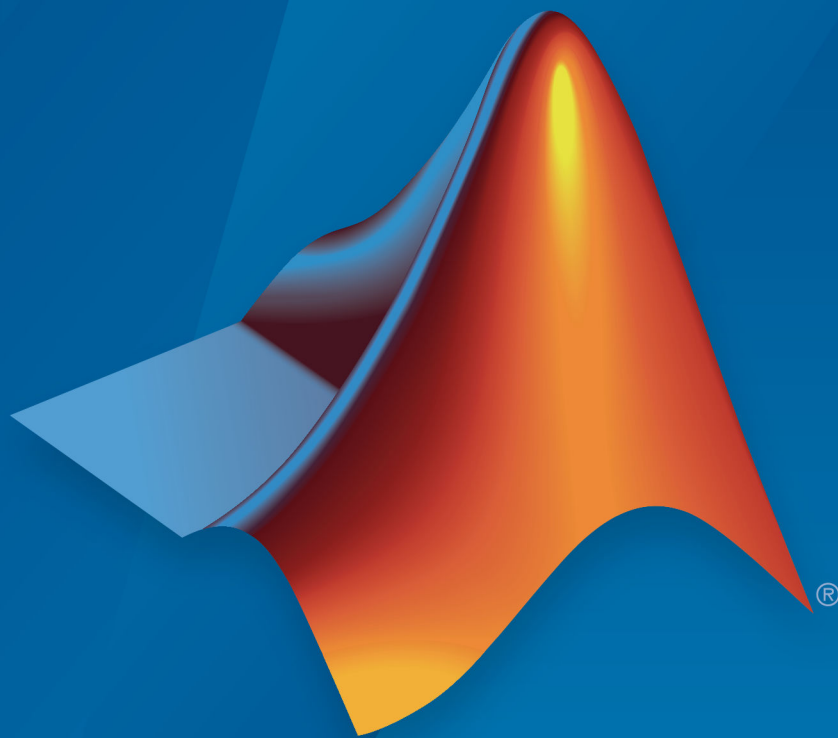


Simulink® Design Verifier™

Reference



MATLAB® & SIMULINK®

R2018a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

Simulink® Design Verifier™ Reference

© COPYRIGHT 2007–2018 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 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 for more information.

Revision History

September 2010	Online only	New 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)

Functions – Alphabetical List

1

Blocks – Alphabetical List

2

Model Advisor Checks

3

Simulink Design Verifier Checks	3-2
Simulink Design Verifier Checks Overview	3-2
Check compatibility with Simulink Design Verifier	3-2
Detect Dead Logic	3-4
Detect Integer Overflow	3-6
Detect Division by Zero	3-7
Detect Out Of Bound Array Access	3-8
Detect Violation of Specified Minimum and Maximum Values	3-10

Functions — Alphabetical List

sldv.assume

Proof assumption function for Stateflow charts and MATLAB Function blocks

Syntax

```
sldv.assume(expr)
```

Description

`sldv.assume(expr)` specifies that `expr` be true for every evaluation while proving properties. Use any valid Boolean expression for `expr`.

This function has no output and no impact on its parenting function, other than any indirect side effects of evaluating `expr`. If you issue this function from the MATLAB® command line, the function has no effect.

Intersperse `sldv.assume` proof assumptions within MATLAB code or separate the assumptions into a verification script.

The **Proof assumptions** option in the **Property proving** pane applies to proof assumptions represented with the `sldv.assume` function, as well as with the Proof Assumption block.

Input Arguments

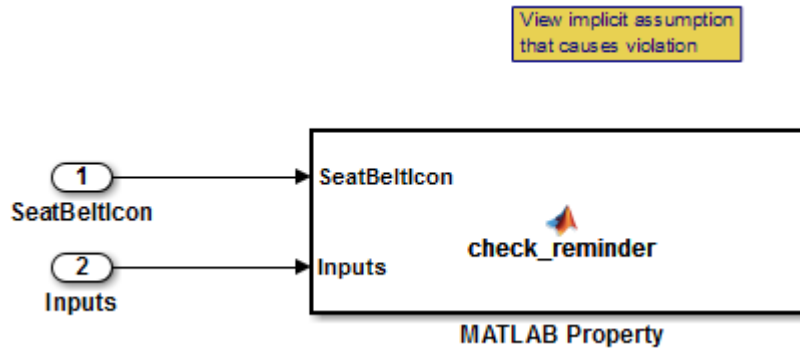
expr

MATLAB expression, for example, `x > 0`

Examples

Specify a property proof objective and proof assumption in a MATLAB Function block:

- 1 Open the sldvdemo_sbr_verification model and save it as ex_sldvdemo_sbr_verification.
- 2 Open the Safety Properties subsystem.



- 3 Open the **MATLAB Property** block, which is a MATLAB Function block.

```

Editor - Block: sldvdemo_sbr_verification/Safety Properties/MATLAB Property
Safety Properties/MATLAB Property x
1  function check_reminder(SeatBeltIcon,Inputs) %#codegen
2      % The seat belt light should be active whenever the key is turned on
3      % and speed is less than 15 and the seatbelt is not fastened
4      activeCond = ((Inputs.KEY ~= 0) && (Inputs.SeatBeltFasten == 0) && ...
5                  (Inputs.Speed < 15));
6
7      sldv.prove(implies(activeCond, SeatBeltIcon));
8
9      function out = implies(cond, result)
10         if (cond)
11             out = result;
12         else
13             out = true;
14         end
15

```

- 4 At the end of the `check_reminder` function definition, add the line `sldv.assume(Inputs.KEY==0 | 1)`; so that the last two lines of the function definition now read:

```
sldv.prove(implies(activeCond, SeatBeltIcon));  
sldv.assume(Inputs.KEY==0 | 1);
```

- 5 In the editor, save the updated code.
- 6 Prove the safety properties. With the model open in the Simulink Editor, select the Safety Properties subsystem and choose **Analysis > Design Verifier > Prove Properties > Selected Subsystem**.

In the Simulink Editor, you can also right-click the Safety Properties subsystem and select **Design Verifier > Prove Subsystem Properties**.

Alternatives

Instead of using the `sldv.assume` function, you can insert a Proof Assumption block in your model. However, using `sldv.assume` instead of a Proof Assumption block offers several benefits, described in “What Is Property Proving?”.

You can also constrain signal values when proving models by using MATLAB for code generation without using the `sldv.assume` function. However, using `sldv.assume` instead of directly using MATLAB for code generation eliminates the need to:

- Express the assumption with a Simulink block
- Explicitly connect the assumption output to a Simulink block

See Also

Proof Assumption | Proof Objective | Test Condition | Test Objective | `sldv.condition` | `sldv.prove` | `sldv.test`

Topics

“Prove Properties in a Model”

“Workflow for Proving Model Properties”

Introduced in R2009b

sldvblockreplacement

Replace blocks for analysis

Syntax

```
[status,newmodel] = sldvblockreplacement(model)
[status,newmodel] = sldvblockreplacement(model,options)
[status,newmodel] = sldvblockreplacement(model,options,showUI)
sldvblockreplacement(model,options)
```

Description

`[status,newmodel] = sldvblockreplacement(model)` copies the model `model` and replaces specified model blocks and other model components for a Simulink Design Verifier analysis. `sldvblockreplacement` replaces the blocks of the model according to the block-replacement rules in the model configuration settings.

`[status,newmodel] = sldvblockreplacement(model,options)` replaces the blocks of the model `model` according to the block-replacement rules specified in the `sldvoptions` object `options`, and returns a handle to the new model in `newmodel`.

`[status,newmodel] = sldvblockreplacement(model,options,showUI)` performs the same tasks as `sldvblockreplacement(model,options)`. If `showUI` is `true`, errors appear in the Diagnostic Viewer. Otherwise, errors appear at the MATLAB command line.

Examples

Replace Blocks in Model by Using Block-Replacement Rules

Replace the blocks in `sldvdemo_sqrt_blockrep` model by using the block-replacement rules specified in `opts`.

Open the `sldvdemo_sqrt_blockrep` example model.

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

Set the sldvoptions and specify the block-replacement rule.

```
opts = sldvoptions;  
opts.BlockReplacement = 'on';  
opts.BlockReplacementRulesList = ['sldvdemo_custom_blkrep_rule_sqrt.m,' ...  
    'blkrep_rule_lookup_normal.m,' ...  
    'blkrep_rule_switch_normal.m'];
```

Create a model by using sldvblockreplacement.

```
[status, newmodel] = sldvblockreplacement('sldvdemo_sqrt_blockrep', opts);
```

- “Replace Multiport Switch Blocks”

Input Arguments

model — Name or handle of model

character vector | string scalar

Name or handle to a Simulink model.

options — Specify analysis parameters

[] (default) | character vector

sldvoptions object that specifies the analysis parameters.

showUI — Display messages during analysis

logical

Logical value indicating where to display messages during analysis.

true to display messages in the log window

false (default) to display messages in the MATLAB command window

Output Arguments

status — Status of block-replacement

boolean

If the operation replaces the blocks, `sldvblockreplacement` returns a status of 1. Otherwise, it returns 0.

newmodel — Handle to new model

character vector

`sldvblockreplacement` returns a handle to the new model in `newmodel`.

See Also

`sldvoptions`

Topics

“Replace Multiport Switch Blocks”

“Define Custom Block Replacements”

Introduced in R2007a

sldvcompat

Check model for compatibility with analysis

Syntax

```
status = sldvcompat(model)
status = sldvcompat(subsystem)
status = sldvcompat(subsystem, options)
status = sldvcompat(model, options, showUI, startCov)
```

Description

`status = sldvcompat(model)` returns a `status` of 1 if the `model` is compatible with Simulink Design Verifier software. Otherwise, `sldvcompat` returns 0.

`status = sldvcompat(subsystem)` converts the Simulink atomic subsystem `subsystem` into a temporary model and checks the compatibility of the temporary model with Simulink Design Verifier software. After the compatibility check, `sldvcompat` closes the temporary model.

`status = sldvcompat(subsystem, options)` checks the subsystem specified by `subsystem` for compatibility with Simulink Design Verifier software by using the `sldvoptions` object `options`.

`status = sldvcompat(model, options, showUI, startCov)` checks the compatibility of the model with Simulink Design Verifier software. If `showUI` is `true`, errors appear in the Diagnostic Viewer. Otherwise, errors appear at the MATLAB command line. The analysis ignores all model coverage objectives satisfied in `startCov`, a `cvdata` object.

Examples

Check Model Compatibility

Check the `sldvdemo_flipflop` model for compatibility with Simulink Design Verifier software.

Open the `sldvdemo_flipflop` example model and check for compatibility.

```
open_system('sldvdemo_flipflop')
status = sldvcompat('sldvdemo_flipflop')
```

Input Arguments

model — Handle to model

[] (default) | character vector | string scalar

Handle to a Simulink model.

Example: `'sldvdemo_cruise_control'`

subsystem — Handle to atomic subsystem

character vector | string scalar

Handle to an atomic subsystem in a Simulink model.

options — Analysis parameters

[] (default) | character vector | string scalar

`sldvoptions` object that specifies the analysis parameters.

showUI — Display messages during analysis

logical

Logical value indicating where to display messages during analysis:

`true` to display messages in the log window.

`false` (default) to display messages in the Command Window.

startCov — Coverage data for model

character vector | string scalar

A `cvdata` object that contains coverage data for the model.

Output Arguments

status — Model is compatible

logical

If the `model` is compatible with Simulink Design Verifier software, the `status` is 1 . Otherwise, `sldvcompat` returns 0.

Alternatives

To check if a model is compatible with the Simulink Design Verifier software, in the Simulink Editor, select **Analysis > Design Verifier > Check Compatibility > Model**.

To check the compatibility of a subsystem, right-click the subsystem and select **Design Verifier > Check Subsystem Compatibility**.

See Also

`sldvoptions` | `sldvrun`

Topics

“Check Compatibility of the Example Model”

Introduced in R2007a

sldv.condition

Test condition function for Stateflow charts and MATLAB Function blocks

Syntax

```
sldv.condition(expr)
```

Description

`sldv.condition(expr)` specifies that `expr` is true for every time step in a generated test case. Use any valid Boolean expression for `expr`.

This function has no output and no impact on its parenting function, other than any indirect side effects of evaluating `expr`. If you issue this function from the MATLAB command line, the function has no effect.

Intersperse `sldv.condition` test conditions within MATLAB code or separate the conditions into a verification script.

The **Test conditions** option in the **Test generation** pane applies to test conditions represented with the `sldv.condition` function and with the Test Condition block.

Input Arguments

expr — Boolean expression for condition

boolean expression

MATLAB expression, for example, `x > 0`.

Examples

Add Test Objective and Test Conditions

Add a test objective and test conditions by using the MATLAB Function block.

Open the `sldvdemo_cruise_control` model and save it as `ex_sldvdemo_cruise_control`.

Remove the Test Condition block for the `speed` block signal. Instead of the Test Condition block, this example uses `sldv.test` and `sldv.condition`.

From the User-Defined Functions library, add a MATLAB Function block:

- Name the block `tests`.
- Open the block and add this code:

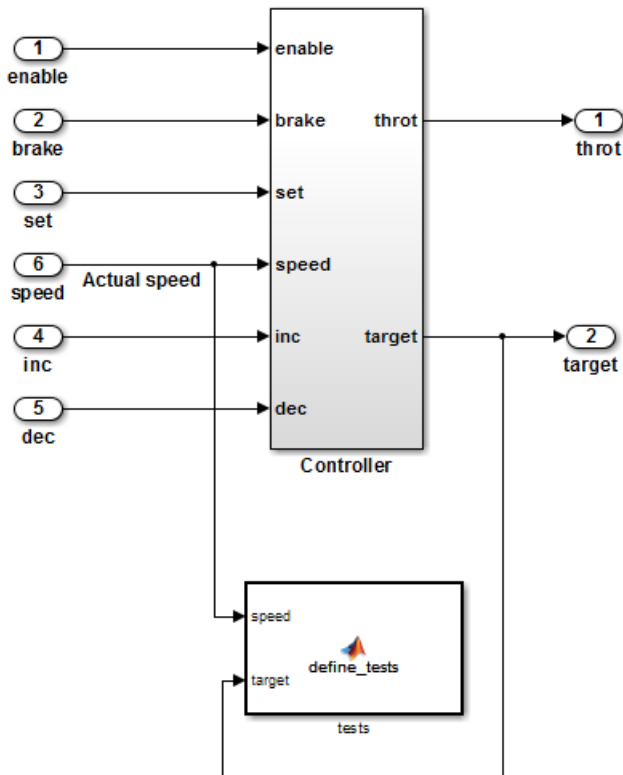
```
function define_tests(speed, target)
%#codegen

sldv.condition(speed >= 0 && speed <= 100);
sldv.test(speed > 60 && target > 40 && target < 50);
sldv.test(speed < 20 && target > 50);
```

- Save the code and close the editor.
- Connect the block to the signals for the `speed` block and for the `target` block.

Save the changes to the `ex_environment_controller` model.

Simulink Design Verifier Cruise Control Test Generation



Generate the test by selecting **Analysis > Design Verifier > Generate Tests > Model**.

- “Generate Test Cases for Model Decision Coverage”

Alternatives

Instead of using the `sldv.condition` function, you can insert a Test Condition block in your model. Using `sldv.condition` instead of a Test Condition block offers several benefits, described in “What Is Test Case Generation?”.

You can also specify test conditions by using MATLAB for code generation without using the `sldv.condition` function. Using `sldv.condition` instead of directly using MATLAB for code generation eliminates the need to:

- Express the constraints with Simulink blocks.
- Explicitly connect the condition output to a Simulink block.

See Also

[Proof Assumption](#) | [Proof Objective](#) | [Test Condition](#) | [Test Objective](#) | [sldv.assume](#) | [sldv.prove](#) | [sldv.test](#)

Topics

[“Generate Test Cases for Model Decision Coverage”](#)

[“Workflow for Test Case Generation”](#)

Introduced in R2009b

sldvextract

Extract subsystem or subchart contents into new model for analysis

Syntax

```
newModel = sldvextract(subsystem)
newModel = sldvextract(subchart)
newModel = sldvextract(subsystem, showModel)
newModel = sldvextract(subchart, showModel)
```

Description

`newModel = sldvextract(subsystem)` extracts the contents of the atomic subsystem `subsystem` and creates a model for the Simulink Design Verifier software to analyze. `sldvextract` returns the name of the new model in `newModel`. If the model name exists, `sldvextract` uses the subsystem name for the model name, appending a number to the model name.

`newModel = sldvextract(subchart)` extracts the contents of the atomic subchart `subchart` and creates a model for the Simulink Design Verifier software to analyze. Specify the full path of the atomic subchart in `subchart`. If the model name exists, `sldvextract` uses the subchart name for the model name, appending a number to the model name.

Note If the atomic subchart calls an exported graphical function that is outside the subchart, `sldvextract` creates the model, but the new model will not compile.

`newModel = sldvextract(subsystem, showModel)` and `newModel = sldvextract(subchart, showModel)` opens the extracted model when you set `showModel` to `true`. If `showModel` is set to `false`, the extracted model is only loaded into workspace.

Examples

Extract the Atomic Subsystem

Extract the atomic subsystem and copy it to a new model.

Extract the atomic subsystem Bus Counter from the `sldemo_mdhref_conversion` model and copy it to a new model.

```
open_system('sldemo_mdhref_conversion');  
newmodel = sldvextract('sldemo_mdhref_conversion/Bus Counter', true);
```

Extract the Atomic Subchart

Extract the atomic subchart and copy it to a new model.

Extract the atomic subchart, Sensor1, from the `sf_atomic_sensor_pair` model and copy it to a new model.

```
open_system('sf_atomic_sensor_pair');  
newmodel = sldvextract('sf_atomic_sensor_pair/RedundantSensors/Sensor1',...  
    true);
```

Input Arguments

subsystem — Path to atomic subsystem

character vector | string scalar

Full path to the atomic subsystem.

subchart — Path to Stateflow atomic subchart

character vector | string scalar

Full path to the Stateflow® atomic subchart.

showModel — Display the extracted model

True (default) | logical

Logical value that indicates whether to display the extracted model.

Output Arguments

newModel — Name of new model

character vector

Name of the new model.

See Also

Topics

“Extract Subsystems for Analysis”

Introduced in R2007a

sldvgencov

Analyze models to obtain missing model coverage

Syntax

```
[status, cvdo] = sldvgencov(model, options, showUI, startCov)
[status, cvdo] = sldvgencov(block, options, showUI, startCov)
[status, cvdo, filenames] = sldvgencov(model, options, showUI,
startCov)
[status, cvdo, filenames, newmodel] = sldvgencov(block, options,
showUI, startCov)
```

Description

[status, cvdo] = sldvgencov(model, options, showUI, startCov) analyzes the model model by using the sldvoptions object options.

[status, cvdo] = sldvgencov(block, options, showUI, startCov) analyzes the atomic subsystem block by using the sldvoptions object options.

[status, cvdo, filenames] = sldvgencov(model, options, showUI, startCov) analyzes the model and returns the file names that the software creates in filenames.

[status, cvdo, filenames, newmodel] = sldvgencov(block, options, showUI, startCov) analyzes the block by using the sldvoptions object options. The software returns a handle to the newmodel, which contains a copy of the block subsystem.

Examples

Collect Missing Coverage Data

Analyze the coverage data and collect the missing coverage data.

Analyze the Cruise Control model and simulate a version of that model by using data from test cases from the previous analysis. Compare the model coverage data and collect the coverage missing from the `sldvdemo_cruise_control_mod` model analysis:

```

opts = sldvoptions;
% Generate test cases
opts.Mode = 'TestGeneration';
% Specify MCDC coverage
opts.ModelCoverageObjectives = 'MDC';
% Don't create harness model
opts.SaveHarnessModel = 'off';
% or report
opts.SaveReport = 'off';
open_system('sldvdemo_cruise_control');
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
open_system('sldvdemo_cruise_control_mod');
[ outData, startCov ] = sldvrntest('sldvdemo_cruise_control_mod',...
    files.DataFile, [], true);
cvhtml('Coverage with the original test suite', startCov);
[ status, covData, files ] = sldvgencov('sldvdemo_cruise_control_mod',...
    opts, false, startCov);

```

- “Generate Test Cases for Model Decision Coverage”

Input Arguments

block — Handle to an atomic subsystem

character vector | string scalar

Handle to an atomic subsystem in a Simulink model.

model — Handle to a model

[] (default) | character vector | string scalar

Handle to a Simulink model.

options — Analysis parameters

[] (default) | character vector | string scalar

`sldvoptions` object that specifies the analysis parameters.

showUI — Display messages during analysis

logical

Logical value that indicates where to display messages during analysis:

true to display messages in the log window.
false (default) to display messages in the MATLAB command window.

startCov — Model coverage data

[] (default) | character vector | string scalar

cvdata object. The analysis ignores model coverage objectives already satisfied in startCov.

Output Arguments

cvdo — Coverage data

character vector

cvdata object containing coverage data for new tests.

filenames — Analysis results file names

structure

A structure whose fields list the file names resulting from the analysis.

DataFile	MAT-file with the raw input data.
HarnessModel	Simulink harness model.
Report	HTML report of the results.
ExtractedModel	Simulink model extracted from the subsystem.
BlockReplacementModel	Simulink model obtained after block replacements.

status — Status of model coverage data

logical

Logical value that indicates if the analysis collected model coverage.
true for analysis collected model coverage data.
false if analysis does not collect model coverage data.

See Also

sldvmergeharness | sldvoptions | sldvrun | sldvruntest

Topics

“Generate Test Cases for Model Decision Coverage”

Introduced in R2007a

sldvharnessopts

Default options for sldvmakeharness

Syntax

```
harnessopts = sldvharnessopts
```

Description

harnessopts = sldvharnessopts generates the default configuration for running sldvmakeharness.

Output Arguments

harnessopts

A structure whose fields specify the default options for sldvmakeharness when creating a Simulink Design Verifier harness model.

The harnessopts structure can have the following fields. If you do not specify values, the configuration uses default values.

Field	Description
harnessFilePath	Specifies the file path for creating the harness model. If an invalid path is specified, sldvmakeharness does not save the harness model, but it creates and opens the harness model. If this option is not specified, sldvmakeharness generates a new harness model and saves it in the MATLAB current folder. Default: ''

Field	Description
modelRefHarness	<p>Generates the test harness model that includes <code>model</code> in a Model block. When <code>false</code>, the test harness model includes a copy of <code>model</code>.</p> <p>Default: <code>true</code></p>
usedSignalsOnly	<p>When <code>true</code>, the Signal Builder block in the harness model has signals only for input signals used in the model. <code>model</code> must be compatible with the Simulink Design Verifier software to detect the used input signals.</p> <p>Default: <code>false</code></p>

Examples

Create a test harness for the `sldvdemo_cruise_control` model using the default options:

```
open_system('sldvdemo_cruise_control');
harnessOpts = sldvharnessopts;
[harnessfile] = sldvmakeharness('sldvdemo_cruise_control',...
    '', harnessOpts);
```

See Also

`sldvmakeharness`

Introduced in R2010b

sldvhighlight

Highlight model using data from Simulink Design Verifier analysis

Syntax

```
sldvhighlight
sldvhighlight(model)
sldvhighlight(model, dataFile)
```

Description

`sldvhighlight` highlights the current model using its active Simulink Design Verifier analysis results. If there are no active results, `sldvhighlight` loads the latest analysis results for the current model. The function highlights the model using these results.

`sldvhighlight(model)` highlights `model` using its active Simulink Design Verifier analysis results. If there are no active results, `sldvhighlight` loads the latest analysis results for `model`. The function highlights the model using these results.

`sldvhighlight(model, dataFile)` loads the Simulink Design Verifier analysis results from `dataFile`. The function highlights `model` using these results.

Examples

Highlight Active Analysis Results on Current Model

Highlight the current model with its active Simulink Design Verifier analysis results.

Open the `sldvdemo_debounce_modelcov` example model.

```
open_system('sldvdemo_debounce_modelcov')
```

Run test generation analysis on the example model using its default settings.

```
status = sldvrun('sldvdemo_debounce_modelcov')
```

```

Starting test generation for model 'sldvdemo_debounce_modelcov'
Compiling model... done
Translating model... done

'sldvdemo_debounce_modelcov' is compatible with Simulink Design Verifier.

Generating tests...
.....
Completed normally.

Generating output files:

  Data file:
  pwd\sldv_output\sldvdemo_debounce_modelcov\ ...
  sldvdemo_debounce_modelcov_sldvdata.mat

  Harness model:
  pwd\sldv_output\sldvdemo_debounce_modelcov\ ...
  sldvdemo_debounce_modelcov_harness.mdl

Results generation completed.

status =

    1

```

Highlight the results of the analysis on the current model, `sldvdemo_debounce_modelcov`.

```
sldvhighlight
```

The example model is highlighted with the analysis results. The Simulink Design Verifier Results Inspector opens.

In the model, click on a highlighted object to view detailed analysis results for that object in the Results Inspector.

Highlight Active Analysis Results on Specified Model

Highlight a specified model with its active Simulink Design Verifier analysis results.

Open the `sldvdemo_debounce_modelcov` example model.

```
open_system('sldvdemo_debounce_modelcov')
```

Run test generation analysis on the example model using its default settings.

```
status = sldvrun('sldvdemo_debounce_modelcov')
```

```

Starting test generation for model 'sldvdemo_debounce_modelcov'
Compiling model... done

```

```
Translating model... done
'sldvdemo_debounce_modelcov' is compatible with Simulink Design Verifier.
Generating tests...
.....
Completed normally.
Generating output files:
    Data file:
    pwd\sldv_output\sldvdemo_debounce_modelcov\ ...
    sldvdemo_debounce_modelcov_sldvdata.mat
    Harness model:
    pwd\sldv_output\sldvdemo_debounce_modelcov\ ...
    sldvdemo_debounce_modelcov_harness.mdl
Results generation completed.
status =
    1
```

Highlight the results of the analysis on `sldvdemo_debounce_modelcov`.

```
sldvhighlight('sldvdemo_debounce_modelcov')
```

The example model is highlighted with the analysis results. The Simulink Design Verifier Results Inspector opens.

In the model, click on a highlighted object to view detailed analysis results for that object in the Results Inspector.

Highlight Analysis Results from Data File on Specified Model

Highlight a specified model with its Simulink Design Verifier analysis results, loaded from a data file.

Open the `sldvdemo_debounce_modelcov` example model.

```
open_system('sldvdemo_debounce_modelcov')
```

Run test generation analysis on the example model using its default settings.

```
status = sldvrun('sldvdemo_debounce_modelcov')
```

```
Starting test generation for model 'sldvdemo_debounce_modelcov'
Compiling model... done
Translating model... done
```



```
'sldvdemo_debounce_modelcov' is compatible with Simulink Design Verifier.

Generating tests...
.....
Completed normally.

Generating output files:

    Data file:
    pwd\sldv_output\sldvdemo_debounce_modelcov\ ...
    sldvdemo_debounce_modelcov_sldvdata.mat

    Harness model:
    pwd\sldv_output\sldvdemo_debounce_modelcov\ ...
    sldvdemo_debounce_modelcov_harness.mdl

Results generation completed.

status =

    1
```

Close the example model and the harness model that the analysis produced.

```
bdclose('sldvdemo_debounce_modelcov')
bdclose('sldvdemo_debounce_modelcov_harness')
```

Reopen the example model.

```
open_system('sldvdemo_debounce_modelcov')
```

Highlight the example model with its analysis results, stored in the data file that the analysis created.

```
sldvhighlight('sldvdemo_debounce_modelcov',[pwd ...
'\sldv_output\sldvdemo_debounce_modelcov\' ...
'sldvdemo_debounce_modelcov_sldvdata.mat'])
```

The Simulink Design Verifier Results Inspector opens. The model is highlighted to show the results of the analysis.

In the model, click on a highlighted object to view detailed analysis results for that object in the Results Inspector.

Input Arguments

model — Name or handle of model to highlight

character vector | string scalar

Name of model or handle of model to highlight.

Example: 'sldvdemo_cruise_control'

Example: 'sldvdemo_flipflop'

dataFile — Name of analysis data file

character vector | string scalar

Name of Simulink Design Verifier analysis data file.

For more information about analysis data files, see “Simulink Design Verifier Data Files”.

Example: 'results.mat'

Example: 'sldv_output\sldvdemo_flipflop
\sldvdemo_flipflop_sldvdata.mat'

Example: 'sldv_output\my_model\my_model_sldvdata.mat'

See Also

sldvloadresults | sldvreport

Topics

“Highlighted Results on the Model”

“Simulink Design Verifier Data Files”

Introduced in R2013b

sldvisactive

Verify updating of a block diagram

Syntax

```
status = sldvisactive
status = sldvisactive(model)
status = sldvisactive(block)
```

Description

`status = sldvisactive` checks if the software is actively analyzing the current Simulink model. If the software is actively analyzing the current model, `sldvisactive` returns 1. Otherwise, it returns 0.

`status = sldvisactive(model)` checks if the software is actively analyzing `model`.

`status = sldvisactive(block)` checks if the Simulink Design Verifier software is actively analyzing the model that contains block `block`.

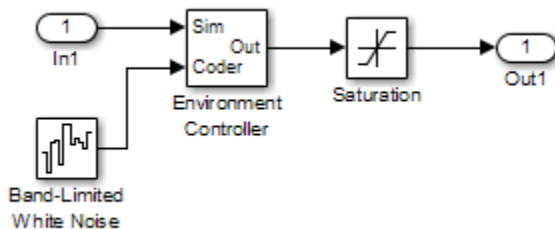
`sldvisactive` customizes the model analysis in the block and model callback functions or mask initialization.

Examples

Eliminate Incompatible Blocks

Eliminate blocks that are incompatible.

Create a Simulink model with the blocks and save as `ex_environment_controller`.



Right-click the Environment Controller block and select **Mask > View Base Mask**.

If the code does not exist, click the **Initialization** tab and add this command:

```
switch_mode = rtwenvironmentmode(bdroot(gcbh)) || ...
    (exist('sldvisactive','file')~=0 && ...
    sldvisactive(bdroot(gcbh)));
```

The software does not support Band-Limited White Noise blocks. If it is analyzing the `mEnvControl` model, the mask initialization of the Environment Controller block:

- Sets the pass-through mode to pass the Sim signal to the output port.
- Eliminates the Coder port that is incompatible.

Save the changes to the `ex_environment_controller` model.

Input Arguments

model — Name or handle of model

character vector | string scalar

Full path name or handle to a Simulink model.

block — Name or handle of block

character vector | string scalar

Full path name or handle to a Simulink block.

Output Arguments

status — **Actively analyzing model**

logical

If the software is actively analyzing the current model, `sldvisactive` returns 1. Otherwise, the `status` is 0.

See Also

`sldvcompat` | `sldvextract`

Topics

“Basic Workflow for Simulink Design Verifier”

“Check Model Compatibility”

“Extract Subsystems for Analysis”

Introduced in R2009a

sldvloadresults

Load Simulink Design Verifier analysis results for model

Syntax

```
status = sldvloadresults(model)
status = sldvloadresults(model, dataFile)
```

Description

`status = sldvloadresults(model)` loads the most recently generated Simulink Design Verifier analysis results for `model` into the Model Explorer. If `model` is not already open, `sldvloadresults` opens `model`. The function loads the results from the data file specified by **Analysis > Design Verifier > Options > Output directory** and **Analysis > Design Verifier > Options > Data file name**.

`status = sldvloadresults(model, dataFile)` loads analysis results for `model` from `dataFile` into the Model Explorer. If `model` is not already open, `sldvloadresults` opens `model`. The function loads the results from `dataFile`.

Examples

Load Active Results for Specified Model

Load active Simulink Design Verifier analysis results for a specified model.

Open the `sldvdemo_flipflop` example model.

```
open_system('sldvdemo_flipflop')
```

Run test generation analysis on the example model using its default settings.

```
status = sldvrun('sldvdemo_flipflop')
```

```
Starting test generation for model 'sldvdemo_flipflop'
Compiling model... done
```

```
Translating model... done
'sldvdemo_flipflop' is compatible with Simulink Design Verifier.
Generating tests...
.....
Completed normally.
Generating output files:
    Data file:
    pwd\sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat
Results generation completed.
status =
    1
```

Close the example model.

```
bdclose('sldvdemo_flipflop')
```

Reopen the example model. Load its most recently generated analysis results.

```
sldvloadresults('sldvdemo_flipflop')
```

```
ans =
    1
```

You can view the loaded analysis results in the Model Explorer or in the Simulink Design Verifier Results Summary window. To open this window, in the Simulink Editor, select **Analysis > Design Verifier > Results > Active**.

Load Results from Data File for Specified Model

Load Simulink Design Verifier analysis results from a data file for a specified model.

Open the `sldvdemo_flipflop` example model.

```
open_system('sldvdemo_flipflop')
```

Run test generation analysis on the example model using its default settings.

```
status = sldvrun('sldvdemo_flipflop')
```

```
Starting test generation for model 'sldvdemo_flipflop'
Compiling model... done
```

```
Translating model... done
'sldvdemo_flipflop' is compatible with Simulink Design Verifier.
Generating tests...
.....
Completed normally.
Generating output files:
    Data file:
    pwd\sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat
Results generation completed.
status =
    1
```

Close the example model.

```
bdclose('sldvdemo_flipflop')
```

Reopen the example model. Load analysis results for the model from the data file that the analysis generated.

```
sldvloadresults('sldvdemo_flipflop',[pwd '\sldv_output ...
\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat'])
ans =
    1
```

You can view the loaded analysis results in the Model Explorer or in the Simulink Design Verifier Results Summary window. To open this window, in the Simulink Editor, select **Analysis > Design Verifier > Results > Active**.

Input Arguments

model — Name or handle of model

character vector | string scalar

Name of the model or handle of model for which to load analysis results.

Example: 'sldvdemo_cruise_control'

Example: 'sldvdemo_flipflop'

dataFile — Name of data file containing analysis results

character vector | string scalar

Name of data file containing analysis results. `dataFile` must contain analysis results for the specified model.

If `dataFile` was generated with a previous version of `model`, when you load the results from `dataFile`, you might see unexpected effects. To avoid inconsistencies between your model and analysis results data, when you specify `dataFile`, choose a data file that contains results from the same version of `model`.

For more information about analysis data files, see “Simulink Design Verifier Data Files”.

Example: 'results.mat'

Example: 'sldv_output\sldvdemo_flipflop
\sldvdemo_flipflop_sldvdata.mat'

Example: 'sldv_output\my_model\my_model_sldvdata.mat'

Output Arguments

status — Outcome of attempt to load results

logical

Outcome of attempt to load results, returned as a logical value.

Logical Value Returned	Status of Loaded Results
true	Processing completed normally
false	An error occurred

See Also

sldvhighlight | sldvreport

Topics

“Review Analysis Results”

“Simulink Design Verifier Data Files”

Introduced in R2013b

sldvlogsignals

Log simulation input port values

Syntax

```
data = sldvlogsignals(model_block)
data = sldvlogsignals(harness_model)
data = sldvlogsignals(harness_model, test_case_index)
```

Note `sldvlogsignals` replaces `sldvlogdata`. Use `sldvlogsignals` instead.

Description

`data = sldvlogsignals(model_block)` simulates the model that contains `model_block` and logs the input signals to the `model_block` block. `model_block` must be a Simulink Model block. `sldvlogsignals` records the logged data in the structure `data`.

`data = sldvlogsignals(harness_model)` simulates every test case in `harness_model` and logs the input signals to the Test Unit block in the harness model. You must generate `harness_model` using Simulink Design Verifier analysis, `sldvmakeharness`, or `slvnmakeharness`.

`data = sldvlogsignals(harness_model, test_case_index)` simulates every test case in the Signal Builder block of the `harness_model` that is specified by `test_case_index`. `sldvlogsignals` logs the input signals to the Test Unit block in the harness model. If you omit `test_case_index`, `sldvlogsignals` simulates every test case in the Signal Builder.

Input Arguments

model_block — Block path or handle to model

character vector | string scalar

Full block path name or handle to a Simulink Model block

harness_model — Name or handle to a harness model

character vector | string scalar

Name or handle to a harness model that the Simulink Design Verifier software, `sldvmakeharness`, or `slvnmakeharness` creates

test_case_index — Simulate test cases

character vector | cell array of character vectors | string array

Array of integers that specifies which test cases in the Signal Builder block of the harness model to simulate

Output Arguments

data

Structure that contains the logged data

Examples

Use logged signals to create a harness model in order to visualize the data:

- 1 Simulate the CounterB Model block, which references the `sldemo_mdhref_counter` model, in the context of the `sldemo_mdhref_basic` model. Then log the data:

```
open_system('sldemo_mdhref_basic');  
data = sldvlogsignals('sldemo_mdhref_basic/CounterB');
```

- 2 Create a harness model for `sldemo_mdhref_counter` using the logged data and the default harness options:

```
load_system('sldemo_mdhref_counter');  
harnessOpts = sldvharnessopts;
```

```
[~, harnessFilePath] = ...  
    sldvmakeharness('sldemo_mdref_counter', data, harnessOpts);
```

See Also

Topics

“Extend Test Cases for Model with Temporal Logic”

“Extend Test Cases for Closed-Loop System”

Introduced in R2010b

sldvmakeharness

Generate harness model

Syntax

```
[savedHarnessFilePath] = sldvmakeharness(model)
[savedHarnessFilePath] = sldvmakeharness(model, dataFile)
[savedHarnessFilePath] = sldvmakeharness(model, dataFile,
harnessOpts)
```

Description

[savedHarnessFilePath] = sldvmakeharness(model) generates a test harness from the model, which is a handle to a Simulink model or the model name. sldvmakeharness returns the path and file name of the generated harness model in savedHarnessFilePath. sldvmakeharness creates an empty harness model. The test harness includes one default test case that specifies the default values for all input signals.

[savedHarnessFilePath] = sldvmakeharness(model, dataFile) generates a test harness from the data file dataFile.

[savedHarnessFilePath] = sldvmakeharness(model, dataFile, harnessOpts) generates a test harness from model by using the dataFile and the harnessOpts, which specifies the harness creation options. If the dataFile is not available, requires '' for dataFile.

If the software generates a harness, it does not imply that your model is compatible with the Simulink Design Verifier software.

Examples

Create a Test Harness

Create a test harness for the `sldvdemo_cruise_control` model by using the default options.

Open the `sldvdemo_cruise_control` model by using the default options and create a harness model:

```
open_system('sldvdemo_cruise_control');  
harnessopts=sldvharnessopts();  
[harnessfile] = sldvmakeharness('sldvdemo_cruise_control', '', harnessopts);
```

Input Arguments

model — Name or handle of model

character vector | string scalar

Handle to a Simulink model or the model name.

dataFile — Name of sldvData file

' ' (default) | character vector | string scalar

Name of the `sldvData` file.

harnessOpts — Configurations for sldvmakeharness

character vector | string scalar

A structure whose fields specify these configurations for `sldvmakeharness`, as listed in this table.

Field	Description
<code>harnessFilePath</code>	<p>Specifies the file path for creating the harness model. If an invalid path is specified, <code>sldvmakeharness</code> does not save the harness model, but it creates and opens the harness model. If this option is not specified, <code>sldvmakeharness</code> generates a new harness model and saves it in the MATLAB current folder.</p> <p>Default: ''</p>
<code>modelRefHarness</code>	<p>Generates the test harness model that includes <code>model</code> in a Model block. When <code>false</code>, the test harness model includes a copy of <code>model</code>.</p> <p>Default: <code>true</code></p>
<code>usedSignalsOnly</code>	<p>When <code>true</code>, the Signal Builder block in the harness model has signals for only input signals used in the model. <code>model</code> must be compatible with the software to detect the used input signals.</p> <p>Default: <code>false</code></p>

Note To create a default `harnessOpts` object, use `sldvharnessopts`.

Output Arguments

savedHarnessFilePath — File name of generated harness model

character vector

The path and file name of the generated harness model.

Alternatives

sldvmakeharness creates a test harness model without analyzing the model. To analyze the model and create a test harness:

- 1 In the Simulink Editor, select **Analysis > Design Verifier > Options**.
In the Configuration Parameters dialog box, the **Design Verifier** node is expanded.
- 2 Select the **Results** node. In **Harness model options**, set the options that you want.
- 3 Click **OK** to save your changes and close the Configuration Parameters dialog box.
- 4 In the Simulink Editor, select **Analysis > Design Verifier > Generate Tests** to run a test-generation analysis.

See Also

sldvharnessopts | sldvmergeharness | sldvrun | slvnharnessopts | slvnmakeharness | slvnmergeharness

Topics

“Simulink Design Verifier Harness Models”

“Create Harness Model”

“Generate Test Harness Model and Record Coverage Data”

Introduced in R2009b

sldvmergeharness

Merge test cases and initializations into one harness model

Note `sldvmergeharness` replaces `sldvharnessmerge`. Use `sldvmergeharness` instead.

Syntax

```
status = sldvmergeharness(name, models, initialization_commands)
```

Description

`status = sldvmergeharness(name, models, initialization_commands)` collects the test data and initialization commands from each test harness model in `models`. `sldvharnessmerge` saves the data and initialization commands in `name`, which is a handle to the new model.

If `name` does not exist, `sldvmergeharness` creates it as a copy of the first model in `models`. `sldvmergeharness` then merges data from other models listed in `models` into this model. If you create `name` from a previous `sldvmergeharness` run, subsequent runs of `sldvmergeharness` for `name` maintain the structure and initialization from the earlier run. If `name` matches an existing Simulink model, `sldvmergeharness` merges the test data from `models` into `name`.

`sldvmergeharness` assumes that `name` and the rest of the models in `models` have only one Signal Builder block on the top level. If a model in `models` does not meet this restriction or its top-level Signal Builder block does not have the same number of signals as the top-level Signal Builder block in `name`, `sldvmergeharness` does not merge that model's test data into `name`.

Use `sldvmergeharness` with `sldvgencov` to combine test cases that use different sets of parameter values.

Input Arguments

name — Name of harness model

character vector | string scalar

Name of the new harness model, to be stored in the default MATLAB folder

models — Names of harness models

character vector | cell array of character vectors | string array

A cell array that represents harness model names

initialization_commands — Parameter settings for test cases

character vector | cell array of character vectors | string array

A cell array the same length as `models`. `initialization_commands` defines parameter settings for the test cases of each test harness model.

Output Arguments

status

If the operation works, `sldvmergeharness` returns a `status` of 1. Otherwise, it returns 0.

Examples

Analyze the `sldvdemo_cruise_control` model for decision and for full coverage and merge the two test harnesses:

```
model = 'sldvdemo_cruise_control';
open_system(model)
% Collect decision coverage
opts1 = sldvoptions;
opts1.Mode = 'TestGeneration';
opts1.ModelCoverageObjectives = 'Decision';
opts1.HarnessModelFileName = 'first_harness';
opts1.SaveHarnessModel = 'on';
sldvrun(model, opts1);
% Collect full coverage
opts2 = sldvoptions;
opts2.Mode = 'TestGeneration';
```

```
opts2.ModelCoverageObjectives = 'ConditionDecision';
opts2.HarnessModelFileName = 'second_harness';
opts2.SaveHarnessModel = 'on';
sldvrun(model, opts2);
% Merge the two harness files:
status = sldvmergeharness('new_harness_model', {'first_harness',...
    'second_harness'});
```

See Also

sldvgencov | sldvmakeharness | sldvrun

Introduced in R2010b

sldvoptions

Create design verification options object

Syntax

```
options = sldvoptions  
options = sldvoptions(model)
```

Description

`options = sldvoptions` returns an object `options` that contains the default values for the design verification parameters.

`options = sldvoptions(model)` returns the object `options` attached to `model`.

Examples

Create an Options Object

Create an options object and set several parameters.

Create an `opts` option for the `sldvdemo_cruise_control` model:

```
opts = sldvoptions;  
opts.AutomaticStubbing = 'on';  
opts.Mode = 'TestGeneration';  
opts.ModelCoverageObjectives = 'MCDC';  
opts.ReportIncludeGraphics = 'on';  
opts.SaveHarnessModel = 'off';  
opts.SaveReport = 'off';  
opts.TestSuiteOptimization = 'LongTestCases';
```

Get the options object for the `sldvdemo_cruise_control` model:

```
sldvdemo_cruise_control  
optsModel = sldvoptions(bdroot);
```

```
optsCopy = optsModel.deepCopy;  
optsCopy.MaxProcessTime = 120;
```

Input Arguments

model — Name or handle to a model

character vector | string scalar

Name or handle to a Simulink model.

Output Arguments

options — Options for design verification

character vector

This table lists the parameters that comprise a Simulink Design Verifier options object.

Parameter	Description	Values
AbsoluteTolerance	Specify an absolute value for tolerance in relational boundary tests.	double { '1.0e-05' }
Assertions	Specify whether Assertion blocks in your model are enabled or disabled.	'EnableAll' 'DisableAll' 'UseLocalSettings' (default)
AutomaticStubbing	Specify whether the software ignores unsupported blocks and functions and proceeds with the analysis.	'on' (default) 'off'

Parameter	Description	Values
BlockReplacement	<p>Specify whether the software replaces blocks in a model before its analysis.</p> <p>When set to 'on', this parameter enables BlockReplacementModel-FileName and BlockReplacementRules-List.</p>	<p>'on'</p> <p>'off' (default)</p>
BlockReplacementModel-FileName	<p>Specify a folder and file name for the model that is the result after applying block replacement rules.</p> <p>This parameter is enabled when BlockReplacement is set to 'on'.</p>	<p>character array</p> <p>'\$modelName \$_replacement' (default)</p>
BlockReplacementRules-List	<p>Specify a list of block replacement rules that execute before its analysis.</p> <p>This parameter is enabled when BlockReplacement is set to 'on'.</p>	<p>character array</p> <p>'<FactoryDefaultRules>' (default)</p>
CoverageDataFile	<p>Specify a folder and file name for the file that contains data about satisfied coverage objectives.</p> <p>This parameter is enabled when IgnoreCovSatisfied is set to 'on'.</p>	<p>character array</p> <p>' ' (default)</p>

Parameter	Description	Values
CovFilter	<p>For test generation analysis, specify whether to ignore test objectives stored in coverage filter file.</p> <p>When set to on, this parameter enables CovFilterFileName.</p>	<p>'on'</p> <p>'off' (default)</p>
CovFilterFileName	<p>For test generation, specify a name for the coverage filter file that contains test objectives to exclude from analysis.</p> <p>This parameter is enabled when CovFilter is set to 'on'.</p>	<p>character array</p> <p>' ' (default)</p>
DataFileName	<p>Specify a folder and file name for the MAT-file that contains the data generated during the analysis, stored in an sldvData structure.</p> <p>This parameter is enabled when SaveDataFile is set to 'on'.</p>	<p>character array</p> <p>'\$modelName\$_sldvdata' (default)</p>
DesignMinMaxCheck	<p>Specify whether to check that the intermediate and output signals in your model are within the range of specified minimum and maximum constraints.</p> <p>This parameter is disabled when DetectDeadLogic is set to 'on'.</p>	<p>'on'</p> <p>'off' (default)</p>
DesignMinMaxConstraints	<p>Specify whether Simulink Design Verifier software generates test cases that consider specified minimum and maximum values as constraints for input signals in your model.</p>	<p>'on' (default)</p> <p>'off'</p>

Parameter	Description	Values
DetectActiveLogic	Specify whether to analyze your model for active logic. This parameter is enabled only if DetectDeadLogic is set to 'on'.	'on' 'off' (default)
DetectDeadLogic	Specify whether to analyze your model for dead logic. When set to 'on', this parameter disables DetectDivisionByZero, DetectIntegerOverflow, DetectOutOfBounds, and DesignMinMaxCheck.	'on' 'off' (default)
DetectDivisionByZero	Specify whether to analyze your model for division-by-zero errors. This parameter is disabled when DetectDeadLogic is set to 'on'.	'on' (default) 'off'
DetectIntegerOverflow	Specify whether to analyze your model for integer and fixed-point data overflow errors. This parameter is disabled when DetectDeadLogic is set to 'on'.	'on' (default) 'off'
DetectOutOfBounds	Specify whether to analyze your model for out of bounds array access errors. This parameter is disabled when DetectDeadLogic is set to 'on'.	'on' 'off' (default)

Parameter	Description	Values
DisplayReport	<p>Display the report that the Simulink Design Verifier analysis generates after completing its analysis.</p> <p>This parameter is enabled when SaveReport is set to 'on'.</p>	'on' (default) 'off'
DisplayUnsatisfiable-Objectives	<p>Specify whether to display warnings if the analysis detects unsatisfiable test objectives.</p> <p>This parameter is enabled when Mode is set to 'TestGeneration'.</p>	'on' 'off' (default)

Parameter	Description	Values
ExistingTestFile	<p>Specify a folder and file name for the MAT-file that contains the logged test case data.</p> <p>This parameter is enabled when Mode is set to 'TestGeneration' and ExtendExistingTests is set to 'on'.</p> <p>When you configure Simulink Design Verifier to treat parameters as variables in analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with ExistingTestFile and apply parameter configurations with ParametersConfigFileName or the Parameter Configuration table, in the analysis, the software reports that your model is incompatible. This error occurs because the existing test cases do not include corresponding parameter values.</p>	<p>character array</p> <p>' ' (default)</p>

Parameter	Description	Values
ExtendExistingTests	<p>Extend the Simulink Design Verifier analysis by importing test cases logged from a harness model or a closed-loop simulation model.</p> <p>When set to 'on', this parameter enables ExistingTestFile and IgnoreExistTestSatisfied.</p> <p>This parameter is enabled when Mode is set to 'TestGeneration'.</p> <p>When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with ExistingTestFile and apply parameter configurations with ParametersConfigFileName or the Parameter Configuration table, in the analysis, the software reports that your model is incompatible. This error occurs because the existing test cases do not include corresponding parameter values.</p>	<p>'on'</p> <p>'off' (default)</p>

Parameter	Description	Values
HarnessModelFileName	Specify a folder and file name for the harness model. This parameter is enabled when SaveHarnessModel is set to 'on'.	character array '\$ModelName\$_harness' (default)
IgnoreCovSatisfied	Specify to analyze the model, ignoring satisfied coverage objectives, as specified in CoverageDataFile.	'on' 'off' (default)
IgnoreExistTestSatisfied	Ignore the coverage objectives satisfied by the logged test cases in ExistingTestFile. This parameter is enabled when Mode is set to 'TestGeneration' and ExtendExistingTests is set to 'on'.	'on' (default) 'off'
IncludeRelationalBoundary	Specify generation of test cases that satisfy relational boundary objectives.	'on' 'off' (default)
MakeOutputFilesUnique	Specify whether the software makes its output file names unique by appending a numeric suffix.	'on' (default) 'off'
MaxProcessTime	Specify the maximum time (in seconds) for analyzing a model.	double '300' (default)

Parameter	Description	Values
MaxTestCaseSteps	<p>Specify the maximum number of simulation steps when attempting to satisfy a test objective.</p> <p>The analysis uses the MaxTestCaseSteps 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 completes 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.</p> <p>To achieve the best performance, set the MaxTestCaseSteps parameter to a value large enough to bound the longest required test case, even if the test cases that are ultimately generated are longer than this value.</p> <p>When you set the TestSuiteOptimization parameter to 'LongTestCases', the analysis uses successive passes of test generation to extend a potential test case so that it satisfies more objectives. The analysis applies the MaxTestCaseSteps parameter</p>	<p>int32</p> <p>'500' (default)</p>

Parameter	Description	Values
	<p>to each individual iteration of test generation.</p> <p>This parameter is enabled when Mode is set to 'TestGeneration'.</p>	
MaxViolationSteps	<p>Specify the maximum number of simulation steps over which the software searches for property violations.</p> <p>This parameter is enabled when Mode is set to 'PropertyProving' and when ProvingStrategy is set to 'FindViolation' or 'ProveWithViolation-Detection'.</p>	<p>int32</p> <p>'20' (default)</p>
Mode	Specify the analysis mode.	<p>'TestGeneration' (default)</p> <p>'PropertyProving'</p> <p>'DesignErrorDetection'</p>
ModelCoverageObjectives	<p>Specify the type of model coverage to achieve.</p> <p>When ModelCoverageObjectives is set to 'MDC', the Simulink Design Verifier software enables every coverage objective for decision coverage and condition coverage. Enabling coverage for condition coverage causes every decision and condition coverage outcome to be enabled.</p> <p>This parameter is enabled when Mode is set to 'TestGeneration'.</p>	<p>'None'</p> <p>'Decision'</p> <p>'ConditionDecision' (default)</p> <p>'MDC'</p>

Parameter	Description	Values
ModelReferenceHarness	Use a Model block to reference the model to run in the harness model.	'on' 'off' (default)
OutputDir	Specify a path name to which the Simulink Design Verifier software writes its output.	character array 'sldv_output/\$ModelName\$' '\$' (default)
Parameters	Specify whether the software uses parameter configurations when analyzing a model. When set to 'on', this parameter enables ParametersConfigFileName.	'on' 'off' (default)

Parameter	Description	Values
ParametersConfigFileName	<p>Specify a MATLAB function that defines parameter configurations for a model.</p> <p>This parameter is enabled when Parameters is set to 'on'. This parameter is disabled when ParametersUseConfig is set to 'on'.</p> <p>When you configure the software to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with ExistingTestFile and apply parameter configurations with ParametersConfigFileName or the Parameter Configuration table, in analysis, the software reports that your model is incompatible. This error occurs because the existing test cases do not include corresponding parameter values.</p>	<p>character array</p> <p>'sldv_params_template.m' (default)</p>

Parameter	Description	Values
ParametersUseConfig	<p>Specify to use the Parameter Configuration table to define parameter configurations for a model.</p> <p>When set to 'on', this parameter disables ParametersConfigFileName.</p> <p>When you configure the software to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with ExistingTestFile and apply parameter configurations with ParametersConfigFileName or the Parameter Configuration table, in analysis, the software reports that your model is incompatible. This error occurs because the existing test cases do not include corresponding parameter values.</p>	'on' 'off' (default)
ProofAssumptions	Specify whether Proof Assumption blocks in your model are enabled or disabled.	'EnableAll' 'DisableAll' 'UseLocalSettings' (default)
ProvingStrategy	Specify the strategy for proving properties.	'FindViolation' 'Prove' (default) 'ProveWithViolation-Detection'

Parameter	Description	Values
RandomizeNoEffectData	Specify whether to use random values instead of zeros for input signals that have no impact on test or proof objectives. This parameter is enabled when SaveDataFile is set to 'on'.	'on' 'off' (default)
ReduceRationalApprox	Specify whether to run additional analysis to reduce instances of rational approximation.	'on' (default) 'off'
RelativeTolerance	Specify a relative value for tolerance to be used in relational boundary tests.	double { '0.01' }
ReportFileName	Specify a folder and file name for the analysis report. This parameter is enabled when SaveReport is set to 'on'.	character array '\$ModelName\$_report' (default)
ReportIncludeGraphics	Includes screen shots of properties in the report. Valid only in property-proving mode. This parameter is enabled when SaveReport is set to 'on' and Mode is set to 'PropertyProving'.	'on' 'off' (default)
SaveDataFile	Save the test data that the analysis generates to a MAT-file. When set to 'on', this parameter enables DataFileName, SaveExpectedOutput, and RandomizeNoEffectData.	'on' (default) 'off'

Parameter	Description	Values
SaveExpectedOutput	<p>Simulate the model by using test case signals and include the output values in the Simulink Design Verifier data file.</p> <p>This parameter is enabled when SaveDataFile is set to 'on'.</p>	'on' 'off' (default)
SaveHarnessModel	<p>Create a harness model generated by the Simulink Design Verifier analysis.</p> <p>When SaveReport is set to 'on', this parameter must also be set to 'on'.</p> <p>When set to 'on', this parameter enables HarnessModelFileName.</p>	'on' 'off' (default)
SaveReport	<p>Generate and save a Simulink Design Verifier report.</p> <p>When this parameter is set to 'on', SaveHarnessModel must also be set to 'on'.</p> <p>When set to 'on', this parameter enables ReportFileName, ReportIncludeGraphics, and DisplayReport.</p>	'on' 'off' (default)
SFcnSupport	<p>Enable support for S-functions that have been compiled to be compatible with Simulink Design Verifier. See “Support Limitations and Considerations for S-Functions and C/C++ Code”.</p>	'on' (default) 'off'

Parameter	Description	Values
SFcnExtraOptions	Extra options for analyzing S-functions that have been compiled to be compatible with Simulink Design Verifier. For example, defaultArraySize = 512. See "Support Limitations and Considerations for S-Functions and C/C++ Code".	character array ' ' (default)
TestConditions	Specify whether Test Condition blocks in your model are enabled or disabled. This parameter is enabled when Mode is set to 'TestGeneration'.	'EnableAll' 'DisableAll' 'UseLocalSettings' (default)
TestgenTarget	Specify the test generation target as model, code generated as top model, or code generated as model reference.	character array 'Model' (default) 'GeneratedCode' 'GeneratedModelReferenceCode'
TestObjectives	Specify whether Test Objective blocks in your model are enabled or disabled. This parameter is enabled when Mode is set to 'TestGeneration'.	'EnableAll' 'DisableAll' 'UseLocalSettings' (default)

Parameter	Description	Values
TestSuiteOptimization	<p>Specify the optimization strategy to use when generating test cases.</p> <p>This parameter is enabled when Mode is set to 'TestGeneration'.</p>	<p>'CombinedObjectives (Nonlinear Extended)' (default)</p> <p>'IndividualObjectives'</p> <p>'LargeModel'</p> <p>'LongTestCases'</p> <p>'CombinedObjectives'</p> <p>'LargeModel (Nonlinear Extended)'</p>

Alternatives

In the Simulink Editor, select **Analysis > Design Verifier > Options** to set the Simulink Design Verifier analysis options.

See Also

sldvblockreplacement | sldvcompat | sldvgencov | sldvrun

Topics

“Simulink Design Verifier Options”

Introduced in R2007a

sldv.prove

Proof objective function for Stateflow charts and MATLAB Function blocks

Syntax

```
sldv.prove(expr)
```

Description

`sldv.prove(expr)` specifies that `expr` be true for every evaluation while proving properties. Use any valid Boolean expression for `expr`.

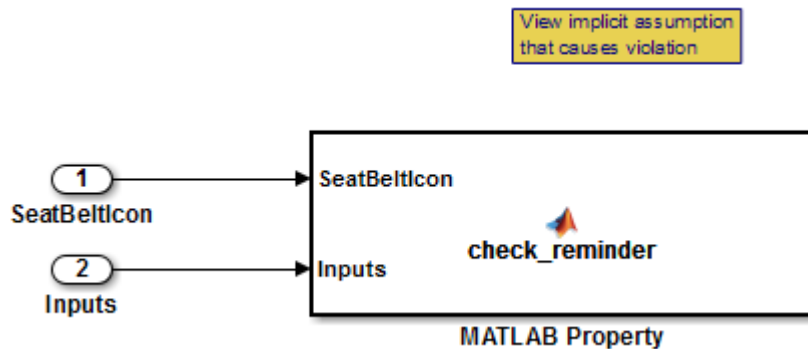
This function has no output and no impact on its parenting function, other than any indirect side effects of evaluating `expr`. If you issue this function from the MATLAB command line, the function has no effect.

Intersperse `sldv.prove` proof assumptions within code or separate the assumptions into a verification script.

Examples

Specify a property proof objective and proof assumption in a MATLAB Function block:

- 1 Open the `sldvdemo_sbr_verification` model and save it as `ex_sldvdemo_sbr_verification`.
- 2 Open the Safety Properties subsystem.



- 3 Open the **MATLAB Property** block, which is a MATLAB Function block.

```

Editor - Block: sldvdemo_sbr_verification/Safety Properties/MATLAB Property
Safety Properties/MATLAB Property x
1  function check_reminder(SeatBeltIcon,Inputs) %#codegen
2      % The seat belt light should be active whenever the key is turned on
3      % and speed is less than 15 and the seatbelt is not fastened
4      activeCond = ((Inputs.KEY ~= 0) && (Inputs.SeatBeltFasten == 0) && ...
5                  (Inputs.Speed < 15));
6
7      sldv.prove(implies(activeCond, SeatBeltIcon));
8
9      function out = implies(cond, result)
10         if (cond)
11             out = result;
12         else
13             out = true;
14         end
15

```

- 4 At the end of the `check_reminder` function definition, add the line `sldv.assume(Inputs.KEY==0 | 1);` so that the last two lines of the function definition now read:

```

sldv.prove(implies(activeCond, SeatBeltIcon));
sldv.assume(Inputs.KEY==0 | 1);

```


- 5 In the editor, save the updated code.
- 6 Prove the safety properties. With the model open in the Simulink Editor, select the Safety Properties subsystem and choose **Analysis > Design Verifier > Prove Properties > Selected Subsystem**.

In the Simulink Editor, you can also right-click the Safety Properties subsystem and select **Design Verifier > Prove Subsystem Properties**.

Alternatives

Instead of using the `sldv.prove` function, you can insert a Proof Objective block in your model.

However, using `sldv.prove` instead of a Proof Objective block offers several benefits, described in “What Is Property Proving?”.

You can also specify a proof objective by using MATLAB for code generation without using the `sldv.prove` function. Using `sldv.prove` instead of directly using MATLAB for code generation eliminates the need to:

- Express the objective with a Simulink block
- Explicitly connect the proof output to a Simulink block

See Also

[Proof Assumption](#) | [Proof Objective](#) | [Test Condition](#) | [Test Objective](#) | [sldv.condition](#) | [sldv.prove](#) | [sldv.test](#)

Topics

“Prove Properties in a Model”

“Workflow for Proving Model Properties”

Introduced in R2009b

sldvreport

Generate report

Syntax

```
[status, reportFilePath] = sldvreport(sldvDataFile)
[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1,
reportOption2, ...})
[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1,
reportOption2, ...}, reportFilePath)
[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1,
reportOption2, ...}, reportFilePath, showUI)
[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1,
reportOption2, ...}, reportFilePath, showUI, FORMAT)
```

Description

[status, reportFilePath] = sldvreport(sldvDataFile) generates a complete HTML report from the data in sldvDataFile. status returns true if sldvreport created the report. reportFilePath contains the actual name of the HTML report created.

[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1, reportOption2, ...}) generates a complete HTML report from the data in sldvDataFile based on the specified options. options is a cell array.

[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1, reportOption2, ...}, reportFilePath) generates a complete HTML report from the data in sldvDataFile based on the specified options and saves it in the location reportFilePath.

[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1, reportOption2, ...}, reportFilePath, showUI) generates a complete HTML report from the data in sldvDataFile based on the specified options and saves it in the location reportFilePath. Also displays the status of the report generation in a UI if showUI is true.

[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1, reportOption2, ...}, reportFilePath, showUI, FORMAT) generates a complete report in the specified FORMAT from the data in sldvDataFile based on the specified options and saves it in the location reportFilePath. Also displays the status of the report generation in a UI if showUI is true.

Input Arguments

sldvDataFile — Data file with analysis results

character vector | string scalar

Name of the data file that contains the analysis results

options — Options for the report

character vector | cell array of character vectors | string array

Cell array that specifies options for the report:

'summary'	Include summary analysis data only (Default: false)
'objectives'	Include test objective data (Default: true)
'objects'	Include data about all model objects (Default: true)
'testcases'	Include data about all generated test cases (Default: true)
'properties'	Include data about all properties proven or falsified (Default: true)

reportFilePath — Generated report

character vector | string scalar

The path and file name for the generated report

showUI — Display messages

logical

Logical value indicating where to display messages during analysis
true to display messages in the log window

false (**default**) to display messages in the MATLAB command window

FORMAT — Generate report format

'HTML' | 'PDF' | {'HTML', 'PDF'}

Entry indicating whether to generate the report in HTML format, PDF format, or in both formats.

'HTML' (**default**) to generate an HTML version of the report

'PDF' to generate a PDF version of the report

{'HTML', 'PDF'} to generate both an HTML version and a PDF version of the report

This parameter is case sensitive. Use only capital letters for this parameter.

Output Arguments

status

true if `sldvreport` creates the report, otherwise false.

reportFilePath

The path and file name for the generated HTML report

Examples

Analyze the model and create a PDF version of the report using `sldvreport`:

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.SaveReport = 'off';
open_system('sldvdemo_cruise_control');
[status, files] = sldvrun('sldvdemo_cruise_control', opts);
[status, reportFilePath] = sldvreport(files.DataFile, ...
    {'summary', 'objectives'}, 'C:\work\sldvdemo_cruise_control_report', false, 'PDF');
```

Alternatives

The Simulink Design Verifier software can create an HTML report after analyzing a model. In the Configuration Parameters dialog box, in the **Design Verifier > Report**

pane, select **Generate report of the results**. If you want to save an additional PDF version of the report, select **Generate additional report in PDF format**.

See Also

sldvoptions | sldvrun

Introduced in R2009b

sldvrun

Analyze model

Syntax

```
status = sldvrun
status = sldvrun(model)
status = sldvrun(subsystem)
status = sldvrun(model, options)
[status, filenames] = sldvrun(model, options)
[status, filenames] = sldvrun(model, options, showUI, startCov)
```

Description

`status = sldvrun` analyzes the current model to generate the test cases that provide the model coverage or prove the model properties.

`status = sldvrun(model)` analyzes `model` to generate the test cases that provide the model coverage or prove the model properties

`status = sldvrun(subsystem)` converts the atomic subsystem `subsystem` into a new model and runs a design verification analysis on the new model.

`status = sldvrun(model, options)` analyzes `model` by using the `sldvoptions` object options.

`[status, filenames] = sldvrun(model, options)` analyzes `model` and returns the filenames that the software creates during the analysis.

`[status, filenames] = sldvrun(model, options, showUI, startCov)` opens the log window during the analysis if you set `showUI` to `true`. If you set `showUI` to `false` (the default), `sldvrun` directs output to the MATLAB command line.

Examples

Analyze the Model by Using Analysis Options

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

Set `sldvoptions` parameters:

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';           % Perform test-generation analysis
opts.ModelCoverageObjectives = 'MCDC';  % MCDC coverage
opts.SaveHarnessModel = 'off';          % Don't save harness as model file
opts.SaveReport = 'on';                 % Save the HTML report
```

Open the `sldvdemo_cruise_control` model and analyze the model by using the specified options:

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

- “Generate Test Cases for Model Decision Coverage”
- “Prove Properties in a Model”

Input Arguments

model — Name or handle of a model

[] (default) | character vector | string scalar

Handle to a Simulink model.

subsystem — Name or handle of an atomic subsystem

[] (default) | character vector | string scalar

Handle to an atomic subsystem in a Simulink model.

options — Analysis options

[] (default) | character vector | string scalar

`sldvoptions` object that specifies the analysis options.

showUI — Display messages during analysis

logical

Logical value indicating where to display the messages during the analysis:

true to display the messages in the log window.
false (default) to display the messages in the Command Window.

startCov — Model coverage objects

[] (default) | character vector | string scalar

cvdata object that specifies the model coverage objects for the software to ignore.

Output Arguments

filenames — Save data to file

character vector

A structure whose fields list the file names that the Simulink Design Verifier software generates.

DataFile	MAT-file with raw input data
HarnessModel	Simulink harness model
Report	HTML report of the simulation results
ExtractedModel	Simulink model extracted from subsystem
BlockReplacementModel	Simulink model obtained after block replacements

status — Analysis status

character vector

These values that list the status of the analysis.

-1	Analysis exceeded the maximum processing time
0	Error
1	Preprocessing completed normally

Alternatives

To run a Simulink Design Verifier analysis, in the Model Editor window, select one of the following:

- **Analysis > Design Verifier > Detect Design Errors**
- **Analysis > Design Verifier > Generate Tests**
- **Analysis > Design Verifier > Prove Properties**

See Also

sldvcompat | sldvgencov | sldvoptions

Topics

“Generate Test Cases for Model Decision Coverage”

“Prove Properties in a Model”

Introduced in R2007a

sldvruncgvtest

Invoke Code Generation Verification (CGV) API and execute model

Syntax

```
cgvObject = sldvruncgvtest(model, dataFile)
cgvObject = sldvruncgvtest(model, dataFile, runOpts)
```

Description

`cgvObject = sldvruncgvtest(model, dataFile)` invokes the Code Generation Verification (CGV) API methods and executes the `model` by using all the test cases in `dataFile`. `cgvObject` is a `cgv.CGV` object that `sldvruncgvtest` creates during the execution of the `model`. `sldvruncgvtest` sets the execution mode for `cgvObject` to 'sim' by default.

`cgvObject = sldvruncgvtest(model, dataFile, runOpts)` invokes CGV API methods and executes the `model` by using the test cases in `dataFile`. `runOpts` defines the options for executing the test cases. The settings in `runOpts` determine the configuration of `cgvObject`.

Examples

Invoke CGV API and Execute the Test Cases

Create the default configuration object for `sldvruncgvtest` and execute the specified test cases on the generated code for the model.

Open the `sldemo_mdhref_basic` example model and log the input signals to the Counter A Model block.

```
open_system('sldemo_mdhref_basic');
load_system('sldemo_mdhref_counter');
loggedData = sldvlogs('sldemo_mdhref_basic/CounterA');
```

Create the default configuration object for `sldvruncgvtest` and allow the model to be configured to execute test cases with the CGV API.

```
run0pts = sldvruntestopts('cgv');
run0pts.allowCopyModel = true;
```

To invoke the CGV API and execute the specified test cases on the generated code for the model, use the logged signals, execute `sldvruncgvtest`—first in the simulation mode, and then in software-in-the-loop (SIL) mode— to invoke the CGV API and execute the specified test cases on the generated code for the model.

```
cgvObjectSim = sldvruncgvtest('sldemo_md1ref_counter', loggedData, run0pts);
run0pts.cgvConn = 'sil';
cgvObjectSil = sldvruncgvtest('sldemo_md1ref_counter', loggedData, run0pts);
```

Use the CGV API to compare the results of the first test case.

```
simout = cgvObjectSim.getOutputData(1);
silout = cgvObjectSil.getOutputData(1);
[matchNames, ~, mismatchNames, ~] = cgv.CGV.compare(simout, silout);
fprintf('\nTest Case: %d Signals match, %d Signals mismatch', ...
        length(matchNames), length(mismatchNames));
```

Input Arguments

model — Name or handle of model

character vector | string scalar

Name or handle of the Simulink model to execute.

dataFile — Name of data file or input data

character vector | string scalar

Name of the data file or a structure that contains the input data. You can generate the data by either of these methods:

- Analyzing the model by using the Simulink Design Verifier software.
- Using the `sldvlogsignals` function.

run0pts — configuration parameters

structure

A structure whose fields specify the configuration of `sldvruncgvtest`.

Field Name	Description
testIdx	Test case index array to execute from <code>dataFile</code> . If <code>testIdx</code> is [], <code>sldvruncgvtest</code> executes all test cases in <code>dataFile</code> . Default: []
allowCopyModel	If you have not configured the model, specifies to create and configure the model to execute test cases with the CGV API. If <code>true</code> and you have not configured <code>model</code> to execute test cases with the CGV API, <code>sldvruncgvtest</code> copies the model, fixes the configuration, and executes the test cases on the copied model. If <code>false</code> (the default), an error occurs if the tests cannot execute with the CGV API. Note If you have not configured the top-level model or any referenced models to execute test cases, <code>sldvruncgvtest</code> does not copy the model, even if <code>allowCopyModel</code> is <code>true</code> . An error occurs.
cgvCompType	Defines the software-in-the-loop (SIL) or processor-in-the-loop (PIL) approach for CGV: <ul style="list-style-type: none">• 'topmodel' (default)• 'modelblock'
cgvConn	Specifies mode of execution for CGV: <ul style="list-style-type: none">• 'sim' (default)• 'sil'• 'pil'

Note `runOpts = sldvruntestopts('cgv')` returns a `runOpts` structure with the default values for each field.

Output Arguments

cgvObject — **cgv.CGV object**
object

cgv.CGV object that sldvruncgvtest creates during the execution of model.

sldvruncgvtest saves the following data for each test case executed in an array of Simulink.SimulationOutput objects inside cgvObject.

Field	Description
tout_sldvruncgvtest	Simulation time
xout_sldvruncgvtest	State data
yout_sldvruncgvtest	Output signal data
logout_sldvruncgvtest	Signal logging data for: <ul style="list-style-type: none"> • Signals connected to outports • Signals that are configured for logging on the model

Tips

To run sldvruncgvtest, you must have Embedded Coder®.

If your model has parameters that are not configured for executing test cases with the CGV API, sldvruncgvtest reports warnings about the invalid parameters. If you see these warnings, do one of the following:

- Modify the invalid parameters and rerun sldvruncgvtest.
- Set allowCopyModel in runOpts to be true and rerun sldvruncgvtest. sldvruncgvtest makes a copy of your model with the same configuration and invokes the CGV API.

See Also

cgv.CGV | sldvlogsignals | sldvruncgvtest | sldvruncgvtestopts

Topics

“Verify a Component for Code Generation”

“Creating and Executing Test Cases”

Introduced in R2010b

sldvruntest

Simulate model using input data

Syntax

```
outData = sldvruntest(model, dataFile)
outData = sldvruntest(model, dataFile, runOpts)
[outData, covData] = sldvruntest(model, dataFile, runOpts)
```

Description

`outData = sldvruntest(model, dataFile)` simulates `model` using all the test cases in `dataFile`. `outData` is an array of `Simulink.SimulationOutput` objects. Each array element contains the simulation output data of the corresponding test case.

`outData = sldvruntest(model, dataFile, runOpts)` simulates `model` using all the test cases in `dataFile`. `runOpts` defines the options for simulating the test cases.

`[outData, covData] = sldvruntest(model, dataFile, runOpts)` simulates `model` using the test cases in `dataFile`. When the `runOpts` field `coverageEnabled` is `true`, the Simulink Coverage™ software collects model coverage information during the simulation. `sldvruntest` returns the coverage data in the `cvdata` object `covData`.

Input Arguments

model — Name of the model

character vector | string scalar

Name or handle of the Simulink model to simulate

dataFile — Name of data file

character vector | string scalar

Name of the data file or structure that contains the input data. You can generate `dataFile` using the Simulink Design Verifier software, or by running the `sldvlogsignals` function.

runOpts — Configurations for sldvruntest

character vector | cell array of character vectors | string array

A structure whose fields specify the configuration of `sldvruntest`.

Field	Description
<code>testIdx</code>	Test case index array to simulate from <code>dataFile</code> . If <code>testIdx</code> is <code>[]</code> , <code>sldvruntest</code> simulates all test cases. Default: <code>[]</code>
<code>coverageEnabled</code>	If <code>true</code> , specifies that the Simulink Coverage software collect model coverage data during simulation. Default: <code>false</code>
<code>coverageSetting</code>	<code>cvtest</code> object for collecting model coverage. If <code>[]</code> , <code>sldvruntest</code> uses the existing coverage settings for <code>model</code> . Default: <code>[]</code>

Note `runOpts = sldvruntestopts` returns a `runOpts` structure with the default values for each field.

Output Arguments

outData

An array of `Simulink.SimulationOutput` objects that simulating the test cases generates. Each `Simulink.SimulationOutput` object has the following fields.

Field Name	Description
tout_sldvrntest	Simulation time
xout_sldvrntest	State data
yout_sldvrntest	Output signal data
logout_sldvrntest	Signal logging data for: <ul style="list-style-type: none"> • Signals connected to outports • Signals that are configured for logging on the model

covData

cvdata object that contains the model coverage data collected during simulation.

Examples

Analyze the `sldvdemo_cruise_control` model. Using data from the three test cases in the test suite, simulate the model. Use the Simulation Data Inspector to examine the signal logging data from the three test cases:

```

opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'off';
open_system('sldvdemo_cruise_control');
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
runOpts = sldvrntestopts;
[ outData ] = sldvrntest('sldvdemo_cruise_control',...
    files.DataFile, runOpts);
Simulink.sdi.createRun('Test Case 1 Output', 'namevalue',...
    {'output'}, {outData(1).find('logout_sldvrntest')});
Simulink.sdi.createRun('Test Case 2 Output', 'namevalue',...
    {'output'}, {outData(2).find('logout_sldvrntest')});
Simulink.sdi.createRun('Test Case 3 Output', 'namevalue',...
    {'output'}, {outData(3).find('logout_sldvrntest')});
Simulink.sdi.view;

```

Tips

The `dataFile` that you create with a Simulink Design Verifier analysis or by running `sldvlogssignals` contains time values and data values. When you simulate a model using these test cases, you might see missing coverage. This issue occurs when the time values in the `dataFile` are not aligned with the current simulation time step due to numeric calculation differences. You see this issue more frequently with multirate models —models that have multiple sample times.

See Also

`cvsim` | `cvtest` | `sim` | `sldvrun` | `sldvruntimeopts`

Introduced in R2007b

sldvruntestopts

Generate simulation or execution options for `sldvruntest` or `sldvruncgvttest`

Syntax

```
run0pts = sldvruntestopts  
run0pts = sldvruntestopts('cgv')
```

Description

`run0pts = sldvruntestopts` generates a `run0pts` structure for `sldvruntest`.

`run0pts = sldvruntestopts('cgv')` generates a `run0pts` structure for `sldvruncgvttest`.

Input Arguments

cgv — Default configuration options

character vector | string scalar

Generates a default `run0pts` structure for `sldvruncgvttest`.

Output Arguments

run0pts

A structure whose fields specify the configuration of `sldvruntest` or `sldvruncgvttest`. `run0pts` can have the following fields. If you do not specify a field, `sldvruncgvttest` or `sldvruntest` uses the default value.

Field Name	Description
testIdx	<p>Test case index array to simulate or execute from dataFile.</p> <p>If testIdx = [], all test cases will be simulated or executed.</p>
outputFormat	<p>Specifies format of output values:</p> <ul style="list-style-type: none">• 'TimeSeries' (default) — sldvrntest/sldvrncgvtest stores the output values in time-series format.• 'StructureWithTime' — sldvrntest/sldvrncgvtest stores the output values in the Structure with time format.
coverageEnabled	<p>Available only for sldvrntest.</p> <p>If true, the Simulink Coverage software collects model coverage data during simulation.</p> <p>Default: false</p>
coverageSetting	<p>Available only for sldvrntest.</p> <p>cvtest object to use for collecting model coverage.</p> <p>If coverageSetting is [], sldvrntestopts returns the coverage settings for the model specified in the call to sldvrntest.</p> <p>Default: []</p>

Field Name	Description
allowCopyModel	<p>Available only for sldvruncgvtest.</p> <p>Specifies to create and configure the model if you have not configured it to execute test cases with the CGV API.</p> <p>If <code>true</code> and you have not configured the <code>model</code> to execute test cases with the CGV API, <code>sldvruncgvtest</code> copies the model, fixes the configuration, and executes the test cases on the copied model.</p> <p>If <code>false</code> (the default), an error occurs if the tests cannot execute with the CGV API.</p> <hr/> <p>Note If you have not configured the top-level model or any referenced models to execute test cases, <code>sldvruncgvtest</code> does not copy the model, even if <code>allowCopyModel</code> is <code>true</code>. An error occurs.</p>
cgvComType	<p>Available only for sldvruncgvtest.</p> <p>Defines the software-in-the-loop (SIL) or processor-in-the-loop (PIL) approach for CGV:</p> <ul style="list-style-type: none"> • 'topmodel' (default) • 'modelblock'
cgvConn	<p>Available only for sldvruncgvtest.</p> <p>Specifies mode of execution for CGV:</p> <ul style="list-style-type: none"> • 'sim' (default) • 'sil' • 'pil'

Examples

Create `runOpts` objects for `sldvruntest` and `sldvruncgvtest`:

```
runtest_options = sldvruntestopts;           ! sldvruntest  
runcgvtest_options = sldvruntestopts('cgv') ! sldvruncgvtest
```

Alternatives

Create a `runOpts` object for `sldvruntest` at the MATLAB command line.

See Also

`sldvruncgvtest` | `sldvruntest`

Introduced in R2010b

sldvsimdata

Get simulation data in Dataset format

Syntax

```
[simData,params] = sldvsimdata(dataFile)
[simData,params] = sldvsimdata(dataFile,index)
[simData,params] = sldvsimdata(data)
[simData,params] = sldvsimdata(data,index)
```

Description

[simData,params] = sldvsimdata(dataFile) returns Simulink.SimulationData.Dataset object `simData`, containing simulation data, and structure array `params`, containing parameter values, from Simulink Design Verifier data file `dataFile`. The elements of `simData` and `params` correspond to each test case or counterexample in `dataFile`.

[simData,params] = sldvsimdata(dataFile,index) returns Simulink.SimulationData.Dataset object `simData`, containing simulation data, and structure array `params`, containing parameter values, for the test case or counterexample represented by integer `index` in Simulink Design Verifier data file `dataFile`.

[simData,params] = sldvsimdata(data) returns Simulink.SimulationData.Dataset object `simData`, containing simulation data, and structure array `params`, containing parameter values, from Simulink Design Verifier data variable `data`.

[simData,params] = sldvsimdata(data,index) returns Simulink.SimulationData.Dataset object `simData`, containing simulation data, and structure array `params`, containing parameter values, for the test case or counterexample represented by integer `index` in the Simulink Design Verifier data variable `data`.

Input Arguments

dataFile — simulation data file

character vector | string scalar

Simulink Design Verifier data file. For more information, see “Simulink Design Verifier Data Files”.

data — simulation data variable

character vector | string scalar

Simulink Design Verifier data variable.

index — index of test case or counterexample in data file

integer

Index of test case or counterexample in data file, specified as an integer.

Output Arguments

simData — Dataset object containing simulation data

object

Simulation data, returned as `Simulink.SimulationData.Dataset` object.

params — model configuration parameters

structure array

Model parameters, returned as a structure array.

See Also

`Simulink.SimulationData.Dataset` | `sldvlogsignals` | `sldvruntime`

Introduced in R2014b

sldv.test

Test objective function for Stateflow charts and MATLAB Function blocks

Syntax

```
sldv.test(expr)
```

Description

`sldv.test(expr)` Specifies that `expr` should be made true when generating tests. Use any valid Boolean expression for `expr`.

This function has no output and no impact on its parenting function, other than any indirect side effects of evaluating `expr`. If you issue this function from the MATLAB command line, the function has no effect.

Intersperse `sldv.test` test objectives within code or separate the objectives into a verification script.

The **Test objectives** option in the **Test generation** pane applies to test objectives represented with the `sldv.test` function, as well as with the Test Objective block.

Examples

Add a test objective and test conditions:

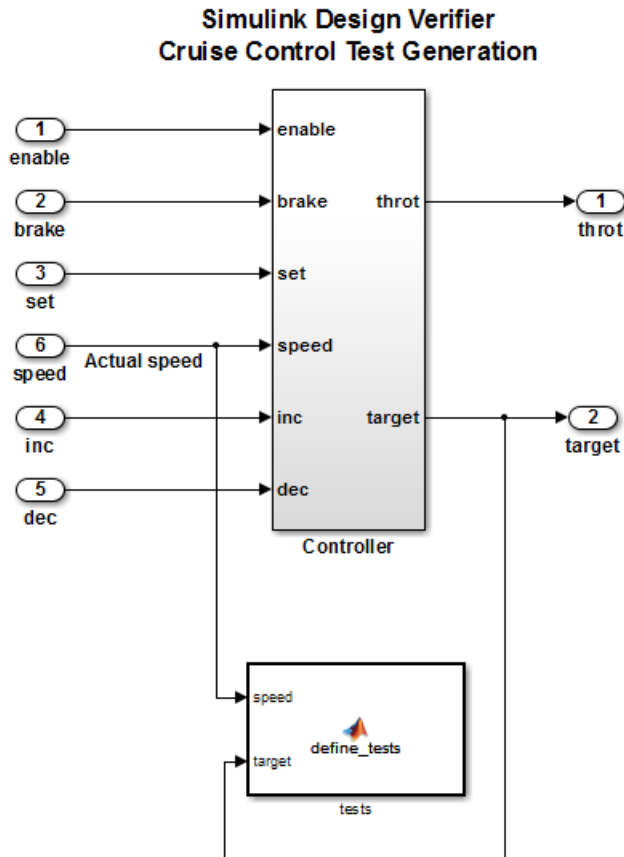
- 1 Open the `sldvdemo_cruise_control` model and save it as `ex_sldvdemo_cruise_control`.
- 2 Remove the Test Condition block for the `speed` block signal. Instead of the Test Condition block, this example uses `sldv.test` and `sldv.condition`.
- 3 From the User-Defined Functions library, add a MATLAB Function block and:
 - a Name the block `tests`.

- b** Open the block and add the following code:

```
function define_tests(speed, target)
%#codegen

sldv.condition(speed >= 0 && speed <= 100);
sldv.test(speed > 60 && target > 40 && target < 50);
sldv.test(speed < 20 && target > 50);
```

- c** Save the code and close the editor.
- d** Connect the block to the signal for the speed block and to the signal for the target block.



- 4 Generate the test: select **Analysis > Design Verifier > Generate Tests > Model**.

Alternatives

Instead of using the `sldv.test` function, you can insert a Test Objective block in your model.

However, using `sldv.test` instead of a Test Objective block offers several benefits, described in “What Is Test Case Generation?”.

See Also

[Proof Assumption](#) | [Proof Objective](#) | [Test Condition](#) | [Test Objective](#) | [sldv.assume](#) | [sldv.condition](#) | [sldv.prove](#)

Topics

[“Generate Test Cases for Model Decision Coverage”](#)
[“Workflow for Test Case Generation”](#)

Introduced in R2009b

sldvtimer

Identify, change, and display timer optimizations

Syntax

```
status = sldvtimer
status = sldvtimer(value)
status = sldvtimer(sldvdata)
status = sldvtimer(sldvdata,display)
status = sldvtimer(model)
```

Description

`status = sldvtimer` returns a `status` of 1 if timer optimizations are enabled for Simulink Design Verifier test generation. Otherwise, `sldvtimer` returns a `status` of 0.

`status = sldvtimer(value)` enables or disables timer optimizations for Simulink Design Verifier test generation.

`status = sldvtimer(sldvdata)` indicates if timer optimizations are recorded in Simulink Design Verifier data file `sldvdata`. Returns a `status` of 1 if timer optimizations are recorded in Simulink Design Verifier data file `sldvdata`. Returns a `status` of 0 if timer optimizations are not recorded. Returns a `status` of -1 if `sldvdata` does not have information about timer optimizations.

`status = sldvtimer(sldvdata,display)` indicates if timer optimizations are recorded in Simulink Design Verifier data file `sldvdata` and identifies model items that are part of recognized timer patterns when `display` is true. Returns a `status` of 1 if timer optimizations are recorded in Simulink Design Verifier data file `sldvdata`. Returns a `status` of 0 if timer optimizations are not recorded. Returns a `status` of -1 if `sldvdata` does not have information about timer optimizations.

`status = sldvtimer(model)` displays timer patterns in the `model` that can be optimized for Simulink Design Verifier test generation.

Input Arguments

value — Enable timer optimizations

logical

Logical value to enable timer optimizations

`true` to enable timer optimizations

`false` (default) to disable timer optimizations

sldvdata — Name of data file

character vector | string scalar

Name of the data file that contains the timer optimization data.

display — Identify model objects in timer patterns

logical

Logical value to identify model objects that are part of recognized timer patterns

`true` to identify model objects that are part of recognized timer patterns

`false` (default) to not identify model objects that are part of recognized timer patterns

model — Handle to a model

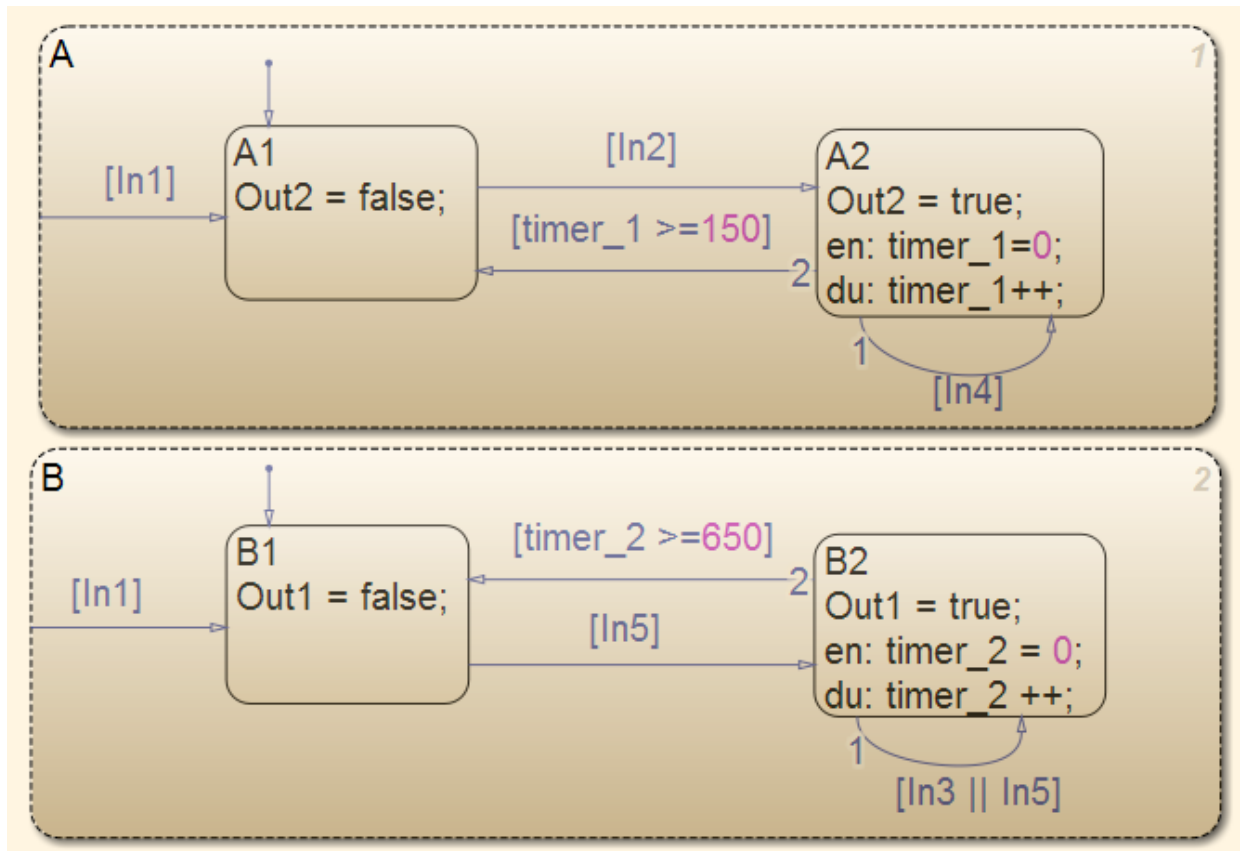
[] (default) | character vector | string scalar

Handle to a Simulink model

Examples



This example shows how to use the `sldvtimer` function to optimize model timers, increasing the number of test generation objectives met during Simulink Design Verifier Test Generation analysis.

- 1 The example model has timers `timer_1` and `timer_2` in a Stateflow chart.




2 Select **Analysis > Design Verifier > Generate Tests > Model**.

- The Simulink Design Verifier log dialog box reports:
 - Test generation exceeded time limit
 - 28 of 32 objectives satisfied
- The Simulink Design Verifier Errors information dialog box indicates that Test generation did not optimize timer patterns.

Message	Source	Reported By	Summary
 Design Verifier analysis error	ex_sldvtimer_control	simulink	Simulink Design Verifier has exceeded the maxi...
 Design Verifier analysis error	ex_sldvtimer_control	simulink	Test Generation did not optimize timer pattern...

.....

 ex_sldvtimer_control

Test Generation did not optimize timer patterns. This model contains timer patterns and you might get better results by enabling timer optimizations with executing command [sldvtimer\(1\)](#) in the MATLAB workspace and restarting Test Generation. Refer to the [sldvtimer](#) command for more information.

- 3 In the MATLAB Command Window, enter:

```
sldvtimer(1)
```
- 4 Select **Analysis > Design Verifier > Generate Tests > Model** to generate test cases again.

Limitations

If relational boundary objectives are included for test case generation, `sldvtimer` can optimize fewer timers. For information on relational boundary objectives, see “Relational Boundary”.

See Also

`sldvruncgvttest` | `sldvrunttest` | `sldvrunttestopts`

Introduced in R2012a

slslicer

Create API object for invoking Model Slicer

Syntax

```
slslicer(model)
slslicer(model,opts)
obj = slslicer(model)
```

Description

`slslicer(model)` creates an API object for the model `model` by exposing the methods for invoking Model Slicer. Uses the Model Slicer configurations associated with `model`, as defined by `slsliceroptions`.

`slslicer(model,opts)` creates an API object `model` `model` by using the options object `opts`, as defined by `slsliceroptions`.

`obj = slslicer(model)` creates a Model Slicer configuration object. You can apply the methods on the Model Slicer object `obj`.

Examples

Add Starting Point and Highlight the Model Slice

Add a new starting point to the active Model Slicer configuration, and then highlight the model.

Open the `sldvSliceClimateControlExample` example model.

```
addpath(fullfile(docroot,'toolbox','sldv','examples'));
open_system('sldvSliceClimateControlExample');
```

Create a Model Slicer configuration object for the model by using `slslicer`.


```
obj = slicer('sldvSliceClimateControlExample');
```

Activate the slice highlighting mode of Model Slicer to compile the model and prepare the model slice for dependency analysis.

```
activate(obj);
```

Add the Out1 output block as the starting point and highlight the model slice.

```
addStartingPoint(obj, 'sldvSliceClimateControlExample/Out1');
highlight(obj);
```

The area of the model upstream of the starting point and which is active during simulation is highlighted.

Terminate the model highlighting mode and discard the analysis data.

```
terminate(obj);
```

- “Programmatically Resolve Unexpected Behavior in a Model with Model Slicer”

Input Arguments

model — Name or handle of model

character vector | string scalar

Name of the model whose Model Slicer options object you configure. `slicer` uses the Model Slicer configurations associated with `model`, as defined by `sliceroptions`.

opts — Options you attach to a model or save to a file

structure

Structure containing the options for the Model Slicer configuration. `sliceroptions` defines the options object `opts`.

Output Arguments

obj — Model Slicer object

method

The table lists the methods that you use on a Model Slicer object.

Method	Description
activate	Activates the model for analysis. Example: <code>activate(obj)</code>
unlock	Discards the analysis data while retaining model highlights. Example: <code>unlock(obj)</code>
terminate	Discards the analysis data and reverts the model highlighting (invoked when the object goes out of scope). Example: <code>terminate(obj)</code>
highlight	Updates the model highlighting. Example: <code>highlight(obj)</code>
unhighlight	Removes the model highlighting without changing the activation status. Example: <code>unhighlight(obj)</code>
slice	Creates a sliced model from the model highlight. Example: <code>slice(obj, 'sldvSliceClimateControlExample_sliced')</code>
<code>simulate(t1,t2)</code>	Simulates a test case for dynamic slicing from time "t1" to time "t2". Example: <code>simulate(obj,0,30)</code>
ActiveBlocks	Returns the active nonvirtual block handles. Example: <code>ActiveBlocks(obj)</code>

Method	Description
addStartingPoint	<p>Adds the block handles, block paths, or Simulink Identifiers (SID) as the slice starting point.</p> <p>Example: <code>addStartingPoint(obj, 'sldvSliceClimateControlExample/Out1')</code></p>
removeStartingPoint	<p>Removes the starting point from the model slice.</p> <p>Example: <code>removeStartingPoint(obj, 'sldvSliceClimateControlExample/Out1')</code></p>
addExclusionPoint	<p>Adds the block handles, block paths, or Simulink Identifiers (SID) as the slice exclusion point.</p> <p>Example: <code>addExclusionPoint(obj, 'sldvSliceClimateControlExample/Refrigeration/On')</code></p>
removeExclusionPoint	<p>Removes the exclusion point from the model slice.</p> <p>Example: <code>removeExclusionPoint(obj, 'sldvSliceClimateControlExample/Refrigeration/On')</code></p>
addConstraint	<p>Adds the constraint on these blocks:</p> <ul style="list-style-type: none"> • Switch or Multiport switch • Stateflow state or transition <p>Example: <code>bpath={'sldvSliceClimateControlExample/Refrigeration/On'};</code> <code>addConstraint(obj,bpath,{1, 1})</code></p>
removeConstraint	<p>Removes the constraint from the model slice.</p> <p>Example: <code>removeConstraint(obj,bpath)</code></p>

Method	Description
addSliceComponent	Adds a model or a subsystem as a slice component. Example: addSliceComponent(obj, 'sldvdemo_cruise_control/Controller/PI Controller')
removeSliceComponent	Removes the slice component from the model slice. Example: removeSliceComponent(obj);
refineDeadLogic	Updates the model highlighting with dead logic refinement. Example: analysis_time=100; refineDeadLogic(obj, 'sldvSlicerdemo_dead_logic', analysis_time)
removeDeadLogic	Removes the dead logic refinement. Example: removeDeadLogic(obj, 'sldvSlicerdemo_dead_logic')

See Also

slsliceroptions | slslicertrace

Topics

“Programmatically Resolve Unexpected Behavior in a Model with Model Slicer”

“Workflow for Dependency Analysis”

“Configure Model Highlight and Sliced Models”

“Model Slicer Considerations and Limitations”

Introduced in R2015b

sliceroptions

Create an options object for configuring Model Slicer

Syntax

```
sliceroptions  
sliceroptions(model)  
sliceroptions(file)  
sliceroptions(model,opts)  
sliceroptions(file,opts)
```

Description

sliceroptions creates an options object for configuring Model Slicer.

sliceroptions(model) creates a copy of the Model Slicer options object associated with the model model.

sliceroptions(file) creates a copy of the Model Slicer options object contained in the SLMS file file.

sliceroptions(model,opts) attaches the slicer options opts to the model model, overwriting the existing options.

sliceroptions(file,opts) attaches the slicer options opts to the SLMS file file, overwriting the existing options.

Examples

Add Starting Points and Exclusion Points to Active Configuration

Add a new starting point and a new exclusion point to the active Model Slicer configuration.

Open the f14 example model.

```
open_system('f14')
```

Define the options file `opts` for the model.

```
opts = slsliceroptions('f14')
```

Add a new starting point on the Gain block.

```
addStartingPoint(opts, 'f14/Gain')
```

Add a new exclusion point on the alpha (rad) block.

```
addExclusionPoint(opts, 'f14/alpha (rad)')
```

Add Starting Points and Exclusion Points to New Configuration

Add a starting point and an exclusion point to the a new Model Slicer configuration without overwriting the original configuration.

Open the f14 example model.

```
open_system('f14')
```

Define the options file `opts` for the model.

```
opts = slsliceroptions('f14')
```

Create a second Model Slicer options configuration for the model.

```
addConfiguration(opts)
```

Add a new starting point on the Gain block for the second Model Slicer options configuration.

```
addStartingPoint(opts.Configuration(2), 'f14/Gain')
```

Add a new exclusion point on the alpha (rad) block for the second Model Slicer options configuration.

```
addExclusionPoint(opts.Configuration(2), 'f14/alpha (rad)')
```

Input Arguments

model — Name or handle of model

character vector | string scalar

Name of the model whose Model Slicer options object you configure.

file — Name of file

character vector | string scalar

Name of the SLMS file containing the Model Slicer options object you configure.

Example: `sliceroptions('f14.slms')`

opts — Options you attach to a model or save to a file

structure

Structure containing the options for the Model Slicer configuration.

See Also

`slicer` | `slicertrace`

Topics

“Workflow for Dependency Analysis”

“Configure Model Highlight and Sliced Models”

“Model Slicer Considerations and Limitations”

Introduced in R2015b

slslicertrace

Return block handles in sliced model or source model after using Model Slicer

Syntax

```
slslicertrace('slice',object)  
slslicertrace('source',object)
```

Description

`slslicertrace('slice',object)` returns the block handles in the sliced model that correspond to blocks specified by `object` in the source model.

`slslicertrace('source',object)` returns the block handles in the source model that correspond to blocks specified by `object` in the sliced model.

Examples

Highlight a Block in the Source Model

Highlight the Switch block in the `sldvSliceClimateControlExample` source model.

Open the `sldvSliceClimateControlExample` example model.

```
addpath(fullfile(docroot,'toolbox','sldv','examples'));  
open_system('sldvSliceClimateControlExample');
```

Create a slicer object `obj` and add `Out1` as the starting point.

```
obj = slslicer('sldvSliceClimateControlExample');  
activate(obj);  
addStartingPoint(obj,'sldvSliceClimateControlExample/Out1');  
highlight(obj)
```

Create a sliced model by using `slice`.


```
slice(obj, 'sldvSliceClimateControlExample_sliced')
```

Highlight the On Switch block in the source model by using `slicertrace`

```
h=slicertrace('SOURCE', 'sldvSliceClimateControlExample_sliced/Refrigeration/On');  
hilite_system(h);  
terminate(obj);
```

Input Arguments

object — Object in source model or sliced model

character vector | cell array of character vectors | string array

An `object` can be specified as an array of block handles, cell arrays of block paths, or cell arrays of Simulink Identifiers (SID).

See Also

`slicer` | `sliceroptions`

Topics

“Workflow for Dependency Analysis”

“Configure Model Highlight and Sliced Models”

“Model Slicer Considerations and Limitations”

Introduced in R2015b

Blocks — Alphabetical List

Detector

Detect true duration on input and construct output true duration based on output type



Library

Simulink Design Verifier

Temporal Operators Terminology

- True duration of a signal — Consecutive time steps during which a signal is true
- Length of the true duration of the signal — The number of time steps that constitute the true duration
- Input detection phase — The phase that is complete at the final time step of the expected length of the input true duration
- Output construction phase— The phase when the block constructs a true duration at the output based on the output type of the block
- Delay duration — The number of time steps of delay after input detection, after which the output signal is true

Description

The inputs and outputs of the Detector block are of Boolean type.

On input detection, the Detector block constructs an output signal based on one of the two output types that you specify:

- **Delayed Fixed Duration**—After the input detection is complete and after an optional delay, the output signal becomes `true` for a fixed number of time steps. The true duration of the output is independent of the input.
- **Synchronized**—In the final time step of the input detection, the output becomes `true` and stays `true` as long as the input signal continues to be `true`. The true duration of the output varies and is synchronized with the true duration of the input.

Parameters

External reset

Specify whether the block can be reset to the start of the input detection by an external Boolean reset signal.

Output type

Select **Delayed Fixed Duration** (the default) to specify a fixed true duration length for the output after an optional delay. Select **Synchronized** to synchronize the output true duration with that of the input.

Time steps for input detection

Length of the true duration for input detection (minimum is 1).

Time steps for delay (optional)

For **Delayed Fixed Duration**, optionally specify the length of the delay duration, after which the output becomes true.

Time steps for output duration

For **Delayed Fixed Duration**, specify the length of the output true duration (minimum is 1).

Examples

In the following examples, use a sample time of 1 second.

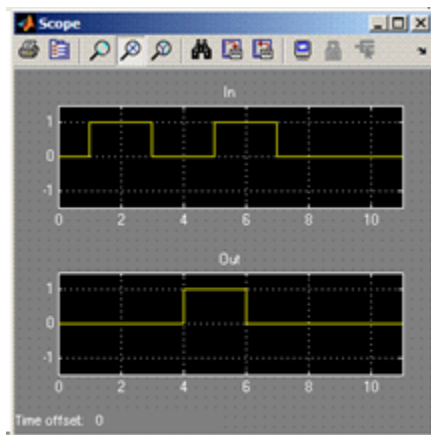
Delayed Fixed Duration

In this example, with **Output type** set to **Delayed Fixed Duration**, the input detection phase does not continue during the output signal construction. The following block parameters for the Detector block are set as follows:

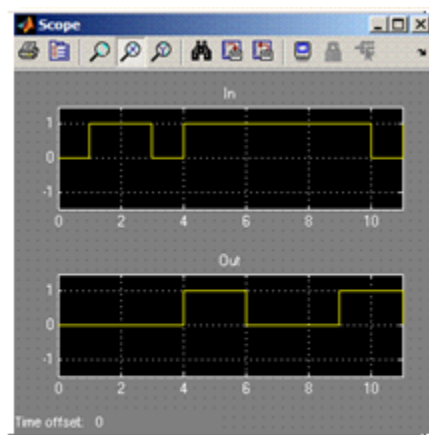
- **Time steps for input detection = 2**
- **Time steps for delay (optional) = 1**
- **Time steps for output duration = 2**

Scope 1 shows a scenario where the second `true` duration is not detected, because some of the `true` time steps occur during output construction.

However, the second `true` duration in Scope 2 is detected because the remaining `true` duration after the output construction satisfies the number of steps required for input detection.



Scope 1

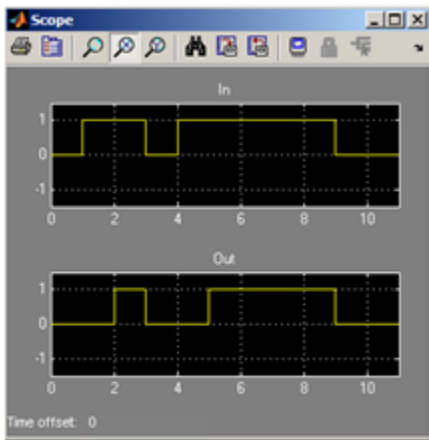


Scope 2

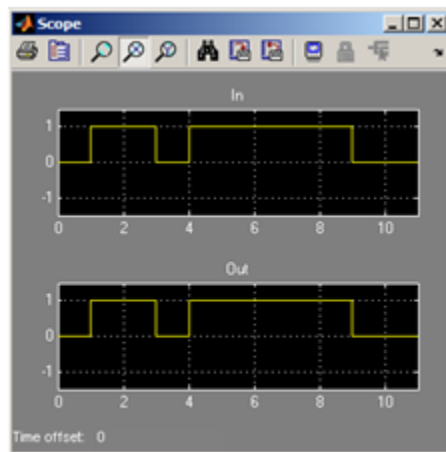
Synchronized

In this example, with the **Output type** set to Synchronized and **Time steps for input detection** set to 2, the output becomes `true` in the final step of input detection. The output continues to be true as long as the input signal is true.

Scope 1 shows that the output becomes true in the second time step, which is the final time step of the input detection phase. When the number of time steps for input detection is set to 1, the output is identical to the input, as you can see in Scope 2.



Scope 1



Scope 2

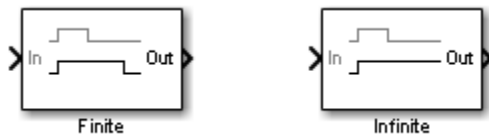
See Also

Extender, Within Implies

Introduced in R2011a

Extender

Extend true duration of input



Library

Simulink Design Verifier

Temporal Operators Terminology

- True duration of a signal — Consecutive time steps during which a signal is true

Description

The Extender block extends the true duration of the input signal by a fixed number of steps (finite extension mode) or indefinitely.

The inputs and outputs of the Extender block are of Boolean type.

Parameters

Extension Period

Select **Finite** (the default) to specify a fixed number of time steps for extension.
Select **Infinite** to specify indefinite extension.

Time steps for extension

For finite extension, specify the number of time steps for extending the true duration (minimum is 1).

External reset

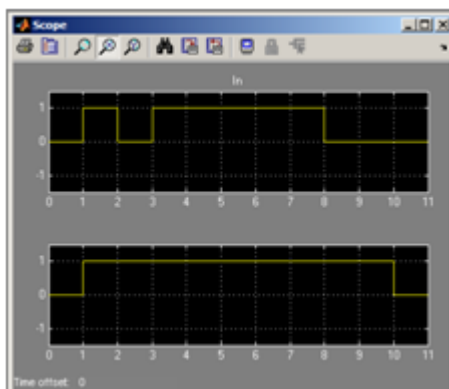
Specify whether an external Boolean reset signal can reset the block extension. The reset signal also resets the infinite extension. The infinite extension with an external reset is an indefinite extension until the external reset signal becomes `true`.

Examples

In the following example, do the following:

- Set the model sample time to 1 second.
- For the Extender block:
 - Set the **Extension Period** parameter to `Finite`.
 - Set the **Time steps for extension** parameter to 2

If the input signal becomes `true` during the extension period, the output continues to be `true` and is extended after the last input `true` duration is complete. You can see this in the following scope.



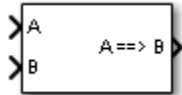
See Also

Detector, Within Implies

Introduced in R2011a

Implies

Specify condition that produces a certain response



Library

Simulink Design Verifier

Description

The Implies block lets you specify a condition to produce a given response; for example, when you press the brake pedal on a car, the cruise control mechanism becomes disabled. If input A is true and input B is false, the output is false; for all other pairs of inputs, the output is true.

You can use the Implies block in any model, not just when you run the Simulink Design Verifier software.

Introduced in R2009a

Proof Assumption

Constrain signal values when proving model properties



Library

Simulink Design Verifier

Description

When operating in property-proving mode, the Simulink Design Verifier software proves that properties of your model satisfy specified criteria (see “What Is Property Proving?”). In this mode, you can use Proof Assumption blocks to define assumptions for signals in your model. The **Values** parameter lets you specify constraints on signal values during a property proof. The block applies the specified **Values** parameter to its input signal, and the Simulink Design Verifier software proves or disproves that the properties of your model satisfy the specified criteria.

The block's parameter dialog box also allows you to:

- Enable or disable the assumption.
- Specify that the block should display its **Values** parameter in the Simulink Editor.
- Specify that the block should display its output port.

Note The Simulink and Simulink Coder™ software ignore the Proof Assumption block during model simulation and code generation, respectively. The Simulink Design Verifier software uses the Proof Assumption block only when proving model properties.

Specifying Proof Assumptions

Use the **Values** parameter to constrain signal values in property proofs. Specify any combination of scalars and intervals in the form of a MATLAB cell array. For information about cell arrays, see “Cell Arrays” (MATLAB).

Tip If the **Values** parameter specifies only one scalar value, you do not need to enter it in the form of a MATLAB cell array.

Scalar values each comprise a single cell in the array, for example:

```
{0, 5}
```

A closed interval comprises a two-element vector as a cell in the array, where each element specifies an interval endpoint:

```
{[1, 2]}
```

Alternatively, you can specify scalar values using the `Sldv.Point` constructor, which accepts a single value as its argument. You can 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, `Sldv.Interval` considers an interval to be closed if you omit its third input argument.

As an example, the **Values** parameter

```
{0, [1, 3]}
```

specifies:

- 0 — a scalar
- [1, 3] — a closed interval

The **Values** parameter

```
{Sldv.Interval(0, 1, '[]'), Sldv.Point(1)}
```

specifies:

- `Sldv.Interval(0, 1, '[]')` — the right-open interval $[0, 1)$
- `Sldv.Point(1)` — a scalar

If you specify multiple scalars and intervals for a Proof Assumption block, the Simulink Design Verifier software combines them using a logical OR operation during the property proof. In this case, the software considers the entire assumption to be satisfied if any single scalar or interval is satisfied.

Data Type Support

The Proof Assumption block accepts signals of all built-in data types supported by the Simulink software. For a discussion on the data types supported by the Simulink software, see “Data Types Supported by Simulink” (Simulink).

Parameters

Enable

Specify whether the block is enabled. If selected (the default), the Simulink Design Verifier software uses the block when proving properties of a model. Clearing this option disables the block, that is, causes the Simulink Design Verifier software to behave as if the Proof Assumption block did not exist. If this option is not selected, the block appears grayed out in the Simulink Editor.

Type

Specify whether the block behaves as a Proof Assumption or Test Condition block. Select **Test Condition** to transform the Proof Assumption block into a Test Condition block.

Values

Specify the proof assumption (see “Specifying Proof Assumptions” on page 2-11).

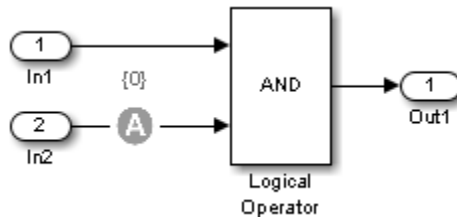
Display values

Specify whether the block displays the contents of its **Values** parameter in the Simulink Editor. By default, this option is selected.

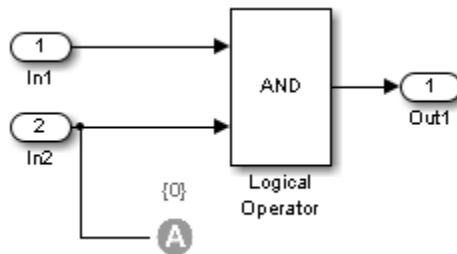
Pass through style (show Output)

Specify whether the block displays an output port in the Simulink Editor. If selected (the default), the block displays its output port, allowing its input signal to pass through as the block output. If not selected, the block hides its output port and terminates the input signal. The following graphics illustrate the appearance of the block in each case.

Pass through style (show Output): Selected



Pass through style (show Output): Deselected



See Also

Proof Objective, Test Condition

Introduced in R2007a

Proof Objective

Define objectives that signals must satisfy when proving model properties



Library

Simulink Design Verifier

Description

When operating in property-proving mode, the Simulink Design Verifier software proves that properties of your model satisfy specified criteria (see “What Is Property Proving?”). In this mode, you can use Proof Objective blocks to define proof objectives for signals in your model.

The **Values** parameter lets you specify acceptable values for the block's input signal. If a signal value deviates from the acceptable values in *any* time step, a property violation occurs and the proof objective is falsified. The block applies the specified **Values** parameter to its input signal, and the Simulink Design Verifier software proves or disproves that the properties of your model satisfy the specified criteria.

The block's parameter dialog box allows you to

- Enable or disable the objective.
- Specify that the block should display its **Values** parameter in the Simulink Editor.
- Specify that the block should display its output port.

Note The Simulink and Simulink Coder software ignore the Proof Objective block during model simulation and code generation, respectively. The Simulink Design Verifier software uses the Proof Objective block only when proving model properties.

Specifying Proof Objectives

Use the **Values** parameter to define values that a signal must achieve during a proof simulation. Specify any combination of scalars and intervals in the form of a MATLAB cell array. For information about cell arrays, see “Cell Arrays” (MATLAB).

Tip If the **Values** parameter specifies only one scalar value, you do not need to enter it in the form of a MATLAB cell array.

Scalar values each comprise a single cell in the array, for example:

```
{0, 5}
```

A closed interval comprises a two-element vector as a cell in the array, where each element specifies an interval endpoint:

```
{[1, 2]}
```

Alternatively, you can specify scalar values using the `Sldv.Point` constructor, which accepts a single value as its argument. You can 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, `Sldv.Interval` considers an interval to be closed if you omit its third input argument.

As an example, the **Values** parameter

```
{0, [1, 3]}
```

specifies:

- 0 — a scalar
- [1, 3] — a closed interval

The **Values** parameter

```
{Sldv.Interval(0, 1, '['), Sldv.Point(1)}
```

specifies:

- `Sldv.Interval(0, 1, '[')` — the right-open interval $[0, 1)$
- `Sldv.Point(1)` — a scalar

If you specify multiple scalars and intervals for a Proof Objective block, the Simulink Design Verifier software combines them using a logical OR operation during the property proof. In this case, the software considers the entire proof objective to be satisfied if any single scalar or interval is satisfied.

Data Type Support

The Proof Objective block accepts signals of all built-in data types supported by the Simulink software. For a discussion on the data types supported by the Simulink software, see “Data Types Supported by Simulink” (Simulink).

Parameters

Enable

Specify whether the block is enabled. If selected (the default), the Simulink Design Verifier software uses the block when proving properties of a model. Clearing this option disables the block, that is, causes the Simulink Design Verifier software to behave as if the Proof Objective block did not exist. If this option is not selected, the block appears grayed out in the Simulink Editor.

Values

Specify the proof objective (see “Specifying Proof Objectives” on page 2-15).

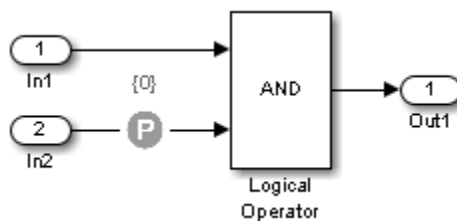
Display values

Specify whether the block displays the contents of its **Values** parameter in the Simulink Editor. By default, this option is selected.

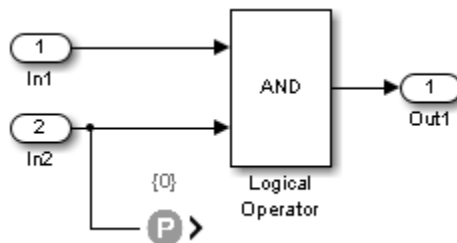
Pass through style

Specify whether the block displays an output port in the Simulink Editor. If selected (the default), the block displays its output port, allowing its input signal to pass through as the block output. If not selected, the block hides its output port and terminates the input signal. The following graphics illustrate the appearance of the block in each case.

Pass through style: Selected



Pass through style: Deselected



Stop simulation when the property is violated

Specify whether to stop the simulation if the simulation encounters a signal that violates the property specified in the **Values** parameter.

If you select this parameter and simulate the model, the simulation stops if it encounters a violation of the specified property.

See Also

Proof Assumption, Test Objective

Introduced in R2007a

Test Condition

Constrain signal values in test cases



Library

Simulink Design Verifier

Description

When operating in test generation mode, the Simulink Design Verifier software produces test cases that satisfy the specified criteria (see “What Is Test Case Generation?”). In this mode, you can use Test Condition blocks to define test conditions for signals in your model. The **Values** parameter lets you specify constraints on signal values during a test case simulation. The block applies the specified **Values** parameter to its input signal, and the Simulink Design Verifier software attempts to produce test cases that satisfy the condition.

The block's parameter dialog box also allows you to

- Enable or disable the condition.
- Specify that the block should display its **Values** parameter in the Simulink Editor.
- Specify that the block should display its output port.

Note The Simulink and Simulink Coder software ignore the Test Condition block during model simulation and code generation, respectively. The Simulink Design Verifier software uses the Test Condition block only when generating test cases for a model.

Specifying Test Conditions

Use the **Values** parameter to constrain signal values in test cases. Specify any combination of scalars and intervals in the form of a MATLAB cell array. For information about cell arrays, see “Cell Arrays” (MATLAB).

Tip If the **Values** parameter specifies only one scalar value, you do not need to enter it in the form of a MATLAB cell array.

Scalar values each comprise a single cell in the array, for example:

```
{0, 5}
```

A closed interval comprises a two-element vector as a cell in the array, where each element specifies an interval endpoint:

```
{[1, 2]}
```

Alternatively, you can specify scalar values using the `Sldv.Point` constructor, which accepts a single value as its argument. You can 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, `Sldv.Interval` considers an interval to be closed if you omit its third input argument.

As an example, the **Values** parameter

```
{0, [1, 3]}
```

specifies:

- 0 — a scalar
- [1, 3] — a closed interval

The **Values** parameter

```
{Sldv.Interval(0, 1, '[]'), Sldv.Point(1)}
```

specifies:

- `Sldv.Interval(0, 1, '[]')` — the right-open interval $[0, 1)$
- `Sldv.Point(1)` — a scalar

Logical Behavior of Specifications

If you specify multiple scalars and intervals for a Test Condition block, the Simulink Design Verifier software combines them using a logical OR operation when generating test cases. Consequently, the software considers the entire test condition to be satisfied if any single scalar or interval is satisfied.

Within a single scalar or interval, a test condition is generated with a logical AND operation. In this case, all signals must satisfy the constraints in order for the input to satisfy the condition.

For example, consider a two-dimensional open interval:

```
Sldv.Interval([-5 -5],[5 2], '()')
```

The zero vector $[0 \ 0]$ satisfies the condition because the zero elements are within the intervals -5 to 5 and -5 to 2.

The vector $[0 \ 3]$ does not satisfy the condition because the second element 3 falls outside the interval -5 to 2.

Data Type Support

The Test Condition block accepts signals of all built-in data types supported by the Simulink software. For a discussion on the data types supported by the Simulink software, see “Data Types Supported by Simulink” (Simulink).

Parameters

Enable

Specify whether the block is enabled. If selected (the default), Simulink Design Verifier software uses the block when generating tests for a model. Clearing this option disables the block, that is, causes the Simulink Design Verifier software to behave as if the Test Condition block did not exist. If this option is not selected, the block appears grayed out in the Simulink Editor.

Type

Specify whether the block behaves as a Test Condition or Proof Assumption block. Select **Assumption** to transform the Test Condition block into a Proof Assumption block.

Values

Specify the test condition (see “Specifying Test Conditions” on page 2-20).

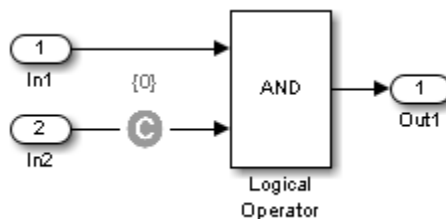
Display values

Specify whether the block displays the contents of its **Values** parameter in the Simulink Editor. By default, this option is selected.

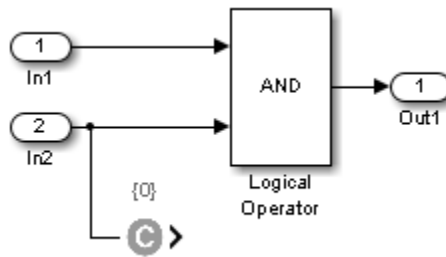
Pass through style

Specify whether the block displays an output port in the Simulink Editor. If selected (the default), the block displays its output port, allowing its input signal to pass through as the block output. If not selected, the block hides its output port and terminates the input signal. The following graphics illustrate the appearance of the block in each case.

Pass through style: Selected



Pass through style: Deselected



See Also

Proof Assumption, Test Objective

Introduced in R2007a

Test Objective

Define custom objectives that signals must satisfy in test cases



Library

Simulink Design Verifier

Description

When operating in test generation mode, the Simulink Design Verifier software produces test cases that satisfy the specified criteria (see “What Is Test Case Generation?”). In this mode, you can use Test Objective blocks to define custom test objectives for signals in your model. The **Values** parameter lets you specify values that a signal must achieve for at least one time step during a test case simulation. The block applies the specified **Values** parameter to its input signal, and the Simulink Design Verifier software attempts to produce test cases that satisfy the objective.

The block's parameter dialog box also allows you to

- Enable or disable the objective.
- Specify that the block should display its **Values** parameter in the Simulink editor.
- Specify that the block should display its output port.

Note The Simulink and Simulink Coder software ignore the Test Objective block during model simulation and code generation, respectively. The Simulink Design Verifier software uses the Test Objective block only when generating test cases for a model.

Specifying Test Objectives

Use the **Values** parameter to define custom objectives that signals must satisfy in test cases. Specify any combination of scalars and intervals in the form of a MATLAB cell array. For information about cell arrays, see “Cell Arrays” (MATLAB).

Tip If the **Values** parameter specifies only one scalar value, you do not need to enter it in the form of a MATLAB cell array.

Scalar values each comprise a single cell in the array, for example:

```
{0, 5}
```

A closed interval comprises a two-element vector as a cell in the array, where each element specifies an interval endpoint:

```
{[1, 2]}
```

Alternatively, you can specify scalar values using the `Sldv.Point` constructor, which accepts a single value as its argument. You can 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, `Sldv.Interval` considers an interval to be closed if you omit its third input argument.

As an example, the **Values** parameter

```
{0, [1, 3]}
```

specifies:

- 0 — a scalar
- [1, 3] — a closed interval

The **Values** parameter

```
{Sldv.Interval(0, 1, '['), Sldv.Point(1)}
```

specifies:

- `Sldv.Interval(0, 1, '[')` — the right-open interval $[0, 1)$
- `Sldv.Point(1)` — a scalar

Logical Behavior of Specifications

If you specify multiple scalars and intervals for a Test Objective block, the Simulink Design Verifier software combines them using a logical OR operation when generating test cases. Consequently, the software considers the entire test objective to be satisfied if any single scalar or interval is satisfied.

Within a single scalar or interval, a test objective is generated with a logical AND operation. In this case, all signals must satisfy the constraints in order for the input to satisfy the objective.

For example, consider a two-dimensional open interval:

```
Sldv.Interval([-5 -5],[5 2], '(')
```

The zero vector $[0 \ 0]$ satisfies the objective because the zero elements are within the intervals -5 to 5 and -5 to 2.

The vector $[0 \ 3]$ does not satisfy the objective because the second element 3 falls outside the interval -5 to 2.

Data Type Support

The Test Objective block accepts signals of all built-in data types supported by the Simulink software. For a discussion on the data types supported by the Simulink software, see “Data Types Supported by Simulink” (Simulink).

Parameters

Enable

Specify whether the block is enabled. If selected (the default), the Simulink Design Verifier software uses the block when generating tests for a model. Clearing this option disables the block, that is, causes the Simulink Design Verifier software to behave as if the Test Objective block did not exist. If this option is not selected, the block appears grayed out in the Simulink Editor.

Values

Specify the test objective (see “Specifying Test Objectives” on page 2-25).

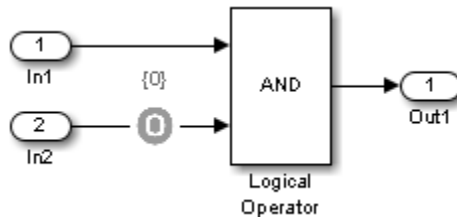
Display values

Specify whether the block displays the contents of its **Values** parameter in the Simulink editor. By default, this option is selected.

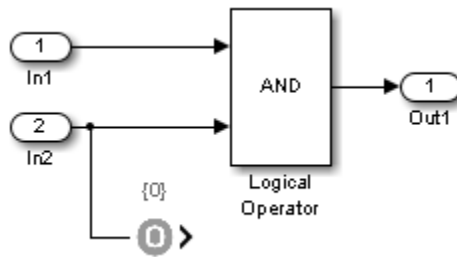
Pass through style

Specify whether the block displays an output port in the Simulink editor. If selected (the default), the block displays its output port, allowing its input signal to pass through as the block output. If not selected, the block hides its output port and terminates the input signal. The following figure illustrates the appearance of the block in each case.

Pass through style: Selected



Pass through style: Deselected



See Also

Proof Objective, Test Condition

Introduced in R2007a

Verification Subsystem

Specify proof or test objectives without impacting simulation results or generated code



Library

Simulink Design Verifier

Description

This block is a Subsystem block that is preconfigured to serve as a starting point for creating a subsystem that specifies proof or test objectives for use with the Simulink Design Verifier software.

The Simulink Coder software ignores Verification Subsystem blocks during code generation, behaving as if the subsystems do not exist. A Verification Subsystem block allows you to add Simulink Design Verifier components to a model without affecting its generated code.

Note If a Verification Subsystem block contains blocks that depend on absolute time, and you select an ERT-based target (Simulink Coder) for code generation, set the software environment to absolute time. Open the Configuration Parameters dialog box. In the **Code Generation > Interface** pane under **Software environment**, select **absolute time**. Do not select **continuous time**. For more information on this setting, see “Support: absolute time” (Simulink Coder).

When collecting model coverage, the Simulink Coverage software only records coverage for Simulink Design Verifier blocks in the Verification Subsystem block; it does not record coverage for any other blocks in the Verification Subsystem.

To create a Verification Subsystem in your model:

- 1 Copy the Verification Subsystem block from the Simulink Design Verifier library into your model.
- 2 Open the Verification Subsystem block by double-clicking it.
- 3 In the Verification Subsystem window, add blocks that specify proof or test objectives. Use Inport blocks to represent input from outside the subsystem.

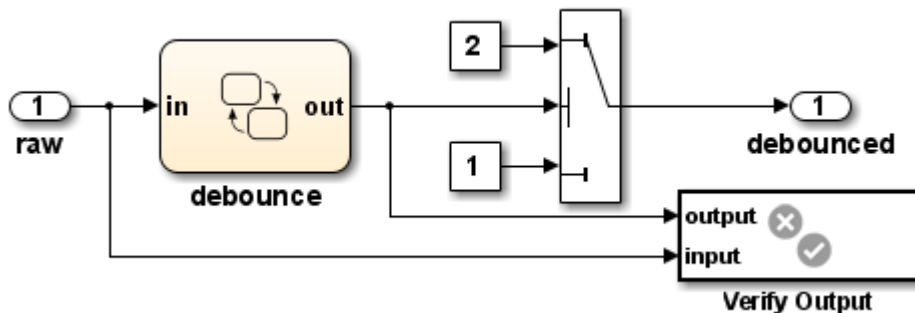
The Verification Subsystem block in the Simulink Design Verifier library is preconfigured to work with the Simulink Design Verifier software. A Verification Subsystem block must:

- Contain no Outputport blocks.
- Enable its **Treat as Atomic Unit** parameter.
- Specify its **Mask type** parameter as `VerificationSubsystem`.

If you alter the Verification Subsystem block so that the preceding conditions are not met, the Simulink Design Verifier software displays a warning.

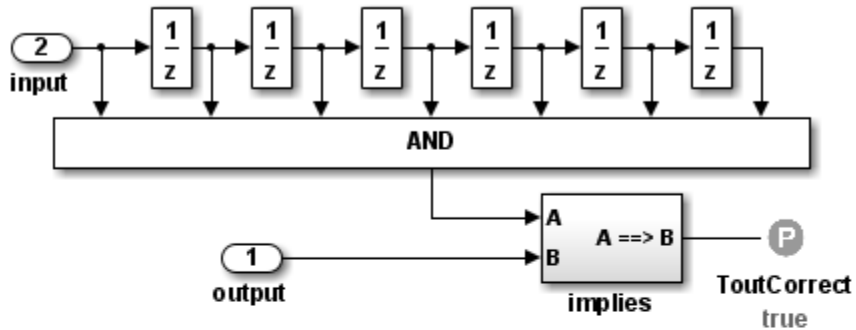
Examples

The `sldvdemo_debounce_validprop` example model includes a Verification Subsystem called Verify Output, as shown in the image below.

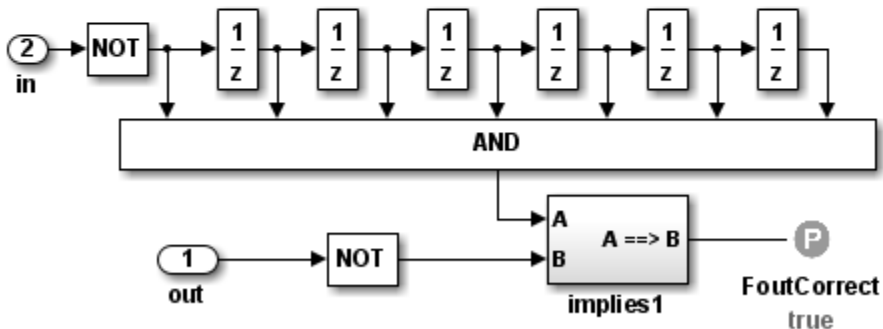


The Verify Output subsystem specifies two proof objectives, detailed in the following image.

sldvdemo_debounce_validprop ▶ Verify Output ▶



Prove that when the current and six previous inputs are true the output is true.



Prove that when the current and six previous inputs are false the output is false.

See Also

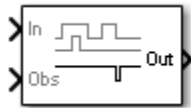
- Implies
- Within Implies
- Proof Assumption
- Proof Objective
- Test Condition

- Test Objective
- Subsystem block in the Simulink documentation
- “Create a Subsystem” (Simulink)

Introduced in R2007b

Within Implies

Verify response occurs within desired duration



Library

Simulink Design Verifier

Temporal Operators Terminology

- True duration of a signal — Consecutive time steps during which a signal is true

Description

The Within Implies block captures the within implication by observing whether the `Obs` input is true for at least one step within each true duration of the first input `In`. Whenever `Obs` is not detected within a particular input true duration, the output becomes false for one time step in the step that follows the input true duration.

Parameters

The Within Implies block has only one user-specified parameter:

External reset

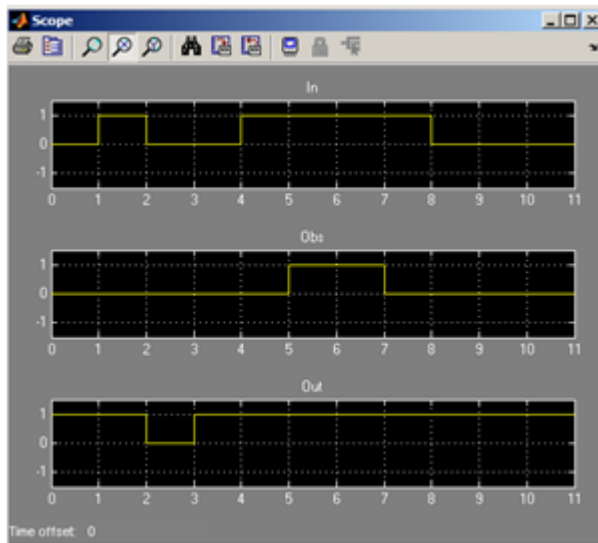
Specify whether the block observation of `Obs` can be reset by an external Boolean reset signal.

Examples

In the following example, consider a sample time of 1 second.

Obs is not observed within the first true duration of In, so Out becomes false for one time step. Obs is observed within the second true duration of In, so Out is true. When there is no true duration of In, Out remains true.

If Obs occurs multiple times, it does not affect the output.



See Also

Detector, Extender

Introduced in R2011a

Model Advisor Checks

Simulink Design Verifier Checks

In this section...
“Simulink Design Verifier Checks Overview” on page 3-2
“Check compatibility with Simulink Design Verifier” on page 3-2
“Detect Dead Logic” on page 3-4
“Detect Integer Overflow” on page 3-6
“Detect Division by Zero” on page 3-7
“Detect Out Of Bound Array Access” on page 3-8
“Detect Violation of Specified Minimum and Maximum Values” on page 3-10

Simulink Design Verifier Checks Overview

These checks help you prepare your model for Simulink Design Verifier analysis. When you run a Simulink Design Verifier check, the Model Advisor checks out the Simulink Design Verifier license.

For more information on the Model Advisor, see “Run Model Checks” (Simulink) and “Automate Model Advisor Check Execution” (Simulink Check).

Check compatibility with Simulink Design Verifier

Check ID: `mathworks.sldv.compatibility`

Identify elements that Simulink Design Verifier analysis does not support.

Description

This check assesses your model for compatibility with Simulink Design Verifier.

Results and Recommended Actions

Condition	Recommended Action
Incompatible	<p>Avoid using the following unsupported software features or Simulink blocks in the model or model component that you want to analyze:</p> <ul style="list-style-type: none"> • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” • “Support Limitations for Model Blocks” • “Support Limitations for Simulink Software Features” • “Support Limitations for Stateflow Software Features” • “Support Limitations for MATLAB for Code Generation”
Partially compatible	<ul style="list-style-type: none"> • Use automatic stubbing to ignore the behavior of unsupported blocks during analysis. See “Handle Incompatibilities with Automatic Stubbing” • Analyze components of your model separately. See “Extract Subsystems for Analysis” and “Bottom-Up Approach to Model Analysis” • If you have a complex model with a large verification state space, see “Sources of Model Complexity” for tips on performing Simulink Design Verifier analysis.
Compatible	Simulink Design Verifier can analyze your model.

See Also

- “Run Model Advisor Checks” (Simulink)
- “Check Model Compatibility”

- “Handle Incompatibilities with Automatic Stubbing”

Detect Dead Logic

Check ID: `mathworks.sldv.deadlogic`

Identify logic that stays inactive during simulation.

Description

This check identifies portions of your model that stay inactive during simulation.

You can run a more detailed analysis that identifies both dead logic and active logic using Simulink Design Verifier design error detection. For more information, see “Detect Dead Logic Caused by an Incorrect Value”.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See: <ul style="list-style-type: none"> • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” • “Support Limitations for Model Blocks” • “Support Limitations for Simulink Software Features” • “Support Limitations for Stateflow Software Features” • “Support Limitations for MATLAB for Code Generation” Also see “Handle Incompatibilities with Automatic Stubbing”.

Result	Recommended Action
Dead logic found in model	<p>Simulink Design Verifier proved that these decision and condition outcomes cannot occur and are dead logic in the model. Dead logic can also be a side effect of specified constraints on parameters or specified minimum and maximum constraints on input ports. In rare cases, dead logic can result from approximations performed by Simulink Design Verifier. It is possible that there are objectives that this analysis did not decide. To extend the results of this analysis, use Simulink Design Verifier design error detection to also identify active logic. From the Simulink Editor, select Analysis > Design Verifier > Options. In the Design Error Detection pane, select both Dead logic and Identify active logic.</p>
Dead logic not found in model	<p>Simulink Design Verifier did not find dead logic in the model. It is possible that there are objectives that this analysis did not decide. To extend the results of this analysis, use Simulink Design Verifier design error detection to also identify active logic. From the Simulink Editor, select Analysis > Design Verifier > Options. In the Design Error Detection pane, select both Dead logic and Identify active logic.</p>

See Also

- MISRA C:2012: Rule 2.1
- CERT C, MSC07-C
- CWE, CWE-561
- “Run Model Checks” (Simulink)
- “Secure Coding Standards” (Embedded Coder)

- “Detect Dead Logic Caused by an Incorrect Value”
- “Design Verifier Pane: Design Error Detection”

Detect Integer Overflow

Check ID: `mathworks.sldv.integeroverflow`

Detects integer or fixed-point data overflow errors in your model

Description

This check identifies operations that exceed the data type range for integer or fixed-point operations.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See <ul style="list-style-type: none">• “Supported and Unsupported Simulink Blocks in Simulink Design Verifier”• “Support Limitations for Model Blocks”• “Support Limitations for Simulink Software Features”• “Support Limitations for Stateflow Software Features”• “Support Limitations for MATLAB for Code Generation” Also see “Handle Incompatibilities with Automatic Stubbing”.

Result	Recommended Action
Integer overflow found in model	To view the conditions that cause the integer overflow, create a harness model. When you simulate the harness, the inputs replicate the error. Click View test case in the Model Advisor report.

See Also

- MISRA C:2012: Directive 4.1
- ISO/IEC TS 17961: 2013, intoflow
- CERT C, INT30-C and INT32-C
- CWE, CWE-190
- “Secure Coding Standards” (Embedded Coder)
- “Design Error Detection”

Detect Division by Zero

Check ID: `mathworks.sldv.divbyzero`

Detects division-by-zero errors in your model

Description

This check identifies operations in your model that cause division-by-zero errors.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See <ul style="list-style-type: none"> • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” • “Support Limitations for Model Blocks” • “Support Limitations for Simulink Software Features” • “Support Limitations for Stateflow Software Features” • “Support Limitations for MATLAB for Code Generation” Also see “Handle Incompatibilities with Automatic Stubbing”.
Division by zero found in model	To view the conditions that cause the division by zero, create a harness model. When you simulate the harness, the inputs replicate the error. Click View test case in the Model Advisor report.

See Also

- MISRA C:2012: Directive 4.1
- ISO/IEC TS 17961: 2013, diverr
- CERT C, INT33-C and FLP03-C
- CWE, CWE-369
- “Secure Coding Standards” (Embedded Coder)
- “Design Error Detection”

Detect Out Of Bound Array Access

Check ID: `mathworks.sldv.arraybounds`

Detects operations that access outside the bounds of an array index

Description

This check detects instances of out of bound array access in Simulink Design Verifier.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See <ul style="list-style-type: none"> • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” • “Support Limitations for Model Blocks” • “Support Limitations for Simulink Software Features” • “Support Limitations for Stateflow Software Features” • “Support Limitations for MATLAB for Code Generation” Also see “Handle Incompatibilities with Automatic Stubbing”.
Out of bound array access found in model	To view the conditions that cause the out of bound array access, create a harness model. When you simulate the harness, the inputs replicate the error. Click View test case in the Model Advisor report.

See Also

- MISRA C:2012: Rule 18.1
- ISO/IEC TS 17961: 2013, invptr
- CERT C, ARR30-C
- CWE, CWE-118

- “Secure Coding Standards” (Embedded Coder)
- “Design Error Detection”

Detect Violation of Specified Minimum and Maximum Values

Check ID: `mathworks.sldv.minmax`

Detect signals which exceed specified minimum and maximum values

Description

This analysis checks the specified minimum and maximum values (the design ranges) on intermediate signals throughout the model and on the output ports. If the analysis detects that a signal exceeds the design range, the results identify where in the model the errors occurred.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See <ul style="list-style-type: none">• “Supported and Unsupported Simulink Blocks in Simulink Design Verifier”• “Support Limitations for Model Blocks”• “Support Limitations for Simulink Software Features”• “Support Limitations for Stateflow Software Features”• “Support Limitations for MATLAB for Code Generation” Also see “Handle Incompatibilities with Automatic Stubbing”.

Result	Recommended Action
Violation of minimum and/or maximum found in model	To view the conditions that cause the violation, create a harness model. When you simulate the harness, the inputs replicate the error. Click View test case in the Model Advisor report.

See Also

- MISRA C:2012: Directive 4.1
- CERT C, API00-C
- CWE, CWE-628
- “Secure Coding Standards” (Embedded Coder)
- “Design Range Checks”

