cādence™

Formal Verifier User Guide

Product Version 9.2

October 2010

Contents

Preface

Incisive Formal Verifier is Cadence's industry-leading tool for formal analysis. It is able to read Verilog, VHDL, and mixed-language designs and it supports properties written using the industry standards assertion languages, such as PSL and OVL. The tool provides industry-leading features and capabilities that enable both beginners and advanced users to effectively use formal analysis.

Organization and Content of the User Guide

This guide is intended to help you quickly understand how to use Formal Verifier. It does not cover every capability available in Formal Verifier, nor does it give detailed information on the various HDL and property languages. This level of detail would distract from the purpose of the guide. Instead, this detailed information is presented in the *Formal Verifier Reference Manual*. Other sources of information, such as methodology documents and tutorials, are described in the section titled Other References.

The first nine chapters of this guide should be read in order, because each chapter covers information that will be used in the later chapters. An understanding of these chapters is sufficient to help you verify your designs using formal analysis. The remaining chapters cover advanced topics that can be read in any desired order.

Methodology plays a major role in getting value out of formal analysis. While it is not the intent of the user guide to give a detailed discussion on methodology, a high level understanding of methodology will provide a framework for the remainder of the user guide. Thus, methodology is covered in Chapter 1, “Methodology Overview,”.

Chapter 2, “Launching and Interacting with Formal Verifier,” gives an overview of Formal Verifier. It also describes some common conventions used in the tool. Finally, this chapter describes the help facilities.

The process of verifying a design (RTL block) using Formal Verifier is similar for most RTL blocks. Chapters 3 through 9 provide details on the process of using Formal Verifier to formally analyze a design block. The tasks involved in this process include:

- Setting up Formal Verifier to read a block in Chapter 4, “Reading a Block,”.
- Specifying clocks and initializing the design Chapter 5, “Clocks and Initialization,”.

- Understanding and debugging messages Chapter 6, “Understanding Messages,”.
- Developing and using properties Chapter 7, “Developing and Using Properties,”.
- Running verification and analyzing results Chapter 8, “Running Verification and Understanding Results,”.
- Debugging failures Chapter 9, “Debugging Failures and Viewing Witnesses,”.

Understanding these tasks is critical for using Formal Verifier productively. Understanding these tasks is also necessary prior to reading any of the remaining chapters.

Chapters 10 through 15 discuss a variety of subjects. These chapters can be read in any order; however, an understanding of the materials contained in chapters 1 through 9 is a prerequisite. The following subjects are covered:

- **Automatic formal analysis**

  Formal Verifier supports several automatic formal analysis capabilities. It can infer certain checks based on the structure of the RTL. These checks are often useful in finding subtle bugs. Chapter 10, “Automatic Formal Analysis,” gives an overview of some of the most common and useful checks, and discusses some methodology related to automatic formal analysis.

- **Assertion coverage metrics**

  Metrics can help assess the completeness of a set of assertions relative to the block under consideration. Formal Verifier provides a set of metrics for this purpose. This chapter discusses how to use these metrics to help assess assertion coverage.

- **Dealing with complexity**

  Complex designs present challenges for formal tools in general. Formal Verifier has many advanced features for the purpose of analyzing and dealing with complex blocks. This chapter discusses the general subject of complexity and then describes how you can use Formal Verifier to understand and deal with complex designs.

- **Synergy with simulation**

  There are several ways to use simulation to help the formal analysis task or to use formal analysis to compliment simulation-based verification. These are:

  - Using formal analysis to augment coverage data
  - Using simulation to initialize a design
  - Generating simulation testbenches based on error waveforms generated by Formal Verifier for failing properties.

Chapter 11, “Synergy with Simulation,” discusses the above topics in detail.

Other References

Throughout the user guide, several other sources of information are mentioned. These sources are described below:

- **Tutorials**

  Tutorials provide a hands-on approach to learning Formal Verifier and demonstrate many of the specific ideas and features mentioned in the user guide. The tutorials are located in the Formal Verifier installation at `<`*`IFV_install_dir`*`>/doc/ifvtut*`.

  Each tutorial contains a guide, RTL, and scripts. The tutorial guides are also available through the Formal Verifier Help menu. Cadence recommends that you go through these tutorials along with the user guide as a way to reinforce the concepts explained in the user guide.

- **The Formal Verifier Reference Manual**

  This manual provides details on command-line options, TCL commands, variables, and other information. The Reference Manual is helpful when you need more information than the user guide provides. The Reference Manual is available in Cadence Help and through the Formal Verifier Help menu.

- **Formal Analysis Methodology Papers**

  The papers provide a much more in-depth discussion of various methodology topics. The user guide refers to several of these papers. These papers are available in the Formal Verfier Methodology, which can be accessed through the Formal Verfier Help menu.

Typographic Conventions

The following typographical conventions are used in this manual:

Font	Meaning
Italic	Titles of books or other documents. Example: See the *NC-VHDL Simulator Help* for more information.

Font	**Meaning**
`Literal`	Program output or text that you type at the command line.
	Example:
	`ncvhdl *.vhd`
`argument`	Variables
	Example:
	`ncelab top`
Hypertext Link	Links that take you to other relevant sections of this book.
	Example:
	See the File Manager section for more information.

Syntax Conventions

The following syntax conventions are used for text commands:

Syntax	**Meaning**
`literal`	Words in the `courier` font indicate keywords that you must enter in literally. These keywords represent command (function, routine) or option names.
`argument`	Words in *`courier italics`* indicate user-defined arguments, for which you must substitute a name or a value.
|	Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other operator.
	Example:
	`command argument1 | argument2`
[ ]	Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices from which you can choose one.
	Example:
	`command [argument1 | argument2 | argument3]`

Syntax	Meaning
<>	Angle brackets denote variable arguments. You need to replace the angle brackets and their text with text that applies to the command. When used with OR-bars, they enclose a list of choices from which you can choose one. Example: `command <argument1 | argument2 | argument3>`
{}	Braces are used with OR-bars and enclose a list of choices from which you must choose one. Example: `command {argument1 | argument2 | argument2}`
...	Three dots (...) indicate that you can repeat the previous argument. If they are used within brackets, you can specify zero or more arguments. If they are used outside of brackets, you must specify at least one argument, but you can specify more. Examples: ■ *argument*... specify at least one ■ [*argument*]... specify zero or more

About Online Help

The Cadence online documentation system displays manuals and documents in your web browser.

Launching the Documentation

- From the `<installation_dir>/tools/bin` directory, type `cdnshelp` at the command prompt.
- From any Cadence product, click the *Help* button on forms and dialog boxes.
- From any Cadence product GUI, select the main *Help* menu.

Getting Help for Cadence Help

After launching Cadence Help, press `F1` or choose *Help – Contents* to display the help page for Cadence Help.

Providing Feedback

Contact Cadence Customer Support to open a service request if you find:

- An error in a manual
- An omission of information in a manual
- A problem using the Cadence Help documentation system

Customer Support

There are several ways that you can get help with your Cadence product:

- Customer support

  Cadence is committed to keeping your design teams productive by providing answers to technical questions, the latest software updates, and education services to keep your skills updated. For information on Cadence support, go to the following web site:

  `http://www.cadence.com/support`

- Cadence Online Support

  Customers with a maintenance contract with Cadence can obtain current information on the tools at the following web site:

  `http://support.cadence.com`

- Feedback about documentation

  Contact Cadence Customer Support to open a service request if you find:

  - An error in a manual
  - An omission of information in a manual
  - A problem using the Cadence Help documentation system

1

Methodology Overview

In most technologies and tools, the methodology behind the application and usage of the tool within the project is critical to getting good return on investment (ROI) and achieving success. This is especially true with formal analysis (FA) tools.

Introduction to Formal Analysis

Formal analysis (FA) is a process that uses sophisticated algorithms to conclusively prove or disprove that a design behaves as desired for all possible operating states. Desired behavior is not expressed through a traditional testbench, but rather as a set of assertions. Formal analysis does not require traditional user-developed test vectors, instead it analyzes all legal input sequences concurrently and automatically.

Because of this automation, formal analysis, when used correctly, improves productivity and reduces time-to-market. With formal analysis, many bugs can be found quickly and very early in the design process without the need to develop large sets of test vectors or random test generators. Furthermore, because of its exhaustive nature, formal analysis, improves quality by finding corner-case bugs, which elude traditional verification (based on testbenches) methods. The dual benefits of increased productivity and increased quality are driving the adoption of formal analysis into the design flows of many companies.

In formal analysis, ***properties*** are the basic units of expression. Properties are formalized statements about the behavior of signals over time. They are expressed either through a property language, such as PSL, or a library of properties, such as OVL. Properties may express desired behavior of a design under test or they may express the behavior of the environment in which such a design is embedded. Properties, which express desired behavior of a design under test, are called ***assertions***. Properties, which express the behavior of the environment, are called ***constraints***. They are called constraints, because the presence of these properties constrains Formal Verifier to generate input sequences that satisfy the properties of the environment.

Benefits of Formal Analysis

The primary benefits of formal analysis are quality and productivity. Quality is improved because formal analysis finds bugs that simulation-based verification may miss. Productivity is improved because of the following reasons:

- Bugs are isolated so that debugging and fixing problems is easier and the turn-around time is faster.
- Bugs are identified more easily and earlier in the design cycle.
- Finding a bug early involves fewer people.
- Eliminating bugs early means that fewer people are impacted by those bugs and the overall project flows more smoothly.

The combination of improved quality and productivity means better time-to-market (due to the productivity gains) and reduced costs (due to improved quality and fewer silicon re-spins).

Relation of Formal Analysis to Simulation

Formal analysis has some very different characteristics as compared to simulation. Simulation requires the specification of test vectors that drive the inputs of the design, usually in the form of a testbench. The simulation results are limited to the selection of test vectors that users provide as inputs. In contrast, formal analysis is exhaustive in nature. A formal analysis tool tries to verify the design for all combinations of inputs to the design, by using assertions. This eliminates the need for test vectors. However, because of the exhaustive nature of the analysis, computation time can become an issue. As a result, unlike simulation, formal analysis is more productively used on relatively small blocks or designs.

Therefore, a methodology in which formal analysis augments simulation will best take advantage of the capabilities of formal analysis. Verifying a block through formal analysis may allow you to avoid some simulation, but you should expect that significant simulation will still be required, especially at higher levels of a design.

Formal Analysis Users

The two main users of formal analysis are designers and verification engineers.

Designers have the most intimate knowledge of the blocks they implement - the intended behavior, relationship to other blocks, and the implementation in RTL. Because of this intimate knowledge, designers are naturally good candidates to write properties related to their blocks. Designers are also good candidates to interactively use the formal tool to check and debug their own designs because they understand the RTL implementation. The clear advantages that designers have when performing FA makes their role critical in an FA project methodology.

Verification engineers can and should take an active role in the FA flow by writing properties and running the formal tool. Verification engineers do not have the white box knowledge that the designers have. This can make creation of properties at lower levels of the design difficult, and it can make debugging of RTL difficult. However, verification engineers have other knowledge that the designers may not have. For example, verification engineers may have a better picture of the overall verification plan and risk areas. Also, verification engineers may have more expertise in using the formal tool and in writing properties.

There are other factors to consider as well:

- Verification engineers may have different schedule constraints than the design engineers, giving the project an additional degree of freedom in resource allocation.
- Verification engineers bring another perspective to how the design should function and may be able to write properties to find bugs that a designer might not have thought about.

For all these reasons, both verification engineers and designers have important roles in the use of FA.

The Role of ROI in Methodology

A guiding principle behind formal analysis methodology is that it should enable high return on investment (ROI) for the design team. ROI applies at the individual user level, the project level, and the company level. This user guide is primarily concerned with individual users. Therefore, we focus on the user level.

At the user level, the *return* is mostly the following:

- Quality and quantity of bugs found by the formal tool
- Savings in debug time and bug-fix time
- Reduced iteration between engineers

The *investment* is the user's time to learn the methodology, write properties, and run the tool.

Since Cadence does not advocate removing simulation from the verification flow, you should always keep ROI in mind and invest in formal analysis only if the return you expect warrants the investment.

For a more in-depth discussion of all aspects of methodology, refer to the Formal Analysis Methodology paper series described in the section titled Other References in the Preface.

2

Launching and Interacting with Formal Verifier

This chapter gives an overview of both the command-line and the graphical user interface of Formal Verifier. It also describes some common conventions used in the tool. Finally, this chapter describes the online help facilities.

Overview

There are two ways to launch Formal Verifier, single-step method and multi-step mode. The single-step mode is by far the most commonly used launching mechanism, and is the one you should use unless you have specific reasons not to use it. The benefits of single-step method are the following:

- There is one command to get a block up and running. All you need to do is launch the tool with the appropriate options and arguments.
- The single-step method automatically manages libraries by using the `-y` and `-v` options.
- The design units in library files are compiled into a library with the same name.
- Mixed-language designs are handled automatically. The tool understands common suffixes, such as `.vhd` and `.v`, and provides straightforward controls for reading mixed-language designs.

On the other hand, the multi-step method provides for certain capabilities that some users need, such as the following:

- The multi-step method is similar to the multi-step method used in the Incisive Unified Simulator and the NC-Verilog and NC-VHDL simulators. If you are familiar with the multi-step method in these simulators, you may prefer the multi-step method for Formal Verifier.
- The multi-step method provides more control over the placement and reuse of intermediate files. This is helpful when you have pre-compiled design and library files. With the multi-step method, these design and library files can be compiled once and

reused multiple times. This can incrementally save compilation time for projects set up this way.

- The multi-step method provides finer control over the update mechanisms.

The following sections give more details on these methods. It is not usually necessary for you to understand the multi-step method. Consequently, the section on multi-step method is optional.

The Single-Step Method

You can run Incisive Formal Verifier by issuing one command, `ifv`. This variant of Formal Verifier is able to read Verilog, VHDL, and mixed-language designs. By default, its log file is named `formalverifier.log`, although you can specify the file name on the command line.

The `ifv` command has the following syntax

```
ifv [ifv_options]
     <vhdl_design_files>
     <verilog_design_files>
     +top+<worklib.entity:architecture>
```

The `ifv` command has many options. The most common are listed in the following table.

Option	**Purpose**
`-f <filename>`	Read the command-line arguments from the specified file.
`-l <filename>`	Set the log file name to the specified name. The default name is `formalverifier.log`.
`-v <filename>`	Use the specified file as a library.
`-y <directory_name>`	Search the specified directory for the library files.
`+gui`	Launch the Formal Verifier GUI.
	Note: The GUI can also be started after the command-line version of Formal Verifier has come up, by typing `simvision` at the Tcl command-line.
`+incdir+<directory_name>`	Search the specified directories for the files included with the `` `include `` directive.
`+libext+<arg>`	List the extensions to be used for reading the library files from the directory specified with the `-y` option.

`+tcl+<tclfile>`	Read Tcl commands from the specified file.
`VHDL:` `+top+<entity>:<arch>` `Verilog:` `+top+<module_name>`	Specify the top-level entity and architecture names (for VHDL) and top-level module name (for Verilog) to be considered as the top-level module or design unit for Formal Verifier. **Note:** This option is needed for VHDL designs only. You may choose to use this option for Verilog designs when the top-level module is ambiguous or not clearly defined.
`+mapfile+<lib_name>` `+sourcefile+<file_name>`	Use the file specified by *`filename`* as the source file for the logic library specified by *`lib_name`*. The *mapfile* and *sourcefile* options are always used together to establish a mapping between the logical library name and the physical source file for that library. **Note:** This option is needed for VHDL designs only.
`+bb_list+< filename>`	Read the specified file for the list of design units and libraries to be blackboxed.

The usage of these options is described in more depth in Chapter 4, "Reading a Block," of this guide. For more in-depth information on all the available options, please refer to Chapter 2, "The Single-step Method," of the *Formal Verifier Reference Manual*.

Files and Directories Created by Formal Verifier

When you run `ifv`, the tool parses the RTL and generates an internal model of the design. This model is then stored in a binary format in the directory `INCA_libs`.

After the internal model has been generated, `ifv` launches a user-specified Tcl file (see the `+tcl+<filename>` option).

Note: If you do not specify a Tcl file, the tool is launched using the default Tcl file in the `INCA_libs/ifv_snap.nc` directory. The following shows the contents of the default Tcl file:

```
# Add all the assertions
assertion -add -specification

# Start the verification
prove

# Show the results of the verification
assertion -summary

# To get the individual verification status of all the assertions
# assertion -show -all
```

This Tcl file verifies the design and generates a summary of the verification run.

Other files and directories are also created by `ifv`. The following table describes some of the files and directories that are created.

File/directory	Purpose
`engine_info.log`	This file is automatically created by the tool. It contains the depth and memory information for all engines.
`-f <filename>`	This directory contains files and directories that are used by the tool for internal consumption only.
`formalverifier.log`	This is the log file. Note that you can change the name of this file through a command-line option.
`formalverifier.key`	This file is automatically created by the tool. It contains the commands that are specified on the command-line in a single run of Formal Verifier.
`INCA_libs`	This directory contains files and directories that are used by the tool. These files are meant for tool use only and are in a non-readable format.
`rtlchecks.log`	This file is for the tool internal use only.

IRUN-IFV Integration

Alternatively, you can also run IFV in single-step mode through IRUN. IRUN is a single integrated launch engine for various verification processes within Incisive Design Team. IFV has been integrated in IRUN. The IRUN command line is as follows:

```
irun -ifv <design_files> <ifv_options>
```

To get IFV specific options of `irun`, use:

```
irun -helpsubject ifv
```

For details on `irun`, refer *IRUN* user guide.

Limitations:

- Reinvoke is not supported in case of `irun -ifv`.

Differences between ifv and irun –ifv invocations:

- -mapfile & -sourcefile options are not available in irun. Instead, -makelib & -endlib options are used.
- While using OVL, IFV automatically recognizes a configuration for OVL. In irun –ifv flow, -ovl option of irun are used for OVL recognition.

The Multi-Step Method

There are several steps involved in the multi-step method:

1. Compile the RTL files using `ncvlog` or `ncvhdl`, as appropriate.
2. Elaborate the design using `ncelab`.
3. Prepare the design for formal analysis using `formalbuild`.
4. Launch `formalverifier` on the design snapshot obtained from `formalbuild`.
5. Use the GUI or Tcl commands to verify properties, view their status, and debug failed assertions.

Consider the following example of a simple mixed-language simulation script that you might use with NC simulator:

```
ncvhdl -V93 -assert -lexpragma -work worklib test.vhd
ncvlog -assert -lexpragma -work worklib test.v
ncelab -access +C worklib.test:module
ncsim worklib.test:module
```

In this script, several Verilog and VHDL files are parsed. Then, elaboration is done on the snapshot. Finally, `ncsim` is launched to simulate the elaborated model. This script can be modified for use with Formal Verifier, as follows:

```
ncvhdl -V93 -assert -lexpragma -work worklib test.vhd
ncvlog -assert -lexpragma -work worklib test.v
ncelab -access +C worklib.test:module
formalbuild worklib.test:module
formalverifier -input run.tcl worklib.test:module
```

The modified script invokes `formalbuild` and `formalverifier` instead of `ncsim`. Also, a Tcl command file is provided to `formalverifier`. It is beyond the scope of this guide to fully explain the multi-step method. For further information, refer to the *Formal Verifier Reference Manual*, *NC-Verilog Simulator Help*, or *NC-VHDL Simulator Help*.

Behavior of IFV on Protected Modules

Whenever there are protected modules present in the design, IFV ignores these modules for verification and continues with the rest of the design verification. Depending on whether the module is completely or partially protected, the behavior of IFV on these modules will be:

- If the module is completely protected, IFV will ignore all instances of protected module and treat it as partial design.

  For example, in the code mentioned below, the instance `m1` of `Mod1` is ignored as module `Mod1` is completely protected. Therefore, IFV treats this module as partial design.

  ```
  module top(out,in,clk);
  input in;
  input clk;
  output out;
  wire mout;
  reg out;
  Mod1 m1(in , mout);
  always @(posedge clk)
    out =  mout ;
  endmodule
  ```

- If the module is partially protected, IFV will glassbox that module.

  In the code mentioned below, the module is partially protected. Therefore, IFV will glassbox this module.

  ```
  module Mod3(out,in,clk);
  input in,clk;
  output out;
  reg out;
  reg a;
  //pragma protect
  //pragma protect begin
  always @(posedge clk)
    out = in ;
  //pragma protect end
  always @(posedge clk)
    a = in ;
  endmodule
  ```

Note: To know about the glassboxed or completely protected modules present in the design, you can use the `check -show` command.

Command-Line Overview

Formal Verifier is capable of running both in a full GUI mode and in a pure command-line mode. In both cases, you can interactively enter at the tool command prompt or through a script. The tool command prompt looks like this:

```
FormalVerifier>
```

Up/Down Arrow Key Support

The Formal Verifier prompt provides interactive command line editing facilities, similar to those of the UNIX `tcsh` shell. You can use the keyboard keys to recall previously typed command lines, by using the arrow keys.

Every time you type a new command, it is appended to a list of historical input lines. You can traverse up and down this list by using the up and down arrow keys. Alternatively, you can also use the *Ctrl+P*, and *Ctrl+N* keys.

Thus, if you press the up-arrow key (*Ctrl+P*) once, the previously entered command will be displayed. Similarly, if you press the down-arrow (*Ctrl+N*) key, you will return to the newer lines or empty command prompt, as the case maybe.

TCL Command Conventions

Formal Verifier's command language is based on Tcl. In Tcl, many commands have modifiers which tell the tool more about what the command should do.   For example, in the following Tcl command, *assertion* is the command name, and *-add* is a modifier:

```
FormalVerifier> assertion -add top.A1
```

In general, the command format is:

```
command [-modifier [-options]] [arguments]
```

Commands consist of a command name, which can be in uppercase or lowercase and can be abbreviated to the shortest unique string. The command name may then be followed either by *arguments* or *-modifiers*. The *-modifiers* may have *-options*. Modifiers and options can also be abbreviated to the shortest unique string.

You can specify more than one command on the command line, using a semicolon to separate the commands. Because of the way Tcl works, only the output from the last command in a script or on the command line is printed to the screen or log file. In the following example, only the output of the help command is printed:

```
FormalVerifier> version; help
```

Wildcarding

Some TCL commands are able to take multiple property names or net names as arguments. For these commands, it is often convenient to use wildcards to specify names that fit a pattern. The wildcard characters are:

- The asterisk (*)

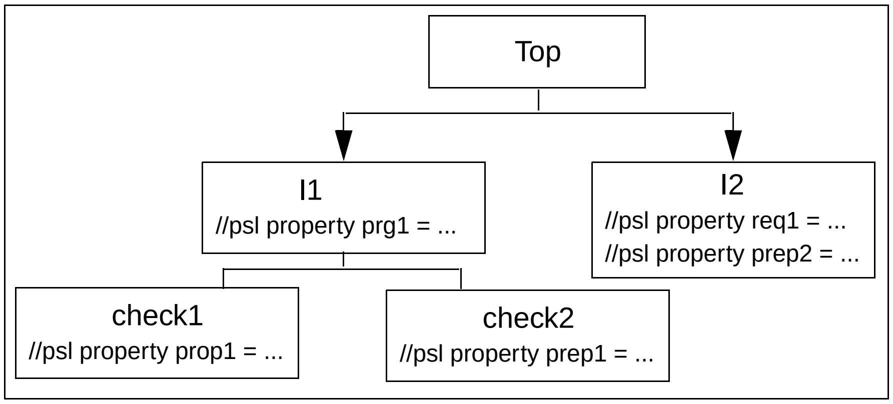
  This character stands for any combination of zero or more characters. For example, the pattern `s*n` matches any object name, regardless of length, that starts with the letter `s` and that ends with the letter `n`. Possible matches include `sn`, `sun`, `son`, and `sudden`.

- The question mark (?)

  This character stands for any single character. For example, the pattern `p?n` matches any three character object name that starts with the letter `p` and that ends with the letter `n`. Possible matches include `pun`, `pan`, and `pin`.

Wildcards in Property Names

Perhaps the most useful situation in which to use wildcards is when specifying property names for use in property manipulation commands. Consider the following diagram of a design with embedded properties


Top
I1
//psl property prg1 = ...
I2
//psl property req1 = ...
//psl property prep2 = ...
check1
//psl property prop1 = ...
check2
//psl property prep1 = ...

Suppose you issue the following command:

- For Verilog

```
FormalVerifier> assertion -add top.I*.check*.p*
```

- For VHDL

```
FormalVerifier> assertion -add :I*:check*:p*
```

By default, the wildcard character `*` will not match the hierarchy separators "**.**" (in Verilog) and "**:**" (in VHDL). Given this behavior, the following properties will be added as assertions:

```
top.I1.check1.prop1
top.I1.check2.prep1
```

Sometimes it is convenient or necessary to have wildcards that match across hierarchies. The `-recursive` option (`-r` for short) facilitates this. It causes the wildcard character `*` to also match the hierarchy separators.

Continuing with the example of the above design, now the following properties in the design will be added as assertions:

```
top.I1.prg1
top.I1.check1.prop1
top.I1.check2.prep1
top.I2.prop2
```

So far, property names have been specified using the full hierarchical name, starting from the top of the hierarchy. It is also possible to specify the property name relative to the current scope. For example, if you set the scope and then add assertions, as shown below:

- For Verilog

```
FormalVerifier> scope -set top.I1
FormalVerifier> assertion -add check*.p*
```

- For VHDL

```
FormalVerifier> scope -set :I1
FormalVerifier> assertion -add check*:p*
```

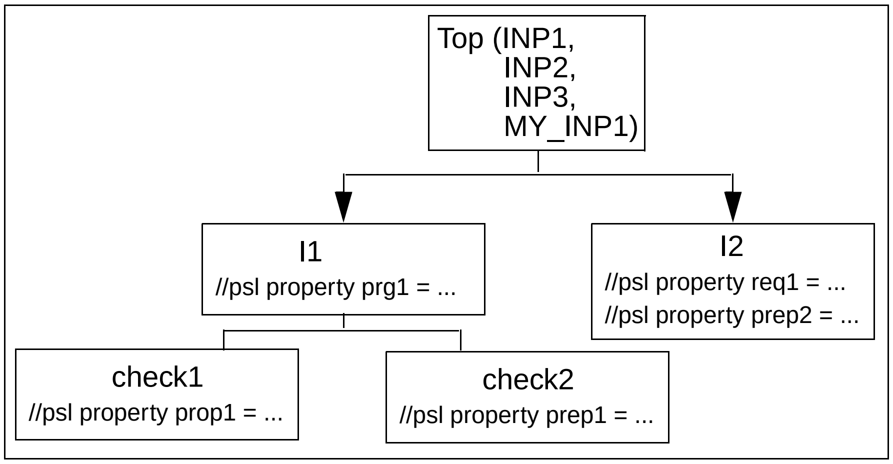
then, the following properties are added as assertions:

```
top.I1.check1.prop1
top.I1.check2.prep1
```

Note: If you are specifying the full hierarchical name, you cannot specify the wildcard character `*` in the design top name.

Wildcards in Net Names

Wildcards can be used to specify net names. Consider the following diagram of a design:


Top (INP1,
INP2,
INP3,
MY_INP1)
I1
//psl property prg1 = ...
I2
//psl property req1 = ...
//psl property prep2 = ...
check1
//psl property prop1 = ...
check2
//psl property prep1 = ...

If you give the following command:

- For Verilog

```
FormalVerifier> constraint -add -pin top.I*=3'b111
```

- For VHDL

```
FormalVerifier> constraint -add -pin :I*=b"111"
```

The `-recursive` option is not allowed when referring to net names, so the wildcard character `*` will not match the hierarchy separators. Given this behavior, the following nets will have pin constraints put on them such that they are tied to the binary value `111`:

```
top.INP1
top.INP2
top.INP3
```

GUI Overview

To launch the Formal Verifier GUI, use the following command:

```
% ifv -f cmd_script.f +gui
```

At startup, the following windows are displayed:

- Formal Verifier window
- Console window

With the Formal Verifier GUI, you can perform the following tasks:

- Control the properties to be verified and to be used as *constraints*.
- Control the proof process when verifying or re-verifying assertions.
- View the status of user-defined properties, automatically extracted properties, and RTL checks.
- Debug assertion failures using the Waveform window and RTL browsers.
- View the design hierarchy and narrow the focus to selected parts of the hierarchy.
- Trace the drivers of the assertions and signals in the design.
- Re-load the design without exiting the tool.

In addition, the GUI provides the following features:

- Interoperability between the Tcl commands and their equivalent steps in the GUI, so that you can switch back and forth between the *Console* and the *Formal Verifier* window. All GUI actions are mapped to Tcl commands, and vice-versa.
- Right-mouse click for easy access to options available from the menus on the *Formal Verifier* window.

Later sections of this guide discuss many of the GUI elements in even more detail to show how they are used in the context of Formal Verifier. For complete information about the GUI, refer to the *Chapter 5* in the *Formal Verifier Reference Manual*.

Getting Help

This section contains information on the following topics:

- Command-Line help
- Tcl command-line help
- Extended Help

Command-Line help

Any time, you can display a list of command-line options for any of the Cadence tools and utilities by typing the tool or utility name followed by the -help option.

The `-help` option displays a list of the command options for the specified tool with a brief description of each option.

The syntax is:

```
% toolname -help
```

Examples:

```
% ncvlog -help

% ncvhdl -help

% ncelab -help

% formalbuild -help

% formalverifier -help
```

Tcl command-line help

To get help on Tcl commands, use the Tcl help command. If you do not specify any arguments, the help command returns an alphabetical list of the available commands. If you request help on a particular command, the help command returns a detailed description of the command and its options.

The syntax of the help command is:

```
help [-brief] [command [command_options]]
```

The help command does not have any modifiers or options.

Examples:

```
FormalVerifier> help

FormalVerifier> help help
```

In addition to the set of interactive commands, the help command also displays help on standard Tcl commands. Only basic information is provided for these commands. Refer to the following Web sites for information on Tcl commands and for other information related to Tcl/Tk:

- *http://www.elf.org*
- *http://dev.scriptics.com/*
- *http://www.tcltk.com/*

Extended Help

Use the `nchelp` utility to display extended help on the brief messages generated by the `compiler`, `elaborator`, `formalbuild`, and `formalverifier`.

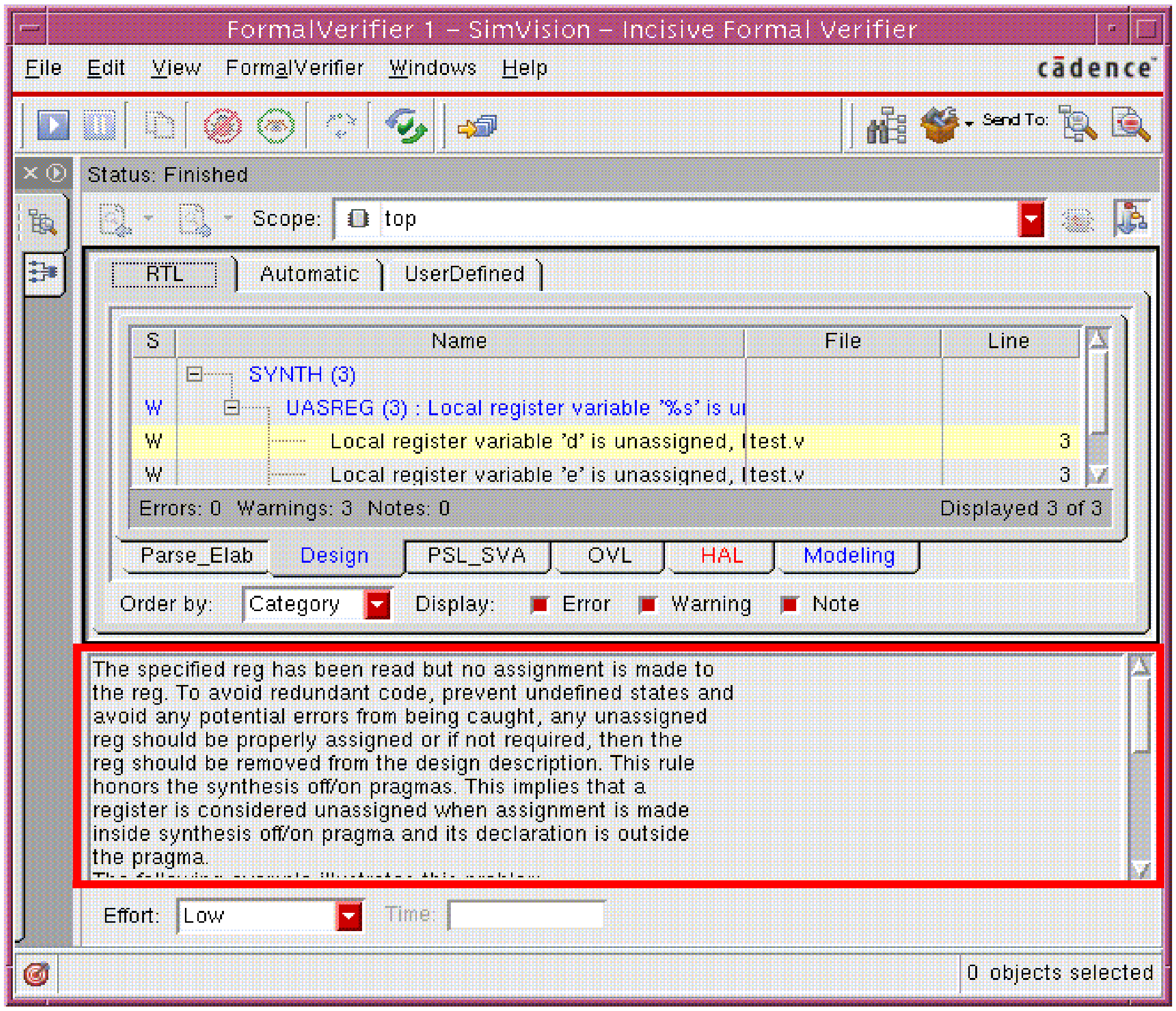
Syntax:

```
% nchelp [options] <tool_name> <message_code>
```

The *message_code* argument is not case-sensitive and can be in uppercase or lowercase. Consider the following examples:

```
% nchelp ncvlog badclp

% nchelp ncelab cuvwsp

% nchelp formalbuild SENCMW

% nchelp formalverifier NOUDPD
```

You can also view the extended or detailed help for the messages that appear during the RTL Checks done by the tool, in the Extended Help pane, in the Formal Verifier GUI, as shown in the figure below:


FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cadence
Send To:
Status: Finished
Scope: top
RTL Automatic UserDefined
Parse_Elab Design PSL_SVA OVL HAL Modeling
Order by: Category Display: Error Warning Note
Effort: Low Time:
0 objects selected

S	Name	File	Line
	SYNTH (3)		
W	UASREG (3) : Local register variable '%s' is u		
W	Local register variable 'd' is unassigned,	test.v	3
W	Local register variable 'e' is unassigned,	test.v	3

Errors: 0 Warnings: 3 Notes: 0 Displayed 3 of 3

The specified reg has been read but no assignment is made to the reg. To avoid redundant code, prevent undefined states and avoid any potential errors from being caught, any unassigned reg should be properly assigned or if not required, then the reg should be removed from the design description. This rule honors the synthesis off/on pragmas. This implies that a register is considered unassigned when assignment is made inside synthesis off/on pragma and its declaration is outside the pragma.

Note: You can use the mouse to resize the Extended Help pane.

3

Overview of the Formal Analysis Process

Most applications of Formal Verifier for block verification have a basic process through which formal analysis is done. Before diving into the details of the process, it is helpful to get an overview of the process.

Reading a Block

The setup phase has the following steps:

1. Create a directory in which to hold user-developed scripts and tool-generated intermediate files.
2. Create a command-line script and a Tcl command script. Initially these scripts contain the commands necessary to read the files. Later, you will add to these scripts, as needed. At this stage, you typically invoke the tool with these basic scripts to debug the scripts themselves, and to resolve syntax and other obvious errors in the RTL.

Clocks and Initialization

The next step is to set up the clocking of the block. You can specify any number of inputs as clocks and provide arbitrary periodic waveforms for these clocks, by using Tcl commands. Along with clocking, it is important to set up the block's initialization or reset sequence. There are several techniques that can be used to accomplish this. The most common and simplest is to use Tcl commands to toggle reset and run simulation for several cycles to propagate reset through the block.

Understanding Messages

During the design verification process, Formal Verifier analyzes the RTL and properties in many ways, from simply parsing the RTL to advanced modeling checks. The analysis process can generate a significant number of messages, ranging from fatals to notes. It is important to understand the various messages, because these messages can indicate bugs in the RTL and/or in the properties. Furthermore, finding and resolving bugs at these early stages is often easier than finding and resolving bugs through debugging failed assertions.

Developing and Using Properties

Properties are the key mechanism which Formal Verifier uses to analyze your design. Chapter 7, “Developing and Using Properties,” discusses several key topics, outlined below:

- **Classes of properties**

  There are several types of properties, including safety properties, liveness properties, and existential properties. Understanding what these different classes of properties are will help you understand when they can be applied, how the tool shows these properties, and how to evaluate the failures of the different types of properties.

- **Property specification mechanisms and coding guidelines**

  Formal Verifier supports several property specification mechanisms such as PSL and OVL. It is important to understand how these mechanisms are supported and how to select which one to use for your particular situation. It is also important to follow some basic coding guidelines to enable you to use Formal Verifier to verify the properties you write, more efficiently.

- **Using properties in Formal Verifier**

  Once you have your properties coded, you need to indicate the properties that are *assertions* to be verified and the properties that are *constraints* to be used during verification of the assertions. There are several Tcl commands that enable you to do this.

- **Recommended approach to developing and analyzing properties**

  Developing properties for a block can be a daunting task for someone new to the process. Cadence recommends a structured approach to property development, which starts with protocol properties on interfaces, then moves on to white-box properties, and finishes with black-box functional properties.

  In addition to this structured approach, Cadence recommends an incremental approach where at first, you turn off or inactivate some of the interfaces to a block, so that you can gradually build up the complexity of the properties. The use of an incremental approach lets you find and resolve bugs more quickly than if you tried to build up a complete and general set of properties prior to starting verification.

Running Verification and Understanding Results

As you develop assertions, you will want to verify them. The first part of Chapter 8, “Running Verification and Understanding Results,” discusses how to start the proof process on a set of assertions and how to control the proof process. After the proof process is completed, each assertion has an associated result.

The second part of this chapter discusses the possible results and how to interpret them.

It is possible for an assertion to have a *Pass* result due to user error, rather than truly indicating that the design is bug free relative to the assertion. Given this, you need to know how to validate the results. The third and final part of this chapter discusses the need for validation and techniques you can use to validate the results.

Debugging Failures

A critical part of the formal analysis process is debugging. Chapter 9, “Debugging Failures and Viewing Witnesses,” discusses how to debug failures using the Waveform Viewer and Source Browser. This chapter also gives hints about the best ways to use some of the tool's features to make the debugging process more efficient.

4

Reading a Block

In the setup phase, you need to do the following steps:

1. Create a directory in which to hold user-developed scripts and tool-generated intermediate files.
2. Create a command-line script and a Tcl command script. Initially these scripts contain the commands necessary to read the files. Later, you will add to these scripts, as needed. At this stage, you typically invoke the tool with these basic scripts to debug the scripts themselves, and to resolve syntax and other obvious errors in the RTL.

Directory Structure and Files

Although the tool can be run from any directory, Cadence recommends that you create a separate directory to hold the script files, log files, and other files generated by Formal Verifier. Further, although not strictly necessary, Cadence recommends that you create a new working directory for each block that you use the tool on. Depending on your preferences, you may decide to create a single directory to hold all of your Formal Verifier-related files, or you may decide to create several subdirectories under a main directory.

Some subdirectories that you may be interested in creating are:

- `prop`

  If you decide to create several different files containing properties and auxiliary RTL code, you may decide to keep these files in a separate *property* directory.

- `cmd`

  If you create numerous Formal Verifier related scripts for different purposes, you may decide to keep them in a separate directory for scripts.

- `log`

  Formal Verifier generates several log files. Some users prefer to maintain several older log files around for a period of time. Having a separate directory for log files is a good way to avoid cluttering up your main working directory.

If you decide on a single directory, you would run Formal Verifier from this directory and all output would be stored in this directory. If you decide on multiple subdirectories, you would typically run the tool out of the `cmd` directory.

Note: Only one run of the tool is supported in a given working directory.

As users and project teams gain experience, they often end up creating auxiliary RTL or properties that are reused in multiple places and by multiple people. To accommodate sharing of these, Cadence recommends that you create a central repository for the reusable material.

The Command-Line Script

The first script file that you need to create is the command-line script. Cadence recommends that you use the name `top_block_name.f`, where `top_block_name` is the name of the top level of the hierarchy.

Formal Verifier can read Verilog, VHDL, or mixed-language designs. Command-line scripts for each of these variations are similar. The next several sections discuss each of the variations.

Verilog Command-Line Script

The following example shows a typical command-line script for a Verilog-only design:

```
1   +define+ABV_ON
2   -y $ABV_LIB
3   +libext+.vlib
4   +incdir+../../rtl
5   -l ../log/bus_arb.log
6   ../../rtl/bus_arb.v
7   ../../rtl/rr_arb.v
8   ../../rtl/vcomp_bus.v
9   +propfile+../prop/bus_arb.psl
10  +tcl+bus_arb.tcl
```

The following is a detailed explanation of each of these lines:

Line #	**Explanation**
`Line 1`	Defines a Verilog macro called `ABV_ON`. The Verilog macro `ABV_ON` is often used along with `` `ifdef `` (and corresponding `` `endif ``) to surround blocks of code intended for verification purposes only.

Line #	Explanation
`Line 2`	Tells Formal Verifier to search the directory referenced. In this case, the `$ABV_LIB` environment variable to find library elements. The tool searches for a library element whenever it finds an instance that has no matching module definition in the user's RTL. The file name that the tool looks for is based on the module name of the undefined instance with a suffix of `.v` or a user-defined suffix.
`Line 3`	Specifies a user-defined suffix as `.vlib`. There can be more than one user-defined suffix.
`Line 4`	Tells Formal Verifier to search the directory `../../rtl` for any file included with the `'include` directive.
`Line 5`	Sets up a log file with a non-default name, `bus_arb.log`. If this line were not present, Formal Verifier would create a log file in the current directory named `formalverifier.log`.
`Lines 6 through 8`	Specify the list of RTL files.
`Line 9`	Specifies a PSL property file to use. External property files contain PSL vunits and can be useful when you cannot, or do not want to, write properties in-line with the RTL code.
`Line 10`	Tells Formal Verifier to use the Tcl command file called `bus_arb.tcl` after generating the internal model of the RTL. Cadence recommends naming the Tcl file using the same base name as the command-line script file, i.e. `top_block_name.tcl`.

VHDL Command-Line Script

The following example shows a typical command-line script for a VHDL-only design:

```
1   ../rtl/fifo_cntl.vhd
2   +generic+"ABV_ON => \"TRUE\""
3   +mapfile+abv_utils +sourcefile+../lib/abv_utils.vhd
4   +top+fifo_cntl:rtl
5   +tcl+fifo_cntl.tcl
6   -l fifo_cntl.log
```

The following is a detailed explanation of each of these lines:

Line #	Explanation
`Line 1`	Specifies the RTL file.

Line #	**Explanation**
`Line 2`	Specifies the value of a VHDL generic variable at the top level of the RTL. In this case, the generic `ABV_ON` is set to the value `TRUE`. Note the backslashes and quotation marks are necessary.
`Line 3`	Shows how library names are mapped to specific VHDL files. In this case, the library name `abv_utils` is mapped to the file `../lib/abv_utils.vhd`.
`Line 4`	Specifies the top of the design. This line is necessary for VHDL and mixed-language designs.
`Line 5`	Specifies which Tcl file to use. Cadence recommends naming the Tcl file using the same base name as the command-line script file, i.e. `top_block_name.tcl`.
`Line 6`	Specify the name and location of the log file. If this line were not present, Formal Verifier would create a log file in the current directory named `ifv.log`.

Mixed-Language Command-Line Script

The scripts for mixed-language designs are very similar to those for Verilog-only or VHDL-only designs. Any of the lines in the scripts in the previous two sections can be mixed and matched. Formal Verifier recognizes Verilog and VHDL files by their suffixes.

The Tcl Command Script

After you create the command file, you should create the Tcl command script file. The following is a typical Tcl command script:

```
1   clock -add clk
2   input init.tcl
3   constraint -add -pin rst_n 1
4   constraint -add *:assume_*
5   assertion -add *:assert_* -r
6   prove
7   assertion -show -status fail
8   constraint -show
```

The following is a detailed explanation of each of these lines:

Line #	**Explanation**
`Line 1`	Specifies that the signal `clk` should be considered a clock. Clocking and clock specification are discussed in Chapter 5, “Clocks and Initialization,”.

Line #	**Explanation**
`Line 2`	Tells the tool to read in another Tcl script file called `init.tcl`. In this case, the Tcl script that is referenced is used for initialization. Initialization is discussed in Chapter 5, “Clocks and Initialization,”.
`Lines 3 and 4`	Adds *constraints* to the system. Constraints and how to use them are discussed in Chapter 7, “Developing and Using Properties,”.
`Line 5`	Adds *assertions* to the system. Assertions and how to use them are discussed in Chapter 7, “Developing and Using Properties,”.
`Lines 6`	Tells the tool to start the verification process on the assertions that were added in Line 5. Details of the `prove` command and how to control it are in Chapter 8, “Running Verification and Understanding Results,”.
`Line 7`	Shows all those assertions that failed during the verification process. Chapter 9, “Debugging Failures and Viewing Witnesses,” discusses the possible verification results.
`Line 8`	Shows all the constraints in the system.

Before creating a full-fledged Tcl command script, Cadence recommends starting Formal Verifier with only the command-line script, to find and resolve errors in the command-line script and in the RTL. After the command-line script is clean and the RTL is read in without errors, you can work to develop the Tcl command script.

Launching Formal Verifier

With the command-line and Tcl scripts in place, the next step is to start the tool. If the name of the command-line script is `top.f`, you launch the tool by giving the following command:

```
% ifv -f top.f +gui
```

This invokes the tool in GUI mode. If there are any problems with the command-line script or with RTL errors, you will see error messages before the GUI starts up.

5

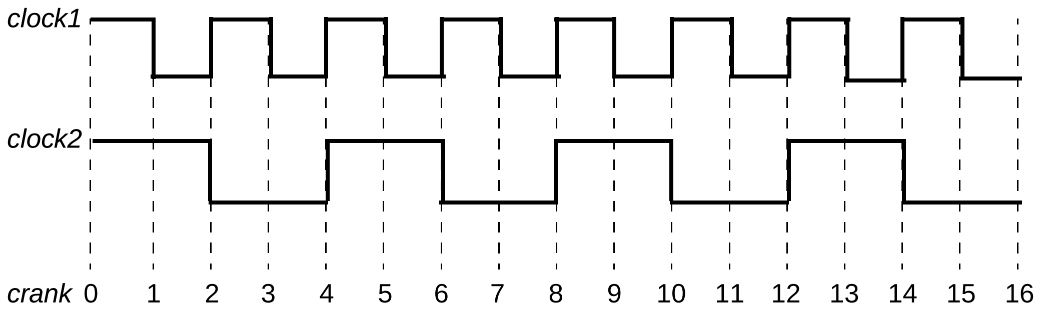
Clocks and Initialization

Once you have the basic scripts in place and the block as been successfully read, the next step is to set up the clocking of the block. You can specify any number of inputs as clocks and provide arbitrary periodic waveforms for these clocks, by using Tcl commands. Along with clocking, it is important to set up the block's initialization or reset sequence. There are several techniques, which can be used to accomplish this. The most common and simplest is to use Tcl commands to toggle reset and run simulation for several cycles to propagate reset through the block.

Setting up Clocks

Clock specification is essential to the verification of synchronous designs. The core engine of Formal Verifier has cycle simulation semantics where verification is performed at a predefined unit-less time step, called the verification time step. This is also referred to as a *crank* and serves as the smallest discrete step at which assertions are verified.

You can specify the behavior of the clock signals in terms of cranks. In the absence of such a specification, the clock signal is treated as any other unconstrained input signal. Using the `clock` command, different kinds of clock waveforms can be generated as shown in the figure below:



In the figure above, `clock1` has a period of two cranks and `clock2` has a period of four cranks.

The clock Command

You can specify the behavior of the clock signals using the `clock` command. The `clock` command has the following parameters:

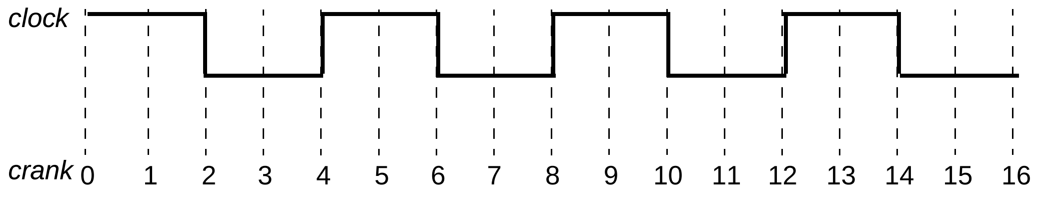
- *Initial* value can be either `0` or `1`, and represents the starting value of the clock waveform The default value is `1`.
- *Offset* is the number of time units for which the initial value of clock signal should be held. By default, the value is held for `1` time unit.
- *Width* can be any positive number, and denotes the time for which the clock waveform remains at the new value after it has toggled from its *initial* value (after offset time units).The default value is `1`.
- *Period* is the time period of the clock signal. By default, time period is the sum of *offset* and *width*.

It is easy to get confused about default values in situations where there is more than one clock. This confusion can lead to incorrect clock specifications. Therefore, if you have more than one clock definition, Cadence recommends that you explicitly define all the parameters for each `clock` command in your script.

For details of the `clock` command, refer to *Chapter 1, "Formal Verifier Command Reference,"* of the *Formal Verifier Reference Manual*.

Ramifications of Large Clock Periods

Large clock periods increase run time and reduce capacity. Therefore, create clocks with large periods only when necessary. For example, consider the following figure:



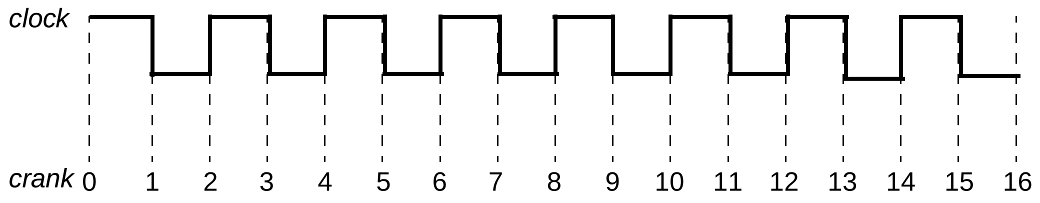
In the figure above, *clock* represents a clock whose high and low phases are two cranks long. Because the tool verifies the design at each crank, this would result in unnecessary verification at cranks 1, 3, 5, 7, 9, and so on. Hence, you should avoid clock specifications with periods greater than two cranks unless required for verification. Such a clock specification could be required in the following situations:

- A design with multiple clocks.

- A single clock design where asynchronous set-reset is active when the clock is stable, that is, asynchronous set-reset are not active either at the positive or negative edge of the clock.
- A single clock design, which is sensitive to both edges of the clock and has asynchronous set-reset.

Default Clock

If you do not specify an explicit clock, Formal Verifier attempts to identify the clock signal in the design and automatically assigns the default value of `1` to *initial*, *offset* and *width*. The clock shown in the figure below represents a default clock waveform:



Any primary input signal, which is directly used to clock a flip-flop is treated as a clock signal. When there are multiple clock signals, the tool does not automatically assign any clock specification; such signals are treated similar to any other input signal.

Multiple Clocks

You can specify the behavior of multiple clocks by using the `clock` command, described above. You can use the clock command to specify any number of different but related clocks. These clocks are related because they are all specified with respect to a crank.

If the clocks are unrelated to each other (asynchronous), you should adopt the following strategy:

1. Specify the clock for one domain and allow the clock signal corresponding to the other domain(s) to remain as an input signal.
2. Verify the assertions in the clock domain whose clock was specified in the previous step.
3. Delete the clock specification and repeat the above mentioned steps for other clock domains.

Performing Initialization

Proper design initialization is critical in formal analysis. Without proper design initialization, verification can produce false assertion failures. Thus, before assertion verification, the design should be reset to a correct initial state. Formal Verifier supports the following multiple ways to initialize the design:

- Tcl-based initialization
- PSL-based initialization
- Initialization by simulation dump

These are described in the sections that follow. Of these, Tcl-based initialization is the most common and easiest to use. In addition to ease of use, it has the advantage of better performance than PSL-based initialization, in most cases. Given these benefits and if you are new user, Cadence recommends that you use Tcl-based initialization unless it does not meet your needs.

Tcl-Based Initialization

Tcl-based initialization is perhaps the most convenient way to initialize a design. Tcl-based initialization uses an internal event simulator to run simulation on the design. Tcl commands force reset and potentially other signals at appropriate times during simulation. The state of the design after simulation is the desired initial state. This state is then used as the initial state during the verification of assertions.

Those state elements (flip-flops and latches) that take the value `X` at the end of simulation are not initialized, and verification is done assuming all possible initial values for those state elements. Because assertions are checked under all possible initial values, the tool is able to provide a very strong check for initialization correctness.

Uninitialized state elements are very common. Some common occurrences of these include address or data holding registers, simple delay registers, memories, and pipeline registers. State elements that should be initialized include state vectors for state machines, registers on handshake signals, and other control-oriented state elements.

The first time you set up initialization, visually scan through the list of initial values that are reported and decide whether there are any incorrectly uninitialized state elements. Although these incorrectly uninitialized state elements would likely be caught by assertion failures later, it is much more time-consuming to identify an initialization problem with an assertion failure than it is to simply scan a list of initial values.

Tcl Commands for Initialization

Tcl-based initialization is supported through the following Tcl commands:

`force <pin_or_wire> [=]<value>`

> This command forces the specified signals to the specified value. Primary inputs that are not forced take the value `X` during simulation.

`run <num_of_cranks>`

> This command runs the internal event simulator for the specified number of cranks. The number of cranks per clock cycle depends on the clock specification. See the section titled Setting up Clocks on page 45 for details on setting up clocks and how clocks relate to internal cranks.

`init -load -current`

> This command loads the initialized values of the state elements from the event simulator into the formal engine. This command should be run after all other run and force command have completed to ensure that the correct initialization values are loaded.

`init -add <stateElement> [=] 0|1|X`

> This command sets the initial value of the specified state element. You should execute this command after `init -load` to ensure that values added will not be overridden by `init -load`. Usage and caveats associated with this command are more fully explained in the section titled Explicit Initial States on page 54.

`init -show [-all | stateElements] [-uninitialized]`

> This command shows the initial values of the specified state elements. Optionally, it can be restricted to show only those elements that are uninitialized. This command is particularly good to run at least once to visually inspect that the state elements that you expect to be initialized are initialized correctly.

`debug -init`

> This command lets you see a waveform of the initialization simulation. The Waveform Viewer is similar to what you see when you debug assertions, however in this case, the waveforms shown are derived from the event simulation. See the section titled Debugging Tcl-Based Initialization on page 56 for more information on debugging initialization.

Some of the above commands have other capabilities not described here. Details on the commands can be found in *Chapter 1, "Formal Verifier Command Reference,"* of the *Formal Verifier Reference Manual*.

Typically, the Tcl command script is edited to include some combination of the above commands. Depending on how the block needs to be initialized, there may be more than once `force` command, and these commands may be separated by one or more `run` commands. In most block-level verification, it is sufficient to have a single `force` command that asserts reset, followed by a single run command.

Initialization Example

To better understand initialization, it helps to look at an example. Below is a simple module which will help illustrate the initialization process.

```
// Counter.v
module counter(clk,reset,count);
 input clk;
 input reset;

 output [2:0] count;
 reg [2:0] count;

 // psl P1: assert never (count == 3'b110) @ posedge clk;

 always @ (posedge clk or negedge reset)
 begin
    if(!reset)
       count = 3'b000;
    else if (count < 3'b101)
       count = count + 1;
 end

endmodule
```

This module contains a counter that counts from zero (0) to five (5) and then stops counting. The assertion checks to see that the counter never counts past 5. Clearly, the counter in the above design can be reset by an active-low transition of the reset signal.

The following script shows some Tcl commands that initialize the design and verify the assertion:

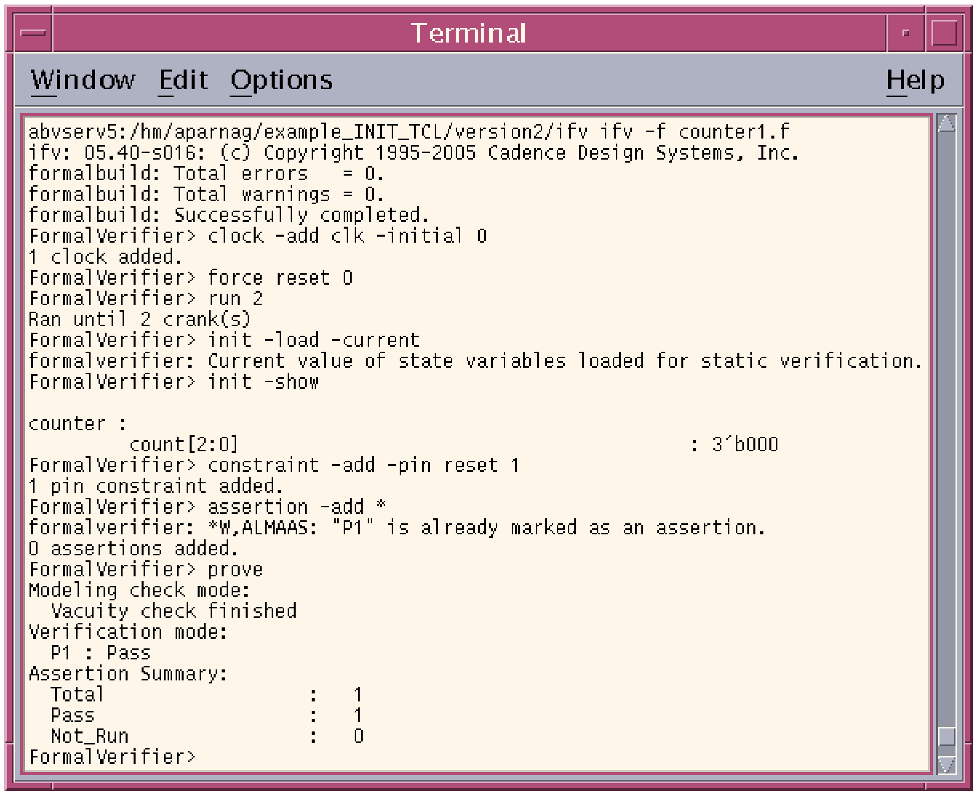
```
// counter1.tcl
clock -add clk -initial 0
force reset 0
run 2
init -load -current
init -show
constraint -add -pin reset 1
assertion -add *
prove
```

The following is a detailed explanation of each of these lines:

Line #	**Explanation**
`Line 1`	Defines the clock so that the signal clk has a 2-crank period and starts out low. The clock is set up prior to running the simulation. If this is not done, then the default clock will be used. Default clock setup is described in the earlier section on clocking.
`Line 2`	Asserts reset active low.
`Line 3`	Runs the simulation for 2 cranks.
`Line 4`	Loads the initialization values from the point just after running for two cranks.
`Line 5`	Displays all the initial values for the state elements.
`Line 6`	Constrains reset to be inactive for later stages of formal analysis. This is often done in formal analysis to avoid having to deal with reset going active at arbitrary times. Constraints in general, and the constraint command specifically, are described in Chapter 7, "Developing and Using Properties,".
`Line 7`	Adds the single assertion shown in the code to the session, so that it can be proved in `Line 8`. Adding assertions is described in detail in Chapter 7, "Developing and Using Properties,". Proving is described in Chapter 8, "Running Verification and Understanding Results,".

If you were to run the above example, you would see something like the following:

Figure 5-1 Verification after Initialization



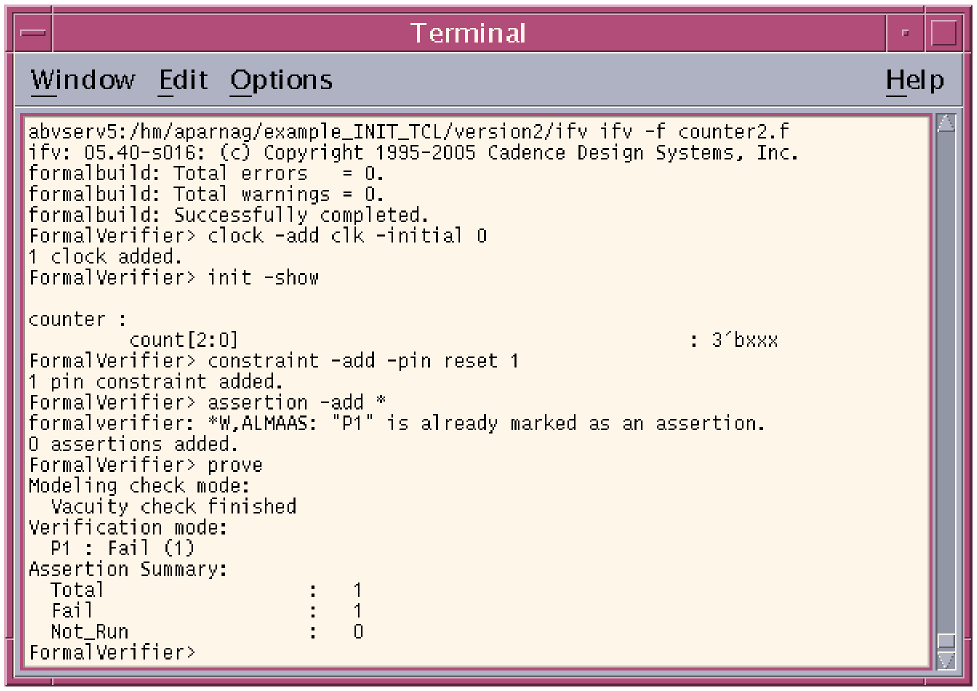
```
abvserv5:/hm/aparnag/example_INIT_TCL/version2/ifv ifv -f counter1.f
ifv: 05.40-s016: (c) Copyright 1995-2005 Cadence Design Systems, Inc.
formalbuild: Total errors   = 0.
formalbuild: Total warnings = 0.
formalbuild: Successfully completed.
FormalVerifier> clock -add clk -initial 0
1 clock added.
FormalVerifier> force reset 0
FormalVerifier> run 2
Ran until 2 crank(s)
FormalVerifier> init -load -current
formalverifier: Current value of state variables loaded for static verification.
FormalVerifier> init -show

counter :
         count[2:0]                                     : 3´b000
FormalVerifier> constraint -add -pin reset 1
1 pin constraint added.
FormalVerifier> assertion -add *
formalverifier: *W,ALMAAS: "P1" is already marked as an assertion.
0 assertions added.
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Verification mode:
  P1 : Pass
Assertion Summary:
  Total                 :   1
  Pass                  :   1
  Not_Run               :   0
FormalVerifier>
```

Here, you see the initial value for count is `3'b000`, and the assertion passed. This is what you would intuitively expect from this simple example. To illustrate the importance of initialization, consider modifying the Tcl script to remove the initialization commands:

```
// counter2.tcl
1   clock -add clk -initial 0
2   init -show
3   constraint -add -pin reset 1
4   assertion -add *
5   prove
```

If you were to run the modified script, you would see something like the following:

Figure 5-2 Verification without Initialization



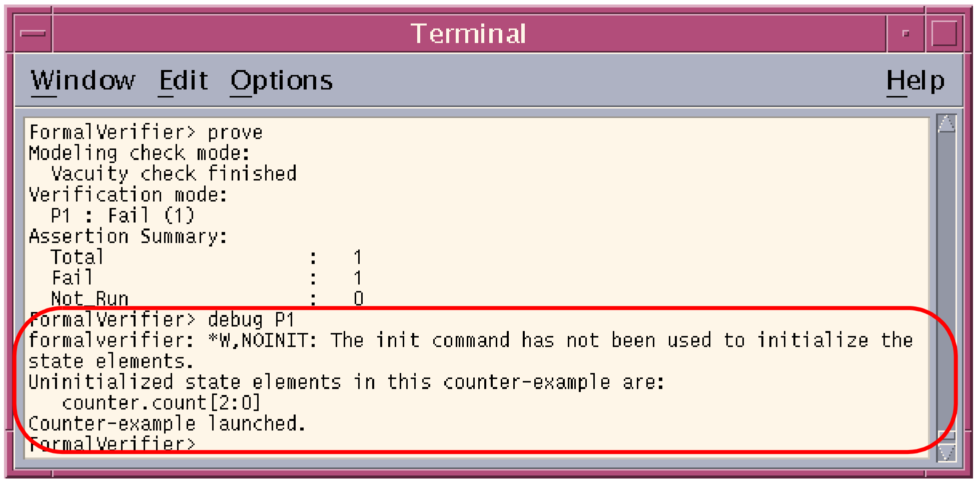
```
abvserv5:/hm/aparnag/example_INIT_TCL/version2/ifv ifv -f counter2.f
ifv: 05.40-s016: (c) Copyright 1995-2005 Cadence Design Systems, Inc.
formalbuild: Total errors   = 0.
formalbuild: Total warnings = 0.
formalbuild: Successfully completed.
FormalVerifier> clock -add clk -initial 0
1 clock added.
FormalVerifier> init -show

counter :
         count[2:0]                                        : 3'bxxx
FormalVerifier> constraint -add -pin reset 1
1 pin constraint added.
FormalVerifier> assertion -add *
formalverifier: *W,ALMAAS: "P1" is already marked as an assertion.
0 assertions added.
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Verification mode:
  P1 : Fail (1)
Assertion Summary:
  Total                   :   1
  Fail                    :   1
  Not_Run                 :   0
FormalVerifier>
```

Here, you see the initial value for count is `3'bxxx`, and the assertion failed. Without initialization, the signal `count` is undefined. Thus, the assertion is analyzed under all possible initial values for `count`. One possible initial value is the value `6`, which would violate the assertion.

Often, as in this example, you can determine by inspection that the design has not been initialized correctly. Other times, you need to invoke the `debug` command as described in Chapter 8, “Running Verification and Understanding Results,”. For the above example, you could use the following command:

```
    FormalVerifier> debug P1
```

This launches the counter-example in the SimVision *Waveform Viewer*, as described in Chapter 9, “Debugging Failures and Viewing Witnesses,”. In addition to the waveforms, Formal Verifier gives other hints that there could be a problem, as shown in the following figure:



```
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Verification mode:
  P1 : Fail (1)
Assertion Summary:
  Total                 :    1
  Fail                  :    1
  Not Run               :    0
FormalVerifier> debug P1
formalverifier: *W,NOINIT: The init command has not been used to initialize the
state elements.
Uninitialized state elements in this counter-example are:
   counter.count[2:0]
Counter-example launched.
FormalVerifier>
```

Here, you see the messages:

The init command has not been used to initialize the state elements.

Uninitialized state elements in this counter-example are: counter.count[2:0]

Be sure to pay attention to these sorts of messages, as they may lead you to the source of a failure quickly.

Explicit Initial States

Sometimes it may help to explicitly specify the initial states of some of the state elements. For example, this can be helpful, when you have several control registers that in real hardware would be initialized through some complex transactions on a bus. To avoid creating these transactions with `force` commands, it is convenient to simply specify the initial values that you want. To do this, you can use the `init -add` command, as in the following script:

```
// counter3.tcl
  clock -add clk -initial 0
  force reset 0
  run 2
  init -load -current
  init -add count = 3'b010
  init -show
```

This can also be helpful when setting memory states to some predefined values. The `init -add` command supports hierarchical references, so that you can initialize memories at arbitrary points in the hierarchy.

You should be careful when setting initial states explicitly, because doing this puts the design into a state that might be unreachable by ordinary reset methods, and in doing so could cause false results for both passes and failures. As a rule, you should strive to do initialization by hardware-supported methods (such as asserting reset) rather than by explicitly setting initial states, unless it is inconvenient to do otherwise.

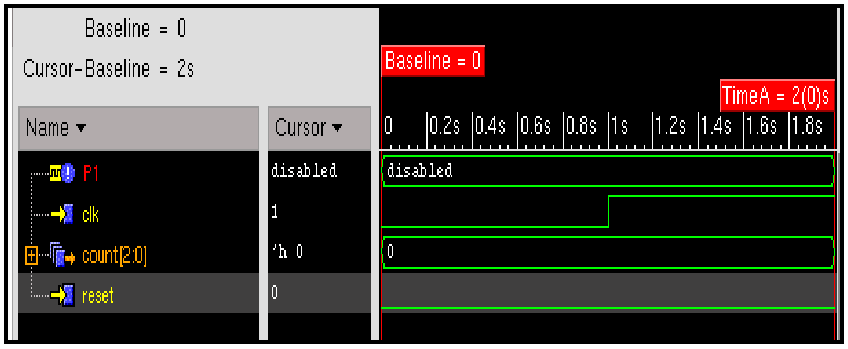
Guidelines for Correct Tcl-Based Initialization

The following guidelines should be adhered to for proper Tcl-based initialization:

- **Race between clock transition and signal changes**

  Signals should be changed (using the `force` command) on the inactive edge of the clock. If a signal is changed on the active clock edge, the race between the clock and the signal can lead to unpredictable results.

  Let us try to understand this better, by using the following waveform, which shows the reset sequence specified in `counter1.tcl` in the section titled Initialization Example on page 50



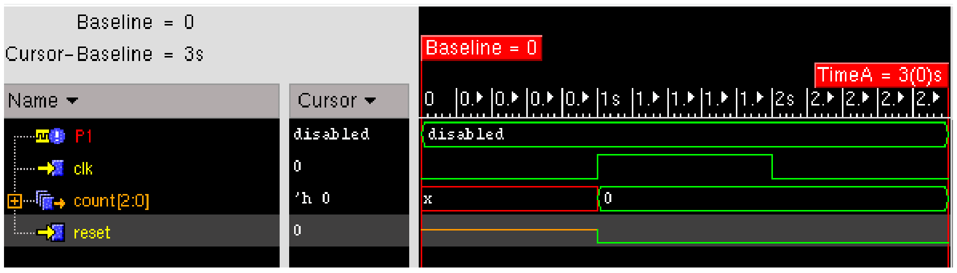
  Here, at positive edge of the clk, reset has a stable value 0, there is no race, and therefore, `count` initializes properly.

  Note: The waveform is launched using the `debug -init` command.

  Now consider the same example, but with the following initialization sequence:

```
// counter4.tcl
1   clock -add clk -initial 0
2   run 1
3   force reset 0
```

```
run 2
debug -init
```



Here, at the positive edge of the `clk`, `reset` is also changing to `0`. Thus, there is a race between `clk` and `reset`, and `count` is not guaranteed to initialize properly. In the above waveform, `count` actually does initialize properly, but it may not always do so.

- **Run simulation for complete clock cycles**

  Although the tool is often able to handle situations where simulation is run for a fractional number of clock cycles, it is advisable to run the simulation for full clock cycles. In case of multiple clocks, run simulation for at least a number of cranks equal to the *least common multiple* of all the clocks' *periods*.

  For example, if the design has two clocks, `clk1` with a period of `2` and `clk2` with a period of `5`, you should run simulation in multiples of `10` (least common multiple of `2` and `5`) cranks.

- **No testbench components in the RTL**

  The design RTL should not have any testbench components, such as initial blocks or clock generators. The initial blocks or clock generators will interfere with the initialization being done by Tcl commands, and can lead to undesired behavior.

Debugging Tcl-Based Initialization

Occasionally, you will encounter situations where initialization is not happening as you expect. In these situations, it is very helpful to view a waveform showing initialization that is happening by the `force` and `run` commands. You can view the initialization waveform any time after the first run command, using the `debug -init` command.

The waveform window is similar to that for debugging failed assertions, however in the case of `debug -init` the waveform window that opens by default, does not contain any signals. You need to add the signals that you want to view.

Refer to the section Adding and Deleting Signals on page 161 in this guide. For details on this command, refer to *Chapter 1, "Formal Verifier Command Reference,"* of the *Formal Verifier Reference Manual*.

PSL-Based Initialization

Tcl-based initialization has a drawback in that it does not allow you to check the correctness of assertions during the reset phase. In addition, constraints are not honored during the initialization simulation. These drawbacks are rarely important. However, if either of these two limitations are important in your situation, you should use PSL-based initialization.

In PSL-based initialization, the behavior of reset is modeled by PSL properties, and the reset signal is asserted and de-asserted as part of the formal analysis process rather than through simulation prior to the formal analysis process.

PSL-based initialization reduces performance, compared to Tcl-based initialization. It also has the disadvantage of a more tedious reset sequence specification. The reduced performance is because the properties themselves add complexity, and because the formal analysis engine must analyze the behavior of the assertions during reset as well as during the post-initialization period. The extra reset properties also add complexity to the process.

The following example shows an example of PSL-based initialization:

```
1   // test.v
2   module counter(clk,reset,count);
3     input clk;
4     input reset;
5
6     output [2:0] count;
7     reg [2:0] count;
8     //psl RST1: assume {1'b1} |-> {!reset[*2];reset};
9     //psl RST2: assume always {reset} |=> {reset};
10
11    // psl P1: assert reset -> never (count == 3'b110);
12
13    always @ (posedge clk)
14    begin
15      if(!reset)
16          count = 3'b000;
17      else if (count < 3'b101)
18          count = count + 1;
19    end
20
21  endmodule
```

In the above design, `Line 8` and `9` show two PSL properties that model the behavior of reset. The first property is active only once, starting at time `0`. It indicates that reset is asserted for two cranks, and then de-asserted. The second property states that once reset is de-asserted, it remains de-asserted forever.

Note:

- The assertion `P1` has been guarded by using the form:

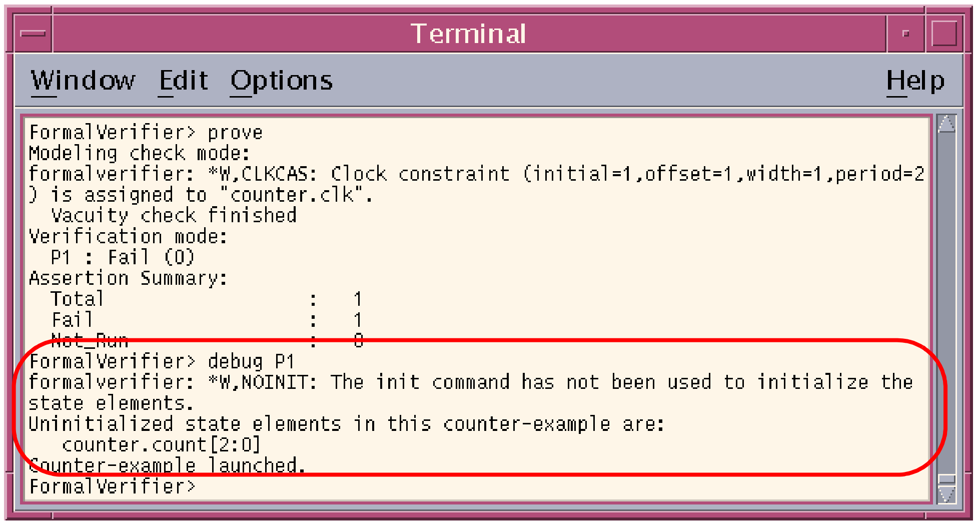
  ```
  reset -> formula
  ```

  The use of the `reset` guard stops the property from being checked during reset. If there are assertions you want to be checked during reset, then there should be no guard.

- The Tcl command `constraint -add -pin reset = 1` should not be used, as used in Tcl-based initialization. It would conflict with the assertion of reset.

- The `init -show` and `debug -init` commands, described in the section Tcl-Based Initialization on page 48 are not relevant when using PSL-based initialization. Instead, the effects of `reset` are seen in the counter-example waveforms. See Chapter 9, "Debugging Failures and Viewing Witnesses," for more information.

- When viewing counter-examples or witnesses in the context of PSL-based initialization, you will see warnings similar to those shown in the following figure:



```
FormalVerifier> prove
Modeling check mode:
formalverifier: *W,CLKCAS: Clock constraint (initial=1,offset=1,width=1,period=2
) is assigned to "counter.clk".
  Vacuity check finished
Verification mode:
  P1 : Fail (0)
Assertion Summary:
  Total                  :   1
  Fail                   :   1
  Not_Run                :   0
FormalVerifier> debug P1
formalverifier: *W,NOINIT: The init command has not been used to initialize the
state elements.
Uninitialized state elements in this counter-example are:
   counter.count[2:0]
Counter-example launched.
FormalVerifier>
```

Initialization Using Simulation Dumps

Formal Verifier supports initialization from a simulation dump in either Cadence's SST2 database format or in (E)VCD format. This initialization mechanism is useful in the following scenarios:

- When the initialization sequence is complex and cannot be written easily in the form of a Tcl testbench.

- When the testbench is already available and can be easily run in simulation.

- When the initialization sequence is not available, but you have a simulation dump, which includes the desired initialization sequence.

Formal Verifier can read the values from the simulation dump at the user-specified time. Therefore, the desired initial state can be the state of simulation at any time in the simulation dump, not necessarily at the end of simulation.

Moreover, when creating a simulation dump, the testbench is usually the design top. The design top in simulation may or may not be the same as the block you are working on. To facilitate using dumps from higher-level simulations, Formal Verifier supports mapping of intermediate signals in the hierarchy of the simulation dump to the inputs of the lower-level block on which you are working.

The following is an example showing how to use a simulation dump initialization. The example is based on the following Verilog code, showing a counter embedded in a testbench:

```
// testb.v
module block_to_test(clk,reset,count);
 input clk, reset;
 output [2:0] count;

 // Internals are unimportant

endmodule

module testbench();
 reg clk, reset;
 wire [2:0] count;

 block_to_test mydut(clk, reset, count);

endmodule
```

For the purposes of this example, assume that the following simulation dumps exist, based on simulations of the module testbench:

- SST2 format dump `../ncsim/rst_sst.shm`
- VCD format dump `../ncsim/rst_vcd.vcd`

For more information on how to create these simulation dumps, see *Chapter 11, “Debugging Your Design”* of the *NC-Verilog Simulator Help* or *Chapter 12, “Debugging Your Design”* of the *NC-VHDL Simulator Help*. Additionally, you can also refer to *Chapter 7, “Managing Simulation Databases”* of the *SimVision User Guide*.

Assuming that you are verifying the module `block_to_test`, you could use one of the following Tcl commands:

- For SST2 format dump

```
FormalVerifier> init -load -sst2 ../ncsim/rst_sst.shm -root testbench.mydut
  -time 10
```

- For VCD format dump

```
FormalVerifier> init -load -vcd ../ncsim/rst_vcd.vcd -root testbench.mydut
   -time 10
```

In the above commands, the option `-root testbench.mydut` specifies that the top block during formal analysis corresponds to `testbench.mydut` in the simulation dump. The option `-time 10` specifies that the initialization values should be read from *time 10* of the simulation dump.

The Tcl command `init -show` works as expected even with simulation dump initialization. See *Chapter 1, “Formal Verifier Command Reference,”* of the *Formal Verifier Reference Manual*.

Delaying Assertion Verification

Proper design initialization is important in formal analysis else verification can produce false assertion failures. Incase the design is supposed to be initialized on its own then you can delay assertion verification till the initialization is complete.

Consider an example, as shown below. This example has a `oneHot` circuit, which ensures that core module always gets a `onehot` input.

```
module top (A,B,clk,...);
input A,B,clk
wire [0:3] Core_in;

....
oneHot OH (Core_in,A,B,clk);
Core C (Core_in,clk);
endmodule

module oneHot (out, A, B, clk);
output [0:3] out;
input clk;
input A, B;

wire A1, A2, A3, A4;
wire O1, O2, O3, O4;

reg [0:3] out;

// One cold for wires A1,A2,A3,A4
nand I1(A1, A, B);
nand I2(A2, !A, B);
nand I3(A3, A, !B);
nand I4(A4, !A, !B);
```

```
always @(posedge Clk)
begin
    out[0] = A1;
    out[1] = A2;
    out[2] = A3;
    out[3] = A4;
end

 // Invert assignment to make output oneHot
assign O1 = !out[0];
assign O2 = !out[1];
assign O3 = !out[2];
assign O4 = !out[3];
endmodule
```

In the above example, if you start the verification of assertion from the first cycle with the assumption that `Core_in` will always be `oneHot` then the assertion will fail in the first cycle as the `oneHot` circuit is not initialized.

The command `define delay_assertion <cranks>` allows you to specify the number of cranks for which the verification of assertion needs to be delayed. By default, the number of cranks is taken as `0` and the maximum number of cranks that you can specify is `50`. Clock extraction will be disabled if the value for `delay_assertion` is set to value other than 0. Failure depth for a failing assertion and a passing depth for cover assertions will be more than or equal to the specified delay in verification.

Example

```
define delay_assertion 4
```

This command will delay the verification of assertion by `4` cranks. You will be able to view the effect of this delay in the counter-example of the properties with conclusive results. The counter-example will be visible from crank 0. The effect of this delay is not visible for properties with *Explored* status.

Limitations

- This delay will not result in verification delay for automatic formal assertions and cycle checks.
- This command will not have any effect on vacuity check.
- There will be no delay in application of constraints.
- The assume-guarantee method will get disabled.
- The combinational loop detection will not get delayed. In such cases, the assertion can get blocked at depth less than the specified delay in assertion verification.

- The abort conditions in a PSL property will not be reached till the delay depth is reached. This can result in failure of properties, which were giving a Pass status earlier as the abort condition was reached.
- During the verification of a delayed assertion, if an assertion has failed during initialization, that is, before the delayed number of cranks then such failures will get ignored and will not be reported.

6

Understanding Messages

During the design verification process, Formal Verifier analyzes the RTL and properties in many ways, ranging from simply parsing the RTL to advanced modeling checks. The analysis process can generate a significant number of messages, ranging from fatal errors to notes. It is important to understand the various messages, because they can indicate bugs in the RTL and properties. Furthermore, finding and resolving bugs using these messages is often easier than finding and resolving bugs through debugging failed assertions.

Message Conventions

There are four major severity levels associated with its messages, described in the following sections.

Fatal Messages

A fatal message results from a situation from which Formal Verifier cannot recover. Examples of situations in which fatal messages might be issued include:

- Syntax and semantic errors in the RTL and properties
- Not enough disc space
- Internal errors

In all these cases, the tool exits.

The following is an example of a fatal message:

```
NODISC  F   "Aborting the verification as the system has run out of disc"
```

The letter `F` indicates a *fatal* error.

Error Messages

An error message is reported in a situation from which Formal Verifier can recover and continue. There are various types of situations that can result in error messages, including:

- Non-synthesizable RTL constructs

  Non-synthesizable constructs are not supported generally in formal analysis. When the tool encounters such a construct, it issues an error message and blackboxes the offending construct. Refer to section titled Design Checks on page 77 for more information on glassboxing, blackboxing, and non-synthesizable RTL constructs.

- Unsupported or erroneous properties

  Unsupported or erroneous properties trigger an error message and are then ignored. Refer to section titled Unsupported Properties on page 84 for more information on unsupported or erroneous properties.

- Tcl command problem

  When Formal Verifier is unable to understand or honor a Tcl command, it issues an error message and returns control to the Tcl prompt.

  The example given below is of a command that Formal Verifier cannot understand:

  ```
  FormalVerifier> assertion -ard P1 P2 P3
  ```

  For the above command, Formal Verifier issues the following message:

  INVOPT E "Invalid option "-ard" specified for command "assertion"

  As a second example, consider the following command:

  ```
  FormalVerifier> assertion -add P1 P2
  ```

  In this example, if `P1` is a property and `P2` does not exist, then the tool is not able to honor the command and issues the following message:

  OBJNFD E "Object "P2" not found in the design"

The letter `E` indicates an *error* message.

Warning Messages

Warning messages signal potential problems and should be evaluated to determine if the potential problem is real. Formal Verifier issues warning messages for a variety of reasons, such as:

- Potential problems in the design

Often, Formal Verifier can detect potential problems in RTL. Refer to section titled RTL Checks on page 68 for more information on how Formal Verifier detects and reports on potential issues in the design.

- A Tcl command that will have no effect

  Sometimes Tcl commands are issued that will have no effect. Consider the following Tcl command:

  ```
  FormalVerifier> assertion -add P1
  ```

  When assertion *P1* has already been added, the tool issues the following warning:

  ```
  ALMAAS  W  "P1 is already marked as an assertion"
  ```

The letter `W` indicates a warning message.

Note Messages

Formal Verifier issues note messages when it might be beneficial to give the user more information. For example:

```
FSMIDN  N "In module/design-unit 'top', FSM for state register 'state_p' has been recognized.
```

The letter `N` indicates a note message.

Build-Time Messages

Processing the RTL to generate the internal model is divided into the following stages:

- Parsing
- Elaboration
- Model generation

Messages issued during these stages are printed to the terminal and stored in the log file. The following sections discuss the messages that can be issued during each of these stages.

Parsing Messages

In the parsing stage, Formal Verifier scans the HDL using syntax driven parsing techniques. In this stage, the parser issues a message when it encounters an unexpected situation such as a syntax error. The parser displays the message and stores it in the log file. If the parser is not able to recover, it stops.

Consider the following design:

```
1   module top();
2      reg s1;
3      assign s1 = s2;
4   endmodule
```

For this design, the tool issues the following message:

```
assign s1 = s2;
|
ncvlog: *E,UNDIDN (test.v,3|15): 's2': undeclared identifier [12.5(IEEE)].
```

You can view the warnings and notes that are generated during the parsing stage, by using the following command:

```
FormalVerifier> check -show parse
```

For details of the `check` command, refer to the *Formal Verifier Reference Manual*.

Elaboration Messages

After the HDL text has been parsed, Formal Verifier elaborates the design. During elaboration, the tool constructs the whole design by linking together the various modules and sub-modules that were parsed earlier. The elaborator issues a message when it encounters an unexpected situation, which are generally related to problems in the design hierarchy. The elaborator displays the message and stores it in the log file. If the elaborator is not able to recover, it stops.

Consider the following design:

```
1   // test.v
2   module top();
3      bot I1();
4   endmodule
5
6   module bot();
7      assign top.x = 1;
8   endmodule
```

For this design, the tool issues the following message:

```
assign top.x = 1;
|
ncelab: *E,CUVHNF (test.v,7|13): Hierarchical name component lookup failed at 'top.x'.
```

You can view the warnings and notes that are generated during the elaboration stage, by using the following command:

```
FormalVerifier> check -show elab
```

For details of the `check` command, refer to the *Formal Verifier Reference Manual*.

Viewing the Parsing and Elaboration Messages

You can view the warnings and notes that are generated during the parsing and elaboration stages. There are two ways to view and manage these warnings and notes:

1. On the TCL prompt, by using the following command:

   ```
   FormalVerifier> check -show parse_elab
   ```

   For details of the `check` command, refer to the *Formal Verifier Reference Manual*.

2. Through the Formal Verifier GUI, by using the *Parse_Elab* sub-tab on the *RTL Checks* tab.

   For details on using the GUI, see the section titled Viewing RTL Checks in the GUI on page 71 in this chapter.

Note: The *Error* messages are stored in `formalverifier.log` and displayed on shell prompt. This is because the tool cannot proceed if the errors reported by the tool during the parsing and elaboration stage are not fixed.

Customizing Message Severity Levels

It is now possible to change the default severity level of a message in IFV. You can convert to and from each of the different severities, that is, error, warning, or a note. Any change to or from a Fatal severity level is not allowed. The updated severity of messages will only be visible at the display level. Internally, the tool will continue to process with the original severity levels.

You can change the default severity level of messages from the command-line option as:

- To convert any warning or note to an error, type the following command at the formalverifier/formalbuild/ifv command-line option:

  ```
  formalverifier -ERRFILE error_file
  formalbuild -ERRFILE error_file
  ifv +ERRFILE+error_file
  ```

  where `error_file` lists the mnemonics of messages to be reported as error.

- To convert any error or note to a warning, type the following command at the formalverifier/formalbuild/ifv command-line option:

  ```
  formalverifier -WARNFILE warning_file
  formalbuild -WARNFILE warning_file
  ifv +WARNFILE+warning_file
  ```

  where `warning_file` lists the mnemonics of messages to be reported as warning.

- To convert any warning or error to a note, type the following command at the formalverifier/formalbuild/ifv command-line option:

```
formalverifier -NOTEFILE note_file
formalbuild -NOTEFILE note_file
ifv +NOTEFILE+note_file
```

where `note_file` lists the mnemonics of messages to be reported as note.

While changing the default severity level of messages, consider the following points:

- Any change in the severity of messages at the formalbuild level will get reflected in the fomalverifier, only when you pass the same file at the formalverifier command-line option also.
- Changing the severity level of `halcheck` and `halstruct` messages is not supported.
- If a particular message mnemonic is specified in more than one severity file, then the message will be changed based on the following order of preference, Note > Warning > Error. The note is given the first preference and an error the least.
- You cannot specify multiple files for same option. For example, type the following command at the `ifv` command-line option:

  ```
  ifv +ERRFILE+file1 +ERRFILE+file2
  ```

  In this case, only `file1` will be considered, whereas `file2` will be ignored.

Model Generation Messages

During the model generation stage, Formal Verifier converts the design into a cycle-based model. During this stage, the tool can issue warning, error, and fatal messages. If the tool encounters an error in the build stage, it drops that portion of the design and tries to proceed with the remaining design.

If the error can be localized, Formal Verifier blackboxes the portion of the design that contains the error. This is referred to as auto-blackboxing. If the tool cannot save any portion of the module or design unit, it glassboxes the entire module or design unit. This is referred to as auto-glassboxing. Blackboxing and glassboxing are discussed in section Blackbox Report on page 79.

RTL Checks

Detecting problems early in the design cycle is very important for efficient functional verification. Formal Verifier automatically performs a set of RTL checks on your design. If one of these checks raises a warning or error, it indicates that there is some problem or a potential problem in your design. Finding bugs with RTL checks can be significantly easier than finding them with assertions. Therefore, Cadence recommends that you analyze the results of these checks and fix any problems prior to assertion verification.

RTL checks are divided into the following categories:

- Design
- PSL_SVA
- OVL
- HAL
- Modeling

You can view the results of the various checks either in the GUI or with the `check` command, which generates a report in the *Console* window.

Viewing RTL Check Reports

You can use the check command to view textual reports in the *Console* window.

- To view a textual report for each of the categories listed above, use one of the following commands:

  ```
  FormalVerifier> check -show design
  FormalVerifier> check -show ovl
  FormalVerifier> check -show psl_sva
  FormalVerifier> check -show hal
  FormalVerifier> check -show modeling
  ```

- To view a textual report for all the categories, except the `HAL` category, use the following command:

  ```
  FormalVerifier> check -show -all
  ```

- To view a report based on the message severity, use following command:

  ```
  FormalVerifier> check -show -fatal
  FormalVerifier> check -show -error
  FormalVerifier> check -show -warn
  FormalVerifier> check -show -note
  ```

- To view a summary report of all checks, use following command:

  ```
  FormalVerifier> check -summary
  ```

Customizing RTL Check Reports

You can use the `check -show` command to view the textual reports of the RTL Checks done on the design. This command displays messages formatted by tag, category, and severity, as explained in the previous section Viewing RTL Check Reports on page 69 of this chapter.

In some cases, the list of messages can be very large. You, however, might be interested in only a subset of the report. You can use the `check -add` and `check -delete` commands to customize the reports as per your requirements.

You can use the `check -delete` command to remove messages of a certain tag, category, severity, or file from the reports and the `check -add` command to include messages of a certain tag, category, severity, or file to the reports.

- To remove messages of a certain *category*, use the following command:

  ```
  FormalVerifier> check -delete OVL
  ```

- To remove messages from a certain `file`, use the following command:

  ```
  FormalVerifier> check -delete -filename test.v
  ```

  Note: You cannot specify multiple files with the `-filename` option, but you can specify wildcard characters in the filename. For example, you can specify `mod*.vhd` as an argument to the `-filename` option.

- To remove messages from *set of lines* in a certain file, use the following command:

  ```
  FormalVerifier> check -delete -filename test.v -fromline 30 -toline 40
  ```

  Note: Keep the following in mind with reference to the above command:

  - You cannot specify the `-fromline` and `-toline` options without the `-filename` option.

    For example, if you do not specify the `test.v` file with the above command, the tool will not process the command and will report an error.

  - If you do not specify the `-fromline` option, the value specified with the `-toline` option is used as the default value for the `-fromline` option.

    For example, if you do not specify any value to the `-fromline` option with the above command, the tool will process the command using the value `40` specified with the `-toline` option.

  - If you do not specify the `-toline` option, the value specified with the `-fromline` option is used as the default value for the `-toline` option.

For example, if you do not specify any value to the `-toline` option with the above command, the tool will process the command using the value `30` specified with the `-fromline` option.

- ❑ The values specified with the `-fromline` and `-toline` options are included while processing the `check` command.

  For example, while the processing the above command, the tool will delete messages from the lines `30` to `40` (both inclusive) and report the other messages from the file.

- To remove messages of a certain *severity*, use the following command:

```
FormalVerifier> check -delete -warn
```

- To remove messages containing a specific *word*, use the following command:

```
FormalVerifier> check -delete -string
```

Note: Currently, you cannot remove a set of words or space-separated text.

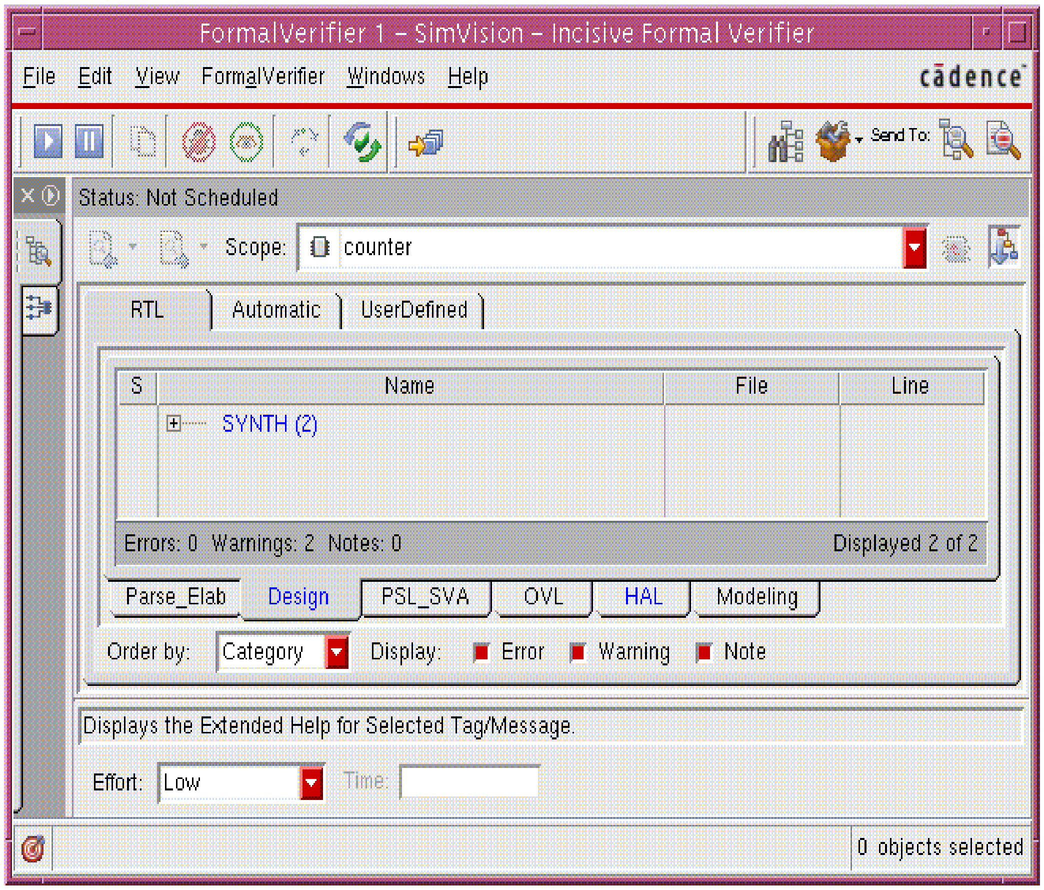
The `check -add` command has the same use model and implementation.

You can use the `check -show` command to view the modified or customized report after adding messages to or deleting messages from the textual reports.

Viewing RTL Checks in the GUI

In the GUI, the results are displayed in the RTL Checks tab of the Formal Verifier window as shown in the figure below:

Figure 6-1 RTL Checks tab


FormalVerifier 1 - SimVision - Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cādence
Send To:
Status: Not Scheduled
Scope: counter
RTL Automatic UserDefined
S Name File Line
SYNTH (2)
Errors: 0 Warnings: 2 Notes: 0
Displayed 2 of 2
Parse_Elab Design PSL_SVA OVL HAL Modeling
Order by: Category Display: Error Warning Note
Displays the Extended Help for Selected Tag/Message.
Effort: Low Time:
0 objects selected

The *RTL Checks* tab has a sub-tab for each of the six categories of checks. You choose a tab to view messages in the respective category.

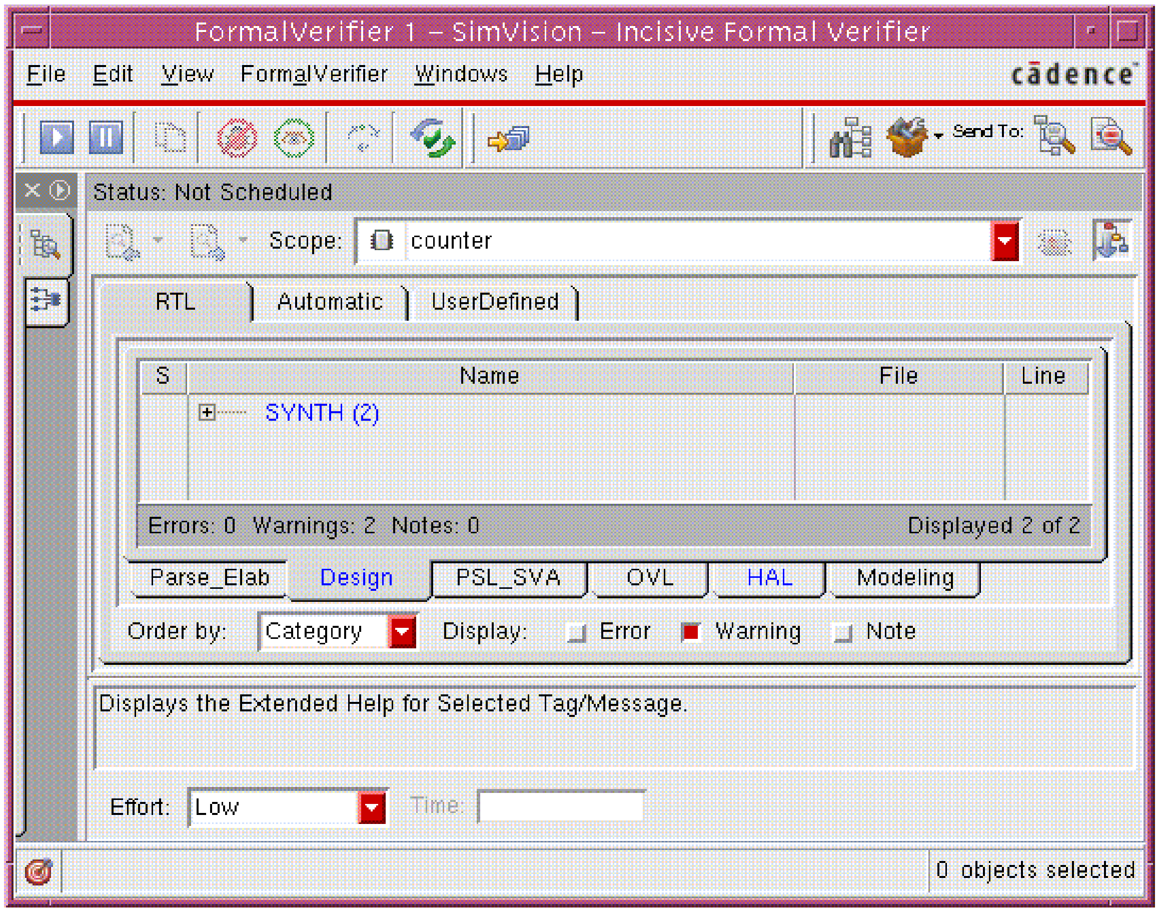
In the GUI, messages are color-coded according to the message severity, as follows:

- Error (Red)
- Warning (Blue)
- Note (Green)

The sub-tab name, within the *RTL Checks* tab, is displayed in the color of the message with the highest severity level. That is, if there is an error in any category within a given sub-tab, the color of that sub-tab is red, such as the *Design* and *HAL* sub-tabs in the Figure 6-1 on page 72. If none of the messages are errors, but there is a warning message, the color of the sub-tab is blue, such as the *Parse_Elab* sub-tab in the Figure 6-1 on page 72. If none of the messages are errors or warnings, but there is a note, then the sub-tab is green. If there are no messages, the sub-tab is not colored, such as the PSL_SVA, OVL, and Modeling sub-tabs in the Figure 6-1 on page 72.

Other things to note:

- You can also view a count of messages reported. Each tab displays the number of errors, warnings, and notes, and the total number of messages within that tab.
- By default, you can view the messages of all severity levels. However, you may choose to view messages of a specific severity level only. In that case, you can deselect the button next to the severity level and those messages will be removed from the display, as shown in the figure below:



Status: Not Scheduled

Scope: counter

S	Name	File	Line
	⊞ SYNTH (2)		

Errors: 0 Warnings: 2 Notes: 0   Displayed 2 of 2

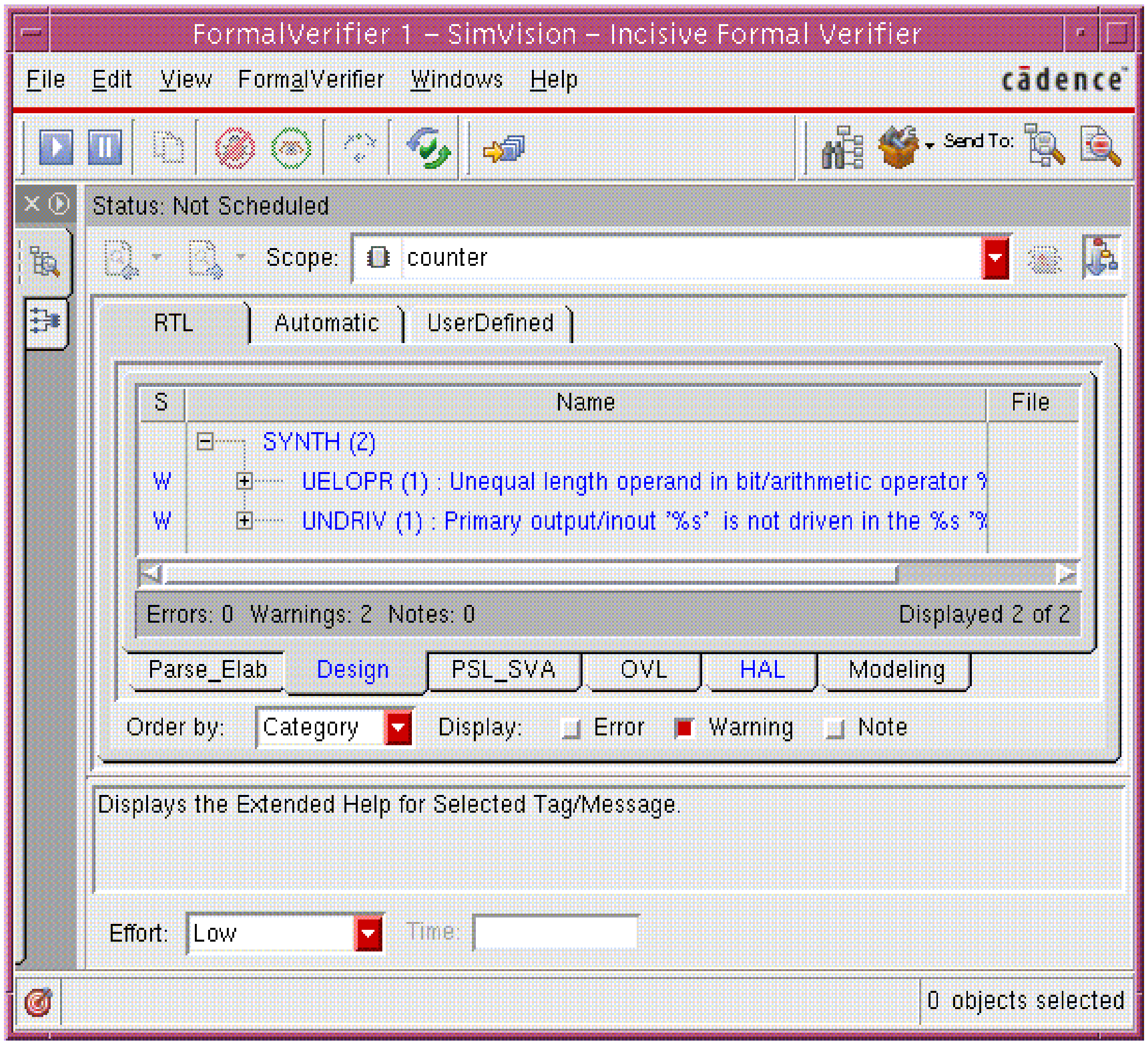
Order by: Category   Display: ☐ Error ■ Warning ☐ Note

Displays the Extended Help for Selected Tag/Message.

Effort: Low   Time:

0 objects selected

- By default, all the categories of messages within a tab are collapsed. You can click the + sign next to a category to expand it, as shown in the figure below:



Status: Not Scheduled

Scope: counter

S	Name	File
	⊟ SYNTH (2)	
W	⊞ UELOPR (1) : Unequal length operand in bit/arithmetic operator %	
W	⊞ UNDRIV (1) : Primary output/inout '%s' is not driven in the %s '%	

Errors: 0 Warnings: 2 Notes: 0 Displayed 2 of 2

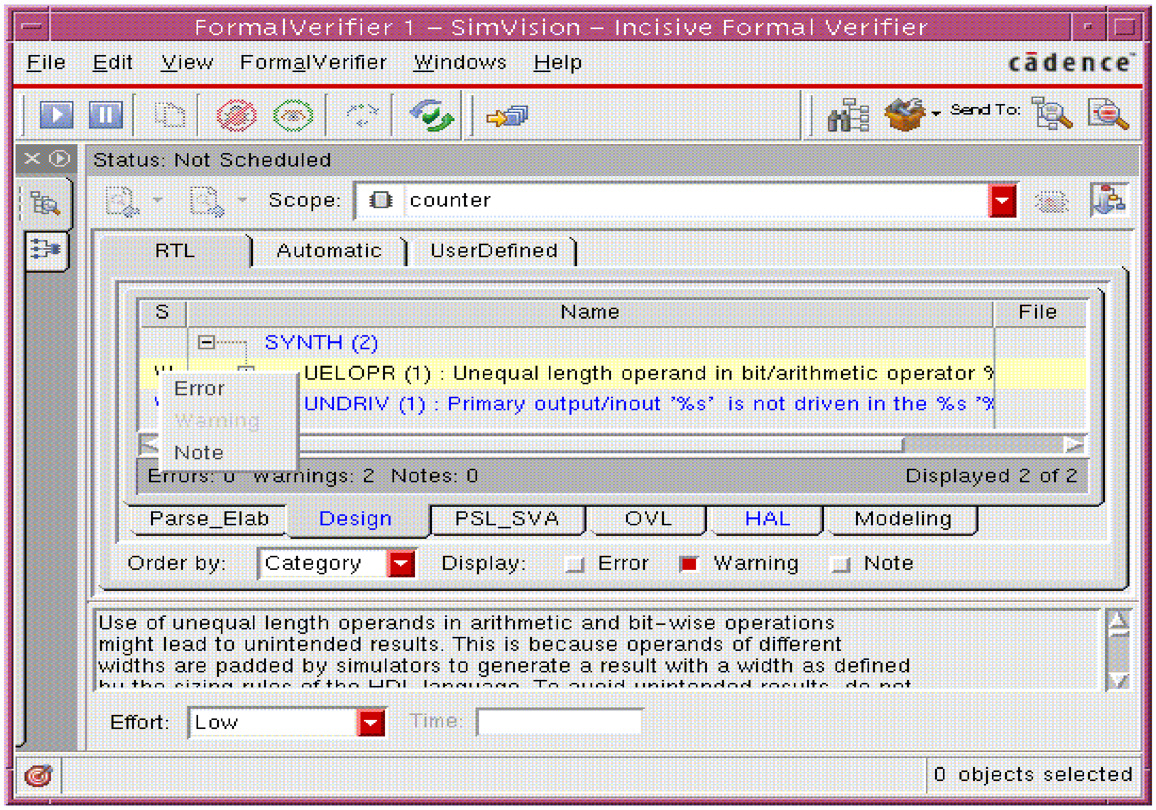
Order by: Category Display: ☐ Error [✓ Warning] ☐ Note

Displays the Extended Help for Selected Tag/Message.

Effort: Low Time:

0 objects selected

- You can modify the severity level of the RTL checks. You can right click the severity column and choose a new severity level, as shown in the figure below:

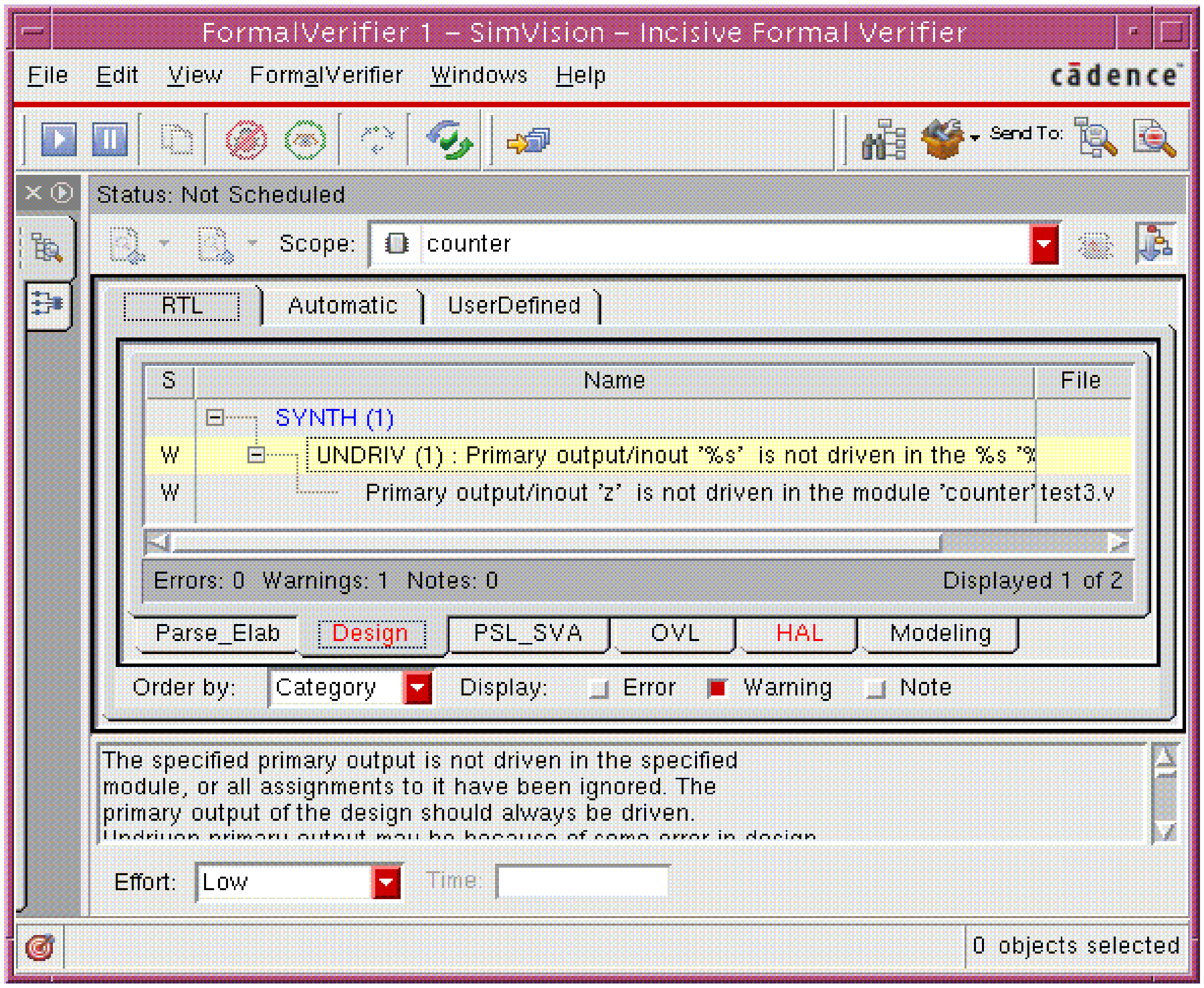

FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cādence
Send To:
Status: Not Scheduled
Scope: counter
RTL
Automatic
UserDefined
S
Name
File
SYNTH (2)
UELOPR (1) : Unequal length operand in bit/arithmetic operator
UNDRIV (1) : Primary output/inout '%s' is not driven in the %s
Error
Warning
Note
Errors: 0 Warnings: 2 Notes: 0
Displayed 2 of 2
Parse_Elab
Design
PSL_SVA
OVL
HAL
Modeling
Order by: Category
Display: Error Warning Note
Effort: Low
Time:
0 objects selected

Use of unequal length operands in arithmetic and bit-wise operations might lead to unintended results. This is because operands of different widths are padded by simulators to generate a result with a width as defined

When you modify the severity of a message, the color of the sub-tab is updated to the color of the message with the highest severity level. For example, the color of the *Design* sub-tab is red, because there is one error and three warnings, as shown in the Figure 6-1 on page 72. Now, if you decide to modify the severity of the error to a warning, the color of the sub-tab will become blue as there are only warnings on that sub-tab.

You can also use the `check -modify` command to change the severity level on the *FormalVerifier* command-line.

- You can view the extended or detailed help for a message in the Extended Help pane, as shown in the figure below:


FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cādence
Send To:
Status: Not Scheduled
Scope: counter
RTL Automatic UserDefined
Parse_Elab Design PSL_SVA OVL HAL Modeling
Effort: Low
Time
0 objects selected

S	Name	File
	SYNTH (1)	
W	UNDRIV (1) : Primary output/inout '%s' is not driven in the %s '%	
W	Primary output/inout 'z' is not driven in the module 'counter'	test3.v

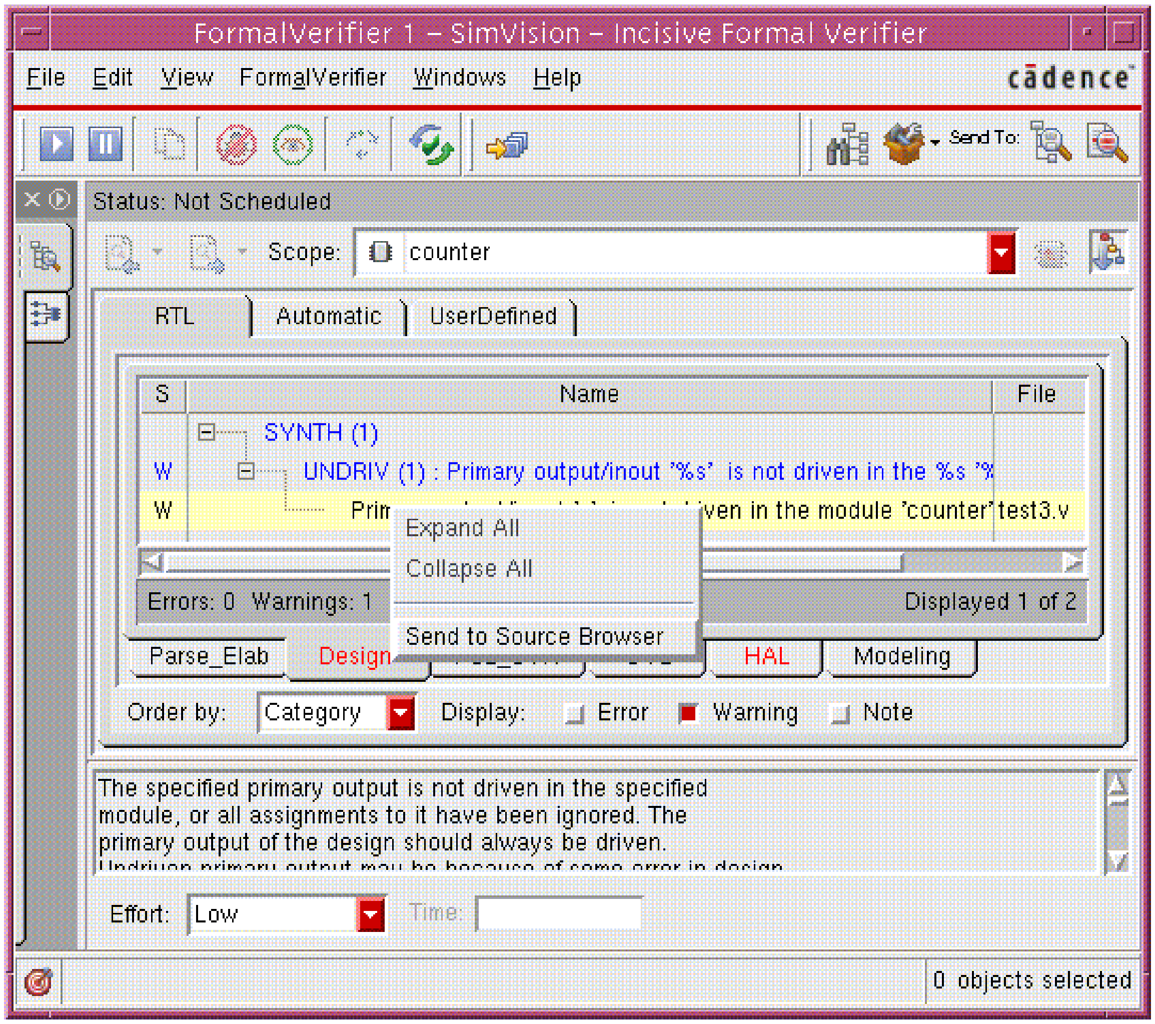
Errors: 0 Warnings: 1 Notes: 0 Displayed 1 of 2

Order by: Category Display: Error Warning Note

The specified primary output is not driven in the specified
module, or all assignments to it have been ignored. The
primary output of the design should always be driven.

Note: You can use the mouse to resize the Extended Help pane.

- To view the source code corresponding to a message, double-click the message, or right-click the message and choose *Send to Source Browser*, as shown below:


FormalVerifier 1 – SimVision – Incisive Formal Verifier

Status: Not Scheduled

Scope: counter

S	Name	File
	SYNTH (1)	
W	UNDRIV (1) : Primary output/inout '%s' is not driven in the %s '%	
W	Prim … ven in the module 'counter'	test3.v

Errors: 0 Warnings: 1 Displayed 1 of 2

Order by: Category Display: Error Warning Note

The specified primary output is not driven in the specified module, or all assignments to it have been ignored. The primary output of the design should always be driven.

Effort: Low

0 objects selected

- You can filter messages in the RTL Checks tab based on *Category*, *Tag*, or *Message*.
- You can sort messages by clicking on the column headers.

Design Checks

Design checks are related to tool setup and contribute to improving the quality of formal analysis. In this domain, the following kinds of checks are done:

- Setup and tool checks
- Unsupported construct checks
- Blackbox report

- Synthesizability and structural checks
- Race condition checks
- Finite state machine checks
- Clock domain crossing checks
- Lint Checks on PSL/SVA Expressions

The following sections describe each of these kinds of checks.

Setup and Tool Checks

These checks include checks related to the tool setup and command-line use. These messages are issued when you do not use correct command syntax or when there are environment or setup issues. Some examples of these checks are:

- The specified blackboxed or glassboxed unit is not present in the design.
- An error occurred while reading a file.
- Processing stopped because virtual memory usage was exceeded.
- The tool was unable to check out a license.
- The specified top-level instance is not present in the design.

Unsupported Construct Checks

When Formal Verifier detects an unsupported construct in the design, it issues a message. You should either remove unsupported constructs, or be aware that the unsupported constructs are not taken into account during formal analysis.

Several HDL constructs are unsupported, due to their rare use or non-synthesizability. These include:

- Use of real variables or time variables
- Guarded assignments
- Shared variables
- System function calls
- Repeat event specification

- Simulation specific `constructs`: forever `construct`, disable `construct`, fork-join `construct`, and wait `construct`
- Named event control

For a more complete discussion of unsupported constructs, refer to the *Formal Verifier Reference Manual*.

Blackbox Report

Sometimes the tool may encounter an unsynthesizable construct, or it may be unable to process some part of the design. For example, if it encounters an unknown language. The tool automatically blackboxes or glassboxes the problematic part of the design.

Blackboxing and glassboxing are defined as follows:

- Glassboxing

  In glassboxing, all the design constructs in a module, other than instances of other modules inside it, are ignored. Only modules are glassboxed. When a module is glassboxed, all instances of that module are also glassboxed, and you cannot see any portion of the design in that instance. However, you can traverse the instances and hierarchy below it.

  Consider the following design:

```
module top(clk, rst, out);
  input clk, rst;
  output out;

  reg out2;

  always @(posedge clk or negedge rst)
  begin
   if (rst == 1)
    out2 = 1'b0;
   else
    assign out2 = 1'b1;
  end

endmodule
```

For this design, Formal Verifier issues the following messages:

*formalbuild: *E,RECFLE: No combinational circuit or sequential element could be recognized for top.out2[0:0], The sensitivity list will be used as the trigger.*

*formalbuild: *W,GLSBOX: Module worklib.top automatically glassboxed.*

- Blackboxing

In blackboxing, all design constructs in a portion of design are ignored. The entire hierarchy below the blackboxed module or design unit is not available for verification.

Consider the following design:

```
ARCHITECTURE rtl OF test IS
BEGIN
  out1 <= (FALSE, TRUE, FALSE, TRUE) ror 2;
END rtl;
```

For this design, Formal Verifier issues the following message:

```
out1 <= (FALSE, TRUE, FALSE, TRUE) ror 2;
|
formalbuild: *E,SHFTOP (test1.vhd,16|0): In design unit test:test, unsupported shift operation
ror encountered.
```

The blackbox category of checks reports those constructs in the design that have been blackboxed, glassboxed, or ignored. These checks include:

- A loop with more iterations than the specified limit
- Memory size greater than the specified limit
- A formal port list of a design unit or subprogram is dropped due to errors

Synthesizability and Structural Checks

Synthesizability and structural checks identify code that is non-synthesizable or that is likely incorrect from a structural perspective. These checks also identify code that could result in differences between synthesis and simulation. Some examples of these checks are:

- Width mismatches
- Incorrect modeling style, such as:
  - Bit-select or part-select outside of the defined range
  - Null slice range of slice expression
- Undriven nets or primary outputs in the design
- Incomplete sensitivity list
- Multiply-driven bus or wire
- Combinational loops
- Array size or shape mismatch
- Infinite and non-static loops

- Ports used but not connected in an instance
- Port size mismatch between module instantiation and declaration
- Register is driven in more than one block or process

Race Condition Checks

Race conditions cause critical and complex issues that can consume significant debugging effort. Formal Verifier helps you resolve these issues by identifying race conditions.

Race conditions occur between signals that are sensitive to the order of execution. When a model contains order-dependent code, a race condition can cause unexpected results between successive clock cycles.

Formal Verifier performs the following race condition checks:

- Write-write race

  This occurs when two or more processes that share the same clock signal simultaneously write to the same signal.
- Read-write race

  This occurs when a signal is referenced in the same clock cycle in which it is written.
- Trigger propagation race

  This occurs when a signal, driven in one process, triggers a second process, which in turn drives a signal that triggers the first process.
- Non-blocking assignment encountered in a combinational block
- Signal is assigned by both blocking and non-blocking assignments

Finite State Machine Checks

Finite state machines (FSMs) are common in RTL designs. Bad coding of FSMs can lead to issues during later stages of the design cycle. Formal Verifier checks the design for compliance with good FSM design practices and common errors, including:

- FSMs that do not adhere to modeling style guidelines
- FSMs that have unreachable states
- FSMs that have terminal states

- Extraneous logic in the sequential portion of the FSM
- A variable tag used in the FSM

Clock Domain-Crossing Checks

Designers often face issues with the reliability of data transfer across clock domains. Synchronization failures between asynchronous clock domains can allow data to cross domains and violate setup and hold requirements. This can cause metastability (an unstable state that is neither 0 nor 1). A design must synchronize all signals that cross clock domains to avoid unpredictable behavior and problems related to metastability.

Formal Verifier performs the following clock domain checks statically without the need for simulation:

- Identification of signals crossing from one clock domain to another without proper synchronization
- Classification of the synchronizer hardware for a given domain crossing

Lint Checks on PSL/SVA Expressions

If there are lint errors in an expression (PSL/SVA properties or design), they can lead to incorrect verification results. For example, in Verilog, you can assign signals of different widths. However, if the width of a signal on the RHS is larger than the width of a signal on the LHS, the value is truncated.

If you have specified signals of different widths by mistake, this can lead to unexpected and inaccurate verification results. To prevent this, Formal Verifier, by default, does lint checks on expressions.

Note: Currently, these lint checks are not extended to the assertion language syntax but limited to Verilog/SystemVerilog/VHDL expressions used inside these languages.

In the TUI, the `check -show` command lists the results of these checks. In the GUI, you can view the results in the *Design* sub-tab of the *RTL Checks* tab.

OVL Checks

Proper definition of properties is important for meaningful verification. Formal Verifier automatically detects problems in OVL properties, thereby reducing the amount of debugging necessary to deal with these problems. Some of the checks done on OVL properties include:

- The `reset` of an assertion is tied low (`reset` asserted low).
- A clock port of the assertion is tied to a constant.
- A port of assertion has an `X` or `Z` value.
- A port of the assertion has an unsynthesizable expression.
- The start event is always disabled.
- The end event is always enabled.

For more information about OVL checks, refer to the *Formal Verifier Reference Manual*.

PSL_SVA Checks

Formal Verifier automatically detects problems in PSL_SVA properties. Some of the checks done on PSL_SVA properties include:

- PSL_SVA trivially true or false property checks
- PSL_SVA unsupported properties
- Miscellaneous PSL_SVA checks

Each of these is discussed in the following sections.

Trivial Property Checks

Trivial properties are those whose verification status is independent of the design context. Formal Verifier may identify a property as being trivially true or trivially false.

Examples of trivial properties include:

```
true    always 1
false   never 0 -> expr
true    always expr -> 1
true    always d || !d
false   always d && !d
true    always {a;b} |-> {true}
true    always data1==0 || data1==1
false   never 1
```

```
true    never d && ! d
```

If a property is trivially true or trivially false it may indicate an error in the property. Therefore, you should examine trivial properties to make sure they are correct.

In some instances, you may want to write properties that can be evaluated at compile time. These properties would most likely be trivial, for example, properties with parameterized code. Cadence recommends that you write assertions for the parameters themselves, to ensure that the module has been instantiated with legal parameter values, as in the following:

```
    // psl param_mem_width_good: assert MEM_WIDTH > 0;
```

Because `MEM_WIDTH` is a constant value, the assertion can be evaluated as either trivially true or trivially false.

Unsupported Properties

Formal Verifier identifies unsupported PSL_SVA properties. A property is unsupported if it contains one or more of the following:

- Unsupported PSL operators, syntax, or SERE
- Unsupported or unsynthesizable HDL expressions
- `X` or `Z` in HDL expressions

For details on currently unsupported constructs, refer to the *Formal Verifier Reference Manual* and your Cadence account manager or application engineer.

Miscellaneous Checks

Some other checks that are performed on PSL_SVA properties include:

- Existence of user-defined properties in the design

  If no user-defined properties exist, you may have inadvertently made a mistake in your setup.

- Property is unclocked

  Unclocked properties have some good uses, but are much less common than clocked properties. More often than not, unclocked properties are a result of a mistake.

- Property has mixed clock-edge expressions

  Mixing clock-edge expressions in a PSL_SVA property is legal, but most likely a mistake.

HAL Checks

HDL Analysis and Lint (HAL) performs over 250 types of checks. The checks done in the *Design Checks* category are a subset of checks done in HAL, considered the checks most likely to relate to real problems. Examining HAL messages is a good practice after all messages have been examined in the other categories.

Enabling/Disabling Individual HAL Checks

You can disable individual HAL checks in IFV by specifying the `on/off` param for a check at the end in the local definition file and then pass this definition file to the IFV. For example, to disable CLKDMN rule within the HAL checks, modify the local.def file as:

```
include default
```

```
params CLKDMN {off}
```

Note: To enable a particular check, you need to specify `on` in place of `off` in the above example.

You can then pass the local.def file to IFV using the '`+rulefile`' option as:

```
% ifv <design file> +rulefile+./local.def
```

For more information about HAL checks, refer to the *HAL User Guide* and *HAL Reference Manual*.

Modeling Checks

Modeling checks are automatically done by Formal Verifier to ensure that the design and properties are set up properly. In the modeling check mode, Formal Verifier performs the following checks:

- Clock Modeling Checks
- Structural Checks
- Initialization Checks
- Vacuity Check

For more information about Modeling checks, refer to section titled Modeling Check Mode on page 114 of this guide.

7 Developing and Using Properties

In formal analysis, ***properties*** are the basic unit of expression. Properties are formalized statements about the behavior of signals over time. They are expressed in a property language, such as PSL, SVA, or by a library of properties, such as OVL and IAL.

Properties can be used to express the:

- Desired behavior of a design under test, or
- Behavior of the environment in which the design is embedded.

Properties that express the desired behavior of a design under test are called *assertions*. Properties that express the behavior of the environment are called *constraints*, because they constrain Formal Verifier to generate only those input sequences that satisfy the properties of the environment.

Because of their importance, developing properties is central to using formal analysis. Developing properties is perhaps, one of the most interesting and challenging activities you will do during formal analysis.

This chapter discusses the following topics related to property development and subsequent usage in the tool:

- **Classes of properties**

  There are several types of properties, including *safety* properties, *liveness* properties, and *existential* properties. Understanding these classes of properties will help you understand how they can be applied, how the tool shows these properties, and how to evaluate property failures.

- **Property specification mechanisms and coding guidelines**

  Formal Verifier supports several property specification mechanisms, such as PSL and SVA. It is important to understand how these mechanisms are supported and how to select the one to use. It is also important to follow some basic coding guidelines to help you use Formal Verifier more efficiently, to verify the properties you write.

- **Using properties in Formal Verifier**

After you have coded your properties, you need to indicate which properties are *assertions* to be verified and which properties are to be used as *constraints* during verification of the assertions. There are several Tcl commands that help you do this.

- **Recommended approach to developing and analyzing properties**

  Developing properties for a block can be a daunting task for someone new to the process. Cadence recommends a structured approach to property development, which starts out with protocol properties on interfaces, then moves on to whitebox properties, and finishes with blackbox functional properties.

  In addition to a structured approach, Cadence recommends an incremental approach, in which you first turn off or inactivate some of the interfaces to a block. This allows you to gradually build up the complexity of the properties. The use of an incremental approach helps you find and resolve bugs more quickly than if you tried to build a complete and general set of properties prior to verification.

Classes of Properties

There are three major classes of properties:

- Safety
- Liveness
- Existential

These three classes are discussed in the following sections.

Safety

A *safety* property is a property whose failure can be demonstrated using a finite sequence of states. For example, if `P` is a Boolean proposition, such as:

- `P := count[2:0]==3'b101`
- `P := x || y && z`

then, the property `never P` is a *safety* property, because a failure of this property can be shown with a finite sequence of states that ends with a state in which `P` is true. Similarly, the properties `always not P`, `always P`, and `never not P` are all *safety* properties.

Many other more complicated properties are safety properties. The following are other examples of safety properties:

- always `P` implies `Q` is true 3 states later

  A failure of this property could be demonstrated by a finite sequence of states in which `P` is true in some state and then three states later, `Q` is not true.

- always `P` implies `Q` will happen before `R`

  A failure of this property could be demonstrated by a finite sequence of states in which:

  a. `P` is true in some state.

  b. Some number of states later `R` is true.

  c. `Q` was not true in any of the intervening states.

Safety properties are by far the most common properties. In fact, these assertions are so common that OVL only supports safety properties.

Liveness

A *liveness* (sometimes called *eventuality*) property is a property whose failure can be demonstrated only using an infinite sequence of states. For example, the property `always P implies eventually Q` is a *liveness* property, because a failure of this property can only be shown with an infinite sequence of states such that:

1. `P` is true in some state.
2. There is an infinite number of states after `P` is true.
3. `Q` is not true in any of the states including and following the one in which `P` is true.

Since all implementable hardware has a finite number of possible states, the infinite sequence of states can be represented as an initial, non-repeating sequence of states followed by an infinitely repeating finite sequence of states, termed a *loop*. An example of such a sequence is:

```
s0, s1, s2, s3, s4, s2, s3, s4, s2, s3, s4, ...
```

In this example, the state sequence {`s0, s1`} is the initial, non-repeating sequence and the state sequence {`s2, s3, s4`} is the repeating sequence.

OVL does not support *liveness* properties; however you can model liveness in PSL using the `eventually!` or `!` operators. The following are two examples of *liveness* properties:

```
// psl property A1 = always P -> eventually! Q;
// psl property A2 = always {P} |-> {[*];Q}!;
```

The two examples mean the same thing; however the first uses the `eventually!` operator and the second uses the `!` operator at the end of the right-hand-side SERE.

Liveness properties are used less often than safety properties; however when applicable, they are very useful. They are especially useful when you do not know much about the internal operation of a design. For example, *liveness* properties are often used with arbiters. A typical arbiter has a set of request inputs and a set of grant outputs. Often, the desired behavior of an arbiter is that a grant should eventually happen if the corresponding request was asserted. This can be stated as follows:

```
    // psl property reqEgnt = always req[i] -> eventually! gnt[i];
```

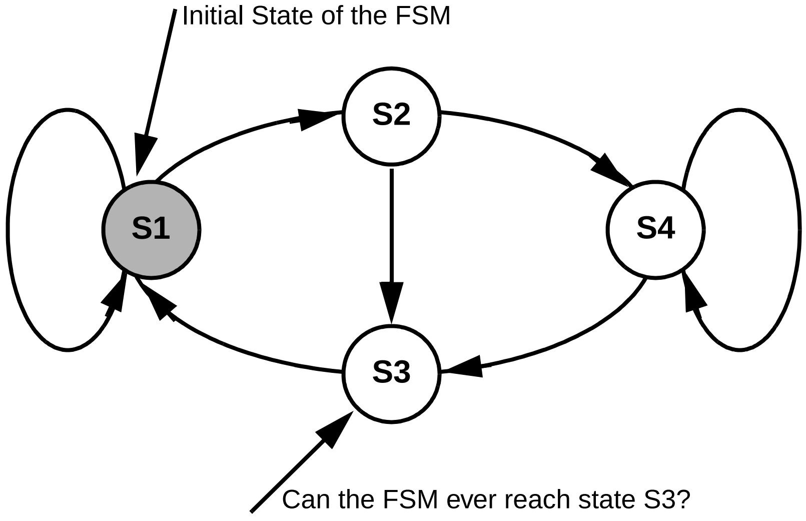
In this property, all that is being said is that the grant will eventually come. It may come after 1 cycle, 2 cycles, 10 cycles or even later.

Existential

The *safety* and *liveness* properties explained above are used to describe valid behavior of the design. *Existential* properties, on the other hand, are used to check if certain behavior of the design (in the context of the constraints) can ever happen.

These assertions are also referred to as *Witness* assertions, because Formal Verifier generates a *witness* waveform for these assertions when they pass. Refer to Chapter 9, “Debugging Failures and Viewing Witnesses,” for more information on waveforms in general and witness waveforms specifically.

Consider the following state diagram:



You may want to check whether the state machine can ever reach state `S3` if it is initialized to state `S1`. In PSL, an *existential* assertion that checks this can be written as follows:

```
// psl coverS3: cover {state == S3} @(posedge clk);
```

In general, the PSL `cover` directive can be applied to any SERE to create an *existential* assertion. Other existential assertions available in Formal Verifier are as follows:

- Deadcode assertions

  Refer to Chapter 10, "Automatic Formal Analysis," of this guide for more information.

- Automatic OVL and PSL trigger checks

  Refer to the section titled Automatic Trigger Checks for OVL, PSL, and SVA on page 141 of this guide for more information.

Note: *Existential* properties cannot be used as *constraints*.

Which Property Language?

Formal Verifier supports both PSL and SVA property language. It also supports OVL and IAL libraries. You can either use property languages or both property languages and libraries in the same file and in the same module. PSL, SVA, OVL, and IAL have complementary strengths and weaknesses.

OVL and IAL strengths are:

- Many simple properties are expressed concisely in these libraries.

  Clock and reset are a part of almost all OVL and IAL libraries. Since clock and reset are always used, this feature of OVL and IAL enables concise properties.

- OVL is supported by all VHDL and Verilog simulators.The OVL library is available in both the Verilog and VHDL versions. An IAL library is available only in Verilog version. However, it provides VHDL package for the components, which enables it to be used for mixed-language designs.

- Commonly used patterns are available in OVLand IAL libraries.

Important

The assertion names for OVL (as they appear in the Assertion Table) do not match with those in the Source Browser. The Source Browser shows the macro names, which are same as the IAL names of the assertions. This is because internally the standard implementation is based on the IAL. Please be aware of this behavior.

Note: To know about the IAL components that are supported by IFV, refer to *Incisive Assertion Library Reference Guide*.

PSL and SVA strengths are:

- Liveness can be expressed

  Liveness can be critical in some verification applications. Use of *liveness* properties can improve verification efficiency in some cases.

- Complicated temporal behaviors are expressed easily

  With OVL, complicated temporal behaviors often require auxiliary state machines and other RTL code.

Basic Property Coding Guidelines

Following basic property coding guidelines improve your productivity. The following sections discuss Cadence's recommended guidelines.

Natural Language Descriptions

Cadence recommends that you write natural language descriptions in comments next to the definition of each property you write. This enables better readability and maintainability.

Location

Cadence recommends that you embed properties in the RTL files if you have write permission on these files. Otherwise, you can create properties in external PSL vunits or an HDL testharness.

A testharness is a higher level RTL file in which the DUT is embedded. Consider the following example of a testharness:

```
module fifo_tb(clk, rst, push, pop, data_in, data_out); input clk, rst, push,
pop;
  input [15:0] data_in, data_out;

fifo myfifo
 (
   .clk(clk),
   .rst(rst),
   .push(push),
   .pop(pop),
   .data_in(data_in),
   .data_out(data_out)
 );
//
// properties and auxiliary code go here.
//
endmodule
```

Embedding Properties in RTL

Whenever possible, Cadence recommends that you place *assertions* and *assumptions* directly in the RTL file. This has the following advantages:

- Embedded properties travel with RTL code, which makes file management and version control easier.
- Embedded properties make the RTL code self-documenting.
- Embedded properties ease debugging by reducing switching between multiple files.

PSL vunits

In some cases it might not be possible, or it is not desirable, to embed properties in the RTL source code. In these cases, Cadence recommends that you place the properties in a property file external to the design modules. The following are some examples:

Verilog Example

```
// test1.prop
vunit controller_props (controller) {
   default clock = (posedge clk);
   // do not overflow FIFO
   no_overflow: assert never
      (write && full && !read && rst_n);
}
```

In this example, a PSL assertion is contained in an external property file using Verilog expression syntax. The `vunit` is bound to a Verilog module named `controller`. To read in this property file along with the rest of the RTL, you need to add the following line to your `.f` script file:

```
+vlogpropfile+test1.prop
```

VHDL Example

```
-- test2.prop
vunit controller_props (CONTROLLER(RTL)) {
   -- do not overflow FIFO
   assert_no_overflow : assert_never
      PORT MAP (clk, rst_n, write_AND_full_AND_no_read);
```

In this example, an OVL property is contained in a property file using VHDL expression syntax. The `vunit` is bound to the architecture `RTL` of an entity named `CONTROLLER`. To read in this property file along with the rest of the RTL, you should add the following line to your `.f` script file:

```
+vhdlpropfile+test2.prop
```

HDL Testharness

Perhaps the most straightforward way to write properties without modifying the RTL is to create an RTL testharness which contains your properties. When using this technique, you should duplicate, in the testharness, the ports of the device under test. Doing this will ensure that Formal Verifier does not issue numerous warning messages about undriven or unused ports.

Naming Conventions

A naming convention can simplify property manipulation by enabling the use of wildcards in Tcl commands. A naming convention also improves readability.

Property names should encapsulate a description of the behavior. The property name can also be viewed as an abbreviated version of the property's description. Properties share the namespace with all other objects in the RTL code, so take care to choose unique names. Also, avoid choosing ambiguous names, such as `assertion1`. As shown in the following PSL/VHDL example, the assertion `no_overflow` is a unique and meaningful name.

```
-- do not overflow FIFO
-- psl no_overflow: assert never
--              (write AND full AND (NOT read) AND rst_n)
-- @(rising_edge(clk));
```

Cadence recommends that you use a prefix to identify whether a property is intended to be an *assertion* or a *constraint*. You should determine whether the property is an *assertion* or a *constraint* when you create it and with this information, choose one of the following prefixes:

- `assert_`

  Use this prefix when you intend a property to be used as an *assertion*.

- `assume_`

  Use this prefix when you intend a property to be used as a *constraint*.

The following OVL/Verilog example illustrates the use of the recommended prefix.

```
// do not underflow FIFO
assert_never assume_no_underflow
```

```
    (clk, rst_n, (read && empty));

// must be able to fill up the FIFO with a sequence of writes
assert_cycle_sequence #(0, FIFO_DEPTH) assert_fill_up
    (clk, rst_n,
        {{FIFO_DEPTH-1 {!read && write && rst_n }}, full}
);
```

Note: If a module to be instantiated is prefixed with `assert_` and has no primary output port, IFV detects that module as an OVL module. This functionality gives you the flexibility to extend OVL library and add new library elements.

However, after the module has been identified as an OVL module, you can either mark the complete instance of the module as an assertion or as a constraint. You cannot independently select each property of that OVL module as an assertion or a constraint. Also, if an assertion within this module fails, then the verification for the rest of the assertions present in this module stops.

There are other coding guidelines pertaining to property libraries and verification components. More information on this and on property coding conventions in general, is available in the methodology paper titled *Property Coding Conventions* from Cadence. Refer to the section titled Other References on page 13 of this guide for more information on how to get this paper.

Auxiliary Code

Auxiliary code is RTL whose purpose is to facilitate writing properties. Signals defined in the auxiliary code can be used in any place where you need more expressiveness than the property language affords. For example, often auxiliary code is used to compute combinational functions. Also, auxiliary code can be used to create counters or other types of RTL that keeps track of states. This can be useful for storing values at one point in time and using those same values later in a property.

Cadence recommends the following:

- Break up properties with complex expressions. Create auxiliary signals for these complex expressions and use the auxiliary signals in the property.

  These auxiliary signals can be observed in the *Waveform Viewer* and *Source Browser*. This improves readability and debug-ability.

- Isolate property related code in Verilog using the following:

```
`ifdef ABV_ON
    <property code>
`endif
```

- Isolate property related code in VHDL using **`generate`** blocks conditional on a **`generic`** of type Boolean called **`ABV_ON`**

  For OVL properties, you can declare auxiliary signals in the `generate` block.

  For PSL properties, you must declare auxiliary signals in the `architecture`. You can place auxiliary signal assignments in the `generate` block.

Out-of-Module References (OOMRs)

The Verilog language supports OOMRs. OVL, PSL, and SVA support OOMRs when used in a Verilog context.

The VHDL language does not support OOMRs. However, it is possible to reference signals in a different scope using a Cadence specific utility called `nc_mirror`.

nc_mirror

Use the `nc_mirror()` procedure or `$nc_mirror()` system task to trace the value of any VHDL signal/port/variable or Verilog variable/wire lying anywhere in the design from within a VHDL architecture or Verilog module. This procedure establishes a wire between the destination and source so that the value of a VHDL signal/port/variable or the value of a Verilog variable/wire can be read.

VHDL nc_mirror() Procedure

Parameters can be associated by position or by using named association.

```
-- Parameters associated by name
nc_mirror (destination => "destination",
   source => "source",
   verbose => "verbose");

-- Parameters associated by position
nc_mirror ("destination", "source", "verbose");
```

The `nc_mirror` procedure should be written inside a process block.

Verilog $nc_mirror() System Task

Parameters can be associated only by position. Named association is not supported.

```
$nc_mirror ("destination", "source", "verbose");
```

The `$nc_mirror` task should be specified in an always block. Since `$nc_mirror` effectively translates into a wire, the sensitivity list and context conditions are ignored.

Parameters

The following table shows the parameters for the `nc_mirror` procedure/system task.

`“destination”`	The path of a VHDL signal/port/variable or Verilog variable/wire to which the value of the source signal is to be mirrored. The path should be a full hierarchical path and should be in the same design unit in which the `nc_mirror` procedure or the `$nc_mirror` system task is written.
`“source”`	The full hierarchical path of a VHDL signal/port/variable or Verilog variable/wire that is to be mirrored or traced.
`“verbose”`	The `verbose` parameter is used for status reporting. This parameter is optional and is not supported in IFV.

The “destination” and “source” parameters should be a full hierarchical path. A full path starts with the name of the top-level module if the top-level is Verilog, or with `:` if the top-level is VHDL. For example:

```
(Top-level is Verilog)
nc_mirror(“tb_top:mon:in2”, “tb_top:DUT:io”, “verbose”);

(Top-level is VHDL)
nc_mirror(“:mon:in2”, “:DUT:io”, “verbose”);
```

The following examples illustrate the use of `nc_mirror` and `$nc_mirror`:

Example 1

```
nc_mirror(destination => “:dest”,
  source      => “:I1:src”,
  verbose     => “verbose”);
```

In this example, a VHDL signal called `dest` has been declared in the top-level VHDL architecture. The `nc_mirror` routine is used to mirror the value of a signal called `src`, which is in a lower-level design unit called `I1`, which could be VHDL or Verilog. The parameters are associated using named association.

Example 2

```
nc_mirror(destination => “:dest”,
  source      => “:I1:src”);
```

In this example, the optional `verbose` parameter is omitted.

Example 3

```
nc_mirror(“:dest”, “:I1:src”, “verbose”);
```

In this example, the parameters are associated by position.

Example 4

```
reg dummy_event;
always @(dummy_event)
begin
$nc_mirror(“top.dest”, “top.I1.src”, “verbose”); // Using absolute paths
$nc_mirror(“top.dest”, “top.I1.src”); // verbose parameter is omitted
end
initial #0 dummy_event=1;
```

In this example, if a signal called `dest` is declared in a Verilog module, you could use the $nc_mirror system task to mirror the value of `src`, which is in a lower-level design unit called `I1`.

Example 5

```
always @(posedge clk or negedge rst)
    if (!rst)
        $nc_mirror(“top.dest1”, “top.I1.src1”);
    else
        $nc_mirror(“top.dest2”, “top.I1.src2”);
```

In this example, even if the `$nc_mirror` is written in an always block, the assignment is not sensitive to the list of always block or if-else condition.

The always block will get translated into two wires as:

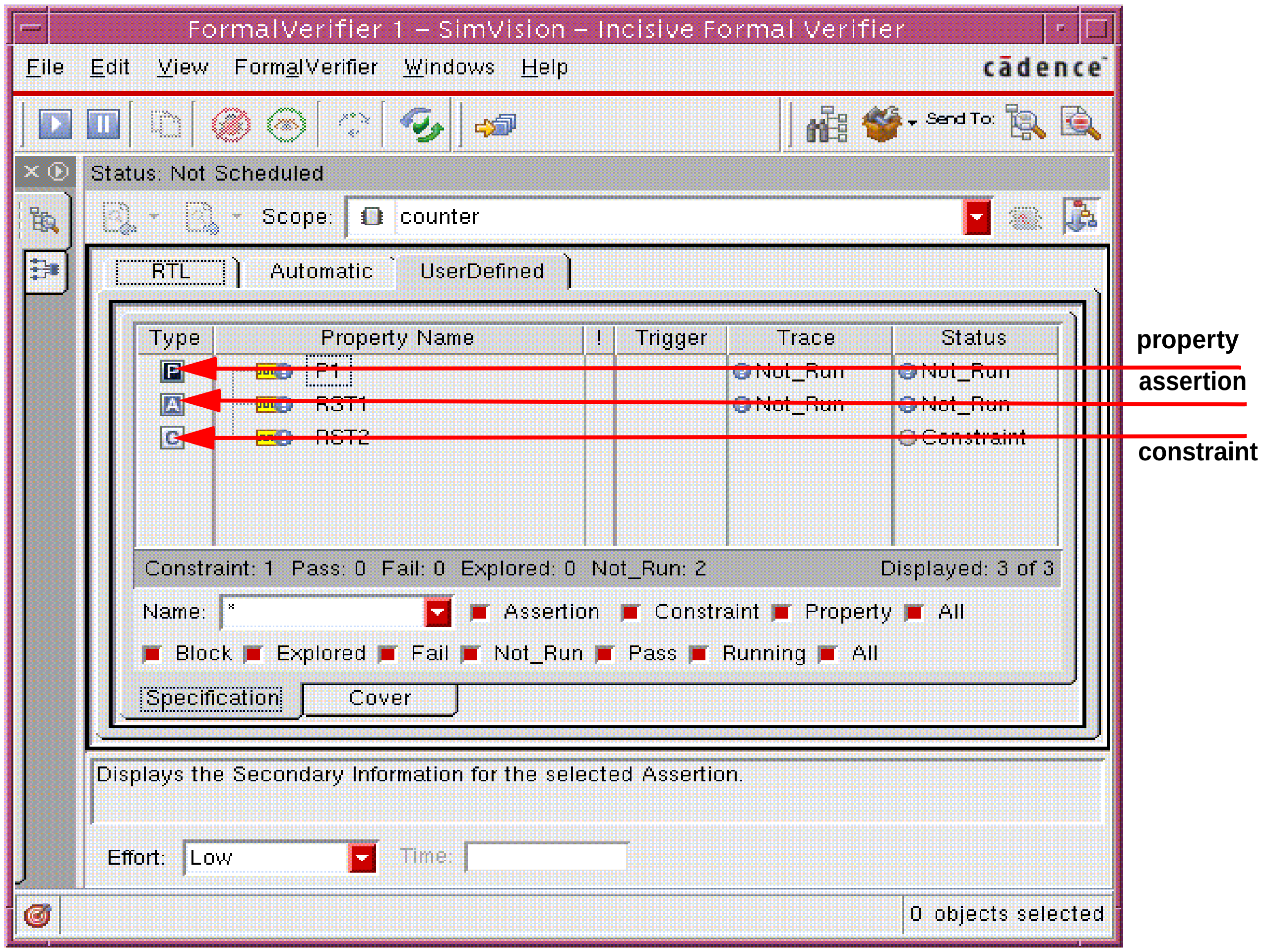
```
assign top.dest1 = top.I1.src1
assign top.dest2 = top.I1.src2.
```

Restriction

In example 4 above, if the dummy signal `dummy_event` is not mentioned in the sensitivity list of the `always` block, then, for dynamic witness/counter-example, the tool hangs as `always` block does not have a sensitivity list.

Manipulating Properties

The following figure shows Formal Verifier's UserDefined property window. Each row in the window represents one property. In Formal Verifier, properties can be in one of three states:



Status: Not Scheduled

Scope: counter

Type	Property Name	!	Trigger	Trace	Status
P	P1			Not_Run	Not_Run
A	RST1			Not_Run	Not_Run
C	RST2				Constraint

Constraint: 1 Pass: 0 Fail: 0 Explored: 0 Not_Run: 2 Displayed: 3 of 3

Name: * 🟥 Assertion 🟥 Constraint 🟥 Property 🟥 All

🟥 Block 🟥 Explored 🟥 Fail 🟥 Not_Run 🟥 Pass 🟥 Running 🟥 All

Displays the Secondary Information for the selected Assertion.

Effort: Low Time:

0 objects selected

- **P** stands for property

  A property in this state is not verified nor is it used as a *constraint*. Properties are essentially ignored by the tool until they are converted to *constraints* or *assertions*.

- **A** stands for assertion

  A property in this state is verified when you issue the `prove` command. See Chapter 8, "Running Verification and Understanding Results," for more information on the `prove` command.

- **C** stands for constraint

A property in this state is used as a `constraint` when verifying properties in state **A**.

PSL properties written with the `assert` or `cover` directive will initialize to the **A** state. PSL properties written with the `assume` directive will initialize to the **C** state. OVL properties and PSL properties not written with a verification directive are initialized to the **P** state.

- You can move a property back and forth between state **P** and **A** using the `assertion` command.
  - To move a property from state **P** to state **A** use the following command:

    ```
    FormalVerifier> assertion -add <property_name>
    ```

  - To move a property from state **A** to state **P** use the following command:

    ```
    FormalVerifier> assertion -delete <property_name>
    ```

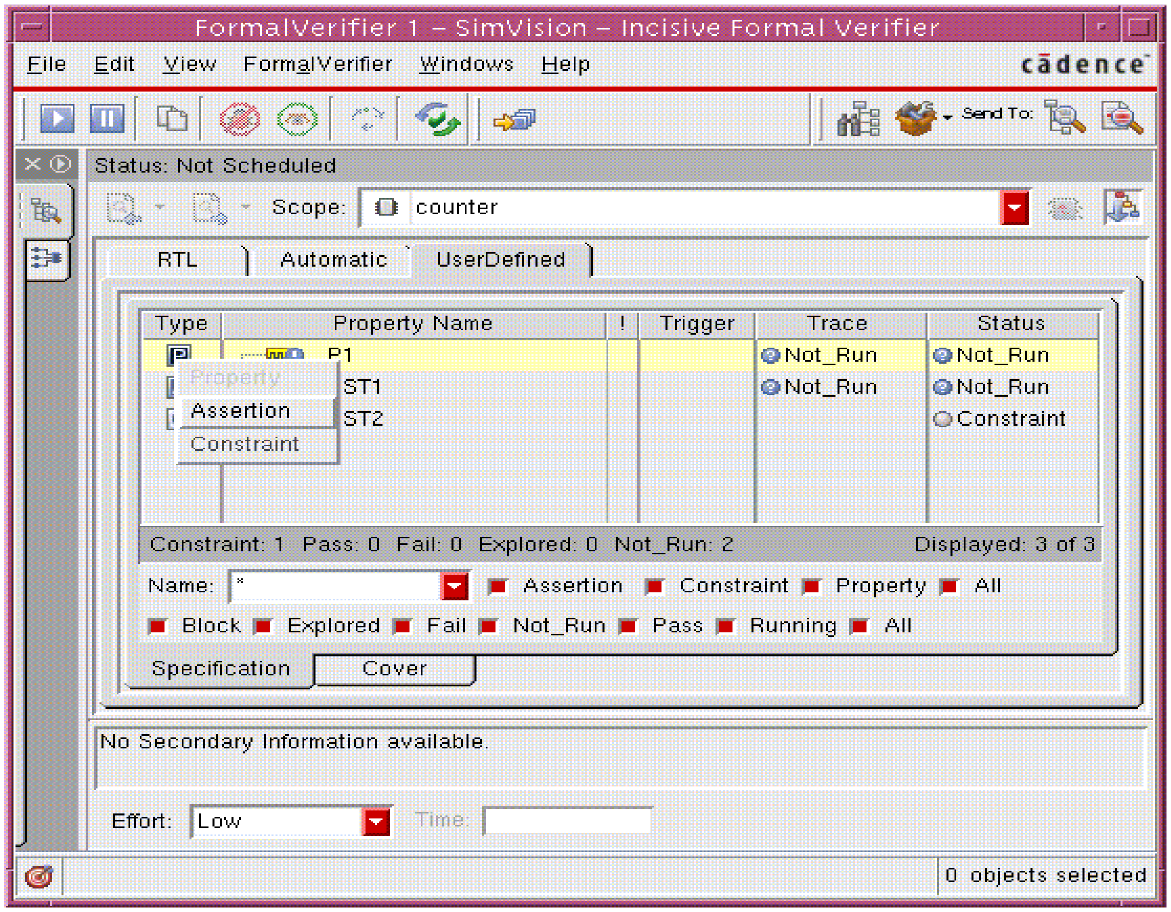
- You can move a property back and forth between state **P** and **C** using the constraint command.
  - To move a property from state **P** to state **C** use the following command:

    ```
    FormalVerifier> constraint -add <property_name>
    ```

  - To move a property from state **C** to state **P** use the following command:

    ```
    FormalVerifier> constraint -delete <property_name>
    ```

You can also move one or more properties between states using the GUI. Select the desired properties, right-click the *Type* column, and select one of the three options - Property, Assertion, or Constraint, as shown below:



Status: Not Scheduled

Scope: counter

Type	Property Name	!	Trigger	Trace	Status
P	P1			Not_Run	Not_Run
	ST1			Not_Run	Not_Run
	ST2				Constraint

Constraint: 1 Pass: 0 Fail: 0 Explored: 0 Not_Run: 2 Displayed: 3 of 3

Name: * 🟥 Assertion 🟥 Constraint 🟥 Property 🟥 All

🟥 Block 🟥 Explored 🟥 Fail 🟥 Not_Run 🟥 Pass 🟥 Running 🟥 All

No Secondary Information available.

Effort: Low Time:

0 objects selected

Using Wildcards

The `assertion` and `constraint` commands can be used with wildcards. The use of consistent naming conventions enables wildcards to be used effective. Refer to section titled Wildcarding on page 28 for more information on using wildcards with Tcl commands such as `assertion` and `constraint`.

Constraint Minimization

You can enable constraint minimization at `prove`, using the following command:

```
FormalVerifier> define minimize_constraints static
```

By default, this option is set to `off`.

When option `minimize_constraints` is set to `static`, the tool reduces the set of constraints required to prove an assertion by static structural analysis. The purpose of this constraint minimization is to improve the performance of the tool. In this case, the constraints that have only clock or reset nets common with the existing COI are not selected. Due to this, constraints in some cases get dropped, which results in better performance.

To view the minimized constraints, you can use the `assert -show -dependent` command.

Limitations

You can get different results in case of the following:

- If there is a psl constraint on clock or reset net.
- If there is a constraint that indirectly controls clock or reset.

  Example:

  ```
  always @(posedge clk)
    count = count + 1'b1;
  // psl C1 : assume (count != 3) @(posedge clk);
  ```

  In this example, once the count reaches 2, the clock stops because of this constraint. The clock cannot tick after two cycles.
- When constraint causes design level vacuity and has only clock as common net.

Incremental Analysis

Incremental analysis is an approach in which:

- You start by forcing many of the interfaces of the design under test to be inactive.

  This allows you to start doing formal analysis even before the RTL is complete.
- You write properties about the remaining active interfaces.

  This allows you to reduce the number and complexity of your properties in the beginning, thereby allowing you to develop and debug your code incrementally.
- You gradually allow the interfaces to become active.

  This allows you to gradually add to your existing set of properties and gain more confidence in the quality of the design.

Specifying Pin Constraints

Pin constraints are the key mechanism that enable incremental analysis. Pin constraints are added using the Tcl command `constraint -add -pin`. The examples given below indicate how to pin constraint a single pin and a bus in Verilog and VHDL.

To pin constraint a single pin and a bus in Verilog, refer to the following examples:

```
FormalVerifier> constraint -add -pin mode 1'b1
FormalVerifier> constraint -add -pin config 4'b0010
FormalVerifier> constraint -add -pin data 8'bx000000x
```

To pin constraint a single pin and a bus in VHDL, refer to the following examples:

```
FormalVerifier> constraint -add -pin mode b"1"
FormalVerifier> constraint -add -pin config b"0010"
FormalVerifier> constraint -add -pin data b"x000000x"
```

The use of x's in the values indicates that those bit positions are not constrained.

Similarly, pin constraints can be deleted using the Tcl command `constraint -delete -pin`.

```
FormalVerifier> constraint -delete -pin mode 1'b1
FormalVerifier> constraint -delete -pin mode b"1"
```

Both these commands support all value specification formats in Verilog and VHDL. For example, `4'b101x`, `4'o13` in Verilog and `x"c"`, `o"13"` in VHDL.

Specifying pin constraints becomes a tedious task if the width of the vector is very large and you need to put constraints only on some specific bits. To ease out this problem, you can use part selection. Part selection allows you to put constraints on both single bit as well as vector range. The part selection feature can be used with command `constraint -add -pin`.

Consider the following example in Verilog where part selection is used:

```
FormalVerifier> constraint -add -pin a[1:3] = 3'b101
```

In the above example, the lower and the upper index of the vector range can also be separated by '`..`'.

```
FormalVerifier> constraint -add -pin a[1..3] = 3'b101
```

The same example in VHDL can be indicated as:

```
FormalVerifier> constraint -add -pin a[1:3] = b"101"
FormalVerifier> constraint -add -pin a[1..3] = b"101"
FormalVerifier> constraint -add -pin a[1 to 3] = b"101"
FormalVerifier> constraint -add -pin a[3 downto 1] = b"101"
```

As shown in the above examples, if '`to`' separator is used, then the left index should be less than or equal to the right index in the vector range. If '`downto`' separator is used, then the left index should be greater than or equal to the right index. The separators '`to`' and '`downto`' are not allowed in the Verilog design.

The width of the pin and the width of the value mentioned should match. The correct example in Verilog and VHDL is indicated as:

```
FormalVerifier> constraint -add -pin a[2:1] = 2'b00
FormalVerifier> constraint -add -pin a[2:1] = b"00"
```

If the width of the value is not specified then the IFV will take the width of the value to be same as that of the pin. If there is a mismatch between the specified width of the value and the actual bits required to represent that value then either truncation will happen or zero's will be appended to the pin. For example:

```
FormalVerifier> constraint -add -pin a[2:1] = 'h0
FormalVerifier> constraint -add -pin a[2:1] = 2'h0
```

In both the above cases, pin `a` will be constrained to `00`.

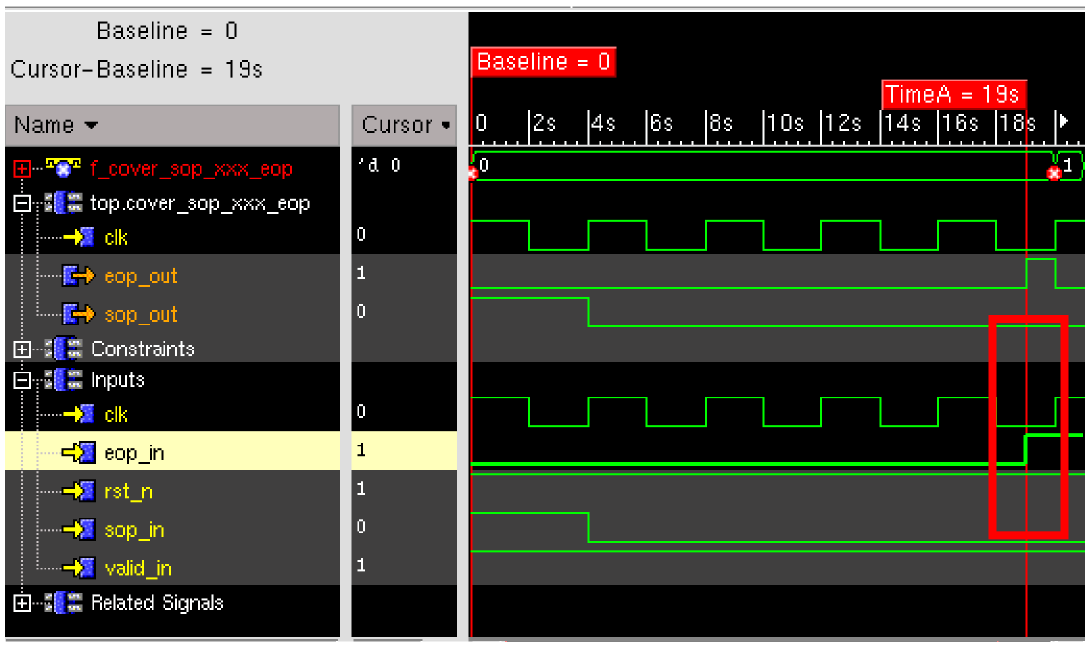
Note: When defining a part selection for pin constraints, consider the following limitations:

- There should be no space between `a` and `[` in the vector range.
- Pin constraints are not supported on multi-dimensional vectors.
- The direction of range in pin constraint should be same as the direction specified in the design.

- Enclose indices in square brackets instead of parenthesis when defining the range in the VHDL design.
- In order to change the value of the pin constraint, first delete the already added pin constraint and then add a new constraint.
- Wildcard characters are not allowed with the part selection.

Specifying the Change on Edge Only Stability Constraints

In the counter-example and/or the witness for an assertion, the value of the primary inputs may not always change on the clock synchronous edge of the design/input. For example, if the design is triggered on the positive-edge of the system clock, you would expect to see the primary inputs changing only on the positive edge instead of at negative edge or at clock level. However, the waveform appears as shown below:



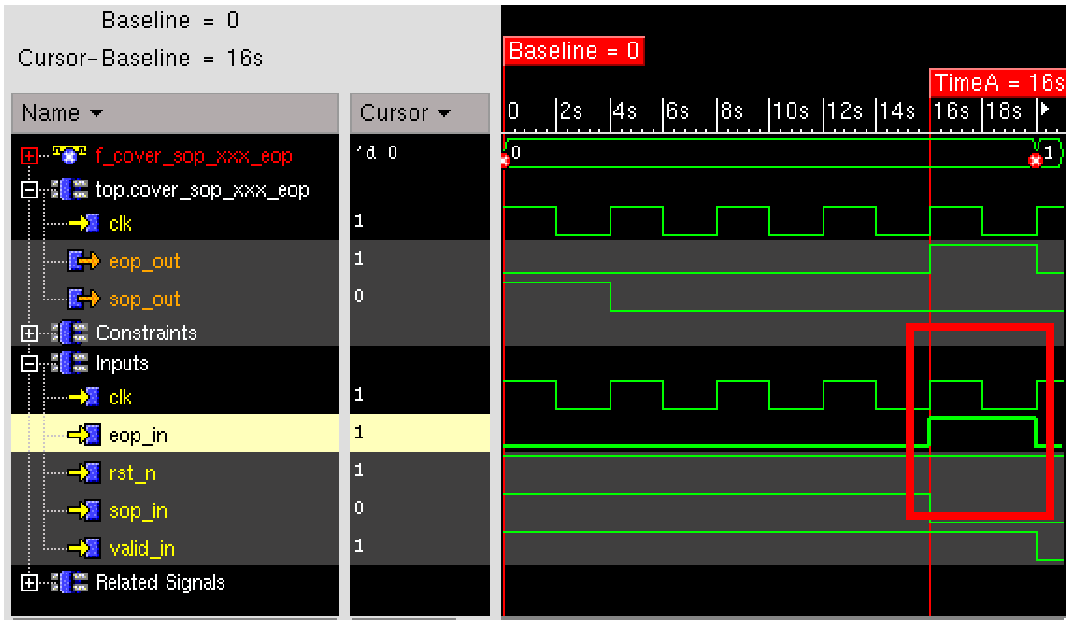
To generate a waveform smoothed on the clock edge, you can add a change on edge only stability constraint on the desired signal with reference to the clock edge. You can use the `constraint -add -change` command to add the constraint.

The change on edge only stability constraint can be applied to primary inputs, undriven nets, and outputs of blackboxed modules. Some examples follow:

- Consider the following command:

```
FormalVerifier> constraint -add -change eop_in -clock clk -edge posedge
```

The above command ensures that value of the input `eop_in` changes only at posedge of signal `clk`. The signal will remain stable at all times and change only on posedge of `clk`. Consider the modified waveform as shown below:



Note: You can add edge stability constraint on scalar, bit or part-select of one-dimensional signal. The clock option specified in the edge stability command can be a scalar variable or a bit-select of one-dimensional clock.

For example:

```
FormalVerifier> constraint -add -change eop_in[1:3]-clock clk[1] -edge posedge
```

Note: Edge stability constraints are not supported on multi-dimensional signals.

- You can specify a change on edge only stability constraint even when the clock and signal names are in different hierarchies. Consider the following example:

```
module Top(input in1, input clk1);
  wire out_d;
  SubModule m1(.in1(in1),.clk2(clk1),.out_d(out_d));
endmodule

module SubModule (input in1, input clk2, output out_d);
...
endmodule
```

The module `Top` has an instance `m1` of module `SubModule`. You can apply the change on edge only stability constraint on `in1`, by using the following command:

```
FormalVerifier> constraint -add -change in1 -clock Top.m1.clk2 -edge posedge
```

The above command ensures that the waveform of the generated counter-example or witness of `in1` remains stable at all times and changes only on the posedge of `Top.m1.clk2`.

- You can specify a change on edge only stability constraint on blackboxed outputs and undriven nets. Consider the following example:

  ```
  module Top (input in1, input clk1);
    wire out_d;
    Bot m1(.in1(in1),.clk2(clk1),.out_d(out_d));
  endmodule

  module Bot (input in1, input clk2, output out_d);
  ...
  endmodule
  ```

  The module `Top` has an instance `m1` of module `Bot`. You can apply the change on edge only stability constraint on the output `out_d` of instance `m1`, by using the following command:

  ```
  FormalVerifier> constraint -add -change m1.out_d -clock Top.m1.clk2
  -edge posedge
  ```

  The above command ensures that the waveform of the generated counter-example or witness of `m1.out_d` remains stable at all times and changes only on the posedge of `Top.m1.clk2`.

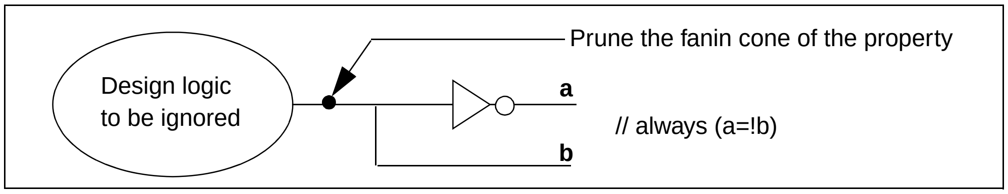
  Note: This is possible only when `out_d` is undriven or module `Submodule` is blackboxed.

-
  ```
  FormalVerifier> constraint -add -change * -clock clk -edge negedge
  ```

  You can use wildcards to add the constraint on multiple signals, as shown by the above example.

Creating Pseudo Inputs

You can create a primary input on a specified net while verifying the design. You may want to create these pseudo primary inputs to ignore some part of the design during verification. You can use the `cutpoint` command to manage the *cutpoints* in the design.


Prune the fanin cone of the property
Design logic to be ignored
a
b
// always (a=!b)

If the design is complex and you want to ignore a part of it, do the following:

1. Break the net by using the `cutpoint -add` command.
2. Add pin constraints on the broken net by using the `constraint -add` command.

Caution

Using the cutpoint command can lead to false negatives or introduce other problems. Therefore, the command needs to be used with great care. It is not intended for regular users, but is recommended for advanced users only.

Structured Approach

Cadence recommends that you take a structured approach to property development:

1. Develop protocol properties for the interfaces.

   Correct interface properties ensure that the design is well-constrained and that the design's interfaces "*speak the right language*". The set of properties for the interfaces will likely contain both *assertions* and *constraints*.

2. Develop whitebox assertions based on implementation knowledge.

   Areas that seem complex to the designer of the block often have bugs. Focus your initial effort on these complex areas to flush out bugs quickly. Whitebox assertions also simplify debugging, because they are usually in close proximity to incorrect RTL. If you did not design the block or do not have implementation knowledge, you may not be able to write whitebox assertions. If this is the case, skip to the next step.

3. Develop functional assertions based on desired behavior of the design.

These assertions often tie together large parts of the overall design and can detect bugs that could not be found with any other type of assertion. These assertions depend on correct interface behavior so they must be written after the interface protocol assertions have been verified.

The above description was necessarily brief. The methodology paper, *Developing Properties*, discussed in the section titled Other References on page 13 of this guide, contains more information on the subject of property development.

VHDL Array of Records

The support for VHDL Array of Records has now been enabled in IFV. A new option, `+vhdl_nxg`, has been added to `ifv` command syntax and `-VHDL_NXG` to `formalbuild` command syntax to enable this support. However, you need to consider the following limitations:

- In case of automatic formal assertions:
  - Xcheck names deviate from the standard naming convention for records, negative ranges, interfaces, and functions.
- In the counter-example:
  - The indices of array are shown as integer values when defined as enums.
  - The value of a vector or a record is not always shown at the top level.
  - The value of a vector in some cases is shown as separated by a comma.
- `init -fsdb` and `init -vcd` options are not supported.
- `debug -tclexport` and `debug -hdlexport` options are not supported.
- Structural checks (halstruct) in HAL are not supported.
- In some cases, the output of command `init -show`, for a VHDL design, displays a Verilog literal instead of a VHDL string.
- If a clock is defined as an array part or a field select of the record, a default clock constraint is not added. The workaround here would be to explicitly add clock constraint on such signals using `clock -add` command.
- The command, `assertion -show -all -verbose -list`, displays all fields of the record, even if only one field was used to prove an assertion.
- If an array index is of enum type then `debug -dynamic` is not supported by the tool. For example:

```
type myEnum is (RED,BLUE,GREEN);
type artype is array(GREEN down to BLUE) of std_logic;
signal myarr: artype;
```

In this example, signal `myarr` is an array whose index is of enum type.

8

Running Verification and Understanding Results

The first part of this chapter discusses how to invoke the proof process on a set of assertions and how to control the proof process. After the proof process has completed, each assertion has an associated result.

The second part of this chapter discusses the possible results and how to interpret them.

It is possible for an assertion to have a *Pass* result due to user error, rather than truly indicating that the design is bug free relative to the assertion. Given this, you need to know how to validate the results. The third and final part of this chapter discusses the need for validation and techniques you can use to validate the results.

Using the Prove Command

The `prove` command allows you to verify one or more `assertions`, under a set of `constraints` and after appropriate initialization of state elements in the design.

The `prove` command can be executed in the following two modes:

- Blocking mode

  In this mode, the Tcl prompt is not available until verification is complete and the `prove` command terminates. This is the default mode.

- Non-blocking mode

  In this mode, the Tcl prompt is available while verification runs in the background. This is particularly useful to enable you to debug a failed assertion while other assertions are being verified.

The syntax of the prove command is:

```
    FormalVerifier> prove -kill | -nowait | -wait
```

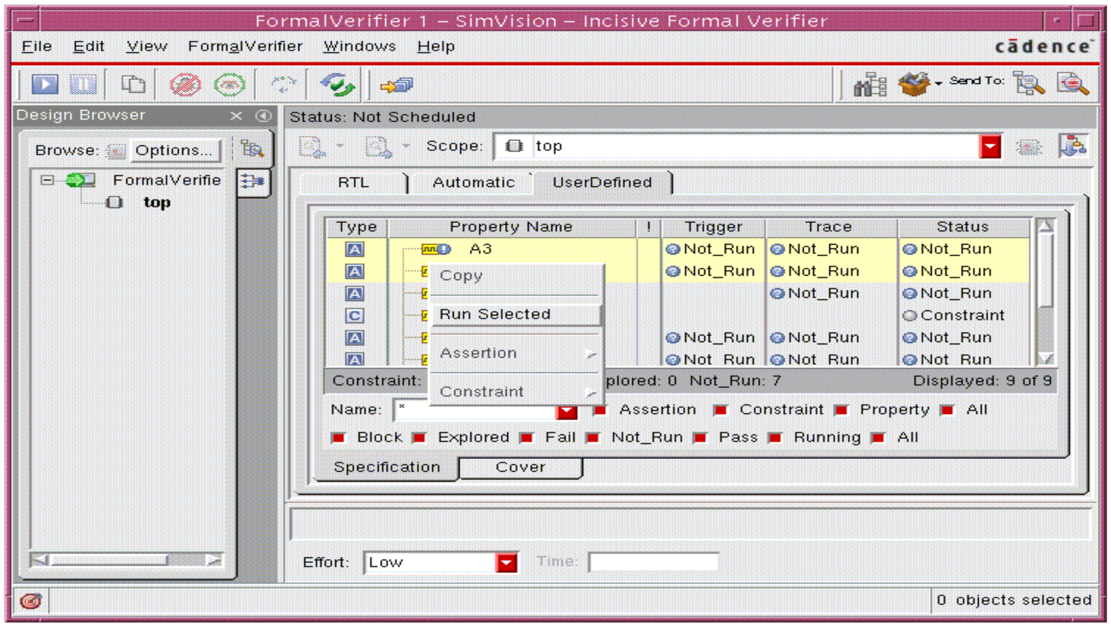
The options are described in the following table:

Modifier	**Description**
`-kill`	Terminates a prove command running in non-blocking mode.
`-wait`	Executes the prove command in blocking mode.
`-nowait`	Executes the prove command in non-blocking mode.

You can also execute the prove command from the GUI by clicking the *Run* (▶) button.

You can terminate a running prove command from the GUI by clicking the *Terminate* (⏸) button. This is equivalent to `prove -kill` command.

You can also execute the prove command on the selected set of assertions from the GUI. Select the desired assertions, right-click, and select *Run Selected*, as shown below:



Status: Not Scheduled

Scope: top

Type	Property Name	!	Trigger	Trace	Status
A	A3		Not_Run	Not_Run	Not_Run
A			Not_Run	Not_Run	Not_Run
A				Not_Run	Not_Run
C					Constraint
A			Not_Run	Not_Run	Not_Run
A			Not_Run	Not_Run	Not_Run

Constraint: … plored: 0 Not_Run: 7 Displayed: 9 of 9

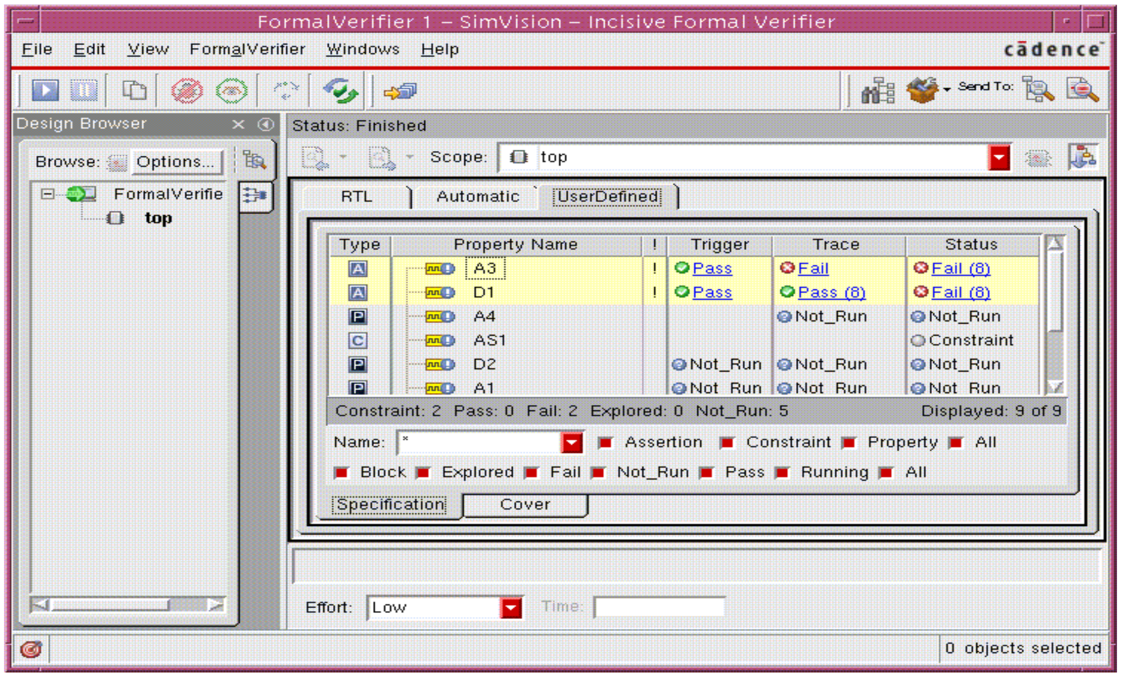
Name: * ; Assertion ; Constraint ; Property ; All

Block ; Explored ; Fail ; Not_Run ; Pass ; Running ; All

Effort: Low Time:

Note: `-nowait` is the default mode, if you use *Run* button to execute the `prove` command or select *Run Selected* in GUI.

The results for the selected assertions will be displayed as:



Status: Finished

Scope: top

Type	Property Name	!	Trigger	Trace	Status
A	A3	!	Pass	Fail	Fail (8)
A	D1	!	Pass	Pass (8)	Fail (8)
P	A4			Not_Run	Not_Run
C	AS1				Constraint
P	D2		Not_Run	Not_Run	Not_Run
P	A1		Not_Run	Not_Run	Not_Run

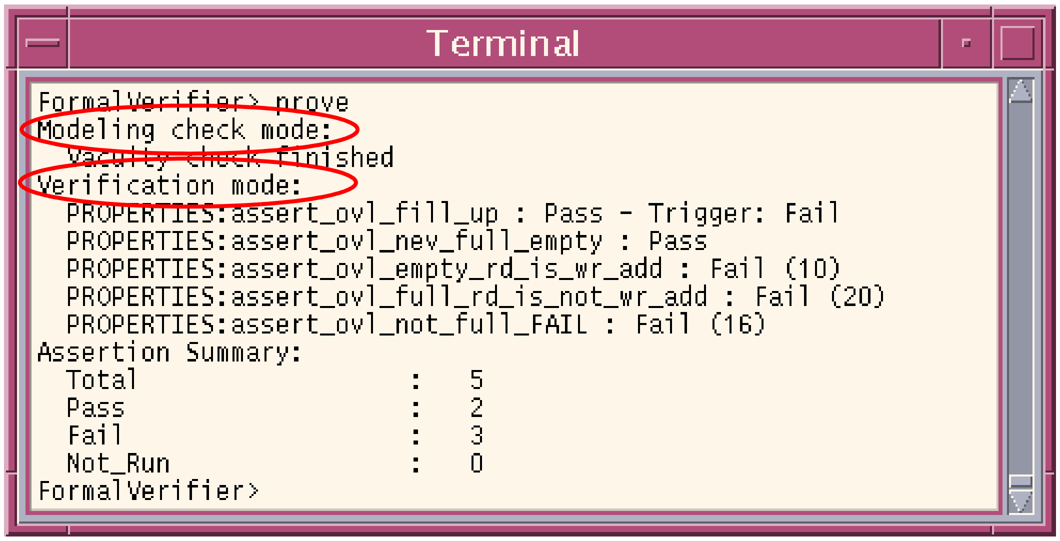
Constraint: 2 Pass: 0 Fail: 2 Explored: 0 Not_Run: 5 Displayed: 9 of 9

Name: * Assertion Constraint Property All

Block Explored Fail Not_Run Pass Running All

Note: You can switch from blocking mode to non-blocking mode by pressing *Ctrl-C* in the *Console* window. This returns you to the Tcl command prompt.

The *prove* command runs in two phases, consisting of *Modeling Check Mode* and *Verification Mode* as shown below:



```
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Verification mode:
  PROPERTIES:assert_ovl_fill_up : Pass - Trigger: Fail
  PROPERTIES:assert_ovl_nev_full_empty : Pass
  PROPERTIES:assert_ovl_empty_rd_is_wr_add : Fail (10)
  PROPERTIES:assert_ovl_full_rd_is_not_wr_add : Fail (20)
  PROPERTIES:assert_ovl_not_full_FAIL : Fail (16)
Assertion Summary:
  Total                     :    5
  Pass                      :    2
  Fail                      :    3
  Not_Run                   :    0
FormalVerifier>
```

You can also enable clock optimization mode by issuing the following command:

```
FormalVerifier> define stop_verification clock_not_opt
```

The clock optimization mode starts immediately after modeling check mode in the `prove` command

These three phases are described in the following sections.

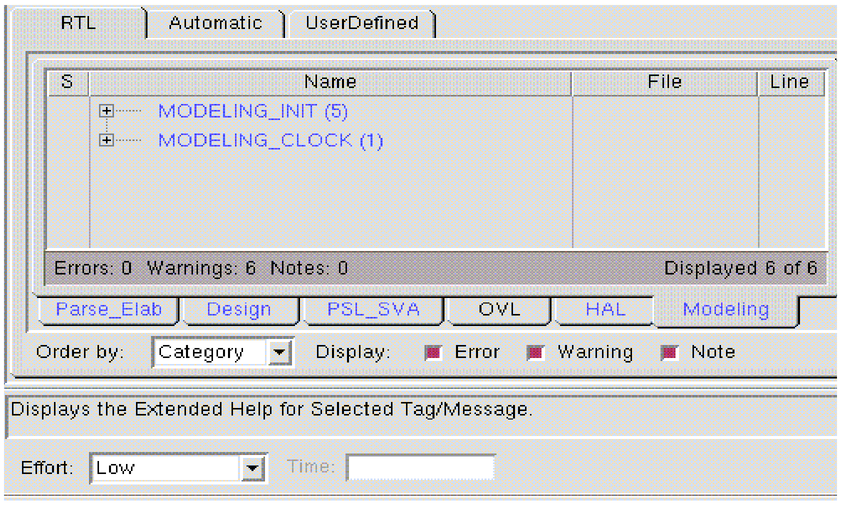
Modeling Check Mode

Modeling checks are automatically done by Formal Verifier to ensure that the design and properties are set up properly. In the modeling check mode, Formal Verifier performs the following checks:

- Clock Modeling Checks
- Structural Checks
- Initialization Checks
- Vacuity Check

The results of the modeling checks may be influenced by TCL commands, such as `clock`, `init`, and `constraint`. For example, if you have used the `constraint -add` command (either indirectly through the GUI, via the *FormalVerifier* command prompt, or via a tcl script), the vacuity violations reported may differ from the violations reported without using the command.

In the GUI, the results are displayed in the *Modeling* sub-tab of the *RTL Checks* tab of the Formal Verifier window as shown in the figure below:



MODELING_INIT (5)
MODELING_CLOCK (1)

Errors: 0 Warnings: 6 Notes: 0 Displayed 6 of 6

Order by: Category Display: Error Warning Note

Displays the Extended Help for Selected Tag/Message.

Effort: Low Time:

For details of how to interpret the results and the features of the Formal Verifier GUI, refer to the section titled Viewing RTL Checks in the GUI on page 71 of this guide.

You can also use the following command to view the modeling check violations:

```
FormalVerifier> check -show modeling
```

For more information about the above command, see the `check` command documentation in the *Formal Verifier Reference Manual*.

Clock Modeling Checks

In a design with ill-defined clocks, Formal Verifier performs the following clock modeling checks:

- Undefined clock pins

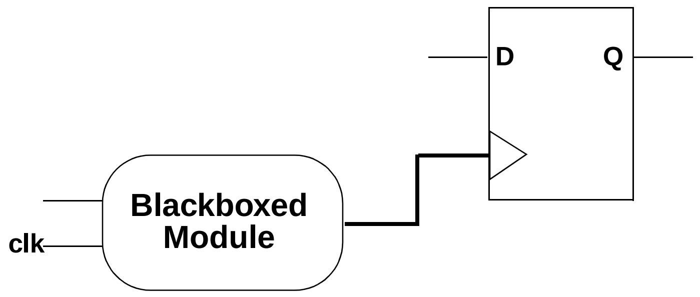
  An undefined clock pin warning indicates that the design has flip-flops whose clock nets do not have any clock behavior specified for them. Without a clock specification, Formal Verifier treats these nets like any other unconstrained signal. Cadence recommends that you assign a clock specification to all clock inputs using the `clock` command. For more information about clock specification, see the section titled Setting up Clocks on page 45 of this guide and the `clock` command documentation in the *Formal Verifier Reference Manual*.

- Gated clock pins

  A gated clock pin message indicates that the design has one or more flip-flops that are clocked by a clock waveform that cannot be fully determined. This message is informative only and does not require you to take any corrective action.

- Clock pins being driven by nets which are not clock nets.

  This check indicates that the design includes clock inputs for a set of flip-flops or latches that are not driven by any clock pin in the fanin cone. Refer to the following figure:



You can use the following command to view the clock modeling check violations:

```
FormalVerifier> check -show modeling_clock
```

Structural Checks

This category of rules check for potential structural errors in the design, such as the following:

- Input to the flip-flops/latches is always high-impedance (`Z`)
- Set/Reset port of flip-flops/latches is always enabled
- Clock port of the flip-flops/latches is tied, disabled, or enabled
- Data port of the flip-flops/latches is tied

These violations might also be reported when pin constraints are not specified properly. For example, `reset` is constrained to always be *active* though it should be constrained to always be *inactive*.

You can use the following command to view the structural modeling check violations:

```
FormalVerifier> check -show modeling_struct
```

More information on structural checks is available in the Formal Verifier Reference manual.

Initialization Checks

In order to properly initialize a design and perform a valid verification, Formal Verifier uses the following rule to check the conditions in your design:

- Uninitialized flip-flops/latches in the design

  This check reports those state elements that are not initialized after the initialization phase. This is not necessarily an error, since most designs contain some state elements that are not intended to be initialized. After initialization, the proof process considers both possible initial states for those state elements that are uninitialized. This ensures that no matter what value a state-element actually takes, there can be no assertion violations.

- Uninitialized FSMs in the design

  These checks report those FSM state elements that are not initialized or initialized to an invalid state. Initialization to an invalid state is probably more serious as it might indicate issues in initializing the design. On the other hand, an uninitialized FSM is not necessarily an error, because some designs contain state elements that are not intended to be initialized. However, FSM checks will not be run in either case.

You can use the following command to view the initialization modeling check violations:

```
FormalVerifier> check -show modeling_init
```

Vacuity Check

This check detects a set of constraints that are mutually conflicting or conflicting with the design and hence cannot be satisfied simultaneously. Inconsistency among constraints and between design and constraints can lead to incorrect verification results.

The following examples illustrate different scenarios that can lead to Vacuity Check failure:

- Example 1: A pair of conflicting constraints

  ```
  // psl property P1 = always (scan == 0);
  // psl property P2 = always (scan == 1);
  ```

  Here, both *constraints* are obviously contradictory.

- Example 2: A set of three mutually conflicting constraints

  ```
  // psl property P1 = always (enable ==1 );
  // psl property P2 = always (enable -> data);
  // psl property P3 = always (enable -> !data);
  ```

  In this case, constraint `P1` forces enable to be equal to `1`. Because enable is equal to `1`, constraint `P2` would imply that data is equal to `0`. However, `P3` would imply that data is equal `1`. Thus, the contradiction.

- Example 3: Vacuity failure because of a single constraint

  ```
  // psl property P1 = always {in;!in};
  ```

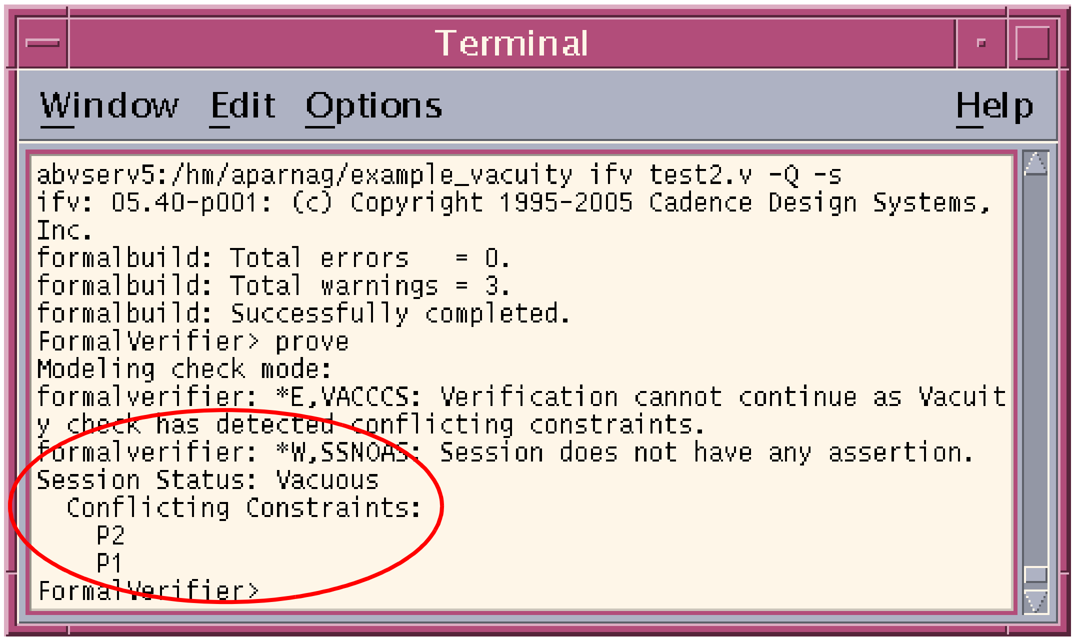
  Because the SERE `{in;!in}` applies at all cranks, it means that `in` and `!in` conflict on every crank except for the first.

- Example 4: Conflict between design and constraints

  ```
  module top (a,b,out1,out2,clk);
  output out1,out2;
  input a,b;
  input clk;

  // psl property P1 : assume always (!out1);
  reg out1,out2:

  always @ (posedge clk)
  begin
      out1 = 1’b1;
  end

  endmodule
  ```

  In this case, as per the design, the value of `out1` will always be `1`. However, constraint `P1` implies that the value of `out1` will always be `0`. Thus, a conflict between design and constraints.

When conflicting constraints are detected, Formal Verifier aborts verification and reports the session status as *Vacuous* and lists the minimum set of constraints that caused vacuity. An example is shown below:



```
abvserv5:/hm/aparnag/example_vacuity ifv test2.v -Q -s
ifv: 05.40-p001: (c) Copyright 1995-2005 Cadence Design Systems,
Inc.
formalbuild: Total errors   = 0.
formalbuild: Total warnings = 3.
formalbuild: Successfully completed.
FormalVerifier> prove
Modeling check mode:
formalverifier: *E,VACCCS: Verification cannot continue as Vacuit
y check has detected conflicting constraints.
formalverifier: *W,SSNOAS: Session does not have any assertion.
Session Status: Vacuous
  Conflicting Constraints:
    P2
    P1
FormalVerifier>
```

In case of a vacuity failure, verification cannot be run until you resolve the conflict amongst the constraints, either by redefining the constraints or by removing one or more of them from the design.

Because vacuity check takes time, you may want to disable it once you are sure that your constraints will not change. You can disable vacuity check by issuing the following command:

```
    FormalVerifier> define vacuity off
```

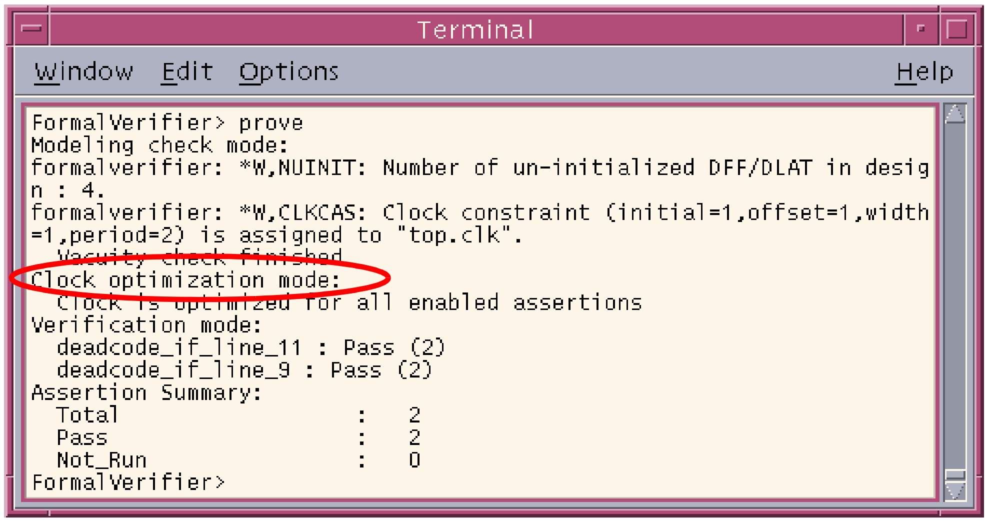
Note: Formal Verifier only detects design-independent conflict among constraints. If there is a conflict of constraints that depends on the design, Formal Verifier may not detect the conflict. Formal Verifier provides other features to assist in detecting conflicting constraints. Refer to the section titled Validating Results on page 140 of this guide, for more information.

Clock Optimization Mode

In this mode, the clock optimization check is done for all assertions in the design. You can enable clock optimization mode by issuing the following command:

```
FormalVerifier> define stop_verification clock_not_opt
```

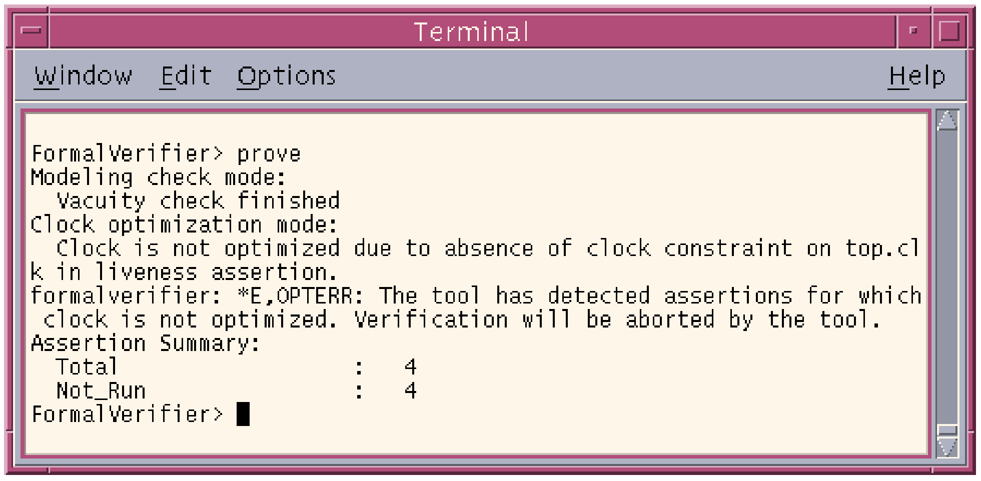
The clock optimization mode starts immediately after modeling check mode during proof, as shown below:



```
FormalVerifier> prove
Modeling check mode:
formalverifier: *W,NUINIT: Number of un-initialized DFF/DLAT in desig
n : 4.
formalverifier: *W,CLKCAS: Clock constraint (initial=1,offset=1,width
=1,period=2) is assigned to "top.clk".
  Vacuity check finished
Clock optimization mode:
  Clock is optimized for all enabled assertions
Verification mode:
  deadcode_if_line_11 : Pass (2)
  deadcode_if_line_9 : Pass (2)
Assertion Summary:
  Total                  :    2
  Pass                   :    2
  Not_Run                :    0
FormalVerifier>
```

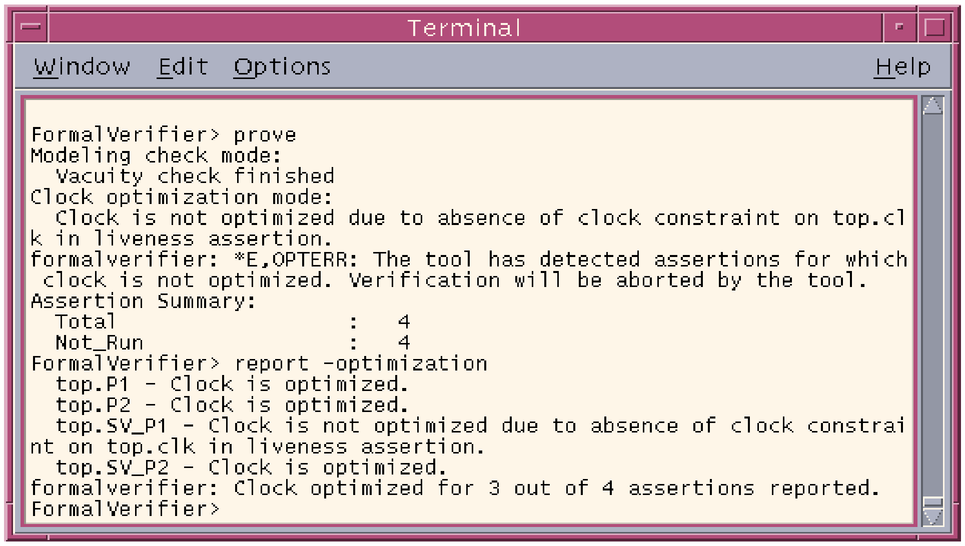
If the clock is optimized for all enabled assertions, then all assertions will be verified in the verification mode.

If the clock is not optimized then the reason for not optimizing the clock is provided, an error message is generated, and the verification is aborted by the tool. The following example displays the message when the clock is not optimized.



```
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Clock optimization mode:
  Clock is not optimized due to absence of clock constraint on top.cl
k in liveness assertion.
formalverifier: *E,OPTERR: The tool has detected assertions for which
 clock is not optimized. Verification will be aborted by the tool.
Assertion Summary:
  Total                  :    4
  Not_Run                :    4
FormalVerifier>
```

However, if you use `report -optimization` command, you will notice that the clock is optimized for some assertions as displayed below.



```
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Clock optimization mode:
  Clock is not optimized due to absence of clock constraint on top.cl
k in liveness assertion.
formalverifier: *E,OPTERR: The tool has detected assertions for which
 clock is not optimized. Verification will be aborted by the tool.
Assertion Summary:
  Total                  :   4
  Not_Run                :   4
FormalVerifier> report -optimization
  top.P1 - Clock is optimized.
  top.P2 - Clock is optimized.
  top.SV_P1 - Clock is not optimized due to absence of clock constrai
nt on top.clk in liveness assertion.
  top.SV_P2 - Clock is optimized.
formalverifier: Clock optimized for 3 out of 4 assertions reported.
FormalVerifier>
```

This is a known behavior, as in clock optimization mode, all assertions are clubbed together for clock optimization check whereas, in `report -optimization`, each assertion is independently checked for clock optimization, thus resulting in a difference in the clock optimization check.

Another case could be that clock is not optimized in clock optimization mode. However, if you use `report -optimization` command, the clock will be optimized for all assertions. In such cases, disable the clock optimization mode and run the `prove` command again.

You can disable clock optimization mode by issuing the following command:

```
FormalVerifier> define stop_verification off
```

Note: By default, the clock optimization mode is disabled.

Verification Mode

In the Verification mode, all the assertions in the design are verified. As the assertions in the design are verified, their status is displayed in the *Console* window and in the GUI.

Variables Affecting Proofs

There are several variables that affect the proof process. Although the default values for these variables are usually adequate, you may want to change one or more. The following are the most common of these variables:

- `effort`

This variable controls the amount of time that Formal Verifier spends verifying an assertion. By default, effort is set to low. If the tool is not able to prove or disprove an assertion within the specified effort, you can increase the effort by setting the effort with the GUI or by using the `define effort` command. The available values are `low`, `mid`, and `high`, which corresponds to `10` seconds, `1` minute, and `5` minutes, respectively. You can also set custom time by using the `define` command or by selecting the time from the *Effort* drop-down box in the GUI.

Note: The effort time per assertion is not honored incase of automatic formal assertions as a set of automatic formal assertions run at the same time. In this case, the entire run effort can be a maximum of, `effort time * total number of properties`.

- `halo`

  In hierarchical designs, Formal Verifier attempts to partition the hierarchy in order to prove assertions more efficiently. The partitioning technique is not always effective. For example, you may know based on the knowledge of your assertion, that in order to prove the assertion, the tool must consider the entire design up to the primary inputs. In this case, it is useless for the tool to attempt partitioning. In such cases, you may want to turn off the partitioning optimization. You can do this by setting the `halo` variable to `off`.

  Formal Verifier also supports multiple halos for verification. For more information, refer to the section titled, Specifying Multiple Halos, in this chapter.

- `engine`

  Formal Verifier has several underlying formal engines. By default, Formal Verifier uses a heuristic to decide what engines to apply to a given assertion. You can override the heuristic by setting the engine variable to one of the available engines. You can view the list of available engines by issuing the `help define` command.

  Formal Verifier also supports parallel engine verification. For more information, refer to the section titled, Distribution, in this chapter.

- `witness_check`

  `witness_check` allows you to enable the witness checks on the user defined assertions in the design. A `witness_check` helps you to generate witness for a trigger and a trace for an assertion. For more information, refer to *define witness_check* in *Chapter 1* of the *Formal Verifier Reference Manual.*

  As described in the sections, titled Automatic Trigger Checks for OVL, PSL, and SVA and Automatic Trace Checks for PSL and SVA, various trigger and trace checks are run by default. Cadence recommends running trigger and trace checks at least until you have verified that all assertions are triggerable and traceable. Because these checks take compute time, you may want to turn them off once you have verified all assertions. You can do this by setting the `witness_check` variable to `off`.

- `vacuity`

  Because vacuity check takes time, you may want to disable it once you are sure that your constraints will not change. You can disable vacuity check by setting the `vacuity` variable to `off`.

For more information on these and other variables, refer to the `define` command documented in the *Formal Verifier Reference Manual*.

Distribution

Whenever you want to verify an assertion, you specify the engine to be used for the verification. If you do not specify the engine, IFV verifies the assertion using an ‘auto’ engine. Based on the complexity of the assertion, the ‘auto’ engine decides the appropriate engine to be used for verification. It is observed that the single engine specified or selected by an auto engine may not provide the best of results for all assertions.

To overcome this problem, IFV now supports parallel engine verification, termed as distribution. In this, you will specify multiple engines. The engines will start with the verification of an assertion. As soon as the results are available for one engine, the rest of the engines will stop verification and will move on to verify the next assertion. The first engine providing the results will be considered as the best engine to verify that assertion. Distribution can apply on all kinds of assertions.

Specifying Multiple Engines

You can use the following command to specify multiple engines for verification:

```
FormalVerifier> define engine sword axe bow
```

You can specify two or more engines in a multi-CPU machine. The multiple engines specified will run on the same machine. The engines specified should be supported by IFV. As of now, IFV supports sword, axe, axe2, and bow engines in the distribution mode. Environment settings specified will be applicable for all engines. You cannot specify engine specific settings.

Note: The number of engines specified should be less than or equal to the number of CPUs on the machine. For example, if the number of engines specified are 3, then the number of CPUs on the machine should be 3 or more.

If the user specifies more engines than the number of CPUs, the following error would be generated.

*formalverifier: *E,CPUCNT: The number of engines specified, 3, exceeds the number of CPU on this machine, 2. This might affect the performance of verification. To specify more engines than the number of CPU, use the command define cpu_sharing on.*

To ignore this error and continue with multiple engine verification, use the following command:

```
FormalVerifier> define cpu_sharing on
```

This command will allow you to proceed with the verification even if the engines specified are more than the number of CPUs on the machine. However, this might effect the performance of the verification.

Specifying Multiple Halos

You can use the following command to specify multiple halos for verification:

```
FormalVerifier> define halo hier off seq
```

For example, if only one engine is specified for verification then the engine will run with `halo hier`, `off`, and `seq` parallely.

If multiple engines and halos are specified, then all combinations of engines and halos will run in parallel.

Automatic Distribution

You can set the auto distribution mode using the following command:

```
FormalVerifier> define engine auto_dist
```

This will automatically run multiple engine and halo combinations in parallel based on the number of CPUs available.

If you use the following command:

```
FormalVerifier> define auto_dist_max full
```

then the number of engine and halo runs is limited by the number of CPUs on the machine.

If you specify the value of the variable as:

```
FormalVerifier> define auto_dist_max 2/3/4
```

then the number of engine and halo runs is limited by the value of `define auto_dist_max` variable.

Secondary Information in Distribution

If a property has result status as Pass, Fail, or Explored in the distribution mode, then the respective engine/halo, which solved the property is reported. Different types of reports will be generated, based on the following:

- In case of engine based distribution, a winning engine will be reported.
- In case of halo based distribution, a winning halo will be reported.
- In case of auto_dist or a combination of multiple halos and engines, then both winning engine and halo will be reported.
- In case of a sub assertions, that is trigger and trace of an assertion, the winning engine/ halo information will be reported for each of the sub assertions also.

This information is reported as secondary information.

You can view a summary of the secondary information by using the following command:

```
FormalVerifier> assertion -show -verbose
```

For example:

```
FormalVerifier> define halo seq off
formalverifier: Setting halo to seq off.
FormalVerifier> define engine Axe sword
formalverifier: Setting engine to Axe sword.
FormalVerifier> prove -silent
FormalVerifier> assertion -show -verbose
  ASSERT_2 : Pass - Trigger: Pass (6)
    Uninitialized nets  : 8
    Winning Engine      :
      Assertion           : Axe
      Trigger             : sword
    Winning Halo        :
      Assertion           : seq
      Trigger             : seq
  ASSERT_SEQ_3_SVA : Fail (8)
    Uninitialized nets  : 8
    Winning Engine      :
      Assertion           : Axe
    Winning Halo        :
      Assertion           : off
FormalVerifier>
```

In case of OVL assertions, the names of the sub assertions are also displayed along with the winning engine/halo information.

For example:

```
check5 : Fail (13) - Trigger: Pass
    Uninitialized nets  : 8
    Winning Engine      :
        check5.ASSERT_HANDSHAKE_ACK_XZ_P: axe
        check5.ASSERT_HANDSHAKE_REQ_XZ_P: axe
        check5.ASSERT_HANDSHAKE_REQ_DEASSERT: axe
    Winning Halo        :
        check5.ASSERT_HANDSHAKE_ACK_XZ_P: seq
        check5.ASSERT_HANDSHAKE_REQ_XZ_P: seq
check5.ASSERT_HANDSHAKE_REQ_DEASSERT: off
```

Licensing for Distribution

In order to use the distribution feature of IFV, you will need additional licenses.

- If you use between two to four engines, you will require one additional license.
- If you use between five to eight engines, you will require two additional licenses.
- If you are using ABVIP along with IFV, an equivalent number of ABVIP and IFV licenses will be required.
- If you use a combination of single engine and two halos, you will require one additional license.
- If you use a combination of two engines and three halos, you will require two additional licenses.
- If you use a combination of three engines and three halos, you will require three additional licenses.

During verification, the multiple engines will run in parallel. Therefore, the machine should have enough memory to run engines in parallel.

Understanding Results

During and after the verification of an assertion, you might encounter many types of results. The possible result types are as follows:

- Pass
- Explored
- Running
- Not-Run
- Abort
- Fail
- Block
- Not-Supported

Beyond the results for the verification of a given assertion, there is other secondary information that is sometimes available. The meaning of *Pass*, *Fail*, and *Explored* for a given assertion depend on the class of the assertion. Property classes are discussed in the section titled Classes of Properties on page 88 of this guide.

The following terms are used in later sections:

- **safety assertion**

  Refer to the section titled Safety on page 88 for a discussion of safety properties.

- **liveness assertion**

  Refer to the section titled Liveness on page 89 for a discussion of liveness properties.

- **existential assertion**

  Refer to the section titled Existential on page 90 for a discussion of existential properties.

- **counter-example**

  A counter-example is a legal sequence of states and input values of a design that demonstrates a failure of an assertion. There may be many ways for an assertion to fail; however a counter-example shows one way. You can examine a counter-example by viewing the waveforms in the *Waveform Viewer* or by viewing the annotated signal values in the *Source Browser*.

- **witness**

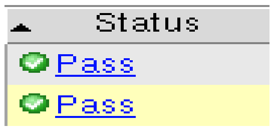
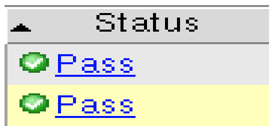
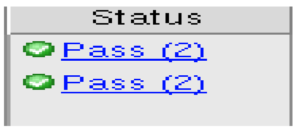
  A witness is a legal sequence of states and input values of a design that provides an example of how an existential assertion can be satisfied. The most common cases of existential assertions are trigger and trace checks (see section Automatic Trigger Checks for OVL, PSL, and SVA and Automatic Trace Checks for PSL and SVA) and deadcode, checks (see Chapter 10). Refer to the section titled Existential on page 90 for a discussion of existential properties.

- **depth**

  Depth is a number that indicates how many cranks the formal engine has processed. Depth is applicable when either a counterexample or a witness has been found. In these cases, a depth of `N` cranks means that the engine had to process `N` cranks in order to find the given counter-example or witness.

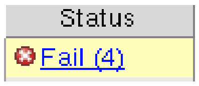
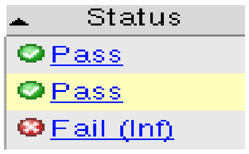
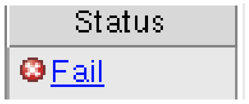
Pass

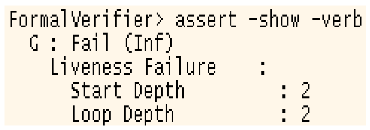
In the GUI, a pass is indicated by a green circle with the word *Pass* next to it. The status *Pass* is undelined and the color is set to blue resembling a hyperlink. You can click the *Pass* status to view the witness waveform. The table below discusses the pass result relative to each of the three main property classes.

Safety Assertion	**Liveness Assertion**	**Existential Assertion**
 Status Pass Pass	 Status Pass Pass	 Status Pass (2) Pass (2)
The *Pass* in the *Status* column indicates that the assertion is satisfied under the given set of constraints and that there is no possible sequence of states that would violate the safety assertion. No depth is reported for a *Pass*. The result of the *Trigger* column in discussed in detail in the section titled Automatic Trigger Checks for OVL, PSL, and SVA on page 141 of this guide.	*Pass* in the *Status* column indicates that the assertion is satisfied under the given set of constraints and that there *does not exist* an infinitely repeating sequence of non-satisfying states for this assertion. No depth is reported for a *Pass*.	*Pass* in the *Status* column indicates that the condition or sequence in the existential assertion is reachable under the given set of constraints. In addition, the number 2 inside the parentheses indicates that a witness was found at a depth of 2 cranks. You can view the witness waveform of a *Pass* existential assertion, by clicking the status *Pass* or clicking the *witness* button ( ).

Fail

In the GUI, a fail is indicated by a red circle with the word *Fail* next to it. The status *Fail* is undelined and the color is set to blue resembling a hyperlink. You can click the *Fail* status to view the counter-example.The table below discusses the Fail result relative to each of the three main property classes.

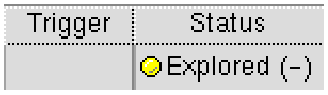
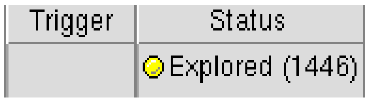
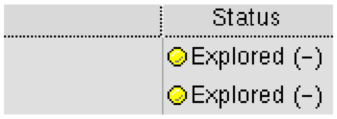
Safety Assertion	Liveness Assertion	Existential Assertion
 Status Fail (4)	 Status Pass Pass Fail (Inf)	 Status Fail
Fail in the *Status* column indicates that the assertion is not satisfied under the given set of constraints and that a counter-example was found for the assertion. The number `4` inside the parentheses indicates that a counter-example was found at a depth of `4` cranks.	*Fail* in the *Status* column indicates that the desired condition never occurs in the generated counter-example.	*Fail* in the *Status* column indicates that it is not possible to excite the condition or sequence being checked, under the given set of constraints.
You can view the counter-example waveform of a *Fail* safety assertion, by clicking the Fail status or the *debug* button ( ).	Since the desired condition can never be reached, depth is shown as *Infinity* (`Inf`). However, you can view the loop depth and loop start depth in the secondary information as:	The debug button is disabled for failed existential assertions, because there is nothing for the tool to show.

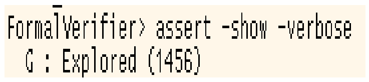
	 `FormalVerifier> assert -show -verb`
	`  G : Fail (Inf)`
	`    Liveness Failure    :`
	`      Start Depth        : 2`
	`      Loop Depth         : 2`	

	You can view the counter-example waveform of a *Fail* liveness assertion, by clicking the Fail status or the *debug* button ( ).	

Explored

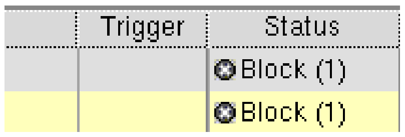
In the GUI, an explored is indicated by a yellow circle with the word *Explored* next to it. The table below discusses the Explored result relative to each of the three main property classes.

Safety Assertion	Liveness Assertion	Existential Assertion
 Trigger / Status: Explored (-)	 Trigger / Status: Explored (1446)	 Status: Explored (-), Explored (-)

Explored in the *Status* column indicates that Formal Verifier is unable to find a counter-example or explore the entire state space of the design within the specified effort or within the memory space of the machine.
 	
The depth shown inside the parentheses indicates the number of cranks till which the engine was able to check. The tool guarantees that no bug exists within this depth.
	Explored in the *Status* column indicates that Formal Verifier is unable to find a counter-example or explore the entire state space of the design within the specified effort or within the memory space of the machine.
	
	Depth is reported inside the parantheses for liveness assertions as:
	
	`FormalVerifier> assert -show -verbose`
	`  G : Explored (1456)`
	
	If the depth reported is (-), it means that the tool has not progressed to any depth as the specified effort was not sufficient to verify the liveness assertion. The tool cannot start from Explored (0) depth as there can be a failure at depth 0 also. In this case, you can increase the effort or try a different engine to proceed with the verification.
		Explored in the *Status* column indicates that Formal Verifier is unable to find a witness or explore the entire state space of the design within the specified effort or within the memory space of the machine.
		
		The depth shown inside the parentheses indicates the number of cranks till which the engine was able to check. The tool guarantees that no witness exists within this depth.

Block

In the GUI, a block is indicated by a black circle with the word *Block* next to it as shown in the following figure:



	Trigger	Status
		● Block (1)
		● Block (1)

Block indicates that the assertion could not be verified, because of one of the following reasons:

- Combinational loops exist in the design

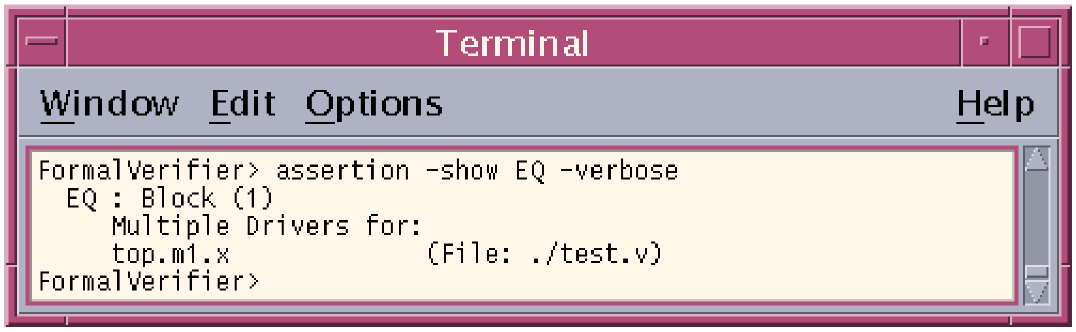
  You can identify and remove combinational loops earlier in the design cycle by using HAL checks. Refer the section titled HAL Checks on page 85 for details.

- Bus Contention errors were detected in the design.

You can use the following command to view the reason for the *Block* status:

```
FormalVerifier> assertion -show <assertionNames> -verbose
```

The output of the above command is as shown below:

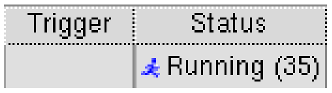


```
FormalVerifier> assertion -show EQ -verbose
  EQ : Block (1)
     Multiple Drivers for:
     top.m1.x                (File: ./test.v)
FormalVerifier>
```

For bus contention errors, you can view a counter-example that shows the bus contention by clicking the debug ( ) button.

Running

In the GUI, running is indicated by a *running* figure in blue with the word *Running* next to it as shown in the following figure:

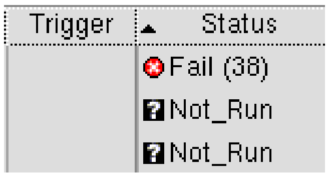


Trigger	Status
	Running (35)

Running indicates that the tool is currently verifying the specified assertion. The depth inside the parentheses indicates the number of cranks verified by the engine so far. The depth is continually updated, as verification proceeds.

Not-Run

In the GUI, a Not_Run is indicated by a question mark with the word *Not_Run* next to it as shown in the following figure:

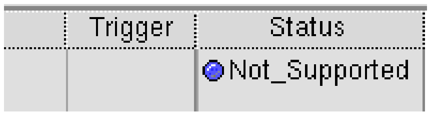


Trigger	Status
	Fail (38)
	Not_Run
	Not_Run

Not_Run indicates that verification of the specified property has not been started.

Not-Supported

In the GUI, an unsupported property is indicated by a blue circle with the word *Not_Supported* next to it as shown in the following figure:



	Trigger	Status
		Not_Supported

Not_Supported indicates that the associated property is not supported. Refer to the section titled Unsupported Properties on page 84 of this guide for more information on unsupported properties.

Abort

An *abort* indicates an unexpected termination of the verification for the associated assertion. This can happen because of one of the following reasons:

- Something abnormal with the tool.
- Something (memory or disk space issues) in the machine.

You can view the details of such problems by accessing the secondary information about the aborted assertion. For details on secondary information, refer to the next section.

Secondary Information

Formal Verifier provides additional information about the verification of some assertions. This is referred to as the secondary information.

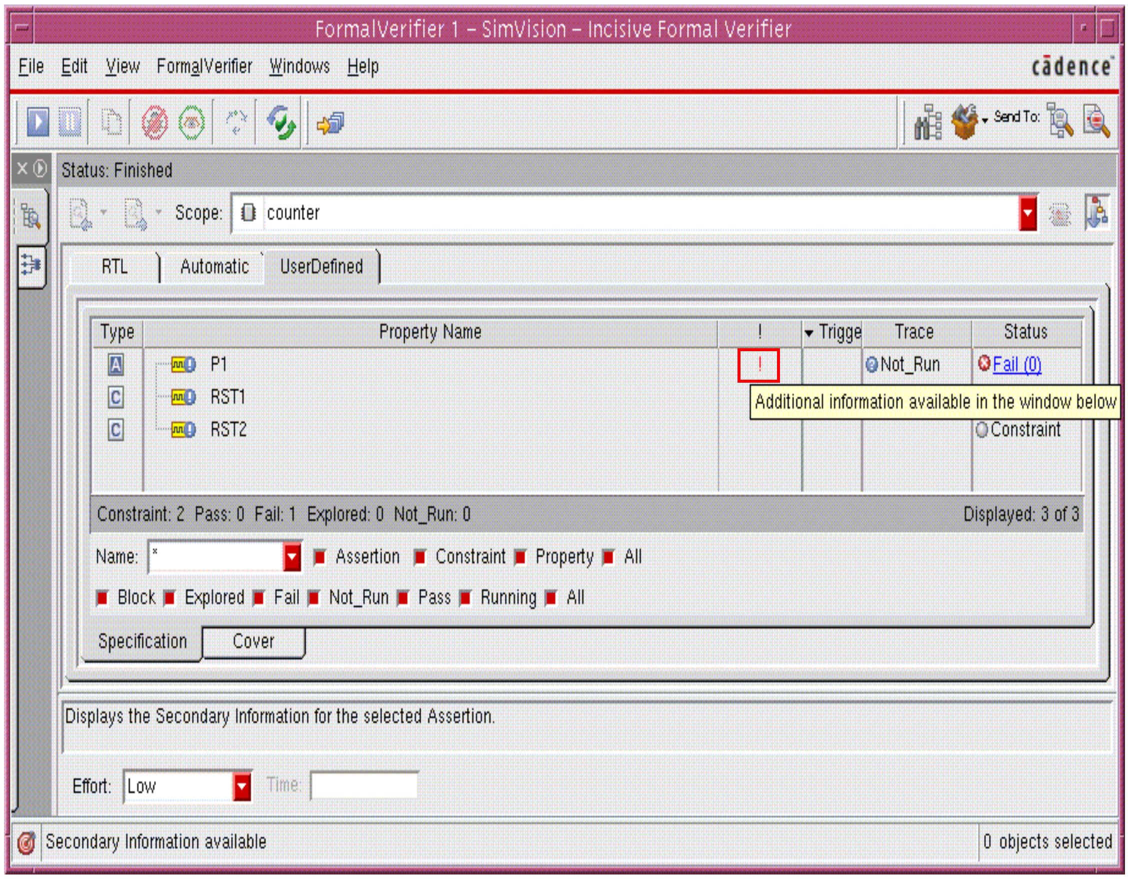
You can view a summary of the secondary information by using the following command:

```
FormalVerifier> assertion -show <assert_name> -verbose
```

You can view a detailed list of secondary information by using the following command:

```
FormalVerifier> assertion -show <assert_name> -verbose -list
```

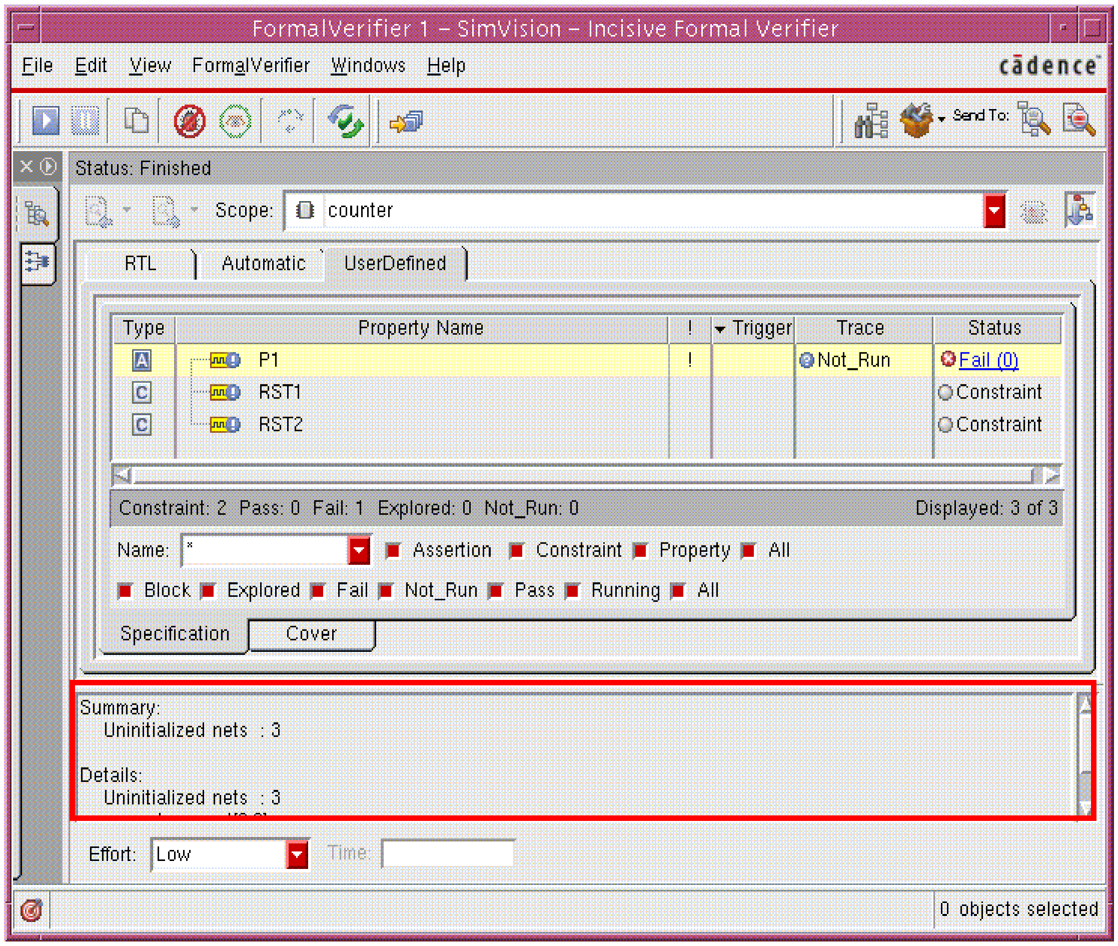
In the GUI, a column indicates if secondary information about an assertion is available. Consider the highlighted section in the figure given below:


FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cādence
Send To:
Status: Finished
Scope: counter
RTL Automatic UserDefined
Type Property Name ! Trigge Trace Status
A P1 ! Not_Run Fail (0)
C RST1
C RST2 Constraint
Additional information available in the window below
Constraint: 2 Pass: 0 Fail: 1 Explored: 0 Not_Run: 0 Displayed: 3 of 3
Name: * Assertion Constraint Property All
Block Explored Fail Not_Run Pass Running All
Specification Cover
Displays the Secondary Information for the selected Assertion.
Effort: Low Time:
Secondary Information available 0 objects selected

The exclamation mark `!` (in red) indicates that secondary information is available for the assertion.

Note: For liveness assertions, there will not be any exclamation mark `!` (in red) indicating that secondary information is available.

When you click the assertion (with the exclamation mark), the secondary information is displayed in the *Secondary Pane*, as shown in the following figure.



Type	Property Name	!	Trigger	Trace	Status
A	P1	!		Not_Run	Fail (0)
C	RST1				Constraint
C	RST2				Constraint

Constraint: 2 Pass: 0 Fail: 1 Explored: 0 Not_Run: 0 Displayed: 3 of 3

Name: * 🟥 Assertion 🟥 Constraint 🟥 Property 🟥 All

🟥 Block 🟥 Explored 🟥 Fail 🟥 Not_Run 🟥 Pass 🟥 Running 🟥 All

```
Summary:
   Uninitialized nets  : 3

Details:
   Uninitialized nets  : 3
```

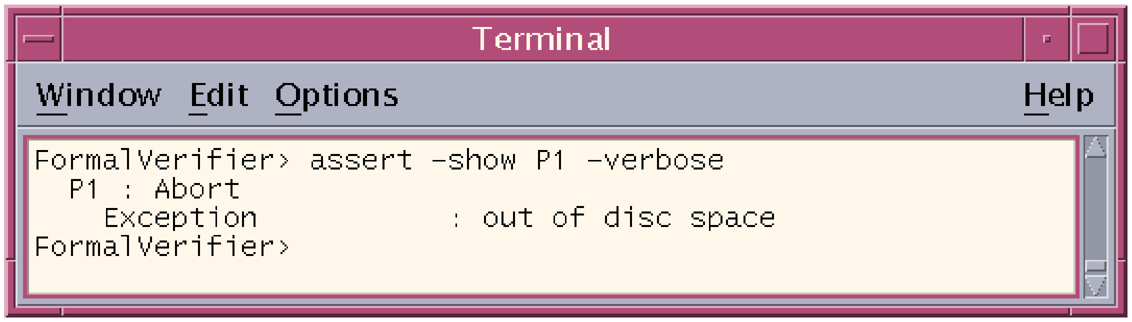
Note: You can use the mouse to resize the Secondary pane.

Secondary information can include the following:

- **Exception**

  When the result for an assertion is either *Explored* or *Abort*, an exception is reported and a reason is given. The reason could be `out of memory`, `out of time`, or `out of disc space`.

The following example shows secondary information for an *Aborted* assertion

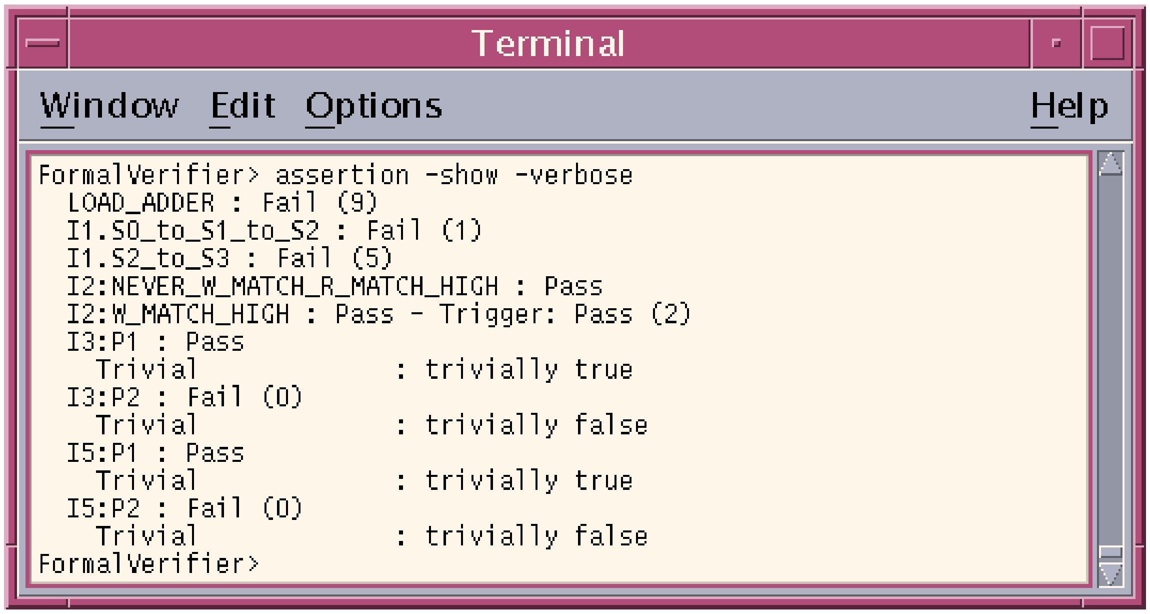


```
FormalVerifier> assert -show P1 -verbose
  P1 : Abort
    Exception           : out of disc space
FormalVerifier>
```

- **Trivial**

  When an assertion is either trivially true or trivially false, the secondary information for this assertion shows this. For more information about trivial properties, refer to the section titled Trivial Property Checks on page 83 of this guide.

  The following example shows secondary information for a set of trivial assertions.

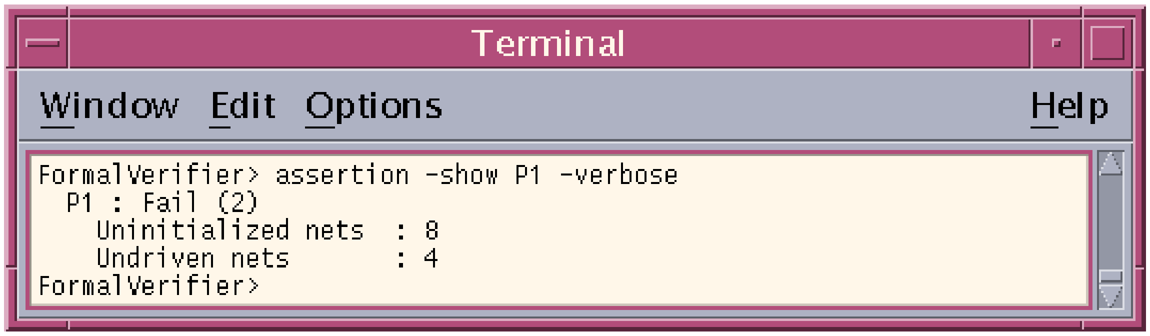


```
FormalVerifier> assertion -show -verbose
  LOAD_ADDER : Fail (9)
  I1.S0_to_S1_to_S2 : Fail (1)
  I1.S2_to_S3 : Fail (5)
  I2:NEVER_W_MATCH_R_MATCH_HIGH : Pass
  I2:W_MATCH_HIGH : Pass - Trigger: Pass (2)
  I3:P1 : Pass
    Trivial             : trivially true
  I3:P2 : Fail (0)
    Trivial             : trivially false
  I5:P1 : Pass
    Trivial             : trivially true
  I5:P2 : Fail (0)
    Trivial             : trivially false
FormalVerifier>
```

- **Uninitialized and Undriven nets**

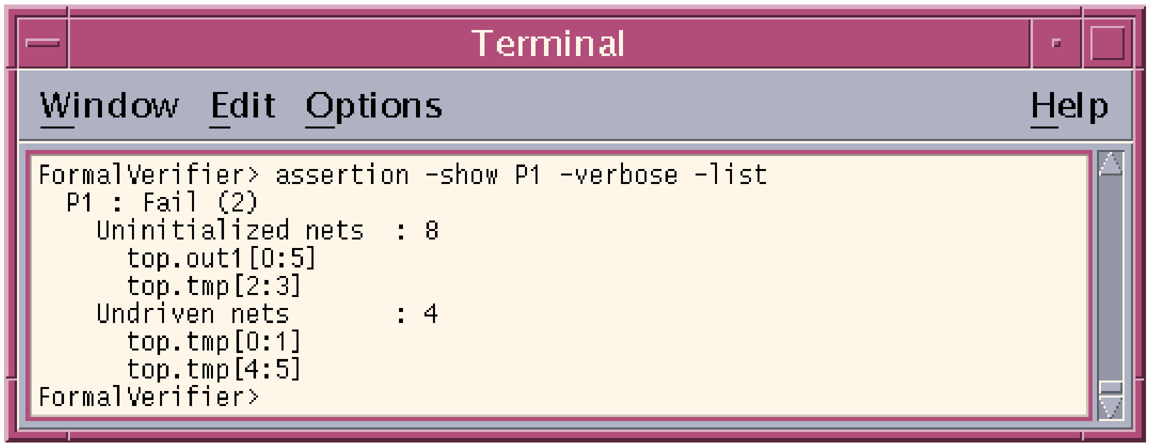
  Since the cause of a failed assertion could be an undriven or uninitialized net, you should be aware of any uninitialized or undriven net in the cone of influence of a failed assertion. The tool helps you understand the impact of these kinds of nets by reporting their existence in secondary information.

The following example shows a summary of uninitialized and undriven nets in the cone of influence of assertion `P1`.



```
FormalVerifier> assertion -show P1 -verbose
  P1 : Fail (2)
    Uninitialized nets  : 8
    Undriven nets       : 4
FormalVerifier>
```

The next example shows more detail about the specific uninitialized and undriven nets.



```
FormalVerifier> assertion -show P1 -verbose -list
  P1 : Fail (2)
    Uninitialized nets  : 8
      top.out1[0:5]
      top.tmp[2:3]
    Undriven nets       : 4
      top.tmp[0:1]
      top.tmp[4:5]
FormalVerifier>
```

- **Block**

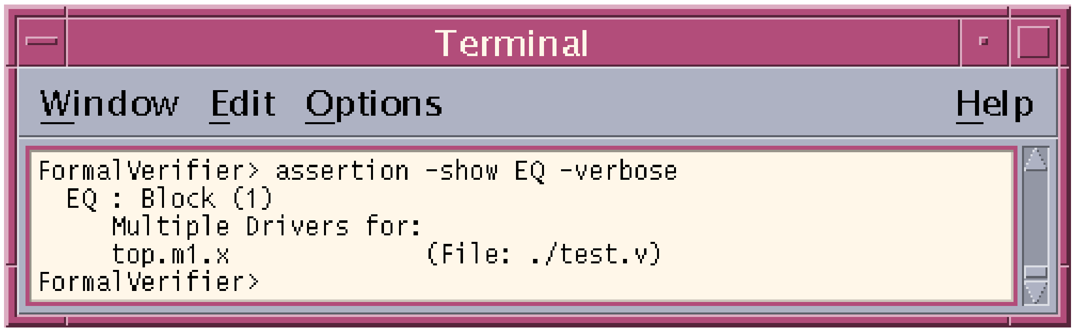
  When the status of an assertion is reported as *Block*, it is because of one of the following reasons:

  - ❑ Combinational loops exist in the design
  - ❑ Bus Contention errors were detected in the design.

  You can use the following command to view the reason for the *Block* status:

```
FormalVerifier> assertion -show <assertionNames> -verbose
```

The output of the above command is as shown below:



```
FormalVerifier> assertion -show EQ -verbose
  EQ : Block (1)
     Multiple Drivers for:
     top.m1.x              (File: ./test.v)
FormalVerifier>
```

In addition to the secondary information, you can also view the list of constraints used to verify the assertions. The list of constraints can include, user defined PSL/OVL constraints, pin constraints, clock constraints, and the properties used to verify the assertions. You can view this information using the `assertion -show -dependent` command. For example:

```
FormalVerifier> assertion -show -dependent
  CHECK1 : Pass
  Properties Used:
    n/a
  Pin Constraints Used:
    reset_n : 1'b1
  Clock Constraints Used:
    test.clk : (0, 1, 1, 2)

  CHECK1_trace : Pass (10)
  Properties Used:
    CHECK1 : Pass
  Pin Constraints Used:
    reset_n : 1'b1
  Clock Constraints Used:
    test.clk : (0, 1, 1, 2)
```

In the above example, the command `assertion -show -dependent` displays the list of constraints used to prove assertions `CHECK1`and the trace of the assertion `CHECK1`. You will notice that against assertion `CHECK1`, no property is being used as a smart constraint to verify the assertion wheras against `CHECK1_trace`, the property `CHECK1` is being used as a smart constraint to verify the assertion.

Clubbing of Assertions

In order to gain performance improvement, user defined assertions/covers can be clubbed together for verification. You can enable clubbing of assertions by issuing the following command:

```
FormalVerifier> define clubbing on
```

When you set the `clubbing` option to `on`, the tool automatically selects the assertions to be clubbed together for verification. Only user defined safety assertions will be clubbed. The clubbing feature is only supported for `bow1`, `bow2`, and `bow3` engines. This setting will not apply to the remaining engines.

Verification of Clubbed Assertions

The following points need to be considered during the verification of clubbed assertions:

- `Total Effort = effort specified for one assertion * number of assertions in a club`
- The command, `assertion -show -time` will report the total verification time of an entire assertion club.
- The command, `assertion -show -dependent` will list the dependents of an entire assertion club.
- The `fanin` reporting for an entire assertion club, can be viewed as:

  ```
  FormalVerifier> report -fanin P1 P2 P3
  ```
- During verification, if a combinational loop is detected in the cone of influence of an assertion in a club, then result status of all assertions within that club will be displayed as *Blocked*.
- Trigger for assertions will continue to run even if an assertion in the club fails.
- In case of OVL assertions, if one subassertion of an OVL fails, then the verification will stop and the result status of each OVL will be displayed as *Fail*.
- In case the command, `define stop_on_fail` has been set to `on`, then the verification will stop on first assertion failure. All other assertions in the club will display the result status as *Explored*.
- In case of a *Blocked* status for all checks clubbed together, you can debug any of the checks to find the nets that are part of the combinational cycle or you can run the checks after setting the clubbing command to `off`.

Validating Results

As in most other engineering endeavors, it is important to cross-check your results. Assertion fail results are easy to check, since the counter-example waveforms tell you how the system is behaving and you can use your engineering judgement to decide the true cause of the failure. Assertion pass results, on the other hand, are more subtle. In addition to correct operation of the design under test, there can be several other reasons for a passing result. These include the following:

- **Error in the assertion**

  An incorrect assertion may never fail. Consider the following:

  ```
  // psl P1: assert sig_a && sig_b -> sig_c;
  ```

  If the design under test cannot drive `sig_a` and `sig_b` to both be high simultaneously, then the assertion, in a sense, never checks that `sig_c` behaves correctly. This situation often arises because of an error in the assertion itself.

- **Over-constraining**

  Over-constraining is the elimination of valid behaviors of design inputs through incorrect constraints. When valid behaviors of the inputs are eliminated, it may cause some of the assertions to never be triggered and therefore pass.

- **Vacuity**

  Vacuity is a situation, resulting from incorrect constraints, which makes it so that there are no input sequences that can satisfy all the constraints simultaneously. In vacuity situations, assertions pass, regardless of the correctness of the assertion or of the design under test. Vacuity problems can be hard to diagnose, because the reason for vacuity is often manifested in an interaction between one or more constraints and the design itself.

Fortunately, Formal Verifier has several features that can help you validate your results. These features include:

- Automatic Vacuity Check
- Automatic Trigger Checks for OVL, PSL, and SVA
- Automatic Trace Checks for PSL and SVA
- PSL and OVL Sanity Check Support

Automatic Vacuity Check

Automated vacuity check is discussed in detail in the section titled Vacuity Check on page 117.

Automatic Trigger Checks for OVL, PSL, and SVA

Formal Verifier automatically generates trigger checks for certain OVL , PSL, and SV assertions. Trigger checks are existential assertions that capture the enabling condition of OVL, PSL, and SV assertions. Consider the following PSL assertion:

```
        // psl a1: assert
        //      always {req} |=> {ack[*1:5]} @posedge(clk);
```

The meaning of this assertion, in natural language is `if req is asserted, then ack should be asserted between 1 and 5 clock cycles later`. The enabling condition of the assertion is `req asserted`. If `req` can never be asserted, then the assertion passes. This is not likely what was intended. A trigger check ensures that there is some way for `req` to be asserted. If there is no witness that shows the enabling condition can happen, then the trigger check fails.

The above trigger check could be manually written, using the PSL `cover` statement, as follows:

```
    // psl cover1: cover {req} @posedge(clk);
```

Trigger checks are automatically generated for the following forms of PSL assertions:

PSL Assertion	Trigger Check Generated

`always (enable -> fullfill) @(posedge clk);`
 	
`always (enable -> (fullfill abort discharge)) @(posedge clk);`
 	
`always (enable -> PSL_formula) @(posedge clk);`
	`cover {enable} @(posedge clk);`

`always ((enable -> fullfill) abort discharge) @(posedge clk);`
 	
`always ((enable -> PSL_Formula) abort discharge) @(posedge clk);`
	`cover {enable && !discharge} @(posedge clk);`

`always (enabling sequence |-> fullfilling sequence) @(posedge clk);`
 	
`always (enabling sequence |-> (fullfilling sequence abort discharge) @(posedge clk);`
 	
`always (enabling sequence |=> fullfilling sequence ) @(posedge clk);`
 	
`always (enabling sequence |=> (fullfilling sequence abort discharge)) @(posedge clk);`
	`cover {enabling sequence} @(posedge clk);`

`always ((enabling sequence |-> fullfilling sequence) abort discharge) @(posedge clk);`
 	
`always ((enabling sequence |=> fullfilling sequence) abort discharge) @(posedge clk);`
	`cover {{enabling sequence} && {!discharge[*]}} @(posedge clk);`

Trigger checks are also automatically generated for OVL assertions. Consider the OVL property `assert_window`:

```
    assert_window(clk, rst_n, start, test_expr, end)
```

In this property, `test_expr` must be asserted the next cycle after `start` is asserted and be continuously true up to and including the cycle when `end` is asserted.

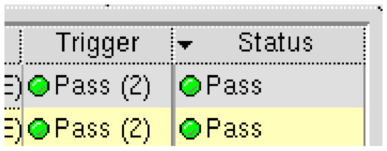
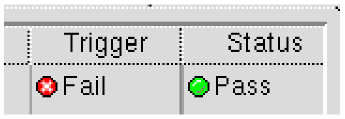
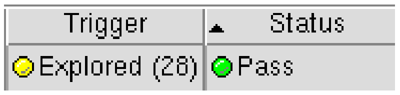
Trigger checks are also automatically generated for the following forms of SV assertions:

SV Assertion	Trigger Check Generated
`@(posedge clk) ((enable) |-> (fulfill));` `@(posedge clk) ((enable) |-> (SV_Formula));`	`cover {a} @(posedge clk);`
`@(posedge clk) disable iff (discharge) ((enable) |-> (fulfill));` `@(posedge clk) disable iff (discharge) ((enable) |-> (SV_Formula));`	`cover {enable && !discharge} @(posedge clk);`
`@(posedge clk) ((enabling sequence) |-> (fulfilling sequence));` `@(posedge clk) ((enabling sequence) |=> (fulfilling sequence));`	`cover {enabling sequence} @(posedge clk);`
`@(posedge clk) disable iff (discharge) ((enabling sequence) |-> (fulfilling sequence));` `@(posedge clk) disable iff (discharge) ((enabling sequence) |=> (fulfilling sequence));`	`cover {{enabling sequence} && {!discharge[*]}} @(posedge clk);`

For a given assertion, the tool verifies the corresponding trigger check only when the assertion passes. This is because when an assertion fails, it means that the assertion was triggered and you can view the triggering condition in the counter-example of the failure.

Hence, the focus of this section is the possible trigger check results when an assertion `passes`.

Trigger check results are shown in a separate column in the GUI as shown in the following figures:

Pass in the Trigger Column	Fail in the Trigger Column	Explored in the Trigger Column
 Trigger / Status; Pass (2) / Pass; Pass (2) / Pass A *Pass* in the *Trigger* and *Status* columns indicates that the enabling condition of the property is reachable. The number `2` inside the parentheses indicates that a witness for the enabling condition was found at a depth of `2` cranks. You can view the witness waveform by right-clicking the trigger check and selecting *Witness* from the menu or by selecting the assertion and clicking the *witness* button ( ).	 Trigger / Status; Fail / Pass A *Fail* in the *Trigger* column and a *Pass* in the *Status* column indicates that the assertion was never triggered. This indicates a *bogus Pass* and the situation should be examined closely as described in Validating Results on page 140. Consider the example of an assertion where `always (a -> next(b))` and `a` never becomes `1`. The assertion will pass, but it will be a *Vacuous Pass* because the tool is not required to verify `b`.	 Trigger / Status; Explored (28) / Pass An *Explored* in the *Trigger* column and a *Pass* in the *Status* column indicates that the assertion was triggered, but at a depth of `28` cranks. This could also indicate a *Vacuous Pass* but it is not possible to confirm this as the assertion was triggered.

Note: Trigger checks are assertions in their own right and can also have the following results:

- In case of a *Block* in the *Trigger* column, investigate the result as any other blocked assertion, by viewing its counter-example.
- In case of an *Explored* in the *Trigger* column along with an *Explored* in the *Status* column, increase the effort (by using the `define effort` command) and focus on getting a definite status for the assertion.

Automatic Trace Checks for PSL and SVA

A trace is a complete witness for an assertion. It is generated considering the design, constraints, and the initialization. Trace check generates and displays a waveform for an input sequence that serves as a witness for an assertion. The witness shows a path in the design that traces the complete assertion sequence. IFV automatically generates trace check for PSL and SV assertions.

Note: Trace check will not be generated for properties, which have a blocked status.

Trace checks are automatically generated for the following forms of PSL assertions. This list covers most of the cases and gives an idea on the desired functionality.

PSL Assertion	Trace Check Generated
`always (a -> b) @(posedge clk);`	`cover {a && b} @(posedge clk);`

`always ((a -> b) abort c)`
`@(posedge clk);`
	`cover {a && b &&!c}`
	`@(posedge clk);`

`always (S1 |-> S2) @(posedge clk);`	`cover {{S1} : {S2}}`
	`@(posedge clk);`

`always ((S1 |-> S2) abort c))`
`@(posedge clk);`
	`cover {{{S1} : {S2} && {!c[*]}} @(posedge clk);`

`always (S1 |=> S2) @(posedge clk);`	`cover {{S1}; {S2}}`
	`@(posedge clk);`

`always ((S1 |=> S2) abort c))`
`@(posedge clk);`
	`cover {{{S1}; {S2} && {!c[*]}} @(posedge clk);`

`always (B1) @(posedge clk);`	`cover {B1} @(posedge clk);`

`always ((B1) abort c)`
`@(posedge clk);`
	`cover {B1 && !c} @(posedge clk);`

`never (B1) @(posedge clk);`	`cover {!B1} @(posedge clk);`

`always (eventually! B1)`
`@(posedge clk);`
	`cover {B1} @(posedge clk);`

`always (B1 -> eventually! B2 ) @(posedge clk);`
`always ({B1} |-> {[*]; B2}!)`
`@(posedge clk);`
	`cover {B1; [*]; B2}`
	`@(posedge clk);`

Trace checks are also automatically generated for the following forms of SV assertions. The derived cover checks are shown in the PSL format.

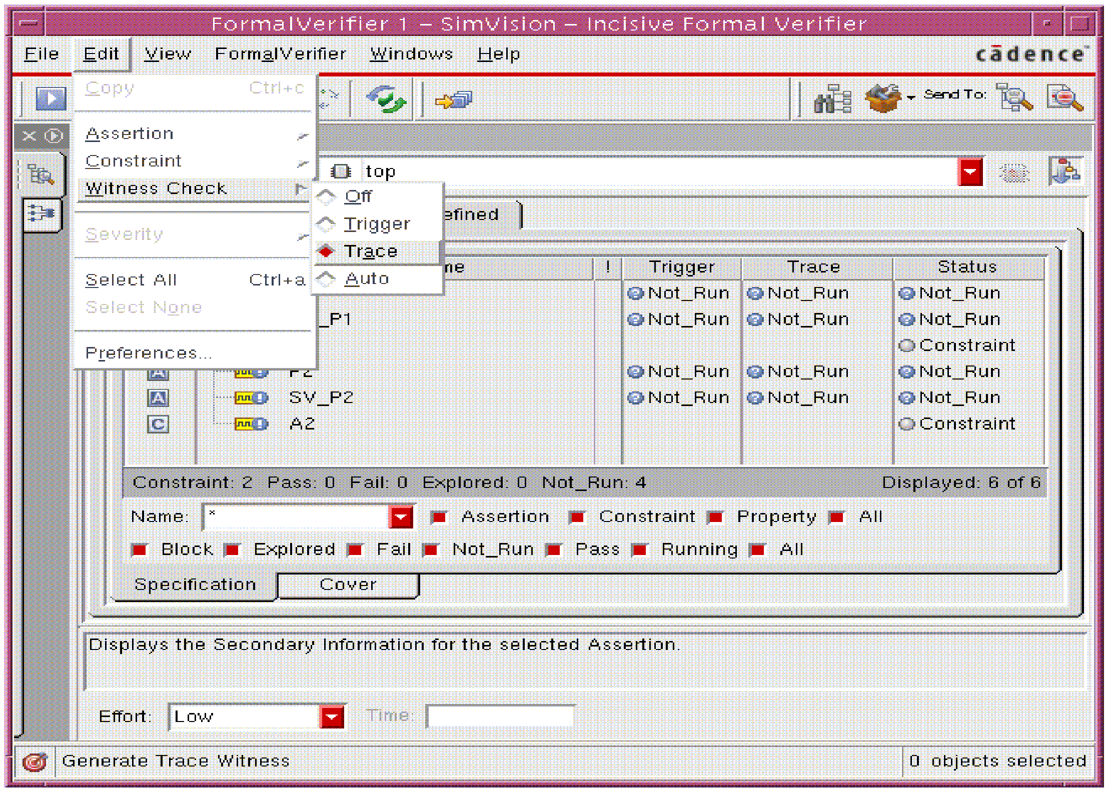
SV Assertion	**Trace Check Generated**
`@(posedge clk) ((a |-> (b));`	`cover {a && b} @(posedge clk);`
`@(posedge clk) disable iff(c) ((a) |-> (b));`	`cover {a && b &&!c} @(posedge clk);`
`@(posedge clk)((S1 |-> S2));`	`cover {{S1} : {S2}} @(posedge clk);`
`@(posedge clk) disable iff(c) ((S1 |-> S2));`	`cover {{{S1} : {S2} && {!c[*]}} @(posedge clk);`
`@(posedge clk) ((S1 |=> S2));`	`cover {{S1}; {S2}} @(posedge clk);`
`@(posedge clk) disable iff(c) ((S1 |=> S2));`	`cover {{{S1}; {S2} && {!c[*]}} @(posedge clk);`
`@(posedge clk) (B1);`	`cover {B1} @(posedge clk);`
`@(posedge clk) disable iff(c) (B1);`	`cover {B1 && !c} @(posedge clk);`
`@(posedge clk) not(B1);`	`cover {!B1} @(posedge clk);`
`@(posedge clk) ((B1 |-> ([*0:$] ##1 B2));`	`cover {B1; [*]; B2} @(posedge clk);`
`@(posedge clk) (1’b1 [*0:$] ##1 B1);`	`cover {B1} @(posedge clk);`

However, there are certain PSL and SV assertions for which trace checks are not generated. These are:

- `never (S1) @(posedge clk);`
- `forall i in {0:15}: always (DATA==i) @(posedge clk);`
- `@(posedge clk) a | -> not (b ##1 c ##1 d);`
- `@(negedge clk) if (a) b |=> c else d |=> e;`

A `witness_check` for an assertion helps you to generate witness for `trigger` as well as for the `trace`.

To generate the trace in the GUI, choose the *Witness Check* menu option from the *Edit* menu and select the *Trace* option, as shown below:



ne	!	Trigger	Trace	Status
		Not_Run	Not_Run	Not_Run
_P1		Not_Run	Not_Run	Not_Run
				Constraint
P2		Not_Run	Not_Run	Not_Run
SV_P2		Not_Run	Not_Run	Not_Run
A2				Constraint

Constraint: 2 Pass: 0 Fail: 0 Explored: 0 Not_Run: 4 Displayed: 6 of 6

Name: * 🟥 Assertion 🟥 Constraint 🟥 Property 🟥 All

🟥 Block 🟥 Explored 🟥 Fail 🟥 Not_Run 🟥 Pass 🟥 Running 🟥 All

Displays the Secondary Information for the selected Assertion.

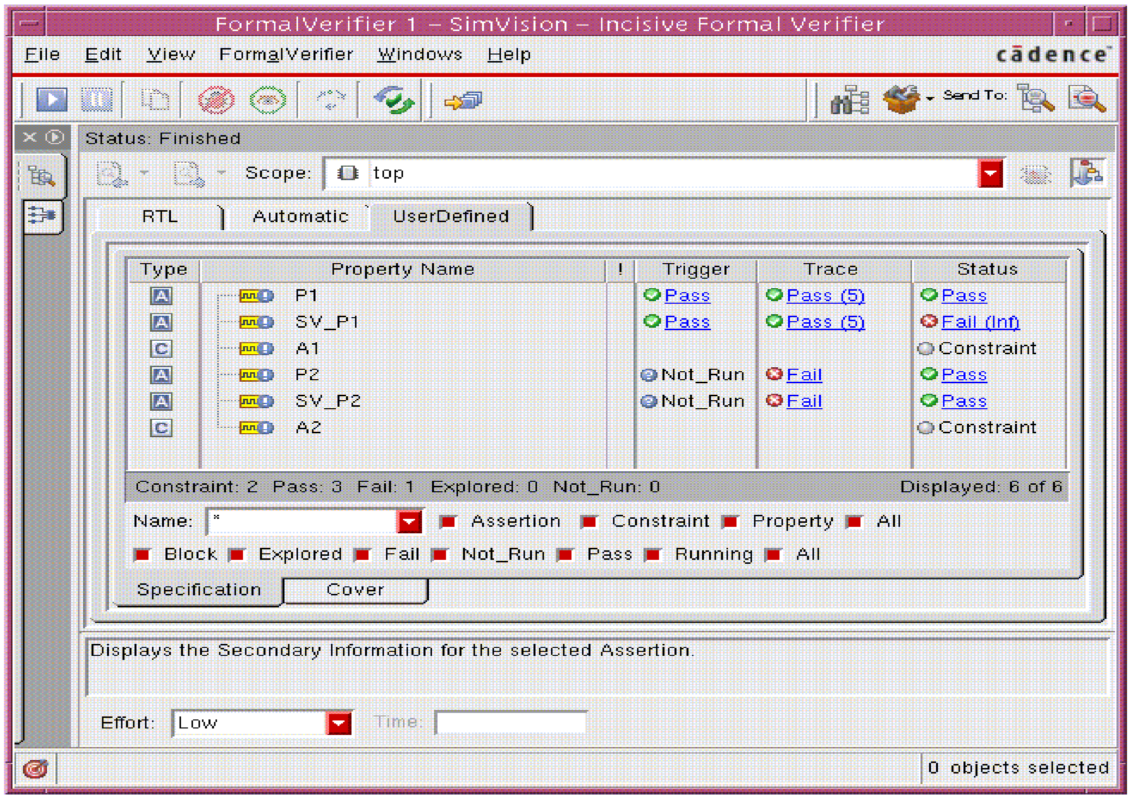
Effort: Low

Generate Trace Witness 0 objects selected

Click the *Run* button to submit the session for verification. Witness check results for both trigger as well as trace will be displayed as shown below.

When a trace for an assertion passes, IFV automatically passes the trigger also but it does not run the trigger. The trace witness includes the trigger therefore IFV marks the trace as a pass. In addition, a new option *Auto*, has been included in the *Witness Check* menu option.

If you select an *Auto* option, the trace for the assertion starts generating. If the trace status is fail or explored, then the trigger is run automatically.


FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cadence
Send To:
Scope: top
RTL
Automatic
UserDefined
Name: *
Specification
Cover
Effort: Low
Time:

Status: Finished

Type	Property Name	!	Trigger	Trace	Status
A	P1		Pass	Pass (5)	Pass
A	SV_P1		Pass	Pass (5)	Fail (Inf)
C	A1				Constraint
A	P2		Not_Run	Fail	Pass
A	SV_P2		Not_Run	Fail	Pass
C	A2				Constraint

Constraint: 2 Pass: 3 Fail: 1 Explored: 0 Not_Run: 0 Displayed: 6 of 6

Assertion Constraint Property All

Block Explored Fail Not_Run Pass Running All

Displays the Secondary Information for the selected Assertion.

0 objects selected

For more information on `witness_check` command, refer to *define witness_check* in *Chapter 1* of the *Formal Verifier Reference Manual.*

PSL and OVL Sanity Check Support

Sanity checks are used to test for the existence of specific valid behaviors. By testing for specific behaviors, you can assess whether assertions incorporating these behaviors are being checked by Formal Verifier. Cadence recommends that you use sanity checks as part of your verification methodology.

Trigger checks, described in the previous section are a form of sanity check that the tool is able to generate automatically. There are situations when the automatically generated checks are not sufficient. Consider the following PSL assertion:

```
    //psl a1: assert always {req} |=> {ack[*1:5]};
```

In this example, assertion `a1` checks for `ack` to be asserted between `1` and `5` cycles after `req` is asserted. As discussed in the section titled Automatic Trigger Checks for OVL, PSL, and SVA on page 141, the tool will generate a trigger check, such as the following:

```
// psl cover1: cover {req} @posedge(clk);
```

This trigger check may not be sufficient. To feel comfortable that assertion `a1` is testing actual behavior, you may want to test that `ack` can actually be asserted. A typical way to write sanity checks is to look for corner case behavior. In this example, the:

- first corner case behavior is for `ack` to be asserted in the next cycle after `req`.
- second corner case behavior is for `ack` to be asserted `5` cycles later.

These two corner case behaviors can be tested with the following two sanity checks:

```
//psl sanity1: cover {req;ack};
//psl sanity2: cover {req;!ack[*4];ack};
```

The assertion `a1` could also be written in OVL as follows:

```
assert_frame #(0, 1, 6) a1(clk, 1'b1, req, ack);
```

Formal Verifier supports an extension of OVL called `assert_cover`, which is analogous to PSL `cover`. The PSL sanity checks above could be recorded using this OVL extension as:

```
assert_cover #(0,2,0) sanity1 (clk,rst_n,{req,ack});
assert_cover #(0,6,0) sanity2
    (clk,rst_n,{req,!ack,!ack,!ack,!ack,ack});
```

When sanity checks passes, Formal Verifier finds a witness for the valid behavior. You can use the `witness` command to view the witness waveform. Alternatively, you can either click the witness button ( ) in the Formal Verifier GUI or you can right-click on the property and select *Witness*.

Debugging Trigger and Sanity Checks

The failure of a trigger check or sanity check has a variety of potential causes, such as the following:

- Over-constraining
- Misunderstanding of some aspect of the system
- An incorrect assertion

The lack of a counter-example compounds the difficulty of debugging failed trigger and sanity checks; however you can use the following basic process to help you debug:

1. **1.** Identify the set of constraints used in the failed sanity check by using the `assertion -dependent` command as follows:

   ```
   FormalVerifier> assertion -show <assertionName> -dependent
   ```

2. **2.** Delete all the constraints and rerun the failed assertion. You can do this by using the command `constraint -delete <constraint_name>` one or more times.

   If the assertion passes after removing constraints then most likely you have an over-constraint or vacuity situation. To debug the over-constraint situation, you should add constraints one at a time, re-running the proof each time. At the point that the assertion fails again, you know that you have identified at least one of the offending constraints.

   You can then continue to delete other constraints until the assertion passes. At this point, you have identified another of the offending constraints.

   By repeatedly adding and delete constraints, you should be able to narrow down the set of constraints that are causing the problem.

   During this process, you may find that all the constraints are correct. This would imply that either the design has bugs or the sanity check itself (or assertion corresponding to the failed trigger) is incorrect.

3. **3.** To help you better understand what might be happening, you could weaken or alter the sanity check. For example, you might try weakening the assertion `sanity2` above, so that it checks whether `ack` can be asserted `4` cycles after `req`:

   ```
   //psl sanity2: cover {req;!ack[*3];ack};
   ```

   You could continue weakening the assertion until you find what is feasible in this design.

9 Debugging Failures and Viewing Witnesses

For failing assertions, Formal Verifier generates a counter-example that shows a sequence of design states and input values leading to the failure. Similarly, for passing existential assertions, Formal Verifier generates a witness that shows a sequence of states and input values that demonstrate that the assertion is satisfied.

There are several common reasons for an assertion failure:

- A bug in the design

  You can correct this problem by identifying and fixing the bug.

- A bug in the assertion

  You can correct this problem by identifying and fixing the bug.

- Incorrect input to the design

  Formal Verifier will analyze all input sequences that obey the given constraints. An incorrect input sequence implies that the constraints are incorrect or insufficient. The solution is to add missing constraints or fix incorrect constraints

You can examine a counter-example or witness by viewing waveforms in the *Waveform Viewer* or by viewing annotated signal values in the *Source Browser*. The failure condition is manifested at the end of the counter-example. Formal Verifier typically shows the shortest sequence of states sufficient to demonstrate the failure.

Waveform Viewer

Formal Verifier uses the SimVision Waveform Viewer to display counter-example waveforms showing conditions leading to failure. To view a counter-example waveform in the *Waveform Viewer*, use the following command:

```
FormalVerifier> debug <assertionName>
```

Alternatively, in the Formal Verifier GUI, you can either click the debug ( ) button or you can right-click the property and select *Debug -> Show Static*.

Note: A counter-example waveform is not available for trivially false assertions. If you try to debug a trivially false assertion, you will get the following warning:

*formalverifier: *W,ASTFNC: Assertion is "Trivially False". The counter-example is not available.*

Similarly, you can view a witness waveform for an existential assertion in the *Waveform Viewer* by using the following command:

```
    FormalVerifier> witness <assertionName>
```

Alternatively, in the Formal Verifier GUI, you can either click the witness ( ) button or you can right-click on the property and select *Witness -> Show Static*.

Waveforms for failed safety assertions and passed existential assertions are slightly different from waveforms for failed *liveness* assertions. Refer to the section titled Classes of Properties on page 88 for more information on property types.

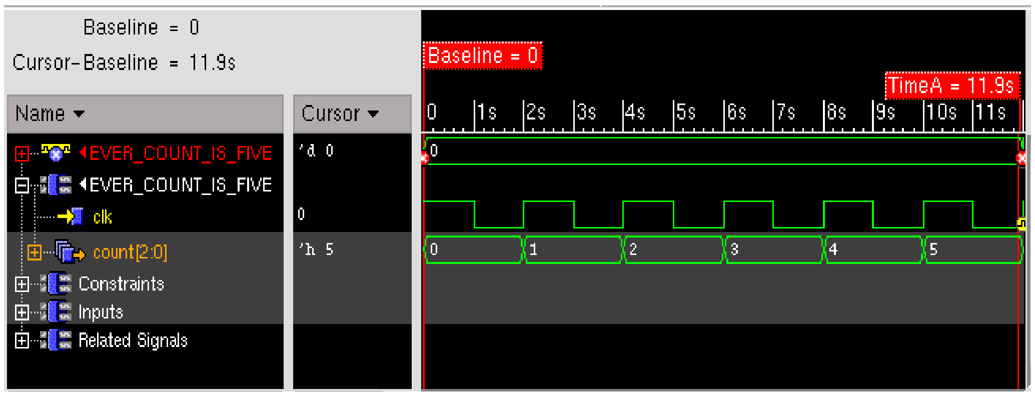
Waveforms for Safety and Existential Assertions

A failed safety assertion has an associated counter-example in which the final state is inconsistent with the stated assertion.

Consider the following assertion:

```
    // psl NEVER_COUNT_IS_FIVE:
    //       assert never (count == 5) @(posedge clk);
```

In the final state of a counter-example, count will have the value 5. The following figure shows a counter-example for this assertion.



In this counterexample, the assertion fails at crank `12` (which is six clock cycles), because count takes the value 5 at the positive edge of `clk`. The failing point is marked by a small red dot with an `x` (✖).

Witness waveforms for passed existential assertions are essentially the same as counter-examples for failed safety assertions except that the final state shows that the desired behavior can be reached.

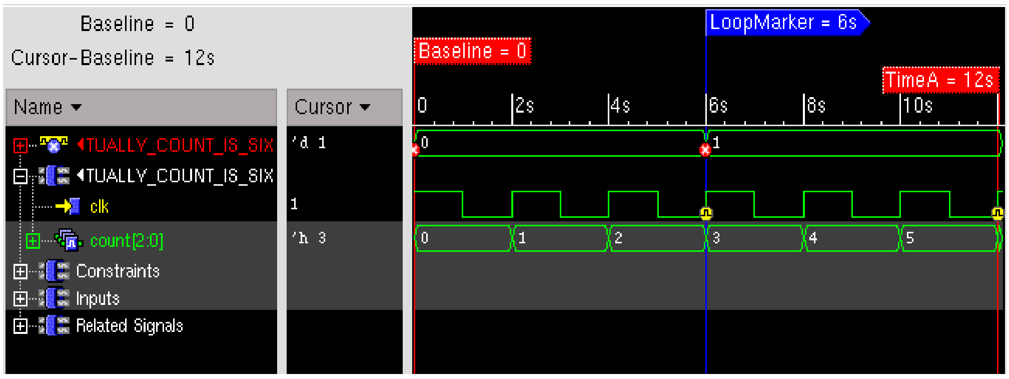
Waveforms for Liveness Assertions

A failed *liveness* assertion has an associated counter-example in which there is an infinitely repeating sequence of non-satisfying states for the assertion. The counterexample thereby shows a case where an expected behavior never happens.

Consider the following simple Verilog RTL:

```
module counter(input clk, rst);

reg [2:0] count;

always @(posedge clk)
    if (rst) count <= 3'd0;
    else if (count==3'd5) count <= 3'd3;
    else count <= count+1'b1;

// psl default clock = (posedge clk);
// psl EVENTUALLY_COUNT_IS_SIX: assert
//        always count==1 -> eventually! count==6;
endmodule
```

In this example, `count` is reset to zero and then counts up to `5`. Once count reaches `5`, it cycles back to `3` and continues counting up. The assertion `EVENTUALLY_COUNT_IS_SIX` says that once `count` equals `1`, then it should eventually equal `6`. Based on the RTL, this assertion should fail. Indeed, that is what the counter-example shows.



The counter-example shows:

- An initial sequence of states in which `count` starts at `zero` and increments to `3`.
- Then, a sequence of states, termed a *loop*, which infinitely repeat. The start of the repeating sequence of states is denoted by *LoopMarker*, which is placed at crank `6`. The end of the repeating sequence is denoted by the marker on the far right side of the waveform.

Because the waveform represents an infinitely long sequence of states and `count` is not equal to `6` anywhere along the entire waveform, `count` can never equal `6`.

Example Waveform of the Property

IFV automatically allows you to visualize the property in the form of a waveform without considering the design or constraints. Seeing a waveform of the property always provides a better understanding of the property written.

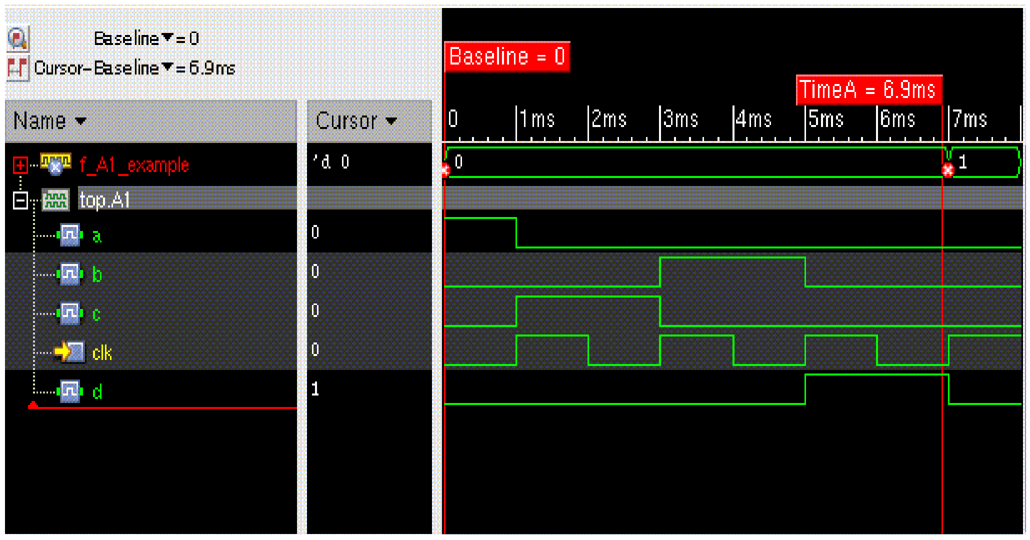
Consider the following property:

```
//psl A1: assert always ({a} |=> {c; b; d}) @(posedge clk);
```

To view the waveform of the property, A1, specify the following command at the tcl prompt.

```
Formal Verifier > example <property name>

Formal Verifier > example A1
```

This launches the waveform in the SimVision Waveform Viewer, as depicted below. This waveform is also known as an example.



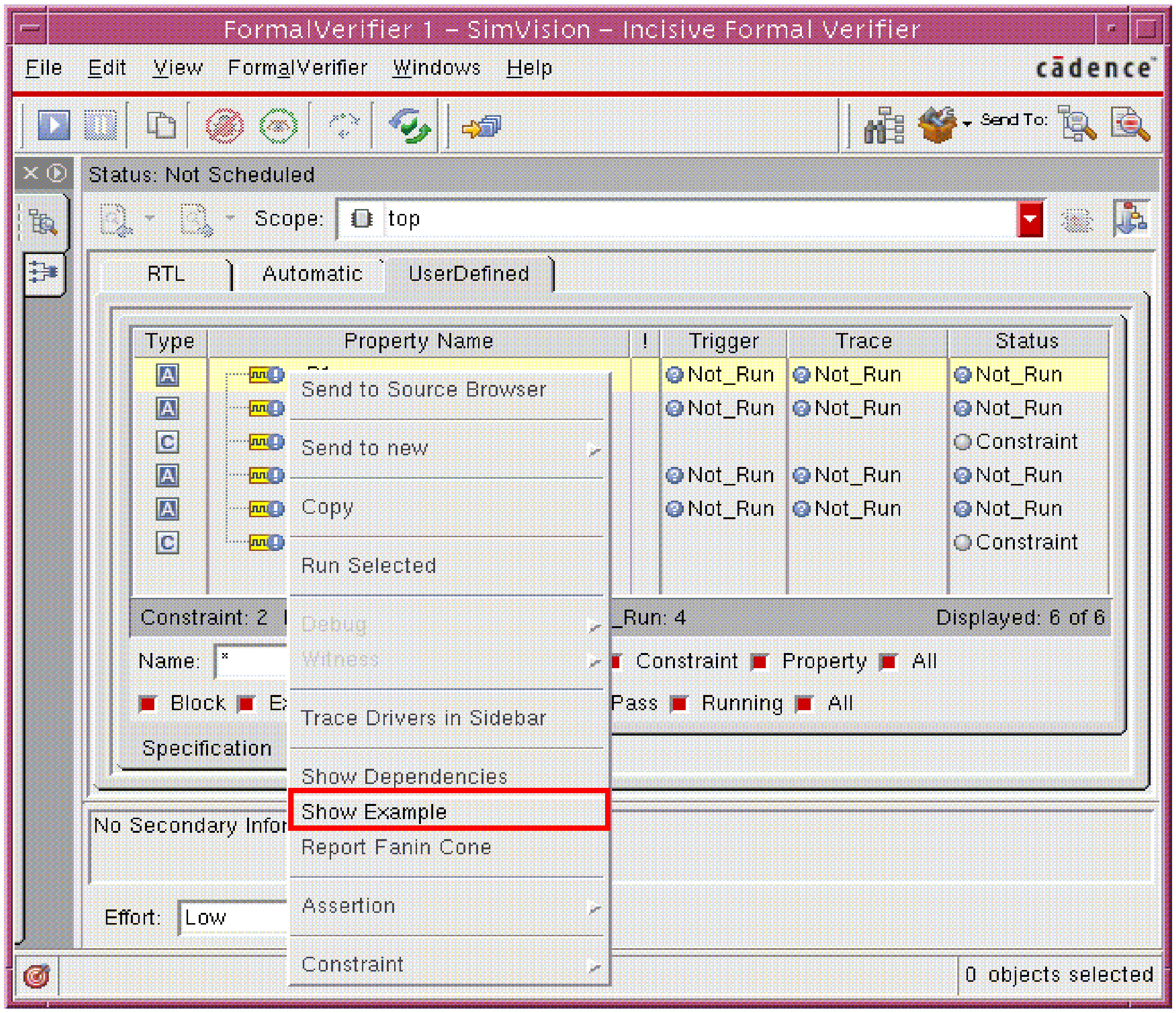
The example can be generated for properties, assertions, and assumptions irrespective of whether their result status is `Pass`, `Fail`, or `Not_Run`. In addition, the command, `example`, is not linked to any other command and can be specified at any point of time. You can specify the command, `example`, before or even after specifying the `prove` command.

However, you need to consider the following limitations when viewing the example for any property:

- You will be able to run `example` command for only one property at a time.

- This command is supported only on covers and those forms of properties for which the trace mode of `witness_check` is valid. For a list of supported PSL and SVA forms, refer to Section, *Automatic Trace Checks for PSL and SVA*, in Chapter, *Running Verification and Understanding Results* of the *IFV User Guide*.
- This command will only run in wait mode. That is, while this command is being run for launching the waveform, and you specify Cntrl-C, then the process will be killed. The command will no longer continue to run in the background.
- The `example` command cannot run while `prove` command is in progress in the no-wait mode.
- The example is available only for PSL and SVA property languages and not for OVL.
- The example is not available for the trivially false properties. Wild carding will be supported for the property name. However, if there are multiple properties matching the given property name, then the example will not be launched.

Note: To launch the example of the property, you can also select the option, *Show Example*, available on the RMB of the property name, in the Formal Verifier GUI, as shown in the figure below:



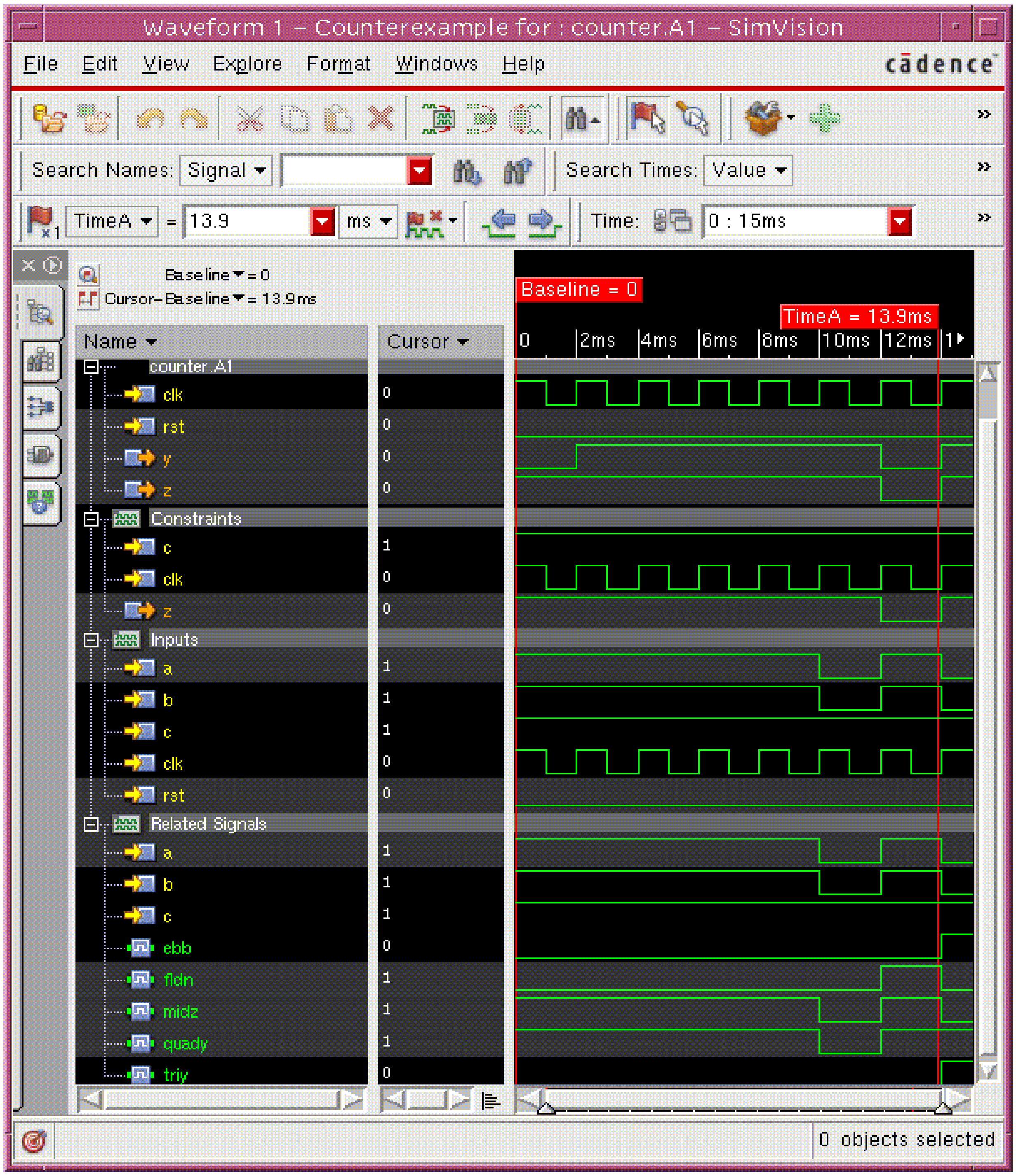
The option, *Show Example*, will be enabled for all properties, which are of the supported form, else the command will be displayed in gray.

Default Signals

When the Waveform Viewer opens, there are several groups of signals displayed by default. These are organized to help you efficiently understand the waveform. Consider the following Verilog module:

```
module counter(input clk, rst, a, b, c, output reg y, z);

reg midz, quady, triy, ebb, fldn;

always @* begin
    midz = a && b;
    quady = midz || triy;
    triy = ebb && y;
    fldn = c ^ z;
end

always @(posedge clk)
    if (rst) begin
        z <= 1'b0;
        y <= 1'b0;
        ebb <= 1'b0;
    end
    else  begin
        z <= midz;
        y <= quady;
        ebb <= fldn;
    end

// psl default clock = (posedge clk);
// psl A1: assert always ({z[*6]} |=> {y}) abort(rst);
// psl C1: assume always c && !z -> next c;
endmodule
```

The following figure shows the default groups created for a counter-example to `A1`.



In this example, there are several groups, each of which has been expanded to show the underlying signals:

- Assertion group

  This group gets its name from the failed assertion. That is, the full hierarchical name of the assertion is the name of the group. This group contains the signals that are referred to by the failed assertion.

  Initially, this is the only group that is expanded.

  In this example, the assertion group contains `clk`, `rst`, `y`, and `z`, because these signals are referred to in assertion `A1`.

  Note: The signal `clk` is referred to indirectly, because the default clock statement applies to `A1`.

- Constraints group

  The name of this group is *Constraints*. This group contains the signals that are directly referred to in constraints used to verify the assertion. The constraints used to verify the assertion can include:

  a. User defined PSL/OVL constraints

  b. Pin constraints added using Tcl command `constraint -add -pin`

  c. Clock constraints added using Tcl command `clock -add`

  d. Properties being used as smart constraints.

  Initially, this group is collapsed. Click the `+` sign next to the group name to expand the group.

  In this example, the Constraints group contains `clk`, `c`, and `z`, because these signals are referred to in constraint `C1`.

  Note: The signal `clk` is referred to indirectly, because the default clock statement applies to `C1`.

- Inputs group

  The name of this group is *Inputs*. This group may contain the following signal types:

  a. Primary inputs in the cone-of-influence of the assertion

  b. Primary inputs in the cone-of-influence of one or more relevant constraints

  c. Undriven nets

  d. Outputs of blackboxed instances

Initially, this group is collapsed. Click the `+` sign next to the group name to expand the group.

In this example, the Inputs group contains `clk`, `rst`, `a`, `b`, and `c`, because these signals are the primary inputs in the cone-of-influence of `A1`and also in the cone-of-influence of `C1`.

- Related Signals group

  This group contains at most 20 signals that are related to signals referenced in the assertion. The tool heuristically selects these signals from among all available signals.

  Initially, this group is collapsed. Click the `+` sign next to the group name to expand the group.

  In this example, the Related Signals group contains `clk`, `rst`, `a`, `b`, `c`, `midz`, `quady`, `triy`, `fldn`, and `ebb`. All of these signals are in the cone-of-influence of `A1`.

You can add and delete groups, move groups around, and do other operations on group. More information on signal groups is available in the *SimVision User Guide*.

Note: If there are no signals to be displayed in a group, then the group is not displayed in the Waveform Viewer.

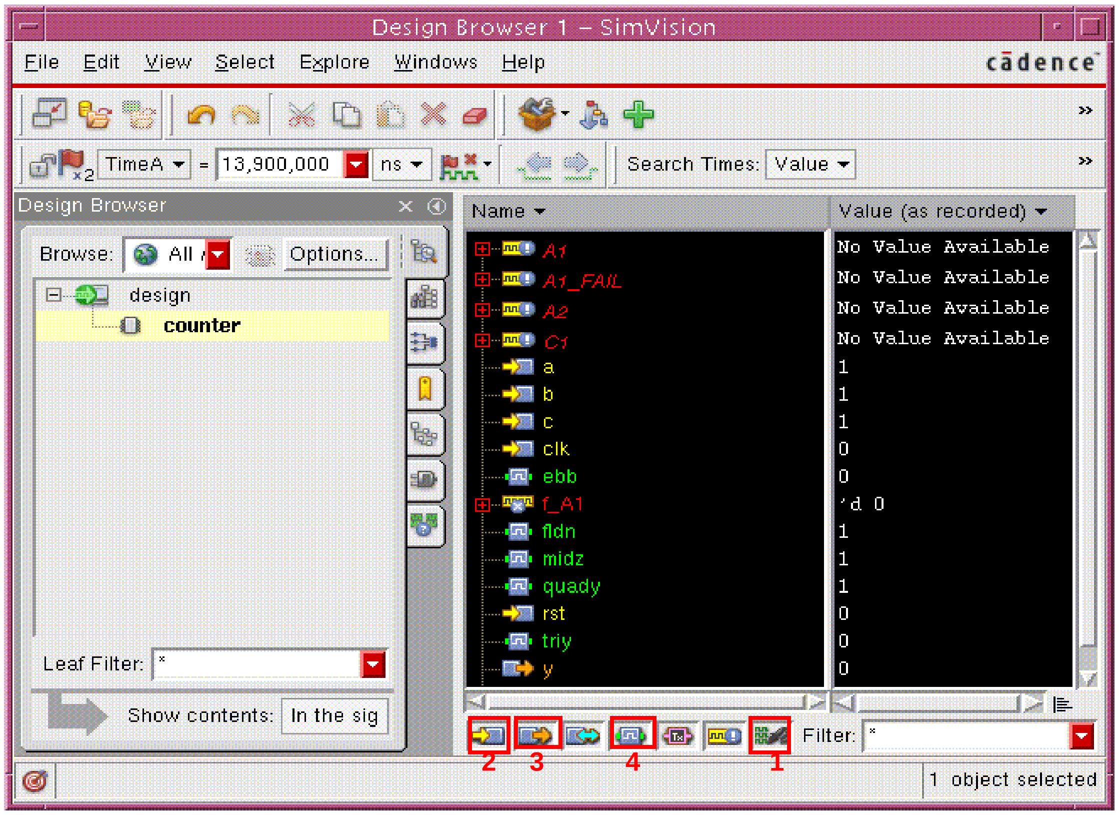
Adding and Deleting Signals

When the signals selected by the tool are not sufficient, you can add other signals to the waveform. The first step is to bring up the *Design Browser.*

In the SimVision Waveform window, click on the *Design Browser* icon on the sidebar. This will expand the Design Browser pane.

Alternately, you can click on the menu *Windows - New - Design Browser* to bring up the *Design Browser* window. The remainder of this section will focus on the Design Browser window, however the behavior of the Design Browser tab in the sidebar is essentially the same.

The Design Browser displays the design hierarchy and all the signals/variables available in the design. The following figure shows the Design Browser window:



Name	Value (as recorded)
A1	No Value Available
A1_FAIL	No Value Available
A2	No Value Available
C1	No Value Available
a	1
b	1
c	1
clk	0
ebb	0
f_A1	'd 0
fldn	1
midz	1
quady	1
rst	0
triy	0
y	0

All the signals and variables in the design are shown in the pane to the right. The signals and variables that are not relevant to the failed assertion are displayed in the *italicized* font and do not have a value associated with them.

To hide the signals/variables that are not relevant to the failed assertion and do not exist in the design database, use the Show/Hide unprobed signals (the highlighted box #1) button. Other buttons of interest include:

- Show/Hide inputs (the highlighted box #2)
- Show/Hide outputs (the highlighted box #3)
- Show/Hide internals (the highlighted box #4)

You can traverse through the hierarchy and select interesting signals. Once you have selected an interesting signal, you can right-click on it and chose *Send to Waveform Window*. This displays the selected signal in the Waveform Viewer.

For more information, refer to the section titled *Adding Signals to the Waveform Window* in *Chapter 16* of the *SimVision User Guide.* Many other useful Waveform Viewer features are also documented in the *SimVision Waveform User Guide*.

Zooming In and Out

There are several ways to zoom in and out in the *Waveform Viewer*.   A common technique for zooming in is to hold down the *Ctrl* key while clicking the left mouse button. With the left mouse button held down, you can drag across the portion of the waveform you wish to zoom in on.

A common technique for zooming out is to use one of the two following keystrokes:

- *Zoom out:Alt-o*
- *Zoom out full:Alt-=*

You can also use the zoom (+ – = ▣) buttons in the Waveform Viewer.

The + button means zoom-in. The - button means zoom out. The = button means zoom out full.

For more information on zooming, refer to the section titled *Zooming In and Out on the Waveform Data* in *Chapter 16* of the *SimVision User Guide*.

User-defined Signal List

Based on your requirement, you can specify the list of signals to be displayed when opening a waveform for showing a counter-example or a witness.

Open the waveform window and select the required signals. Save the file from the waveform window, as *File ->Save Command Script*. The file will be saved as Simvision Control File (SVCF).

To view a counter-example waveform with the signals specified in the SVCF, use the following command:

```
FormalVerifier> debug assertionName -input [filename]
```

This command will take a filepath as parameter instead of loading the default signal list in the waveform window. Specify the path of the SVCF. A valid SVCF will generate the counter-example waveform with required signals. To append the already existing SVCF, use the following command:

```
FormalVerifier> debug assertionName -input [filename] -append
```

Similarly, to view a witness waveform with the signals specified in the SVCF, use the following command:

```
FormalVerifier> witness assertionName -input [filename]
```

Source Browser

The Waveform Viewer enables you to see relationships between signals over time; however it does not enable you to understand how signal values are computed. The Source Browser, on the other hand, enables you to understand how signal values are computed but provides no ability to see signal relationships over time.

The Source Browser has many features, including:

- It recognizes and color codes Verilog, VHDL, and PSL keywords as well as comments and message strings that it detects in the source files.
- It lets you request information about or otherwise use design objects which are underlined in the Source Browser. For example, you can select a signal and traverse up or down the hierarchy to components connected by this signal.
- It lets you send groups of signals to other windows, such as the Waveform Viewer.
- It is aware of the cursor in the Waveform Viewer. For example, based on the time at which the cursor is positioned, the Waveform Viewer annotates signal values onto corresponding signals in the Source Browser at that time.
- It displays the scope history for easy access to previously visited scopes.

You cannot edit the source code directly in the Source Browser, because it would no longer represent the design snapshot. However, you can use the Source Browser to invoke a text editor, make changes to the source file, and then re-load the new snapshot.

Using Go to - Cause to Locate a Signal Assignment

By analyzing the waveform, you can determine which signal transitions caused an assertion failure. To understand why these transitions happened, you can use the Explore - Go to - Cause feature of SimVision. To use this feature, follow these steps:

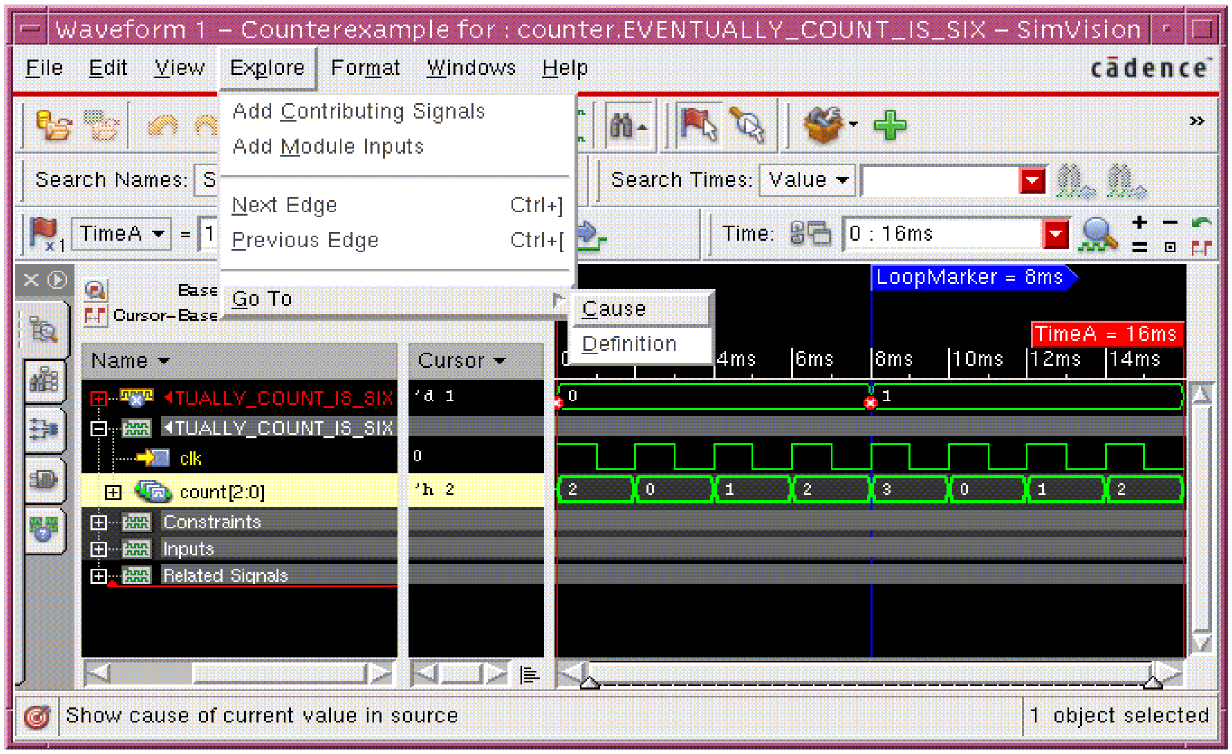
1. Click at the time of interest on the waveform to set the cursor.
2. Click on the signal name of interest to select the signal.
3. Right-click on the signal name and select *Go to - Cause* to bring up the Source Browser.

Note: Double-click in Waveform window has been mapped to *GoTo->Cause.*

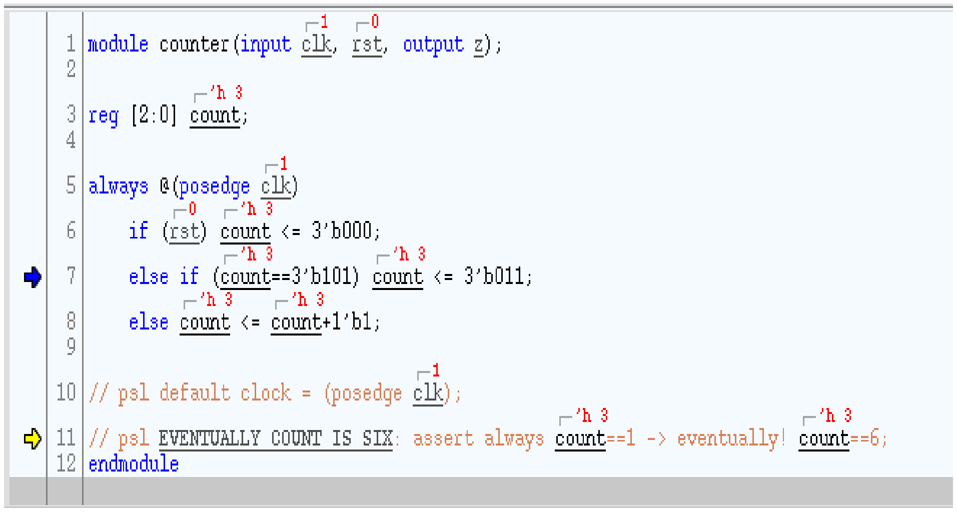
To understand this better, consider the following Verilog RTL:

```
module counter(input clk, rst);

reg [2:0] count;

always @(posedge clk)
    if (rst) count <= 3'd0;
    else if (count==3'd5) count <= 3'd3;
    else count <= count+1'b1;

// psl default clock = (posedge clk);
// psl EVENTUALLY_COUNT_IS_SIX: assert
//        always count==1 -> eventually! count==6;
endmodule
```

The counter-example for `EVENTUALLY_COUNT_IS_SIX` is shown in the following figure:



Crank `12` is interesting, because this is the end of the repeating sequence of states. With the cursor positioned at crank `12` as shown, select signal `count` and choose the *Explore - Go to - Cause* menu options to bring up the *Source Browser*.

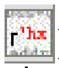


```
 1 module counter(input clk, rst, output z);
 2
 3 reg [2:0] count;
 4
 5 always @(posedge clk)
 6     if (rst) count <= 3'b000;
 7     else if (count==3'b101) count <= 3'b011;
 8     else count <= count+1'b1;
 9
10 // psl default clock = (posedge clk);
11 // psl EVENTUALLY_COUNT_IS_SIX: assert always count==1 -> eventually! count==6;
12 endmodule
```

The *Source Browser* window contains a blue arrow that points to the line of interest, `line 7`. This shows that `count` is assigned the value `3` at the end of the repeating sequence of states.

Repositioning the cursor in the *Waveform Viewer* will update the annotated values in the *Source Browser*; however the blue arrow will remain pointed at the same location. To update the blue arrow, you must reinvoke *Go to Cause* from the *Waveform Viewer*.

If you use *Go to Cause* capability on a sequential variable at `0`s, the blue arrow in the *Source Browser* window will point to the declaration of the variable in the design. In this example, if you use *Go to Cause* to view the cause of the transition of count at crank `0`, the blue arrow will point to line `6`, where `count` is declared.

Note: You can click on the value annotation () button in the Source Browser to turn-off displaying the signal values next to the signals in the HDL.

Go to - Cause in Hierarchical Designs

When invoking *Go to Cause* on a non-leaf-level signal in a hierarchical design, the blue arrow in the *Source Browser* points to the actual line in the RTL, which assigns the signal.

Searching for Signal References

Once you locate a signal assignment, you often need to understand the behavior of other related signals. To search the HDL code for signal references, you can select *Edit - Text Search* (or the hot-keys `Ctrl-F`). If a signal is connected hierarchically, you can also use the *Follow Signal* capability of the *Source Browser* to traverse up or down the hierarchy. Right-click the signal and select *Follow Signal.* This will pop up a menu which will show the available places in the hierarchy. You can then select the place of interest.

Adding Signals to the Waveform Viewer

Once you find a signal that may be involved in a bug, you can add it to the Waveform Viewer to allow you to see how this signal changes over time. Right-click on the signal and select *Send to Waveform Window*.

Tracing Drivers and Loads

For complete information on this refer to *Chapter 16*, titled *Tracing Paths with the Trace Signals Sidebar* of the *SimVision User Guide*.

Signals Outside the COI

Formal Verifier does a cone-of-influence abstraction while verifying an assertion. This means that for each assertion that is verified, Formal Verifier ignores the part of design not in the transitive fanin cone of the assertion. Because the signals that are not in the cone-of-influence are not used during verification of the assertion, they are not available in a counter-example for the assertion.

You can use the command `debug <assertionName> -dynamic` to generate values for all the signals in the design, regardless of whether they are in the cone-of-influence of the specified assertion. This command causes Formal Verifier to run an internal event-based simulator on the full design using input values derived from the assertion failure. The results of this event-based simulation can be viewed in the *Waveform Viewer*.

Since the internal model that Formal Verifier uses is a cycle-based model, there could be differences between it and the event-based model that the simulator uses. Refer to the chapter titled *Formal Analysis and Simulation Result Differences* of the *Formal Verifier Reference* for more information on cycle and event differences.

To better understand how the cone-of-influence affects available signals in both the default and the dynamic case, consider the following VHDL example:

```
package pack is
  type CVAL is (ZRO,ONE,TWO,TRE,FAR,FIV,SIX,SVN);
end;
package body pack is
end;

library ieee;
use work.pack.all;
use ieee.std_logic_1164.all;

entity top is
  port (clk : in std_logic;
  reset: in std_logic;
  count : buffer std_logic_vector(2 downto 0));
  end;

architecture rtl of top is
  signal verifcount: CVAL;
  begin

-- psl property P1 =
--      always({(count="000")}|=>
--     {(count="101");(count="001");(count="100")}) @ (rising_edge(clk));
-- psl assert P1;
verifcount <= ZRO when (count = "111") else
ONE when (count = "010") else
TWO when (count = "110") else
TRE when (count = "011") else
FAR when (count = "000") else
FIV when (count = "101") else
SIX when (count = "001") else
SVN when (count = "100") ;

process (clk,reset)
    begin
    if (reset='1') then
        count <= "111";
        elsif (clk'event and clk = '1') then
            case count is

               when ("111") => count <= "010";
               when ("010") => count <= "110";
               when ( "110") => count <= "011";
               when ( "011") => count <= "000";
               when ( "000") => count <= "101";
               when ("101") => count <= "001";
               when ( "001") => count <= "111";
               when ("100") => count <= "111";
               when others => null;
               end case;
    end if;
    end process;
    end;
```

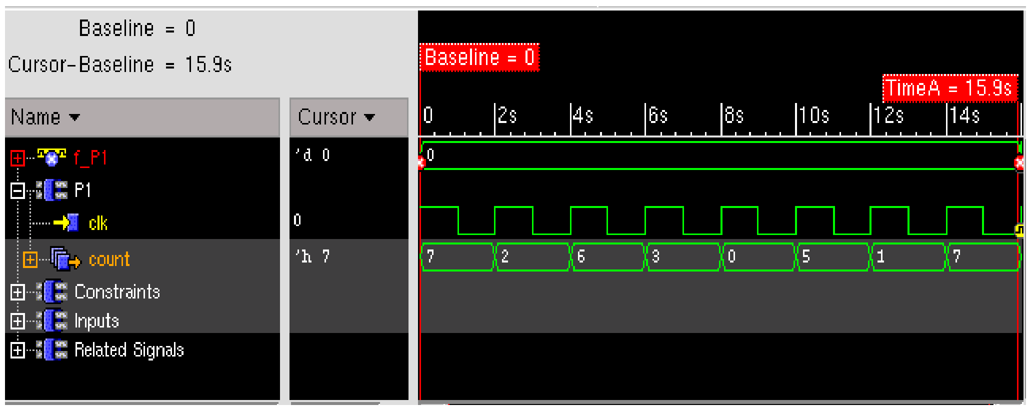
In the above design, the signal `verifcount` is not in the transitive fanin cone of the assertion P1. In this example, P1 fails when verified. The command:

```
FormalVerifier> debug P1
```

displays the following message:

Counter-example launched.

and shows the following counter-example:



When you try to add `verifcount`, which is outside the cone-of-influence, *No Value Available* is displayed in the waveform window.

So, you can give the command:

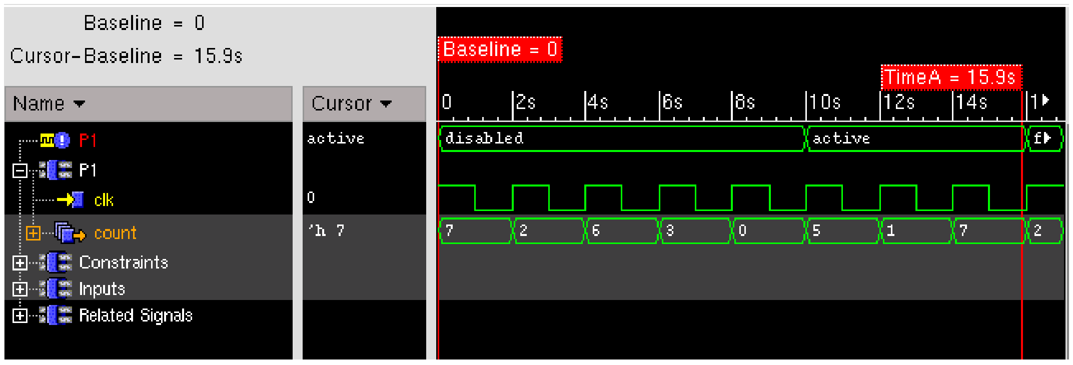
```
FormalVerifier> debug P1 -dynamic
```

that displays the following messages:

Running Simulation…

Waveform display launched.

and shows the following counter-example:



This time, when you try to add `verifcount`, you see the expected waveform.

Viewing Signals Outside the Cone of Influence

Incase of an assertion failure or a cover pass, you can generate and view the associated counter-example or a witness. This counter-example or a witness will enable you to identify the path for states within the COI, which led to the result as fail or pass.

If you want to generate and view the complete set of states of the design, you will need to view signals inside the COI and also signals that are outside the COI.

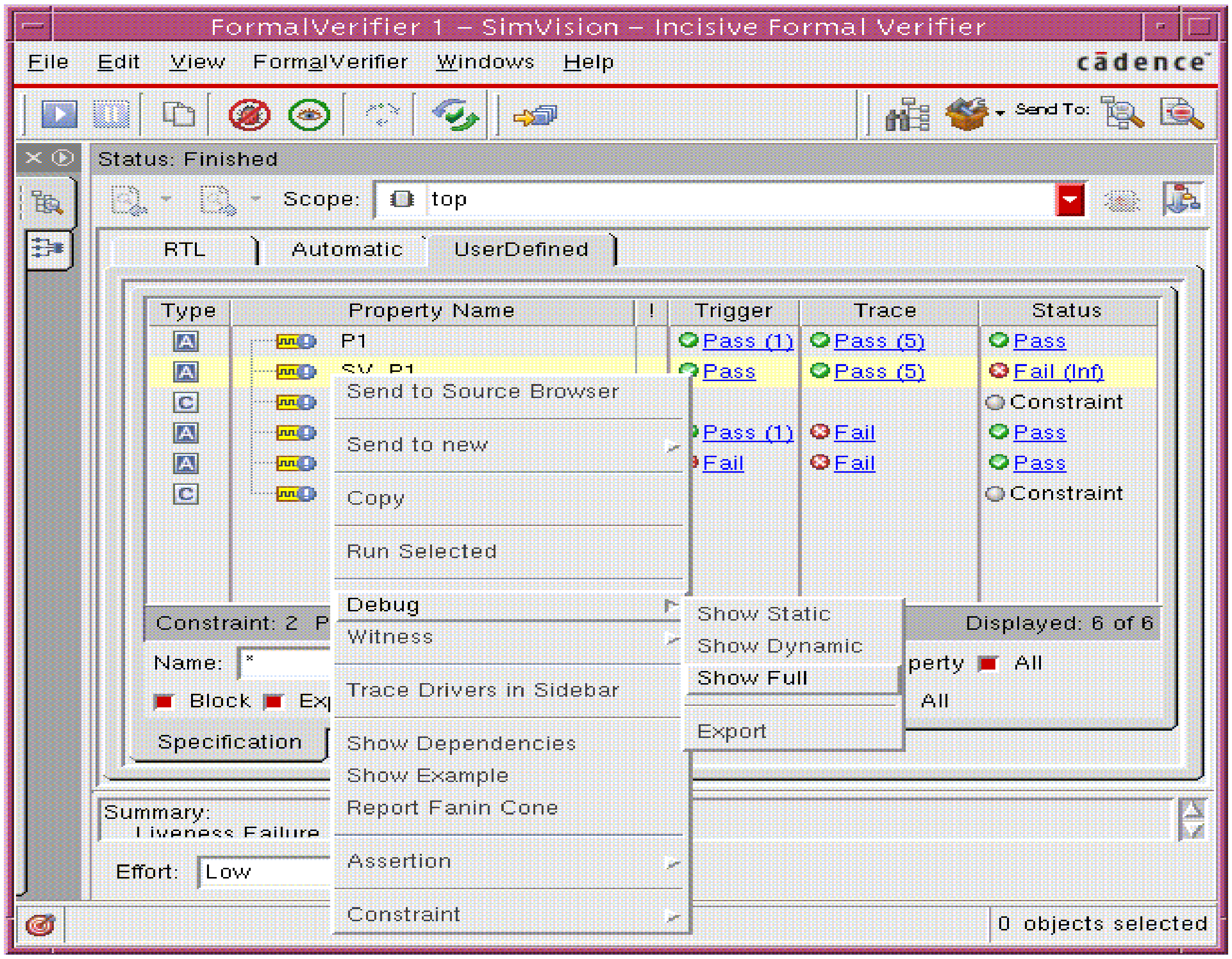
IFV enables you to view:

- Signals, which are part of the COI of an assertion or a cover.
- Signals, which are not in the COI of an assertion or cover. These signals however have an influence of other constraints of the design or on fanout of the COI of an assertion or cover.

To view these signals, generate a counter-example or a witness using the following command:

```
FormalVerifier> debug <assertionName> -full
FormalVerifier> witness <assertionName> -full
```

To view these signals in a counter-example waveform in the Formal Verifier GUI, click *View -> Debug -> Show Full*. Alternately, you can also right-click the property and select *Debug -> Show Full*, as shown below.



Status: Finished

Scope: top

Type	Property Name	!	Trigger	Trace	Status
A	P1		Pass (1)	Pass (5)	Pass
A	SV_P1		Pass	Pass (5)	Fail (Inf)
C					Constraint
A			Pass (1)	Fail	Pass
A			Fail	Fail	Pass
C					Constraint

Constraint: 2 P

Displayed: 6 of 6

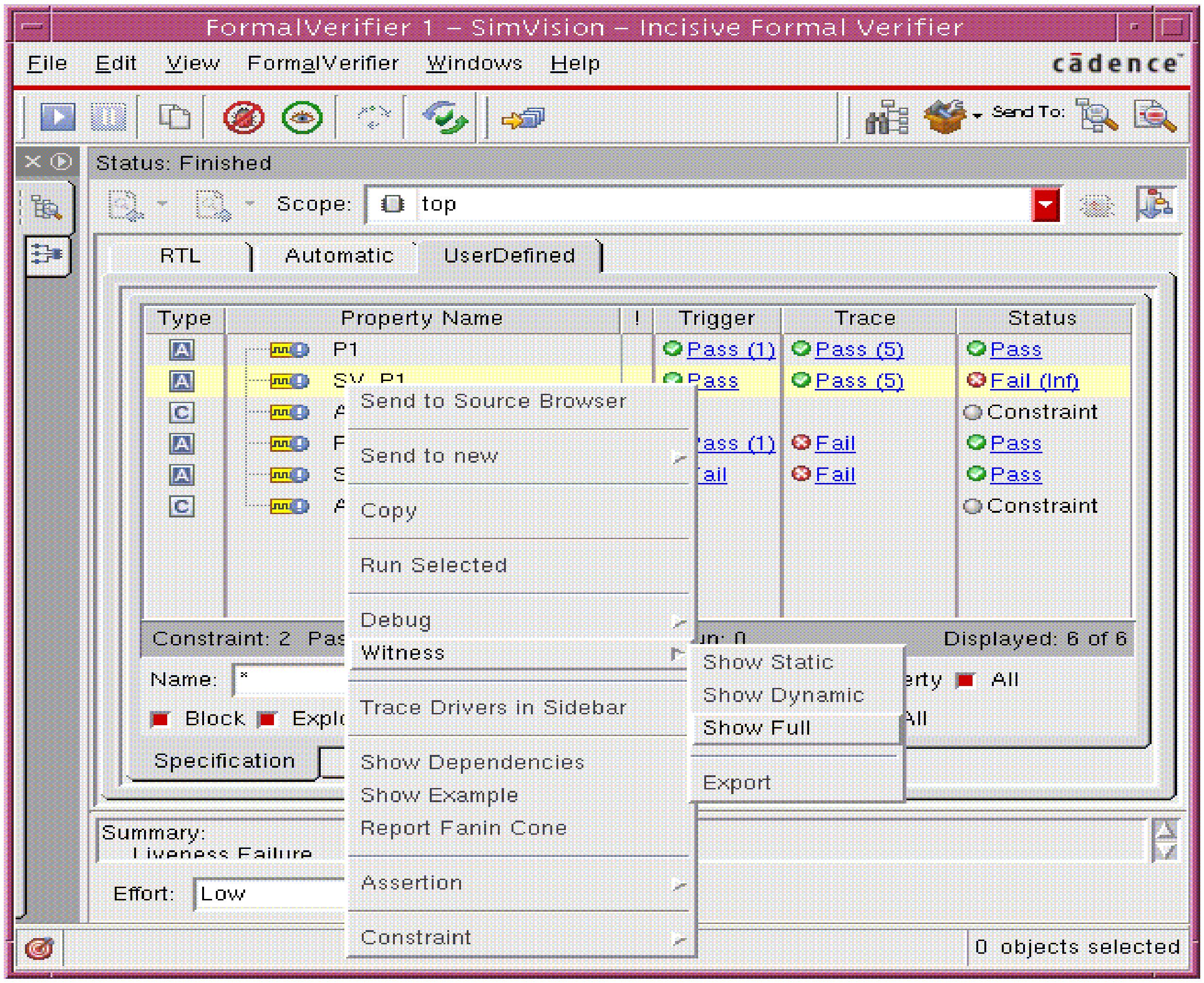
Name: *

Summary:
Liveness Failure

Effort: Low

0 objects selected

To view these signals in a witness waveform in the Formal Verifier GUI, click *View -> Witness -> Show Full*. Alternately, you can also right-click the property and select *Witness -> Show Full*, as shown below.



Generating the Counter-example in Batch Mode

You can generate and view the counter-example only by using the `debug` command from the `FormalVerifier>` prompt. You can also generate counter-examples in batch mode and view them later. This can be very useful if the counter-example has to be passed to a different engineer or team for detailed analysis.

You can generate the counter-example, save it in an SST database, and view it later using SimVision. To do these, you need to use the `-sstexport` option of the debug command as explained below:

- To generate the counter-example and save it in the SST database in a specified directory, give the following command:

```
debug <assertionName> -sstexport <dirName>
```

This command create a new directory named `dirName` in the current directory and saves the SST database in it. If the specified directory exists, the tool reports an error.

Consider the following example:

```
FormalVerifier> debug LIVE1 -sstexport NEW
```

This command creates the directory `NEW` and saves the counter-example of assertion `LIVE1` in SST database format in `NEW`.

- To generate the counter-example and save it in the SST database in an existing directory, give the following command:

  ```
  debug <assertionName> -sstexport <dirName> -overwrite
  ```

  This command is similar to the first command. However, it overwrites the existing directory `dirName` and saves the SST database in it.

  Consider the following example:

  ```
  FormalVerifier> debug LIVE1 -sstexport NEW -overwrite
  ```

  This command re-creates the directory `NEW` and saves the counter-example of assertion `LIVE1` in SST database format in `NEW`.

- To save the counter-example generated using event-based simulation, give the following command:

  ```
  debug <assertionName> -sstexport <dirName> -dynamic
  ```

  Consider the following example:

  ```
  FormalVerifier> debug LIVE1 -sstexport NEW -dynamic
  ```

  This command runs simulation on assertion `LIVE1`, creates the directory `NEW`, and saves the counter-example of assertion `LIVE1` in SST database format in `NEW`.

Viewing the Saved Counter-example

To view the saved counter-example, give the following command from the current working directory:

```
viewWaveform
     [-dbDirName <dbdirname>]
     [-snapshot <snapshotname>]
     [-cdslib <cds.lib>]
     [ -hdlvar <hdl.var>]]
```

Consider the following example:

```
FormaLVerifier> viewWaveform -dbDirName NEW -snapshot worklib.counter:v -
cdslib ./INCA_libs/ifv_snap.nc/sc_cds.lib -hdlvar ./INCA_libs/ifv_snap.nc/
sc_hdl.var
```

This command launches the counter-example, from the directory `NEW`, using the snapshot `worklib.counter:v`.

Note:

1. If the snapshot name is not specified, then the *Go To-Cause* feature is disabled.
2. If the directory name is not specified by using the `-dbDirName` option, then the `viewWavform` command has to be executed from the user-specified directory. That is, you will need to `cd` to the directory `NEW`, before giving the command.

Converting the SST database to the VCD database

You can convert the default SST database to the VCD database by using the utility `simvisdbutil`. To change the format, give the following command:

```
simvisdbutil -vcd <dbDirName>/cex.trn -output <dbDirName>/cex.vcd
```

Consider the following example:

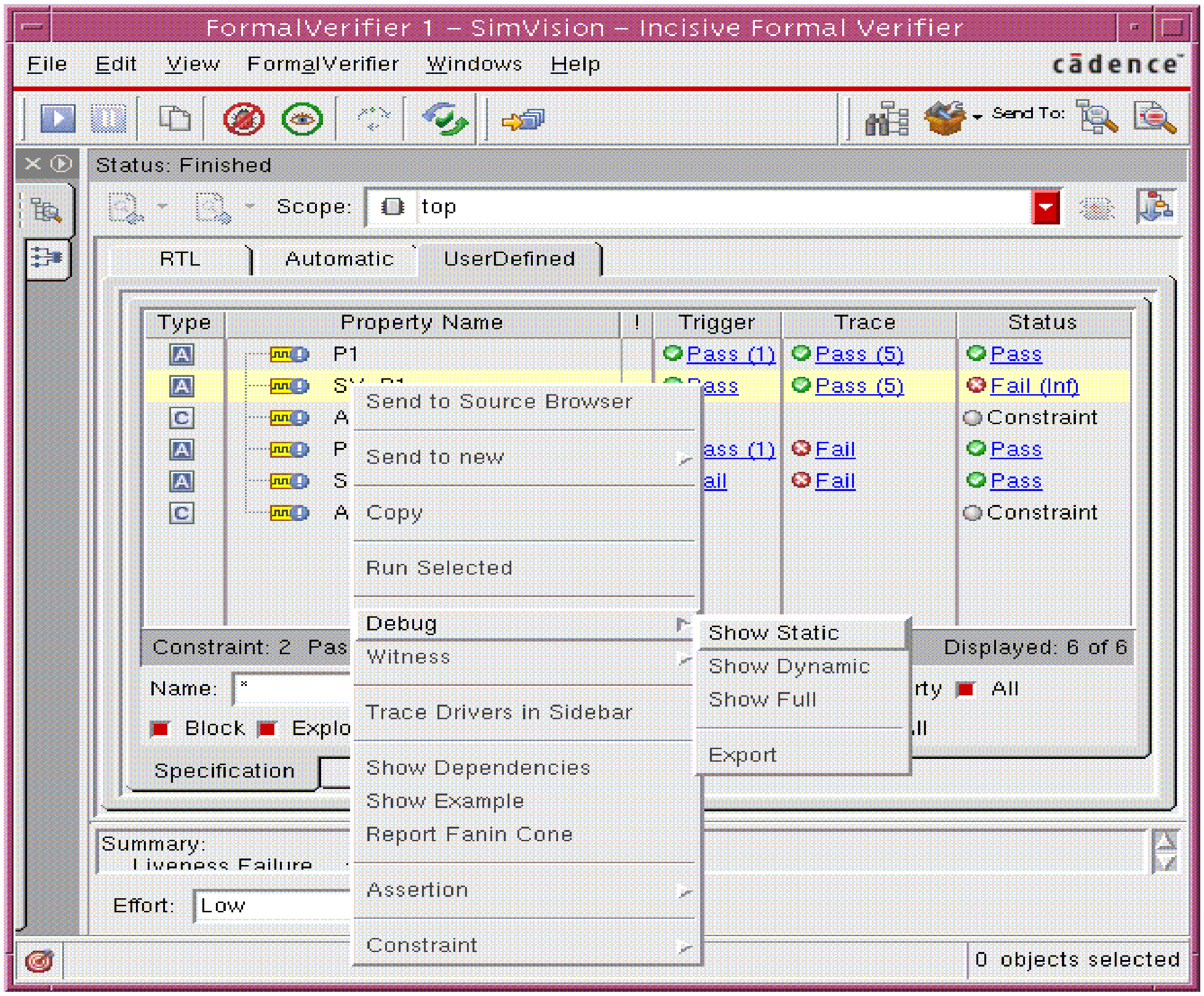
```
% simvisdbutil -vcd NEW/cex.trn -output NEW/cex.vcd
```

This command converts the SST database `cex.trn` in the directory `NEW` to the VCD database `cex.vcd` in the same directory `NEW`.

Note: Exit the tool before converting the database format.

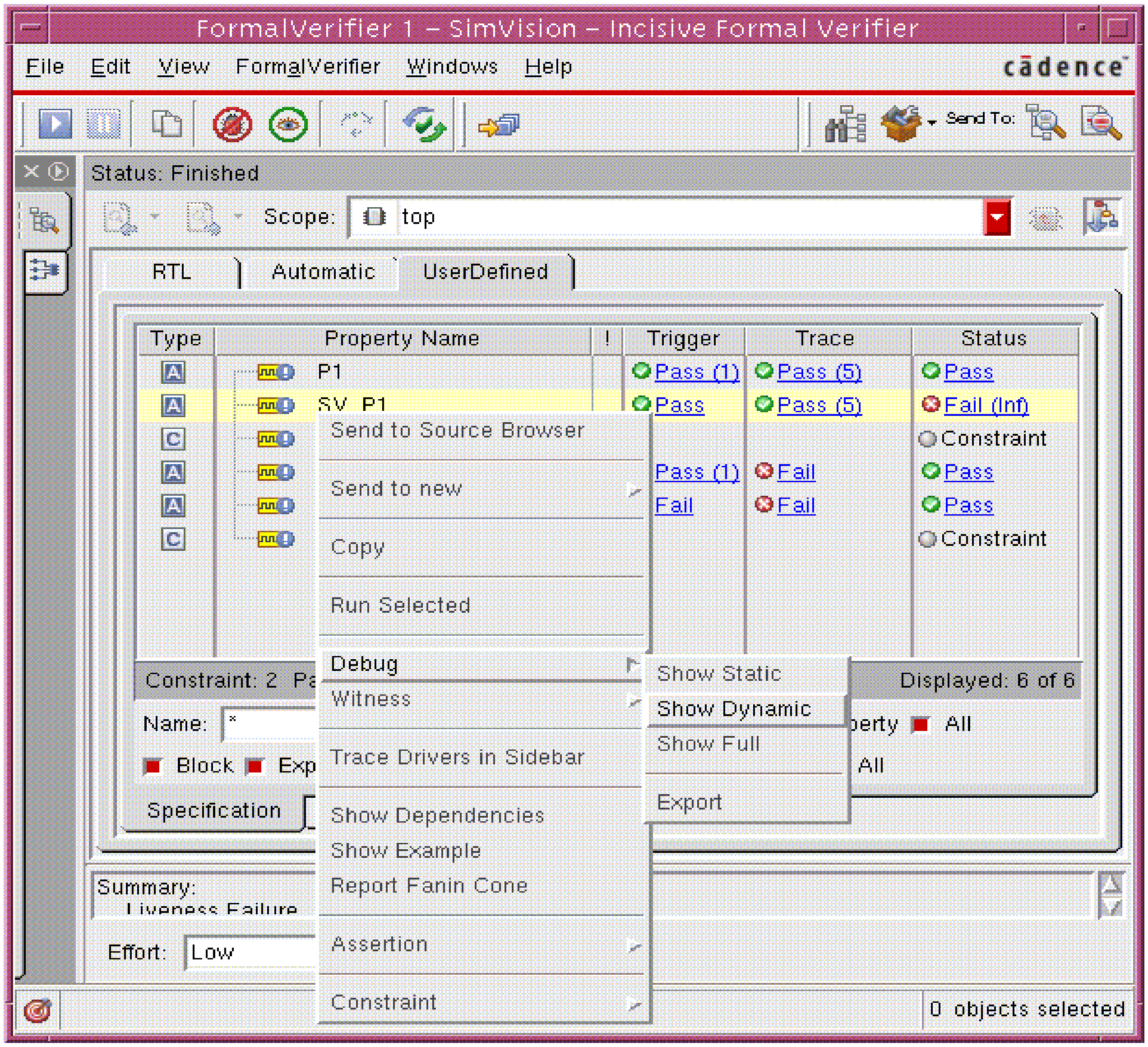
Exporting the Counter-example in the GUI Mode

To view a counter-example waveform in the Formal Verifier GUI, click *View -> Debug -> Show Static*. Alternately, you can also right-click the property and select *Debug -> Show Static*, as shown below.



Type	Property Name	!	Trigger	Trace	Status
A	P1		Pass (1)	Pass (5)	Pass
A	S…		Pass	Pass (5)	Fail (Inf)
C	A				Constraint
A	P		Pass (1)	Fail	Pass
A	S		Fail	Fail	Pass
C	A				Constraint

To display the waveform generated by internal simulation in the Formal Verifier GUI, click *View -> Debug -> Show Dynamic*. Alternately, you can also right-click the property and select *Debug -> Show Dynamic*, as shown below.



Status: Finished

Scope: top

Type	Property Name	!	Trigger	Trace	Status
A	P1		Pass (1)	Pass (5)	Pass
A	SV_P1		Pass	Pass (5)	Fail (Inf)
C					Constraint
A			Pass (1)	Fail	Pass
A			Fail	Fail	Pass
C					Constraint

Constraint: 2 Pa

Displayed: 6 of 6

Name: *

erty All

Block Exp

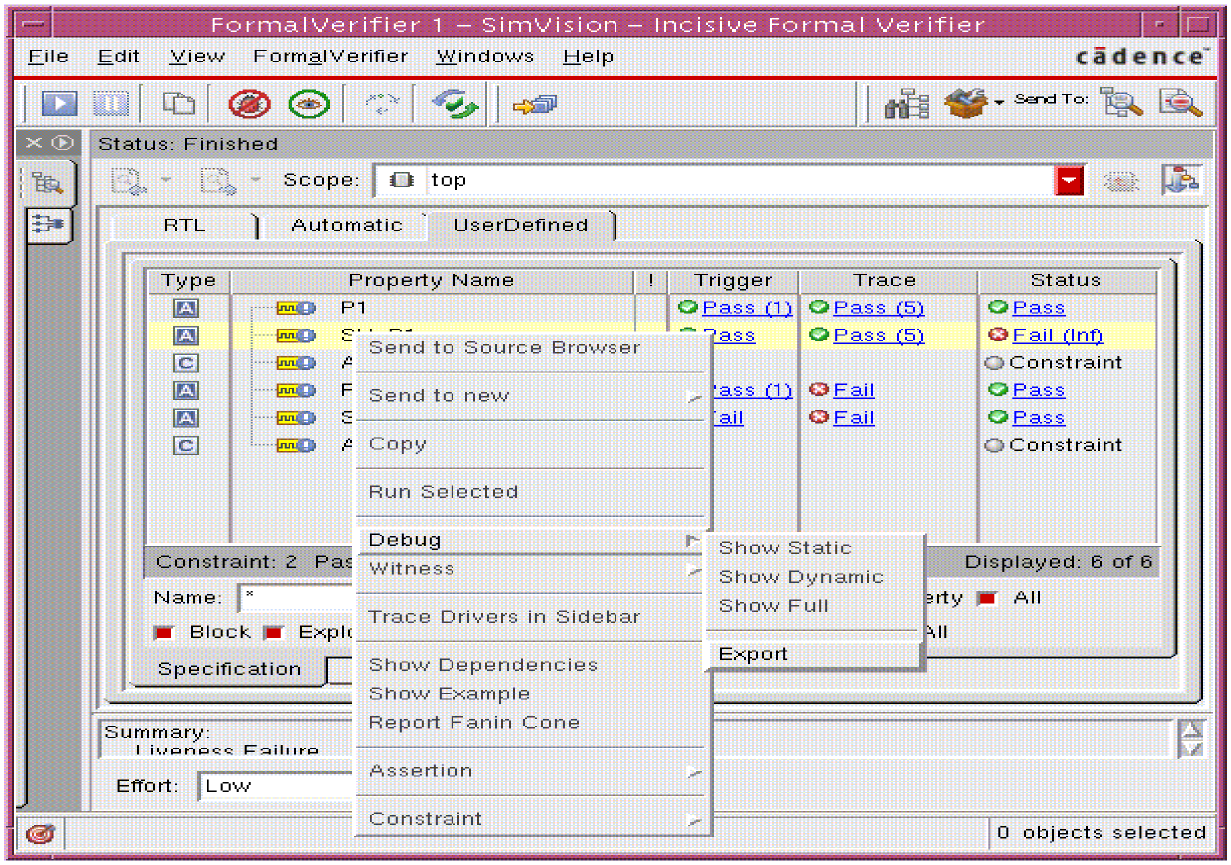
All

Summary:
Liveness Failure

Effort: Low

0 objects selected

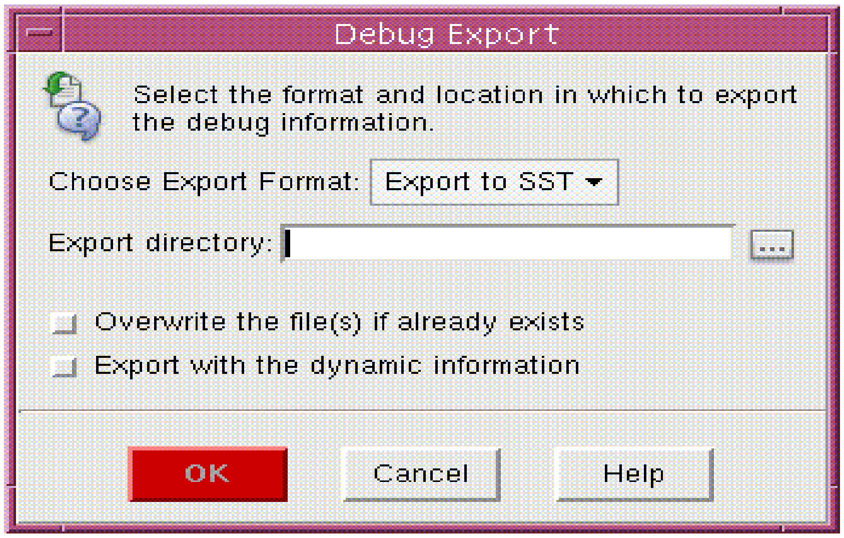
To export the counter-example in various formats in the Formal Verifier GUI, click *View-> Debug -> Export*. Alternately, you can also right-click the property and select *Debug -> Export*, as shown below.



Type	Property Name	!	Trigger	Trace	Status
A	P1		Pass (1)	Pass (5)	Pass
A	S...		Pass	Pass (5)	Fail (inf)
C	A...				Constraint
A	F...		Pass (1)	Fail	Pass
A	S...		Fail	Fail	Pass
C	A...				Constraint

Constraint: 2 Pas...
Name: *
Block Expl...
Specification
Summary:
Liveness Failure
Effort: Low

The *Debug Export* dialog box will be displayed as:


Debug Export

Select the format and location in which to export the debug information.

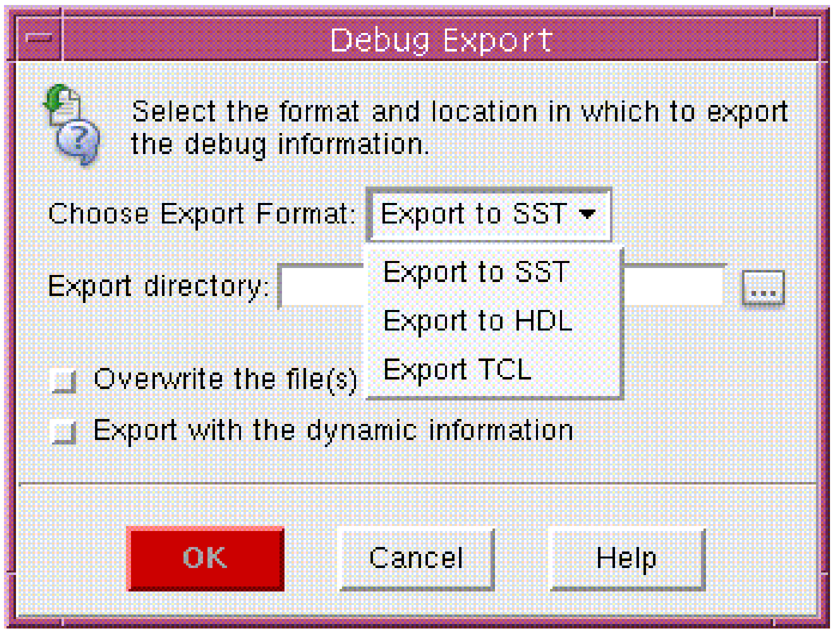
Choose Export Format: Export to SST

Export directory:

Overwrite the file(s) if already exists

Export with the dynamic information

Select the format and location to export the debug information, by selecting from the drop-down menu as shown below.


Debug Export

Select the format and location in which to export the debug information.

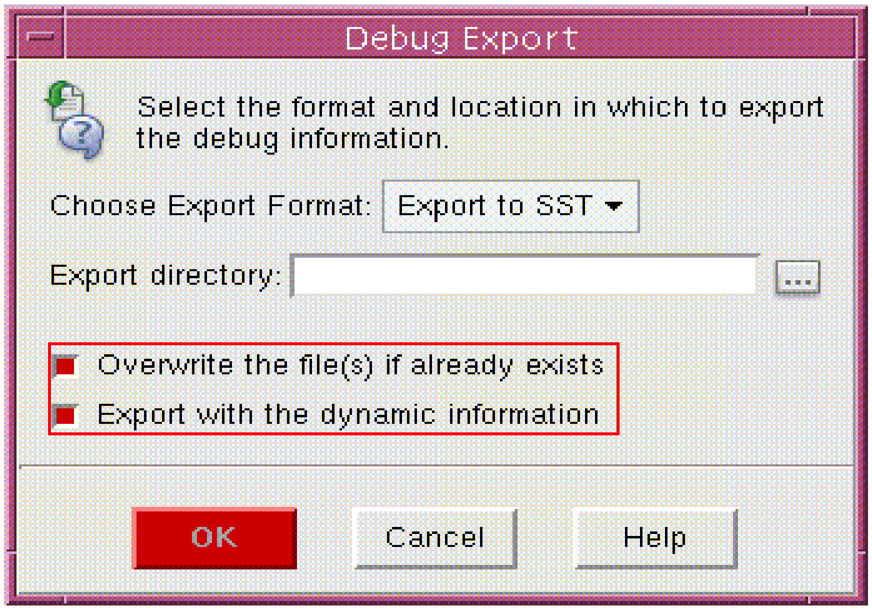
Choose Export Format: Export to SST

Export directory:

The three options specified are:

- Export to SST: Exports the counter-example as an SST database to the specified directory.
- Export to HDL: Generates a testbench for the specified assertion with the specified filename.
- Export TCL: Generates a NC Simulator tcl file that can be used for simulating the failure.

You can also appropriately select the highlighted options in the *Debug Export* dialog box.



Select the format and location in which to export the debug information.

Choose Export Format: Export to SST

Export directory:

- [✓] Overwrite the file(s) if already exists
- [✓] Export with the dynamic information

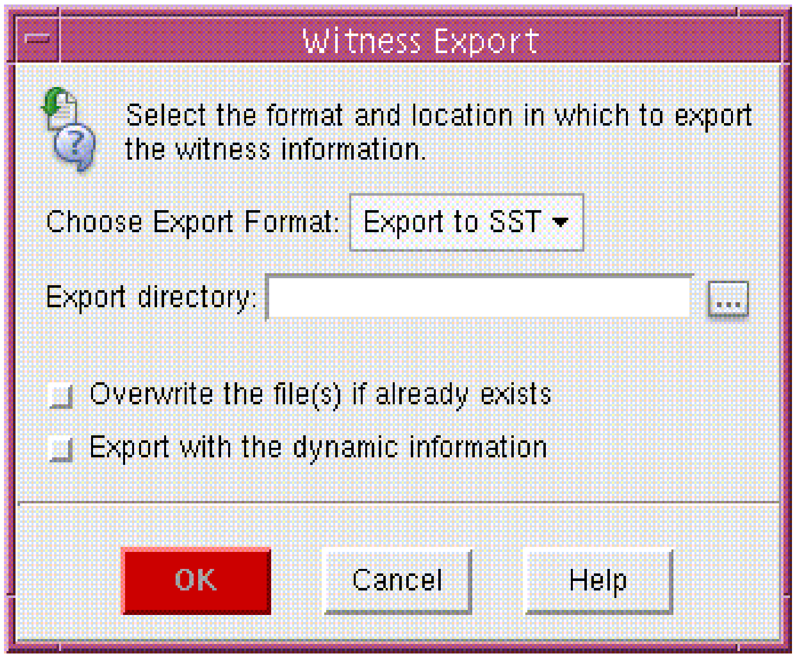
Exporting the Witness in the GUI Mode

To view a witness waveform in the Formal Verifier GUI, click *View -> Witness -> Show Static*. Alternately, you can also right-click the property and select *Witness -> Show Static*.

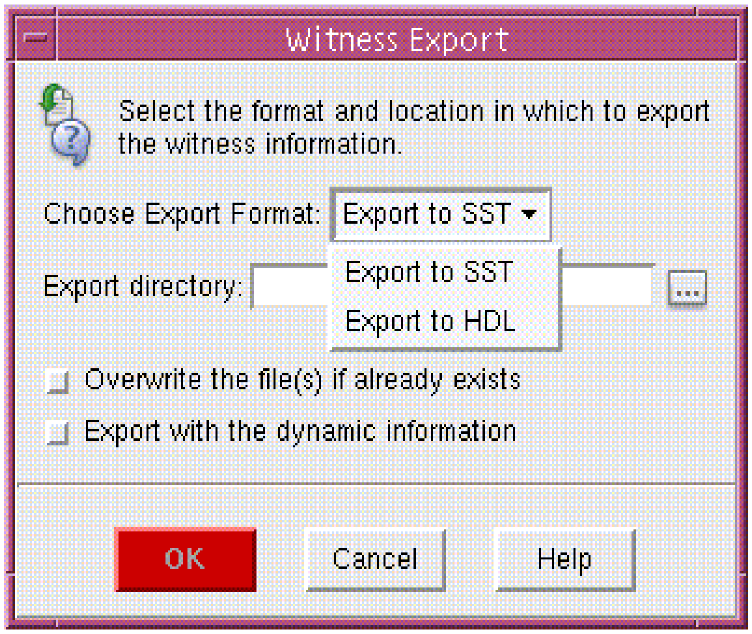
To display the witness waveform generated by internal simulation in the Formal Verifier GUI, click *View -> Witness -> Show Dynamic*. Alternately, you can also right-click the property and select *Witness -> Show Dynamic*.

To export the witness in various formats in the Formal Verifier GUI, click *View -> Witness -> Export*. Alternately, you can also right-click the property and select *Witness -> Export*.

The *Witness Export* dialog box will be displayed as:


Witness Export
Select the format and location in which to export the witness information.
Choose Export Format: Export to SST
Export directory:
...
Overwrite the file(s) if already exists
Export with the dynamic information
OK
Cancel
Help

Select the format and location to export the witness information, by selecting from the drop-down menu as shown below.


Witness Export
Select the format and location in which to export the witness information.
Choose Export Format: Export to SST
Export to SST
Export to HDL
Export directory:
...
Overwrite the file(s) if already exists
Export with the dynamic information
OK
Cancel
Help

The two options specified are:

- Export to SST: Exports the counter-example in SST database to the specified directory.
- Export to HDL: Generates the testbench for the specified assertion with the specified filename.

Note: Export TCL option is not available incase of Witness command.

You can also appropriately select the options specified in the *Witness Export* dialog box.

Exporting Waveform Data to a File

For complete information on this, refer to the section titled *Exporting a Database* in *Chapter 7*, of the *SimVision User Guide*.

Saving a SimVision Layout for Later Use

For complete information on this, refer to *Chapter 3*, titled *Saving and Restoring Your Debugging Environment* in the *SimVision User Guide*.

Similar Counterexamples

Formal Verifier does not verify an assertion if a similar one has already been verified. Consider an example of an assertion that is defined inside one module and a higher level design contains multiple instances of that module. Then, multiple instances of the assertion will exist and will form a set of similar assertions. All these assertions can be verified without running the formal engines if one of them can be verified in the context of the lower level module.

Due to this notion of similarity, only one counter-example is generated for a set of similar assertions. The counter-example is in the context of the assertion that was verified and found to be failing. When the `debug` command is issued on an assertion that is similar to an already failing one, Formal Verifier displays the counter-example based on the one assertion that was verified and issues the following warning:

*formalverifier: *W,WRNSCX: Displaying the counter-example which was generated during verification of the assertion "top.A1", and is similar to the counter-example of the assertion "top.A2"*

10

Automatic Formal Analysis

Assertions are usually written by engineers using a property specification language, such as PSL, OVL, or SVA. This allows the engineer tremendous flexibility to verify functional behavior. In addition to the ability to verify user-written assertions, Formal Verifier can automatically generate assertions by analyzing the RTL. These automatically generated assertions provide tremendous value because:

- They do not require any engineering time to create.
- Formal Verifier generates assertions consistently across the entire RTL.
- Many of the assertions that are generated would be very tedious for a user to write.
- Correctness of the assertions is guaranteed, which means less time spent in debugging.

Formal Verifier can automatically generate the following classes of assertions:

- Deadcode Checks

  Formal Verifier creates an assertion for every conditional branch or loop in the design. If one of these assertion fails, it means that the segment of code associated with that assertion can never be exercised with any legal set of stimuli. The purpose of these assertions is to identify `dead code` that is not exercisable under the given set of constraints. If the dead code is detected and eliminated early in the design cycle, it will result in lesser area on the silicon. Sometimes dead code indicates incorrect constraints. Finally, deadcode checks can be used in conjunction with traditional code coverage to assess whether specific code coverage holes are actually due to code that is simply not reachable.

- Synthesis Pragma Checks

  Formal Verifier creates assertions that validate whether synthesis pragmas actually hold true for a design. Synthesis pragmas that Formal Verifier understands include: `parallel_case`, `full_case`, `one_hot`, and `one_cold`.

- FSM Checks

  Finite State Machines (FSM) are used very frequently. Formal Verifier creates properties to check for the following:

    - State reachability
    - Transition reachability
    - Deadlocks

- Bus Checks

  Formal Verifier creates assertions to automatically detect the buses in the design. It identifies the signal that is driven by multiple drivers as bus. IFV creates properties to ensure that the bus adheres to some or all of the following checks:

    - At least one driver is driving the bus
    - Not more than one driver is driving the bus
    - No two drivers are driving the bus with conflicting values

- Xchecks

  Formal Verifier analyzes HDL designs for an X assigned to a signal. If there are any X assignments, IFV will create a check to see if these X assignments are reachable.

- Range Overflow Checks

  IFV analyzes HDL designs for indexed expressions. If there are indexed expressions, IFV creates a check to see if these expressions are indexed within the defined range. These created checks are termed as range overflow checks.

Note: Formal Verifier does not allow you to add automatically generated properties as constraints.

Deadcode Checks

A block is a statement or sequence of statements in Verilog or VHDL that execute as a set, one-after-another. Either none or all of the statements in a block are executed. Formal Verifier considers statements in `conditional`, `case`, `if-else`, and `loop` statements as a block and creates *existential* assertions that check whether the associated blocks can ever be executed. Refer to Chapter 7, “Developing and Using Properties,” for more information on *existential* properties.

If a block is determined to be unreachable, the status of the corresponding deadcode assertion is reported as *Fail*. Conversely, if a block is determined to be reachable, the status of the corresponding deadcode assertion is reported as *Pass*.

A deadcode assertion as an *existential* assertion implies that a *passing* deadcode assertion has a *witness* associated with it. The witness shows the sequence of inputs that are required to execute the corresponding block of code.

Failing deadcode assertions do not have witnesses. This can make debugging more difficult, because there is nothing specific to look at. Therefore, it is best to look at the RTL from which the deadcode property originated and try to determine the cause. To further isolate the signals that are responsible for the deadcode, you could write user-defined existential assertions to better understand why the deadcode assertion is failing. Consider the following code:

```
assign z = a && b && c ? x : y;
```

Assume that `deadcode_cond_assign_2` failed. Some existential assertions that might be helpful in assessing the reason for failure are:

```
// psl cover_cond_assign_2_a: cover {a};
// psl cover_cond_assign_2_b: cover {b};
// psl cover_cond_assign_2_c: cover {c};
// psl cover_cond_assign_2_ab: cover {a && b};
// psl cover_cond_assign_2_ac: cover {a && c};
// psl cover_cond_assign_2_bc: cover {b && c};
```

You can use the results of the above set of assertions to help narrow down the reasons for the failure of the original deadcode assertion.

Note: Deadcode properties will not be generated for sub modules, which do not have an output port.

Deadcode Property Naming Scheme

In general, the name of a deadcode property is of the following form:

```
deadcode_<type>_line_<line_number>
```

where:

1. `type` can take any one of the following values:

`if`	the deadcode property is generated for an `if-else` statement.
`loop`	the deadcode property is generated for a `for/while` (looping) statement.
`case_default`	the deadcode property is generated for the *default* branch of a `case` statement.

`case`	the deadcode property is generated for one of the *non-default* branches of the `case` statement.
`cond_assign`	the deadcode property is generated for `ternary operator` (`?`) in Verilog or `when-else` in VHDL.

2. `line_number` in the property name is the starting line number of the block for which the property has been generated.

If there are two deadcode properties of the same type and at the same line number, then the names of the two deadcode properties will be `deadcode_<type>_line_<line_number>` and `deadcode_<`*type*`>_line_<`*line_number*`>_<`*column_number*`>`.

Consider the following example:

```
1   module decoder ( out,in1, in2, cond);
2     input in1, in2, cond;
3     output out;
4
5   assign out = cond ? in1 : in2;
6
7   endmodule
```

In the above example, two deadcode properties are generated for the ternary operator on line `5` and their names are `deadcode_cond_assign_line_5` and `deadcode_cond_assign_line_5_28`. Here, `28` is the column number of the second branch (`in2`) of the ternary operator.

Types of Deadcode Properties

Formal Verifier generates the deadcode properties for the following conditional statement blocks:

Note: All the examples shown below are in Verilog. In VHDL, the corresponding statements can be used to construct the examples.

- **If-Else Statement**

Formal Verifier generates one deadcode property for every block within an if statement. Consider the following example:

```
1   if (sel)
2       out <= a;
3   else
4       out <= b;
```

Here, Formal Verifier generates two deadcode properties named `deadcode_if_line_2` and `deadcode_if_line_4` to check whether the statements on the corresponding lines will ever be executed. As the names suggest, property `deadcode_if_line_2` is created to check the statement on `line 2` and the property `deadcode_if_line_4` is generated to check the statement on `line 4`.

- **Case Statement**

  Formal Verifier generates one deadcode property for every branch in a case statement. Consider the following example:

```
1   case (sel)
2      2'b01 : out = a;
3      2'b10 : out = b;
4      default: out = c;
5   endcase
```

  Here, Formal Verifier generates three deadcode properties named `deadcode_case_line_2`, `deadcode_case_line_3`, and `deadcode_case_default_line_4` to check whether the statements in the corresponding branch will ever be executed.

- **Loop Statement**

  Formal Verifier generates one deadcode property for every loop (such as `for` and `while`). Consider the following example:

```
1   for (i=0; i<2; i=i+1) begin
2      o = o + 1;
3   end
```

  Here, Formal Verifier generates one deadcode property named `deadcode_loop_line_2`.

- **Ternary Operators**

  For the ternary operators in Verilog (`when-else` in VHDL), deadcode properties are generated to check whether the expressions in the branches of the condition of the ternary operator can ever be selected. Consider the following example:

```
1   assign q = cond1 ?
2       in1 + in2
3      : in1 - in2;
```

  Here, Formal Verifier generates two deadcode properties. The property `deadcode_cond_assign_line_2` to check whether `in1 + in2` will ever be selected by the ternary operator and the property `deadcode_cond_assign_line_3` to check whether i`n1 - in2` will ever be selected by the ternary operator.

  Note: To disable the generation of these properties, launch the tool with the `+disable_conditional_deadcode` option. For details of this command, refer to *Chapter 2, "The Single-step Method,"* of the *Formal Verifier Reference Manual*.

Synthesis Pragma Checks

Synthesis pragmas are comments that define some aspects of the functionality of an HDL design. Synthesis tools assume that these pragmas are correct and optimize the synthesized implementation accordingly.

Formal Verifier creates assertions that validate whether synthesis pragmas actually hold true for a design. The following Cadence, Synopsys, and Ambit synthesis pragmas are supported (for Verilog designs only):

- `one_hot`

  In the following examples of synthesis pragma, the entry in double quotes contains the names of the signals to be checked:

  ```
  // cadence one_hot "set, reset"
  // ambit synthesis one_hot "set, reset"
  // synopsys one_hot "set, reset"
  ```

  This means that the corresponding pragma property checks that only one of the signals, specified in double quotes (set or reset) can be *high* at any given point of time.

  The name of the generated property is `assert__one_hot_<`*`integer_value`*`>`.

- `one_cold`

  In the following example of synthesis pragma, the entry in double quotes contains the names of the signals to be checked:

  ```
  // cadence one_cold "set, reset"
  ```

  This means that the corresponding synthesis pragma property checks that only one of the signals, specified in double quotes (set or reset) can be *low* at any given point of time.

  The name of the generated property is `assert__one_cold_<`*`integer_value`*`>`.

- `full_case`

  The `full_case` synthesis pragma can be specified on a `case` statement. This means that for every legal value of selector of the case statement there will be a corresponding branch.

  Consider the following example with the `full_case` synthesis pragma:

  ```
  module decoder ( out,in);
    output [3:0] out;
        input [1:0] in;
        reg [3:0] out;
        wire [1:0] selector;

    assign selector = in;

  ```

```
  always @ (selector)
  begin
    case (selector)      // cadence full_case
      2'b01 : out = 4'h1;
      2'b10 : out = 4'h2;
      2'b11 : out = 4'h3;
    endcase
  end

endmodule
```

As per the `full_case` pragma, the valid value of the signal `selector` is `2'b01`, `2'b10`, or `2'b11`. The generated property checks that the value of the signal `selector` is always `2'b01`, `2'b10`, or `2'b11` and will fail when `selector` gets the value `2'b00`.

The name of the generated property is `assert__full_case_`<*`integer_value`*>.

- `parallel_case`

  The `parallel_case` synthesis pragma can be specified on a `case` statement. This means that for every legal value of selector of the case statement there will not be more than one branch.

  Consider the following example with the parallel_case synthesis pragma:

```
always @ (selector)
begin
       case (selector)      // cadence parallel_case
         cond1 : out = 4'h1;
         cond2 : out = 4'h2;
         cond3 : out = 4'h3;
       endcase
    end
```

  As per the parallel_case pragma, the corresponding synthesis pragma property checks that the value of the signal selector is not matched to more than one conditions, `cond1`, `cond2`, and `cond3`.

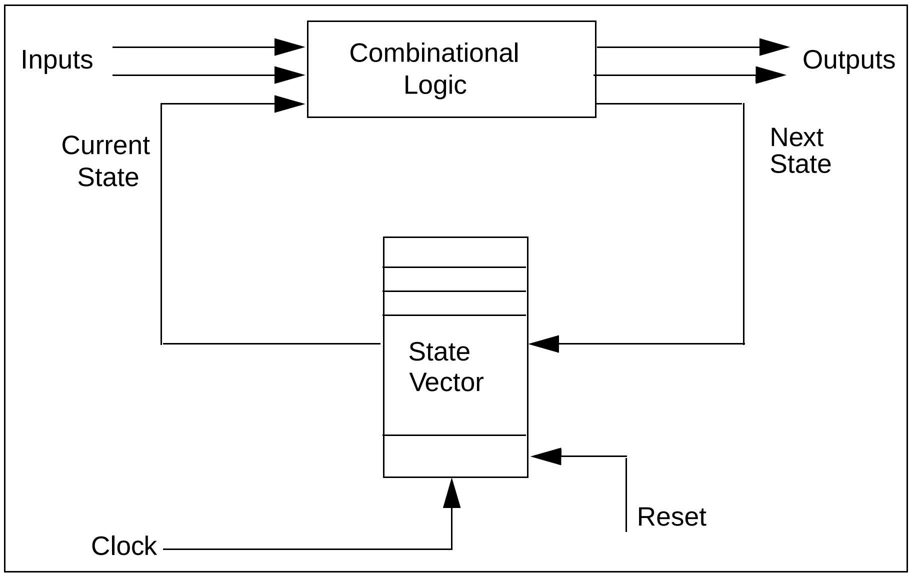
  The name of the generated property is `assert__parallel_case_`<*integer_value*>.

Note: `full_case` and `parallel_case` synthesis pragmas on `casex` or `casez` statements are not translated into properties.

FSM Checks

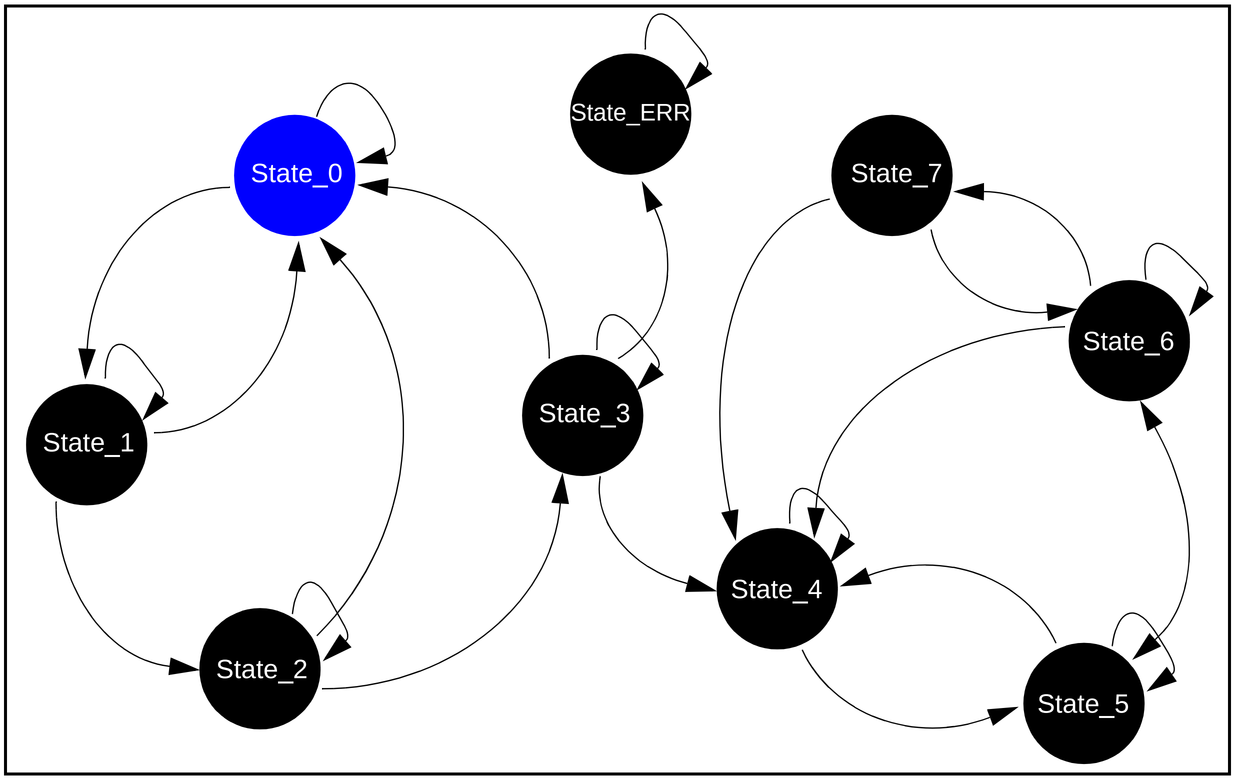
Designers frequently use Finite State Machines (FSMs) in the control logic and Formal Verifier applies formal techniques to verify standard FSMs and ensure that they are coded correctly and efficiently.

Synchronous finite-state controllers are a special class of *sequential logic*. The following figure shows the basic model of an FSM.



An FSM consists of a combinational block that computes the next state for the next cycle and the output values for the current clock cycle and memory elements that preserve the present-state of the machine.The next-state computation typically depends on the machine's present-state and input values. Output can be either *Mealy* or *Moore*. A *Mealy* output is a function of both present-state and inputs, while a *Moore* output is decoded solely from the present state of the machine.

Conceptually, an FSM can be viewed as a state graph, as shown below:



The FSM starts at `State_0` and depending on input values at each clock edge, transitions to other states and generates appropriate outputs. The diagram shows eight states. `State_0` is the reset state and each edge indicates the next-state transition and output value corresponding to the present-state and the input value.

Types of FSM Properties

Formal Verifier extracts the FSM properties automatically when the design is given to the tool. There are no commands to specify an FSM or modify an extracted FSM.

Formal Verifier automatically creates the following properties for FSMs:

- State Reachability

  The state reachability property checks if the state is reachable from the initial state of the FSM.

  The naming convention followed by automatically generated `reachability` property is as follows:

  ```
  fsm_<fsmname>_<statename>_reachable
  ```

where:

1. `fsmname` in the property name is the name of the FSM as mentioned in the design.
2. `statename` in the property name is the name of the state as mentioned in the design.

Consider the following example:

```
...
     case (fifo)                                  fsmname
          STATE_0 : begin                         statename
                        if(!state_sig[1])
                          fifo <= STATE_1;
                        else
                          fifo <= STATE_0;
                        end
....
```

In the above example, the *fsmname* is `fifo` and the *statenames* are `STATE_0`, `STATE_1`, and so on. Therefore, the *reachability* property for `STATE_0` will be named:

```
fsm_fifo_STATE_0_reachable
```

This property passes if the specified state can be reached from the initial state.

Note: If the tool is able to identify that a particular FSM state is not reachable, just based on the structural analysis, then the reachability property will be reported as a trivially false property. For example, in an FSM, the state register evaluated to a constant and all states except one was not reachable. For more information about trivial properties, refer to the section titled Trivial Property Checks on page 83 of this guide.

- State Transition

  The state transition property checks if it is possible to transition between the specified states.

  The naming convention followed by automatically generated `transition` property is as follows:

  `fsm_<`*`fsmname`*`>_<`*`fromstatename`*`>_to_<`*`tostatename`*`>_transition`

  Consider the following example:

```
...
     case (fifo)                                  fsmname
          STATE_0 : begin                         fromstatename
                        if(!state_sig[1])
                          fifo <= STATE_1;        tostatename
                        else
                          fifo <= STATE_0;        tostatename
                          end
    ...
```

In the above example, the *fsmname* is `fifo`, the *fromstatename* is `STATE_0` and the *tostatenames* are `STATE_0` and `STATE_1`. Therefore, the *transition* property from `STATE_0` to `STATE_1` will be named:

```
fsm_fifo_STATE_0_to_STATE_1_transition
```

This property passes if the transition from the specified state to the next sate is possible.

The goal is to get these properties to pass. However, these first two checks can generate a *witness* only if they pass and cannot generate a *counter-example* if they fail. For more information on this, refer to the section titled "Analyzing the Counter-example and Witness Waveforms".

- Deadlock State

  The deadlock state property checks that the FSM cannot take any outgoing transition from the current state under any possible input combination. This could be due to following reasons:

  - There is no outgoing edge from the current state, or
  - The condition of the outgoing edge cannot be met

  The naming convention for an automatically generated `deadlock` property is as follows:

  ```
  fsm_<fsmname>_<statename>_no_deadlock
  ```

  Consider the following example:

  ```
      ...
           case (fifo)                                   fsmname
      ...
      ...
      ...
                  STATE_ERR : begin                       statename
                                fifo <= STATE_ERR;
                              end
      ...
  ```

  In the above example, the *fsmname* is `fifo` and the *statename* is `STATE_ERR`. Therefore, the *deadlock* property of `STATE_ERR` will be named:

  ```
  fsm_fifo_STATE_ERR_no_deadlock
  ```

  This property passes if there is no deadlock in the state. In the example shown above, the property will fail and this means that a possible deadlock situation has been detected. After reaching `STATE_ERR`, the design will remain in that state.

  The third FSM check is very similar to a regular user-defined assertion in that you can get a counter-example if a property fails. However, they are potentially more difficult to debug. Consider the following examples:

Example 1

```
case(foo)
   FOO_STATE0 : next_state = FOO_STATE1;
   FOO_STATE1 : next_state = FOO_STATE1; // state has deadlock
endcase
```

There really is no way to debug this easily other than looking at the RTL.

Example 2

```
case(foo)
   FOO_STATE0 : next_state = FOO_STATE1;
   FOO_STATE1 : if (foo_input)
                   next_state = FOO_STATE0;
endcase
```

In this case, if `FOO_STATE1` has a deadlock, then nothing can be gained by debugging the failure by using the `debug` command. However, the interesting thing is that the transition check generated from `FOO_STATE1` to `FOO_STATE0` will also fail. This then is probably the best place to start debugging by using `cover` statements to track what `foo_input` is derived from and whether those items are reachable. Therefore, this is a very different debug strategy that you may choose to adopt.

Note: FSMs must be initialized to a valid initial state. If the FSMs are not correctly initialized, the FSM checks will not run, even if the properties are automatically generated by the tool.

Supported FSM Modeling Styles

FSMs can be modelled in multiple ways, but Formal Verifier recognizes FSMs that follow standard FSM modeling styles. Formal Verifier automatically generates properties for the following FSM modeling styles only:

- **One always Block/Single Process Modeling Style**

  One always block or single process modeling style is the most common modeling style where one sequential `always` block is used to code the FSM. The entire logic is described in a single process sensitive to the edge of clock and asynchronous resets. All signals being assigned using this style are registered signals.

  In this modeling style, one unique signal qualifies as the present state signal. The process is sensitive to appropriate edge of clock and asynchronous resets. To model reset priorities, this process should have a single `if` block where the last `else` defines the state machine behavior after all the resets were applied. Asynchronous resets are specified first, their priorities are specified by the order in which they are specified.

  In this modeling style, the present state and the next state are combined and there is a single process block that determines the state.

An example of this modeling style guide is given below:

```
module FSM1
(output reg grant,
input dly, done, req, clk, reset);

parameter [1:0] IDLE = 2’d0,
BUSY = 2’d1,
WAIT = 2’d2,
FREE = 2’d3;
reg [1:0] state;

always @(posedge clk or posedge reset)
  if (reset) begin
         state <= IDLE;
         grant <= 1’b0;
  end
  else begin
         state <= 2’bx;
         grant <= 1’b0;
         case (state)
              IDLE: if (req) begin
                         state <= BUSY;
                         grant <= 1’b1;
                      end
                      else

                         state <= IDLE;
              BUSY: if (!done) begin
                         state <= BUSY;
                         grant <= 1’b1;
                      end
                      else if ( dly) begin
                         state <= WAIT;
                         grant <= 1’b1;
                      end
                      else
                        state <= FREE;

              WAIT: if ( dly) begin
                         state <= WAIT;
                         grant <= 1’b1;
                      end
                      else
                         state <= FREE;

              FREE: if (req) begin
                         state <= BUSY;
                         grant <= 1’b1;
                      end
                      else
                         state <= IDLE;
         endcase
  end
endmodule
```

The below mentioned example, a new style of FSM, is coded in One always Block/Single Process Modeling using `if-else` statements in place of `case` statement.

```
module top(input clk, rst, output reg [1:0] state);
```

```
parameter RESET = 2'b00, RUN = 2'b11, STOP = 2'b10, HOLD = 2'b01;
    always @( posedge clk or posedge rst )
    begin
    if ( rst )
        state <= RESET;
    else
             if(state == RESET)
        state =  RUN  ;
         else if(state == RUN)
        state =  STOP;
         else if( state == STOP)
        state = HOLD ;
    end
endmodule
```

- **Two always Block/Two Process Modeling Style**

  This is one of the best and most commonly used coding styles to code FSMs. This uses two `always` blocks, one for the sequential state register and one for the combinational next-state and combinational output logic.

  In this modeling style, the process describing the sequential logic (flip-flops) is sensitive to the scalar clock edge event and all asynchronous resets (if any). The combinational process is sensitive to all signals, including the state register, that are read in the combinational process. The two-process model should have a unique signal that is computed from the combinational block as the next-state and a present-state signal that is assigned in a sequential block from the next-state signal.

  Note: Complex clock expressions are not supported.

  In a two process modeling style, there is a process block for the present state and there is another process block for the next state.

  An example of this style is given below:

```
module FSM
(output reg grant,
input dly, done, req, clk, reset);
parameter [1:0] IDLE = 2’b00,
BUSY = 2’b01,
WAIT = 2’b10,
FREE = 2’b11;
reg [1:0] state, next;

always @(posedge clk or posedge reset)
  if (reset)
     state <= IDLE;
  else
     state <= next;
```

```
always @(state or dly or done or req)
  begin
    next = 2'bx;
    grant = 1'b0;
    case (state)
         IDLE: if (req)
                 next = BUSY;
               else
                 next = IDLE;

         BUSY: begin
                 grant = 1'b1;
                 if (!done)
                    next = BUSY;
                 else if ( dly)
                    next = WAIT;
                 else
                    next = FREE;
               end

         WAIT: begin
                 grant = 1'b1;
                 if (!dly)
                    next = FREE;
                 else
                    next = WAIT;
               end

         FREE: if (req)
                 next = BUSY;
               else
                 next = IDLE;
    endcase
  end
endmodule
```

The below mentioned example, a new style of FSM, is coded in Two always Block/Two Process Modeling Style where Combinational block uses `If-else` statements instead of `case` statement.

```
module top(input clk, rst, output reg [1:0] current_state);
parameter RESET = 2'b00, RUN = 2'b11, STOP = 2'b10, HOLD = 2'b01;
    reg    [1:0]  next_state;
    always @( posedge clk or posedge rst )
    begin
    if ( rst )
        current_state <= RESET;
    else
        current_state <= next_state;
    end
    always @( current_state )
    begin
    if ( current_state == RESET  )
        next_state = RUN;
    else if ( current_state == RUN )
        next_state = STOP;
    else if ( current_state == STOP )
        next_state = HOLD;
    else if ( current_state == HOLD )
        next_state = HOLD;
    end
endmodule
```

- **One Always Block/Single Process with Next State in Conditional Wire Assignment**

In this style, FSM is coded in One always Block/Single Process Modeling using conditional assignment statements. An example of this style is given below:

```
module top(input clk, rst, output reg [1:0] state);
parameter RESET = 2'b00, RUN = 2'b11, STOP = 2'b10, HOLD = 2'b01;
    always @( posedge clk or posedge rst )
    begin
    if ( rst )
        state <= RESET;
    else
        state <= state == RESET? RUN  :
            state == RUN ? STOP   :
         state == STOP ? HOLD  : state ;
    end
endmodule
```

In the below mentioned example, the sequential block is coded in one always/Process Block and combinational block is coded in one conditional assignment statement.

```
module top(input clk, rst, output reg [1:0] current_state);
parameter RESET = 2'b00, RUN = 2'b11, STOP = 2'b10, HOLD = 2'b01;
reg[1:0] next_state;
    always @( posedge clk or posedge rst ) begin
        if ( rst )
    current_state <= RESET;
        else
    current_state <= next_state;
    end
assign next_state = current_state == RESET? RUN  :
    current_state == RUN? STOP   :
```

```
    current_state == STOP? HOLD   : current_state ;
endmodule
```

■ **One-hot Encoding Style**

In one-hot encoded style, states are defined using parameter declarations and their values are the bit position in the state register (which is a “one” when the FSM is in that particular state). The next-state computation makes the bit corresponding to the next state into a one using the abstract state names as index for the bit position. Following code is an example of one-hot encoded style.

```
module example (a, rst, clock);
input a, rst, clock;
parameter [1:0]
    S0=2’b00,
    S1=2’b01,
    S2=2’b10;
reg [2:0] present_state, next_state;
always @ (posedge clock or posedge rst)
begin
    if (rst)
        begin
           present_state = 0;
           present_state[S0] = 1’b1;
        end
    else
        present_state = next_state;
end
always @ (present_state or a)
begin
    next_state = 0;
    case (1’b1)
       present_state[S0]:
          begin
            if (a == 1’b1)
               next_state[S1] = 1’b1;
            else
               next_state [S0] = 1’b1;
          end
       present_state[S1]:
          begin
            casex (a)
              1’b1:
              begin
                 next_state[S2] = 1’b1;
              end
              default:
              begin
                 next_state[S0] = 1’b1;
              end
            endcase
          end
       present_state[S2]:
          begin
            if (a == 1’b1)
              next_state[S2] = 1’b1;
            if (a == 1’b0)
              next_state[S0] = 1’b1;
          end
    endcase
end
endmodule
```

Coding Guidelines

To ensure that the tool is able to generate FSM properties for the design, you should consider the following guidelines, while coding for FSMs:

- Reset block should set all the bits of the state register. For `N` bit state register, all the `N` bits should be appropriately set inside the reset block, as shown in the code below.

  ```
  if (rst)
     begin
       present_state = 0;
       present_state[S0] = 1'b1;
     end
  ```

- State vector (`P_S`) and next state vector (`N_S`) should not be assigned as part or bit select.
- The tool generates internal state names in the format `State_<state_value>` if you:
  - Do not define FSM case tags as parameters.
  - Mix use of constants/variables with parameters as case tags.
- If the FSM next state computation logic includes incomplete branches in Verilog, it is recommended that you use `//synopsys full_case` and `//synopsys parallel_case` pragmas and compile the design with `-pragma` option to ncvlog.
- State transitions are ignored for variable assignments to the state register.
- Modules with FSMs must not include any SystemVerilog construct that cannot be handled by the FSM extraction engine.

Unsupported Modeling Styles

Currently, FSM extraction is not supported in following scenarios:

- **FSM is coded as nested case blocks**

  FSM extraction is not supported if FSM has been coded as case blocks inside another case block or `if/else` block. Consider the following example:

  ```
  case (control)
      1'b0:
          case (state)
             < STATE ASSIGNMENTS>
      1'b1:
  ```

```
        case (state)
           < STATE ASSIGNMENTS>
```

Incisive Comprehensive Coverage auto extraction will work only if there is one top level case statement with selector as the state register. Consider the following revised statements:

```
case ( state )
            S0 : state <= (control)  ?  <STATE ASSIGNMENT>;
            S1 : state <= (control)  ?  <STATE ASSIGNMENT>;
            S2 : state <= (control)  ?  <STATE ASSIGNMENT>;
```

- **FSM is coded in multiple sequential if/elseif/else blocks**

  FSM extraction is not supported if FSM is coded in multiple sequential `if`/`elseif`/`else` blocks, that check for instance input signals instead of present state. This is similar to `case` block within a `case` block as discussed in the above point. The following code demonstrates the use of multiple sequential if/elseif/else blocks.

```
module DUT ( KEYS, BRAKE, ACCELERATE, SPEED );
input KEYS, BRAKE, ACCELERATE;
output [1:0] SPEED;
reg [2:0] SPEED;
parameter
      Stop=2’b00,
      Slow=2’b01,
      Medium=2’b10,
      Fast=2’b11;
wire CLOCK;
always @(posedge CLOCK or negedge KEYS)
    begin:FSM1
        if(!KEYS)
            SPEED=Stop;
        else if(ACCELERATE)
            case(SPEED) // synopsys full_case
                Stop:SPEED = Slow;
                Slow:SPEED = Medium;
                Medium:SPEED = Fast;
                Fast:SPEED = Fast;
            endcase
        else if(BRAKE)
            case(SPEED) // synopsys full_case
                Stop:SPEED = Stop;
                Slow:SPEED = Stop;
                Medium: SPEED = Slow;
                Fast:SPEED = Medium;
            endcase
        else
            SPEED = SPEED;
    end
endmodule
```

- **FSM extraction for a single-bit FSM is not supported:**

  The following code illustrates this.

```
    always @(posedge CLK or negedge RST)
    begin
        if (RST == 1’b0)
```

```
              begin
                 i_count <= 1'b0 ;
                end
            else
                begin
                   case (i_count)
                   1'b0: i_count <= 1'b1;
                   1'b1 : i_count <= 1'b0;
                   endcase
                end
      end
```

In case of a single-bit FSM, both true and false tags should be explicitly specified and default case tag should not be used.

- **Combinational or sequential always block modeled using module instantiation**

  The combinational or sequential always block is defined within a module and is instantiated within the FSM design. An example of this style is given below:

```
....
....
always @ (posedge clk or posedge reset)
   if(reset)
      state <= 0;
   else
      state <= next_state;
      m1 i1 (next_state, state);
endmodule

module m1 (next_state, state);
input [1:0] state;
output [1:0] next_state;
reg [1:0] next_state;

always @(state)
   case(state)
     0 : next_state = 1;
     1 : next_state = 2;
     2 : next_state = 3;
     3 : next_state =4;
   endcase
endmodule
```

- **Concat operator for state assignment**

  Concat operator is used for state assignment. An example of this style is given below:

```
...
if (rst)
{state[2],state[1],state[0]} <= 3'b000;
...
or consider this example:
case(state)
   3'b000: begin
   next = 3'b100;
```

```
if (!req1) {next[2],next[1],next[0]} = 3’b000;
....
```

- **Concat operator in case selector**

  In the combinational always block, concat operator is used. An example of this style is given below:

```
    ...
    case ({state[2],state[1],state[0]})
    3’b000 : next_state = 3’b001;
    ...
```

- **Non-blocking assignment within the combinational always block**

  In the combinational always block, non-blocking assignment is used. An example of this style is given below:

```
    always @ (cState or IOR or IOW or ALE or WE_m or RE_m)
      begin
        case (cState)
          S0: begin
             Ald = 0;
             if (ALE)
               nState = S1;
             else
               nState = S0;
             end

          S1 : begin
                 Ald = 1;
                 nState <= S2;
               end

          S2 : begin
                 Ald = 1;
    ....
    ...
```

  or consider another example:

```
    always @(state or req1 or req2 or req3 or req4)
       begin
         next <= 3’bxxx;
         err = 0; n_ack1 = 1;
        ack2 = 0; ack3 = 0; ack4 = 0;
         case(state)
              3’b000: begin
                   next = 3’b100;
                   if (!req1) next = 3’b000;
                   if (req1 & req2) next = 3’b001;
                   if (req1 & !req2 & req3) next = 3’b010;
              end
    ....
    ....
```

- **Part-select in case selector**

Part select is used in the case selector within the combinational always block. An example of this style is given below:

```
always @(state)
   begin
     case(state[1:2])
       3: case(state[3:4])
       3: nstate = 1;
       2: nstate = 2;
       1: nstate = 3;
       0: nstate = 4;
     endcase

     2: case(state[3:4])
     3: nstate = 1;
     2: nstate = 2;
     1: nstate = 3;
     0: nstate = 4;
     endcase
....
....
```

- **State assignment through a for loop**

  State assignment is done through a for-loop. An example of this style is given below:

```
    ...
    3'b100: begin
        for (i = 0; i < 3; i = i +1)
            next[i] = 0;
    ...
```

- **Combinational block using continuous assignment**

  The combinational always block is implemented through continuous assignment. An example of this style is given below:

```
    assign next = (req[0])?(3'b001)
        :(req[1])?(3'b010)
        :(req[2])?(3'b011)
       :(req[3])?(3'b100):(3'b000);.
```

Bus Checks

In hardware designs, a bus is a subsystem that transfers data between different components of the design. Unlike a point-to-point connection, a bus can logically connect several components over the same set of wires.

To simplify the verification of bus systems, IFV automatically detects the buses in the design. It identifies the signal that is driven by multiple drivers as a bus. As multiple drivers are driving the bus, it is a challenge to verify the bus system. Depending on the kind of design, it is important to ensure that the bus adheres to the following checks:

- At least one driver is driving the bus
- Not more than one driver is driving the bus
- No two drivers are driving the bus with conflicting values

Consider the following examples:

Example 1:

```
1   module test1(in1, in2, c1, c2, out1);
2   input  in1, in2;
3   input c1, c2;
4   output out1;
5   assign out1 = c1 ? in1 : 1’bz;
6   assign out1 = c2 ? in2 : 1’bz;
7   endmodule
```

In this example, the output `out1` is driven by two continuous assignments in lines 5 and 6. Therefore,the signal `out1` will be recognized as a bus.

Example 2:

```
module test2(clk1, clk2, in1, in2, out1);
output out1;
input clk1, clk2;
input in1, in2;
reg out1;
always @(clk1)
if (clk1)
begin
    if (clk2)
    begin
    if (in1 === in2)
    out1 = in1;
    else
    out1 = 1’bx;
    end
    else
    out1 = in1;
end


always @(clk2)
if (clk2)
begin
    if (clk1)
    begin
    if (in1 === in2)
    out1 = in1;
    else
    out1 = 1’bx;
    end
    else
    out1 = in2;
end
endmodule
```

In this example, there are two always blocks and `out1` is being driven in both these blocks. Therefore, IFV will recognize `out1` as BUS.

Types of Bus Checks

IFV performs the following types of checks on a bus:

- Floating bus: This check determines if the bus can ever have no active driver.
- Multiple drivers: This check determines if the bus can ever have multiple active drivers.
- Contention: This check determines if the bus can ever have multiple active drivers with conflicting data.

Bus Check Property Naming Scheme

For each of the bus checks, IFV creates a property. The naming scheme of the generated properties is based on the type of bus in the design, that is, scalar and vector.

Considering the bus name as `data`, the names of the automatically generated properties for the scalar bus are of the following form:

Type of Check	**Name of the Property**	**Example**
Floating	bus_<*bus_name*>_floating	bus_data_floating
Multiple drivers	bus_<*bus_name*>_muldrv	bus_data_muldrv
Contention	bus_<*bus_name*>_contention	bus_data_contention

For a vector bus, the properties are created for every bit. As an example, for a vector bus `data [3:0]`, the names of these properties are of the following form:

Type of Check	Name of the Property	Example
Floating	bus_<*bus_name*>_<*bit_number*>_floating	bus_data_0_floating bus_data_1_floating bus_data_2_floating bus_data_3_floating

Type of Check	Name of the Property	Example
Multiple drivers	bus_<*bus_name*>_<*bit_number*>_muldrv	bus_data_0_muldrv
		bus_data_1_muldrv
		bus_data_2_muldrv
		bus_data_3_muldrv
Contention	bus_<*bus_name*>_<*bit_number*>_contention	bus_data_0_contention
		bus_data_1_contention
		bus_data_2_contention
		bus_data_3_contention

Understanding the Results of Bus Checks

During and after the verification of the bus check assertions, you might encounter the following results:

Type of Check	**Pass**	**Fail**
Floating	If a floating assertion passes, then it means that at least one driver is always driving the bus.	If a floating assertion fails, then it means that in some scenarios. the bus can be floating.
Multiple drivers	If a multiple driven assertion passes, then it means that the bus is never driven by multiple drivers.	If a muldrv assertion fails, then it means that in some scenarios, the bus can be multiply-driven.
Contention	If a bus contention assertion passes, then it means that the bus can be driven by multiple drivers but never by drivers with conflicting data.	If a bus contention assertion fails, then it means that in some scenarios, the bus can have contention.

Consider the following example for a scalar bus:

```
module top(in1, in2, in3, c1, c2, c3, out1);
input in1, in2, in3; input c1, c2, c3;
```

```
output out1;
// psl CS1 : assume always (onehot({c1, c2, c3}));
// psl CS2 : assume always(in2 == in3);
// psl CS3 : assume always(in2 == in1);
assign out1 = c1 ? in1 : 1'b1z;
M1 I1(in2, in3, c2, c3, out1);
endmodule

module M1(in2, in3, c2, c3, out1);
input in2, in3;
input c2, c3;
output out1;
assign out1 = c2 ? in2 : 1'b1z;
assign out1 = c3 ? in3 : 1'b1z;
endmodule
```

In the above example, the bus `out1` is connected with three drivers. Let us consider, the constraint CS1 is enabled and constraints CS2 and CS3 are disabled. CS1 puts a constraint on signals, c1, c2, and c3 that at any point of time, only one of these signals will be active and the other two will be inactive. Considering the constraint CS1, you will encounter the following results for different bus checks:

Type of Check	Result	Reason
Floating	The floating check with constraint CS1 will give the result as PASS.	The result is PASS because at any given point of time, one of the inputs, c1, c2, or c3 will always be active. That means, one driver will always be driving the bus.
Multiple drivers	The multiple drivers check with constraint CS1 will give the result as PASS.	The result is PASS because at any given point of time, the bus must be driven by only one driver as defined in C1.
Contention	The contention check with constraint CS1 will give the result as PASS.	The result is PASS because only one driver is driving the bus so there are no conflicting values.

Now, consider another situation, in which constraint CS1 is disabled and constraints CS2 and CS3 are enabled. In this situation, signals, in1, in2, and in3 will have the same value at any given point of time. Considering the constraints CS2 and CS3, you will encounter the following results for different bus checks:

Type of Check	Result	Reason
Floating	The floating check with constraint CS2 and CS3 will give the result as FAIL.	The result is FAIL because if c1, c2, and c3 are all inactive, then out1 is floating.
Multiple drivers	The multiple drivers check with constraints CS2 and CS3 will give the result as FAIL.	The result is FAIL because if c1, c2, and c3 are all active, then all the three drivers of out1 are active.
Contention	The contention check with constraints CS2 and CS3 will give the result as PASS.	The result is PASS because there are no conflicting values. Constraints CS2 and CS3 indicate that in1, in2, and in3 will have the same value. Therefore, even if all the three drivers are active, the bus is not driven by conflicting values.

Xchecks

In simulation, "X" is treated as unknown, whereas in synthesis “X” is treated as don’t care. Synthesis is free to map "X" to any value (0 or 1) to meet area and/or timing constraints. This can lead to a mismatch between synthesis and simulation in presence of "X" in RTL. Consider the following example:

```
always @(posedge clk)
begin
    condition = en ? in1 : 1’bx;
        case (conditon)
        1’b0: out = in2;
        1’b1: out = in3;
        default: out = in4;
    endcase
end
```

In the above example, if the condition takes value "X", then in simulation the default branch of case statement will get executed. However, post-synthesis, "X" will get mapped to either a 0 or a 1 value.Therefore default will never get executed. This will lead to a difference in the simulation of pre- and post- synthesized design. It is therefore important to identify variables that can take the value "X" in RTL itself.

Xchecks in IFV will help in identifying the variables, which can take the value "X". Formal Verifier analyzes RTL designs for an X assigned to a signal. If there are any X assignments, IFV will create a check to see if these X assignments are reachable. If an X-assignment is reachable, then the corresponding Xcheck property is reported as Fail. If an X-assignment is unreachable, then the corresponding Xcheck property is reported as Pass.

Xcheck Property Naming Scheme

The name of the Xcheck property helps in identifying the cause of the property. In general, the name of an Xcheck property is of the following form:

```
xcheck_<VariableName>[_IndexInformation]<_lineNo>[_xc<Digit>]
```

where:

`xcheck`	denotes that this is an Xcheck property.
`VariableName`	denotes the variable that has been assigned the value X.
`IndexInformation`	specifies the index information, which is only generated if the vector variable is partly assigned to X. For an index range of a variable that is assigned to X, the corresponding index information is represented by two literals.
`lineNo`	specifies the line number where the assignment has lead to the value x on the variable.
`xc[Digit]`	denotes that if two checks are made for the same variable and the same line number, then to create an assertion with a unique name, xc[digit] is appended to the name.

Multiple X Assignments to a Variable

Consider the following example:

```
1   module top( input en, output  out1 , out2);
2   assign out1 = en ? 1'bx : 1'bx; // X assignments on the same line
3   assign out2 = en ? 1'bx :
4                         1'bx; // X assignments on the different lines
3   endmodule
```

For this design, the following Xcheck properties will be generated:

```
xcheck_out1_line2
xcheck_out1_line2_xc1
xcheck_out2_line3
xcheck_out2_line4
```

The Xcheck properties in this design are generated for lines 2, 3 and 4. As there are two X-assignments in line 2, the second property, xcheck_out1_line2_xc1, indicates the value `xc1`, that is, if there are two X assignments in the same line then the property generated for the second X-assignment is represented as `xc1`. However, if the second X-assignment is mentioned in the next line, as shown in line 4, then the property generated will be represented as xcheck_out2_line4.

X Assignments to the Selected Bits of a Variable

Consider the following example:

```
1   module top( input [1:0] cond, output  out1);
2   reg [4:0] out1;
3   reg [4:0] out2[4:0];
4
5   always @(cond)
6   case (cond)
7   2’b00:  out1 = 5’b00000;
8   2’b01:  out1 = {3’bx,2’b0};
9   2’b10:  out1 = {3’b0, 2’bx};
10  2’b11:  out1 = 5’b00011;
11  default: out1 = 5’b0;
12  endcase
13
14  always @(cond)
15  case (cond)
16  2’b00:  out2[3] = 5’b00000;
17  2’b01:  out2[3] = {3’bx,2’b0};
18  2’b10:  out2[3] = {3’b0, 2’bx};
19  2’b11:  out2[3] = 5’b00011;
20  default: out2[3] = 5’b0;
21  endcase
```

For this design, the following Xcheck properties will be generated:

```
xcheck_out2_3_3_1_0_line18
xcheck_out2_3_3_4_2_line17
xcheck_out1_1_0_line9
xcheck_out1_4_2_line8
```

In the property, `xcheck_out1_1_0_line9`, `xcheck` literal depicts an xcheck assertion, `out1` is the variable name, `1_0` is the index information of the signal, that is bits 1 to down 0 will take the value X, and `line9` is the line number of the HDL design where X-assignment has been identified.

In the property, `xcheck_out2_3_3_1_0_line18`, `xcheck` literal depicts an xcheck assertion, `out2` is the variable name, `3_3_1_0` is the index information of the signal, where, `3_3`, the first two literals are for the first dimension of the variable and the next two literals, `1_0`, are for the second dimension of the variable that has been assigned the value X. Line 18 is the line number of the HDL design where X-assignment has been identified.

X-Assignment at Module Instance Port

The name of an Xcheck property with X-assignment at module instance port is of the following form:

```
xcheck__<instanceName>_<portNo>_<lineNo>[_xc<Digit>]
```

Consider the following example:

```
1   module top( input clk, rst ,en, a , b, c, output [1:0] out1, out2);
2   sub t1module (  clk, rst ,en, 1’bx , 1’bx , c,  out1 );
3   sub t2module ( .clk(clk), .rst(rst) , .a(1’bx) , .en(en), .b(1’bx),
4   .c(c), .xout(out2) );
5   endmodule
6   module sub( input clk, rst ,en, a , b, c, output [1:0] xout );
7   reg  [1:0]xout;
8   always @(posedge clk)
9   begin
10      if(rst)
11          xout = 0;
12      else if(en)
13          xout = {1’b0,1’b1};
14      else if(a)
15          xout = ’b1;
16      else if(b)
17          xout = {1’b1,1’b1};
18      else
19          xout = ’b1;
20  end
21  endmodule
```

For this design, the following Xcheck properties will be generated:

```
xcheck__t1module_4_line2
xcheck__t1module_5_line2
xcheck__t2module_4_line3
xcheck__t2module_5_line3
```

In the property, `xcheck__t1module_4_line2`, `xcheck` literal depicts an xcheck assertion, `t1module` is the instance name, `4` is the port number, and `line2` is the line number of the HDL design where X-assignment has been identified. The port number in the HDL design is defined based on the instantiated module port order.

X-Assignment in HDL Expressions

Consider the following example:

```
1   module top( input clk, a , b, c, output [3:0] e1, output [2:0] e2);
2   reg [3:0] e1;
3   reg [2:0] e2;
4   always @(clk) begin
5       if (clk) begin
6        {e1[3], e1[2], e1[1:0]} <= {a,c, 1’bx, b};
7            e2 = {2’b0, b} & 3’bx;
8       end
```

```
9   end
10  endmodule
```

For this design, the Xcheck properties generated will be:

```
xcheck_e1_1_1_line6
xcheck_e2_line7
```

In the property,`xcheck_e1_1_1_line6`, the literal `1_1` denotes that an X-assignment is made to `e[1].`

Understanding the Results of Xchecks

The Xcheck properties are generated to identify the X assignments that can result in functional bugs. If an X-assignment is determined to be unreachable, then the status of the corresponding Xcheck assertion is reported as Pass. If an X-assignment is determined to be reachable, then the status of the corresponding Xcheck assertion is reported as Fail.

Consider the following example:

```
1   module top(input [1:0] in0,in1,in2, output [1:0] dout);
2
3   mux3ds  mux (in0,in1,in2,3'b010,dout);
4
5   endmodule
6
7   module mux3ds (input [1:0] in0,in1,in2, input [2:0] sel, output reg
[1:0] dout) ;
8
9   always @( in0 or  in1 or  in2 or sel)
10      case (sel)
11           3'b000 : dout = in0;
12           3'b001 : dout = in1;
13           3'b010 : dout = 2'bx;
14           3'b011 : dout = in2;
15           3'b100 : dout = 2'bx;
16           3'b101 : dout = 2'bx;
17           3'b110 : dout = 2'bx;
18           3'b111 : dout = 2'bx;
19           default : dout = 2'bx;
20          endcase
21
22  endmodule
```

For this design, the Formal Verifier generates the following Xcheck properties:

```
mux.xcheck_dout_line13 : Fail (0)
mux.xcheck_dout_line15 : Pass
mux.xcheck_dout_line16 : Pass
mux.xcheck_dout_line17 : Pass
mux.xcheck_dout_line18 : Pass
```

The Xcheck encapsulates all the ‘X’ in the design that lead to the assignment of X to a variable. The Xcheck assertions generated for lines 15, 16, 17, and 18, have a Pass status because signal `sel` is always tied to the value `3'b010` and therefore, X assignments on these lines are not reachable. Xcheck assertion, `mux.xcheck_dout_line13`, generated for line 13 has a status as Fail because this assignment is reachable.

A failed Xcheck assertion has a counter-example associated with it. The counter-example shows the sequence of inputs that are required to execute the corresponding X-assignment in the code. A pass Xcheck indicates that corresponding X-assignment is not reachable.

Range Overflow Checks

IFV analyzes HDL designs for indexed expressions. If there are indexed expressions, IFV creates a check to see if these expressions are indexed within the defined range. These created checks are termed as range overflow checks. The range overflow assertions are reported as Pass if the value assigned to the variable is within the defined index range. If the value is not within the defined index range then the corresponding range overflow assertion is reported as Fail.

Consider the following example:

```
module top(input clk, input[1:0] in1, in2,input [3:0] in3, output reg out);
always@ (posedge clk)
begin
    out = in3[in2];
    if(in2 > 1)
    out =  in1[in2];
end
endmodule
```

In the above example, the following range overflow property will be generated:

```
rangeoverflow_in1_line8
```

The range overflow property is generated for expression `in1[in2]` on line number 8 because if variable `in2` takes any value greater than 1, then that value is not within the defined index range of `in1`. However, the expression, `in3[in2]`, on line number 6 can never lead to range overflow condition because variable `in2` in this case will always take values between 3 and 0, which is within the defined range of `in3`.

Range Overflow Property Naming Scheme

The name of the range overflow property helps in identifying the cause of the property. In general, the name of a range overflow property is of the following form:

```
rangeoverflow_<VariableName>_<lineNo>[_ro<Digit>]
```

where:

`rangeoverflow`	denotes that this is a range overflow property.
`VariableName`	denotes the variable for which the range overflow assertion has been made.
`lineNo`	specifies the line number of the HDL design, where the range overflow has occurred.
`ro<Digit>`	denotes that if two checks are made for the same variable and the same line number, then to create an assertion with a unique name, ro<Digit> is appended to the name.

For example, in the property, `rangeoverflow_in1_line8`, `rangeoverflow` literal depicts a range overflow assertion, `in1` is the variable name, and `line8` is the line number of the HDL design where the variable, `in1`, has been indexed.

For a variable indexed inside a function/task, the name of the corresponding rangeoverflow check is of the following form:

```
rangeoverflow_<function/task_name>_<VariableName>_<lineNo>[_ro<Digit>]
```

Consider the following example:

```
1   module top(input in, clk,
2   output  out1, out2);
3   wire [0:2] in ;
4
5   function  GETFUNCTION;
6       input integer first; input [0:2] select;
7           begin
8               GETFUNCTION = select[first];
9           end
10  endfunction
11  assign out2 = GETFUNCTION(12 , in);
12  endmodule
```

For the above example, the following range overflow property will be generated:

```
rangeoverflow_GETFUNCTION_select_line8
```

In the property, `rangeoverflow_GETFUNCTION_select_line8`, `rangeoverflow` literal depicts a range overflow assertion, `GETFUNCTION` is the function name, `select` is the variable name in the function, and `line8` is the line number of the design where the variable has been indexed.

If the expression indexed is other than the HDL variable, then the range overflow property can be of the following form:

```
rangeoverflow_<ROF>_<lineNo>[_ro<Digit>]
```

Consider the following example:

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.std_logic_arith.all;
4
5   ENTITY top is
6    GENERIC(
7    SIZE_IN : integer := 8
8    );
9    PORT (
10      inInt  : in integer;
11      selBit : in integer;
12      outBit : out std_logic
13   );
14  END top;
15
16  architecture rtl of top is
17  begin
18  outBit <= CONV_STD_LOGIC_VECTOR(inInt, SIZE_IN)(selBit);
19  end rtl;
```

For this design, IFV generates the following range overflow property:

```
rangeoverflow_ROF_line18
```

In this example, a property is made to check the range overflow for an indexing of a VHDL function call, therefore, the name of the generated property contains the ROF literal.

Understanding the Results of Range Overflow

The range overflow check properties are generated to identify out of range expressions in the design. IFV verifies these properties by using formal methods. If the value of the variable is within the defined index range, then the assertion is reported as Pass. If the value of the variable is outside of its defined index range, then the assertion is reported as Fail. The corresponding failed counter-example can be viewed using the debug command.

Consider the following examples of range overflow checks:

Variable Index is out of Range

```
1   module top(input in, in1,in2,clk, output out );
2   wire [0:3] in ;
3   wire [0:2] in1, in2;
4   reg [0:3] out;
5   always @(posedge clk)
6   begin
7    out[in1] = in[in2];
8   end
9   endmodule
```

For this design, IFV generates the following range overflow assertions:

```
rangeoverflow_in_line7 : Fail (2)
rangeoverflow_out_line7 : Fail (2)
```

Any value of `in1` and `in2`, which is greater than 3, will lead to out of range condition, therefore, both the rangeoverflow assertions for variables, `in` and `out`, are reported as Fail.

If you add a constraint `in2` is equal to 0 to the above design, the result status of the range overflow assertions will change to:

```
rangeoverflow_in_line7 : Pass
rangeoverflow_out_line7 : Fail (2)
```

Due to the constraint, variable `in2` will take the value as 0. The value 0 is within the defined index range of `in[0:3]`, therefore, the range overflow assertion generated for variable `in` will result in Pass.

Ports in Function, Task, Module, and Gate Instance are out of Range

```
1   module top(input sel,in, clk,
2   output  out1, out2,out3 , out4, out5);
3   wire [0:2] in ;
4
5   function  GETFUNCTION;
6    input  first;
7       begin
8       GETFUNCTION = first;
9       end
10  endfunction
11
12  task  GETTASK;
13   input first;
14   output reg out;
15      begin
16      out  = first;
17      end
18  endtask
19
20  assign out1 = GETFUNCTION(in[ sel +: 3 ]  ); // Range overflow in a port of

                                                        the function call
21  reg out3;
```

```
22  always @(posedge clk)
23      begin
24      GETTASK ( in[ sel +: 3 ] , out3); // Range overflow in a port of the
                                           task call
25      end
26  wire  out4;
27  sub s (in[ sel +: 3 ], out4); // Range overflow in a port of the module
                                  instance
28  wire out5;
29  and aa(out5, in[ sel +: 3 ], in[ sel +: 3 ]); // Range overflow in a port
                                              of the gate instance
30  endmodule
31  module sub( input [2:0] in, output out);
32  assign out  = in ? 3'b0: 3'b1;
33  endmodule
```

For this design, IFV generates the following range overflow assertions:

```
rangeoverflow_in_line20 : Fail (0)
rangeoverflow_in_line27 : Fail (0)
rangeoverflow_in_line29 : Fail (0)
rangeoverflow_in_line29_ro1 : Fail (0)
rangeoverflow_in_line24 : Fail (2)
```

In the above design, all range overflow assertions are reported as Fail when variable `sel` takes value 1, because the value of the index `(sel +:3)` for `variable in` is not within the defined index range `[0:2]`. If `sel` takes value 0, then the variable `in` can take values within the defined index range `[0:2]` and all the assertions will be reported as Pass.

Expression in Loop Statement is out of Range

```
1   module top(c, out);
2    input c;
3    output [2:6] out;
4    reg    [2:6] out;
5    integer i;
6
7    always begin
8    for (i=0 ; i<= 6 ;  i=i+1)
        begin
9           out[i] = c;
        end
10   end
11  endmodule
```

For this design, IFV generates the following range overflow assertion:

```
rangeoverflow_out_line9 : Fail (0)
```

In the above design, the values 0 and 1 for `i` is not within the defined index range of out[2:6], therefore the generated assertion shows the result as Fail.

Using Automatically Generated Properties

To verify the automatically generated properties, you need to do the following:

1. View the properties generated by the tool.
2. Add the desired properties as assertions.
3. Specify reset constraints
4. Verify the assertions.
5. View the verification results.
6. Analyze counter-examples and witnesses.

The following sections discuss the details of each of these steps, using the following example:

```
module decoder ( out, in);
  output [3:0] out;
  input [1:0] in;
  reg [3:0] out;

  always @ (in)
  begin
    case (in)      // cadence full_case
      2'b01 : begin
                 out = 4'h1;
              end
      2'b10 : begin
                 out = 4'h2;
              end
      2'b11 : begin
                 out = 4'h3;
              end
    endcase
  end

endmodule
```

Viewing Generated Properties

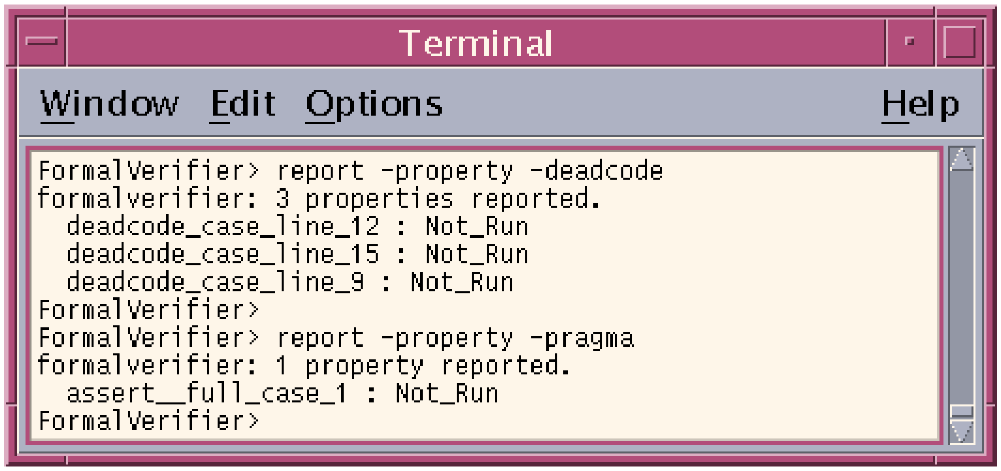
You can view the generated properties in the console window by using one of the following commands:

```
FormalVerifier> report -property -deadcode
FormalVerifier> report -property -pragma
FormalVerifier> report -property -fsm
FormalVerifier> report -property -bus
FormalVerifier> report -property -xcheck
FormalVerifier> report -property -rangeoverflow
```

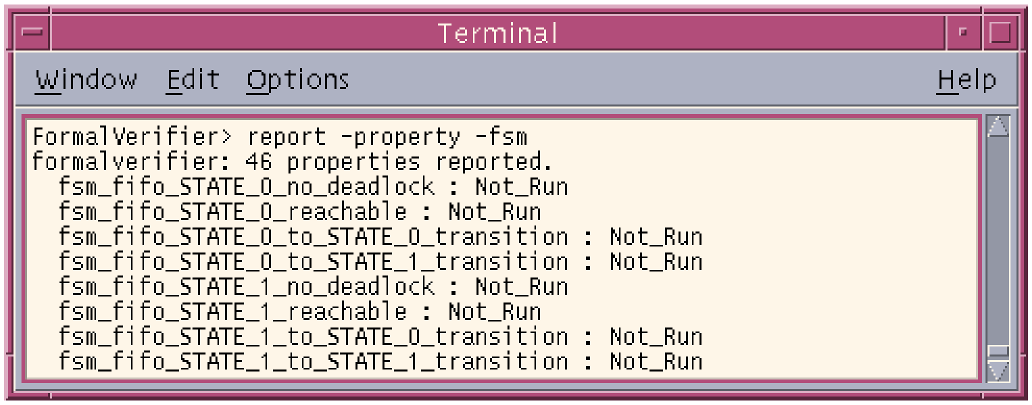
Note: If you want to view only user-defined properties and not the automatically generated properties, you can use the following command:

```
        FormalVerifier> report -property -specification
```

Examples of the use of the `report` commands are shown below:



```
FormalVerifier> report -property -deadcode
formalverifier: 3 properties reported.
  deadcode_case_line_12 : Not_Run
  deadcode_case_line_15 : Not_Run
  deadcode_case_line_9 : Not_Run
FormalVerifier>
FormalVerifier> report -property -pragma
formalverifier: 1 property reported.
  assert__full_case_1 : Not_Run
FormalVerifier>
```

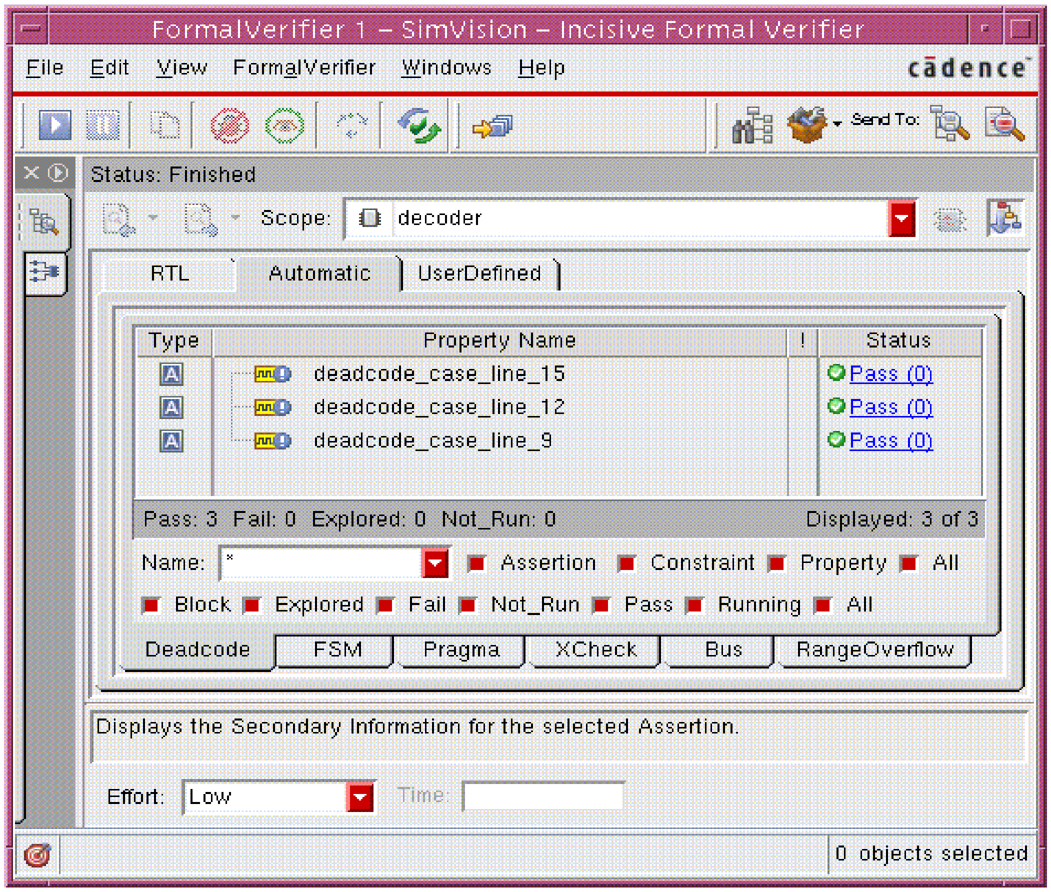


```
FormalVerifier> report -property -fsm
formalverifier: 46 properties reported.
  fsm_fifo_STATE_0_no_deadlock : Not_Run
  fsm_fifo_STATE_0_reachable : Not_Run
  fsm_fifo_STATE_0_to_STATE_0_transition : Not_Run
  fsm_fifo_STATE_0_to_STATE_1_transition : Not_Run
  fsm_fifo_STATE_1_no_deadlock : Not_Run
  fsm_fifo_STATE_1_reachable : Not_Run
  fsm_fifo_STATE_1_to_STATE_0_transition : Not_Run
  fsm_fifo_STATE_1_to_STATE_1_transition : Not_Run
```

For details of the `report` command, refer to *Chapter 1, “Formal Verifier Command Reference,”* of the *Formal Verifier Reference Manual*.

Alternatively, you can view the list of checks in the GUI by selecting the desired tab. For deadcode properties, click on the *Automatic* tab and then on the *Deadcode* tab. For pragma properties, click on the *Automatic* tab and then on the *Pragma* tab. For FSM properties, click on the *Automatic* tab and then on the *FSM* tab. For *Bus* check, click the Automatic tab and then the bus tab. For Xcheck, click the Automatic tab and then the XCheck tab.

The example below shows the *Deadcode* tab in the GUI.



Status: Finished

Scope: decoder

Type	Property Name	!	Status
A	deadcode_case_line_15		Pass (0)
A	deadcode_case_line_12		Pass (0)
A	deadcode_case_line_9		Pass (0)

Pass: 3 Fail: 0 Explored: 0 Not_Run: 0 Displayed: 3 of 3

Name: *

Displays the Secondary Information for the selected Assertion.

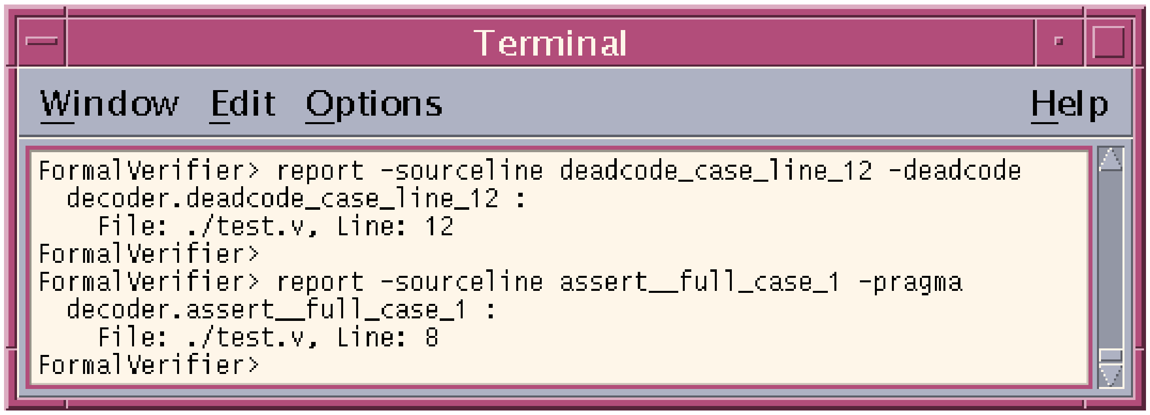
Effort: Low

0 objects selected

For each generated property you can get the line number in the HDL source code from which it was derived. To do this in the console, use one of the following commands:

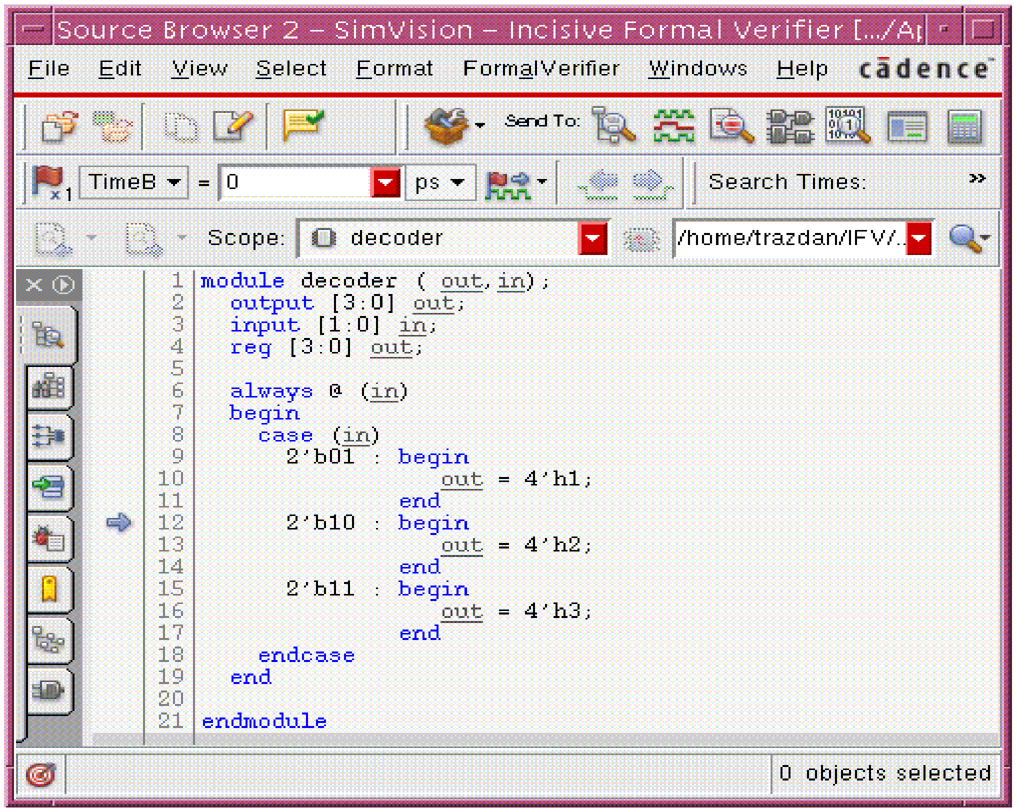
```
FormalVerifier> report -sourceline <name> -deadcode
FormalVerifier> report -sourceline <name> -pragma
FormalVerifier> report -sourceline <name> -bus
FormalVerifier> report -sourceline <name> -xcheck
```

An example of the use of the two `report` commands is shown below:



```
FormalVerifier> report -sourceline deadcode_case_line_12 -deadcode
  decoder.deadcode_case_line_12 :
    File: ./test.v, Line: 12
FormalVerifier>
FormalVerifier> report -sourceline assert__full_case_1 -pragma
  decoder.assert__full_case_1 :
    File: ./test.v, Line: 8
FormalVerifier>
```

In the GUI, you can locate the line in the HDL source code by double-clicking on the property. This opens the *Source Browser* window, with the pointer pointing to the line number of the code for the corresponding deadcode check. An example is shown below:


Source Browser 2 – SimVision – Incisive Formal Verifier [.../A
File Edit View Select Format FormalVerifier Windows Help cādence
Send To:
TimeB = 0 ps
Search Times:
Scope: decoder
/home/trazdan/IFV/.
0 objects selected

```
module decoder ( out, in);
  output [3:0] out;
  input [1:0] in;
  reg [3:0] out;

  always @ (in)
  begin
    case (in)
      2'b01 : begin
                out = 4'h1;
              end
      2'b10 : begin
                out = 4'h2;
              end
      2'b11 : begin
                out = 4'h3;
              end
    endcase
  end
endmodule
```

Adding Properties as Assertions

To be able to verify the generated properties, you need to add them as assertions. You can do this in several ways:

```
FormalVerifier> assertion -add -deadcode
FormalVerifier> assertion -add -pragma
FormalVerifier> assertion -add -fsm
```

These commands add only the specified types of generated properties. These are useful if you want to focus your efforts on automatic assertions.

```
FormalVerifier> assertion -add -all
```

This command adds all automatically generated properties in addition to user-defined assertions.

Note: If you want to add only user-defined properties and not the automatically generated properties, you can use the following command:

```
FormalVerifier> assertion -add -specification
```

In the GUI, adding automatic properties is similar to adding user-defined. Once the desired properties have been added as assertions, you can proceed to prove them as you would prove any other assertion. Refer to the section titled Manipulating Properties on page 99 for a description of how to add properties as assertions and prove them.

Specifying Reset Constraints

A design typically has two modes, the reset mode and the functional mode. The reset mode of the design is used to initialize the state elements in the design. The functional mode of the design is used during normal operation.

In general, assertions are verified when the design is in the functional mode. You can force the design to be in the functional mode by constraining the reset signal(s) to be inactive. This is often done by using the following command:

```
FormalVerifier> constraint -add -pin reset 0
```

Using this command can cause problems for FSM-state, FSM-transition, and deadcode assertions. If reset is tied inactive then some assertions associated with reset code might incorrectly be reported as failing. Consider the following example with deadcode assertions to illustrate this problem.

```
1   ...
2   always @(posedge clk or posedge reset)
3       if (reset)
4            out1 = 1'b0;
5       else
```

```
6           out1 = d1;
7   ...
```

In this example, when the signal `reset` is high, the design is in the reset mode.

Formal Verifier creates the following two deadcode assertions for the above HDL code:

- `deadcode_if_line_4`
- `deadcode_if_line_6`

If reset is tied inactive, then `deadcode_if_line_4` will fail even though it is clearly reachable during reset conditions.

To avoid such situations, Formal Verifier allows one or more constraints to be denoted as `reset constraints`. Reset constraints are ignored for AFA checks, such as deadcode, FSM-state, and FSM-transition. For verification of other assertions including the rest of AFA, the reset constraint is honored.

- To specify a pin constraint as a reset constraint, you need to give the following command:

  ```
  FormalVerifier> constraint -add -pin reset 0 -reset
  ```

- To specify an assertion constraint as a reset constraint, you need to give the following command:

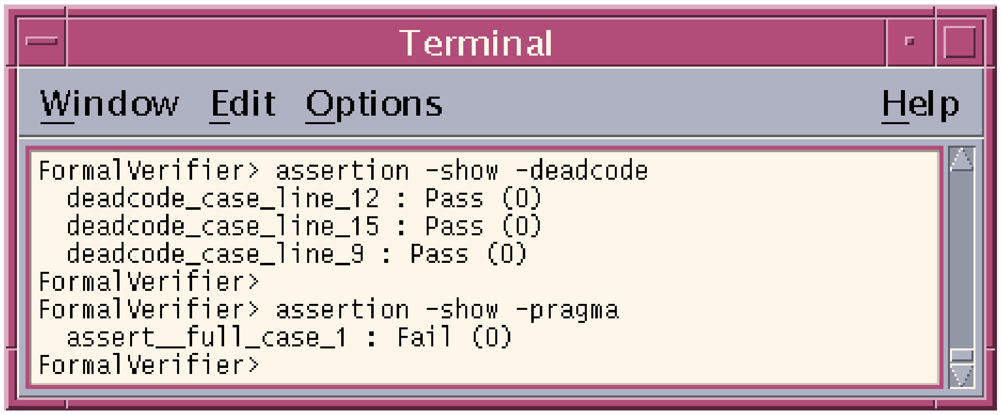
  ```
  FormalVerifier> constraint -add assume_reset -reset
  ```

Viewing Results

You can view the status of the automatic assertions after verification by using one or more of the following commands:

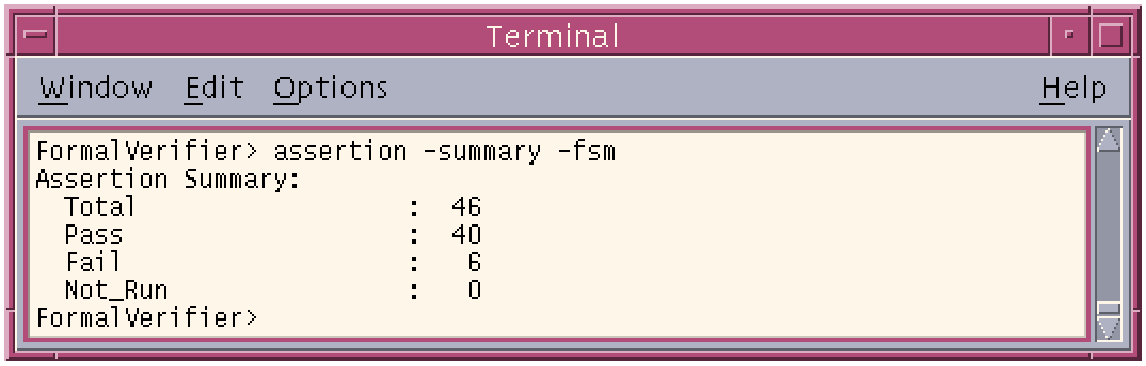
```
FormalVerifier> assertion -show -deadcode
FormalVerifier> assertion -show -pragma
FormalVerifier> assertion -show -fsm
FormalVerifier> assertion -show -bus
FormalVerifier> assertion -show -xcheck
FormalVerifier> assertion -show -rangeoverflow
```

An example of the use of the first two `assertion` commands is shown below:



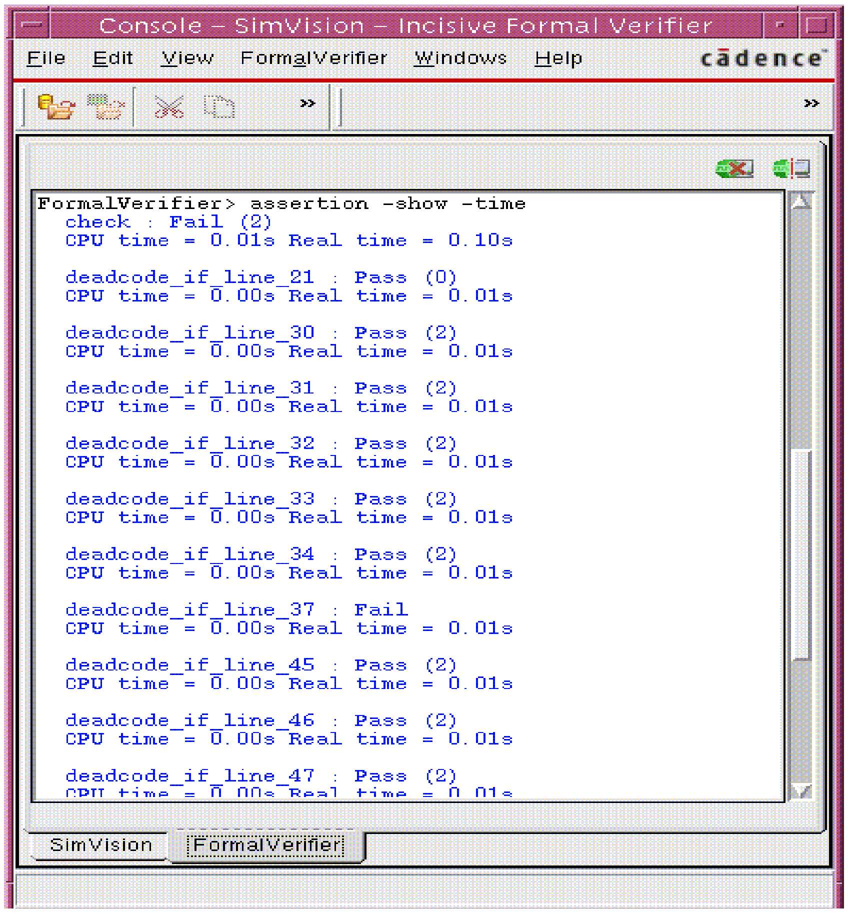
```
FormalVerifier> assertion -show -deadcode
  deadcode_case_line_12 : Pass (0)
  deadcode_case_line_15 : Pass (0)
  deadcode_case_line_9 : Pass (0)
FormalVerifier>
FormalVerifier> assertion -show -pragma
  assert__full_case_1 : Fail (0)
FormalVerifier>
```

You can also use the `assertion -summary` command to view a summary report, as shown below:



```
FormalVerifier> assertion -summary -fsm
Assertion Summary:
  Total                    :  46
  Pass                     :  40
  Fail                     :   6
  Not_Run                  :   0
FormalVerifier>
```

You can also use the `assertion -show -time` command to view the CPU and real time taken by each of these assertions, during verification. The report for this command is shown below:


Console – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cādence
SimVision
FormalVerifier

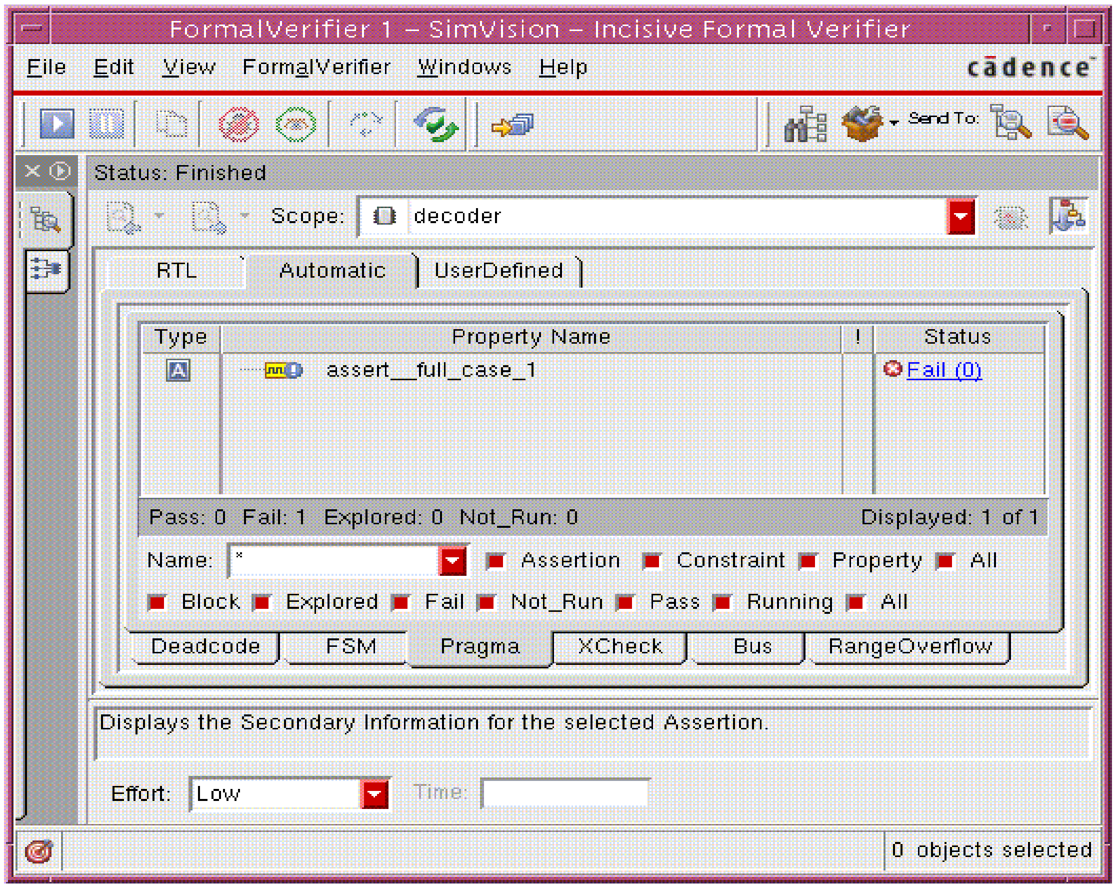
```
FormalVerifier> assertion -show -time
  check : Fail (2)
  CPU time = 0.01s Real time = 0.10s

  deadcode_if_line_21 : Pass (0)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_30 : Pass (2)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_31 : Pass (2)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_32 : Pass (2)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_33 : Pass (2)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_34 : Pass (2)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_37 : Fail
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_45 : Pass (2)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_46 : Pass (2)
  CPU time = 0.00s Real time = 0.01s

  deadcode_if_line_47 : Pass (2)
  CPU time = 0.00s Real time = 0.01s
```

In this report, all assertions are shown to be having the same time status. This is because automatic formal assertions are large in number and IFV tries to run them together for better performance. Due to this, you get to view the same time status against all assertions even though some of the assertions would have reached the conclusive result earlier than others.

Other variations of the `assertion -show` command are possible. For details of the `assertion` command, refer to *Chapter1, “Formal Verifier Command Reference,”* of the *Formal Verifier Reference Manual* or the online help for more details.

Alternatively, you can view the results for a given type of automatic property in the GUI by selecting the desired tab. For deadcode, click the *Automatic* tab and then the *Deadcode* tab. For pragma properties, click the *Automatic* tab and then the *Pragma* tab. For FSM properties, click the *Automatic* tab and then the *FSM* tab. For bus properties, click the *Automatic* tab and then the *Bus* tab. For Xcheck properties, click the *Automatic* tab and then the *XCheck* tab.

The example below shows the *Pragma* tab in the GUI.


FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cadence
Send To:
RTL Automatic UserDefined
Deadcode FSM Pragma XCheck Bus RangeOverflow

Status: Finished

Scope: decoder

Type	Property Name	!	Status
A	assert__full_case_1		Fail (0)

Pass: 0 Fail: 1 Explored: 0 Not_Run: 0 Displayed: 1 of 1

Name: * 🟥 Assertion 🟥 Constraint 🟥 Property 🟥 All

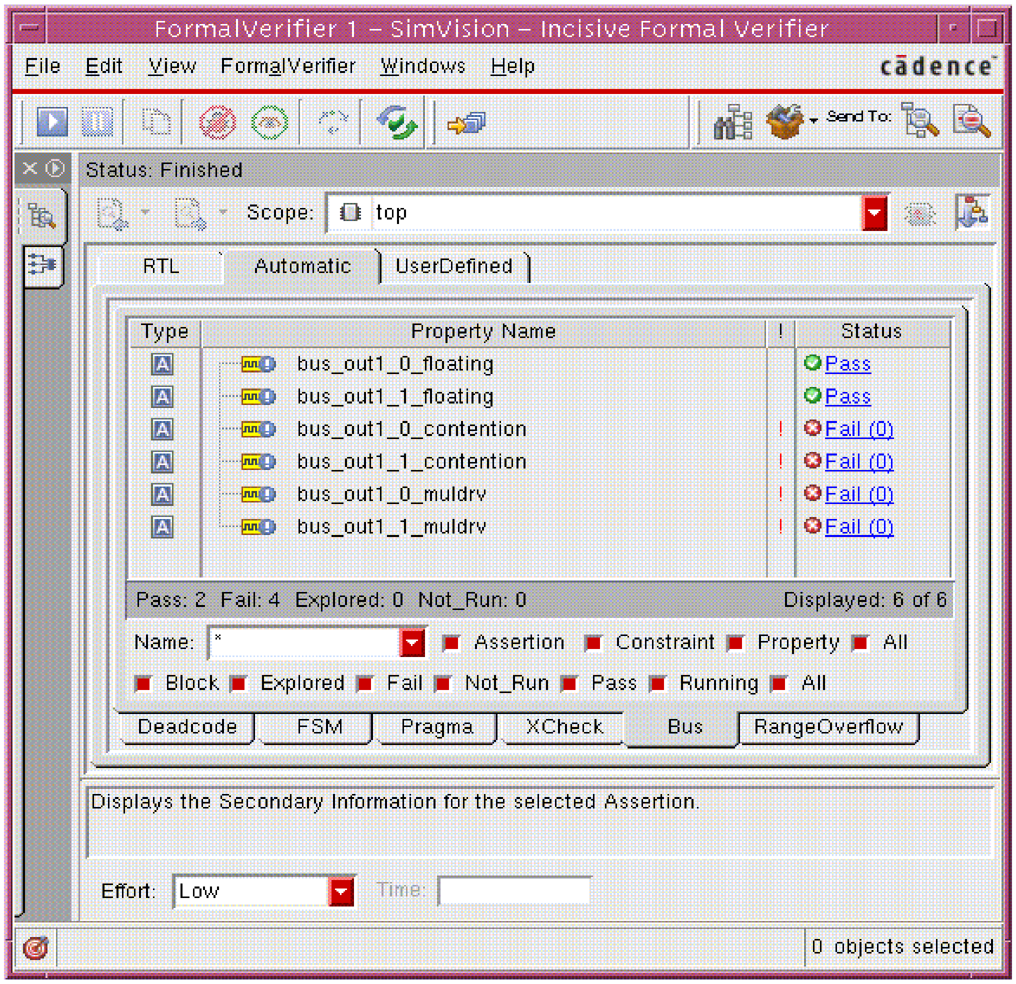
🟥 Block 🟥 Explored 🟥 Fail 🟥 Not_Run 🟥 Pass 🟥 Running 🟥 All

Displays the Secondary Information for the selected Assertion.

Effort: Low Time:

0 objects selected

The *Bus* check tab in the GUI is displayed as:

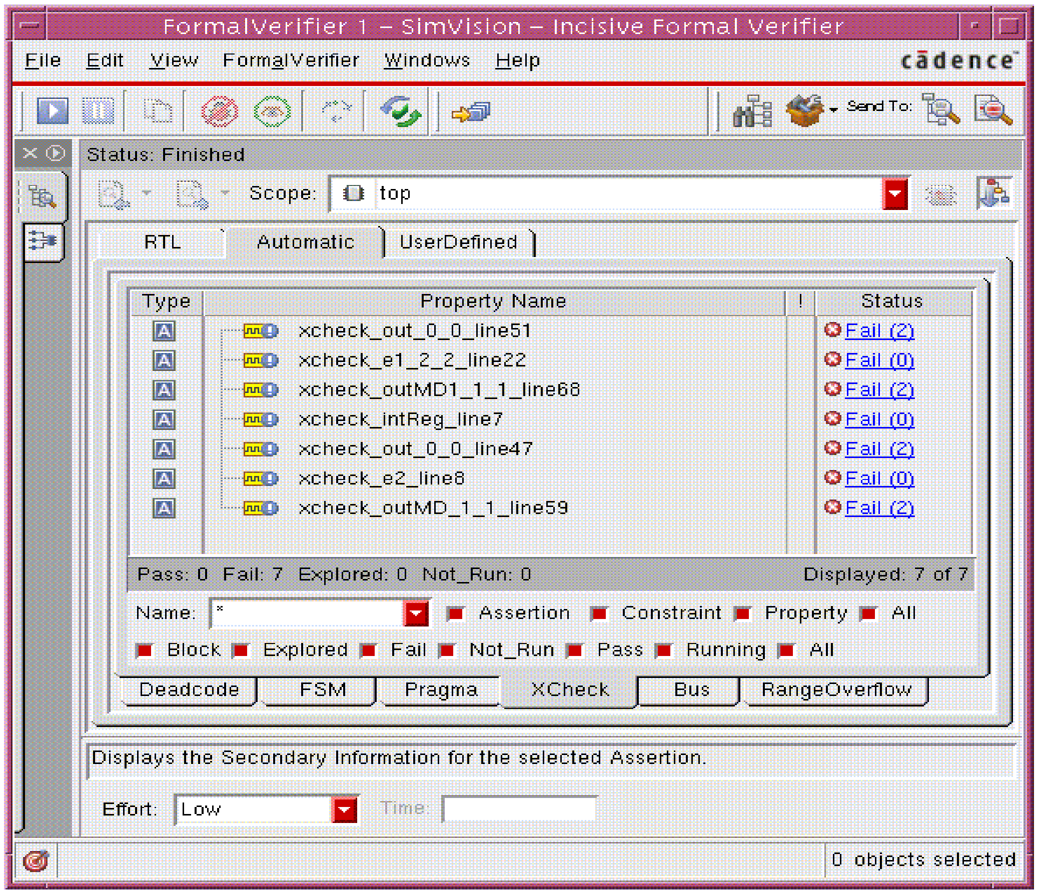


Type	Property Name	!	Status
A	bus_out1_0_floating		Pass
A	bus_out1_1_floating		Pass
A	bus_out1_0_contention	!	Fail (0)
A	bus_out1_1_contention	!	Fail (0)
A	bus_out1_0_muldrv	!	Fail (0)
A	bus_out1_1_muldrv	!	Fail (0)

Pass: 2 Fail: 4 Explored: 0 Not_Run: 0 Displayed: 6 of 6

Displays the Secondary Information for the selected Assertion.

The *XCheck* tab in the GUI is displayed as:


FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cādence
Send To:
RTL
Automatic
UserDefined
Deadcode
FSM
Pragma
XCheck
Bus
RangeOverflow

Status: Finished

Scope: top

Type	Property Name	!	Status
A	xcheck_out_0_0_line51		Fail (2)
A	xcheck_e1_2_2_line22		Fail (0)
A	xcheck_outMD1_1_1_line68		Fail (2)
A	xcheck_intReg_line7		Fail (0)
A	xcheck_out_0_0_line47		Fail (2)
A	xcheck_e2_line8		Fail (0)
A	xcheck_outMD_1_1_line59		Fail (2)

Pass: 0 Fail: 7 Explored: 0 Not_Run: 0 Displayed: 7 of 7

Name: * 🟥 Assertion 🟥 Constraint 🟥 Property 🟥 All

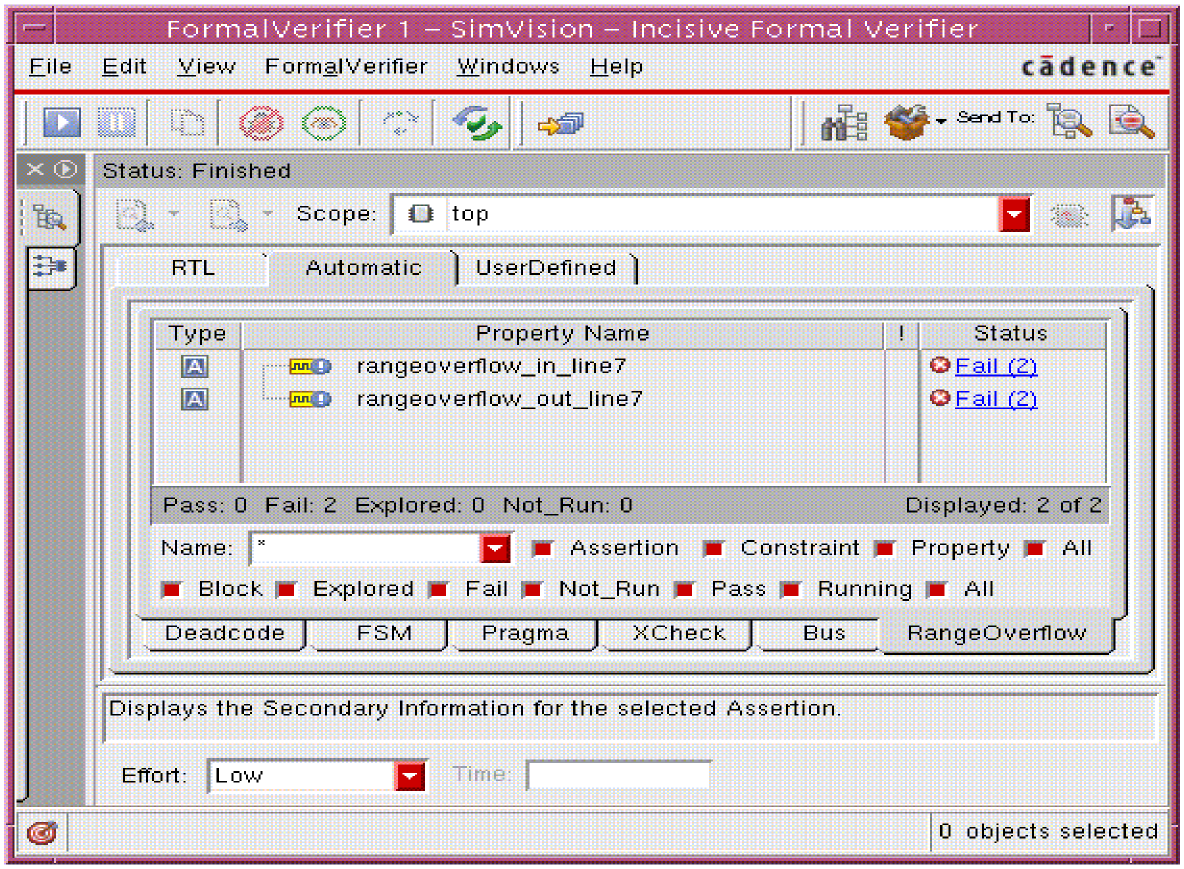
🟥 Block 🟥 Explored 🟥 Fail 🟥 Not_Run 🟥 Pass 🟥 Running 🟥 All

Displays the Secondary Information for the selected Assertion.

Effort: Low Time:

0 objects selected

The *RangeOverflow* tab in the GUI is displayed as:


FormalVerifier 1 – SimVision – Incisive Formal Verifier
File Edit View FormalVerifier Windows Help
cadence
Send To:
Status: Finished
Scope: top
RTL
Automatic
UserDefined
Deadcode
FSM
Pragma
XCheck
Bus
RangeOverflow
Name: *
Assertion Constraint Property All
Block Explored Fail Not_Run Pass Running All
Effort: Low
Time:
0 objects selected

Type	Property Name	!	Status
A	rangeoverflow_in_line7		Fail (2)
A	rangeoverflow_out_line7		Fail (2)

Pass: 0 Fail: 2 Explored: 0 Not_Run: 0 Displayed: 2 of 2

Displays the Secondary Information for the selected Assertion.

Analyzing the Counter-example and Witness Waveforms

Counter-examples and witnesses for the automatic assertions are viewed in exactly the same way as they are for user-defined assertions. Refer to Chapter 9, "Debugging Failures and Viewing Witnesses," for more details.

Important

By default, the witness is not generated for the passing deadcode and FSM assertions. To generate the witness, you need to do the following:

❑ If you have not verified the design:

a. Set the witness generation flag by using the following command:

```
FormalVerifier> define debugmode on
```

b. Verify the design by using the following command:

```
FormalVerifier> prove
```

Note: There is a performance penalty for generating the witness.

❑ If you have verified the design:

a. Set the witness generation flag by using the following command:

```
FormalVerifier> define debugmode on
```

b. Delete those deadcode assertions whose witness you want to view by using the following command:

```
FormalVerifier> assertion -delete <assertionName> -deadcode
FormalVerifier> assertion -delete <assertionName> -fsm
```

c. Add those deadcode assertions whose witness you want to view by using the following command:

```
FormalVerifier> assertion -add <assertionName> -deadcode
FormalVerifier> assertion -add <assertionName> -fsm
```

d. Verify the design by using the following command:

```
FormalVerifier> prove
```

11

Synergy with Simulation

Formal Verifier generates a counter-example in case of an assertion failure. The counter-example shows the sequence of values taken by various signals in the design leading to the assertion failure. Formal Verifier can export this counter-example as an HDL or Tcl testbench. You can then use this automatically generated testbench to recreate the assertion failure using dynamic (simulation-based) ABV.

HDL Testbench Generation

To generate an HDL testbench, you can use the following command in Formal Verifier:

```
FormalVerifier> debug <assertionName> -dynamic
```

Refer to Chapter 9, “Debugging Failures and Viewing Witnesses,” of this guide or to *Chapter 1, “Formal Verifier Command Reference,”* of the *Formal Verifier Reference Manual* for more information about the `debug` command and its uses.

The `debug` command in Formal Verifier enables you to:

- Generate a testbench.
- Run simulation with the generated testbench.
- Generate the simulation waveform.
- View the simulation waveform.

While generating an HDL testbench, Formal Verifier generates the following files:

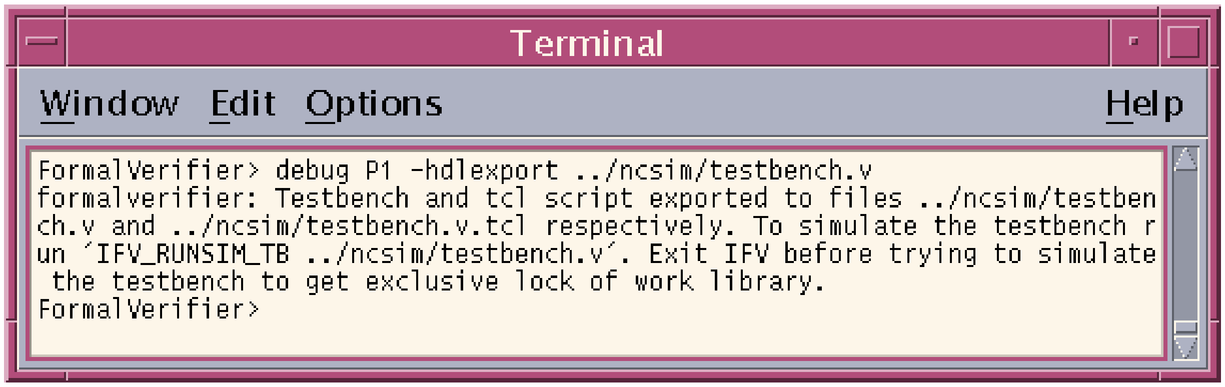
- Verilog testbench file.
- Input Tcl file for `ncsim`.
- `IFV_RUNSIM_TB` shell script to run simulation using `ncverilog`.

  Refer to *Chapter 4, “Running NC-Verilog with the ncverilog Command”* of the *NC-Verilog Simulator Help* for details on the `ncverilog` command.

For example, assume that you have a Formal Verifier run in which assertion `P1` fails. You can generate a testbench illustrating the failure by issuing the following command:

```
FormalVerifier> debug P1 -hdlexport ../ncsim/testbench.v
```

This would generate the testbench in file `../ncsim/testbench.v`. It would also generate `IFV_RUNSIM_TB` and `testbench.v.tcl` in the `../ncsim` directory. The image below shows an example of running this command:



```
FormalVerifier> debug P1 -hdlexport ../ncsim/testbench.v
formalverifier: Testbench and tcl script exported to files ../ncsim/testben
ch.v and ../ncsim/testbench.v.tcl respectively. To simulate the testbench r
un 'IFV_RUNSIM_TB ../ncsim/testbench.v'. Exit IFV before trying to simulate
 the testbench to get exclusive lock of work library.
FormalVerifier>
```

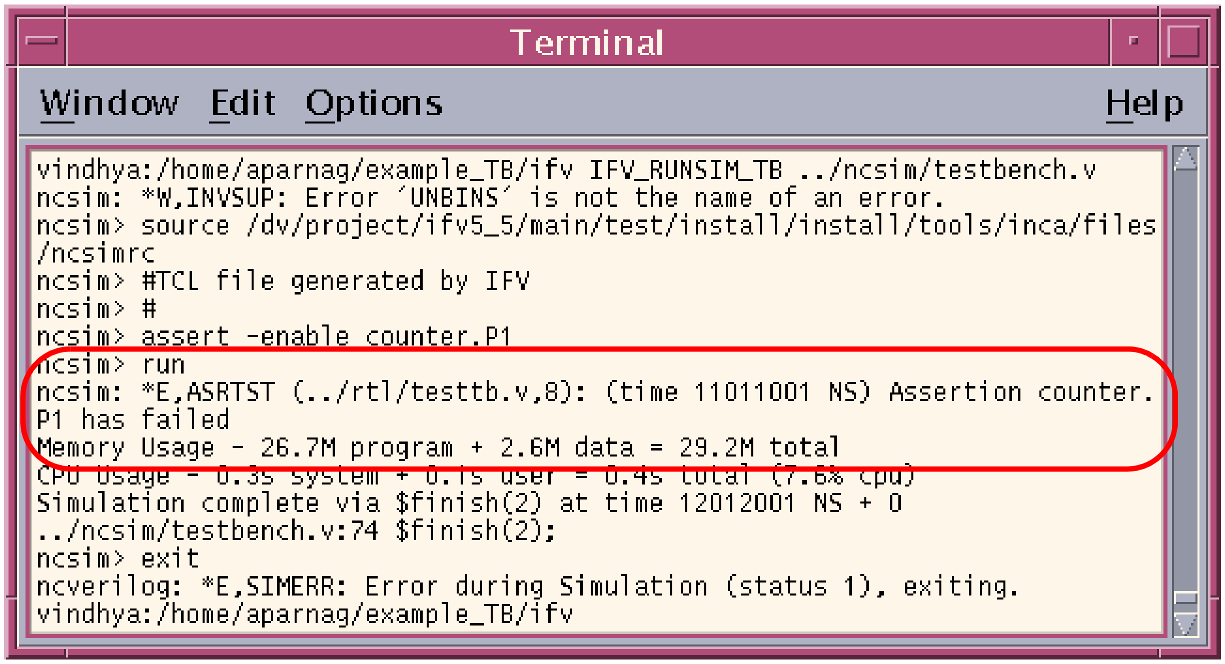
Note: You can generate a testbench from the witness of deadcode checks by issuing the following command:

```
FormalVerifier> witness P1 -hdlexport ../ncsim/testbench.v
```

Later, you can run the testbench by issuing the following command:

```
% IFV_RUNSIM_TB ../ncsim/testbench.v
```

The image below shows an example of running this command:



```
vindhya:/home/aparnag/example_TB/ifv IFV_RUNSIM_TB ../ncsim/testbench.v
ncsim: *W,INVSUP: Error 'UNBINS' is not the name of an error.
ncsim> source /dv/project/ifv5_5/main/test/install/install/tools/inca/files
/ncsimrc
ncsim> #TCL file generated by IFV
ncsim> #
ncsim> assert -enable counter.P1
ncsim> run
ncsim: *E,ASRTST (../rtl/testtb.v,8): (time 11011001 NS) Assertion counter.
P1 has failed
Memory Usage - 26.7M program + 2.6M data = 29.2M total
CPU Usage - 0.3s system + 0.1s user = 0.4s total (7.6% cpu)
Simulation complete via $finish(2) at time 12012001 NS + 0
../ncsim/testbench.v:74 $finish(2);
ncsim> exit
ncverilog: *E,SIMERR: Error during Simulation (status 1), exiting.
vindhya:/home/aparnag/example_TB/ifv
```

Note: The highlighted area in the image above shows how the simulator flags the assertion failure.

Handling Outputs of Blackboxed Modules

Formal Verifier treats the outputs of blackboxed modules as undriven nets and therefore they are non-deterministic during analysis. This means that Formal Verifier will check the validity of assertions assuming all possible values for these nets. While generating testbenches, you can choose from one of the following options:

- By default, Formal Verifier creates testbenches without any blackboxed elements. This means that the simulator will simulate the internals of the blackbox even if these elements were treated as blackboxes for formal analysis.

Important

Because blackboxes are present during formal analysis and absent during simulation, assertion results may differ.

- You can choose to honour blackboxing during simulation to recreate the assertion failure. In this case, you need to pass an additional argument (`+define+IFV_FORCE_BBOUT`) to the automatically generated `IFV_RUNSIM_TB` script. Below is an example:

```
    % IFV_RUNSIM_TB testbench.v +define+IFV_FORCE_BBOUT
```

HDL Testbench Generation Limitations

Testbench generation has the following limitations:

- Testbench generation is supported only for Verilog designs.
- A testbench cannot be generated for liveness assertions. This is because a liveness assertion can fail only on an infinite path, the liveness failure cannot be recreated in dynamic ABV environment.
- For unclocked assertions, the failure may not be generated using an event simulator. This is because of difference in the meaning of an evaluation cycle for Formal Verifier and an event-based simulator.
- Assertion failure may not be recreated during simulation of the generated testbench due to static-simulation differences. Refer to *Chapter 6, "Formal Analysis and Simulation Result Differences,"* of the *Formal Verifier Reference* for details of these differences.

Tcl Testbench Generation

Currently, you can generate an HDL testbench for a failed assertion for Verilog designs only. To enable you to generate a testbench for VHDL and mixed-language designs, Formal Verifier automatically generates a Tcl command language based testbench.

This feature is an extension of the HDL testbench generation described in the previous sections and is supported for Verilog, VHDL, and Mixed-language designs. The HDL testbench is not required when using the Tcl testbench.

Formal Verifier generates an input Tcl file generating a Tcl testbench. The Tcl testbench can be used only with the NC simulators of the IUS releases.

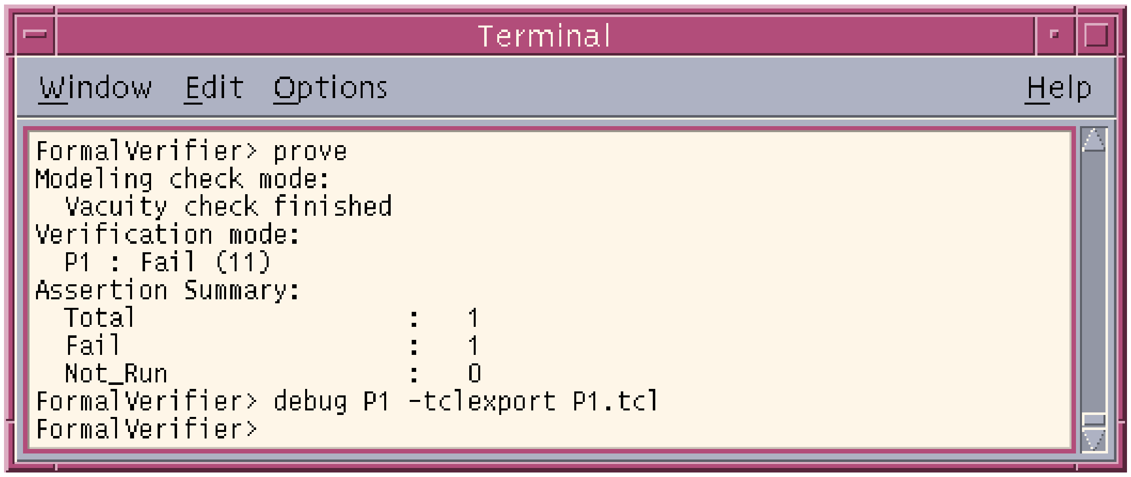
As an example, assume that you have a Formal Verifier run in which assertion `P1` fails. You could generate a tcl testbench `test.tcl` illustrating the failure by issuing the following command:

```
FormalVerifier> debug P1 -tclexport P1.tcl
```

Note: If you do not specify the file name, by default, a tcl file with the following name will be created in the current directory:

```
IFV_tcltb_<assertionName>.tcl
```

The image below shows an example of running this command:

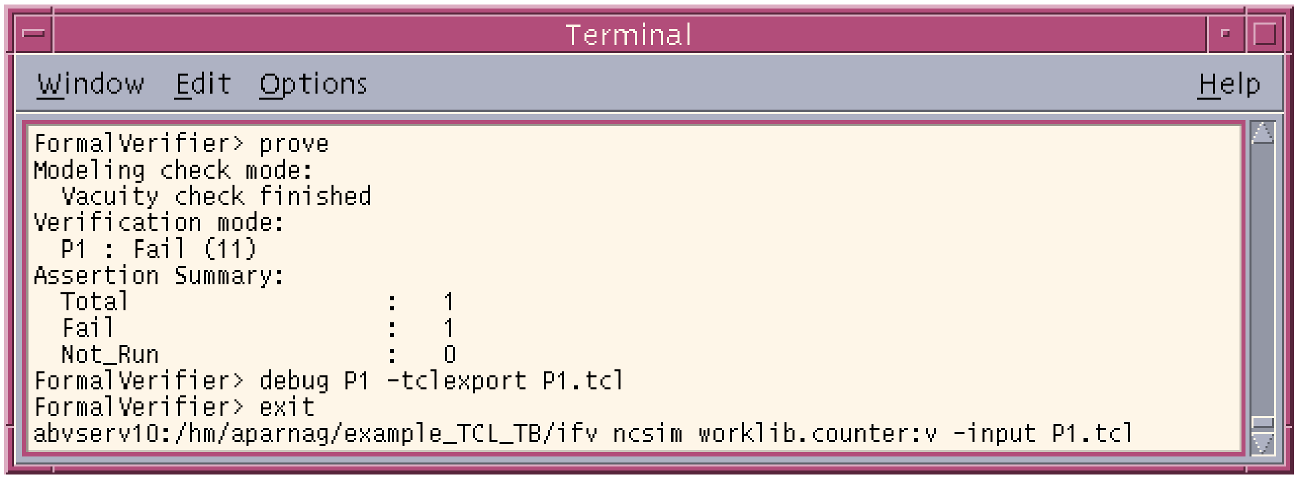


```
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Verification mode:
  P1 : Fail (11)
Assertion Summary:
  Total                  :    1
  Fail                   :    1
  Not_Run                :    0
FormalVerifier> debug P1 -tclexport P1.tcl
FormalVerifier>
```

To run the testbench, exit from the tool and give the following command:

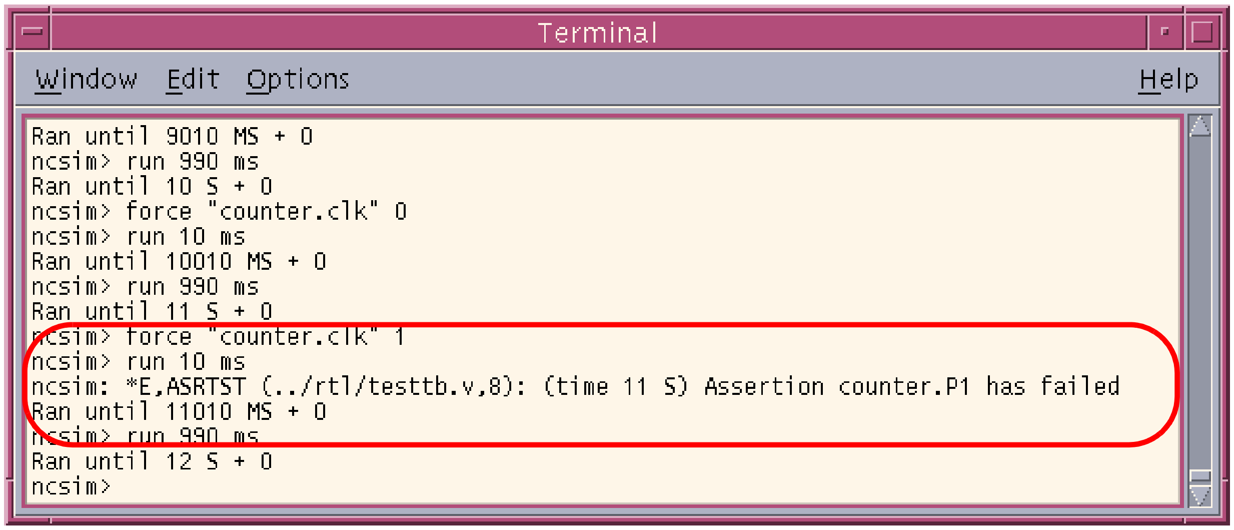
```
% ncsim <snapshot_name> -input <tcl_filename>
```

The image below shows an example of this command:



```
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Verification mode:
  P1 : Fail (11)
Assertion Summary:
  Total                  :    1
  Fail                   :    1
  Not_Run                :    0
FormalVerifier> debug P1 -tclexport P1.tcl
FormalVerifier> exit
abvserv10:/hm/aparnag/example_TCL_TB/ifv ncsim worklib.counter:v -input P1.tcl
```

The highlighted area in the image below shows how the simulator flags the assertion failure:



```
Ran until 9010 MS + 0
ncsim> run 990 ms
Ran until 10 S + 0
ncsim> force "counter.clk" 0
ncsim> run 10 ms
Ran until 10010 MS + 0
ncsim> run 990 ms
Ran until 11 S + 0
ncsim> force "counter.clk" 1
ncsim> run 10 ms
ncsim: *E,ASRTST (../rtl/testtb.v,8): (time 11 S) Assertion counter.P1 has failed
Ran until 11010 MS + 0
ncsim> run 990 ms
Ran until 12 S + 0
ncsim>
```

Tcl Testbench Generation Limitations

- The Tcl testbench works only with the NC simulators of the IUS releases.
- The testbench does not contain any commands to open or probe into any shm/vcd database. You need to add these commands to the .tcl file as per your requirements.
- The testbench recreates the assertion failure in simulation but results may vary due to synthesis/simulation mismatch for certain constructs in the design.

12

Binding Properties Using bind Directive

To facilitate verification without modifying the design, SV allows you to specify properties separately and bind them to specific modules or instances. This is a very convenient mechanism of writing additional HDL/assertions without modifying the design. Using this feature, you can bind a module, interface, or program instance to a module or a module instance.

In SV, a bind directive can be specified in a module, an interface, or a compilation-unit scope.

Bind Syntax

The generic bind syntax is defined as:

`bind` *target bind_obj* `[(`*params*`)]` *bind_inst* `(`*ports*`);`

To know more about the details on the bind syntax, refer to *Chapter 7, “Binding SVA to SystemVerilog and VHDL”* of the *Assertion Writing Guide*.

IFV Use Model for Binding

The typical use model of bind followed in IFV is when you want to add assertions in a design without modifying the design. Consider the following example:

File name: test.v

```
module design_top(clk,a,b);
input clk;
output a,b;
bot11(clk,a,b);
endmodule


module bot(clk,a,b);
input clk;
output a,b;
reg a,b;
always (posedge clk)
```

```
if (a==5'b0)
a=5'b1;
else
a=a<<1;

always @(posdge clk)
b=a;
endmodule
```

In the above example, if you need to verify whether signal `b` follows signal `a` in module `bot`, you will need to write a separate module as follows:

File name: verif_unit.v

```
module verif_unit(clk, la, lb);
input clk, la, lb;
//psl A1: assert always (la -> next(lb)) @posedge clk);
endmodule
```

Instantiate this module in module `bot` as given below:

```
verif_unit verif_inst(clk, a, b);
```

Instantiating this will require the modifications in the module `bot`. Usage of bind directive will allow this instantiation without modifying the module `bot`. The bind statement required will be specified as:

File name: bind_info.v

```
module bind_info();
bind bot verif_unit verif_inst(clk, a, b);
endmodule
```

To compile the design along with bind directive, you will need to pass all the SystemVerilog files to IFV.

In the above example, multiple top modules have been defined, that is, design_top and bind_info. In this case, you will explicitly specify additional tops having bind information using +bind_top option as:

```
ifv test.v verif_unit.v bind_info.v +bind_top+bind_info
```

Note: The extra top modules specified with `+bind_top` should only have bind directives.

IFV also provides an additional use model, which allows you to pass the bind information through the IFV command line option. You can specify bind commands in a text file and pass that file using an `+EXTBIND` option to IFV.

For the example given above, instead of writing the bind directive in a module, write the bind statement in a text file as:

File name: bind_info.txt

```
    bind bot verif_unit verif_inst(clk, a, b);
```

To compile the bind information, use the following command:

```
    ifv test.v verif_unit.v +extbind+bind_info.txt
```

For more information on SVA binding, refer to *Chapter 7, "Binding SVA to SystemVerilog and VHDL*" of the *Assertion Writing Guide*.

13

Using Desktop Option for Enterprise Manager

Desktop Option for Enterprise Manager is a limited-capability version of Enterprise Manager, intended for use by every designer and verification engineer. In Desktop Option, you can perform many day-to-day verification tasks without the need to check out an Enterprise Manager license.

Installation and Environment Setup

The EMGR install is now part of an IFV installation. When you will install IFV, you will be able to see an *EMGR* directory under the *IFV* directory*.*

For step-by-step instructions on installation and environment setup, refer to the *Enterprise Manager Installation and Configuration* in the EMGR online documentation.

Licensing

With Desktop Option for Enterprise Manager, you can perform many day-to-day verification tasks without needing to check out an Enterprise Manager license. For your convenience, Desktop Option displays a GUI button so that you can check out an Enterprise Manager license and access the advanced features without restarting your session.

One **desktop_manager** license key is provided with every IFV license.

Invocation

To invoke Desktop Option for Enterprise Manager, use the **-desktop** switch with the invocation command. For example:

```
    emanager -desktop ...
```

In this case, the GUI buttons for Enterprise Manager’s advanced features are greyed out.

For more information on running the Desktop Option for Enterprise Manager, see *Enterprise Manager Getting Started* in the EMGR online documentation.

14 Assertion Distributor

Assertion Distributor enables you to easily distribute assertions across server farm computers. It shortens runtime by running assertions in parallel. It also enables you to view results in a single verification plan and view waveforms directly from Desktop Option for Enterprise Manager.

Assertion Distributor is enabled by the Desktop Mode for Enterprise Manager and is automatically invoked using Assertion Distributor button from IFV GUI.

Tool and Testcase Setup

Tool Setup

Besides setting up your environment to run IFV, you also need to setup your environment to run Desktop Option of Enterprise Manager. Ensure that both IFV and Desktop Option of Enterprise Manager are located in your path.

Testcase Setup

The following are the requirements for the testcase setup:

- All file references in `.f` files must specify a complete path. Relative paths are not allowed in `.f` files.
  - Consider specifying each file with reference to an environment variable. This will make the testcase easier to relocate in a file structure:

    For example, change any relative file references, such as `./rtl/test.v` to `$TESTCASE/rtl/test.v`, where you define the environment variable TESTCASE (`setenv TESTCASE /home/user/testcase`) to be the full path to the rtl directory. You can also add the setting of additional environment variables to the `env.csh` file for your convenience.
- Ensure that ifv tcl file has a prove command.

Invoking Assertion Distributor from IFV GUI

To invoke Assertion Distributor, run IFV with `+coverage` option and `+gui` added to the command line. For example:

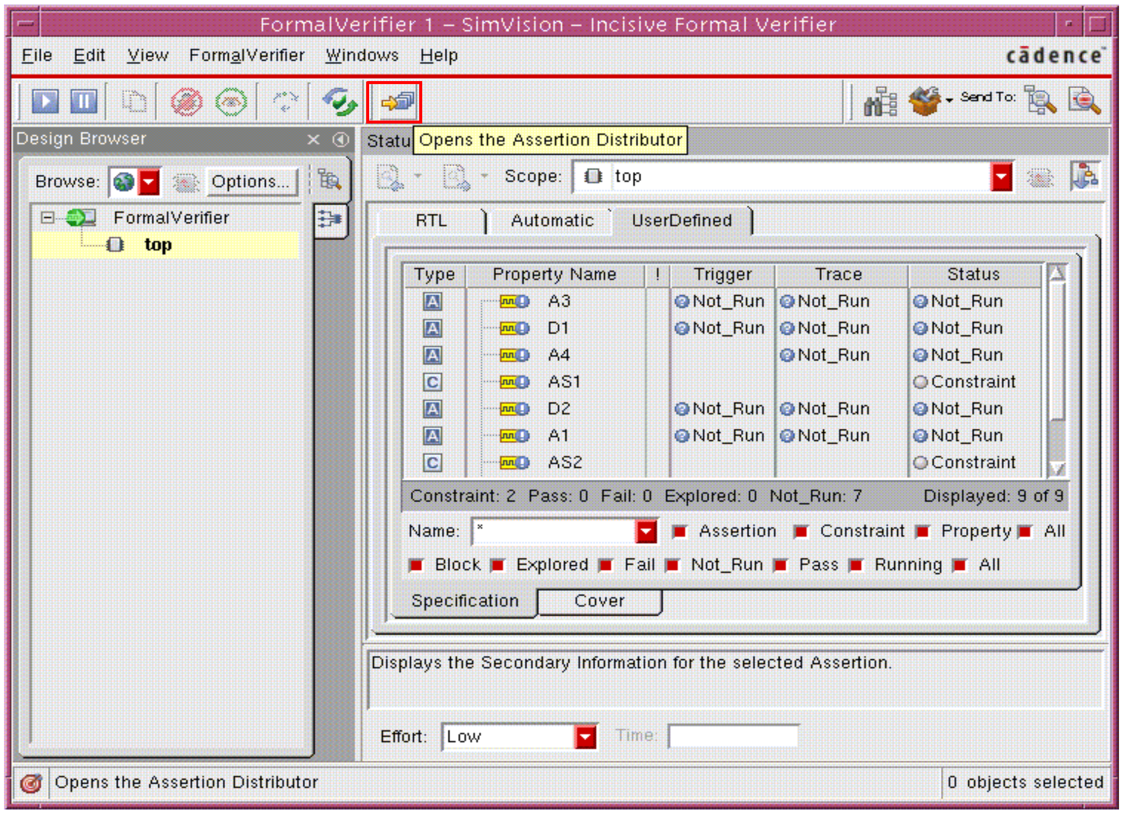
```
ifv -f test.f +coverage +gui
```

Note: Do not mention `+gui` option in a `.f` file. It must be added to the IFV command line. Specifying `+gui` option in a `.f` file will enable all runs to bring up IFV in GUI mode, therefore, invoking multiple windows on your desktop.

For details on `+coverage` option, refer to Chapter 2, "The Single-step Method," of the *Formal Verifier Reference Manual*.

You need not complete the formal analysis and should terminate the current formal verification run by clicking the *Pause* button in the IFV GUI.

Launch Assertion Distributor, from the Incisive Formal Verifier window, by clicking the *Assertion Distributor* button, as shown below:



Scope: top

Type	Property Name	!	Trigger	Trace	Status
A	A3		Not_Run	Not_Run	Not_Run
A	D1		Not_Run	Not_Run	Not_Run
A	A4			Not_Run	Not_Run
C	AS1				Constraint
A	D2		Not_Run	Not_Run	Not_Run
A	A1		Not_Run	Not_Run	Not_Run
C	AS2				Constraint

Constraint: 2 Pass: 0 Fail: 0 Explored: 0 Not_Run: 7 Displayed: 9 of 9

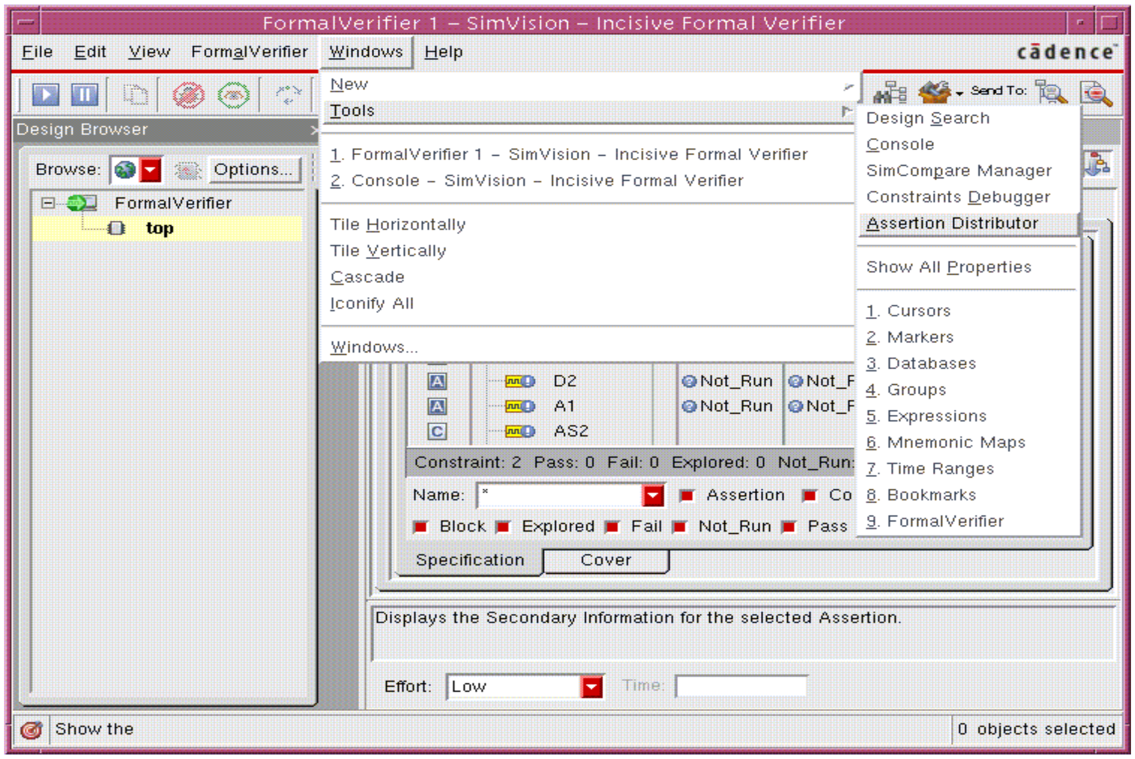
Name: * ☑ Assertion ☑ Constraint ☑ Property ☑ All
☑ Block ☑ Explored ☑ Fail ☑ Not_Run ☑ Pass ☑ Running ☑ All

Displays the Secondary Information for the selected Assertion.

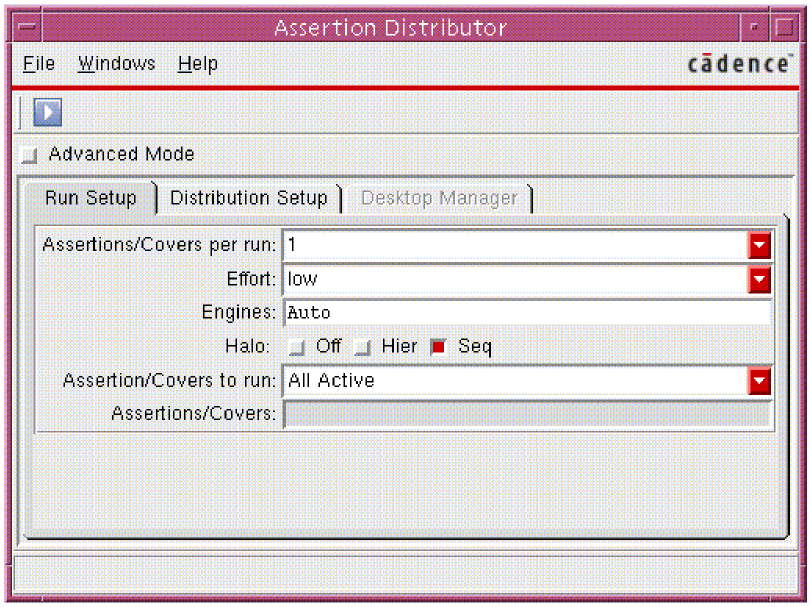
Effort: Low Time:

Opens the Assertion Distributor 0 objects selected

Alternatively, you can select Assertion Distributor from Windows menu bar. Click *Windows -> Tools -> Assertion Distributor*, as shown below:



The *Assertion Distributor* window will be displayed as:



Assertions/Covers per run:	1
Effort:	low
Engines:	Auto
Halo:	Off, Hier, [✓ Seq]
Assertion/Covers to run:	All Active
Assertions/Covers:	

There are two modes in which you can run Assertion Distributor in the GUI:

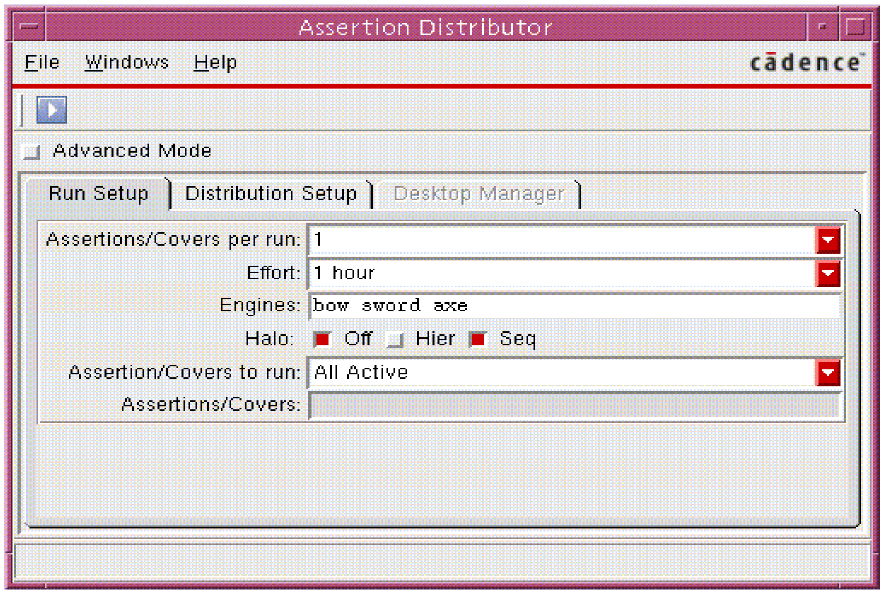
- Basic Mode
- Advanced Mode

Basic Mode

Note: The basic mode is a default mode. Even if you click the *Run* button, without making any updates in the Run Setup and Distribution Setup, the Assertion Distributor will launch the Desktop Mode of Enterprise Manager with default options. You are recommended to select a Distribution Scheme of other than Serial Local or Parallel Local in the Distribution Setup to achieve increased performance. A scheme that leverages your compute farm (LSF, SunGrid, Load Leveler or User Defined) is highly recommended.

To run Assertion Distributor in basic mode:

1. Update the fields of Run Setup tab.



Assertions/Covers per run: 1
Effort: 1 hour
Engines: bow sword axe
Halo: Off Hier Seq
Assertion/Covers to run: All Active
Assertions/Covers:

   a. The *Assertions/Covers per run* drop-down list, enables you to select the number of assertions to be proved per run. By default, the number is set to 1. For example, if you have 10 assertions, the default setting of 1 will create 10 runs, each proving one assertion and cover. If you have the same 10 assertions and you have a setting of 2, it will create five runs, each proving two assertions.

   Note: Each run proves either assertions or covers. As a consequence, if you have one assertion and a one cover to prove and you select two Assertions/Covers per run, two runs will be created, that is, one each for an assertion and cover. Therefore, the total run count you will see can be greater than (assertions+covers)/2 due to the current implementation.

b. The *Effort* drop-down list, enables you to set the effort that the tool spends to verify a single assertion. By default, the effort is the current setting in the IFV run.

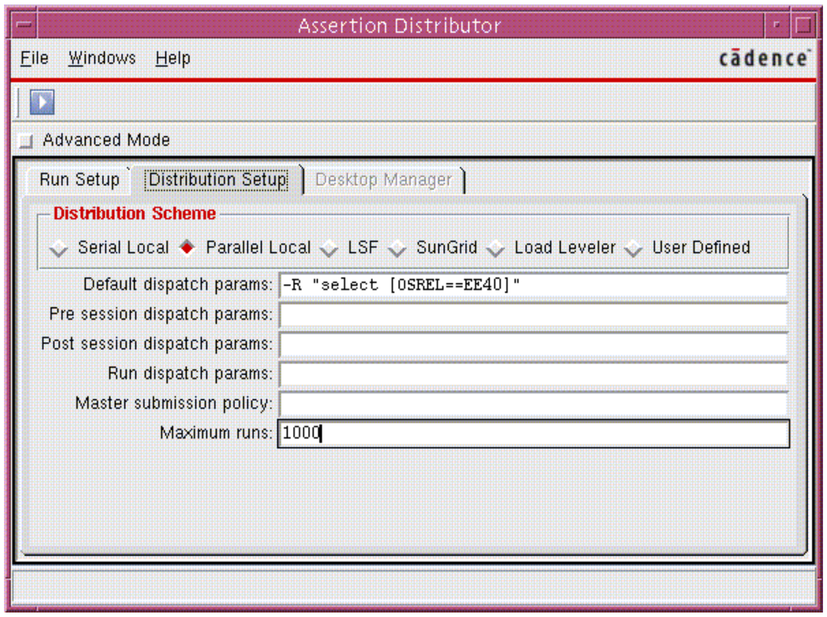
c. The *Engines* field, enables engine selection. By default, the engine is set to the current engine setting in the IFV run. You may have multiple engines set.

d. The *Halo* field, enables halo selection. By default, the halo is set to the current halo setting in the IFV run. However, you may see multiple halos selected.

e. The *Assertion/Covers to run* drop-down list, enables you to select the assertions to run. By default, the option, *All Active* is set. *All Active* assertions/covers are assertions/covers marked with the letter A in the IFV GUI*.* If you select *Not Run*, *Explored,* or *Failed*, then this field will filter the assertions with this status to run. If you select a *Custom List*, then *Assertions/Covers* field gets activated. You can manually enter the path of assertions/covers in the field separating them by spaces. Alternatively, you can also copy the assertions/covers from the IFV window and paste these in the Assertions/Covers field.

Note: When an Assertion Distributor window is displayed at startup, the default options for all the above fields are already set.

2. Update the fields of Distribution Setup tab:



Distribution Scheme

Serial Local [✓ Parallel Local] LSF SunGrid Load Leveler User Defined

Default dispatch params:	-R "select [OSREL==EE40]"
Pre session dispatch params:	
Post session dispatch params:	
Run dispatch params:	
Master submission policy:	
Maximum runs:	1000

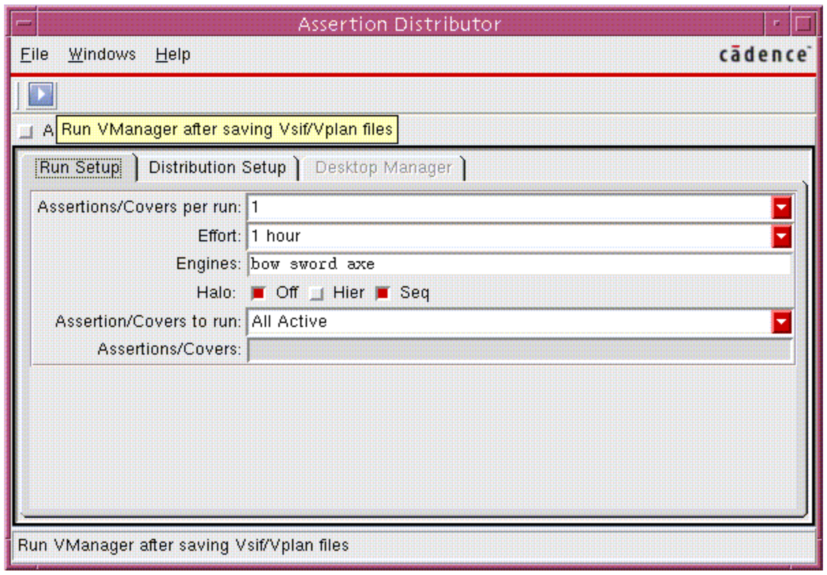
a. Select an option in the Distribution Scheme.

b. Update all dispatch parameters. If you select, *Serial Local* or *Parallel Local* option, you need not update any dispatch params or Master submission policy. For complete details on Distribution Scheme and each of these parameters, refer to section, *Configuring the Distributed Resource Manager (DRM)*, in the *Enterprise Manager Managing Regressions* guide.

   Note: Minimally filling in “Default dispatch params” will serve to apply this setting to all other blank params including “Master submission policy” fields.

Note: Fields in this section will be stored for future sessions, even after exiting the tool.

3. After you have updated Run Setup and Distribution Setup tab, click the *Run* button as shown below:

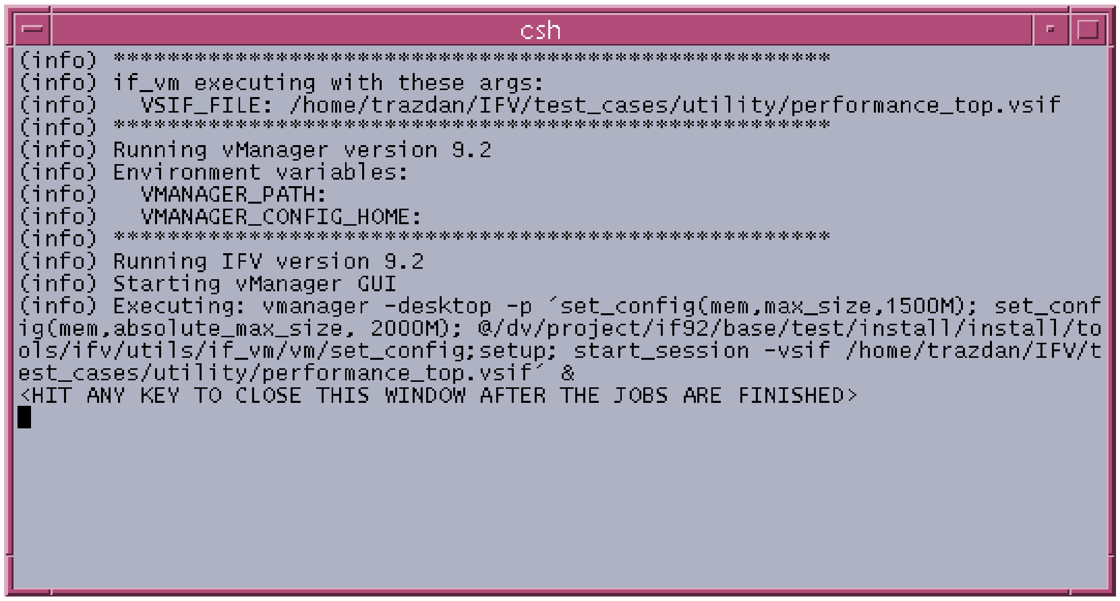


Run VManager after saving Vsif/Vplan files

Assertions/Covers per run:	1
Effort:	1 hour
Engines:	bow sword axe
Halo:	🟥 Off ☐ Hier 🟥 Seq
Assertion/Covers to run:	All Active
Assertions/Covers:	

Run VManager after saving Vsif/Vplan files

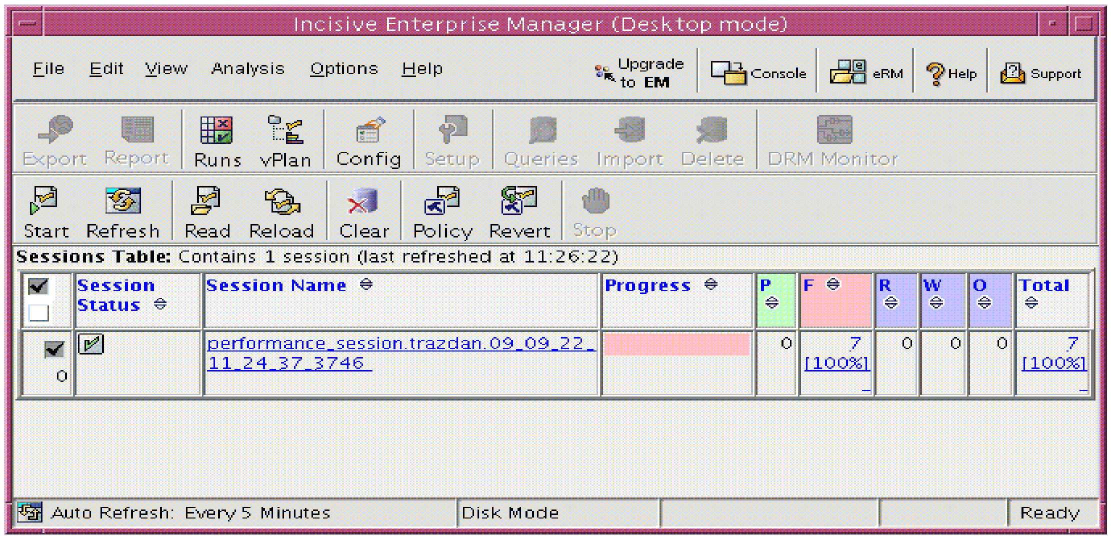
4. When the *Run* button is clicked, the Desktop Option is forked off in an xterm window as shown below:



```
(info) *****************************************************
(info) if_vm executing with these args:
(info)   VSIF_FILE: /home/trazdan/IFV/test_cases/utility/performance_top.vsif
(info) *****************************************************
(info) Running vManager version 9.2
(info) Environment variables:
(info)   VMANAGER_PATH:
(info)   VMANAGER_CONFIG_HOME:
(info) *****************************************************
(info) Running IFV version 9.2
(info) Starting vManager GUI
(info) Executing: vmanager -desktop -p 'set_config(mem,max_size,1500M); set_conf
ig(mem,absolute_max_size, 2000M); @/dv/project/if92/base/test/install/install/to
ols/ifv/utils/if_vm/vm/set_config;setup; start_session -vsif /home/trazdan/IFV/t
est_cases/utility/performance_top.vsif' &
<HIT ANY KEY TO CLOSE THIS WINDOW AFTER THE JOBS ARE FINISHED>
```

Note: Closing this xterm window will terminate the Desktop Manager. Therefore, do not hit any key in this window until your runs are complete.

5. The Desktop GUI of Incisive Enterprise Manager will be launched as:



Sessions Table: Contains 1 session (last refreshed at 11:26:22)

✓	Session Status	Session Name	Progress	P	F	R	W	O	Total
✓ 0	✓	performance_session.trazdan.09_09_22_11_24_37_3746	🟥	0	7 [100%]	0	0	0	7 [100%]

Auto Refresh: Every 5 Minutes | Disk Mode | Ready

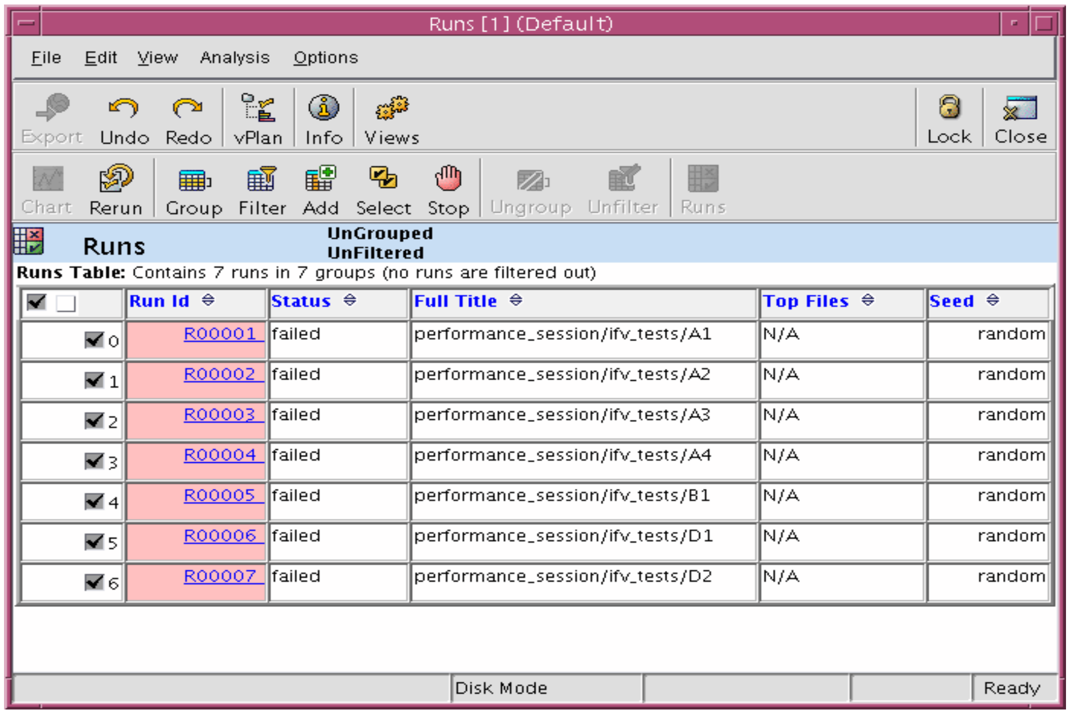
When the *Run* button is clicked in the basic mode, the following steps occur:

■ A vsif and vplan gets generated, which is located in the working directory. The name of the generated vsif/vplan will be *performance_<topmodule>.vsif/.vplan*.

- The generated .vsif file is used to run Incisive Enterprise Manager in Desktop Mode.
- All sessions of Desktop Option are stored under ‘pwd’ /sessions directory.

Viewing Results in Desktop Manager

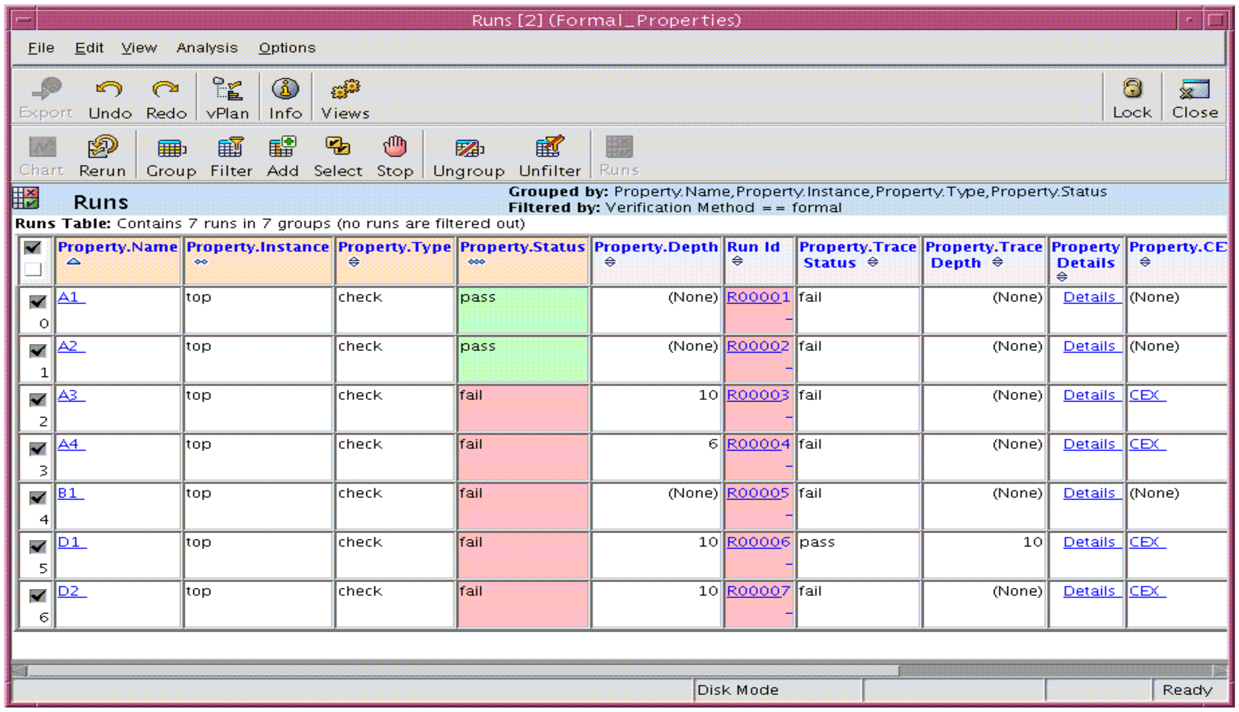
You can view results in *Formal_Properties* window of Desktop Option for Enterprise Manager. Click *Refresh* or wait for an auto refresh cycle and observe completion of all runs. Click the *Total* runs column and observe the Runs view as shown below:


Runs [1] (Default)
File Edit View Analysis Options
Export Undo Redo vPlan Info Views Lock Close
Chart Rerun Group Filter Add Select Stop Ungroup Unfilter Runs
Runs
UnGrouped
UnFiltered
Disk Mode
Ready

Runs Table: Contains 7 runs in 7 groups (no runs are filtered out)

	Run Id	Status	Full Title	Top Files	Seed
✓ 0	R00001	failed	performance_session/ifv_tests/A1	N/A	random
✓ 1	R00002	failed	performance_session/ifv_tests/A2	N/A	random
✓ 2	R00003	failed	performance_session/ifv_tests/A3	N/A	random
✓ 3	R00004	failed	performance_session/ifv_tests/A4	N/A	random
✓ 4	R00005	failed	performance_session/ifv_tests/B1	N/A	random
✓ 5	R00006	failed	performance_session/ifv_tests/D1	N/A	random
✓ 6	R00007	failed	performance_session/ifv_tests/D2	N/A	random

Click *Views ->Formal_Properties* to bring up the Formal_Properties window with all the properties listed along with their status as shown below:



Runs [2] (Formal_Properties)

Runs

Grouped by: Property.Name,Property.Instance,Property.Type,Property.Status
Filtered by: Verification Method == formal

Runs Table: Contains 7 runs in 7 groups (no runs are filtered out)

	Property.Name	Property.Instance	Property.Type	Property.Status	Property.Depth	Run Id	Property.Trace Status	Property.Trace Depth	Property Details	Property.CE
✓ 0	A1	top	check	pass	(None)	R00001	fail	(None)	Details	(None)
✓ 1	A2	top	check	pass	(None)	R00002	fail	(None)	Details	(None)
✓ 2	A3	top	check	fail	10	R00003	fail	(None)	Details	CEX
✓ 3	A4	top	check	fail	6	R00004	fail	(None)	Details	CEX
✓ 4	B1	top	check	fail	(None)	R00005	fail	(None)	Details	(None)
✓ 5	D1	top	check	fail	10	R00006	pass	10	Details	CEX
✓ 6	D2	top	check	fail	10	R00007	fail	(None)	Details	CEX

Advanced Mode

In order to run Desktop Option in an advanced mode, you need to be familiar with Enterprise Manager. Running Desktop Option in an advanced mode provides the following additional capabilities over basic mode:

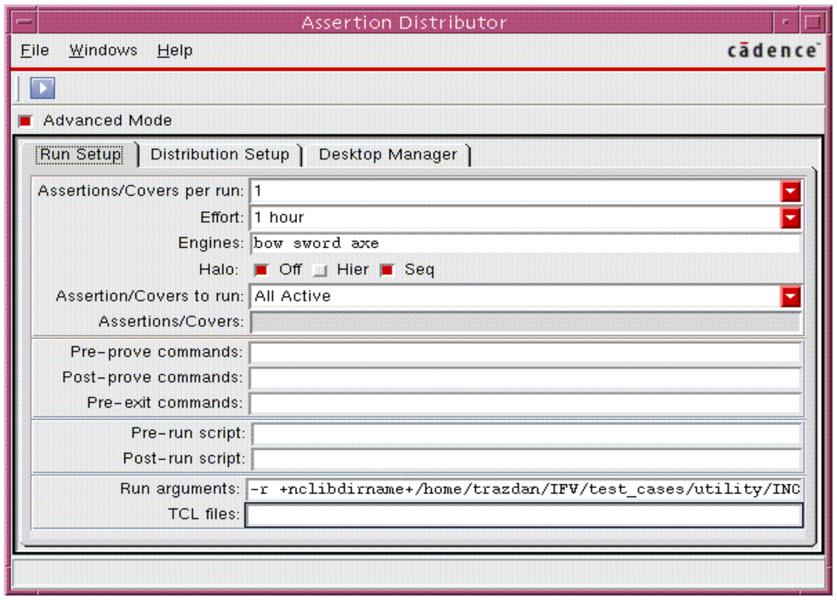
- Allows you to add more attributes to the vsif file.
- Allows you to change default locations and filenames for vsif and vplan files.
- Enables you to leverage the existing compiled snapshot or create a new one.
- Controls availability of waveforms in Desktop Option.
- Enables to create versions of vsifs without launching new Desktop Option sessions.

To run Desktop Option for Enterprise Manager in an advanced mode:

1. Click *Advanced Mode* option in Assertion Distributor window to enable it. You will notice that various advanced attributes are enabled in Run Setup tab. These fields allow you to add the IFV Tcl commands and scripts, which are executed according to the name of the field.

a. *Pre-prove commands*, *Post-prove commands*, and *Pre-exit commands* are Tcl commands to be executed by the IFV run. For example, *Pre-prove commands* specified will be executed before the `prove` command has been initiated.

b. *Pre-run script* and *Post-run script* are Unix scripts or commands to be run before and after the IFV run.

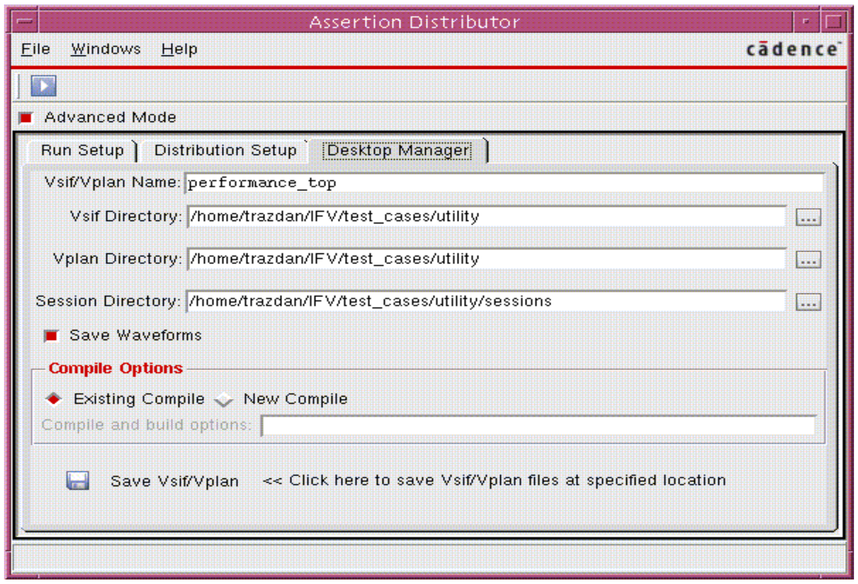


Advanced Mode

Assertions/Covers per run:	1
Effort:	1 hour
Engines:	bow sword axe
Halo:	[✓ Off] Hier [✓ Seq]
Assertion/Covers to run:	All Active
Assertions/Covers:	
Pre-prove commands:	
Post-prove commands:	
Pre-exit commands:	
Pre-run script:	
Post-run script:	
Run arguments:	-r +nclibdirname+/home/trazdan/IFV/test_cases/utility/INC
TCL files:	

Note: All file references specified must mention the full path. Review all file references to ensure that paths are correct.

2. An additional tab, Desktop Manager also gets activated and is displayed as:



Advanced Mode

Vsif/Vplan Name:	performance_top
Vsif Directory:	/home/trazdan/IFV/test_cases/utility
Vplan Directory:	/home/trazdan/IFV/test_cases/utility
Session Directory:	/home/trazdan/IFV/test_cases/utility/sessions

[✓ Save Waveforms]

[✓ Existing Compile] New Compile
Compile and build options:

<< Click here to save Vsif/Vplan files at specified location

a. In the first field, specify the Vsif/Vplan base filename. By default, performance_<top_module_name> is mentioned.

b. Browse locations for Vsif, Vplan, and Sessions Directory.

c. By default, the *Save Waveforms* option will be enabled. If you do not want waveforms to be available, you can disable this option.

d. In *Compile Options* section, an *Existing Compile* option will be selected by default. If you select, *New Compile* option, then *Compile and build options* field will mention the command line to compile the design. Verify that the command line specified is correct.

e. You can save Vsif/Vplan by clicking the *Save Vsif/Vplan* option. This way, you can generate multiple vplan/vsif's with different settings without running the Incisive Enterprise Manager.

3. Run Desktop Manager by clicking the *Run* button.
4. The rest of the process followed is same as the basic mode.

For further information on running in Desktop Option, see *Enterprise Manager Getting Started* in the EMGR online documentation.

Batch Mode Operation

There is an advanced mode capability in which a user can run parallel assertion runs outside the IFV GUI. It involves generating a Vsif from the Assertion Distributor GUI, followed by running the Vsif on the command line. The steps to perform this operation are specified below:

1. Select *Advanced Mode* in the Assertion Distributor window.
2. Select *Desktop Manager* tab and select *New Compile* under Compile Options section.
3. Review all other tabs and make appropriate entries and selections.
4. Return to Desktop Manager tab and click on Save Vsif/Vplan button. This action will create Vsif and Vplan files but will not launch Desktop Mode of Enterprise Manager.

You can optionally exit from the IFV run, as you will not need this session to run the Vsif file in Desktop Mode of Enterprise Manager.

To invoke Desktop Mode of Enterprise Manager and run the jobs defined in the created Vsif file, pass the Vsif file to the `if_vm` script as:

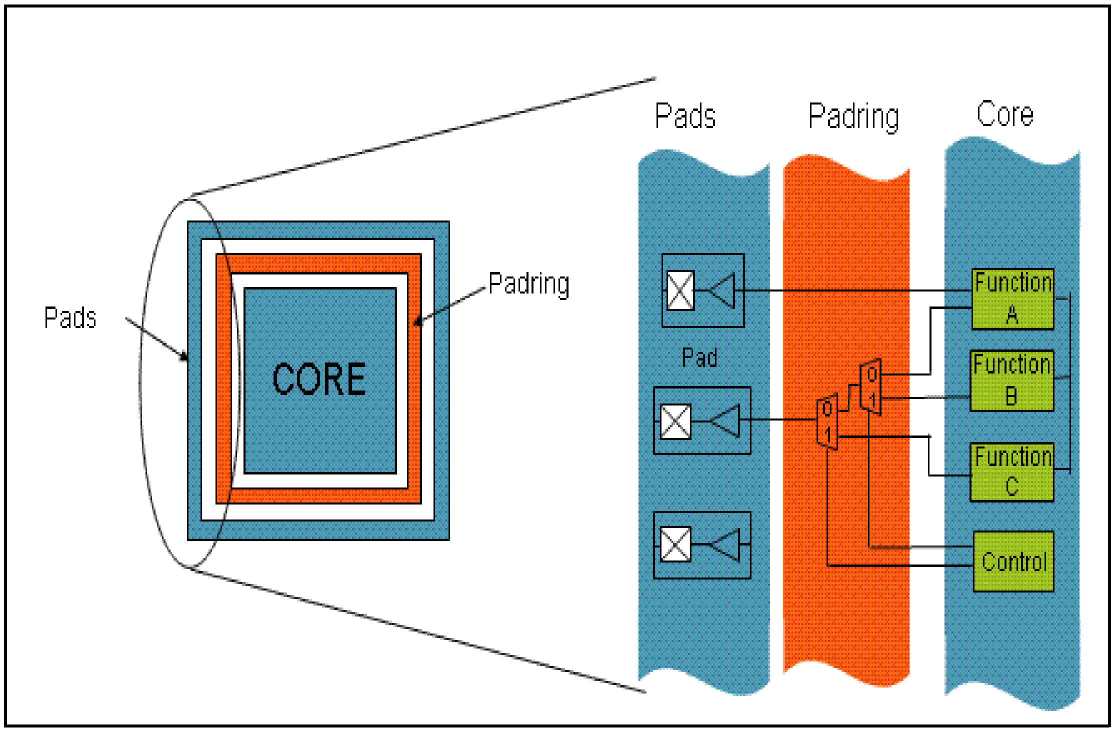
```
if_vm -vsif <vsif filename>
```

You will now see Desktop Mode of Enterprise Manager launched and run the Vsif supplied on the command line.

15

Connectivity Verification

Connectivity refers to the verification of connections between different IPs and between SOCs and the outside world. Connection between IPs test the connection logic at block level, cluster level, or chip level integration. This connectivity checking can be point to point or point to multi-point. Connectivity checking between IPs and the outside world requires a level of multiplexing between the core logic and the pads and is known as *padmux*. A padmux is typically a multiplexing structure. Both IP to IP connectivity and padmux connectivity is shown in the figure below:



Design and verification teams are faced with several challenges surrounding the verification of connections. SOC integration teams are under great pressure to quickly integrate pieces of IP and tape out chips in quick time. The SOC connectivity checking is a dynamic problem impacted by day to day requirement changes and also by implementation considerations, such as, layout limitations and test logic. The two main connectivity challenges in the simulation-based method are controllability and completeness.

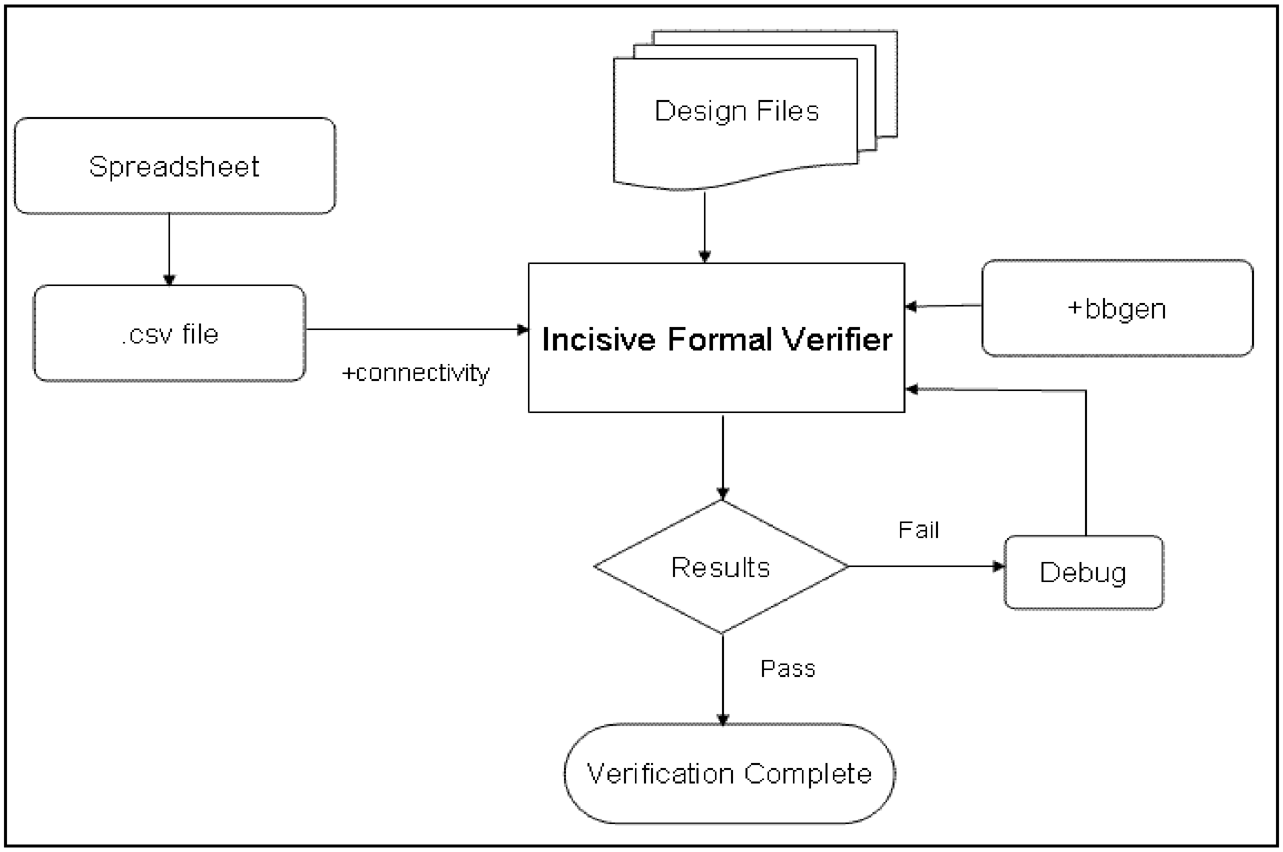
In order to keep pace with the dynamic nature of the problem, you can apply formal analysis verification approach to solve connectivity challenges. You can use IFV to solve these challenges. Controllability challenge can be handled by blackboxing the unnecessary logic and completeness challenge by comprehensive analysis using the set of assertions and formal analysis.

Checking Connectivity Using IFV

Steps to check connectivity using IFV are:

1. The complete connectivity information should be captured in a spreadsheet. For more information on how to capture this data, refer to section, Creating a Spreadsheet.
2. Export the created Microsoft Excel file to .csv file. Depending on the operating system, you can either use OpenOffice or Microsoft Excel to export .csv file. For more information on how to export .csv file, refer to section, Exporting the .csv File.
3. The connectivity information present in the .csv file needs to be converted to a set of assertions. IFV automatically converts the information specified in a .csv file to a set of assertions in a property file.
4. IFV also has a built-in capability to automatically blackbox modules not contributing to the connectivity verification. These modules are identified and excluded from the connectivity verification. If you do not want IFV to automatically blackbox modules, then you need to manually create a blackbox list of modules and pass it to IFV.
5. Launch IFV to verify connections. There are two ways to launch IFV, single-step method and multi-step method. For complete information, refer to section, Verifying Connections in IFV.

The following figure illustrates the connectivity verification flow in IFV:



The subsequent sections describe each of these steps in detail.

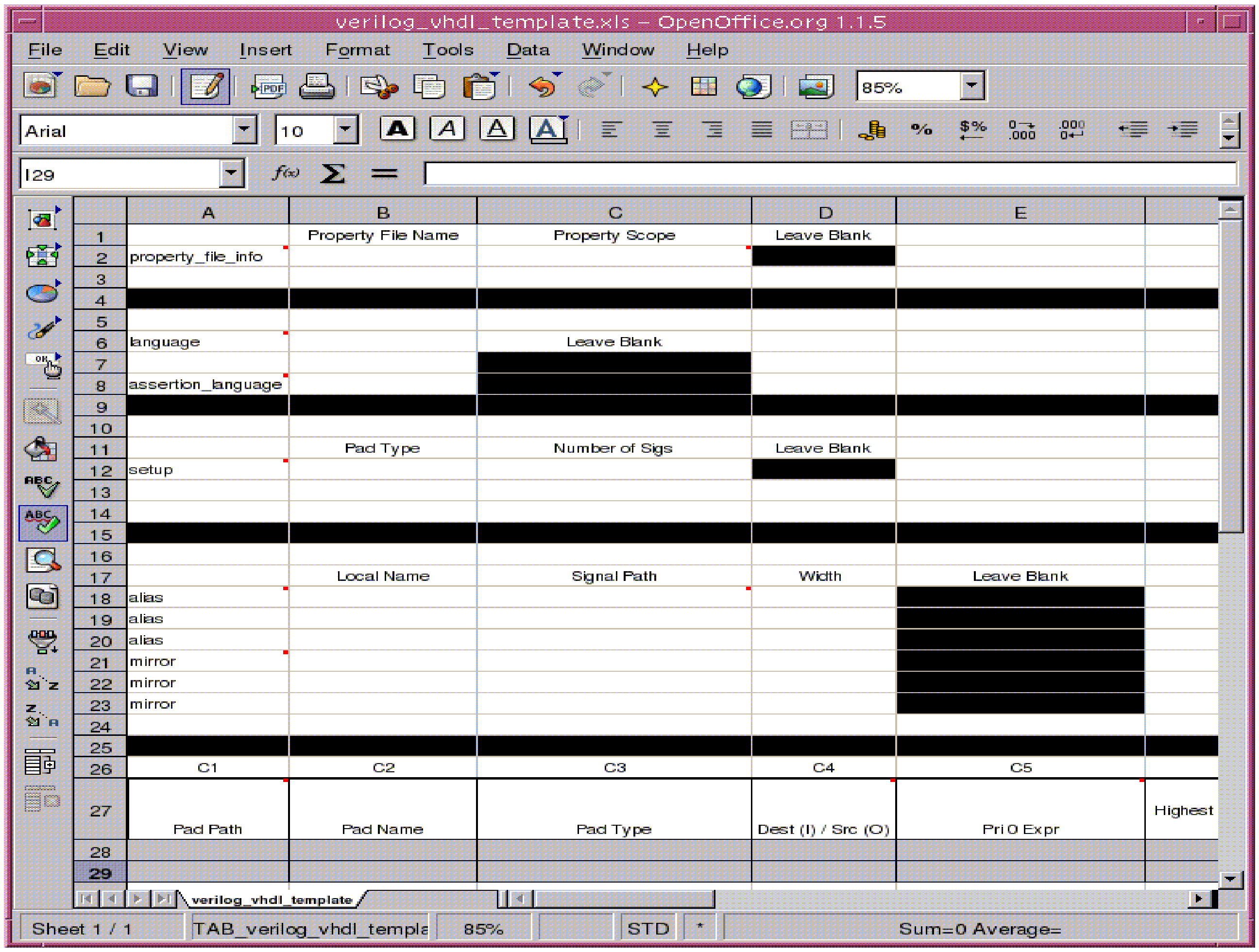
Note: The templates of the spreadsheet and examples are included in the installation. You can customize and reuse these. These can be accessed from:

<*IFV_install_dir*>`/doc/ifvuser/examples/Connectivity`

where, `IFV_install_dir` is the top of your Cadence installation hierarchy.

Creating a Spreadsheet

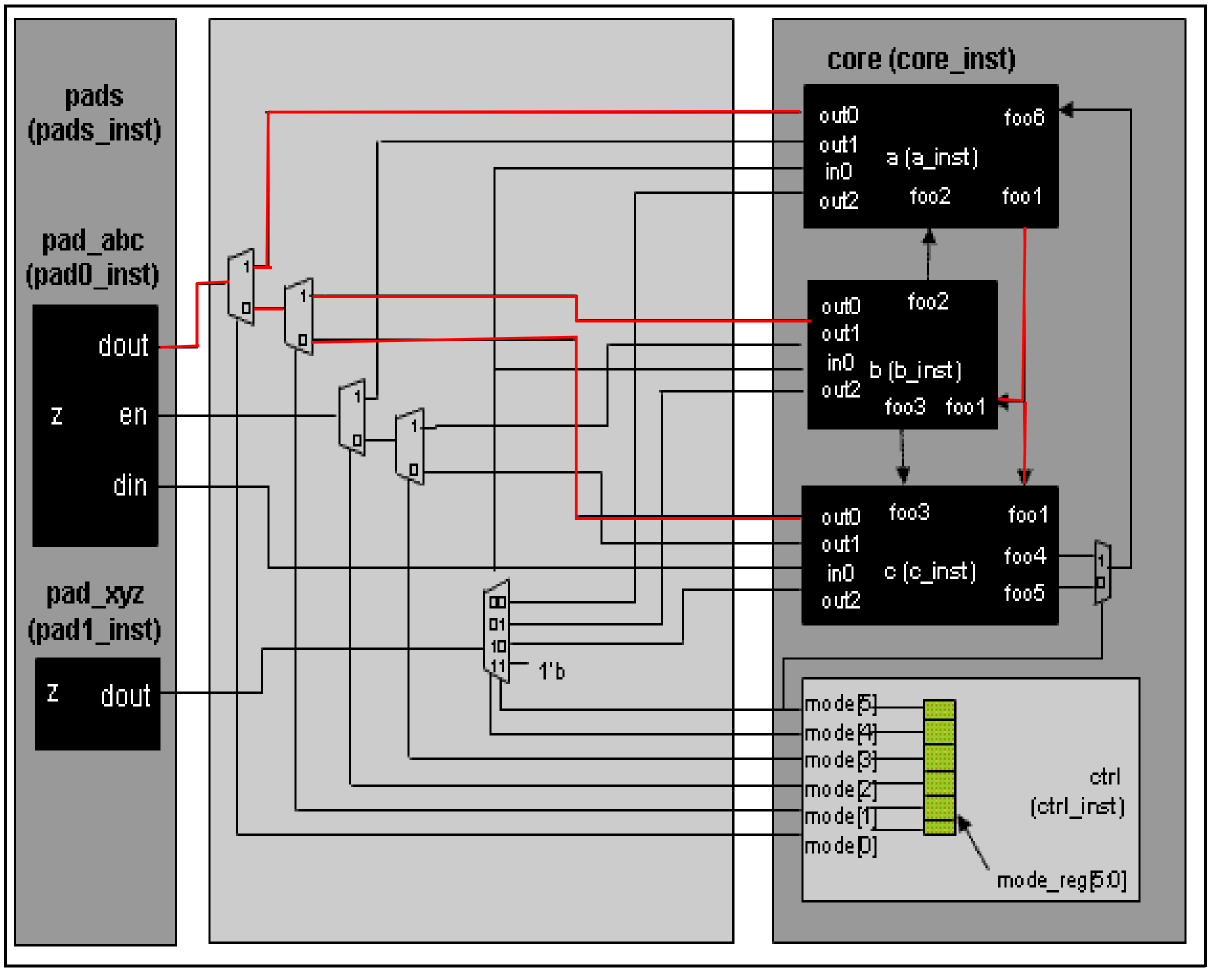
In order to start with the connectivity verification, create a spreadsheet in OpenOffice or a Microsoft Excel. The data in the sheet will represent the set of requirements for generating assertions for each connection. The template available for Verilog and VHDL design is shown as:


verilog_vhdl_template.xls – OpenOffice.org 1.1.5
File Edit View Insert Format Tools Data Window Help
85%
Arial
10
I29
verilog_vhdl_template
Sheet 1 / 1
TAB_verilog_vhdl_templa
85%
STD
Sum=0 Average=

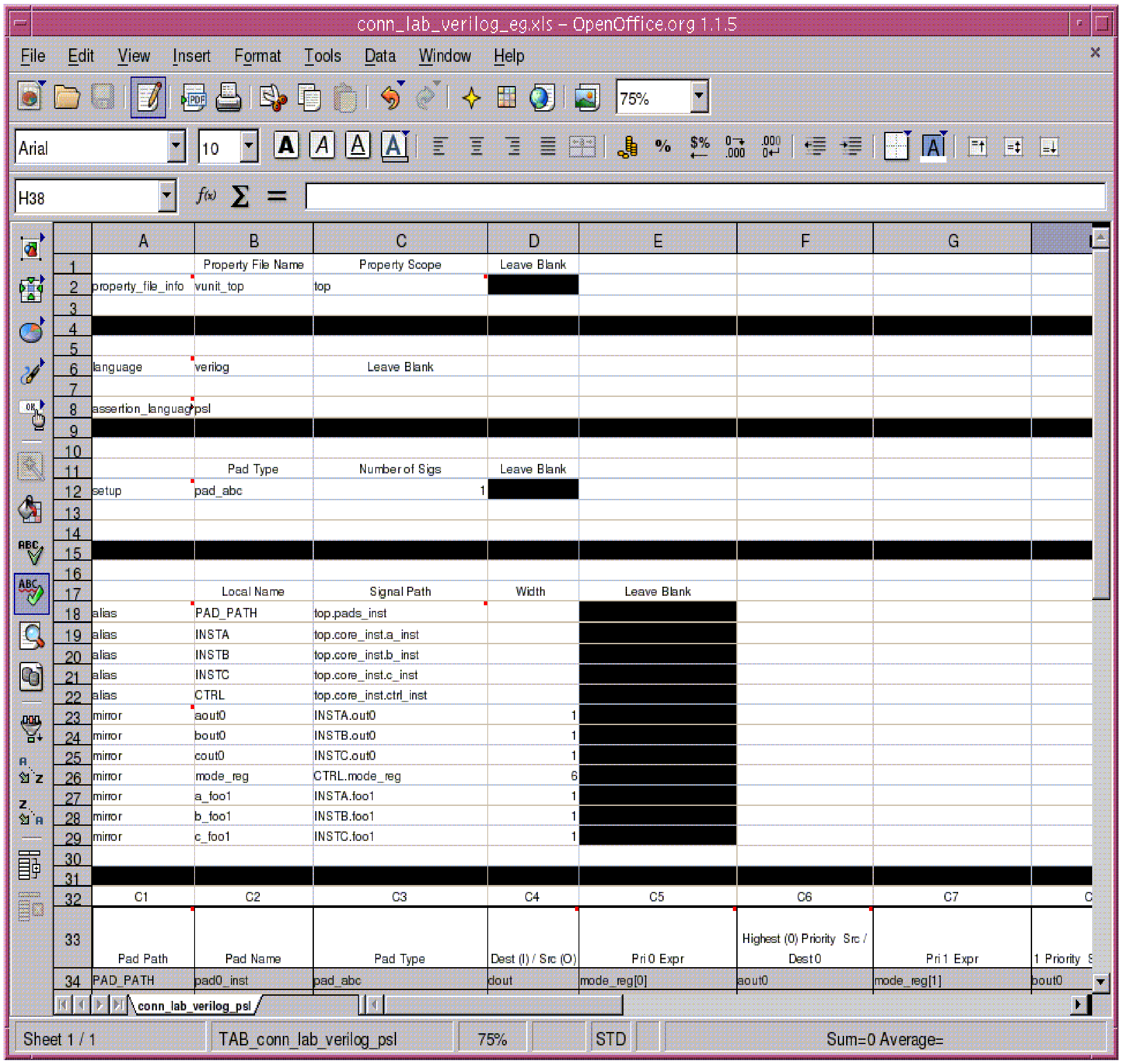
	A	B	C	D	E	
1		Property File Name	Property Scope	Leave Blank		
2	property_file_info			■		
3						
4	■	■	■	■	■	■
5						
6	language		Leave Blank			
7			■			
8	assertion_language		■			
9	■	■	■	■	■	■
10						
11		Pad Type	Number of Sigs	Leave Blank		
12	setup			■		
13						
14						
15	■	■	■	■	■	■
16						
17		Local Name	Signal Path	Width	Leave Blank	
18	alias				■	
19	alias				■	
20	alias				■	
21	mirror				■	
22	mirror				■	
23	mirror				■	
24						
25	■	■	■	■	■	■
26	C1	C2	C3	C4	C5	
27	Pad Path	Pad Name	Pad Type	Dest (I) / Src (O)	Pri O Expr	Highest
28						
29						

Note: The Leave Blank column should be maintained as is and left blank.

Consider a design with possible connections as shown below:



The Microsoft Excel sheet shown below captures the data for the connections highlighted in red in the design. The data in the sheet will represent the set of requirements for generating assertions for each connection.

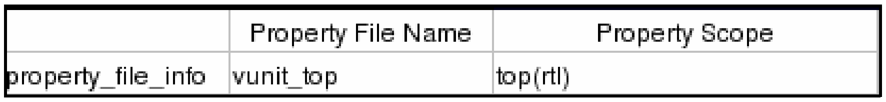

conn_lab_verilog_eg.xls - OpenOffice.org 1.1.5
File Edit View Insert Format Tools Data Window Help
75%
Arial
10
H38
conn_lab_verilog_psl
Sheet 1 / 1
TAB_conn_lab_verilog_psl
75%
STD
Sum=0 Average=

	A	B	C	D	E	F	G	H
1		Property File Name	Property Scope	Leave Blank				
2	property_file_info	vunit_top	top	■				
3								
4	■	■	■	■	■	■	■	■
5								
6	language	verilog	Leave Blank					
7								
8	assertion_languag	psl						
9	■	■	■	■	■	■	■	■
10								
11		Pad Type	Number of Sigs	Leave Blank				
12	setup	pad_abc	1	■				
13								
14								
15	■	■	■	■	■	■	■	■
16								
17		Local Name	Signal Path	Width	Leave Blank			
18	alias	PAD_PATH	top.pads_inst		■			
19	alias	INSTA	top.core_inst.a_inst		■			
20	alias	INSTB	top.core_inst.b_inst		■			
21	alias	INSTC	top.core_inst.c_inst		■			
22	alias	CTRL	top.core_inst.ctrl_inst		■			
23	mirror	aout0	INSTA.out0	1	■			
24	mirror	bout0	INSTB.out0	1	■			
25	mirror	cout0	INSTC.out0	1	■			
26	mirror	mode_reg	CTRL.mode_reg	6	■			
27	mirror	a_foo1	INSTA.foo1	1	■			
28	mirror	b_foo1	INSTB.foo1	1	■			
29	mirror	c_foo1	INSTC.foo1	1	■			
30								
31	■	■	■	■	■	■	■	■
32	C1	C2	C3	C4	C5	C6	C7	C
33	Pad Path	Pad Name	Pad Type	Dest (I) / Src (O)	Pri 0 Expr	Highest (0) Priority Src / Dest 0	Pri 1 Expr	1 Priority S
34	PAD_PATH	pad0_inst	pad_abc	dout	mode_reg[0]	aout0	mode_reg[1]	bout0

There are different keywords associated with this Microsoft Excel sheet. These keywords are used by the script to generate assertions and should not be changed. The description for each of these keywords is given as:

- property_file_info: The information to be captured against this keyword is the Property File Name and Property Scope. Property file captures the details of the automatically generated assertions and the Property File Name is used as the output file name. For the above example, `vunit_top.psl` property file will get generated. Property Scope in the third column specifies the module with which to bind the property. In the above example, `top` has been defined as the property scope.

For VHDL design, you need to mention architecture name along with the `top`. For example:



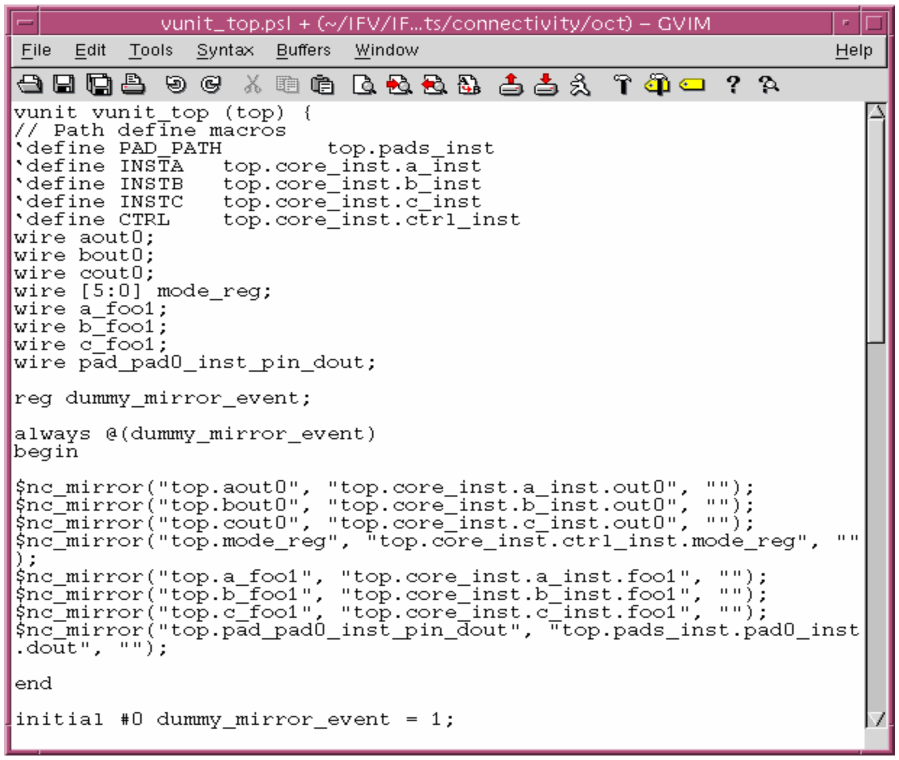
	Property File Name	Property Scope
property_file_info	vunit_top	top(rtl)

- language: The language mentioned here helps to generate the assertions in appropriate HDL syntax. You need to specify the language to indicate to IFV script to generate Verilog or VHDL OOMRs. This option is not case sensitive. For a mixed language design, the language option is the language of the top level design.
- assertion_language: Specifies the assertion syntax to be placed in the property file. Valid options are PSL and SVA. The default assertion language is PSL. SVA is only supported for Verilog language. This option is not case sensitive. SVA does not allow unclocked properties. Therefore, a dummy signal in the generated SVA property file acts as an event for the assertions. SVA assertions are triggered only with the dummy clock signal. The dummy signal is generated in the following format in the SVA property file:

  ```
  wire ifv_connectivity_clk;
  default clocking default_clk @(ifv_connectivity_clk); endclocking
  ```

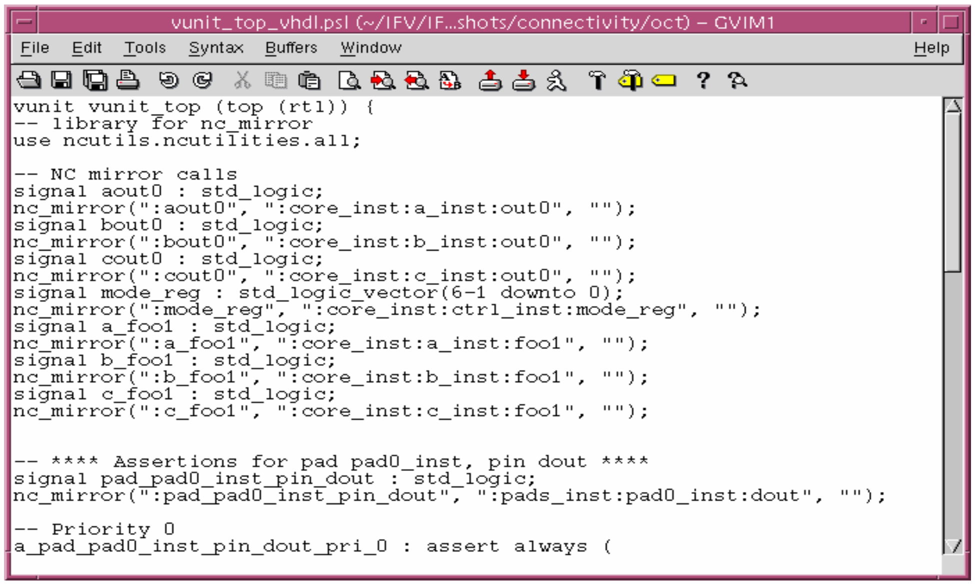
  Note: SVA support cannot be generated with VHDL designs.
- setup: This information is used to define the different types of pad and the associated number of signals to be verified against each Pad Type. The number of signals mentioned against Pad Type should be equal to the number of signals listed in the connectivity information section.
- mirror: The keyword `mirror` is used to specify the hierarchy information in spreadsheet at signal level. The script uses these values to create `$nc_mirror` calls for Verilog design and `nc_mirror` calls for VHDL design to assign OOMR paths to local signal names.

The property file in case of Verilog with assertion language PSL is depicted as:


vunit_top.psl + (~/IFV/IF...ts/connectivity/oct) - GVIM
File Edit Tools Syntax Buffers Window Help

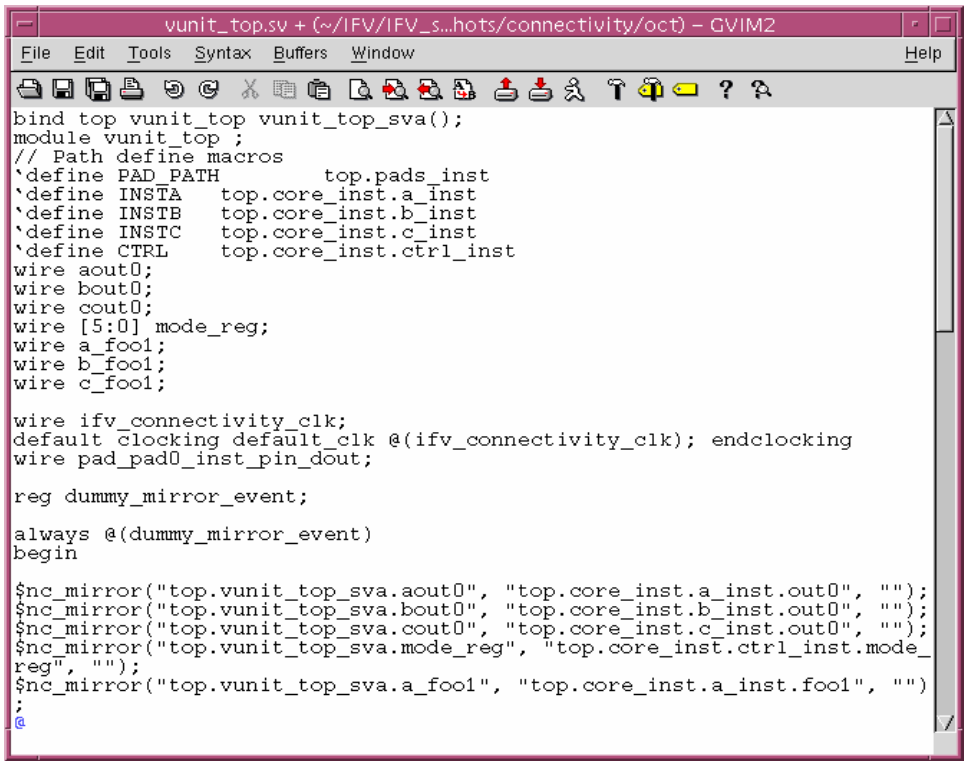
```
vunit vunit_top (top) {
// Path define macros
`define PAD_PATH        top.pads_inst
`define INSTA    top.core_inst.a_inst
`define INSTB    top.core_inst.b_inst
`define INSTC    top.core_inst.c_inst
`define CTRL     top.core_inst.ctrl_inst
wire aout0;
wire bout0;
wire cout0;
wire [5:0] mode_reg;
wire a_foo1;
wire b_foo1;
wire c_foo1;
wire pad_pad0_inst_pin_dout;

reg dummy_mirror_event;

always @(dummy_mirror_event)
begin

$nc_mirror("top.aout0", "top.core_inst.a_inst.out0", "");
$nc_mirror("top.bout0", "top.core_inst.b_inst.out0", "");
$nc_mirror("top.cout0", "top.core_inst.c_inst.out0", "");
$nc_mirror("top.mode_reg", "top.core_inst.ctrl_inst.mode_reg", ""
);
$nc_mirror("top.a_foo1", "top.core_inst.a_inst.foo1", "");
$nc_mirror("top.b_foo1", "top.core_inst.b_inst.foo1", "");
$nc_mirror("top.c_foo1", "top.core_inst.c_inst.foo1", "");
$nc_mirror("top.pad_pad0_inst_pin_dout", "top.pads_inst.pad0_inst
.dout", "");

end

initial #0 dummy_mirror_event = 1;
```

The property file in case of VHDL with assertion language PSL is depicted as:


vunit_top_vhdl.psl (~/IFV/IF...shots/connectivity/oct) - GVIM1
File Edit Tools Syntax Buffers Window Help

```
vunit vunit_top (top (rtl)) {
-- library for nc_mirror
use ncutils.ncutilities.all;

-- NC mirror calls
signal aout0 : std_logic;
nc_mirror(":aout0", ":core_inst:a_inst:out0", "");
signal bout0 : std_logic;
nc_mirror(":bout0", ":core_inst:b_inst:out0", "");
signal cout0 : std_logic;
nc_mirror(":cout0", ":core_inst:c_inst:out0", "");
signal mode_reg : std_logic_vector(6-1 downto 0);
nc_mirror(":mode_reg", ":core_inst:ctrl_inst:mode_reg", "");
signal a_foo1 : std_logic;
nc_mirror(":a_foo1", ":core_inst:a_inst:foo1", "");
signal b_foo1 : std_logic;
nc_mirror(":b_foo1", ":core_inst:b_inst:foo1", "");
signal c_foo1 : std_logic;
nc_mirror(":c_foo1", ":core_inst:c_inst:foo1", "");


-- **** Assertions for pad pad0_inst, pin dout ****
signal pad_pad0_inst_pin_dout : std_logic;
nc_mirror(":pad_pad0_inst_pin_dout", ":pads_inst:pad0_inst:dout", "");

-- Priority 0
a_pad_pad0_inst_pin_dout_pri_0 : assert always (
```

The property file in case of Verilog with assertion language as SVA is depicted as:


vunit_top.sv + (~/IFV/IFV_s...hots/connectivity/oct) - GVIM2
File Edit Tools Syntax Buffers Window Help

```
bind top vunit_top vunit_top_sva();
module vunit_top ;
// Path define macros
`define PAD_PATH        top.pads_inst
`define INSTA   top.core_inst.a_inst
`define INSTB   top.core_inst.b_inst
`define INSTC   top.core_inst.c_inst
`define CTRL    top.core_inst.ctrl_inst
wire aout0;
wire bout0;
wire cout0;
wire [5:0] mode_reg;
wire a_foo1;
wire b_foo1;
wire c_foo1;

wire ifv_connectivity_clk;
default clocking default_clk @(ifv_connectivity_clk); endclocking
wire pad_pad0_inst_pin_dout;

reg dummy_mirror_event;

always @(dummy_mirror_event)
begin

$nc_mirror("top.vunit_top_sva.aout0", "top.core_inst.a_inst.out0", "");
$nc_mirror("top.vunit_top_sva.bout0", "top.core_inst.b_inst.out0", "");
$nc_mirror("top.vunit_top_sva.cout0", "top.core_inst.c_inst.out0", "");
$nc_mirror("top.vunit_top_sva.mode_reg", "top.core_inst.ctrl_inst.mode_
reg", "");
$nc_mirror("top.vunit_top_sva.a_foo1", "top.core_inst.a_inst.foo1", "")
;
@
```

- alias: This information is used to define hierarchical paths to pads. It allows you to define macros to minimize the amount of typing required to explain the connectivity.

  For example:

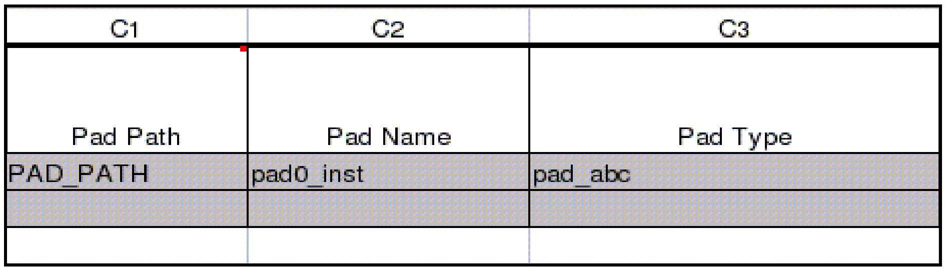
alias	INSTA	top.core_inst.a_inst
mirror	aout0	INSTA.out0

  Note: If you use alias directly in the connectivity section of the spreadsheet in case of Verilog, then use ".".

  The macros and associated paths mentioned here are automatically converted to define statements in the property file, as shown below:

```
vunit vunit_top (top) {
// Path define macros
`define PAD_PATH        top.pads_inst
`define INSTA   top.core_inst.a_inst
`define INSTB   top.core_inst.b_inst
`define INSTC   top.core_inst.c_inst
`define CTRL    top.core_inst.ctrl_inst
```

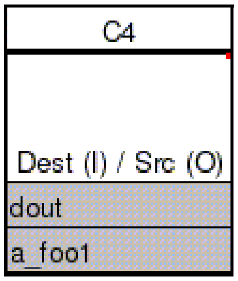
- PadPath: This keyword of the Microsoft Excel sheet populates information on various connections. The columns, 1, 2, and 3 captures information on the pads that are being checked for connectivity. The columns, 1, 2, and 3 are only used for padmux verification and are empty for IP to IP connectivity.



C1	C2	C3
Pad Path	Pad Name	Pad Type
PAD_PATH	pad0_inst	pad_abc

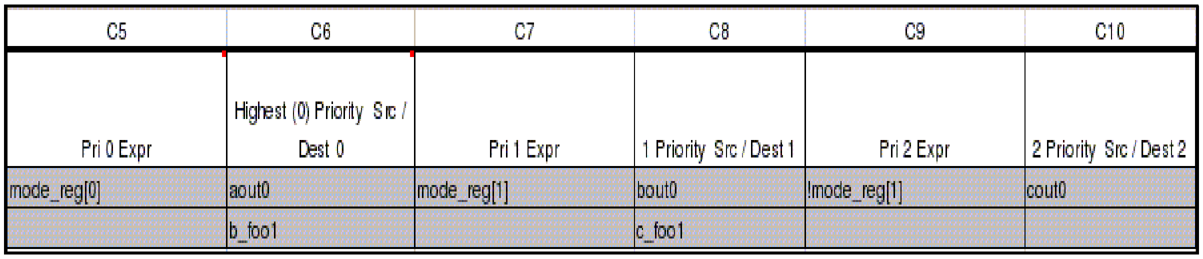
Column1 specifies the path to the parent module of the path. This is typically an alias name. Column 2 is the actual pad instance name and Column 3 specifies the type of pad.

Column 4 mentions the pin name of the source of a point to multi-point connection and destination of the priority multiplexed net. For example, `INSTA.foo1` is the output and `pad0_inst.dout` is the input as mentioned below:



C4
Dest (I) / Src (O)
dout
a_foo1

The entries mentioned in Columns 5 to 10 are shown below:



C5	C6	C7	C8	C9	C10
Pri 0 Expr	Highest (0) Priority Src / Dest 0	Pri 1 Expr	1 Priority Src / Dest 1	Pri 2 Expr	2 Priority Src / Dest 2
mode_reg[0]	aout0	mode_reg[1]	bout0	!mode_reg[1]	cout0
	b_foo1		c_foo1		

Odd columns, 5, 7, and 9 mention the multiplexing control logic associated with the columns to their R.H.S. Even columns, 6, 8, and 10 mention the pin name of the source of a priority multiplexing net and the destination of a point to multi-point connection.

Starting at Column 5, the columns can be used in pairs to define priority muxing.

For example, the first entry in Column 5 specifies the multiplexing control logic associated with Column 6 under Priority 0 as:

```
if ‘CTRL.mode_reg[0] then ‘PAD_PATH.pad0_inst.dout = ‘aout0
```

The first entry in Column 7 specifies the multiplexing control logic associated with Column 8 under Priority 1 as:

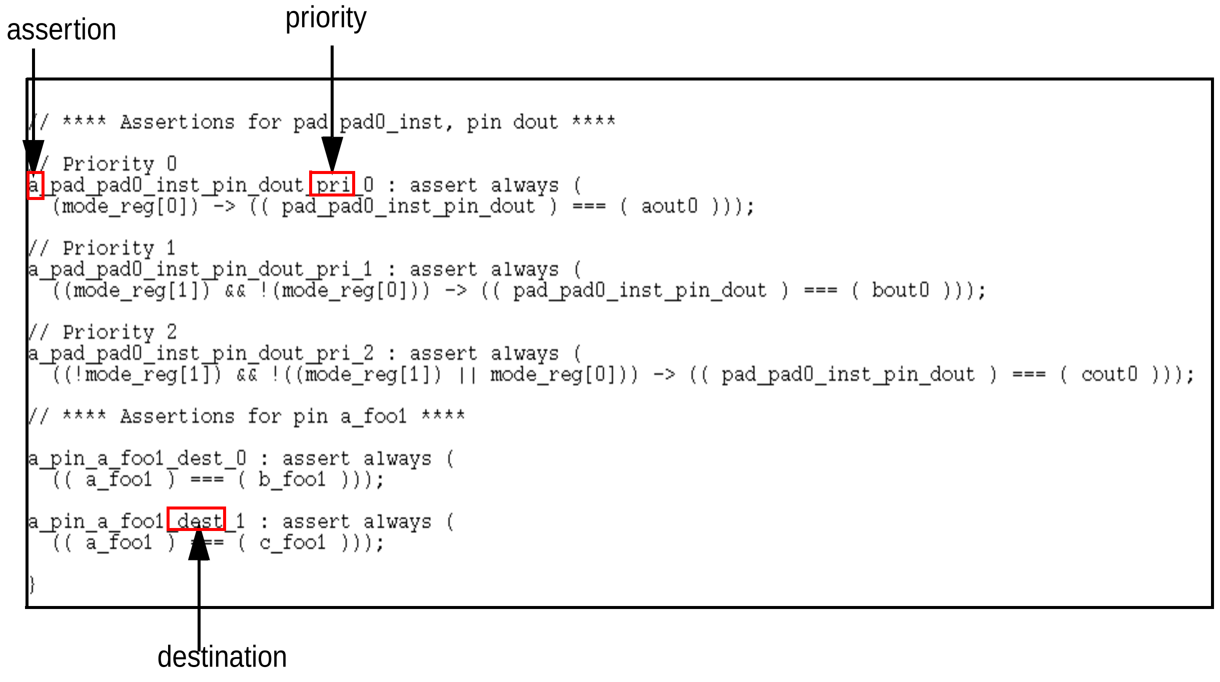
```
if ‘CTRL.mode_reg[1] && !‘CTRL.mode_reg[0] then ‘PAD_PATH.pad0_inst.dout =
‘bout0
```

The first entry in Column 9 specifies the multiplexing control logic associated with Column 10 under Priority 2 as:

```
if !‘CTRL.mode_reg[1] && !(‘CTRL.mode_reg[1] || ‘CTRL.mode_reg[0]) then
‘PAD_PATH.pad0_inst.dout = ‘c.out0
```

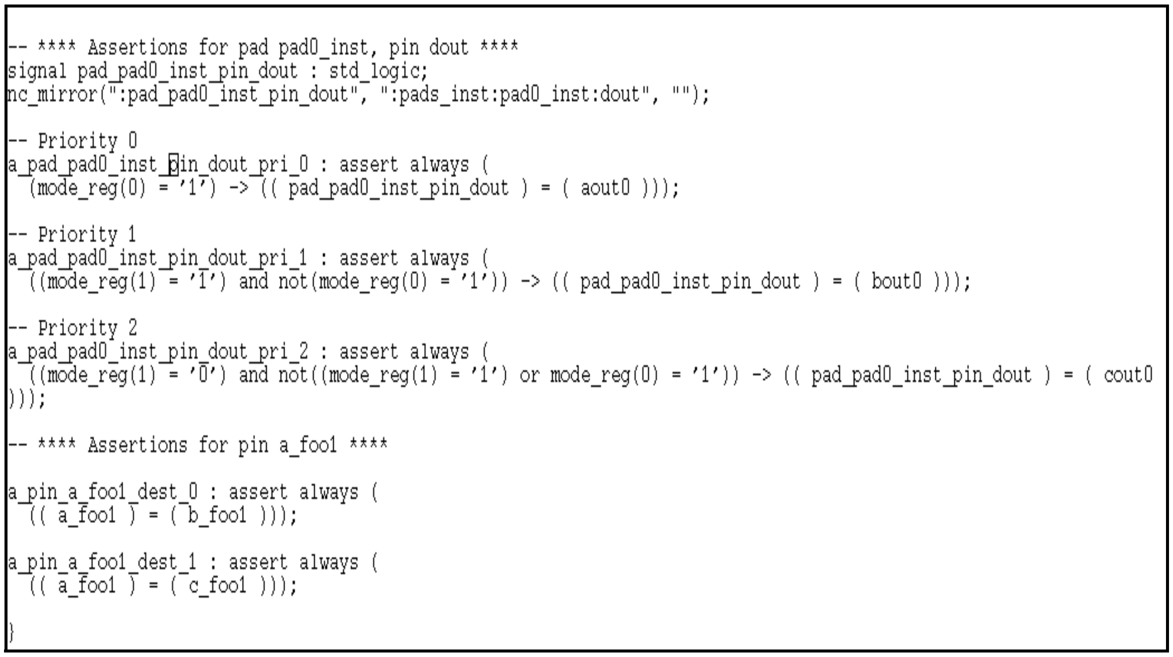
More column pairs can be added as needed.

The assertions in the property file are generated as:



```
// **** Assertions for pad pad0_inst, pin dout ****

// Priority 0
a_pad_pad0_inst_pin_dout_pri_0 : assert always (
  (mode_reg[0]) -> (( pad_pad0_inst_pin_dout ) === ( aout0 )));

// Priority 1
a_pad_pad0_inst_pin_dout_pri_1 : assert always (
  ((mode_reg[1]) && !(mode_reg[0])) -> (( pad_pad0_inst_pin_dout ) === ( bout0 )));

// Priority 2
a_pad_pad0_inst_pin_dout_pri_2 : assert always (
  ((!mode_reg[1]) && !((mode_reg[1]) || mode_reg[0])) -> (( pad_pad0_inst_pin_dout ) === ( cout0 )));

// **** Assertions for pin a_foo1 ****

a_pin_a_foo1_dest_0 : assert always (
  (( a_foo1 ) === ( b_foo1 )));

a_pin_a_foo1_dest_1 : assert always (
  (( a_foo1 ) === ( c_foo1 )));

}
```

If the language defined is VHDL, then the assertions in the property file are generated as:



```
-- **** Assertions for pad pad0_inst, pin dout ****
signal pad_pad0_inst_pin_dout : std_logic;
nc_mirror(":pad_pad0_inst_pin_dout", ":pads_inst:pad0_inst:dout", "");

-- Priority 0
a_pad_pad0_inst_pin_dout_pri_0 : assert always (
  (mode_reg(0) = '1') -> (( pad_pad0_inst_pin_dout ) = ( aout0 )));

-- Priority 1
a_pad_pad0_inst_pin_dout_pri_1 : assert always (
  ((mode_reg(1) = '1') and not(mode_reg(0) = '1')) -> (( pad_pad0_inst_pin_dout ) = ( bout0 )));

-- Priority 2
a_pad_pad0_inst_pin_dout_pri_2 : assert always (
  ((mode_reg(1) = '0') and not((mode_reg(1) = '1') or mode_reg(0) = '1')) -> (( pad_pad0_inst_pin_dout ) = ( cout0
)));

-- **** Assertions for pin a_foo1 ****

a_pin_a_foo1_dest_0 : assert always (
  (( a_foo1 ) = ( b_foo1 )));

a_pin_a_foo1_dest_1 : assert always (
  (( a_foo1 ) = ( c_foo1 )));

}
```

Based on the design, you can specify all entries in the spreadsheet and save it.

After you have identified all connections in a spreadsheet, export that file as a .csv file.

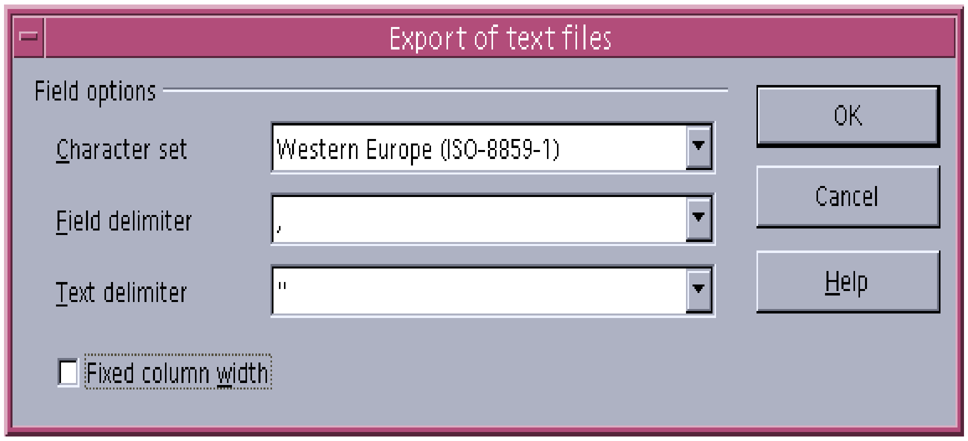
Exporting the .csv File

You can export .csv file from OpenOffice or from Microsoft Excel

Exporting .csv file from OpenOffice

The steps to export .csv file from OpenOffice are:

1. Open the spreadsheet.
2. Select *File -> Save As* from the menu bar.
3. In the *Save As* dialog box, select option, Text CSV (.csv; .txt) from the *File Type* drop-down field.
4. Click *Save*. The dialog box, *Export of text files* is displayed as:



Field options

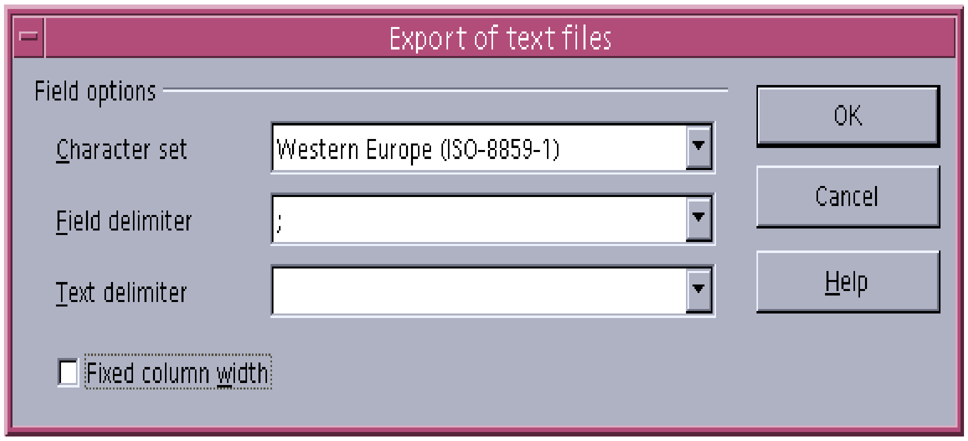
Character set: Western Europe (ISO-8859-1)

Field delimiter: ,

Text delimiter: "

☐ Fixed column width

5. In this dialog box, change the Field delimiter value to “;” and leave the Text delimiter drop-down field blank, as shown below:



Field options

Character set: Western Europe (ISO-8859-1)

Field delimiter: ;

Text delimiter:

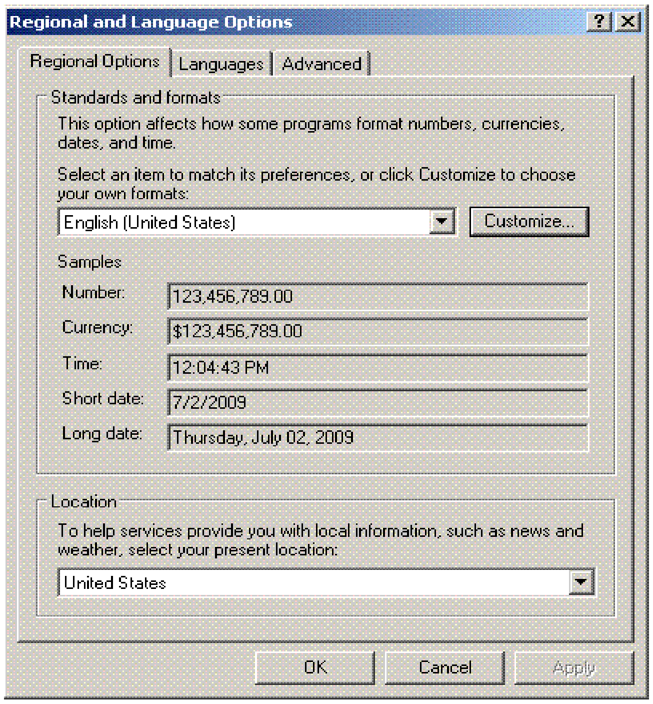
☐ Fixed column width

6. Click *OK*. View the exported *.csv* file.

Exporting .csv file from Microsoft Excel

The steps to export .csv file from Microsoft Excel in Microsoft Windows XP are:

1. Go to *Start -> Settings -> Control Panel -> Regional and Language Options*
2. A *Regional and Language Options* dialog box is displayed as:



Standards and formats

This option affects how some programs format numbers, currencies, dates, and time.

Select an item to match its preferences, or click Customize to choose your own formats:

English (United States)

Samples

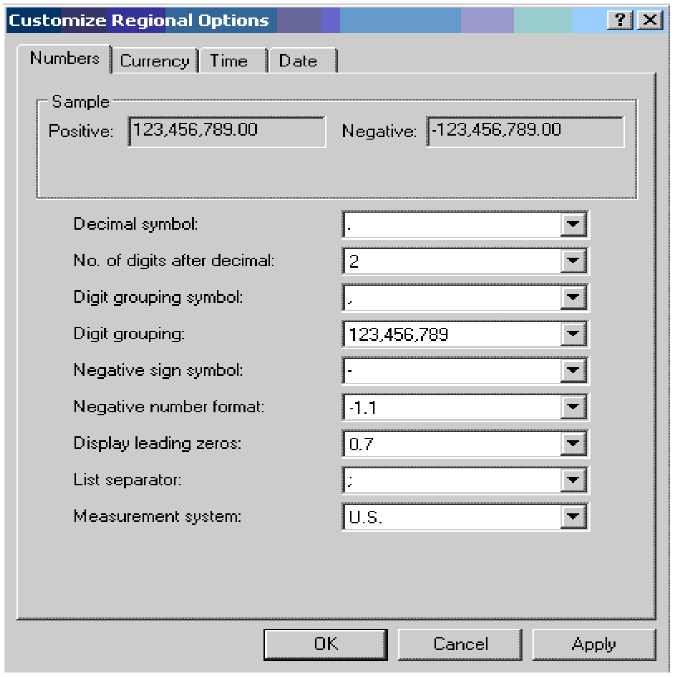
Number:	123,456,789.00
Currency:	$123,456,789.00
Time:	12:04:43 PM
Short date:	7/2/2009
Long date:	Thursday, July 02, 2009

Location

To help services provide you with local information, such as news and weather, select your present location:

United States

3. Select *Customize*. A *Customize Regional Options* dialog box is displayed as:



Sample

Positive: 123,456,789.00 Negative: -123,456,789.00

Decimal symbol:	.
No. of digits after decimal:	2
Digit grouping symbol:	,
Digit grouping:	123,456,789
Negative sign symbol:	-
Negative number format:	-1.1
Display leading zeros:	0.7
List separator:	;
Measurement system:	U.S.

4. Change the List Separator value to “;”.
5. Click *OK*.
6. Now, open Microsoft Excel spreadsheet.
7. Select *File -> Save As* from the menu bar.
8. In the *Save As* dialog box, select option, *CSV (Comma delimited) (*.csv)* from the *save as type* drop-down field.
9. Click *OK*.
10. View the exported *.csv* file.

Verifying Connections in IFV

After a .csv file is created, pass it for verification to IFV. You can run IFV in both 32-bit and 64-bit mode. The two ways to verify connections in IFV are, single-step and multi-step method.

32-bit Mode

- Single-step method: In this method, you can check connectivity using the following command:

  ```
  ifv <options> +connectivity +<filename.csv> +bb_gen
  ```

  The option, `+bb_gen` is optional and is given only when you want to automatically generate the list of blackboxed modules.

  Note: When you specify `+connectivity`, the default option for `define` variable, `drive_system_inout` will be `on`.

- Multi-step method: There are several steps involved in the multi-step method:

  a. Run the connectivity utility, `genConnChecks`, as:

     ```
     genConnChecks <example_vlog.csv>
     ```

     It takes `.csv` file as input and generates `vunit_top.psl` and `connectivity_pairs.txt` file.

  b. Pass the generated `vunit_top.psl` file along with the design files to `irun`.

     ```
     irun -propfile_vlog <vunit_top.psl> -f ifv_top.f -c -access rwc
     ```

  c. Run the `bbgen` utility. This utility takes `connectivity_pairs.txt` file generated by `genConnChecks` as input and creates a list of blackboxed modules.

     ```
     bbgen connectivity_pairs.txt
     ```

d. Pass the created list at the formalbuild level to specify the blackboxed modules.

`formalbuild -bb_list` *snapshot_name*

e. Launch formalverifier and the connections will be verified.

`formalverifier` *snapshot_name*

64-bit Mode

- Single-step method: In this method, you can check connectivity using the following command:

  ```
  ifv <options> +connectivity +<filename.csv> +bb_gen +64bit
  ```

  The option, `+bb_gen` is optional and is given only when you want to automatically generate the list of blackboxed modules.

- Multi-step method: There are several steps involved in the multi-step method:

  a. Run the connectivity utility, `genConnChecks`, as:

  ```
  genConnChecks <example_vlog.csv>
  ```

  It takes `.csv` file as input and generates `vunit_top.psl` and `connectivity_pairs.txt` file.

  b. Pass the generated `vunit_top.psl` file along with the design files to `irun`.

  ```
  irun -64bit -propfile_vlog <vunit_top.psl> -f ifv_top.f -c -access rwc
  ```

  c. Run the `bbgen` utility. This utility takes `connectivity_pairs.txt` file generated by `genConnChecks` as input and creates a list of blackboxed modules.

  ```
  bbgen -64bit connectivity_pairs.txt
  ```

  d. Pass this list at the formalbuild level to specify the blackboxed modules.

  `formalbuild -64bit -bb_list` *snapshot_name*

  e. Launch formalverifier and the connections will be verified.

  `formalverifier -64bit` *snapshot_name*

Structural Connectivity

To verify structural connectivity, the tool checks whether two scalar ports are connected in a hierarchy through port mapping. The ports may be connected through a wire, bit select, or part select of internal vector ports or wires. This check does not consider other criteria for connectivity such as assignments, black boxing, or cut points.

To verify structural connectivity using the tool, specify the `+structural` option to the tool along with the `+connectivity` option.

```
ifv +connectivity <file.csv> +structural
```

The format of the input CSV file is same as in functional connectivity. The tool leverages the source and destination column entries to verify structural connectivity. However, unlike functional connectivity, no assertions are generated for verifying structural connectivity.

Functional and structural connectivity cannot be verified in the same run. Therefore, the tool does not check functional connectivity when invoked with `+structural` option. However, you can use following command to check functional connectivity.

```
ifv +connectivity <file.csv>
```

The structural connectivity checks are viewed using the following command:

```
check –show MODELING_CONNECTIVITY
```

The tool reports three types of messages under MODELING_CONNECTIVITY category.

1. INVNAM for invalid signal names specified in the connectivity information.
2. ILLPRT for signal names that are not scalar by definition.
3. UNCONP for signal names that are not connected as per definition of structural connectivity.

Example:

```
FormalVerifier> check -show
MODELING
   MODELING_STRUCTURAL
     UNCONN
       *E: top.in1.i and top.in2.o are not connected
       *E: top.in3.io1 and top.in2.io2 are not connected
     ILLPRT
       *E: top.vec1 is not scalar port. It is ignored for connectivity check
     INVNAM
      W: Invalid port/signal top.vec2 specified. Verify input specified to the tool
```

Same information is also available in the `Modeling check` tab in GUI.

Limitations

1. Structural connectivity is verified only on scalar ports. These ports can be bit select and may be connected through bit/part select of vector ports.

Example:

- Scalar ports are connected through vector part select.

```
module top ();
    wire [7:0] mbit;
    t1 t1(mbit[7]);
    other other(mbit[7:5]);
endmodule

module t1(x);
    input x;
endmodule

module other(y);
    input [3:1] y;
    wire w, w1;
    t2 t2(y[3],y[1]);
endmodule

module t2 (a, b);
    input a,b;
endmodule
```

In this case `top.t1_i.x` is connected to `top.other_i.t2_i.a`.

- This is vhdl example of connecting bit select of port with scalar.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity top is
end ;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bot is
  port ( in1,in2: in std_logic_vector(3 downto 0) ; out1: out std_logic);
end;

architecture arc of bot is
begin
```

```
end;


architecture arc of top is
      component bot
         port (in1, in2: in std_logic_vector(3 downto 0);
                out1: out std_logic);
      end component;
  signal sub_1, sub_2,sub_3,sub_4:  std_logic;
  signal sub_5 : std_logic_vector (3 downto 0);
  signal out_2: std_logic;

begin

  s2:  bot port map (
                           in1 (3) => sub_1,
                           in1 (2) => sub_2,
                           in1 (1) => sub_3,
                           in1 (0) => sub_4,
                           in2     => sub_5,
                           out1     =>out_2);

  s3:  bot port map (
                           in1 (0) => sub_1,
                           in1 (1) => sub_2,
                           in1 (2) => sub_3,
                           in1 (3) => sub_4,
                           in2     => sub_5,
                           out1     =>out_2);

end ;

 In this case top.s2.in1[0] is connected to top.s3.in1[3]
```

Limitations in Connectivity Verification

- If a protected module is encountered during the connectivity verification by IFV, then the module is blackboxed by the `bbgen` utility.
- If there are any escape characters, such as, backslash(\ ) or a space in the design, then `bbgen` utility will blackbox that design.
- The `bbgen` utility takes `connectivity_pairs.txt` as input. If you do not have a compiled library, that is, all libraries are in a default INCA_lib directories then `bbgen` utility picks the default details and proceeds. If you have a compiled library, for example, `worklib.mylib`, then `bbgen` cannot detect this library by default. In this case, `bbgen` reports that design cannot be read.

  In such cases, pass the following as input:

  ```
  bbgen <connectivity_pairs.txt> -snapshot <snapshot name> -cdslib <path for
  cdslib> -hdlvar <path for hdlvar>
  ```

  This issue exists only with multi-step mode. In single-step mode, tool automatically picks these details.