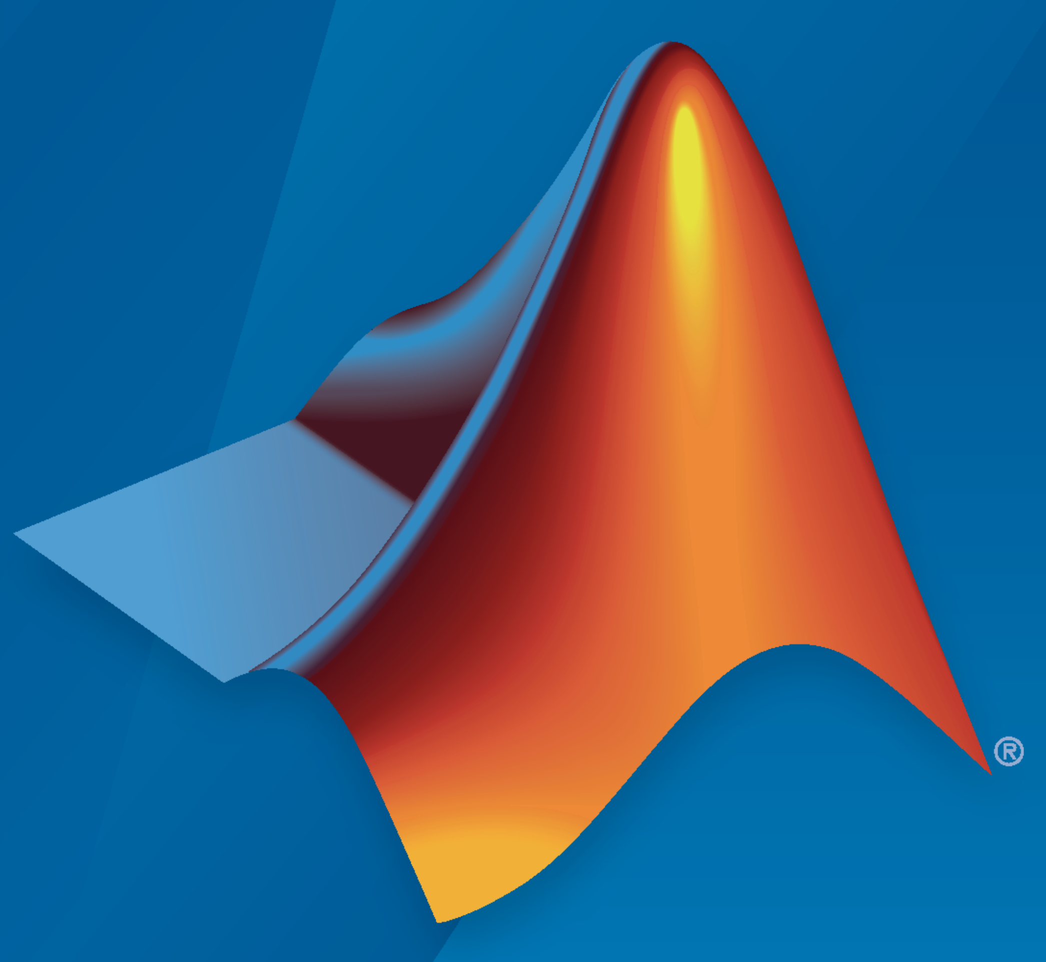
Simulink® Design Verifier™

Reference



MATLAB® & SIMULINK®

R2021b

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Simulink® Design Verifier™ Reference

© COPYRIGHT 2007–2021 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)
September 2018	Online only	Revised for Version 4.0 (Release 2018b)
March 2019	Online only	Revised for Version 4.1 (Release 2019a)
September 2019	Online only	Revised for Version 4.2 (Release 2019b)
March 2020	Online only	Revised for Version 4.3 (Release 2020a)
September 2020	Online only	Revised for Version 4.4 (Release 2020b)
March 2021	Online only	Revised for Version 4.5 (Release 2021a)
September 2021	Online only	Revised for Version 4.6 (Release 2021b)

Contents

1 Functions

2 Blocks

3 Model Advisor Checks

Simulink Design Verifier Checks **3-2**
Simulink Design Verifier Checks Overview **3-2**
Check compatibility with Simulink Design Verifier **3-2**
Detect Dead Logic **3-3**
Detect Out Of Bound Array Access **3-5**
Detect Division by Zero **3-6**
Detect Integer Overflow **3-7**
Detect Non-finite and NaN Floating-Point Values **3-8**
Detect Subnormal Floating-Point Values **3-8**
Detect Specified Minimum and Maximum Value Violations **3-9**
Detect Data Store Access Violations **3-10**
Detect Block Input Range Violations **3-11**
Check usage of rem and reciprocal operations **3-12**
Check usage of Sqrt operations **3-13**
Check usage of log and log10 operations **3-13**
Check usage of Reciprocal Sqrt blocks **3-13**

Functions

sldv.assume

Proof assumption function for Stateflow charts and MATLAB Function blocks

Syntax

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

Description

`sldv.assume(expr)` specifies that `expr` is 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 the proof assumptions represented by the `sldv.assume` function and by the Proof Assumption block.

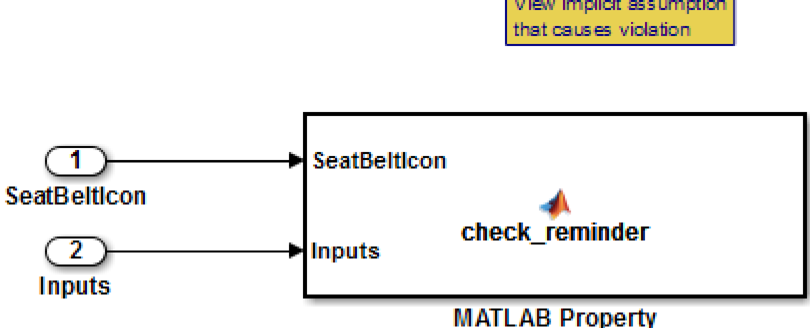
Examples

Specify Property Proof Objective and Proof Assumption

Specify a property proof objective and proof assumption in the `sldvdemo_sbr_verification` model by using a MATLAB Function block.

Open the `sldvdemo_sbr_verification` model and save it as `ex_sldvdemo_sbr_verification`.

Open the Safety Properties subsystem.



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

Safety Properties/MATLAB Property

```
function check_reminder(SeatBeltIcon,Inputs) %#codegen
    % The seat belt light should be active whenever the key is turned on
    % and speed is less than 15 and the seatbelt is not fastened
    activeCond = ((Inputs.KEY ~= 0) && (Inputs.SeatBeltFasten == 0) && ...
                  (Inputs.Speed < 15));

    sldv.prove(implies(activeCond,SeatBeltIcon));

function out = implies(cond, result)
    if (cond)
        out = result;
    else
        out = true;
    end

```

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

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

To save the updated code, in the **Editor** tab, click **Save** and close the editor.

To prove the safety properties, in the Simulink Editor, select the Safety Properties subsystem. On the **Design Verifier** tab, click **Prove Properties**.

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

Input Arguments

expr — Boolean expression for assumption

boolean expression

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

Alternatives

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

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

- Express the assumption by using a Simulink block.

- Explicitly connect the assumption output to a Simulink block.

See Also

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

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);
```

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

Output Arguments

status — Status of block-replacement
logical

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

newmodel — Handle to new model
double

sldvblockreplacement returns a handle to the new model in newmodel.

See Also
sldvoptions

Topics
“Block Replacements for Unsupported Blocks”

Introduced in R2007a

sldvcompat

Check model for compatibility with analysis

Syntax

```
status = sldvcompat(model)
[status, errmsgs] = 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, errmsgs] = sldvcompat(model)` returns a `status` of `1` (true) if the `model` is compatible with Simulink Design Verifier and `0` (false) otherwise and also returns a list of incompatibilities.

`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_flipflop'

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.

errmsgs — Returns a list of incompatibilities

If the model is compatible with Simulink Design Verifier, the errmsgs will be [] (empty). If the model is incompatible, the errmsgs is a struct that returns a list of incompatibilities.

Alternatives

To check if a model is compatible with Simulink Design Verifier, on the **Design Verifier** tab, in the **Analyze** section, click **Check Compatibility**.

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.

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`.

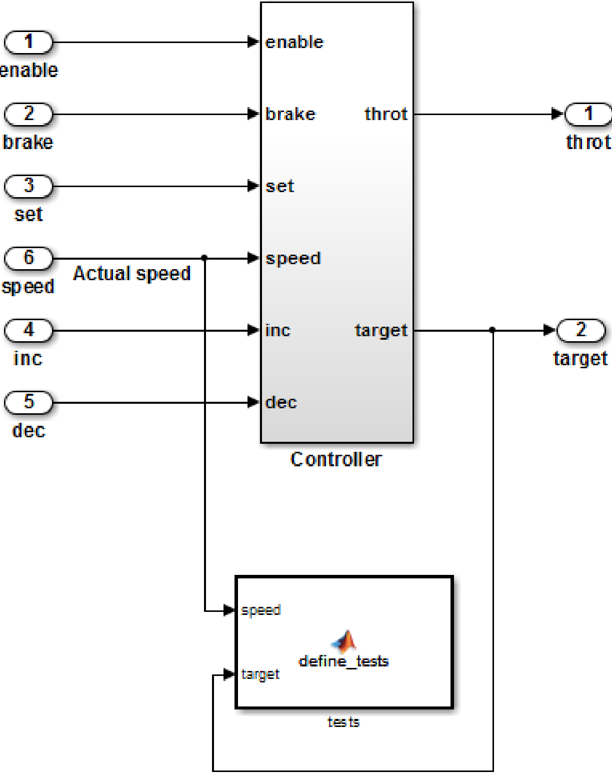
From the User-Defined Functions library, add a MATLAB Function block:

1. Name the block `tests`.
2. 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);
   ```
3. In the **Editor** tab, click **Save** and close the editor.
4. Connect the block to the signals for the `speed` block and for the `target` block.

Save the changes to the `ex_environment_controller` model.



To generate test cases, on the **Design Verifier** tab, click **Generate Tests**.

Input Arguments

`expr` — Boolean expression for condition
boolean expression

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

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

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

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(exportfcnmodel)
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(exportfcnmodel)` creates a scheduler model that invokes the export-function model `exportfcnmodel` for analysis by Simulink Design Verifier. `sldvextract` returns the name of the new model in `newModel`. The `newModel` consists of `exportfcnmodel` model name with a suffix `SldvScheduler`. If the model name already exists, `sldvextract` uses the `exportfcnmodel` name for the model name, appending a number to the model name.

`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 `Bus Counter` from the `sldemo_mdlref_conversion` model and copy it to a new model.

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

Extract the Atomic Subchart

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);
```

Create the Scheduler for Export-Function Model

Create the scheduler model that invokes the `sldvExportFunction_autosar_multirunnables` model.

```
addpath(fullfile(docroot,'toolbox','sldv','examples'));
open_system('sldvExportFunction_autosar_multirunnables');
newmodel = sldvextract('sldvExportFunction_autosar_multirunnables');
```

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.

`exportfcnmodel` — Path to export-function model
character vector | string scalar

Full path to the export-function model.

`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"
"Analyze Export-Function Models"

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 = 'MCDC';
% 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 ] = sldvruntest('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);
```

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.

`newmodel` — Handle to new model
`double`

If the API is invoked with `block`, then the software returns the handle to the new model in `newmodel` which contains a copy of the atomic subsystem `block`.

See Also

`sldvruntest` | `sldvmergeharness` | `sldvoptions` | `sldvrun`

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`.

Examples

Create Test Harness Model by Using Default Options

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

Open the `sldvdemo_cruise_control` example model.

```
open_system('sldvdemo_cruise_control');
```

Using the default options, create a test harness for the `sldvdemo_cruise_control` model.

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

Output Arguments

`harnessopts` — Options for harness model
structure

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

The `harnessopts` structure can have these 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 you do not specify this option, `sldvmakeharness` generates a new harness model and saves it in the MATLAB current folder. Default: `''`
`modelRefHarness`	Generates the test harness model that includes the model in a Model block. When `false`, the test harness model includes a copy of the model. Default: `true`
`usedSignalsOnly`	When `true`, the Signal Builder block in the harness model has signals only for input signals in the model. Model must be compatible with the Simulink Design Verifier software to detect those input signals. When the `harnessSource` parameter is set to `Signal Editor`, the Signal Editor block in the harness model consists of unused bus signals in addition to the input signals in the model. Default: `false`
`harnessSource`	When set to `Signal Builder`, Simulink Design Verifier generates a harness model with the Signal Builder block as the Inputs block. When set to `Signal Editor`, Simulink Design Verifier generates a harness model with the Signal Editor block as the Inputs block. Default: `Signal Builder`

See Also

`sldvmakeharness`

Topics

"Simulink Design Verifier Harness Models"
"Create Harness Model"
"Generate Test Harness Model and Record Coverage Data"

Introduced in R2010b

sldvhighlight

Highlight model using data from Simulink Design Verifier analysis

Syntax

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

Description

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

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

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

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

Examples

Highlight Active Analysis Results on Current Model

Highlight the current model that has active Simulink Design Verifier analysis results.

Open the `sldvdemo_debounce_modelcov` example model.

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

Using the default settings, run test generation analysis on the example model.

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

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

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

Generating tests using model representation from 15-Mar-2019 13:12:06...
...................

Completed normally.
```

```
Generating output files:

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

Results generation completed.

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

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.

To view the detailed analysis results for the object in the Results Inspector, click a highlighted object in the model.

Highlight Active Analysis Results on Specified Model

Highlight a specified model that has active Simulink Design Verifier analysis results.

Open the `sldvdemo_debounce_modelcov` example model.

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

Using the default settings, run the test generation analysis on the model.

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

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

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

Generating tests using model representation from 15-Mar-2019 13:12:06...
...................

Completed normally.

Generating output files:

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

Results generation completed.

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

status =

     1
```

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

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

The example model is highlighted with the analysis results.

To view the detailed analysis results for the object in the Results Inspector, click a highlighted object in the model.

Highlight Analysis Results from Data File on a Model

Highlight a specified model that has Simulink Design Verifier analysis results, loaded from a data file.

Open the `sldvdemo_debounce_modelcov` example model.

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

Using the default settings, run the test generation analysis on the example model.

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

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

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

Generating tests using model representation from 15-Mar-2019 13:12:06...
...................

Completed normally.

Generating output files:

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

Results generation completed.

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

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 that has 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.

To view the detailed analysis results for the object in the Results Inspector, click highlighted object in the model.

Input Arguments

`model` — Name or handle of model to highlight
character vector | string scalar

Name of the model or handle of the 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'`

`Filter` — Name of analysis filter file with justification rules
character vector | string scalar

Name of the analysis filter file that contains the justification rules.

Example: `'sldvdemo_design_error_detection_sldv_filter.cvf'`

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 translating the current Simulink model. If the software is actively translating the current model, `sldvisactive` returns `1`. Otherwise, it returns `0`.

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

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

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

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 translating model
logical

If the software is actively translating 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 recently generated Simulink Design Verifier analysis results for `model` in the Results Summary window. If `model` is not open, `sldvloadresults` opens the `model`.

To load the results from the data file, on the **Design Verifier** tab, in the **Review Results** section, click **Load Earlier Results** and open the data file. The `sldvloadresults` loads the result from this specified data file.

`status = sldvloadresults(model, dataFile)` loads the analysis results for `model` from `dataFile` in the Results Summary window. If `model` is not open, `sldvloadresults` opens the `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 the test generation analysis on the example model by using its default settings.

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

```
Checking compatibility for test generation: model 'sldvdemo_flipflop'
Compiling model...done
Building model representation...done
'sldvdemo_flipflop' is compatible for test generation with Simulink Design Verifier.

Generating tests using model representation from 14-Dec-2018 16:33:34...
............

Completed normally.

Generating output files:

Results generation completed.

    Data file:
    H:\work\sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat

status =

     1
```

Close the example model.

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

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

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

```
ans =

     1
```

You can view the loaded analysis results in the Simulink Design Verifier Results Summary window. To open the Results Summary window, on the **Design Verifier** tab, in the **Review Results** section, click **Results Summary**.

Load Results from Data File for Specified Model

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

Open the `sldvdemo_flipflop` example model.

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

Using the default settings, run the test generation analysis on the example model.

```
[status, filenames] = sldvrun('sldvdemo_flipflop')
```

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

Generating tests using model representation from 14-Dec-2018 16:36:43...
............

Completed normally.

Generating output files:

Results generation completed.

    Data file:
    H:\work\sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat

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',filenames.DataFile)
```

```
ans =

     1
```

You can view the loaded analysis results in the Simulink Design Verifier Results Summary window. To open the Results Summary window, on the **Design Verifier** tab, in the **Review Results** section, click **Results Summary**.

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

The name of the data file containing the analysis results. dataFile must contain analysis results for the specified model.

If dataFile is 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 the analysis results data, when you specify dataFile, choose a data file that contains results from the same version of the 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, which returns 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`.

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 the `harness_model` and logs the input signals to the Test Unit block in the harness model. Generate `harness_model` by using Simulink Design Verifier analysis, `sldvmakeharness`, or `slvnvmakeharness`.

`data = sldvlogsignals(harness_model, test_case_index)` simulates every test case in the Signal Builder or Signal Editor block of the `harness_model` that specifies the `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 or Signal Editor.

Examples

Create a Harness Model by Using Logged Signals

Use logged signals to create a harness model to visualize the data.

Simulate the CounterB Model block,which references the `sldemo_mdlref_counter` model, in the context of the `sldemo_mdlref_basic` model. Then log the data:

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

Create a harness model for `sldemo_mdlref_counter` model by using the logged data and the default harness options:

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

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

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 `slvnvmakeharness` creates.

`test_case_index` — Simulate test cases
character vector | cell array of character vectors | string array

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

Output Arguments

`data` — Structure of logged data
structure

Structure that contains the logged data.

See Also
`sldvmakeharness` | `slvnvmakeharness` (Simulink Coverage)

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
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: ''
modelRefHarness	Generates the test harness model that includes model in a Model block. When false, the test harness model includes a copy of model. Default: true
usedSignalsOnly	When true, the Signal Builder block in the harness model has signals for only input signals used in the model. model must be compatible with the software to detect the used input signals. When the harnessSource parameter is set to Signal Editor, the Signal Editor block in the harness model consists of unused bus signals in addition to the input signals in the model. Default: false
harnessSource	When set to Signal Builder, Simulink Design Verifier generates a harness model with the Signal Builder block as the Inputs block. When set to Signal Editor, Simulink Design Verifier generates a harness model with the Signal Editor block as the Inputs block. Default: Signal Builder

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. Open the model.
2. On the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.
3. In the Configuration Parameters dialog box, on the **Design Verifier > Results** pane, in **Harness model options**, set the options that you want.
4. Click **OK**.
5. To perform test generation analysis, on the **Design Verifier** tab, in the **Mode** section, select **Test Generation**.
6. Click **Generate Tests**.

See Also

`sldvharnessopts` | `sldvmergeharness` | `sldvrun` | `slvnvharnessopts` | `slvnvmakeharness` | `slvnvmergeharness`

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.

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. The function saves the data and initialization commands in name, which is a handle to the new model.

If name does not exist, sldvmergeharness creates name as a copy of the first model in models. sldvmergeharness then merges the 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 maintains 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 has more than one Signal Builder block on the top level 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 the model test data into name.

To combine the test cases that use different sets of parameter values, use sldvmergeharness with sldvgencov.

Examples

Merge the Two Test Harnesses

Analyze the sldvdemo_cruise_control model for decision objectives and full coverage, and then merge the two test harnesses.

Open the sldvdemo_cruise_control example model.

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

Set the sldvoptions and collect decision coverage.

```
opts1 = sldvoptions;
opts1.Mode = 'TestGeneration';
opts1.ModelCoverageObjectives = 'Decision';
```

```
opts1.HarnessModelFileName = 'first_harness';
opts1.SaveHarnessModel = 'on';
sldvrun(model, opts1);
```

Set the `sldvoptions` and 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'});
```

Input Arguments

name — Name of new harness model

character vector | string scalar

Name of the new harness model, which is 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 of the same length as `models`. `initialization_commands` defines parameter settings for the test cases of each test harness model.

Output Arguments

`status` — Status of merge harness

boolean

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

See Also

`sldvgencov` | `sldvmakeharness` | `sldvrun`

Topics

"Extend an Existing Test Suite"
"Creating and Executing Test Cases"
"Using Command Line Functions to Support Changing Parameters"

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'`
`BlockReplacement`	Specify whether the software replaces blocks in a model before its analysis. When set to `'on'`, this parameter enables `BlockReplacementModel-FileName` and `BlockReplacementRulesList`.	`'on'` `'off'` (default)
`BlockReplacementModel-FileName`	Specify a folder and file name for the model that is the result after applying block replacement rules. This parameter is enabled when `BlockReplacement` is set to `'on'`.	character array `'$`*`ModelName`*`$_replacement'` (default)
`BlockReplacementRulesList`	Specify a list of block replacement rules that execute before its analysis. This parameter is enabled when `BlockReplacement` is set to `'on'`.	character array `'<FactoryDefaultRules>'` (default)
`CodeAnalysisExtraOptions`	Extra options for analyzing 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”.	character array `''` (default)

Parameter	Description	Values
`CoverageDataFile`	Specify a folder and file name for the file that contains data about satisfied coverage objectives. This parameter is enabled when `IgnoreCovSatisfied` is set to `'on'`.	character array `''` (default)
`CovFilter`	For test generation and design error detection analysis, specify whether to ignore objectives stored in coverage filter file. When set to on, this parameter enables `CovFilterFileName`.	`'on'` `'off'` (default)
`CovFilterFileName`	For test generation and design error detection analysis, specify a name for the coverage filter file that contains objectives to exclude from analysis. This parameter is enabled when `CovFilter` is set to `'on'`.	character array `''` (default)
`DataFileName`	Specify a folder and file name for the MAT-file that contains the data generated during the analysis, stored in an `sldvData` structure.	character array `'$`*`ModelName`*`$_sldvdata'` (default)
`DesignMinMaxCheck`	Specify whether to check that the intermediate and output signals in your model are within the range of specified minimum and maximum constraints.	`'on'` `'off'` (default)
`DesignMinMaxConstraints`	Specify whether Simulink Design Verifier software generates test cases that consider specified minimum and maximum values as constraints for input signals in your model.	`'on'` (default) `'off'`
`DetectActiveLogic`	Specify whether to analyze your model for active logic. This parameter is enabled only if `DetectDeadLogic` is set to `'on'`.	`'on'` `'off'` (default)

Parameter	Description	Values
`DetectBlockInputRange-Violations`	Specify whether to analyze your model for block input range violations. For more information, see “Specified block input range violations”.	`'on'` `'off'` (default)
`DetectDeadLogic`	Specify whether to analyze your model for dead logic.	`'on'` `'off'` (default)
`DetectDivisionByZero`	Specify whether to analyze your model for division-by-zero errors.	`'on'` (default) `'off'`
`DetectDSMAccessViolations`	Specify whether to analyze your model for data store access violations.	`'on'` `'off'` (default)
`DetectHISMViolationsHisl_0002`	Specify whether to check the usage of `rem` and `reciprocal` operations that cause non-finite results.	`'on'` `'off'` (default)
`DetectHISMViolationsHisl_0003`	Specify whether to check the usage of Square Root operations with inputs that can be negative.	`'on'` `'off'` (default)
`DetectHISMViolationsHisl_0004`	Specify whether to check the usage of `log` and `log10` operations that cause non-finite results.	`'on'` `'off'` (default)
`DetectHISMViolationsHisl_0028`	Specify whether to check the usage of Reciprocal Square Root blocks with inputs that can go zero or negative.	`'on'` `'off'` (default)
`DetectInfNaN`	Specify whether to analyze your model for non-finite and NaN floating-point values.	`'on'` `'off'`(default)
`DetectIntegerOverflow`	Specify whether to analyze your model for integer and fixed-point data overflow errors.	`'on'` (default) `'off'`
`DetectOutOfBounds`	Specify whether to analyze your model for out of bounds array access errors.	`'on'` (default) `'off'`
`DetectSubnormal`	Specify whether to analyze your model for subnormal floating-point values.	`'on'` `'off'`(default)
`DisplayReport`	Display the report that the Simulink Design Verifier analysis generates after completing its analysis. This parameter is enabled when `SaveReport` is set to `'on'`.	`'on'` (default) `'off'`

Parameter	Description	Values
ExistingTestFile	Specify a folder and file name for the MAT-file that contains the logged test case data. This parameter is enabled when Mode is set to 'TestGeneration' and ExtendExistingTests is set to 'on'.	character array '' (default)
ExtendExistingTests	Extend the Simulink Design Verifier analysis by importing test cases logged from a harness model or a closed-loop simulation model. When set to 'on', this parameter enables ExistingTestFile and IgnoreExistTestSatisfied. This parameter is enabled when Mode is set to 'TestGeneration'.	'on' 'off' (default)
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)
HarnessSource	Specify the type of the Inputs block for the harness model. This parameter is enabled when SaveHarnessModel is set to 'on'.	'Signal Builder' (default) 'Signal Editor'
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'

Parameter	Description	Values
`MaxProcessTime`	Specify the maximum time (in seconds) that Simulink Design Verifier spends analyzing a model. You can set the value of maximum analysis time to the value that you are willing to provide to the analysis. You can also stop the analysis at any time.	double `300` (default)
`MaxTestCaseSteps`	Specify the maximum number of simulation steps when attempting to satisfy a test objective. 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. 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. 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 to each individual iteration of test generation. This parameter is enabled when `Mode` is set to `'TestGeneration'`.	int32 `10000` (default)

Parameter	Description	Values
MaxViolationSteps	Specify the maximum number of simulation steps over which the software searches for property violations. This parameter is enabled whenMode is set to 'PropertyProving' and when ProvingStrategy is set to 'FindViolation' or 'ProveWithViolation-Detection'.	int32 20 (default)
Mode	Specify the analysis mode.	'TestGeneration' (default) 'PropertyProving' 'DesignErrorDetection'
ModelCoverageObjectives	Specify the type of model coverage to achieve. When ModelCoverageObjectives is set to 'MCDC', 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. This parameter is enabled when Mode is set to 'TestGeneration'.	'None' 'Decision' 'ConditionDecision' (default) 'MCDC' 'EnhancedMCDC'
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 ParametersConfigFile-Name.	'on' 'off' (default)

Parameter	Description	Values
ParametersConfigFileName	Specify a MATLAB function that defines parameter configurations for a model. This parameter is enabled when Parameters is set to 'on'. This parameter is disabled when ParametersUseConfig is set to 'on'.	character array 'sldv_params_template.m' (default)
ParametersUseConfig	Specify to use the Parameter Configuration table to define parameter configurations for a model. When set to 'on', this parameter disables ParametersConfigFileName.	'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'
RandomizeNoEffectData	Specify whether to use random values instead of zeros for input signals that have no impact on test or proof objectives.	'on' 'off' (default)
RebuildModelRepresentation	Specify whether to rebuild the model representation for Simulink Design Verifier analysis.	'Always' 'IfChangeIsDetected' (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)

Parameter	Description	Values
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)
SaveExpectedOutput	Simulate the model by using test case signals and include the output values in the Simulink Design Verifier data file.	'on' 'off' (default)
SaveHarnessModel	Create a harness model generated by the Simulink Design Verifier analysis. When SaveReport is set to 'on', this parameter must also be set to 'on'. When set to 'on', this parameter enables HarnessModelFileName.	'on' 'off' (default)
SaveReport	Generate and save a Simulink Design Verifier report. When this parameter is set to 'on', SaveHarnessModel must also be set to 'on'. When set to 'on', this parameter enables ReportFileName, ReportIncludeGraphics, and DisplayReport.	'on' 'off' (default)
SFcnSupport	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”.	'on' (default) 'off'
StrictEnhancedMCDC	Specify whether to use strict propagation conditions for enhanced MCDC analysis. This parameter is enabled when Mode is set to 'TestGeneration' and ModelCoverageObjectives is set to 'EnhancedMCDC'.	'on' 'off' (default)

Parameter	Description	Values
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) 'GenCodeTopModel' 'GenCodeModelRef'
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)
TestSuiteOptimization	Specify the optimization strategy to use when generating test cases. This parameter is enabled when Mode is set to 'TestGeneration'. If you analyze your model by using the LargeModel (Nonlinear Extended), the software displays a warning message that this option has been removed and suggests that you use Auto instead.	'Auto' (default) 'IndividualObjectives' 'LongTestCases' 'LargeModel (Nonlinear Extended)'
UseParallel	Specify whether to validate test cases or counterexamples with parallel computing. This option requires a Parallel Computing Toolbox™ license.	'on' 'off' (default)

Alternatives

To set the analysis options, on the **Design Verifier** tab, in the **Prepare** section, from the drop-down menu for the mode settings, click **Settings**.

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` is 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 the code or separate the assumptions into a verification script.

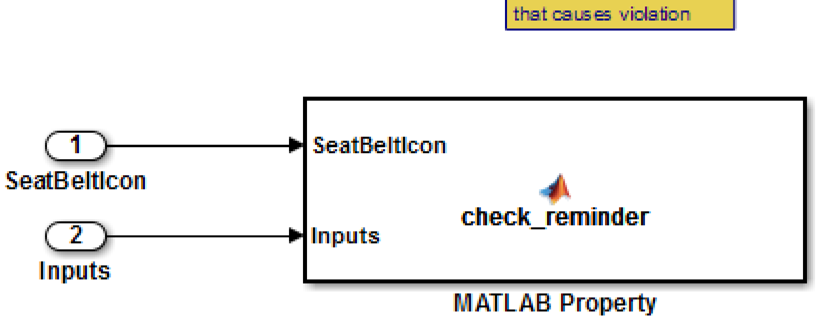
Examples

Specify Property Proof Objective and Proof Assumption

Specify a property proof objective and proof assumption in `sldvdemo_sqrt_blockrep` model by using a MATLAB Function block.

Open the `sldvdemo_sbr_verification` model and save it as `ex_sldvdemo_sbr_verification`.

Open the Safety Properties subsystem.



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

Safety Properties/MATLAB Property

```
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
```

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

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

To save the updated code, in the **Editor** tab, click **Save** and close the editor.

To prove the safety properties, in the Simulink Editor, select the Safety Properties subsystem. On the **Design Verifier** tab, click **Prove Properties**.

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

Input Arguments

expr — Boolean expression for assumption

boolean expression

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

Alternatives

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

- Explicitly connect the proof output to a Simulink block.

See Also

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

Topics

"Prove Properties in a Model"
"Workflow for Proving Model Properties"

Introduced in R2009b

sldvreport

Generate Simulink Design Verifier 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)
[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1,
reportOption2, ...}, reportFilePath, showUI, FORMAT, Filter)
```

Description

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

`[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`. If `showUI` is true, the software displays the status of the report generation in the user interface.

`[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`. If `showUI` is true, the software displays the status of the report generation in UI.

`[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1, reportOption2, ...}, reportFilePath, showUI, FORMAT, Filter)` generates a complete report in the specified `FORMAT` by applying the justification rules specified in the analysis filter file `Filter` and the data in `sldvDataFile`.

Examples

Create PDF of the Report

Analyze the model and create a PDF of the Simulink Design Verifier analysis report by using `sldvreport`.

Open the `sldvdemo_cruise_control` example model.

```
open_system('sldvdemo_cruise_control');
```

Set the `sldvoptions` and create a PDF version of the analysis results.

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

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 these 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 that are proven or are falsified (**Default**: true)

`reportFilePath` — Generated report

character vector | string scalar

The path and the 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, PDF, or both.
`'HTML'` (**default**) to generate an HTML format of the report.
`'PDF'` to generate a PDF version of the report.
`{'HTML', 'PDF'}` to generate HTML and PDF formats of the report.

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

`Filter` — Name of analysis filter file with justification rules
character vector | string scalar

Name of the analysis filter file that contains the justification rules.

Example: `'sldvdemo_design_error_detection_sldv_filter.cvf'`

Output Arguments

`status` — Status of report creation
logical

If the operation creates the analysis report, `sldvreport` returns a `status` of `true`. Otherwise, it returns `false`.

`reportFilePath` — Name of file and path
character vector

The path and the file name for the generated HTML report.

Alternatives

Simulink Design Verifier software creates 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 a PDF of the report, select **Generate additional report in PDF format**.

See Also
`sldvrun` | `sldvoptions`

Topics
"Simulink Design Verifier Reports"

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);
```

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`	Processing completed normally

Alternatives

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

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

See Also

`sldvcompat` | `sldvoptions` | `sldvgencov`

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

```
open_system('sldemo_mdlref_basic');
load_system('sldemo_mdlref_counter');
loggedData = sldvlogsignals('sldemo_mdlref_basic/CounterA');
```

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

```
runOpts = sldvruntestopts('cgv');
runOpts.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.

```
cgvObjectSim = sldvruncgvtest('sldemo_mdlref_counter', loggedData, runOpts);
runOpts.cgvConn = 'sil';
cgvObjectSil = sldvruncgvtest('sldemo_mdlref_counter', loggedData, runOpts);
```

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.

runOpts — configuration parameters

structure

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

Field Name	Description
`testIdx`	Test case index array to execute from `dataFile`. If `testIdx` is `[]`, `sldvruncgvtest` executes all test cases in `dataFile`. **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 `true` and you have not configured `model` to execute test cases with the CGV API, `sldvruncgvtest` copies the model, fixes the configuration, and executes the test cases on the copied model. If `false` (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, `sldvruncgvtest` does not copy the model, even if `allowCopyModel` is `true`. An error occurs.
`cgvCompType`	Defines the software-in-the-loop (SIL) or processor-in-the-loop (PIL) approach for CGV: • `'topmodel'` (default) • `'modelblock'`

Field Name	Description
cgvConn	Specifies mode of execution for CGV: • '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
logsout_sldvruncgvtest	Signal logging data for: • 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 | sldvrun | sldvruntest | sldvruntestopts

Topics

“Verify a Component for Code Generation”

"Creating and Executing Test Cases"

Introduced in R2010b

sldvruntest

Simulate model by 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 the `model` by using all the test cases in the `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 the `model` by using all the test cases in the `dataFile`. `runOpts` defines the options for simulating the test cases.

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

Examples

Analyze Model by Using Data from Test Cases

Analyze the `sldvdemo_cruise_control` model. Simulate the model by using the data from the three test cases. Examine the signal logging data by using the Simulation Data Inspector.

Open the `sldvdemo_cruise_control` example model.

```
open_system('sldvdemo_cruise_control');
```

Set the `sldvoptions` and analyze the model.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'off';
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
```

Simulate the model by using the data from the three test cases. Examine the signal logging data by using the Simulation Data Inspector.

```
runOpts = sldvruntestopts;
[ outData ] = sldvruntest('sldvdemo_cruise_control',...
    files.DataFile, runOpts);
Simulink.sdi.createRun('Test Case 1 Output', 'namevalue',...
    {'output'}, {outData(1).find('logsout_sldvruntest')});
```

```
Simulink.sdi.createRun('Test Case 2 Output', 'namevalue',...
    {'output'}, {outData(2).find('logsout_sldvruntest')});
Simulink.sdi.createRun('Test Case 3 Output', 'namevalue',...
    {'output'}, {outData(3).find('logsout_sldvruntest')});
Simulink.sdi.view;
```

Generate Coverage Report by Using the Simulation Data

Generate coverage report by simulating the test cases for the `sldvdemo_cruise_control` model.

Open the `sldvdemo_cruise_control` example model.

```
open_system('sldvdemo_cruise_control');
```

Set the `sldvoptions` and analyze the model.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'off';
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
```

Enable the option to collect model coverage data and simulate the model by using the data. Then, generate coverage report by using `cvhtml` (Simulink Coverage).

```
runOpts = sldvruntestopts;
runOpts.coverageEnabled = true;
[ outData,covData ] = sldvruntest('sldvdemo_cruise_control',...
                      files.DataFile,runOpts);
cvhtml('coverageReport',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 the `dataFile` by using the Simulink Design Verifier analysis or by running the `sldvlogsignals` function. For more information, see “Simulink Design Verifier Data Files”.

`runOpts` — Configurations for sldvruntest

character vector | cell array of character vectors | string array

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

Field	Description
testIdx	Test case index array to simulate from dataFile. If testIdx is [], sldvruntest simulates all the test cases. **Default:** []
signalLoggingSaveFormat	Specify the format for signal logging data for signals that connect to the outport of the model and for intermediate signals that are configured for logging. If you specify Dataset, data is stored in the Simulink.SimulationData.Dataset objects. **Default:** 'Dataset'
coverageEnabled	If true, specifies that the Simulink Coverage software collects the model coverage data during simulation. **Default:** false
coverageSetting	cvtest object for collecting the model coverage. If [], sldvruntest uses the existing coverage settings for model. **Default:** []
fastRestart	If true, Simulink Design Verifier uses fast restart mode for model simulation. **Default:** true
useParallel	If true, Simulink Design Verifier simulates test cases with parallel computing. This option requires a Parallel Computing Toolbox license. **Default:** false

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

Output Arguments

outData — Array of Simulink.SimulationOutput objects
object

An array of Simulink.SimulationOutput objects that the software generates after simulating the test cases. Each Simulink.SimulationOutput object has the fields listed in the table.

Field Name	Description
tout_sldvruntest	Simulation time

Field Name	Description
`xout_sldvruntest`	State data
`yout_sldvruntest`	Output signal data
`logsout_sldvruntest`	Signal logging data for: • Signals connected to outports • Signals that are configured for logging on the model

`covData` — Model coverage data
object

The `cvdata` object that contains the model coverage data, which the software collects during simulation.

Note `covdata` references a file containing the coverage results. The coverage data from the referenced file is automatically loaded into memory when `covdata` is used by a coverage function. This file gets stored in the `sldv_covoutput` folder inside the current directory.

Tips

- The `dataFile` that you create with a Simulink Design Verifier analysis or by running `sldvlogsignals` contains time values and data values. When you simulate a model by using these test cases, you can achieve missing coverage. The missing coverage occurs when the time values in the `dataFile` are not aligned with the current simulation time step due to numeric calculation differences. You achieve missing coverage more frequently with multirate models—models that have multiple sample times.
- For `useParallel`, the following points must be considered when simulating test cases using parallel computing:
  - Starting a parallel pool can take time, which impacts the overall analysis time. To reduce the analysis time:
    - Make sure that the parallel pool is already running before you run a test generation analysis. By default, the parallel pool shuts down after being idle for a specified number of minutes. To change the setting, see the topic 'Specify Your Parallel Preferences' in Parallel Computing Toolbox.
    - Load Simulink on all the parallel pool workers.
  - The simulation occurs sequentially when:
    - The cluster is not `local`. Configure parallel preferences to use the `local` cluster only. To change the setting, see the topic 'Specify Your Parallel Preferences' in Parallel Computing Toolbox.
    - The model is in `dirty state` prior to launching the SLDV analysis.
    - The model has `ToFile` blocks.
    - The model is in Software-in-the-loop (SIL) simulation mode.
    - The model is an internal harness.

See Also

`cvsim` | `cvtest` | `sim` | `sldvrun` | `sldvruntestopts`

Topics

"Extend an Existing Test Suite"
"Defining and Extending Existing Tests Cases"

Introduced in R2007b

sldvruntestopts

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

Syntax

```
runOpts = sldvruntestopts
runOpts = sldvruntestopts('cgv')
```

Description

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

`runOpts = sldvruntestopts('cgv')` generates a `runOpts` structure for `sldvruncgvtest`.

Examples

Create runopts for sldvruntest

Create `runOpts` for the `sldvdemo_cruise_control` model. Then, analyze the model by using `sldvruntest`.

Open the `sldvdemo_cruise_control` example model.

```
open_system('sldvdemo_cruise_control');
```

Set the `sldvoptions` and create `runOpts` for `sldvruntest`.

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'off';
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
runOpts = sldvruntestopts;
```

Analyze the model by using the `runOpts`.

```
[ outData ] = sldvruntest('sldvdemo_cruise_control',...
    files.DataFile, runOpts);
```

Input Arguments

cgv — Default options for sldvruncgvtest

character vector | string scalar

Generates a default `runOpts` structure for `sldvruncgvtest`.

Example: `runcgvtest_options = sldvruntestopts('cgv');`

Output Arguments

runOpts — Configuration of sldvruntest or sldvruncgvtest
structure

A structure whose fields specify the configuration of sldvruntest or sldvruncgvtest. If you do not specify a field name, sldvruncgvtest or sldvruntest uses the default value. runOpts can contain the fields listed in the table.

Field Name	Description
testIdx	Test case index array to simulate or execute from dataFile. If testIdx = [], all test cases are simulated or executed.
signalLoggingSaveFormat	Available only for sldvruntest. Specifies the format of signal logging data for signals that connect to the outport of the model and for intermediate signals that are configured for logging. If you specify Dataset, data is stored in the Simulink.SimulationData.Dataset objects. **Default:** 'Dataset'
coverageEnabled	Available only for sldvruntest. If true, the Simulink Coverage software collects model coverage data during simulation. **Default:** false
coverageSetting	Available only for sldvruntest. cvtest object to use for collecting model coverage. If coverageSetting is [], sldvruntestopts returns the coverage settings for the model specified in the call to sldvruntest. **Default:** []

Field Name	**Description**
`allowCopyModel`	Available only for `sldvruncgvtest`. If you have not configured `sldvruncgvtest` to execute test cases with the CGV API, `allowCopyModel` specifies creating and configuring the model. If `true` and you have not configured the `model` to execute test cases with the CGV API, `sldvruncgvtest` copies the model, fixes the configuration, and executes the test cases on the copied model. If `false` (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, `sldvruncgvtest` does not copy the model, even if `allowCopyModel` is `true`. An error occurs.
`cgvComType`	Available only for `sldvruncgvtest`. Defines the software-in-the-loop (SIL) or processor-in-the-loop (PIL) approach for CGV: • `'topmodel'` (default) • `'modelblock'`
`cgvConn`	Available only for `sldvruncgvtest`. Specifies mode of execution for CGV: • `'sim'` (default) • `'sil'` • `'pil'`
`fastRestart`	Available only for `sldvruntest`. If `true`, Simulink Design Verifier uses fast restart mode for model simulation. **Default:** `true`
`useParallel`	Available only for `sldvruntest`. If `true`, Simulink Design Verifier simulates test cases with parallel computing. This option requires a Parallel Computing Toolbox license. **Default:** `false`

Alternatives

Create a `runOpts` object for `sldvruntest` from the MATLAB Command Window.

See Also

`sldvruncgvtest` | `sldvruntest`

Topics

“Verify a Component for Code Generation”
“Creating and Executing Test Cases”
“Achieve Missing Coverage in Referenced Model”

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`.

Examples

Generate Tests and Report Coverage

Generate test cases for the `sldvdemo_cruise_control` model. Then, report coverage by using `sldvsimdata` for test case `1`.

Open the `sldvdemo_cruise_control` example model.

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

Analyze the model and generate test cases by using `sldvrun`.

```
[~,files] = sldvrun(model);
s = load(files.DataFile);
```

Generate a `Simulink.SimulationData.Dataset` object for test case `1` from the Simulink Design Verifier data variable `s`.

```
simData = sldvsimdata(s.sldvData,1);
```

Simulate and generate model coverage report for test objects.

```
cvdata = cvsim(model, 10, [], simData);
 cvhtml('Test Coverage', cvdata);
```

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

An integer that specifies the index of a test case or counterexample in a data file.

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` | `sldvruntest` | `sldvlogsignals`

Topics
“Create and Edit Signal Data”
“Import Signal Data for Root Inport Mapping”

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` is 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 by the `sldv.test` function and by the Test Objective block.

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

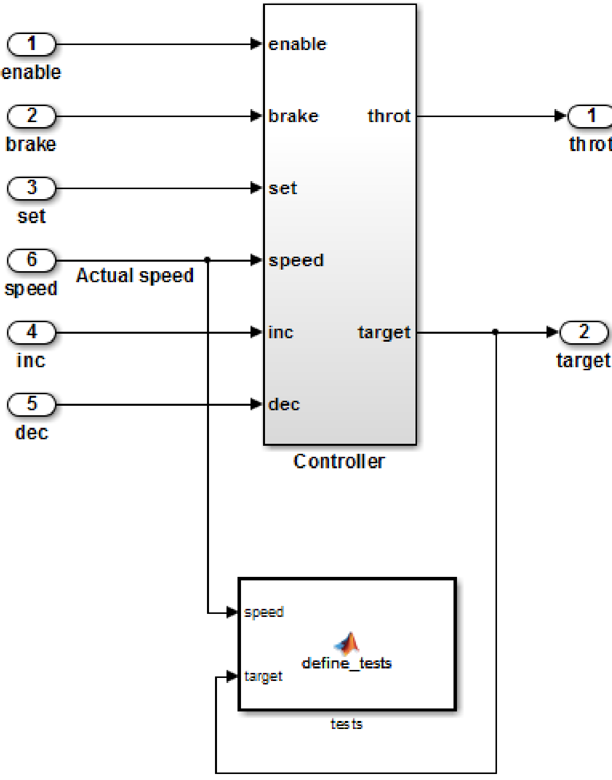
From the User-Defined Functions library, add a MATLAB Function block.

- 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);
  ```
- In the **Editor** tab, click **Save** 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_sldvdemo_cruise_control` model.



To generate test cases, on the **Design Verifier** tab, click **Generate Tests**.

Input Arguments

`expr` — Boolean expression for tests

boolean expression

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

Alternatives

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

See Also

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

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`. If timer optimizations are recorded, the function returns a `status` of `1`. If timer optimizations are not recorded, the function returns a `status` of `0`. If `sldvdata` does not have information about the timer optimizations, the function returns a `status` of `-1`.

`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. If timer optimizations are recorded, the function returns a `status` of `1`. If timer optimizations are not recorded, the function returns a `status` of `0`. If `sldvdata` does not have information about timer optimizations, the function returns a `status` of `-1`.

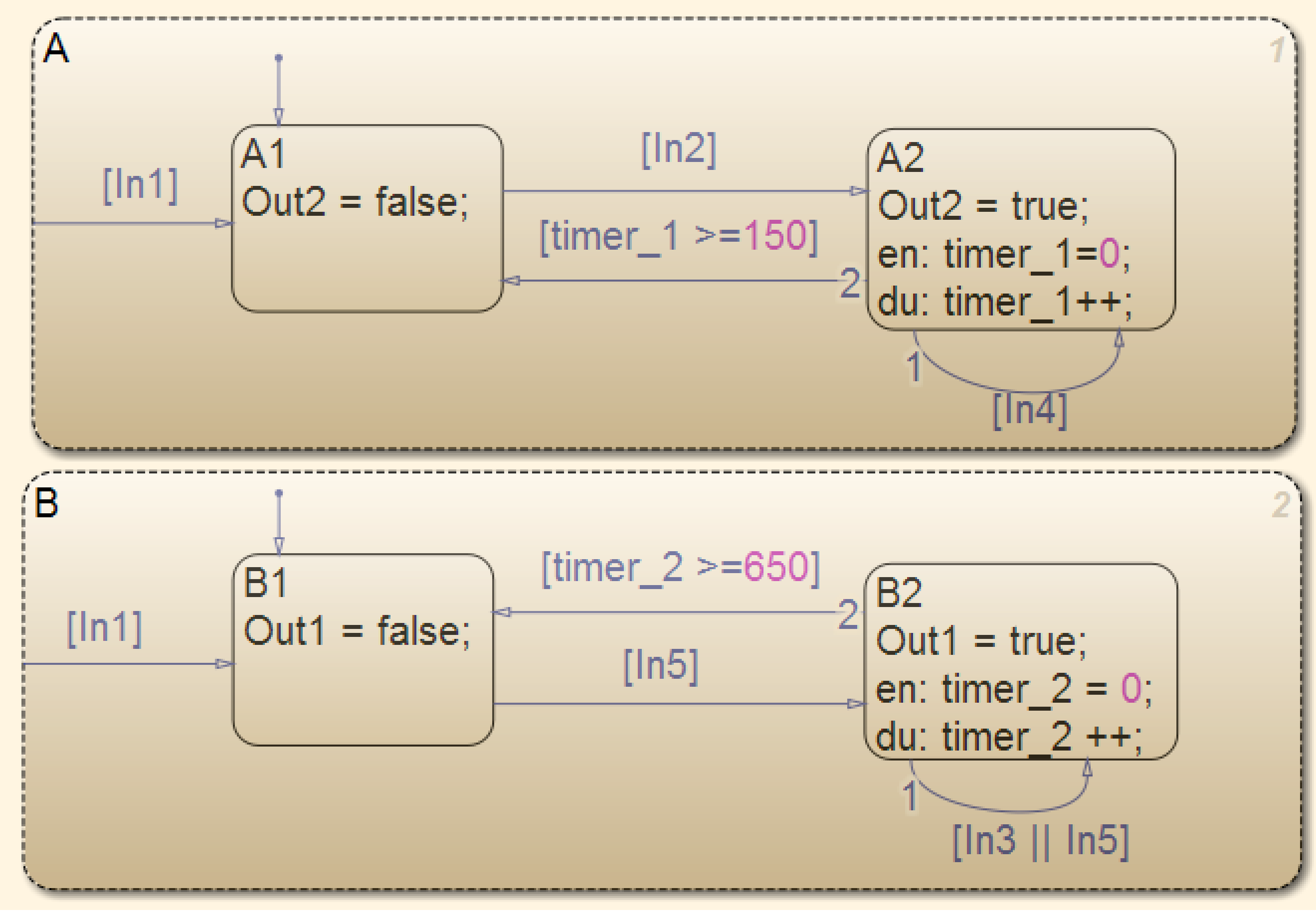
`status = sldvtimer(model)` displays timer patterns in the `model` that you can optimize for Simulink Design Verifier test generation.

Examples

Optimize Model Timers

Using the `sldvtimer` function to optimize model timers, increase the number of test generation objectives met during Simulink Design Verifier Test Generation analysis.

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



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

- The Simulink Design Verifier log dialog box displays that:
  - `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.

To enable the timer optimizations, in the MATLAB Command Window, enter:

```
sldvtimer(1)
```

To generate test cases, on the **Design Verifier** tab, in the **Mode** section, select **Test Generation**, then click **Generate Tests**.

The test generation analysis completes and the software generates the test cases.

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) not to identify model objects that are part of recognized timer patterns

`model` — Handel to a model
[ ] (default) | character vector | string scalar

Handle to a Simulink model

Output Arguments

`status` — Outcome of recorded timers
integer

Outcome of attempt to record timer optimization, which returns an integer value.

Logical Value Returned	Status of Timer Optimizations
`1`	Timer optimizations recorded.
`0`	Timer optimizations not recorded.
`-1`	`sldvdata` does not have information about timer optimizations.

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

`sldvruncgvtest` | `sldvruntest` | `sldvruntestopts`

Topics
“Counters and Timers”

Introduced in R2012a

sldvchecksum

Returns checksum of model

Syntax

```
checksum = sldvchecksum(model)
checksum = sldvchecksum(system)
```

Description

`checksum = sldvchecksum(model)` returns the `checksum` of the model passed as the input argument. The checksum represents the state of the model being analyzed.

`checksum = sldvchecksum(system)` returns the `checksum` for the specified component `system`. The component `system` must be an atomic subsystem or a Model block.

Examples

Compare Checksum

You can compute the checksum of model and compare it with an updated model checksum. You can also compare the checksum of a model with the checksum provided in the Simulink Design Verifier analysis report.

Compare this checksum with the checksum in the Simulink Design Verifier analysis report by using:

```
open_system ('sldvdemo_cruise_control');
[checksum] = sldvchecksum('sldvdemo_cruise_control');
```

Open the same `sldvdemo_cruise_control` model and make changes to it. Save the updated model. Repeat the previous step to calculate the checksum, then compare the checksums of the original model and the updated model.

Input Arguments

`model` — Handle to model
character vector | string scalar

Handle to a Simulink model, specified as a character vector or string scalar.

Example: `sldvdemo_cruise_control`

`system` — Handle to a component block
character vector | string scalar

Handle to an atomic subsystem or Model block in a Simulink model, specified as a character vector or string scalar.

Output Arguments

`checksum` — 128-bit checksum value
character vector

128-bit checksum of the model, returned as a character vector.

Limitations

The checksum of the model does not change if:

- The model contains MATLAB System blocks and you modify the source code of the System object™ to which the MATLAB System blocks refer.
- You modify the custom block replacement rules for the model.

Note The checksum of a single model may not match in different MATLAB versions due to internal changes.

See Also

`sldvoptions` | "Simulink Design Verifier Reports"

Topics
"Generate Test Cases for Model Decision Coverage"

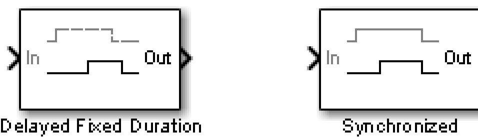
Introduced in R2021a

2

Blocks

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.

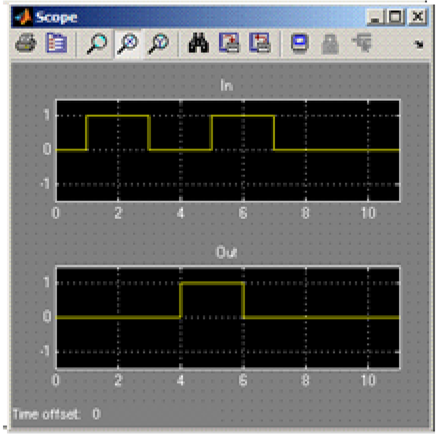
Delayed Fixed Duration

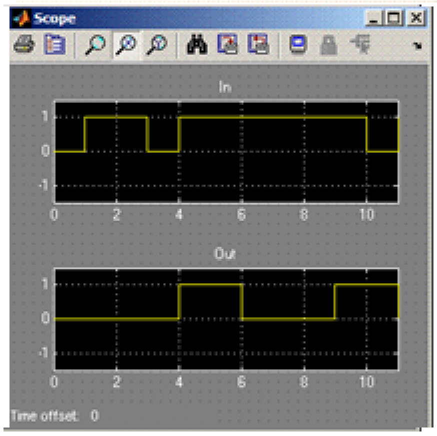
In this example, with **Output type** set to `Delayed Fixed Duration`, the input detection phase does not continue during the output signal construction. The following block parameters for the Detector block are set as follows:

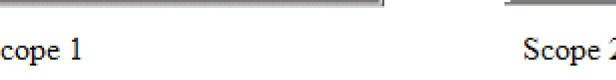
- **Time steps for input detection** = 2
- **Time steps for delay (optional)** = 1
- **Time steps for output duration** = 2

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.





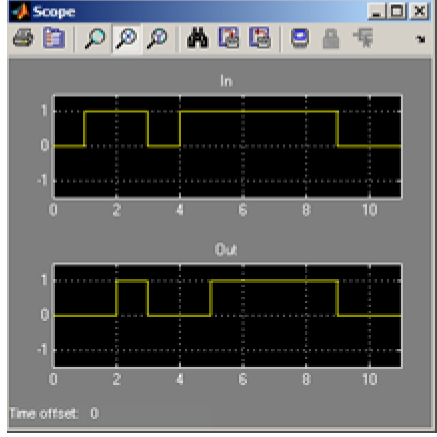


Scope 1 Scope 2

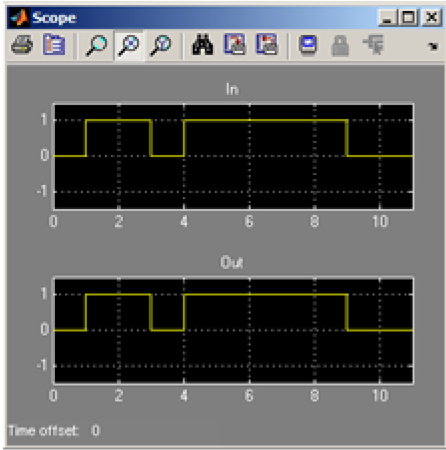
Synchronized

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

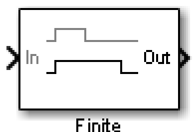
See Also

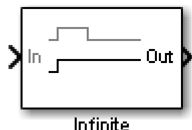
Extender, Within Implies

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`.

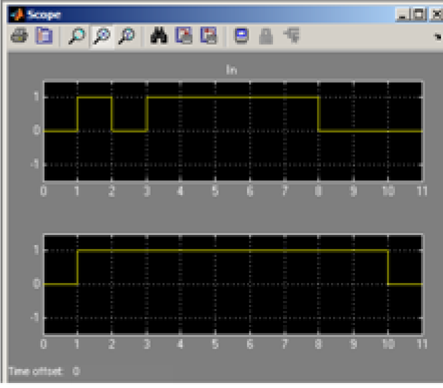
Examples

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.



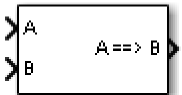
See Also

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.

Data Type Support

The Implies block accepts signals of all numeric 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”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

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”.

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”. The block does not support complex input signals.

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 2-8).

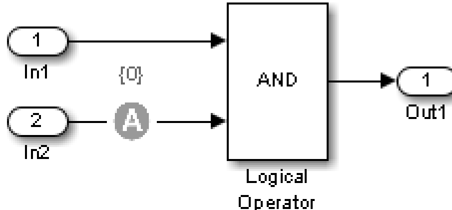
Display values

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

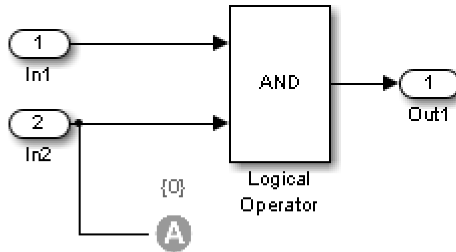
Pass through style (show Outport)

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 Outport): Selected



Pass through style (show Outport): 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”.

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”. The block does not support complex input signals.

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 2-11).

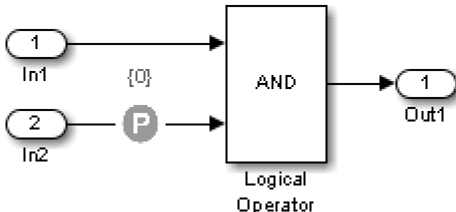
Display values

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

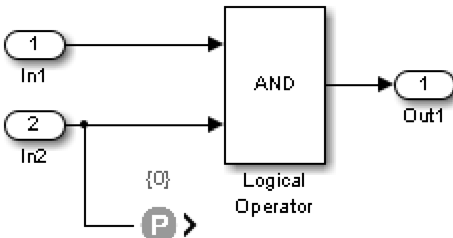
Pass through style

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 / Objectives and Constraints

Description

The Test Condition block defines test conditions for signals in models that are operating in test generation mode. (See “What Is Test Case Generation?”). Use the **Values** parameter to specify constraints on signal values. The block applies the specified **Values** parameter to its input signal and attempts to produce test cases.

You can use the block to:

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

When you generate code using Simulink and Simulink Coder, the Test Condition block is ignored. The Test Condition block does not modify the signal during model simulation, and only uses the signal to generate test conditions.

Ports

Input

`Port_1` — Input signal specified as a scalar or cell array
scalar | cell array

The Test Condition block accepts signals of all built-in data types supported by the Simulink software. For more information about supported data types, see “Data Types Supported by Simulink”. The block does not support complex input signals.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Output

`Port_1` — Output signal returned as a scalar or a cell array
scalar | cell array

The Test Condition block returns signals of all built-in data types supported by the Simulink software. For more information about supported data types, see “Data Types Supported by Simulink”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Dependencies

To enable this port, select **Pass through style (show Outport)**.

Parameters

`Enable` — Whether block is enabled

`on` (default) | `off`

Select this parameter to generate tests for a model. Clearing this option disables the block. If this option is cleared, the block appears dimmed in the Simulink Editor.

Type — Whether the block behaves as a Test Condition or Proof Assumption block

`Test Condition` (default) | `Assumption`

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` — Test condition

scalar | cell array

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 more information, see "Define Constraint Values for Parameters in MATLAB Code Files".

Alternatively, you can specify scalar values using the `Sldv.Point` constructor, which accepts a single value as its argument. You can also specify intervals using the `Sldv.Interval` constructor, which requires two input arguments, one for the lower bound and one for the upper bound for the interval.

For 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 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`.

`Display values` — Display the contents of values parameter
on (default) | `off`

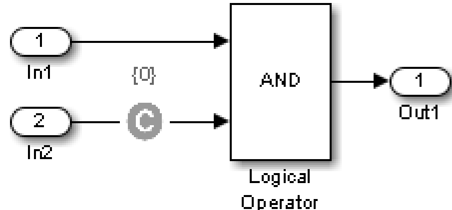
Specify whether the block displays the contents of its **Values** parameter in the Simulink Editor.

`Pass through style(show Outport)` — Whether block displays an output port in the Simulink Editor
on (default) | `off`

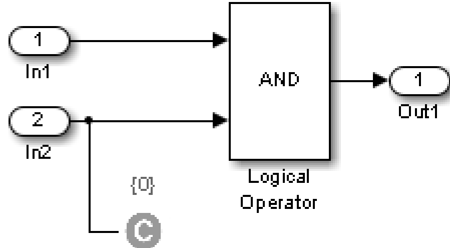
If selected, the block displays its output port, which allows the input signal to pass through to the block output. If not selected, the block hides its output port and terminates the input signal.

Example: The following graphics illustrate the appearance of the block in each case.

Pass through style: Selected



Pass through style: Cleared



See Also

Proof Assumption | Test Objective

Topics
"Test Condition Block"

Introduced in R2007a

Test Objective

Define custom objectives that signals must satisfy in test cases

Library: Simulink Design Verifier / Objectives and Constraints

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.

Ports

Input

`Values` — Input signal

scalar | cell array

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”. The block does not support complex input signals.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

`Enable` — Specify whether the block is enabled

on (default) | `off`

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 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

scalar | cell array

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”.

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`.

`Display values` — Display the contents of values parameter

`on` (default) | `off`

Specify whether the block displays the contents of its **Values** parameter in the Simulink Editor.

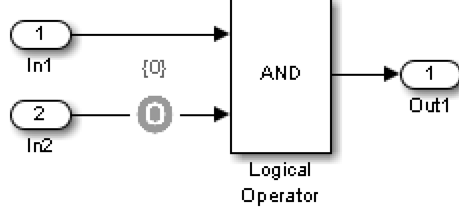
`Pass through style` — Specify whether the block displays an output port in the Simulink Editor

`on` (default) | `off`

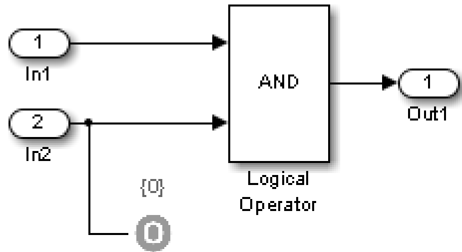
If selected, the block displays its output port, allowing its input signal to pass through as the block output. If not selected, the block hides its output port and terminates the input signal.

Example: The following graphics illustrate the appearance of the block in each case.

Pass through style (show Outport): Selected



Pass through style (show Outport): Deselected



See Also

Proof Objective | Test Condition

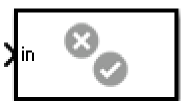
Topics

"Test Objective Block"

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" (Embedded 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.

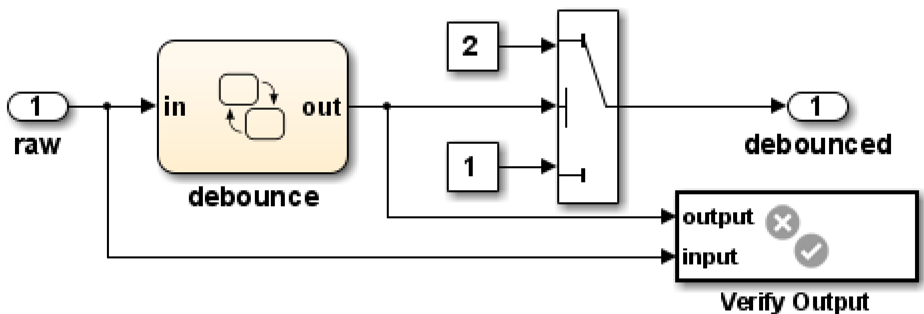
The Verification Subsystem block in the Simulink Design Verifier library is preconfigured to work with the Simulink Design Verifier software. A Verification Subsystem block must:

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

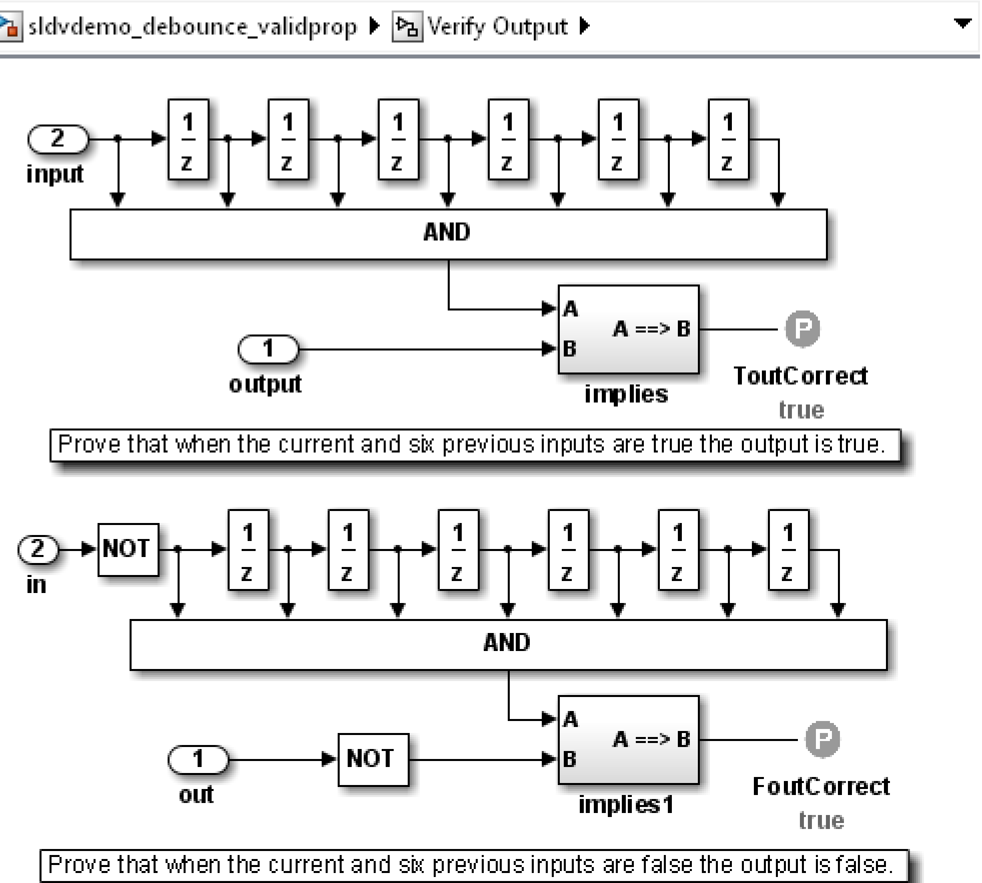
If you alter the Verification Subsystem block so that the preceding conditions are not met, the Simulink Design Verifier software displays a warning.

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.



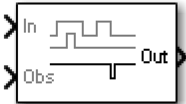
See Also

- Implies
- Within Implies
- Proof Assumption
- Proof Objective
- Test Condition
- Test Objective
- Subsystem block in the Simulink documentation
- “Create Subsystems”

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

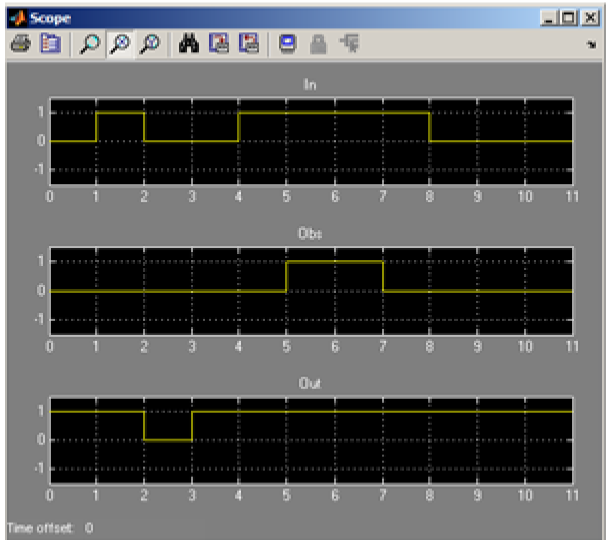
Specify whether the block observation of Obs can be reset by an external Boolean reset signal.

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 3-3
“Detect Out Of Bound Array Access” on page 3-5
“Detect Division by Zero” on page 3-6
“Detect Integer Overflow” on page 3-7
“Detect Non-finite and NaN Floating-Point Values” on page 3-8
“Detect Subnormal Floating-Point Values” on page 3-8
“Detect Specified Minimum and Maximum Value Violations” on page 3-9
“Detect Data Store Access Violations” on page 3-10
“Detect Block Input Range Violations” on page 3-11
“Check usage of rem and reciprocal operations” on page 3-12
“Check usage of Sqrt operations” on page 3-13
“Check usage of log and log10 operations” on page 3-13
“Check usage of Reciprocal Sqrt blocks” on page 3-13

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 Advisor Checks” and “Automate Model Advisor Check Execution”.

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	Avoid using the following unsupported software features or Simulink blocks in the model or model component that you want to analyze: • “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	• 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 and Review Results”
- “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: • “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”.
Dead logic found in model	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 **Apps > Design Verifier > Settings**. In the **Configuration Parameters** window, from **Design Verifier > Design Error Detection** pane, select both **Dead logic** and **Identify active logic**.
Dead logic not found in model	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 **Apps > Design Verifier > Settings**. In the **Configuration Parameters** window, from **Design Verifier > Design Error Detection** pane, select both **Dead logic** and **Identify active logic**.

See Also

- MISRA C:2012: Rule 2.1
- CERT C, MSC07-C
- CWE, CWE-561
- “Run Model Advisor Checks”
- “Secure Coding Standards” (Embedded Coder)
- “Detect Dead Logic Caused by an Incorrect Value”
- “Design Verifier Pane: 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 • “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 Out of Bound Array Access Errors”

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 • “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 Integer Overflow and Division-by-Zero Errors”

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 • “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”.
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 Integer Overflow and Division-by-Zero Errors”

Detect Non-finite and NaN Floating-Point Values

Check ID: `mathworks.sldv.infnan`

Detects Nonfinite and NaN floating-point values in your model

Description

This check detects the occurrences of nonfinite and NaN floating-point values in your model.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See • “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”.
Nonfinite and NaN floating-point values found in model	To view the conditions that cause the occurrence of nonfinite and NaN floating-point values, 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

- “Design Error Detection”
- “Detect Non-Finite, NaN, and Subnormal Floating-Point Values”

Detect Subnormal Floating-Point Values

Check ID: `mathworks.sldv.subnormal`

Detects subnormal floating-point values in your model

Description

This check detects the occurrences of subnormal floating-point values in your model.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See • "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".
Subnormal floating-point values found in model	To view the conditions that cause the occurrence of subnormal floating-point values, 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

- "Design Error Detection"
- "Detect Non-Finite, NaN, and Subnormal Floating-Point Values"

Detect Specified Minimum and Maximum Value Violations

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

<table>
<tr><th>Result</th><th>Recommended Action</th></tr>
<tr><td>Failed, model incompatible</td><td>Resolve the model incompatibility. See
“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”.</td></tr>
<tr><td>Violation of minimum and/or maximum found in model</td><td>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.</td></tr>
</table>

See Also

- MISRA C:2012: Directive 4.1
- CERT C, API00-C
- CWE, CWE-628
- “Secure Coding Standards” (Embedded Coder)
- “Design Range Checks”
- “Check for Specified Minimum and Maximum Value Violations”

Detect Data Store Access Violations

Check ID: `mathworks.sldv.dsmaccessviolations`

Detect data store access violations in your model.

Description

This check detects these data store access violations:

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

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See: • "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" • "Handle Incompatibilities with Automatic Stubbing".
Data store access violations found	In the Model Advisor report, click **View test case**. The software creates a harness model and the Signal Builder block displays the test case that replicates the error.

See Also

- "Design Error Detection"
- "Detecting Access Order Errors"
- "Detect Data Store Access Violations"

Detect Block Input Range Violations

Check ID: `mathworks.sldv.blockinputrangeviolations`

Detect block input range violations in your model.

Description

This check detects input range violations for blocks with these settings:

- For these blocks, when the **Diagnostic for out-of-range input** parameter is set to `Warning` or `Error`:
  - n-D Lookup Table
  - Interpolation Using Prelookup
  - Prelookup
  - Direct Lookup Table (n-D)
- Multiport Switch blocks, when the **Diagnostic for default case** parameter is set to `Warning` or `Error`.
- Trigonometric Function blocks, when the **Approximation method** parameter is set to `CORDIC`

Note The check does not flag block input range violations for n-D Lookup Table blocks, when the **Interpolation method** is set to `Akima spline` or `Cubic spline`.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See: • "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" • "Handle Incompatibilities with Automatic Stubbing"
Block input range violations found	In the Model Advisor report, click **View test case**. The software creates a harness model and the Signal Builder block displays the test case that replicates the error.

See Also

- "Design Error Detection"

Check usage of rem and reciprocal operations

Check ID: `mathworks.sldv.hismviolationshisl_0002`

Description

Identifies the usage of rem and reciprocal operations that cause non-finite results.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains `rem` or `reciprocal` operations that might result in non-finite output signals. Non-finite signals are not supported in real-time embedded systems.	When using the `rem` or `reciprocal` operation, protect the corresponding input from being equal to zero.

See Also

- hisl_0002: Usage of Math Function blocks (rem and reciprocal)

Check usage of Sqrt operations

Check ID: `mathworks.sldv.hismviolationshisl_0003`

Description

Identify Sqrt operations with inputs that can be negative.

Results and Recommended Actions

Condition	Recommended Action
One or more Sqrt operations in the model have inputs that can go negative during simulation.	Remodel to protect the input of the Sqrt operations from going negative.

See Also

- "hisl_0003: Usage of square root operations"

Check usage of log and log10 operations

Check ID: `mathworks.sldv.hismviolationshisl_0004`

Description

Identifies the log and log10 operations that cause non-finite results.

Results and Recommended Actions

Condition	Recommended Action
One or more `log` and `log10` operations used in the model might require non-finite number support, which is not supported in real-time embedded systems.	Consider protecting the input of these operations such that it is not less than or equal to zero.

See Also

- hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)

Check usage of Reciprocal Sqrt blocks

Check ID: `mathworks.sldv.hismviolationshisl_0028`

Description

Identifies Reciprocal Sqrt blocks with inputs that can go zero or negative.

Results and Recommended Actions

Condition	Recommended Action
One or more Reciprocal Sqrt blocks in the model have inputs that can go to zero or negative during simulation.	Remodel to protect the input of the Reciprocal Sqrt blocks from going negative.

See Also

- “hisl_0028: Usage of Reciprocal Square Root blocks”