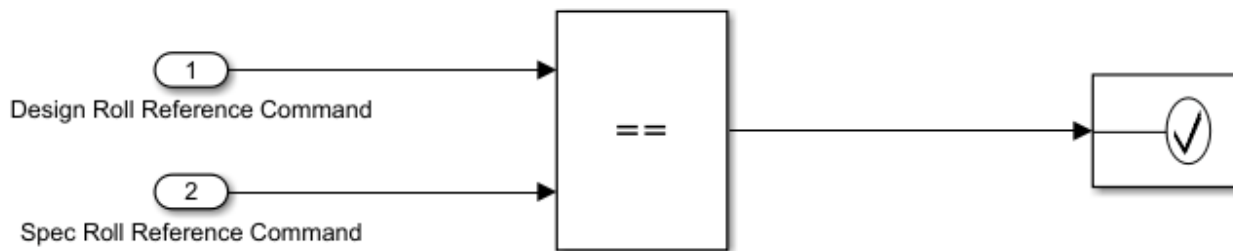
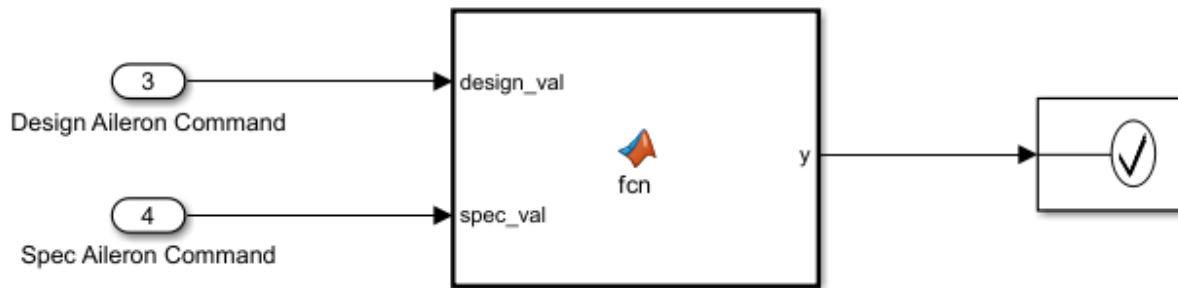
After you create tests that satisfy the test objectives, you can run the tests on the design model. In this example, the design model is the model for the aircraft autopilot controller, `sldvexRollApController`.

Before you run tests on the design model, you must interface the specification model with the design model. Typically, the specification model does not produce or use the same signals as the design model. These differences can be simple or abstract. For example, the design model might use different input and output signal types than the specification model, or you may want to compare a scalar output from the design model against a range in the specification model. As a result, you need to construct an interface between the design model and the specification model.

### **Interface the Design Model with the Specification Model**

In this example, the specification model `spec_model_final` and design model `sldvexRollApController` inputs can interface directly, but one of the outputs is different. `spec_model_final` represents the aileron command as a range of values, but the aileron command value produced by `sldvexRollApController` is a scalar double. The interface uses a MATLAB Function block to compare the aileron command values. The interface then verifies both outputs with Assertion blocks. Open the model, `spec_model_test_interface`, to view the interface.

```
test_interface = "spec_model_test_interface";  
open_system(test_interface);
```



The MATLAB Function block compares the two signals by using this code:

```
function y = fcn(design_val, spec_val)
switch spec_val
case Ail_Cmd.All
    y = true;
case Ail_Cmd.Zero
    y = (design_val == 0);
otherwise
    y = false;
end
```

### Run the Updated Tests on the Design Model

To test and verify the design model, create a harness model that contains the:

- 1 Specification model
- 2 Design model

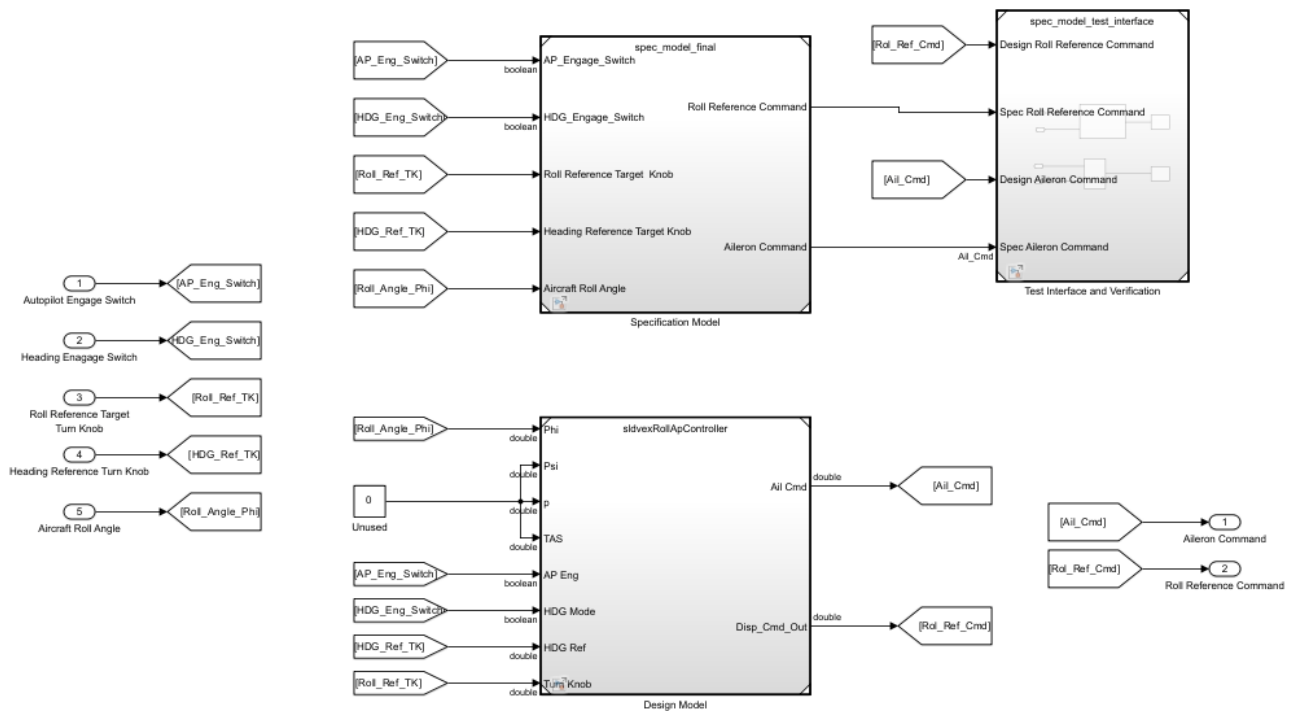
### 3 Test interface and verification model

In the harness model, attach the models together. Then run the tests on the design model and verify the outputs correspond to the requirements in the harness model.

To view the harness model, open the model, `sldvexDesignHarnessFinal`.

```
harness_model = "sldvexDesignHarnessFinal";
open_system(harness_model);
```

Like with the interface model, not all design model inputs may directly correspond to specification model inputs. In this example, the harness model prepares the design model for testing with the five inputs specified by the specification model.



Run the updated tests on the design model from within the harness model. Use the `sldvruntest` function to run the tests and save the results. If you have Simulink Coverage™, you can view the results of the tests from the output of `sldvruntest` in a coverage report. View the coverage report by using the `cvhtml` (Simulink Coverage) function.







```
cvopts = sldvruntestopts;
cvopts.coverageEnabled = true;
[finalData, finalCov] = sldvruntest(...
    harness_model, files.DataFile, cvopts);
cvhtml("finalCov", finalCov);
```

The coverage report shows that full coverage is achieved on the design model, `sldvexRollApController`.



## Summary

### Model Hierarchy/Complexity Test 1

|  |   | Decision |   | Execution |   |
|--|---|----------|---|-----------|---|
| 1. <a href="#">sldevexRollApController</a> | 8 | 100%     |  | 100%      |  |
| 2. ... <a href="#">Roll Reference</a>      | 5 | 100%     |  | 100%      |  |
| 3. .... <a href="#">Latch Phi</a>          | 1 | 100%     |  | 100%      |  |

```
bdclose("all");
slreq.clear;
```

### See Also

Requirements Table

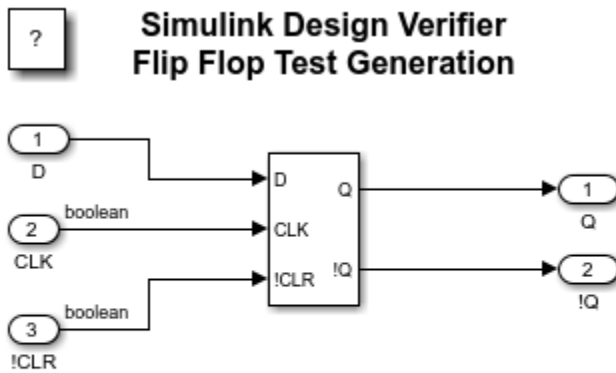
### Related Examples

- “What Is a Specification Model?” on page 7-60
- “Add Assumptions to Requirements” (Requirements Toolbox)
- “Export Tests from Models That Contain Requirements Table Blocks with Simulink Design Verifier” on page 13-30

## Flip Flop Test Generation

This example shows how to generate test cases that achieve complete model coverage for a flip-flop. The outcome of each model coverage point in this example model is a test objective. If you configure Simulink Design Verifier to generate the fewest test cases, it will satisfy as many objectives as possible in each test case.

```
open_system('sldvdemo_flipflop');
```

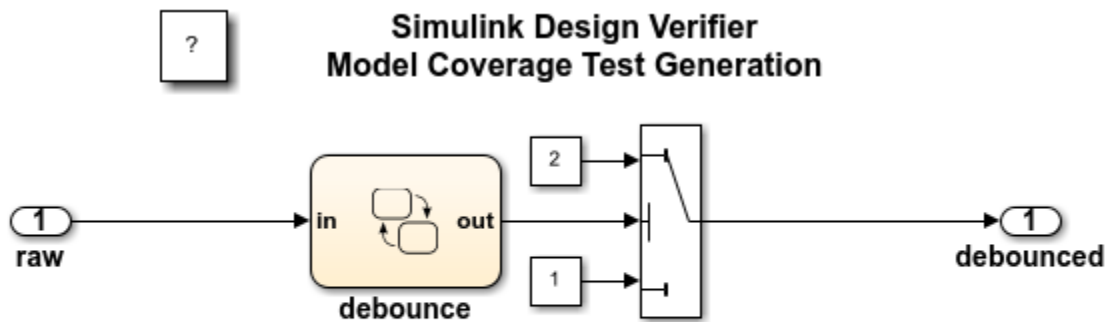


Copyright 2006-2023 The MathWorks, Inc.

## Model Coverage Test Generation

This example shows how to generate test cases that achieve complete model coverage for a debouncer. The outcome of each model coverage point in this example model is a test objective. If you configure Simulink Design Verifier to generate the fewest test cases, it will satisfy as many objectives as possible in each test case.

```
open_system('sldvdemo_debounce_modelcov');
```

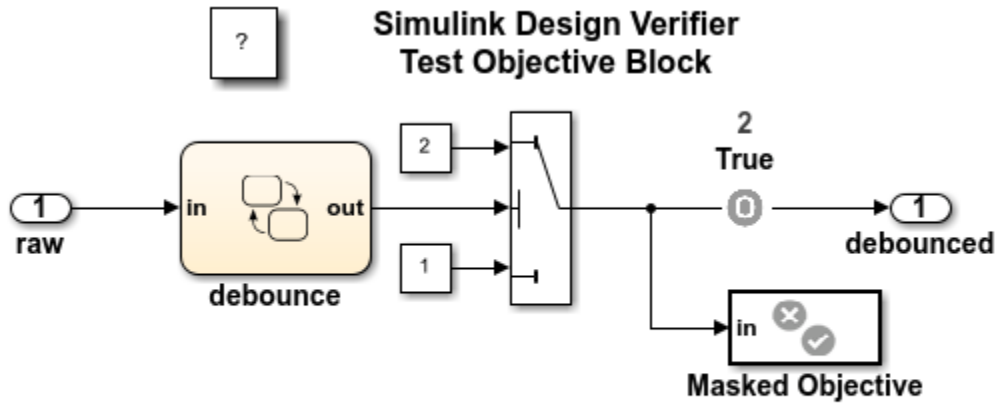


Copyright 2006-2023 The MathWorks, Inc.

## Test Objective Block

This example shows the use of two custom Test Objective blocks. The block "True" forces the output signal to be 2. The block "Edge" inside "Masked Objective" specifies that the output signal transition from 2 to 1.

```
open_system('sldvdemo_debounce_testobjblks');
```

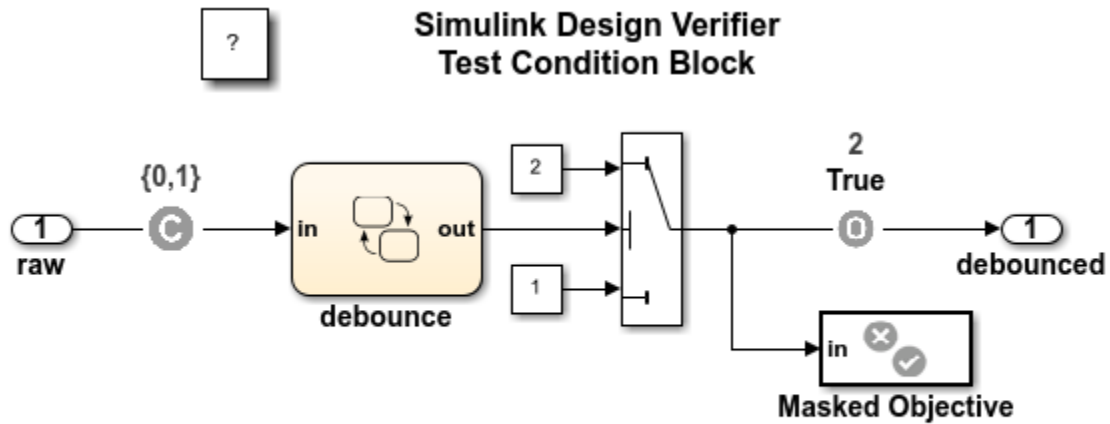


Copyright 2006-2023 The MathWorks, Inc.

## Test Condition Block

This example shows how to constrain input values. The Test Condition block forces the input value to be either 0 or 1.

```
open_system('sldvdemo_debounce_testconblk');
```

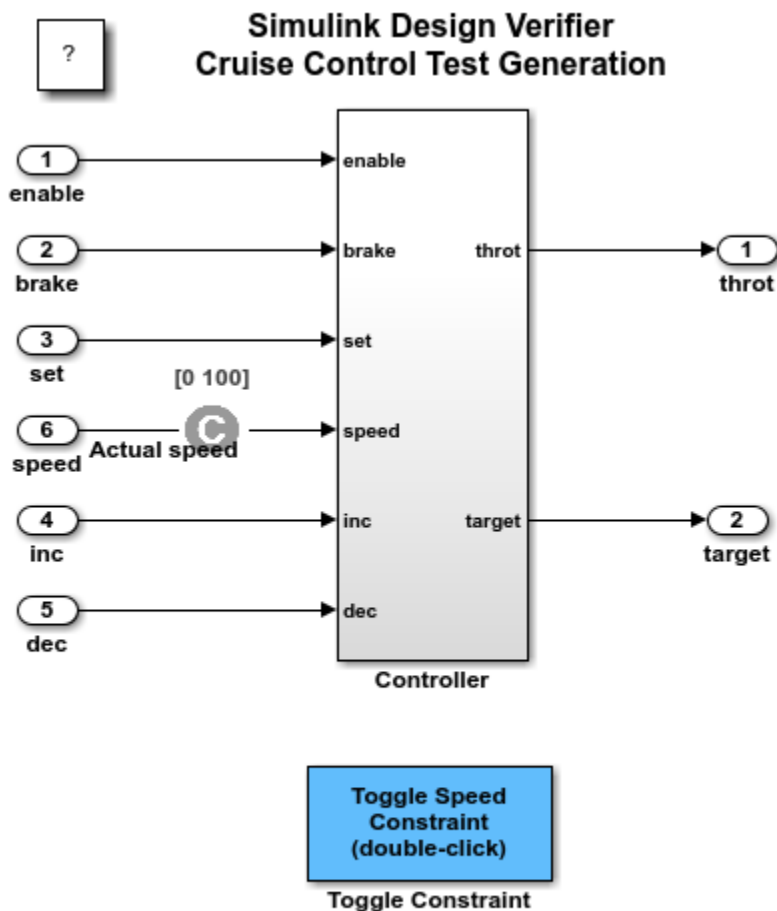


Copyright 2006-2023 The MathWorks, Inc.

## Cruise Control Test Generation

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

```
open_system('sldvdemo_cruise_control');
```

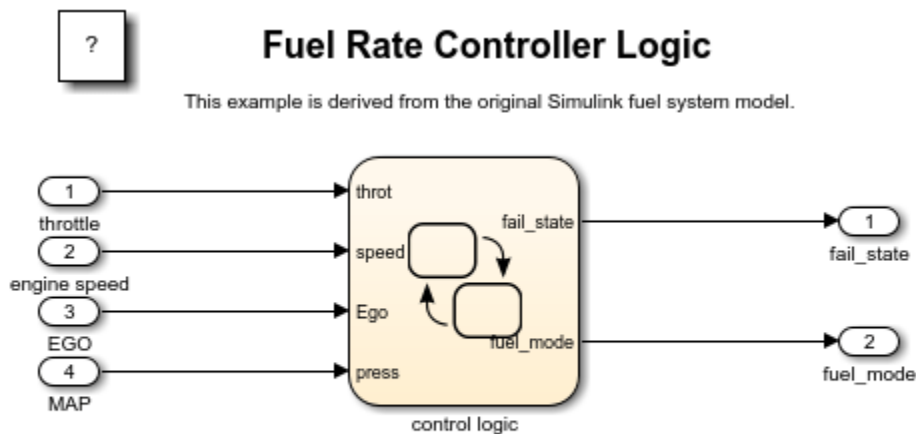


Copyright 2006-2023 The MathWorks, Inc.

## Fuel Rate Controller Logic

This example shows how to generate test cases that satisfy Decision, Condition, and MCDC coverage. Simulink® Design Verifier™ automatically generates test data and proves properties of models. It produces sequences of input values that satisfy a testing criteria or demonstrate a counterexample of a proof. The configuration options associated with the model specify the objectives of the analysis. When you analyze the model, Simulink Design Verifier uses exhaustive searching techniques to generate input data. When successful, it generates test data and creates a new harness model containing a Signal Builder block with the data values that satisfy the analysis objectives. NOTE: The complexity of this model might prevent test generation from completing in the allotted time. You can stop test generation and generate partial results, or you can extend the time limit by editing the Simulink Design Verifier options.

```
open_system('sldvdemo_fuelsys_logic');
```



Copyright 2006-2023 The MathWorks, Inc.

## Extend an Existing Test Suite

This example shows how to use Simulink® Design Verifier™ to extend an existing test suite to obtain missing model coverage.

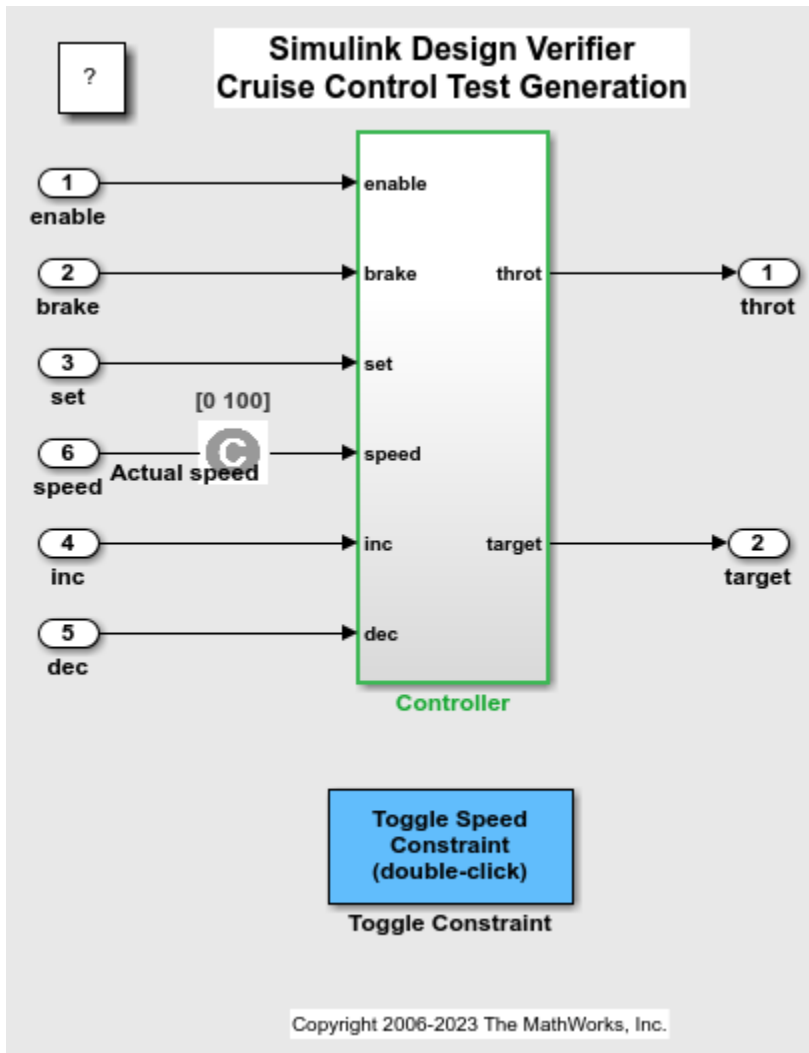
You analyze an example model and generate test suite to achieve full coverage. Then, modify the model such that test cases no longer achieve full coverage. Finally, you analyze the modified model to obtain missing coverage by using Simulink® Design Verifier™.

### Generate an Initial Test Suite

Analyze the `sldvdemo_cruise_control` model and generate a test suite that achieves full model coverage. To analyze the model to generate test cases that provide model coverage, use the `sldvrun` function. Set the design verification parameters with `sldvoptions`.

```
open_system('sldvdemo_cruise_control');
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'MCDC';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts, true);
```





The test generation analysis result appears in the Simulink Design Verifier Results Summary window.

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











### Verify Complete Coverage

The `sldvrntest` function simulates the model with the existing test suite. The `cvhtml` function produces a coverage report that indicates the initial coverage of the `sldvdemo_cruise_control` model.

```
open_system('sldvdemo_cruise_control');
[ outData, initialCov ] = sldvrntest('sldvdemo_cruise_control', files.DataFile, [], true);
cvhtml('Initial coverage',initialCov);
close_system('sldvdemo_cruise_control',0);
```

## Summary

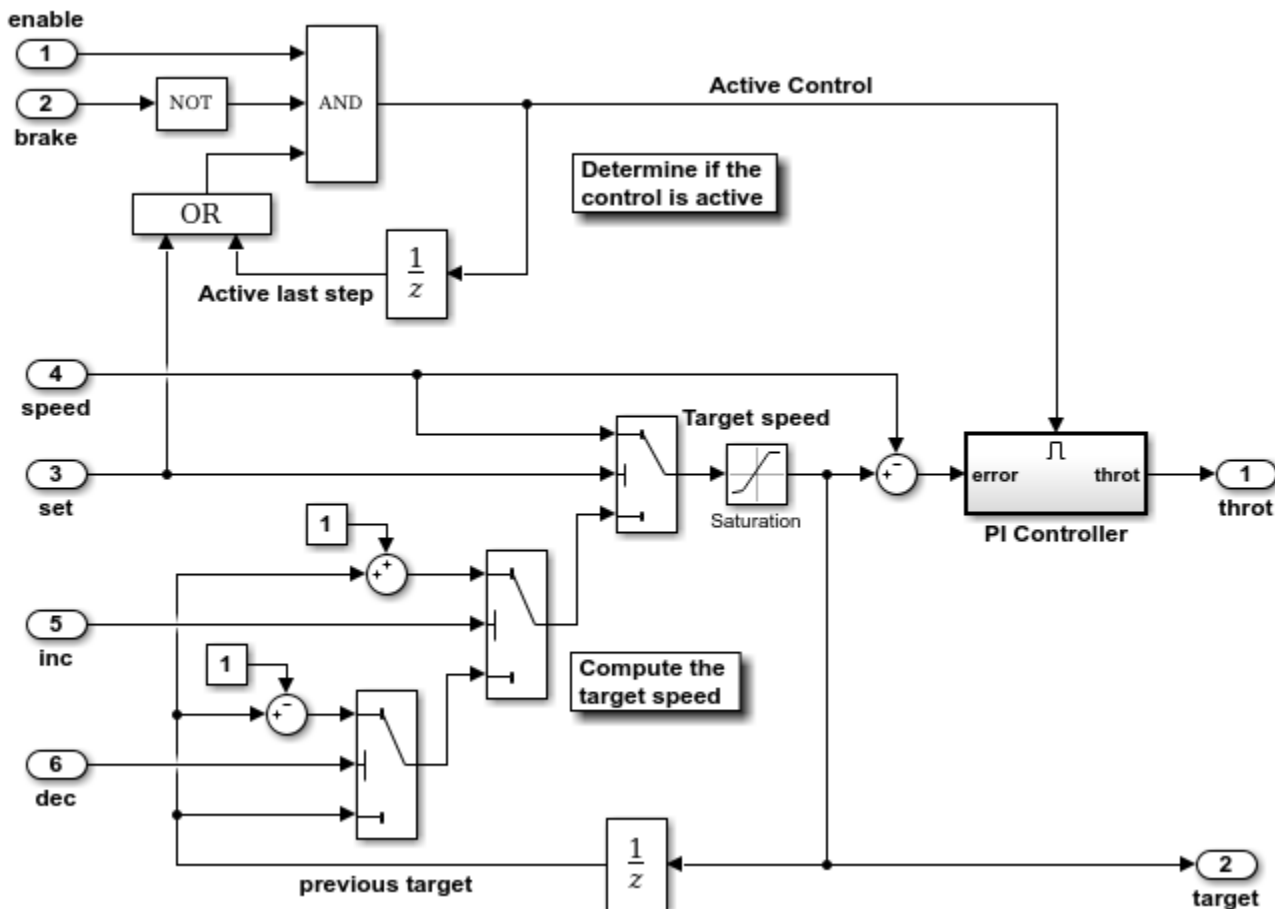
### Model Hierarchy/Complexity Test 1

|  | Decision   | Condition  | MCDC   | Execution  |
|--|--|--|--|--|
| 1. <a href="#">sldvdemo_cruise_control</a> | 8 100%  | 100%  | 100%  | 100%  |
| 2. .... <a href="#">Controller</a>         | 7 100%  | 100%  | 100%  | 100%  |
| 3. .... <a href="#">PI Controller</a>      | 4 100%  | NA   | NA   | 100%  |

### Modify the Model

Load the modified `sldvdemo_cruise_control_mod` model. The controller target speed value is limited to 70, by using a Saturation block.

```
load_system 'sldvdemo_cruise_control_mod';
load_system 'sldvdemo_cruise_control_mod/Controller';
```













## Measure the Coverage Achieved by the Existing Test Suite

The `sldvrntest` function simulates the modified `sldvdemo_cruise_control_mod` model with an existing test suite and inputs identical to `sldvdemo_cruise_control` model. The `cvhtml` function produces a coverage report that indicates the modified `sldvdemo_cruise_control_mod` model no longer achieves full coverage.

```
[ outData, startCov ] = sldvrntest('sldvdemo_cruise_control_mod', files.DataFile, [], true);
cvhtml('Coverage with the original testsuite',startCov);
```

## Summary

| Model Hierarchy/Complexity                     | Test 1 | Test 1  |  |  |  |
|--|--------|---|--|--|--|
|  |        | Decision  | Condition  | MCDC   | Execution  |
| 1. <a href="#">sldvdemo_cruise_control_mod</a> | 10     | 88%  | 100%  | 100%  | 100%  |
| 2. .... <a href="#">Controller</a>             | 9      | 88%  | 100%  | 100%  | 100%  |
| 3. .... <a href="#">PI Controller</a>          | 4      | 67%  | NA   | NA   | 100%  |

## Extend an Existing Test Suite

To achieve full model coverage, the `sldvgencov` function analyzes the model and extends the existing test suite.


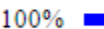
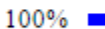
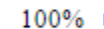

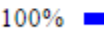
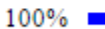
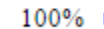

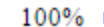
```
[ status, covData, files ] = sldvgencov('sldvdemo_cruise_control_mod', opts, true, startCov);
```

## Verify Complete Coverage

Verify that the new test suite achieves full coverage for the `sldvdemo_cruise_control_mod` modified model. The `sldvrntest` function simulates the modified model with the extended test suite. The `cvhtml` report shows the total coverage achieved by the `sldvdemo_cruise_control_mod` model.

```
[ additionalOut, additionalCov ] = sldvrntest('sldvdemo_cruise_control_mod', files.DataFile, []
totalCov = startCov + additionalCov;
cvhtml('With additional coverage',totalCov);
```

## Summary

| Model Hierarchy/Complexity                     | Test 1 | Test 1   |  |   |  |
|--|--------|--|--|---|--|
|  |        | Decision   | Condition  | MCDC  | Execution  |
| 1. <a href="#">sldvdemo_cruise_control_mod</a> | 10     | 100%  | 100%  | 100%  | 100%  |
| 2. .... <a href="#">Controller</a>             | 9      | 100%  | 100%  | 100%  | 100%  |
| 3. .... <a href="#">PI Controller</a>          | 4      | 100%  | NA   | NA  | 100%  |

To complete the example, close the model.

```
close_system('sldvdemo_cruise_control_mod');
```

## Defining and Extending Existing Tests Cases

This example shows how Simulink® Design Verifier™ can extend test cases with additional time steps to efficiently generate complete test suites.

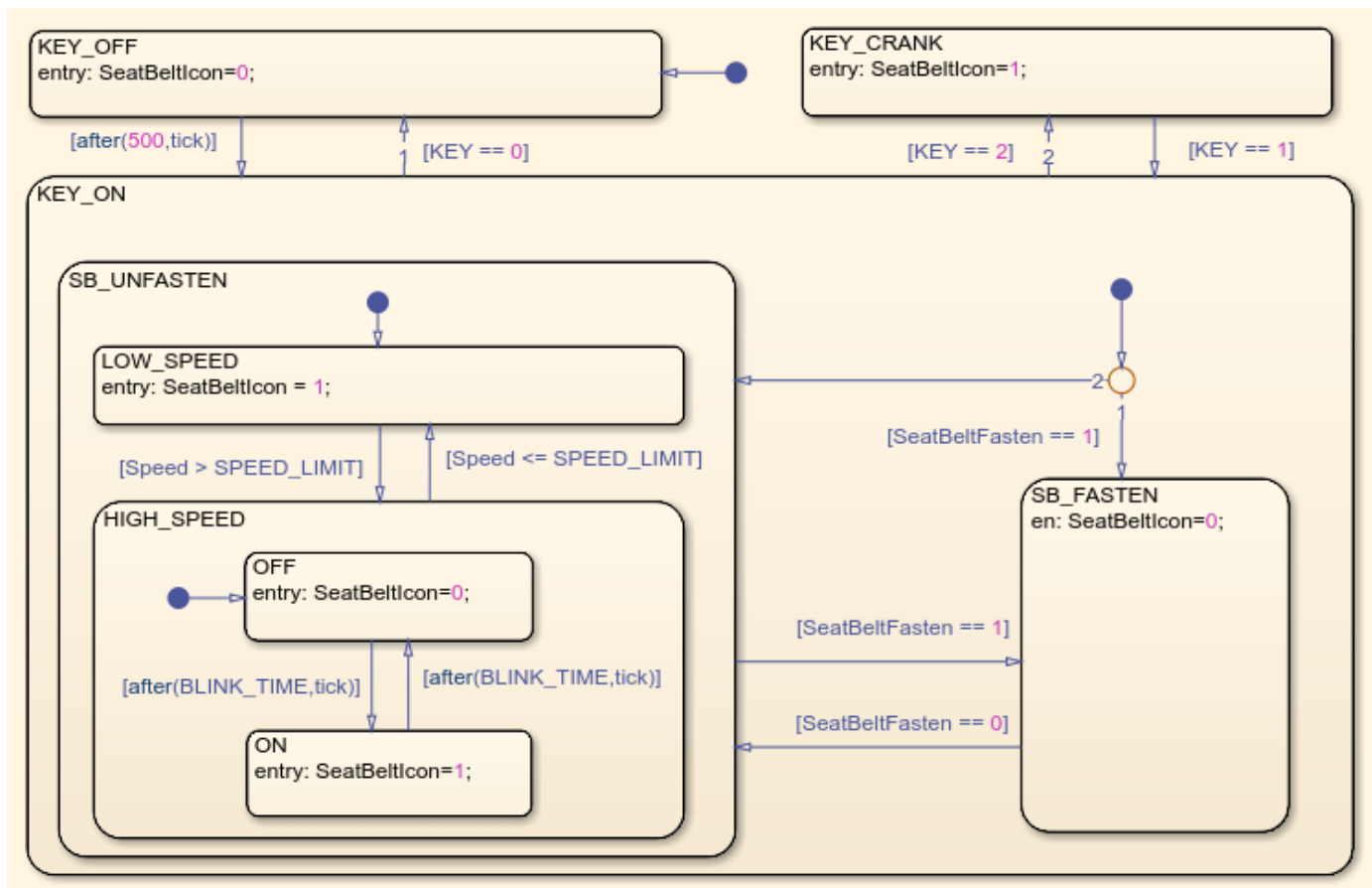
The example starts with a model containing time-delay characteristics that make test generation challenging. By creating a default test harness model and manually authoring one test, the critical obstacle to efficient test generation is removed. Simulink Design Verifier takes as input the logged values from the harness model and efficiently extends this test to create a complete test suite.

### Model Characteristics That Motivate Test Case Extension

The `sldvdemo_sbr_extend_design` model includes the Stateflow® Chart SBR that uses temporal logic so that very long test cases are required to make a transition from the `KEY_OFF` state to the `KEY_ON` state. This type of time-delay characteristic is common in designs where a delay is used to reject spurious behavior or to wait for a physical system or user to respond. In this design, satisfying the temporal logic in this transition is a common obstacle to testing any of the states and transitions within the `KEY_ON` state.

Fortunately, this type of time-delay characteristic is usually easy to identify and satisfy with a manually authored test case.

```
open_system('sldvdemo_sbr_extend_design');  
sf('Open',sldvdemo_ssid_to_sfid('sldvdemo_sbr_extend_design/SBR',11));
```



### Creating a Harness Model and Defining Starting Tests

The Simulink Design Verifier function `sldvmakeharness` creates a harness model with a block that generates input values to the test model included by way of a Model block.

You can modify the test data in a harness model by manually editing the data values using the Signal Builder user interface. You can also add more test cases by creating new signal groups in the block. Alternatively, you can use the `signalbuilder` command to programmatically accomplish the same thing.

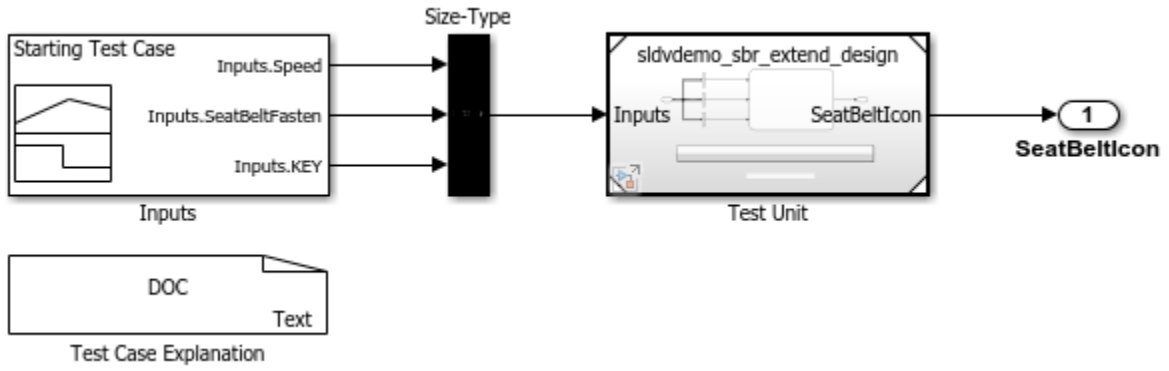
In this example, you specify a test case that keeps the system in the `KEY_OFF` state for 5 seconds:

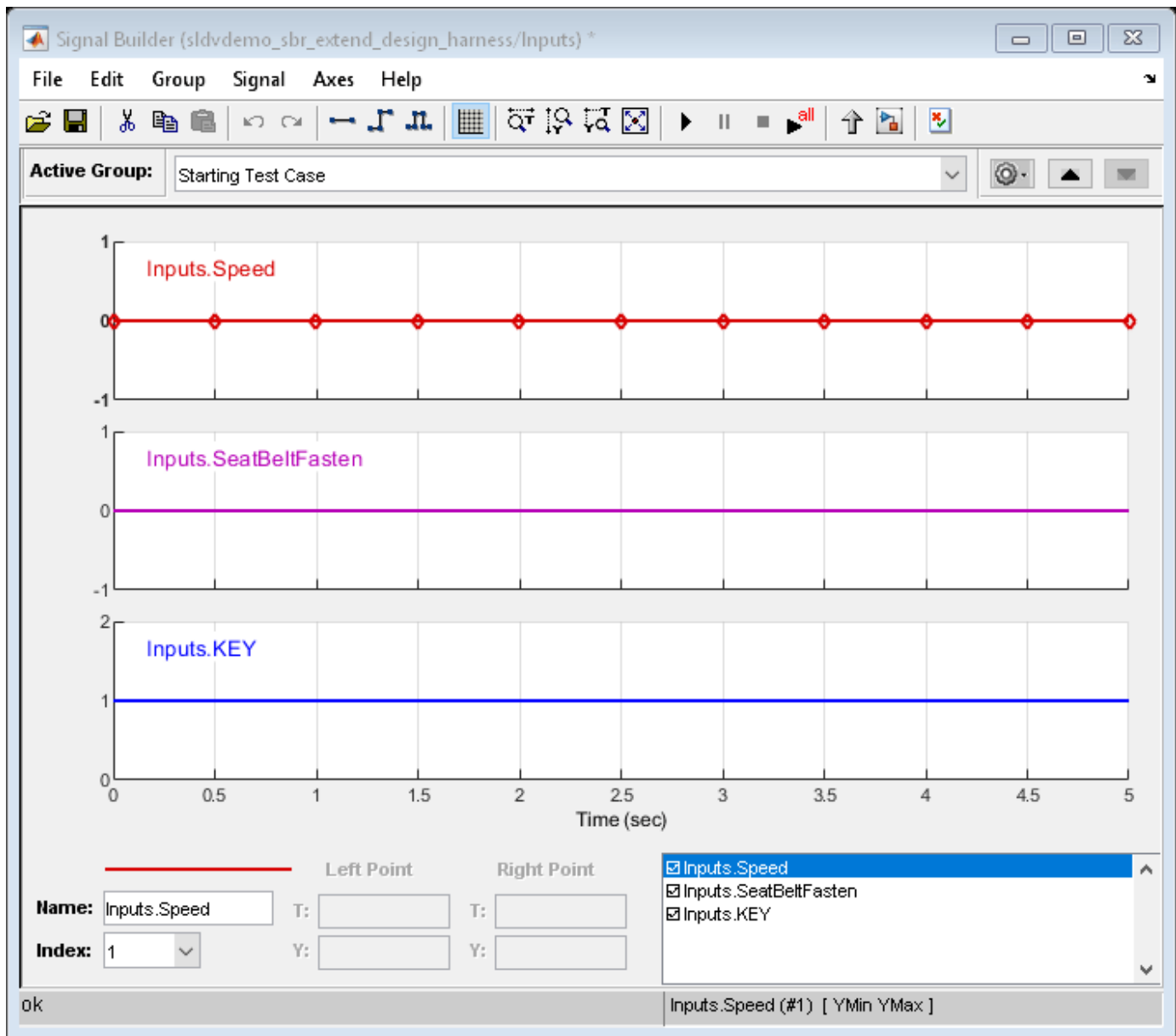
```
[~, harnessModelFilePath] = sldvmakeharness('sldvdemo_sbr_extend_design', [], [], true);
[~, harnessModel] = fileparts(harnessModelFilePath);
```

```
startingTestTime = 0:0.5:5;
startingTestData = cell(3, 1);
lengthStartingTest = length(startingTestTime);
startingTestData{1} = zeros(1, lengthStartingTest);
startingTestData{2} = zeros(1, lengthStartingTest);
startingTestData{3} = ones(1, lengthStartingTest);
```

```
signalBuilderBlock = sldvdemo_signalbuilder_block(harnessModel);
signalbuilder(signalBuilderBlock, 'Append', ...
    startingTestTime, startingTestData, ...
```

```
{'Inputs.Speed', 'Inputs.SeatBeltFasten', 'Inputs.KEY'}, 'Starting Test Case');
signalbuilder(signalBuilderBlock, 'ActiveGroup', 2);
open_system(signalBuilderBlock);
```





### Logging Starting Tests

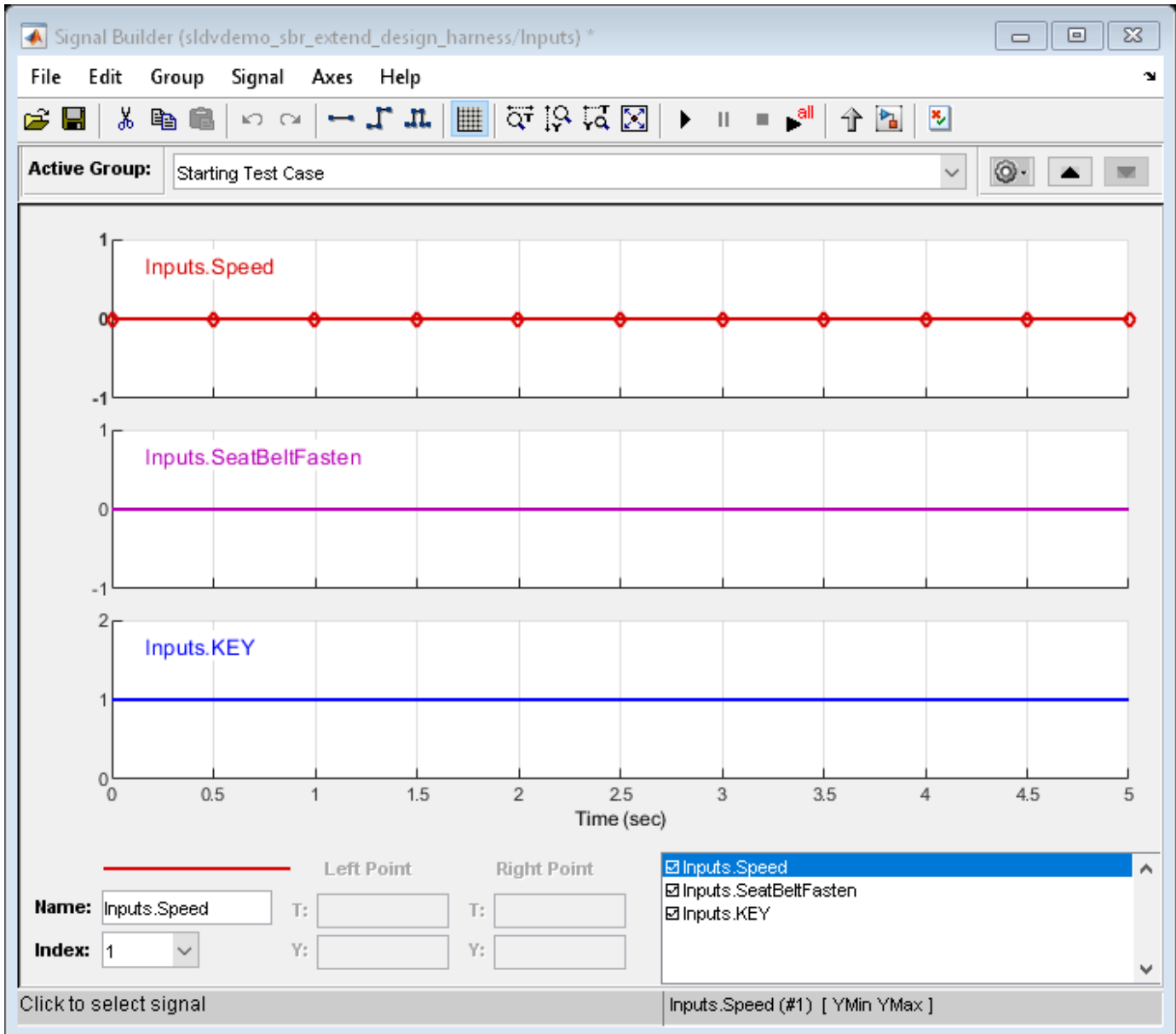
In order to leverage the starting test case defined above, you use the `sldvlogsignals` function to capture the input values in the necessary logged data format.

The first input to `sldvlogsignals` is the path to a Model block, and the second input is the index of signal group(s) within the harness model. When you invoke `sldvlogsignals`, the parent model that contains the Model block is simulated.

The parent model is not restricted to Simulink Design Verifier harness models. Alternatively, you might log data from a closed-loop simulation model that uses a Model block to include the controller so that controller test cases more realistically reflect the continuous time behavior expected in the closed-loop system.



```
[~, modelBlock] = find_mdrefs(harnessModel, false);
loggeddata = sldvlogsignals(modelBlock{1},2);
```



### Extending Existing Tests During Test Generation

Before you can use existing test data during test generation, the data must be saved to a MAT-file. You enable test case extension in the Test Generation pane of the Simulink Design Verifier configuration parameters. Select **Extend existing test cases**, and specify the MAT file in the **Data file** field.

Generated tests either extend one of the starting test cases with one or more new time steps or will specify one or more time steps starting from the initial, or default, configuration.

```
save('existingtestcase.mat', 'loggeddata');
```

```

opts = sldvoptions;
opts.ExtendExistingTests = 'on';
opts.ExistingTestFile = 'existingtestcase.mat';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';

[~, fileNames] = sldvrun('sldvdemo_sbr_extend_design', opts, true);

```

### Verifying Complete Coverage







The `sldvruntest` function verifies that the new test suite achieves complete model coverage. The `cvhtml` function produces a coverage report that indicates 100% Decision coverage is achieved with the generated test vectors.

```

[~, finalCov] = sldvruntest('sldvdemo_sbr_extend_design', fileNames.DataFile, [], true);
cvhtml('Final Coverage', finalCov);

```

## Summary

| Model Hierarchy/Complexity                    | Test 1  | Decision  |
|---|---------|---|
| 1. <a href="#">sldvdemo_sbr_extend_design</a> | 21 100% |    |
| 2. ... <a href="#">SBR</a>                    | 20 100% |   |
| 3. .... <a href="#">SF: SBR</a>               | 19 100% |  |
| 4. .... <a href="#">SF: KEY_ON</a>            | 13 100% |  |
| 5. .... <a href="#">SF: SB_UNFASTEN</a>       | 8 100%  |  |
| 6. .... <a href="#">SF: HIGH_SPEED</a>        | 4 100%  |  |

### Clean Up

To complete the demo, close all models and remove the saved logged data file.

```

close_system(harnessModel,0);
close_system('sldvdemo_sbr_extend_design');
delete('existingtestcase.mat');

```

## Using Existing Coverage Data During Subsystem Analysis

This example shows how Simulink® Design Verifier™ can target its analysis to a single subsystem within a continuous-time closed-loop simulation and generate test cases for missing coverage in that subsystem.

The example starts by measuring the coverage of a subsystem in a closed-loop simulation model. Simulink Design Verifier finds new test cases that achieve the missing coverage of the subsystem.

### Measure Coverage of the Subsystem

The `sldvdemo_autotrans` model is a closed-loop simulation model. The subsystem `ShiftLogic` is a Stateflow® chart and represents the controller part of this model. Test cases designed in the Signal Editor block `ManeuversGUI` drive the closed-loop simulation. You can use the `cvtest` and `cvsim` functions to measure the model coverage achieved for this subsystem inside the closed-loop simulation model. In this example, specifying the input to `cvtest` as a path to the subsystem rather than to the model name results in measuring the coverage for the subsystem only. Also, the second input to `cvsim` specifies the time interval to simulate the model and it is derived from the time range of the current pane in the block `ManeuversGUI`.

The `cvhtml` function produces a report that indicates that 87% Decision, 67% Condition, and 33% MCDC coverage is achieved by simulating the test case authored in the block `ManeuversGUI`.

```
open_system('sldvdemo_autotrans');
open_system('sldvdemo_autotrans/ManeuversGUI');

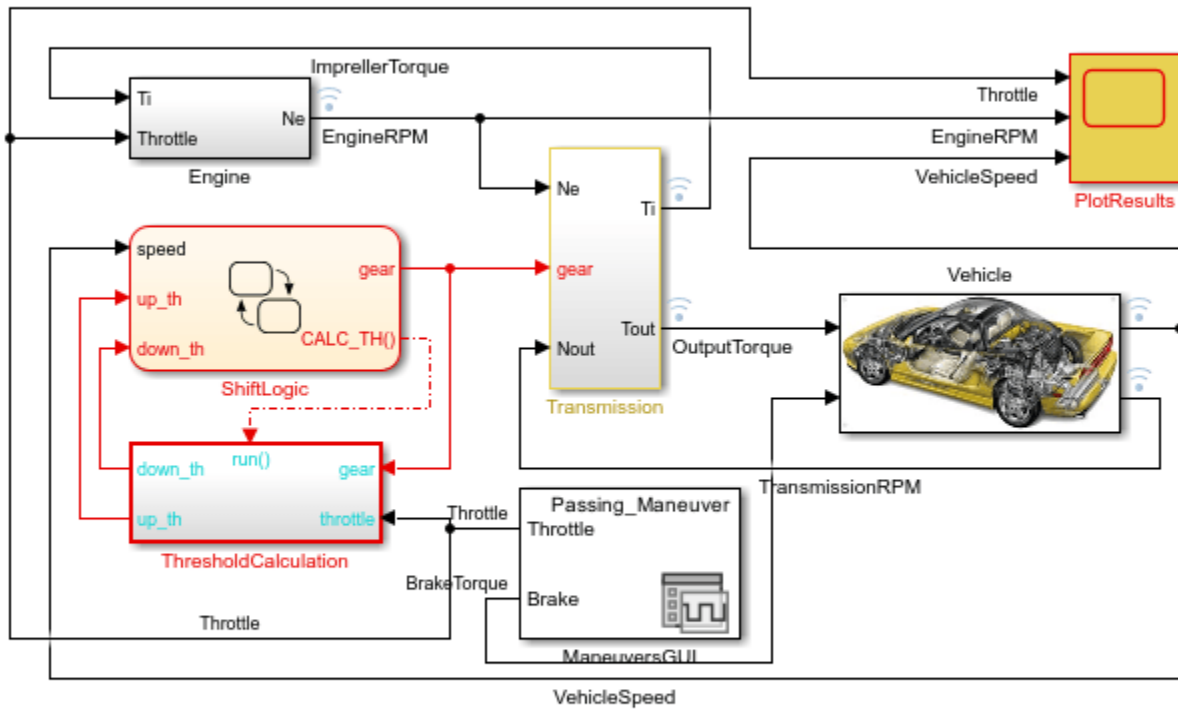
test = cvtest('sldvdemo_autotrans/ShiftLogic');
test.settings.decision = 1;
test.settings.condition = 1;
test.settings.mcdc = 1;

signalEditorBlock = sldvdemo_signaleditor_block('sldvdemo_autotrans');
signalEditorTime = sldvdemo_signaleditor_DataTime(signalEditorBlock);
simulationStopTime = signalEditorTime{1,1}(end);

existingCovData = cvsim(test,[0 simulationStopTime]);
cvhtml('Existing Coverage', existingCovData);
```



## Simulink Design Verifier Modeling an Automatic Transmission Controller



Double-click on ManeuversGUI and select a maneuver



Copyright 1990-2019 The MathWorks, Inc.

### Find Test Cases for Missing Coverage

To use existing coverage data during test generation, save existing coverage data to a .cvt coverage data file. You can use existing coverage data by specifying the coverage data path in the **Coverage data file** parameter and setting **Ignore objectives satisfied in existing coverage data** parameter to on in the **Test Generation** pane of Simulink Design Verifier configuration parameters.

In this example, the first input to `sldvrun` specifies the subsystem to analyze. Instructing Simulink Design Verifier to analyze a subsystem is beneficial when the controller part of a model needs to be tested separately or when you want to divide the analysis of a large model into smaller, manageable parts.

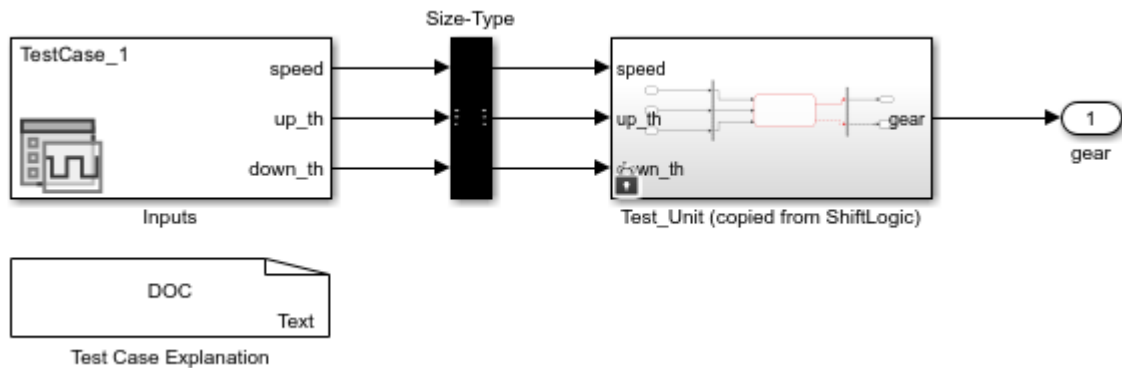
As you can see in the report, Simulink Design Verifier only finds test cases for the coverage objectives that are not covered in the existing coverage file. Notice that 4 coverage objectives in the subsystem `ShiftLogic` are proven to be unsatisfiable. This is expected because the logic inside the Stateflow chart `ShiftLogic` uses temporal events and since this chart updates at every sample time, usage of temporal conditions should be satisfactory. Also note that, dead code within a subsystem will always be a dead code in the model containing that subsystem.

To generate the harness model, Simulink Design Verifier extracts the contents of the subsystem ShiftLogic into a Test Unit component fed by a Signal Editor block containing the generated test cases.

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

```
opts = sldvoptions;
opts.IgnoreCovSatisfied = 'on';
opts.CoverageDataFile = 'existingcov.cvt';
opts.ModelCoverageObjectives = 'MCDC';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'on';
```

```
[status, fileNames] = sldvrun('sldvdemo_autotrans/ShiftLogic',opts,true);
[~, harnessModel] = fileparts(fileNames.HarnessModel);
open_system(harnessModel);
```



## Clean Up

To complete the demo, close all models and remove the saved coverage data file.

```
close_system('sldvdemo_autotrans');
close_system(fileNames.ExtractedModel,0);
close_system(fileNames.HarnessModel,0);
delete('existingcov.cvt');
```

## Creating and Executing Test Cases

This example shows how to use Simulink® Design Verifier™ functions to log input signals, create a harness model, generate test cases for missing coverage, merge harness models, and execute test cases.

The example starts by logging input signals to the component that implements the controller in its parent model and creating harness model for the controller from that logged data. You use Simulink Design Verifier to find a new test case that achieves the missing coverage. Then you merge the first harness model with the harness model generated after the Simulink Design Verifier analysis. Finally, you capture all test cases and execute the controller with those test cases in simulation mode and Software-In-the-Loop (SIL) mode, and compare the results using CGV API.

### Check Product Availability

This example requires a valid Stateflow® license. To demonstrate test execution in Software-In-the-Loop (SIL) mode it also requires valid Simulink® Coder™ and Embedded Coder™ licenses.

```
if ~license('test','Stateflow')
    return;
end

canUseSIL = license('test','Real-Time_Workshop') && ...
    license('test','RTW_Embedded_Coder');
```

### Logging Input Signals to the Component and Creating the Harness Model

The `slvndemo_powerwindow` model contains a power window controller and a low-order plant model. The component `slvndemo_powerwindow/power_window_control_system/control` is a Model block that references the model `slvndemo_powerwindow_controller`, which implements the controller with a Stateflow® chart.

To create a harness model for the controller with the signals that simulate the controller in the plant model, first log the input signals and then invoke harness generation with that logged data.

```
open_system('slvndemo_powerwindow');
load_system('slvndemo_powerwindow_controller');

loggedSignalsPlant = ...
    sldvlogsignals('slvndemo_powerwindow/power_window_control_system/control');

harnessModelFilePath = ...
    sldvmakeharness('slvndemo_powerwindow_controller',loggedSignalsPlant);
[~,harnessModel] = fileparts(harnessModelFilePath);

### Starting serial model reference simulation build.
### Successfully updated the model reference simulation target for: slvndemo_powerwindow_contro
```

Build Summary

Simulation targets built:

| Model                           | Action                       | Rebuild Reason                  |
|---------------------------------|------------------------------|---------------------------------|
| slvndemo_powerwindow_controller | Code generated and compiled. | slvndemo_powerwindow_controller |

```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 20.931s
### Starting serial model reference simulation build.
### Model reference simulation target for slvndemo_powerwindow_controller is up to date.
```

Build Summary

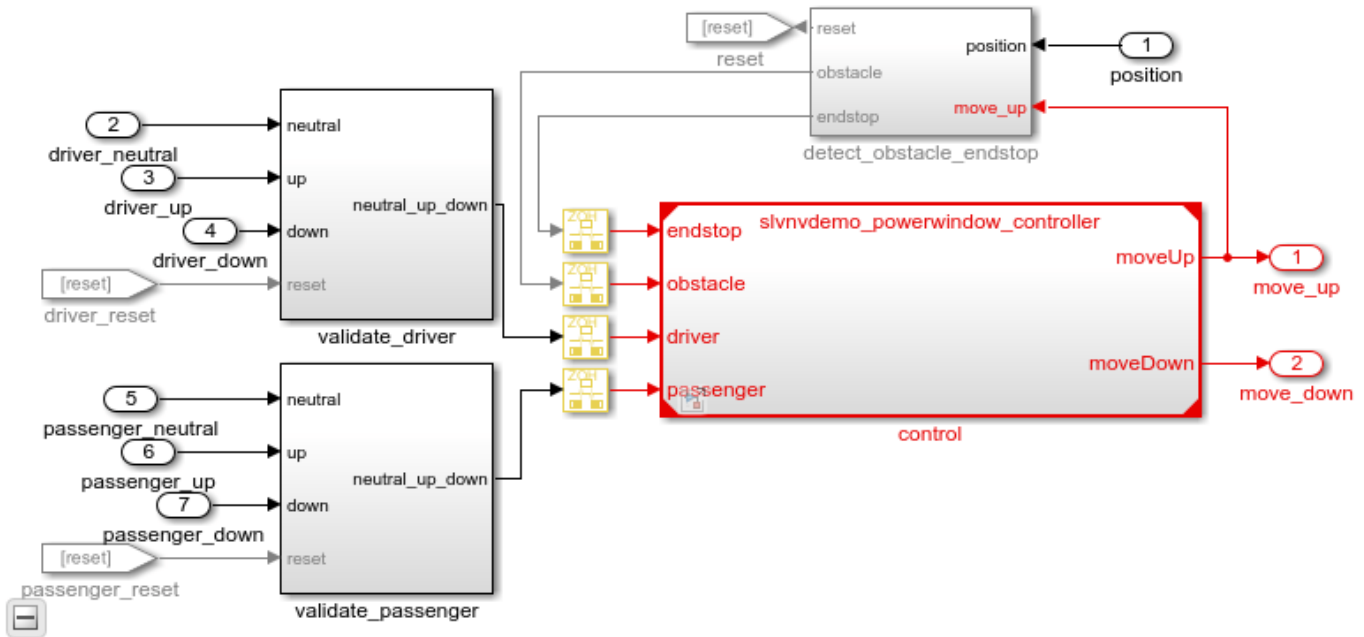
```
0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 0.55583s
### Starting serial model reference simulation build.
### Successfully updated the model reference simulation target for: slvndemo_powerwindow_contro
```

Build Summary

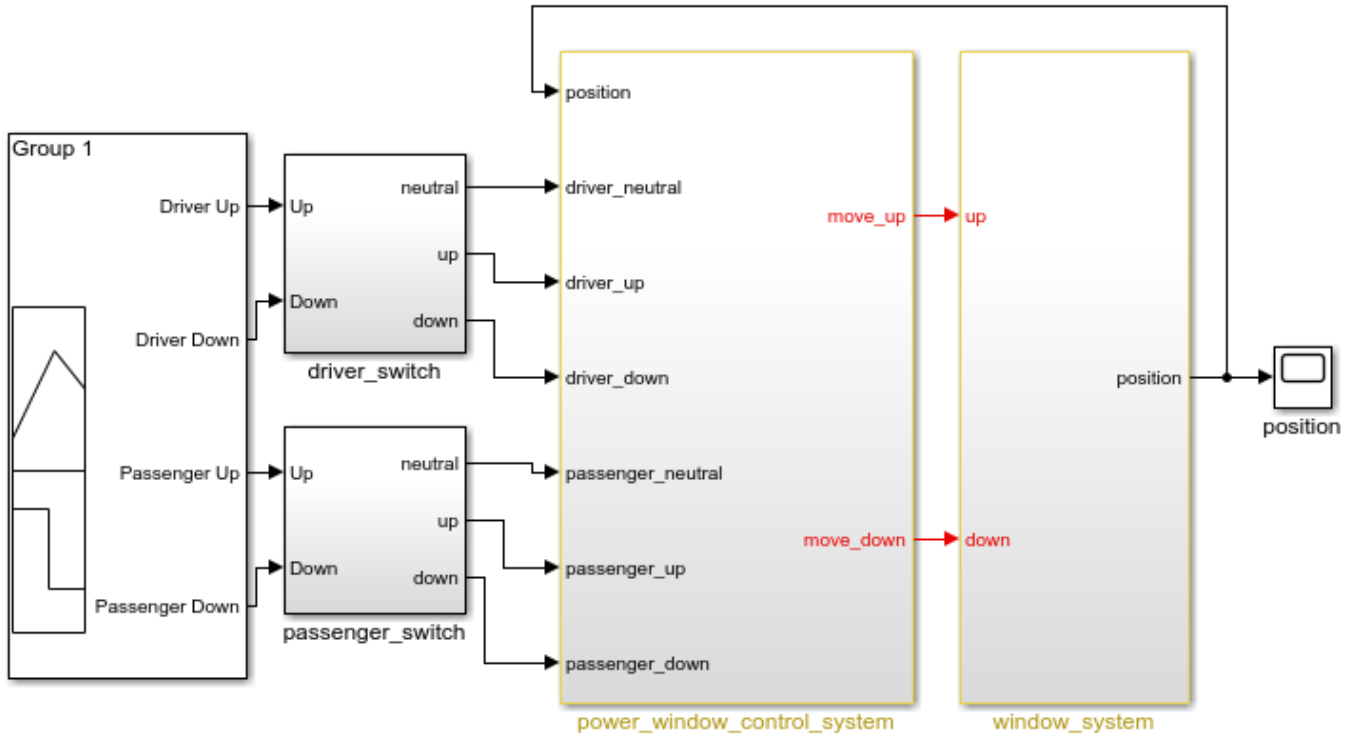
Simulation targets built:

| Model                           | Action                       | Rebuild Reason |
|---------------------------------|------------------------------|----------------|
| slvndemo_powerwindow_controller | Code generated and compiled. |                |

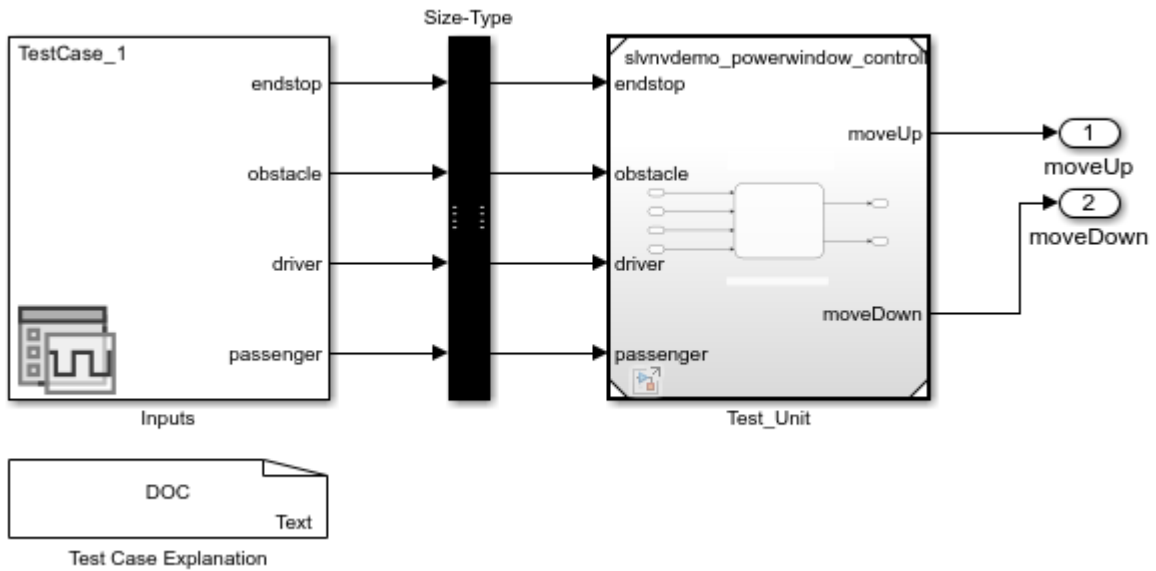
```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 13.322s
```



### Simulink Coverage Power Window Controller Hybrid System Model



Copyright 1990-2017 The MathWorks, Inc.





## Measuring the Coverage with Logged Signals

Use the `cvtest` and `cvsim` functions to measure the model coverage achieved for the controller model `slvndemo_powerwindow_controller` with the logged signals that are captured in the harness model.

The `cvhtml` function produces a report that indicates that 40% Decision, 35% Condition, and 10% MCDC coverage is achieved by simulating the test cases captured from the closed-loop model.

```
test = cvtest(harnessModel);
test.modelRefSettings.enable = 'On';
test.modelRefSettings.excludeTopModel = 1;

covDataFromLoggedSignals = cvsim(test);
cvhtml('Coverage with Logged Test Cases', covDataFromLoggedSignals);
```

## Finding Test Cases for Missing Coverage

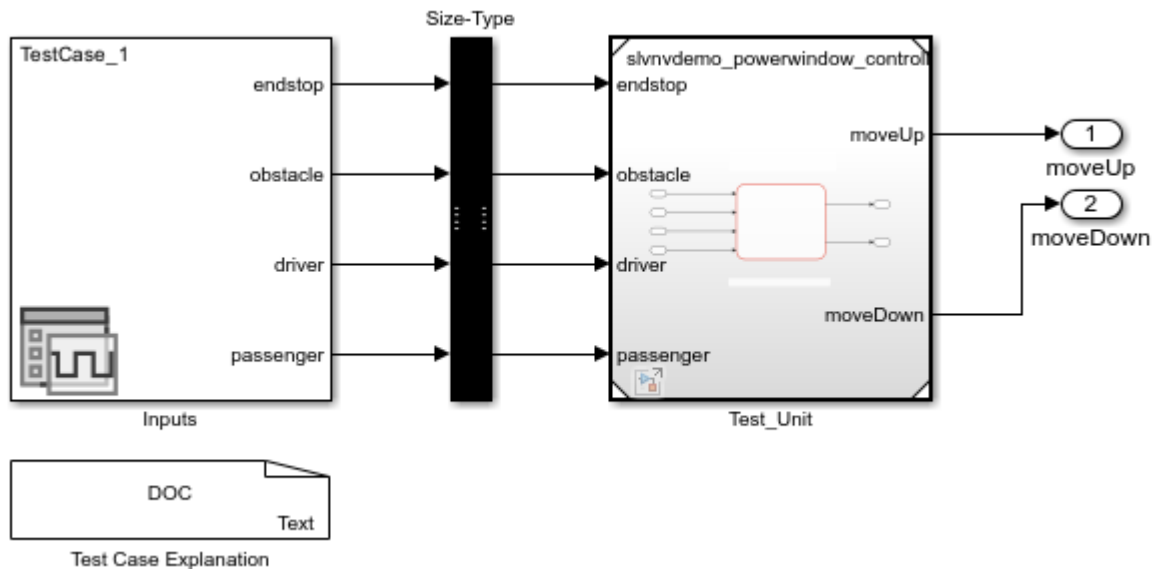
Before you can use existing coverage data during test generation, the data must be saved to a coverage data file (.cvt). You can use the existing coverage data by specifying the coverage data path in the **Coverage data file** parameter and setting the **Ignore objectives satisfied in existing coverage data** parameter to on in the **Test Generation** pane of Simulink Design Verifier configuration parameters.

As you can see in the report, Simulink Design Verifier restricts test generation to the coverage objectives that are not covered in the existing coverage file. Notice that 8 coverage objectives in the Stateflow chart `control` are proven to be unsatisfiable. This indicates unnecessary redundant logic that cannot be tested.

```
cvsave('existingCovFromLoggedSignals', covDataFromLoggedSignals);

opts = sldvoptions;
opts.IgnoreCovSatisfied = 'on';
opts.CoverageDataFile = 'existingCovFromLoggedSignals.cvt';
opts.ModelCoverageObjectives = 'MCDC';
opts.TestSuiteOptimization = 'LongTestcases';
opts.SaveHarnessModel = 'on';
opts.ModelReferenceHarness = 'on';
opts.MaxProcessTime = 500;

[status, fileNames] = sldvrun('slvndemo_powerwindow_controller', opts, true);
[~, newHarnessModel] = fileparts(fileNames.HarnessModel);
open_system(newHarnessModel);
```



### Merging Test Cases from Harness Models

Now use `sldvmergeharness` to combine generated test cases with logged test case. The command takes a list of harness models as arguments.

```
sldvmergeharness(harnessModel, newHarnessModel);
```

### Logging Test Cases of the Harness Model

In order to programmatically execute the model `slvndemo_powerwindow_controller` with the test cases captured in the merged harness model, first use the `sldvlogsignals` function to obtain the input values of all test cases in the necessary data format.

```
loggedSignalsMergedHarness = sldvlogsignals(harnessModel);
disp(loggedSignalsMergedHarness);
```

```
    LoggedTestUnitInfo: [1x1 struct]
        TestCases: [1x2 struct]
```

### Execute the Model in Simulation Mode with CGV API

Use the `sldvruncgvtest` function to execute the model `slvndemo_powerwindow_controller` in simulation mode, with test cases captured from the harness model.

```
runopts = sldvruntestopts('cgv');
disp(runopts);

runopts.cgvConn = 'sim';
cgvSim = sldvruncgvtest('slvndemo_powerwindow_controller',...
    loggedSignalsMergedHarness, runopts);

    testIdx: []
    allowCopyModel: 0
    cgvCompType: 'topmodel'
    cgvConn: 'sim'
```

```

Starting execution:
  ComponentType: topmodel
  Connectivity: sim
  InputData:
  C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex67947267\cgv_runtest\slvndemo_powerwindow_controller
End CGV execution: status completed.
Starting execution:
  ComponentType: topmodel
  Connectivity: sim
  InputData:
  C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex67947267\cgv_runtest\slvndemo_powerwindow_controller
End CGV execution: status completed.

```

### Execute the Model in Software-In-the-Loop (SIL) Mode with CGV API

Now use the `sldvruncgvtest` function to execute the model `slvndemo_powerwindow_controller` in SIL mode, with the same test cases.

```

if canUseSIL
    runopts.cgvConn = 'sil';
else
    % When SIL is not possible, the example runs another simulation.
    runopts.cgvConn = 'sim';
end
cgvSil = sldvruncgvtest('slvndemo_powerwindow_controller',...
    loggedSignalsMergedHarness,runopts);

```

```

Starting execution:
  ComponentType: topmodel
  Connectivity: sil
  InputData:
  C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex67947267\cgv_runtest\slvndemo_powerwindow_controller
### Starting build procedure for: slvndemo_powerwindow_controller
### Successful completion of build procedure for: slvndemo_powerwindow_controller

```

#### Build Summary

Top model targets built:

| Model                           | Action                       | Rebuild Reason                   |
|---------------------------------|------------------------------|----------------------------------|
| slvndemo_powerwindow_controller | Code generated and compiled. | Code generation information file |

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 10.463s

### Preparing to start SIL simulation ...

Building with 'Microsoft Visual C++ 2019 (C)'.

MEX completed successfully.

### Starting SIL simulation for component: slvndemo\_powerwindow\_controller

### Application stopped

### Stopping SIL simulation for component: slvndemo\_powerwindow\_controller

End CGV execution: status completed.

Starting execution:

ComponentType: topmodel

Connectivity: sil

InputData:

C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\28\tp27a1e6fc\sldv-ex67947267\cgv\_runtest\slvndemo\_powerwindow\_controller

```
### Starting build procedure for: slvndemo_powerwindow_controller
### Successful completion of build procedure for: slvndemo_powerwindow_controller
```

Build Summary

Top model targets built:

| Model                           | Action                       | Rebuild Reason                  |
|---------------------------------|------------------------------|---------------------------------|
| slvndemo_powerwindow_controller | Code generated and compiled. | Generated code was out of date. |

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 10.192s

### Preparing to start SIL simulation ...

Building with 'Microsoft Visual C++ 2019 (C)'.

MEX completed successfully.

### Starting SIL simulation for component: slvndemo\_powerwindow\_controller

### Application stopped

### Stopping SIL simulation for component: slvndemo\_powerwindow\_controller

End CGV execution: status completed.

### Compare Results of Simulation and SIL Modes

The `sldvruncgvtest` returns a `cgv.CGV` object after running tests. Use the CGV API to compare the results of executions in simulation and SIL modes for each test case designed in the harness model and show that they are equal.

```
for i=1:length(loggedSignalsMergedHarness.TestCases)
    simout = cgvSim.getOutputData(i);
    silout = cgvSil.getOutputData(i);

    [matchNames, ~, mismatchNames, ~ ] = ...
        cgv.CGV.compare(simout, silout);

    fprintf('\nTest Case(%d): %d Signals match, %d Signals mismatch', ...
        i, length(matchNames), length(mismatchNames));
end
```

Test Case(1): 4 Signals match, 0 Signals mismatch

Test Case(2): 4 Signals match, 0 Signals mismatch

### Clean Up

To complete the example, close all models.

```
close_system(harnessModel,0);
close_system(newHarnessModel,0);
close_system('slvndemo_powerwindow',0);
close_system('slvndemo_powerwindow_controller',0);
```

## Using Specified Input Minimum and Maximum Values as Constraints

This example shows how to use input port minimum and maximum values as analysis constraints by Simulink® Design Verifier™ during both test generation and property proving.

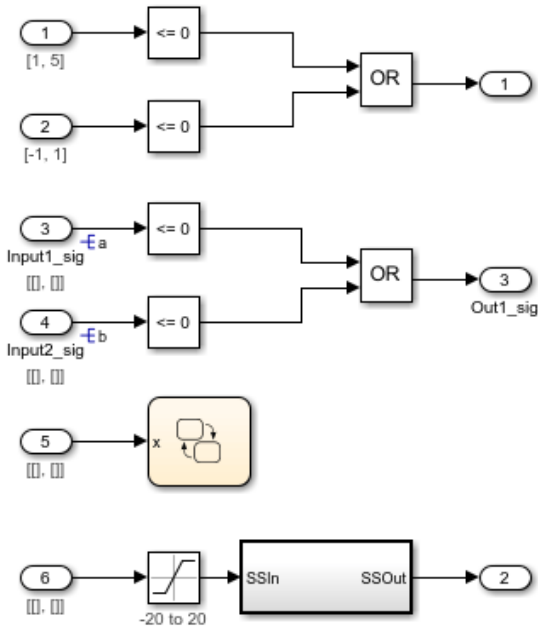
This model is preconfigured to generate tests for MCDC. The specified minimum and maximum values are displayed in square brackets. The constraints in this example prevent some of the coverage objectives from being satisfied. When you generate tests without considering these constraints, all of the coverage objectives are satisfied.

1. The Input1 and Input2 minimum and maximum values are captured directly on their respective inport signal attributes.
2. The minimum and maximum values are specified on the Simulink.Signal objects associated with signals a and b. Simulink Design Verifier uses the signal object's values as constraints. When multiple minimum and maximum values are specified, e.g., on the inport and on the signal object, Simulink Design Verifier considers their tightest range.
3. Simulink Design Verifier considers the minimum and maximum limit ranges specified on Stateflow® data that is directly connected to the root-level input ports
4. For subsystem analysis, the subsystem root-level specified input minimum and maximum values are considered. Observe that generating tests for the Subsystem uses the constraints specified on SSIn, but ignores them for the system-level analysis.

```
open_system('sldvdemo_minmaxconstraints');
```



### Simulink Design Verifier Using Specified Input Minimum and Maximum Values as Constraints



1. The Input1 and Input2 minimum and maximum values are captured directly on their respective inport signal attributes.

2. The minimum and maximum values are specified on the Simulink.Signal objects associated with signals a and b. Simulink Design Verifier uses the signal object's values as constraints. When multiple minimum and maximum values are specified, e.g., on the inport and on the signal object, Simulink Design Verifier considers their tightest range.

3. Simulink Design Verifier considers the minimum and maximum limit ranges specified on Stateflow data that is directly connected to the root-level input ports.

4. For subsystem analysis, the subsystem root-level specified input minimum and maximum values are considered. Observe that generating tests for the Subsystem uses the constraints specified on SSIn, but ignores them for the system-level analysis.

## Configuring S-Function for Test Case Generation

This example shows how to compile an S-Function to be compatible with Simulink® Design Verifier™ for test case generation. Simulink Design Verifier supports S-Functions that are:

- Generated with the Legacy Code Tool, with `def.Options.supportCoverageAndDesignVerifier` set to `true`,
- Generated with the SFunctionBuilder, with **Enable support for Design Verifier** selected on the **Build Info** tab of the SFunctionBuilder dialog box, or
- Compiled with the function `slcovmex`, with the option `-sldv` passed.

### Compile S-Function to be Compatible with Simulink Design Verifier

The handwritten S-Function is found in the file `sldvexSFunctionHandlingSFcn.c`, and the user source code for the lookup table is found in the file `sldvexSFunctionHandlingSource.c`. Call the function `slcovmex` to compile the C-MEX S-Function and make it compatible with Simulink Design Verifier.

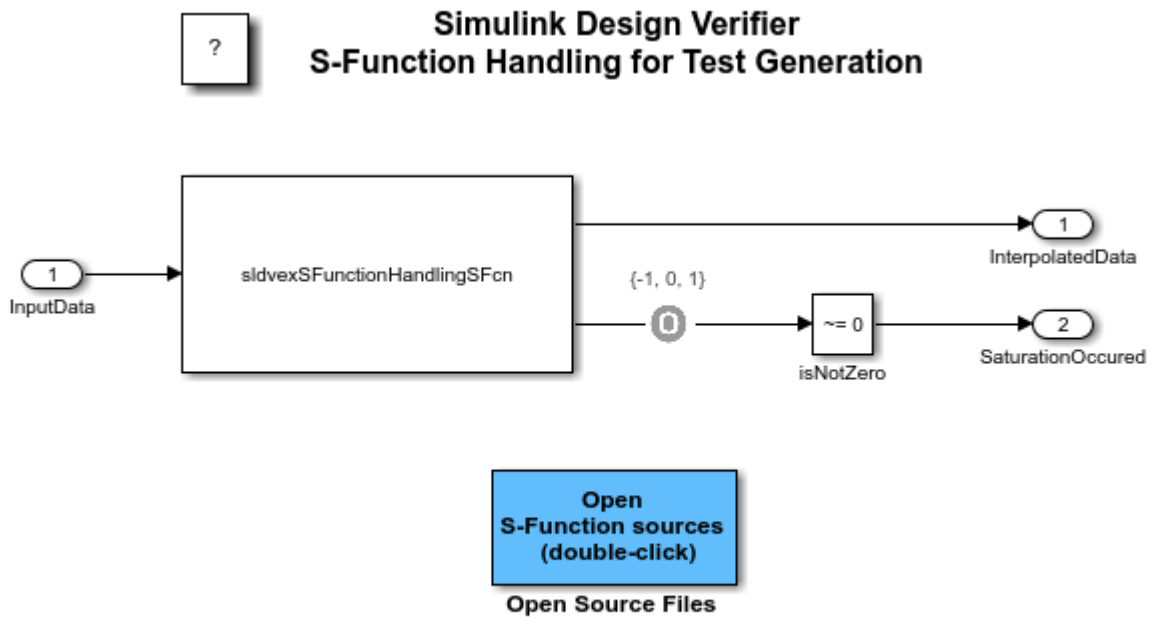
```
slcovmex('-sldv', ...
        '-output', 'sldvexSFunctionHandlingSFcn',...
        ['-I', fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src')], ...
        fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src', 'sldvexSFunctionHandlingSource.c'),
        fullfile(matlabroot, 'toolbox', 'sldv', 'sldvdemos', 'src', 'sldvexSFunctionHandlingSFcn.c')
    );
```

```
mex -IB:\matlab\toolbox\sldv\sldvdemos\src C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tpac1a0bb4_63
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
mex -IB:\matlab\toolbox\sldv\sldvdemos\src B:\matlab\toolbox\sldv\sldvdemos\src\sldvexSFunctionHandlingSource.c
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
```

### Create Test Suite

The example model `sldvexSFunctionHandlingExample` contains the handwritten S-Function, which implements a lookup table algorithm. The S-Function block returns the interpolated value at the first output port and returns the status of the interpolation at the second output port. The second output port returns the value `-1` if a lower saturation occurs, `1` if an upper saturation occurs, and `0` otherwise. Open the `sldvexSFunctionHandlingExample` model and configure the analysis options by turning on S-Function support for test generation. On running the analysis, Simulink Design Verifier returns a test suite that satisfies all coverage objectives.

```
open_system('sldvexSFunctionHandlingExample');
```



Copyright 2015-2019 The MathWorks, Inc.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'ConditionDecision';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
opts.SFcnSupport = 'on';
```

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

### Verifying Complete Coverage

The `sldvruntest` function verifies that the test suite achieves complete model coverage. The `cvhtml` function produces a coverage report that indicates 100% Condition and Decision coverage is achieved with the generated test vectors.

```
[~, finalCov] = sldvruntest('sldvexSFunctionHandlingExample', fileNames.DataFile, [], true);
cvhtml('Final Coverage', finalCov);
```

### Clean Up

To complete the demo, close all models.

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



## Code Coverage Test Generation

This example shows how to use Simulink® Design Verifier™ to generate test cases to obtain complete code coverage.

You first collect code coverage for an example model configured for software-in-the-loop (SIL) simulation mode. Then you use Simulink® Design Verifier™ to create a test suite that generates test cases. Finally, you execute the generated test cases in SIL simulation mode to verify the complete coverage.

### Check Product Availability

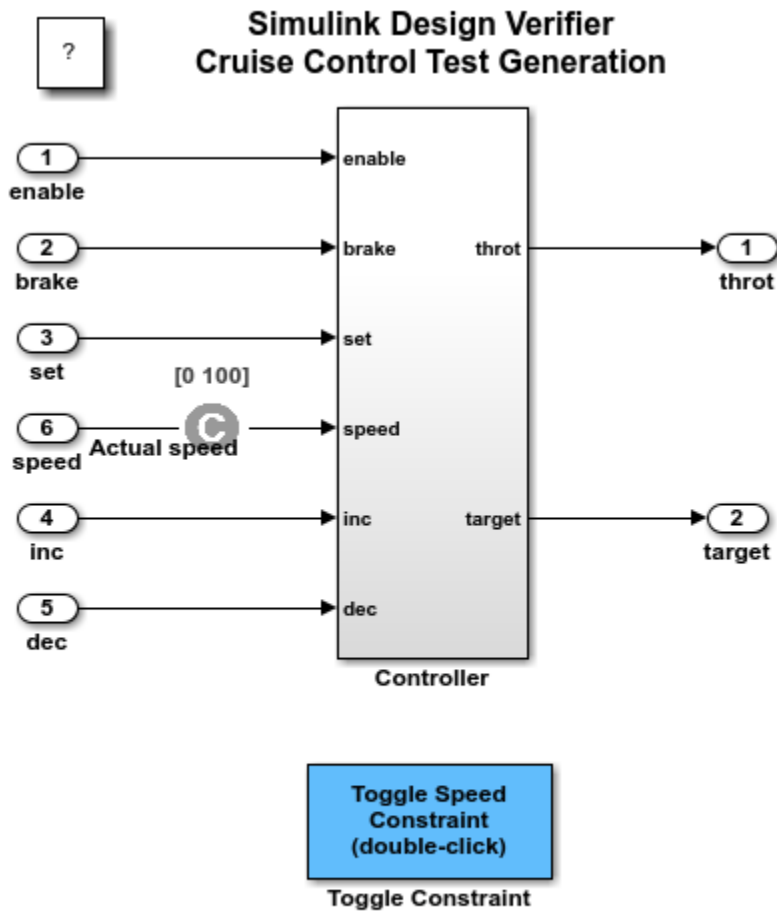
Make sure that you have Simulink® Coder™ and Embedded Coder™ software installed on your machine.

```
if ~(license('test', 'Real-Time_Workshop') && ...  
    license('test', 'RTW_Embedded_Coder'))  
    return  
end
```

### Initial Setup

Make sure that an unedited version of the model is open.

```
model = 'sldvdemo_cruise_control';  
close_system(model, 0)  
open_system(model)
```



Copyright 2006-2023 The MathWorks, Inc.

### Configure the Model for SIL based test generation

1. In the **Configuration Parameters** window, click **Code Generation** and set **System Target File** to `ert.tlc`. Alternatively, enter:

```
set_param(model, 'SystemTargetFile', 'ert.tlc');
```

2. Click **Hardware Implementation**, then set **Device vendor** and **Device type** to the vendor and type of your SIL system. For example, for a 64-bit Linux machine, set **Device vendor** to `Intel` and **Device type** to `x-86-64(Windows)`. Alternatively, enter:

```
if ismac
    lProdHWDeviceType = 'Intel->x86-64 (Mac OS X)';
elseif isunix
    lProdHWDeviceType = 'Intel->x86-64 (Linux 64)';
else
    lProdHWDeviceType = 'Intel->x86-64 (Windows64)';
end

set_param(model, 'ProdHWDeviceType', lProdHWDeviceType);
```

## Find Test Cases for Coverage Computation

Analyze the `sldvdemo_cruise_control` model by using Simulink® Design Verifier™ to generate a test suite that achieves increased code coverage. Set the Simulink® Design Verifier™ options to generate test cases to achieve MCDC coverage for the top model.

```

opts = sldvoptions;
opts.TestgenTarget = 'GenCodeTopModel';
opts.Mode = 'TestGeneration';
[~, files] = sldvrun(model, opts, true);

### Starting build procedure for: sldvdemo_cruise_control
### Loading TLC function libraries
.....
### Initial pass through model to cache user defined code
.
### Caching model source code
.....
### Writing header file sldvdemo_cruise_control_types.h
### Writing header file sldvdemo_cruise_control.h
### Writing header file rtwtypes.h
.
### Writing source file sldvdemo_cruise_control.c
### Writing header file sldvdemo_cruise_control_private.h
### Writing source file ert_main.c
### TLC code generation complete (took 3.551s).

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029

Microsoft (R) Program Maintenance Utility Version 14.29.30137.0
Copyright (C) Microsoft Corporation. All rights reserved.

    cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERPRETER
sldvdemo_cruise_control.c
### Successfully generated all binary outputs.

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
### Successful completion of build procedure for: sldvdemo_cruise_control
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502968\IntelWin64\sldv-ex15502968'
### Building 'sldvdemo_cruise_control_ca': nmake -f sldvdemo_cruise_control_ca.mk all

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029

```

```
Microsoft (R) Program Maintenance Utility Version 14.29.30137.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DINTEGGER_CODE=
coder_assumptions_hwimpl.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DINTEGGER_CODE=
coder_assumptions_flt.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DINTEGGER_CODE=
sldvdemo_cruise_control_ca.c
### Creating static library ".\sldvdemo_cruise_control_ca.lib" ...
lib /nologo -out:.\sldvdemo_cruise_control_ca.lib @sldvdemo_cruise_control_ca.rsp
### Created: .\sldvdemo_cruise_control_ca.lib
### Successfully generated all binary outputs.
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502968\IntelWin64\sldv
### Building 'sldvdemo_cruise_control': nmake -f sldvdemo_cruise_control.mk all
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
```

```
Microsoft (R) Program Maintenance Utility Version 14.29.30137.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
xil_interface_lib.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
xil_data_stream.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
xil_services.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
xil_interface.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
xilcomms_rtiostream.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
xil_rtiostream.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
rtiostream_utils.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_app.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_data_stream.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
coder_assumptions_rtiostream.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
sil_main.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
target_io.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
rtiostream_tcpip.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERI
```

```

xil_instrumentation.c
  cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERP
codeinstr_data_stream.c
  cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTERP
codeinstr_rtiostream.c
### Creating standalone executable ".\sldvdemo_cruise_control.exe" ...
  link /RELEASE /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsock.lib advapi32.lib -out
### Created: .\sldvdemo_cruise_control.exe
### Successfully generated all binary outputs.

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
### Starting SIL simulation for component: sldvdemo_cruise_control
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Starting build procedure for: sldvdemo_cruise_control
### Generating code and artifacts to 'Target environment subfolder' folder structure
### Generating code into build folder: C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-
### Generated code for 'sldvdemo_cruise_control' is up to date because no structural, parameter c
### Saving binary information cache.
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502968\IntelWin64\sldvdemo_cru
### Building 'sldvdemo_cruise_control': nmake -f sldvdemo_cruise_control.mk buildobj

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502

Microsoft (R) Program Maintenance Utility Version 14.29.30137.0
Copyright (C) Microsoft Corporation. All rights reserved.

### Successfully generated all binary outputs.

mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
### Successful completion of build procedure for: sldvdemo_cruise_control

Build Summary

Top model targets built:

Model                Action                Rebuild Reason
=====
sldvdemo_cruise_control  Code compiled.  Compilation artifacts were out of date.

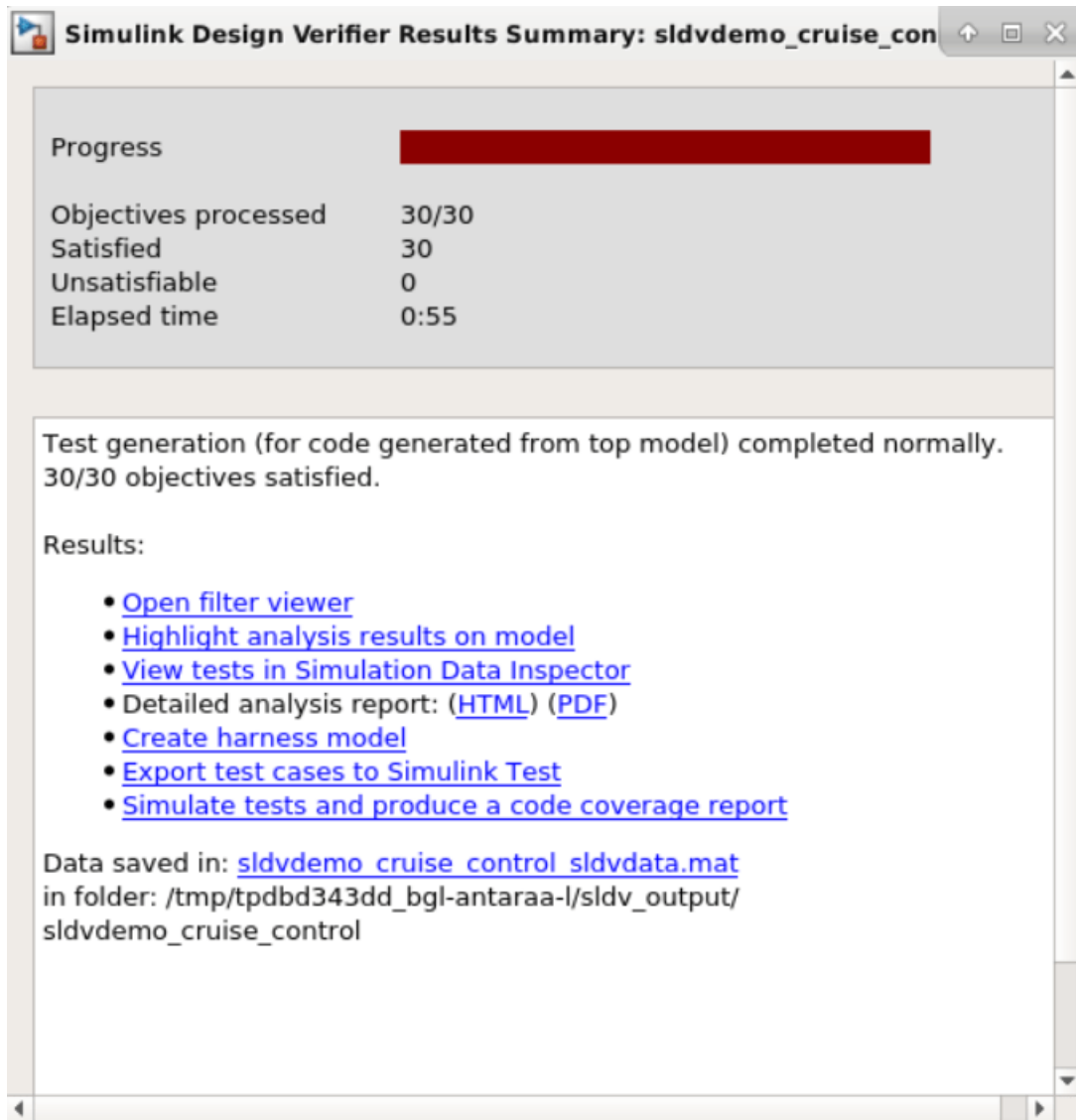
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 3.6671s
### Preparing to start SIL simulation ...
### Skipping makefile generation and compilation because C:\TEMP\Bdoc23a_2213998_3568\ib570499\28
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:

```

```

### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis

```



Note: When you run the script, the SIL test generation regenerates and recompiles the code.

## Verify Complete Coverage

The `sldvrntest` function simulates the model by using the generated test suite. The `cvhtml` function produces a coverage report that indicates the final coverage of the `sldvdemo_cruise_control` model.

```
[~, finalCov] = sldvrntest(model, files.DataFile, [], true);
cvhtml('sil_final_coverage', finalCov);
close_system(model, 0);
```

```
### Starting build procedure for: sldvdemo_cruise_control
### Generating code and artifacts to 'Target environment subfolder' folder structure
### Generating code into build folder: C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-
### Generated code for 'sldvdemo_cruise_control' is up to date because no structural, parameter
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502968\IntelWin64\sldvdemo_cru
### Building 'sldvdemo_cruise_control': nmake -f sldvdemo_cruise_control.mk buildobj
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
Microsoft (R) Program Maintenance Utility Version 14.29.30137.0
```

```
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
### Successfully generated all binary outputs.
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
### Successful completion of build procedure for: sldvdemo_cruise_control
```

Build Summary

Top model targets built:

| Model                   | Action         | Rebuild Reason                          |
|-------------------------|----------------|---|
| sldvdemo_cruise_control | Code compiled. | Compilation artifacts were out of date. |

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 2.7723s

```
### Preparing to start SIL simulation ...
```

```
Building with 'Microsoft Visual C++ 2019 (C)'.
```

```
MEX completed successfully.
```

```
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
```

```
### 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502968\IntelWin64\sldvdemo_cru
```

```
### Building 'sldvdemo_cruise_control': nmake -f sldvdemo_cruise_control.mk all
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex155029
```

```
Microsoft (R) Program Maintenance Utility Version 14.29.30137.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTER
xil_interface.c
cl -c -nologo -GS -W4 -DWIN32 -D_MT -MT -D_CRT_SECURE_NO_WARNINGS /Od /Oy- -DCLASSIC_INTER
xil_instrumentation.c
### Creating standalone executable ".\sldvdemo_cruise_control.exe" ...
link /RELEASE /INCREMENTAL:NO /NOLOGO kernel32.lib ws2_32.lib mswsock.lib advapi32.lib -out
### Created: .\sldvdemo_cruise_control.exe
### Successfully generated all binary outputs.
```

```
mathworks\batserve@BAT6234WIN64 C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldv-ex15502
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
### Starting SIL simulation for component: sldvdemo_cruise_control
rtw.connectivity.HostLauncher: started executable with host process identifier <a href="matlab:
rtw.connectivity.HostLauncher: stopped executable with host process identifier <a href="matlab:
### Stopping SIL simulation for component: sldvdemo_cruise_control
### Completed code coverage analysis
```

## Summary

| File Contents/Complexity                                | Test 1   |           |      |           |          |
|---|----------|-----------|------|-----------|----------|
|   | Decision | Condition | MCDC | Statement | Function |
| 1. <a href="#">sldvdemo_cruise_control.c</a>            | 9 100%   | 100%      | 100% | 100%      | 100%     |
| 2... <a href="#">sldvdemo_cruise_control_step</a>       | 7 100%   | 100%      | 100% | 100%      | 100%     |
| 3... <a href="#">sldvdemo_cruise_control_initialize</a> | 1 --     | --        | --   | 100%      | 100%     |
| 4... <a href="#">sldvdemo_cruise_control_terminate</a>  | 1 --     | --        | --   | 100%      | 100%     |

Note: When you run the script, the SIL test generation regenerates and recompiles the code.



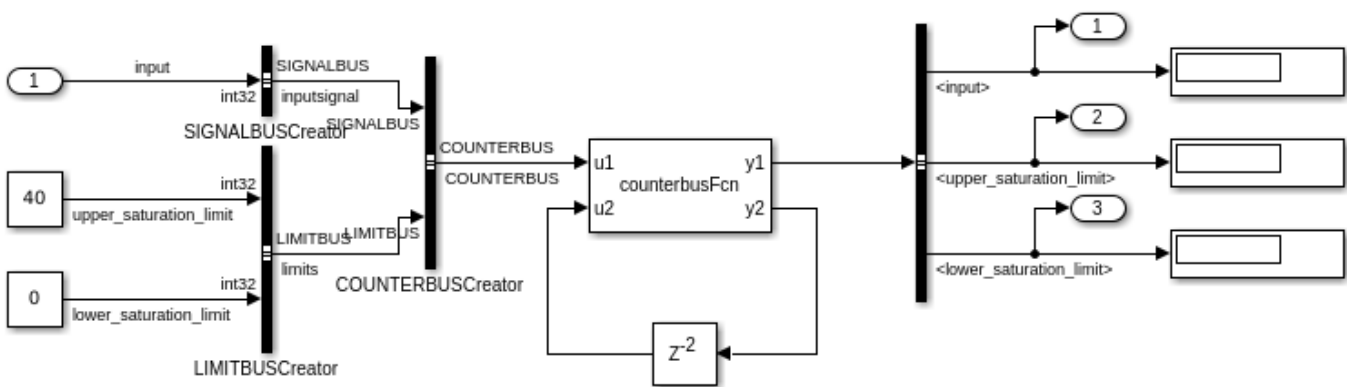
## Test Generation on Model with C Caller Block

This example shows how to use test generation on a model with a C Caller block and custom C code

### Open the model containing the C Caller block and custom code

```
open_system('sldvexCCallerBlockExample');
```

### Simulink Design Verifier Test Case Generation with C Caller Block



Copyright 2018 The MathWorks, Inc.

### Generate tests to ensure coverage of the model

Use the `sldvrun` function to run Simulink® Design Verifier™ analysis.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'ConditionDecision';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
```

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

```
27-Feb-2023 10:36:01
Checking compatibility for test generation: model 'sldvexCCallerBlockExample'
Compiling model...done
Building model representation...done
```

```
27-Feb-2023 10:36:27
```

```
'sldvexCCallerBlockExample' is compatible for test generation with Simulink Design Verifier.
```

```
Generating tests using model representation from 27-Feb-2023 10:36:27...
```

```
.....
```

27-Feb-2023 10:36:44

Completed normally.

Generating output files:

27-Feb-2023 10:36:46

Results generation completed.

Data file:

/home/lucyzeng/Documents/MATLAB/ExampleManager/lucyzeng.BR2023ad.j2194193.1/sldv-ex07804984/

### Verify the coverage

Use the `sldvruntest` function to verify that the test suite achieves complete model coverage.

```
[~, finalCov] = sldvruntest('sldvexCCallerBlockExample', fileNames.DataFile, [], true);  
cvhtml('Final Coverage', finalCov);
```

### Clean Up

To complete the example, close all models.

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

## Debug Enhanced Modified Condition Decision Coverage Using Model Slicer

This example shows how to find the Simulink® Design Verifier™ generated objectives related to a specific model object using Model Slicer. Once the objectives are identified, Model Slicer highlights the slice at the step when the objective is observable.

This example uses the following products to demonstrate debugging enhanced Modified Condition Decision Coverage (MCDC):

- Simulink Design Verifier
- Model Slicer

Enhanced MCDC analyzes the detectability of each objective in the model and generates non-masking test cases for each objective. It coordinates the effect of downstream blocks to avoid masking effects. It also calculates detection sites for each detectable objective where the effect of the objective can be observed. This data is available in the `sldvdemo_cruise_control_sldvdata.mat` file generated by the analysis. These detection sites can be added to the equivalence criteria of back-to-back testing.

This example uses the following slicer configuration:

- Starting point is set as the model object to be observed.
- Exclusion point is set as the detection point relevant to the objective generated by Simulink Design Verifier.
- Signal propagation is set to downstream (forward slice).

### Step 1: Prepare the Model

1. Open the model.

```
model = 'sldvdemo_cruise_control';
open_system(model);
```

2. Load the data file generated by Simulink Design Verifier (sldvData) for test generation using Enhanced MCDC.

```
load('sldvdemo_cruise_control_sldvdata.mat');
```

3. Choose the model object for which the objective must be highlighted and find its SID.

```
modelObjIdentifier = 'sldvdemo_cruise_control/Controller/Switch3';
modelObjSID = Simulink.ID.getSID(modelObjIdentifier);
```

### Step 2: Setting Up Model Slicer

1. Enable FastRestart for the model.

```
set_param(model, 'FastRestart', 'on');
```

Enabling FastRestart will simulate the model and collect the simulation data at various time stamps. This will allow us to use **Step Back** and **Step Forward** options.

2. Create and activate Model Slicer object.

```
slicerObject = slslicer(model);
activate(slicerObject);
```

3. Set the signal propagation to **downstream**.

```
slicerObject.Configuration.SignalPropagation = 'downstream';
```

### Step 3: Find Objectives Related to the Model Object

1. Access sldvData with an object of SldvDataExplorer class.

```
sldvObj = SldvDataExplorer(sldvData);
```

**Note:** The class **SldvDataExplorer** is a helper class. You can edit it as per your requirements.

2. Find all objectives related to the model object and the details of the objectives.

```
[objectives, tableOfObjectives] = sldvObj.getObjectivesForModelObj(modelObjSID);
disp(tableOfObjectives);
```

| ObjectiveNum | Type       | Description   |
|--------------|------------|---|
| 1            | "Decision" | "logical trigger input false (output is from 3rd input port)" |
| 2            | "Decision" | "logical trigger input true (output is from 1st input port)"  |

The following details of the objectives are saved in `tableOfObjectives` table:

- ObjectiveNum - Objective number.
- Type - MCDC/Decision/Condition.
- Description - Description of the objective as generated by Simulink Design Verifier.
- Detectability - The detectability status of an objective.
- Status - The status of an objective.
- TestCaseId - Integer that represents the index of a test case or counterexample that addresses an objective.

### Step 4: Highlighting the Objectives

For this example, we will highlight the first objective from the table.

1. Obtain the simulation input object with the input values set according to the test case that corresponds to the objective.

```
[simIn, atStep, ~] = sldvObj.getSimInObjForObjective(objectives(1));
```

2. Allow rollback in the model so it is possible to step backwards in the model and set the number of simulation rollback steps to 1.

```
simIn = simIn.setModelParameter('EnableRollBack', 'on');
simIn = simIn.setModelParameter('NumberOfSteps', 1);
```

3. Apply Simulink input object to model.

```
slicerObject.applySimInToModel(simIn);
```

4. Find all the detection sites for the selected objective.

```
objectDetectionSites = sldvObj.getObjectDetectionSites(objectives(1));
```

5. Add all detection sites as exclusion points.

```
for n = 1:length(objectDetectionSites)
    detectionSite = objectDetectionSites(n).modelObj;
    slicerObject.addExclusionPoint(detectionSite);
end
```

6. Add the model object as an starting point.

```
slicerObject.addStartingPoint(modelobjSID);
```

7. Step to the point in the testcase where the objective is observable.

```
for q = 1:atStep
    slicerObject.stepForward();
end
```

Now you can observe that the slice is highlighted.

### Cleanup

Perform the following actions to cleanup the model:

1. Clear the slicer object.
2. Clear the Simulink input object.

```
clear slicerObject simIn
```

3. Reset the FastRestart parameter of the model.

```
set_param(model, 'FastRestart', 'off');
```

### See Also

- “Use Model Coverage Objectives for Enhanced MCDC Coverage” on page 7-42

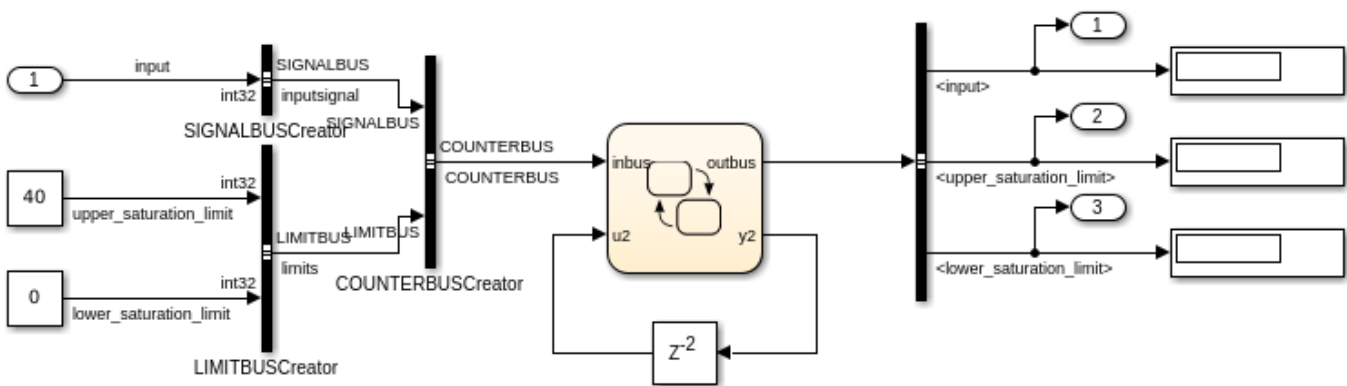
## Test Generation for Custom Code in a Stateflow Chart

This example shows how to use test generation on a model with custom code in a Stateflow® chart.

### Open the Model Containing Custom Code in a Stateflow Chart

```
open_system('sldvexSFCustomCodeExample');
```

### Simulink Design Verifier Test Case Generation with C/C++ Custom Code



Copyright 2018 The MathWorks, Inc.

### Generate Tests to Ensure Coverage of the Model

Use the `sldvrun` function to run the Simulink® Design Verifier™ analysis.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'ConditionDecision';
opts.SaveHarnessModel = 'off';
opts.SaveReport = 'off';
```

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

```
27-Feb-2023 10:49:57
Checking compatibility for test generation: model 'sldvexSFCustomCodeExample'
Compiling model...done
Building model representation...done
```

```
27-Feb-2023 10:50:13
```

```
'sldvexSFCustomCodeExample' is compatible for test generation with Simulink Design Verifier.
```

```
Generating tests using model representation from 27-Feb-2023 10:50:13...
```

```
.....
```

27-Feb-2023 10:50:29

Completed normally.

Generating output files:

27-Feb-2023 10:50:30

Results generation completed.

Data file:

/home/lucyzeng/Documents/MATLAB/ExampleManager/lucyzeng.BR2023ad.j2194193.1/sldv-ex18712703/

### **Verify the Coverage**

Use the `sldvruntest` function to verify that the test suite achieves complete model coverage.

```
[~, finalCov] = sldvruntest('sldvexSFCustomCodeExample', fileNames.DataFile, [], true);  
cvhtml('Final Coverage', finalCov);
```

### **Clean Up**

To complete the example, close all models.

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

## Generate Test Cases for Model Blocks

This example shows how to generate a test case for Model block that models a power window controller in Simulink® Design Verifier™.

### **Step 1: Open the Model**

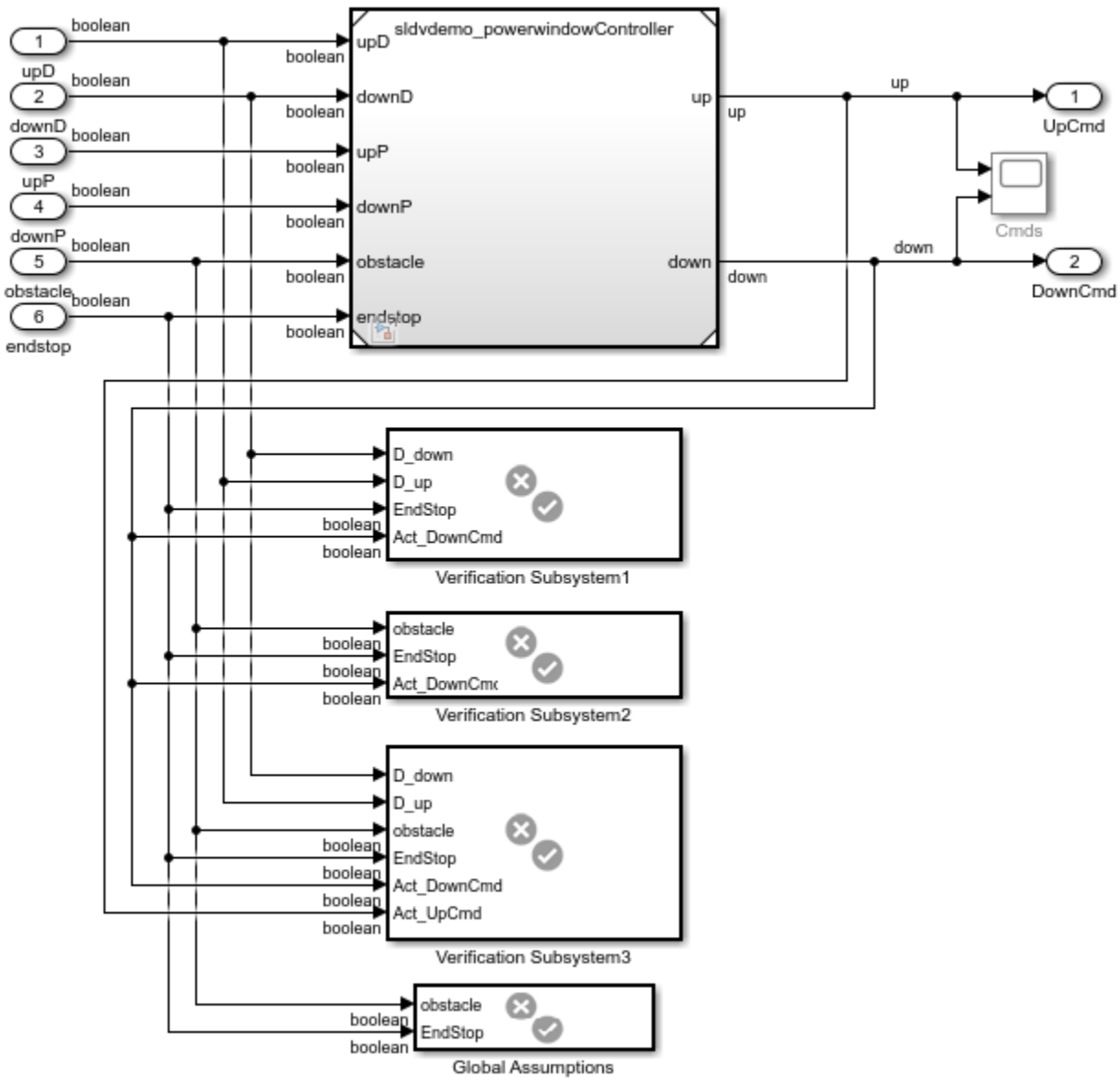
The top-level model represents a power window verification system. The model contains a model reference that represents a power window controller model and that specifies the controller behavior and the modeled requirements.

To open the model of the top-level verification system, enter:

```
open_system('sldvdemo_powerwindow_vs');
```

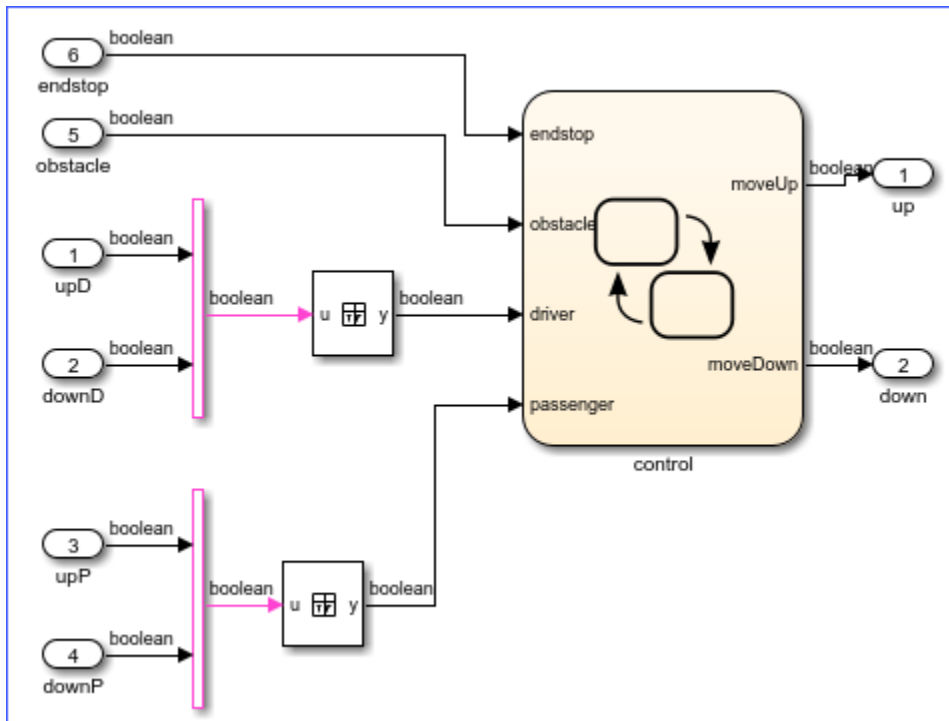


### Power Window Controller Temporal Property Specification



Copyright 1990-2010 The MathWorks, Inc.

The model reference points to the model `slvdemo_powerwindowController`, which responds to the driver and passenger commands by giving the commands for moving the window up or down. The model also responds if the window encounters an obstacle or if it reaches the end of the window frame in either direction.



### Step 2: Specify Analysis Options

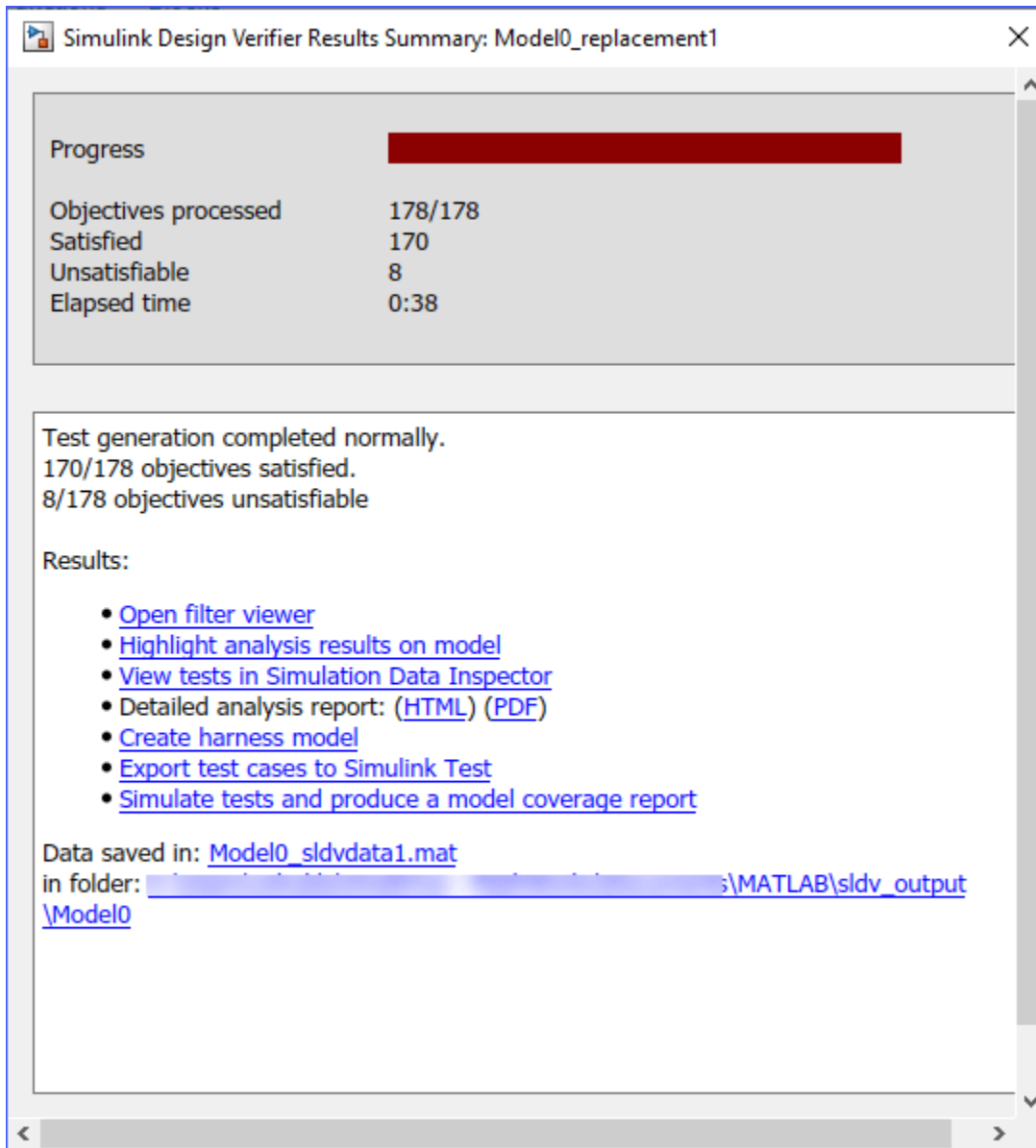
Specify the analysis options for test case generation:

1. On the **Design Verifier** tab, change the **mode** to **Test Generation**.
2. Click **Test Generation Settings**.
3. From **Test Generation** pane in the Configuration Parameters dialog box, set **Model coverage objectives** to MCDC.
4. Click **OK**.

### Step 3: Perform Analysis and Review Results

Perform test case generation on the Model block:

1. Right-click the **Model** block and select **Design Verifier > Generate Tests for Referenced Model**. Alternatively, in the **Design Verifier** pane, in the **Analyze** section, click the unpin button, then select the Model block. Then click Generate Tests.
2. Simulink Design Verifier generates test cases for the Model block. The Results window shows that the test generation completed normally.



3. To access the detailed analysis report, click **HTML** in the Results window. The analysis report shows that 170 objectives are satisfied and eight objectives are unsatisfiable out of the 178 objectives processed.

#### Step 4: Clean Up

To complete the example, close the opened model.

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

#### Related Topics

- “What Is Test Case Generation?” on page 7-3

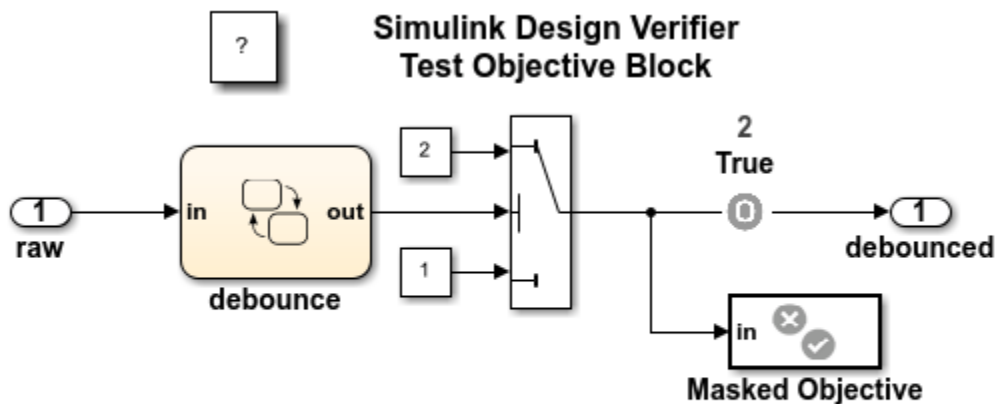
## Use Observer Reference Block for Test Case Generation

This example shows how to generate test cases for two custom Test Objective blocks using Observer Reference block and use model representation to reanalyze the design model. For more information, see “Isolate Verification Logic with Observers” on page 12-29. To reanalyze the model, you update the verification logic and set the **Rebuild model representation** option to **If change is detected**. For more information, see “Model Representation for Analysis” on page 2-28.

### Step 1: Open the Model and Replace Verification Subsystem

In the Test Objective block, the block "True" forces the output signal to be 2. The block "Edge" inside "Masked Objective" specifies that the output signal transitions from 2 to 1. To open the model, enter:

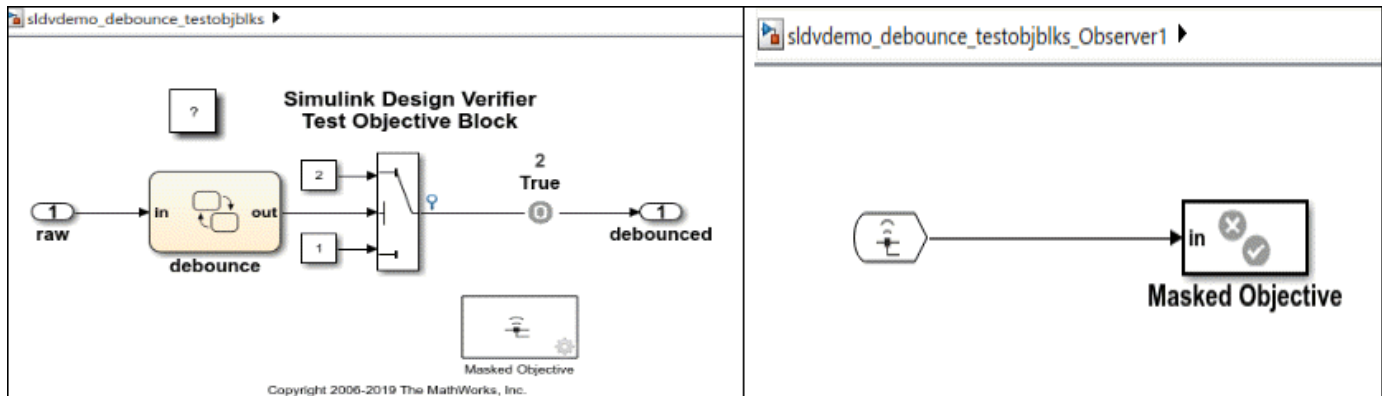
```
open_system('sldvdemo_debounce_testobjblks');
```



Copyright 2006-2023 The MathWorks, Inc.

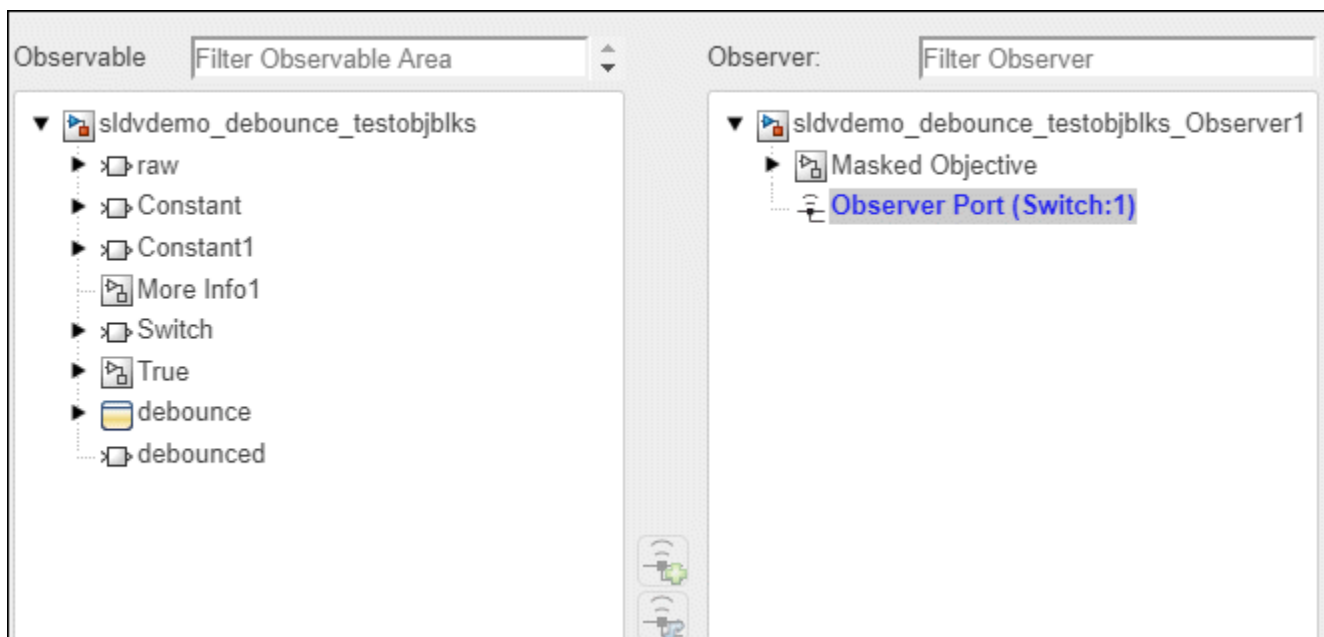
To replace the Verification Subsystem **Masked Objective** in the model by the Observer Reference block, follow these steps:

- Right-click on the **Masked Objective** in the `sldvdemo_debounce_testobjblks` model. In the context menu, click **Observers > Move selected block to Observer > New Observer**.
- Click **Yes** on **move 'Verify Output' to Observer** dialog box that appears after step (a).
- An Observer Reference block is added to your system model, and an Observer model `sldvdemo_debounce_testobjblks_observer1` is created and opened.



(d) Save the file `sldvdemo_debounce_validprop_Observer1` in a writable folder on the MATLAB path.

(e) Double-click on the Observer port to open the Manage Observer configuration window. The signal `Switch 1` is automatically mapped to the Observer Port block in the `sldvdemo_debounce_testobjblks_Observer1`.



(f) Select the input signal to the **Masked Objective** subsystem in the `sldvdemo_debounce_testobjblks` and click on **Test Point** in the **Signal** pane to make sure that Simulink Design Verifier successfully build the model representation for analysis.

## Step 2: Perform Test Generation Analysis

To perform the test generation analysis, follow these steps:

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

After the analysis completes, the Results Summary window displays that both objectives are satisfied with the test case.

To view the detailed analysis report, in the Results Summary window, click **HTML**. In the report, the Test Objectives Status chapter lists the status of the objectives for **Design Model** and **Observers Model(s)** in separate subsections.

[3.1.1. Design Model](#)  
[3.1.2. Observer Model\(s\)](#)

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

### 3.1.1. Design Model

This section contains information about 'Objectives Satisfied' in the design model.

| # | Type           | Model Item           | Description  | Analysis Time (sec) | Test Case         |
|---|----------------|----------------------|--------------|---------------------|-------------------|
| 2 | Test objective | <a href="#">True</a> | Objective: 2 | 12                  | <a href="#">1</a> |

### 3.1.2. Observer Model(s)

This section contains information about 'Objectives Satisfied' in the observer model(s).

| # | Type           | Model Item                            | Description  | Analysis Time (sec) | Test Case         |
|---|----------------|---------------------------------------|--------------|---------------------|-------------------|
| 1 | Test objective | <a href="#">Masked Objective/Edge</a> | Objective: 1 | 12                  | <a href="#">2</a> |

## Step 3: Modify Observer model and reanalyze without rebuilding design model representation

To generate the test case for the functional requirement, the debounced signal transitions from 1 to 2 without rebuilding the model representation for design model. To enable the reuse of design model representation, follow these steps:

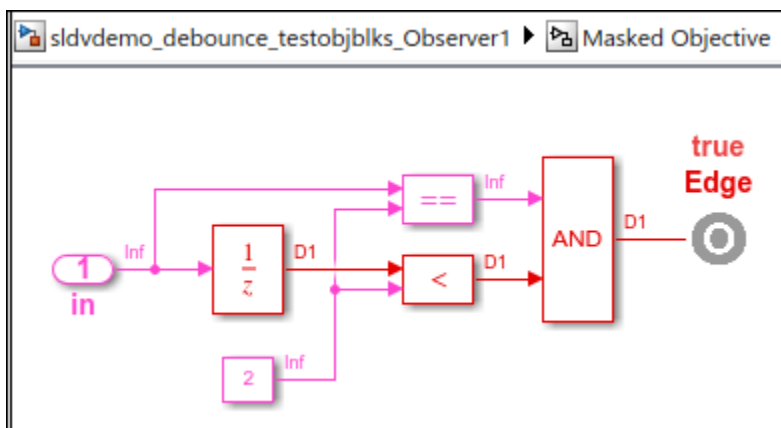
(a) On the **Design Verifier** tab, click **Test Generation Settings > Settings**.

(b) In the Configurations Parameters dialog box, on the Design Verifier pane, in Advanced parameters, set the **Rebuild model representation** option to **If change is detected** and Click **OK**.

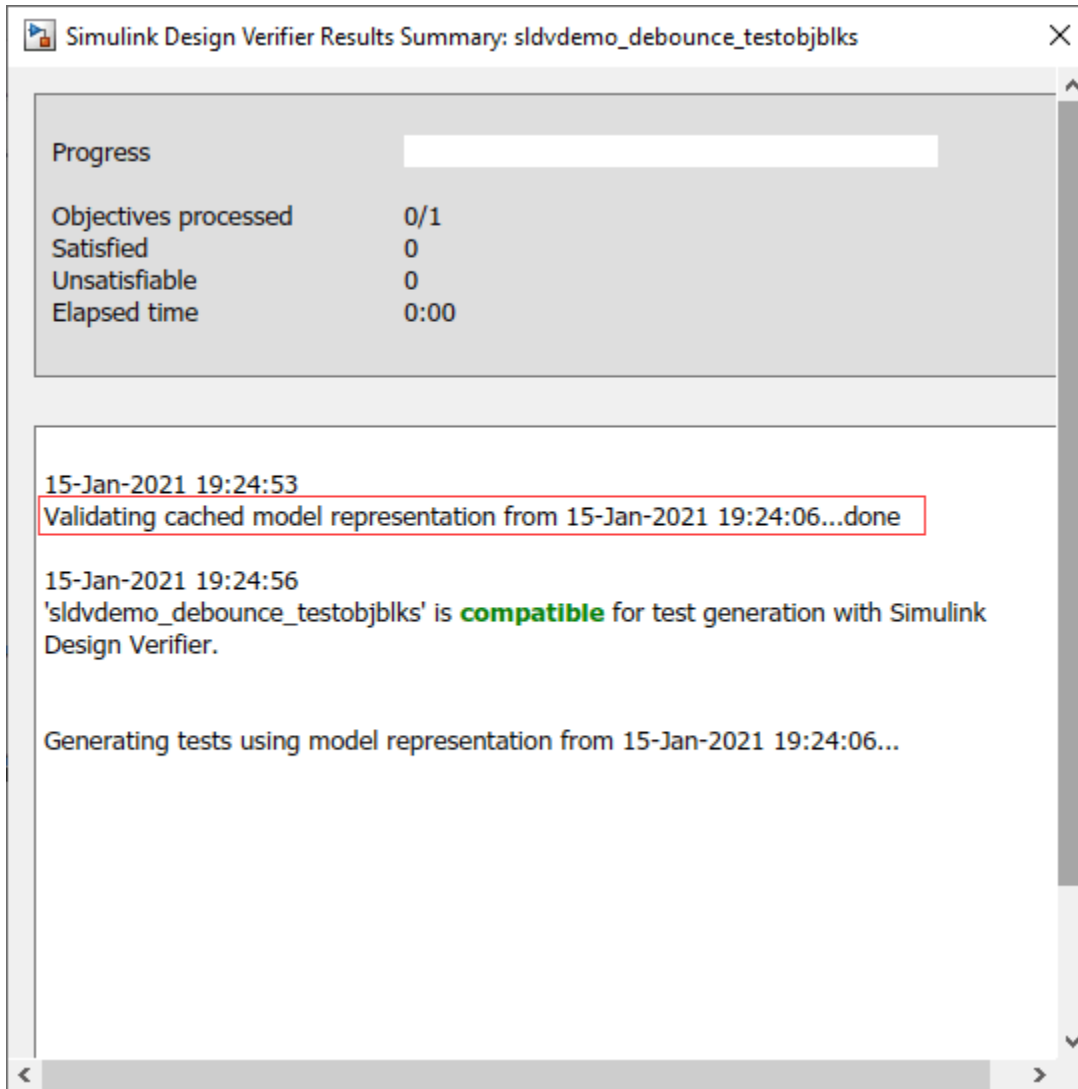
(c) To update the model parameters, follow these steps:

1. In the `sldvdemo_debounce_testobjblks_Observer1` window, double-click to open the **Masked Objective** subsystem and change the value of constant `In1` from 1 to 2 and relational operator from `>` to `<`.

2. Save the changes in a writable MATLAB path.



(d) Perform Test Case Generation Analysis and Review Results. On the **Design Verifier** tab, click **Generate Tests**. The software validates the cached design model representation, detects no change in design model and reuses the representation for analysis.



After the analysis completes, the Results Summary window displays that only one test objective is satisfied.

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

**Summary**

Length: 0.06 second (7 sample periods)

Objectives Satisfied: 2

**Objectives**

| Step | Time | Model Item                                 | Objectives   |
|------|------|--|--|
| 7    | 0.06 | <a href="#">Masked Objective/Edge True</a> | <a href="#">1. Objective: 1</a><br><a href="#">2. Objective: 2</a> |

**Generated Input Data**

|             |   |           |           |      |
|-------------|---|-----------|-----------|------|
| <b>Time</b> | 0 | 0.01-0.02 | 0.03-0.05 | 0.06 |
| <b>Step</b> | 1 | 2-3       | 4-6       | 7    |
| raw         | 2 | 1         | 2         | 1    |

**Note:** If you create a new model, by default, the **Rebuild model representation** option is set to **If change is detected**. The software validates the cache model representation, detects no change, and reuses the model representation for analysis.

**Related Topics**

- “Access Model Data Wirelessly by Using Observers” (Simulink Test).
- Verification Subsystem.



## Inspect Test Generation Objectives by Using Model Slicer

This example shows how to use Model Slicer to inspect test generation objectives in a Simulink model. The Simulink® Design Verifier™ analysis generates test cases and propagates the test cases to a Model block. You can see or inspect the values in the Model block at the time step at which the objective is observable and highlight the path and values using Model Slicer.

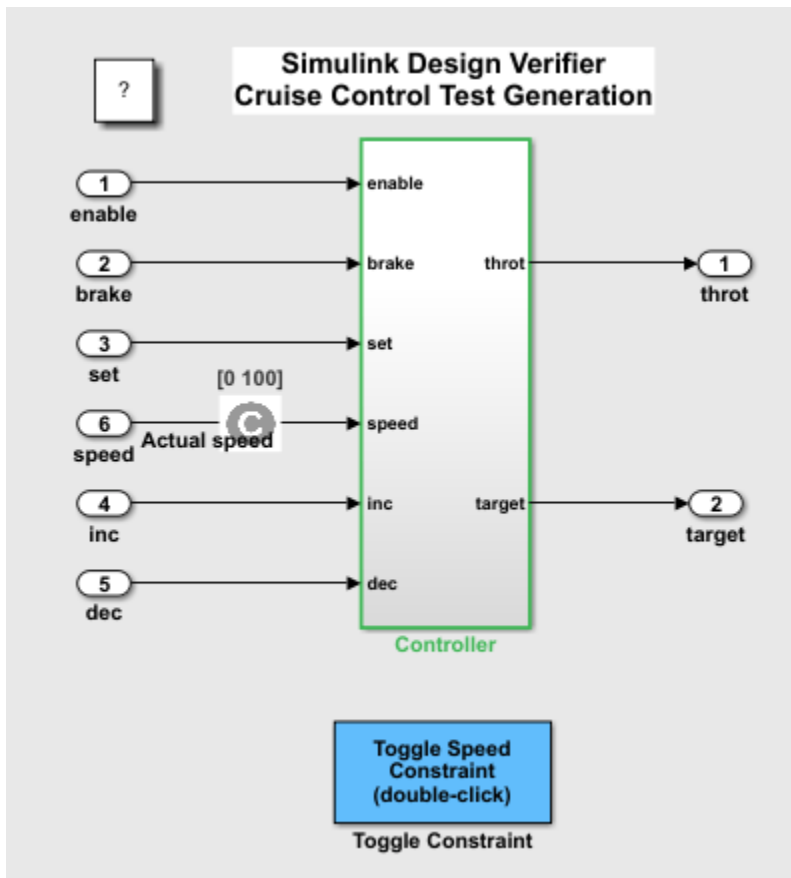
### Prepare the Model

Open the model.

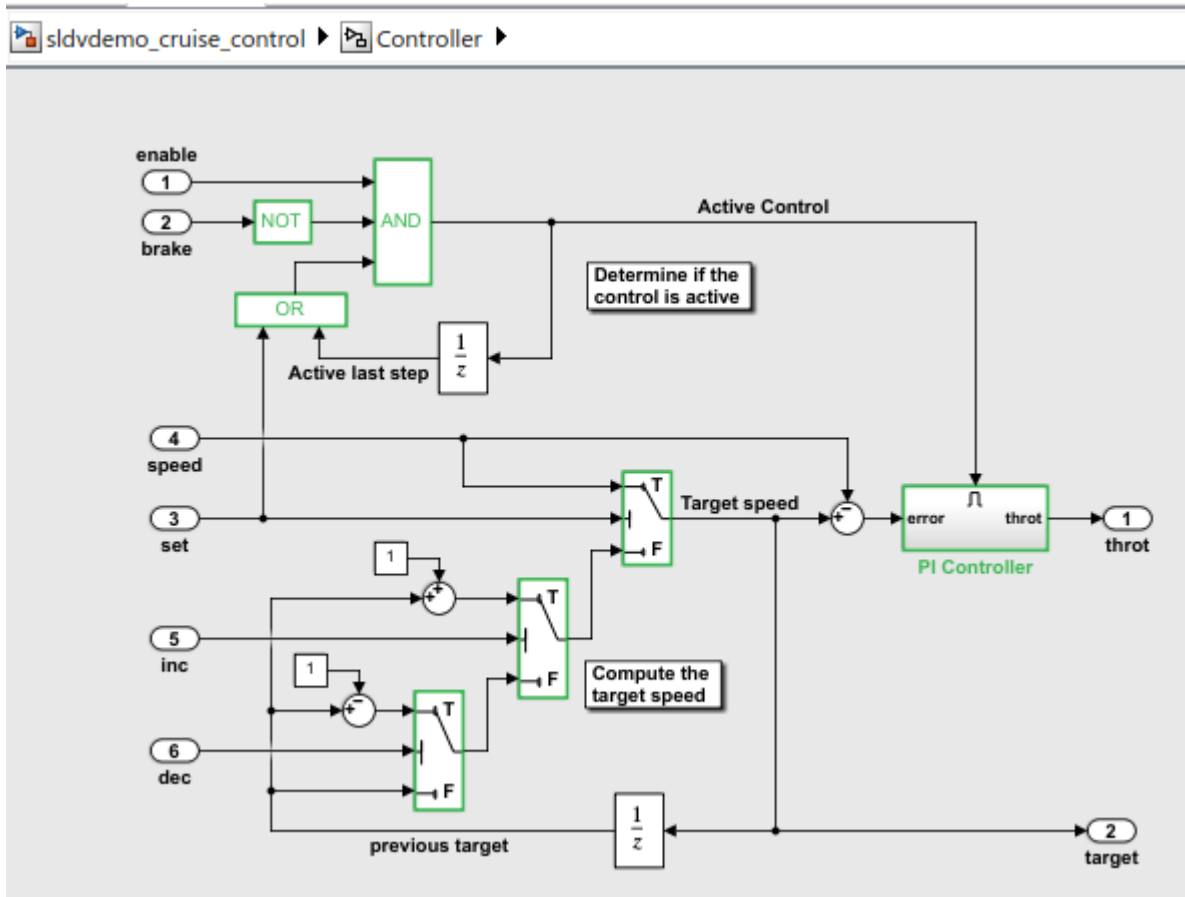
```
model = 'sldvdemo_cruise_control';
open_system(model);
```

### Generate Test Objectives

1. Open **Simulink Design Verifier** by clicking on **Apps > Design Verifier**.
2. In the **Design Verifier** tab, click **Generate Tests**. Simulink Design Verifier analyzes the model and displays the results in Simulink Design Verifier Results Summary window.
3. In the model, the analysis highlights the Controller subsystem where the objectives are located.

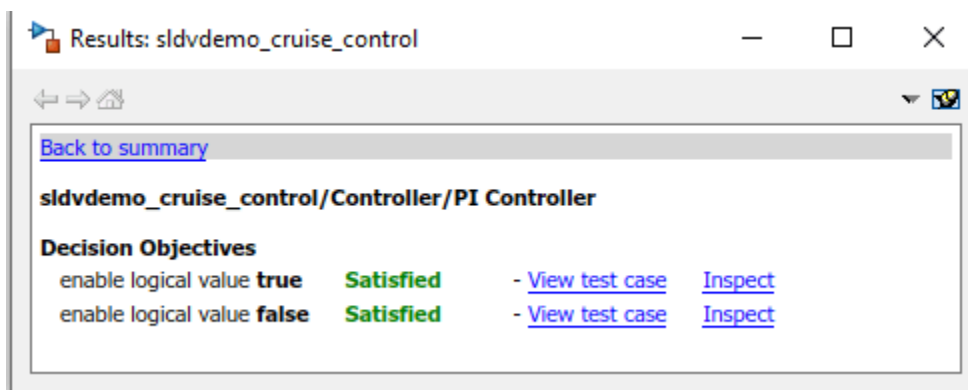


4. Open the Controller subsystem and click the PI Controller subsystem. Alternatively, you can select any of the blocks highlighted in green color. The objectives appear in the Results window.



### Inspect Test Objectives using Model Slicer

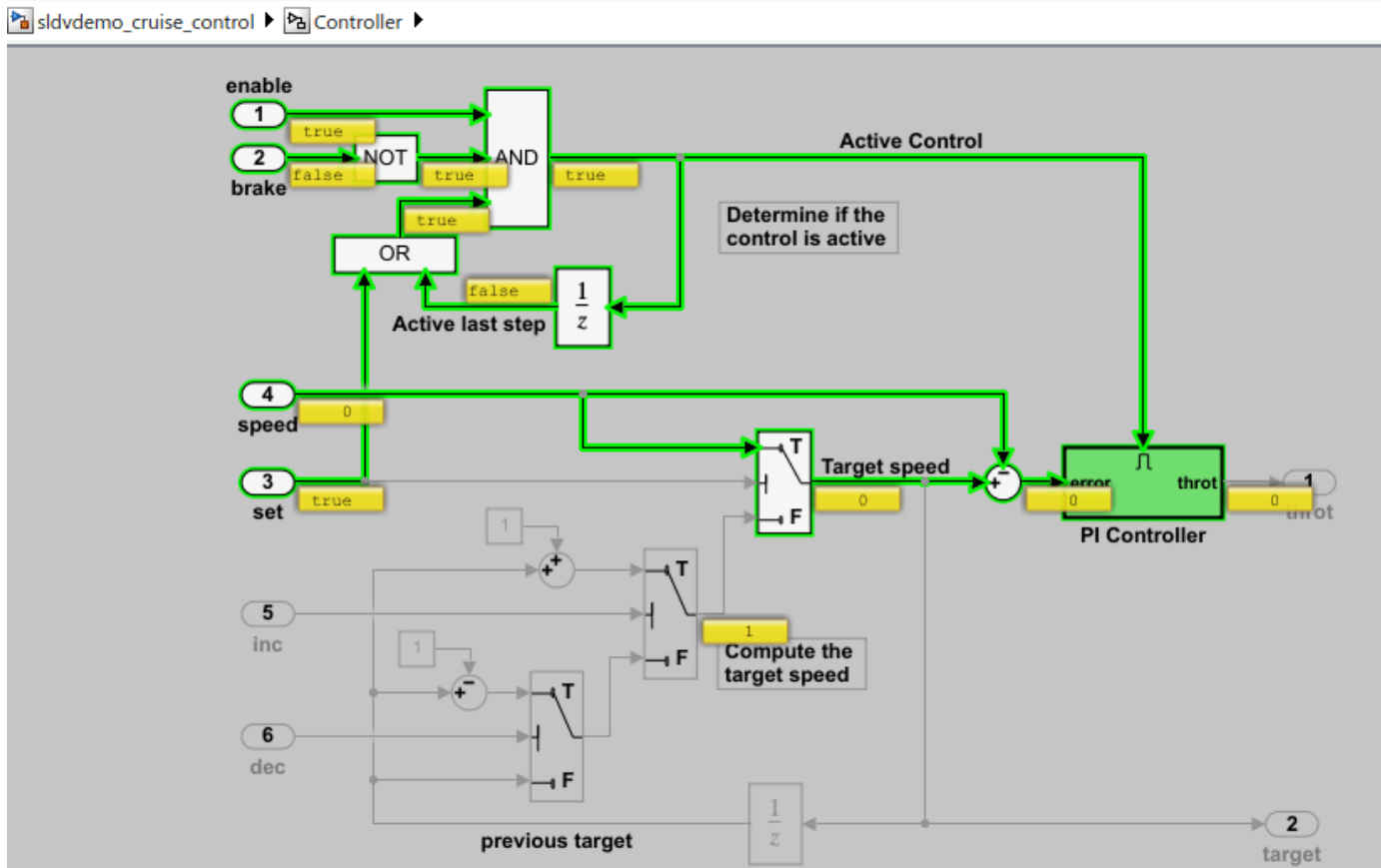
1. In the **Results** window, click **Inspect** to launch Model Slicer and analyze the objective. Alternatively, in the Design Verifier tab, in **Review Results** section, click **Review Results > Inspect Using Slicer**.



As part of the model setup, Model Slicer:

- Uses the selected block as a starting point
- Highlights the slice that represents the objective
- Simulates the model and pauses it at the time of observation

You can analyze the model by inspecting the port labels or observe the values of the test case propagated to the objective block and the path it takes.



Note that when you set the model coverage objective to enhanced MCDC, you can also inspect the objective detectability. In this case, the Model Slicer configuration allows you to switch to different modes by using the **Slice Configuration list**. For more information, see “Inspect Enhanced MCDC Objectives using Model Slicer” on page 7-50.

### Related Links

- “Basic Workflow for Enhanced MCDC Analysis” on page 7-47

## Generate Tests for Model Block Component by Using Default Simulation

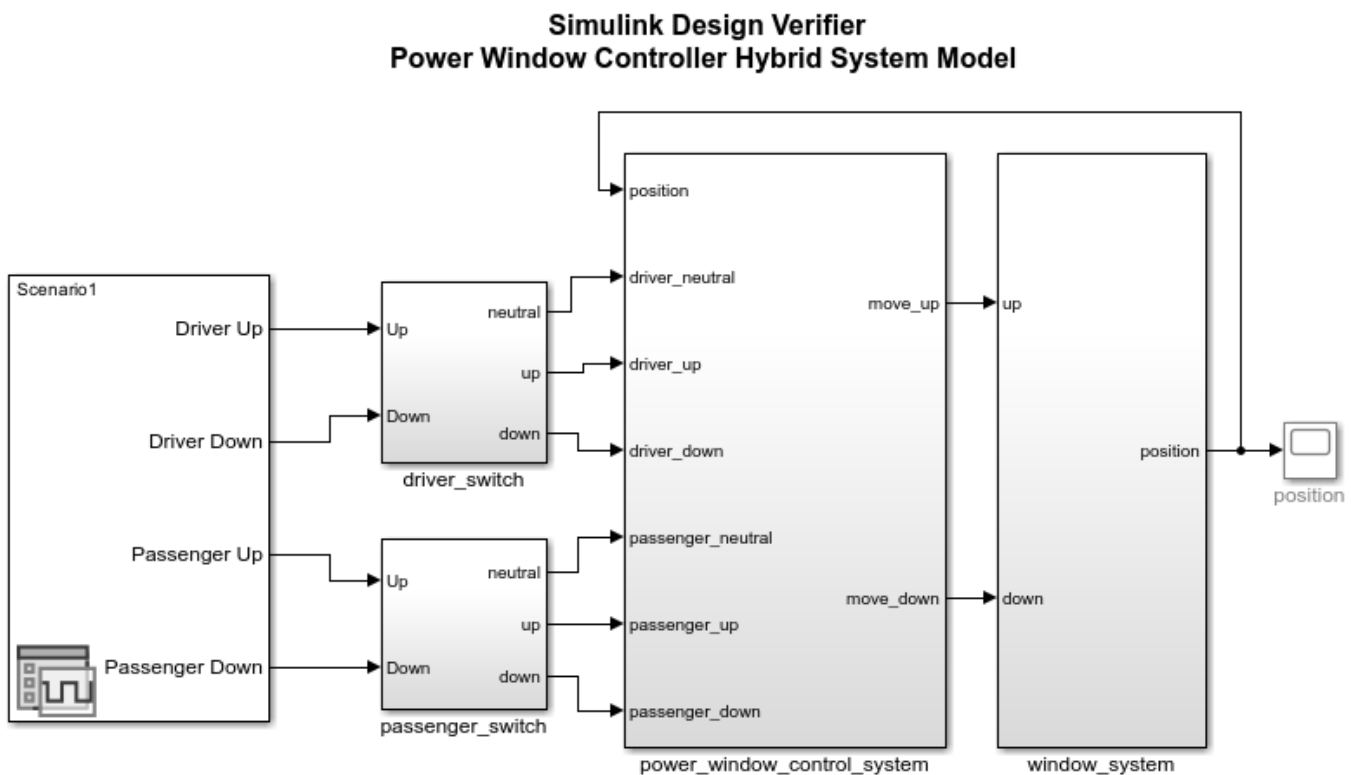
This example shows how to use Simulink® Design Verifier™ to generate test cases for a Model block by using a default top model simulation.

This example contains Model block that acts as a controller. The top model is configured for plant-in-loop simulation. You can generate test cases for a controller by using the top model simulation.

### Set Up the Default Plant-in-Loop Controller Simulation

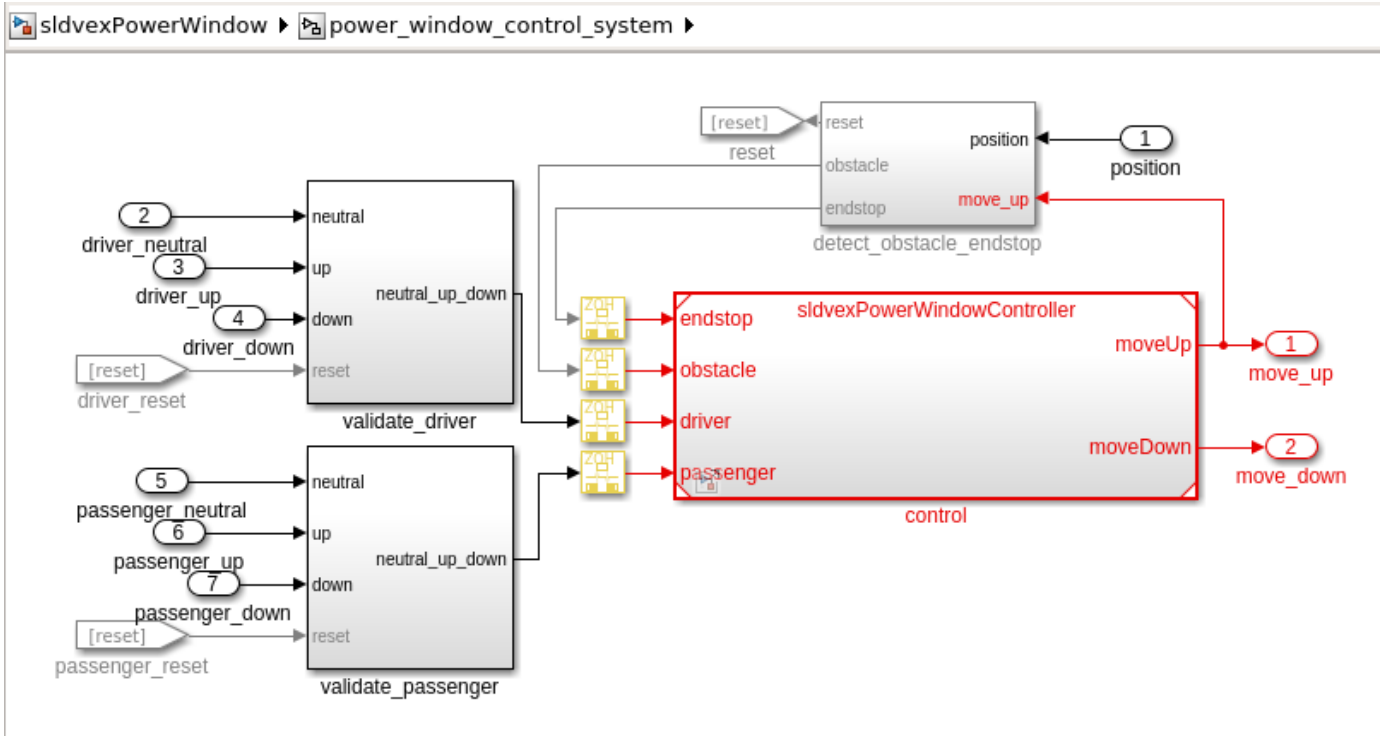
The model contains a power window controller and a low-order plant model. `sldvexPowerWindow/power_window_control_system/control` is a Model block that references the model `sldvexPowerWindowController`, which implements the controller with a Stateflow® chart.

```
open_system('sldvexPowerWindow');
```



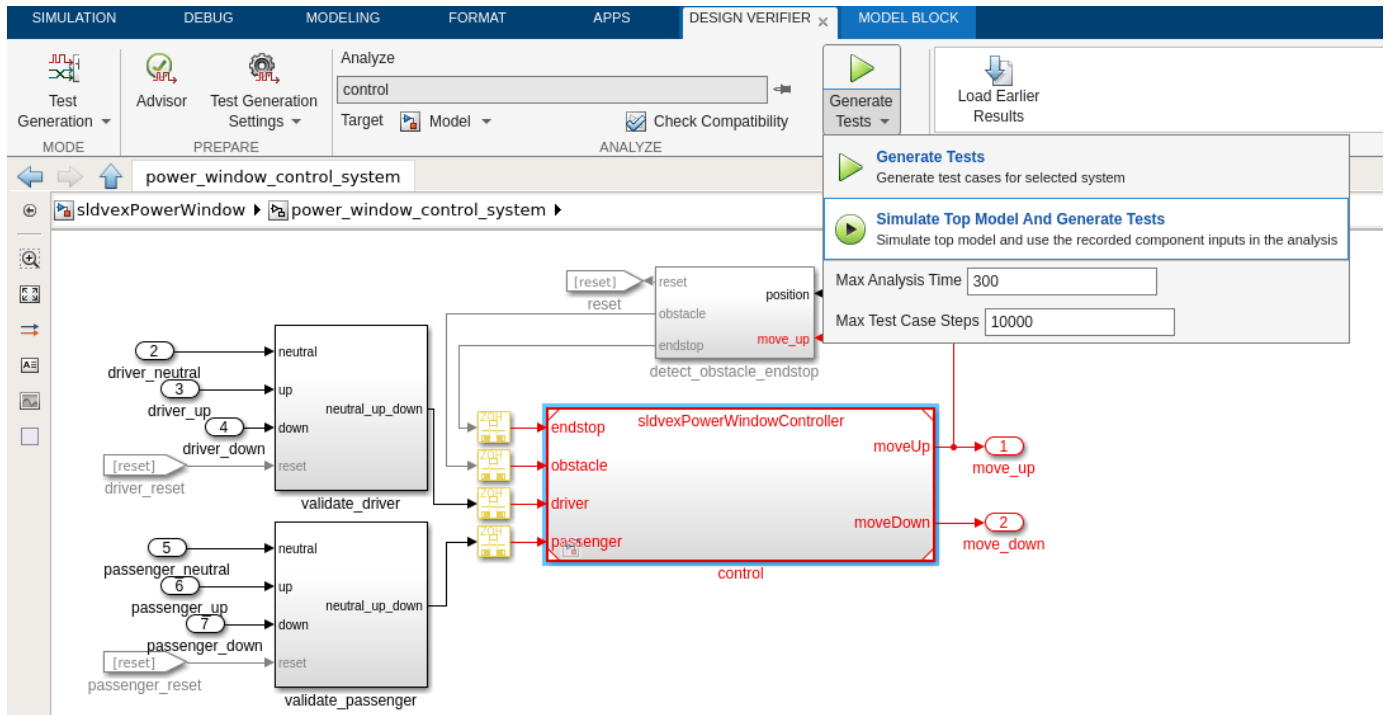
Copyright 1990-2021 The MathWorks, Inc.

This model contains a Signal Editor block at the top level. The simulation is set up as a plant-in-loop controller simulation.



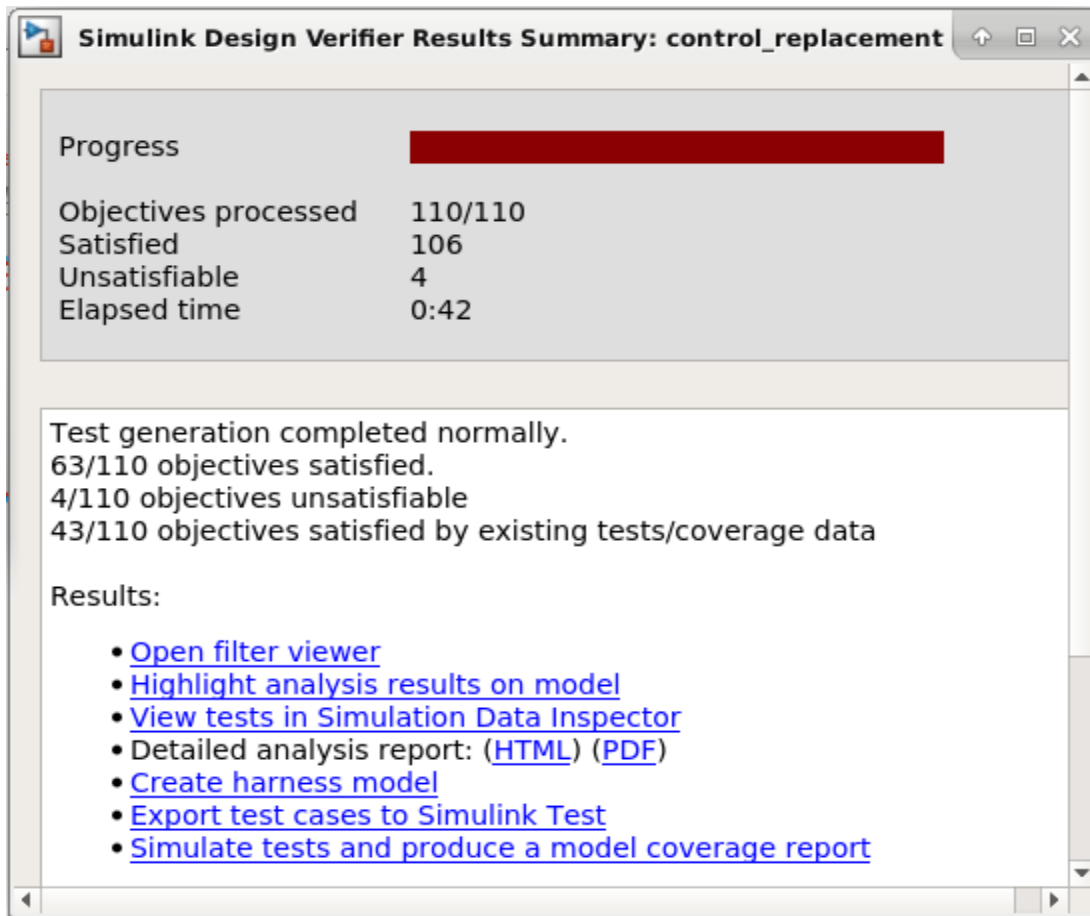
### Simulate the Top Model and Generate Test Cases for the Controller

1. In the **Apps** pane, open **Design Verifier**.
2. In the **Analyze** section, click the Remember Selection icon to unpin the current selection.
3. Select the Model block `sldvexPowerWindow/power_window_control_system/control`.
4. In the **Design Verifier** tab, expand **Generate Test** and click **Simulate Top Model And Generate Tests**.



### View Test Generation Results

Design Verifier runs the default simulation to log inputs for the Model block `sldvexPowerWindow/power_window_control_system/control`. Then Design Verifier runs a test extension on logged inputs to generate additional test cases for the controller.



### Clean Up

Close the model.

```
close_system('sldvexPowerWindow');
```

## Add Test Cases Using Excel File

This example shows how to create test cases incrementally by using Simulink® Design Verifier™ supported Excel® file.

Simulink® Design Verifier™ generates test cases to satisfy testing criteria, such as model objectives. Owing to support limitations and model complexity, occasionally it can generate test cases that do not cover all model objectives. Use this example to understand how to:

- Create test cases in Excel file format.
- Write a new test case manually in an Excel file.
- Create additional test cases by using test extension with Excel file.

### Generate Test Cases by Using Design Verifier

Open the model and create test cases.

```
model = 'sldvexSpreadsheetTopoff';
open_system(model);
[~, files] = sldvrun(model);
```

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

```
'sldvexSpreadsheetTopoff' is partially compatible for test generation with Simulink Design Verifier.
```

```
The model can be analyzed by Simulink Design Verifier.
It contains unsupported elements that will be stubbed out during analysis. The results of the analysis are shown in the Diagnostic Viewer.
See the Diagnostic Viewer for more details on the unsupported elements.
```

```
Generating tests using model representation from 22-Jul-2022 20:02:02...
```

```
Generating output files:
```

```
Results generation completed.
```

```
Data file:
```

```
C:\Users\pdasbasu\OneDrive - MathWorks\Documents\MATLAB\ExampleManager\pdasbasu.BR2022bd.j200
```

### Save Design Verifier Test Cases to Excel File

Simulink Design Verifier creates test cases in the MAT-file format by default. Save the test cases generated in the previous section in an Excel file by using any of these methods:

- Click Save to spreadsheet button in Results.
- Click Save to spreadsheet link in the results window, or results inspector.
- Use `sldvgenspreadsheet` function.

For this example, use the `sldvgenspreadsheet` function to save the test cases.

```
excelFilePath = sldvgenspreadsheet(model, files.DataFile);
```

**Note:** Importing or exporting to an Excel file is not supported for an array of bus signals. For more information, see Microsoft Excel Import, Export, and Logging Format.



## Identify Missing Coverage Objectives

Simulate the model by using all test cases from the Excel file and create a coverage report. `sldvruntest` supports test cases from a spreadsheet as simulation input.

```
runOpts = sldvruntestopts;
runOpts.coverageEnabled = true; % Enable coverage
[~, initialCov] = sldvruntest(model, excelFilePath, runOpts); % Use test cases from Excel file for
cvhtml('Initial coverage', initialCov);
```

Observe that the value of Switch block logical trigger input is never false in the coverage report.


## Switch block "[Switch](#)"

### [Justify or Exclude](#)

**Parent:** [/sldvexSpreadsheetTopoff](#)

| Metric                | Coverage                      |
|-----------------------|-------------------------------|
| Cyclomatic Complexity | 1                             |
| Decision              | 50% (1/2) decision outcomes   |
| Execution             | 100% (1/1) objective outcomes |

### Decisions analyzed

|                                       |  |
|---------------------------------------|--|
| logical trigger input                 | 50%  |
| false (output is from 3rd input port) | 0/2<br> |
| true (output is from 1st input port)  | 2/2  |

### Write Test Case to Satisfy Coverage Objective

Determine the cause of missing the coverage objective. In this example, the model contains unsupported block *Sqrt*, which limits Simulink Design Verifier analysis.

To make the value of trigger input of *Switch* block false, observe that the value of inport *In3* should be greater than 100. Add a new sheet in the Excel file with the test case.

|   | A    | B                                      | C                                      | D                                      |
|---|------|--|--|--|
| 1 | time | In1                                    | In2                                    | In3                                    |
| 2 |      | BlockPath: sldvexSpreadsheetTopoff/In1 | BlockPath: sldvexSpreadsheetTopoff/In2 | BlockPath: sldvexSpreadsheetTopoff/In3 |
| 3 |      | PortIndex: 1                           | PortIndex: 2                           | PortIndex: 3                           |
| 4 |      | Interp: zoh                            | Interp: zoh                            | Interp: zoh                            |
| 5 |      | Source: Input                          |  |  |
| 6 | 0    | 5.95663                                | 1.91836                                | 101                                    |

Verify whether the new test case satisfies the required coverage objective.

```
excelFilePath = 'WithNewTestCase.xlsx';
runOpts = sldvruntestopts;
runOpts.testIdx = 2; % Simulate only the newly added test case
runOpts.coverageEnabled = true;
[~, newTestCov] = sldvruntest(model, excelFilePath, runOpts);
cvhtml('New test coverage', newTestCov);
```

### Run Test Extension by Using Excel file

Simulink Design Verifier test extension workflow generates new test cases by extending the existing test cases. This helps to satisfy additional coverage objectives by extending your new test case.

```
opts = sldvoptions(model);
opts.ExistingTestFile = excelFilePath; % Use Excel file with new test cases as input for test ext
opts.ExtendExistingTests = 'on'; % Enable test extension
[~, files] = sldvrun(model, opts);
```

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

```
'sldvexSpreadsheetTopoff' is partially compatible for test generation with Simulink Design Verifier.
```

```
The model can be analyzed by Simulink Design Verifier.
It contains unsupported elements that will be stubbed out during analysis. The results of the analysis
See the Diagnostic Viewer for more details on the unsupported elements.
```

```
Loading initial test data...done
```

```
Generating tests using model representation from 22-Jul-2022 20:02:34...
```

```
Generating output files:
```

```
Results generation completed.
```

```
Data file:
C:\Users\pdasbasu\OneDrive - MathWorks\Documents\MATLAB\ExampleManager\pdasbasu.BR2022bd.j20220722
```

### Verify Complete Coverage



Simulate the model using the new test cases and verify that you now have complete coverage.

```
runOpts = sldvruntestopts;
runOpts.coverageEnabled = true; % Enable coverage
[~, finalCov] = sldvruntest(model, files.DataFile, runOpts);
cvhtml('Final coverage', finalCov);
close_system(model, 0);
```

If the new test cases still yield partial coverage, you can write a new test case in an Excel file and then run test extension workflow till complete coverage is achieved.

# Summary

## Model Hierarchy/Complexity Test 1

|  |   | <b>Decision</b>   | <b>Execution</b>   |
|--|---|---|--|
| 1. <a href="#">sldvexSpreadsheetTopoff</a> | 3 | 100%  | 100%  |

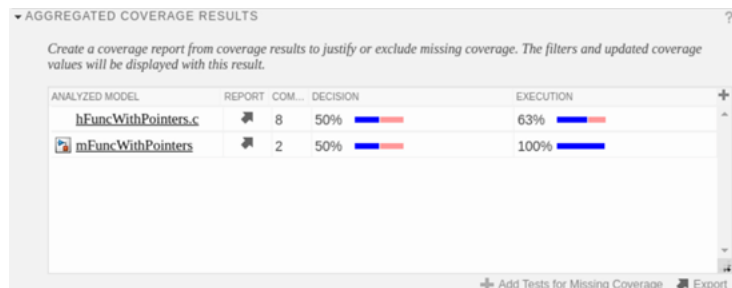
## Achieve Missing Coverage in Custom Code

This example shows you how to test for missing coverage in custom code. You can use these steps to also test for missing coverage in external C code. If you simulate a model with custom code through C Caller block, C Caller Library, or coder.ceval function, then coverage of the custom code is reported. If the code does not achieve full coverage, you can use Simulink® Design Verifier™ to generate test cases that achieve full coverage. You can then use Simulink® Test Manager™ to perform unit testing by generating test cases only for the custom code.

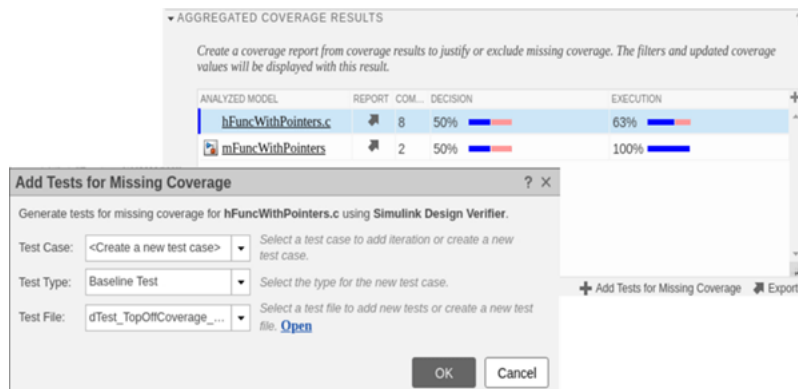
1. Open the test file.

```
testFile = 'dTest_TopOffCoverage_mFuncWithPointers.mldatx';
sltest.testmanager.load(testFile);
sltest.testmanager.view;
```

2. Simulate the test file and observe the coverage value in the Aggregated Coverage Results window.



3. As the coverage of the custom code is not 100%, click Add Tests for Missing Coverage.

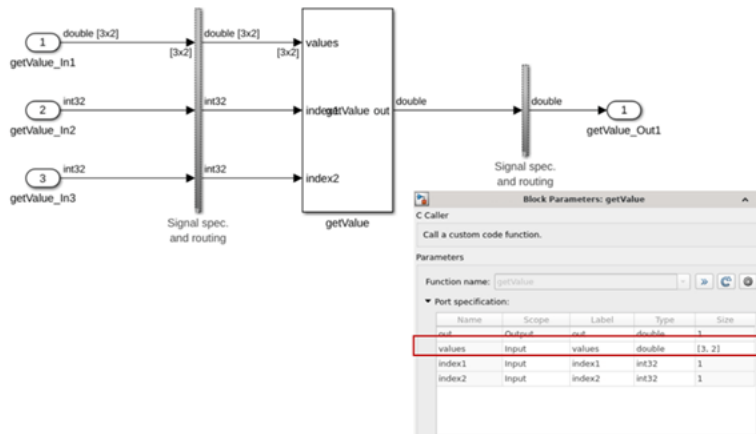


4. Simulink Design Verifier generates additional test cases. The custom code file, hFuncWithPointers.c contains two functions: getValue and getValuePointer.

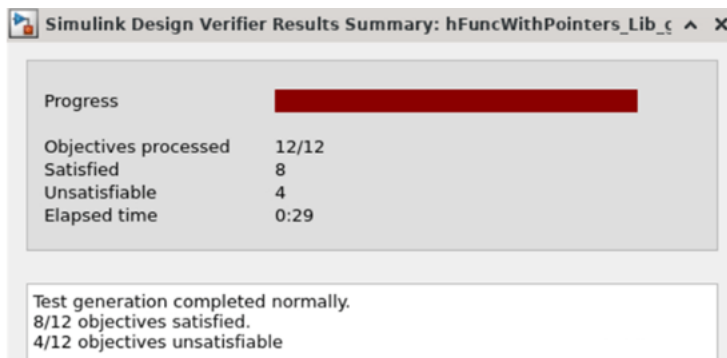
Function 1: The function **getValue** has vector and scalar inputs. The dimensions are specified for the vector inputs.

The inputs of the harness constructed from this code contain the proper dimensions of the signals. Therefore, Simulink Design Verifier may not be able to generate correct test cases for the code.

The harness generated for this code is as shown:

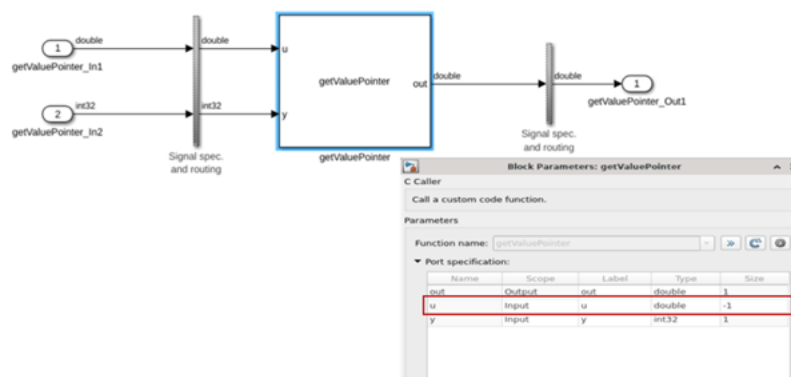


The analysis result is as shown here:

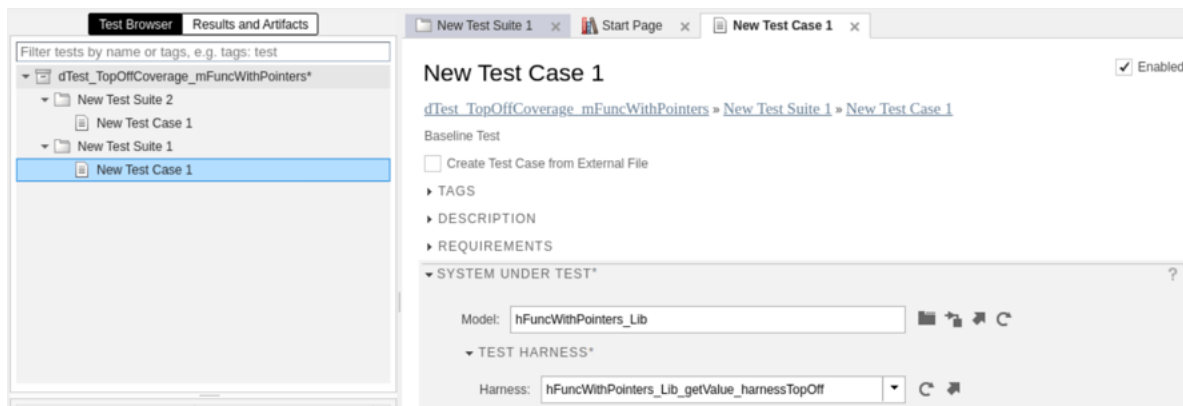


Function 2: The function **getValuePointer** also has vector and scalar inputs. The dimensions are not specified for the vector inputs.

The inputs of the harness constructed from this code contain the dimensions of the signals to be inherited. Simulink Design Verifier cannot generate correct test cases for the code.



5. Add new test cases for the function **getValue** as shown:



From the warning message, you will get a list of functions for which Simulink Design Verifier is not invoked, and the list of corresponding harness names which you need to update manually, to achieve additional coverage. In this example, for the function **getValuePointer** no additional test case is generated, and **hFuncWithPointers\_Lib\_getValuePointer\_harnessTopOff.slx** is the corresponding harness that you need to update.

6. To manually add additional test cases for the function, **getValuePointer**, update the port specification of the **C Caller** block **getValuePointer** of the generated library model **hFuncWithPointers\_Lib**. Open the block parameter of the **C Caller** block **getValuePointer**, and update the required port dimension and save the model.

7. Add a new test suite for the updated harness, in the existing test file.

8. Simulate the test suite and generate additional test cases by using **Add Tests for Missing Coverage**. You have now generated test cases for missing coverage in custom code.

## Achieve Missing Coverage in Generated Code of RLS

This example shows you how to use Simulink® Design Verifier™ to generate test cases that achieve full coverage. If you simulate a harness of a reusable library subsystem (RLS) in the software-in-the-loop (SIL) simulation mode, then coverage of the generated code of the RLS is reported. Using Simulink® Test Manager™, you can easily achieve full coverage by using the following steps:

Generate the top-model code before invoking simulation on the harness of the RLS. Before generating the code, you need to set up the code generation target environment. For more information on how to set up the code generation environment, see “Generate Test Cases for RLS in Software-in-the-Loop Mode” on page 7-21. After code generation, open the test file.

1. Open the test file.

```
orig = Simulink.fileGenControl('get','CodeGenFolderStructure');
Simulink.fileGenControl('set','CodeGenFolderStructure', Simulink.filegen.CodeGenFolderStructure);
load_system('mRLS');
slbuild('mRLS');
```

```
testFile = 'dTest_TopOffCoverage_Controller.mldatx';
sltest.testmanager.load(testFile);
sltest.testmanager.view;
```

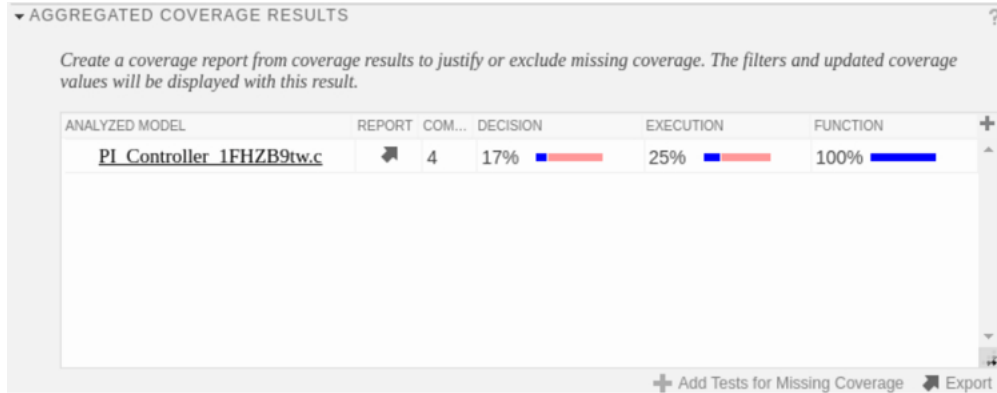
```
### Starting build procedure for: Controller_CodeSpecification1
### Generating code and artifacts to 'Target_environment_subfolder' folder structure
### Generating code into build folder: C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldev-
### Invoking Target Language Compiler on Controller_CodeSpecification1.rtw
### Using System Target File: B:\matlab\rtw\c\ert\ert.tlc
### Loading TLC function libraries
.....
### Initial pass through model to cache user defined code
.
### Caching model source code
.....
### Writing header file Controller_Lp0dbbft.c
### Writing header file Controller_CodeSpecification1_types.h
### Writing header file Controller_CodeSpecification1.h
### Writing header file rtwtypes.h
.
### Writing header file Controller_Lp0dbbft.h
### Writing source file Controller_CodeSpecification1.c
### Writing header file Controller_CodeSpecification1_private.h
### Writing source file ert_main.c
### TLC code generation complete (took 5.586s).
### Saving binary information cache.
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldev-ex58892879\IntelWin64\_sha
### Using toolchain: Microsoft Visual C++ 2019 v16.0 | nmake (64-bit Windows)
### Creating 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\28\tp27a1e6fc\sldev-ex58892879\IntelWin64\Cont
### Successful completion of code generation for: Controller_CodeSpecification1
```

The following files will be copied from IntelWin64\\_shared to C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\28\tp27a1e6fc\sldev-ex58892879\IntelWin64\\_shared:

```
Controller_Lp0dbbft.c
Controller_Lp0dbbft.h
```

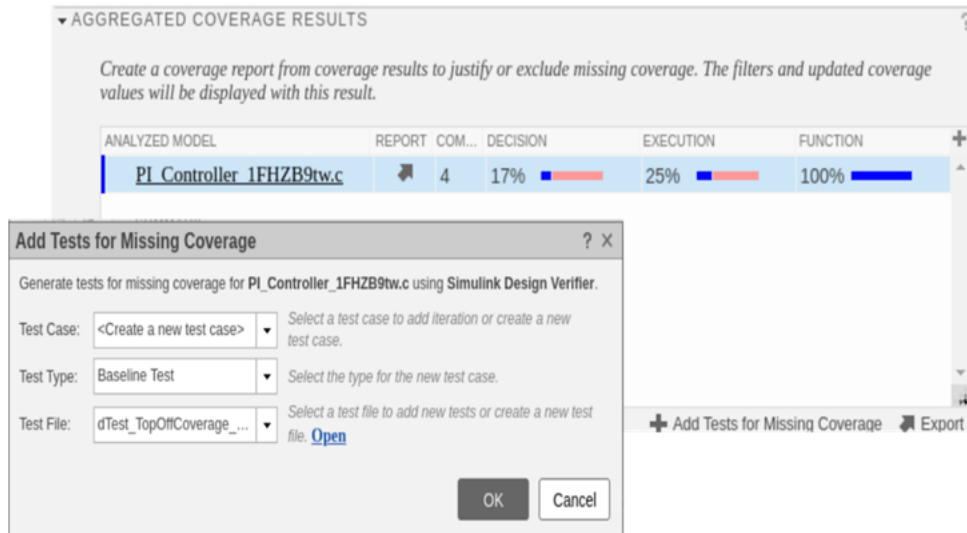
shared\_file.dmr

Files copied from IntelWin64\\_shared to C:\TEMP\Bdoc23a\_2213998\_3568\ib570499\28\tp27a1e6fc\slvdv



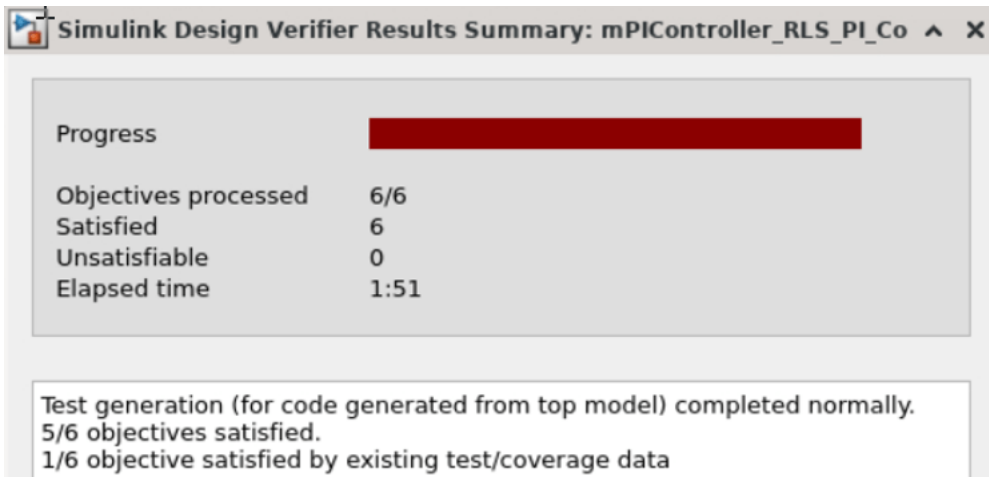
2. Simulate the test file and observe the coverage value.

3. Click Add Tests for Missing Coverages as coverage of the generated code for the RLS is not 100%.



This workflow invokes Simulink® Design Verifier™ to generate additional testcases.





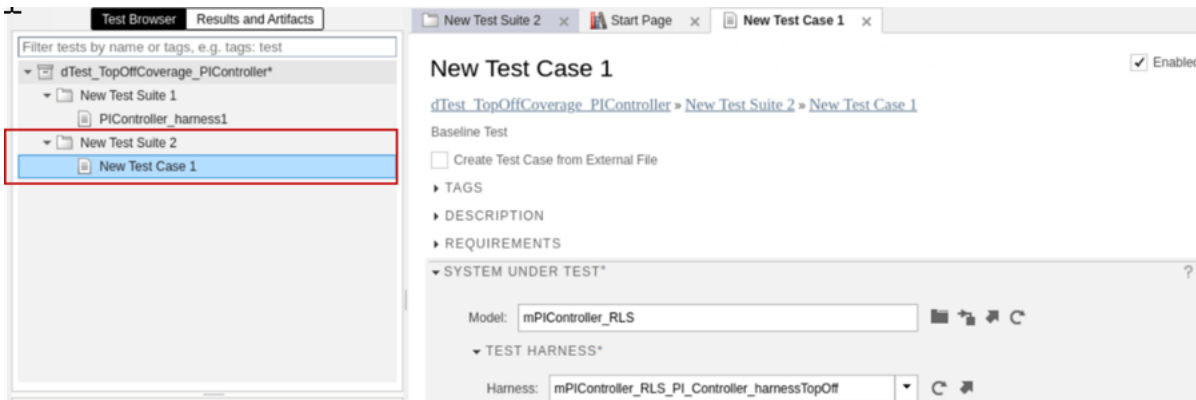
**Simulink Design Verifier Results Summary: mPIController\_RLS\_PI\_Co**

Progress

|                      |      |
|----------------------|------|
| Objectives processed | 6/6  |
| Satisfied            | 6    |
| Unsatisfiable        | 0    |
| Elapsed time         | 1:51 |

Test generation (for code generated from top model) completed normally.  
5/6 objectives satisfied.  
1/6 objective satisfied by existing test/coverage data

New test cases are added to the test file.



Test Browser | Results and Artifacts

Filter tests by name or tags, e.g. tags: test

- dTest\_TopOffCoverage\_PIController\*
  - New Test Suite 1
    - PIController\_harness1
  - New Test Suite 2
    - New Test Case 1

**New Test Case 1** Enabled

dTest\_TopOffCoverage\_PIController » New Test Suite 2 » New Test Case 1

Baseline Test

Create Test Case from External File

▶ TAGS

▶ DESCRIPTION

▶ REQUIREMENTS

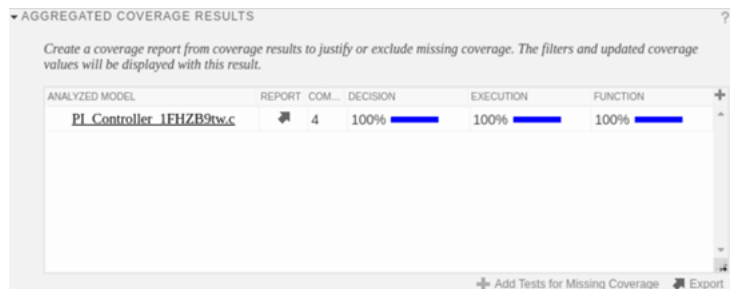
▼ SYSTEM UNDER TEST\*

Model: mPIController\_RLS

▼ TEST HARNESS\*

Harness: mPIController\_RLS\_PI\_Controller\_harnessTopOff

4. Simulate the overall test file and check if you now have full coverage for the generated code for the RLS.



AGGREGATED COVERAGE RESULTS

Create a coverage report from coverage results to justify or exclude missing coverage. The filters and updated coverage values will be displayed with this result.

| ANALYZED MODEL           | REPORT | COM. | DECISION | EXECUTION | FUNCTION |
|--------------------------|--------|------|----------|-----------|----------|
| PI_Controller_1FHZB9rw.c | 4      | 100% | 100%     | 100%      | 100%     |

+ Add Tests for Missing Coverage | Export



# Extending Existing Test Cases

---

- “When to Extend Existing Test Cases” on page 8-2
- “Extend Test Cases for Model with Temporal Logic” on page 8-4
- “Extend Test Cases for Closed-Loop System” on page 8-10
- “Extend Test Cases for Modified Model” on page 8-15
- “Create and Run Back-to-Back Tests Using Enhanced MCDC” on page 8-18

## When to Extend Existing Test Cases

### In this section...

“Common Workflow for Extending Existing Test Cases” on page 8-2

“Considerations for Starting Test Cases” on page 8-3

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

- You encounter delays trying to analyze your model, or you see incomplete results. This can happen if your model has any of the following characteristics:
  - Temporal logic
  - Large counters
  - Model objects that are difficult to test due to complex or nonlinear logic

Analyzing the model and considering the existing test cases allows you to focus the analysis on those parts of the model that are difficult to analyze. You can combine the generated test cases to create a complete test suite for the full model.

For an example of extending existing test cases for a model that uses temporal logic, see “Extend Test Cases for Model with Temporal Logic” on page 8-4.

- You have a closed-loop simulation model that uses a Model block to include the controller. First, log the data from the Model block and then analyze the model referenced by the Model block. Using this technique, the test cases for the controller can realistically reflect the continuous time behavior expected in the closed-loop system.

For an example of extending existing test cases for a closed-loop system, see “Extend Test Cases for Closed-Loop System” on page 8-10.

- You change an existing model for which you have already generated test cases. In this situation, you can reanalyze the model, omitting the analysis results from the original version of the model. The combined test cases give you a complete test suite for the new model.

For an example of extending existing test cases for modified models, see “Extend Test Cases for Modified Model” on page 8-15.

- You apply parameter configurations or update the parameter constraint values of an existing model for which you have generated test cases. In this situation, you can reanalyze the model by reusing the previously generated test cases and extend them to achieve full model coverage. For an example of extending existing test cases when you modify parameter configurations, see “Extend Existing Test Cases After Applying Parameter Configurations” on page 5-46.

## Common Workflow for Extending Existing Test Cases

Use the following workflow for extending existing test cases during a test-generation analysis:

- Create the starting test cases.
- Log the starting test cases.

- Extend the existing test cases during test-generation analysis.
- Verify that you have created a complete test suite.

The examples in this category use some or all of these tasks when extending existing test cases during analysis.

## **Considerations for Starting Test Cases**

If the existing test cases are inconsistent with the model, Simulink Design Verifier ignores the test cases during test case extension. For example, if you update the constraint values of parameters and the existing test case violates the specified constraint values, the test case will be ignored.

## **See Also**

### **More About**

- “Extend Test Cases for Model with Temporal Logic” on page 8-4
- “Extend Test Cases for Closed-Loop System” on page 8-10
- “Extend Test Cases for Modified Model” on page 8-15

## Extend Test Cases for Model with Temporal Logic

### In this section...

“Create Starting Test Case” on page 8-4

“Log Starting Test Case” on page 8-6

“Extend Existing Test Cases” on page 8-7

“Verify Analysis Results” on page 8-8

### Create Starting Test Case

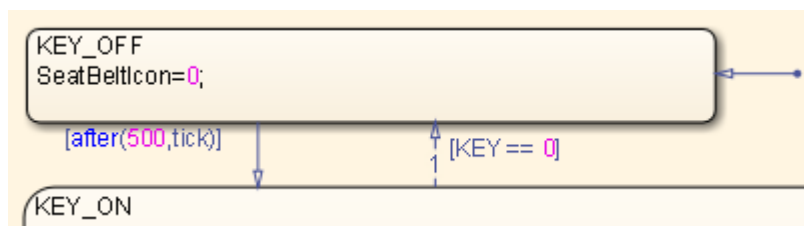
This example uses the `sldvdemo_sbr_extend_design` model. This model includes a Stateflow chart SBR that uses temporal logic. The transition from the `KEY_OFF` state to the `KEY_ON` state occurs after the Stateflow chart has been simulated 500 times. To test this transition requires a test case with 500 time steps.

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

- 1 Open the example model:

```
sldvdemo_sbr_extend_design
```

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

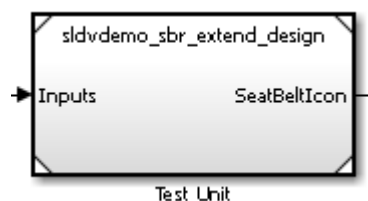


- 3 Create a model reference harness model:

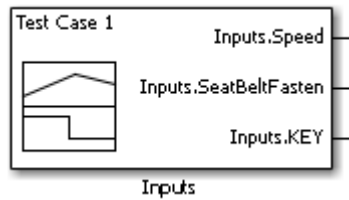
```
[~, harnessModelFilePath] = ...
sldvmakeharness('sldvdemo_sbr_extend_design', [], [], true);
```

The harness model, `sldvdemo_sbr_extend_design_harness`, includes:

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

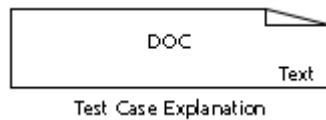


- A Signal Builder block named Inputs that contains the test-case inputs to the model referenced in the Model block.



Initially, the Signal Builder block contains only the default test case, with all three inputs set to 0.

- A DocBlock block named Test Case Explanation that documents the test case.



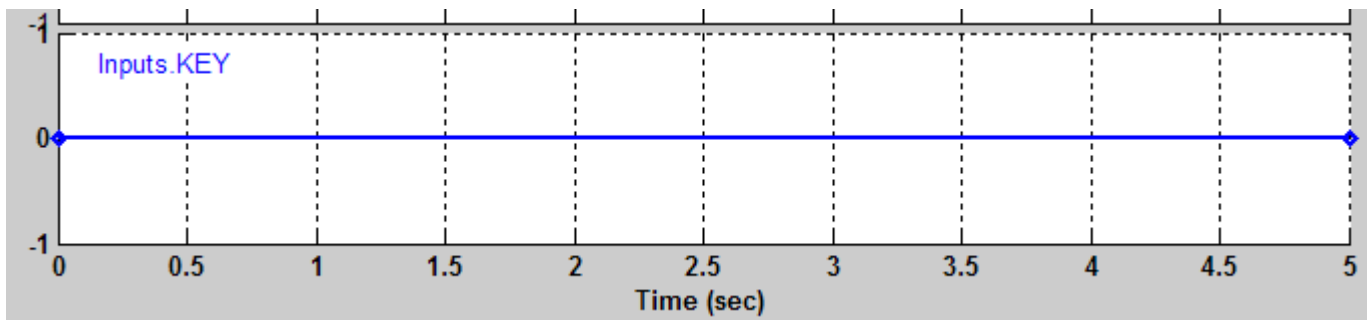
Initially, the Test Case Explanation block does not have any content for the default test case.

- 4 `sldvmakeharness` returns the path to the harness model file in `harnessModelFilePath`. Extract the name of the harness model file into `harnessModel`, for later use:

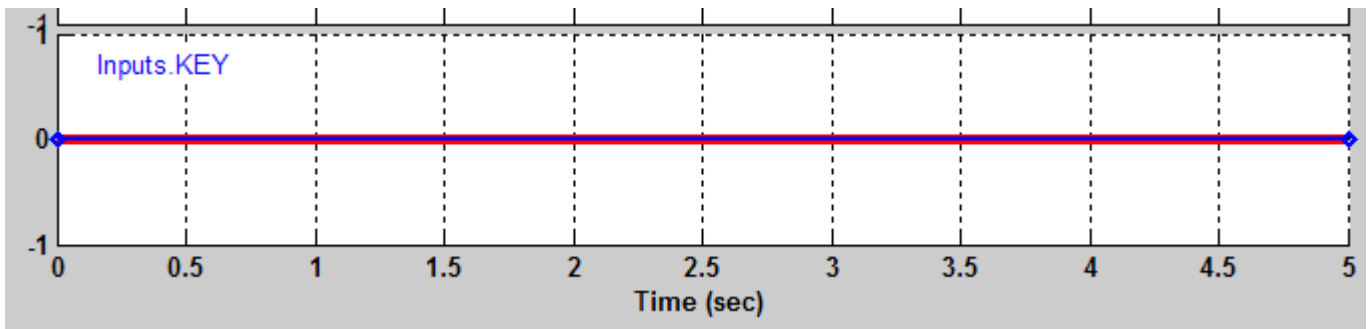
```
[~, harnessModel] = fileparts(harnessModelFilePath);
```

In order to analyze the KEY\_OFF to KEY\_ON state transition, create a test case that makes the transition to the KEY\_ON state in 500 time steps:

- 1 Open the Signal Builder dialog box for the harness model.
- 2 Select **Axes > Change Time Range**.
- 3 The Signal Builder's time range determines the span of time over which its output is explicitly defined. In the Set the total time range dialog box, set the **Max time** field to 5 seconds, creating 500 time steps of 0.01 seconds duration each.
- 4 Set the KEY input to 1 for the duration of this starting test case, forcing the transition to the KEY\_ON state. Selecting the `Inputs.KEY` signal requires two clicks. First, click the signal so that dots appear at both ends of the signal.

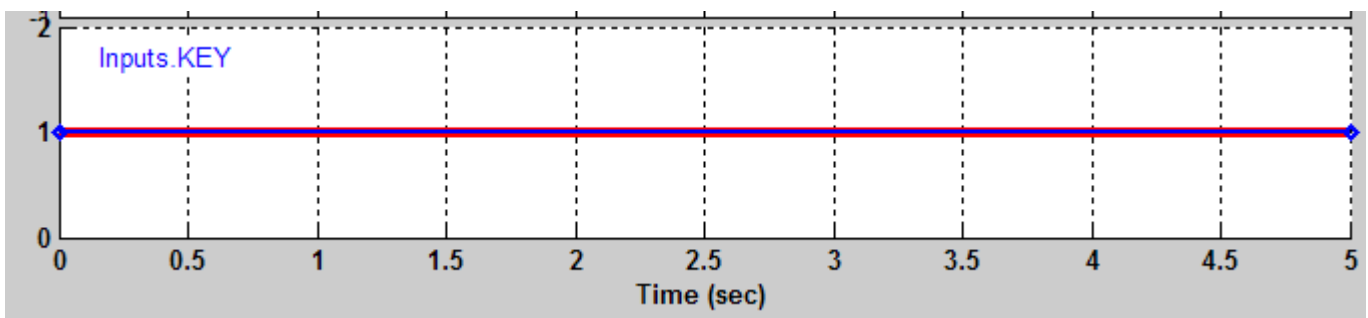


- 5 Click the `Inputs.KEY` signal again. The Signal Builder thickens the signal to indicate that it is selected.



- 6 At the bottom of the Signal Builder dialog box, under **Left Point**, enter 1 for **Y**.
- 7 Press **Enter** to apply the change.

The `Inputs.KEY` signal is set to 1 for the duration of the test case.



- 8 Close the Signal Builder dialog box.

## Log Starting Test Case

The next step is to log the starting test case that you created. You can then specify that Simulink Design Verifier ignore the objectives satisfied by that test case when performing an analysis.

The `sldvlogsignals` function records the test case data in a MAT-file that contains an `sldvData` structure. This structure stores all the data that the software gathers and produces during the analysis.

To log the starting test cases:

- 1 Save the name of the Model block in the harness model that references the `sldvdemo_sbr_extend_design` model:
 

```
[~, modelBlock] = find mdlrefs(harnessModel, false);
```
- 2 Simulate the model referenced by the Model block using the new test case, and log the input signals in the workspace variable `loggeddata`:
 

```
loggeddata = sldvlogsignals(modelBlock{1});
```
- 3 Save the logged data in a MAT-file named `existingtestcase.mat`:
 

```
save('existingtestcase.mat', 'loggeddata');
```

You will specify this file when you analyze the `sldvdemo_sbr_extend_design` model.



## Extend Existing Test Cases

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

Specify the starting test case and analyze the model:

- 1 Open the model.

```
open_system('sldvdemo_sbr_extend_design');
```

- 2 On the **Design Verifier** tab, click **Test Generation Settings**.

- 3 In the Configuration Parameters dialog box, on the **Test Generation** pane, under **Existing test cases**, select **Extend existing test cases**.

- 4 In the **Data file** field, enter the name of the MAT-file that contains the logged data:

```
existingtestcase.mat
```

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

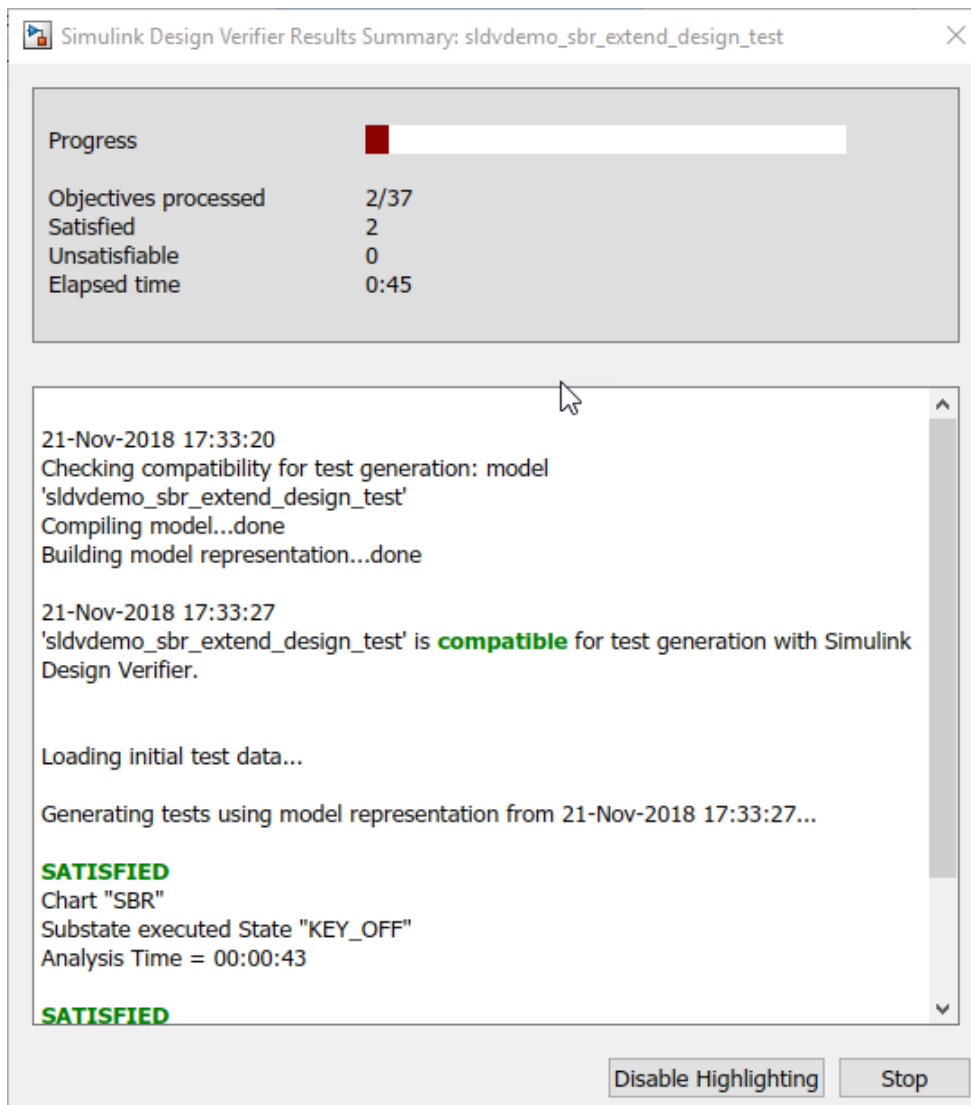
When you clear this option, the software includes the starting test case in the final test suite. You will see that the complete test suite achieves 100% model coverage.

- 6 To close the Configuration Parameters dialog box, click **OK**.

- 7 Save the `sldvdemo_sbr_extend_design` model on the MATLAB path with the name `sldvdemo_sbr_extend_design_test`.

- 8 Click **Generate Tests**.

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



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

## Verify Analysis Results

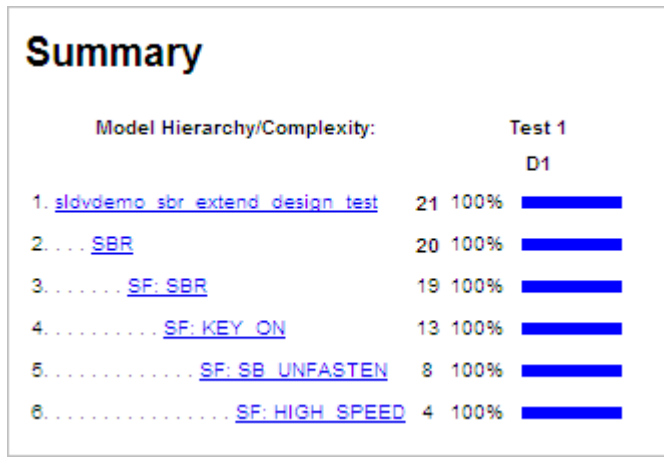
To make sure that this analysis creates a complete test suite, generate the harness model so you can simulate the model with the generated test cases:

- 1 On the **Design Verifier** tab, in the **Review Results** section, click **Create Test Harness Model**.
- 2 In the harness model `sldvdemo_sbr_extend_design_test_harness`, open the Signal Builder block named `Inputs`.
- 3 To simulate the model using all the test cases, click the **Run all and produce coverage** button



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

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



## See Also

### Related Examples

- “Component-Based Modeling with Model Reference”

### More About

- “When to Extend Existing Test Cases” on page 8-2
- “Extend Test Cases for Closed-Loop System” on page 8-10
- “Extend Test Cases for Modified Model” on page 8-15

## Extend Test Cases for Closed-Loop System

### In this section...

“Log Starting Test Case” on page 8-10

“Extend 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.

### Log Starting Test Case

This example uses the “Component-Based Modeling with Model Reference” example model `sldemo_mdref_basic`. `sldemo_mdref_basic` model. The CounterA Model block references the model `sldemo_mdref_counter`. When you simulate the parent model, `sldemo_mdref_basic`, and collect coverage, you record only 75% coverage for `sldemo_mdref_counter`. Log the data from the simulation and extend those test cases to achieve 100% coverage for the referenced model.

- 1 Open the “Component-Based Modeling with Model Reference” example model `sldemo_mdref_basic`.

```
openExample('sldemo_mdref_basic')
```

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

Under **Model Verification, Validation, and Test**, click **Coverage Analyzer**.

- 3 On the **Coverage** tab, click **Settings**.
- 4 In the **Coverage** pane of the Configuration Parameters, select **Enable coverage analysis**.
- 5 Select **Referenced Models**.

Note that the analysis records coverage only for referenced models with **Simulation mode** set to Normal, SIL, or PIL. In `sldemo_mdref_basic`, the CounterC Model block has **Simulation mode** set to Accelerator, so you cannot record coverage for it.

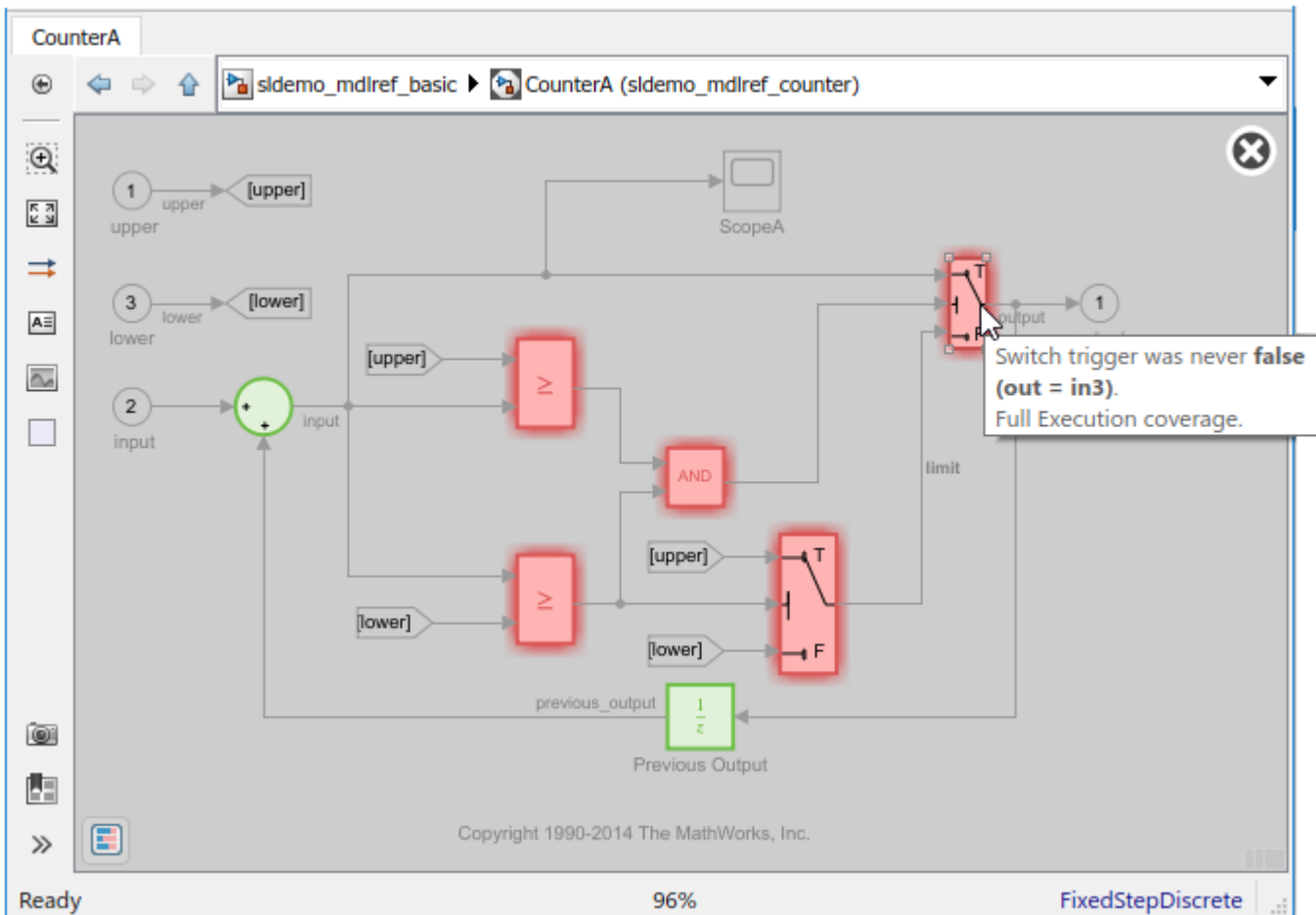
- 6 Under **Coverage metrics**, set the structural coverage level to **Modified Condition Decision Coverage (MCDC)** to record decision, condition, and modified condition/decision coverage.
- 7 Click **OK**.
- 8 Click **Analyze Coverage**.

To open the coverage report, in the **Review Results** section, click **Generate Report**.

When the simulation completes, the generated coverage report opens in a browser window. The report shows the following coverage results for the referenced model:

- Condition: 50% (2/4) condition outcomes
- Decision: 25% (1/4) decision outcomes
- MCDC: 0% (0/2) conditions reversed the outcome

The coverage results are also highlighted in the referenced model, `sldemo_mdref_counter`. You can select individual model objects to view specific coverage results in the Coverage dialog box, as shown in the following screenshot.



- 9 To log the input signals for the CounterA Model block in `sldemo_mdref_basic` during simulation, at the MATLAB command prompt, enter the following code:

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

- 10 Save the logged data in a MAT-file named `existingtestcase.mat`:

```
save('existingtestcase.mat', 'logged_data');
```

When you analyze the model referenced in CounterA (`sldemo_mdref_counter`) to extend existing test cases, you specify this MAT-file.

## Extend Existing Test Cases

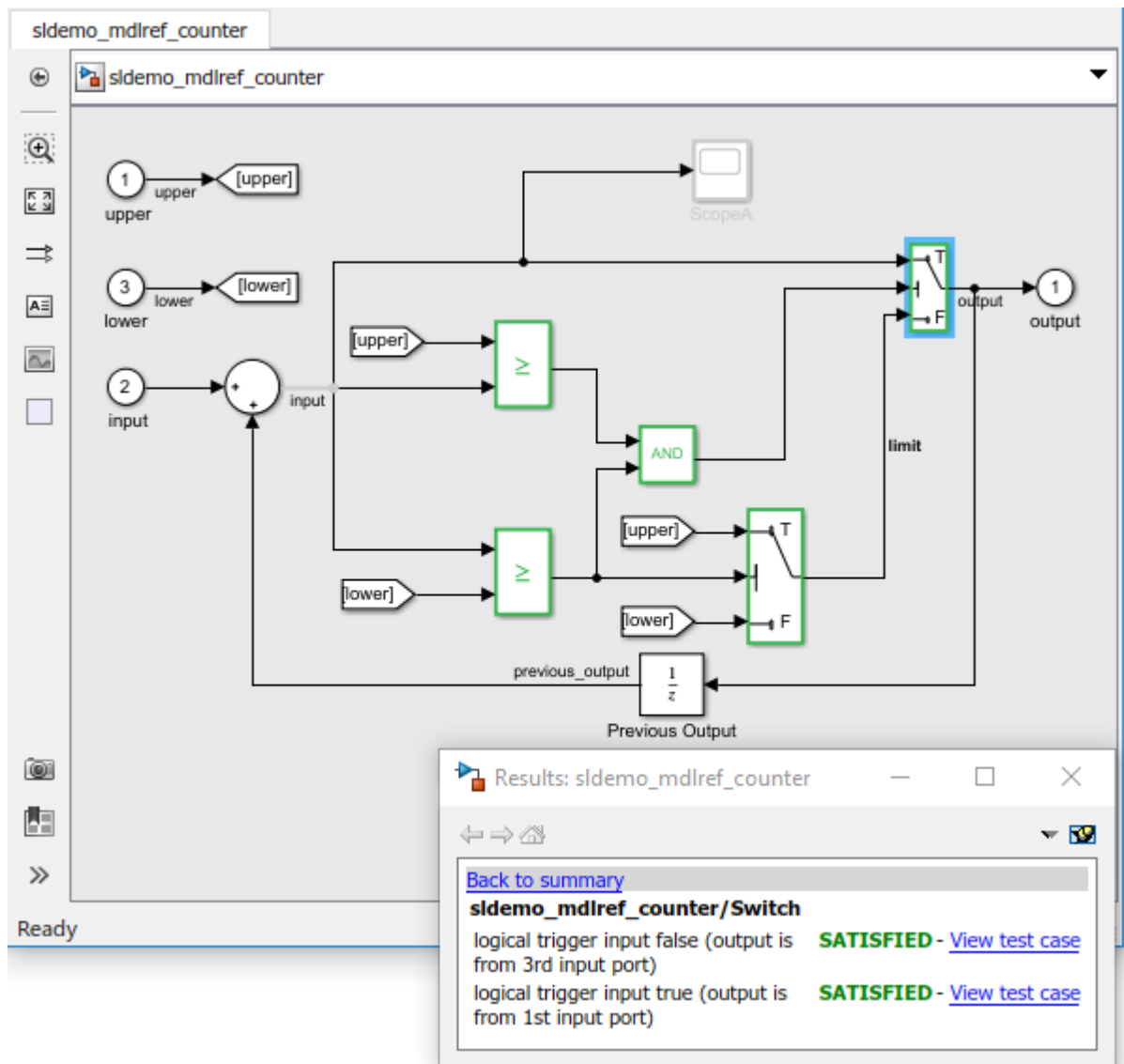
Analyze the `sldemo_mdhref_counter` model, specifying that the analysis extend the test cases already satisfied:

- 1 To open the `sldemo_mdhref_counter` model, in the `sldemo_mdhref_basic` model, double-click the CounterA Model block.
- 2 On the **Design Verifier** tab, click **Test Generation Settings**.
- 3 In the Configuration Parameters dialog box, on the **Test Generation** pane, in the **Model coverage objectives** box, select MCDC.
- 4 Under **Advanced parameters**, select **Add tests for the missing coverage**.
- 5 Select the **Extend using existing data** check box.
- 6 In **Coverage Data** field, specify the name of the MAT-file that contains the logged data, in this case, `existingtestcase.mat`
- 7 Click **OK**.
- 8 Click **Generate Tests**.

The analysis first loads the objectives satisfied by the logged test cases. Then it adds extra time steps to those test cases and tries to satisfy any missing objectives. When the analysis completes, the Simulink Design Verifier log window opens and indicates that all 12 objectives are satisfied.

- 9 To view the analysis results on the model, in the Simulink Design Verifier log window, select **Highlight analysis results on model**.

The Simulink Design Verifier results are highlighted in the referenced model, `sldemo_mdhref_counter`. You can select individual model objects to view specific analysis results in the Simulink Design Verifier Results dialog box, as shown in the following screenshot.



- 10 To verify the results of the analysis and review the generated test cases, in the Simulink Design Verifier log window, select **Generate detailed analysis report**.
- 11 To collect model coverage using the extended test suite, in the Simulink Design Verifier log window, select **Simulate tests and produce a model coverage report**.

When the simulation completes, the generated coverage report opens in a browser window. The report now shows the following coverage results for the referenced model `sldemo_mdref_counter`:

- Condition: 100% (4/4) condition outcomes
- Decision: 100% (4/4) decision outcomes
- MCDC: 100% (2/2) conditions reversed the outcome

## **See Also**

### **Related Examples**

- “Component-Based Modeling with Model Reference”

### **More About**

- “When to Extend Existing Test Cases” on page 8-2
- “Extend Test Cases for Model with Temporal Logic” on page 8-4
- “Extend Test Cases for Modified Model” on page 8-15



## Extend Test Cases for Modified Model

### In this section...

“Create Starting Test Cases” on page 8-15

“Extend Existing Test Cases” on page 8-15

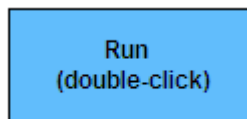
Suppose that you have a model that you have already analyzed using Simulink Design Verifier, and you modify the model. The original test suite may not record 100% coverage for the modified model. Reanalyze the modified model to make sure that it satisfies all the new test objectives. Instead of reanalyzing the entire model, you focus the new analysis on just the modified part of the model. In this way, you leverage the test cases created for the original model, extending them to satisfy any new objectives.

This example uses the `sldvdemo_cruise_control` model. You analyze the model and generate test cases. Then you analyze a modified version of that model, `sldvdemo_cruise_control_mod`, extending the test cases from the original analysis. The analysis returns a complete test suite for the new model.

### Create Starting Test Cases

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

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



Run Simulink Design Verifier

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

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

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

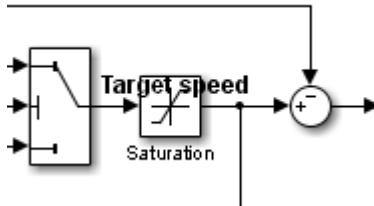
- 3 Close the `sldvdemo_cruise_control` model and all the files created by the analysis. If asked, do not save any changes you made to the model.

### Extend Existing Test Cases

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

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

- 1 Open the example model, the modified version of `sldvdemo_cruise_control`:  
`sldvdemo_cruise_control_mod`
- 2 Double-click the Controller subsystem to see the change to the original model, a Saturation block that specifies the maximum speed:



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

`sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_sldvdata.mat`

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

When you clear this option, the analysis includes the test cases recorded in the file `sldvdemo_cruise_control_sldvdata.mat` in the final test suite.

- 8 Click **Apply** to save these settings.
- 9 To start the analysis, click **Generate Tests**.

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

- 10 In the Results Summary window, click **Generate detailed analysis report**.

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

## Objectives Satisfied

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

| # | Type     | Model Item                            | Description   | Test Case          |
|---|----------|---------------------------------------|---|--------------------|
| 1 | Decision | <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> |

## See Also

### More About

- “When to Extend Existing Test Cases” on page 8-2
- “Extend Test Cases for Model with Temporal Logic” on page 8-4
- “Extend Test Cases for Closed-Loop System” on page 8-10

## Create and Run Back-to-Back Tests Using Enhanced MCDC

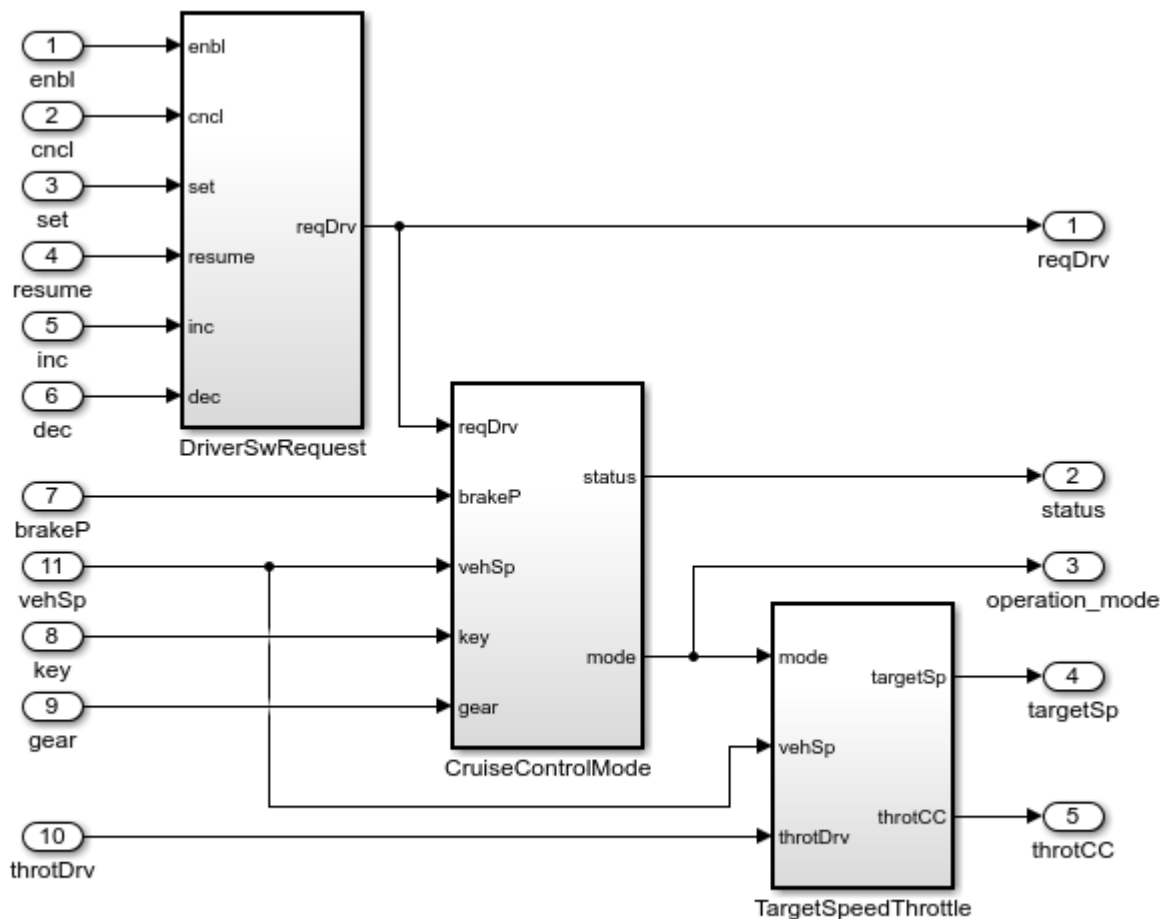
This example shows you how to create and run a back-to-back test using enhanced MCDC. Enhanced MCDC analyzes the detectability of each objective in the model and generates non-masking test cases for each objective. For more information, see “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42.

Back-to-back tests in Simulink® Test™ compare the results of normal simulations with the generated code results from software-in-the-loop, processor-in-the-loop, or hardware-in-the-loop simulations.

### Section 1: Prepare the Model

1. Open the model:

```
model = ('sldvSliceCruiseControl');
open_system(model);
```



Copyright 2015 The MathWorks, Inc.

2. Prepare the model for code generation and logging.

```

set_param(model, 'ProdHWDeviceType', 'Intel->x86-64 (Linux 64)');
set_param(model, 'ProdLongLongMode', 'on');
set_param(model, 'SaveOutput', 'on');
set_param(model, 'SignalLogging', 'on');
set_param(model, 'SaveFormat', 'Dataset');

```

Note: You can also optionally mark internal signals in the model as test-pointed logged signals (for example, `sldvSliceCruiseControl/CruiseControlMode/opMode/Switch`), so that these signals are prioritized as detection sites during the enhanced MCDC analysis. For more information, see “Configure Detection Sites using Test-pointed Logged Signals” on page 7-48.

3. Generate the code.

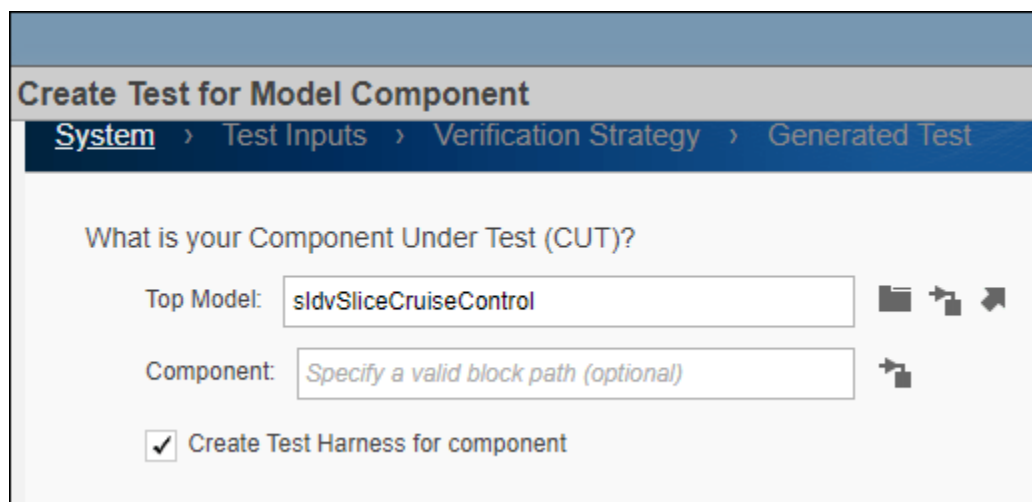
In the **Apps** tab, click **Embedded Coder**, and then click **Generate Code**.

Embedded coder generates the code generation report for model. Close the generated report window. Simulink Design Verifier uses information on logged signals from the generated code to configure the detection sites for enhanced MCDC. If you do not generate the code, Simulink Design Verifier uses the information on test-pointed logged signals from the model to configure the detection sites for enhanced MCDC.

## Section 2: Create Back-to-Back Tests Using Enhanced MCDC

Follow these steps to create back-to-back tests in the **Simulink Test** Test Manager:

1. To open the **Simulink Test** tab, in the **Apps** tab, in the **Model Verification, Validation, and Test** section, click **Simulink Test**.
2. To open the Test Manager, in the **Tests** tab, click **Simulink Test Manager**.
3. Click **New > Test for Model Component**. The Create Test for Model Component wizard opens.
4. To specify the **Top Model** to test, fill the fields by clicking the Use currently selected model component button next to the **Top Model** field.



5. Click **Next** to specify how to use the Simulink Design Verifier to generate test inputs. Select **Use Design Verifier to generate test input scenarios**. This option runs the model and creates inputs using Simulink Design Verifier.

### Create Test for Model Component

System › Test Inputs › Verification Strategy › Generated Test

How do you want to setup the inputs?

Use component input from the top model as test input  
*Create harness inputs by simulating the top model and recording the component inputs*

Use Design Verifier to generate test input scenarios  
*Create inputs using Simulink Design Verifier. [Design Verifier Settings](#)*

Specify inputs in the created harness  
*Create a new test harness for component. Inputs should be added to the harness*

6. Click **Next** to select the testing method. Select **Perform back-to-back testing**. For **Simulation1**, select Normal. For **Simulation2**, select Software-in-the-Loop (SIL). Select **Set Model coverage objectives as Enhanced MCDC**.

### Create Test for Model Component

System › Test Inputs › Verification Strategy › Generated Test

How do you want to test the component?

Use component under test output as baseline  
*Simulate the top model and record the outputs of the component to be used as baseline*

Perform back-to-back testing  
*Set up a test to compare the component under test outputs in different simulation modes*

Select simulation modes:

Simulation1:  ▼

Simulation2:  ▼

Set Model coverage objectives as Enhanced MCDC

Define the verification logic in the created harness  
*No verification logic will be automatically added to the test*

7. Click **Next** to specify the input source, format, and where to save the test data and generated tests. For **Specify the file format**, select MAT. For **Specify location to save test data**, use the default location name.

**Create Test for Model Component**

[System](#) > [Test Inputs](#) > [Verification Strategy](#) > [Generated Test](#)

How do you want to save the test data?

Select test harness input source:  Specify the file format:

Specify location to save test data:

Where do you want to save the generated test(s)?

Add tests to the currently selected test file.

Create a new test file containing the test(s).

Test File Location:

8. Click **Done**. **Simulink Test** creates the test cases and closes the wizard.

### Section 3: Run Back-to-Back Tests

To run the back-to-back test, click **Run** in Simulink Test Manager.

#### Clean Up

To complete the example, close the model.

```
bdclose(model);
```

#### Related Topics

- “Create Back-to-Back Tests Using Enhanced MCDC” on page 16-20
- “Generate Tests and Test Harnesses for a Model or Components” (Simulink Test)





# Achieving Test Cases for Missing Model Coverage

---

- “Generate Test Cases for Missing Coverage Data” on page 9-2
- “Achieve Missing Coverage in Referenced Model” on page 9-3
- “Achieve Missing Coverage in Subsystems and Model Blocks” on page 9-10
- “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11
- “Analyze Coverage for Lookup Table Boundary Values” on page 9-14
- “Modified Condition and Decision Coverage in Simulink Design Verifier” on page 9-21
- “Achieve Coverage in Models with Variable-Size Inputs” on page 9-24

## **Generate Test Cases for Missing Coverage Data**

If you simulate your model and record coverage data, but your model does not achieve 100% coverage, Simulink Design Verifier can find test cases that achieve the missing coverage. The software targets the test-generation analysis for the part of the model that is missing coverage, ignoring the model coverage data that was recorded during simulation.

The following examples describe how to focus the test-generation analysis on a part of the model that did not achieve 100% coverage:

- “Achieve Missing Coverage in Referenced Model” on page 9-3
- “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11

## Achieve Missing Coverage in Referenced Model

If you simulate a referenced model that does not achieve full coverage, you can use Simulink Design Verifier to generate test cases that achieve full coverage. There are two approaches:

- Programmatically achieve missing coverage: Generate test cases for a referenced model with APIs for test-generation analysis.
- Incrementally increase coverage: Generate test cases for the test harness model with missing coverage analysis features.

### Programmatically Achieve Missing Coverage in Referenced Model

- “Record Coverage Data for Example Model” on page 9-3
- “Find Test Cases for the Missing Coverage” on page 9-4
- “Achieve Missing Coverage” on page 9-5
- “Verify Complete Model Coverage” on page 9-5

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

#### Record Coverage Data for Example Model

Simulate the example model. Record condition, decision, and MCDC coverage.

- 1 Open the “Component-Based Modeling with Model Reference” example model `sldemo_mdref_basic`.

```
openExample('sldemo_mdref_basic');
```

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

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

Under **Model Verification, Validation, and Test**, click **Coverage Analyzer**.

- 3 On the **Coverage** tab, click **Settings**.

- 4 On the **Coverage** pane of the Configuration Parameters dialog box, set the following options:

- Select **Enable coverage analysis**.
- Select **Referenced Models**.
- Click **Select Models**. In the Select Models for Coverage Analysis dialog box, select the check box for the referenced model `sldemo_mdref_counter`. Click **OK**.

The check box for `sldemo_mdref_counter` becomes visible, corresponding to CounterA and CounterB. Coverage is not enabled for CounterC because the reference model CounterC is in Accelerator simulation mode.

- Specify which types of coverage to record during simulation. Under **Coverage metrics**, select **MCDC**.
- 5 In the **Coverage > Results** pane of the Configuration Parameters. Set the following options:

- Select **Save last run in workspace variable** to save the collected coverage data from the most recent simulation run in a variable in the MATLAB workspace.
- Select **Generate report automatically after analysis** to specify that the simulation create a coverage report.
- In the **cvdata object name** field, enter `covdata_original` to specify a unique name for the coverage data workspace variable.

6 Click **OK**.

7 To record the coverage data, start the simulation of the `sldemo_mdref_basic` model.

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

- Decision: 25%
- Condition: 50%
- MCDC: 0%

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

8 Save the coverage data in a file on the MATLAB path:

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

Keep the model open as you continue through this example.

### Find Test Cases for the Missing Coverage

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

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

```
open_system('sldemo_mdref_counter');
```

2 Create an `sldvoptions` object:

```
opts = sldvoptions;
```

When you create the `sldvoptions` object, specify:

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

Enter the following commands to specify these options:

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

3 Analyze the referenced model, `sldemo_mdref_counter`, by using the specified options:

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

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

The next procedure simulates the referenced model, `sldemo_mdref_counter`, with the test case that the analysis created.

## Achieve Missing Coverage

To achieve the missing coverage for the referenced model, `sldemo_mdhref_counter`, simulate the model by using the test case from the Simulink Design Verifier analysis.

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

```
open_system('sldemo_mdhref_counter');
```

- 2 Create a `cvtest` object for the simulation and specify recording decision, condition, and MCDC coverage.

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

- 3 Specify recording coverage and set the name of the `cvtest` object.

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

- 4 Simulate the model with the `cvtest` object, `cvt`, and the test case, as defined in `fileNames.DataFile`. Save the recorded coverage data in the workspace variable `covdata_missing`.

```
[~, covdata_missing] = sldvruntest('sldemo_mdhref_counter', fileNames.DataFile, runOpts);
```

## Verify Complete Model Coverage

You saved the coverage data from the simulation of the top-level model, `sldemo_mdhref_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_mdhref_counter`, enter the following command:

```
cvhtml('Coverage Summary', covdata_original, covdata_missing);
```

The report shows that by analyzing the referenced model and using those results to record coverage, you can achieve 100% decision, condition, and MCDC coverage.

## Summary

| Model Hierarchy/Complexity:              | Test 1 |     |      | Test 2 |    |      | Total |     |      |      |  |    |  |      |  |      |  |      |  |
|--|--------|-----|------|--------|----|------|-------|-----|------|------|--|----|--|------|--|------|--|------|--|
|  | D1     | C1  | MCDC | D1     | C1 | MCDC | D1    | C1  | MCDC |      |  |    |  |      |  |      |  |      |  |
| 1. <a href="#">sldemo_mdhref_counter</a> | 3      | 25% |      | 50%    |    | 0%   |       | 75% |      | 100% |  | 0% |  | 100% |  | 100% |  | 100% |  |

## Increase Coverage for Referenced Models in a Test Harness

- “Generate Test Harness Model and Record Coverage Data” on page 9-6
- “Generate Test Cases for the Missing Coverage” on page 9-6
- “Update Simulink Design Verifier Analysis Options” on page 9-9
- “View Active Results for Missing Coverage Analysis” on page 9-9
- “Limitations” on page 9-9

You can incrementally achieve full coverage for a generated test harness model. This example shows how to first generate a test harness model that does not achieve full coverage. Next, it shows how to run missing coverage analysis on the test harness model to generate test cases for 100% coverage.

---

**Note** This approach supports only test harness models generated by Simulink Design Verifier that reference the input model. The **Design Verifier** app is not available for test harness models when the test unit is copied from the top model. For more information see, “Reference input model in generated harness” on page 15-60.

---

### Generate Test Harness Model and Record Coverage Data

To achieve full coverage for the `sldemo_mdhref_counter` model, run a missing coverage analysis on the Simulink Design Verifier generated harness model.

- 1 Open the example model:

```
open_system('sldemo_mdhref_counter');
```





- 2 Create a harness model for referenced model `sldemo_mdhref_counter`:

```
[savedHarnessFilePath] = sldvmakeharness('sldemo_mdhref_counter');
```

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

- 3 In the harness model `sldemo_mdhref_counter_harness`, the **Format** parameter must be **Dataset** to make the referenced model `sldemo_mdhref_counter` and the harness model `sldemo_mdhref_counter_harness` have the same parameter settings. For more information see, “Model Configuration Parameters: Data Import/Export”.
- 4 Simulate the `sldemo_mdhref_counter_harness` model to record the coverage achieved by the test cases in the harness model. After the simulation, the coverage report appears. The report indicates that the following coverage is achieved for `sldemo_mdhref_counter`:

## Summary

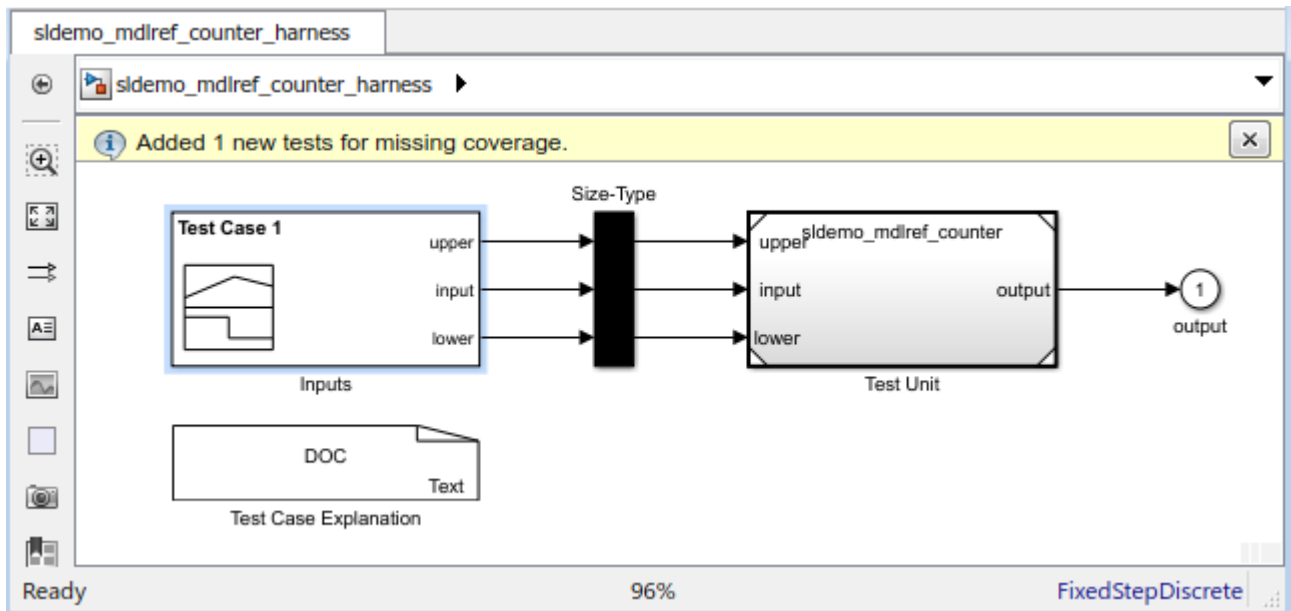
| Model Hierarchy/Complexity               | Test 1 | Decision  | Condition   | MCDC | Execution   | Relational Boundary   |
|--|--------|---|---|------|---|---|
| 1. <a href="#">sldemo_mdhref_counter</a> | 3      | 25%  | 50%  | 0%   | 86%  | 50%  |

### Generate Test Cases for the Missing Coverage

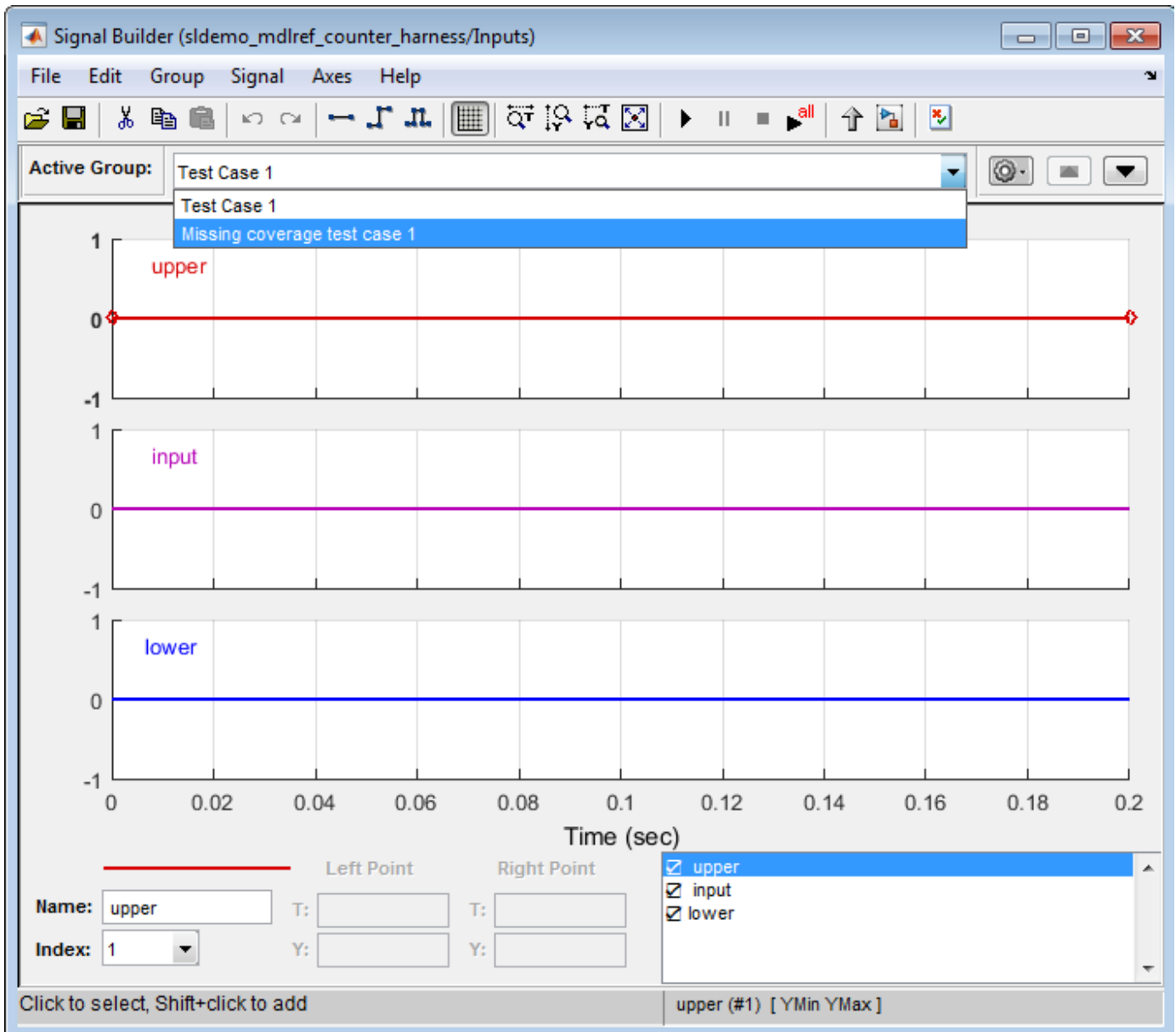
- 1 Open the harness model:

```
open_system('sldemo_mdhref_counter_harness');
```


To generate test cases for the missing coverage, on the **Design Verifier** tab, click **Add Missing Coverage**. A notification indicates the number of new tests that are added.



- 2 The Signal Builder dialog box shows the **Missing coverage test case 1** added to the previous **Test Case 1**.



3

In the Signal Builder dialog box, click **Run all** . The software simulates the harness model by using all the test cases, collects model coverage information, and displays a coverage report. The coverage report indicates that the missing coverage analysis records 100% coverage for `sldemo_mdref_counter`.



## Summary

### Model Hierarchy/Complexity Test 1

|  |   | Decision   | Condition  | MCDC   | Execution   | Relational Boundary   |
|--|---|--|--|--|---|---|
| 1. <a href="#">sldemo_mdhref_counter</a> | 3 | 100%  | 100%  | 100%  | 100%  | 50%  |

### Update Simulink Design Verifier Analysis Options

- 1 Open the harness model.

```
open_system('sldemo_mdhref_counter_harness');
```

On the **Design Verifier** tab, click **Test Generation Settings**. The Configuration Parameters dialog box for referenced model `sldemo_mdhref_counter` opens. You can set design verifier options for missing coverage analysis. For more information see, “Options in Configuration Parameters Dialog Box” on page 15-2.

### View Active Results for Missing Coverage Analysis

- 1 Open the referenced model.

```
open_system('sldemo_mdhref_counter');
```

On the **Design Verifier** tab, in the **Review Results** section, click **Load Earlier Results**. Browse to the previously generated data file and click **Open**.

To view active results for missing coverage test cases, click **Results Summary**. The Results Summary window opens with the missing coverage analysis results. For more information on active results, see “Review Analysis Results” on page 13-57. The missing coverage test cases data is stored in a MAT-file that contains a structure named `sldvData`. For more information see, “Generate sldvData Structure” on page 13-7.

### Limitations

- 1 Missing Coverage analysis is a user interface-based workflow. Command-line functions are not available for Missing Coverage analysis.
- 2 Constraining values for parameters is not supported in the Missing Coverage analysis workflow. For more information see, “Use Parameter Table” on page 5-7.

### See Also

### More About

- “Generate Test Cases for Missing Coverage Data” on page 9-2
- “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11

## Achieve Missing Coverage in Subsystems and Model Blocks

If your model has a Subsystem block that does not achieve full coverage, you can convert it to model referenced in a Model block. “Convert Subsystems to Referenced Models” describes how to convert a subsystem to a referenced model. You can then follow the steps described in “Achieve Missing Coverage in Referenced Model” on page 9-3.

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

It is possible that you have a Stateflow chart or a MATLAB Function block that does not achieve full coverage. You cannot convert Stateflow charts and MATLAB Function blocks to referenced models.

When you cannot use a Model block, follow the steps described in “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11.

### See Also

### More About

- “Achieve Missing Coverage in Referenced Model” on page 9-3
- “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11

## Achieve Missing Coverage in Closed-Loop Simulation Model

### In this section...

“Record Coverage Data for the Model” on page 9-11

“Find Test Cases for Missing Coverage” on page 9-12

If you have a subsystem or a Stateflow chart that does not achieve 100% coverage, and you do not want to convert the subsystem or chart to a Model block, follow this example to achieve full coverage.

The example uses a closed-loop controller model. A *closed-loop controller* passes instructions to the controlled system and receives information from the environment as the control instructions are executed. The controller can adapt and change its instructions as it receives this information.

The `sldvdemo_autotrans` model is a closed-loop simulation model. The ShiftLogic Stateflow chart represents the controller part of this model. Test cases designed in the ManeuversGUI Signal Builder block drive the closed-loop simulation.

### Record Coverage Data for the Model

To simulate the model, recording condition, decision, and MCDC coverage for the ShiftLogic controller:

- 1 Open the example model:  
`sldvdemo_autotrans`
- 2 On the **Apps** tab, click the arrow on the right of the **Apps** section.  
  
Under **Model Verification, Validation, and Test**, click **Coverage Analyzer**.
- 3 On the **Coverage** tab, click **Settings**.
- 4 On the **Coverage** pane in the Configuration Parameters dialog box, set the following options:
  - Select **Enable coverage analysis**.
  - Select **Subsystem** and click **Select Subsystem**.
  - In the Subsystem Selection dialog box, select ShiftLogic and click **OK**.
- 5 Under **Coverage metrics**, select Modified Condition Decision Coverage (MCDC).
- 6 Clear the **Other metrics** if they are selected.
- 7 In the **Coverage > Results** pane of the Configuration Parameters dialog box, set the following options:
  - In the **cvdata object name** field, enter `covdata_original_controller` to specify a unique name for the coverage data workspace variable.
  - Select **Generate report automatically after analysis**.
- 8 Click **OK**.
- 9 Start the simulation of the `sldvdemo_autotrans` model to record the coverage data.

After the simulation, the coverage report opens. The report indicates that the following coverage is achieved for the ShiftLogic Stateflow chart:

- Decision: 87% (27/31)
- Condition: 67% (8/12)
- MCDC: 33% (2/6) conditions reversed the outcome

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

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

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

## Find Test Cases for Missing Coverage

To find the missing coverage for the ShiftLogic chart, run a subsystem analysis on that block. Use this technique to focus your analysis on an individual part of the model.

To achieve 100% coverage for the ShiftLogic controller, run a test-generation analysis that uses the existing coverage data.

- 1 Right-click the ShiftLogic block and select **Design Verifier > Options**.
- 2 In the Configuration Parameters dialog box, under the **Select** tree, choose the **Design Verifier** node. Under **Analysis options** in the **Mode** field, select **Test generation**.
- 3 Under the **Design Verifier** node, select **Test Generation**. Under **Existing coverage data**, select **Ignore objectives satisfied in existing coverage data**.
- 4 In the **Coverage data file** field, enter the name of the file containing the coverage data that you recorded during simulation:

```
existingcov.cvt
```

- 5 Click **Apply** to save these settings.
- 6 Under the **Select** tree, click **Design Verifier**.
- 7 On the main **Design Verifier** pane, click **Generate Tests**.

The analysis extracts the Stateflow chart into a new model named `ShiftLogic0`. The analysis analyzes the new model, ignoring the coverage objectives previously satisfied and recorded in the `existingcov.cvt` file.

- 8 When the test-generation analysis is complete, in the Simulink Design Verifier log window, select **Simulate tests and produce a model coverage report**.

The report indicates that the following coverage is achieved for the ShiftLogic chart in simulation with the test cases generated by Simulink Design Verifier:

- Decision: 84% (26/31)
- Condition: 83% (10/12)
- MCDC: 67% (4/6) conditions reversed the outcome

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

The Simulink Design Verifier report indicates that two coverage objectives in the Stateflow chart ShiftLogic are proven unsatisfiable. The implicit event `tick` is never `false` because the

ShiftLogic chart is updated at every time step. The analysis cannot satisfy condition or MCDC coverage for either instance of the temporal event `after(TWAIT, tick)`.

`after(TWAIT, tick)` is semantically equivalent to

```
Event == tick && temporalCount(tick) >= TWAIT
```

If you move `after(TWAIT, tick)` into the condition, as in

```
[after(TWAIT, tick) && speed < down_th]
```

Simulink Design Verifier determines that `tick` is always true, so it only tests the `temporalCount(tick) >= TWAIT` part of `after(TWAIT, tick)`. The analysis is able to find test objectives that satisfy condition and MCDC coverage for `after(TWAIT, tick)`.

## See Also

### More About

- “Generate Test Cases for Missing Coverage Data” on page 9-2
- “Achieve Missing Coverage in Referenced Model” on page 9-3

## Analyze Coverage for Lookup Table Boundary Values

Lookup tables are standard block sets that approximate functions. Simulink Coverage defines the coverage of lookup tables by checking if all grid points defined by breakpoints are covered or satisfied during simulation. For more information, see “N-Dimensional Lookup Table” (Simulink Coverage).

You can leverage Simulink Design Verifier to generate tests that hit the boundary values of a lookup table. The minimum and maximum breakpoints for each dimension, also known as the *corner points*, define the boundaries of the lookup table. Achieving such coverage is a common use case in the aerospace and automotive domains.

Consider this two-dimensional lookup table:

| Breakpoints | Column | (1) | (2) | (3) |
|-------------|--------|-----|-----|-----|
| Row         |        | 1   | 2   | 3   |
| (1)         | 10     | 1   | 2   | 3   |
| (2)         | 20     | 4   | 5   | 6   |
| (3)         | 30     | 7   | 8   | 9   |

The breakpoints that represent corner points are:

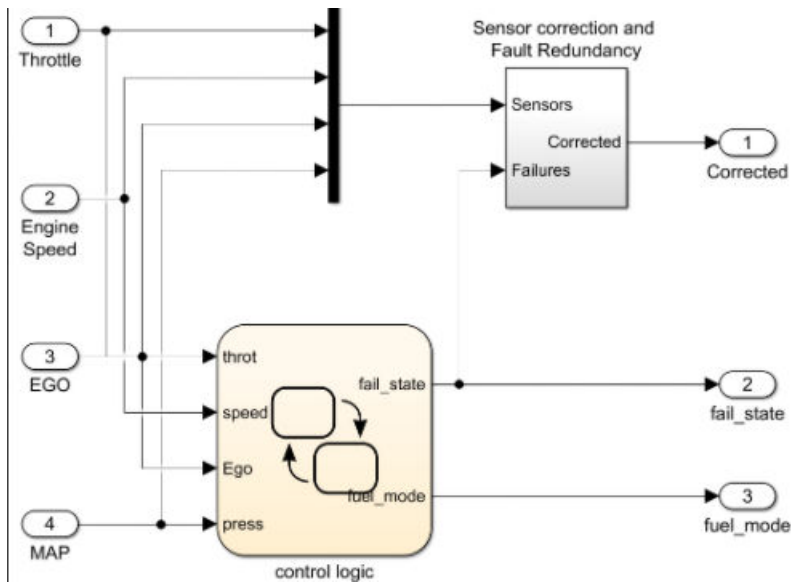
Dimension 1 : 10, 30  
Dimension 2: 1, 3

The breakpoints that represent the boundaries of this lookup table are (10, 1), (10, 3), (30, 1), and (30,3). The tests corresponding to the lookup table boundary satisfy the following coverage metrics:

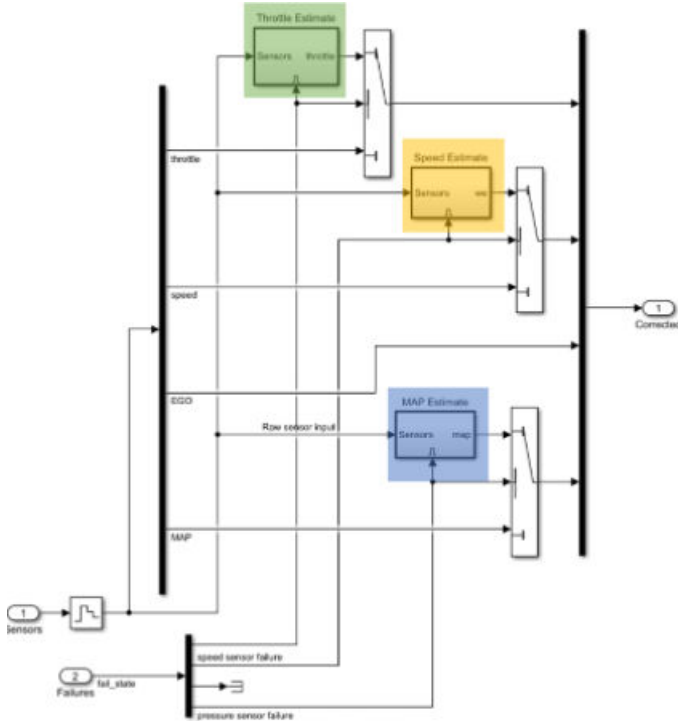
Corner 1: Input 1 < 10, Input2 < 1  
Corner 2: Input 1 < 10, Input2 > 3  
Corner 3: Input 1 > 30, Input2 < 1  
Corner 4: Input 1 > 30, Input2 > 3

This example leverages block replacement framework provided by Simulink Design Verifier to generate tests.

Consider the controller unit (`control logic`) and the fault approximation unit (`Sensor correction and Fault Redundancy`) as shown:



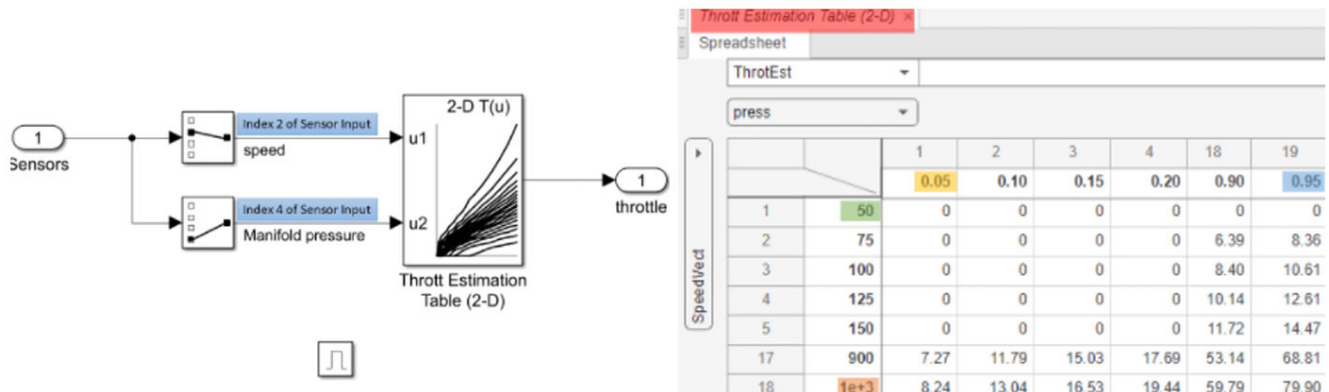
- control logic: The control logic statechart checks for the normal and failure modes of throttle, speed, and manifold absolute pressure (MAP).
- Sensor correction and Fault Redundancy: If there is a failure in the throttle, speed, and MAP values, the subsystem Sensor correction and Fault Redundancy approximates their values by using these tables:



- Throttle Estimate: The lookup table Thrott Estimation Table (2-D) estimates the throttle position based on the values of speed and pressure.

- **Speed Estimate:** The lookup table Speed Table (2-D) estimates the speed based on the estimated value of throttle position and pressure.
- **MAP Estimate:** The lookup table Pressure Estimate (2-D) estimates the MAP based on the estimated value of speed and throttle.

Simulink Design Verifier generates boundary value coverage tests for each of these lookup tables. To describe the results, consider lookup table defined for Throttle Estimate as shown:



- Throttle estimation lookup table uses speed (Sensor index '2') and manifold pressure (Sensor index '4') sensor values as inputs.
- The lookup table has 19 breakpoints for pressure with 0.05 . . . . 0.95 as corner points, and 18 breakpoints for speed with 50 . . . . 1e3 as corner points.

The corner points for this lookup table are:

Breakpoints 1 (speedVect): 50,1000  
 Breakpoints 2 (press): 0.05,0.95

The generated tests attempt to hit the boundary points highlighted in the table. The tests require these breakpoint combinations to cover the boundary values.

---

**Note** The Engine Speed and MAP inputs are inputs to the breakpoints speedVect and press, respectively.

---

- Corner 1: Engine Speed < 50, MAP < 0.05
- Corner 2: Engine Speed < 50, MAP > 0.95
- Corner 3: Engine Speed > 1000, MAP < 0.05
- Corner 4: Engine Speed > 1000, MAP > 0.95

The replacement rule, InstrumentLUTForCornerValueCoverage is shipped with the example mentioned here.

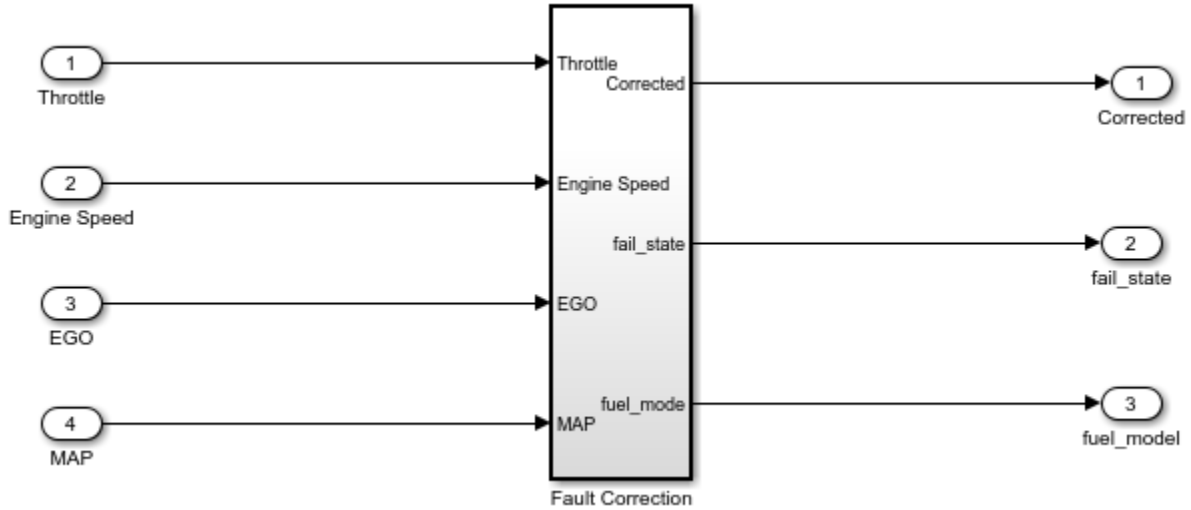
## Generate Tests for Lookup Table Boundary Values

This example shows how to generate tests for lookup table boundary value coverage.



## Open the Model

```
open_system('sldvdemo_fuelsys_lookup_corner_value_coverage');
```



## Specify the Analysis Options

To specify the analysis options:

1. In the **Apps** tab, click **Coverage Analyzer**.
2. In the **Coverage** tab, click **Settings** to open the **Configuration Parameters** window.
3. Expand **Other metrics**, then select the **Lookup table** option. Click **OK**.
4. In the left pane, click **Block Replacements**. Select **Apply block replacements**.
5. In **List of block replacement rules (in order of priority)**, specify the rule as **InstrumentLUTForCornerValueCoverage**. Click **OK**.
6. In the **Test Generation** pane, set **Model Coverage Objectives** to **None**.

Note: Because the test generation settings, **Test conditions** and **Test Objectives** are enabled, Simulink Design Verifier generates tests for the custom test objectives defined in the block replacement rule.

## Perform Analysis

To generate tests only for the custom test objectives for the lookup table:

1. In the **Design Verifier** tab, click **Generate Tests**.
2. To view the coverage results, click the **Simulate Tests and produce a model coverage report** link.

## Review Results

After simulating the test cases and generating a coverage report, the top left and right corners are covered while the bottom left and right corners are uncovered.

### 6. SubSystem block "[Throttle Estimate](#)"

[Justify or Exclude](#)


Parent: [sldvdemo\\_fuelsys\\_lookup\\_corner\\_value\\_coverage/Fault Correction/Sensor correction and Fault Redundancy](#)

| Metric                | Coverage (this object) | Coverage (inc. descendants)                      |
|-----------------------|------------------------|--|
| Cyclomatic Complexity | 2                      | 2  |
| Lookup Table          | NA                     | 2% (7/380) interpolation/extrapolation intervals |
| Execution             | NA                     | 100% (1/1) objective outcomes                    |

Lookup\_n-D block "[Thrott Estimation Table \(2-D\)](#)"

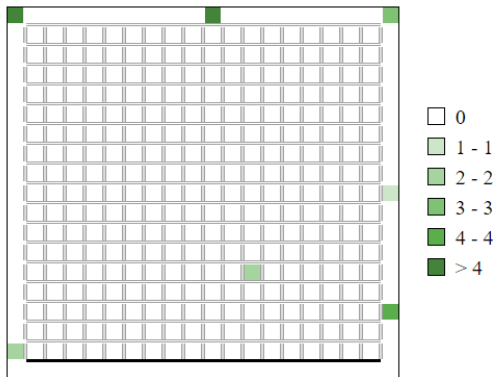
[Justify or Exclude](#)

Parent: [sldvdemo\\_fuelsys\\_lookup\\_corner\\_value\\_coverage/Fault Correction/Sensor correction and Fault Redundancy/Throttle Estimate](#)

Uncovered Links: 

| Metric                | Coverage   |
|-----------------------|--|
| Cyclomatic Complexity | 0  |
| Lookup Table          | 2% (7/380) interpolation/extrapolation intervals |
| Execution             | 100% (1/1) objective outcomes                    |

#### Look-up Table Details



The status of the test objectives associated with the 'Thrott Estimation Table (2-D)' Lookup table:

**Status:** Objectives Satisfied

**Summary**

Length: 0.24 second (25 sample periods)  
 Objectives Satisfied: 4

**Objectives**

| Step | Time | Model Item  | Objectives  |
|------|------|---|---|
| 4    | 0.03 | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/Throttle Estimate/Thrott Estimation Table (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a> | 9. sldv.test(u < BreakpointsForDimension1(1) && v < BreakpointsForDimension2(1))    |
| 11   | 0.1  | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/Speed Estimate/Speed Table (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a>                | 6. sldv.test(u > BreakpointsForDimension1(end) && v < BreakpointsForDimension2(1))  |
| 22   | 0.21 | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/Throttle Estimate/Thrott Estimation Table (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a> | 11. sldv.test(u < BreakpointsForDimension1(1) && v > BreakpointsForDimension2(end)) |
| 25   | 0.24 | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/MAP Estimate/Pressure Estimate (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a>            | 3. sldv.test(u < BreakpointsForDimension1(1) && v > BreakpointsForDimension2(end))  |

**Generated Input Data**

| Time         | 0-0.02   | 0.03-0.04 | 0.05 | 0.06-0.07 | 0.08-0.09 | 0.1-0.11 | 0.12-0.13 | 0.14-0.15 | 0.16-0.18 | 0.19-0.2 | 0.21-0.22 | 0.23     | 0.24 |
|--------------|----------|-----------|------|-----------|-----------|----------|-----------|-----------|-----------|----------|-----------|----------|------|
| Step         | 1-3      | 4-5       | 6    | 7-8       | 9-10      | 11-12    | 13-14     | 15-16     | 17-19     | 20-21    | 22-23     | 24       | 25   |
| Throttle     | 102.5913 | 0         | 0    | 4         | 0         | 120      | 46.9065   | 3         | 3         | 114.8857 | 0         | 1.052    | 120  |
| Engine Speed | 758.0162 | 0         | 1000 | 0         | 0         | 0        | 171.8017  | 1000      | 1         | 534.8037 | 0.05      | 302.5962 | 1    |
| EGO          | 7.4011   | 0         | 0    | 0         | 0         | 0        | 6.866     | 0         | 0         | 10.6192  | 0         | 0.91743  | 0    |
| MAP          | 1.0093   | 0         | 0    | 0         | 0.5005    | 0.001    | 0.37509   | 0.001     | 0.001     | 0.61965  | 1         | 1.3732   | 0    |

From the above figure:

**LUT Boundary:** Engine Speed < 50, MAP < 0.05

**Corner:** Top Left

**Analysis:** This objectives is satisfied at time 0.03 (Step 4).

**LUT Boundary:** Engine Speed < 50, MAP > 0.95

**Corner:** Top Right

**Analysis:** This objectives is satisfied at time 0.21 (Step 22).

**Status:** Objectives Unsatisfiable

**3.2. Objectives Unsatisfiable**

Simulink Design Verifier proved these test objectives to be unreachable by any test case. This often indicates the presence of dead logic in the model. This can be a side effect of parameter configurations or minimum and maximum constraints specified on inputs. In Test Generation, this can also be a result of constraints resulting from Test Condition blocks.

| #  | Type           | Model Item  | Description   | Analysis Time (sec) |
|----|----------------|---|---|---------------------|
| 1  | Test objective | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/MAP Estimate/Pressure Estimate (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a>            | sldv.test(u < BreakpointsForDimension1(1) && v < BreakpointsForDimension2(1))     | 17                  |
| 2  | Test objective | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/MAP Estimate/Pressure Estimate (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a>            | sldv.test(u > BreakpointsForDimension1(end) && v < BreakpointsForDimension2(1))   | 17                  |
| 4  | Test objective | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/MAP Estimate/Pressure Estimate (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a>            | sldv.test(u > BreakpointsForDimension1(end) && v > BreakpointsForDimension2(end)) | 17                  |
| 5  | Test objective | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/Speed Estimate/Speed Table (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a>                | sldv.test(u < BreakpointsForDimension1(1) && v < BreakpointsForDimension2(1))     | 17                  |
| 7  | Test objective | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/Speed Estimate/Speed Table (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a>                | sldv.test(u < BreakpointsForDimension1(1) && v > BreakpointsForDimension2(end))   | 17                  |
| 10 | Test objective | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/Throttle Estimate/Thrott Estimation Table (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a> | sldv.test(u > BreakpointsForDimension1(end) && v < BreakpointsForDimension2(1))   | 17                  |
| 12 | Test objective | <a href="#">Fault Correction/Sensor correction and Fault Redundancy/Throttle Estimate/Thrott Estimation Table (2-D)/MATLAB Function. Defined by block replacement rule 'InstrumentLUTForCornerValueCoverage'.</a> | sldv.test(u > BreakpointsForDimension1(end) && v > BreakpointsForDimension2(end)) | 17                  |

From the above figure:

**LUT Boundary:** Engine Speed > 1000, MAP < 0.05

**Corner:** Bottom Left

**Analysis:** In the **Control logic** statechart, the input speed has the following values:

Minimum speed: 0 Maximum Speed: 1000

The statechart input speed is directly connected to the root inport Engine Speed. Hence, the same range constraints are applied to it. Due to these min/max constraints, the value of **Engine Speed** can never be set to a value >1000, hence this test objective becomes unsatisfiable.

**LUT Boundary:** Engine Speed > 1000, MAP > 0.95

**Corner:** Bottom Right

**Analysis:** The value of 'Engine Speed' can never be set to a value >1000, hence this test objective becomes unsatisfiable.

### Clean Up

To complete the example, close the model.

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

# Modified Condition and Decision Coverage in Simulink Design Verifier

Depending on the settings you apply for Simulink Coverage coverage recording, there can be a difference between the definition of modified condition and decision (MCDC) coverage used for model coverage analysis in Simulink Coverage and the definition used for test case generation analysis in Simulink Design Verifier.

## MCDC Definitions for Simulink Coverage and Simulink Design Verifier

Simulink Design Verifier and Simulink Coverage represent MCDC objectives in two different ways:

- Simulink Coverage treats each condition of a logical expression as an MCDC objective.
- Simulink Design Verifier treats the true and false halves of each independence pair as separate MCDC objectives.

The Simulink Design Verifier Results window shows **Justified** for any justified MCDC objectives. Click on the corresponding **View** link to see the filter rule in the Simulink Design Verifier Analysis Filter window.

Unsatisfiable or undecided MCDC objectives include a **Justify** link. Click on this link to create a corresponding filter rule. Because every MCDC objective in Simulink Coverage corresponds to two MCDC objectives in Simulink Design Verifier, the Simulink Design Verifier MCDC objectives are justified in pairs.

For example, in the image below, when you click on the **Justify** link for the MCDC expression **expression for output with input port 4 false**, creates a filter rule that justifies this MCDC objective as well as the MCDC objective for when that expression is **true**.

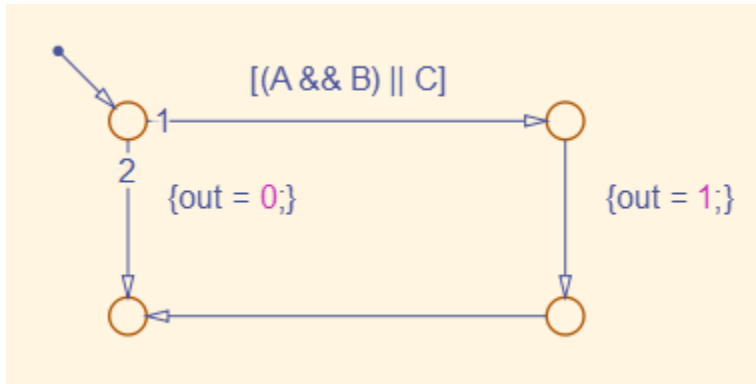
The left screenshot shows the 'Results: mMCDC\_covfilt\_tg' window. It displays a list of MCDC objectives under the heading 'mMCDC\_covfilt\_tg/AND\_block'. The objectives are categorized into 'Condition Objectives' and 'MCDC Objectives'. The 'Condition Objectives' list includes logic for input ports 1 through 4, with 'true' conditions marked as 'Satisfied' and 'false' conditions marked as 'Unsatisfiable'. The 'MCDC Objectives' list includes expressions for output with input ports 1 through 4, with 'true' conditions marked as 'Satisfied' and 'false' conditions marked as 'Unsatisfiable'. Links for 'View test case' and 'Justify' are provided for each objective.

The right screenshot shows the 'Analysis Filter: myModel' window. It displays a table of filter rules. The table has four columns: 'Name', 'Type', 'Mode', and 'Rationale'. The first rule is 'input port 2 outcome of ... by MCDC outcome' with 'Justified' mode and 'some rationale'. The second rule is 'input port 3 outcome of ... by MCDC outcome' with 'Justified' mode and 'another rationale'. There are 'Remove rule' and 'View in model' buttons to the right of the table. Below the table, the 'Selected rule' is 'input port 2 outcome of expression for output in Logic block "AND\_block1"'. At the bottom, there are fields for 'Filename: .../mcdcFilter/mcdcFilter.cvf' and buttons for 'Save filter', 'View', and 'Load filter'.

Simulink Design Verifier always uses the masking MCDC definition for test case generation. By default, Simulink Coverage also uses the masking MCDC definition when recording coverage. However, if you set the `CovMcdcMode` model configuration parameter to 'UniqueCause', Simulink Coverage instead uses the unique-cause MCDC definition when recording coverage. For information

on the differences between the masking MCDC definition and the unique-cause MCDC definition, see “Modified Condition and Decision Coverage (MCDC) Definitions in Simulink Coverage” (Simulink Coverage).

Setting the `CovMcdcMode` model configuration parameter to 'UniqueCause' can result in differences between MCDC reporting in Simulink Coverage and test generation in Simulink Design Verifier. An example of this difference can be seen in analysis results for logical expressions containing a mixture of AND and OR operators, as in this Stateflow transition.



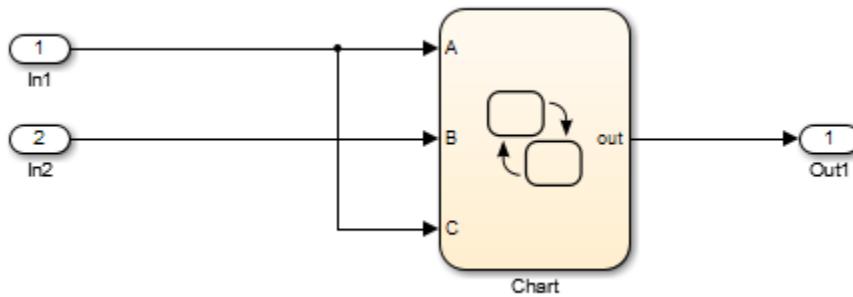
Given that A, B, and C are each separate inputs, there are five possible ways to evaluate the condition on the Stateflow transition, shown in the following table.

|   | A | B | C | (A && B)    C |
|---|---|---|---|---------------|
| 1 | F | x | F | F             |
| 2 | F | x | T | T             |
| 3 | T | F | F | F             |
| 4 | T | F | T | T             |
| 5 | T | T | x | T             |

Satisfying MCDC for a Boolean variable requires a pair of condition evaluations, showing that a change in that variable alone changes the evaluation of the entire expression. In this example, MCDC can be satisfied for C with either the pair 1, 2 or the pair 3, 4. In both of those cases, the value of the expression changed because the value of C changed, while all other variable values stayed the same.

Each pair has a different set of values for A and B which are held constant, but each pair contains one evaluation where C and out are true and one evaluation where C and out are false. To satisfy MCDC for C, Simulink Design Verifier test generation analysis accepts any pair containing one evaluation of true values and one evaluation of false values for C and out. In this example, Simulink Design Verifier test generation analysis accepts not only pair 1, 2 and pair 3, 4 but also pair 1, 4 and pair 2, 3. Simulink Coverage model coverage analysis using the unique-cause MCDC definition is satisfied only by pair 1, 2 or by pair 3, 4.

The preceding example assumes that A, B, and C are all separate inputs. When input A is constrained to be the same value as C, as in this model, only a subset of condition evaluations are possible.



This subset of condition evaluations for the Stateflow transition is shown in the following table.

|   | A | B | C | (A && B)    C |
|---|---|---|---|---------------|
| 1 | F | x | F | F             |
| 4 | T | F | T | T             |
| 5 | T | T | x | T             |

Evaluations 2 and 3 are no longer possible, so neither pair 1, 2 nor pair 3, 4 is possible. As a result, unique-cause MCDC for C can no longer be satisfied in Simulink Coverage model coverage analysis. Since pair 1, 4 is still possible, however, Simulink Design Verifier test generation analysis reports that MCDC for C is satisfiable.

The complexity of MCDC analysis for logical expressions with a mixture of AND and OR operators causes this difference between results from Simulink Coverage set to unique-cause MCDC analysis and Simulink Design Verifier. The default `CovMcdcMode` model configuration parameter value of 'Masking' does not cause this discrepancy. However, if you require the use of unique-cause MCDC analysis in Simulink Coverage, you can minimize this effect by using the `IndividualObjectives` test suite optimization for test generation analysis in Simulink Design Verifier. For more information, see the Tip section of "Test suite optimization" on page 15-34.

## See Also

### More About

- "MCDC" on page 7-31

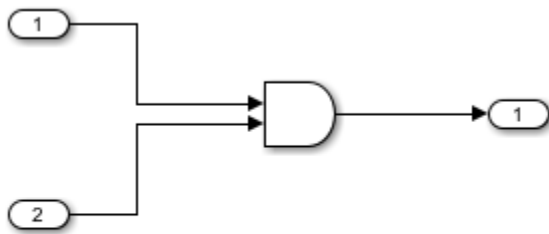
## Achieve Coverage in Models with Variable-Size Inputs

This example shows you how to achieve model coverage in models with variable-size input signals by using Simulink Design Verifier™.

### Open the Model

The model in this example has two input ports that pass variable-size signals. Input port 1 (**in1**) of the model is of variable size signal with maximum dimension [3,3].

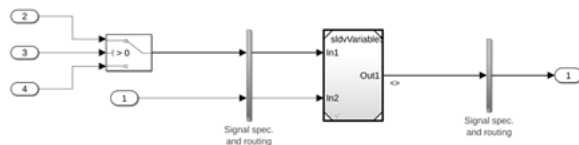
```
open_system('sldvVariableSizeAtInport');
```



Copyright 2023 The MathWorks, Inc.

### Create and Detach Harness Model

1. Open **Simulink Test** in the **Apps** pane.
2. Click **Add Test Harness** in the **Create Test Harness** section.
3. Click **OK** in the **Create Test Harness** dialog box.
4. Click **Detach And Export** in the **Harness** tab.

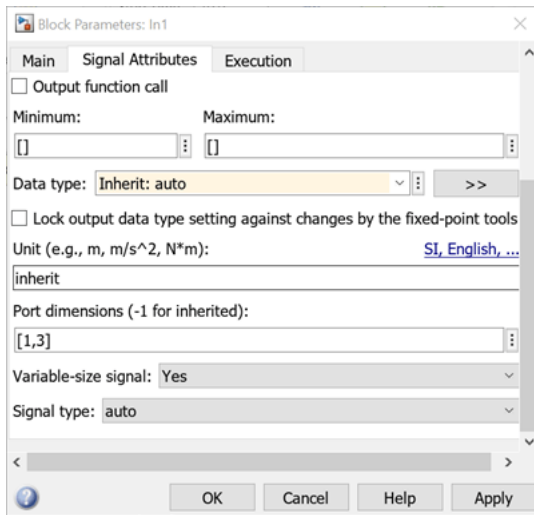


### Drive Variable-size Input Port with Fixed Dimension Signals

Simulink Design Verifier™ supports models with fixed-size signals at the input ports. Enhance the harness model such that it has fixed-size input ports but produces variable-size signals at the design interface.

1. Add a **Switch** block to the harness model and in the **Block Parameters** dialog box on the **Signal Attributes** tab, select **Allow different data input sizes** checkbox.
2. Set the dimensions of the two data ports of the Switch block to [3,3] and [2,2] to ensure that you have a variable-dimension signal **in1** of the design model.








### Generate Tests to Achieve Coverage

1. Open **Design Verifier** in the **Apps** pane of the updated harness.
2. Click **Generate Tests** in the **Analyze** section.
3. Click **Simulate tests** and generate the model coverage report in the Results summary window.

### Summary

#### Model Hierarchy/Complexity Test 1

|   | Decision | Condition  | MCDC   | Test Objective | Proof Objective | Test Condition | Proof Assumption | Execution  |
|---|----------|--|--|----------------|-----------------|----------------|------------------|--|
| 1. <a href="#">sldvVariableSizeAtInport</a> | NA       | 100%  | 100%  | NA             | NA              | NA             | NA               | 100%  |



# Verifying Model Components

---

- “What Is Component Verification?” on page 10-2
- “Functions for Component Verification” on page 10-3
- “Verify a Component for Code Generation” on page 10-4

## What Is Component Verification?

**In this section...**

“Component Verification Approaches” on page 10-2

“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.

### Simulink Design Verifier Tools for Component Verification

By isolating the component to verify, and using tools that Simulink Design Verifier provides, you create test cases that let you expand the scope of the testing for large models. This expanded testing helps you accomplish the following:

- **Achieve 100% model coverage** — If certain model components do not record 100% coverage, the top-level model cannot achieve 100% coverage. By verifying these components individually, you can create test cases that fully specify the component interface, allowing the component to record 100% coverage.
- **Debug the component** — To verify that each model component satisfies the specified design requirements, you can create test cases that verify that specific components perform as designed.
- **Test the robustness of the component** — To verify that a component handles unexpected inputs and calculations properly, you can create test cases that generate data. Then, test the error-handling capabilities in the component.

## Functions for Component Verification

The Simulink Design Verifier software provides several functions that facilitate the tasks associated with component verification.

| Function                      | Task  |
|-------------------------------|---|
| <code>sldvlogsignals</code>   | 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.   |
| <code>sldvmakeharness</code>  | Create a harness model for a component, using logged input signals if specified, or using the default signals.<br><br>For more information about harness models, see “Manage Simulink Design Verifier Harness Models” on page 13-13.  |
| <code>sldvmergeharness</code> | Merge test cases from several harness models into a single harness model.   |
| <code>sldvextract</code>      | Extract an atomic subsystem or atomic subchart into a new model.  |
| <code>sldvruntime</code>      | Simulate a model, executing the specified test cases to record model coverage and output values.  |
| <code>sldvruncgvtest</code>   | Invoke the Code Generation Verification (CGV) API, and execute the specified test cases on the generated code for the model.<br><br><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” (Embedded Coder). |

Component verification functions do not support the following Simulink features:

- Variable-step solvers for `sldvruntime`
- Component interfaces that contain:
  - Variable-size signals
  - Multiword fixed-point data types larger than 128 bits

## Verify a Component for Code Generation

| In this section...   |
|--|
| “About the Example Model” on page 10-4   |
| “Prepare the Component for Verification” on page 10-6                              |
| “Record Coverage for the Component” on page 10-7                                   |
| “Use Simulink Design Verifier Software to Record Additional Coverage” on page 10-7 |
| “Combine the Harness Models” on page 10-8  |
| “Execute the Component in Simulation Mode” on page 10-9                            |
| “Execute the Component in Software-in-the-Loop (SIL) Mode” on page 10-10           |

### About the Example Model

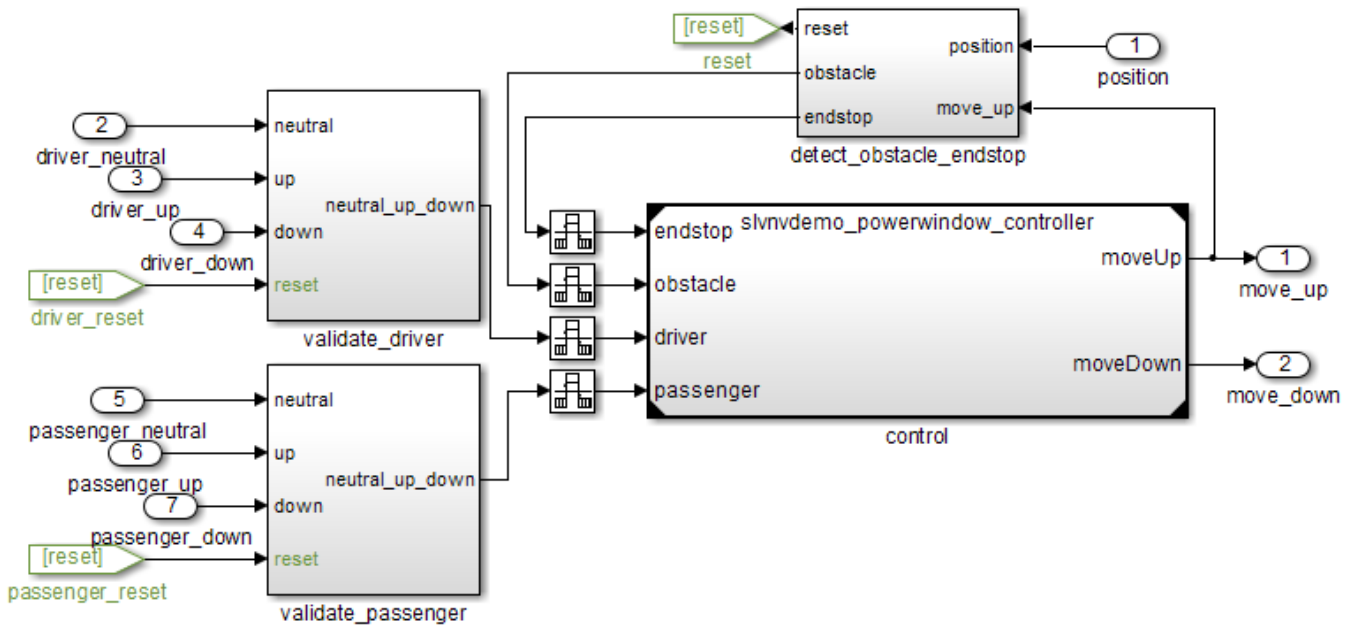
This example uses the `slvnvdemo_powerwindow` model to show how to verify a component in the context of the model that contains that component. As you work through this example, you use the Simulink Design Verifier component verification functions to create test cases and measure coverage for a referenced model. In addition, you can execute the referenced model in both simulation mode and Software-in-the-Loop (SIL) mode using the Code Generation Verification (CGV) API.

---

**Note** You must have the following product licenses to run this example:

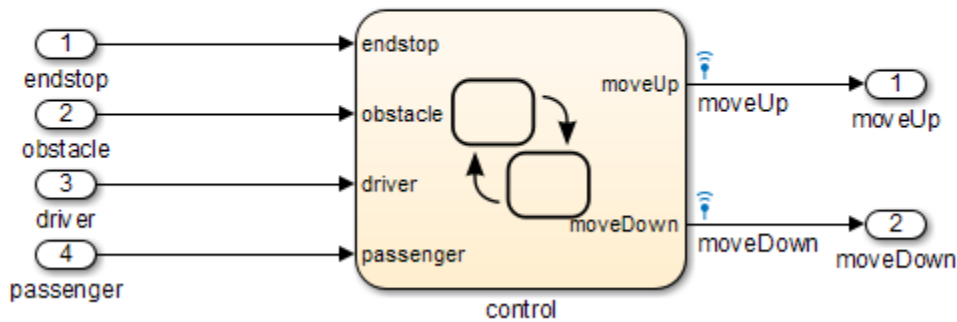
- Stateflow
  - Embedded Coder
  - Simulink Coder
- 

The component that you verify is a Model block named `control`. This component resides inside the `power_window_control_system` subsystem in the top level of the `slvnvdemo_powerwindow` model. The `power_window_control_system` subsystem is shown below.

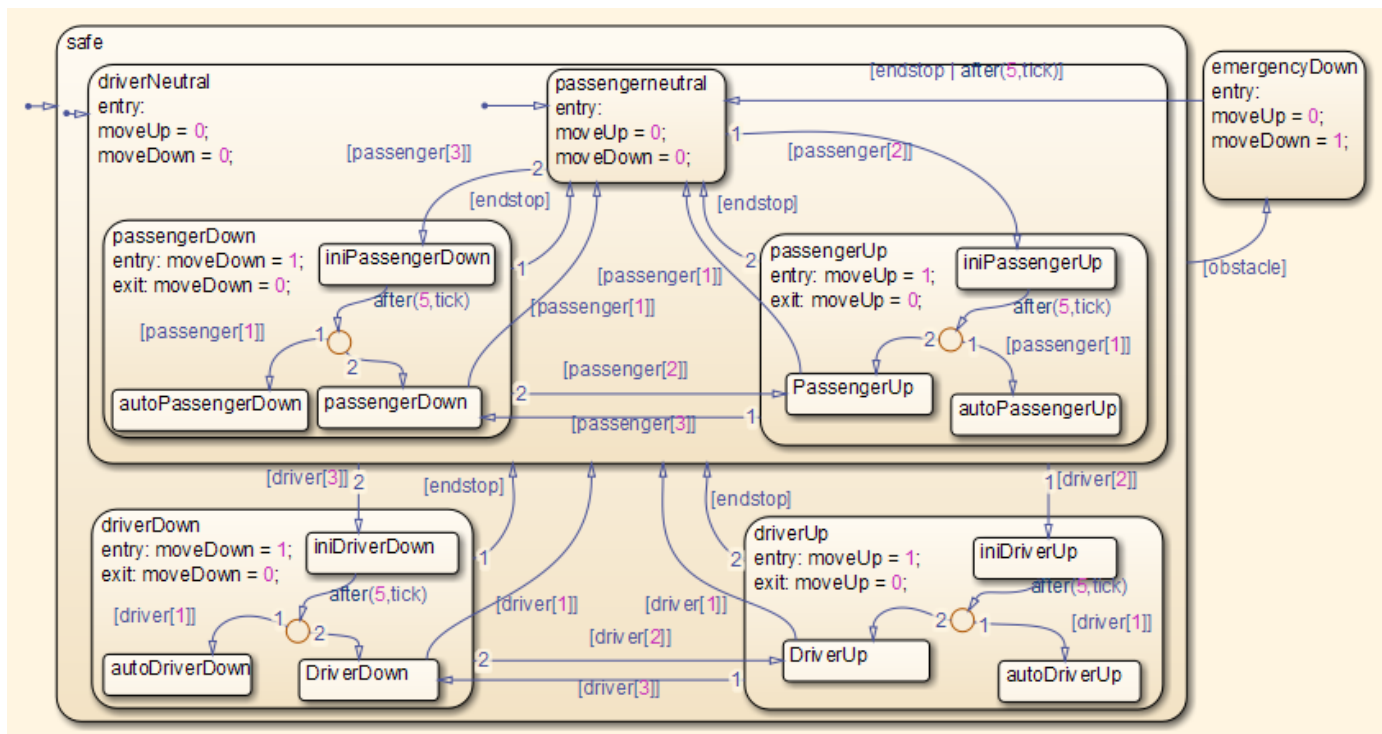


The control Model block references the slvndemo\_powerwindow\_controller model.

### Simulink Coverage Power Window Controller



The referenced model contains a Stateflow chart control, which implements the logic for the power window controller.



## Prepare the Component for Verification

To verify the referenced model `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` example model and the referenced model:

```
open_system('slvndemo_powerwindow');
open_system('slvndemo_powerwindow_controller');
```

- 2 Open the `power_window_control_system` subsystem in the example model.

The Model block named `control` in the `power_window_control_system` subsystem references the component that you verify during this example, `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.



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

- 5 For use later in this example, save the name of the harness model:

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

- 6 Leave all windows open for the next part of this example.

Next, you will record coverage for the `slvndemo_powerwindow_controller` model.

## Record Coverage for the Component

Model coverage is a measure of how thoroughly a test case tests a model, and the percentage of pathways that a test case exercises. To record coverage for the `slvndemo_powerwindow_controller` model:

- 1 Create a default options object, required by the `sldvrntest` function:

```
runOpts = sldvrntestopts;
```

- 2 Specify to simulate the model, and record coverage:

```
runOpts.coverageEnabled = true;
```

- 3 Simulate the referenced model and record coverage:

```
[~, covDataFromLoggedSignals] = sldvrntest( ...
    '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%

For more information about decision coverage, condition coverage, and MCDC coverage, see “Types of Model Coverage” (Simulink Coverage).

Because you did not achieve 100% coverage for the `slvndemo_powerwindow_controller` model, next, you will analyze the model to record additional coverage and create additional test cases.

## Use Simulink Design Verifier Software to Record Additional Coverage

You can use Simulink Design Verifier to analyze the `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 generate test cases to record decision, condition, and modified condition/decision coverage:

```
opts.ModelCoverageObjectives = 'MCDC';
```

- 4 Specify that the analysis ignore objectives that you satisfied when you logged the signals to the Model block:

```
opts.IgnoreCovSatisfied = 'on';
```

- 5 Specify the name of the file that contains the satisfied objectives data:

```
opts.CoverageDataFile = 'existingCovFromLoggedSignal.cvt';
```

- 6 Specify that the analysis 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 “Prepare the Component for Verification” on page 10-6 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 Simulink Design Verifier:

```
[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 74 test objectives.

You can combine this test case with the test case that you created in “Prepare the Component for Verification” on page 10-6, to record additional coverage for the `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.

## Combine 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 `sldvrntest`:

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

- 4 Simulate the model and record and display the coverage data:

```
[~, covDataFromMergedSignals] = sldvrntest( ...  
    '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%

## Execute the Component in Simulation Mode

To verify that the generated code for the model produces the same results as simulating the model, use the Code Generation Verification (CGV) API methods.

---

**Note** To execute a model in different modes of execution, use the CGV API to verify the numerical equivalence of results. For more information about the CGV API, see “Programmatic Code Generation Verification” (Embedded Coder).

---

When you perform this procedure, the simulation compiles and executes the model code using both test cases.

- 1 Create a default options object for `sldvruncgvtest`:

```
runcgvopts = sldvrntestopts('cgv');
```

- 2 Specify to execute the model in simulation mode:

```
runcgvopts.cgvConn = 'sim';
```

- 3 Execute the `slvnx_powerwindow_controller` model using the two test cases and the `runcgvopts` object:

```
cgvSim = sldvruncgvtest('slvnxdemo_powerwindow_controller', ...  
    loggedSignalsMergedHarness, runcgvopts);
```

These steps save the results in the workspace variable `cgvSim`.

Next, you will execute the same model with the same test cases in Software-in-the-Loop (SIL) mode and compare the results from both simulations.

For more information about Normal simulation mode, see “Execute the Model” (Embedded Coder).

## Execute the Component in Software-in-the-Loop (SIL) Mode

When you execute a model in Software-in-the-Loop (SIL) mode, the simulation compiles and executes the generated code on your host computer.

In this section, you execute the `slvnxdemo_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 `slvnx_powerwindow_controller` model using the two test cases and the `runcgvopts` object:

```
cgvSil = sldvruncgvtest('slvnxdemo_powerwindow_controller', ...  
    loggedSignalsMergedHarness, runcgvopts);
```

The workspace variable `cgvSil` contains the results of the SIL mode execution.

- 3 Compare the results in `cgvSil` to the results in `cgvSim`, created from the simulation mode execution. Use the `compare` (Embedded Coder) method to compare the results from the two simulations:

```
for i=1:length(loggedSignalsMergedHarness.TestCases)  
    simout = cgvSim.getOutputData(i);  
    silout = cgvSil.getOutputData(i);  
    [matchNames, ~, mismatchNames, ~] = ...  
        cgv.CGV.compare(simout, silout);  
end
```

- 4 Display the results of the comparison in the MATLAB Command Window:

```
fprintf(['\nTest Case(%d):%d Signals match, %d Signals mismatch\r'],...  
    i, length(matchNames), length(mismatchNames));
```

As expected, the results of the two simulations match.

For more information about Software-in-the-Loop (SIL) simulations, see “What Are SIL and PIL Simulations?” (Embedded Coder).

# Considering Specified Minimum and Maximum Values for Inputs During Analysis

---

- “Minimum and Maximum Input Constraints” on page 11-2
- “Specify Input Ranges on Simulink and Stateflow Elements” on page 11-4
- “Specification of Input Ranges in sldvData Fields” on page 11-10

## Minimum and Maximum Input Constraints

|                           |
|---------------------------|
| <b>In this section...</b> |
|---------------------------|

|  |
|--|
| “Simulink Design Verifier Support for Specified Input Minimum and Maximum Values” on page 11-2 |
|--|

|   |
|---|
| “Limitations of Simulink Design Verifier Support for Specified Minimum and Maximum Values” on page 11-2 |
|---|

When creating a model, you can specify minimum and maximum values on input ports to mimic environmental constraints as part of your design. The Simulink Design Verifier analysis can automatically consider these values as constraints for:

- Design error detection
- Test case generation
- Property proving

Specifying minimum and maximum input values is similar to using the Test Condition block to constrain signals for test case generation or the Proof Assumption block to constrain signals for property proving. The Test Condition and Proof Assumption blocks capture the analysis constraints. The Simulink Design Verifier software can also consider the design constraints captured in the Inport block minimum and maximum parameters as constraints for analysis.

---

**Note** For more information about signal values, see “Investigate Signal Values”.

---

### Simulink Design Verifier Support for Specified Input Minimum and Maximum Values

By default, Simulink Design Verifier considers any minimum and maximum input values specified for Inport blocks in your model. To enable this capability:

- 1 On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
- 2 In the Configuration Parameters dialog box, on the **Design Verifier** pane, select the **Use specified input minimum and maximum values** parameter.
- 3 After the analysis completes, to view the design minimum and maximum constraints for your model, click **Generate detailed analysis reports**.

The constraints are listed in the **Analysis Information** chapter of the Simulink Design Verifier report.

### Limitations of Simulink Design Verifier Support for Specified Minimum and Maximum Values

Simulink Design Verifier support for specified minimum and maximum values has the following limitations:

- The analysis considers specified minimum and maximum values on root-level Inport blocks only. The analysis ignores minimum and maximum values specified on other Simulink blocks.

## **See Also**

### **More About**

- “Specify Signal Ranges”

## Specify Input Ranges on Simulink and Stateflow Elements

When you specify input range constraints on Simulink and Stateflow elements, Simulink Design Verifier considers these constraints during analysis.

### In this section...

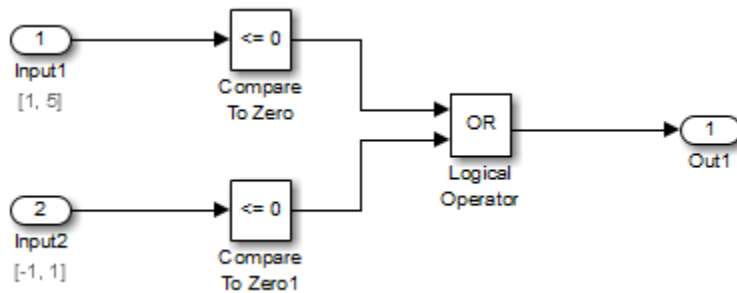
- “Specify Input Ranges for Inport Blocks” on page 11-4
- “Specify Input Ranges for Simulink.Signal Objects” on page 11-5
- “Specify Input Ranges for Stateflow Data Objects” on page 11-5
- “Specify Input Ranges for Subsystems” on page 11-6
- “Specify Input Ranges for Global Data Stores” on page 11-7
- “Specify Input Ranges for Bus Elements” on page 11-8

### Specify Input Ranges for Inport Blocks

After you specify the output minimum and maximum values on Inport blocks, Simulink Design Verifier analysis uses the minimum and maximum values as constraints.

The following example model restricts the signals from two Inport blocks:

- Input1 block: Minimum: 1, Maximum: 5
- Input2 block: Minimum: -1, Maximum: 1



When you use Simulink Design Verifier, to analyze this model, the analysis produces these results:

- The output from Input1 is never less than 0, therefore the first input to the Logical Operator block is never `false`. The objective that the first input to the Logical Operator equals `false` is unsatisfiable.
- The Logical Operator block cannot achieve 100% modified condition/decision coverage (MCDC) coverage because the condition where the first input is `false` never occurs.

The detailed analysis report shows the values you use as constraints for Input1 and Input2.

---

**Note** Simulink Design Verifier considers the full range of possible values (and any minimum and maximum constraints) for root-level inports only.

---

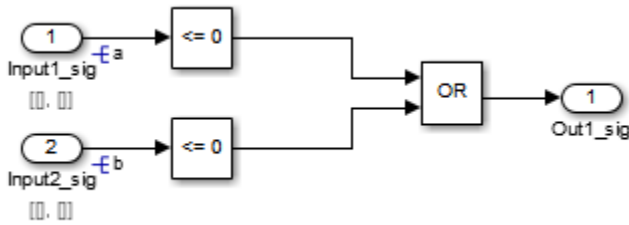


## Specify Input Ranges for Simulink.Signal Objects

Using the Model Explorer, in the model workspace, you can specify minimum and maximum values on `Simulink.Signal` objects associated with input signals.

The following example model uses the `Simulink.Signal` objects associated with the input signals `a` and `b` to restrict the signal values:

- Signal `a`: Minimum: 1, Maximum: 5
- Signal `b`: Minimum: -1, Maximum: 1



When you analyze this model, the results are the same as if you specified the minimum and maximum values on the input ports.

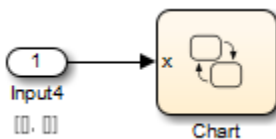
### Specifying Signal Ranges on Inport Blocks and Signals

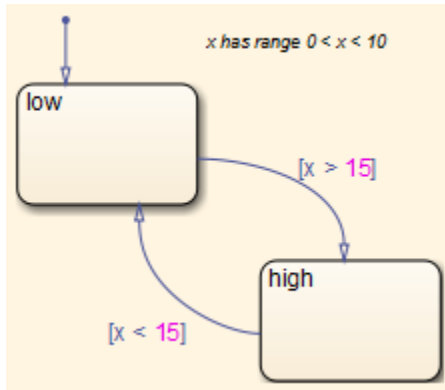
If you specify ranges on the Inport blocks and on the signals, the analysis considers the smallest range for the values. For example, if you specify a range of `4 . . 12` on an input port and a range of `1 . . 8` on the signal from the input port, the analysis considers the range `4 . . 8`.

## Specify Input Ranges for Stateflow Data Objects

Using the Model Explorer, you can specify ranges on data objects that are directly connected to the root-level input ports for a Stateflow chart.

In the following example model, the Stateflow chart named `Chart` has a data object, `x`, whose range you specified as  $0 < x < 10$ . In this chart, `x` must be greater than 15 to trigger the transition from low to high.





The value of  $x$  ranges from 0 through 10, therefore the transition condition  $[x > 15]$  is never true. The transition from `low` to `high` never occurs. Because the `high` state is never entered, the transition condition  $[x < 15]$  is never tested, and the transition from `high` to `low` never occurs. The chart is always in the `low` state.

When you analyze this model, these objectives are proven unsatisfiable:

- The high state is never entered.
- The transition condition  $[x > 15]$  is always false, never true.
- The condition  $[x < 15]$  is never tested, so it is never true or false.

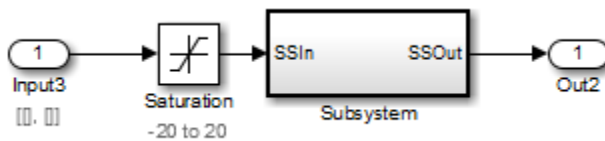
The analysis report indicates the values that you use as constraints for  $x$ :  $[0, 10]$ .

## Specify Input Ranges for Subsystems

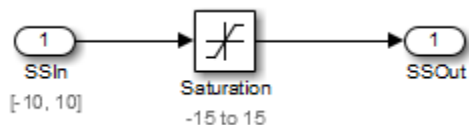
The Simulink Design Verifier software considers specified input minimum and maximum values as constraints only at the top level of a model. You can specify minimum and maximum values on Input ports on subsystems, but when you analyze the top-level model, the software ignores those values.

When you perform the subsystem analysis, the software considers specified minimum and maximum values on the input ports of the subsystem.

For example, consider the following model and its subsystem.



In Subsystem, the specified minimum and maximum values for input port `SSIn` are -10 and 10, respectively. The lower and upper limits for the Saturation block are -15 and 15, respectively.



If you right-click Subsystem in the top-level model and select **Design Verifier > Generate Tests for Subsystem**, the analysis considers the specified minimum and maximum values as constraints on the SSIn port.

| Name | Design Min Max Constraint |
|------|---------------------------|
| SSIn | [-10, 10]                 |

### Constraints: Design Min Max Constraints

The analysis identifies two unsatisfiable objectives:

- input > lower limit F: The input is always greater than the lower limit on the Saturation block (-15).
- input >= upper limit T: The input is never greater than or equal to the upper limit on the Saturation block (15).

If you analyze the model that contains Subsystem, the analysis does not consider the values specified on the input port SSIn in the subsystem. The analysis considers only the root-level input ports at the respective level of the hierarchy for analysis.

### Specify Input Ranges for Global Data Stores

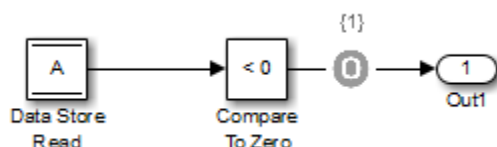
A data store is a repository to which you can write data and from which you can read data, without having to connect an input or output signal directly to the data store. You create a data store by using a Data Store Memory block or a Simulink.Signal object. You can specify minimum and maximum values for any data store.

During subsystem analysis, Simulink Design Verifier creates an input port to mimic the execution context for a global data store. For more information, see “Extract Subsystems for Analysis” on page 14-15. If the data store has specified minimum and maximum values, those values are assigned as minimum and maximum values on the new input port. Simulink Design Verifier analysis considers the input minimum and maximum values as subsystem-level analysis constraints.

In the following example model, data store A has a minimum value of 0 and a maximum value of 10.



The atomic subsystem reads values from the data store and checks to see if the input is less than 0. The Compare To Zero block outputs 1 if the input is less than 0, and outputs 0 if the input is greater than or equal to 0. The Test Objective block checks to see if the output is ever 1.



In the top-level model, if you right-click Subsystem and select **Design Verifier > Generate Tests for Subsystem**, the analysis considers the constraints for data store A to be [0, 10].

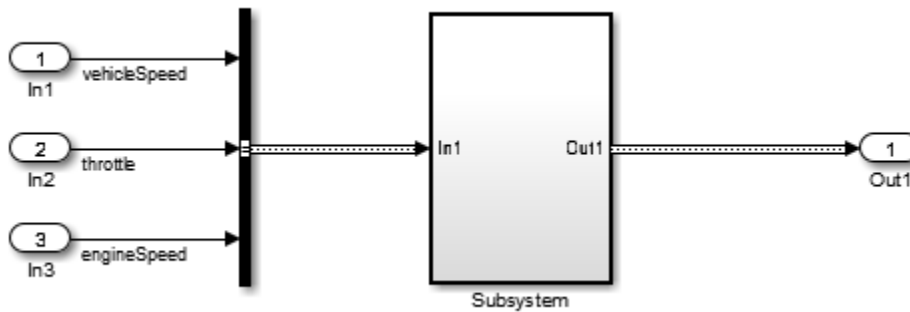
The analysis does not satisfy the objective specified in the Test Objective block. The input is always greater than or equal to 0, therefore the output from the Compare To Zero block is always 0.

## Specify Input Ranges for Bus Elements

When you define a bus, you can specify minimum and maximum values for the elements in the bus. Simulink Design Verifier considers these minimum and maximum values when analyzing subsystems and models that use the bus as an input signal.

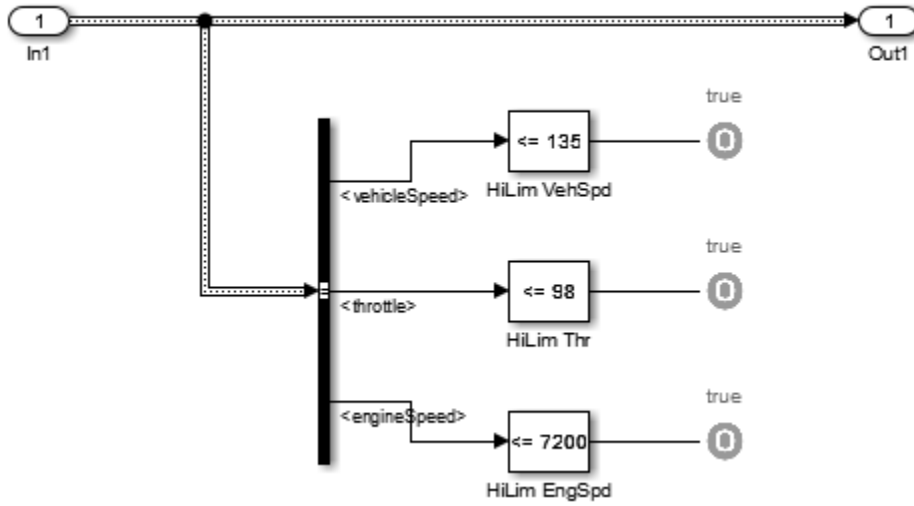
Consider a subsystem that inputs a bus of three fields, each with a defined minimum and maximum. To view this subsystem, add the examples folder to the current MATLAB path and at the command line, enter:

```
open_system('sldvBusMinMaxExample');
```



| Bus Element  | Bus Element Minimum | Bus Element Maximum |
|--------------|---------------------|---------------------|
| vehicleSpeed | 0                   | 125                 |
| throttle     | 0                   | 100                 |
| engineSpeed  | 0                   | 7600                |

The subsystem has test objectives that confirm that each element does not exceed a constant. The `vehicleSpeed` signal is limited to a maximum value lower than the test objective.



Set the current folder to a writable folder. In the top-level mode, right-click Subsystem and select **Design Verifier > Generate Tests for Subsystem**. The Condition Objective for testing `vehicleSpeed > 135` is not satisfiable due to the maximum specification on the `vehicleSpeed` element.

## Specification of Input Ranges in sldvData Fields

When you analyze a model, Simulink Design Verifier generates a data file when it completes its analysis. The data file is a MAT-file that contains an `sldvData` structure. The `sldvData` structure stores all the data that the software gathers and produces during the analysis. You can use the data file to customize your own analysis or to generate a custom report.

If your model contains specified minimum and maximum values on the input ports, the `sldvData` structure contains information about those values. For example, after analyzing the `ex_minmax_on_inports` model in “Specify Input Ranges for Inport Blocks” on page 11-4, the data file contains the following values:

- For the Input1 block:

```
sldvData.Constraints.DesignMinMax(1).value{1}.low
```

```
ans =
```

```
1
```

```
sldvData.Constraints.DesignMinMax(1).value{1}.high
```

```
ans =
```

```
5
```

- For the Input2 block:

```
sldvData.Constraints.DesignMinMax(2).value{1}.low
```

```
ans =
```

```
-1
```

```
sldvData.Constraints.DesignMinMax(2).value{1}.high
```

```
ans =
```

```
1
```

# Proving Properties of a Model

---

- “What Is Property Proving?” on page 12-2
- “Workflow for Proving Model Properties” on page 12-4
- “Prove Properties in a Model” on page 12-5
- “Prove System-Level Properties Using Verification Model” on page 12-20
- “Prove Properties in a Subsystem” on page 12-23
- “Model Requirements” on page 12-24
- “Isolate Verification Logic with Observers” on page 12-29
- “Property Proving with an Invalid Property” on page 12-32
- “Property Proving with Multiple Properties” on page 12-33
- “Property Proving with an Assumption Block” on page 12-34
- “Property Proving Workflow for Cruise Control” on page 12-35
- “Property Proving Workflow for Fixed-Point Cruise Control with Block Replacements” on page 12-39
- “Property Proving Using MATLAB Function Block” on page 12-40
- “Property Proving Using MATLAB Truth Table Block” on page 12-41
- “Property Proving Workflow for Thrust Reverser” on page 12-42
- “Debounce Temporal Properties” on page 12-43
- “Power Window Controller Temporal Properties” on page 12-46
- “Debug Property Proving Violations by Using Model Slicer” on page 12-55
- “Design and Verify Properties in a Model” on page 12-60
- “Validate Requirements by Analyzing Model Properties” on page 12-63
- “Use Observer Reference Blocks for Property Proving Analysis” on page 12-70
- “Prove Properties with Requirements Table Blocks” on page 12-73

## What Is Property Proving?

A property is a requirement that you model in Simulink or Stateflow, or by using MATLAB Function or Requirements Table 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”.

---

### Proof Functions

The Simulink Design Verifier software provides two Stateflow and MATLAB for code generation functions to specify property proving for a Simulink model or Stateflow chart:

- `sldv.prove` — Specifies a proof objective
- `sldv.assume` — Specifies a proof assumption

These functions:

- Identify mathematical relationships for proving properties in a form that can be more natural than using block parameters
- Support specifying multiple objectives, assumptions, or conditions without complicating the model.
- Provide access to the power of MATLAB.
- Support separation of verification and model design.

For an example of how to use these proof functions, see the `sldv.prove` reference page.



---

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

---

## Workflow for Proving Model Properties

To prove properties of your design model, use the following workflow:

- 1** Determine the verification objectives for your design model, e.g., based on your requirements specifications.
- 2** Instrument your design model to specify proof objectives and proof assumptions.
  - For simple properties, instrument your model with blocks or MATLAB functions that specify the proof objectives.
  - For system-level properties, construct a verification model that contains a Model block that references the design model and define the properties on the design model interface using the same inputs and outputs.
- 3** Define analysis constraints using the Proof Assumption block or `sldv.assume`. These constraints apply to all enabled proof objectives.

---

**Note** The proof assumptions are applied to all enabled proof objectives. Make sure that you do not specify any contradictory assumptions because that might invalidate the entire analysis.

---

- 4** Specify options that control how Simulink Design Verifier proves the properties of your model.
- 5** Execute the Simulink Design Verifier analysis and review the results.

For an exercise that demonstrates this workflow, see “Prove Properties in a Model” on page 12-5.

### See Also

#### More About

- “Property Proving Workflow for Cruise Control” on page 12-35
- “Property Proving Workflow for Fixed-Point Cruise Control with Block Replacements” on page 12-39
- “Property Proving Workflow for Thrust Reverser” on page 12-42

## Prove Properties in a Model

### In this section...

“About This Example” on page 12-5  
 “Construct Example Model” on page 12-5  
 “Check Compatibility of Example Model” on page 12-6  
 “Instrument Example Model” on page 12-7  
 “Configure Property-Proving Options” on page 12-8  
 “Analyze Example Model” on page 12-8  
 “Review Analysis Results” on page 12-8  
 “Customize Example Proof” on page 12-15  
 “Reanalyze Example Model” on page 12-16  
 “Review Results of Second Analysis” on page 12-16  
 “Analyze Contradictory Models” on page 12-18  
 “Prove Properties in a Large Model” on page 12-19

### About This Example

The following sections describe a Simulink model, for which you prove a property that you specify using a Proof Objective block. This example demonstrates the property-proving capabilities of Simulink Design Verifier.

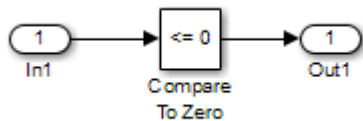
In this example, you perform the following tasks.

| Task | Description   | See...  |
|------|---|---|
| 1    | Construct the example model.  | “Construct Example Model” on page 12-5              |
| 2    | Verify that your model is compatible with Simulink Design Verifier. | “Check Compatibility of Example Model” on page 12-6 |
| 3    | Add a Proof Objective block to your model to prepare for its proof. | “Instrument Example Model” on page 12-7             |
| 4    | Configure Simulink Design Verifier to prove properties.             | “Configure Property-Proving Options” on page 12-8   |
| 5    | Prove a property of your model.                                     | “Analyze Example Model” on page 12-8                |
| 6    | Review the analysis results.  | “Review Analysis Results” on page 12-8              |
| 7    | Add proof assumptions to specify analysis constraints.              | “Customize Example Proof” on page 12-15             |
| 8    | Prove a property of the customized model and interpret the results. | “Reanalyze Example Model” on page 12-16             |

### Construct Example Model

Construct a Simulink model to use in this example:

- 1 Create an empty Simulink model.
- 2 Copy the following blocks into your empty model window:
  - From the Sources library, an Inport block to initiate the input signal whose value Simulink Design Verifier controls
  - From the Logic and Bit Operations library, a Compare To Zero block to provide simple logic
  - From the Sinks library, an Outport block to receive the output signal
- 3 Connect these blocks such so your model appears similar to the following model:



- 4 On the **Modeling** tab, click **Model Settings**.
- 5 On the Configuration Parameters dialog box, in the **Solver** pane, in the **Solver selection**:
  - Set the **Type** option to Fixed-step.
  - Set the **Solver** option to Discrete (no continuous states).

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

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

## Check Compatibility of Example Model

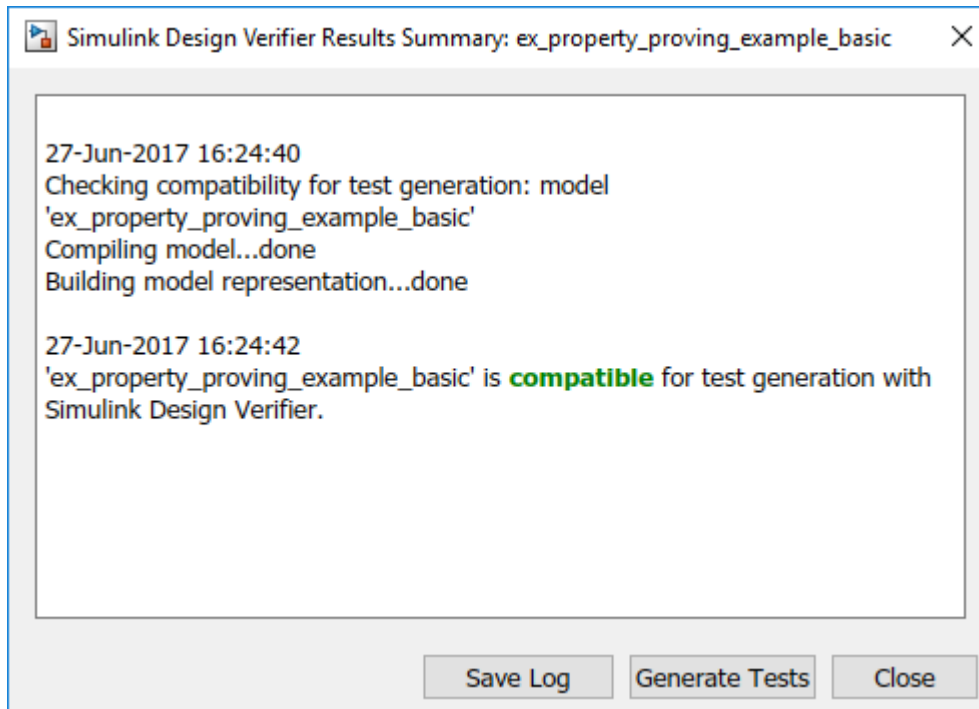
Every time Simulink Design Verifier software analyzes a model, before the analysis begins, the software performs a compatibility check. If your model is not compatible, the software cannot analyze it.

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

- 1 Open the `ex_property_proving_example_basic` model.
- 2 On the **Design Verifier** tab, click **Check Compatibility**.

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

The model you just created is compatible.



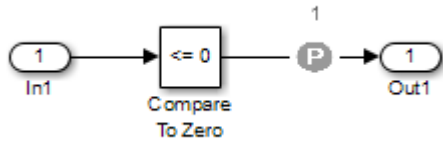
### What If a Model Is Partially Compatible?

If the compatibility check indicates that your model is partially compatible, your model contains at least one object that Simulink Design Verifier does not support. You can analyze a partially compatible model, but, by default, unsupported objects are stubbed out. The results of the analysis may be incomplete. For detailed information about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.

### Instrument Example Model

Prepare your example model so that you can prove its properties with Simulink Design Verifier. Specifically, instrument the model by adding and configuring a Proof Objective block:

- 1 In the MATLAB Command Window, enter `sldvlib`.  
The Simulink Design Verifier library appears.
- 2 Open the Objectives and Constraints sublibrary.
- 3 Copy the Proof Objective block to your model and insert it between the Compare To Zero and Output blocks.
- 4 In your model, double-click the Proof Objective block.  
The Proof Objective block parameters dialog box opens.
- 5 In the **Values** box, enter 1.  
The Simulink Design Verifier software will attempt to prove that the signal output by the Compare To Zero block always attains this value for any signals that it receives.
- 6 Click **OK** to apply your changes and close the Proof Objective block parameters dialog box.



- 7 Save your model and keep it open.

## Configure Property-Proving Options

Configure Simulink Design Verifier to prove properties of the `ex_property_proving_example_basic` model that you instrumented:

- 1 Open the `ex_property_proving_example_basic` model.
- 2 On the **Design Verifier** tab, in the **Mode** section, select **Property Proving**.
- 3 Click **Property Proving Settings**.
- 4 Click **OK** to apply your changes and close the Configuration Parameters dialog box.

---

**Note** On the **Property Proving** pane, you can optionally specify values for other parameters that control how Simulink Design Verifier proves properties of your model. For more information, see “Design Verifier Pane: Property Proving” on page 15-52.

---

- 5 Save the `ex_property_proving_example_basic` model.

## Analyze Example Model

To analyze the `ex_property_proving_example_basic` model, on the **Design Verifier** tab, click **Prove Properties**. Simulink Design Verifier begins a property-proving analysis.

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

To terminate the analysis at any time, in the log window, click **Stop**.

## Review Analysis Results

When the analysis is complete, the log window displays the following options for reviewing the results:

- Highlight the analysis results on the model
- Generate a detailed HTML analysis report
- Create a harness model with test cases
- Simulate the test cases created by the model and produce a model coverage report

You can also view the Simulink Design Verifier data file. For detailed information about the data file, see “Manage Simulink Design Verifier Data Files” on page 13-7.

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

- “Review Results on Model” on page 12-9
- “Review Detailed Analysis Report” on page 12-10
- “Review Harness Model” on page 12-12
- “Simulate Model with Counterexample” on page 12-13
- “Review Analysis Results” on page 12-15

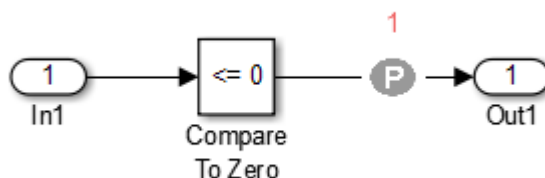
### Review Results on Model

You can review the analysis results at a glance by viewing the blocks that are highlighted in the model window. The highlighting can have four colors:

- Green — The analysis proved all the proof objectives valid.
- Red — The analysis disproved a proof objective and generated a counterexample that falsified that objective.
- Orange — The analysis disproved a proof objective, but it could not generate a counterexample or the proof objective remained undecided. This result occurs due to:
  - A proof objective on a signal whose value the software cannot control, for example, a Constant block
  - A proof objective that depends on nonlinear computation
  - A proof objective that creates an arithmetic error, such as division by zero
  - Automatic stubbing being enabled, and the analysis encountering an unsupported block whose operation it does not understand but that the analysis requires to generate the counterexample
  - The analysis timing out
  - Limitations of the analysis engine
- Gray — The model object was not part of the analysis.

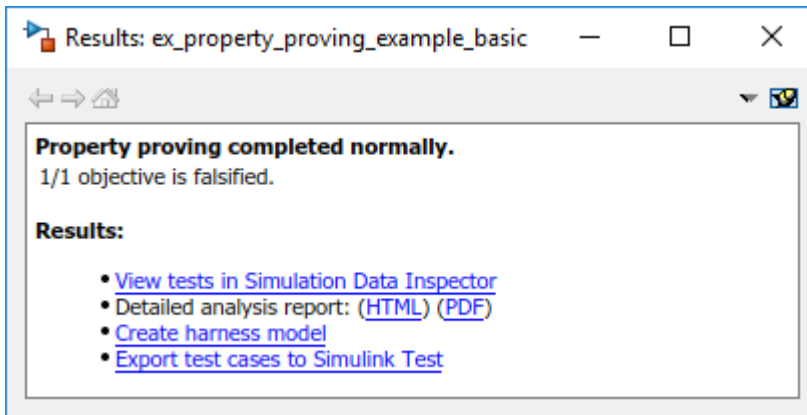
Highlight the analysis results on the example model:

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

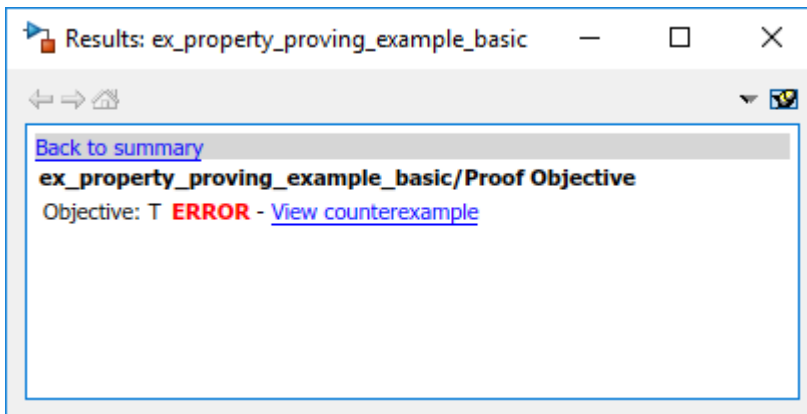


The Proof Objective block is highlighted in red, which indicates that a proof objective was falsified with a counterexample.


The Simulink Design Verifier Results window appears.



As you click objects in the model, this window changes to display detailed analysis results for that object.




---

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

---

- 2 Click the highlighted Proof Objective block.

The Simulink Design Verifier Results window indicates that the proof objective that the output signal from the Compare to Zero was not 1 was disproved with a counterexample.

### Review Detailed Analysis Report

To create a detailed HTML analysis report:

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

The HTML report opens in a browser window.

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



| Table of Contents |   |
|-------------------|---|
| 1.                | <a href="#">Summary</a>                 |
| 2.                | <a href="#">Analysis Information</a>    |
| 3.                | <a href="#">Proof Objectives Status</a> |
| 4.                | <a href="#">Properties</a>              |

- 3 In the **Table of Contents**, click Summary.

## Chapter 1. Summary

### Analysis Information

Model: ex\_property\_proving\_example\_basic  
 Mode: Property proving  
 Status: Completed normally  
 Analysis Time: 11s

The Summary provides an overview of the analysis results, and it indicates that Simulink Design Verifier identified a counterexample that falsifies an objective in your model.

- 4 In the **Table of Contents**, click Proof Objectives Status.

### Objectives Falsified with Counterexamples

| # | Type            | Model Item                      | Description  | Analysis Time (sec) | Counterexample    |
|---|-----------------|---------------------------------|--------------|---------------------|-------------------|
| 1 | Proof objective | <a href="#">Proof Objective</a> | Objective: T | 12                  | <a href="#">1</a> |

The Objectives Falsified with Counterexamples table lists the proof objectives that Simulink Design Verifier disproved using a counterexample that it generated. You can locate the objective in your model window by clicking [Proof Objective](#); the software highlights the corresponding Proof Objective block in your model window.

- 5 In the Objectives Falsified with Counterexamples table, under the **Counterexample** column, click 1.

## Proof Objective

### Summary

Model Item: [Proof Objective](#)

Property: Objective: T

Status: Falsified

### Counterexample

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

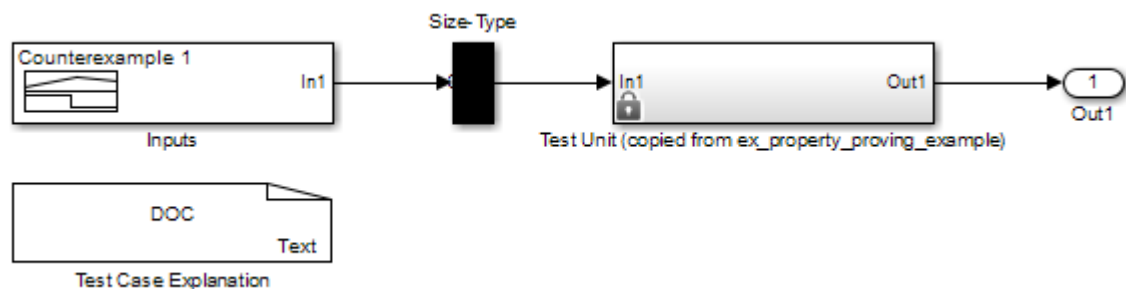
This section displays information about proof objective 1 and provides details about the counterexample that Simulink Design Verifier generated to disprove that objective. In this counterexample, a signal value of 99 falsifies the objective that you specified using the Proof Objective block. That is, 99 is not less than or equal to 0, which causes the Compare To Zero block to return 0 (false) instead of 1 (true).

### Review Harness Model

Create a harness model with counterexamples that falsify the proof objectives in your model:

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

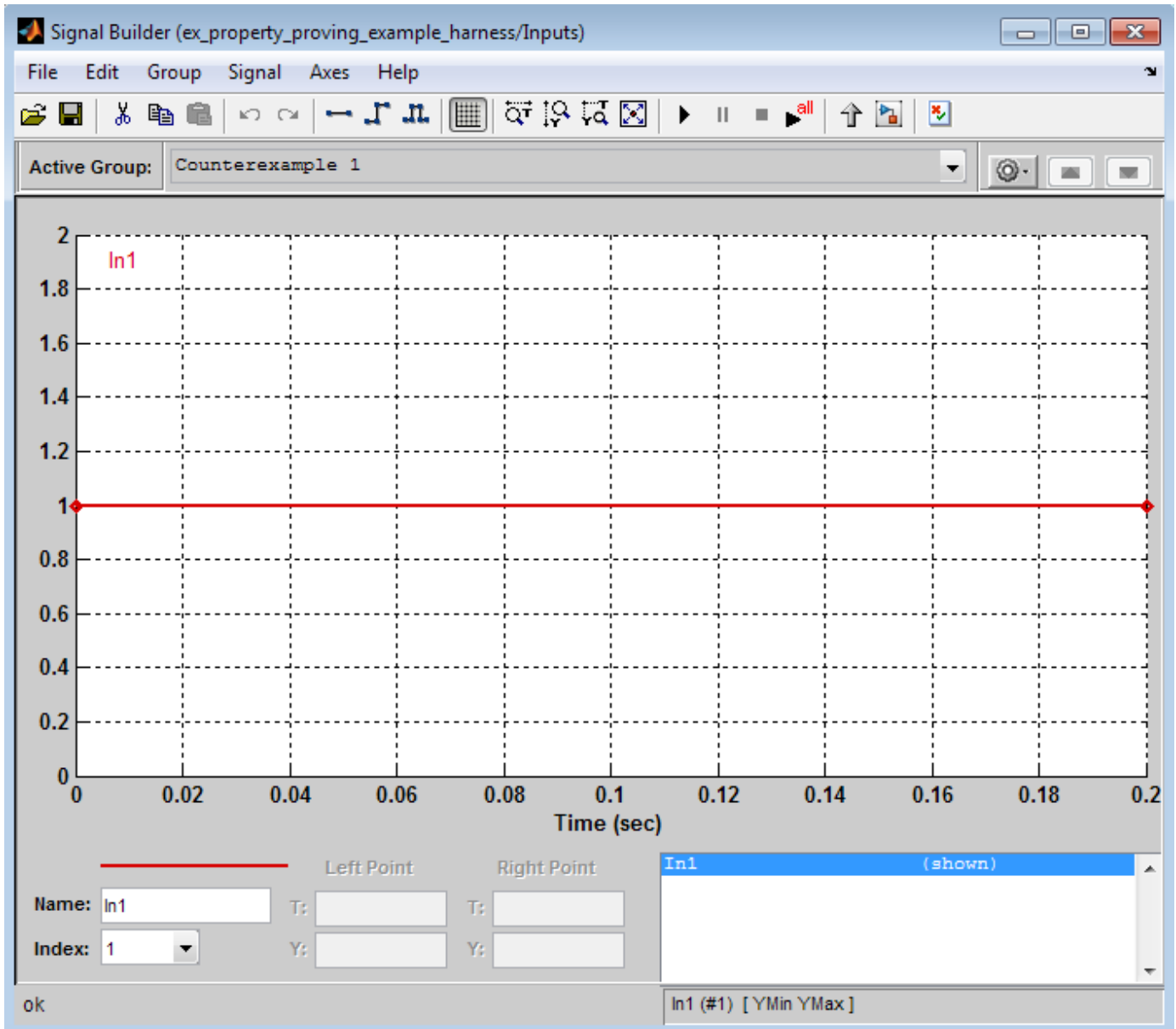
The software creates a harness model named `ex_property_proving_example_basic_harness`.



The harness model contains the following items:

- Signal Builder block named **Inputs** — A group of signals that falsify proof objectives.
- Subsystem block named **Test Unit** — A copy of your model.
- DocBlock named **Test Case Explanation** — A textual description of the counterexamples that the analysis generates.

- A Size-Type block — A subsystem that transmits signals from the Inputs block to the Test Unit block. This block verifies that the size and data type of the signals are consistent with the Test Unit block.
- 2 Double-click the Inputs block.

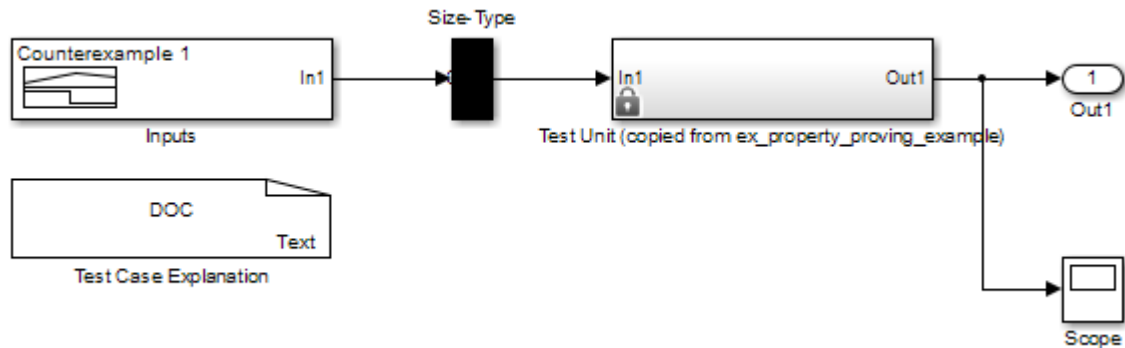


The input signal 1 causes the output of the Compare to Zero block to be 0. This counterexample violates the proof objective that specifies that the output of the Compare to Zero block be 1.

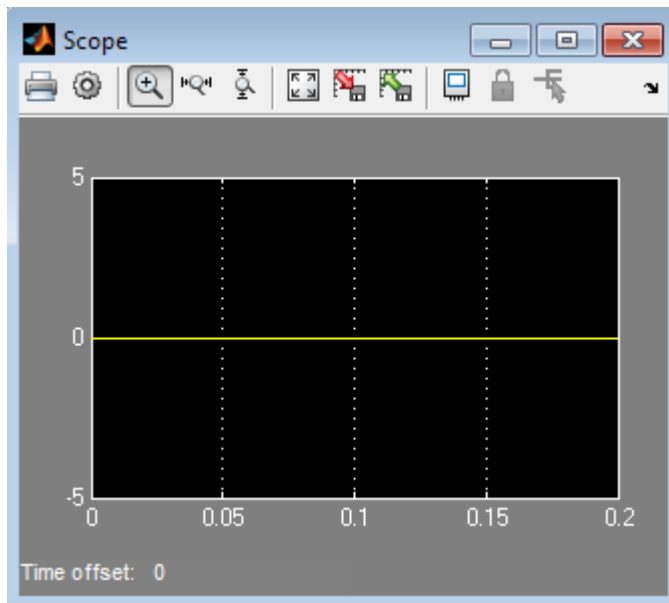
### Simulate Model with Counterexample

Simulate the harness model to observe the counterexample that falsifies the proof objective in your model:

- 1 Open the `ex_property_proving_example_basic` model.
- 2 On the **Simulation** tab, click **Library Browser**.
- 3 From the Sinks library, copy a Scope block into your harness model window. The Scope block allows you to see the value of the signal output by the Compare To Zero block in your model.
- 4 In your harness model window, connect the output signal of the Test Unit subsystem to the Scope block.



- 5 To simulate your harness model, on the **Simulation** tab, click **Run**.  
The Simulink software simulates the harness model.
- 6 In your harness model window, double-click the Scope block to open its display window.



The Scope block displays the value of the signal output by the Compare To Zero block in your model. In this example, the Compare To Zero block returns 0 (false) throughout the simulation, which falsifies the proof objective that the output of the Compare to Zero block be 1 (true). The counterexample that the Signal Builder block supplies falsifies the proof objective.

## Review Analysis Results

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

On the **Design Verifier** tab, in the **Review Results** section, click **Results Summary**. The Results Summary window opens displaying the results of the latest Simulink Design Verifier analysis.

For any Simulink Design Verifier analysis, from the Results Summary window, you can perform the following tasks.

| Task  | For more information   |
|---|--|
| Highlight the analysis results on the model.  | “Highlight Results on the Model” on page 13-2                  |
| Generate a detailed analysis report.  | “Review Results” on page 13-35                                 |
| Create the harness model, or if the harness model already exists, open it.<br><br>If no counterexamples were created during the analysis, this option is not available. | “Manage Simulink Design Verifier Harness Models” on page 13-13 |
| View the data file.   | “Manage Simulink Design Verifier Data Files” on page 13-7      |
| View the log file.  | “View Log Files” on page 13-56                                 |

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

## Customize Example Proof

Modify the simple Simulink model whose proof objective Simulink Design Verifier disproved in the previous task. Specifically, customize the proof by adding and configuring a Proof Assumption block:

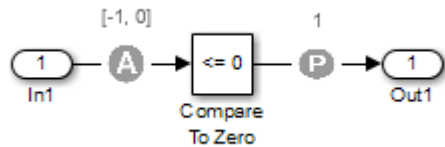
- 1 In the MATLAB Command Window, type `sldvlib`.

The Simulink Design Verifier library opens.

- 2 Open the Objectives and Constraints sublibrary.
- 3 Copy the Proof Assumption block to your model.
- 4 In your model window, insert the Proof Assumption block between the Inport and Compare To Zero blocks.
- 5 In your model, double-click the Proof Assumption block to access its attributes.

The Proof Assumption block parameter dialog box opens.

- 6 In the **Values** box, enter `[-1, 0]`. When proving properties of this model, Simulink Design Verifier constrains the signal values entering the Compare To Zero block to the specified range. If the input to the Compare to Zero block is always within this range, the output of the Compare to Zero block will always be 1.
- 7 Click **Apply** and then **OK** to apply your changes and close the Proof Assumption block parameter dialog box.



- 8 Save the `ex_property_proving_example_basic` model and keep it open.

## Reanalyze Example Model

Analyze the model that you modified to see how the Proof Assumption block affects the property-proving analysis.

Open the `ex_property_proving_example_basic` model. On the **Design Verifier** tab, click **Prove Properties**.

When the analysis is complete, the log window displays the options. There is no option to create a harness model, because the analysis satisfied all proof objectives in your model, so there are no counterexamples.

## Review Results of Second Analysis

Review the results of the second analysis:

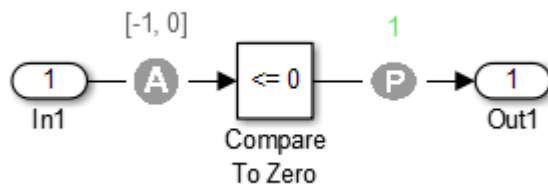
- “Review Results on the Model” on page 12-16
- “Review Analysis Report” on page 12-17

### Review Results on the Model

Highlight the model to see the analysis results:

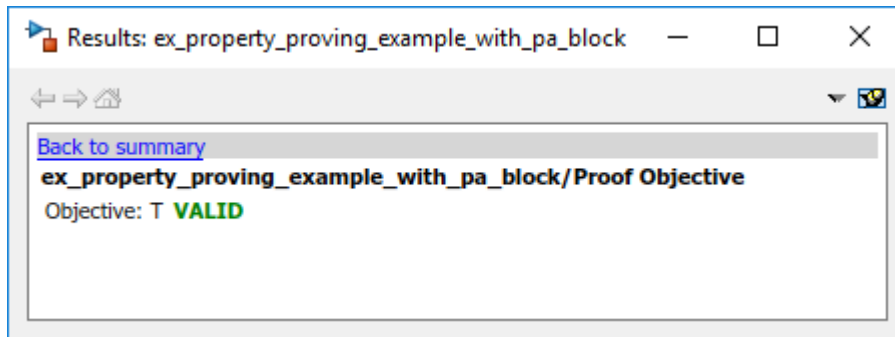
- 1 Click **Highlight analysis results on model**.

The Proof Objective is now highlighted in green.



- 2 Click the Proof Objective block.

The Simulink Design Verifier Results window shows that the proof objective that states that the signal be 1 is valid.



### Review Analysis Report

Review the analysis results in the detailed report:

- 1 Click **Generate detailed analysis report**.
- 2 In the **Table of Contents**, click Summary.

## Chapter 1. Summary

### Analysis Information

Model: ex\_property\_proving\_example\_with\_pa\_block  
 Mode: Property proving  
 Status: Completed normally  
 Analysis Time: 11s

### Objectives Status

**Number of Objectives: 1**  
 Objectives Valid: 1

The Summary chapter indicates that Simulink Design Verifier proved a proof objective in the model.

- 3 The Constraints section lists the analysis constraint you specified in the Proof Assumption block.

| Constraints                |                     |
|----------------------------|---------------------|
| Analysis Constraints       |                     |
| Name                       | Analysis Constraint |
| <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.

## Objectives Valid

| # | Type            | Model Item                      | Description  | Analysis Time (sec) | Counterexample |
|---|-----------------|---------------------------------|--------------|---------------------|----------------|
| 1 | Proof objective | <a href="#">Proof Objective</a> | Objective: T | 5                   | n/a            |

The Objectives Proven Valid table lists the proof objectives that Simulink Design Verifier proved to be valid.

- 5 Scroll down to view the Properties chapter or go to the top of the browser window and in the **Table of Contents**, click Properties.

## Proof Objective

### Summary

Model Item: [Proof Objective](#)

Property: Objective: T

Status: Valid

The Proof Objective summary indicates that Simulink Design Verifier proved an objective that you specified in your model. The Proof Assumption block restricts the domain of the input signals to the interval  $[-1, 0]$ . Therefore, the software proves that this interval does not contain values that are greater than zero, thereby satisfying the proof objective.

## Analyze Contradictory Models

If the analysis produces the error `The model is contradictory in its current configuration`, the software detected a contradiction in your model and it cannot analyze the model. You can have a contradiction if your model has Proof Assumption blocks with incorrect parameters. For example, an assumption could state that a signal must be between 0 and 5 when the signal is constant 10.

If the software detects a contradiction, all previous results are invalidated and the software reports that all the properties are falsified.

---

**Note** Constraints added at the inputs either through design minimum/maximum or test conditions/proof assumptions do not lead to a contradiction. However, if you constrain signals that are downstream of a computation using test conditions/proof assumptions, you must ensure that the constrained condition is feasible through the model computation. Otherwise, the resulting model is contradictory that may produce invalid results with or without an explicit analysis error. To ensure that the constraints are feasible, first try the same condition using a Test Objective. If it can be satisfied, you can use the same condition safely as a constraint.

---



## Prove Properties in a Large Model

A thorough proof of your model requires that Simulink Design Verifier search through all reachable configurations of your model—even the ones that are reached only after long time delays. The computation time and memory required to search a model completely often make an exhaustive proof impractical.

“Prove Properties in Large Models” on page 14-24 gives detailed information about strategies you can use to improve the performance of a property-proving analysis of a large model.

## See Also

### More About

- “Property Proving with an Invalid Property” on page 12-32
- “Property Proving with Multiple Properties” on page 12-33
- “Property Proving with an Assumption Block” on page 12-34

## Prove System-Level Properties Using Verification Model

### In this section...

“When to Use a Verification Model for Property Proving” on page 12-20

“About This Example” on page 12-20

“Understand the Verification Model” on page 12-20

“Prove the Properties of the Design Model” on page 12-21

“Fix the Verification Model” on page 12-22

### 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.

### Understand the Verification Model

Take these steps to understand how the verification model works:

- 1 Open the verification model:

```
sldvdemo_sbr_verification
```

The Design Model block is a Model block that references `sldvdemo_sbr_design`. The SBR Stateflow chart in the design model assumes that the KEY input is initially 0.

- 2 Open the Safety Properties subsystem that specifies the properties of the design model that you want to prove.

This subsystem contains a MATLAB Function block called **MATLAB Property**. The code in this block specifies the property that the seat belt reminder should be on when the ignition is on, the seat belt is not fastened, and the speed is less than 15:

- 3 Close the Safety Properties subsystem.

- 4 Open the Input Constraints subsystem.

This subsystem defines the following constraints:

- The key can have three positions: 0, 1, 2
  - The speed is constrained to fall between 10 and 30.
  - The key must start at 0 and can only change by one increment at a time. For example, the key can change from 0 to 1 or 1 to 2, but not from 0 to 2. In this verification model, this constraint is not enabled.
- 5 Close the Input Constraints subsystem, but keep the `sldvdemo_sbr_verification` model open.

## Prove the Properties of the Design Model

Analyze the `sldvdemo_sbr_verification` model to prove the properties:

- 1 In the `sldvdemo_sbr_verification` model window, to start the analysis, double-click the **Run** button to start the analysis.

When the analysis completes, the Simulink Design Verifier log window indicates that one objective is falsified - needs simulation. For more information, see “Objectives Falsified - Needs Simulation” on page 13-49.

- 2 To see which objective was falsified, click **Highlight analysis results on model**.

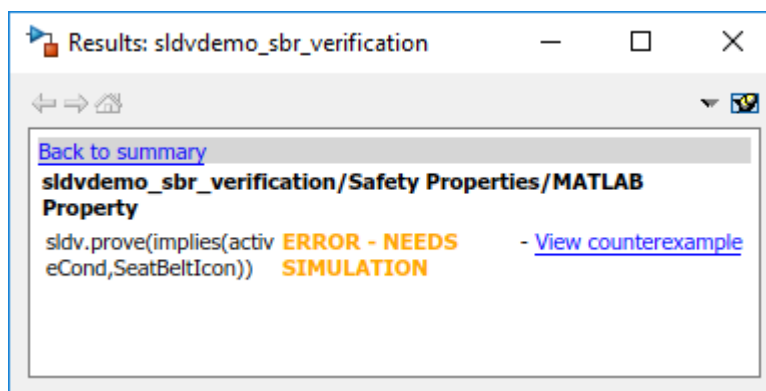
The Safety Properties subsystem is highlighted in orange.

- 3 Open the Safety Properties subsystem and click the MATLAB Property block.

The Simulink Design Verifier Results window indicates that the statement

```
sldv.prove(implies(activeCond,SeatBeltIcon))
```

was false during at least one time step.



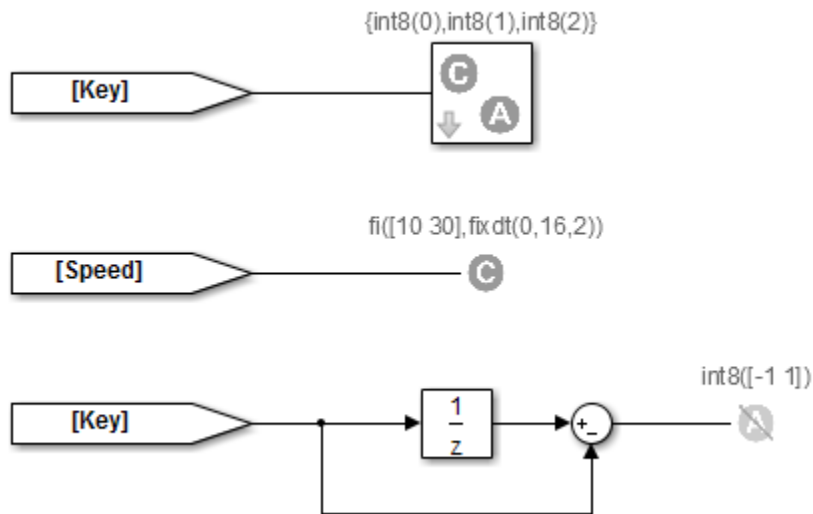
- 4 Click **View counterexample** to see the signal values that violated this property.

The Signal Builder block opens with the counterexample. The KEY input was initially 2, which is invalid.

To validate the property specified in the Safety Properties subsystem, you have to make sure that the initial value of KEY is 0.

## Fix the Verification Model

The Input Constraints subsystem in the verification model contained three constraints. The third constraint, which requires that the initial value of KEY be 0, and that KEY can only change in increments of 1, is disabled.



To see how this property is validated when you enable the third constraint:

- 1 In the `sldvdemo_sbr_verification` model, click **Open Fixed Model**.

The `sldvdemo_sbr_verification_fixed` verification model opens.

- 2 Open the Input Constraints subsystem.

This third constraint is now enabled so that KEY has an initial value of 0 and changes in increments of 1.

- 3 Close the Input Constraints subsystem.
- 4 In the `sldvdemo_sbr_verification_fixed` model, to start the analysis, double-click the **Run** block.

The analysis proves the validity of the property.

## See Also

### More About

- “Property Proving Using MATLAB Function Block” on page 12-40
- “Property Proving Using MATLAB Truth Table Block” on page 12-41

## Prove Properties in a Subsystem

If you have a large model, you can prove the properties of a subsystem in the model and review the analyses in smaller, manageable reports. The workflow for proving properties in a subsystem is:

- 1 Open the model that contains the subsystem.
- 2 Make the subsystem atomic.
- 3 Run Simulink Design Verifier using the **Prove Properties of Subsystem** option.
- 4 Review the results.

The tutorial in “Generate Test Cases for a Subsystem” on page 7-18 explains how to generate test cases for the Controller subsystem in the Cruise Control Test Generation model. The steps for proving properties are similar to those for generating test cases, except that you select the **Prove Properties of Subsystem** option instead of the **Generate Tests for Subsystem** option.

## Model Requirements

The Simulink Design Verifier block library includes a sublibrary Example Properties. The Example Properties sublibrary includes:

- “Basic Properties” on page 12-24 — Four examples that demonstrate how to prove basic properties.
- “Temporal Properties” on page 12-26 — Four examples that demonstrate how to define temporal properties on Boolean signals

The workflow for using these examples in your model is:

- 1 Copy these examples into your Verification Subsystem block.
- 2 Adapt them, if required, for the specific properties that you want to prove.
- 3 Run the Simulink Design Verifier analysis to prove that the assertions in these examples never fail.
- 4 If the assertion fails, the software creates a counterexample that causes the assertion to fail and then generates a harness model.
- 5 On the harness model, execute the counterexample to confirm that the assertion fails with that counterexample.

### Basic Properties

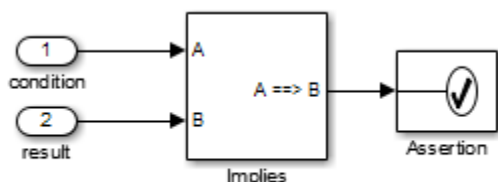
To view the Basic Properties examples:

- 1 Open the Simulink Design Verifier block library. Type:  
`sldvlib`
- 2 Double-click the Examples sublibrary.
- 3 Double-click the **Basic Properties** block that contains the examples.

The sections that follow describe each example in the Block Properties sublibrary in detail.

#### Conditions that Trigger a Result

The Simulink Design Verifier Implies block allows you to test for conditions that trigger a result. This example specifies that if condition A is true, result B must always be true.

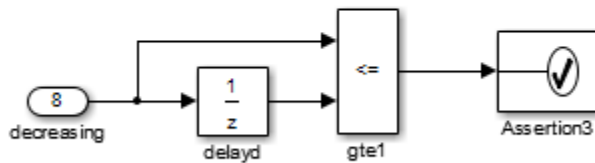
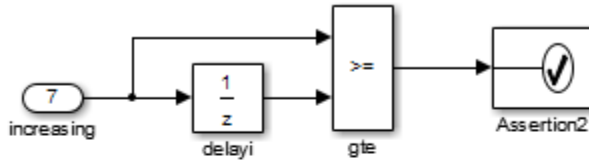


Implies operation describes conditions that should trigger a result.

#### Increasing or Decreasing Signals

The two examples in this section specify that a signal is either:

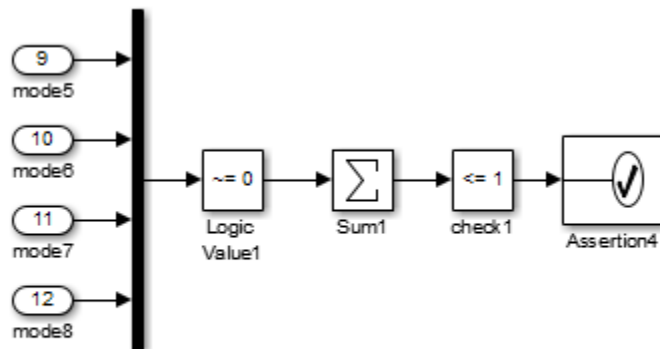
- Always increasing or staying constant
- Always decreasing or staying constant



Increasing and decreasing operations describe signals that should increase or decrease.

### Exclusivity Operation

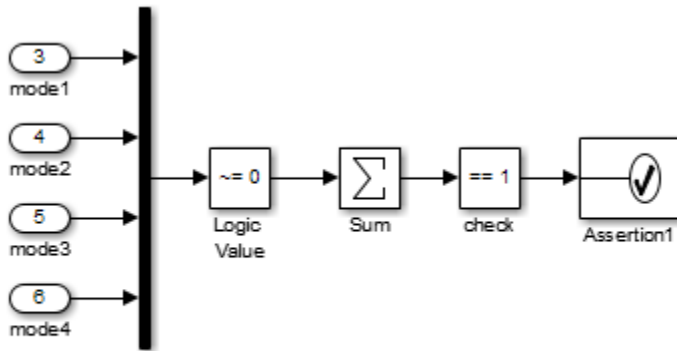
This example describes four conditions that should not be true at the same time.



Exclusivity operation describes conditions that should never be true at same time.

### Conditions with One True Element

This example specifies that only one of the four input signals can be true.



Mutual exclusivity operation describes conditions that should have exactly one true element.

## Temporal Properties

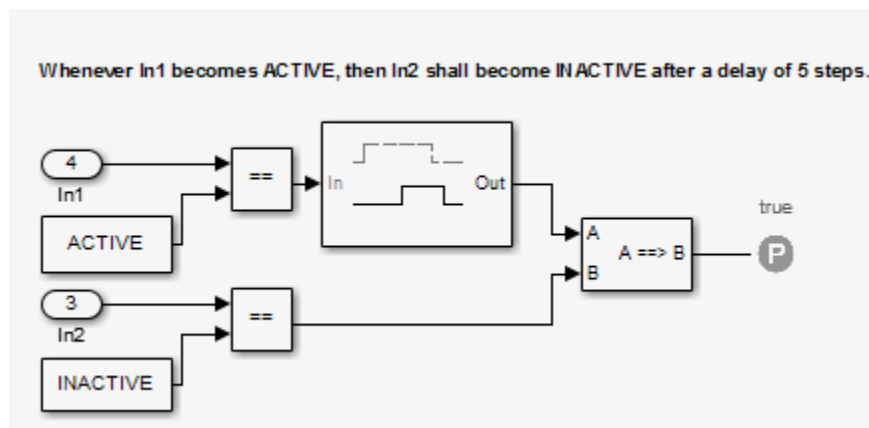
To view the Temporal Properties examples:

- 1 Open the Simulink Design Verifier block library. Type: `sldvlib`
- 2 Double-click the Temporal Properties sublibrary.
- 3 Double-click the **Temporal Properties** block that contains the examples.

The sections that follow describe each example in the Temporal Properties sublibrary in detail.

### Synchronize the Output with the Input

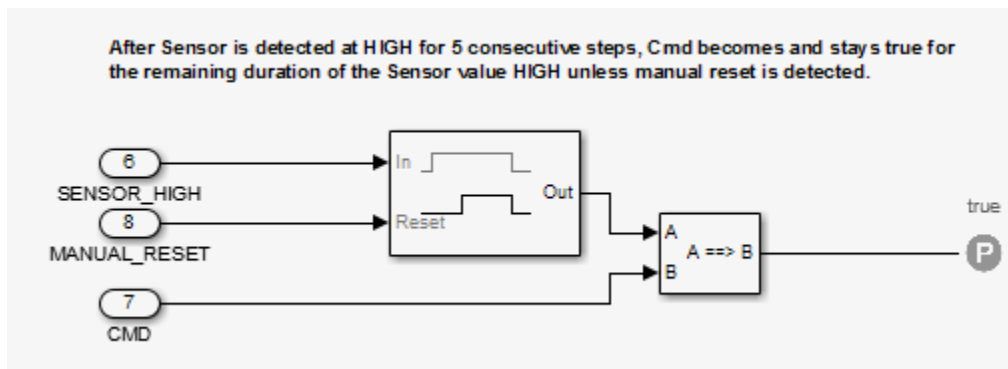
When the input In1 equals ACTIVE, the input In2 is set to INACTIVE after five time steps.



### Make a Signal Inactive After a Delay

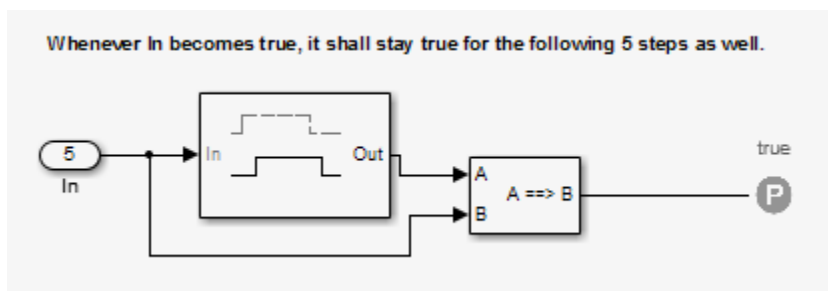
In this example, after five consecutive time steps where the `SENSOR_HIGH` input is true, the `CMD` signal becomes true. `CMD` is true as long as `SENSOR_HIGH` is true, unless the block is reset by the `MANUAL_RESET` signal.





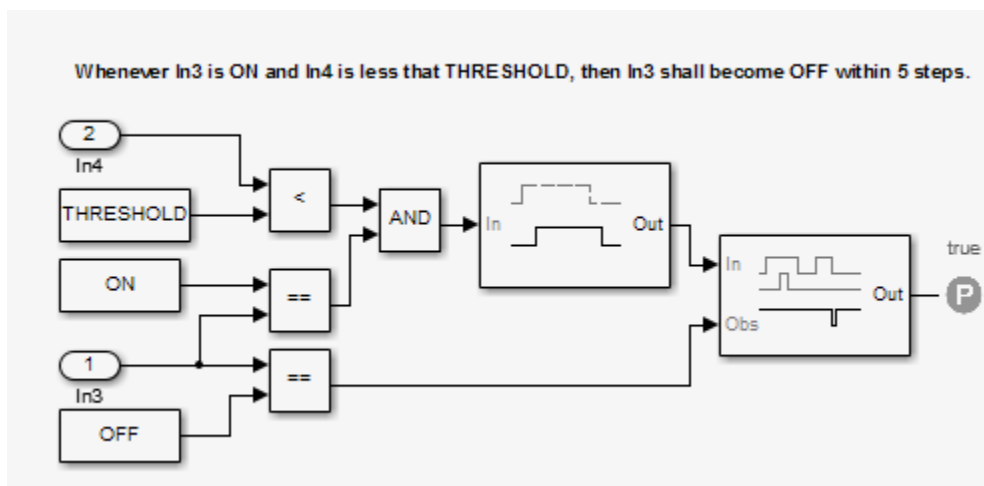
### Extend a True Signal

In this example, after the input becomes true, the output becomes true for the number of time steps specified in the Detector block, in this case, 5. The input remains true for 5 time steps as well.



### Test the Input Against a Specified Threshold

When the input In3 equals ON and the input In4 is less than the constant THRESHOLD, In3 is set to OFF within five time steps.



### See Also

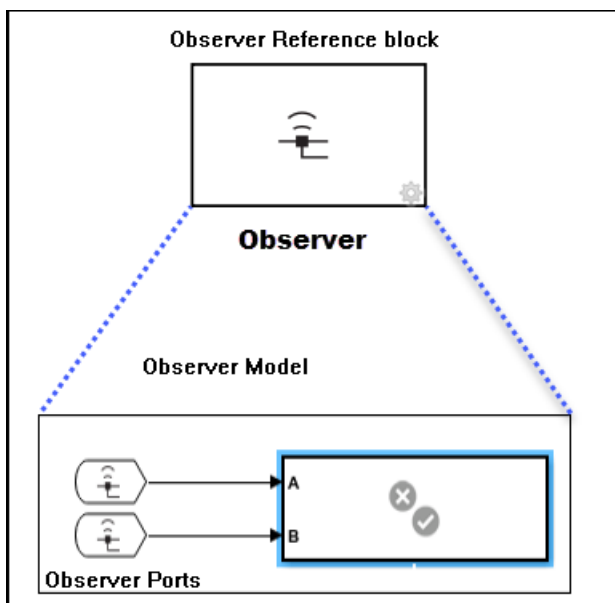
### More About

- “Debounce Temporal Properties” on page 12-43
- “Power Window Controller Temporal Properties” on page 12-46

## Isolate Verification Logic with Observers

You can isolate the verification logic in a model by using Observer Reference blocks. Use Observer Reference blocks when you want to keep the verification logic separate from your design model. When you use an Observer Reference, you can make changes to the Observer model without changing the design model. Using Observer Reference blocks can help you specify properties or requirements early in the model design or across multiple model designs. The Observer Reference block also allows you to:

- Model design requirements as properties and prove them using Simulink Design Verifier.
- Establish baseline results based on the captured output and detect model regressions.
- Generate test cases for functional design requirements using custom test objectives.

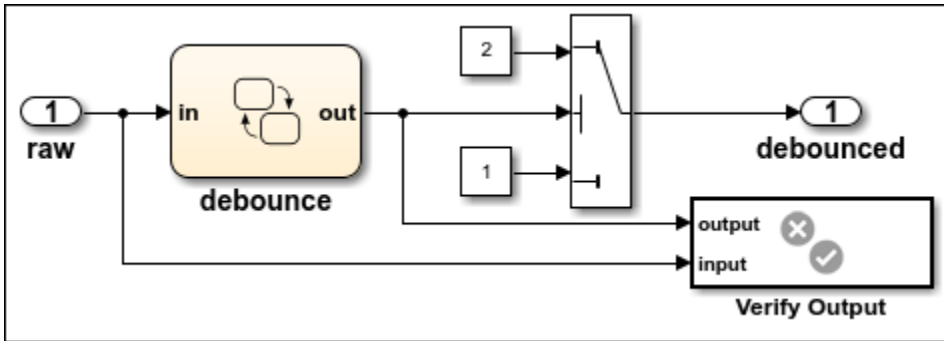


Double-click an Observer Reference block to open the Observer model. Observer Reference blocks can only be at the top level of a system model and do not have input ports. For more information, see “Access Model Data Wirelessly by Using Observers” (Simulink Test).

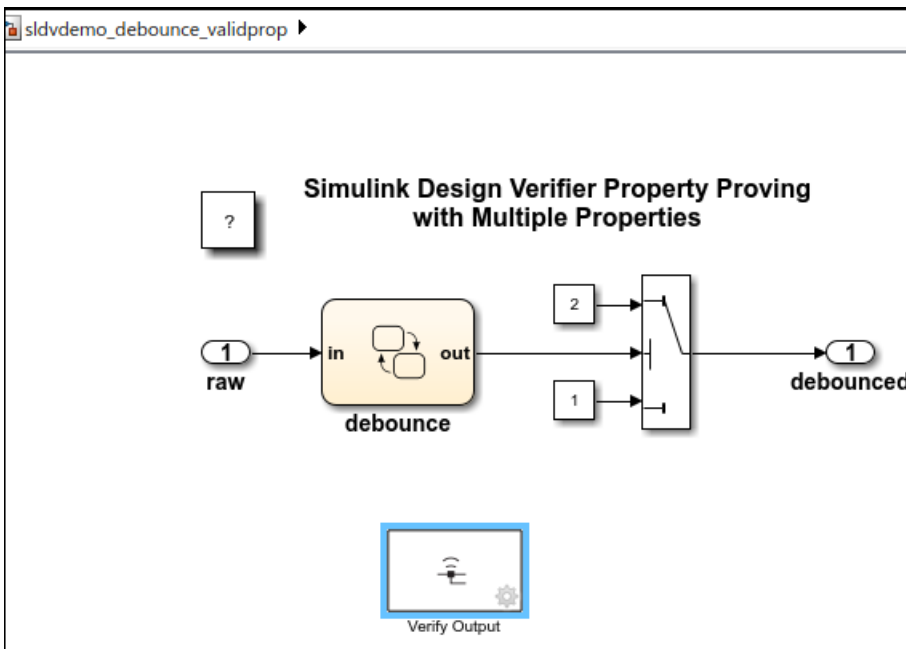
### Replace a Verification Subsystem with an Observer Reference Block

When authoring custom verification objectives, the Observer Reference block can be used in place of the Verification Subsystem block. The Observer Reference block references a separate verification model called the Observer model that you use to verify your system model. Converting a Verification Subsystem block to an Observer Reference block can declutter a system model. To convert a Verification Subsystem block to an Observer Reference block, right-click the verification subsystem and select **Observers > Move selected block to Observer > New Observer**. This operation cannot be undone. This action adds an Observer Reference block to your system model and opens the Observer model. You must save the Observer model in a writable folder on the MATLAB path.

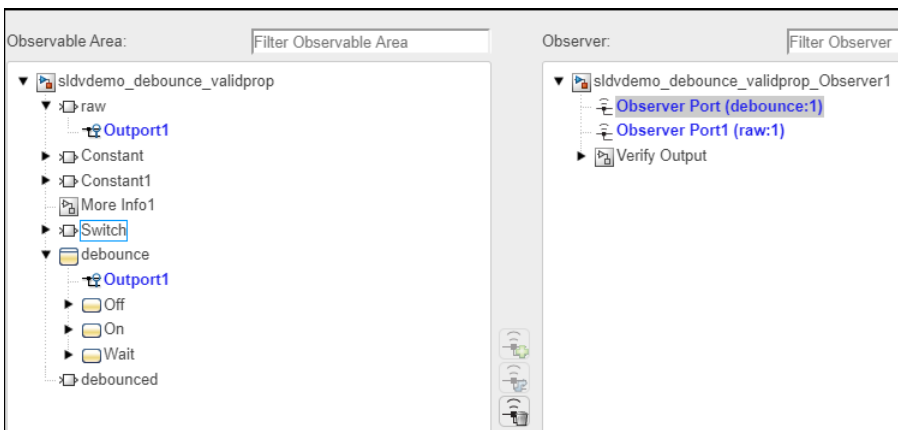
Consider the case where the model `sldvdemo_debounce_validprop` contains the Verification Subsystem block `Verify Output`.



By converting the subsystem to an Observer Reference block, you remove the signals that connect subsystem to the system model while preserving the ability to test the integrity of the system.



The two signals, `debounce` and `raw`, are automatically mapped to two Observer Port blocks in the Observer model, `sldvdemo_debounce_validprop_Observer1`.



You can verify the properties of `sldvdemo_debounce_validprop` without making any changes to the design model.

## Report on Observer Reference Blocks

If your model includes an Observer Reference block, the Simulink Design Verifier analysis report shows the property proving, test case generation, and design error information for the Observer Reference blocks in the **Observer Model(s)** subsection and the design model information in the **Design Model** subsection. For more information, see “Review Results” on page 13-35.

## Limitations

- Simulink Design Verifier does not support:
  - Observer models that include Model blocks
  - Observer models observing a constant signal
  - Applying block replacement rules to Observer models
  - Observer models that run at a different base rate than the design model
  - Tuning the parameters inside an Observer model
  - Test generation for code generated by Embedded Coder for models that contain Observer Reference blocks
  - Observer model that uses variable-step solver settings to execute the analysis

---

**Note** If an Observer model includes any of the restrictions in this list, the software ignores the corresponding Observer Reference block during the analysis.

---

- Simulink Design Verifier analysis returns an error when you:
  - Analyze standalone Observer models
  - Perform subsystem extraction on an Observer Reference block

## See Also

Observer Port (Simulink Test) | Observer Reference (Simulink Test) | “Use Observer Reference Blocks for Property Proving Analysis” on page 12-70 | “Use Observer Reference Block for Test Case Generation” on page 7-130

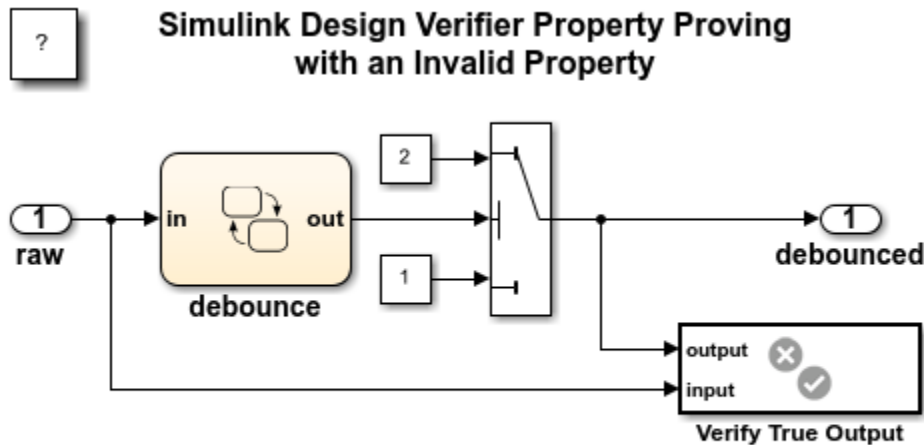
## External Websites

- “Access Model Data Wirelessly by Using Observers” (Simulink Test)

## Property Proving with an Invalid Property

This example shows how to find an invalid property using Simulink® Design Verifier™ property proving analysis. It attempts to prove that when the sum of the current and six previous input values is greater than 6, the output equals 2. In this case, the property is invalid because a single large input value (e.g. 255) causes the sum to be greater than 6. Simulink Design Verifier produces a counterexample that demonstrates the violation.

```
open_system('sldvdemo_debounce_falseprop');
```



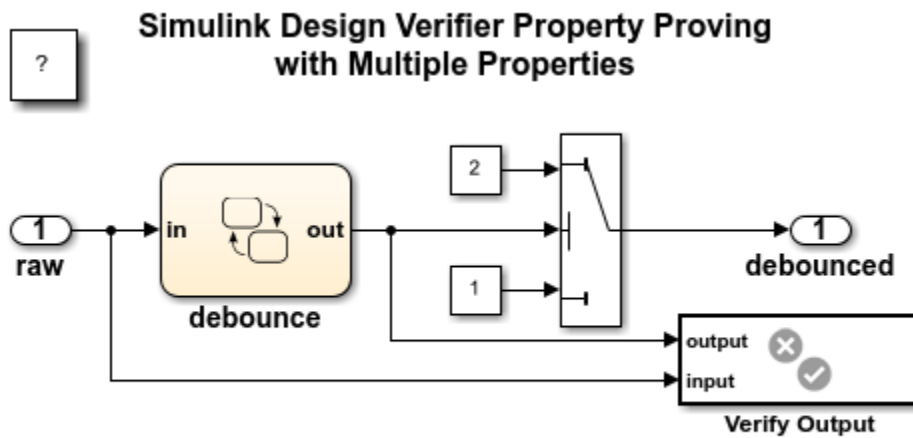
Copyright 2006-2023 The MathWorks, Inc.

## Property Proving with Multiple Properties

This example shows how to perform a property proving analysis with multiple properties. The model is configured for the analysis to attempt to prove that:

- When the current and six previous input values are true, the output will be true.
- When the current and six previous input values are false, the output will be false.

```
open_system('sldvdemo_debounce_validprop');
```

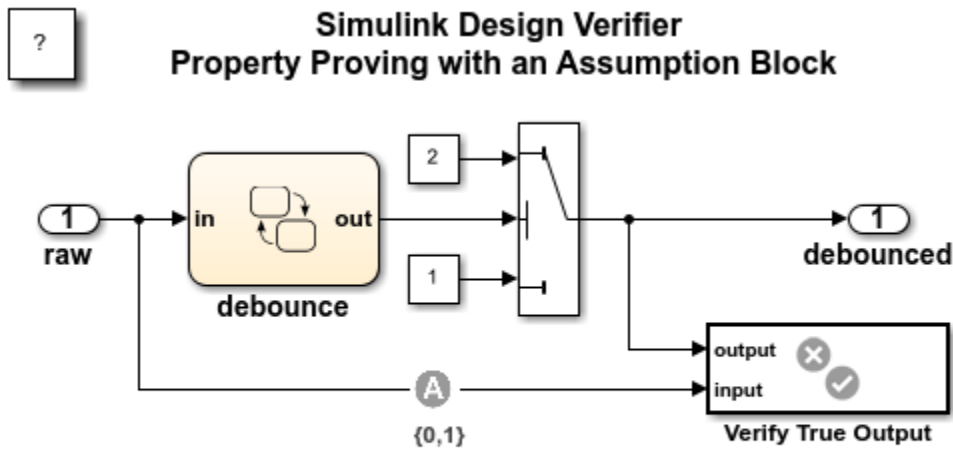


Copyright 2006-2023 The MathWorks, Inc.

## Property Proving with an Assumption Block

This example shows how to perform a Simulink® Design Verifier™ property proof using a Proof Assumption block. It attempts to prove that when the sum of the current and six previous input values is greater than 6, the output equals 2. The model includes a Proof Assumption block that constrains the input to be 0 or 1. Simulink Design Verifier searches for violations of 20 or fewer time steps. It is unable to find a violation because the property is valid under the assumption.

```
open_system('sldvdemo_debounce_assumeblk');
```



Copyright 2006-2023 The MathWorks, Inc.



## Property Proving Workflow for Cruise Control

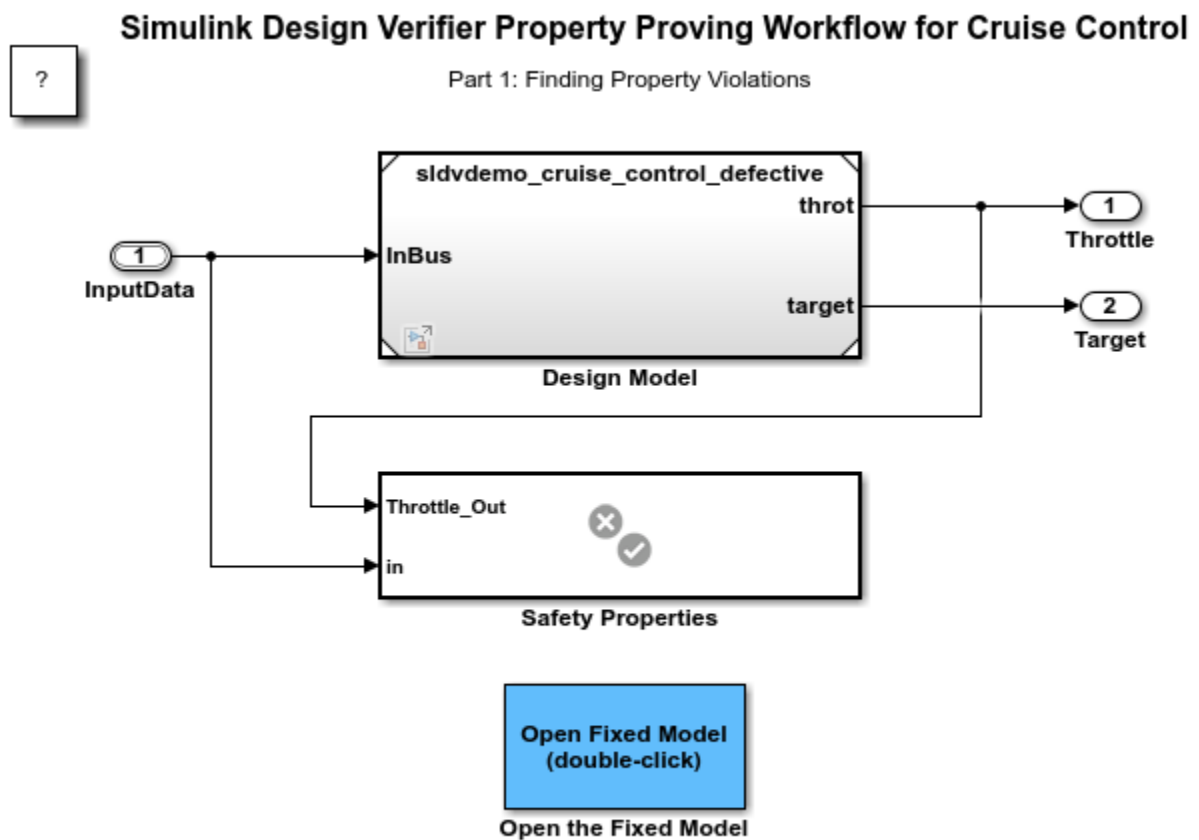
This example shows how to find a property violation by using Simulink® Design Verifier™ property proving analysis. You model safety requirements as properties and then verify the design model against requirements.

When you perform property proving analysis, Simulink Design Verifier generates a counterexample that you use to debug the property violation.

### Step 1: Open the Model

The `sldvdemo_cruise_control_verification` model contains a model reference to the `sldvdemo_cruise_control_defective` design model. The design model is a cruise control system that consists of a PI Controller that computes the throttle output based on the difference between the actual and target speed.

```
open_system('sldvdemo_cruise_control_verification');
```

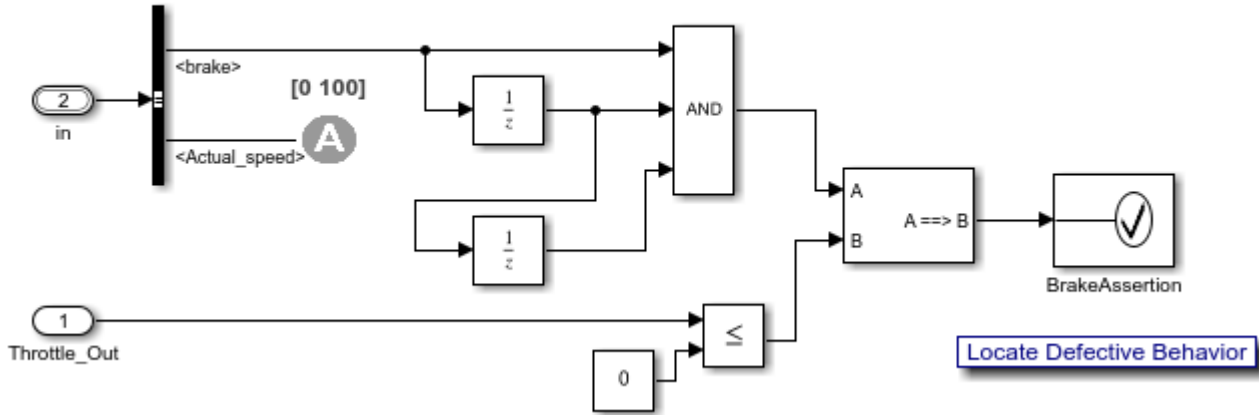


Copyright 2006-2020 The MathWorks, Inc.

The safety properties for the throttle output are modeled in the Safety Properties verification subsystem by the Assertion block.

```
open_system('sldvdemo_cruise_control_verification/Safety Properties');
```

Property: When the brake is applied for three consecutive steps, the throttle goes to zero.

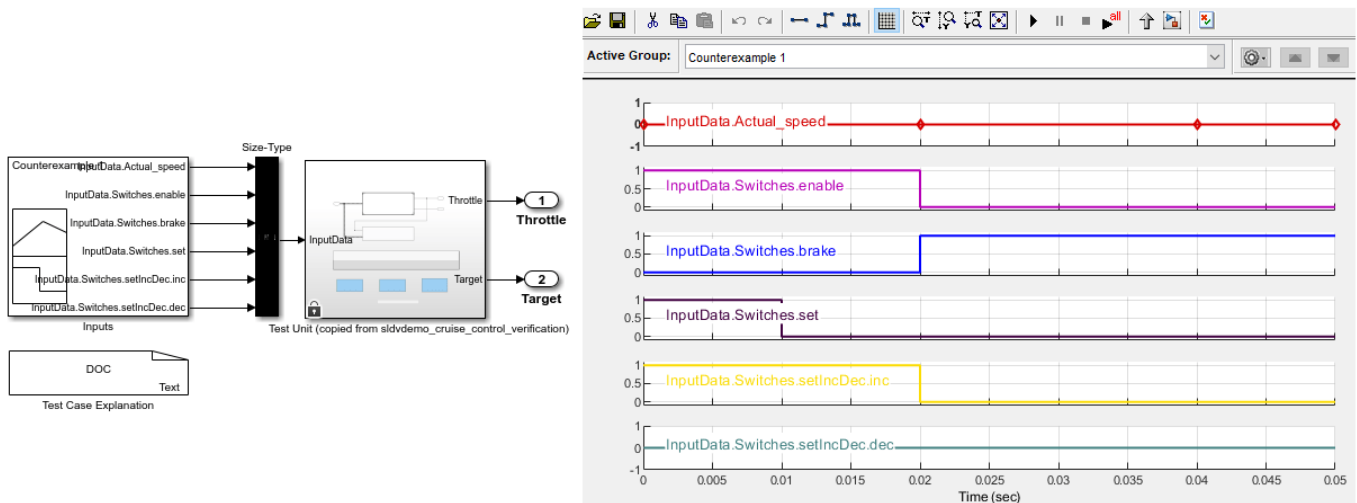


### Step 2: Perform Property Proving Analysis

On the **Design Verifier** tab, click **Prove Properties**.

After the analysis completes, the Results Summary window reports that one objective was falsified.

The harness model is generated and the Signal Builder dialog box opens and displays the counterexample.



### Step 3: Simulate the Counterexample to Replicate the Error

In the Signal Builder dialog box, click the **Start simulation** button ▶ .

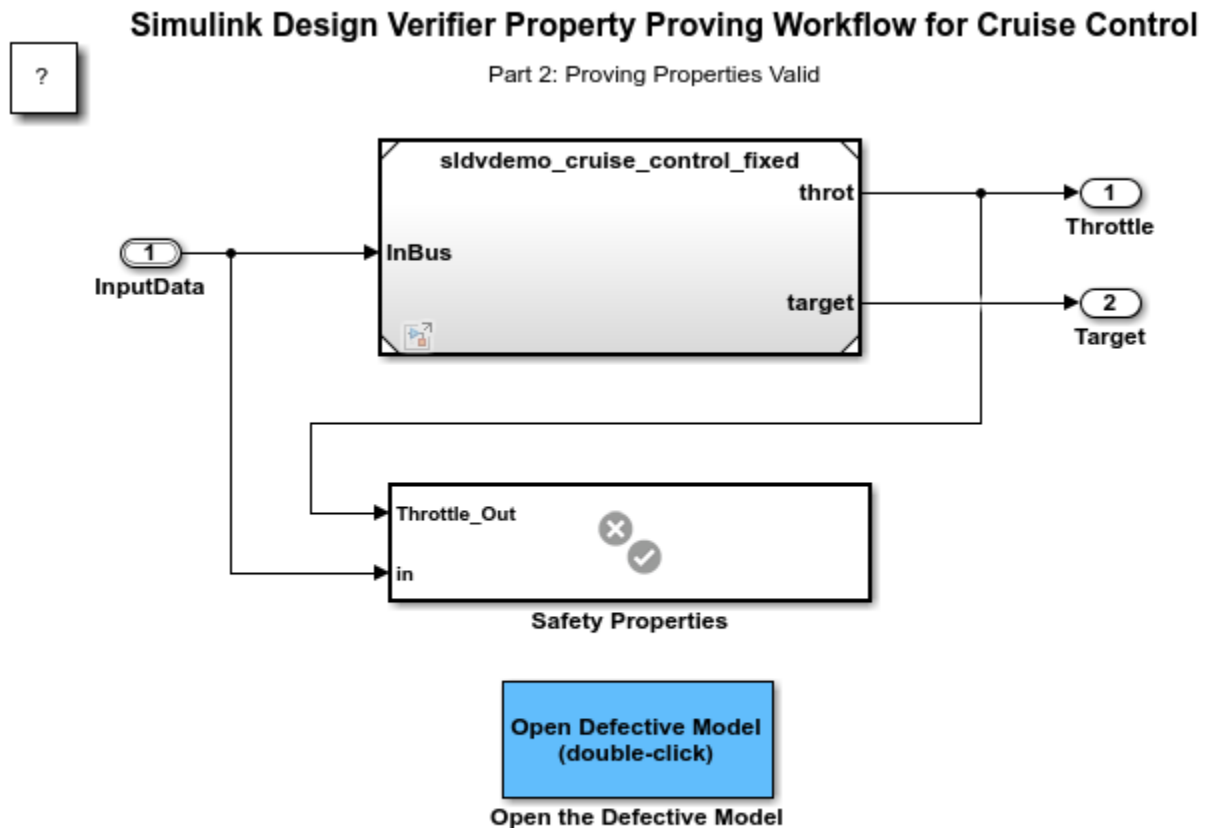
The Diagnostic Viewer window displays an error stating that the simulation was terminated because an assertion occurred at time 0.04.

Optionally, you can debug the property violation by using the Model Slicer. For more information, see “Debug Property Proving Violations by Using Model Slicer” on page 12-55.

#### Step 4: Open the Fixed Model

The erroneous behavior exhibited by the counter example is fixed in the `sldvdemo_cruise_control_verification_fixed` model.

```
open_system('sldvdemo_cruise_control_verification_fixed');
```



Copyright 2006-2020 The MathWorks, Inc.

In the property proving workflow, you may be required to redesign the system and/or redefine the property and perform such iterations.

Open the referenced model `sldvdemo_cruise_control_fixed` and open the Controller subsystem. In this subsystem, the updated design model resets the throttle output when Active Control is active.

On the **Design Verifier** tab, click **Prove Properties**. After the analysis completes, the Results Summary window reports that the objective is valid.

#### See Also

- “Workflow for Proving Model Properties” on page 12-4

- “Prove System-Level Properties Using Verification Model” on page 12-20

## Property Proving Workflow for Fixed-Point Cruise Control with Block Replacements

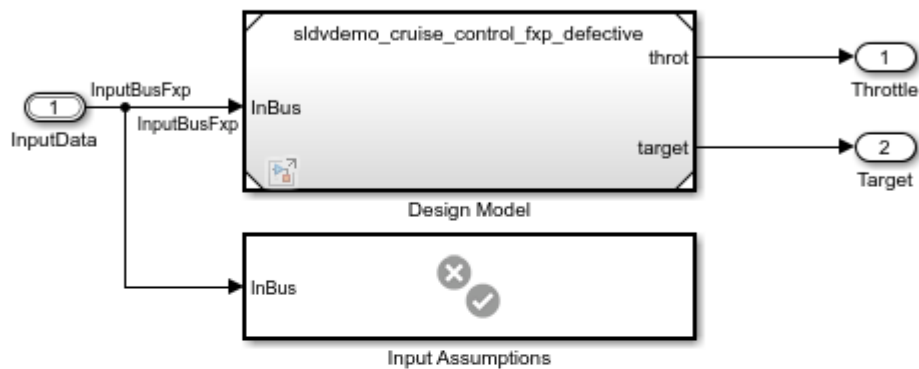
This example shows how to prove properties in a fixed-point cruise control algorithm. It references the design model using model reference so that the original design model is unchanged. A block replacement rule specifies the property that checks if an overflow is possible. The verification subsystem specifies an assumption on the range of the speed input during property proving. This model configures Simulink Design Verifier to apply a block replacement to the Sum block that feeds the output of the fixed-point PI Controller in the referenced model and return a counterexample that demonstrates an overflow.

```
open_system('sldvdemo_cruise_control_fxp_verification');
```



### Simulink Design Verifier Property Proving Workflow for Fixed-Point Cruise Control with Block Replacements

Part 1: Finding Property Violations



**Sum Replacement Rule  
(double-click)**

Open the Sum Replacement Rule

**Sum Replacement Contents  
(double-click)**

Open the Sum Replacement Contents

**Open Fixed Model  
(double-click)**

Open the Fixed Model

Copyright 2006-2019 The MathWorks, Inc.

## Property Proving Using MATLAB Function Block

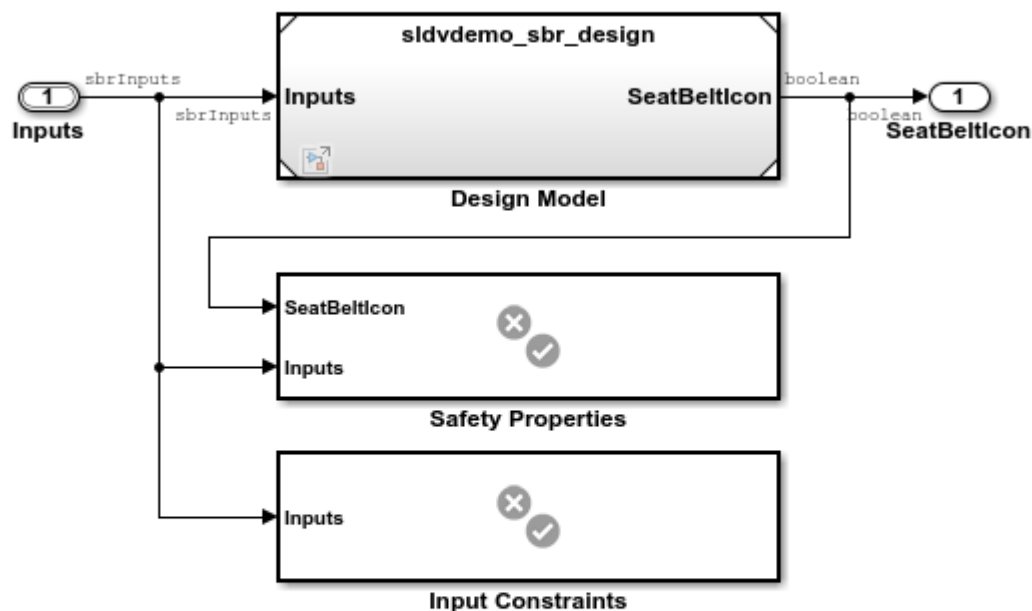
This example shows how to verify the seat belt reminder design model. The Safety Properties block below it contains a property specified in MATLAB® that indicates when the icon should be active. Simulink® Design Verifier™ analyzes the design model and safety property to prove correctness or to identify counterexamples. In this model, the property is violated because the design implicitly assumes that the KEY input starts at 0 and changes by increments of 1.

```
open_system('sldvdemo_sbr_verification');
```



### Simulink Design Verifier Property Proving Using MATLAB Function Block

Part 1: Finding property violations



Open Fixed Model  
(double-click)

Open Fixed Model

Copyright 2006-2023 The MathWorks, Inc.

## Property Proving Using MATLAB Truth Table Block

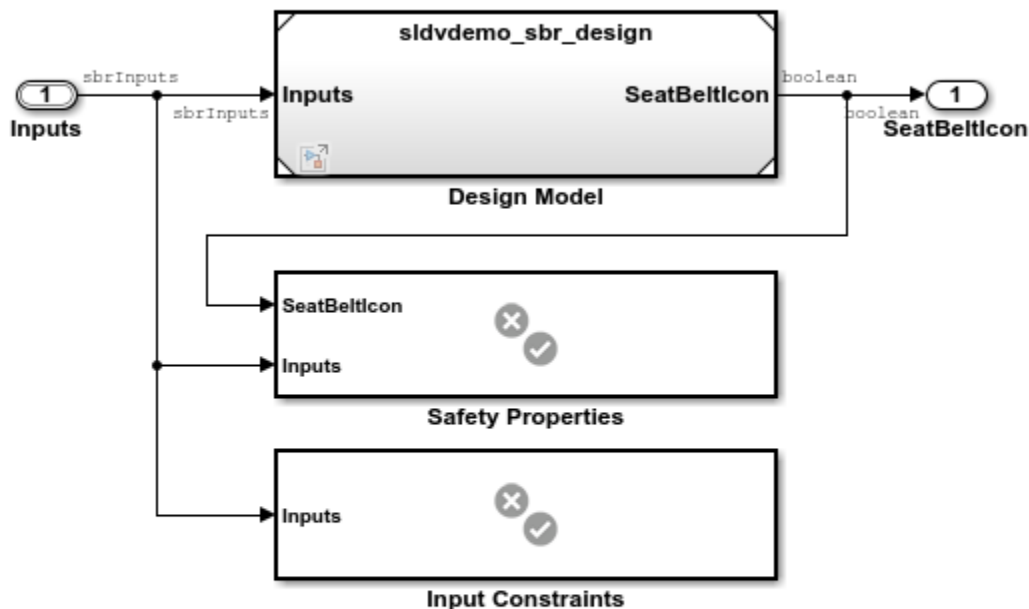
This example shows how to verify the seat belt reminder design model referenced in the top block above. The Safety Properties block below it contains a property specified in MATLAB Truth Table that indicates when the SeatBeltIcon output should be active. Simulink Design Verifier analyzes the design model and safety property to prove correctness or to identify counterexamples. In this model, the property is proven under the explicit assumption that the KEY input starts at 0 and changes by increments of 1.

```
open_system('sldvexSBRVerificationTruthTableFixedExample');
```

### Simulink Design Verifier Property Proving Using MATLAB Truth Table Block



Part 2: Proving properties valid



Open Defective Model  
(double-click)

Open Defective Model

Copyright 2013-2023 The MathWorks, Inc.

## Property Proving Workflow for Thrust Reverser

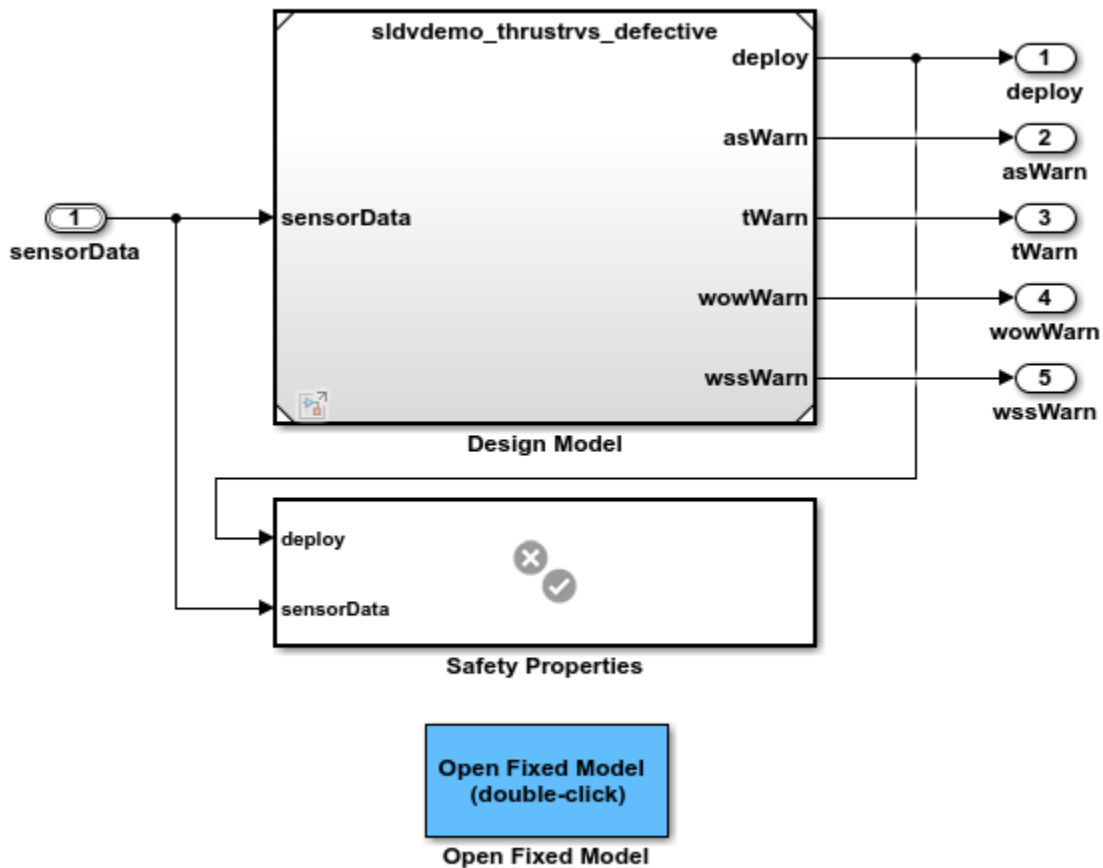
This example shows how to verify safety properties in a thrust reverser design model. The Properties block below it contains four safety properties. Simulink® Design Verifier™ analyzes the design model and safety properties to prove correctness or to identify counterexamples. The use of model referencing eliminates the need to add verification content to the design model, allowing the verification content to exist independently from the design.

```
open_system('sldvdemo_thrustrvs_verification');
```



### Simulink Design Verifier Property Proving Workflow for Thrust Reverser

Part 1: Finding property violations





## Debounce Temporal Properties

This example shows how to model temporal system requirements for property proving and test case generation using Simulink® Design Verifier™ Temporal Operator blocks.

### Temporal Operators

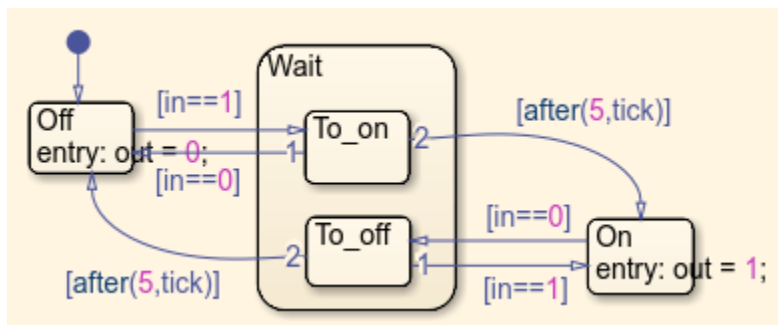
The Simulink® Design Verifier™ library provides three basic temporal operator blocks can be used to model temporal properties. The intent of the temporal operators is to support the specification of temporal requirements, such that the modeled property has a closer co-relation to the actual textual requirement. These blocks are low-level building blocks for constructing more complex temporal properties.

### Debounce Model and Requirements

Consider a debounce logic that debounces between values of 0 and 1 based on the input holding a value for a fixed number of time steps.

The debounce functionality is captured in the containing Stateflow® chart.

```
open_system('sldvdemo_debounce_to')
open_system('sldvdemo_debounce_to/debounce')
```



Consider two requirements of the debounce model that you would like to verify.

#### Requirement 1:

Whenever the input equals 1 for more than 6 steps, the output shall be equal to 2.

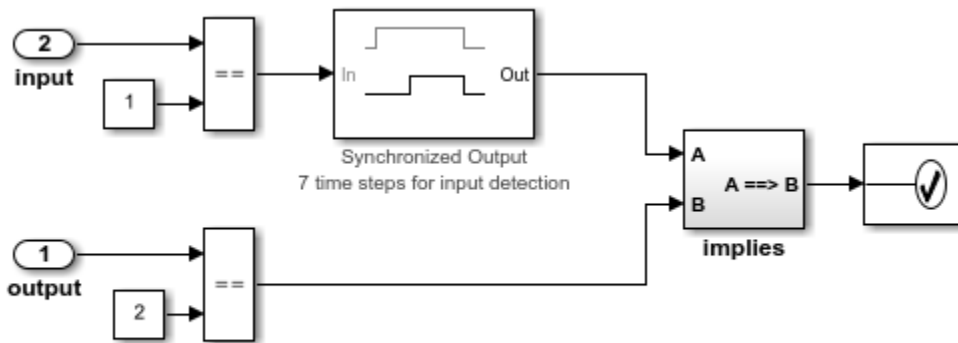
#### Requirement 2:

Whenever the input becomes 0 for more than 5 steps after the output was 2, the output shall equal 1 as long as the input stays at 0.

### Property Specification

For specifying **Requirement 1**, you first represent the constraint that **input equals 1 for more than 6 steps**. This can be captured by the Detector block from the Temporal Operator Blocks Library. On detecting that the input value is 1 for 7 (or more than 6) time steps, you want to check that the output equals 2 as long as input stays equal to 1 after the detection. Use the "Synchronized" option of the Detector block followed by an Implies block to capture this.

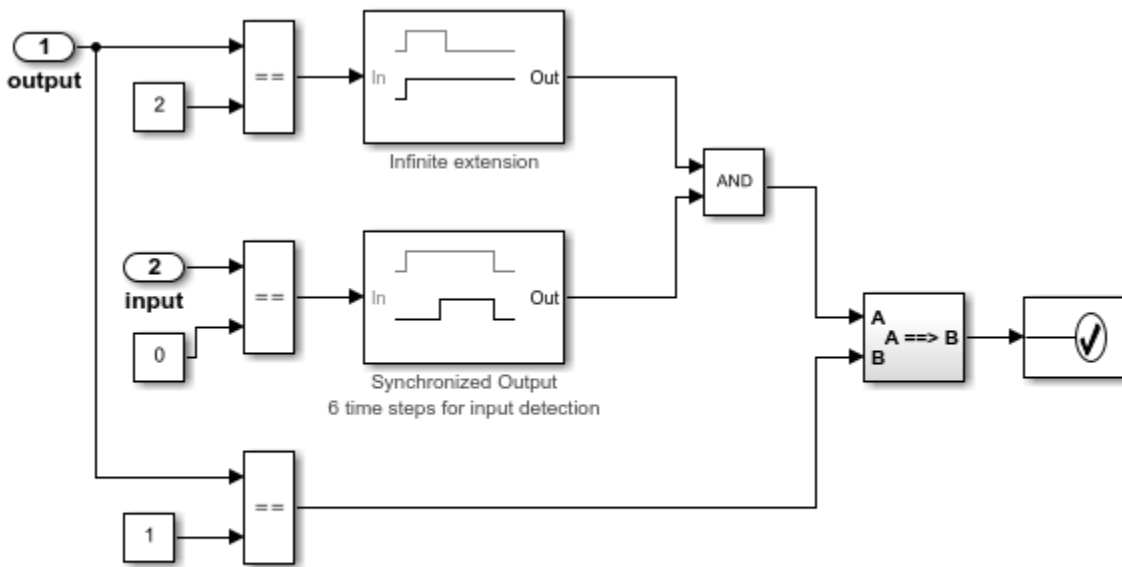
```
open_system('sldvdemo_debounce_to/Verify True Output1')
```



Whenever input value stays 1 for more than 6 steps then the output shall be 2.

Multiple temporal operator blocks can be combined to construct more complex temporal properties. Consider **Requirement 2**.

```
open_system('sldvdemo_debounce_to/Verify True Output2')
```



Whenever the input becomes 0 for more than 5 steps after the output was 2, the output shall equal 1 as long as the input stays at 0.

For illustration, this requirement is broken down roughly into three pieces of interest:

- 1 **After the output was 2:** This is an enabling condition for your property. While checking the rest of the constraints, you want to know if this condition was true at some point in the past. This type of an enabling condition is typically followed by an Extender (either "Finite" or "Infinite") that, in combination with other constraints, might form the first input to your implication.
- 2 **The input becomes 0 for more than 5 steps** and check something **as long as input stays 0:** For the same reason as the first property, you use a Detector with "Synchronized" output ("Time steps for input detection" = 6).

- 3 The output shall equal 1:** This is the condition that you want to verify whenever the first two constraints hold. This is captured through a logical Implies block. Note that you cannot use Within Implies block here.

### Property Proving

Once the temporal requirements have been modeled, you can perform property proving on these using Simulink Design Verifier.

### Clean Up

To complete the example, close all the opened models.

```
close_system('sldvdemo_T0Blocks',0);  
close_system('sldvdemo_debounce_to',0);
```

## Power Window Controller Temporal Properties

This example shows how to model temporal system requirements in a power window controller model for property proving and test case generation using Simulink® Design Verifier™ Temporal Operator blocks.

### Temporal Operators

The Simulink® Design Verifier™ library provides three basic temporal operator blocks which can be used to model temporal properties. The intent of the temporal operators is to support the specification of temporal requirements, such that the modeled property has a closer correlation to the actual textual requirement. These blocks are low-level building blocks for constructing more complex temporal properties.

### Power Window Controller

The power window controller responds to the driver and passenger commands by giving the commands for moving the window up or down. It also responds to an obstacle and to reaching the end of the window frame in either direction.

Consider the following two requirements for the power window controller:

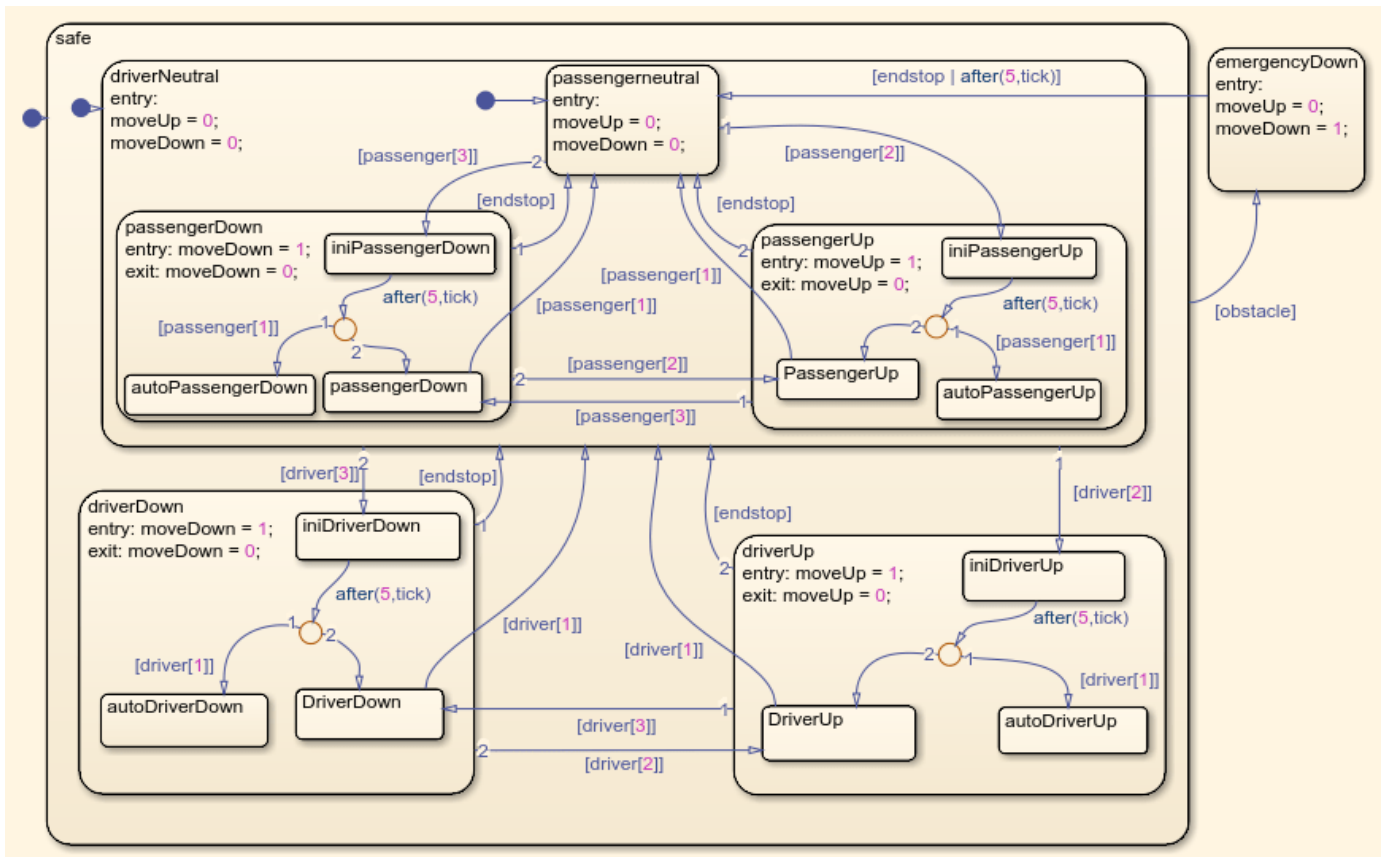
#### Requirement 1 (Obstacle Response)

Whenever an obstacle is detected, the controller shall give the down command for 1 second.

#### Requirement 2 (AutoDown feature)

If the driver presses the down button for less than 1 second, the controller keeps giving the down command until the end has been reached or the driver presses the up button.

```
%Model of the power window controller
open_system('sldvdemo_powerwindowController')
open_system('sldvdemo_powerwindowController/control')
```

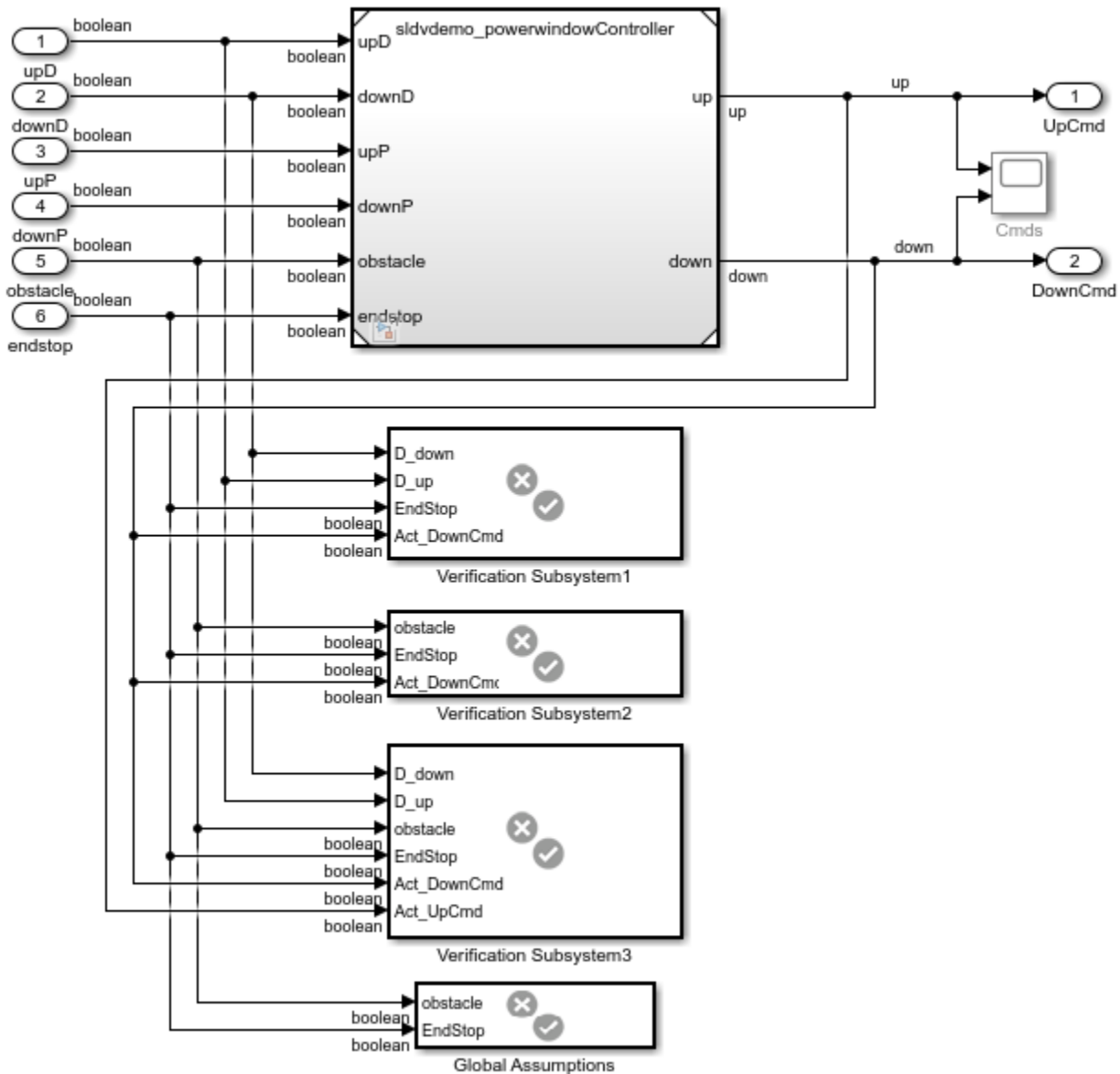


### Property Specification

The power window verification system is the top-level model that contains a model reference to the power window controller model specifying the controller behavior and the modeled requirements.

```
%Model of the top-level verification system
open_system('sldvdemo_powerwindow_vs')
```

### Power Window Controller Temporal Property Specification



Copyright 1990-2010 The MathWorks, Inc.

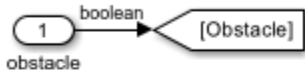
**Global Assumptions:** The power window controller is an open system. This makes the environment controlled inputs, obstacle and endstop (end of the window frame) to occur freely. To constrain the environment, add two global assumptions for your controller model.

- 1) The obstacle and the endstop inputs never become true at the same time.
- 2) The obstacle does not occur multiple times within the following 1-second interval.

For the temporal assumption on obstacle, use a Detector block with output type of "Delayed Fixed Duration" to capture the fixed duration of 1 second (5 time steps with 0.2 sample time).

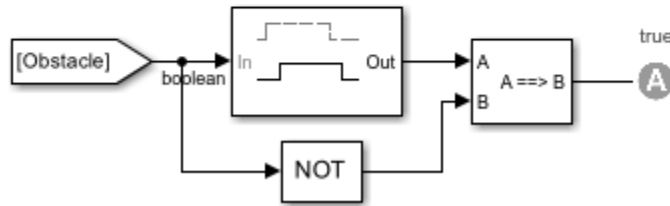
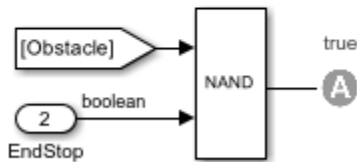
```
% Global Assumptions
open_system('sldvdemo_powerwindow_vs/Global Assumptions')
```

These assumptions using the 'Assumption' blocks apply globally to all property proofs



1. Obstacle and EndStop not true simultaneously

2. Obstacle shall not occur multiple times within the following 1 sec interval.

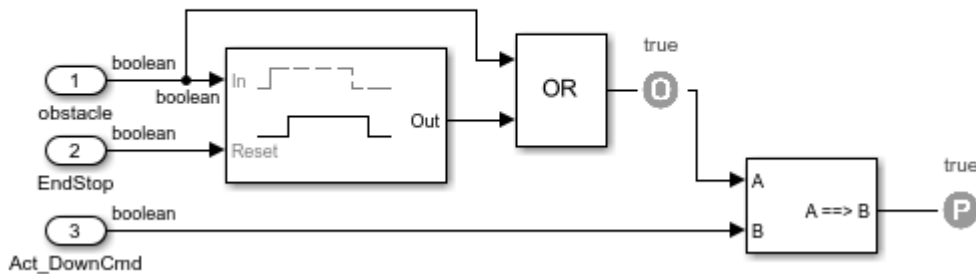


Now consider the first controller requirement for **Obstacle Response**.

```
% Obstacle Response
open_system('sldvdemo_powerwindow_vs/Verification Subsystem2')
```

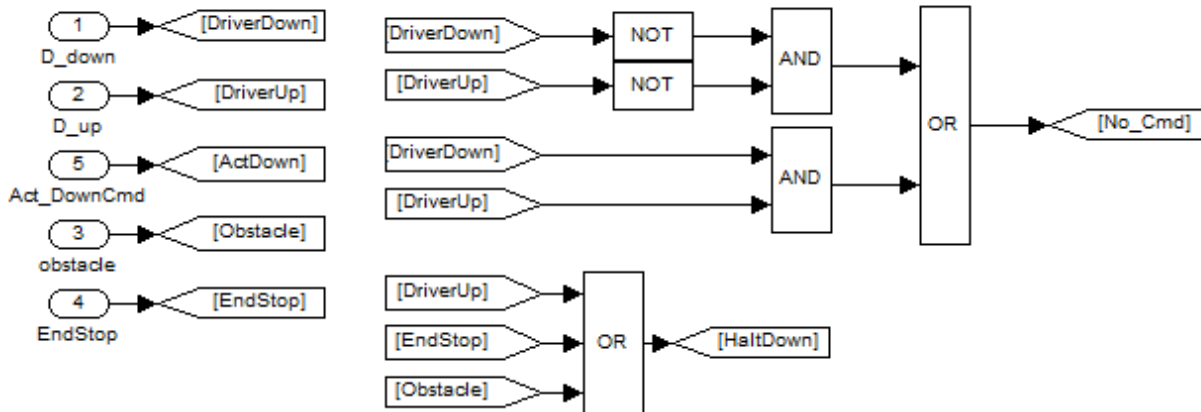
**Requirement:**

Whenever an obstacle is detected, then the down command shall be given for 1 second.



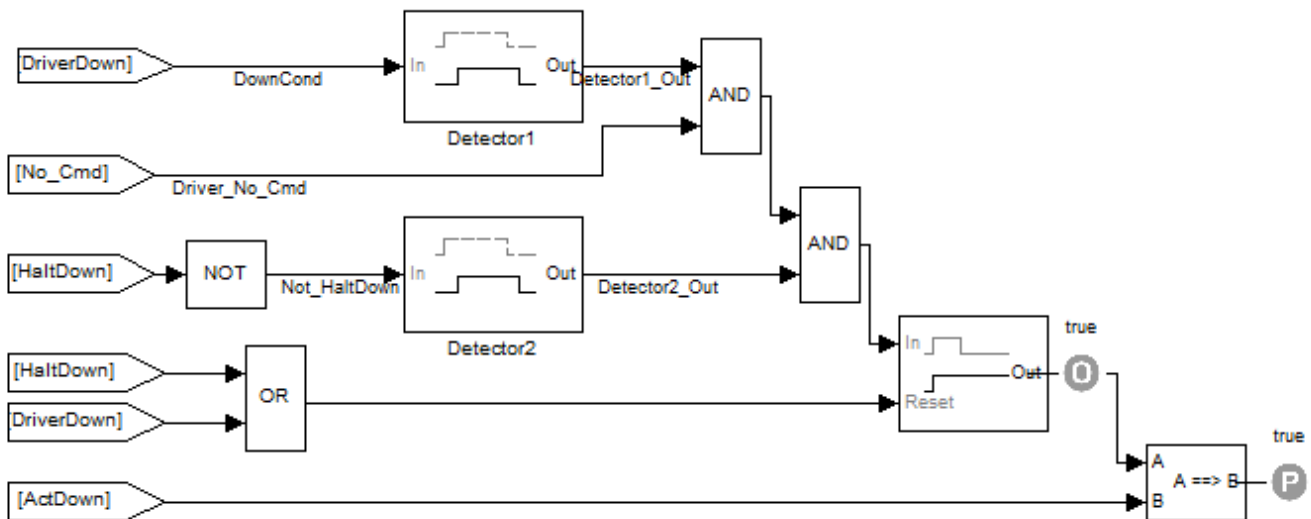
Here, use the Detector block with output type of "Delayed Fixed Duration" for the property specification. After detection of the obstacle, construct a fixed interval of 4 steps. Note that the input is not observed during the output construction phase for the Detector with "Delayed Fixed Duration" output type. In the case where the obstacle can occur freely in absence of the assumption, you might wish to observe all the intermediate occurrences of the obstacle. This can be achieved through an Extender block with "Finite" extension duration of 4 time steps.

Now consider the **AutoDown feature** of the power window controller.



**Requirement (Autodown)**

If the driver presses the down button for less than 5 steps, then the controller gives the down command as long as end has not been reached or the driver presses the up button.



For illustration, consider this property specification in smaller parts:

- 1 The first temporal duration of interest, "driver presses the down button for less than 1 second", is captured by Detector1. At sample rate of 0.2, the 1-second interval is broken down into 5 time steps. On detection of the down signal, Detector1 constructs a 5-step fixed temporal duration at its output, which you will subsequently use in combination with other constraints.
- 2 For the AutoDown feature, you know that the down signal cannot be pressed for more than 1 second, or 5 time steps. Thus, you want to ensure that both driver up and down are "true" or both are "false" in less than 5 steps after down is pressed. By taking the AND of this driver neutral and the Detector output, enforce the constraint that driver down can be pressed for any number of consecutive time steps less than 5.
- 3 You also need to ensure that, during this period, other signals such as obstacle, EndStop and DriverUp are not true, since these will take the controller out of responding to the down press. This is captured using Detector2 by enforcing that NOT(HaltDown) is true for 5 time steps.



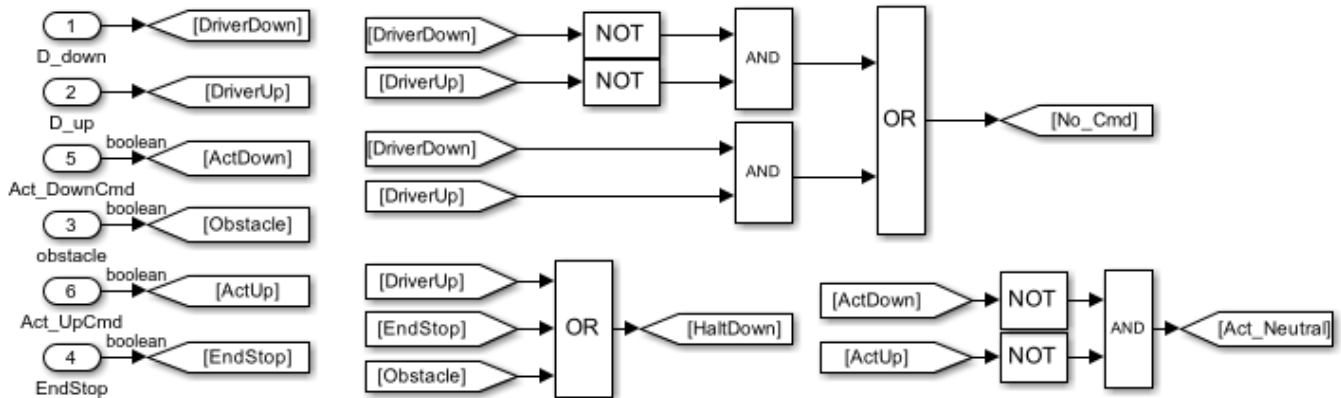
Detector2 has "Delayed Fixed Duration" output type. It also has "Time steps for input detection" = 5 and "Time steps for output duration" = 1.

- 4 Take the AND of these constructed durations.
- 5 For the AutoDown feature, you do not want to limit the number of time steps for which the controller gives the down command. You know that you want the controller to keep giving the down command as long as the driver does not press an up or down command again, or an obstacle or the physical end of the window frame is not hit. This behavior can be captured by the Extender block with "Infinite" extension period and an external reset signal that encodes the condition to end the extension.
- 6 The final piece is an Implies block that takes the temporal duration constructed as explained above and checks if the controller down command is true for every time step of this duration.

Once you have this initial property specification, you can use it for property proving with Simulink Design Verifier. You will get a counterexample for this property. The counterexample shows a scenario where the down command is given when the controller was in the emergency down state due to the response to an earlier detected obstacle. After you add a constraint to avoid this, you will get another counterexample: if the down button is pressed when previously the up command was being given, the AutoDown feature is disabled and the down command is given only as long as the down button is pressed. Looking at these counterexamples and observing the model, you can see a pattern that the AutoDown feature is enabled only when the controller is in a neutral state to begin with when the driver presses the down button.

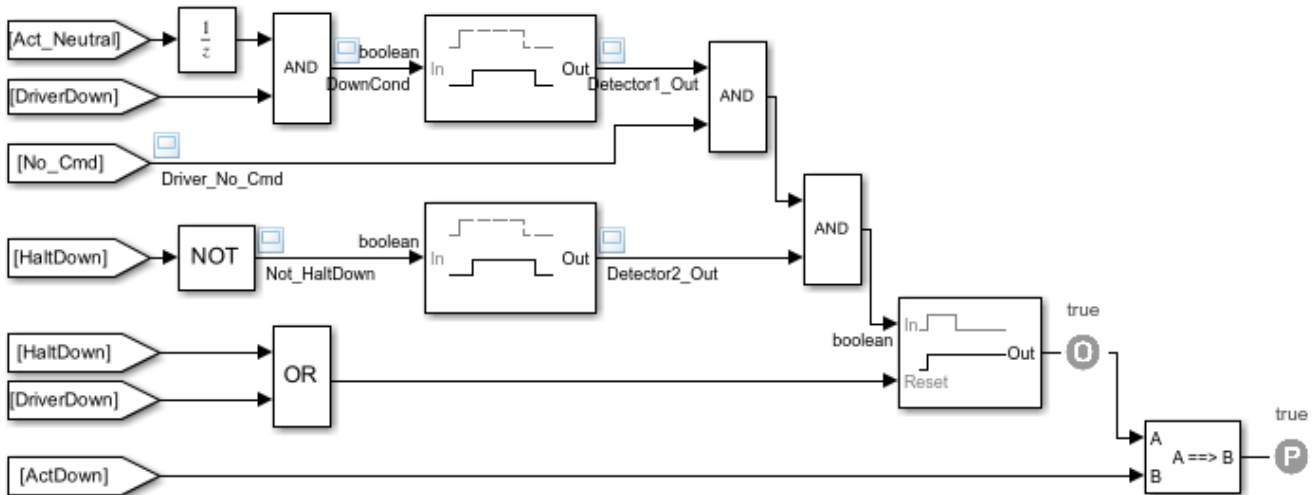
Incorporate this constraint by forcing the controller output to be neutral - neither up nor down command is true - as a precondition for the AutoDown property. This property is proven valid.

```
% Valid AutoDown
open_system('sldvdemo_powerwindow_vs/Verification Subsystem3')
```



**Requirement (Autodown)**

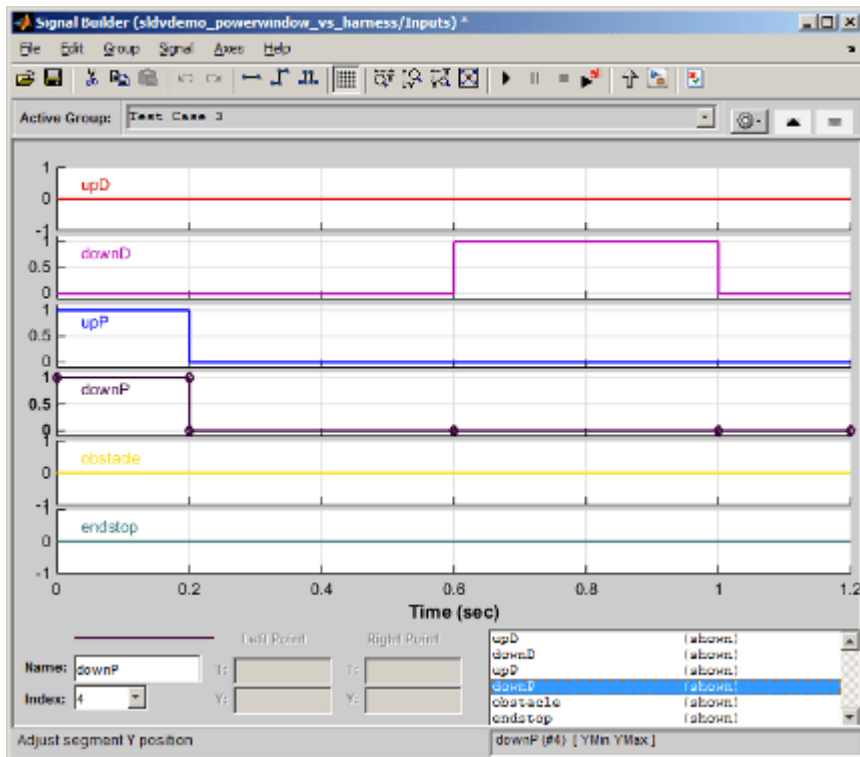
If the driver presses the down button for less than 5 steps, then the controller gives the down command as long as end has not been reached or the driver presses the up button.



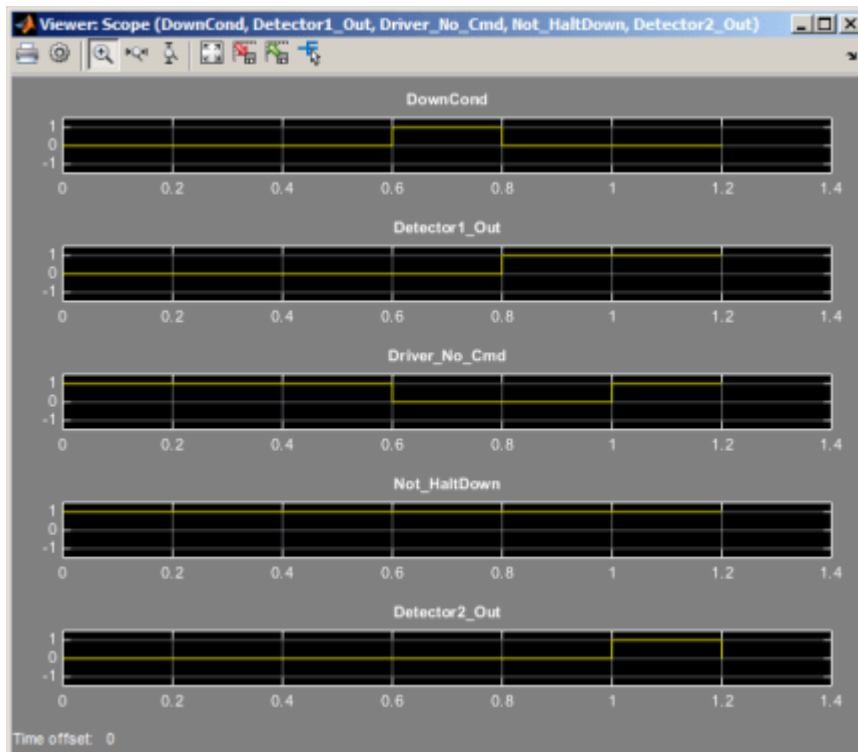
**Test Case Generation for Property Validation**

Once the properties are specified, in addition to property proving, you can run Simulink Design Verifier to automatically generate test cases that exercise various conditions in the property. This can be achieved by placing custom Test Objective blocks at appropriate locations in the property.

One such location to place a Test Objective block (with "true" value) is on the signal feeding into the first input of the Implies block (as shown in the above property). On running test generation, this Test Objective is satisfied and you will get a test case exercising the various constraints encoded in the property. Simulink Design Verifier can also create a test harness to simulate this test case. The Signal Builder block with relevant signals is shown below.



One can now simulate this test case, and see how the temporal durations are created in the property by placing a scope that displays the input and output values of the two Detector blocks and No\_Cmd.



Manually inspecting the test case values enables you to see if the specified property behaves as intended.

This Test Objective block helps in identifying a scenario where the property is valid while the Implies block is not trivially true. An Implies block is trivially true when its output is true because of its first input being false. When you get a test case satisfying this Test Objective, you know that there is at least one case where the first input to the Implies block is true.

This exercise can help you validate your property specifications by manually inspecting the test cases automatically generated by Simulink Design Verifier.

### **Clean Up**

To complete the example, close all the opened models.

```
close_system('sldvdemo_TOBlocks',0);  
close_system('sldvdemo_powerwindowController',0);  
close_system('sldvdemo_powerwindow_vs',0);
```

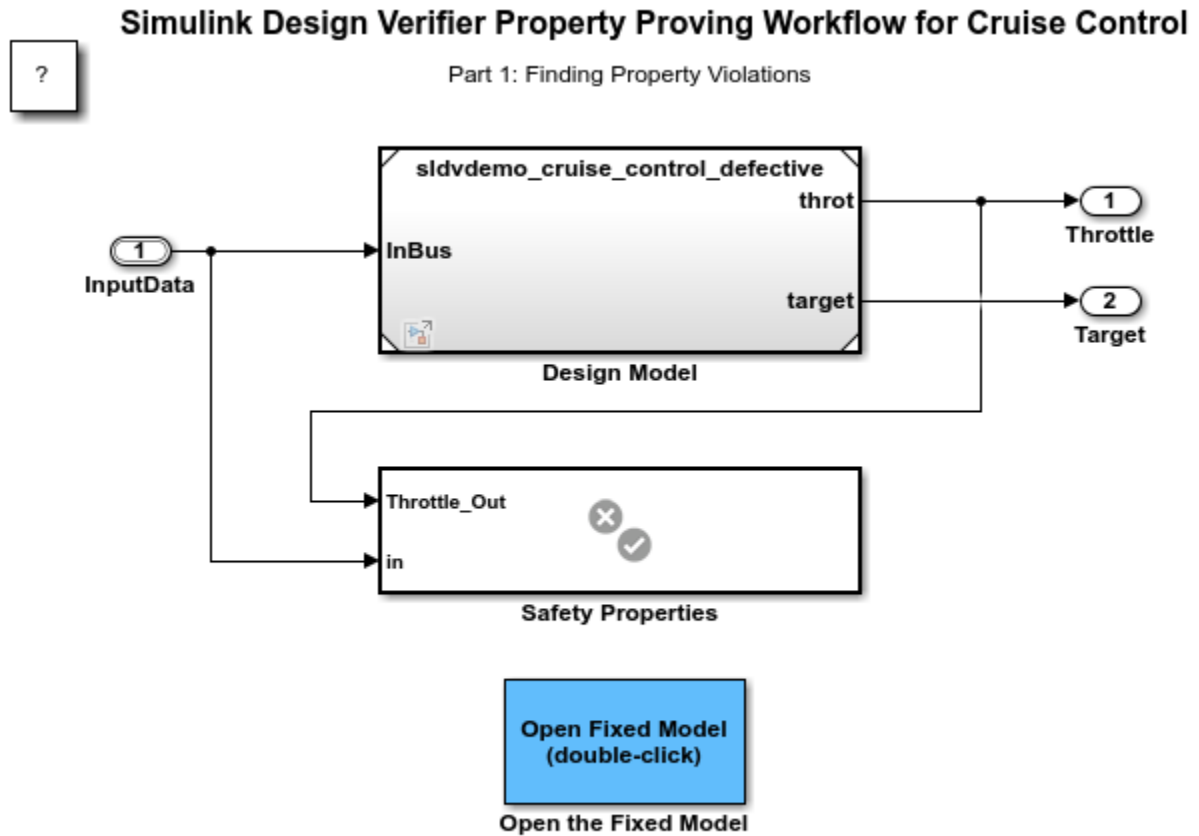
## Debug Property Proving Violations by Using Model Slicer

This example shows how to debug property proving violations by using Model Slicer. Consider the model `sldvdemo_cruise_control_verification`. This model contains an **Assertion** block.

The Verification subsystem **Safety Properties** models a property that should hold true for the design model. This subsystem contains an **Assertion** Block (BrakeAssertion) that verifies the property. Simulink Design Verifier Property Proving analysis will try to falsify the assertion. If Simulink Design Verifier is successful it will generate a counterexample falsifying the assertion. We can use Model Slicer to debug this falsified assertion.

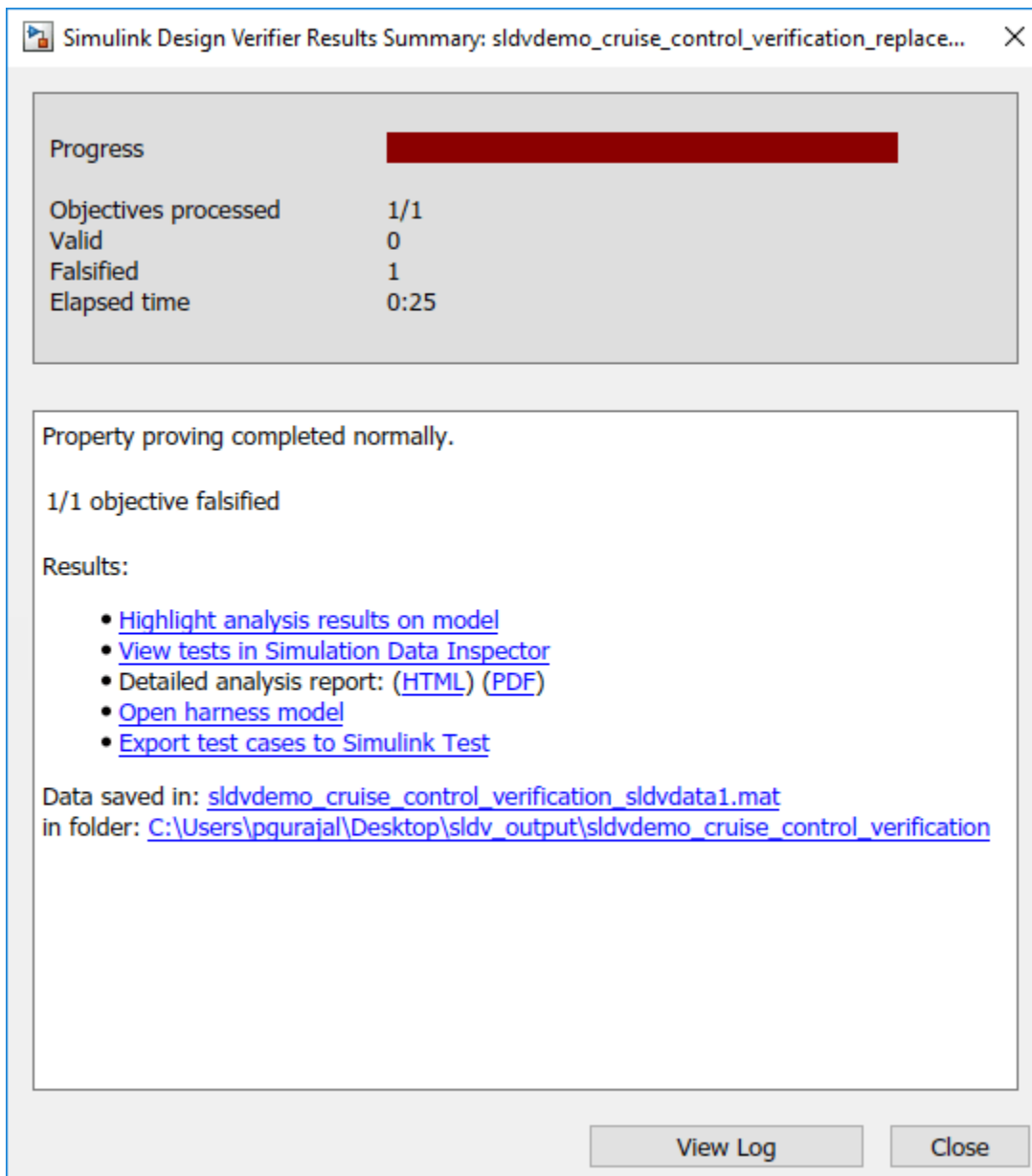
1. Open model `sldvdemo_cruise_control_verification`.

```
open_system ('sldvdemo_cruise_control_verification')
```

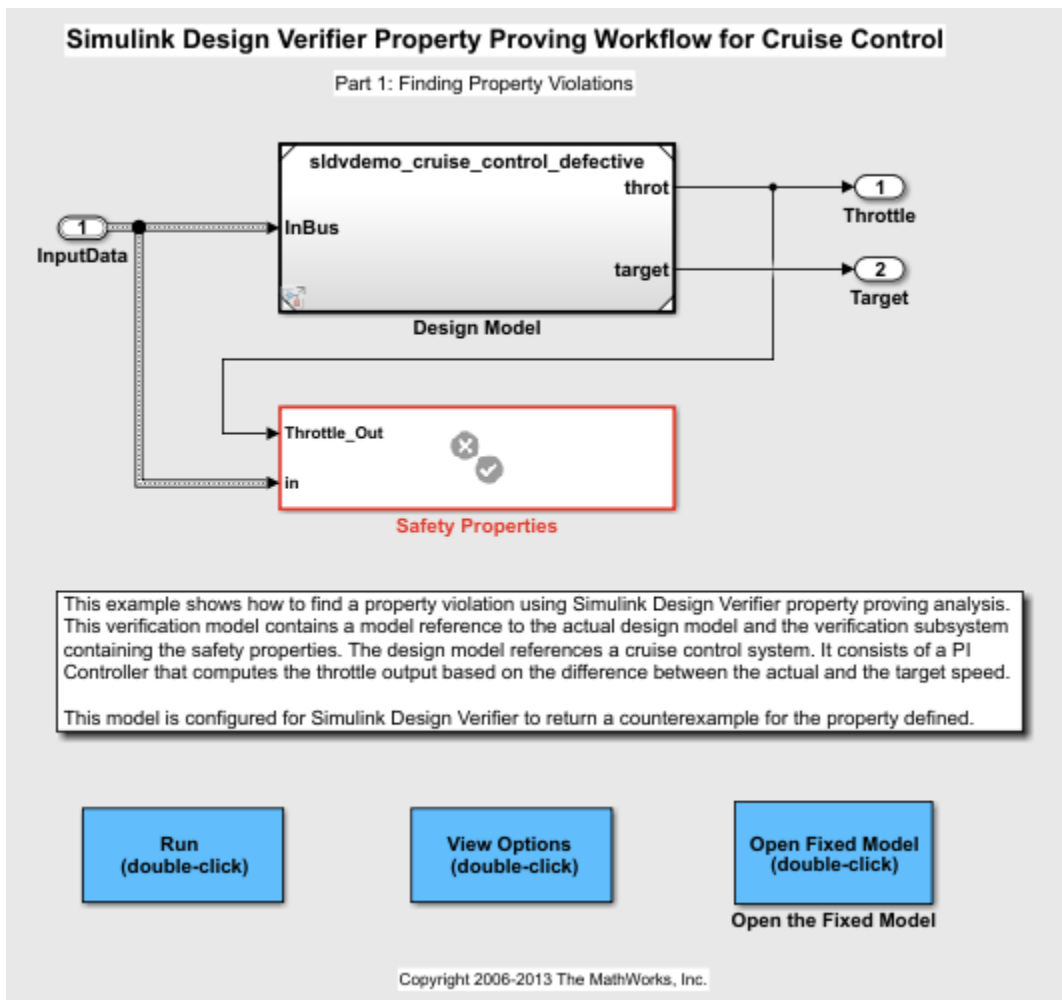


Copyright 2006-2020 The MathWorks, Inc.

2. Open Simulink Design Verifier by clicking on **Apps > Design Verifier**.
3. Click **Prove Properties**. Simulink Design Verifier analyses the model and displays the results in Results Summary window.



The model highlights the subsystem where the **Assertion** block is located.



4. Open Safety Properties subsystem and select the falsified **Assertion** block.

5. Click **Debug Using Slicer** from the toolstrip menu to debug the violation using Model Slicer. Alternatively, you can click **Debug** in the results Inspector window.

On Clicking either of the entry points the following setup is done on the model:

a. The Assertion block is added as a starting point for Model Slicer.

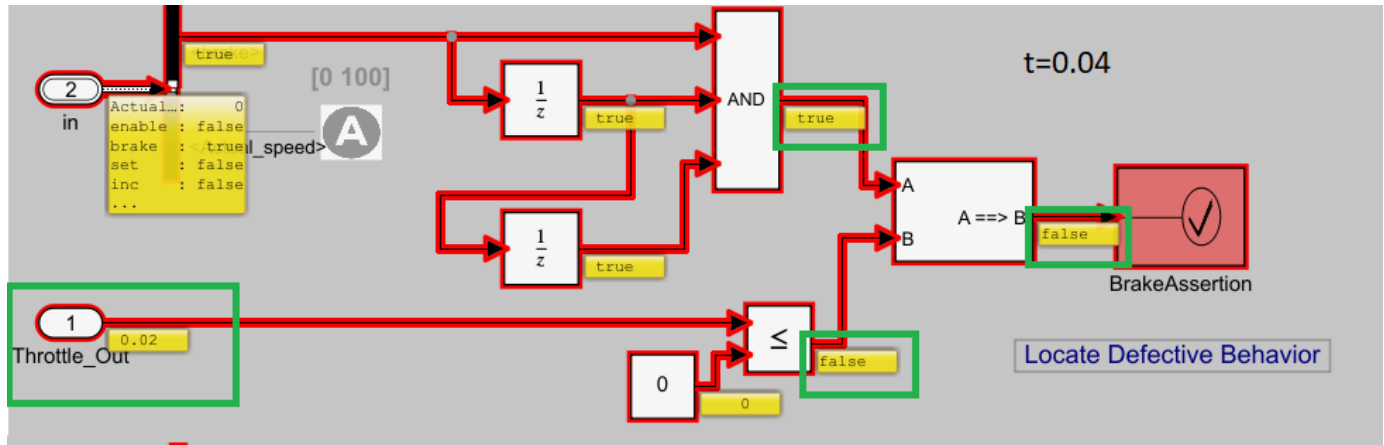
b. The model is highlighted with the counterexample generated by Simulink Design Verifier analysis.

c. The design model is simulated and paused at the time-step of assertion failure.

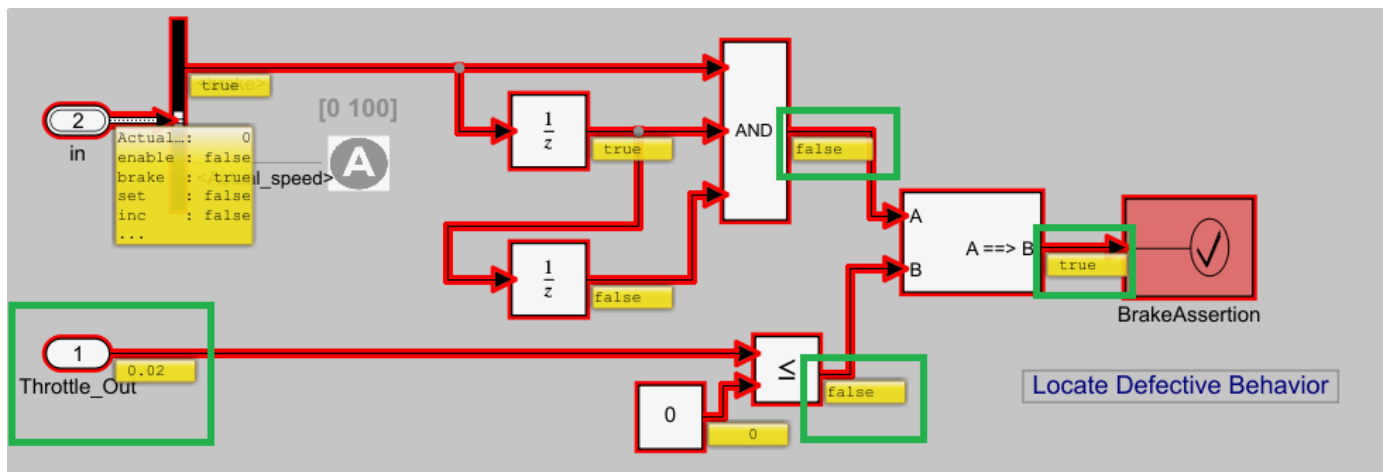
6. Debug and analyze the model by using the Step Back and Step Forward buttons, and inspecting the Port labels.

- The Assert block tests if the output of **A** implies **B** ( $A \implies B$ ) is **false**.
- **A** is **true** when the brake input in is **true** for three consecutive time steps.
- **B** is **true** when the **Throttle\_out**  $\leq 0$

You can notice that the simulation is stopped at  $t=0.04$  when the condition  $A \implies B$  is false. This can be observed from the Port labels.



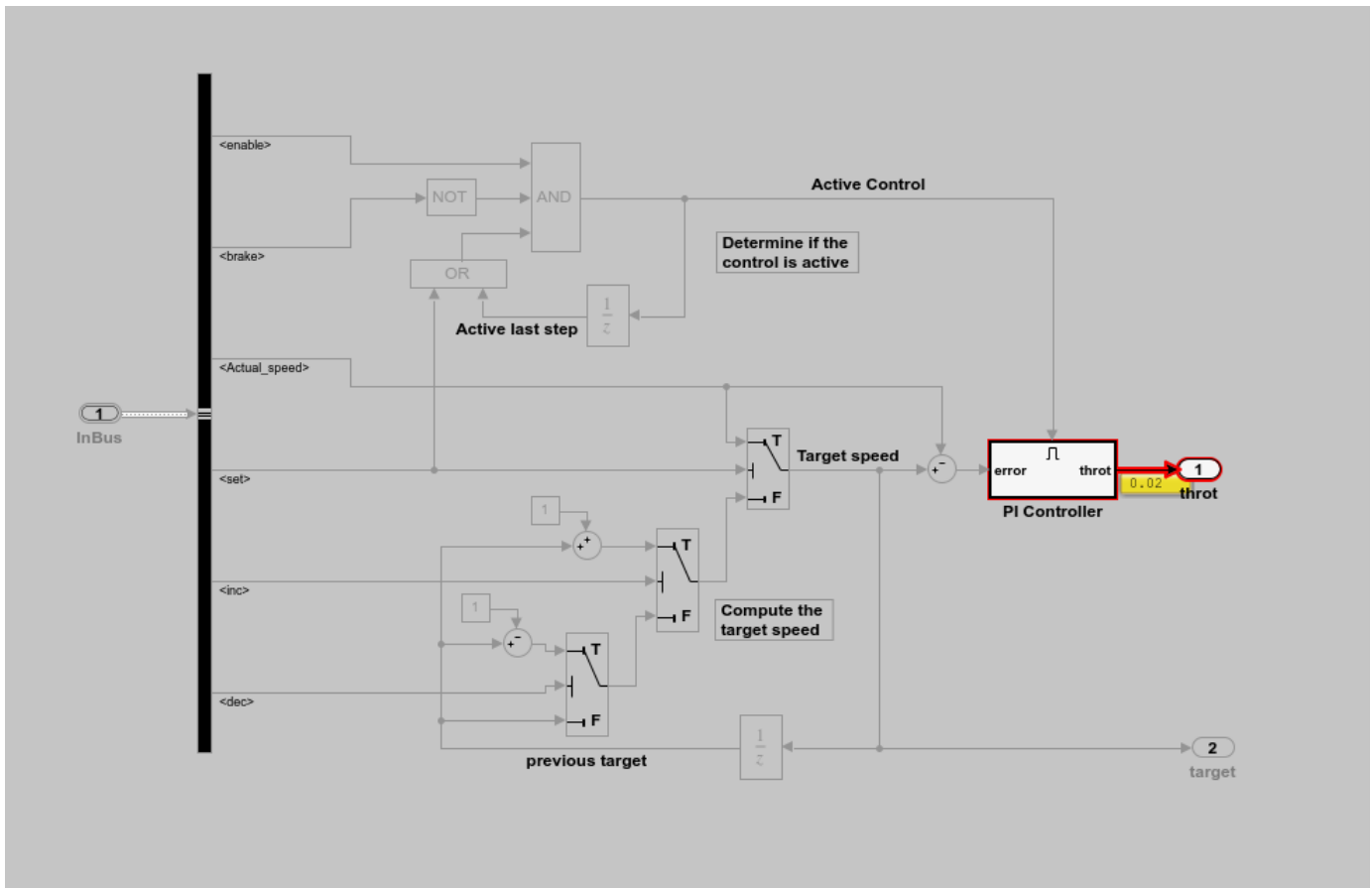
a. On the **Simulation** tab, click the **Step Back** to see the port labels of all the blocks at  $T = (T-0.1)$ .



You can notice that the Port label of **A** is **false** till  $T=0.04$ , when it becomes **true**. At this point the Port label of **B** is **false** ( $\text{Throttle\_Out} > 0$ ). The property is falsified because **Throttle\_Out** is greater than 0.

b. To view the blocks that results in the failure, open the **Design Model > Controller**. The dependent blocks and path are highlighted.





To view the fix, open `sldvdemo_cruise_control_verification` model and then click the **Open Fixed Model** button on the canvas.

## Design and Verify Properties in a Model

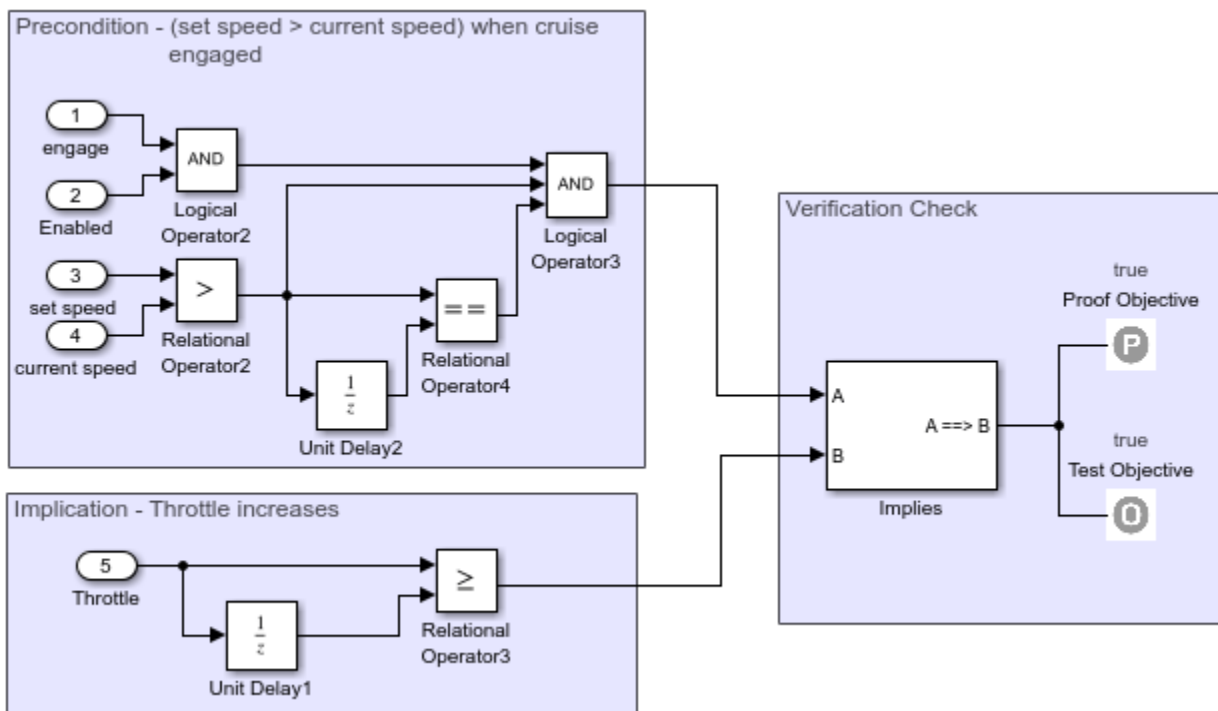
You can use Simulink® Design Verifier™ to model design requirements as properties and then prove properties in a model. To verify that the properties associated with the model requirements hold under all possible input values, use property proving analysis. If the requirement fails, Simulink Design Verifier provides counterexamples to debug the failure.

This example explains how you can model design requirements as properties by using a Proof Objective block and then verify the property for simplified cruise control model discussed in “Analyze a Simple Cruise Control Model”.

### Step 1: Design Property Using Verification Subsystem

The model `sldvexSimpleCruiseControlProperties` consists of Verification Subsystem, that consists of function requirements modeled by using Proof Objective block.

```
load_system('sldvexSimpleCruiseControlProperty');
open_system('sldvexSimpleCruiseControlProperty/Verification Subsystem');
```



### Step 2: Perform Property Proving Analysis

On the **Apps** tab, click arrow on the far right of the Apps section. Under **Model Verification, Validation, and Test** gallery, click **Design Verifier**.

To perform property proving analysis, click **Prove Properties**. The software analyzes the model and displays the results in the Results Summary window. The result indicates that one objective is valid under approximation.

|                      |   |
|----------------------|---|
| Progress             | <div style="width: 100%; height: 10px; background-color: #800000;"></div> |
| Objectives processed | 1/1   |
| Valid                | 0   |
| Falsified            | 0   |
| Elapsed time         | 0:52  |

Property proving completed normally.

1/1 objective valid under approximation

Results:

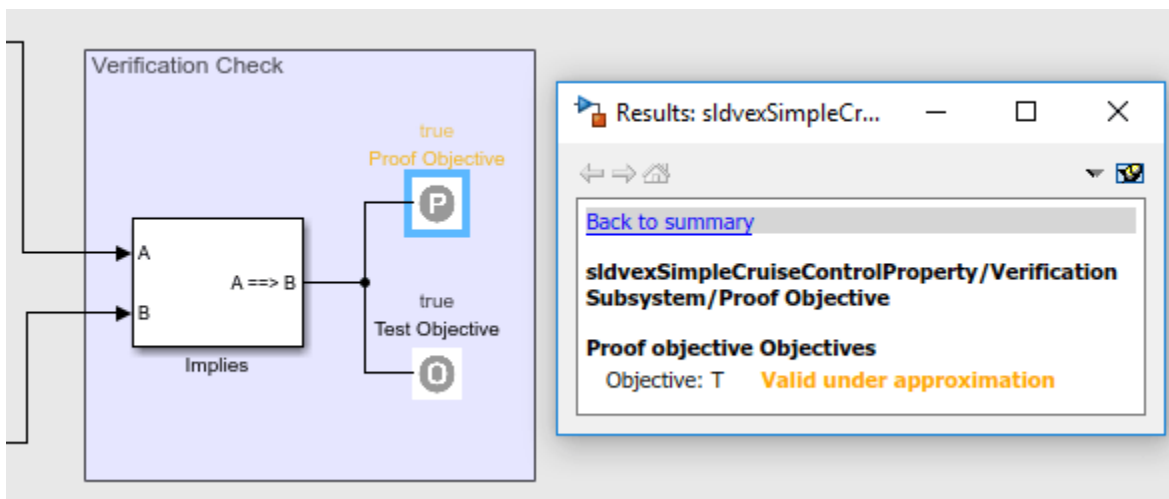
- [Highlight analysis results on model](#)
- Detailed analysis report: ([HTML](#)) ([PDF](#))

Data saved in: [sldvexSimpleCruiseControlProperty\\_sldvdata.mat](#)  
 in folder: [H:\Documents\MATLAB\sldv\\_output](#)  
[\sldvexSimpleCruiseControlProperty](#)

### Step 3: Review Analysis Results

On the **Design Verifier** tab, in the Review Results section, click **Highlight in Model**.

The property that is valid under approximation is highlighted in orange. Click the Proof Objective block. The Results Inspector window displays the objectives of the Proof Objective block.



To view the HTML report, in the Review Results section, click **HTML Report**. The Proof Objective Status chapter lists the proof objective that is found valid under approximation.

## Objectives Valid under Approximation

| # | Type            | Model Item   | Description  | Analysis Time (sec) | Counterexample |
|---|-----------------|--|--------------|---------------------|----------------|
| 1 | Proof objective | <a href="#">Verification Subsystem/Proof Objective</a> | Objective: T | 51                  | n/a            |

**See also**

- “What Is Property Proving?” on page 12-2
- “Prove Properties in a Model” on page 12-5

# Validate Requirements by Analyzing Model Properties

Validate a requirement set by analyzing properties that model individual requirements. Falsified properties indicate design and requirement set incompleteness.

## Overview

In this example, you analyze model properties that are based on four requirements of an engine thrust reverser system. Falsified results from the property analysis suggest that the system design requirements are incomplete -- the system allows behavior that violates several of the following requirements:

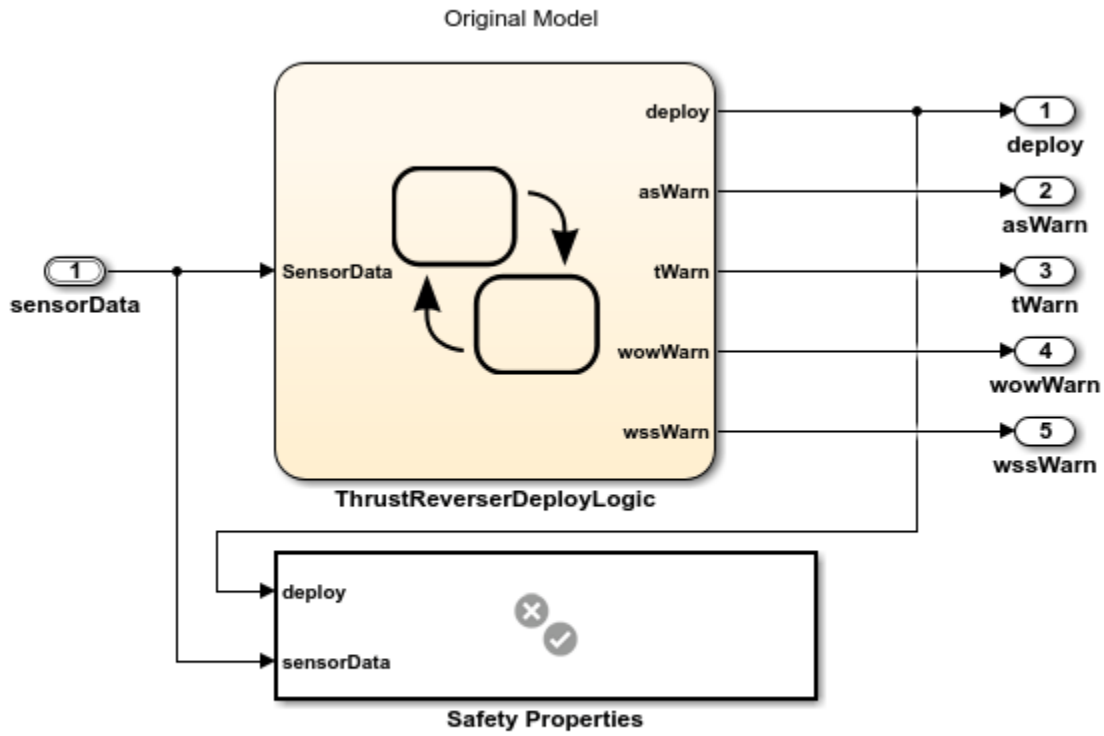
- 1 The thrust reverser shall not deploy if the airspeed is greater than 150 knots.
- 2 The thrust reverser shall not deploy if the aircraft is in the air, as indicated by the value of the weight on wheels sensors. If the aircraft is in the air, the signal value for each of two weight on wheels (WOW) sensors is `false`.
- 3 The thrust reverser shall not deploy if the value of either thrust sensor is positive.
- 4 The thrust reverser shall not deploy if the rotational speed of the landing gear wheels is less than a threshold value.

To better understand the model behavior, you analyze dependencies for a time series input that causes undesirable model behavior because the system lacks required control logic. Then, you analyze a revised control system model which passes the property analysis.

## Analyze the Safety Properties

1. Click the **Open Model** button to open the original model and create a working directory of the example files.

## Requirements Validation Using Property Analysis Example



The Safety Properties subsystem is a Verification Subsystem block from the Simulink® Design Verifier™ library. The verification logic in Safety Properties models the safety requirements. For example, the airspeed requirement is verified by:



If average airspeed > 150 knots, deploy cannot be true.

For more information about Verification Subsystem blocks, see Verification Subsystem.

2. View the requirements. In the model, click the **Show Perspectives views** icon at the lower right and select **Requirements**. The **Requirements** pane opens. Expand thrust\_reverser\_safety\_requirements.

The safety requirements link to the Assertion blocks in the Safety Properties subsystem. The Assertion blocks are considered proof objectives. The verification status for each requirement reflects the property analysis results of its corresponding Assertion block.

3. Display the verification status for the requirements. Right-click one of the requirements in the browser and select **Verification Status**. The **Verified** column indicates that the requirements are unexecuted.

| Index                                 | ID   | Summary              | Verified |
|---------------------------------------|------|----------------------|----------|
| ▼ thrust_reverser_safety_requirements |      |                      |          |
| 1                                     | R1.1 | Airspeed Condition   |          |
| 2                                     | R1.2 | WOW Condition        |          |
| 3                                     | R1.3 | Throttle Condition   |          |
| 4                                     | R1.4 | Wheelspeed Condition |          |

4. Analyze the model properties. In the **Apps** tab, click **Design Verifier**. In the **Design Verifier** tab, click **Prove Properties**.

When the property analysis completes, click the **Refresh** button to update the status in the **Verified** column. The results show that three out of four objectives are falsified -- analysis found a signal condition that falsifies the properties, and therefore violates the requirements.

| Index                                 | ID   | Summary              | Verified |
|---------------------------------------|------|----------------------|----------|
| ▼ thrust_reverser_safety_requirements |      |                      |          |
| 1                                     | R1.1 | Airspeed Condition   |          |
| 2                                     | R1.2 | WOW Condition        |          |
| 3                                     | R1.3 | Throttle Condition   |          |
| 4                                     | R1.4 | Wheelspeed Condition |          |

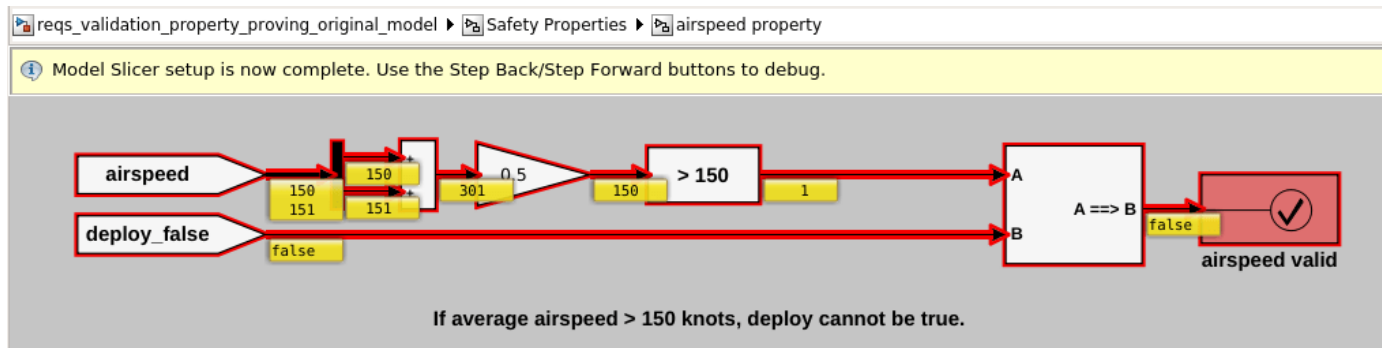
### Analyze Model Behavior with Counterexamples

From the Design Verifier Results Summary window, click **HTML** to open the detailed analysis report. In Chapter 4, each falsified property lists a counterexample. For example, in the counterexample that falsifies the airspeed requirement:

- At  $t = 0.1$ , the thrust reverser is deployed with airspeed below the threshold.
- At  $t = 0.2$ , the thrust reverser is still deployed with airspeed above the threshold.

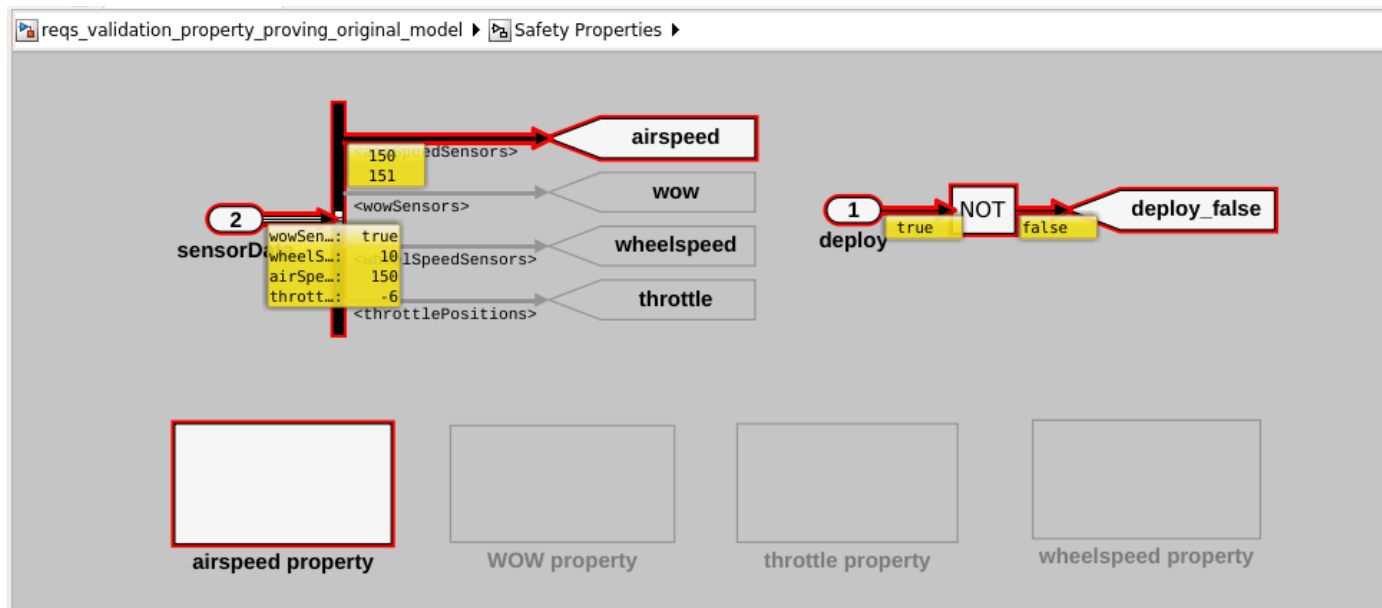
The counterexample time series indicates a condition that might be difficult to encounter in simulation, but nonetheless causes model behavior that violates a requirement. Investigate the behavior by analyzing signal dependencies in the counterexample:

1. In the **Design Verifier** tab, click the **Highlight in Model** button.
2. Select the airspeed valid assertion block in the **Test Unit > Safety Properties > airspeed property** subsystem.
3. In the **Design Verifier** tab, click the **Debug Using Slicer** button. The model highlights dependencies of the airspeed valid assertion, and displays signal values at  $T = 0.2$ .



4. Move up one level in the model, to the **Safety Properties** subsystem. Step back through the counterexample simulation time. In the **Simulation** tab, click **Step Back**.

5. At T = 0.1, the average airspeed is below the threshold, and the thrust reverser is deployed. Stepping forward, at T = 0.2, the average airspeed is above the threshold, and the thrust reverser is still deployed. This violates a requirement.



The falsified property and the dependency analysis suggest that the control system algorithm is incompletely designed, and the requirements are incomplete.

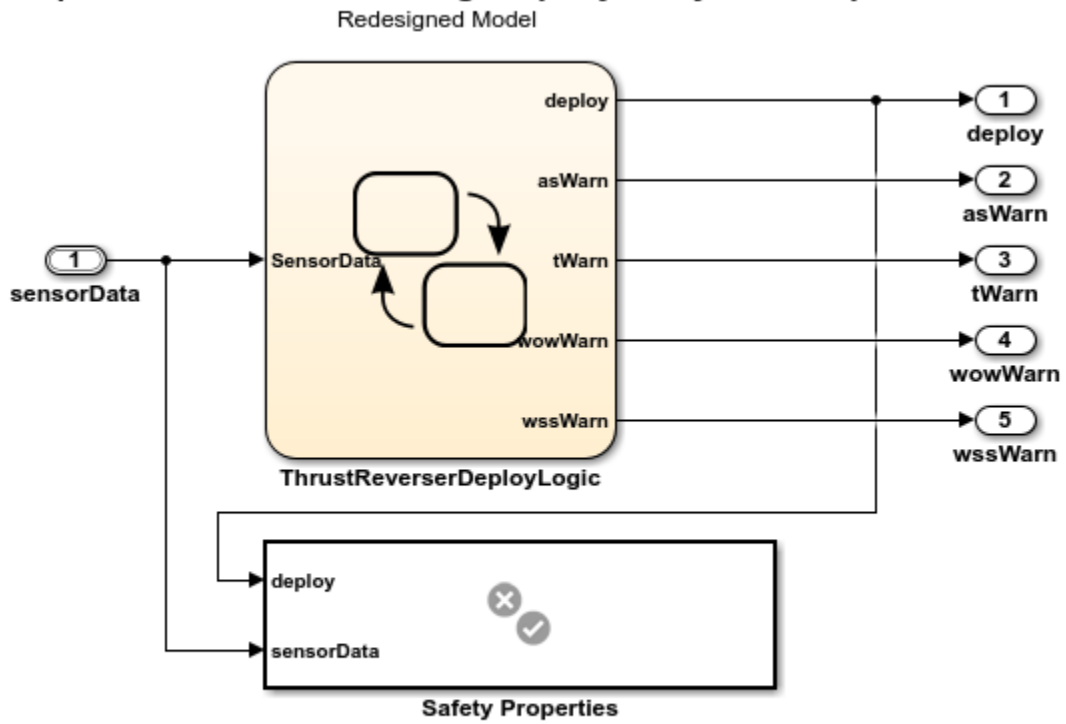
### Analyze the Redesigned System

Redesigning a control system requires rethinking requirements. In this case, the lack of an intermediate standby state allows undesirable system behavior when inputs change suddenly. Adding an intermediate deployment mode which delays thrust reverser response resolves the issue.

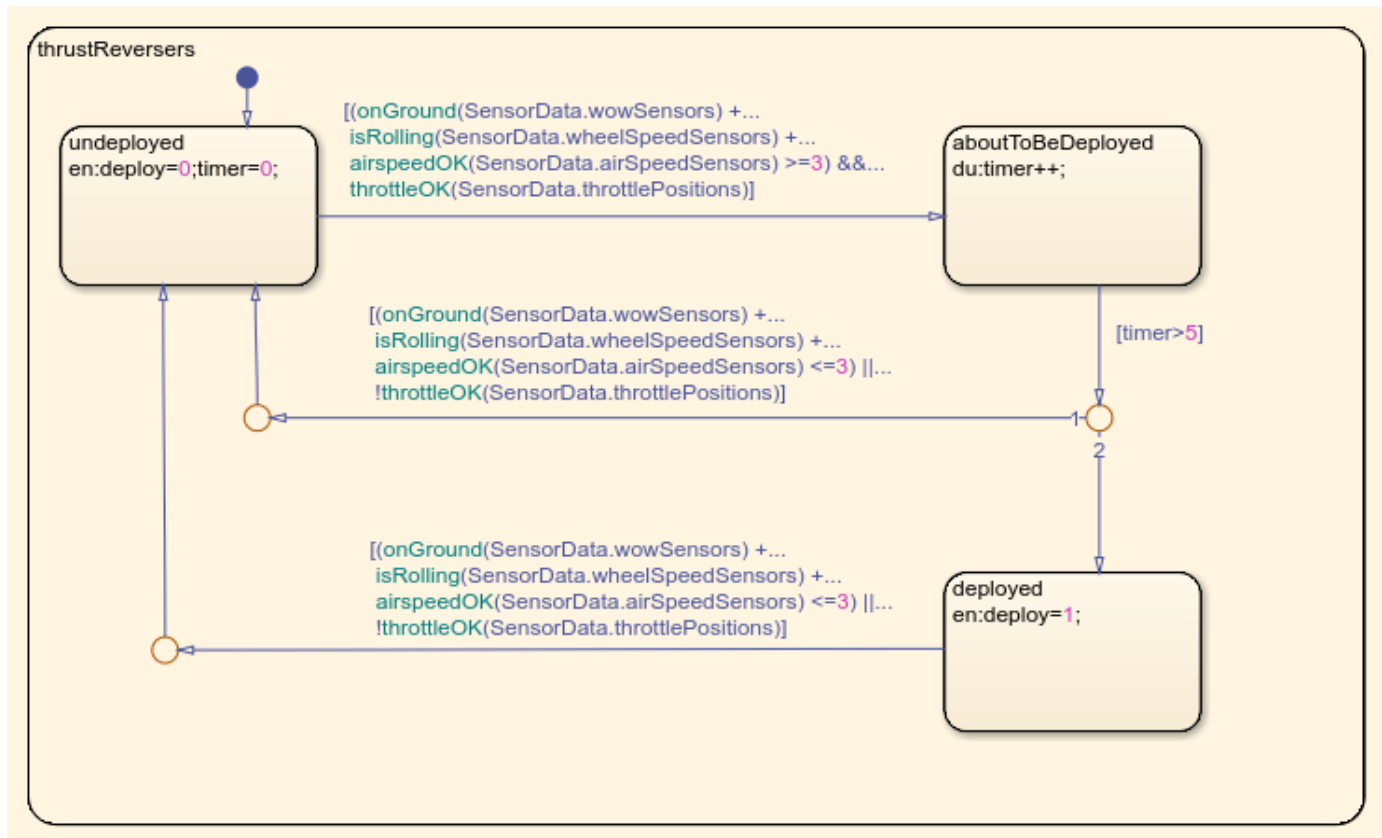
Open the reqs\_validation\_property\_proving\_redesigned\_model model. Open the thrustReversers chart.



## Requirements Validation Using Property Analysis Example



Copyright 2019-2020 The MathWorks, Inc.



The additional design requirement states that the thrust reverser shall deploy after a pause. The redesigned model includes:

- An additional `aboutToBeDeployed` state.
- Expanded transition conditions that return to `undeployed`.

Create links from the verification blocks in the redesigned model to the requirements:

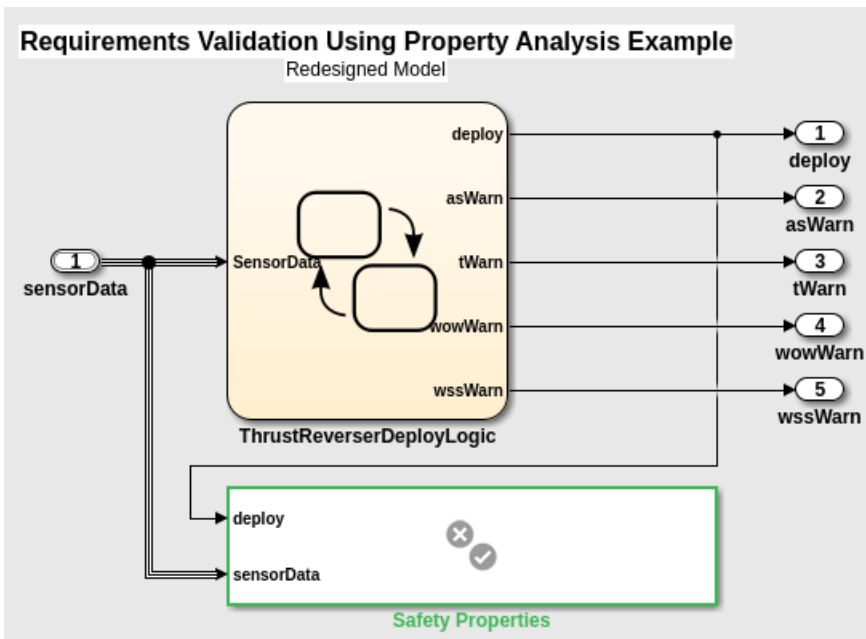
1. In the model, from the **Apps** tab, click **Requirements Manager**.
2. In the **Requirements** tab, click **Requirements Editor**.
3. Open `thrust_reverser_safety_requirements` in the **Requirements Editor**.
4. For requirement 1.1, Airspeed Condition, link to the airspeed valid block in the Safety Properties > airspeed property subsystem. Drag R1.1 from the requirements browser to the airspeed valid block in the model.
5. The new link appears in the Requirements Editor, in the right pane, under **Links**.
6. Delete the link to the assert block in the original model. If the original model is closed, this link appears unresolved. Next to the link, click the **Delete Link** icon.

▼ **Links**

↳ Verified by:

- ⚠ reqs\_validation\_property\_proving\_original\_model:5:29 [✗](#)
- 📄 airspeed valid [?](#) Delete Link

7. Repeat for the other three requirements and verification blocks in the redesigned model. Run the property analysis on the revised model. View the results in the Requirements pane.



| Index                                 | ID   | Summary              | Verified  |
|---------------------------------------|------|----------------------|---|
| ▼ thrust_reverser_safety_requirements |      |                      | <div style="background-color: green; width: 100%; height: 15px;"></div> |
| 1                                     | R1.1 | Airspeed Condition   | <div style="background-color: green; width: 100%; height: 15px;"></div> |
| 2                                     | R1.2 | WOW Condition        | <div style="background-color: green; width: 100%; height: 15px;"></div> |
| 3                                     | R1.3 | Throttle Condition   | <div style="background-color: green; width: 100%; height: 15px;"></div> |
| 4                                     | R1.4 | Wheelspeed Condition | <div style="background-color: green; width: 100%; height: 15px;"></div> |

The results show that the properties are valid.

## Use Observer Reference Blocks for Property Proving Analysis

This example shows you how to use an Observer Reference block to perform property proving analysis with multiple properties without making changes to the model. In this example, you replace an existing verification subsystem with an Observer Reference block. However, you can add an Observer Reference block to your model even if your model does not have a verification subsystem to replace. For more information, see “Access Model Data Wirelessly by Using Observers” (Simulink Test).

The model `sldvdemo_debounce_validprop` is configured for analysis and attempts to prove that:

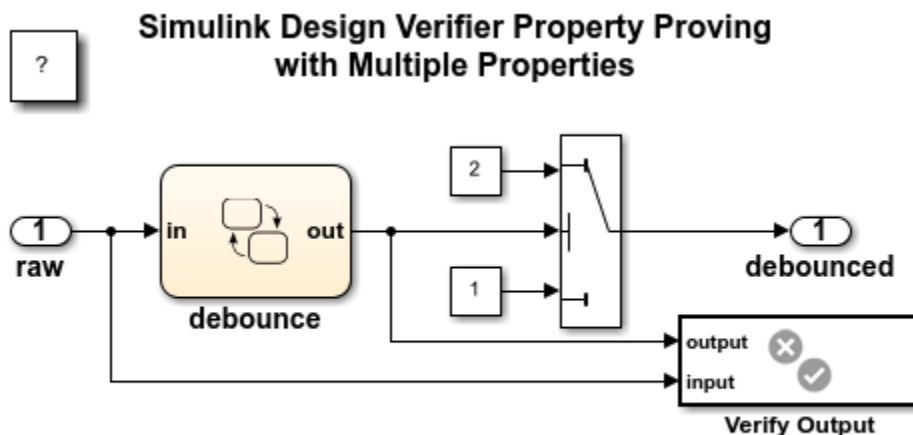
- 1 When the current and six previous input values are true, the output will be true.
- 2 When the current and six previous input values are false, the output will be false.

For a detailed description of the Observer Reference block, see “Isolate Verification Logic with Observers” on page 12-29.

### Step 1: Open the Model

The `sldvdemo_debounce_validprop` model contains a verification subsystem called Verify Output. For more information on the Verification Subsystem, see Verification Subsystem. To open the model, enter:

```
open_system('sldvdemo_debounce_validprop');
```



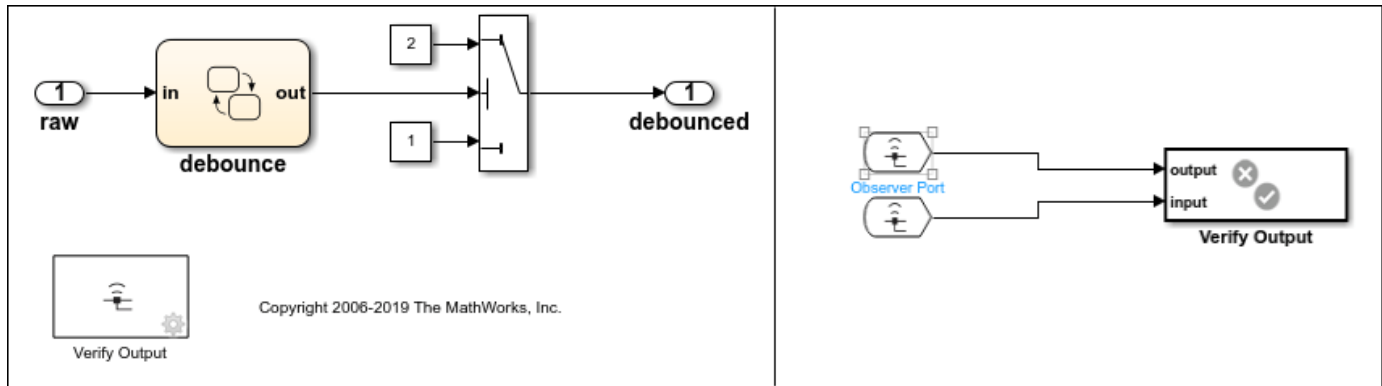
Copyright 2006-2023 The MathWorks, Inc.

### Step 2: Replace the Verification Subsystem with an Observer Reference Block

Perform these steps to create a new Observer Reference block and replace the Verify Output verification subsystem.

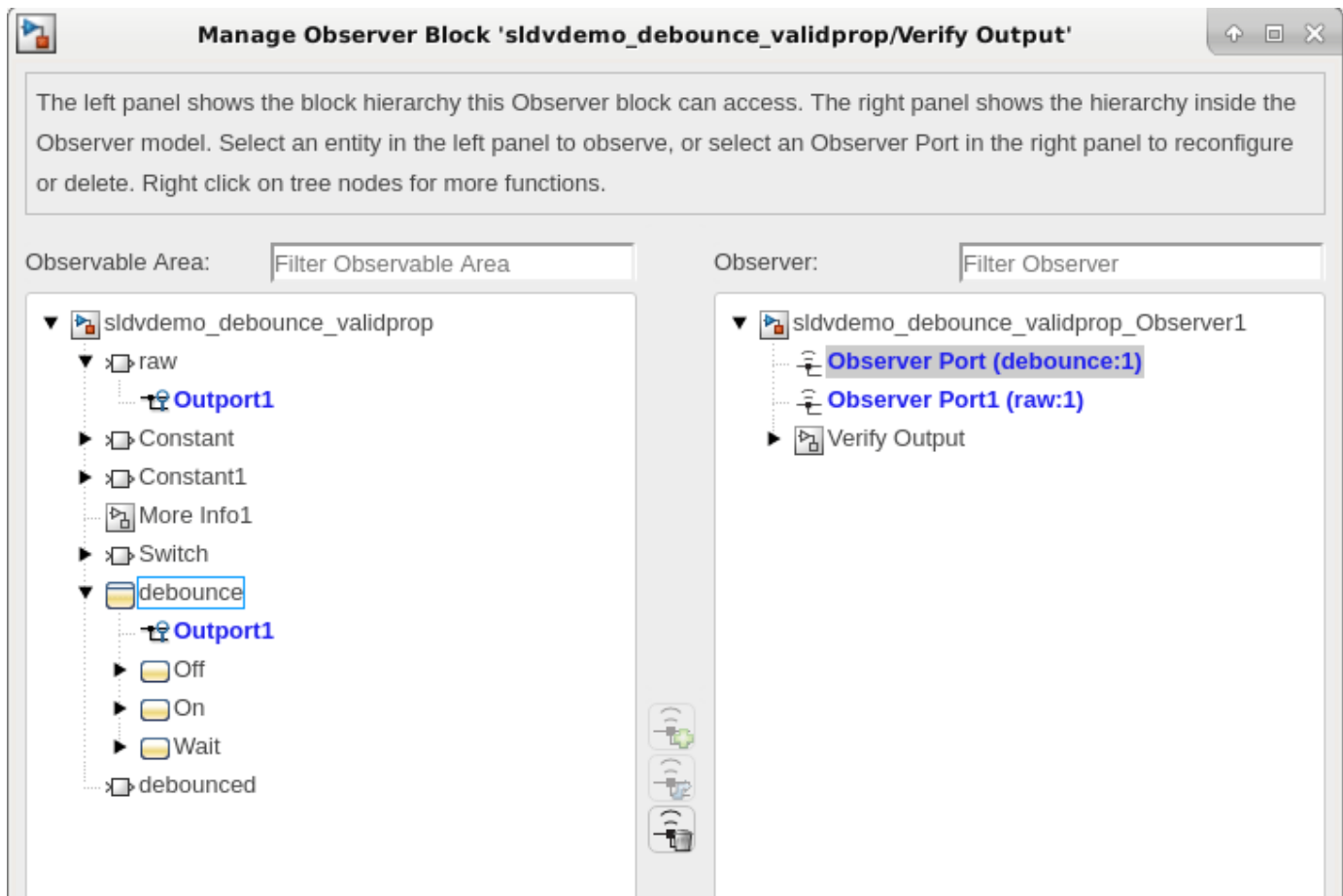
1. Right-click the the Verify Output subsystem. In the context menu, click **Observers > Move selected block to Observer > New Observer**.
2. Click **Yes** in the dialog box that appears.

3. An Observer Reference block replaces the verification subsystem. The `sldvdemo_debounce_validprop_Observer1` Observer model opens.



4. Save `sldvdemo_debounce_validprop_Observer1` in a writable folder on the MATLAB path.

5. Double-click one of the Observer Port blocks to open the Manage Observer Block window. The two signals, `debounce` and `raw`, are automatically map to the two Observer Port blocks in the `sldvdemo_debounce_validprop_Observer1` Observer model.



### Step 3: Perform Property Proving Analysis

To perform the property proving analysis, follow these steps:

1. In the top-level model, on the **Design Verifier** tab, click **Prove Properties**.
2. After the analysis completes, the Simulink Design Verifier Results Summary window reports that two objectives are satisfied.
3. Open the HTML analysis report to see a detailed report that includes information about the top-level model and Observers.

### Step 4: Review the Property Proving Analysis Report

The analysis report shows the Observer information for property proving in the Observer Model(s) section of the Properties chapter..

## 4.2. Observer Model(s)

This section contains information about each property in the observer model(s).

### 4.2.1. Verify Output/FoutCorrect

#### Summary

Model Item: [Verify Output/FoutCorrect](#)

Property:    Objective: T

Status:      Valid

### 4.2.2. Verify Output/ToutCorrect

#### Summary

Model Item: [Verify Output/ToutCorrect](#)

Property:    Objective: T

Status:      Valid

### Step 5: Cleanup

Close the model.

```
bdclose('sldvdemo_debounce_validprop');
```

### Related Topics

- “Isolate Verification Logic with Observers” on page 12-29

## Prove Properties with Requirements Table Blocks

This example shows how to use a Requirements Table block and Simulink® Design Verifier™ to prove the properties of a engine thrust reverser system.

When you use a Requirements Table block and Simulink Design Verifier for property proving:

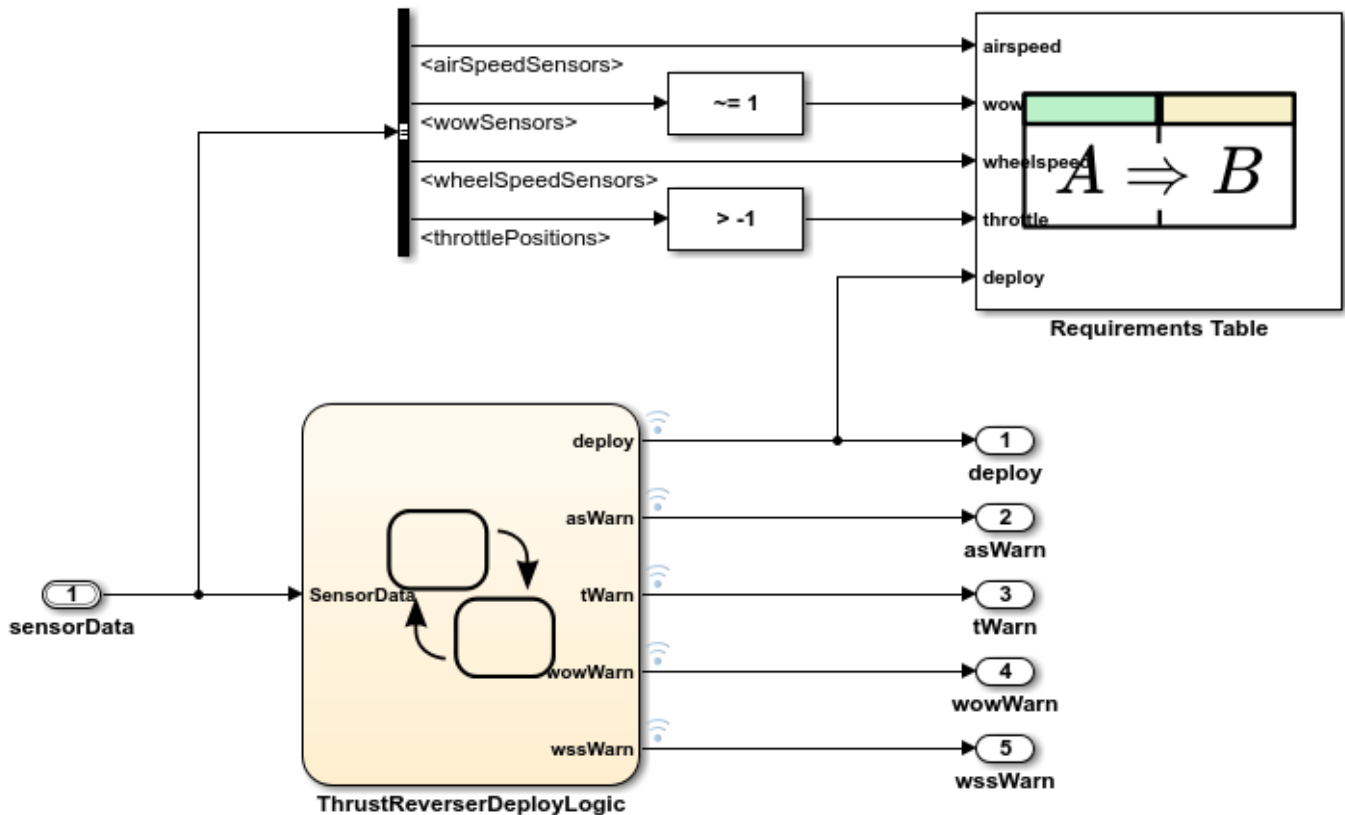
- 1 Each requirement in the block defines a formal requirement that you can use to test properties of a model, subsystem, or block. In this example, the Requirements Table block represents the requirements as preconditions and postconditions.
- 2 Each requirement in the block produces a corresponding requirement in the Requirements Editor. See “Configure Properties of Formal Requirements” (Requirements Toolbox).
- 3 Simulink Design Verifier produces proof objectives for the requirements in the requirement set. The postconditions define the logical conditions that you would normally define in Proof Objective blocks. The block evaluates the proof objective associated with a postcondition only if the precondition is true.

For more information, see “Use a Requirements Table Block to Create Formal Requirements” (Requirements Toolbox) and “What Is Property Proving?” on page 12-2.

### View the Requirements in the Requirements Table Block

Open the example model, `property_proving_reqtable`. In this example, you test the properties of the engine thrust reverser system, which is modeled by the chart, `ThrustReverserDeployLogic`. The Requirements Table block uses the chart input signals and deploy output signal to observe the chart behavior. The block defines data in expressions to assess the inputs and outputs of the chart. See “Define Data in Requirements Table Blocks” (Requirements Toolbox).

## Prove Properties with a Requirements Table Block



Copyright 2022 The MathWorks, Inc.

To view the verification logic associated with each requirement, open the Requirements Table block. The requirements correspond to the requirements in the requirement set used in the example “Validate Requirements by Analyzing Model Properties” on page 12-63. The block proves these properties:

- 1 The thrust reverser shall not deploy if the airspeed is greater than 150 knots.
- 2 The thrust reverser shall not deploy if the aircraft is in the air, as indicated by the value of the weight on wheels sensors. If the aircraft is in the air, the signal value for each of two weight on wheels (WOW) sensors is false.
- 3 The thrust reverser shall not deploy if the value of either thrust sensor is positive.
- 4 The thrust reverser shall not deploy if the rotational speed of the landing gear wheels is less than a threshold value.

Each requirement defines a property. If the preconditions are valid, the postcondition must also be satisfied to prove the property.



| Requirements |   | Assumptions  |               |
|--------------|---|--|---------------|
| Index        | Summary   | Precondition   | Postcondition |
|              |   |  |               |
| 1            | Thrust reverser shall not deploy if airspeed is greater than 150.   | $\text{mean}(\text{airspeed}) > 150$                           | false         |
| 2            | Thrust reverser shall not deploy if the aircraft is in the air, as indicated by the value of the weight on wheels sensors. If the aircraft is in the air, the signal value for each of two weight on wheels (WOW) sensors is false. | $\text{sum}(\text{wow}) \geq 3$                                | false         |
| 3            | Thrust reverser shall not deploy if the value of either throttle position is positive.  | $\text{sum}(\text{throttle}) \geq 3$                           | false         |
| 4            | Thrust reverser shall not deploy if landing gear wheels rotational speed is less than 10.   | $\text{wheelspeed}(1) < 10 \ \&\& \ \text{wheelspeed}(2) < 10$ | false         |

### Prove Properties

To prove the properties, in the **Design Verifier** tab, click **Prove Properties**. In this example, the properties of the chart are proven. The Requirements Table block highlights the postconditions associated with the proven proof objectives in green.

| Postcondition |
|---------------|
| deploy        |
| false         |
| false         |
| false         |
| false         |

If the requirement proof objective is falsified, the block highlights the requirement in red. Otherwise, if Simulink Design Verifier is unable to prove or disprove the proof objective, the block highlights the requirement in yellow. You can investigate this behavior by replacing the chart in this example with

the first iteration of the chart used in the “Validate Requirements by Analyzing Model Properties” on page 12-63 example.

### **See Also**

Requirements Table

### **Related Examples**

- “Prove Properties in a Model” on page 12-5
- “Use a Requirements Table Block to Create Formal Requirements” (Requirements Toolbox)
- “Export Tests from Models That Contain Requirements Table Blocks with Simulink Design Verifier” on page 13-30

# Reviewing the Results

---

- “Highlight Results on the Model” on page 13-2
- “Manage Simulink Design Verifier Data Files” on page 13-7
- “Manage Simulink Design Verifier Harness Models” on page 13-13
- “Simulate Harness Model with Signal Editor Inputs Block” on page 13-22
- “Export Test Cases to Simulink Test” on page 13-27
- “Export Tests from Models That Contain Requirements Table Blocks with Simulink Design Verifier” on page 13-30
- “Review Results” on page 13-35
- “View Log Files” on page 13-56
- “Review Analysis Results” on page 13-57

## Highlight Results on the Model

| In this section...  |
|---|
| “Results Review with Model Highlighting” on page 13-2     |
| “Simulink Design Verifier Results Inspector” on page 13-2 |
| “Highlight Results on Model Automatically” on page 13-2   |
| “Green Highlighting on Model” on page 13-4                |
| “Red Highlighting on Model” on page 13-4                  |
| “Orange Highlighting on Model” on page 13-4               |
| “Gray Highlighting on Model” on page 13-6                 |

### Results Review with Model Highlighting

When you analyze a model by using Simulink Design Verifier, the analyzed model objects are automatically highlighted in one of these colors:

- Green
- Red
- Orange
- Gray

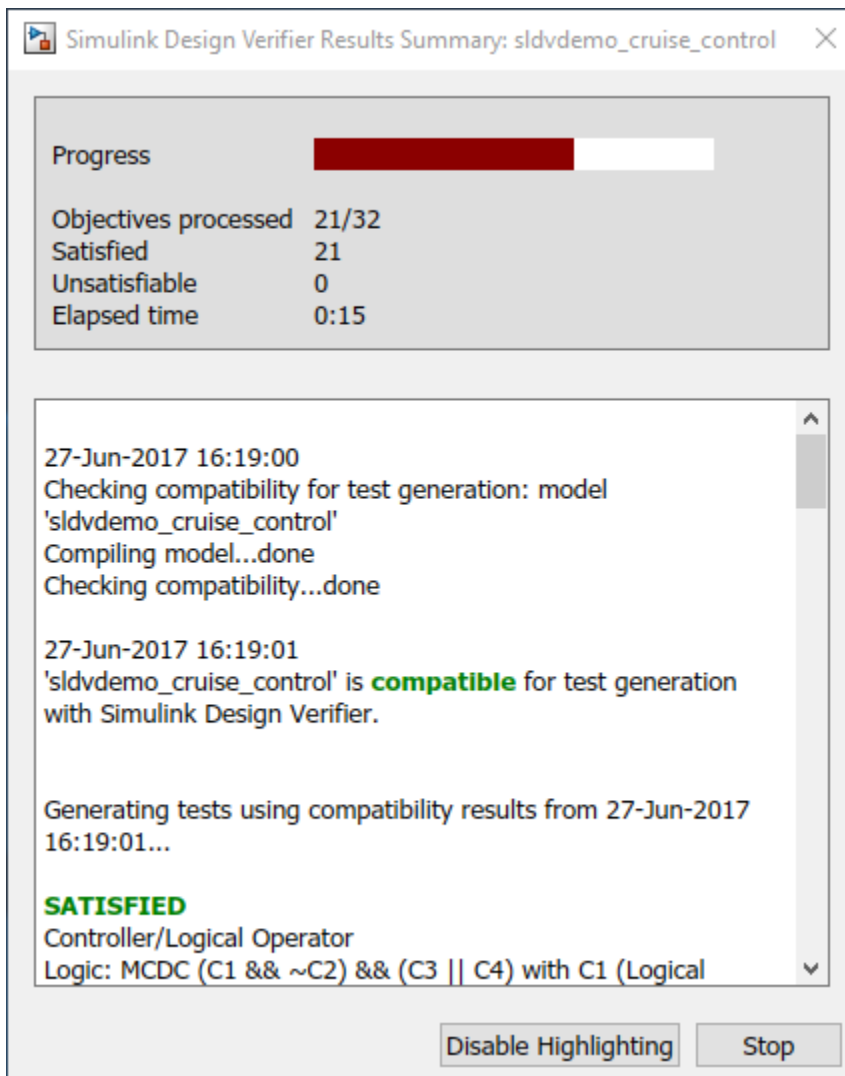
You can review the analysis results at a glance by viewing the objects that are highlighted in the Simulink Editor.

### Simulink Design Verifier Results Inspector

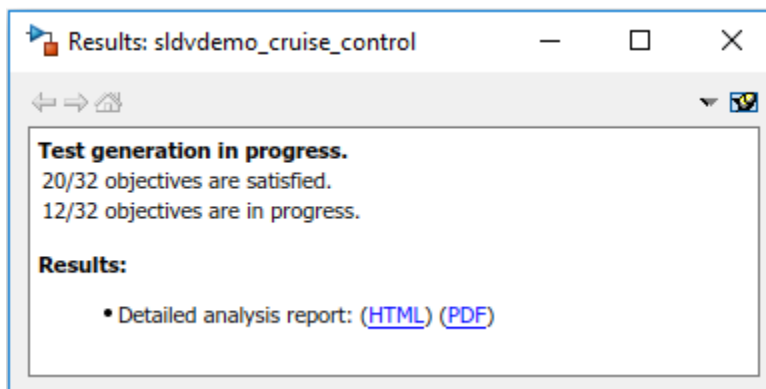
When a model is highlighted, you can click an object for which the analysis recorded results. The Simulink Design Verifier Results Inspector then displays the detailed analysis results for that object.

### Highlight Results on Model Automatically

During analysis, Simulink Design Verifier highlights the model objects automatically when the objectives status is updated. By default, the automatic highlighting is enabled. To disable the highlighting, click **Disable Highlighting** in the Results Summary window.



In the Simulink Editor, results highlighting appears on the model. When highlighting is enabled, the Results Inspector opens displaying the summary of status for analysis objectives.



---

**Note** Simulink Design Verifier does not highlight the Stateflow state transition tables. The Simulink Design Verifier reports, data files, and log files include the analysis data for the state transition tables. Using the report, you can navigate to the state transition tables.

---

## Green Highlighting on Model

Objects that are highlighted in green have the following meaning for each type of analysis.

| Analysis Mode          | Green highlighting  |
|------------------------|---|
| Design error detection | <ul style="list-style-type: none"> <li>The analysis did not find overflow or division-by-zero errors.</li> <li>The analysis did not find dead logic.</li> <li>The analysis did not find intermediate or output signals outside the range of user-specified minimum and maximum constraints.</li> <li>The analysis did not find out of bound array access errors.</li> </ul> |
| Test generation        | The analysis found test cases that satisfy the test objectives.   |
| Property proving       | The analysis found all the proof objectives as valid.   |

## Red Highlighting on Model

Objects that are highlighted in red have the following meaning, depending on the analysis type.

| Analysis Mode          | Red highlighting  |
|------------------------|---|
| Design error detection | <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> <li>The analysis found at least one test case that causes an out of bound array access error.</li> </ul> |
| Test generation        | The analysis did not satisfy certain test objectives.   |
| Property proving       | The analysis disproved a proof objective and generated a counterexample that falsified that objective.  |

If your model contains at least one object highlighted in red, there might be further design errors in your model that Simulink Design Verifier does not highlight in red. If an object in your design causes run-time errors, Simulink Design Verifier might not be able to determine further errors on objects that are downstream of or rely on the results of the object that causes the run-time errors. Resolve the errors that cause the initial red highlighting and rerun the analysis to determine if Simulink Design Verifier highlights other objects in your model as red.

## Orange Highlighting on Model

Objects that are highlighted in orange have the following meaning, depending on the analysis type.

| Analysis Mode          | Orange highlighting   |
|------------------------|---|
| Design error detection | <p>For the highlighted model object,</p> <ul style="list-style-type: none"> <li>• The analysis did not decide at least one design error detection objective. This situation can occur when: <ul style="list-style-type: none"> <li>• The analysis is still in progress.</li> <li>• The analysis times out.</li> <li>• The analysis cannot decide a design error detection objective because of division by zero or nonlinear arithmetic.</li> <li>• The software cannot decide a design error detection objective because of stubbing. For more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.</li> <li>• The software cannot decide a design error detection objective because of limitations of the analysis engine. For example, if the analysis encounters an unbounded while loop, it performs an approximation. For more information, see “Role of Approximations During Model Analysis” on page 2-20.</li> </ul> </li> <li>• The analysis found dead logic that approximations can impact. For more information, see “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23.</li> <li>• The analysis found valid objectives that approximations can impact. For more information, see “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23.</li> </ul> |
| Test generation        | <p>For the highlighted model object,</p> <ul style="list-style-type: none"> <li>• The analysis did not decide at least one test objective. This situation can occur when: <ul style="list-style-type: none"> <li>• The analysis is still in progress.</li> <li>• The analysis times out.</li> <li>• The analysis cannot decide a test objective because of division by zero or nonlinear arithmetic.</li> <li>• The software cannot decide a test objective because of stubbing. For more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.</li> <li>• The software cannot decide a test objective because of limitations of the analysis engine. For example, if the analysis encounters an unbounded while loop, it performs an approximation. For more information, see “Role of Approximations During Model Analysis” on page 2-20.</li> </ul> </li> <li>• The analysis found unsatisfiable objectives that approximations can impact. For more information, see “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23.</li> <li>• The analysis is unable to confirm the satisfied status through validation results. For more information, see “Objectives Satisfied - Needs Simulation” on page 13-46.</li> </ul>  |

| Analysis Mode    | Orange highlighting  |
|------------------|--|
| Property proving | <p>For the highlighted model object,</p> <ul style="list-style-type: none"> <li>• The analysis did not decide at least one proof objective. This situation can occur when: <ul style="list-style-type: none"> <li>• The analysis is still in progress.</li> <li>• The analysis times out.</li> <li>• A proof objective exists on a signal whose value the software cannot control, for example, a Constant block.</li> <li>• The analysis cannot decide a proof objective because of division by zero or nonlinear arithmetic.</li> <li>• The software cannot decide a proof objective because of stubbing. For more information, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.</li> <li>• The software cannot decide a proof objective because of limitations of the analysis engine. For example, if the analysis encounters an unbounded while loop, it performs an approximation. For more information, see “Role of Approximations During Model Analysis” on page 2-20.</li> </ul> </li> <li>• The analysis found valid objectives that approximations can impact. For more information, see “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23.</li> <li>• The software is unable to confirm the falsified status through validation results. For more information, see “Objectives Falsified - Needs Simulation” on page 13-49.</li> </ul> |

### Gray Highlighting on Model

Objects that are highlighted in gray have the following meaning.

| Analysis Mode   | Gray Highlighting                              |
|---|--|
| <ul style="list-style-type: none"> <li>• Design error detection</li> <li>• Test generation</li> <li>• Property proving</li> </ul> | The model object was not part of the analysis. |



## Manage Simulink Design Verifier Data Files

Simulink Design Verifier generates a data file after completing the analysis. The data file is a MAT-file that contains the `sldvData` structure. 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 for further custom analysis or to generate a custom report.

### Generate `sldvData` Structure

Complete these steps to explore the contents of the `sldvData` structure.

- 1 Generate test cases for the `sldvdemo_flipflop` model.

```
sldvdemo_flipflop;
sldvrun('sldvdemo_flipflop');
```

- 2 Load the `sldvData` structure for the `sldvdemo_flipflop` model to the MATLAB workspace.

```
load('sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat')
```

- 3 Display the names of the fields in 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:

#### Model Information

The `ModelInformation` field contains information about the model you analyze in Simulink Design Verifier. This table describes the subfields in the `ModelInformation` field.

| Subfield Name    | Description  |
|------------------|--|
| Name             | Model name.  |
| Version          | Model number.  |
| Author           | User name.   |
| TimeStamp        | Date and time of last update.  |
| SubsystemPath    | Full path name of the subsystem (if any) that was analyzed.                  |
| ExtractedModel   | Name of model extracted to analyze subsystem in <code>SubsystemPath</code> . |
| ReplacementModel | Name of model containing block replacements.                                 |

| Subfield Name     | Description  |
|-------------------|--|
| HarnessOwnerModel | Name of owner of analyzed Simulink Test harness model. |

### Analysis Information

The AnalysisInformation field lists the analysis options and related information. The table describes the AnalysisInformation field.

| Subfield Name           | Description   |
|-------------------------|---|
| Status                  | Status of analysis.   |
| AnalysisTime            | Duration in seconds of analysis   |
| Options                 | Deep copy of the Simulink Design Verifier options object used during the analysis.  |
| InputPortInfo           | Cell array of structures with information about every Inport block in top level of system.  |
| OutputPortInfo          | Cell array of structures with information about every Outport block in top level of system.   |
| SampleTimes             | For internal use only.  |
| Parameters              | For internal use only.  |
| AbstractedBlocks        | For internal use only.  |
| Approximations          | Structure describing approximations performed during analysis. For more information, see "Role of Approximations During Model Analysis" on page 2-20. |
| ReplacementInfo         | For internal use only.  |
| PreProcessingTime       | Time in seconds to build or reuse model representation.   |
| ModelRepresentationInfo | Date and time of model representation used in analysis.   |

### Model Objects

The ModelObjects field lists the model items and their associated objectives. The table describes the ModelObjects field.

| Subfield Name  | Description   |
|----------------|---|
| descr          | Full path to model object, including objects in Stateflow chart.                      |
| typeDesc       | Type of model object, returned as S for a state object and T for a transition object. |
| slPath         | Full path to Simulink model object.   |
| sfObjType      | Type of Stateflow object. Example: S for state and T for transition.                  |
| sfObjNum       | Integer representing a unique identifier for Stateflow object.                        |
| sid            | For internal use only.  |
| designSid      | For internal use only.  |
| replacementSid | For internal use only.  |

| Subfield Name | Description  |
|---------------|--|
| objectives    | Vector of integers representing indices of objectives related to model object. |

### Constraints

The **Constraints** field lists information about specified minimum and maximum values (if any) on input ports in your model. The table describes the **Constraints** field.

| Subfield Name | Description   |
|---------------|---|
| DesignMinMax  | Cell array of structures containing name and minimum and maximum values of each input port. |

### Objectives

The **Objectives** field lists information about each objective, such as its type, status, and description. The table describes the **Objectives** field.

| Subfield Name    | Description   |
|------------------|---|
| type             | Type of objective.  |
| status           | Status of objective.  |
| descr            | Description associated with objective.  |
| label            | Label of objective.   |
| outcomeValue     | Integer representing outcome of objective.  |
| coveragePointIdx | Integer representing index of coverage point associated with objective.   |
| linkInfo         | For internal use only.  |
| range            | For internal use only.  |
| detectability    | Detectability status of objective.<br><br>This field appears in the data file when you set the analysis "Mode" on page 15-10 to <b>Test Generation</b> and "Model coverage objectives" on page 15-31 to <b>Enhanced MCDC</b> .  |
| detectionSites   | Simulink Identifier (SID) array of detection sites for detectable objectives.<br><br>This field appears in the data file when you set the analysis "Mode" on page 15-10 to <b>Test Generation</b> and "Model coverage objectives" on page 15-31 to <b>Enhanced MCDC</b> . |
| modelObjectIdx   | Integer representing index of model object associated with objective.   |
| analysisTime     | Analysis time of objective.   |
| testCaseIdx      | Integer representing index of test case or counterexample addressed in objective.   |

### Test Cases or Counterexamples

This field name depends on the type of check:

- If you set the **Mode** parameter to **Design error detection**, the **CounterExamples** field provides information on each test case that results in an integer-overflow or division-by-zero error.
- If you set the **Mode** parameter to **Test generation**, the **TestCases** field lists information about each test case, such as its signal values and test objectives.
- If you set the **Mode** parameter to **Property proving**, the **CounterExamples** field lists information about each counterexample and the proof objective it falsifies.

The table describes the **TestCases** and **CounterExamples** fields.

| Subfield Name         | Description  |
|-----------------------|--|
| <b>timeValues</b>     | Vector of time values associated with signals in test case or counterexample.  |
| <b>dataValues</b>     | Vector of data values associated with signals in test case or counterexample.  |
| <b>paramValues</b>    | Structure representing details of parameters associated with test case or counterexample containing these fields:<br><br><b>name</b> — Parameter name.<br><br><b>value</b> — Parameter value.<br><br><b>noEffect</b> — Logical to signify if parameter value affects an objective.   |
| <b>stepValues</b>     | Vector that specifies the number of time steps that comprise signals in a test case or counterexample.   |
| <b>objectives</b>     | Structure that specifies objectives that a test case or a counterexample addresses. Its fields include:<br><br><b>objectiveIdx</b> — Integer that represents the index of an objective that a test case achieves or a counterexample falsifies.<br><br><b>atTime</b> — Time value at which either a test case achieves an objective or a counterexample falsifies an objective.<br><br><b>atStep</b> — Time step at which either a test case achieves an objective or a counterexample falsifies an objective. |
| <b>dataNoEffect</b>   | Cell array of logical vectors that specifies whether a signal's data values affect an objective. The vector uses <b>1</b> to indicate that a signal's data value does not affect an objective; otherwise, it uses <b>0</b> .   |
| <b>expectedOutput</b> | 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 <b>Output</b> block in the top-level system. This subfield is populated if you select <b>Include expected output values</b> .   |

**Version Field**

The **Version** field lists the of Simulink Design Verifier version used in the analysis.

## Dead Logic Field

If you analyze your model for dead logic by using the “Run a Partial Check for Dead Logic” on page 6-7 option, the `DeadLogic` field in the `sldvData` structure lists information about each dead logic objective.

This table describes each subfield of the `DeadLogic` field.

| Subfield Name             | Description   |
|---------------------------|---|
| <code>label</code>        | Description of dead logic objective.  |
| <code>descr</code>        | Full path to model object, including objects in Stateflow chart.                            |
| <code>modelObjIdx</code>  | Integer representing index of model object associated with objective.                       |
| <code>coverageType</code> | Type of coverage objective.   |
| <code>coverageIdx</code>  | Integer that represents the index of a coverage point that is associated with an objective. |
| <code>ObjectiveIdx</code> | Integer that represents the index of an objective that is associated with a model object.   |

## Simulate Models Using Data Files

You can use the `sldvruntest` function to simulate a model by using test cases or counterexamples that reside in a Simulink Design Verifier data file. Complete these steps to simulate the `sldvdemo_flipflop` model using a test case in its data file.

- 1 Simulate the `sldvdemo_flipflop` model and generate test cases:

```
sldvdemo_flipflop
```

- 2 Save the location of the data file that Simulink Design Verifier generates after analyzing the model.

```
sldvDataFile = 'sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat'
```

- 3 Use the `sldvruntest` function to simulate the `sldvdemo_flipflop` model using the second test case in the data file:

```
[ outdata ] = sldvruntest('sldvdemo_flipflop', sldvDataFile, 2)
```

The `sldvruntest` outputs is an array of `Simulink.SimulationOutput` objects.

- 4 Examine the output data from the first test case using the `Simulink.SimulationOutput` object:

```
tout_sldvruntest = outdata(1).find('tout_sldvruntest');
xout_sldvruntest = outdata(1).find('xout_sldvruntest');
yout_sldvruntest = outdata(1).find('yout_sldvruntest');
logcout_sldvruntest = outdata(1).find('logcout_sldvruntest');
```

## Load Results from Data Files

You can load the results of a previous analysis of a model from a data file. For more information, see “Load Previous Results” on page 13-57 and `sldvloadresults`.

**See Also**

“Review Analysis Results” on page 13-57 | sldvreport | “Load Previous Results” on page 13-57

## Manage Simulink Design Verifier Harness Models

### In this section...

“Harness Model Generation” on page 13-13  
 “Create a Harness Model” on page 13-13  
 “Contents of a Harness Model” on page 13-13  
 “Configuration of the Harness Model” on page 13-19  
 “Simulate the Harness Model” on page 13-19

### Harness Model Generation

A harness model provides an isolated environment to test design changes. You can create a harness model during Simulink Design Verifier analysis or after the analysis.

The contents of the harness model depends on the value of the **Mode** parameter, set in the Configuration Parameters dialog box on the **Design Verifier** pane:

- **Design error detection** — The harness model contains the test cases that result in errors during simulation.
- **Test generation** — The harness model contains the test cases that achieve test objectives.
- **Property proving** — The harness model contains counterexamples that falsify the proof objectives.

By default, the **Generate separate harness model after analysis** parameter is disabled.

---

**Note** The Simulink Design Verifier software generates a harness model only when the top-level model that you are analyzing contains an Inport block.

---

### Create a Harness Model

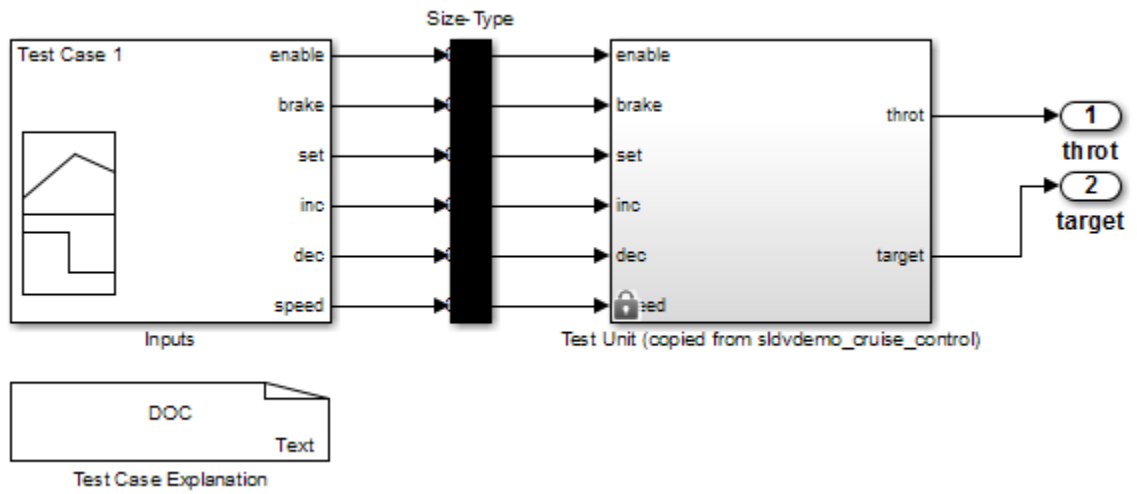
To create a harness model before or after the analysis, use these methods:

- Before the analysis, in the Configuration Parameters dialog box, on the **Design Verifier > Results** pane, select **Generate separate harness model after analysis**.
- After the analysis, in the Simulink Design Verifier Results Summary window, select **Create harness model**.

### Contents of a Harness Model

Simulink Design Verifier software creates a harness model that contains these items:

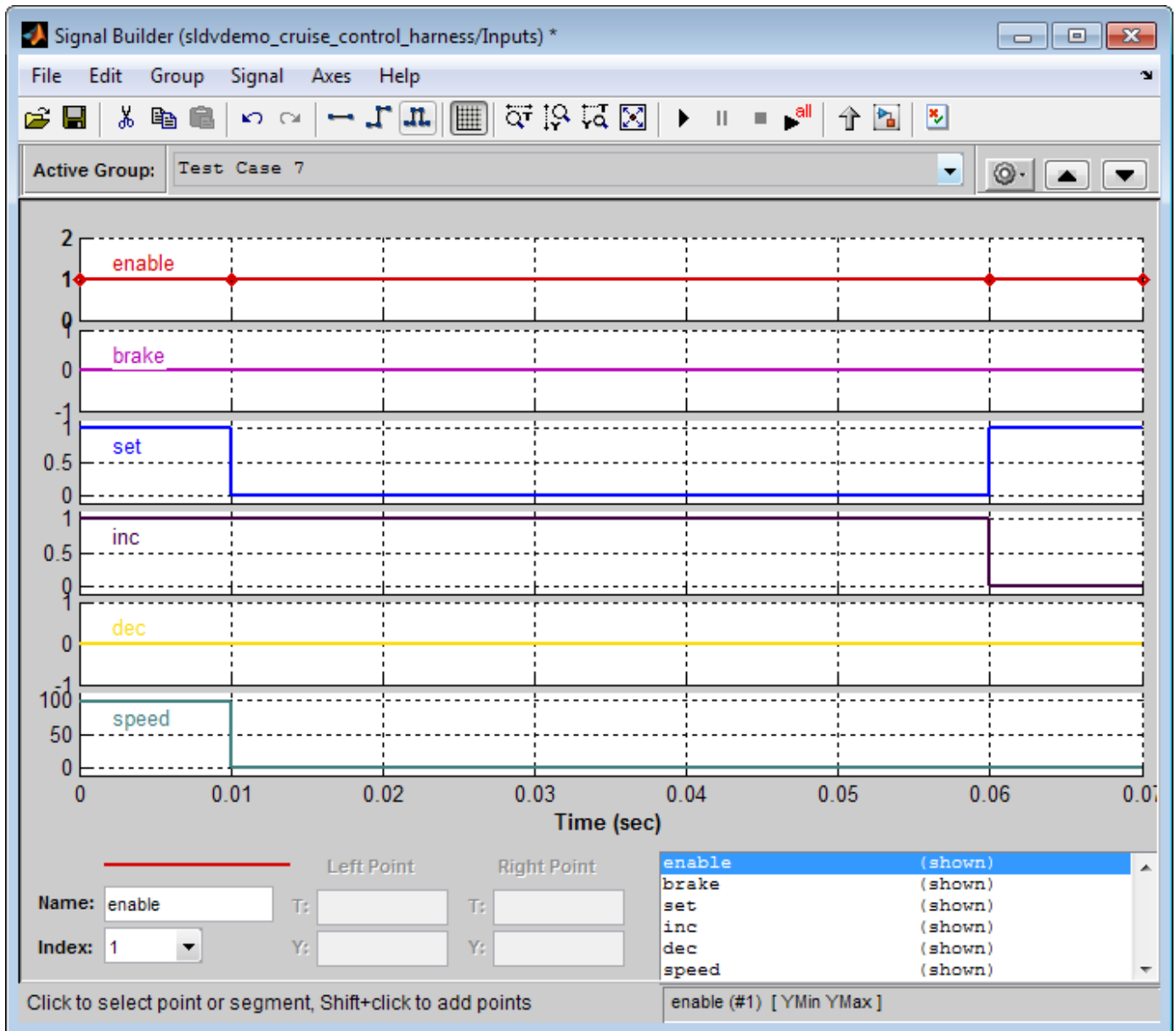
- **Inputs** — The Inputs block is a Signal Builder or Signal Editor block based on the “Harness source” on page 15-61 option set in the **Design Verifier > Results** pane.
  - **Signal Builder:** This block contains signals that are comprised of the test cases or counterexamples that Simulink Design Verifier generates. The Signal Builder block contains signals only for input signals that are used in the model. If an input signal has no effect on the output of the model, that signal is not included in the Signal Builder block.



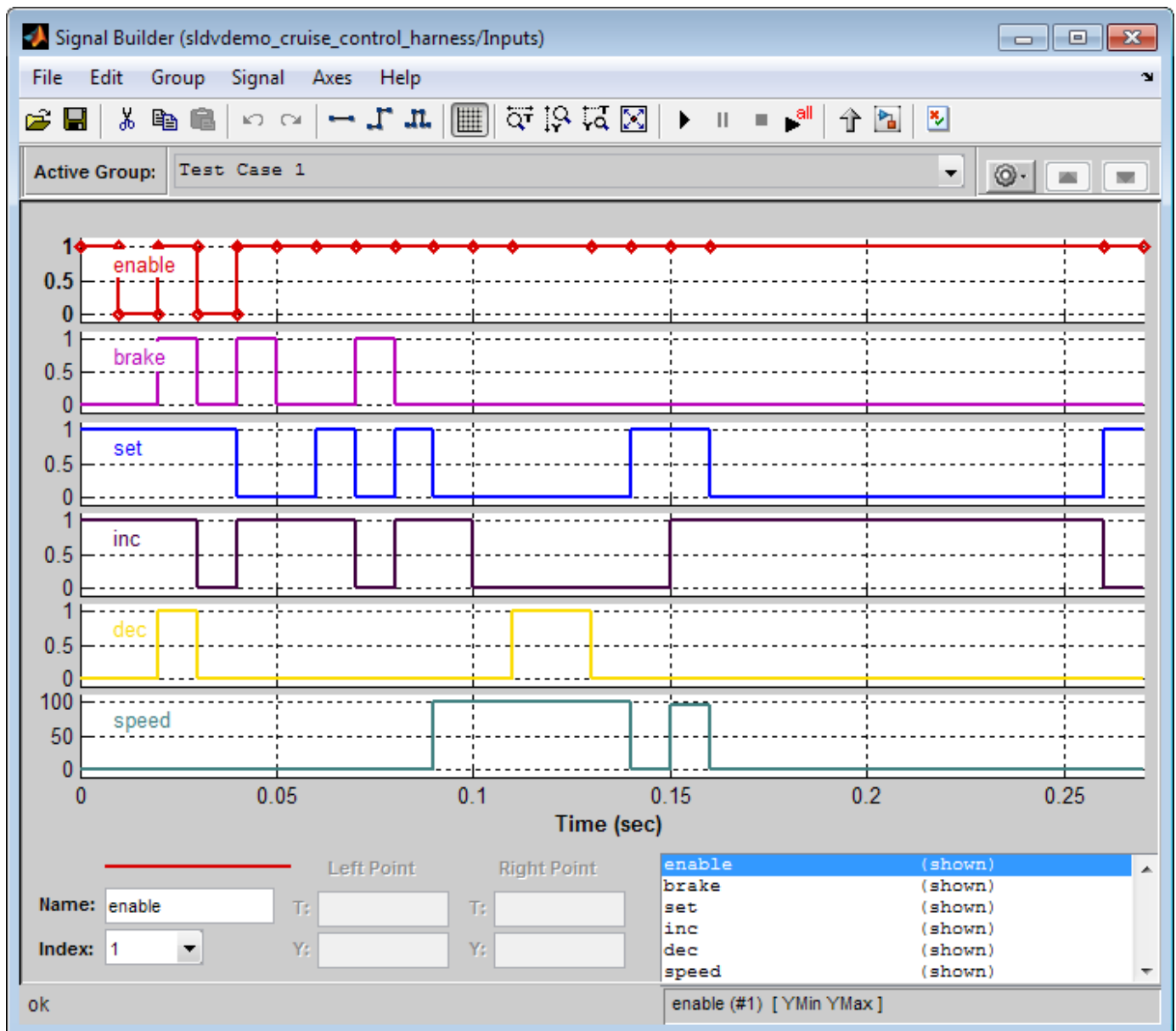
To open the Signal Builder dialog box and view its signals, double-click the Inputs block. Each signal group represents a unique test case or counterexample. To view the signals associated with a particular test case or counterexample, in the Signal Builder dialog box, select **Active Group**.

After Simulink Design Verifier performs test generation analysis on the `sldvdemo_cruise_control` model with the default options, this Signal Builder block shows the signals for Test Case 7.



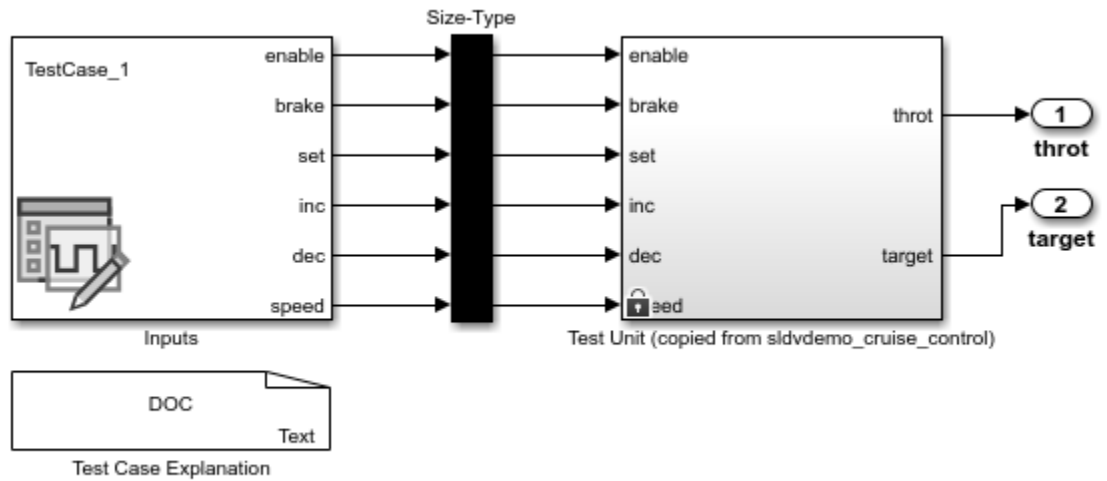


If you select the LongTestcases option of the **Test suite optimization** parameter, the analysis creates fewer, longer test cases. For example, if you select the LongTestcases option for the sldvdemo\_cruise\_control model, the analysis produces one long test case instead of nine shorter test cases. This Signal Builder dialog box shows the signals for the long test case. For more information about the Signal Builder dialog box, see "Signal Groups".

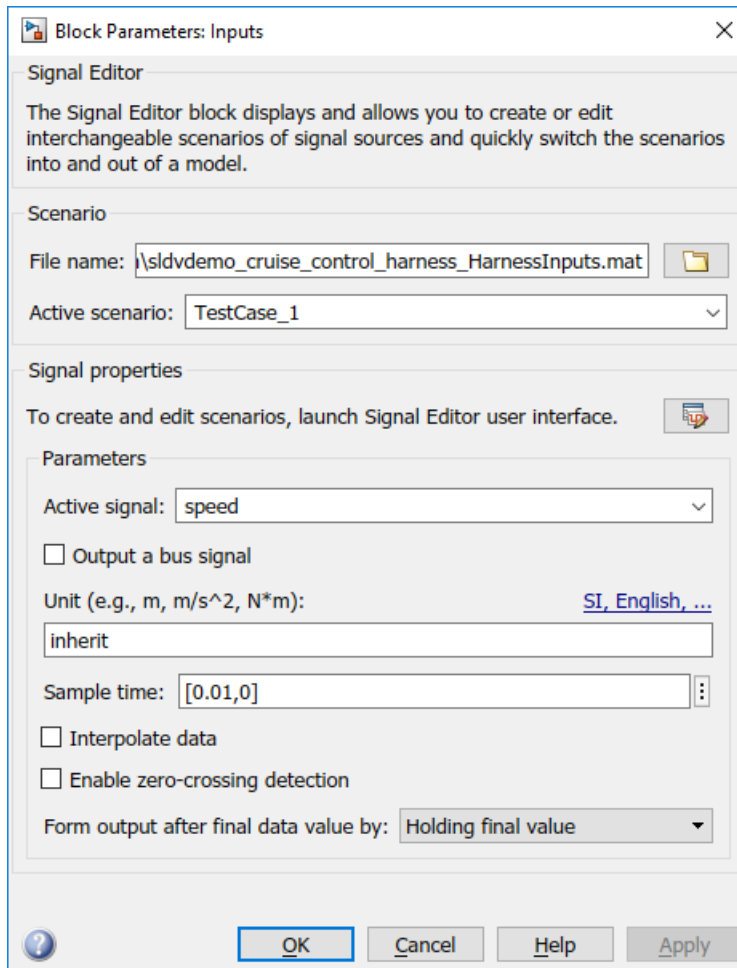


- Signal Editor: This block contains scenarios that are comprised of the test cases or counterexamples that Simulink Design Verifier generates. The Signal Editor block contains signals only for input signals that are used in the model. If an input signal has no effect on the output of the model, that signal is not included in the Signal Editor block.

After Simulink Design Verifier generates harness model, the input MAT-file for the Signal Editor block is saved at the default location `<current_folder>\sldv_output \<model_name>\<model_name>_harness_HarnessInputs.mat`.



To open the Signal Editor dialog box and view the scenarios of signal sources, double-click the Inputs block. The **Active scenario** lists the test cases or counterexamples. To create and edit scenarios, launch the Signal Editor user interface. For more information, see "Create and Edit Signal Data".



- **Size-Type** — This Subsystem block transmits signals from the Inputs block to the Test Unit block. It verifies that the size and data type of the signals are consistent with the Test Unit block.
- **Test Unit** — This Model block references the original model that Simulink Design Verifier analyzed.

If you do not select the **Reference input model in generated harness** on the **Design Verifier > Results** pane in the **Configuration Parameters** dialog box, the **Test Unit** created is a Subsystem block.

If the Test Unit in the harness model is a subsystem, the values of the parameters on the **Optimization** and **Math and Data Types** panes might impact the coverage results.

- **Test Case Explanation** — This DocBlock block documents the test cases or counterexamples that Simulink Design Verifier generates. To view the description of each test case or counterexample, double-click the Test Case Explanation block. The block lists either the test objectives that each test case achieves or the proof objectives that each counterexample falsifies.

```

1 Test Case 1 (8 Objectives)
2   Parameter values:
3
4   1. Controller/Switch3 - logical trigger input false (output is from 3rd input port) @ T=0.00
5   2. Controller/Switch2 - logical trigger input true (output is from 1st input port) @ T=0.00
6   3. Controller/Switch1 - logical trigger input true (output is from 1st input port) @ T=0.00
7   4. Controller/Logical Operator1 - Logic: input port 1 false @ T=0.00
8   5. Controller/Logical Operator2 - Logic: input port 1 true @ T=0.00
9   6. Controller/Logical Operator - Logic: input port 1 false @ T=0.00
10  7. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C1 (Logical Operator In1) false @ T=0.00
11  8. Controller/PI Controller - enable logical value false @ T=0.00
12
13 Test Case 2 (4 Objectives)
14   Parameter values:
15
16   1. Controller/Logical Operator1 - Logic: input port 1 true @ T=0.00
17   2. Controller/Logical Operator - Logic: input port 1 true @ T=0.00
18   3. Controller/Logical Operator - Logic: input port 2 false @ T=0.00
19   4. Controller/Logical Operator - (C1 && ~C2) && (C3 || C4) with C2 (Logical Operator1 In1) false @ T=0.00
20
21 Test Case 3 (1 Objectives)
22   Parameter values:
23
24   1. Controller/Switch2 - logical trigger input false (output is from 3rd input port) @ T=0.00
25
26 Test Case 4 (1 Objectives)
27   Parameter values:
28
29   1. Controller/Switch3 - logical trigger input true (output is from 1st input port) @ T=0.00
30
31 Test Case 5 (6 Objectives)
32   Parameter values:

```

## Configuration of the Harness Model

Simulink Design Verifier generates the harness model with these settings.

- The harness model start time is always 0. If the original model uses a nonzero start time, the software ignores the start time and uses 0 for the simulation start time for test cases and counterexamples.
- The harness model stop time always equals the stop time of the longest test case in the Inputs block.
- By default, the software enables coverage analysis and generates a coverage report for the harness models that contain test cases. The coverage reporting is enabled with default options. You can customize these settings by using “Specify Coverage Options” (Simulink Coverage).
- By default, if you select **Ignore objective based on filter** and provide a coverage filter file for the Test Unit, the coverage filter file applies to the harness model. For more information, see “Coverage data” on page 15-38.
- For models that use the complex type Inport block, a Signal Editor block is used as the harness source regardless of the **Harness source** that you specify.

## Simulate the Harness Model

The harness model enables you to simulate a copy of your original model by using the test cases or counterexamples that Simulink Design Verifier generates. Using the harness model, you can simulate:

- A counterexample.

- A single test case, for which the Simulink Coverage software collects and displays model coverage information.
- All the test cases, for which the Simulink Coverage software collects and displays cumulative model coverage information.

---

**Note** If you analyze a model that is simulated with sample time warnings, when you simulate the harness model, the warnings might be reported as errors, causing the simulation to fail.


---

### **Simulate Harness Model by Using the Signal Builder Source Block**

To simulate a single test case or counterexample:


- 1 In the harness model, double-click the Inputs block.
- 2 In the Signal Builder dialog box, select the **Active Group** with a particular test case or counterexample.

The Signal Builder dialog box displays the signals that comprise the selected test case or counterexample.

- 3 Click the **Start simulation** button .

The Simulink software simulates the harness model by using the signals associated with the selected test case or counterexample. When simulating a test case, the Simulink Coverage software collects model coverage information and displays a coverage report.

To simulate all test cases and measure their combined model coverage:

- 1 In the harness model, double-click the Inputs block.
- 2 In the Signal Builder dialog box, click the **Run all** button .

The Simulink software simulates the harness model by using all test cases, while the Simulink Coverage software collects model coverage information and displays a coverage report.

When you click **Run all**, the software simulates all the test cases by using the stop time for the harness model. The stop time equals the stop time for the longest test case, so you might accumulate additional coverage when you simulate the shorter test cases.

For more information, see “Simulating with Signal Groups”.

### **Simulate Harness Model by Using the Signal Editor Inputs Block**

To simulate a single test case or counterexample:

- 1 In the harness model, double-click the Inputs block.
- 2 In the Signal Editor dialog box, select the **Active scenario** with a particular test case or counterexample and click **OK**.
- 3 In the Simulink editor, click the **Run** button.

The Simulink software simulates the harness model by using the scenario of signal sources associated with the selected test case or counterexample. When simulating a test case, the Simulink Coverage software collects model coverage information and displays a coverage report.

To simulate all the test cases and measure their combined model coverage, use `cvsim` (Simulink Coverage) or `parsim` command. For example, see Simulate Harness Model with Signal Editor Inputs Block on page 13-22.

**See Also**

“Creating and Executing Test Cases” on page 7-100 | “Create Harness Model” on page 1-12

## Simulate Harness Model with Signal Editor Inputs Block

This example shows how to generate model coverage report by simulating the test harness model with the Signal Editor Inputs block. You can simulate a single test case or counterexample by selecting the active scenario in the Signal Editor dialog box. For more information see, “Simulate Harness Model by Using the Signal Editor Inputs Block” on page 13-20.

To simulate all the test cases and measure their combined model coverage, use the `cvsim` or the `parsim` command.

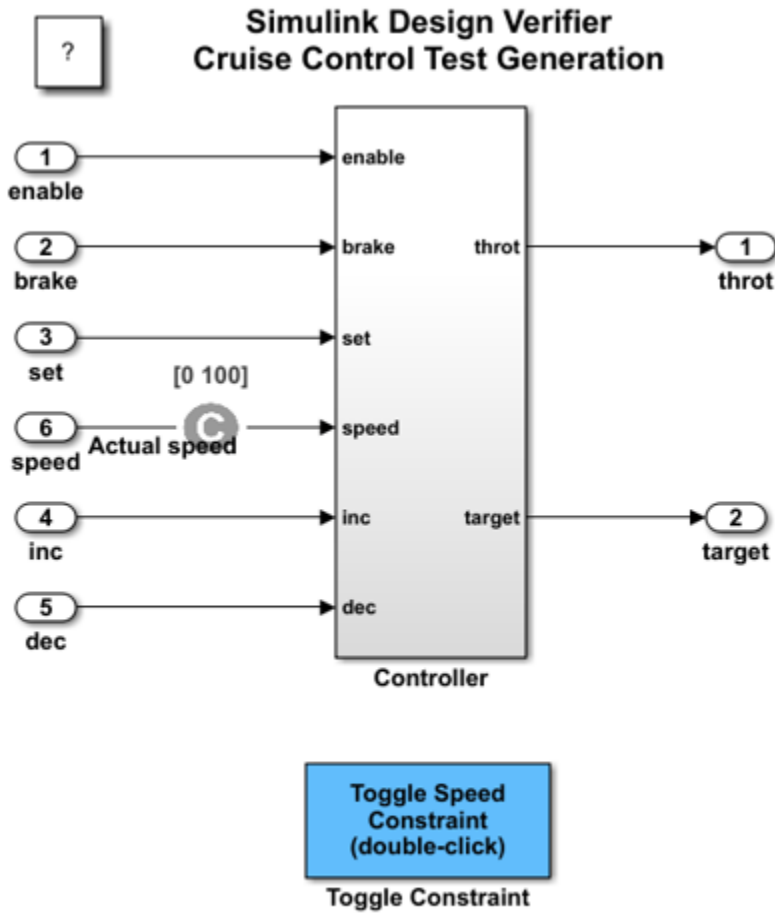
In this example, you generate a harness model by selecting the Signal Editor as the harness source. The Signal Editor scenarios consists of signal sources that are associated with the test cases or counterexamples. Then, to generate combined model coverage report, you simulate all the scenarios by using the `cvsim` or `parsim` function.

### 1. Open the model and configure harness options

Create a harness model for the `sldvdemo_cruise_control` model by using the `sldvharnessopts` options. Set the `HarnessSource` option to `Signal Editor`.

```
model = 'sldvdemo_cruise_control';
open_system(model);
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.SaveHarnessModel = 'on';
opts.HarnessSource = 'Signal Editor';
opts.HarnessModelFileName = 'sldvdemo_cruise_control_harness';
opts.SaveReport = 'off';
```





Copyright 2006-2019 The MathWorks, Inc.

## 2. Generate test cases

Analyze the model by using the `sldvrun` function and `sldvoptions`.

```
sldvrun('sldvdemo_cruise_control', opts);
save_system('sldvdemo_cruise_control_harness');
```

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

```
'sldvdemo_cruise_control' is compatible for test generation with Simulink Design Verifier.
```

```
Generating tests using model representation from 31-Dec-2021 02:59:04...
.....
```

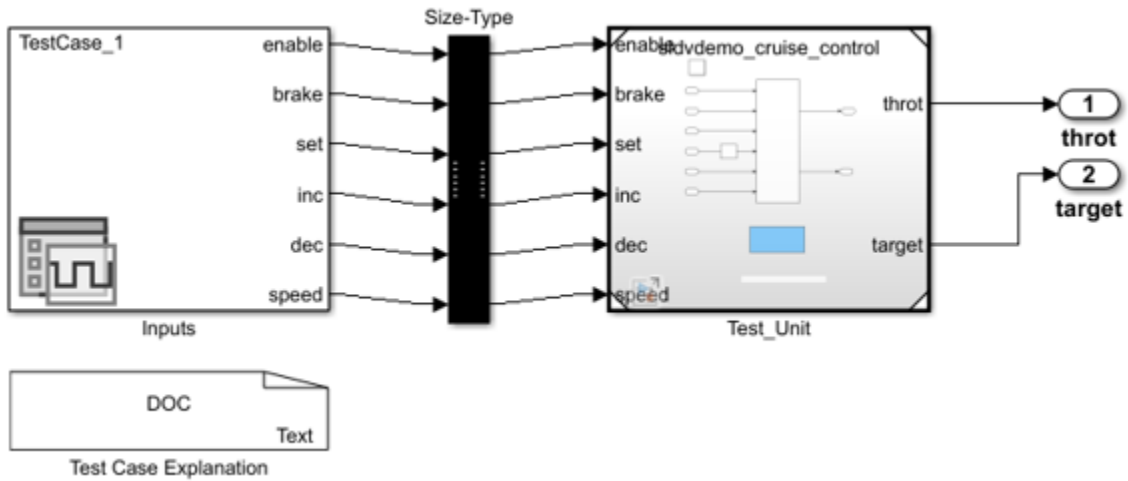
```
Completed normally.
```

```
Generating output files:
```

The analysis did not produce a harness model.  
 Unable to read MAT-file C:\Users\pdasbasu\AppData\Roaming\MathWorks\MATLAB\R2022a\matlabprefs.mat

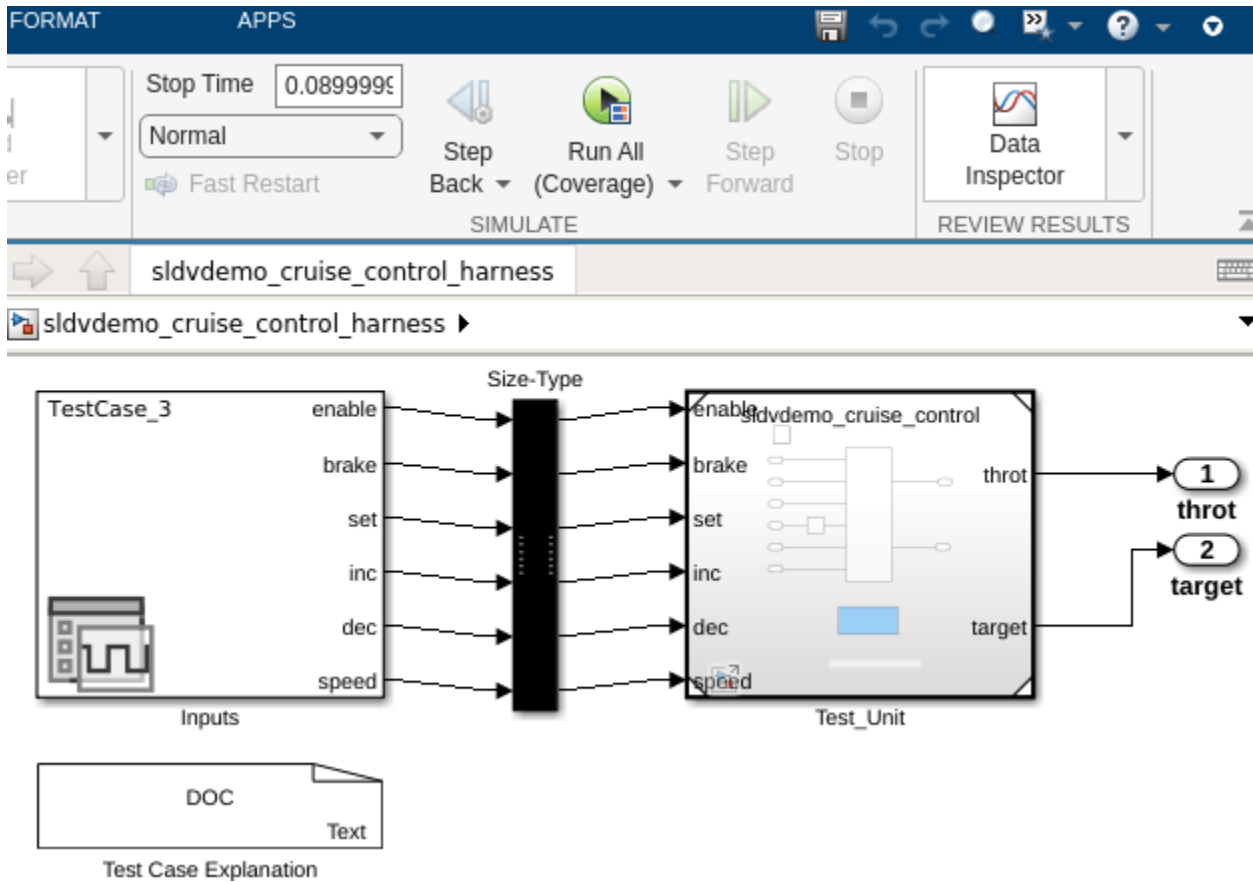
Results generation completed.

Data file:  
 C:\Users\pdasbasu\OneDrive - MathWorks\Documents\MATLAB\ExampleManager\pdasbasu.Bdoc22a.j183



**3. Generate combined model coverage report**

Simulink Design Verifier automatically configures Signal Editor harness in multiple simulation mode. To simulate the generated test cases and gather coverage for Test Unit, click **Run all (Coverage)** button on Simulation toolstrip menu.



Alternatively, after the analysis generates the harness model, you can use this code that uses `cvtest` and `cvsim` functions to generate the combined model coverage report.

```

signalEditorBlock = 'sldvdemo_cruise_control_harness/Inputs';
numOfScenarios = str2double(get_param(signalEditorBlock, 'NumberOfScenarios'));
harnessModel = 'sldvdemo_cruise_control_harness';
test = cvtest(harnessModel);
test.modelRefSettings.enable = 'On';
test.modelRefSettings.excludeTopModel = 1;
covData = [];
for id = 1:numOfScenarios
    set_param(signalEditorBlock, 'ActiveScenario', id);
    aCovData = cvsim(harnessModel);
    if isempty(covData)
        covData = aCovData;
    else
        covData = covData + aCovData;
    end
end
save_system('sldvdemo_cruise_control_harness');
cvhtml('Coverage_Harness', covData);

```

Optionally, you can use this code that uses the `parsim` function to generate the combined model coverage report.

```

signalEditorBlock = 'sldvdemo_cruise_control_harness/Inputs';
numOfScenarios = str2double(get_param(signalEditorBlock, 'NumberOfScenarios'));
harnessModel = 'sldvdemo_cruise_control_harness';

simIn = Simulink.SimulationInput.empty(0,numOfScenarios);
for id = 1:numOfScenarios
    simIn(id) = Simulink.SimulationInput(harnessModel);
    simIn(id) = simIn(id).setBlockParameter(signalEditorBlock, 'ActiveScenario', id);
    simIn(id) = simIn(id).setModelParameter('CovEnable', 'on');
    simIn(id) = simIn(id).setModelParameter('CovSaveSingleToWorkspaceVar', 'on');
end

simOut = parsim(simIn);
cvhtml('Coverage_Harness', simOut.covdata);

```

```










[31-Dec-2021 02:59:45] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
[31-Dec-2021 03:00:49] Starting Simulink on parallel workers...
[31-Dec-2021 03:01:16] Configuring simulation cache folder on parallel workers...
[31-Dec-2021 03:01:17] Loading model on parallel workers...
[31-Dec-2021 03:01:26] Running simulations...
[31-Dec-2021 03:01:45] Completed 1 of 3 simulation runs
[31-Dec-2021 03:01:45] Completed 2 of 3 simulation runs
[31-Dec-2021 03:01:45] Completed 3 of 3 simulation runs
[31-Dec-2021 03:01:45] Cleaning up parallel workers...

```

The coverage report indicates that 100% coverage is achieved by simulating all the test cases for `sldvdemo_cruise_control_model`.

## Summary

### Model Hierarchy/Complexity Test 1

|   | Decision   | Condition  | Test Objective | Proof Objective | Test Condition   | Proof Assumption | Execution  |
|---|--|--|----------------|-----------------|--|------------------|--|
| 1. <code>sldvdemo_cruise_control</code> | 8 100%  | 100%  | NA             | NA              | 100%  | NA               | 100%  |
| 2. .... <code>Controller</code>         | 7 100%  | 100%  | NA             | NA              | NA   | NA               | 100%  |
| 3. .... <code>PI Controller</code>      | 4 100%  | NA   | NA             | NA              | NA   | NA               | 100%  |

## 5. Clean Up

```

% To complete this example, close the models.
close_system('sldvdemo_cruise_control_harness', 0);
close_system('sldvdemo_cruise_control', 0);

```

## Export Test Cases to Simulink Test

Model verification often requires repeated testing to achieve certain objectives or coverage criteria. If you run repeated tests, consider using the Test Manager in Simulink Test to structure your test cases, archive test results, and generate reports. You can generate test cases using Simulink Design Verifier and export the test inputs to new test cases automatically created in the Simulink Test Manager.

To export generate inputs to new test cases in Simulink Test:

- 1 Choose an existing Simulink Design Verifier results file or generate new results by analyzing your model.
  - If you use an existing results file, you can load results by either:
    - Using the Simulink Test command `sltest.import.sldvData`.
    - Using **Load Earlier Results** in the **Design Verifier** tab. Select the MAT-file or Excel<sup>®</sup> file with the analysis results.
  - If you run a model analysis, the Design Verifier Results Summary window appears after the analysis completes.
- 2 In the results summary window, click **Export test cases to Simulink Test**. The Export Design Verifier Test Cases dialog box opens.
- 3 In the Export Design Verifier Test Cases dialog box, you can:
  - Choose **Harness Source to Inport, Signal Editor or Signal Builder**.
  - Set the **Test Data Format** to MAT or Excel.
  - Click **OK** to generate the test file and test harness.
- 4 Simulink Test generates the test file and test harness. In the Test Manager, expand the new test file in the **Test Browser** to see the individual test cases.

## Generate and Export Test Cases to Simulink Test

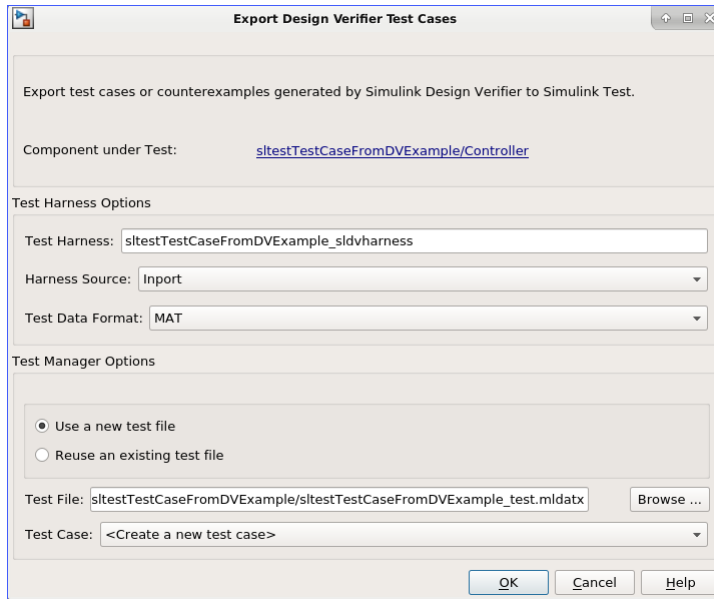
This example shows how to generate test cases to achieve coverage objectives for a controller subsystem. The example also shows how to add functional test cases from test harnesses in the model. This example requires a Simulink Test license.

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

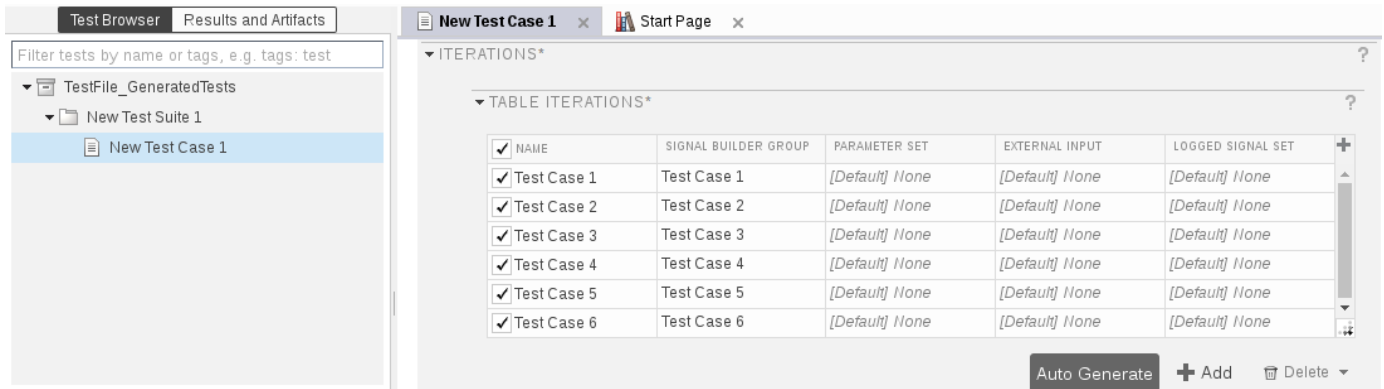
- 1 Open the model.
 

```
open_system('sltestTestCaseFromDVExample.slx');
```
- 2 Set the current working folder to a writable folder.
- 3 In the model, generate tests for the Controller subsystem. Right-click the Controller block and select **Design Verifier > Generate Tests for Subsystem**.
- 4 In the Simulink Design Verifier Results Summary window, click **Export test cases to Simulink Test**.

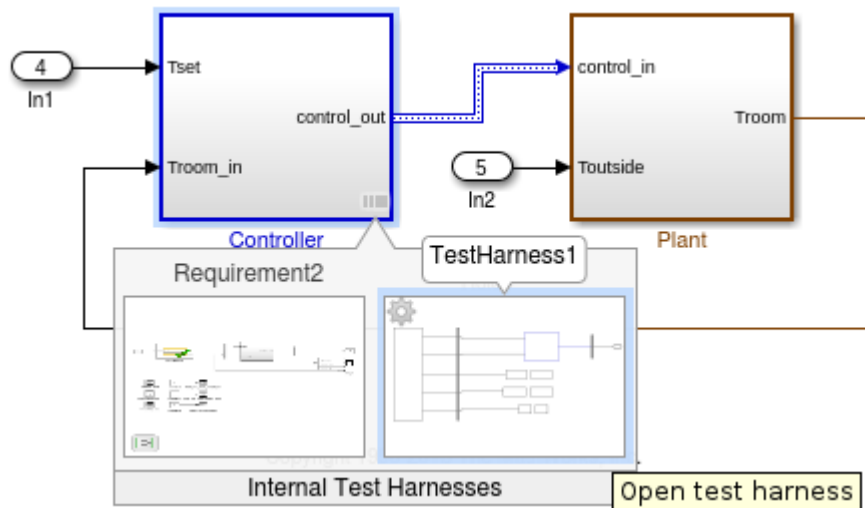
- In the Export Design Verifier Test Cases dialog box, click **OK**.




The Test Manager displays six new test cases in the test file.



- In the model, click the perspective view badge to see the new test harness.



- 7 Add a test case to the other test harness in the model. In the Test Manager, point to the new test file name and click the Synchronize Test File button .
- 8 The Test Manager prompts you to add tests for the Requirement2 test harness. Select Simulation for the test type and click **Update Test File**.

The Test Manager adds the Requirement2 test case to the test file.

## See Also

sltest.import.sldvData

## Export Tests from Models That Contain Requirements Table Blocks with Simulink Design Verifier

If you create models that contain Requirements Table blocks and you generate tests with Simulink Design Verifier, you can export the tests to the **Test Manager**. When configuring the tests, you can specify the test harness that you want to use. After running the tests, you can determine if the tests fail or pass, and inspect the results further.

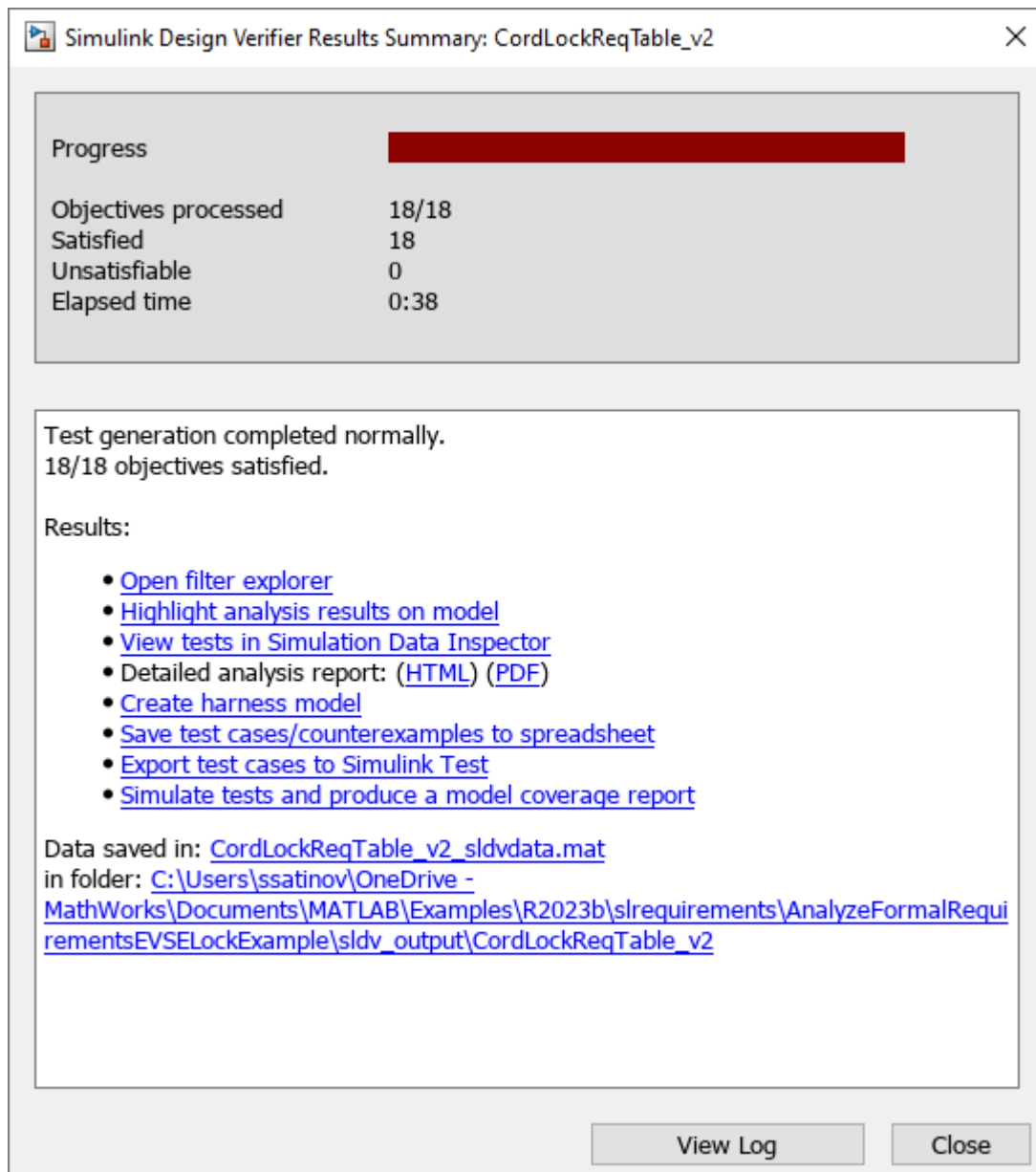
### Construct the Model and Generate Tests

Simulink Design Verifier creates test objectives from the requirements defined in Requirements Table blocks. To generate tests, construct a model, called the specification model, with Requirements Table blocks and Simulink blocks that do not define test objectives. See “What Is a Specification Model?” on page 7-60. After you create the requirements, confirm that the requirement set in each Requirements Table block is complete and consistent by analyzing them. See “Identify Inconsistent and Incomplete Formal Requirement Sets” (Requirements Toolbox). If you do not create complete and consistent requirements, Simulink Design Verifier may not be able to create tests that satisfy the test objectives.

After constructing the requirements, generate tests in the specification model.

- 1 In the **Apps** tab, click **Design Verifier**.
- 2 In the **Mode** section, set the mode to **Test Generation**.
- 3 In the **Prepare** section, click **Test Generation Settings**. The Requirements Table block supports test generation with decision, condition, MCDC, enhanced MCDC, and relational boundary coverage objectives. Specify the objectives in the **Model coverage objectives** parameter. For more information on these options, See “Model Coverage Objectives for Test Generation” on page 7-30. Click **OK**.
- 4 In the **Analyze** section, click **Generate Tests**. Simulink Design Verifier indicates how many objectives from the requirements are satisfied.





- 5 If at least one of the objectives is not satisfied, update your requirements. If you did not analyze the requirements before, analyze them now.

## Export the Tests to the Test Manager

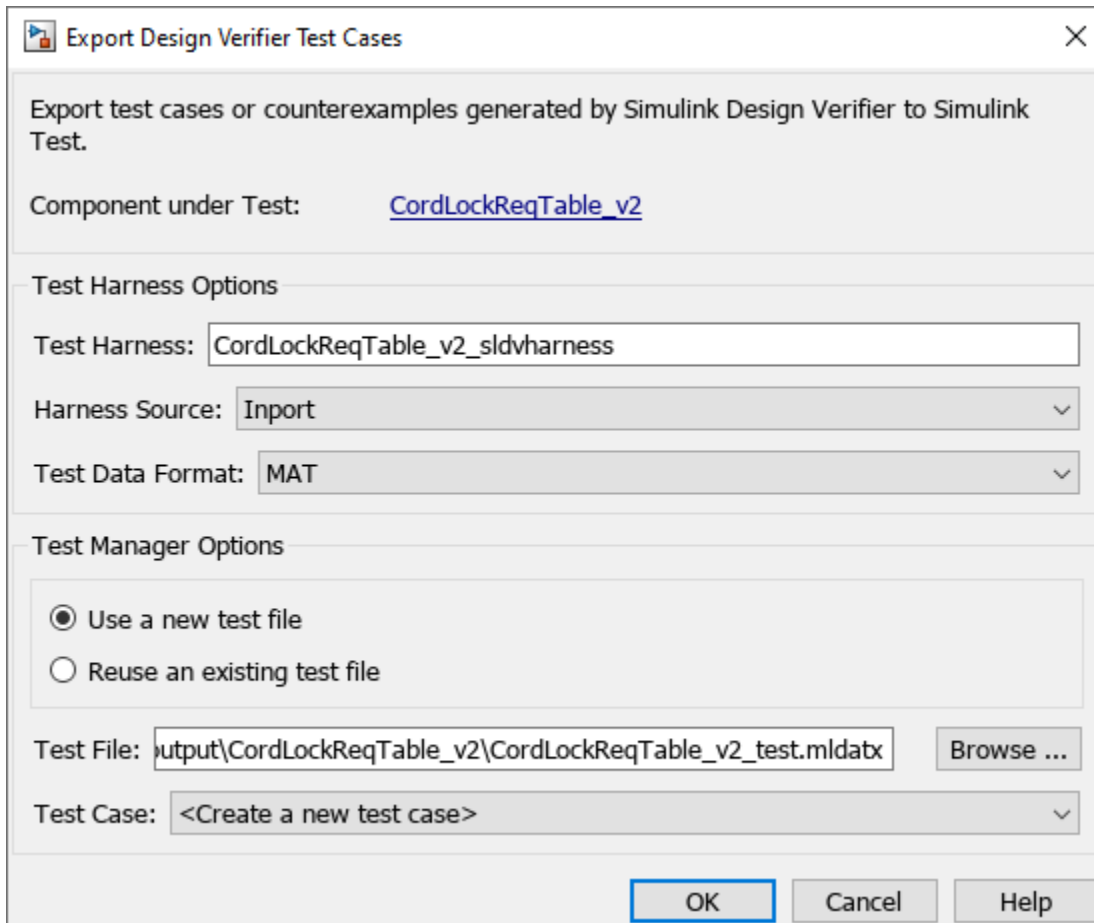
If you have Simulink Test, you can export the tests to the **Test Manager**. In the Simulink Design Verifier Results Summary window, click **Export test cases to Simulink Test**.

Test generation completed normally.  
18/18 objectives satisfied.

Results:

- [Open filter explorer](#)
- [Highlight analysis results on model](#)
- [View tests in Simulation Data Inspector](#)
- Detailed analysis report: ([HTML](#)) ([PDF](#))
- [Create harness model](#)
- [Save test cases/counterexamples to spreadsheet](#)
- [Export test cases to Simulink Test](#)
- [Simulate test and produce a model coverage report](#)

The Export Design Verifier Test Cases window displays the properties that you can adjust before exporting.



For more information on the properties you can select, see “Export Test Cases to Simulink Test” on page 13-27.

## Run the Tests

After exporting the tests, Simulink Test registers the requirements in the Requirements Table blocks to the test cases they generate. To view these assignments, open the **Test Manager**. Then select the test case in the **Test Browser** pane. In the **Test Case** pane, expand the **Iterations** section.

▼ ITERATIONS\* ?

▼ TABLE ITERATIONS\* ?

| <input checked="" type="checkbox"/> NAME        | DESCRIPTION | REQUIREMENTS  | EXTERNAL INPUT |
|---|-------------|---|----------------|
| <input checked="" type="checkbox"/> Test Case:1 | None        | <a href="#">Requirement 1: Lock when evse c...</a>  | Test Case:1    |
| <input checked="" type="checkbox"/> Test Case:2 | None        | <a href="#">Requirement 6: Unlock during em...</a>  | Test Case:2    |
| <input checked="" type="checkbox"/> Test Case:3 | None        | <a href="#">Requirement 2: Unlock during nor...</a> | Test Case:3    |
| <input checked="" type="checkbox"/> Test Case:4 | None        | <a href="#">Requirement 7: Unlock SessionSto...</a> | Test Case:4    |
| <input checked="" type="checkbox"/> Test Case:5 | None        | <a href="#">Requirement 9: Unlock when com...</a>   | Test Case:5    |
| <input checked="" type="checkbox"/> Test Case:6 | None        | <a href="#">Requirement 3: Unlock during em...</a>  | Test Case:6    |

Auto Generate + Add Delete ▼

If you have a manually created test harness that you want to run the tests on, in the **Test Browser** pane, select the test case and expand **System Under Test**. In the **Model** field, specify the model, and clear the model from the **Harness** field.

## Inspect Test Failures

If one of your tests fail, you may need investigate causes of the failure. If you use verification blocks in your harness to verify the outputs of your design and specification model, or if you capture design model outputs in requirement postconditions, you can use property proving on the test harness and run **Model Slicer** to identify the conditions that cause an assertion failure.

- 1 Open the test harness model.
- 2 In the **Design Verifier** tab, in the **Mode** section, select **Property Proving**.
- 3 In the **Prepare** section, click **Property Proving Settings**. If your specification model uses at least one postcondition, the Requirements Table block highlights the postconditions green if the associated proof objectives are satisfied, red if they are not, and orange for other conditions.

| Requirements |  | Assumptions  |               |
|--------------|--|--|---------------|
| Index        | Summary  | Precondition   | Postcondition |
|              |  |  | deploy        |
| 1            | The thrust reverser system shall not deploy if the average airspeed is greater than 150 knots. | $\text{mean}(\text{airspeed}) > 150$                           | false         |
| 2            | The thrust reverser system shall not deploy if any two WOW sensors are false.                  | $\text{sum}(\text{wow}) \geq 3$                                | false         |
| 3            | The thrust reverser system shall not deploy if the two thrust sensors are greater than 0.      | $\text{sum}(\text{throttle}) \geq 3$                           | false         |
| 4            | The thrust reverser system shall not deploy if either wheelspeed sensor is less than 10 knots. | $\text{wheelspeed}(1) < 10 \ \&\& \ \text{wheelspeed}(2) < 10$ | false         |

If you use verification blocks, Simulink Design Verifier highlights the blocks that assert a failure in red.

- If you use verification blocks, select the highlighted verification block. In the Results window, click **debug**. The model displays the values that correspond to each signal that caused the failure. These values only display outside of the Requirements Table block.

For more information, see “Debug Property Proving Violations by Using Model Slicer” on page 12-55 and “Prove Properties with Requirements Table Blocks” on page 12-73.

## See Also

Requirements Table

## Related Examples

- “Use a Requirements Table Block to Create Formal Requirements” (Requirements Toolbox)
- “What Is Property Proving?” on page 12-2
- “What Is a Specification Model?” on page 7-60
- “Use Specification Models for Requirements-Based Testing” on page 7-69

## Review Results

### In this section...

“Simulink Design Verifier Report Generation” on page 13-35

“Create Analysis Reports” on page 13-35

“Front Matter” on page 13-35

“Summary Chapter” on page 13-36

“Analysis Information Chapter” on page 13-36

“Derived Ranges Chapter” on page 13-40

“Objectives Status Chapters” on page 13-42

“Model Items Chapter” on page 13-50

“Design Errors Chapter” on page 13-51

“Test Cases Chapter” on page 13-52

“Properties Chapter” on page 13-54

## Simulink Design Verifier Report Generation

After an analysis, Simulink Design Verifier can generate an HTML report that contains detailed information about the analysis results.

The analysis report contains hyperlinks that allow you to:

- Navigate to a specific part of the report
- Navigate to the object in your Simulink model for which the analysis recorded results

You can also generate an additional PDF version of the Simulink Design Verifier report.

## Create Analysis Reports

To create a detailed analysis report before or after the analysis, do one of the following:

- Before the analysis, in the Configuration Parameters dialog box, on the **Design Verifier > Report** pane, select **Generate report of the results**. If you want to save an additional PDF version of the Simulink Design Verifier report, select **Generate additional report in PDF format**.
- After the analysis, in the Simulink Design Verifier log window, you can choose HTML or PDF format and **Generate detailed analysis report**.

## Front Matter

The report begins with two sections:

- “Title” on page 13-35
- “Table of Contents” on page 13-36

### Title

The title section lists the following information:

- Model or subsystem name Simulink Design Verifier analyzed
- User name associated with the current MATLAB session
- Date and time that Simulink Design Verifier generated the report

### Table of Contents

The table of contents follows the title section. Clicking items in the table of contents allows you to navigate quickly to particular chapters in the report.

## Summary Chapter

The **Summary** chapter of the report lists the following information:

- Name of the model
- MATLAB release in which the analysis was performed
- Checksum value that represents the state of the model analyzed
- Analysis mode
- Model Representation
- Test generation target (applicable for test case generation analysis)
- Analysis status
- Preprocessing time
- Analysis time
- Status of objectives analyzed. This includes the percentage number for each status

| <b>Analysis Information</b>  |  |          |
|------------------------------|--|----------|
| Model:                       | sldvdemo_cruise_control                    |          |
| Release:                     | R2021a Prerelease                          |          |
| Checksum:                    | 863976376 4198703694 2414576155 2412704551 |          |
| Mode:                        | Test generation                            |          |
| Model Representation:        | Built on 09-Nov-2020 14:28:46              |          |
| Test Generation Target:      | Model                                      |          |
| Status:                      | Completed normally                         |          |
| PreProcessing Time:          | 4s   |          |
| Analysis Time:               | 25s  |          |
| <b>Objectives Status</b>     |  |          |
| <b>Number of Objectives:</b> | <b>32</b>                                  |          |
| Objectives Satisfied:        | 32   | ( 100% ) |

## Analysis Information Chapter

The **Analysis Information** chapter of the report includes the following sections:

- “Model Information” on page 13-37
- “Analysis Options” on page 13-37
- “Unsupported Blocks” on page 13-38
- “User Artifacts” on page 13-39
- “Constraints” on page 13-39
- “Block Replacements Summary” on page 13-39
- “Approximations” on page 13-39
- “Analysis Harness Information” on page 13-40

### **Model Information**

The Model Information section provides the following information about the current version of the model:

- Path and file name of the model that Simulink Design Verifier analyzed
- Model version
- Date and time that the model was last saved
- Name of the person who last saved the model

### **Analysis Options**

The Analysis Options section provides information about the Simulink Design Verifier analysis settings.

The Analysis Options section lists the parameters that affected the Simulink Design Verifier analysis. If you enabled coverage filtering, the name of the filter file is included in this section.

## Analysis Options

|  |                    |
|--|--------------------|
| Mode:  | TestGeneration     |
| Rebuild Model Representation:                                      | Always             |
| Test generation target:  | Model              |
| Test Suite Optimization:   | CombinedObjectives |
| Maximum Testcase Steps:  | 500time steps      |
| Test Conditions:   | UseLocalSettings   |
| Test Objectives:   | UseLocalSettings   |
| Model Coverage Objectives:   | MCDC               |
| Include Relational Boundary Objectives:                            | off                |
| Maximum Analysis Time:   | 300s               |
| Block Replacement:   | off                |
| Parameters Analysis:   | off                |
| Include expected output values:                                    | off                |
| Randomize data that do not affect the outcome:                     | off                |
| Additional analysis to reduce instances of rational approximation: | off                |
| Save Data:   | on                 |
| Save Harness:  | off                |
| Save Report:   | off                |

---

**Note** For more information about these parameters, see “Simulink Design Verifier Options” on page 15-2.

---

### Unsupported Blocks

If your model includes unsupported blocks, by default, automatic stubbing is enabled to allow the analysis to proceed. With automatic stubbing enabled, the software considers only the interface of the unsupported blocks, not their actual behavior. This technique allows the software to complete the analysis. However, the analysis may achieve only partial results if any of the unsupported model blocks affect the simulation outcome.

The Unsupported Blocks section appears only if the analysis stubbed unsupported blocks; it lists the unsupported blocks in a table, with a hyperlink to each block in the model.

| Block                                | Type               |
|--------------------------------------|--------------------|
| <a href="#">Discrete State-Space</a> | DiscreteStateSpace |

For more information about automatic stubbing, see “Handle Incompatibilities with Automatic Stubbing” on page 2-7.



## User Artifacts

The User Artifacts section provides information about test data and coverage data in the Simulink Design Verifier analysis.

## Constraints

The Constraints section provides information about test conditions that Simulink Design Verifier applied when it analyzed a model.

| Name                       | Analysis 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 Simulink Design Verifier executed. It appears only if Simulink Design Verifier replaced blocks in a model.

Each row of the table corresponds to a particular block replacement rule that Simulink Design Verifier applied to the model. The table lists the following:

- Name of the file that contains the block replacement rule and the value of the `BlockType` parameter the rule specifies
- Description of the rule that the `MaskDescription` parameter of the replacement block specifies
- Names of blocks that Simulink Design Verifier replaced in the model

To locate a particular block replacement in your model, click on the name for that replacement in the Replaced Blocks column of the table; the software highlights the affected block in your model window and opens a new window that displays the block in detail.

| #: | Replacement Rule / Block Type     | Rule Description   | Replaced Blocks   |
|----|-----------------------------------|--|---|
| 1  | blkrep_rule_switch_normal /Switch | Inserts test objectives for switch blocks that require each switch position be demonstrated when the values of input ports 1 and 3 differ. | <a href="#">Switch1</a><br><a href="#">Switch2</a><br><a href="#">Switch3</a> |

## Approximations

Each row of the Approximations table describes a specific type of approximation that Simulink Design Verifier used during its analysis of the model.

| # | Type                   | Description  |
|---|------------------------|--|
| 1 | Rational approximation | The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic. |

---

**Note** Review the analysis results carefully when the software uses approximations. In rare cases, an approximation may result in test cases that fail to achieve test objectives or counterexamples that fail to falsify proof objectives. For example, a floating-point round-off error might prevent a signal from exceeding a designated threshold value.

---

### Analysis Harness Information

The **Analysis Harness Information** section provides an overview of the analysis harness generated by Simulink Design Verifier used to analyze the model. The **Analysis Harness Information** section shows these sub-sections based on whether the model is an export-function model or a model that contains Function Caller blocks without corresponding Simulink functions.

#### Schedule for Export Function analysis

Simulink Design Verifier assumes an analysis harness for invoking **Export Functions** during analysis. For example, this table shows the analysis harness for the model `sldvExportFunction_autosar_multirunnables`:

| Order | Function-Call Inport      | Sample Time(sec) | Number of times invoked per sample hit |
|-------|---------------------------|------------------|--|
| 1     | <a href="#">Runnable1</a> | 1                | 1                                      |
| 2     | <a href="#">Runnable2</a> | 1                | 1                                      |
| 3     | <a href="#">Runnable3</a> | 10               | 1                                      |

#### Stubbed Simulink Functions for Analysis

This table in the **Stubbed Simulink Functions for Analysis** lists the function prototypes that correspond to the stubbed Simulink functions which are stubbed in the analysis harness:

| Function Prototype   |
|--|
| <a href="#">[EventFailed,ERR] = TPS2StuckLow_GetEventFailed()</a>  |
| <a href="#">[EventFailed,ERR] = TPS2StuckHigh_GetEventFailed()</a> |
| <a href="#">[EventFailed,ERR] = TPS1StuckLow_GetEventFailed()</a>  |
| <a href="#">[EventFailed,ERR] = TPS1StuckHigh_GetEventFailed()</a> |
| <a href="#">ERR = TPS_SetEventStatus(EventStatus)</a>              |

---

**Note** Simulink Design Verifier assumes the outputs of stubbed Simulink functions do not change when the function is invoked multiple times during a single time step.

---

### Derived Ranges Chapter

In a design error detection analysis, the analysis calculates the derived ranges of the signal values for the Outports for each block in the model. This information can help you identify the source of data overflow or division-by-zero errors.

The table in the **Derived Ranges** chapter of the analysis report lists these bounds.

| Signal  | Derived Ranges              |
|---|-----------------------------|
| <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/Switch. Defined by block replacement rule 'blkrep rule switch normal'.- output 1</a> | [-Inf..Inf]                 |
| <a href="#">Controller/Switch2/Switch. Defined by block replacement rule 'blkrep rule switch normal'.- output 1</a> | [-Inf..Inf]                 |
| <a href="#">Controller/Switch1/Switch. Defined by block replacement rule 'blkrep rule switch normal'.- 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..3.5954e+306] |
| <a href="#">target- output 1</a>  | [-Inf..Inf]                 |

If an Observer Reference block is used in the design error detection analysis, then this section will show the observer information in a **Observer Model (s)** subsection and design model information in **Design Model** subsection.

The table in the Design Model subsection shows the list of each derived range in the sldvdemo\_debounce\_validprop example model.

| Signal                                     | Derived Ranges |
|--|----------------|
| <a href="#">Constant- Output 1</a>         | 2              |
| <a href="#">Constant1- Output 1</a>        | 1              |
| <a href="#">Chart "debounce"- Output 1</a> | [F..T]         |
| <a href="#">Switch- Output 1</a>           | [1..2]         |
| <a href="#">debounced- Output 1</a>        | [1..2]         |

The Observer model(s) section will not show derived ranges reported as the observers are ignored for design error detection analysis.

## Objectives Status Chapters

This section of the report provides information about all the objectives in a model, including the type of the objective, the model item that corresponds to the type, and objective description.

- “Design Error Detection Objectives Status” on page 13-43
- “Test Objectives Status” on page 13-45
- “Proof Objectives Status” on page 13-47
- “Objectives Undecided due to Runtime Error” on page 13-49
- “Objectives Undecided Due to Division by Zero” on page 13-49
- “Objectives Undecided Due to Nonlinearities” on page 13-50
- “Objectives Undecided Due to Stubbing” on page 13-50
- “Objectives Undecided Due to Array Out of Bounds” on page 13-50
- “Objectives Undecided” on page 13-50

The software identifies the presence of approximations and reports them at the level of each objective status. For more information, see “How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23. This table summarizes the objective status for Simulink Design Verifier analysis modes.

| Analysis Mode          | Objective Status   |
|------------------------|--|
| Design error detection | <ul style="list-style-type: none"> <li>• “Dead Logic” on page 13-44</li> <li>• “Dead Logic under Approximation” on page 13-44</li> <li>• “Active Logic - Needs Simulation” on page 13-44</li> <li>• “Objectives Valid” on page 13-44</li> <li>• “Objectives Valid under Approximation” on page 13-45</li> <li>• “Objectives Falsified with Counterexamples” on page 13-45</li> <li>• “Objectives Error - Needs Simulation” on page 13-45</li> <li>• “Objectives Undecided Due to Division by Zero” on page 13-49</li> <li>• “Objectives Undecided Due to Nonlinearities” on page 13-50</li> <li>• “Objectives Undecided Due to Stubbing” on page 13-50</li> <li>• “Objectives Undecided” on page 13-50</li> <li>• “Objectives Undecided Due to Array Out of Bounds” on page 13-50</li> </ul> |

| Analysis Mode    | Objective Status  |
|------------------|---|
| Test generation  | <ul style="list-style-type: none"> <li>• “Objectives Satisfied” on page 13-46</li> <li>• “Objectives Satisfied - Needs Simulation” on page 13-46</li> <li>• “Objectives Unsatisfiable” on page 13-47</li> <li>• “Objectives Unsatisfiable under Approximation” on page 13-47</li> <li>• “Objectives Undecided with Testcases” on page 13-47</li> <li>• “Objectives Undecided due to Runtime Error” on page 13-49</li> <li>• “Objectives Undecided Due to Division by Zero” on page 13-49</li> <li>• “Objectives Undecided Due to Nonlinearities” on page 13-50</li> <li>• “Objectives Undecided Due to Stubbing” on page 13-50</li> <li>• “Objectives Undecided” on page 13-50</li> <li>• “Objectives Undecided Due to Array Out of Bounds” on page 13-50</li> </ul>            |
| Property proving | <ul style="list-style-type: none"> <li>• “Objectives Valid” on page 13-48</li> <li>• “Objectives Valid under Approximation” on page 13-48</li> <li>• “Objectives Falsified with Counterexamples” on page 13-48</li> <li>• “Objectives Falsified - Needs Simulation” on page 13-49</li> <li>• “Objectives Undecided with Counterexamples” on page 13-49</li> <li>• “Objectives Undecided due to Runtime Error” on page 13-49</li> <li>• “Objectives Undecided Due to Division by Zero” on page 13-49</li> <li>• “Objectives Undecided Due to Nonlinearities” on page 13-50</li> <li>• “Objectives Undecided Due to Stubbing” on page 13-50</li> <li>• “Objectives Undecided” on page 13-50</li> <li>• “Objectives Undecided Due to Array Out of Bounds” on page 13-50</li> </ul> |

### Design Error Detection Objectives Status

If you run a design error detection analysis, the **Design Error Detection Objectives Status** section can include the following objective statuses:

- “Dead Logic” on page 13-44
- “Dead Logic under Approximation” on page 13-44
- “Active Logic - Needs Simulation” on page 13-44
- “Objectives Valid” on page 13-44
- “Objectives Valid under Approximation” on page 13-45
- “Objectives Falsified with Counterexamples” on page 13-45
- “Objectives Error - Needs Simulation” on page 13-45

If an Observer Reference block is used in the design error detection analysis, then this section will show the observer information in **Observer Model(s)** subsection and design model information in **Design Model** subsection. This section will be empty when there are no active logic present in the model.

The table in the Design model subsection shows the list of active logic in the `sldvdemo_debounce_validprop` example model.

| # | Type     | Model Item                       | Description                    | Analysis Time (sec) |
|---|----------|----------------------------------|--------------------------------|---------------------|
| 3 | Decision | <a href="#">Chart "debounce"</a> | Substate executed State "Off"  | 11                  |
| 4 | Decision | <a href="#">Chart "debounce"</a> | Substate executed State "On"   | 14                  |
| 5 | Decision | <a href="#">Chart "debounce"</a> | Substate executed State "Wait" | 11                  |

The Observer model(s) section will not show any active logic reported as the observers are ignored for design error detection analysis.

### Dead Logic

The **Dead Logic** section lists the items for which the analysis found dead logic.

This image shows the **Dead Logic** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` model.

| # | Type      | Model Item   | Description   |
|---|-----------|--|---|
| 1 | Condition | Transition " <a href="#">!speed==0 &amp; press &lt; zero_th...</a> " from " <a href="#">speed_norm</a> " to " <a href="#">speed_fail</a> " | " <a href="#">press &lt; zero_thresh</a> " can only be true |
| 2 | Decision  | Transition " <a href="#">!m(Sens_Failure_Counter.Mu...</a> " from Junction #2 to "Shutdown"  | trigger expression can only be true                         |

### Dead Logic under Approximation

The **Dead Logic under Approximation** section lists the model items for which the analysis found dead logic under the impact of approximation.

In releases before R2017b, this section can include objectives that were marked as **Dead Logic**.

This image shows the **Dead Logic under Approximation** section of the generated analysis report.

| # | Type      | Model Item               | Description                            | Analysis Time (sec) | Test Case |
|---|-----------|--------------------------|--|---------------------|-----------|
| 2 | Condition | <a href="#">enblock1</a> | Script: <code>isequal(A1.A1eq)F</code> | 13                  | n/a       |

### Active Logic - Needs Simulation

The **Active Logic - Needs Simulation** section lists the model items for which the analysis found active logic. To confirm the active logic status, you must run additional simulations of test cases.

In releases before R2017b, this section can include objectives that were marked as **Active Logic**.

This image shows a portion of the **Active Logic - Needs Simulation** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` model.

| # | Type     | Model Item                                     | Description                                      | Analysis Time (sec) | Test Case |
|---|----------|--|--|---------------------|-----------|
| 3 | Decision | State " <a href="#">Oxygen_Sensor_Mode</a> "   | Substate executed " <a href="#">O2_fail</a> "    | 28                  | 1         |
| 4 | Decision | State " <a href="#">Oxygen_Sensor_Mode</a> "   | Substate executed " <a href="#">O2_normal</a> "  | 27                  | 1         |
| 5 | Decision | State " <a href="#">Oxygen_Sensor_Mode</a> "   | Substate executed " <a href="#">O2_warmup</a> "  | 27                  | 1         |
| 6 | Decision | State " <a href="#">Pressure_Sensor_Mode</a> " | Substate executed " <a href="#">press_fail</a> " | 28                  | 1         |

### Objectives Valid

The **Objectives Valid** section lists the design error detection objectives that the analysis found valid. For these objectives, the analysis determined that the described design errors cannot occur.

In releases before R2017b, this section can include objectives that were marked as **Proven Valid**.

This image shows the **Objectives Valid** section of the generated analysis report for the `sldvdemo_design_error_detection` model.

| #  | Type     | Model Item  | Description | Analysis Time (sec) | Test Case |
|----|----------|---|-------------|---------------------|-----------|
| 3  | Overflow | <a href="#">Controller.Sum</a>                                    | Overflow    | 8                   | n/a       |
| 18 | Overflow | <a href="#">Controller.Pl Controller.Discrete Time Integrator</a> | Overflow    | 8                   | n/a       |
| 21 | Overflow | <a href="#">Controller.Pl Controller.Kp</a>                       | Overflow    | 8                   | n/a       |
| 24 | Overflow | <a href="#">Controller.Pl Controller.Kp1</a>                      | Overflow    | 8                   | n/a       |
| 27 | Overflow | <a href="#">Controller.Pl Controller.Sum</a>                      | Overflow    | 8                   | n/a       |

### Objectives Valid under Approximation

The **Objectives Valid under Approximation** section lists the design error detection objectives that the analysis found valid under the impact of approximation.

In releases before R2017b, this section can include objectives that were marked as **Proven Valid**.

This image shows the **Objectives Valid under Approximation** section of the generated analysis report.

| #  | Type             | Model Item             | Description      | Analysis Time (sec) | Test Case |
|----|------------------|------------------------|------------------|---------------------|-----------|
| 12 | Division by zero | <a href="#">Divide</a> | Division by zero | 40                  | n/a       |

### Objectives Falsified with Counterexamples

The **Objectives Falsified with Counterexamples** lists the set of design error detection objects whose counterexamples have been simulated and verified to observe the reported errors.

This image shows the **Objectives Falsified with Counterexamples** section of the generated analysis report for the `sldvdemo_design_error_detection` model.

| #  | Type             | Model Item                      | Description | Analysis Time (sec) | Test Case         |
|----|------------------|---------------------------------|-------------|---------------------|-------------------|
| 7  | Integer overflow | <a href="#">Controller.Sum2</a> | Overflow    | 39                  | <a href="#">1</a> |
| 12 | Integer overflow | <a href="#">Controller.Sum1</a> | Overflow    | 39                  | <a href="#">2</a> |

### Objectives Error - Needs Simulation

The **Objectives Error- Needs Simulation** section lists the design error detection objectives for which the analysis found test cases that demonstrate design errors. To confirm the falsified status, you must run additional simulations of test cases.

In releases before R2017b, this section can include objectives that were marked as **Falsified**.

This image shows the **Objectives Error - Needs Simulation** section of the generated analysis report for the `sldvdemo_array_bounds` model.

| #  | Type         | Model Item                   | Description     | Analysis Time (sec) | Test Case         |
|----|--------------|------------------------------|-----------------|---------------------|-------------------|
| 16 | Array bounds | State <a href="#">"Diff"</a> | Array bounds: u | 26                  | <a href="#">1</a> |
| 17 | Array bounds | State <a href="#">"Diff"</a> | Array bounds: u | 26                  | <a href="#">2</a> |

### Test Objectives Status

If you run a test case generation analysis, the **Test Objectives Status** section can include the following objective statuses:

- “Objectives Satisfied” on page 13-46

- “Objectives Satisfied - Needs Simulation” on page 13-46
- “Objectives Unsatisfiable” on page 13-47
- “Objectives Unsatisfiable under Approximation” on page 13-47
- “Objectives Undecided with Testcases” on page 13-47

When you analyze a model with **Model coverage objectives** set to **Enhanced MCDC**, the software reports the detection status of model items. For more information, see “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42.

If an Observer Reference block is used in the test case generation analysis, then each test objective status section will show the observer information in **Observer Model(s)** sub-section an design model information in **Design Model** subsection. These subsections will be empty if no test objective found in the model.

The table shows a part of **Objectives Satisfied** test objectives for the design model in the `sldvdemo_debounce_testobjblks` example model.

| # | Type           | Model Item           | Description  | Analysis Time (sec) | Test Case         |
|---|----------------|----------------------|--------------|---------------------|-------------------|
| 2 | Test objective | <a href="#">True</a> | Objective: 2 | 29                  | <a href="#">1</a> |

The table shows a part of **Objectives Satisfied** test objectives for observer model in the `sldvdemo_debounce_testobjblks` example model.

| # | Type           | Model Item                            | Description  | Analysis Time (sec) | Test Case         |
|---|----------------|---------------------------------------|--------------|---------------------|-------------------|
| 1 | Test objective | <a href="#">Masked Objective/Edge</a> | Objective: 1 | 29                  | <a href="#">2</a> |

### Objectives Satisfied

The **Objectives Satisfied** section lists the test objectives that the analysis satisfied. The generated test cases cover the objectives.

This image shows a portion of the **Objectives Satisfied** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` example model.

| # | Type     | Model Item   | Description                                 | Analysis Time (sec) | Test Case          |
|---|----------|--|---|---------------------|--------------------|
| 1 | Decision | <a href="#">control logic.Oxygen_Sensor_Mode</a>   | State: Substate executed State "O2_fail"    | 97                  | <a href="#">35</a> |
| 2 | Decision | <a href="#">control logic.Oxygen_Sensor_Mode</a>   | State: Substate executed State "O2_normal"  | 94                  | <a href="#">31</a> |
| 3 | Decision | <a href="#">control logic.Oxygen_Sensor_Mode</a>   | State: Substate executed State "O2_warmup"  | 72                  | <a href="#">1</a>  |
| 4 | Decision | <a href="#">control logic.Pressure_Sensor_Mode</a> | State: Substate executed State "press_fail" | 79                  | <a href="#">9</a>  |
| 5 | Decision | <a href="#">control logic.Pressure_Sensor_Mode</a> | State: Substate executed State "press_norm" | 72                  | <a href="#">1</a>  |

### Objectives Satisfied - Needs Simulation

The **Objectives Satisfied - Needs Simulation** section lists the test objectives that the analysis satisfied. To confirm the satisfied status, you must run additional simulations of test cases.

In releases before R2017b, this section can include objectives that were marked as **Satisfied**.

This image shows the **Objectives Satisfied - Needs Simulation** section of the generated analysis report.



| # | Type     | Model Item                        | Description            | Analysis Time (sec) | Test Case         |
|---|----------|-----------------------------------|------------------------|---------------------|-------------------|
| 1 | Decision | <a href="#">Smaulink Function</a> | Function call executed | 11                  | <a href="#">1</a> |

### Objectives Unsatisfiable

The **Objectives Unsatisfiable** section lists the test objectives that the analysis determined could not be satisfied.

In releases before R2017b, this section can include objectives that were marked as **Proven Unsatisfiable**.

This image shows the **Objectives Unsatisfiable** section of the generated analysis report for the `sldvdemo_fuelsys_logic_simple` example model.

| #   | Type      | Model Item   | Description  | Analysis Time (sec) | Test Case |
|-----|-----------|--|--|---------------------|-----------|
| 61  | Condition | <a href="#">controlLogic.Speed_Sensor_Mode."   speed==0 &amp; press == zero_th_..."</a>    | Transition: Condition 2, "press < zero_thresh" F   | 13                  | n/a       |
| 67  | MCDC      | <a href="#">controlLogic.Speed_Sensor_Mode."   speed==0 &amp; press == zero_th_..."</a>    | Transition: MCDC Transition trigger expression with Condition 2, "press < zero_thresh" F | 13                  | n/a       |
| 106 | Decision  | <a href="#">controlLogic.Fueling_Mode.Fuel_Disabled."   (mSens_Failure_Counter.Mu_..."</a> | Transition: Transition trigger expression F  | 13                  | n/a       |

### Objectives Unsatisfiable under Approximation

The **Objectives Unsatisfiable under Approximation** section lists the test objectives that the analysis determined could not be satisfied due to approximation during analysis.

In releases before R2017b, this section can include objectives that were marked as **Proven Unsatisfiable**.

This image shows the **Objectives Unsatisfiable under Approximation** section of the generated analysis report.

| # | Type     | Model Item                                  | Description   | Analysis Time (sec) | Test Case |
|---|----------|---|---------------|---------------------|-----------|
| 5 | Decision | <a href="#">Chart_WithLengthGuard.Box.B</a> | State: Mloc F | 21                  | n/a       |

### Objectives Undecided with Testcases

The **Objectives Undecided with Testcases** section lists the test objectives that are undecided due to the impact of approximation during analysis.

In releases before R2017b, this section can include objectives that were marked as **Satisfied**.

This image shows the **Objectives Undecided with Testcases** section of the generated analysis report for the `sldvApproximationsExample` example model.

| # | Type     | Model Item             | Description   | Analysis Time (sec) | Test Case         |
|---|----------|------------------------|---|---------------------|-------------------|
| 1 | Decision | <a href="#">Switch</a> | logical trigger input false (output is from 3rd input port) | 14                  | <a href="#">2</a> |

### Proof Objectives Status

If you run a property-proving analysis, the **Proof Objectives Status** section can include:

- “Objectives Valid” on page 13-48
- “Objectives Valid under Approximation” on page 13-48

- “Objectives Falsified with Counterexamples” on page 13-48
- “Objectives Falsified - Needs Simulation” on page 13-49
- “Objectives Undecided with Counterexamples” on page 13-49

If an Observer Reference block is used in the property-proving analysis, then each proof objective status section will show the observer information in **Observer Model(s)** subsection and design model information in **Design Model** subsection. These subsections will be empty when no objective is found in the model.

The table shows **Objectives Valid** proof objectives for the Observer model in the `sldvdemo_debounce_validprop` example model.

| # | Type            | Model Item                                 | Description  | Analysis Time (sec) |
|---|-----------------|--|--------------|---------------------|
| 1 | Proof objective | <a href="#">Verify: Output/FoutCorrect</a> | Objective: T | 8                   |
| 2 | Proof objective | <a href="#">Verify: Output/ToutCorrect</a> | Objective: T | 8                   |

### Objectives Valid

The **Objectives Valid** section lists the proof objectives that the analysis found valid.

In releases before R2017b, this section can include objectives that were marked as **Proven Valid**.

This image shows the **Objectives Valid** section of the generated analysis report for the `sldvdemo_debounce_validprop` example model.

| # | Type            | Model Item                                 | Description  | Analysis Time (sec) | Counterexample |
|---|-----------------|--|--------------|---------------------|----------------|
| 1 | Proof objective | <a href="#">Verify: Output/FoutCorrect</a> | Objective: T | 16                  | n/a            |
| 2 | Proof objective | <a href="#">Verify: Output/ToutCorrect</a> | Objective: T | 17                  | n/a            |

### Objectives Valid under Approximation

The **Objectives Valid under Approximation** section lists the proof objectives that the analysis found valid under the impact of approximation.

In releases before R2017b, this section can include objectives that were marked as **Objectives Proven Valid**.

This image shows the **Objectives Valid under Approximation** section of the generated analysis report.

| # | Type            | Model Item                      | Description     | Analysis Time (sec) | Counterexample |
|---|-----------------|---------------------------------|-----------------|---------------------|----------------|
| 1 | Proof objective | <a href="#">MATLAB Function</a> | sldv.prove(x>0) | 9                   | n/a            |

### Objectives Falsified with Counterexamples

The **Objectives Falsified with Counterexamples** section lists the proof objectives that the analysis disproved. The generated counterexample shows the violation of the proof objective.

This image shows the **Objectives Falsified with Counterexamples** section of the generated analysis report for the `sldvdemo_debounce_falseprop` example model.

| # | Type   | Model Item                                   | Description | Analysis Time (sec) | Counterexample    |
|---|--------|--|-------------|---------------------|-------------------|
| 1 | Assert | <a href="#">Verify True Output/Assertion</a> | Assert      | 1                   | <a href="#">1</a> |

### Objectives Falsified - Needs Simulation

The **Objectives Falsified - Needs Simulation** section lists the proof objectives that the analysis disproved. To confirm the falsified status, you must run additional simulations of counterexamples.

In releases before R2017b, this section can include objectives that were marked as **Objectives Falsified with Counterexamples**.

This image shows the **Objectives Falsified - Needs Simulation** section of the generated analysis report.

| # | Type            | Model Item  | Description                                  | Analysis Time (sec) | Counterexample    |
|---|-----------------|---|--|---------------------|-------------------|
| 1 | Proof objective | <a href="#">Safety Properties/MATLAB Property</a> | slsv.prove(implies(activeCond.SeatBeltIcon)) | 12                  | <a href="#">1</a> |

### Objectives Undecided with Counterexamples

The **Objectives Undecided with Counterexamples** section lists the proof objectives undecided due to the impact of approximation during analysis.

In releases before R2017b, this section can include objectives that were marked as **Falsified**.

This image shows the **Objectives Undecided with Counterexamples** section of the generated analysis report.

| # | Type            | Model Item                      | Description       | Analysis Time (sec) | Counterexample    |
|---|-----------------|---------------------------------|-------------------|---------------------|-------------------|
| 1 | Proof objective | <a href="#">Proof Objective</a> | Objective: [1, 2] | 11                  | <a href="#">1</a> |

### Objectives Undecided due to Runtime Error

For proof objectives and test objectives, the **Objectives Undecided due to Runtime Error** section lists the undecided objectives during analysis due to a run-time error. The run-time error occurred during simulation of a test case or counterexample.

In releases before R2017b, this section can include objectives that were marked as **Falsified** or **Satisfied**.

This image shows the **Objectives Undecided due to Runtime Error** section of the generated analysis report.

| # | Type      | Model Item                          | Description                               | Analysis Time (sec) | Test Case         |
|---|-----------|-------------------------------------|---|---------------------|-------------------|
| 1 | Condition | <a href="#">Relational Operator</a> | RelationalOperator: input1 == input2<br>T | 13                  | <a href="#">1</a> |

### Objectives Undecided Due to Division by Zero

For all types of objectives, the **Objectives Undecided Due to Division by Zero** section lists the undecided objectives during analysis due to division-by-zero errors in the associated model items. To detect division-by-zero errors before running further analysis on your model, follow the procedure in "Detect Integer Overflow and Division-by-Zero Errors" on page 6-19.

| # | Type     | Model Item                 | Description            | Analysis Time (sec) | Test Case |
|---|----------|----------------------------|------------------------|---------------------|-----------|
| 1 | Decision | <a href="#">Saturation</a> | input > lower limit F  | 0                   | n/a       |
| 2 | Decision | <a href="#">Saturation</a> | input > lower limit T  | 0                   | n/a       |
| 3 | Decision | <a href="#">Saturation</a> | input >= upper limit F | 0                   | n/a       |
| 4 | Decision | <a href="#">Saturation</a> | input >= upper limit T | 0                   | n/a       |

## Objectives Undecided Due to Nonlinearities

For all types of objectives, the **Objectives Undecided Due to Nonlinearities** section lists the undecided objectives during analysis due to required computation of nonlinear arithmetic. Simulink Design Verifier does not support nonlinear arithmetic or nonlinear logic.

| #  | Type     | Model Item                               | Description                         | Analysis Time (sec) | Test Case |
|----|----------|--|-------------------------------------|---------------------|-----------|
| 30 | Decision | <a href="#">BasicRollMode/Integrator</a> | integration result <= lower limit T | 2                   | n/a       |
| 32 | Decision | <a href="#">BasicRollMode/Integrator</a> | integration result >= upper limit T | 2                   | n/a       |

## Objectives Undecided Due to Stubbing

For all types of objectives, the **Objectives Undecided Due to Stubbing** section lists model items with undecided objectives during analysis due to stubbing. In releases before R2013b, these objectives can include objectives that were marked as **Objectives Satisfied - No Test Case** or **Objectives Falsified - No Counterexample**.

| # | Type     | Model Item                 | Description            | Analysis Time (sec) |
|---|----------|----------------------------|------------------------|---------------------|
| 2 | Decision | <a href="#">Saturation</a> | input > lower limit F  | 12                  |
| 3 | Decision | <a href="#">Saturation</a> | input > lower limit T  | 12                  |
| 4 | Decision | <a href="#">Saturation</a> | input >= upper limit F | 12                  |
| 5 | Decision | <a href="#">Saturation</a> | input >= upper limit T | 12                  |

## Objectives Undecided Due to Array Out of Bounds

For all types of objectives, the **Objectives Undecided Due to Array Out of Bounds** section lists the undecided objectives during analysis due to array out of bounds errors in the associated model items. To detect out of bounds array errors in your model, see “Detect Out of Bound Array Access Errors” on page 6-28.

| # | Type           | Model Item                     | Description         | Analysis Time (sec) | Test Case |
|---|----------------|--------------------------------|---------------------|---------------------|-----------|
| 1 | Test objective | <a href="#">Test Objective</a> | Objective (3, Inf)  | 18                  | n/a       |
| 2 | Test objective | <a href="#">Test Objective</a> | Objective (-Inf, 0) | 18                  | n/a       |

## Objectives Undecided

For all types of objectives, the **Objectives Undecided** section lists the objectives for which the analysis was unable to determine an outcome in the allotted time.

In this property-proving example, either the software exceeded its analysis time limit (which the **Maximum analysis time** parameter specifies) or you aborted the analysis before it completed processing these objectives.

| # | Type            | Model Item                                | Description  | Analysis Time (sec) | Counterexample |
|---|-----------------|---|--------------|---------------------|----------------|
| 1 | Proof objective | <a href="#">Verify Output/FoutCorrect</a> | Objective: T | -1                  | n/a            |
| 2 | Proof objective | <a href="#">Verify Output/ToutCorrect</a> | Objective: T | -1                  | n/a            |

## Model Items Chapter

The **Model Items** chapter of the report includes a table for each object in the model that defines coverage objectives. The table for a particular object lists all of the associated objectives, the objective types, objective descriptions, and the status of each objective at the end of the analysis.

The table for an individual object in the model looks similar to this one for the Discrete-Time Integrator in the PI Controller subsystem of the `sldvdemo_cruise_control` example model.

| #: | Type     | Description                         | Status    | Test Case         |
|----|----------|-------------------------------------|-----------|-------------------|
| 31 | Decision | integration result <= lower limit F | Satisfied | <a href="#">3</a> |
| 32 | Decision | integration result <= lower limit T | Satisfied | <a href="#">8</a> |
| 33 | Decision | integration result >= upper limit F | Satisfied | <a href="#">3</a> |
| 34 | Decision | integration result >= upper limit T | Satisfied | <a href="#">9</a> |

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.

If an Observer Reference block is used in the property-proving analysis, then each model item section will show the observer information in **Observer Model(s)** subsection and design model information in **Design Model** subsection. These subsections will be empty if no test objective found in the model.

The table shows one of the test objectives for the design model in the `sldvdemo_debounce_testobjblks` example model.

| #: | Type           | Description  | Status    | Test Case         |
|----|----------------|--------------|-----------|-------------------|
| 2  | Test objective | Objective: 2 | Satisfied | <a href="#">1</a> |

The table shows one of the test objectives for the observer model in the `sldvdemo_debounce_testobjblks` example model.

| #: | Type           | Description  | Status    | Test Case         |
|----|----------------|--------------|-----------|-------------------|
| 1  | Test objective | Objective: 1 | Satisfied | <a href="#">2</a> |

## Design Errors Chapter

If you perform design error detection analysis and the analysis detects design errors in the model, the report includes a **Design Errors** chapter. This chapter summarizes the design errors that the analysis falsified:

- “Table of Contents” on page 13-51
- “Summary” on page 13-51
- “Test Case” on page 13-52

### Table of Contents

The Design Errors chapter contains a table of contents. Each item in the table of contents is a hyperlink to results about a specific design error.

### Summary

The Summary section lists:

- The model item
- The type of design error that was detected (overflow or division by zero)
- The status of the analysis (Falsified or Proven Valid)

In the following example, the software analyzed the `sldvdemo_debounce_falseprop` model to detect design errors. The analysis detected an overflow error in the Sum block in the Verification Subsystem named Verify True Output.

Model Item: [Verify True Output/Sum](#)  
 Type: Overflow  
 Status: Falsified

### Test Case

The Test Case section lists the time step and corresponding time at which the test case falsified the design error objective. The Inport block raw had a value of 255, which caused the overflow error.

|      |        |
|------|--------|
| Time | 0-0.01 |
| Step | 1-2    |
| raw  | 255    |

### Test Cases Chapter

If you run a test generation analysis, the report includes a **Test Cases** chapter. This chapter includes sections that summarize the test cases the analysis generated:

- “Table of Contents” on page 13-52
- “Summary” on page 13-52
- “Objectives” on page 13-53
- “Generated Input Data” on page 13-53
- “Expected Output” on page 13-53
- “Long Test Cases” on page 13-54

### Table of Contents

The Test Cases chapter contains a table of contents. Each item in the table of contents is a hyperlink to information about a specific test case.

### Summary

The Summary section lists:

- Length of the signals that comprise the test case
- Total number of test objectives that the test case achieves

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

## Objectives

The Objectives section lists:

- The time step at which the test case achieves that objective.
- The time at which the test case achieves that objective.
- A link to the model item associated with that objective. Clicking the link highlights the model item in the Simulink Editor.
- The objective that was achieved with a link to navigate between the **Test Objectives Status** and **Test Cases** chapters.

| Step | Time | Model Item   | Objectives  |
|------|------|--|---|
| 2    | 0.01 | <a href="#">State "Running"</a><br><a href="#">State "Low_Emissions"</a> | <a href="#">112. Substate exited when parent exits "Low_Emissions"</a><br><a href="#">123. Substate exited when parent exits "Warmup"</a> |
| 7    | 0.06 | <a href="#">Transition "DEC" from "FL1" to "FL0"</a>                     | <a href="#">S6.trigger expression true</a>  |

## Generated Input Data

For each input signal associated with the model item, the Generated Input Data section lists the time step and corresponding time at which the test case achieves particular test objectives. If the signal value does not change over those time steps, the table lists the time step and time as ranges.

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

**Note** The Generated Input Data table displays a dash (-) instead of a number as a signal value when the value of the signal at that time step does not affect the test objective. The table does not include the entire signal if all values of a signal are having no impact. In the harness model, the Inputs block represents these values with zeros unless you enable the **Randomize data that does not affect outcome** parameter (see "Randomize data that do not affect the outcome" on page 15-58).

## 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.

|             |          |             |             |             |             |             |             |
|-------------|----------|-------------|-------------|-------------|-------------|-------------|-------------|
| <b>Time</b> | <b>0</b> | <b>0.01</b> | <b>0.02</b> | <b>0.03</b> | <b>0.04</b> | <b>0.05</b> | <b>0.06</b> |
| <b>Step</b> | <b>1</b> | <b>2</b>    | <b>3</b>    | <b>4</b>    | <b>5</b>    | <b>6</b>    | <b>7</b>    |
| throt       | 0        | 1.96        | 1.9898      | 2.0197      | 2.0497      | 2.0798      | 0.05        |
| target      | 97       | 98          | 99          | 100         | 101         | 102         | 0           |

### Long Test Cases

If you set the **Test suite optimization** option to LongTestcases, the Test Cases chapter in the report includes fewer sections about longer test cases.

#### [Test Case 1](#)

This section contains detailed information about each generated test case.

#### Summary

Length: 0.26 second (27 sample periods)  
 Objectives Satisfied: 259

### Properties Chapter

If you run a property-proving analysis, the report includes a **Properties** chapter. This chapter includes sections that summarize the proof objectives and any counterexamples the software generated:

- “Table of Contents” on page 13-54
- “Summary” on page 13-55
- “Counterexample” on page 13-55

#### Table of Contents

The 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.

If an Observer Reference block is used in the property-proving analysis, then each properties chapter will show the observer information in **Observer Model(s)** subsection and design model information in **Design Model** subsection. It will be empty when there are no properties in the model.

The table shows one of the properties for the observer model in the sldvdemo\_debounce\_validprop example model.

|             |   |
|-------------|---|
| Model Item: | <a href="#">Verify Output/FoutCorrect</a> |
| Property:   | Objective: T                              |
| Status:     | Valid                                     |



## Summary

The Summary section lists:

- The model item that the software analyzed
- The type of property that was evaluated
- The status of the analysis

In the following example, the software analyzed the `sldvdemo_cruise_control_verification` model for property proving. The analysis proved that the input to the Assertion block named `BrakeAssertion` was nonzero.

Model Item:    [Safety Properties/BrakeAssertion](#)  
 Property:        Assert  
 Status:          Falsified

## Counterexample

The Counterexample section lists the time step and corresponding time at which the counterexample falsified the property. This section also lists the values of the signals at that time step.

| Time                             | 0 | 0.01 | 0.02-0.04 |
|----------------------------------|---|------|-----------|
| Step                             | 1 | 2    | 3-5       |
| InputData.Actual_speed           | 0 | 0    | 0         |
| InputData.Switches.enable        | 1 | 1    | 0         |
| InputData.Switches.brake         | 0 | 0    | 1         |
| InputData.Switches.set           | 1 | 0    | 0         |
| InputData.Switches.setIncDec.inc | 1 | 1    | 0         |
| InputData.Switches.setIncDec.dec | 0 | 0    | 0         |

## View Log Files

Every time you analyze a model, Simulink Design Verifier creates a log file. To view the log file, click **View Log** in the Simulink Design Verifier log window.

The log file contains a list of the analysis results for each object in the model. The content of the log file corresponds to the analysis results displayed in the log window during the analysis.

```
1
2 20-Mar-2019 15:49:20
3 Checking compatibility for test generation: model 'sldvdemo_cruise_control'
4 Compiling model...done
5 Building model representation...done
6
7 20-Mar-2019 15:49:42
8 'sldvdemo_cruise_control' is compatible for test generation with Simulink Design Verifier.
9
10
11 Generating tests using model representation from 20-Mar-2019 15:49:42...
12
13 SATISFIED
14 Controller/Switch3
15 logical trigger input true (output is from 1st input port)
16 Analysis Time = 00:00:12
17
18 SATISFIED
19 Controller/PI Controller
20 enable logical value true
21 Analysis Time = 00:00:12
22
23 SATISFIED
24 Controller/PI Controller/Discrete-Time Integrator
25 integration result >= upper limit false
26 Analysis Time = 00:00:12
```

# Review Analysis Results

## In this section...

“View Active Results” on page 13-57

“Load Previous Results” on page 13-57

“Explore Results” on page 13-57

## View Active Results

After analysis is complete, the Simulink Design Verifier Results Summary window opens, showing different ways you can use the results. See “Explore Results” on page 13-57.

If you close the Results Summary window so you can fix the cause of any analysis errors in your model, you might need to review the analysis results again. If you have not closed your model since you ran the analysis, you can reopen the latest analysis results for your model.

On the **Design Verifier** tab, click **Results Summary** to view the Results Summary window. The Results Summary window reopens with the latest analysis results for your model.

## Load Previous Results

If you want to review results of a previous analysis on a model, you can load these results from the analysis data file. On the **Design Verifier** tab, click **Load Earlier Results** and browse to the data file that corresponds to the analysis you want to review. Click **Results Summary**.

For more information on analysis data files, see “Manage Simulink Design Verifier Data Files” on page 13-7.

If you load analysis results for a model from a data file that was generated with a previous version of that model, you might see unexpected effects. To avoid inconsistencies between your model and analysis results data, when you load results for a model, choose a data file that contains results from the same version of that model.

## Explore Results

With active or previous analysis results loaded in the Results Summary window, you can perform the following tasks.

| Task   | For more information                          |
|--|---|
| Highlight the analysis results on the model. | “Highlight Results on the Model” on page 13-2 |
| Generate a detailed analysis report.         | “Review Results” on page 13-35                |

| <b>Task</b>   | <b>For more information</b>                                    |
|---|--|
| Create the harness model, or if the harness model already exists, open it.<br><br>You will not be able to create the harness model if: <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> | “Manage Simulink Design Verifier Harness Models” on page 13-13 |
| View the data file.   | “Manage Simulink Design Verifier Data Files” on page 13-7      |
| View the log file.  | “View Log Files” on page 13-56                                 |

## See Also

### More About

- “Design Verifier Pane: Results” on page 15-56
- “Manage Simulink Design Verifier Data Files” on page 13-7
- “Review Results” on page 13-35

# Analyzing Large Models and Improving Performance

---

- “Sources of Model Complexity” on page 14-2
- “Analyze a Large Model” on page 14-3
- “Increase Allocated Memory for Analysis Report Generation” on page 14-7
- “Manage Model Data to Simplify the Analysis” on page 14-8
- “Partition Model Inputs for Incremental Test Generation” on page 14-11
- “Bottom-Up Approach to Model Analysis” on page 14-13
- “Extract Subsystems for Analysis” on page 14-15
- “Logical Operations” on page 14-21
- “Analyzing Models with Large Verification State Space” on page 14-22
- “Counters and Timers” on page 14-23
- “Prove Properties in Large Models” on page 14-24

## Sources of Model Complexity

Some characteristics of Simulink models can cause problems during a Simulink Design Verifier analysis in the following ways:

- Complexity of model inputs due to:
  - Large number of inputs (The number of inputs can vary, depending on the individual model.)
  - Types of inputs (floating-point values, for example)
  - The way the inputs affect the model state and the objectives of the analysis
- Number of possible simulation paths through a model
- Portions of the model that cannot be reached
- Large counters in the model

The topics in “Reduce Model Complexity” describe techniques designed to reduce the impact of this complexity and achieve the best performance from Simulink Design Verifier.

Most of these techniques focus on test generation for large models. However, you can use many of them to detect design errors or prove the properties of a large model and generate counterexamples when a property is disproved. In addition, “Prove Properties in Large Models” on page 14-24 describes specific techniques for proving properties in a large model.

# Analyze a Large Model

## In this section...

- “Types of Large Model Problems” on page 14-3
- “Summarize Model Hierarchy and Compatibility” on page 14-3
- “Use the Default Parameter Values” on page 14-4
- “Modify the Analysis Parameters” on page 14-5
- “Stop the Analysis Before Completion” on page 14-5

## Types of Large Model Problems

The Simulink Design Verifier software may encounter some of these problems when analyzing a large model:

- Unsatisfiable objectives — The software proved there are no test cases that exercise these test objectives, and did not generate any test cases.
- Undecided objectives — The software was not able to satisfy or falsify these objectives.
- Objectives with errors — This problem usually occurs when a model component uses nonlinear arithmetic, which can affect a test objective.
- Cannot complete the analysis in the time allotted — This problem may indicate an area of your model where the software encountered problems, or you may need to increase value of the **Maximum analysis time** parameter.
- Analysis hangs — If the number of objectives processed remains constant for a considerable length of time, the software has likely encountered complexity between the model and its objectives.
- Does not achieve a high percentage of model coverage — When you run the test cases on the harness model, the percentage of model coverage is insufficient for your design.

The next few sections describe the initial steps to take when analyzing a large model. Although these steps address test generation, you can use a similar approach when detecting design errors or proving properties in a model.

## Summarize Model Hierarchy and Compatibility

You can use the Test Generation Advisor to summarize test generation compatibility, condition and decision objectives, and dead logic for the model and model components.

The Test Generation Advisor performs a high-level analysis and fast dead logic detection. You can use the results to better understand your model, particularly large models, complex models, or models for which you are uncertain of their compatibility with Simulink Design Verifier. For example, you can:

- Identify incompatibilities with test case generation.
- Identify complex components that might be time-consuming to analyze.
- Determine instances of dead logic.
- Get a summary of the component hierarchy.
- Get recommended test generation parameters.

To access the Test Generation Advisor, on the **Design Verifier** tab, in the **Mode** section, click **Test Generation**. In the **Prepare** section, click **Advisor**. For more information see “Use Test Generation Advisor to Identify Analyzable Components” on page 7-24.

## Use the Default Parameter Values

When you generate test cases, you should generally begin by analyzing the model using the Simulink Design Verifier default parameter values:

- 1 Check to see if your model is compatible with Simulink Design Verifier, as described in “Check Model Compatibility” on page 3-2.
- 2 Using the default parameter values, analyze the model. The following table lists the default values for parameters in the Configuration Parameters dialog box that you might change when analyzing large models.

| Parameter                        | Default Value      | Description   |
|----------------------------------|--------------------|---|
| <b>Maximum analysis time (s)</b> | 300 (seconds)      | If the analysis does not finish within the specified time, the analysis times out and terminates. |
| <b>Test suite optimization</b>   | Auto               | Generates test cases that address more than one test objective.                                   |
| <b>Model coverage objectives</b> | Condition/Decision | Generates test cases that achieve condition and decision coverage.                                |

- 3 Review the following information in the Simulink Design Verifier log window while the analysis runs:
  - Number of objectives processed — How many objectives were processed? Did the analysis hang after processing a certain number of objectives? The answers to these questions might give you a clue about where a problem might lie.
  - Number of objectives satisfied/Number of objectives falsified — Which objectives were falsified?
  - Time elapsed — Did the analysis time out, or did it finish within the specified maximum analysis time?
- 4 When the analysis completes, you can highlight the results in the model and individually review the analysis of each model object, as described in “Highlight Results on the Model” on page 13-2. You can also generate and review the Simulink Design Verifier HTML report. This report contains links to the model elements for satisfied and falsified objectives so you can see what portions of the model might have problems. For more information, see “Review Results” on page 13-35.
- 5 For a test generation analysis, if all the test objectives have been satisfied, run the test cases on the harness model to determine model coverage.

If model coverage is enough for your design, you do not need to do anything else. If the coverage is insufficient, take additional steps to improve the analysis performance, as described in the following sections.

---

**Note** A large percentage of falsified objectives and poor model coverage often indicate that you need to change model parameter values to get complete coverage. This can occur when you have tunable parameters in Constant blocks that are connected to enabled subsystems or to the trigger inputs of Switch blocks. In these situations, configure Simulink Design Verifier parameter support as described in the example “Specify Parameter Configuration for Full Coverage” on page 5-17.

---



## Modify the Analysis Parameters

If the analysis satisfied most but not all of the objectives, try the following steps:

- 1 Increase the **Maximum analysis time** parameter. This gives the analysis more time to satisfy all the objectives.
- 2 Set the **Model coverage objectives** parameter to **Decision**. Selecting this option generates only test cases that achieve decision coverage. These test cases are a subset of the **MCDC** option.
- 3 Rerun the analysis and review the report.

If the results are still not satisfactory, try the techniques described in the following sections.

## Stop the Analysis Before Completion

Watch the **Objectives processed** value in the log window. If about 50 percent of the **Maximum analysis time** parameter has elapsed and this value does not increase, the model analysis may have trouble processing certain objectives. If the analysis does not progress, take the following steps:

- 1 Click **Stop** in the log window.

A dialog box appears, informing you that the analysis was aborted and asking you if you still want to produce results.

- 2 Click **Yes** to save the results of the analysis so far.

The log window lists the following options, depending on which analysis mode you ran:

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

- 3 Click **Generate detailed analysis report**.

- 4 In the HTML report, review the following sections to identify the model elements that are causing problems:

- **Objectives Undecided when the Analysis was Stopped**
- **Objectives Producing Errors**

- 5 Review the model elements that have undecided objectives or objectives with errors to see if these problems are present. Consult the pages in the **More Information** column for specific techniques to improve the analysis.

| Problem in Your Model | More Information  |
|-----------------------|---|
| Floating-point inputs | <ul style="list-style-type: none"> <li>• “Manage Model Data to Simplify the Analysis” on page 14-8</li> <li>• “Bottom-Up Approach to Model Analysis” on page 14-13</li> </ul> |
| Nonlinear operations  | <ul style="list-style-type: none"> <li>• “Logical Operations” on page 14-21</li> <li>• “Bottom-Up Approach to Model Analysis” on page 14-13</li> </ul>                        |

| <b>Problem in Your Model</b> | <b>More Information</b>   |
|------------------------------|---|
| Large state spaces           | <ul style="list-style-type: none"><li>• “Analyzing Models with Large Verification State Space” on page 14-22</li><li>• “Bottom-Up Approach to Model Analysis” on page 14-13</li></ul> |
| Large timers and time delays | <ul style="list-style-type: none"><li>• “Counters and Timers” on page 14-23</li><li>• “Bottom-Up Approach to Model Analysis” on page 14-13</li></ul>                                  |

## Increase Allocated Memory for Analysis Report Generation

When you analyze a model with a large root-level input signal count, you may encounter an insufficient memory error when Simulink Design Verifier is generating the report.

When this occurs, you need to increase the amount of memory the Sun® Java® Virtual Machine (JVM™) software can allocate. For steps on how to increase this memory, see “Increase the MATLAB JVM Memory Allocation Limit” (MATLAB Report Generator).

## Manage Model Data to Simplify the Analysis

### In this section...

“Simplify Data Types” on page 14-8

“Constrain Data” on page 14-8

### Simplify Data Types

One way to simplify your model is to use for the designated signal data type a data type requiring the least amount of space for the expected data. For example, do not use an `int` data type for Boolean data, because only one bit is required for Boolean data.

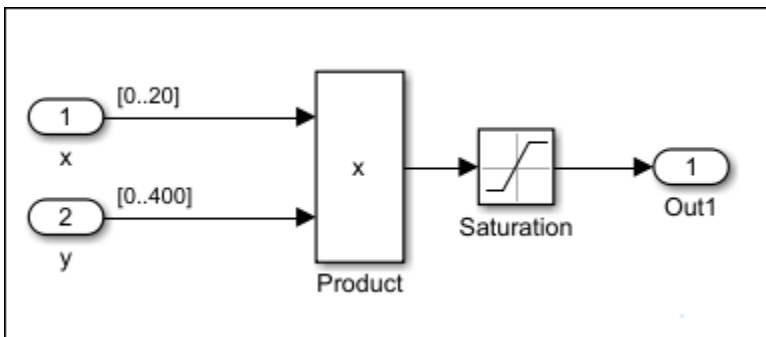
In another example, suppose you have a Sum block with two inputs that are always integers between -10 and 10. Set the **Output data type** parameter to `int8`, rather than `int32` or `double`.

To display the signal data types, on the **Debug** tab, click **Information Overlays > Port Data Type**.

### Constrain Data

Another effective technique for reducing complexity is to restrict the inputs to a set of representative values or, ideally, a single constant value. This process, called discretization, treats the input as if it were an enumeration. Discretization allows you to handle nonlinear arithmetic from multiplication and division in the simplest way possible.

The following model has a Product block feeding a Saturation block. The inputs `x` and `y` have specific design ranges as shown and the Saturation block limits the input signal to the upper and lower saturation values which is 8000 and 0 RPM.



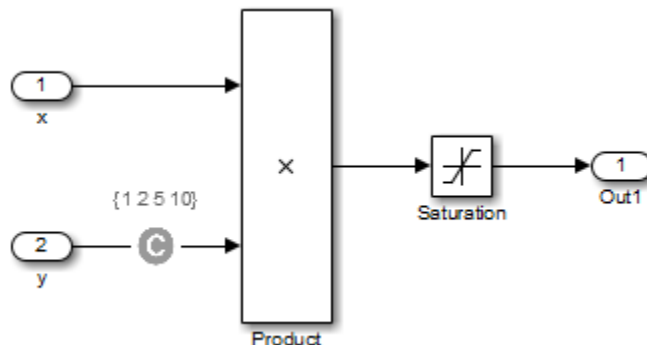
The Simulink Design Verifier software generates errors when attempting to satisfy the upper and lower limits of the Saturation block, because the software does not support nonlinear arithmetic. To work around these errors, restrict one of the inputs to a set of discrete values.

Identify discrete values that are required to satisfy your testing needs. For example, you may have an input for model speed, and your design contains paths of execution that are conditioned on speed above or below thresholds of 80, 150, 600, and 8000 RPM. For an effective analysis, constrain speed values to be 50, 100, 200, 1000, 5000, or 10000 RPM so that every threshold can be either active or inactive.

If you need to use more than two or three values, consider specifying the constrained values using an expression like

```
num2cell(minval:increment:maxval)
```

Using the previous example model, restrict the second input ( $y$ ) to be either 1, 2, 5, or 10 using the Test Condition block as shown in the following model. The Simulink Design Verifier software produces test cases for all inputs.



You can also constrain signals that are intermediate or output values of the model. Constraining such signals makes it easier to work around multiplication or division inside lower level subsystems that do not depend on model inputs.

---

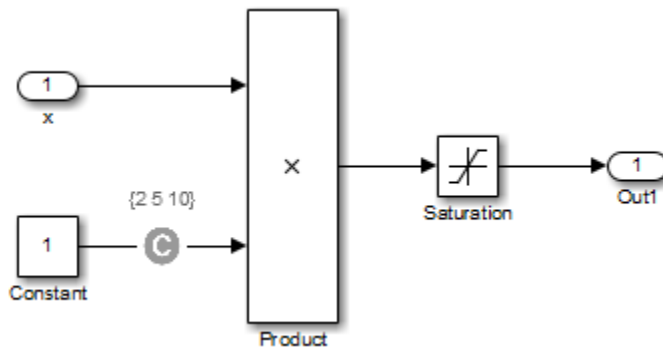
**Note** Discretization is best limited to a small number of inputs (less than 10). If your model requires discretization of many inputs, try to achieve model coverage through successive simulations, as described in “Partition Model Inputs for Incremental Test Generation” on page 14-11.

---

Test Condition blocks do not need to be placed exactly on the inputs. In deciding where to place the constraints in your model, consider the following guidelines:

- Favor constraints on the input values because the software can process inputs easier.
- If you need to place constraints on both the input and the output, for example, to avoid nonlinear arithmetic, one of the constraints should be a range such as  $[\text{minval } \text{maxval}]$ . The software first tests the values at both ends of the range and can return a test case, even if the underlying calculations are nonlinear.
- Make sure that constraints at corresponding input and output points are not contradictory. Do not constrain the output signals to values that are not achievable because of the constraints on the input values.
- Avoid creating constraints that contradict the model. Such contradictions occur when a constraint can never be satisfied because it contradicts some aspect of the model or another constraint. Analyzing contradictory models can cause Simulink Design Verifier to hang.

The next model is a simple example of a contradictory model. The second input to the Multiply block is the constant 1, but the Test Condition block constrains it to a value of 2, 5, or 10. The analysis cannot achieve all the test objectives in this model.



- When you work with large models that have many multiplication and division operations, you may find it easier to add constraints to all of the floating-point inputs rather than to identify the precise set of inputs that require constraints.

## Partition Model Inputs for Incremental Test Generation

As described in “Constrain Data” on page 14-8, you can constrain the values of model inputs using the Simulink Design Verifier Test Condition block.

Like other Simulink parameters, constraint values can be shared across several blocks by referencing a common workspace variable; you can initialize constraint values using MATLAB commands. If you have several inputs related to speed, such as desired speed, measured speed, and average speed, you might choose to constrain all of them to the same set of values.

As an advanced technique for experienced MATLAB programmers, you can use parameterized constraints and successive runs of Simulink Design Verifier to implement an incremental test generation technique:

- 1 Partition model inputs so that some are held constant, some are constrained to sets of constants using the Test Condition block, and some can have any value.
- 2 Generate test cases and run those test cases to collect model coverage.
- 3 Choose new values and partition the inputs with these new values.
- 4 Generate test cases for missing coverage using the `sldvgencov` function and the current test coverage.

---

**Note** To view an example of extending an existing test suite to achieve missing model coverage, enter the following at the command prompt in the MATLAB Command Window:

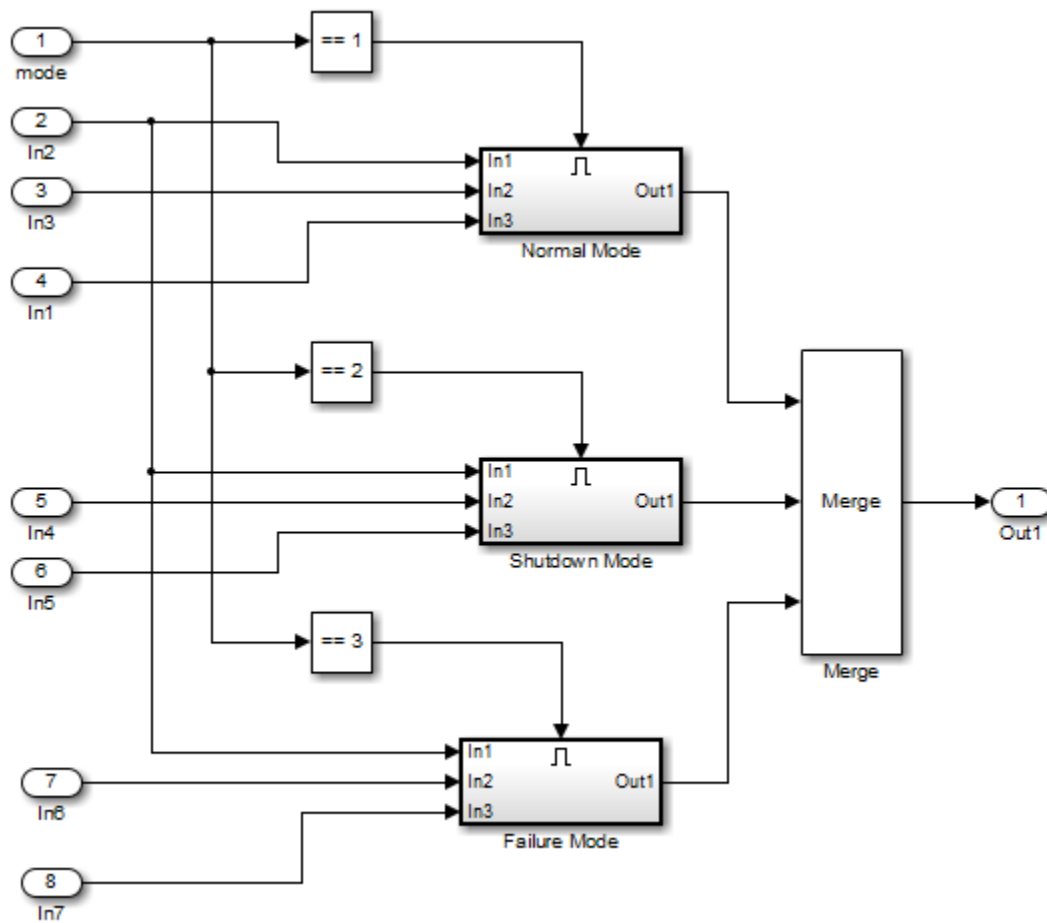
```
showdemo('sldvdemo_incremental_test_generation')
```

---

- 5 Repeat steps 3 and 4 until you have achieved the desired coverage.

Partition the model inputs that enable further simplification when an analysis runs. Consider the following model, which has three mutually independent enabled subsystems:

- Normal Mode
- Shutdown Mode
- Failure Mode



You can incrementally generate test cases for each subsystem by constraining the first input to a constant value before running an analysis. In this way, as you create test cases for each subsystem, the software ignores the complexity of the other two subsystems.



## Bottom-Up Approach to Model Analysis

### In this section...

“Reuse of Analysis Results from Subsystems at the System level” on page 14-13

“Limitations” on page 14-14

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 default `Auto test suite optimization`.

The bottom-up approach offers several advantages:

- It allows you to solve the problems that slow down error detection, test generation, or property proving in a controlled environment.
- Solving problems with small model components before analyzing the model as a whole is more efficient, especially if you have unreachable components in your model that you can only discover in the context of the model.
- You can iterate more quickly—find a problem and fix it, find another problem and fix it, and so on.
- If one model component has a problem—for example, a component is unreachable in simulation—that can prevent the software from generating tests for *all* the objectives in a large model.

Try this workflow with your large model:

- 1 Use the Test Generation Advisor to identify analyzable model components and generate tests for these components. For more information, see “Use Test Generation Advisor to Identify Analyzable Components” on page 7-24.
- 2 Fix any problems by adding constraints or specifying block replacements.
- 3 After you analyze the smaller components, reapply the required constraints and substitutions to the original model. Analyze the full model.

When you finish a bottom-up analysis, you have a top-level model that Simulink Design Verifier can analyze quickly.

### Reuse of Analysis Results from Subsystems at the System level

This section explains how the results for Simulink Design Verifier run on the unit level generalize to the system level. This could be used in certain circumstances as a replacement for running Simulink Design Verifier at the system level, or to restrict the checks that need to run at the system level.

These points describe how Simulink Design Verifier generalizes the results on the unit level to the system level:

- When the design errors prove to be valid or, if you find dead logic at the unit level, the same results are considered valid (or dead logic) at the integration level. Without the system context, analysis at the unit level allows for a less constrained set of behaviors than those experienced in a unit when running at the system level. In other words, when the design error is valid in an unconstrained setting, it is valid in the more constrained setting.
- When there are design errors or an absence of dead logic at the unit level, the results might be different at the integration level. You must then reanalyze these objectives at the integration level.

## Limitations

These limitations are for reusing of analysis results from subsystems, at the system level:

- If the configuration parameter values between the unit level and the system level differ, the Simulink Design Verifier results may change at the system level.
- If floating-point Inf/NaN check is run at the unit level, the inputs to the unit are assumed to be finite, and similarly if the subnormal check is run at the unit level, the inputs to the unit are assumed to be normal. If you need to consider Inf/NaN and subnormal as inputs to the unit level, consider either disabling these checks or analyzing at the integration level. For more information, see “Assumptions and Limitations” on page 6-33.
- If you use `sldvextract` function, in order to extract a unit for analysis, Simulink Design Verifier in some cases, inserts a Data Store Memory block and Data Store Read and/or Data Store Write blocks. For more information, see “Analyze Subsystems That Read from Global Data Storage” on page 14-16. This leads to a different simulation behavior for the unit level. Additionally, the data store access violation checks may experience different results.

# Extract Subsystems for Analysis

**In this section...**

“Overview of Subsystem Extraction” on page 14-15

“sldvextract Function” on page 14-15

“Structure of the Extracted Model” on page 14-15

“Analyze Subsystems That Read from Global Data Storage” on page 14-16

“Analyze Function-Call Subsystems” on page 14-17

“Analyze Global Simulink Function” 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 Simulink Design Verifier. This technique allows you to implement a bottom-up approach to analyzing a large model, as described in “Bottom-Up Approach to Model Analysis” on page 14-13.

When you analyze a subsystem or atomic subchart, the software:

- Extracts the subsystem or subchart into a new model.
- If required, adds blocks to the newly created model that replicate the execution context of the subsystem or subchart within its parent model.
- Analyzes the extracted model and produces results.

---

**Note** The Simulink Design Verifier software can only analyze atomic subsystems and atomic subcharts.

For more information about analyzing subsystems, see “Generate Test Cases for a Subsystem” on page 7-18.

For more information about analyzing atomic subcharts, see “Analyze a Stateflow Atomic Subchart” on page 1-17.

---

## sldvextract Function

The `sldvextract` function allows you to extract subsystems and atomic subcharts for component verification. By extracting the subsystem or atomic subchart, you can verify the component in isolation from the rest of the system, allowing you to test the component algorithm. For more information, see “What Is Component Verification?” on page 10-2 and “Functions for Component Verification” on page 10-3.

## Structure of the Extracted Model

When you analyze a subsystem or atomic subchart, Simulink Design Verifier creates a new model that contains the subsystem or atomic subchart, and any input and output ports that correspond to the ports connected to the original subsystem.

The software assigns the following properties to the ports in the new model, as determined by compiling the original model:

- Data types
- Sample rates
- Signal dimensions
- Minimum and maximum values of the signal ranges

The software names the new model *subsystem\_name*, where *subsystem\_name* is the name of the subsystem.

The next sections provide examples of how Simulink Design Verifier extracts and analyzes subsystems.

## Analyze Subsystems That Read from Global Data Storage

A data store is a repository to which you can write data, and from which you can read data, without having to connect an input or output signal directly to the data store.

You create a data store using a Data Store Memory block or a `Simulink.Signal` object. The Data Store Memory block or `Simulink.Signal` object represents the data store and specifies its properties. Every data store must have a unique name.

When you analyze a subsystem that reads data from a data store that is accessed outside the subsystem, the analysis:

- Adds a Data Store Memory block to the new model.
- Adds an input port that writes to the data store. Since the input writes to the data store, the data can have any values (within the specified data type) for the purpose of the Simulink Design Verifier analysis.
- If the data store specifies minimum and maximum values, those values are assigned to the new input port.

The following example analyzes a subsystem in the `sl_subsys_fcncall8` example model:

1. Open the `sl_subsys_fcncall8` example model:

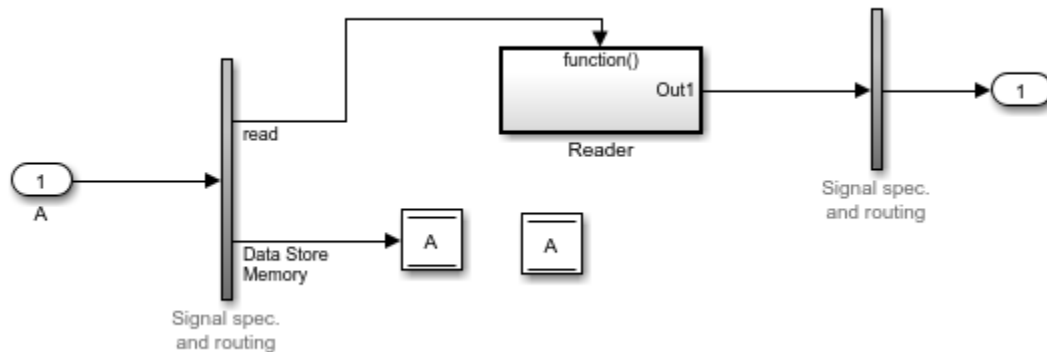
```
open_system('sl_subsys_fcncall8');
```

This model defines a data store A, from which the atomic subsystem Reader reads data using a Data Store Read block.

2. Right-click the Reader subsystem and select **Design Verifier > Generate Tests** for Subsystem.

The Simulink Design Verifier log window shows that the software extracts the subsystem into a new model named `Reader`, analyzes the extracted model, and offers you the choice of which results to produce.

3. Open the new Reader model that the software created in `<current_folder>\sldv_output\Reader`.



The new Inport block A writes into the data store, which is used by the subsystem Reader in the new model.

## Analyze Function-Call Subsystems

A function-call subsystem is a triggered subsystem whose execution is determined by logic internal to a C MEX S-function instead of by the value of a signal. Function-call subsystems are always atomic.

Note: For more information, see “Implement Function-Call Subsystems with S-Functions”.

When you analyze a model with a function-call subsystem, Simulink Design Verifier creates a new model with an Inport block that mimics the trigger and a copy of the subsystem. The software then analyzes the new model.

The following example analyzes a function-call subsystem in the `sl_subsys_fcncall2` model:

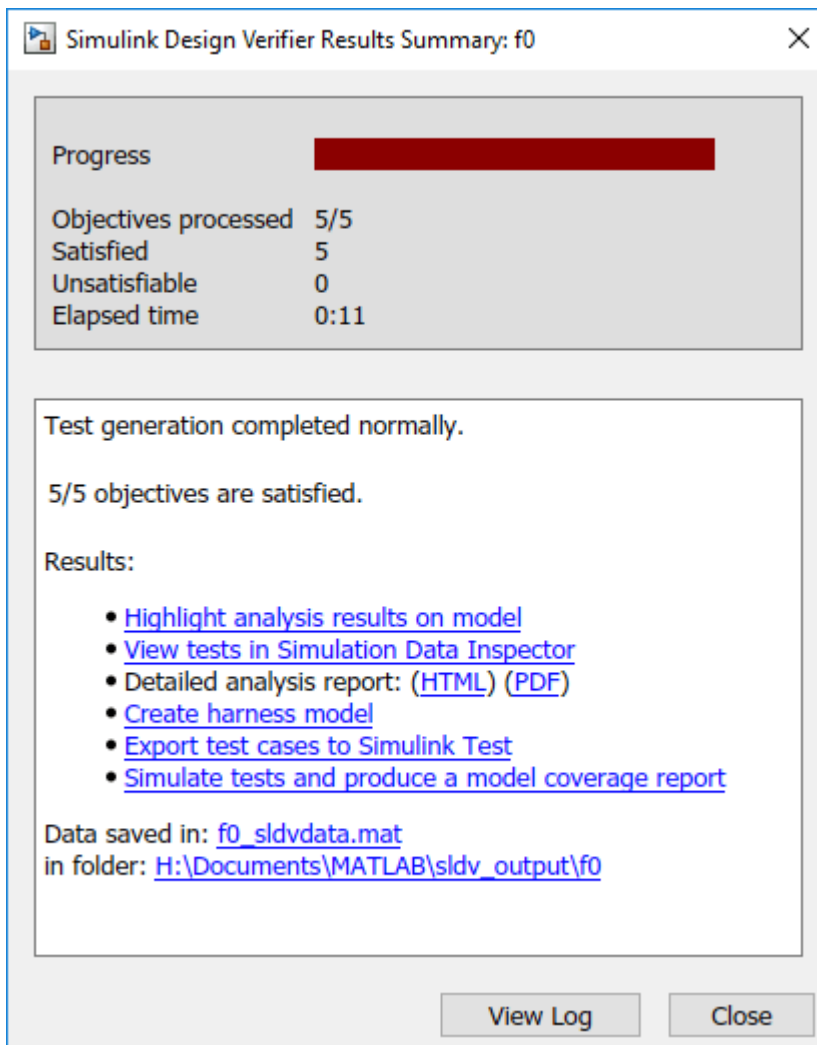
1. Open the `sl_subsys_fcncall2` example model:

```
open_system("sl_subsys_fcncall2");
```

2. This model contains a Stateflow™ chart named Chart that triggers the function-call subsystem `f`.

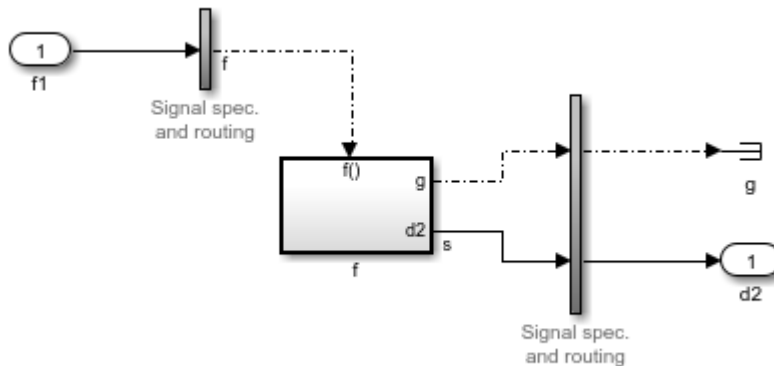
Right-click the `f` subsystem and select **Design Verifier > Generate Tests** for Subsystem.

The software extracts the subsystem into a new model named `f0`, analyzes the extracted model, and produces results.



3. Open the f0 model that the software created in `<current_folder>\sldv_output\f0`.

The Inport block and the new subsystem block mimic the trigger for the function-call subsystem f in the new f0 model.



## Analyze Global Simulink Function

A Simulink® function is a computational unit that calculates a set of outputs when provided with a set of inputs.

When you analyze Simulink Function subsystem, Simulink Design Verifier™ creates a new model containing a MATLAB function block `_SldvExportFcnScheduler` and a copy of the subsystem. This MATLAB Function block invokes Simulink Functions aperiodically and is driven by inports which represent the input arguments of the Simulink Function. An additional Inport block called `FcnTriggerPort`, the value of which indicates whether to invoke a particular function in a time step or not.

The following example analyzes a global Simulink function in the `sldvexGlobalSimFcn` model:

1. Open the `sldvexGlobalSimFcn` model.

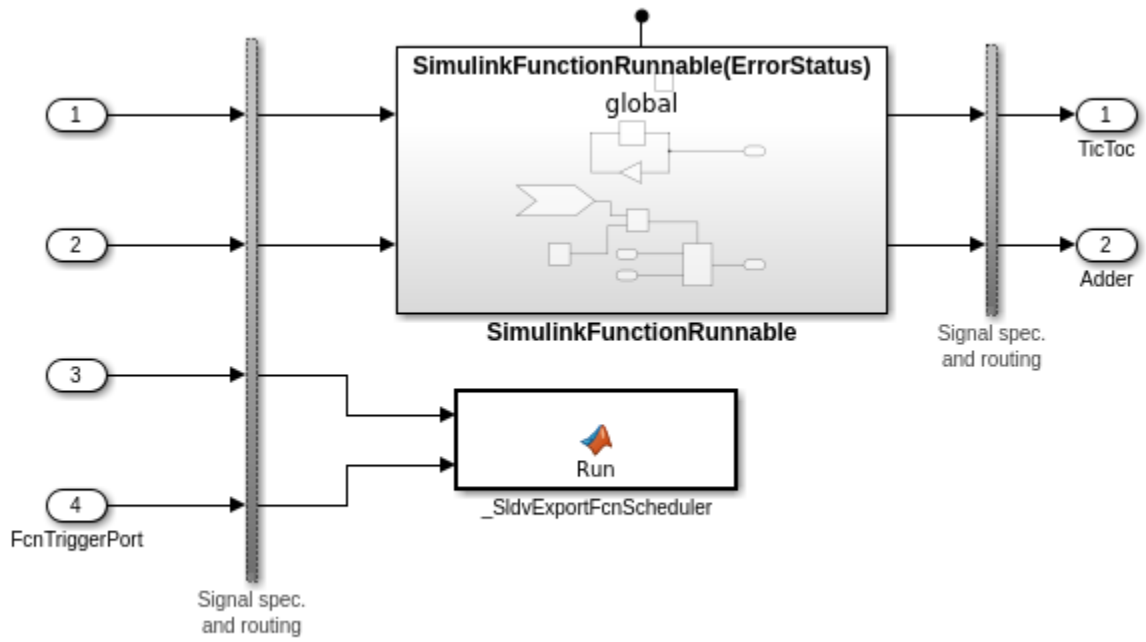
```
open_system("sldvexGlobalSimFcn");
```

2. Right-click the subsystem and select **Design Verifier > Generate Tests** for Subsystem.

The software extracts the subsystem into a new model and analyzes the extracted model, and produces results.

3. Open the new model `SimulinkFunctionRunnable0` that the software creates in `<current_folder>\sldv_output\`.

The Inport block `FcnTriggerPort`, invokes the Simulink Function `SimulinkFunctionRunnable` in the new `SimulinkFunctionRunnable0` model.





## Logical Operations

If you have a Simulink model with both logical and arithmetic operations, consider analyzing only the logical operations.

The Simulink Design Verifier software does not support nonlinear arithmetic of floating-point numbers, as occurs with multiplication or division, unless one of the multiply operands or the divisor is a constant.

To simplify models that contain integers or floating-point numbers, the software maps the model computations into expressions of Boolean variables. For example, the software might represent an eight-bit number as a set of eight Boolean values, with one for each digit. It might represent a bit-wise OR operation of two eight-bit integers as eight separate logical OR operations.

Mapping problems of one data type into Boolean variables is complex, and this complexity increases when the software performs such mapping. The software handles models with predominantly logical signals more efficiently than it does those with large integer or floating-point signals.

---

**Note** Simulink Design Verifier software can handle floating-point inputs when their values impact the design through linear inequalities such as  $x < y$  or  $a > 0$ .

In addition, input complexity can result from certain cast operations. For example, casting a double to an `int8` can introduce a non-linearity in certain situations.

---

## Analyzing Models with Large Verification State Space

Persistent design variables (variables that are assigned in one time step and used in a later time step during simulation) affect the complexity of analysis in much the same way as input complexity. You can use one or more of the following techniques to simplify the complexity of the state space you want to search:

- Apply constraints to input signals that are delayed.
- Constrain the inputs to states that are contained within conditionally executed subsystems.
- Limit the number of test case steps by setting the **Maximum test case step** parameter to 20.
- Increase the sample time for part or all of the model. (This procedure is similar to reducing timer thresholds, as described in “Counters and Timers” on page 14-23.) A test case that you generate at a lower sample rate often has similarities to the test case with a high sample rate that you need to achieve an objective.
- Use tight variable types where ever possible. For example, if a flag with values of 0 or 1 only is defined as a `double`, restrict the type to `Boolean`.

States that are computed from previous state values present a special challenge. For example, if you want to restrict the integrator value in a PID controller, you can only use a set of values that includes all reachable values from the initial value. Otherwise, the input must be forced to 0. Neither of these limitations is practical and would probably make the analysis less complete.

Alternatively, you can use existing simulation data to help satisfy your testing needs. If you have existing test data, run it on your model and collect model coverage. For an example of extending an existing test suite to achieve missing model coverage, see “Extend an Existing Test Suite” on page 7-86.

## Counters and Timers

Simulink Design Verifier analysis searches through sequences of states to find input values that drive the analysis to reach a state that satisfies an objective. Each counter value or timer step corresponds to a different state, so the presence of long timers or counters can dramatically increase the size of the state representation. Since analysis complexity depends on the size of the state representation, you must give special consideration to counters and timers in your model to avoid over complicating Simulink Design Verifier analysis.

---

**Note** For the purposes of Simulink Design Verifier analysis, the term configuration refers to a set of values for all the persistent information in your model.

---

The search process investigates all configurations that can be reached in a single timer step before considering any of the configurations that can be reached in two timer steps. Likewise, the search investigates all configurations that can be reached in two timer steps before it considers any configuration that requires three or more timer steps, and so on. The number of timer steps required to exhaust the counter directly affects the number of states that the analysis needs to search. Models that contain time delays, such as countdown timers, complicate the analysis by forcing the search to span a large number of states.

You may see similar effects when systems use extensive averaging and filtering to delay the response to a change in inputs. Any aspect of the design that delays the response causes the test sequences to contain more timer steps, resulting in longer test cases that are more difficult to identify.

Some basic techniques you can use to improve analysis performance in models with counters or timers include the following:

- Choose very small values for time delays. A system with a logical error when a time delay is set to 2000 steps usually demonstrates that error if the time delay is changed to 2 steps. If your system has several delays, choose small but unique values for each of them so that your delays are progressively satisfied.
- Make the initial values of counters and timers parameter values that Simulink Design Verifier can modify. The software finds initial values that allow shorter test cases to exceed thresholds. For more information, see “Parameter Configuration for Analysis” on page 5-2.
- Choose higher frequency cutoffs for filters and fewer samples to average to minimize filtering delays.

Some more advanced techniques you can use to improve analysis performance in models with counters or timers include the following:

- Use `sldvtimer` to identify timer patterns that can be optimized for Simulink Design Verifier test generation.
- Use an existing test case or set of test cases that exhausts the counter or timer, and extend those test cases to create a full test suite. For more information, see “Defining and Extending Existing Tests Cases” on page 7-91.

## Prove Properties in Large Models

Property proving uses the same underlying techniques as design error detection and test generation and suffers from the same performance limitations. However, unlike design error detection or test generation, you often cannot simplify the problem without compromising the validity of the results.

You can quickly prove simple proof objectives that are not affected by model dynamics. However, a thorough proof requires that Simulink Design Verifier search through all reachable configurations of your model—even the ones that are reached only after long time delays. The computation time and memory required to search a model completely often make an exhaustive proof impractical.

There are two techniques you can use to improve the performance of property proving in a large model:

### In this section...

“Find Property Violations While Designing Your Model” on page 14-24

“Combine Proving Properties and Finding Proof Violations” on page 14-24

### Find Property Violations While Designing Your Model

Simulink Design Verifier software offers a strategy that quickly identifies property violations in larger, more complicated models. While designing your model, analyze your model using this strategy so that you can fix any property violations before finalizing your design.

To identify property violations of a model, on the **Design Verifier > Property Proving** pane of the Configuration Parameters dialog box, specify the value of the **Strategy** parameter as **FindViolation**. When you use this strategy, the **Maximum violation steps** parameter becomes active so that you can specify an upper bound for the number of time steps in the search.

During analysis, the software searches only for property violations within the specified number of time steps. By identifying and fixing the property violations first, you improve the performance of a property-proving analysis that uses the **Prove** strategy.

If a violation is not detected, it is impossible to violate the property with any input sequence having fewer time steps than the specified limit. However, you cannot prove that the property is true because there might be a counterexample within more time steps than the specified limit.

### Combine Proving Properties and Finding Proof Violations

Use the following technique for proving properties in large model. This technique combines proving and searching for violations:

- 1 On the **Design Verifier > Property Proving** pane, set the **Strategy** parameter to **Prove**.
- 2 On the **Design Verifier** pane, use a relatively short value for the **Maximum analysis time** parameter, such as 5-10 minutes. If trivial counterexamples exist — or if your properties do not depend on model dynamics—the analysis should complete in that amount of time.
- 3 Change the **Strategy** parameter to **FindViolation**, and choose a small bound for the **Maximum violation steps** parameter, such as 4, 5, or 6. If your properties have simple counterexamples, the software should discover them.
- 4 If you do not find any violations with a small bound, increase the bound and look for longer counterexamples.

- a** Increase the bound in several increments, and observe the processing time and memory consumption. System resources might limit the length of violation that can be searched.
  - b** In addition, consider the dynamics of your model and the number of time steps required to transition between an arbitrary pair of configurations. If you choose too large a bound, the violation search can be more complex than the unbounded proof.
- 5** If you can run violation searches with relatively large bounds, e.g., 30-50 time steps, switch back to the Prove strategy, and use a longer time limit, such as several hours.



# Simulink Design Verifier Configuration Parameters

---

- “Simulink Design Verifier Options” on page 15-2
- “Design Verifier Pane” on page 15-9
- “Design Verifier Pane: Block Replacements” on page 15-19
- “Design Verifier Pane: Parameters and Variants” on page 15-22
- “Design Verifier Pane: Test Generation” on page 15-30
- “Design Verifier Pane: Design Error Detection” on page 15-42
- “Design Verifier Pane: Property Proving” on page 15-52
- “Design Verifier Pane: Results” on page 15-56
- “Design Verifier Pane: Report” on page 15-63

## Simulink Design Verifier Options

### In this section...

“Options in Configuration Parameters Dialog Box” on page 15-2

“Design Verification Options Objects” on page 15-2

“Command-Line Parameters for Design Verification Options” on page 15-2

### Options in Configuration Parameters Dialog Box

You can set options for Simulink Design Verifier analysis in the Configuration Parameters dialog box. To view the options, open **Design Verifier** tab. In the **Prepare** section, from the drop-down menu for the mode settings, and click **Settings**. The **Design Verifier** pane of the model configuration parameters opens.

By default, options for Simulink Design Verifier do not appear in the Configuration Parameters dialog box. When you open the **Design Verifier** tab, Simulink Design Verifier associates its default options with the model. After you save the model, you can access options for Simulink Design Verifier directly from the Configuration Parameters dialog box.

See “Set Model Configuration Parameters for a Model” for more information about working with this interface.

### Design Verification Options Objects

You can use the `sldvoptions` function to specify Simulink Design Verifier options at the command line.

To view in the MATLAB Command Window the design verification options associated with a Simulink model, use the following syntax:

```
opts = sldvoptions('model_name');
get(opts)
```

### Command-Line Parameters for Design Verification Options

Use the following parameters to configure the behavior of Simulink Design Verifier. Use the `get_param` and `set_param` functions to retrieve and specify values for these parameters programmatically.

For each parameter, the **Location** column indicates where you can set its value in the Configuration Parameters dialog box. The **Values** column shows the type of value required, the possible values (separated with a vertical line), and the default value (enclosed in braces).

| Parameter           | Location   | Values               |
|---------------------|--|----------------------|
| DVAbsoluteTolerance | Set by the <b>Floating point absolute tolerance</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane. | double { '1.0e-05' } |



| Parameter                        | Location   | 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.                      | character array {'\$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.                    | character array {'<FactoryDefaultRules>'}         |
| DVCodeAnalysisExtraOptions       | Set by the <b>Additional options for code analysis</b> parameter on the <b>Design Verifier</b> pane.                                       | character array {''}                              |
| DVCoverageDataFile               | Set by the <b>Coverage data file</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.                                    | character array {''}                              |
| DVCovFilter                      | Set by the <b>Ignore objectives based on filter</b> parameter on the <b>Design Verifier</b> pane.  | 'on'   {'off'}                                    |
| DVCovFilterFileName              | Set by the <b>Filter file(s)</b> parameter on the <b>Design Verifier</b> pane.   | character array {''}                              |
| DVDataFileName                   | Set by the <b>Data file name</b> parameter on the <b>Design Verifier &gt; Results</b> pane.  | character array {'\$modelName \$_sldvdata'}       |
| DVDeadLogicObjectives            | Set by the <b>Coverage objectives to be analyzed</b> parameter on the <b>Design Verifier &gt; Design Error Detection</b> pane.             | 'Decision'   {'ConditionDecision'}   'MCDC'       |
| DVDesignMinMaxCheck              | Set by the <b>Specified minimum and maximum value violations</b> parameter on the <b>Design Verifier &gt; Design Error Detection</b> pane. | 'on'   {'off'}                                    |
| DVDesignMinMaxConstraints        | Set by the <b>Use specified input minimum and maximum values</b> parameter on the <b>Design Verifier</b> pane.                             | {'on'}   'off'                                    |

| Parameter                          | Location   | Values                                    |
|------------------------------------|--|---|
| DVDetectActiveLogic                | Set by <b>Run exhaustive analysis</b> on the <b>Design Verifier &gt; Design Error Detection</b> pane.                                | 'on'   {'off'}                            |
| DVDetectBlockInputRange-Violations | Set by <b>Specified block input range violations</b> on the <b>Design Verifier &gt; Design Error Detection</b> pane.                 | 'on'   {'off'}                            |
| DVDetectDeadLogic                  | Set by <b>Dead logic (partial)</b> 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'                            |
| DVDetectDSM-AccessViolations       | Set by the <b>Data store access violations</b> parameter on the <b>Design Verifier &gt; Design Error Detection</b> pane.             | 'on'   {'off'}                            |
| DVDetectInfNaN                     | Set by the <b>Non-finite and NaN floating-point values</b> parameter on the <b>Design Verifier &gt; Design Error Detection</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'                            |
| DVDetectOutOfBounds                | Set by the <b>Out of bound array access</b> parameter on the <b>Design Verifier &gt; Design Error Detection</b> pane.                | {'on'}   'off'                            |
| DVDetectSubnormal                  | Set by the <b>Subnormal floating-point values</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'                            |
| 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.                                       | character array {' '}                     |
| DVHarnessModelFileName             | Set by the <b>Harness model file name</b> parameter on the <b>Design Verifier &gt; Results</b> pane.                                 | character array {'\$modelName\$_harness'} |

| Parameter                    | Location   | Values  |
|------------------------------|--|---|
| DVHarnessSource              | Set by the <b>Harness source</b> parameter on the <b>Design Verifier &gt; Results</b> pane.  | {'Signal Builder'}   'Signal Editor'                                  |
| 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'}  |
| DVIgnoreExistTestSatisfied   | 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'  |
| DVIncludeRelational-Boundary | Set by the <b>Include relational boundary objectives</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'  |
| 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 {10000}   |
| 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'}   'DesignErrorDetection'   'PropertyProving'       |
| DVModelCoverageObjectives    | Set by the <b>Model coverage objectives</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.   | 'None'   'Decision'   {'ConditionDecision'}   'MCDC'   'EnhancedMCDC' |
| 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 <b>Output folder</b> on the <b>Design Verifier</b> pane.  | character array {'sldv_output/\$modelName\$'}                         |

| Parameter                     | Location  | Values  |
|-------------------------------|---|---|
| DVParameterConstraints        | Set by <b>Constraint</b> column in Parameter Table on the <b>Design Verifier &gt; Parameters</b> pane.  | double array {[]}   |
| DVParameterNames              | Set by <b>Name</b> column in Parameter Table on the <b>Design Verifier &gt; Parameters</b> pane.  | double array {[]}   |
| DVParameterUseInAnalysis      | Set by <b>Use</b> column in Parameter Table on the <b>Design Verifier &gt; Parameters</b> pane.   | cell array {[]}   |
| DVParameters                  | Set by <b>Enable parameter configuration</b> on the <b>Design Verifier &gt; Parameters</b> pane.  | 'on'   {'off'}  |
| DVParametersConfigFileName    | Set by <b>Parameter configuration file</b> on the <b>Design Verifier &gt; Parameters</b> pane.<br><br>This parameter is disabled when DVParametersUseConfig is set to 'on'. | character array<br>{'sldv_params_template.m'}               |
| DVParametersUseConfig         | Set by <b>Use parameter table</b> on the <b>Design Verifier &gt; Parameters</b> pane.<br><br>When set to 'on', this parameter disables DVParametersConfig-FileName.         | 'on'   {'off'}  |
| 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 do not affect the outcome</b> parameter on the <b>Design Verifier &gt; Results</b> pane.  | 'on'   {'off'}  |
| DVRebuildModel-Representation | Set by the <b>Rebuild model representation</b> parameter on the <b>Design Verifier</b> pane.  | 'Always'   {'If change is detected'}                        |
| DVReduceRationalApprox        | Set by the <b>Run additional analysis to reduce instances of rational approximation</b> parameter on the <b>Design Verifier</b> pane.                                       | {'on'}   'off'  |

| Parameter               | Location  | Values  |
|-------------------------|---|---|
| DVRelativeTolerance     | Set by the <b>Floating point relative tolerance</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.      | double {'0.01'}                                   |
| DVReportFileName        | Set by the <b>Report file name</b> parameter on the <b>Design Verifier &gt; Report</b> pane.                                | character array {'\$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'}                                    |
| DVReportPDFFormat       | Set by the <b>Generate additional report in PDF format</b> parameter on the <b>Design Verifier &gt; Report</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'}                                    |
| DVSaveHarnessModel      | Set by the <b>Generate separate harness model after analysis</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'}                                     |
| DVSupportSFunctions     | Set by the <b>Support S-Functions in the analysis</b> parameter on the <b>Design Verifier</b> pane.                         | {'on'}   off'                                     |
| DVTestHarnessName       | Set by the <b>Test Harness Name</b> parameter on the <b>Design Verifier &gt; Results</b> pane.                              | character array {'\$ModelName \$ _sldvharness'}   |
| DVTestFileName          | Set by the <b>Test File Name</b> parameter on the <b>Design Verifier &gt; Results</b> pane.                                 | character array {'\$ModelName \$ _test'}          |
| DVStrictEnhancedMCDC    | Set by the <b>Use strict propagation conditions</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.      | 'on'   {'off'}                                    |
| DVTestConditions        | Set by the <b>Test conditions</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.                        | 'EnableAll'   'DisableAll'   {'UseLocalSettings'} |
| DVTestgenTarget         | Set by the <b>Test generation target</b> parameter on the <b>Design Verifier &gt; Test Generation</b> pane.                 | {'Model'}   'GenCodeTopModel'   'GenCodeModelRef' |

| Parameter               | Location  | Values  |
|-------------------------|---|---|
| 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.<br><br>If you analyze your model by using the Legacy <code>LargeModel (Nonlinear Extended)</code> , the software displays a warning message that this option has been removed and suggests that you use <code>Auto</code> instead. | {'Auto'}   'IndividualObjectives'   'LongTestcases'   'LargeModel (Nonlinear Extended)' |
| DVUseParallel           | Set by the <b>Validate test cases or counterexamples with parallel computing</b> parameter on the <b>Design Verifier</b> pane.  | 'on'   {'off'}  |

## See Also

### More About

- “Design Verifier Pane” on page 15-9
- `sldvoptions`

## Design Verifier Pane

Analysis options

Mode:

Maximum analysis time (s):

Output

Output folder:

Make output file names unique by adding a suffix

▼ Advanced parameters

Rebuild model representation:

Automatic stubbing of unsupported blocks and functions

Support S-Functions in the analysis

Use specified input minimum and maximum values

Run additional analysis to reduce instances of rational approximation

Validate test cases or counterexamples with parallel computing

Additional options for code analysis:

Exclude and justify objectives

Ignore objectives based on filter

Filter file:

### In this section...

“Design Verifier Pane Overview” on page 15-10

“Mode” on page 15-10

“Maximum analysis time” on page 15-11

“Output folder” on page 15-11

“Make output file names unique by adding a suffix” on page 15-12

“Check Model Compatibility” on page 15-13

“Generate Tests/Detect Errors/Prove Properties” on page 15-13

**In this section...**

“Rebuild model representation” on page 15-13  
 “Automatic stubbing of unsupported blocks and functions” on page 15-13  
 “Support S-Functions in the analysis” on page 15-14  
 “Use specified input minimum and maximum values” on page 15-15  
 “Run additional analysis to reduce instances of rational approximation” on page 15-15  
 “Validate test cases or counterexamples with parallel computing” on page 15-16  
 “Additional options for code analysis” on page 15-17  
 “Ignore objectives based on filter” on page 15-17  
 “Filter file(s)” on page 15-18  
 “Browse...” on page 15-18

**Design Verifier Pane Overview**

Specify analysis options and configure Simulink Design Verifier output.

**Mode**

Specify the analysis mode for Simulink Design Verifier.

**Settings**

**Default:** Test generation

Design error detection

Detects integer and fixed-point overflow errors and division-by-zero errors in a model

Test generation

Generates test cases for a model.

Property proving

Proves properties of a model.

**Tip**

Simulink Design Verifier specifies the value of this option when you select one of these analysis options from the **Design Verifier** tab, in the **Mode** section:

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

**Dependency**

When you set the **Mode** parameter, the button below **Check Model Compatibility** changes as follows:



- **Mode:** Test generation, button reads: **Generate Tests**
- **Mode:** Design error detection, button reads: **Detect Errors**
- **Mode:** Property proving, button reads: **Prove Properties**

### Command-Line Information

**Parameter:** DVMode

**Type:** character array

**Value:** 'TestGeneration' | 'DesignErrorDetection' | 'PropertyProving'

**Default:** 'TestGeneration'

### See Also

- “Overview of the Simulink Design Verifier Workflow” on page 1-19
- “What Is Design Error Detection?” on page 6-2
- “What Is Test Case Generation?” on page 7-3
- “What Is Property Proving?” on page 12-2

## Maximum analysis time

Specify the maximum time (in seconds) that Simulink Design Verifier spends analyzing a model. You can set the value of maximum analysis time to the value that you are willing to provide to the analysis. You can also stop the analysis at any time.

### Settings

**Default:** 300

The value that you enter represents the maximum number of seconds Simulink Design Verifier analyzes your model.

### Command-Line Information

**Parameter:** DVMaxProcessTime

**Type:** double

**Value:** any valid value

**Default:** 300

## Output folder

Specify a path name to which Simulink Design Verifier writes its output.

### Settings

**Default:** sldv\_output/\$ModelName\$

- Enter a path that is either absolute or relative to the current folder.
- \$ModelName\$ is a token that represents the model name.

**Tip**

You can use the following parameters to customize the names and locations of Simulink Design Verifier output:

- On the **Results** pane:
  - **Data file name**
  - **Harness model file name**
  - **Simulink Test options > Test File name**
- On the **Report** pane:
  - **Report file name**
  - **File path of the output model**
- On the **Block Replacements** pane:
  - **File path of the output model**

**Command-Line Information**

**Parameter:** DVOutputDir

**Type:** character array

**Value:** any valid path

**Default:** 'sldv\_output/\$ModelName\$'

**See Also**

“Review Analysis Results”

## Make output file names unique by adding a suffix

Specify whether Simulink Design Verifier makes its output file names unique by appending a numeric suffix.

**Settings**

**Default:** On

On

Appends an incremental numeric suffix to Simulink Design Verifier output file names. Selecting this option prevents the software from overwriting existing files that have the same name.

Off

Does not append a suffix to Simulink Design Verifier output file names. In this case, the software might overwrite existing files that have the same name.

**Command-Line Information**

**Parameter:** DVMakeOutputFilesUnique

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

**See Also**

“Review Analysis Results”

**Check Model Compatibility**

Run a check to assess your model for compatibility with Simulink Design Verifier. For more information, see “Simulink Design Verifier Checks”.

**Generate Tests/Detect Errors/Prove Properties**

When you set the **Mode** parameter, this button changes as follows:

- **Mode:** Test generation, button reads: **Generate Tests**  
For more information, see “What Is Test Case Generation?” on page 7-3.
- **Mode:** Design error detection, button reads: **Detect Errors**  
For more information, see “What Is Design Error Detection?” on page 6-2.
- **Mode:** Property proving, button reads: **Prove Properties**  
For more information, see “What Is Property Proving?” on page 12-2.

**Rebuild model representation**

Specify whether to rebuild model representation for Simulink Design Verifier analysis.

**Settings**

**Default:** If change is detected

Always

Always rebuild the model representation.

If change is detected

Rebuild the model representation only when the software detects any change in the model.

**Command-Line Information**

**Parameter:** DVRebuildModelRepresentation

**Type:** character array

**Value:** 'Always' | 'IfChangeIsDetected'

**Default:** 'If change is detected'

**See Also**

“Check Model Compatibility” on page 3-2

**Automatic stubbing of unsupported blocks and functions**

Specify whether to ignore unsupported blocks and functions during analysis.

### Settings

**Default:** On

On

Ignores unsupported blocks and functions and proceeds with the analysis.

Off

Displays a warning when Simulink Design Verifier encounters an unsupported block or function and asks if you want to continue the analysis.

### Command-Line Information

**Parameter:** DVAutomaticStubbing

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

“Handle Incompatibilities with Automatic Stubbing” on page 2-7

## Support S-Functions in the analysis

Specify whether to enable support for S-Functions that have been compiled to be compatible with Simulink Design Verifier.

### Settings

**Default:** On

On

Enables support for S-Functions that have been compiled to be compatible with Simulink Design Verifier.

Off

Simulink Design Verifier automatically stubs S-Functions during analysis.

### Command-Line Information

**Parameter:** DVSFcnSupport

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

“Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-28

“Configuring S-Function for Test Case Generation” on page 7-109

“Handle Incompatibilities with Automatic Stubbing” on page 2-7

## Use specified input minimum and maximum values

Specify whether to generate test cases that consider specified minimum and maximum values as constraints for all input signals in your model.

### Settings

**Default:** On

On

Considers specified minimum and maximum values as constraints for all input signals.

Off

Ignores any specified minimum and maximum values.

### Command-Line Information

**Parameter:** DVDesignMinMaxConstraints

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

“Minimum and Maximum Input Constraints” on page 11-2

## Run additional analysis to reduce instances of rational approximation

Specify whether Simulink Design Verifier attempts to reduce the use of rational approximation during analysis.

### Settings

**Default:** On

On

When you use Simulink Design Verifier to analyze models, Simulink Design Verifier attempts to reduce the use of rational approximation if the model. Enabling this setting may increase analysis time.

Off

Simulink Design Verifier does not attempt to reduce the use of rational approximation during analysis.

### Command-Line Information

**Parameter:** DVReduceRationalApprox

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

## Validate test cases or counterexamples with parallel computing

Specifies whether to validate test cases or counterexamples with parallel computing. This option requires a Parallel Computing Toolbox™ license.

### When to Use Parallel Computing for Validation

In general, parallel execution can help reduce the validation time if:

- You have a complex Simulink model that takes a long time to simulate.
- The Simulink Design Verifier analysis exceeds the maximum analysis time and results in a number of objectives with the Needs Simulation status. For more information, see “Objectives Satisfied - Needs Simulation” on page 13-46 and “Objectives Falsified - Needs Simulation” on page 13-49.
- The test generation analysis generates long test cases. This may be because you have set **Test suite optimization** to LongTestcases or the **Maximum test case steps** value is greater than the default value. For more information, see “Test Generation Pane Overview” on page 15-31.

The following points must be considered when using parallel computing for validation:

- Starting a parallel pool can take time, which impacts the overall analysis time. To reduce the analysis time:
  - Make sure that the parallel pool is already running before you run a test generation analysis. By default, the parallel pool shuts down after being idle for a specified number of minutes. To change the setting, see the topic 'Specify Your Parallel Preferences' in Parallel Computing Toolbox.
  - Load Simulink on all the parallel pool workers.
- The simulation occurs sequentially when:
  - The cluster is not local. Configure parallel preferences to use the local cluster only. To change the setting, see the topic 'Specify Your Parallel Preferences' in Parallel Computing Toolbox.
  - The model is in dirty state prior to launching the SLDV analysis.
  - The model has ToFile blocks.
  - The model is an internal harness.
- Cross-product features such as **functional testing and coverage analysis** from Simulink Test Manager do not support parallel computing for validation. For more information, see “Perform Functional Testing and Analyze Test Coverage” (Simulink Test).

### Settings

**Default:** Off

On

If you have a Parallel Computing Toolbox license, then Simulink Design Verifier validates test cases or counterexamples in parallel across multiple workers on the same machine.

Off

Simulink Design Verifier validates test cases or counterexamples in serial.

**Command-Line Information****Parameter:** DVUseParallel**Type:** character array**Value:** 'on' | 'off'**Default:** 'off'**See Also**

“How Simulink Design Verifier Reports Approximations Through Validation Results” on page 2-23

**Additional options for code analysis**

Specify additional options for analyzing S-functions that have been compiled to be compatible with Simulink Design Verifier. For more information, see “Support Limitations and Considerations for S-Functions and C/C++ Code” on page 3-28.

**Settings****Default:** ''

Enter additional options for analyzing S-Functions that have been compiled to be compatible with Simulink Design Verifier. For example, to specify the maximum size of arrays, enter `defaultArraySize = 512`.

**Command-Line Information****Parameter:** DVCodeAnalysisExtraOptions**Type:** character array**Value:** any valid option for analyzing S-Functions**Default:** ''**Ignore objectives based on filter**

Specify to analyze the model, ignoring the objectives in the **Filter file(s)**. The **Filter file(s)** contains the model coverage objectives for test generation, dead logic detection, and design error detection objectives that you want to filter from analysis.

**Settings****Default:** Off

On

Ignores objectives in the **Filter file(s)** during test generation and design error detection analysis.



Off

Generates results for all the objectives during test generation and design error detection analysis, including those in the **Filter file(s)**.

**Dependency**

This parameter enables **Filter file(s)**.

**Command-Line Information****Parameter:** DVCovFilter**Type:** character array**Value:** 'on' | 'off'**Default:** 'off'**See Also**

“Coverage Filtering” (Simulink Coverage)

**Filter file(s)**

Specify folder and file name(s) for the file(s) that contains the model coverage objectives and design error detection objectives that you want to filter from analysis.

**Settings****Default:** ''

- Specify the name of the folder and file name(s) that contain the objectives that you want to ignore from test generation and design error detection analysis.

Click the **Browse** button to select an existing **Filter file(s)**.

**Command-Line Information****Parameter:** DVCovFilterFileName**Type:** character array**Value:** valid file paths separated by comma or semi-colon**Default:** ''**See Also**

“Coverage Filter Rules and Files” (Simulink Coverage)

Filter Objectives by Using Analysis Filter Viewer on page 6-46

**Browse...**

Browse to the file that contains the objectives that you want to ignore from design error detection and test generation analysis.

**Dependency**

This button is enabled by **Ignore objectives based on filter**.



## Design Verifier Pane: Block Replacements

Block Replacements

Apply block replacements

List of block replacement rules (in order of priority):

Output model

File path of the output model:

### In this section...

- “Block Replacements Pane Overview” on page 15-19
- “Apply block replacements” on page 15-19
- “List of block replacement rules” on page 15-20
- “File path of the output model” on page 15-20

## Block Replacements Pane Overview

Specify options that control how Simulink Design Verifier preprocesses the models it analyzes.

### See Also

“Perform Block Replacement”

## Apply block replacements

Specify whether Simulink Design Verifier replaces blocks in a model before its analysis.

### Settings

**Default:** Off

On

Replaces blocks in a model before Simulink Design Verifier analyzes it.

Off

Does not replace blocks in a model before Simulink Design Verifier analyzes it.

## Dependencies

This parameter enables **List of block replacement rules** and **File path of the output model**.

### Command-Line Information

**Parameter:** DVBlockReplacement

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

“Perform Block Replacement”

## List of block replacement rules

Specify a list of block replacement rules that Simulink Design Verifier executes before its analysis.

### Settings

**Default:** <FactoryDefaultRules>

- Specify block replacement rules as a list delimited by spaces, commas, or carriage returns.
- The Simulink Design Verifier software processes block replacement rules in the order that you list them.
- If you specify the default value, Simulink Design Verifier uses its factory default block replacement rules.

### Dependency

This parameter is enabled when you select **Apply block replacements**.

### Command-Line Information

**Parameter:** DVBlockReplacementRulesList

**Type:** character array

**Value:** any valid rules

**Default:** '<FactoryDefaultRules>'

### See Also

“Perform Block Replacement”

## File path of the output model

Specify a folder and file name for the model that results after applying block replacement rules.

### Settings

**Default:** \$ModelName\$\_replacement

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.

- Enter a file name for the model that results after applying block replacement rules.
- `$ModelName$` is a token that represents the model name.

**Dependency**

This parameter is enabled when you select **Apply block replacements**.

**Command-Line Information**

**Parameter:** DVBlockReplacementModelFileName

**Type:** character array

**Value:** any valid path and file name

**Default:** '\$ModelName\$\_replacement'

**See Also**

“Perform Block Replacement”

## Design Verifier Pane: Parameters and Variants

Parameters

Parameter configuration: Treat all parameters as constants

Parameter specification

Parameter table

Enable Disable Clear Highlight in Model

| Use | Name | Constraint | Value | Min | Max | Model Element |
|-----|------|------------|-------|-----|-----|---------------|
|     |      |            |       |     |     |               |

Find parameters Import Export

Parameter configuration file: <empty> Browse... Edit...

Variants

Analyze all Startup Variants Launch Variant Manager...

OK Cancel Help Apply

### In this section...

- “Parameters Pane Overview” on page 15-23
- “Parameter configuration” on page 15-23
- “Enable” on page 15-23
- “Disable” on page 15-23
- “Clear” on page 15-23
- “Highlight in Model” on page 15-24
- “Use” on page 15-24
- “Name” on page 15-24
- “Constraint” on page 15-25
- “Value” on page 15-25
- “Min” on page 15-26
- “Max” on page 15-26
- “Model Element” on page 15-26
- “Find parameters” on page 15-27
- “Import” on page 15-27
- “Export” on page 15-27
- “Parameter configuration file” on page 15-27
- “Browse...” on page 15-28
- “Edit...” on page 15-28
- “Analyze all Startup Variants” on page 15-28
- “Launch Variant Manager...” on page 15-29

## Parameters Pane Overview

Specify options that control how Simulink Design Verifier uses parameter configurations when analyzing models.

### Parameter configuration

Specify the parameter configuration from these options available in drop-down:

- Treat all parameters as constants
- Automatically infer parameter specification. See “Automatically Infer Parameter Specification” on page 5-32
- Determine from generated code. See “Determine from Generated Code” on page 5-36
- Use parameter table. See “Use Parameter Table” on page 5-7
- Use parameter configuration file. See “Use Parameter Configuration File” on page 5-29

#### Settings

**Default:** None

#### Command-Line Information

**Parameter:** DVParameterConfiguration

**Type:** enum

**Value:** 'None' | 'Auto' | 'DetermineFromGeneratedCode' | 'UseParameterTable' | 'UseParameterConfigFile'

**Default:** 'None'

#### See Also

“Use Parameter Table” on page 5-7

### Enable

#### Dependency

This button is enabled by setting **Parameter configuration** to **Use parameter table**.

### Disable

#### Dependency

This button is enabled by setting **Parameter configuration** to **Use parameter table**.

### Clear

**Dependency**

This button is enabled by setting **Parameter configuration** to **Use parameter table**.

**Highlight in Model****Dependency**

This button is enabled by setting **Parameter configuration** to **Use parameter table**.

**Use**

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Use** column specifies whether to use this row's named parameter and specified constraint in the current parameter configuration.

**Settings**

**Default:** Off

On

Use this parameter and its specified constraint in the current parameter configuration.

Off

Do not use this parameter and its specified constraint in the current parameter configuration.

**Dependency**

This column is enabled by setting **Parameter configuration** to **Use parameter table**.

**See Also**

"Use Parameter Table" on page 5-7

**Name**

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Name** column displays the name of the parameter.

**Settings**

**Default:** empty

### Tips

To load the model parameters into the Parameter Table, at the bottom of the table, click **Find in Model**. When possible, the software automatically generates constraint values for each parameter.

### Dependency

This column is enabled by setting **Parameter configuration** to **Use parameter table**.

### See Also

“Use Parameter Table” on page 5-7

## Constraint

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Constraint** column contains the specified value range for the parameter.

### Settings

**Default:** empty

### Tips

To autogenerate parameter constraints, at the bottom of the Parameter Table, click **Find in Model**.

### Dependency

This column is enabled by setting **Parameter configuration** to **Use parameter table**.

### See Also

“Use Parameter Table” on page 5-7

## Value

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

The **Value** column contains the value of the parameter in the base workspace. If the parameter is defined in a Simulink data dictionary that is linked to the model, the **Value** column contains the value of the parameter in the data dictionary.

### Settings

**Default:** empty

### Dependency

This column is enabled by setting **Parameter configuration** to **Use parameter table**.

**See Also**

“Use Parameter Table” on page 5-7

**Min**

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

For parameters of type `Simulink.Parameter` with a specified minimum value, the **Min** column contains the specified minimum value for the parameter.

**Settings**

**Default:** empty

**Dependency**

This column is enabled by setting **Parameter configuration** to **Use parameter table**.

**See Also**

- “Use Parameter Table” on page 5-7
- `Simulink.Parameter`

**Max**

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.

For parameters of type `Simulink.Parameter` with a specified maximum value, the **Max** column contains the specified maximum value for the parameter.

**Settings**

**Default:** empty

**Dependency**

This column is enabled by setting **Parameter configuration** to **Use parameter table**.

**See Also**

- “Use Parameter Table” on page 5-7
- `Simulink.Parameter`

**Model Element**

In the Parameter Table, each row represents a parameter that can be constrained to specified values during Simulink Design Verifier analysis.



The **Model Element** column displays the path to the model elements where the parameter is used.

### Settings

**Default:** empty

### Dependency

This column is enabled by setting **Parameter configuration** to **Use parameter table**.

### See Also

“Use Parameter Table” on page 5-7

## Find parameters

The software searches your model for parameters that you can configure and loads them in the **Parameter Table**. If your model uses a configuration reference, Simulink Design Verifier does not support the search for parameters when using the **Find in Model** button. For more information, see “Share a Configuration with Multiple Models”.

### Dependency

This button is enabled by setting **Parameter configuration** to **Use parameter table**.

## Import

Adds parameters to the **Parameter Table** from a list stored in a file.

### Dependency

This button is enabled by setting **Parameter configuration** to **Use parameter table**.

## Export

Exports the current parameters in the **Parameter Table** to a file.

### Dependency

This button is enabled by setting **Parameter configuration** to **Use parameter table**.

## 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 setting **Parameter configuration** to **Use parameter table**.

### Command-Line Information

**Parameter:** DVParametersConfigFileName

**Type:** character array

**Value:** any valid MATLAB file

**Default:** 'sldv\_params\_template.m'

### See Also

“Use Parameter Table” on page 5-7

## Browse...

Browse to the parameter configuration file.

### Dependency

This button is enabled by **Enable parameter configuration**. This button is disabled by **Use parameter table**.

## Edit...

Edit the current parameter configuration file.

### Dependency

This button is enabled by **Enable parameter configuration**. This button is disabled by **Use parameter table**.

## Analyze all Startup Variants

Specify to analyze models that contain variant blocks where the **Variant activation time** parameter is startup.

### Settings

**Default:** On

On

Simulink Design Verifier analyze models that contain variant blocks with the **Variant activation time** parameter set to startup.

Off

Simulink Design Verifier analyzes only active variant blocks with **Variant activation time** parameter set to `startup`.

**Command-Line Information**

**Parameter:** DVAnalyzeAllStartupVariants

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

**See Also**

“Variant Activation Time for Variant Blocks”

**Launch Variant Manager...**

Launch the Variant Manager to view or define constraints on variant control parameters. Simulink Design Verifier applies these constraints during the analysis.

**See Also**

- “Variant Manager for Simulink”
- “Parameter Configuration for Analysis” on page 5-2
- “Verify and Validate Variant Models with Startup Activation Time”

## Design Verifier Pane: Test Generation

Test Generation

Test generation target:

Model coverage objectives:

Test conditions:

Test objectives:

Maximum test case steps:

Test suite optimization:

Relational Boundary Objectives

Include relational boundary objectives

Floating point absolute tolerance:  Floating point relative tolerance:

▼ Advanced parameters

Enhanced MCDC

Use strict propagation conditions

Add tests for the missing coverage

Extend using existing coverage data

Coverage data:

Extend using existing test data

Test data:

Separate objectives satisfied with the existing tests/coverage data in the report

### In this section...

- “Test Generation Pane Overview” on page 15-31
- “Test generation target” on page 15-31
- “Model coverage objectives” on page 15-31
- “Test conditions” on page 15-32
- “Test objectives” on page 15-33
- “Maximum test case steps” on page 15-33
- “Test suite optimization” on page 15-34

**In this section...**

“Include relational boundary objectives” on page 15-35

“Floating point absolute tolerance” on page 15-36

“Floating point relative tolerance” on page 15-36

“Use strict propagation conditions” on page 15-37

“Extend using existing coverage data” on page 15-38

“Coverage data” on page 15-38

“Browse” on page 15-39

“Extend using existing test data” on page 15-39

“Test data” on page 15-39

“Browse” on page 15-40

“Separate objectives satisfied with the existing tests/coverage data in the report” on page 15-40

**Test Generation Pane Overview**

Specify options that control how Simulink Design Verifier generates tests for the models it analyzes.

**See Also**

- “Workflow for Test Case Generation” on page 7-5

**Test generation target**

Specify the target for test generation.

- **Default:** Model generates test cases for the model.
- Code Generated as Top Model generates the code for the target as top model followed by test cases generation using the generated code.
- Code Generated as Model Reference generates the code for target as model reference followed by test cases generation using the generated code.

**Command-Line Information**

**Parameter:** DVTestgenTarget

**Type:** character array

**Value:** 'Model' | 'GenCodeTopModel' | 'GenCodeModelRef' |

**See Also**

- “Code Coverage Test Generation” on page 7-111
- “Generate Test Cases for Embedded Coder Generated Code” on page 7-28

**Model coverage objectives**

Specify the type of model coverage that Simulink Design Verifier attempts to achieve.

## Settings

**Default:** Condition Decision

None

Generates test cases that achieve only the custom objectives that you specified in your model using, for example, Test Objective blocks.

Decision

Generates test cases that achieve decision coverage. For more information, see “Decision” on page 7-30.

Condition Decision

Generates test cases that achieve condition and decision coverage. For more information, see “Condition” on page 7-30.

MCDC

Generates test cases that achieve modified condition decision coverage (MCDC). When you select MCDC, Simulink Design Verifier automatically enables every coverage objective for decision and condition coverage. For more information, see “MCDC” on page 7-31.

Enhanced MCDC

Generates test cases that achieve enhanced MCDC coverage. When you select Enhanced MCDC, Simulink Design Verifier automatically enables MCDC coverage. For more information, see “Enhanced MCDC” on page 7-31.

## Command-Line Information

**Parameter:** DVModelCoverageObjectives

**Type:** character array

**Value:** 'None' | 'Decision' | 'ConditionDecision' | 'MCDC' | 'EnhancedMCDC'

**Default:** 'ConditionDecision'

## See Also

- “What Is Test Case Generation?” on page 7-3
- “Workflow for Test Case Generation” on page 7-5

## Test conditions

Specify whether Test Condition blocks in your model are enabled or disabled.

### Settings

**Default:** Use local settings

Use local settings

Enables or disables Test Condition blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

Enable all

Enables all Test Condition blocks in the model regardless of the settings of their **Enable** parameters.

## Disable all

Disables all Test Condition blocks in the model regardless of the settings of their **Enable** parameters.

### Command-Line Information

**Parameter:** DVTestConditions

**Type:** character array

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Test Condition
- “Workflow for Test Case Generation” on page 7-5

## Test objectives

Specify whether Test Objective blocks in your model are enabled or disabled.

### Settings

**Default:** Use local settings

#### Use local settings

Enables or disables Test Objective blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

#### Enable all

Enables all Test Objective blocks in the model regardless of the settings of their **Enable** parameters.

#### Disable all

Disables all Test Objective blocks in the model regardless of the settings of their **Enable** parameters.

### Command-Line Information

**Parameter:** DVTestObjectives

**Type:** character array

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Test Objective
- “Workflow for Test Case Generation” on page 7-5

## Maximum test case steps

Specify the maximum number of simulation steps Simulink Design Verifier takes when attempting to satisfy a test objective.

The analysis uses the **Maximum test case steps** parameter during certain parts of the test-generation analysis to bound the number of steps that test generation uses. When you set a small value for this parameter, the parts of the analysis that are bounded complete in less time. When you set a larger value, the bounded parts of the analysis take longer, but it is possible for these parts of the analysis to generate longer test cases.

To achieve the best performance, set the **Maximum test case steps** parameter to a value just large enough to bound the longest required test case, even if the test cases that are ultimately generated are longer than this value.

When you also specify LongTestcases for the **Test suite optimization** parameter, the analysis uses successive passes of test generation to extend a potential test case so that it satisfies more objectives. When this happens, the analysis applies the **Maximum test case steps** parameter to each individual iteration of test generation.

### Settings

**Default:** 10000

You can specify a value that represents the maximum number of simulation steps Simulink Design Verifier takes when attempting to satisfy a test objective.

### Command-Line Information

**Parameter:** DVMaxTestCaseSteps

**Type:** int32

**Value:** any valid value

**Default:** 10000

### See Also

- “Workflow for Test Case Generation” on page 7-5

## Test suite optimization

Specify the optimization strategy to use when generating test cases.

### Settings

**Default:** Auto

#### Auto

Analyzes the model by using a strategy that automatically adapts to the model for better analysis performance and precision.

#### IndividualObjectives

Maximizes the number of test cases in a suite by generating cases that each address only one test objective. Each test case tends to be short, that is, it includes only a few time steps.

#### LongTestcases

Combines test cases to create a smaller number of test cases. This strategy generates fewer, but longer, test cases that each satisfy multiple test objectives.

#### Legacy LargeModel (Nonlinear Extended)

Analyzes the model by using a static strategy that does not adapt to the model. When you analyze a model by using Legacy LargeModel (Nonlinear Extended), Simulink Design Verifier



displays a warning message that this option is deprecated and suggests that you use Auto. Auto is most likely to produce better analysis results than Legacy LargeModel (Nonlinear Extended).

### Command-Line Information

**Parameter:** DVTestSuiteOptimization

**Type:** character array

**Value:** 'Auto' | 'IndividualObjectives' | 'LongTestcases' | Legacy LargeModel (Nonlinear Extended)

**Default:** 'Auto'

### See Also

- “Workflow for Test Case Generation” on page 7-5
- Simulink Design Verifier Options on page 15-2

## Include relational boundary objectives

Specify generation of test cases that satisfy relational boundary objectives. The objective applies to blocks such as Relational Operator that have an explicit or implicit relational operation. The tests check the relational operations in these blocks with:

- Equal operand values for integer and fixed-point operands.
- Operand values within a certain tolerance for all operands. For integer and fixed-point operands, the tolerance is fixed. For floating-point operands, the tolerance is computed using the inputs and a tolerance value that you specify. If you do not specify a tolerance value, the default values are used.

### Settings

**Default:** Off

On

For supported blocks, generates the test cases to satisfy relational boundary objectives.

Off

Ignores the relational boundary objectives for generating the test cases.

### Dependencies

If you select this option, you can use default values or specify values for:

- “Floating point absolute tolerance” on page 15-36
- “Floating point relative tolerance” on page 15-36

### Command-Line Information

**Parameter:** DVIncludeRelationalBoundary

**Type:** character array

**Value:** 'on' || 'off'

**Default:** 'off'

**See Also**

- “Relational Boundary” on page 7-31
- “Model Objects That Receive Coverage” (Simulink Coverage)
- “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” on page 3-7

**Floating point absolute tolerance**

Specify a value for absolute tolerance used in relational boundary tests. The relational boundary objectives apply to blocks such as Relational Operator that have an explicit or implicit relational operation. The tolerance value applies only if the relational operations in those blocks use floating point operands.

- For integer operands, the tolerance value is fixed at 1.
- For fixed-point operands, the tolerance value is the least significant bit.

**Settings**

**Default:** 1.0000e-05

For supported blocks, the relational boundary tests check the relational operations in the block with operand values that differ by a certain tolerance. The software calculates the tolerance value using the following formula

$\max(\text{absTol}, \text{relTol} * \max(|\text{lhs}|, |\text{rhs}|))$ , where:

- `absTol` is the absolute tolerance value that you specify.
- `relTol` is a relative tolerance value that you can specify.
- `lhs` is the left operand and `rhs` the right operand.
- `max(x, y)` returns `x` or `y`, whichever is greater.

**Dependencies**

To enter a value for this option, select “Include relational boundary objectives” on page 15-35.

**Command-Line Information**

**Parameter:** DVAbsoluteTolerance

**Type:** double

**Value:** Any valid value

**Default:** 1.0000e-05

**See Also**

- “Relational Boundary” on page 7-31
- “Model Objects That Receive Coverage” (Simulink Coverage)

**Floating point relative tolerance**

Specify a value for relative tolerance used in relational boundary tests. The relational boundary objectives apply to blocks such as Relational Operator that have an explicit or implicit relational

operation. The tolerance value applies only if the relational operations in those blocks use floating point operands.

- For integer operands, the tolerance value is fixed at 1.
- For fixed-point operands, the tolerance value is the least significant bit.

### Settings

**Default:** 0.01

For supported blocks, the relational boundary tests check the relational operations in the block with operand values that differ by a certain tolerance. The software calculates the tolerance value using the following formula

$\max(\text{absTol}, \text{relTol} * \max(|\text{lhs}|, |\text{rhs}|))$ , where:

- `absTol` is an absolute tolerance value that you can specify.
- `relTol` is the relative tolerance value that you specify.
- `lhs` is the left operand and `rhs` the right operand.
- `max(x, y)` returns `x` or `y`, whichever is greater.

### Dependencies

To enter a value for this option, select “Include relational boundary objectives” on page 15-35.

### Command-Line Information

**Parameter:** DVRelativeTolerance

**Type:** double

**Value:** Any valid value

**Default:** 0.01

### See Also

- “Relational Boundary” on page 7-31
- “Model Objects That Receive Coverage” (Simulink Coverage)

## Use strict propagation conditions

Specify whether to use strict propagation conditions for enhanced MCDC analysis.

### Settings

**Default:** Off

On

Use strict propagation condition for enhanced MCDC analysis.

Off

Does not use strict propagation conditions for enhanced MCDC analysis.

## Dependency

This parameter is enabled when you select Enhanced MCDC as **Model coverage objectives**.

### Command-Line Information

**Parameter:** DVStrictEnhancedMCDC

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

- “Enhanced MCDC” on page 7-31

## Extend using existing coverage data

Specify whether to use your existing coverage data for test generation. Simulink Design Verifier generates test cases for the objectives not covered in your existing coverage data.

### Settings

**Default:** Off

On

Extend the coverage in **Coverage data** by generating additional test cases.

Off

Analysis ignores existing **Coverage data**.

### Command-Line Information

**Parameter:** DVIgnoreCovSatisfied

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

## Coverage data

Specify the folder and file name for a file that contains data about satisfied coverage objectives.

### Settings

**Default:** ''

- Specify the folder and file name for a file that contains the satisfied coverage objectives data.
- Click **Browse** to navigate to and select an existing file.

### Command-Line Information

**Parameter:** DVCoverageDataFile

**Type:** character array

**Value:** any valid path and file name

**Default:** ''

## Browse

Browse to the coverage file that contains the data about satisfied coverage objectives.

## Dependencies

To enable this parameter, select **Extend using existing coverage data**.

## See Also

- “Achieve Missing Coverage in Closed-Loop Simulation Model” on page 9-11
- “Generate Tests”

## Extend using existing test data

Specify whether to extend the set of generated test cases in Simulink Design Verifier by importing previously generated test cases, test cases logged from a harness model, or a closed-loop simulation model.

## Settings

**Default:** Off

On

Use test cases specified in **Test data** to extend the set of generated test cases.

Off

Analysis ignores the existing **Test data**.

## Command-Line Information

**Parameter:** DVExtendExistingTests

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

## Test data

Specify a folder and file name for the MAT-file that contains the generated or 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 **Browse** to navigate to and select an existing file.

**Command-Line Information****Parameter:** DVExistingTestFile**Type:** character array**Value:** any valid path and file name**Default:** ' '**Browse**

Browse to the MAT-file that contains the generated or logged test case data and data about satisfied coverage objectives.

**Dependencies**

To enable this parameter, select **Extend using existing test data**.

**See Also**

- “When to Extend Existing Test Cases” on page 8-2
- “Common Workflow for Extending Existing Test Cases” on page 8-2

**Separate objectives satisfied with the existing tests/coverage data in the report**

Specify whether to separate the test objective statuses that are satisfied by the existing tests or coverage data from the extended coverage and test data in the analysis report.

**Settings****Default:** On On

Generates an analysis report where the existing tests and coverage data are separate from the extended test and coverage data.

 Off

Generates a report that combines existing and extended coverage and test data.

**Command-Line Information****Parameter:** DVIgnoreExistTestSatisfied**Type:** character array**Value:** 'on' | 'off'**Default:** 'on'**See Also**

- “Extend Test Cases for Closed-Loop System” on page 8-10
- “Manage Simulink Design Verifier Data Files” on page 13-7

## **See Also**

### **More About**

- “Design Verifier Pane” on page 15-9
- “Generate Test Cases for Model Decision Coverage” on page 7-6
- “Workflow for Test Case Generation” on page 7-5

## Design Verifier Pane: Design Error Detection

Design Error Detection

**Modeling Errors**

- Dead logic (partial)
  - Run exhaustive analysis
- Out of bound array access
- Data store access violations

**Numerical Errors**

- Division by zero
- Integer overflow
- Non-finite and NaN floating-point values
- Subnormal floating-point values

**Signal Range Errors**

- Specified minimum and maximum value violations
- Specified block input range violations

**High-Integrity Systems Modeling Checks**

- Usage of remainder and reciprocal operations - hisl\_0002
- Usage of square root operations - hisl\_0003
- Usage of log and log10 operations - hisl\_0004
- Usage of Reciprocal Square Root blocks - hisl\_0028

### In this section...

- “Design Error Detection Pane Overview” on page 15-43
- “Dead logic (partial)” on page 15-43
- “Run exhaustive analysis” on page 15-43
- “Coverage objectives to be analyzed” on page 15-44
- “Out of bound array access” on page 15-45
- “Data store access violations” on page 15-45
- “Division by zero” on page 15-46
- “Integer overflow” on page 15-46
- “Non-finite and NaN floating-point values” on page 15-47
- “Subnormal floating-point values” on page 15-47



**In this section...**

“Specified minimum and maximum value violations” on page 15-48

“Specified block input range violations” on page 15-48

“Usage of rem and reciprocal operations - hisl\_0002” on page 15-49

“Usage of Square Root operations - hisl\_0003” on page 15-50

“Usage of log and log10 operations - hisl\_0004” on page 15-50

“Usage of Reciprocal Square Roots blocks - hisl\_0028” on page 15-51

## Design Error Detection Pane Overview

Specify options that control how Simulink Design Verifier detects runtime errors in the models it analyzes.

### Dead logic (partial)

Specify whether to analyze your model for dead logic. This may result in a partial analysis. Select **Run exhaustive analysis** to always run an exhaustive analysis.

#### Settings

**Default:** Off

On

Reports dead logic identified in your model.

Off

Does not analyze for dead logic.

#### Command-Line Information

**Parameter:** DVDetectDeadLogic

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

#### See Also

- “Dead Logic Detection” on page 6-7

### Run exhaustive analysis

Specify whether to run an exhaustive analysis for dead logic in the model.

#### Settings

**Default:** Off

On

Perform an exhaustive analysis for dead logic in your model.

 Off

Does not perform exhaustive analysis for dead logic in your model.

**Command-Line Information****Parameter:** DVDetectActiveLogic**Type:** character array**Value:** 'on' | 'off'**Default:** 'off'**Dependency**

To enable this parameter, select **Dead logic (partial)**.

**See Also**

“Dead Logic Detection” on page 6-7

**Coverage objectives to be analyzed**

Specify the coverage objectives to analyze for dead logic in the model.

**Settings****Default:** 'ConditionDecision'

Decision

Analyze decision coverage objectives for dead logic.

Condition Decision

Analyze condition and decision coverage objectives for dead logic.

MCDC

Analyze modified condition decision coverage (MCDC) objectives for dead logic.

**Command-Line Information****Parameter:** DVDeadLogicObjectives**Type:** character array**Value:** 'Decision' | 'ConditionDecision' | 'MCDC'**Default:** 'ConditionDecision'**Dependency**

This parameter is dependent upon **Dead logic (partial)** and works only when **Dead logic (partial)** is also enabled.

**See Also**

“Dead Logic Detection” on page 6-7

## Out of bound array access

Specify whether to analyze your model for out of bound array access errors.

### Settings

**Default:** On

On

Reports out of bound array access errors in your model.

Off

Does not report out of bound array access errors in your model.

### Command-Line Information

**Parameter:** DVDetectOutOfBounds

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

### See Also

“Detect Out of Bound Array Access Errors” on page 6-28

## Data store access violations

Specify whether to analyze your model for data store access violations. Design error detection checks for these violations related to Data Store Memory blocks:

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

### Settings

**Default:** Off

On

Reports data store access violations in your model.

Off

Does not report data store access violations in your model.

### Command-Line Information

**Parameter:** DVDetectDSMAccessViolations

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

**See Also**

“Detecting Access Order Errors”

**Division by zero**

Specify whether to analyze your model for division-by-zero errors.

**Settings**

**Default:** On



On

Reports division-by-zero errors in your model.



Off

Does not report division-by-zero errors in your model.

**Command-Line Information**

**Parameter:** DVDetectDivisionByZero

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

**See Also**

“Detect Integer Overflow and Division-by-Zero Errors” on page 6-19

**Integer overflow**

Specify whether to analyze your model for integer and fixed-point data overflow errors.

**Settings**

**Default:** On



On

Reports integer or fixed-point data overflow errors in your model.



Off

Does not report integer or fixed-point data overflow errors in your model.

**Command-Line Information**

**Parameter:** DVDetectIntegerOverflow

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

## See Also

“Detect Integer Overflow and Division-by-Zero Errors” on page 6-19

## Non-finite and NaN floating-point values

Specify whether to analyze your model for non-finite and NaN floating-point values.

### Settings

**Default:** Off



On

Reports non-finite and NaN floating-point values in your model.



Off

Does not report non-finite and NaN floating-point values in your model.

### Command-Line Information

**Parameter:** DVDetectInfNaN

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

“Detect Non-Finite, NaN, and Subnormal Floating-Point Values” on page 6-33

## Subnormal floating-point values

Specify whether to analyze your model for subnormal floating-point values.

### Settings

**Default:** Off



On

Reports subnormal floating-point values in your model.



Off

Does not report subnormal floating-point values in your model.

### Command-Line Information

**Parameter:** DVDetectSubnormal

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

**See Also**

“Detect Non-Finite, NaN, and Subnormal Floating-Point Values” on page 6-33

**Specified minimum and maximum value violations**

Specify whether to check that the intermediate and output signals in your model are within the range of user-specified minimum and maximum constraints.

**Settings**

**Default:** Off



On

Checks that intermediate and output signals are within the range of user-specified minimum and maximum constraints.



Off

Does not check that intermediate and output signals are within the range of user-specified minimum and maximum constraints.

**Command-Line Information**

**Parameter:** DVDesignMinMaxCheck

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

**See Also**

“Check for Specified Minimum and Maximum Value Violations” on page 6-23

**Specified block input range violations**

Specify whether to analyze your model for block input range violations. The check detects input range violations for blocks with these settings:

- For these blocks, when the **Diagnostic for out-of-range input** parameter is set to Warning or Error:
  - n-D Lookup Table
  - Interpolation Using Prelookup
  - Prelookup
  - Direct Lookup Table (n-D)
- Multiport Switch blocks, when the **Diagnostic for default case** parameter is set to Warning or Error.
- Trigonometric Function blocks, when the **Approximation method** parameter is set to CORDIC

---

**Note** The check does not flag block input range violations for n-D Lookup Table blocks, when the **Interpolation method** is set to Akima spline or Cubic spline.

---

**Note** The check does not flag block input range violations for Trigonometric Function blocks with CORDIC **Approximation method**, for which the **Function** parameter is atan2 and the data types of the input signals are double.

---

## Settings

**Default:** Off



On

Reports block input range violations in your model.



Off

Does not report block input range violations in your model.

## Command-Line Information

**Parameter:** DVDetectBlockInputRangeViolations

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

“Detect Block Input Range Violations”

## Usage of rem and reciprocal operations - hisl\_0002

Specify whether to check the usage of rem and reciprocal operations that cause non-finite results.

This corresponds to the hisl\_0002 check for High-Integrity Systems Modeling. For more information, see hisl\_0002: Usage of Math Function blocks (rem and reciprocal).

## Settings

**Default:** Off



On

Reports violations of the hisl\_0002 check in your model.



Off

Does not report violations of the hisl\_0002 check in your model.

## Command-Line Information

**Parameter:** DVDetectHISMViolationsHisl\_0002

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

**See Also**

“Model Advisor Checks for High-Integrity Systems Modeling Guidelines”

Math Function

**Usage of Square Root operations - hisl\_0003**

Specify whether to check the usage of Square Root operations with inputs that can be negative.

This corresponds to the hisl\_0003 check for High-Integrity Systems Modeling. For more information, see hisl\_0003: Usage of Square Root blocks.

**Settings**

**Default:** Off

On  
Report violations of the hisl\_0003 check in your model.

Off  
Does not report violations of the hisl\_0003 check in your model.

**Command-Line Information**

**Parameter:** DVDetectHISMViolationsHisl\_0003

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

**See Also**

“Model Advisor Checks for High-Integrity Systems Modeling Guidelines”

Sqrt

**Usage of log and log10 operations - hisl\_0004**

Specify whether to check the usage of log and log10 operations that cause non-finite results.

This corresponds to the hisl\_0004 check for High-Integrity Systems Modeling. For more information, see hisl\_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm).

**Settings**

**Default:** Off

On  
Report violations of the hisl\_0004 check in your model.

Off  
Does not report violations of the hisl\_0004 check in your model.



**Command-Line Information****Parameter:** DVDetectHISMViolationsHisl\_0004**Type:** character array**Value:** 'on' | 'off'**Default:** 'off'**See Also**

“Model Advisor Checks for High-Integrity Systems Modeling Guidelines”

**Usage of Reciprocal Square Roots blocks - hisl\_0028**

Specify whether to check the usage of Reciprocal Square Root blocks with inputs that can go zero or negative.

This corresponds to the hisl\_0028 check for High Integrity Systems Modeling. For more information, see hisl\_0028: Usage of Reciprocal Square Root blocks.

**Settings****Default:** Off

On

Report violations of the hisl\_0028 check in your model.



Off

Does not report violations of the hisl\_0028 check in your model.

**Command-Line Information****Parameter:** DVDetectHISMViolationsHisl\_0028**Type:** character array**Value:** 'on' | 'off'**Default:** 'off'**See Also**

“Model Advisor Checks for High-Integrity Systems Modeling Guidelines”

## Design Verifier Pane: Property Proving

| Property Proving         |  |
|--------------------------|--|
| Assertion blocks:        | Enable all <input type="button" value="v"/>    |
| Proof assumptions:       | Enable all <input type="button" value="v"/>    |
| Strategy:                | FindViolation <input type="button" value="v"/> |
| Maximum violation steps: | 20   |

### In this section...

“Property Proving Pane Overview” on page 15-52

“Assertion blocks” on page 15-52

“Proof assumptions” on page 15-53

“Strategy” on page 15-53

“Maximum violation steps” on page 15-54

## Property Proving Pane Overview

Specify options that control how Simulink Design Verifier proves properties for the models it analyzes.

### See Also

- “What Is Property Proving?” on page 12-2
- “Workflow for Proving Model Properties” on page 12-4
- “Prove Properties in a Model” on page 12-5

## Assertion blocks

Specify whether Assertion blocks in your model are enabled or disabled.

### Settings

**Default:** Use local settings

Use local settings

Enables or disables Assertion blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

Enable all

Enables all Assertion blocks in the model regardless of the settings of their **Enable** parameters.

Disable all

Disables all Assertion blocks in the model regardless of the settings of their **Enable** parameters.

### Command-Line Information

**Parameter:** DVAssertions

**Type:** character array

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Assertion
- “Workflow for Proving Model Properties” on page 12-4
- “Prove Properties in a Model” on page 12-5

## Proof assumptions

Specify whether Proof Assumption blocks in your model are enabled or disabled.

### Settings

**Default:** Use local settings

Use local settings

Enables or disables Proof Assumption blocks based on the value of the **Enable** parameter of each block. If a block's **Enable** parameter is selected, the block is enabled; otherwise, the block is disabled.

Enable all

Enables all Proof Assumption blocks in the model regardless of the settings of their **Enable** parameters.

Disable all

Disables all Proof Assumption blocks in the model regardless of the settings of their **Enable** parameters.

### Command-Line Information

**Parameter:** DVProofAssumptions

**Type:** character array

**Value:** 'UseLocalSettings' | 'EnableAll' | 'DisableAll'

**Default:** 'UseLocalSettings'

### See Also

- Proof Assumption
- “Workflow for Proving Model Properties” on page 12-4
- “Prove Properties in a Model” on page 12-5

## Strategy

Specify the strategy that Simulink Design Verifier uses when proving properties.

## Settings

**Default:** Prove

Prove

Performs property proofs.

FindViolation

Searches only for property violations within the number of simulation steps specified by the **Maximum violation steps** option.

ProveWithViolationDetection

Searches both for property violations, as well as tries to prove properties for which it failed to detect a violation. This strategy is a relatively optimal balance between the ProveWithViolationDetection and FindViolation strategies.

## Dependency

Selecting FindViolation or ProveWithViolationDetection enables the **Maximum violation steps** parameter.

## Command-Line Information

**Parameter:** DVProvingStrategy

**Type:** character array

**Value:** 'Prove' | 'FindViolation' | 'ProveWithViolationDetection'

**Default:** 'Prove'

## See Also

- “What Is Property Proving?” on page 12-2
- “Workflow for Proving Model Properties” on page 12-4
- “Prove Properties in a Model” on page 12-5

## Maximum violation steps

Specify the maximum number of simulation steps over which Simulink Design Verifier searches for property violations.

## Settings

**Default:** 20

The Simulink Design Verifier software does not search beyond the maximum number of simulation steps that you specify. Therefore, it cannot identify violations that might occur later in a simulation.

## Dependency

This parameter is enabled when you set **Strategy** to FindViolation or ProveWithViolationDetection.

## Command-Line Information

**Parameter:** DVMaxViolationSteps

**Type:** int32

**Value:** any valid value

**Default:** 20

**See Also**

- “What Is Property Proving?” on page 12-2
- “Workflow for Proving Model Properties” on page 12-4
- “Prove Properties in a Model” on page 12-5

## Design Verifier Pane: Results

Data file options

Data file name:

Include expected output values

Randomize data that do not affect the outcome

Harness model options

Generate separate harness model after analysis

Harness model file name:

Reference input model in generated harness

Harness source:

Simulink Test options

Test File name:

Test Harness name:

OK Cancel Help Apply

### In this section...

- “Results Pane Overview” on page 15-56
- “Data file name” on page 15-57
- “Include expected output values” on page 15-57
- “Randomize data that do not affect the outcome” on page 15-58
- “Generate separate harness model after analysis” on page 15-59
- “Harness model file name” on page 15-59
- “Reference input model in generated harness” on page 15-60
- “Harness source” on page 15-61
- “Test File Name” on page 15-61
- “Test Harness Name” on page 15-62

## Results Pane Overview

Specify options that control how Simulink Design Verifier handles the results that it generates.

### See Also

“Review Analysis Results”

## Data file name

Specify a folder and file name for the MAT-file that contains the data generated during the analysis, stored in an `sldvData` structure.

### Settings

**Default:** `$ModelName$_sldvdata`

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.
- Enter a file name for the MAT-file.
- `$ModelName$` is a token that represents the model name.

### Command-Line Information

**Parameter:** `DVDataFileName`

**Type:** character array

**Value:** any valid path and file name

**Default:** `'$ModelName$_sldvdata'`

### See Also

- “Manage Simulink Design Verifier Data Files” on page 13-7
- “Review Analysis Results”

## 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 Output 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 “Generate sldvData Structure” on page 13-7.
- When **Include expected output values** is enabled, Simulink Design Verifier successively simulates the model using each test case that it generates. Enabling this option requires more time for Simulink Design Verifier to complete its analysis.

**Command-Line Information****Parameter:** DVSaveExpectedOutput**Type:** character array**Value:** 'on' | 'off'**Default:** 'off'**See Also**

- “Manage Simulink Design Verifier Data Files” on page 13-7
- “Review Analysis Results”

**Randomize data that do not affect the outcome**

Specify whether to use random values instead of zeros for input signals that have no impact on test or proof objectives.

**Settings****Default:** Off On

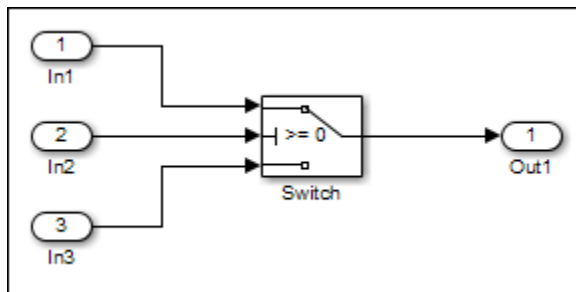
Assigns random values to test case or counterexample signals that do not affect the outcome of test or proof objectives in a model. This option can enhance traceability and improve your regression tests.

 Off

Assigns zeros to test case or counterexample signals that do not affect the outcome of test or proof objectives in a model.

**Tips**

- This option replaces default data values with random values when the Simulink Design Verifier internal analysis engine does not specify a value. When a value does not influence the satisfaction of a test or proof objective, the generated analysis report indicates that value with a dash (-).
- Simulink Design Verifier generated analysis reports show the setting of this option.
- Enable this option to enhance traceability when simulating test cases or counterexamples. For instance, consider the following model:



Only the signal entering the Switch block control port impacts its decision coverage. If the **Randomize data that does not affect outcome** parameter is off, Simulink Design Verifier uses



zeros to represent the signals from In1 and In3. When inspecting the results from test case or counterexample simulations, it is unclear which of these signals passes through the Switch block because they have the same value. But if the **Randomize data that does not affect outcome** parameter is on, the software uses unique values to represent each of those signals. In this case, it is easier to determine which signal passes through the Switch block.

### Command-Line Information

**Parameter:** DVRandomizeNoEffectData

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

- “Manage Simulink Design Verifier Data Files” on page 13-7
- “Review Analysis Results”

## Generate separate harness model after analysis

Create a harness model generated by the Simulink Design Verifier analysis.

### Settings

**Default:** Off

On

Saves the harness model that Simulink Design Verifier generates as a model file.

Off

Does not save the harness model that Simulink Design Verifier generates.

### Dependency

This parameter enables **Harness model file name**.

### Command-Line Information

**Parameter:** DVSaveHarnessModel

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

- “Manage Simulink Design Verifier Harness Models” on page 13-13
- “Review Analysis Results”

## Harness model file name

Specify a folder and file name for the harness model.

**Settings****Default:** \$modelName\$\_harness

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.
- Enter a file name for the harness model.
- \$modelName\$ is a token that represents the model name.

**Dependency**This parameter is enabled by **Generate separate harness model after analysis**.**Command-Line Information****Parameter:** DVHarnessModelFileName**Type:** character array**Value:** any valid path and file name**Default:** '\$modelName\$\_harness'**See Also**

- “Manage Simulink Design Verifier Harness Models” on page 13-13
- “Review Analysis Results”

**Reference input model in generated harness**

Use a Model block to reference the model to run in the harness model.

**Settings****Default:** On On

Uses a Model block to reference the model to run in the harness model.

 Off

Uses a copy of the model in the harness model.

**Tips**

- If the Test Unit in the harness model is a subsystem, the values of the Simulink simulation optimization parameters on the Configuration Parameters dialog box can affect the coverage results.

---

**Note** The simulation optimization parameters are on the following Configuration Parameters dialog box panes:

- **Optimization** pane
  - **Optimization > Signals and Parameters** pane
  - **Optimization > Stateflow** pane
-

- On the **Design Verifier > Parameters** pane, if you select the **Apply parameters** parameter, Simulink Design Verifier uses a subsystem that contains a copy of the original model in the harness model, even if you select **Reference input model in generated harness**.

### Command-Line Information

**Parameter:** DVModelReferenceHarness

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

### See Also

- “Manage Simulink Design Verifier Harness Models” on page 13-13
- “Review Analysis Results”

## Harness source

Specify the type of Inputs block for the harness model.

### Settings

**Default:** Signal Editor

Signal Editor

Generates a separate harness model with the Signal Editor block as the Inputs block.

Signal Builder

Generates a separate harness model with the Signal Builder block as the Inputs block.

### Dependency

This parameter is enabled by **Generate separate harness model after analysis**.

### Command-Line Information

**Parameter:** DVHarnessSource

**Type:** character array

**Value:** 'Signal Editor' | 'Signal Builder'

**Default:** 'Signal Editor'

### See Also

- “Manage Simulink Design Verifier Harness Models” on page 13-13

## Test File Name

Name and path for test file name in Simulink Test

### Settings

**Default:** \$modelName\$\_test

- Enter a file name for the test file containing Simulink Design Verifier results.
- `$modelName$` is a token that represents the model name.
- You can enter an absolute path, or a path relative to that specified by **Output folder** in the Design Verifier pane.

### Dependency

This parameter is visible and enabled if you have a Simulink Test license.

### Command-Line Information

**Parameter:** DVSLTestFileName

**Type:** character array

**Value:** any valid path and file name

**Default:** '\$modelName\$\_test'

### See Also

- “Increase Coverage by Generating Test Inputs” (Simulink Test)

## Test Harness Name

Name of the test harness in Simulink Test

### Settings

**Default:** `$modelName$_sldvharness`

- Enter a valid name for the test harness built to simulate Simulink Design Verifier test cases. The test harness corresponds to the test file specified by the parameter **Test File name**.
- The `$modelName$` token represents the model name.
- Enter a valid MATLAB identifier for the test harness name.

### Dependency

This parameter is visible and enabled with a Simulink Test license.

### Command-Line Information

**Parameter:** DVSLTestHarnessName

**Type:** character array

**Value:** any valid file name

**Default:** '\$modelName\$\_sldvharness'

### See Also

- “Increase Coverage by Generating Test Inputs” (Simulink Test)

## Design Verifier Pane: Report

Report

Generate report of the results

Generate additional report in PDF format

Report file name:

Include screen shots of properties

Display report

### In this section...

“Report Pane Overview” on page 15-63

“Generate report of the results” on page 15-63

“Generate additional report in PDF format” on page 15-64

“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 Simulink Design Verifier reports its results.

### See Also

- “Review Results” on page 13-35
- “Review Analysis Results”

## Generate report of the results

Generate and save a Simulink Design Verifier report.

### Settings

**Default:** Off

- On  
Saves the HTML report that Simulink Design Verifier generates.
- Off  
Does not generate a Simulink Design Verifier report.

## Dependencies

This parameter enables the following parameters:

- **Generate additional report in PDF format**
- **Report file name**
- **Include screen shots of properties**
- **Display report**

## Command-Line Information

**Parameter:** DVSaveReport

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

- “Review Results” on page 13-35
- “Review Analysis Results”

## Generate additional report in PDF format

Save an additional PDF version of the Simulink Design Verifier report.

### Settings

**Default:** Off

On

Saves an additional PDF version of the Simulink Design Verifier report.

Off

Does not save an additional PDF version of the Simulink Design Verifier report.

### Dependency

This parameter is enabled by **Generate report of the results**.

## Command-Line Information

**Parameter:** DVReportPDFFormat

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

## See Also

- “Review Results” on page 13-35
- “Review Analysis Results”

## Report file name

Specify a folder and file name for the report that Simulink Design Verifier analysis generates.

### Settings

**Default:** \$modelName\$\_report

- Optionally, enter a path that is either absolute or relative to the path name specified in **Output folder**.
- Enter a file name for the report that the analysis generates.
- \$modelName\$ is a token that represents the model name.

### Dependency

This parameter is enabled by **Generate report of the results**.

### Command-Line Information

**Parameter:** DVReportFileName

**Type:** character array

**Value:** any valid path and file name

**Default:** '\$modelName\$\_report'

### See Also

- “Review Results” on page 13-35
- “Review Analysis Results”

## 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** in the Reports pane and **Generate separate harness model after analysis** in the Results pane.

### Command-Line Information

**Parameter:** DVReportIncludeGraphics

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'off'

**See Also**

- “Review Results” on page 13-35
- “Review Analysis Results”

## Display report

Display the report that the Simulink Design Verifier analysis generates after completing its analysis.

**Settings**

**Default:** On

On

Displays the report that the analysis generates after completing its analysis.

Off

Does not display the report that the analysis generates after completing its analysis.

**Dependency**

This parameter is enabled by **Generate report of the results**.

**Command-Line Information**

**Parameter:** DVDisplayReport

**Type:** character array

**Value:** 'on' | 'off'

**Default:** 'on'

**See Also**

- “Review Results” on page 13-35
- “Review Analysis Results”



# Verification and Validation

---

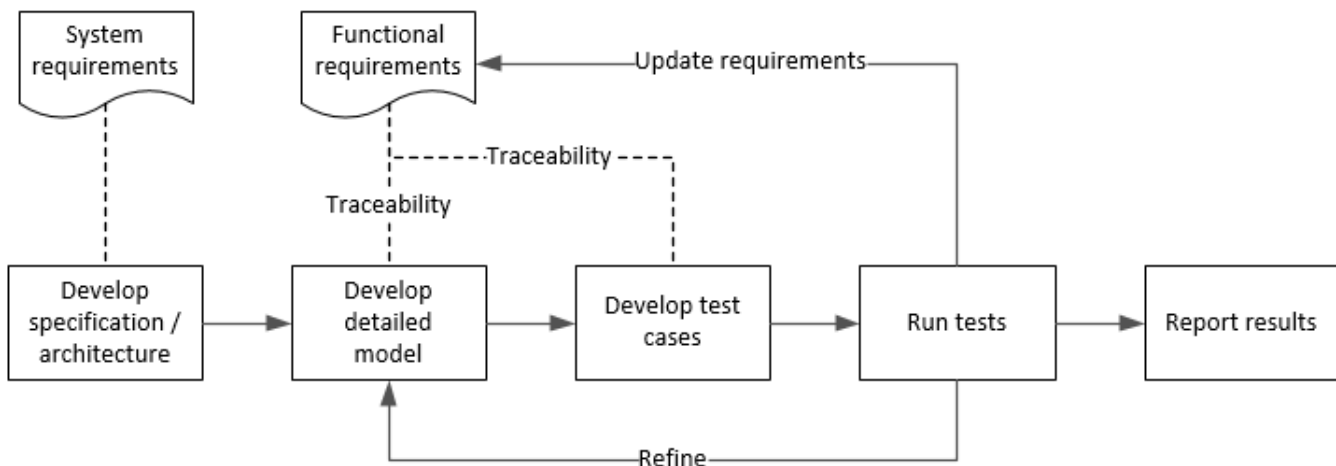
- “Test Model Against Requirements and Report Results” on page 16-2
- “Analyze Models for Standards Compliance and Design Errors” on page 16-7
- “Perform Functional Testing and Analyze Test Coverage” on page 16-9
- “Analyze Code and Test Software-in-the-Loop” on page 16-12
- “Create Back-to-Back Tests Using Enhanced MCDC” on page 16-20

## Test Model Against Requirements and Report Results

### Requirements - Test Traceability Overview

Traceability between requirements and test cases helps you interpret test results and see the extent to which your requirements are verified. You can link a requirement to elements that help verify it, such as test cases in the Test Manager, `verify` statements in a Test Sequence block, or Model Verification blocks in a model. When you run tests, a pass/fail summary appears in your requirements set.

This example demonstrates a common requirements-based testing workflow for a cruise control model. You start with a requirements set, a model, and a test case. You add traceability between the tests and the safety requirements. You run the test, summarize the verification status, and report the results.




In this example, you conduct a simple test of two requirements in the set:

- That the cruise control system transitions to disengaged from engaged when a braking event has occurred
- That the cruise control system transitions to disengaged from engaged when the current vehicle speed is outside the range of 20 mph to 90 mph.

### Display the Requirements

- 1 Open the example project.

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

- 2 In the `models` folder, open the `simulinkCruiseAddReqExample` model.
- 3 Display the requirements. Click the  icon in the lower-right corner of the model canvas, and select **Requirements**. The requirements appear below the model canvas.
- 4 Display the verification and implementation status. Right-click a requirement and select **Verification Status** and **Implementation Status**.

The screenshot displays the Simulink environment with a model diagram on the left, a Requirements table in the middle, and a Property Inspector on the right.

**Model Diagram:** Shows a block named "Compute target speed" with inputs: CruiseOnOff (boolean), Brake (boolean), Speed (single), CoastSetSw (boolean), and AccelResSw (boolean). It has outputs: engaged (boolean) and tspeed (single).

**Requirements Table:**

| Index                 | ID                      | Summary                      | Verified | Implemented |
|-----------------------|-------------------------|------------------------------|----------|-------------|
| simulinkCruiseChar... |                         |                              |          |             |
| 1                     | Architecture            | Architecture                 |          |             |
| 1.1                   | A 1.1                   | Enable Disable Switch        |          |             |
| 1.2                   | A 1.2                   | Set Speed / Decelerate Bu... |          |             |
| 1.3                   | A 1.3                   | Resume Speed / Accelerat...  |          |             |
| 1.4                   | A 1.4                   | Engaged Output               |          |             |
| 1.5                   | A 1.5                   | Target Speed Output          |          |             |
| 1.6                   | A 1.6                   | Vehicle Speed Input          |          |             |
| 1.7                   | A 1.7                   | Vehicle Brake Input          |          |             |
| 2                     | Functional Requirements | Functional Requirements      |          |             |
| 3                     | Safety Requirements     | Safety Requirements          |          |             |

**Property Inspector (Requirement: A 1.2):**

- Type: Functional
- Index: 1.2
- Custom ID: A 1.2
- Summary: Set Speed / Decelerate Button
- Description: The controller shall have an input button to: set the target speed to the current vehicle speed when the cruise control is **not engaged (active)** decelerate (reduce) the target speed when the cruise control is **engaged (active)**
- Keywords:
- Revision information:
- Links: Implemented by: CoastSetSw
- Comments:

- 5 In the Project window, open the Simulink Test file `s1ReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

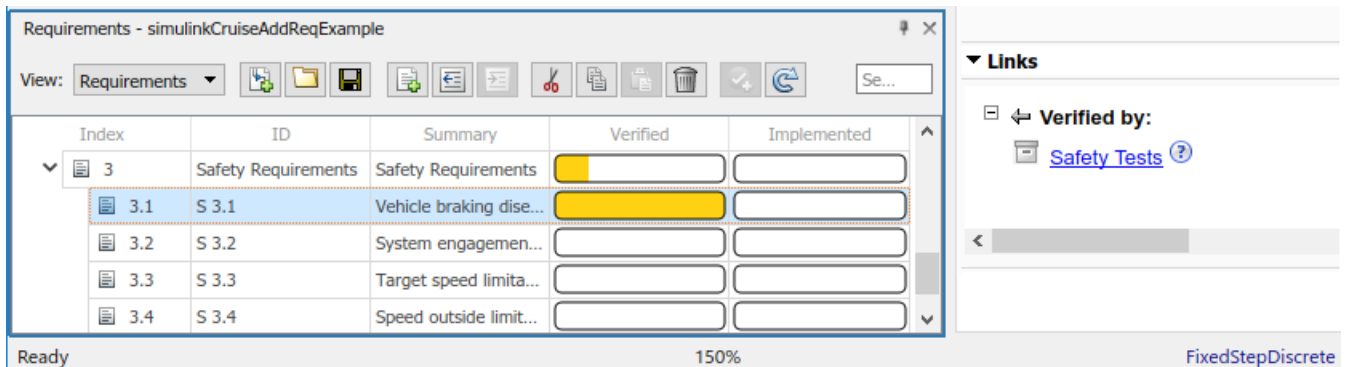
## Link Requirements to Tests

Link the requirements to the test case.

- 1 In the Project window, open the Simulink Test file `s1ReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager. Explore the test suite and select `Safety Tests`.

Return to the model. Right-click on requirement `S 3.1` and select **Link from Selected Test Case**.

A link to the `Safety Tests` test case is added to **Verified by**. The yellow bars in the **Verified** column indicate that the requirements are not verified.



- 2 Also add a link for item S 3.4.

## Run the Test

The test case uses a test harness `SafetyTest_Harness1`. In the test harness, a test sequence sets the input conditions and checks the model behavior:

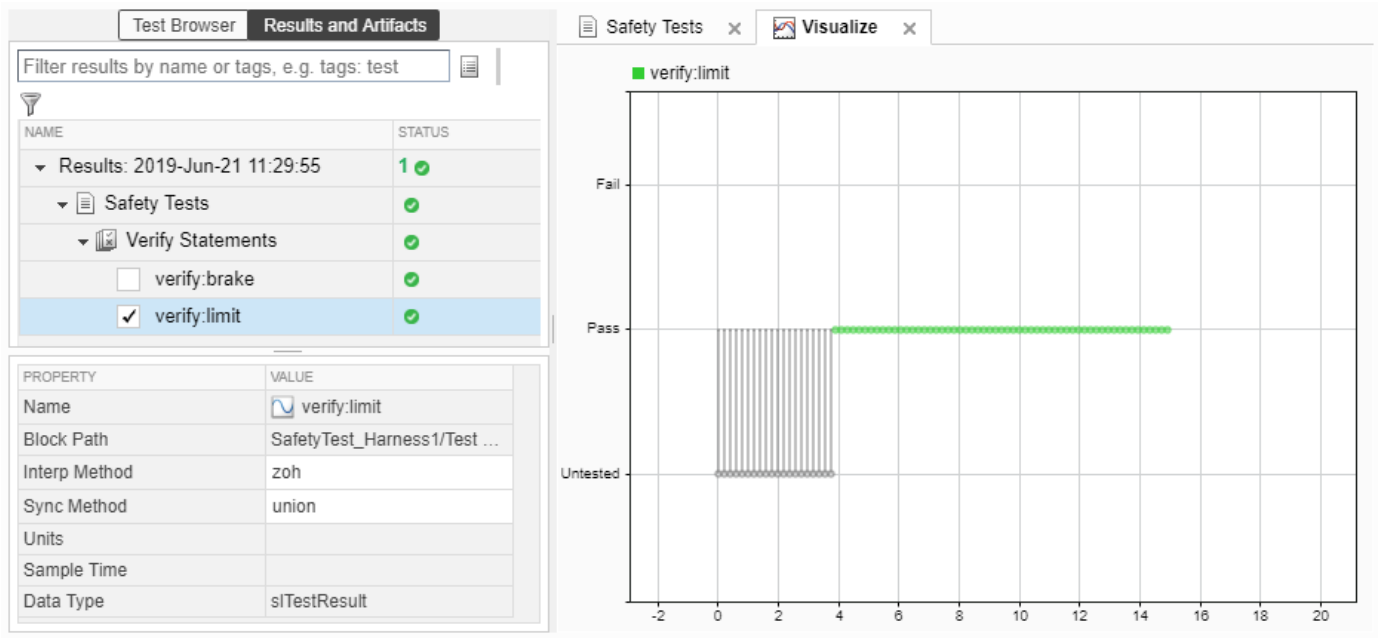
- The `BrakeTest` sequence engages the cruise control, then applies the brake. It includes the `verify` statement

```
verify(engaged == false,...
      'verify:brake',...
      'system must disengage when brake applied')
```

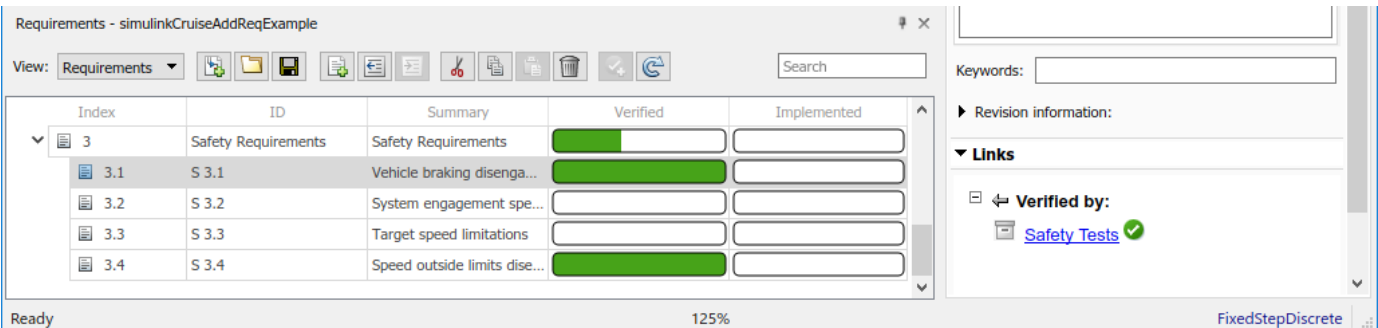
- The `LimitTest` sequence engages the cruise control, then ramps up the vehicle speed until it exceeds the upper limit. It includes the `verify` statement.

```
verify(engaged == false,...
      'verify:limit',...
      'system must disengage when limit exceeded')
```

- 1 Return to the Test Manager. To run the test case, click **Run**.
- 2 When the test finishes, review the results. The Test Manager shows that both assessments pass and the plot provides the detailed results of each `verify` statement.



- 3 Return to the model and refresh the Requirements. The green bar in the **Verified** column indicates that the requirement has been successfully verified.



## Report the Results

- 1 Create a report using a custom Microsoft Word template.
  - a From the Test Manager results, right-click the test case name. Select **Create Report**.
  - b In the Create Test Result Report dialog box, set the options:
    - Title — SafetyTest
    - Results for — All Tests
    - File Format — DOCX
    - For the other options, keep the default selections.
  - c Enter a file name and select a location for the report.
  - d For the **Template File**, select the ReportTemplate.dotx file in the **documents** project folder.
  - e Click **Create**.

- 2 Review the report.
  - a The **Test Case Requirements** section specifies the associated requirements
  - b The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

## See Also

### Related Examples

- “Link to Requirements” (Simulink Test)
- “Validate Requirements Links in a Model” (Requirements Toolbox)
- “Customize Requirements Traceability Report for Model” (Requirements Toolbox)

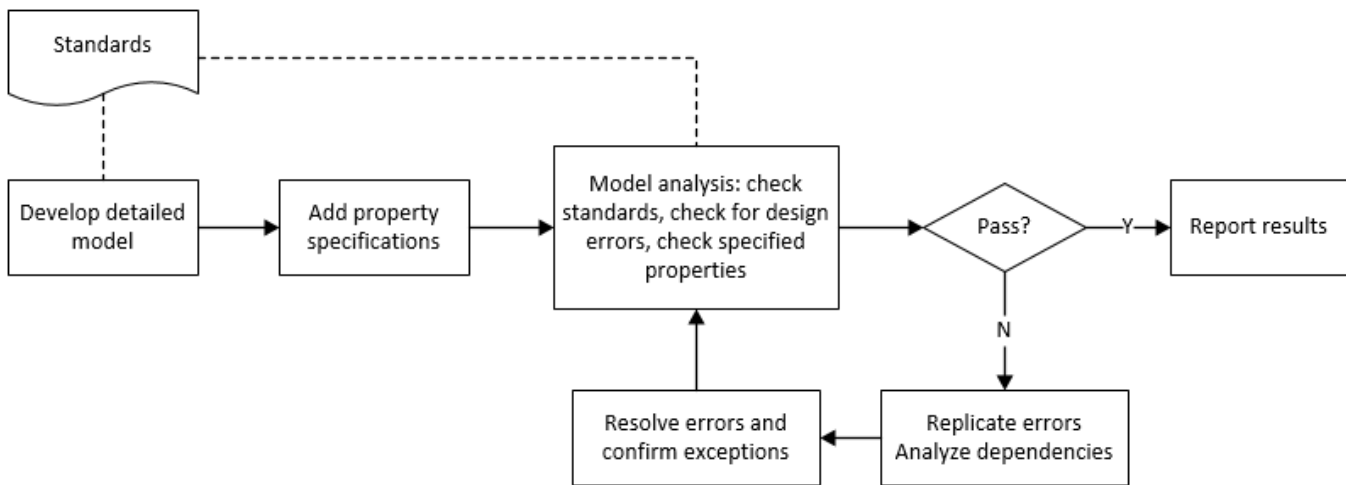
### External Websites

- Requirements-Based Testing Workflow

# Analyze Models for Standards Compliance and Design Errors

## Standards and Analysis Overview

During model development, check and analyze your model to increase confidence in its quality. Check your model against standards such as MAB style guidelines and high-integrity system design guidelines such as DO-178 and ISO 26262. Analyze your model for errors, dead logic, and conditions that violate required properties. Using the analysis results, update your model and document exceptions. Report the results using customizable templates.



## Check Model for Style Guideline Violations and Design Errors

This example shows how to use the Model Advisor to check a cruise control model for MathWorks® Advisory Board (MAB) style guideline violations and design errors. Select checks and run the analysis on the model. Iteratively debug issues using the Model Advisor and rerun checks to verify that it is in compliance. After passing your selected checks, report results.

### Check Model for MAB Style Guideline Violations

Check that your model complies with MAB guidelines by using the Model Advisor.

- 1 Open the example project. On the command line, enter
 

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```
- 2 Open the `simulinkCruiseErrorAndStandardsExample` model.
 

```
open_system simulinkCruiseErrorAndStandardsExample
```
- 3 In the **Modeling** tab, select **Model Advisor**.
- 4 Click **OK** to select `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
- 5 Check your model for MAB style guideline violations using Simulink Check.
  - a In the left pane, in the **By Product > Simulink Check > Modeling Standards > MAB Checks** folder, select:

- **Check Indexing Mode**
  - **Check model diagnostic parameters**
- b** Right-click on the **MAB Checks** node and select **Run Checks**.
  - c** To review the configuration parameter settings that violate MAB style guidelines, run the **Check model diagnostic parameters** check.
  - d** The analysis results appear in the right pane on the **Report** tab. Report displays the violation details and the recommended action.
  - e** Click the parameter hyperlinks, which opens the Configuration Parameters dialog box, and update the model diagnostic parameters. Save the model.
  - f** To verify that your model passes, rerun the check. Repeat steps from c to e, if necessary, to reach compliance.
  - g** To generate a results report of the Simulink Check checks, select the **MAB Checks** node, and then, from the toolbar, click **Report**.

### Check Model for Design Errors

While in the Model Advisor, you can also check your model for hidden design errors using Simulink Design Verifier.

- 1** In the left pane, in the **By Products > Simulink Design Verifier** folder, select **Design Error Detection**.
- 2** If not already checked, click the box beside **Design Error Detection**. All checks in the folder are selected.
- 3** From the tool strip, click **Run Checks**.
- 4** After the Model Advisor analysis, from the tool strip, click **Report**. This generates a HTML report of the check analysis.
- 5** In the generated report, click a **Simulink Design Verifier Results Summary** hyperlink. The dialog box provides tools to help you diagnose errors and warnings in your model.
  - a** Review the analysis results on the model. Click the **Compute target speed** subsystem. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.
  - b** Review the harness model or create one if it does not already exist.
  - c** View tests and export test cases.
  - d** Review the analysis report. To see a detailed analysis report, click **HTML** or **PDF**.

### See Also

#### Related Examples

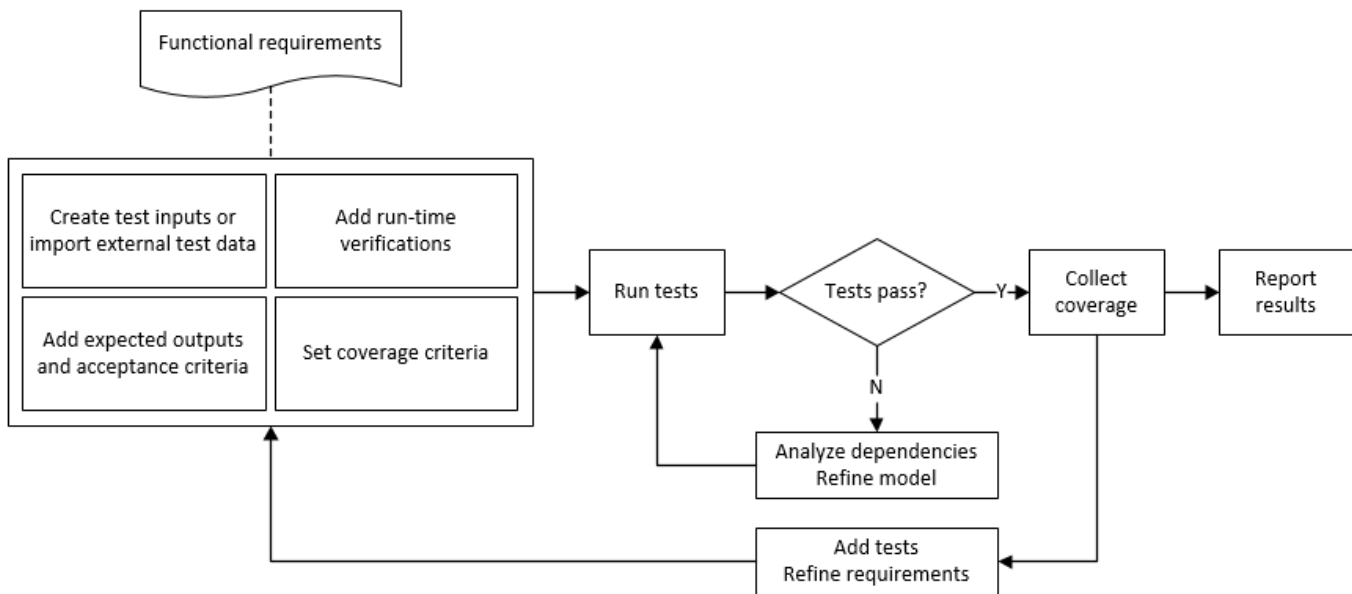
- “Check Model Compliance by Using the Model Advisor”
- “Collect Model Metrics Using the Model Advisor”
- “Analyze Models for Design Errors” on page 6-4
- “Prove Properties in a Model” on page 12-5



## Perform Functional Testing and Analyze Test Coverage

Functional testing begins with building test cases based on requirements. These tests can cover key aspects of your design and verify that individual model components meet requirements. Test cases include inputs, expected outputs, and acceptance criteria.

By collecting individual test cases within test suites, you can run functional tests systematically. To check for regression, add baseline criteria to the test cases and test the model iteratively. Coverage measurement reflects the extent to which these tests have fully exercised the model. Coverage measurement also helps you to add tests and requirements to meet coverage targets.



### Incrementally Increase Test Coverage Using Test Case Generation

This example shows how to perform requirements-based tests for a cruise control model. The tests link to a requirements document. You:

- 1 Run the tests.
- 2 Determine test coverage by using Simulink Coverage.
- 3 Increase coverage with additional tests generated by Simulink Design Verifier.
- 4 Report the results.

#### Open the Test Harness and Model

- 1 Open the project:

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

- 2 Open the model and the test harness. At the command line, enter:

```
open_system simulinkCruiseAddReqExample
sltest.harness.open("simulinkCruiseAddReqExample", "SafetyTest_Harness1")
```

- 3 Load the test suite from “Test Model Against Requirements and Report Results” (Simulink Test) and open the Simulink Test Manager.

```
pf = fullfile(pr.RootFolder,"tests","slReqTests.mldatx");
tf = sltest.testmanager.TestFile(pf);
sltest.testmanager.view
```

- 4 Open the Test Sequence block. The sequence verifies system disengagement when either:
  - The brake pedal is pressed.
  - Speed exceeds a limit.

### Measure Model Coverage

- 1 In the Simulink Test Manager, select the `slReqTests` test file.
- 2 To enable coverage collection, in the right page under **Coverage Settings**:
  - Select **Record coverage for referenced models**.
  - Specify a coverage filter by using **Coverage filter filename**.
  - Select **Decision**, **Condition**, and **MCDC**.
- 3 Click **Run** on the Test Manager toolstrip.
- 4 After the test completes, select **Results**. The test achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.

| ANALYZED MODEL              | REPORT CO... | DECISION | CONDITION | MCDC |
|-----------------------------|--------------|----------|-----------|------|
| simulinkCruiseAddReqExample | 31           | 50%      | 41%       | 25%  |

### Generate Tests to Increase Model Coverage

- 1 Use Simulink Design Verifier to generate additional tests to increase model coverage. In **Results and Artifacts**, select the `slReqTests` test file and open the **Aggregated Coverage Results** section located in the right pane.
- 2 Right-click the test results and select **Add Tests for Missing Coverage**.
- 3 Under **Harness**, choose **Create a new harness**.
- 4 Click **OK** to add tests to the test suite using Simulink Design Verifier. The model being tested must either be on the MATLAB path or in the working folder.
- 5 On the Test Manager toolstrip, click **Run** to execute the updated test suite. The test results include coverage for the combined test case inputs, achieving increased model coverage.

Alternatively, you can create and use tests to increase coverage programmatically by using `sltest.testmanager.addTestsForMissingCoverage` and `sltest.testmanager.TestOptions`.

## See Also

### Related Examples

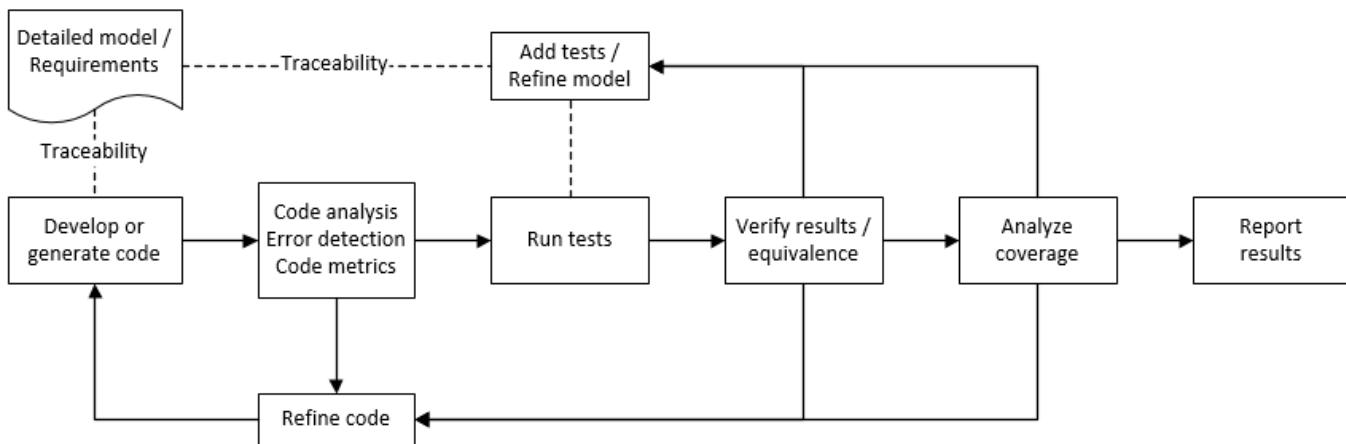
- “Link to Requirements” (Simulink Test)
- “Assess Model Simulation Using verify Statements” (Simulink Test)
- “Compare Model Output to Baseline Data” (Simulink Test)
- “Generate Test Cases for Model Decision Coverage” on page 7-6
- “Increase Test Coverage for a Model” (Simulink Test)

## Analyze Code and Test Software-in-the-Loop

### Code Analysis and Testing Software-in-the-Loop Overview

You can analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. For handwritten code, you typically check for run-time errors with static code analysis and run test cases that evaluate the code against requirements and evaluate code coverage. Based on the results, you refine the code and add tests.

In this example, you generate code and demonstrate that the code execution produces equivalent results to the model by using the same test cases and baseline results. Then you compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.



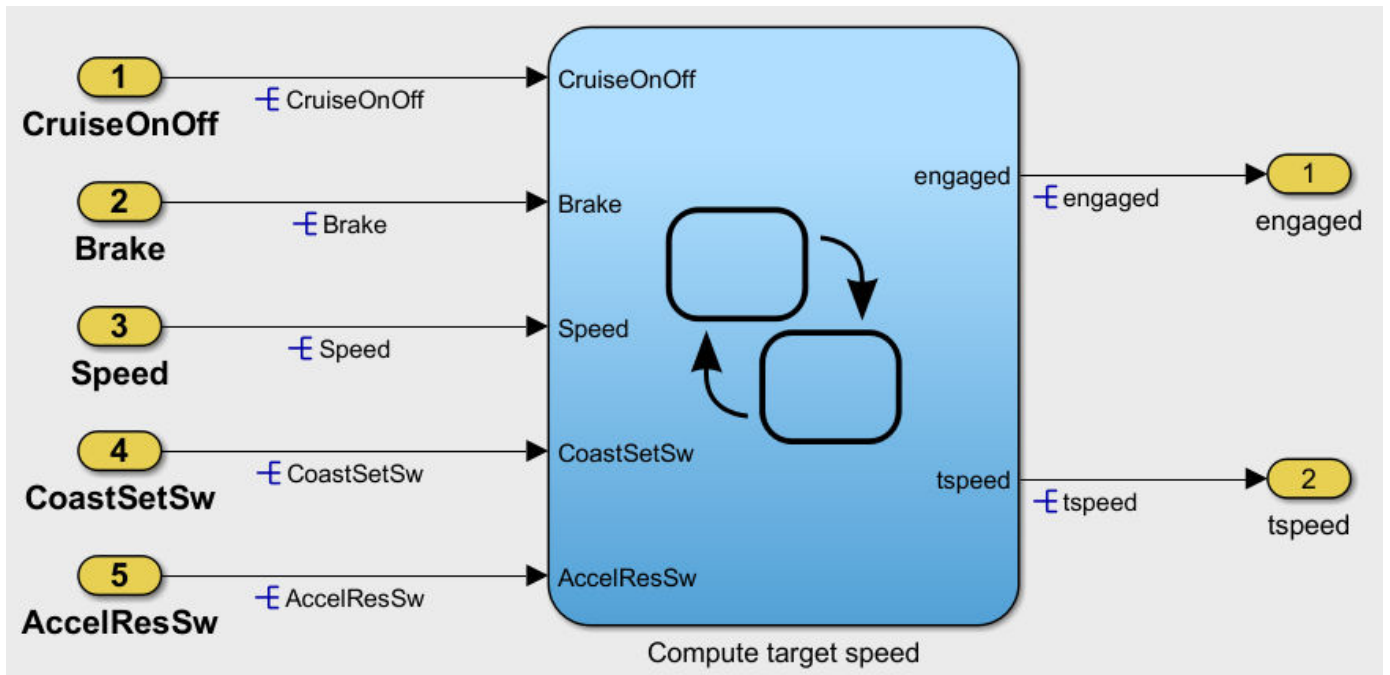
### Analyze Code for Defects, Metrics, and MISRA C:2012

This workflow describes how to check if your model produces MISRA™ C:2012 compliant code and how to check your generated code for code metrics and defects. To produce more MISRA compliant code from your model, you use the code generation and Model Advisor. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

- 1 Open the project.

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

- 2 From the project, open the model `simulinkCruiseErrorAndStandardsExample`.

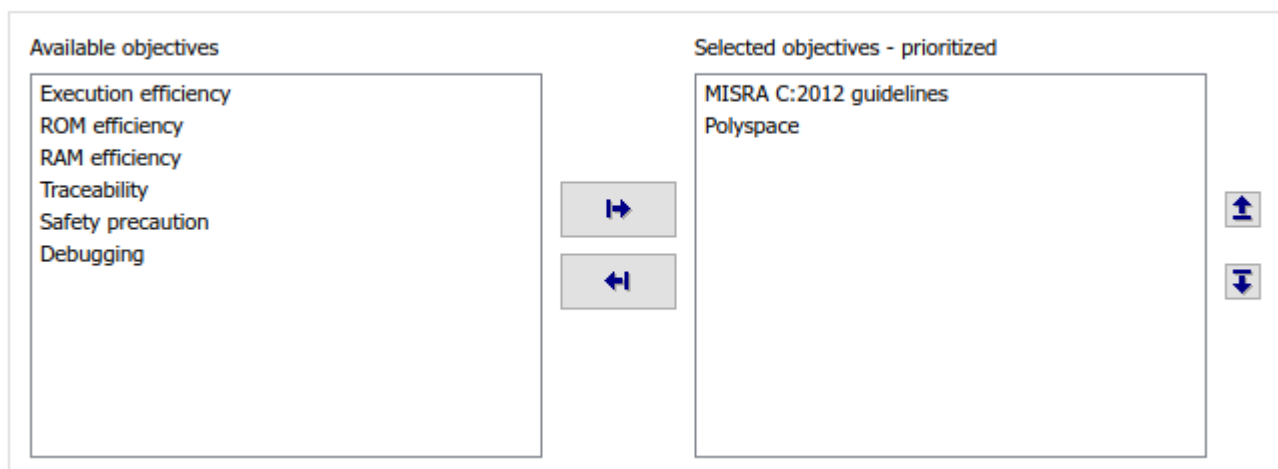


### Run Code Generator Checks

Check your model by using the Code Generation Advisor. Configure code generation parameters to generate code more compliant with MISRA C and more compatible with Polyspace.

- 1 Right-click Compute target speed and select **C/C++ Code > Code Generation Advisor**.
- 2 Select the Code Generation Advisor folder. In the right pane, move Polyspace to **Selected objectives - prioritized**. The MISRA C:2012 guidelines objective is already selected.

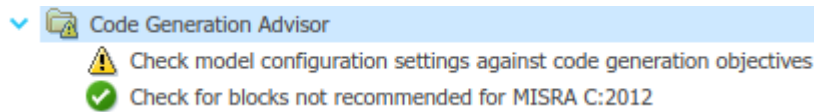
Code Generation Objectives (System target file: ert.tlc)



- 3 Click **Run Selected Checks**.

The Code Generation Advisor checks whether the model includes blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this

model, the check for incompatible blocks passes, but some configuration settings are incompatible with MISRA compliance and Polyspace checking.



- 4 Click the check that did not pass. Accept the parameter changes by selecting **Modify Parameters**.
- 5 Rerun the check by selecting **Run This Check**.

### Run Model Advisor Checks

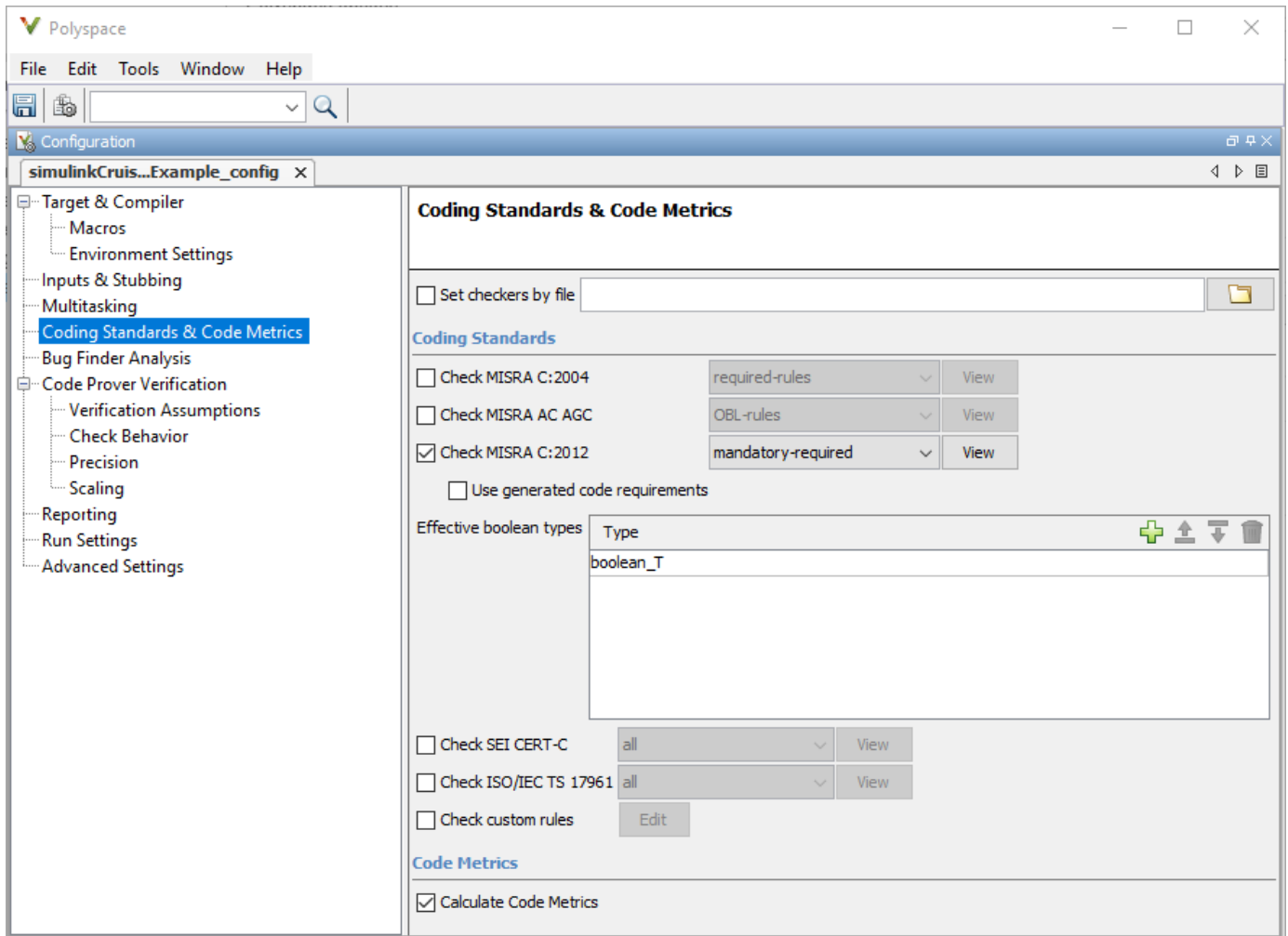
Before you generate code from your model, use the Model Advisor to check your model for MISRA C and Polyspace compliance. This example shows you how to use the Model Advisor to check your model before generating code.

- 1 At the bottom of the Code Generation Advisor window, select **Model Advisor**.
- 2 Under the **By Task** folder, select the **Modeling Standards for MISRA C:2012** advisor checks.
- 3 Click **Run Checks** and review the results.
- 4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

### Generate and Analyze Code

After you have done the model compliance checking, you can generate the code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

- 1 In the Simulink editor, right-click Compute target speed and select **C/C++ Code > Build This Subsystem**.
- 2 Use the default settings for the tunable parameters and select **Build**.
- 3 After the code is generated, in the Simulink Editor, right-click Compute target speed and select **Polyspace > Options**.
- 4 Click **Configure** to choose more advanced Polyspace analysis options in the Polyspace configuration window.



- 5 On the left pane, click **Coding Standards & Code Metrics**, then select **Calculate Code Metrics** to enable code metric calculations for your generated code.
- 6 Save and close the Polyspace configuration window.
- 7 From your model, right-click Compute target speed and select **Polyspace > Verify > Code Generated For Selected Subsystem**.

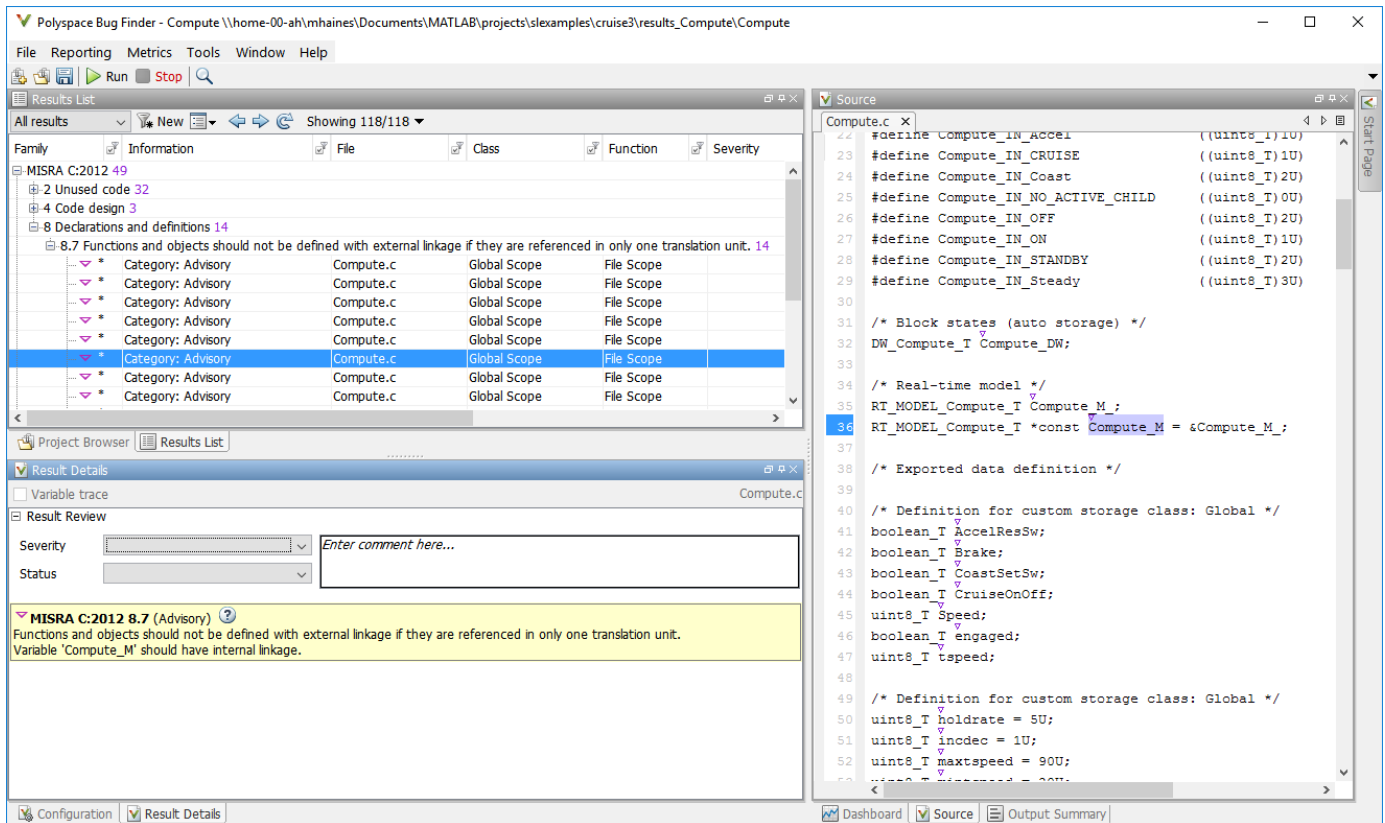
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks. You can see the progress of the analysis in the MATLAB Command Window. After the analysis finishes, the Polyspace environment opens.

## Review Results

The Polyspace environment shows you the results of the static code analysis.

- 1 Expand the tree for rule 8.7 and click through the different results.

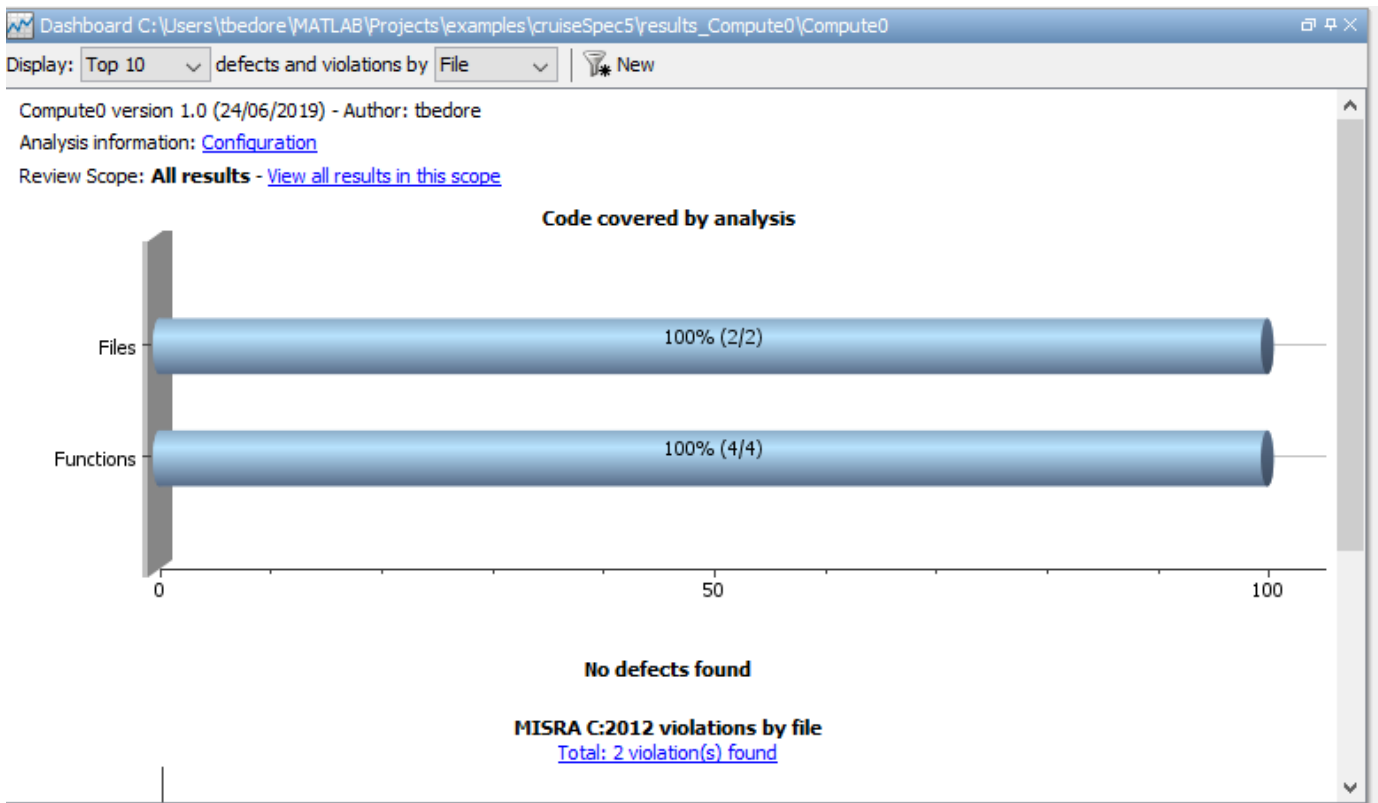
Rule 8.7 states that functions and objects should not be global if the function or object is local. As you click through the 8.7 violations, you can see that these results refer to variables that other components also use, such as `CruiseOnOff`. You can annotate your code or your model to justify every result. Because this model is a unit in a larger program, you can also change the configuration of the analysis to check only a subset of MISRA rules.



- 2 In your model, right-click Compute target speed and select **Polyspace > Options**.
- 3 Set the **Settings from** option to Project configuration to choose a subset of MISRA rules in the Polyspace configuration.
- 4 Click **Configure**.
- 5 In the Polyspace window, on the left pane, click **Coding Standards & Code Metrics**. Then select **Check MISRA C:2012** and, from the drop-down list, select **single-unit-rules**. Now Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.
- 6 Save and close the Polyspace configuration window.
- 7 Rerun the analysis with the new configuration.

The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, Polyspace found only two violations.





When you integrate this model with its parent model, you can add the rest of the MISRA C:2012 rules.

### Generate Report

To demonstrate compliance with MISRA C:2012 and report on your generated code metrics, you must export your results. If you want to generate a report every time you run an analysis, see [Generate report \(Polyspace Bug Finder\)](#).

- 1 If they are not open already, open your results in the Polyspace environment.
- 2 From the toolbar, select **Reporting > Run Report**.
- 3 Select **BugFinderSummary** as your report type.
- 4 Click **Run Report**.

The report is saved in the same folder as your results.

- 5 To open the report, select **Reporting > Open Report**.

### Test Code Against Model Using Software-in-the-Loop Testing

You previously showed that the model functionality meets its requirements by running test cases based on those requirements. Now run the same test cases on the generated code to show that the code produces equivalent results and fulfills the requirements. Then compare the code coverage to the model coverage to see the extent to which the tests exercised the generated code.

- 1 In MATLAB, in the project window, open the `tests` folder, then open `SILTests.mldatx`. The file opens in the Test Manager.

- 2 Review the test case. On the **Test Browser** pane, navigate to SIL Equivalence Test Case. This equivalence test case runs two simulations for the `simulinkCruiseErrorAndStandardsExample` model using a test harness.
  - Simulation 1 is a model simulation in normal mode.
  - Simulation 2 is a software-in-the-loop (SIL) simulation. For the SIL simulation, the test case runs the code generated from the model instead of running the model.

The equivalence test logs one output signal and compares the results from the simulations. The test case also collects coverage measurements for both simulations.

- 3 Run the equivalence test. Select the test case and click **Run**.
- 4 Review the results in the Test Manager. In the **Results and Artifacts** pane, select **SIL Equivalence Test Case** to see the test results. The test case passed and the results show that the code produced the same results as the model for this test case.

The screenshot shows the Test Manager interface with the following components:

- Toolbar:** Contains icons for New, Open, Save, Cut, Copy, Paste, Delete, Test Spec Report, Run, Run with Stepper, Stop, Parallel, Report, Visualize, Highlight in Model, Export, Import, Testing Dashboard, Preferences, and Help.
- Test Browser:** Shows a list of test cases under the heading "Results: 2021-Feb-04 10:35:44". The "SIL Equivalence Test Case" is selected and highlighted in blue. Below it are sub-items: "Equivalence Criteria Result", "Verify Statements 1", "Verify Statements 2", "Current: Sim Output 1 (simulink)", and "Current: Sim Output 2 (simulink)".
- Results and Artifacts:** Shows the details for the selected test case. It includes sections for SUMMARY, LOGS, DESCRIPTION, and COVERAGE RESULTS. The COVERAGE RESULTS section contains a table with the following data:

| ANALYZED MODEL                                      | REPORT | SIM MODE   | COM... | DECISION | CONDITION | MCDC | EXECUTION | FUNCTION | FUNCTION... |
|---|--------|------------|--------|----------|-----------|------|-----------|----------|-------------|
| <code>dlv_su32.c</code>                             |        | SIL        | 2      | 0%       | --        | --   | 0%        | 0%       | --          |
| <code>simulinkCruiseErrorAndStandardsExample</code> |        | ModelRe... | 22     | 54%      | 44%       | 17%  | 70%       | 100%     | 33%         |
| <code>simulinkCruiseErrorAndStandardsExample</code> |        | Normal     | 31     | 50%      | 41%       | 25%  | --        | --       | --          |

At the bottom right of the coverage table, there are buttons for "Add Tests for Missing Coverage" and "Export".

Below the Test Browser, there is a table showing the properties of the selected test case:

| PROPERTY   | VALUE                 |
|------------|-----------------------|
| Name       | SIL Equivalence Te... |
| Status     | 1                     |
| Start Time | 02/04/2021 10:36:10   |
| End Time   | 02/04/2021 10:38:36   |
| Type       | Equivalence Test      |

- 5 Expand the **Coverage Results** section of the results. The coverage measurements show the extent to which the test case exercised the model and the code. When you run multiple test cases, you can view aggregated coverage measurements in the results for the whole run. Use the coverage results to add tests and meet coverage requirements, as shown in "Perform Functional Testing and Analyze Test Coverage".

You can also test the generated code on your target hardware by running a processor-in-the-loop (PIL) simulation. By adding a PIL simulation to your test cases, you can compare the test results and coverage results from your model to the results from the generated code as it runs on the target hardware. For more information, see "Code Verification Through Software-in-the-Loop and Processor-in-the-Loop Execution" (Embedded Coder).

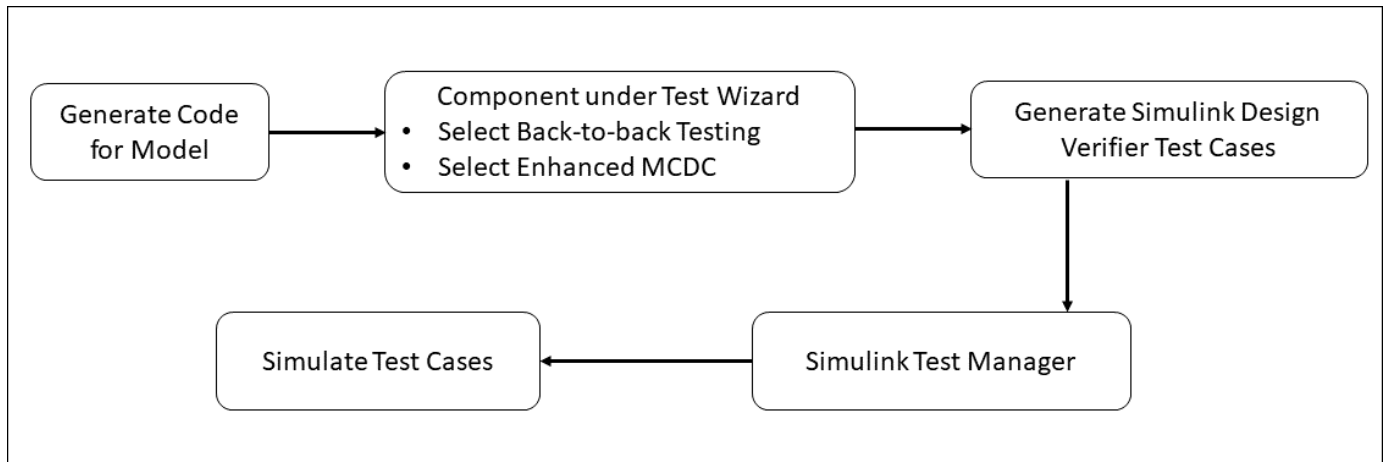
## See Also

### Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Bug Finder)
- “Test Two Simulations for Equivalence” (Simulink Test)
- “Export Test Results” (Simulink Test)

## Create Back-to-Back Tests Using Enhanced MCDC

Back-to-back tests, or equivalence tests, compare the results of normal simulations with the generated code results from software-in-the-loop (SIL), processor-in-the-loop (PIL), or hardware-in-the-loop (HIL) simulations. You can generate back-to-back tests in Simulink Test that use Enhanced MCDC.



### Set Up Test Inputs and Verification Strategy

If you want to test a component under test or subsystems in Simulink Test, you can use the **Create Test for Component wizard** by selecting **New > Create Test for Model Component** Simulink Test Test Manager, **Use Design Verifier to generate test input scenarios**. For detailed information, see “Generate Tests and Test Harnesses for a Model or Components” (Simulink Test).

To compare the results of running the component in two different simulation modes, select **Perform back-to-back testing** on the **Verification Strategy** tab of the wizard. For SIL testing an atomic subsystem or a reusable library subsystem, the subsystem or library that contains the subsystem must already have generated code. See “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42 for more information.

### Create Test for Model Component

[System](#) > [Test Inputs](#) > [Verification Strategy](#) > [Generated Test](#)

How do you want to test the component?

Use component under test output as baseline  
*Simulate the top model and record the outputs of the component to be used as baseline*

Perform back-to-back testing  
*Set up a test to compare the component under test outputs in different simulation modes*

Select simulation modes:

Simulation1:  ▼

Simulation2:  ▼

Set Model coverage objectives as Enhanced MCDC

Define the verification logic in the created harness  
*No verification logic will be automatically added to the test*

If, under **Perform back-to-back testing** you select Software-in-the-Loop or Processor-in-the-Loop for **Simulation2**, the **Set Model Coverage Objective as Enhanced MCDC** option appears. Enhanced MCDC extends decision coverage by generating test cases that avoid masking effects from downstream blocks.

## See Also

### Related Examples

- “Create and Run Back-to-Back Tests Using Enhanced MCDC” on page 8-18
- “Enhanced MCDC Coverage in Simulink Design Verifier” on page 7-42
- “Generate Tests and Test Harnesses for a Model or Components” (Simulink Test)



## Glossary

|                                     |  |
|-------------------------------------|--|
| <b>abstraction</b>                  | The process of ignoring certain aspects of model behavior that do not affect the test objective or property under investigation.   |
| <b>analysis model</b>               | 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.  |
| <b>assumption</b>                   | A property that is assumed to be true during a property proof. The proof result holds only when the assumption is true.  |
| <b>block replacement rule</b>       | A rule that is registered with Simulink Design Verifier and defines how instances of specific blocks are replaced by an alternate implementation. The software uses MATLAB commands to define when and how to apply a block replacement rule (see “Block Replacements for Unsupported Blocks” on page 4-7).  |
| <b>component verification</b>       | 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.  |
| <b>condition coverage</b>           | Measures the percentage of the total number of logic conditions associated with logical model objects that the simulation actually exercised. Enabling condition coverage causes every decision and condition coverage outcome to be enabled. See “Types of Model Coverage” (Simulink Coverage).   |
| <b>constraint</b>                   | A property that is forced to be true during test case generation.  |
| <b>counterexample</b>               | A test case that demonstrates a property violation.  |
| <b>coverage objective</b>           | A test objective that defines when a coverage point results in a particular outcome.   |
| <b>coverage point</b>               | A decision, condition, or MCDC expression associated with a model object. Each coverage point has a fixed number of mutually exclusive outcomes.   |
| <b>decision coverage</b>            | Measures the percentage of the total number of simulation paths through model objects that the simulation actually traversed. Decision coverage is a subset of modified decision/condition coverage. See “Types of Model Coverage” (Simulink Coverage).  |
| <b>floating-point approximation</b> | The process of approximating floating-point numbers using rational numbers (i.e., fractions whose numerator and denominator are small integers). The Simulink Design Verifier software performs floating-point approximations during its analysis. It can generate invalid test cases that result from numerical differences. For example, given a large enough floating-point number $x$ , the expression $x == (x+1)$ can be true; however, this expression never holds if $x$ is a rational number. |
| <b>invalid test case</b>            | A test case that does not satisfy its objectives.  |

**modified condition/  
decision coverage  
(MCDC)**

Measures the independence of logical block inputs and transition conditions associated with logical model objects during the simulation. When you set the coverage objective to MCDC, Simulink Design Verifier automatically enables every coverage objective for decision coverage and condition coverage as well.

Note that MCDC test cases are not generated for XOR configured logic operators. You can achieve MCDC by using the same tests that would be generated from AND configured blocks or OR configured blocks.

See “Types of Model Coverage” (Simulink Coverage).

**nonlinear arithmetic**

A computation in the model that cannot be expressed as a combination of mutually exclusive linear expressions. Nonlinear arithmetic can affect a property or test objective, and it can cause the analysis to return an error. In this case, you should apply simplifying approximations and abstractions.

**property**

A logical expression of the signals and data values, within a model, that is intended to be proven true during simulation. Properties evaluate at specific points in the model.

**property violation**

The condition during a simulation when a property is false.

**test case**

A sequence of numeric values and input data time that you input to a model during its simulation.

**test harness**

A model that runs test cases on an analysis model.

**test objective**

A logical expression of the signals and data values, within a model, that is intended to be true at least once in the resulting test case during simulation. Test objectives evaluate at specific points in the model.

**Test Objective block**

The block that you add to a model to define test objectives. In the block mask, define test objectives as values or ranges that an input signal must satisfy during a test case.

**unsatisfiable test  
objective**

The status of a test objective that indicates a test case cannot be generated for the specified approximations. This includes floating-point approximations and maximum-step limitations specified in the **Design Verifier > Test Generation** pane of the Configuration Parameters dialog box.

**validated property**

The status of a property that indicates no counterexample exists, subject to floating-point approximations and the settings specified in the **Property Proving** pane of the Configuration Parameters dialog box.