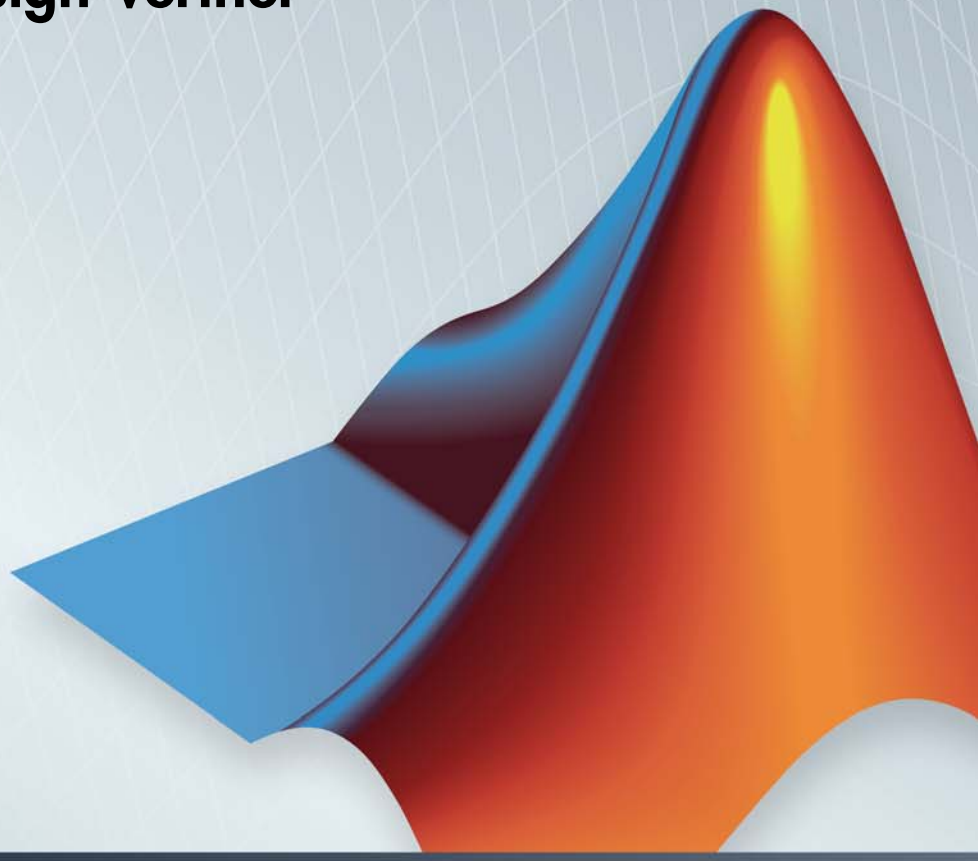
Simulink® Design Verifier™

Reference

R2014**a**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Simulink® Design Verifier™ Reference

Trademarks

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)

Contents

1

Functions — Alphabetical List

sldv.assume

Purpose

Proof assumption function for Stateflow charts and MATLAB Function blocks

Syntax

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

Description

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

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

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

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

Input Arguments

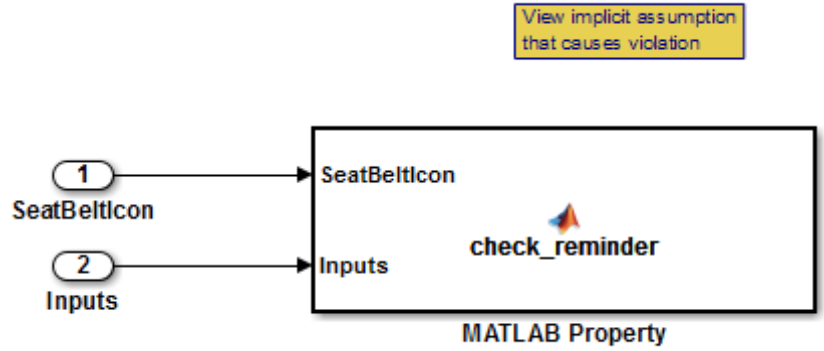
expr

MATLAB expression, for example, `x > 0`

Examples

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

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



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

Editor - Block: sldvdemo_sbr_verification/Safety Properties/MATLAB Property

Safety Properties/MATLAB Property

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

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

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

5. In the editor, save the updated code.

6. Prove the safety properties. With the model open in the Simulink® Editor, select the Safety Properties subsystem and choose **Analysis > Design Verifier > Prove Properties > Selected Subsystem**.

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

Alternatives

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

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

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

See Also

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

Tutorials

- "Prove Properties in a Model"

How To

- "Workflow for Proving Model Properties"

Purpose

Replace blocks for analysis

Syntax

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

Description

`[status, newmodel] = sldvblockreplacement(model)` copies `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. `sldvblockreplacement` returns a handle to the new model in `newmodel`. If the operation replaces the blocks, `sldvblockreplacement` returns a `status` of `1`. Otherwise, it returns `0`.

`[status, newmodel] = sldvblockreplacement(model, options)` replaces the blocks of `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.

Input Arguments

model

Handle to a Simulink model

options

`sldvoptions` object that specifies analysis parameters

Default: `[]`

showUI

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

Examples

Replace the blocks in `sldvdemo_blockreplacement_unsupportedblocks` using the block-replacement rules specified in `opts`:

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

See Also

`sldvoptions`

Tutorials

- "Replace Multiport Switch Blocks"

How To

- "Define Custom Block Replacements"

Purpose

Check model for compatibility with analysis

Syntax

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

Description

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

`status = sldvcompat(block)` converts the Simulink `block` into a temporary model and checks the compatibility of that 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 the Simulink Design Verifier software 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.

Input Arguments

model

Handle to a Simulink model

Default: `[]`

block

Handle to a block in a Simulink model

subsystem

Handle to a subsystem in a Simulink model

options

`sldvoptions` object that specifies analysis parameters

Default: `[]`

showUI

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

startCov

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

Examples

Check the `sldvdemo_flipflop` model to see if it is compatible with Simulink Design Verifier software:

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

Alternatives

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

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

See Also

`sldvoptions` | `sldvrun`

How To

- "Check Compatibility of the Example Model"

Purpose

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, as well as with the Test Condition block.

Input Arguments

expr

MATLAB expression, for example, `x > 0`

Examples

Add a test objective and test conditions:

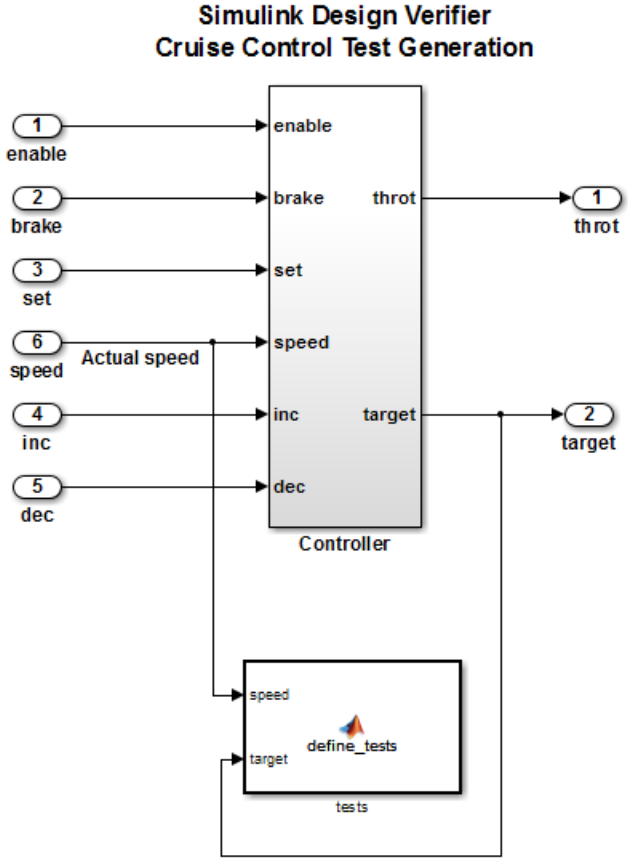
1. Open the `sldvdemo_cruise_control` model and save it as `ex_sldvdemo_cruise_control`.
2. Remove the Test Condition block for the `speed` block signal. Instead of the Test Condition block, this example uses `sldv.test` and `sldv.condition`.
3. From the User-Defined Functions library, add a MATLAB Function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

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

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

c Save the code and close the editor.

d Connect the block to the signal for the `speed` block and to the signal for the `target` block.



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

Alternatives

Instead of using the `sldv.condition` function, you can insert a Test Condition block in your model. However, 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. However, 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

Tutorials

- "Generate Test Cases for Model Decision Coverage"

How To

- "Workflow for Test Case Generation"

Purpose

Extract subsystem or subchart contents into new model for analysis

Syntax

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

Description

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

`newModel = sldvextract(subchart)` extracts the contents of the atomic subchart `subchart` and creates a model for the Simulink Design Verifier software to analyze. `subchart` should specify the full path of the Atomic Subchart. `sldvextract` uses the subchart name for the model name, appending a numeral to the model name if that model name already exists.

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

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

Input Arguments

subsystem

Full path to the atomic subsystem

subchart

Full path to the Stateflow® atomic subchart

showModel

Boolean that indicates whether to display the extracted model

Default: `True`

Output Arguments

newModel

Name of the new model

Examples

Extract the atomic subsystem, Bus Counter, from the `sldemo_mdlref_conversion` model and copy it into a new model:

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

Extract the atomic subchart, Sensor1, from the `sf_atomic_sensor_pair` model and copy it into a new model:

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

Purpose

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 `model` using the `sldvoptions` object `options`.

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

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

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

Input Arguments

block

Handle to an atomic subsystem in a Simulink model

model

Handle to a Simulink model

Default: `[]`

options

`sldvoptions` object that specifies analysis parameters

Default: `[]`

showUI

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

startCov

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

Default: `[]`

Output Arguments

cvdo

`cvdata` object containing coverage data for new tests

filenames

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

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

status

Logical value that indicates if the analysis collected model coverage

```
true
false
```

Examples

Analyze the Cruise Control model and simulate a version of that model 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);
```

See Also

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

Tutorials

- “Generate Test Cases for Model Decision Coverage”

sldvharnessopts

Purpose

Default options for `sldvmakeharness`

Syntax

```
harnessopts = sldvharnessopts
```

Description

`harnessopts = sldvharnessopts` generates the default configuration for running `sldvmakeharness`.

Output Arguments

harnessopts

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

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

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

Field	Description
`usedSignalsOnly`	When `true`, the Signal Builder block in the harness model has signals only for input signals used in the model. `model` must be compatible with the Simulink Design Verifier software to detect the used input signals. Default: `false`
`systemTestHarness`	When `true`, generates a SystemTest harness. This option requires `dataFile` path in addition to `model`. Default: `false`

Examples

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

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

See Also

`sldvmakeharness`

sldvhighlight

Purpose

Highlight model using data from Simulink Design Verifier analysis

Syntax

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

Description

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

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

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

Input Arguments

model - Name or handle of model to highlight

string | handle

Name of model to highlight, specified as a string. Or, handle of model to highlight.

Example: `'sldvdemo_cruise_control'`

Example: `'sldvdemo_flipflop'`

dataFile - Name of analysis data file

string

Name of Simulink Design Verifier analysis data file, specified as a string.

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'`

Examples

Highlight Active Analysis Results on Current Model

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

Open the `sldvdemo_debounce_modelcov` example model.

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

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

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

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

'sldvdemo_debounce_modelcov' is compatible with Simulink Design Verifier.

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

Generating output files:

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

    Harness model:
    pwd\sldv_output\sldvdemo_debounce_modelcov\ ...
```

```
  sldvdemo_debounce_modelcov_harness.mdl

Results generation completed.

status =

     1
```

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

```
sldvhighlight
```

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

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

Highlight Active Analysis Results on Specified Model

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

Open the `sldvdemo_debounce_modelcov` example model.

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

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

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

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

'sldvdemo_debounce_modelcov' is compatible with Simulink Design Verifier.

Generating tests...
```

```
...................
Completed normally.

Generating output files:

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

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

Results generation completed.

status =

     1
```

Highlight the results of the analysis on `sldvdemo_debounce_modelcov`.

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

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

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

Highlight Analysis Results from Data File on Specified Model

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

Open the `sldvdemo_debounce_modelcov` example model.

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

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

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

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

'sldvdemo_debounce_modelcov' is compatible with Simulink Design Verifier.

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

Generating output files:

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

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

Results generation completed.

status =

     1
```

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

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

Reopen the example model.

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

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

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

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

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

See Also

`sldvloadresults` | `sldvreport`

Concepts

- "Highlighted Results on the Model"
- "Simulink Design Verifier Data Files"

sldvisactive

Purpose

Check if Simulink Design Verifier software is updating block diagram

Syntax

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

Description

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

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

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

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

Input Arguments

model

Full path name or handle to a Simulink model

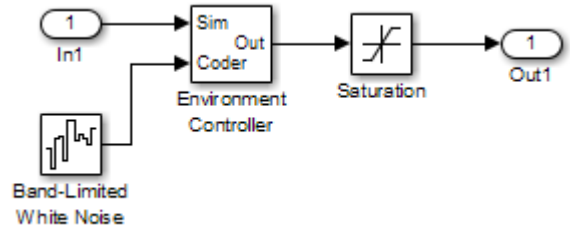
block

Full path name or handle to a Simulink block

Examples

Eliminate blocks that are incompatible with the Simulink Design Verifier software:

1. Create a Simulink model and save it as `ex_environment_controller`.



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

3 Click the **Initialization** tab and add the following command, if it does not exist:

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

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

- Sets the pass-through mode to pass the `Sim` signal to the output port.
- Eliminates the `Coder` port, which is incompatible with the Simulink Design Verifier software.

4 Save the changes to the `ex_environment_controller` model.

sldvloadresults

Purpose

Load Simulink Design Verifier analysis results for model

Syntax

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

Description

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

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

Input Arguments

model - Name or handle of model for which to load analysis results

string | handle

Name of model for which to load analysis results, specified as a string. 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

string

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

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

when you specify `dataFile`, choose a data file that contains results from the same version of `model`.

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

Example: `'results.mat'`

Example: `'sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat'`

Example: `'sldv_output\my_model\my_model_sldvdata.mat'`

Output Arguments

status - Outcome of attempt to load results

logical

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

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

Examples

Load Active Results for Specified Model

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

Open the `sldvdemo_flipflop` example model.

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

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

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

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

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

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

Generating output files:

    Data file:
    pwd\sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat

Results generation completed.

status =

     1
```

Close the example model.

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

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

```
sldvloadresults('sldvdemo_flipflop')

ans =

      1
```

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

Load Results from Data File for Specified Model

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

Open the `sldvdemo_flipflop` example model.

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

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

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

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

'sldvdemo_flipflop' is compatible with Simulink Design Verifier.

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

Generating output files:

    Data file:
    pwd\sldv_output\sldvdemo_flipflop\sldvdemo_flipflop_sldvdata.mat

Results generation completed.

status =

     1
```

Close the example model.

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

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

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

```
ans =

     1
```

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

See Also

`sldvhighlight` | `sldvreport`

Concepts

- "Review Analysis Results"
- "Simulink Design Verifier Data Files"

Purpose

Log simulation input port values

Syntax

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

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

Description

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

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

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

Input Arguments

model_block

Full block path name or handle to a Simulink Model block

harness_model

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

test_case_index

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

Output Arguments

data

Structure that contains the logged data

Examples

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

1. Simulate the CounterB Model block, which references the `sldemo_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');
   ```

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

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

How To

- “Extend Test Cases for Model with Temporal Logic”
- “Extend Test Cases for Closed-Loop System”

Purpose

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 `model`, which is a handle to a Simulink model or a string with 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` using the `dataFile` and `harnessOpts`, which specifies the harness creation options. Requires `''` for `dataFile` if `dataFile` is not available.

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

Input Arguments

model

Handle to a Simulink model or a string with the model name

dataFile

Name of the `sldvData` file.

Default: `''`

harnessOpts

A structure whose fields specify the configuration for sldvmakeharness:

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` **Note** If your model contains bus objects and you set `modelRefHarness` to `true`, in the **Configuration Parameters > Diagnostics > Connectivity** pane, you must set the **Mux blocks used to create bus signals** parameter to `error`. For more information, see "Prevent Bus and Mux Mixtures".

Field	Description
`usedSignalsOnly`	When `true`, the Signal Builder block in the harness model has signals only for input signals used in the model. `model` must be compatible with the Simulink Design Verifier software to detect the used input signals. Default: `false`
`systemTestHarness`	When `true`, generates a SystemTest harness. This option requires `dataFile` path in addition to `model`. Default: `false`

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

Output Arguments

savedHarnessFilePath

String containing the path and file name of the generated harness model

Examples

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

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

Alternatives

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

1. In the Simulink Editor, select **Analysis > Design Verifier > Options**.

The Configuration Parameters dialog box opens. Under the **Select** tree, the **Design Verifier** node is expanded.

2. Under **Design Verifier**, select the **Results** node. Under **Harness model options**, set the desired options.

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

4. In the Simulink Editor, select **Analysis > Design Verifier > Generate Tests** to run a test-generation analysis.

See Also

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

Purpose

Merge test cases and initializations into one harness model

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

Syntax

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

Description

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

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

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

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

Input Arguments

name

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

models

A cell array of strings that represent harness model names

initialization_commands

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

Output Arguments

status

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

Examples

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

```
model = 'sldvdemo_cruise_control';
open_system(model)
% Collect decision coverage
opts1 = sldvoptions;
opts1.Mode = 'TestGeneration';
opts1.ModelCoverageObjectives = 'Decision';
opts1.HarnessModelFileName = 'first_harness';
opts1.SaveHarnessModel = 'on';
sldvrun(model, opts1);
% Collect full coverage
opts2 = sldvoptions;
opts2.Mode = 'TestGeneration';
opts2.ModelCoverageObjectives = 'ConditionDecision';
opts2.HarnessModelFileName = 'second_harness';
opts2.SaveHarnessModel = 'on';
sldvrun(model, opts2);
% Merge the two harness files:
status = sldvmergeharness('new_harness_model', {'first_harness',...
  'second_harness'});
```

See Also

`sldvgencov` | `sldvmakeharness` | `sldvrun`

sldvoptions

Purpose

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`.

Input Arguments

model

Name or handle to a Simulink model

Output Arguments

options

The following table describes the parameters that comprise a Simulink Design Verifier options object.

Parameter	Description	Values

`Assertions`	Specify whether Assertion blocks in your model are enabled or disabled.	`'EnableAll'`
		`'DisableAll'`
		`'UseLocalSettings'` (default)

`AutomaticStubbing`	Specify whether or not Simulink Design Verifier software should ignore unsupported blocks and functions and proceed with the analysis.	`'on'` (default) `'off'`

Parameter	Description	Values
`BlockReplacement`	Specify whether the Simulink Design Verifier software replaces blocks in a model before its analysis. When set to `'on'`, this parameter enables `BlockReplacementModelFileName` and `BlockReplacementRulesList`.	`'on'` `'off'` (default)
`BlockReplacementModelFileName`	Specify a folder and file name for the model that results after applying block replacement rules. This parameter is enabled when `BlockReplacement` is set to `'on'`.	string `'$ModelName$_replacement'` (default)
`BlockReplacementRulesList`	Specify a list of block replacement rules that the Simulink Design Verifier software executes before its analysis. This parameter is enabled when `BlockReplacement` is set to `'on'`.	string `'<FactoryDefaultRules>'` (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'`.	string `''` (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. This parameter is enabled when `SaveDataFile` is set to `'on'`.	string `'$`*`ModelName`*`$_sldvdata'` (default)
`DesignMinMaxCheck`	Specify whether to check that the intermediate and output signals in your model are within the range of user-specified minimum and maximum constraints. **Note** This parameter is disabled when `DetectDeadLogic` is set to `'on'`.	`'on'` `'off'` (default)

Parameter	Description	Values
`DesignMinMaxConstraints`	Specify whether or not Simulink Design Verifier software should generate test cases that consider specified minimum and maximum values as constraints for input signals in your model.	`'on'` (default) `'off'`
`DetectDeadLogic`	Specify whether to analyze your model for dead logic. **Note** When set to `'on'`, this parameter disables `DetectDivisionByZero`, `DetectIntegerOverflow`, `DetectOutOfBounds`, and `DesignMinMaxCheck`.	`'on'` `'off'` (default)
`DetectDivisionByZero`	Specify whether to analyze your model for division-by-zero errors. **Note** This parameter is disabled when `DetectDeadLogic` is set to `'on'`.	`'on'` (default) `'off'`

Parameter	Description	Values
`DetectIntegerOverflow`	Specify whether to analyze your model for integer and fixed-point data overflow errors. **Note** This parameter is disabled when `DetectDeadLogic` is set to `'on'`.	`'on'` (default) `'off'`
`DetectOutOfBounds`	Specify whether to analyze your model for out of bound array access errors. **Note** This parameter is disabled when `DetectDeadLogic` is set to `'on'`.	`'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'`
`DisplayResultsOnModel`	Specify whether to display analysis results by highlighting the model and providing context-sensitive details about the results.	`'on'` `'off'` (default)

Parameter	Description	Values
`DisplayUnsatisfiable-Objectives`	Specify whether to display warnings if the analysis detects unsatisfiable test objectives. This parameter is enabled when `Mode` is set to `'TestGeneration'`.	`'on'` `'off'` (default)
`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'`.	string `''` (default)

Parameter	Description	Values
	Note When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with `ExistingTestFile` and apply parameter configurations with `ParametersConfigFileName` or the Parameter Configuration table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.	

Parameter	Description	Values
`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)

Parameter	Description	Values
	Note When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with `ExistingTestFile` and apply parameter configurations with `ParametersConfigFileName` or the Parameter Configuration table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.	
`HarnessModelFileName`	Specify a folder and file name for the harness model. This parameter is enabled when `SaveHarnessModel` is set to `'on'`.	string `'$ModelName$_harness'` (default)

Parameter	Description	Values
`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'`
`MakeOutputFilesUnique`	Specify whether the Simulink Design Verifier software makes its output file names unique by appending a numeric suffix.	`'on'` (default) `'off'`
`MaxProcessTime`	Specify the maximum time (in seconds) that the Simulink Design Verifier software spends analyzing a model.	double `'300'` (default)

Parameter	Description	Values
`MaxTestCaseSteps`	Specify the maximum number of simulation steps the Simulink Design Verifier software takes 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 complete in less time. When you set a larger value, the bounded parts of the analysis take longer, but it is possible for these parts of the analysis to generate longer test cases. To achieve the best performance, set the `MaxTestCaseSteps` parameter to a value just large enough to bound the longest required test case, even if the test cases that are ultimately generated are longer than this value.	int32 `'500'` (default)

Parameter	Description	Values
	Note 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. When this happens, the analysis applies the `MaxTestCaseSteps` parameter to each individual iteration of test generation. This parameter is enabled when `Mode` is set to `'TestGeneration'`.	
`MaxViolationSteps`	Specify the maximum number of simulation steps over which the Simulink Design Verifier software searches for property violations. This parameter is enabled when`Mode` is set to `'PropertyProving'` and when `ProvingStrategy` is set to `'FindViolation'` or `'ProveWithViolationDetection'`.	int32 `'20'` (default)

Parameter	Description	Values

`Mode`	Specify the analysis mode for the Simulink Design Verifier software.	`'TestGeneration'` (default)
		`'PropertyProving'`
		`'DesignErrorDetection'`

`ModelCoverageObjectives`	Specify the type of model coverage that the Simulink Design Verifier software attempts to achieve.
	
	Note When `ModelCoverageObjectives` is set to `'MCDC'`, the Simulink Design Verifier software automatically enables every coverage objective for decision coverage and condition coverage as well. Similarly, enabling coverage for condition coverage causes every decision and condition coverage outcome to be enabled.
	
	This parameter is enabled when `Mode` is set to `'TestGeneration'`.
		`'None'`
		`'Decision'`
		`'ConditionDecision'` (default)
		`'MCDC'`

`ModelReferenceHarness`	Use a Model block to reference the model to run in the harness model.	`'on'` `'off'` (default)

Parameter	Description	Values
`OutputDir`	Specify a path name to which the Simulink Design Verifier software writes its output.	string `'sldv_output/$ModelName$'` (default)
`Parameters`	Specify whether the Simulink Design Verifier software uses parameter configurations when analyzing a model. When set to `'on'`, this parameter enables `ParametersConfigFile-Name`.	`'on'` `'off'` (default)
`ParametersConfigFile-Name`	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'`.	string `'sldv_params_template.m'` (default)

Parameter	Description	Values
	Note When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with `ExistingTestFile` and apply parameter configurations with `ParametersConfigFileName` or the Parameter Configuration table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.	

Parameter	Description	Values
`ParametersUseConfig`	Specify to use the Parameter Configuration table to define parameter configurations for a model. When set to `'on'`, this parameter disables `ParametersConfigFileName`. **Note** When you configure Simulink Design Verifier to treat parameters as variables in its analysis, you cannot also use the analysis to extend existing test cases. If you specify your model to extend existing test cases with `ExistingTestFile` and apply parameter configurations with `ParametersConfigFileName` or the Parameter Configuration table, when you attempt to perform Simulink Design Verifier analysis, the software reports that your model is incompatible. This occurs because the existing test cases do not include corresponding parameter values.	`'on'` `'off'` (default)

Parameter	Description	Values

`ProofAssumptions`	Specify whether Proof Assumption blocks in your model are enabled or disabled.	`'EnableAll'`
		`'DisableAll'`
		`'UseLocalSettings'` (default)

`ProvingStrategy`	Specify the strategy that the Simulink Design Verifier software uses when proving properties.	`'FindViolation'`
		`'Prove'` (default)
		`'ProveWithViolationDetection'`

`RandomizeNoEffectData`	Specify whether to use random values instead of zeros for input signals that have no impact on test or proof objectives. This parameter is enabled when `SaveDataFile` is set to `'on'`.	`'on'` `'off'` (default)
`ReportFileName`	Specify a folder and file name for the report that Simulink Design Verifier analysis generates. This parameter is enabled when `SaveReport` is set to `'on'`.	string `'$`*`ModelName`*`$_report'` (default)

Parameter	Description	Values
`ReportIncludeGraphics`	Includes screen shots of properties in the Simulink Design Verifier report. Only valid in property-proving mode. This parameter is enabled when `SaveReport` is set to `'on'` and `Mode` is set to `'PropertyProving'`.	`'on'` `'off'` (default)
`SaveDataFile`	Save the test data that the Simulink Design Verifier analysis generates to a MAT-file. When set to `'on'`, this parameter enables `DataFileName`, `SaveExpectedOutput`, and `RandomizeNoEffectData`.	`'on'` (default) `'off'`
`SaveExpectedOutput`	Simulate the model using test case signals and include the output values in the Simulink Design Verifier data file. This parameter is enabled when `SaveDataFile` is set to `'on'`.	`'on'` `'off'` (default)

Parameter	Description	Values
`SaveHarnessModel`	Create a harness model generated by the Simulink Design Verifier analysis. **Note** 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. **Note** 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)

Parameter	Description	Values
`SaveSystemTestHarness`	Save the analysis results as a SystemTest TEST-file so you can run test cases using the SystemTest capabilities. When set to `'on'`, this parameter enables `SystemTestFileName`. This parameter is enabled when `Mode` is set to `'TestGeneration'`.	`'on'` `'off'` (default)
`SystemTestFileName`	Specify a folder and file name for the SystemTest TEST-file. This parameter is enabled when `SaveSystemTestHarness` is set to `'on'`.	string `'$`*`ModelName`*`$_harness'`

`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)

Parameter	Description	Values

`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'`.
		`'CombinedObjectives'` (default)
		`'IndividualObjectives'`
		`'LargeModel'`
		`'LongTestCases'`
		`'CombinedObjectives (Nonlinear Extended)'`
		`'LargeModel (Nonlinear Extended)'`

Examples

Create an options object and set several parameters:

```
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;
```

Alternatives

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

See Also

`sldvblockreplacement` | `sldvcompat` | `sldvgencov` | `sldvrun`

sldv.prove

Purpose

Proof objective function for Stateflow charts and MATLAB Function blocks

Syntax

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

Description

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

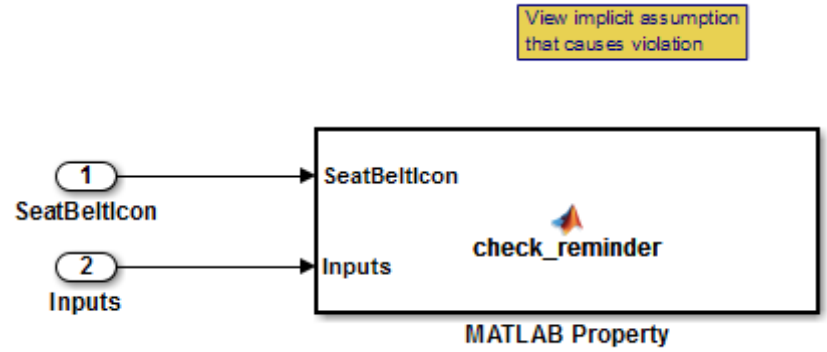
This function has no output and no impact on its parenting function, other than any indirect side effects of evaluating `expr`. If you issue this function from the MATLAB command line, the function has no effect.

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

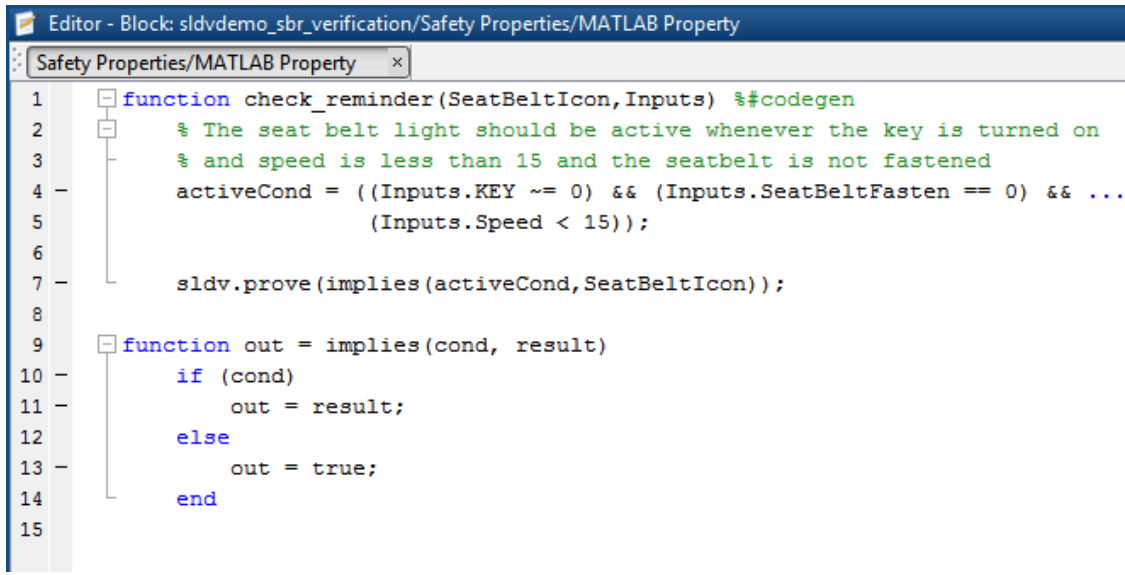
Examples

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

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



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



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

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

5 In the editor, save the updated code.

6 Prove the safety properties. With the model open in the Simulink Editor, select the Safety Properties subsystem and choose **Analysis > Design Verifier > Prove Properties > Selected Subsystem**.

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

Alternatives

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

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

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

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

See Also

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

Tutorials

- "Prove Properties in a Model"

How To

- "Workflow for Proving Model Properties"

Purpose

Generate report

Syntax

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

Description

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

`[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1, reportOption2, ...})` generates a report from `sldvDataFile` based on the specified options. `options` is a cell array of strings.

`[status, reportFilePath] = sldvreport(sldvDataFile, {reportOption1, reportOption2, ...}, reportFilePath, showUI)` generates a report and saves it in the location `reportFilePath`.

Input Arguments

sldvDataFile

Name of the data file that contains the analysis results

Default: `''`

options

Cell array of strings that specify options for the report:

`'summary'`	Include summary analysis data only
`'objectives'`	Include test objective data
`'object'`	Include data about all model objects
`'testcases'`	Include data about all generated test cases
`'properties'`	Include data about all properties proven or falsified

Default: {}

reportFilePath

The path and file name for the generated HTML report

Default: `''`

showUI

Logical value indicating where to display messages during analysis

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

Output Arguments

status

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

reportFilePath

The path and file name for the generated HTML report

Examples

Analyze the model and create the report using `sldvreport`:

```
opts = sldvoptions;                          % Create  options structure
```

```
opts.Mode = 'TestGeneration';                  % Do test-gen analysis
opts.SaveReport = 'off';                       % Don't save HTML report
open_system 'sldvdemo_cruise_control';         % Open the model
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);  %Analyze model
[ status, reportFilePath] = sldvreport(files.DataFile,...
        {'objectives', 'objects', 'testcases'} );  % Create report
```

Alternatives

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

See Also

`sldvrun`

sldvrun

Purpose

Analyze model

Syntax

```
status = sldvrun
status = sldvrun(model)
status = sldvrun(block)
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 test cases that provide model coverage or prove the model properties.

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

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

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

`[status, filenames] = sldvrun(model, options)` analyzes `model` and returns the file names the software created 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), it directs output to the MATLAB command line.

Input Arguments

model

Handle to a Simulink model

Default: `[]`

block

Handle to a block in a Simulink model

Default: `[]`

options

`sldvoptions` object specifying the analysis options

Default: `[]`

showUI

Logical value indicating where to display messages during the analysis

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

startCov

`cvdata` object specifying model coverage objects for the software to ignore

Default: `[]`

Output Arguments

filenames

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
`SystemTestFile`	SystemTest TEST-file
`Report`	HTML report with the results
`ExtractedModel`	Simulink model extracted from subsystem
`BlockReplacementModel`	Simulink model obtained after block replacements

status

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

Examples

Set `sldvoptions` parameters, open the `sldvdemo_cruise_control` model, and analyze the model using the specified options:

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

Alternatives

In the Model Editor window, select **Analysis > Design Verifier > Detect Design Errors**, **Analysis > Design Verifier > Generate Tests**, or **Analysis > Design Verifier > Prove Properties** to run a Simulink Design Verifier analysis.

See Also

`sldvcompat` | `sldvoptions` | `sldvgencov`

Tutorials

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

Purpose

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` using all 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` using test cases in `dataFile`. `runOpts` defines the options for executing the test cases. The settings in `runOpts` determine the configuration of `cgvObject`.

Tips

To run `sldvruncgvtest`, you must have a Embedded Coder® license.

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.

Input Arguments

model

Name or handle of the Simulink model to execute

dataFile

Name of the data file or a structure that contains the input data. Data can be generated either by:

- Analyzing the model using the Simulink Design Verifier software.

- Using the `sldvlogsignals` function.

runOpts

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`	Specifies to create and configure the model if you have not configured it 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.

Field Name	Description
`cgvCompType`	Defines the software-in-the-loop (SIL) or processor-in-the-loop (PIL) approach for CGV: • `'topmodel'` (default) • `'modelblock'`
`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 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

Field	Description
`yout_sldvruncgvtest`	Output signal data
`logsout_sldvruncgvtest`	Signal logging data for: • Signals connected to outports • Signals that are configured for logging on the model

Examples

Open the `sldemo_mdlref_basic` example model and log the input signals to the CounterA 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;
```

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

```
cgvObjectSim = sldvruncgvtest('sldemo_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));
```

See Also

cgv.CGV | sldvlogsignals | sldvrun | sldvruntest | sldvruntestopts

sldvruntest

Purpose

Simulate model using input data

Syntax

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

Description

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

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

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

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 using these test cases, you might see missing coverage. This issue occurs when the time values in the `dataFile` are not aligned with the current simulation time step due to numeric calculation differences. You see this issue more frequently with multirate models—models that have multiple sample times.

Input Arguments

model

Name or handle of the Simulink model to simulate

dataFile

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

runOpts

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 test cases. **Default:** `[]`
`signalLoggingSaveFormat`	Specifies signal logging data format for: • Signals connected to the outports of the model • Intermediate signals that are already configured for logging Valid values are: • `'Dataset'` (default) — `sldvruntest` stores the data in `Simulink.SimulationData.Dataset` objects. • `'ModelDataLogs'` — `sldvruntest` stores the data in `Simulink.ModelDataLogs` objects.

Field	Description
`coverageEnabled`	If `true`, specifies that the Simulink Verification and Validation software collect model coverage data during simulation. **Default:** `false`
`coverageSetting`	`cvtest` object for collecting model coverage. If `[]`, `sldvruntest` uses the existing coverage settings for `model`. **Default:** `[]`

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

Output Arguments

outData

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

Field Name	Description
`tout_sldvruntest`	Simulation time
`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

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

Examples

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

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

See Also

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

sldvruntestopts

Purpose

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`.

Output Arguments

runOpts

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

Field Name	Description
`testIdx`	Test case index array to simulate or execute from `dataFile`. If `testIdx = []`, all test cases will be simulated or executed.
`outputFormat`	Specifies format of output values: • `'TimeSeries'` (default) — `sldvruntest/sldvruncgvtest` stores the output values in time-series format. • `'StructureWithTime'` — `sldvruntest/sldvruncgvtest` stores the output values in the `Structure with time` format.

Field Name	Description
`coverageEnabled`	Available only for `sldvruntest`. If `true`, the Simulink Verification and Validation 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:** `[]`
`allowCopyModel`	Available only for `sldvruncgvtest`. Specifies to create and configure the model if you have not configured it to execute test cases with the CGV API. 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.

Field Name	Description
	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'`

Examples

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

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

Alternatives

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

See Also

`sldvruncgvtest` | `sldvruntest`

Purpose

Test objective function for Stateflow charts and MATLAB Function blocks

Syntax

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

Description

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

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

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

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

Examples

Add a test objective and test conditions:

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

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

sldv.condition(speed >= 0 && speed <= 100);
```

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

c Save the code and close the editor.

d Connect the block to the signal for the `speed` block and to the signal for the `target` block.

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

Alternatives

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

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

See Also

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

Tutorials

- "Generate Test Cases for Model Decision Coverage"

How To

- "Workflow for Test Case Generation"

Purpose

Identify, change, and display timer optimizations

Syntax

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

Description

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

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

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

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

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

Input Arguments

value

Logical value to enable timer optimizations

`true` to enable timer optimizations

`false` (default) to disable timer optimizations

sldvdata

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

display

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

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

model

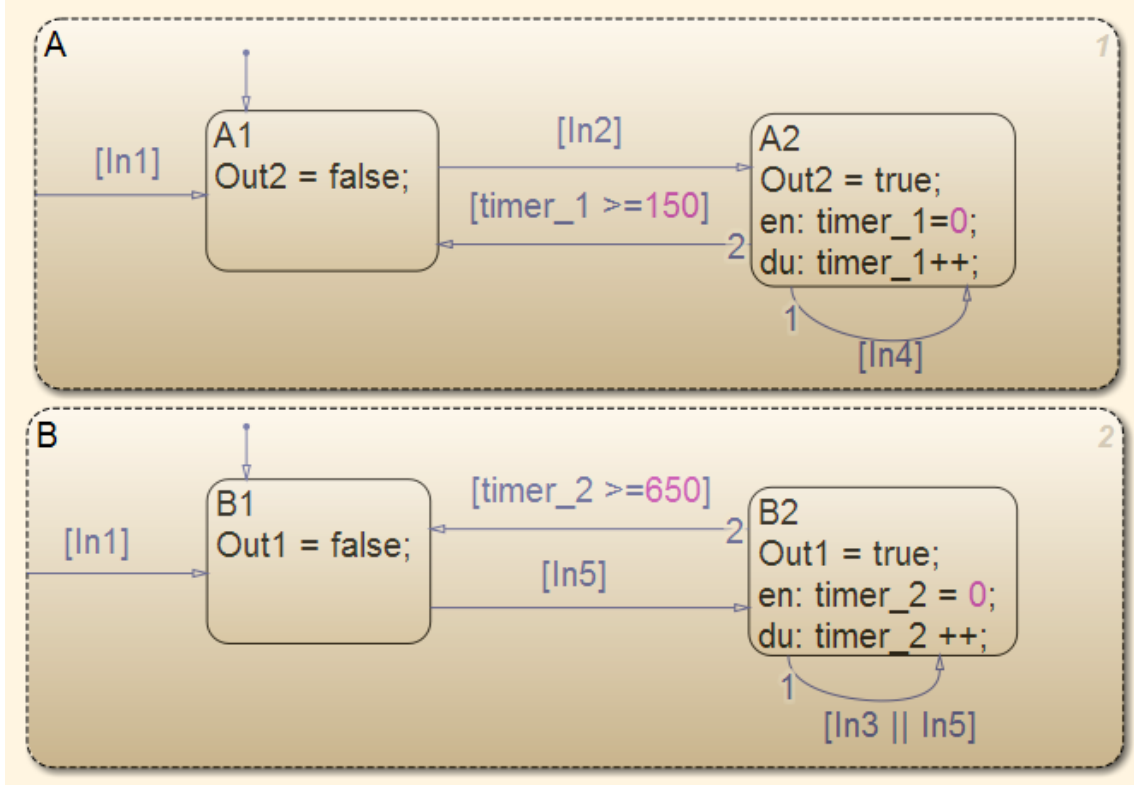
Handle to a Simulink model

Default: `[]`

Examples

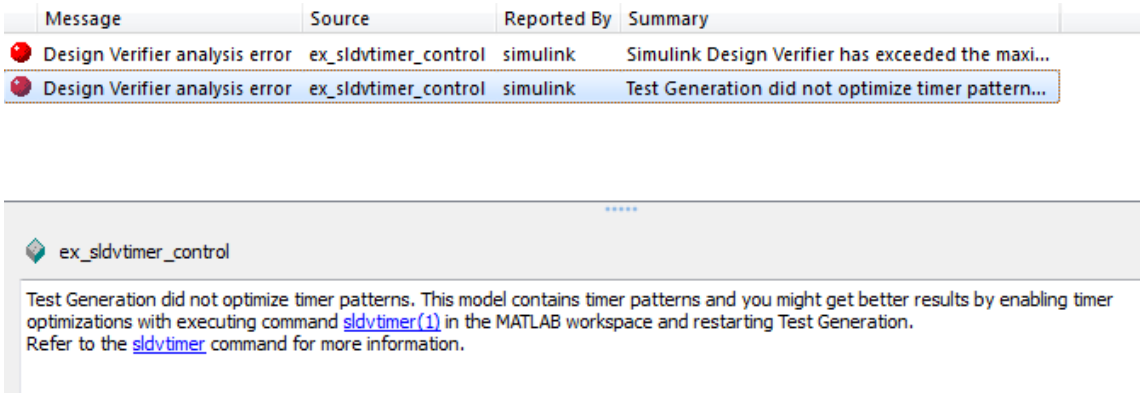
This example shows how to use the `sldvtimer` function to optimize model timers, increasing the number of test generation objectives met during Simulink Design Verifier Test Generation analysis.

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



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

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



3. In the MATLAB Command Window, enter:

   ```
   sldvtimer(1)
   ```

4. Select **Analysis > Design Verifier > Generate Tests > Model** to generate test cases again.

See Also

`sldvruncgvtest` | `sldvruntest` | `sldvruntestopts`

2

Blocks — Alphabetical List

Detector

Purpose

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

Library

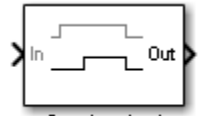
Simulink Design Verifier

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

Delayed Fixed Duration



Synchronized

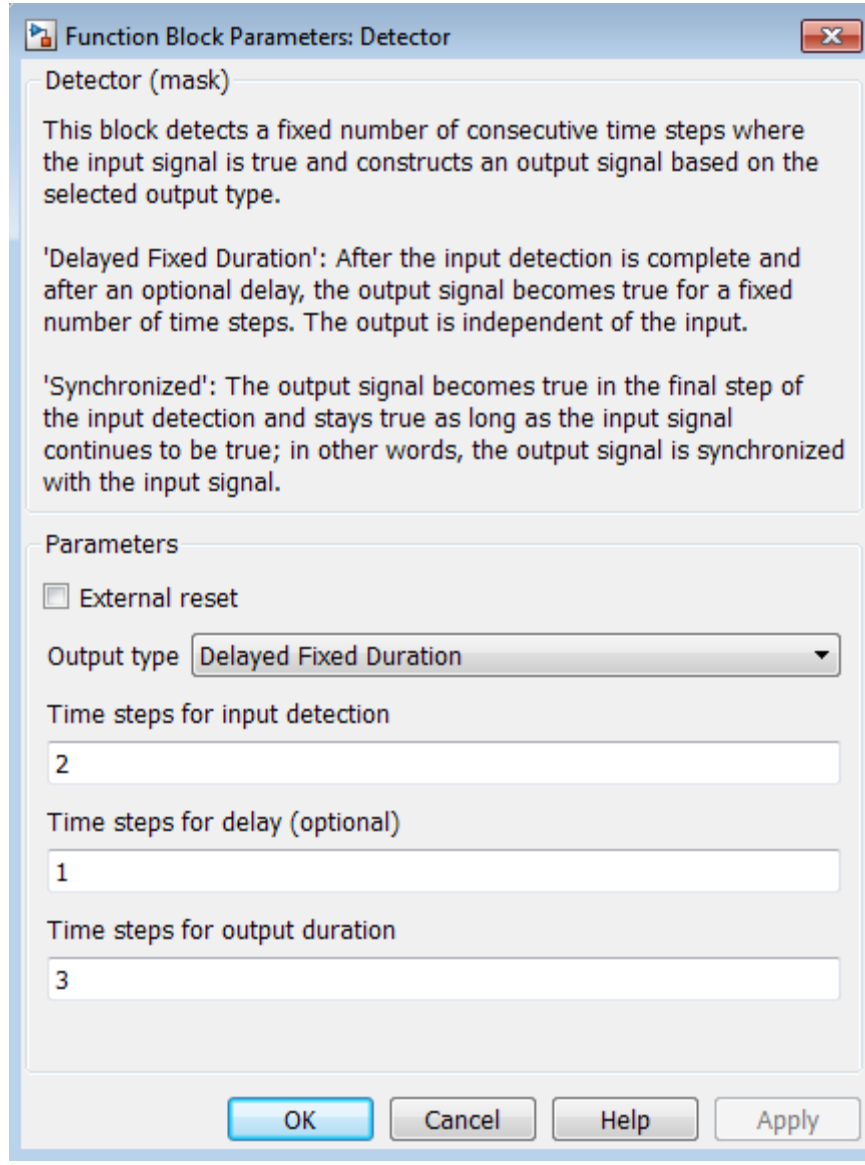
The inputs and outputs of the Detector block are of Boolean type.

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

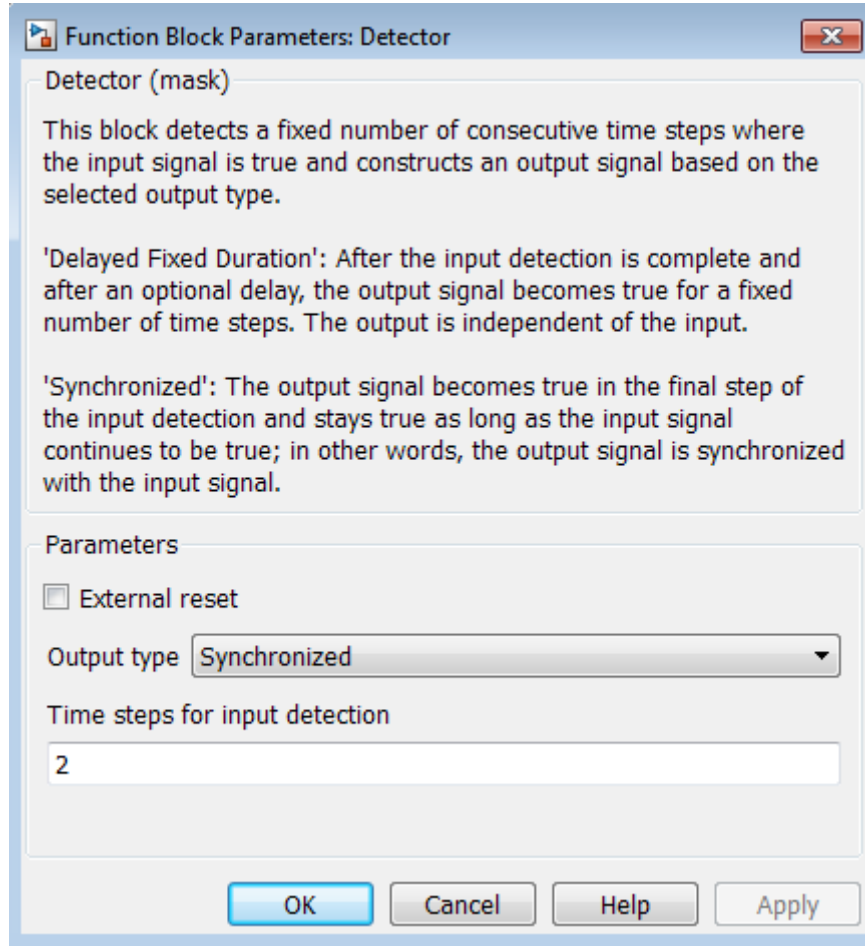
- `Delayed Fixed Duration`—After the input detection is complete and after an optional delay, the output signal becomes `true` for a fixed number of time steps. The true duration of the output is independent of the input.
- `Synchronized`—In the final time step of the input detection, the output becomes `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


Function Block Parameters: Detector
Detector (mask)
This block detects a fixed number of consecutive time steps where the input signal is true and constructs an output signal based on the selected output type.
'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 output is independent of the input.
'Synchronized': The output signal becomes true in the final step of the input detection and stays true as long as the input signal continues to be true; in other words, the output signal is synchronized with the input signal.
Parameters
External reset
Output type
Delayed Fixed Duration
Time steps for input detection
2
Time steps for delay (optional)
1
Time steps for output duration
3
OK
Cancel
Help
Apply

and Dialog Box



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.

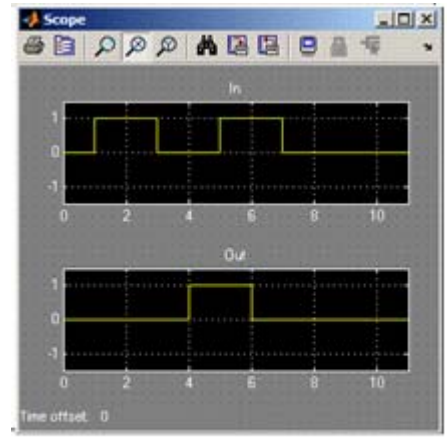
Delayed Fixed Duration

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:

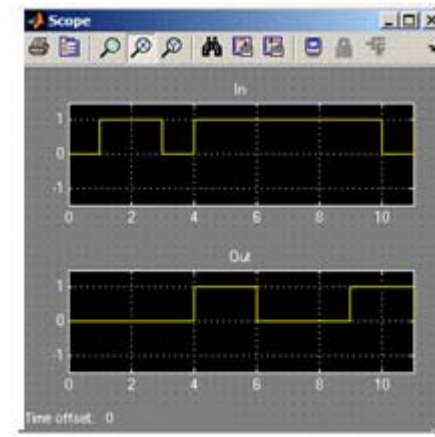
- **Time steps for input detection** = `2`
- **Time steps for delay (optional)** = `1`
- **Time steps for output duration** = `2`

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

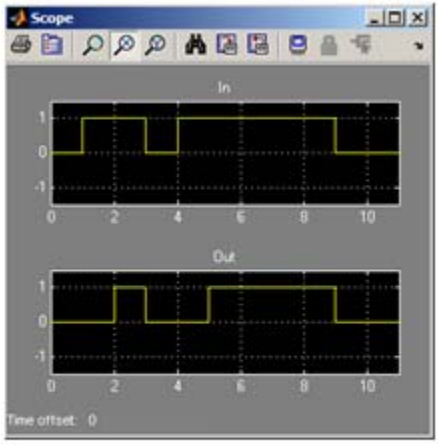


Scope 2

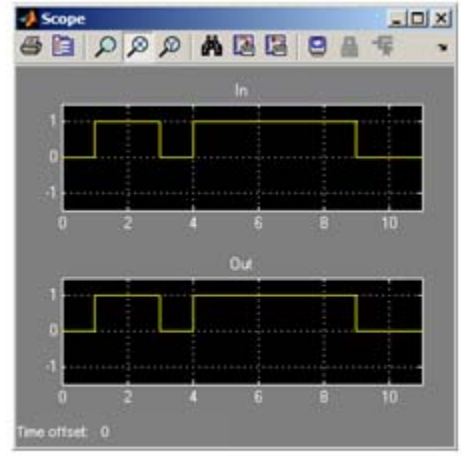
Synchronized

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

Purpose

Extend true duration of input

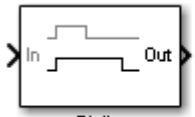
Library

Simulink Design Verifier

Temporal Operators Terminology

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

Description



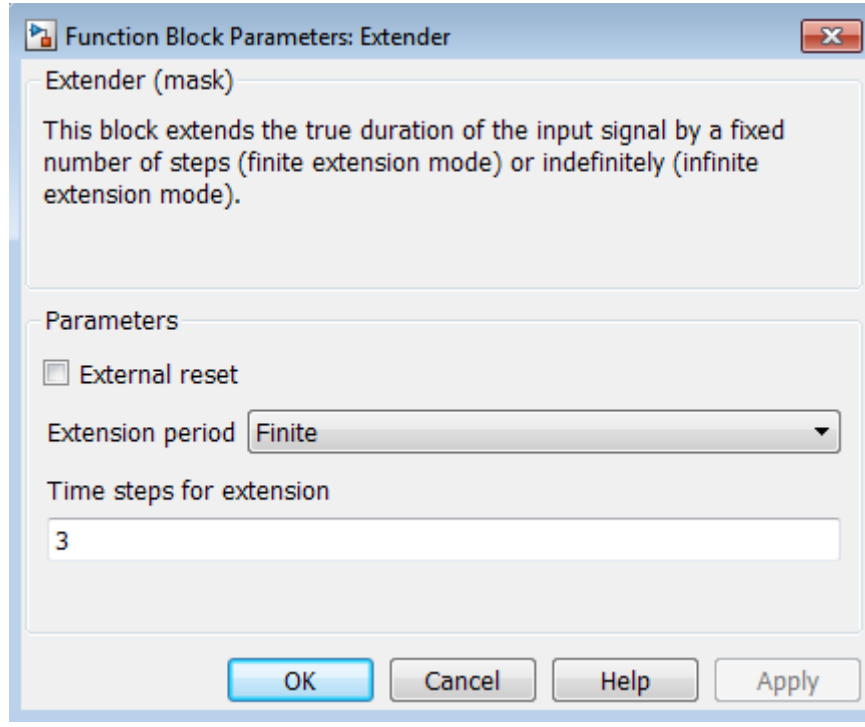
Finite

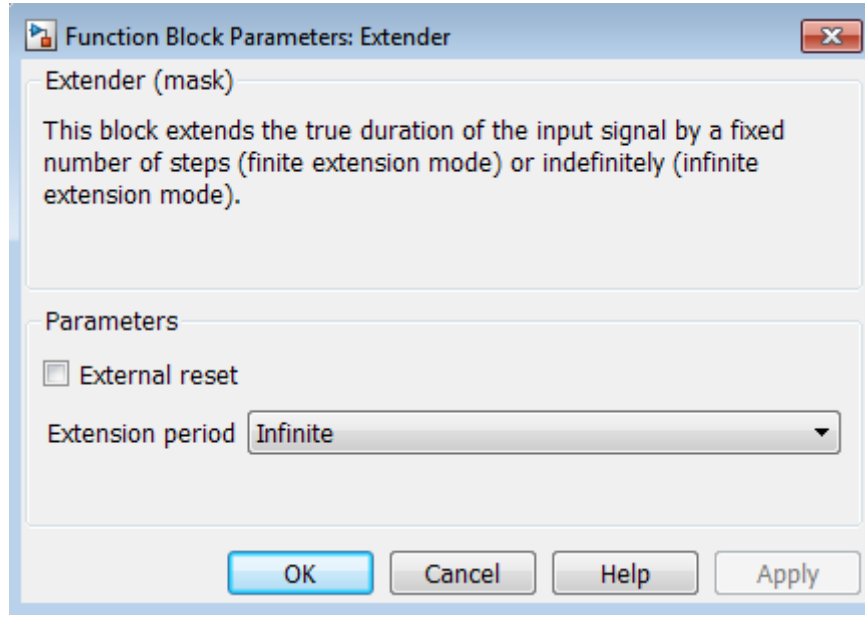
Infinite

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 and Dialog Box





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`).

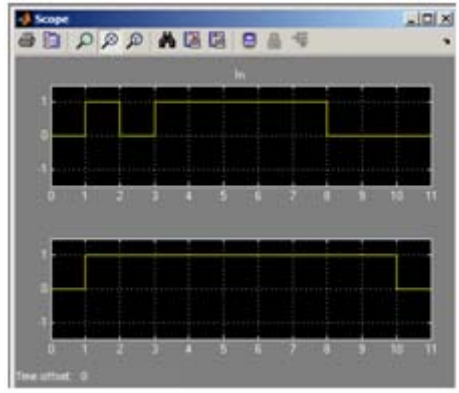
External reset
Specify whether an external Boolean reset signal can reset the block extension. The reset signal also resets the infinite extension. The infinite extension with an external reset is an indefinite extension until the external reset signal becomes `true`.

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

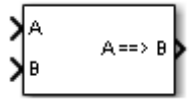
Purpose

Specify condition that produces a certain response

Library

Simulink Design Verifier

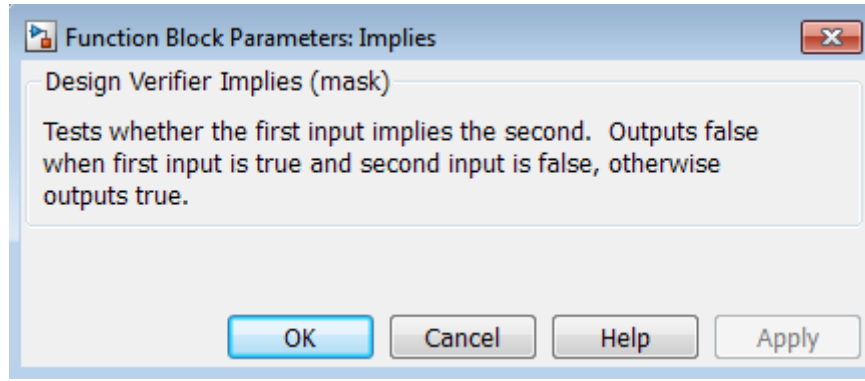
Description



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.

Parameters and Dialog Box



Proof Assumption

Purpose

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" in the MATLAB documentation.)

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 strings as a third input argument that specifies inclusion or exclusion of the interval endpoints:

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

Note By default, `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:

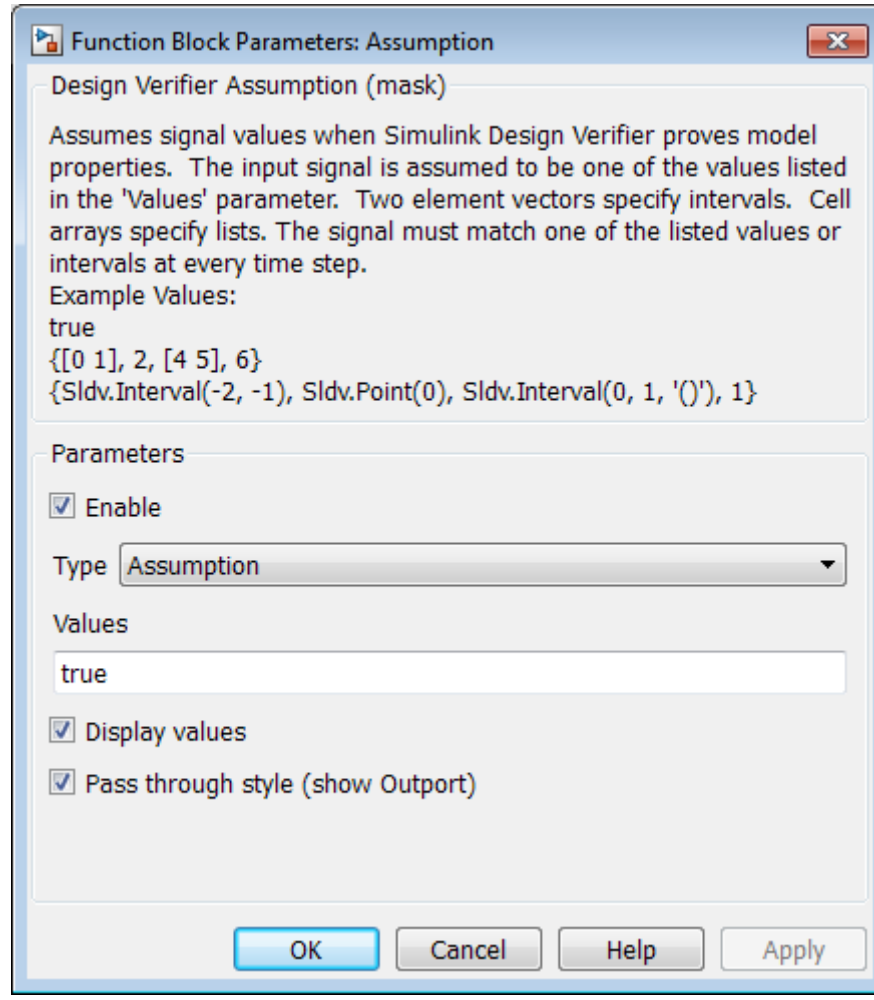
- `Sldv.Interval(0, 1, '[)')` — the right-open interval [0, 1)
- `Sldv.Point(1)` — a scalar

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".

Parameters and Dialog Box



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 2-14).

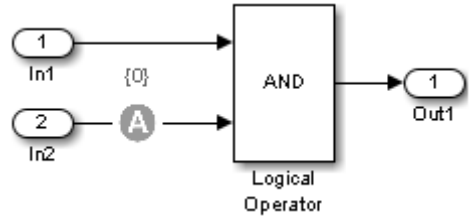
Display values

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

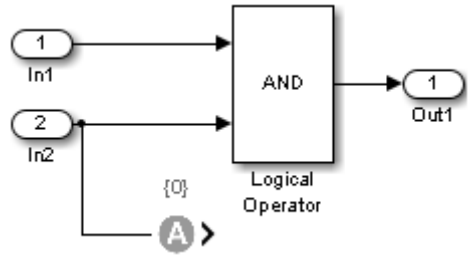
Pass through style

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

Pass through style: Selected



Pass through style: Deselected



See Also Proof Objective, Test Condition

Proof Objective

Purpose

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" in the MATLAB documentation.)

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 strings as a third input argument that specifies inclusion or exclusion of the interval endpoints:

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

Note By default, `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:

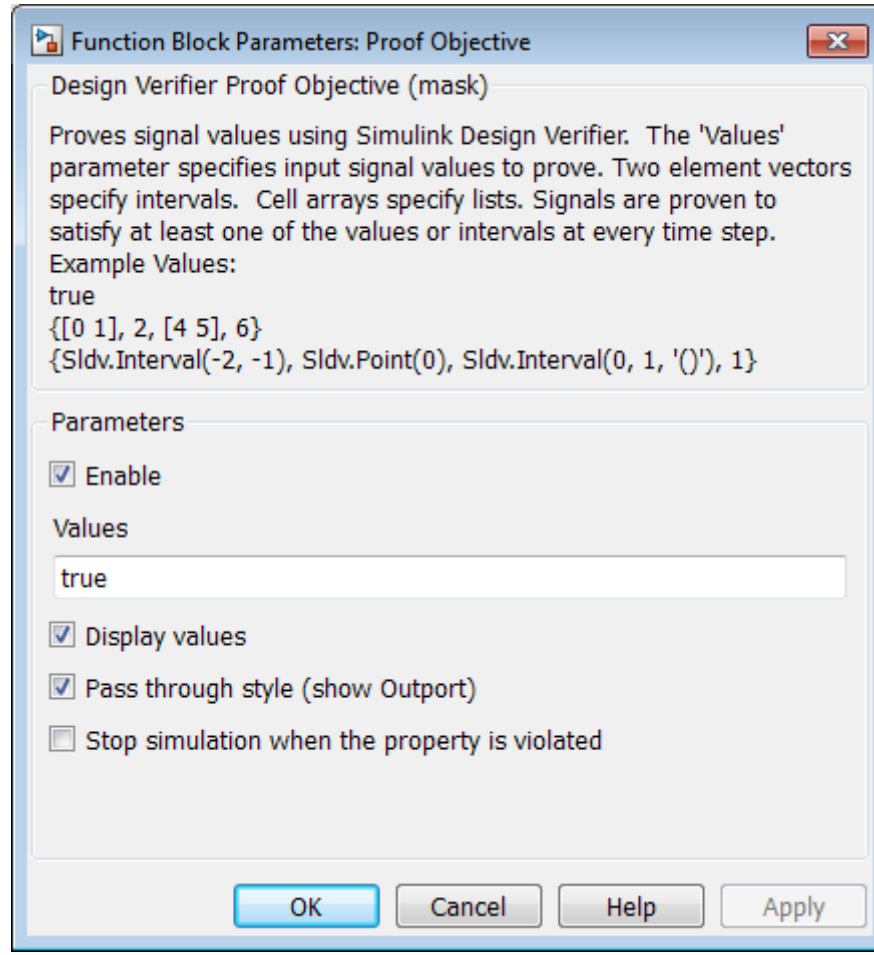
- `Sldv.Interval(0, 1, '[)')` — the right-open interval [0, 1)
- `Sldv.Point(1)` — a scalar

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".

Parameters and Dialog Box



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 2-20).

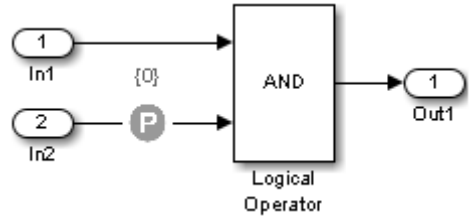
Display values

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

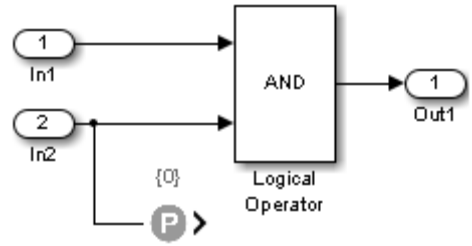
Pass through style

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

Pass through style: Selected



Pass through style: Deselected



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

Test Condition

Purpose

Constrain signal values in test cases

Library

Simulink Design Verifier

Description

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

The block's parameter dialog box also allows you to

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

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

Specifying Test Conditions

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

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 strings as a third input argument that specifies inclusion or exclusion of the interval endpoints:

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

Note By default, `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:

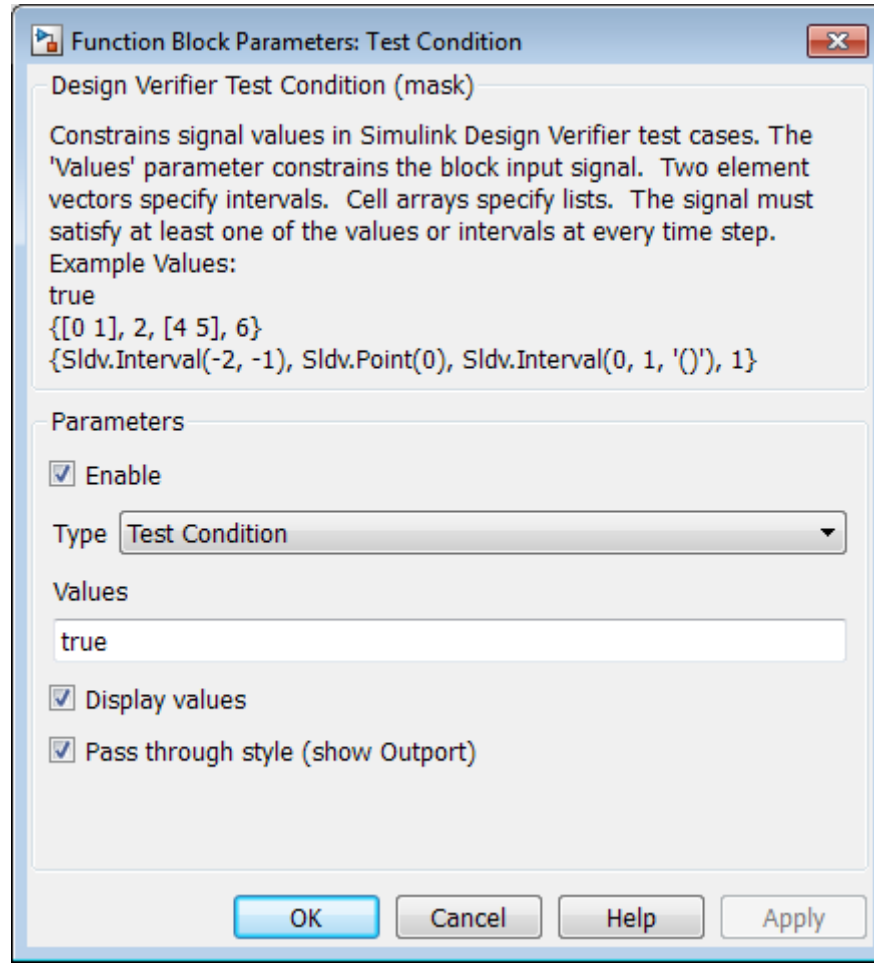
- `Sldv.Interval(0, 1, '[)')` — the right-open interval [0, 1)
- `Sldv.Point(1)` — a scalar

If you specify multiple scalars and intervals for a 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.

Data Type Support

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

Parameters and Dialog Box



Enable

Specify whether the block is enabled. If selected (the default), Simulink Design Verifier software uses the block when generating tests for a model. Clearing this option disables the block, that is, causes the Simulink Design Verifier software to behave as if the

Test Condition block did not exist. If this option is not selected, the block appears grayed out in the Simulink Editor.

Type

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

Values

Specify the test condition (see "Specifying Test Conditions" on page 2-26).

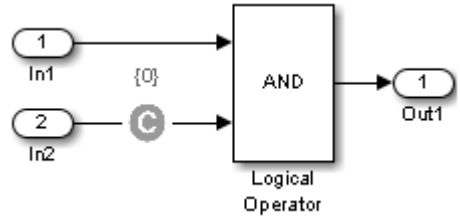
Display values

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

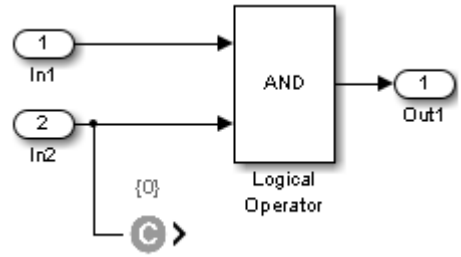
Pass through style

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

Pass through style: Selected



Pass through style: Deselected



See Also Proof Assumption, Test Objective

Test Objective

Purpose

Define custom objectives that signals must satisfy in test cases

Library

Simulink Design Verifier

Description

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

The block's parameter dialog box also allows you to

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

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

Specifying Test Objectives

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

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 strings as a third input argument that specifies inclusion or exclusion of the interval endpoints:

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

Note By default, `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)}
```

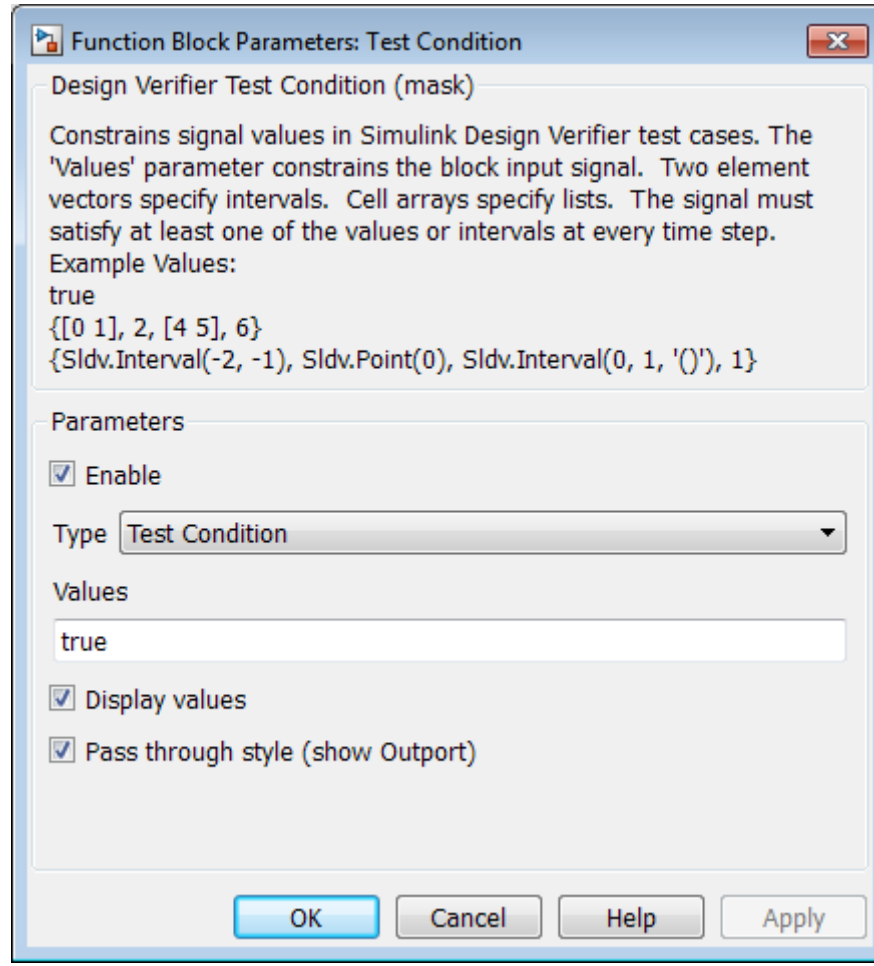
specifies:

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

Data Type Support

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

Parameters and Dialog Box



Enable

Specify whether the block is enabled. If selected (the default), the Simulink Design Verifier software uses the block when generating tests for a model. Clearing this option disables the block, that is, causes the Simulink Design Verifier software to behave as if the

Test Objective block did not exist. If this option is not selected, the block appears grayed out in the Simulink Editor.

Values

Specify the test objective (see “Specifying Test Objectives” on page 2-32).

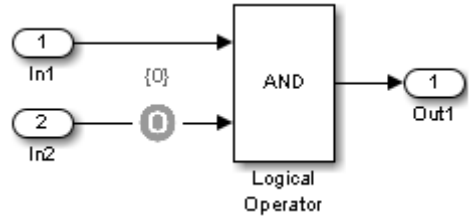
Display values

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

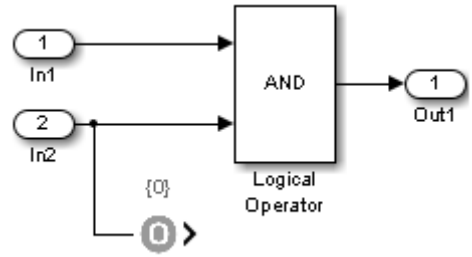
Pass through style

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

Pass through style: Selected



Pass through style: Deselected



See Also Proof Objective, Test Condition

Verification Subsystem

Purpose

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 for code generation, open the Configuration Parameters dialog box and on 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" in the Simulink Coder documentation.

When collecting model coverage, the Simulink Verification and Validation 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.

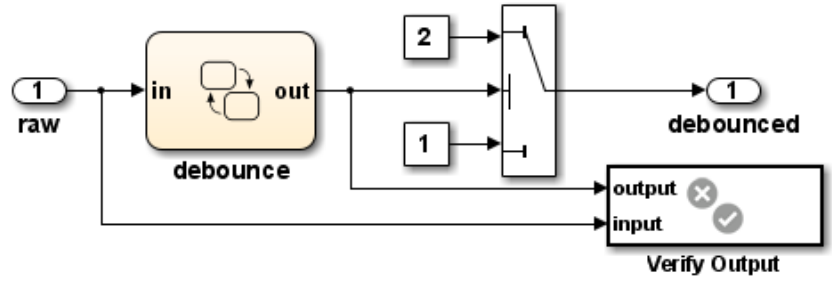
The Verification Subsystem block in the Simulink Design Verifier library is preconfigured to work with the Simulink Design Verifier software. A Verification Subsystem block must:

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

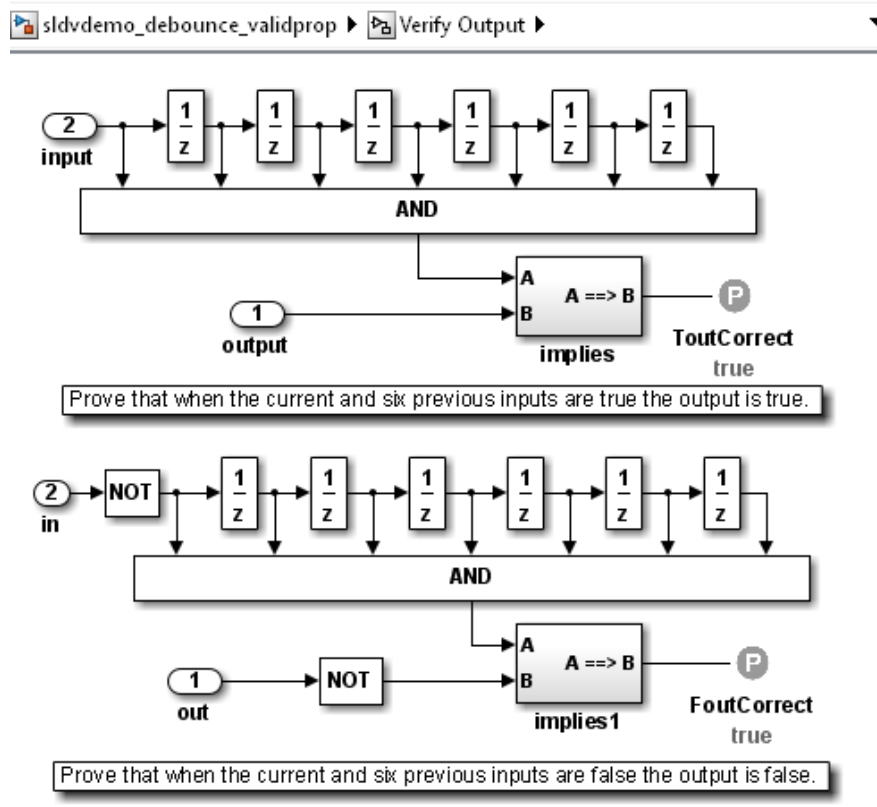
If you alter the Verification Subsystem block so that the preceding conditions are not met, the Simulink Design Verifier software displays a warning.

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 a Subsystem” in the Simulink documentation

Within Implies

Purpose

Verify response occurs within desired duration

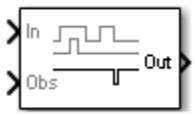
Library

Simulink Design Verifier

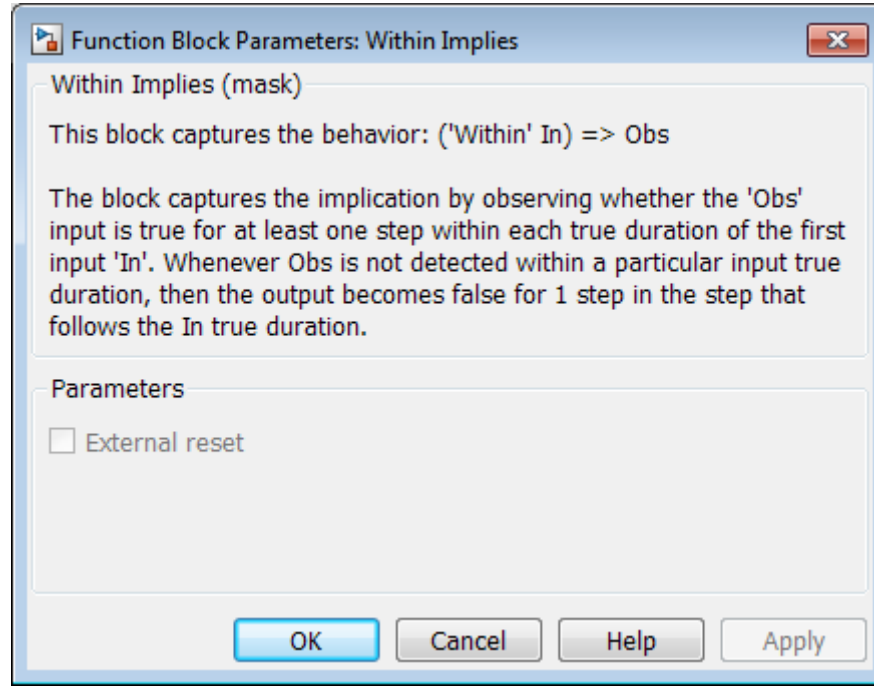
Temporal Operators Terminology

- *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 and Dialog Box

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

External reset

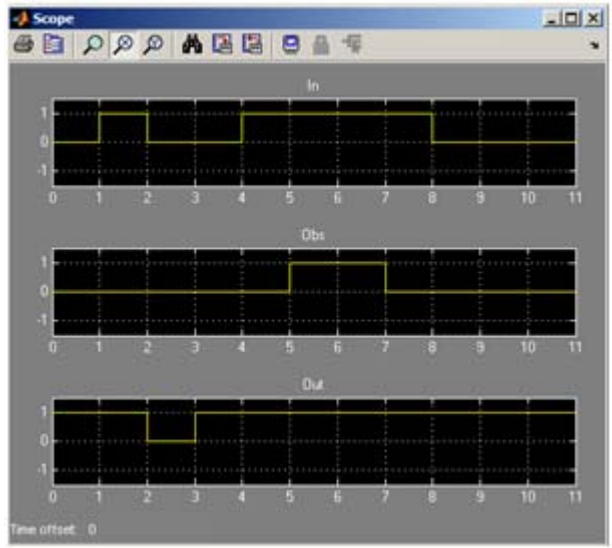
Specify whether the block observation of `Obs` can be reset by an external Boolean reset signal.

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

3

Model Advisor Checks

Simulink Design Verifier Checks

In this section...

Simulink Design Verifier Checks Overview

Simulink Design Verifier checks help you prepare your model for Simulink Design Verifier analysis by identifying elements of your model that might require special attention.

When you run a Simulink Design Verifier check, the Model Advisor performs a checkout of the Simulink Design Verifier license.

Using the Model Advisor, you can:

- Save check results in HTML files. See "View and Save Model Advisor Reports".
- Run checks programmatically. See "Run the Model Advisor Programmatically".

For more information on using the Model Advisor, see "Consult the Model Advisor". For more information on customizing the Model Advisor, see "Automating Check Execution".

Check compatibility with Simulink Design Verifier

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
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	Your model is ready to be analyzed with Simulink Design Verifier.

See Also

- "Consult the Model Advisor"
- "Check Model Compatibility"
- "Handle Incompatibilities with Automatic Stubbing"