Verification Continuum™

VC Static Platform User Guide

Version T-2022.06-SP1, September 2022

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Contents

1 Introduction

This document describes the common capabilities of the Verification Compiler ™ (VC) Static Platform tools provided by Synopsys. This includes tasks such as reading and querying the design and handling black boxes. It also defines terms and concepts used through out the platform tools.

Only some of the available options for the VC Static commands are explained in this document. For a complete list of commands with the options and descriptions, see the VC Static Platform Reference Guide. For more information on the usage of the commands and for commands applicable to each VC Static application, see the respective application's user guide.

The VC Static Platform provides comprehensive high-performance and high capacity static verification solutions for functional verification. Formal and static functional verification solutions may be used very early in the design process and do not require complex setups, testbenches. Formal property checking needs properties or assertions to be developed by the user. Formal mathematical techniques are used to test properties or assertions to ensure correct operations of RTL designs. Static checking techniques analyze design intent and structures to check conformance to specified intent an33d behaviors.

2 VC Static Platform

VC Static Verification Platform offers next-generation comprehensive Formal Verification solution (VC Formal) and Low Power verification solution (VC LP). This advanced platform centric seamless integration of various static verification applications provides 3-5x better performance and capacity to enable efficient and effective verification of the largest System-on-Chip (SoC) designs. The best-in-class technology has unique offerings in terms of ease-of-use, compatibility with Synopsys DC and ICC for use model and flows, accuracy of results, precise reporting and advanced debug infrastructure.

VC LP application offers production proven low power verification of the most advanced UPF based designs. VC Formal provides property/assertion checking with unparalleled capacity and runtimes to handle full chip runs. VC Formal offers connectivity checking and formal coverage analysis. All the applications on the VC Static platform share the common look and feel, design intake, Tcl and GUI infrastructure that greatly improves user experience, simplified use models, faster turnaround time and identify design bugs early in the design cycle. The advantages of 3-5x performance and capacity coupled with user productivity are great enablers for earlier verification closure for complex SoCs.

3 Applications

This chapter is organized into the following sections:

3.1 VC LP Checking

VC LP is a multi-voltage, static low power rule checker. It allows engineers to rapidly verify designs that use voltage control-based techniques for power management.

It helps in pipe-cleaning the power intent of the design that is captured in the IEEE 1801 UPF before such intent is used as a golden reference for implementation and other verification tools. VC LP also validates the power intent of the design that is captured in the IEEE 1801 UPF, and verifies the design throughout the implementation flow, from the RTL to the physically routed netlist.

3.1.1 LP Static Checking

LP static checking performs the following checks:

- Detailed checks between the UPF and design to ensure that the UPF is correct and complete.
- Detailed electrical checks on the design to ensure that the design functions accurately in all power states.
- Checks to ensure that the UPF power supply intent is accurately reflected in the design for power or ground routed designs.

3.2 VC Formal

VC Formal offers core model checking capabilities which can be used for multiple formal-based applications. Design inputs can be in RTL (Verilog/VHDL/SystemVerilog) representations, gate-level netlists or a combination of the two. It also requires properties. These can be written in System Verilog Assertions (SVA) or Tool Command Language (TCL) and be embedded in the design files or in separate modules or they can be automatically generated by the tool. VC Formal offers the following applications as part of its capabilities:

1. AEP: Used to automatically extract properties from the design and verify them.
2. CC: Used for checking connectivity between various nodes in the design.
3. SEQ: Used to check cycle-by-cycle equivalence of two RTL designs.
4. FPV: Used for verifying properties in the design.
5. FRV: Used for checking the registers' attributes specification.

6. COV: Used to find unreachable line, condition, toggle and fsm coverage targets.
7. FTA: Used for checking the quality of assertions using fault injection in conjunction with Formal.
8. FSV: Used for checking data integrity and data leakage in the designs.
9. FXP: Used for checking if unknown signal values can propagate through a design
10. DPV: Used for building formal representations of RTL and C++ design blocks, and then analyzing the lemmas for the behavior of those blocks.
11. Navigator: Used to explore the design step-by-step and modify the waveform for one scenario.

4 VC Static Platform Requirements

This chapter is organized into the following sections:

4.1 Platform Support

Red Hat Linux (32 bit and 64 bit) and SUSE Linux (32 bit and 64 bit) platforms are supported.

4.2 Design Language Support

The VC Static Platform requires a synthesizable or synthesized design in one of the following formats:

- Verilog
- VHDL
- Mixed Verilog and VHDL
- SystemVerilog Design
- Gate-level netlist
- Mixed gate-level and RTL
- .db cell libraries

Property language support is described in the *VC Formal User Guide*. UPF requirements are described in the *VC LP User Guide*.

5 Getting Help

This chapter describes the usage of the help command to access online documentation for the VC Static platform.

All commands in the VC Static Platform have online documentation that is accessible from the Tcl command line by entering the help command describing their usage.

The help command by itself returns a list of all available commands grouped by their usage.

```
%vc_static_shell> help
```

The online documentation is also accessible from the help viewer GUI. To get help on a specific command:

- `command -help` lists all the options with their respective explanations for the command.
- `help command` shows a description of the command.

For example, to get help on the help command:

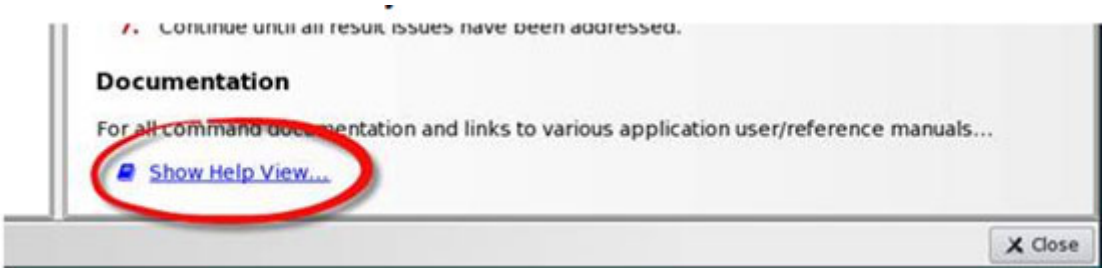
```
%vc_static_shell>help -help
Usage: help    # Display quick help for one or more commands.
[-verbose]              (Display options like -help)
[-groups]               (Display command groups only)
[pattern]               (Display commands matching pattern)

%vc_static_shell>help
# Display quick help for one or more commands.
```

The help viewer GUI is started from the command line using the `view_help` command as shown below:

```
%vc_static_shell>view_help
```

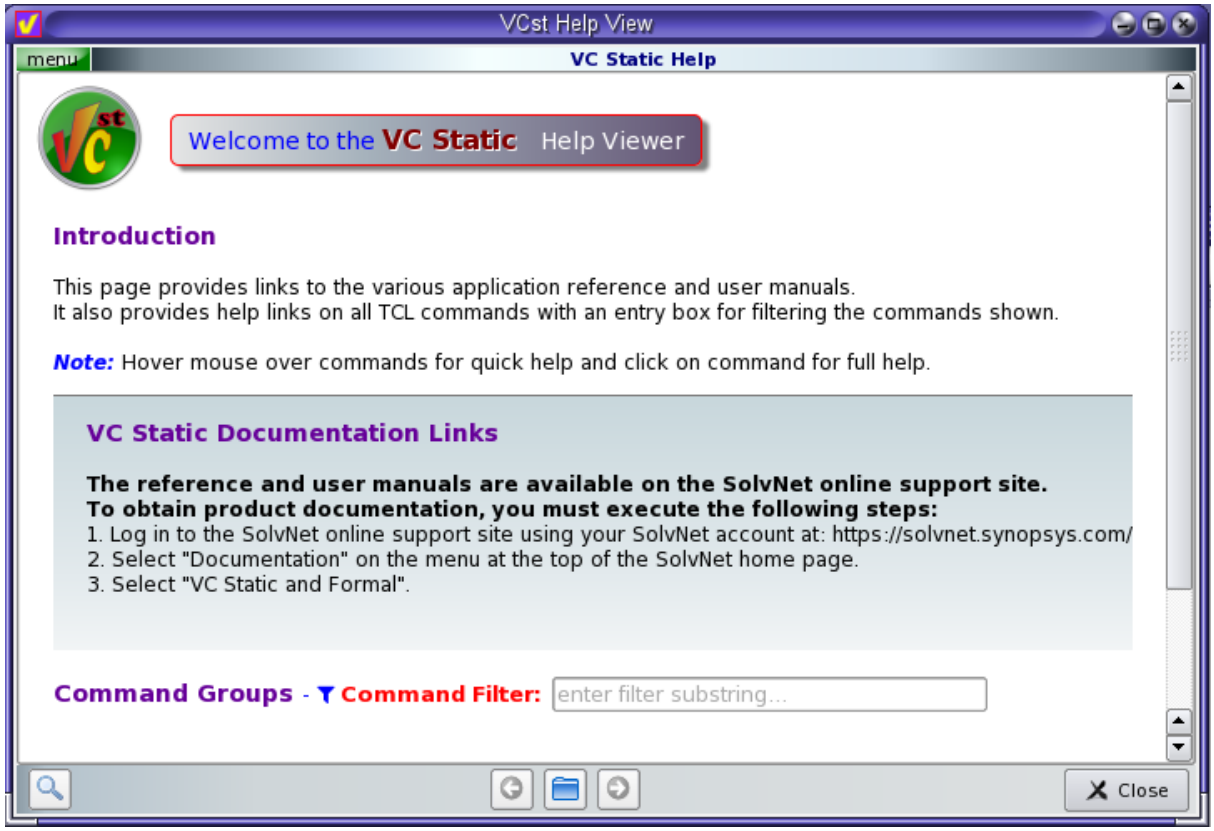
It can also be started from within the activity view GUI as shown in Figure 5-1.

Figure 5-1 VC Static Platform Activity View GUI



This brings up the help viewer which shows all Tcl commands and provides links to each application user guide as shown in Figure 5-2:

Figure 5-2 VC Static Platform Help Viewer



6 Starting VC Static

This chapter describes the different ways to start the VC Static Platform.

The chapter is organized into the following sections:

6.1 Prerequisites for Starting the VC Static Platform

The following are the prerequisites that must be taken care of before starting the VC Static Platform:

- Read and understand the licensing of the product. For more information, see section *Licensing Information* in the *VC Static Platform Product Installation Notes*.
- Set the `VC_STATIC_HOME` environment variable before starting the tool. This should be set to point to the installation directory.
- Add `$VC_STATIC_HOME/bin` to your path.

```
setenv VC_STATIC_HOME /user/bin/install/VC_STATIC_BINARY
setenv PATH $VC_STATIC_HOME/bin:$PATH
```

6.2 Interactive Shell

To start the VC Static Platform with an interactive shell, enter the following command:

```
%vc_static_shell
```

The `vc_static_shell` command has multiple options, for example, `[-mode64 | -full64]` to run in 64 bit mode and `-f` to read commands from a script file. The default prompt is `vc_static_shell`. This can be changed using the command line option `-prompt` to `-prompt vcst`.

6.3 GUI

To start the VC Static Platform with the GUI, type the following command:

```
%vc_static_shell -gui
```

The GUI can also be started from the interactive shell by using the `start_gui` command, as follows:

```
vc_static_shell> start_gui
```

7 Reading the Design

The chapter is organized into the following sections:

7.1 Reading the Design

VC Static Platform has two methods for reading designs depending on the format:

- Verilog and SystemVerilog designs are read using the `read_file` command.
- VHDL and mixed Verilog and VHDL designs are read in two steps, using the `analyze` command first and then the `elaborate` command.

SVA and PSL properties needs to be read at the same time as the design files in VC Formal. Other files are read with dedicated commands in their respective applications. For example, use the `read_upf` command to read UPF files for running LP checks.

The `read_file` command reads designs, libraries and properties into memory. Only one `read_file` command is allowed. To read the design, set the `search_path` and `link_library` Tcl variables and then execute the `read_file` command. The following example is for netlist runs only:

```
set search_path ". ../lib"
set link_library tiny.db
read_file -netlist iso1.v -top top
```

Use the `analyze/elaborate` commands to read the RTL designs as shown in the following code snippet:

```
analyze -format verilog "inv.v top.v"
elaborate top
```

By default, the `read_file`, `analyze` and `elaborate` commands use Design Compiler (DC) syntax for command line options but VCS syntax may also be used by adding the `-vcs` switch. For example to read a Verilog design with SVA properties:

```
read_file -format verilog -sva -top arb -vcs {-sverilog arb.v arb.sva arb_bind.v}
```

To read in a mixed Verilog and VHDL design with SVA properties the `analyze` and `elaborate` commands are used. Multiple analyze commands are allowed but only one elaborate command is allowed.

```
analyze -format verilog -vcs {arb.v arb.sva arb_bind.v}
analyze -format vhdl -vcs {arb.vhd}
elaborate arb -sva
```

VC Static also supports the Design Compiler commands `read_verilog`, `read_sverilog` and `read_vhdl` for reading netlists and library files in Verilog, SystemVerilog, and VHDL respectively.

For more information on how each application of the *VC Static Platform* read in designs, see their respective user guides.

7.2 Black Box Handling

Black boxing certain design modules impact how LP and Formal checks are applied. There may be many reasons why you may want to black box certain design modules. For example, blackboxing helps remove modules from design that are not needed for proving connectivity checks. User defined black boxing can be done using `set_blackbox` command. It needs to be done before `read_design` command is used. For example, to black-box moduleA module, use the following command:

```
set_blackbox -designs {moduleA}
read_file -format verilog -top top top.v
```

When a module is blackboxed all the ports are treated as unconstrained (as inputs only) in the design. When the design is read in, a message is printed for each black-boxed module instance:

```
Note-[SM_BB_SKIP] Skipping blackboxed Module/Entity
```

Undefined modules will be automatically black-boxed as the design is read in. This can be disabled so that the VC Static Platform instead gives an error for undefined modules by setting the following variable:

```
set autobb_unresolved_modules false
```

Note

If there is a complex sequential lib cell without a state table defined in the library, PT black boxes all instances of that lib cell. VC Static cannot black box them since the functionality inside those cells are required for static analysis.

7.3 Ignoring Verilog Modules

You can specify and report the Verilog modules which should be ignored during compilation and design read using the following commands:

Note

This support is available only for Verilog designs.

- `set_ignorebox`: Use this command to specify the Verilog modules to be ignored. Specify the modules with the module names, Verilog file names containing the modules, or directory names with the Verilog files. Specify the `set_ignorebox` command before loading the design.

  Syntax

  ```
  set_ignorebox [-files <file list>] [-directories <directory list>] [<design list>]
  ```

  Where:

  - `[-files <file list>]`: Specify the list of files whose modules should be ignored

- `[-directories <directory list>]`: Specify the list of directories whose modules should be ignored
- `[<design list>]`: Specify the list of modules to be ignored

- `remove_ignorebox`: Use this command to remove Verilog modules from the ignore module list. Specify the modules with the module names, Verilog file names containing the modules, or directory names with the Verilog files.

  Syntax

  ```
  remove_ignorebox [-files <file list>] [-directories <directory list>] [<design list>]
  ```

  The `remove_ignorebox` command reverses the effect of the `set_ignorebox` command.

- `report_ignorebox`: Use this command to get a report the ignored modules and corresponding file or directory list. This command reports all the ignored modules or files, and unused ignore specifications if any.

  The output of the `report_ignorebox` command has four sections: *Ignore Module List*, *Ignore File List*, *Unused Ignore Module Specifications* and *Unused Ignore File Specifications*.

  Notes:

  The `report_link` command reports the ignored modules under the *UserIgnored* category. The report is available in *vcst_rtdb/logs/ignore_summary.rpt*, and contains all the used and unused ignore specifications.

  When the `enable_blackbox_tags` application variable is set to true, the *InfoAnalyzeBBox* violations are reported for all the ignored modules at the end of the design read.

7.4 Managing Violations Through the Violation Life Cycle

VC Static platform provides a violation life cycle for all message and violation tags to provide an easy tracking mechanism of the violations.

All the violations can be grouped in one of the following violation life cycle states:

- *Open*: Default state for all violations
- *Acknowledged*: When the violation has been analyzed, the fix is understood, execution of the fix is needed and no further discussions are required.
- *Waived*: When the violation is analyzed and fix is not required.
- *NeedsInfo*: When the violation is partially analyzed and further discussions are required.
- *Ignore*: When the violation can be ignored.
- *Waive_Temp*: When the violation can be temporarily waived.

You can change the state of the violation using the `set_violation_state` Tcl command. For more details on the `set_violation_state` command, see the man page.

The following snippets shows an example report with the different violation states:

```
vc_static_shell> report_cdc

-------------------------------------------------------------------------------
Management Summary
-------------------------------------------------------------------------------
Stage       Family                  Errors  Warnings     Infos  NeedsInfo  Acknowledged
-----       ----------------------  ------  --------  --------  ----------  ------------
CORRUPTION  RESET                        1         0         0           1             0
SETUP       CLKPROP                      0         0         0           0             1
SETUP       RESET                        0         0         2           0             0
-----       ----------------------  ------  --------  --------  ----------  ------------
Total                                    1         0         2           1             1


-------------------------------------------------------------------------------
Tree Summary
-------------------------------------------------------------------------------
Severity Stage       Tag                                      Count  NeedsInfo  Acknowledged
-------- ----------  ---------------------------------------  -----  ---------  ------------
error    CORRUPTION  RDC_CORRUPT_OBSERVED                         1          0             0
info     CORRUPTION  RDC_CORRUPT_BLOCKED                          0          1             0
info     SETUP       SETUP_CLOCK_PROPAGATED                       0          0             1
info     SETUP       SETUP_RESET_PROPAGATED                       2          0             0
-------- ----------  ---------------------------------------  -----  ---------  ------------
Total                                                             3          1             1
```

The following snippets shows an example report with the required violation states:

```
vc_static_shell> report_cdc -include_viol_state Needs

-------------------------------------------------------------------------------
Management Summary
-------------------------------------------------------------------------------
Stage       Family                  Errors  Warnings     Infos  NeedsInfo
-----       ----------------------  ------  --------  --------  ----------
CORRUPTION  RESET                        1         0         0           1
SETUP       RESET                        0         0         2           0
-----       ----------------------  ------  --------  --------  ----------
Total                                    1         0         2           1


-------------------------------------------------------------------------------
Tree Summary
-------------------------------------------------------------------------------
Severity Stage       Tag                                      Count  NeedsInfo
-------- ----------  ---------------------------------------  -----  ---------
error    CORRUPTION  RDC_CORRUPT_OBSERVED                         1          0
info     CORRUPTION  RDC_CORRUPT_BLOCKED                          0          1
info     SETUP       SETUP_RESET_PROPAGATED                       2          0
-------- ----------  ---------------------------------------  -----  ---------
Total                                                             3          1

1
vc_static_shell> report_cdc -viol_state Ack

-------------------------------------------------------------------------------
Management Summary
-------------------------------------------------------------------------------
Stage       Family                  Errors  Warnings     Infos  Acknowledged
-----       ----------------------  ------  --------  --------  ------------
SETUP       CLKPROP                      0         0         0             1
-----       ----------------------  ------  --------  --------  ------------
Total                                    0         0         0             1


-------------------------------------------------------------------------------
Tree Summary
-------------------------------------------------------------------------------
Severity Stage       Tag                                      Count  Acknowledged
-------- ----------  ---------------------------------------  -----  ------------
info     SETUP       SETUP_CLOCK_PROPAGATED                       0             1
-------- ----------  ---------------------------------------  -----  ------------
Total                                                             0             1
```

7.5 Using the report_link Command

The VC Static platform is equipped to identify empty modules, black boxes, modules that do not have a definition and those that do not qualify the synthesis stage. You can do this using the Tcl command `report_link`.

7.5.1 Use Model

```
report_link [-sim_match]
   -sim_match: include library cells where a simulation model is given
```

The output of the `report_link` command includes module names where both simulation definition and liberty definition are given. However, the port name, port width and port direction specified in an instance connection does not match with ports specified in a liberty definition. In this case, the liberty definition is discarded and the simulation definition is used for those instances. When the `-sim_match` option is specified, the output of the `report_link` command includes all modules that have simulation and liberty definition included.

Example:

```
report_link
Module      Instances     Reason
------      ----------    ------
Module1        3          SimPort*, Synthesis
Module2        1          SimDirection*
Module3       17          Empty
Module4       11          AutoBB, Synthesis
Module5       12          SimWidth*, SimDirection*, SimPort*
```

The command `report_link` reports issues for the following reasons:

Table 7-1 The report_link Reasons Table

report_link Reasons	Description
Empty	Modules definition is empty, that is no wire or signal in module definition has a driver.
Synthesis	Any part of the module is left out due to non-synth constructs or unsupported constructs, even if the majority of the module is synthesized successfully. The entire circuit is black-boxed due to tool support or if the module is detected as simulation memory.
UserBB	Module is black-boxed if a set_blackbox command is issued on this module.
AutoBB	Module is black-boxed giving synthesis as a reason.
SimPort*	Both liberty and simulation definition appear; the simulation definition has been kept for some of its instances because the port name of a port in instance connection does not match with any ports in the liberty definition.
SimDirection*	Both liberty and simulation definition appear; the simulation definition has been kept for some of its instances because the port direction of a port in instance connection does not match port direction of corresponding port specified in the liberty definition.
SimWidth*	Both liberty and simulation definition appear; the simulation definition has been kept for some of its instances because the port width of a port in instance connection does not match port width of corresponding port specified in the liberty definition.
DirtyData*	Module definition is picked after doing dirty data handling for some of its instances in design.
SimMatch*	Both liberty and simulation definition appear; the liberty definition has been kept because the port list, directions and widths match exactly for some of its instances.
Unresolved	There is no definition for the module among any of the Simulation definition or liberty files.

Note
STAR (*) marked on report_link reason indicates this reason is instance specific and you need to look into the warning or error messages to know instance specific details.

7.6 DesignWare Components

The VC Static Platform supports designs that include the Synopsys DesignWare (DW) components. The DW source libraries shipped by Synopsys contain two versions of the DW parts.Therefore, additional steps are needed to ensure the correct version is used. The following are the two versions behavioral models written in Verilog and used for simulation. If used in a VC Static application, the behavioral code is surrounded by Synopsys translate off/on pragmas that result in an empty model or black box.

Encrypted source files written in Verilog/VHDL. These are synthesizable and can be used within Synopsys applications. These parts may be hierarchical and use other encrypted DW subparts. To use these parts in the VC Static flow, all of the parts referenced by a DW part must be analyzed prior to elaboration.

Supporting DW in the VC Static platform is a two step approach.

- Pre-analyze the DW library as part of the VC Static installation. This is done once per customer site.
- Compile designs with DW using the pre-analyzed library.

The following Tcl variables needs to be set prior to using DW components:

Table 7-2 Tcl Variable for DW Components

Tcl Variable	Type	Default Value	Descriptions
hdlin_dwroot	String	“”	DW source tree
vsi_dwroot	String	“”	

Pre-analyzing the DW library allows users to specify the location of the library in their VC Static compile flow without having to reanalyze the parts each time. Pre-analyzing the library is done using the `dw_analyze` command.

7.7 Support for Synopsys SLR

Synopsys Live recording (SLR) support through SIGUSR2 Linux signal is supported.

Use Model

1. Set the following environment variable

   ```
   setenv SNPS_SLR_RECORD_FILE <path to a file with write permission>
   ```
2. Invoke VC SpyGlass, VC LP or VC Formal.
3. To start the SLR, from another terminal execute the following command

   ```
   kill -s USR2 <PID of SVI process>.
   ```
4. Execute the tool commands for recording.
5. To stop the SLR, from another terminal execute the following command:

   ```
   kill -s USR2 <PID of SVI process>
   ```
6. Run the `report_slr_status` command to confirm if SLR is not running.

   Finally, the SLR recording is saved to the file pointed by `SNPS_SLR_RECORD_FILE` environmental variable.

8 Design Query Commands

This chapter describes access commands, and attributes supported in the VC Static Platform for the various design objects. These commands operate using collections in a similar manner as Design Compiler (DC) and PrimeTime (PT). The chapter is organized into the following sections:

- Terminology Used in the VC Static Tools
- Design Query Commands
- Using Collections
- Using Attributes

8.1 Terminology Used in the VC Static Tools

This section describes the terminology used in the VC Static tools.

8.1.1 Designs

Designs are circuit descriptions that perform logical functions. They are described in various design formats, such as VHDL or Verilog HDL. Logic-level designs are represented as sets of boolean operators. Gate-level designs, such as netlists, are represented as interconnected library cells. Designs can be flat or hierarchical.

8.1.1.1 Flat Designs

Flat designs contain no sub-designs and have only one structural level. They contain library cells, nets and ports.

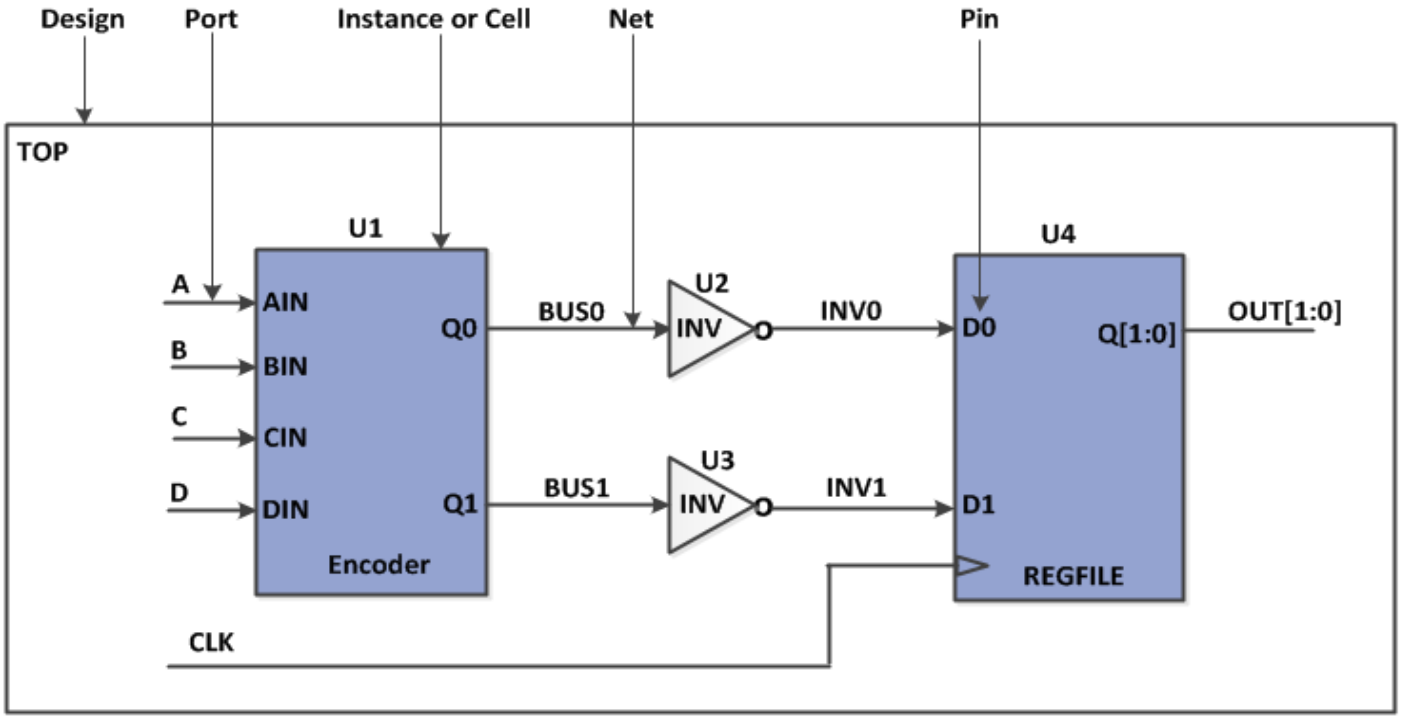
8.1.1.2 Hierarchical Designs

A hierarchical design contains one or more designs as sub-designs. Each sub-design can further contain other sub-designs, creating multiple levels of design hierarchy. Designs that contain sub designs are called parent designs.

8.1.2 Design Objects

Figure 8-1 shows the design objects in a design called top. The VC Static Platform design-query commands and attributes are directed toward specific design objects.

Figure 8-1 Design Objects in a Design Called TOP



8.1.2.1 Design

A design consists of instances, nets, ports, and pins. It can contain sub-designs and library cells. In Figure 8-1, the designs are top, encoder, and regfile. The active design (the design being worked on) is called the current design.

Note

Unlike DC, the VC Static Platform has no concept of current_design which is changeable. In this sense, the VC Static Platform current_design is always the design specified as the top design using the -top option in the read_file command.

8.1.2.2 Instance or Cell

An instance is an occurrence in a circuit of a reference (a library component or design) loaded in memory. Each instance has a unique name. A design can contain multiple instances. Each instance may point to the same or a different reference, but always has a unique name to distinguish it from other instances. An instance is also known as a cell.

A unique instance of a design within another design is called a hierarchical instance. A unique instance of a library cell within a design is called a leaf cell. Some commands work within the context of a hierarchical instance of the current design.

8.1.2.3 Ports

Ports are the inputs and outputs of a design. The port direction is designated as input, output, or inout.

8.1.2.4 Pins

Pins are the input and output of cells (such as gates and flip-flops) within a design. The ports of a sub design are pins within the parent design.

8.1.2.5 Nets

Nets are the wires that connect ports to pins and pins to each other.

8.2 Design Query Commands

This section lists the various Tcl commands that is available for you to access design objects after the design is successfully built.

- `all_connected`
- `all_designs`
- `all_fanin`
- `all_fanout`
- `all_inputs`
- `all_instances`
- `all_outputs`
- `all_registers`
- `get_blackbox`
- `get_cells`
- `get_designs`
- `get_lib_cells`
- `get_lib_pins`
- `get_nets`
- `get_sources`
- `get_trace_paths`
- `get_trace_path_elements`
- `get_object_name`
- `get_pins`
- `get_ports`
- `remove_blackbox`
- `report_trace_paths`
- `set_app_var`
- `set_blackbox`
- `set_message_severity`

8.2.1 Common Behavior

Many commands supported by the VC Static Platform, construct collections of objects. In almost all cases, the commands allow various mechanisms for flexible specifications of what objects should be included in these collections. These mechanisms include:

- Wildcard name specification. These wild-cards can be glob-style, or regular expression style.
- Boolean expressions of object attributes and values.

This section describes behavior that is common to many commands.

8.2.1.1 Regular Expression Behavior

Many commands have a `-regexp` switch. When used, this switch has the following behavior:

`-regexp` - The command views the patterns (name) argument as a regular expression rather than a simple wildcard pattern.

This option also modifies the behavior of the =~ and !~ filter operators to use regular expressions rather than simple wildcard patterns.

Regular expression matching is similar to the Tcl regexp command. When using the `-regexp` option, take care in quoting the pattern argument and filter expression. Using rigid quoting with curly braces around regular expressions is recommended. Note that regular expressions are always anchored; that is, the expression is assumed to begin matching at the beginning of an object name and end matching at the end of an object name. You can widen the search by adding ".*" to the beginning or end of the expressions, as needed.

For commands that have both switches, the -regexp and -exact options are mutually exclusive. You can specify only one of these options.

8.2.1.2 Filter Expression Behavior

Many commands have a `-filter` switch which allows the collection to be composed as a function of attributes of the objects being collected.

`-filter expression` - Filters the collection with the specified expression.

For each object which might be added to the collection, the expression is evaluated based on the objects's attributes. If the expression evaluates to true, the object is included in the result.

To see the list of object attributes that you can use in the expression, use the `list_attributes -application -class <class>` command.

8.2.1.3 Object Name Filtering

Almost all commands which return collections have a patterns field. This field is used as a convenience to filter the collection based on object name and wild-card value for the name.

`patterns` - Creates a collection of objects whose names match the specified patterns. Patterns can include the * (asterisk) and ? (question mark) wildcard characters. For more information on using and escaping wildcards, see the wildcards man page. Pattern matching is case sensitive unless you use the `-nocase` option. If `-filter` is also specified, then objects must evaluate to true for the filter expression, and the object name must pass the patterns field.

If you do not specify any of these arguments, the command uses * (asterisk) as the default pattern.

8.3 Using Collections

VC Static Platform builds an internal database of the design structure and the attributes applied to the design. This database consists of several classes of objects, such as designs, libraries, ports, cells, nets, and pins. Many VC Static Platform commands operate on these objects.

A collection is a group of related objects exported to the Tcl user interface. Collections have an internal representation of the objects, and sometimes a string representation. The string representation is generally used only for error messages.

You can create collections of objects, and then apply a set of commands to interact with those collections. Collections can be homogeneous (contain objects of one type) or heterogeneous (contain objects of many types).

The collection based commands are divided into the following three categories:

- Commands that create collections of objects for use by another command.
- Commands that manipulate collections.
- Commands that query objects for you to view.

You can use wildcards and filtering criteria to narrow the focus of the collection or filter a collection. You can refer to collections in variables for reading (and in some cases writing) attributes, performing custom reporting or use in analysis scripts.

8.3.1 Creating Collections

In the VC Static Platform, you can pass a collection to a command in the following two ways:

- Use the result of a command that creates the collection.
- Use a variable that contains a collection.

For example, the following command creates a collection of nets and assigns that collection to a variable named mynets:

```
set mynets [get_nets]
```

8.3.1.1 Primary Commands that Create Collections

The following commands are used to directly create collections of design objects:

- `get_cells`: Creates a collection of cells or instances
- `get_designs`: Creates a collection of designs
- `get_lib_cells`: Creates a collection of library cells
- `get_lib_pins`: Creates a collection of library cell pins
- `get_nets`: Creates a collection of nets
- `get_pins`: Creates a collection of pins
- `get_ports`: Creates a collection of ports

When used at the command prompt, any command that creates a collection will implicitly query the collection. You can also directly view the contents of a collection by using the `query_objects` command.

Note

Do not use the echo, puts, or printvar commands as they will only return the string representation of the collection. This is the name arbitrarily assigned by the VC Static Platform to serve as an opaque handle to the collection.

8.3.1.2 Primary Collection Command Example

The `get_cells` command creates a collection of cells in the design. The syntax is as follows:

```
get_cells [-hierarchical] [-quiet][-regexp]
[-nocase] [-exact] [-filter expression]
patterns | -of_objects objects
```

For example, enter the following command to query the cells with name beginning with `o` and referencing an FD2 library cell:

```
get_cells "o*" -filter {ref_name == FD2}
{"o_reg1", "o_reg2", "o_reg3", "o_reg4"}
```

Although the output looks like a list, it is not. The output is only a display of the `get_cells` result. The collection was not assigned to a variable nor passed to another command. So, it was deleted as soon as it was displayed.

Given a collection of pins, use the following commands to query the cells connected to those pins:

```
set pins [get_pins o*/CP]
{"o_reg1/CP", "o_reg2/CP"}
get_cells -of_objects $pins
{"o_reg1", "o_reg2"}
```

The following commands also create collections of design objects based on some sort of analysis or algorithm:

- `all_fanin`: Creates a collection of fanin of pins, ports, or nets
- `all_fanout`: Creates a collection of fanout of pins, ports, or nets
- `all_connected`: Creates a collection of objects connected to a net, pin, or port
- `all_inputs`: Creates a collection of input ports
- `all_outputs`: Creates a collection of output ports
- `all_registers`: Creates a collection of register cells or pins

Many other commands in the VC Static Platform return collections of objects other than design objects. The use of collections for those objects is exactly the same as described here.

Each collection created by a command such as `get_nets` has its own context. You cannot add to, remove from, or compare collections from different contexts such as different designs, different ports, or different cells. To compare objects, create one large collection containing all the objects of interest, and then sort, filter, or manipulate the objects in that collection.

For example, the following commands create two different sets of cells that cannot be compared:

```
set someCells [get_cells -filter …]
set moreCells [get_cells -filter …]
```

Even if the two collections represent the same cells, they are different objects, so comparing them will always report a mismatch:

```
if {$someCells != $moreCells} { echo “Mismatch!” }
```

The `compare_collections` command is used to compare two collections.

8.3.2 Saving Collections

To save a collection for later use, assign the result of the collection-creating command to a variable. For example:

```
set domA [get_power_domains -name domainA]{"domainA"}
set domB [get_power_domains -name domainB]{"domainB"}
set badXovers [get_crossovers -source $domA -dest  $domB]
```

The result of the first two commands is two collections of one power domain object each. The third command then creates a collection of crossovers between the two power domains, saving the collection in the `badXover` variable.

In many cases, the collection consumes significant memory. Saving the collection in a variable helps the script to be more memory efficient. You can deallocate the collection by resetting the variable to which a

collection was assigned (domA and domB). Alternatively, you can assign some other value to the variable which references the collection.

A collection is active only as long as it is referenced. Typically, a collection is referenced when a variable is set to the result of a command that creates it, or when it is passed as an argument to a command or a procedure. For example, if you save a collection of ports as follows:

```
set myports [get_ports *]
```

Either of the following two commands will delete the collection referenced by the myports variable:

```
unset myports
set myports "newvalue"
```

Collections can be implicitly deleted when they go out of scope, in other words, when the parent of the objects within the collection is deleted. For example, if your collection of ports is owned by a design, the collection is implicitly deleted when the design that owns the ports is deleted. When a collection is implicitly deleted, the variable that referenced the collection still holds a string representation of the collection. However, since the collection is gone, this string value is not useful.

8.3.3 Querying Objects in a Collection

To view the contents of a collection, use the `query_objects` command. This command searches for and displays objects in the VC Static Platform database. Do not use the `echo`, `puts`, or `printvar` commands, as doing so will only return the string representation of the collection, a name arbitrarily assigned by the VC Static Platform to serve as a handle to the collection.

The output of the `query_objects` command is similar to the DC find command. The `query_objects` command does not have a meaningful return value; it simply displays the objects found and returns the empty string.

8.3.3.1 Implicit Query

When commands that create collections are issued from the command prompt, they implicitly query the collection.

Use the following command to retrieve ports that match the pattern in*:

```
get_ports in*
{"in0", "in1", "in2"}
```

You can also use the following commands to see the same ports.

```
query_objects [get_ports in*]
{"in0", "in1", "in2"}
query_objects -class port in*
{"in0", "in1", "in2"}
```

8.3.3.2 Using Wildcard Characters

Most commands that create collections allow a list of patterns, which contain wildcard characters. The VC Static Platform uses some UNIX global-style matching operators, including:

- * – Matches 0 to n characters
- ? – Matches one character

Example:

Given ports i0, i1, in0, and in1, notice the queries in this sequence of commands:

```
get_ports  i*
```

```
{"i0", "i1", "in0", "in1"}
query_objects [get_ports i?]
{"i0", "i1"}
```

Commands that create explicit collections, such as `get_cells`, allow you to search in the current instance hierarchically by using the `-hierarchical` option. The rules for different kinds of searches are as follows:

- Using a wildcard pattern alone matches leaf names in the current instance.

  Example - `get_cells i1*`

- Using a wildcard pattern with the `-hierarchical` option matches leaf names at each level of the hierarchy.

  For example, to find all cells in the hierarchy with the leaf name containing n1, enter:

  ```
  get_cells *n1* -hierarchical
  ```

- When you search for a wildcard pattern that contains the hierarchy separator, the VC Static Platform breaks up the pattern around the hierarchy separator and matches each piece at progressively deeper levels of the hierarchy.

  For example, to find the cells in i1 that begin with i2, then return a collection of cells in each i1/i2* that has a leaf name of n1, enter:

  ```
  get_cells i1/i2*/n1
  ```

- If an object you specify has a name that contains a backslash character (\) or any wildcard characters (* or ?), it is better to use the `-exact` option to find the object. Using `-exact` makes the primary collection creation command such as `get_cells` or `get_ports` consider the patterns argument to be a list of exact strings rather than a list of patterns with wildcards.

  For example:

  ```
  get_cells -exact [list {*cell*1}] ;# finds cell actually named *cell*1
  get_cells -exact [list {a\b1}] ;# finds cell actually named a\b1
  ```

- Wildcard matching uses the same logic as the Tcl string match command. Without the -exact option, each wildcard match character needs to be backslash-escaped in order to be considered exactly and not as a wildcard.

  For example:

  ```
  get_cells [list {\*cell\*1}]
  get_cells [list {a\\b1}]
  ```

Remember that the patterns argument to the collection creation command is a list. If you do not supply a well-formed list, additional backslashes will be necessary to communicate these special characters.

8.3.4 Filtering Collections

You can filter collections by using the `-filter` option with the primary commands that create collections.

8.3.4.1 Using the -filter Option

Many commands that create collections accept a `-filter` option that specifies a filter expression. A filter expression is a string composed of a series of logical expressions describing a set of constraints you want to place on a collection.

Each sub expression of a filter expression is a relation contrasting an attribute name such as name or direction with a value such as N23 or input, by means of an operator such as == or !=.

For example, the following command gets the cells in U1 that have an area no greater than 12 or reference a design (or library cell) named AN2P, AO2P, and so on. The command then assigns the collection to the Cells variable.

```
set Cells [get_cells "U1/*"  -filter {area <= 12 || ref_name =~ "A*P"}]
```

For more information, see section "Using Attributes".

8.3.4.2 Using Filter Operators

The filter language supports the following logical operators:

- AND or && – Logical AND (not case-sensitive)
- OR or || – Logical OR (not case-sensitive)

You can group logical expressions with parentheses to enforce precedence; otherwise, the VC Static Platform evaluates the expressions from left to right.

The filter language supports the following relational operators:

- == (Equal)
- != (Not equal)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)
- =~ (Matches pattern)
- !~ (Does not match pattern)

  For example, in the following filter expression,

  ```
  {area <= 12 || ref_name =~ "A*P"}
  ```

- Area is an attribute (or identifier)
- <= is a relational operator
- 12 is a value
- || is the logical OR operator

The filter language also supports the following existence operators:

- defined
- undefined

An existence operator determines whether an attribute is defined for an object.

For example:

```
sense == setup_clk_rise and defined(sdf_cond)
```

The right side of a relation can consist of a string or a numeric literal. You do not need to enclose strings in quotation marks. This method is useful because a filter expression is usually the value for an argument, and the entire expression is enclosed in quotation marks.

As shown in the following command, you do not need to enclose the word in with quotation marks:

```
set iports [get_ports * -filter {direction == in}]
```

However, if an expression contains characters that are part of the filter language syntax, you must use curly braces {} to enclose the expression and double quotation marks “” to enclose string operands.

As shown in the following example, parentheses are part of the filter language, so they are double quoted, and the complete expression is grouped in curly braces:

```
set x [filter_collection $ports {full_name =~ "D(*)"}
```

8.3.4.3 Relation Rules

Parsing a filter expression can fail due to the following reasons:

- Syntax problems
- An invalid attribute name
- A type mismatch between an attribute and the value

The following are the basic relational rules:

- String attributes can be compared with any operator.
- Numeric attributes cannot be compared with patterns.
- Boolean attributes can be compared only with == and !=. The value can be only true or false.

8.3.4.4 Using Pattern Match Filter Operators

The pattern match filter operators, =~ and !~, behave differently in different circumstances. They do one of two types of pattern matching as follows:

- Simple wildcard matching (the default)
- Full regular expression matching

Commands that provide a `-filter` option also provide a `-regexp` option to enable you to change the filtering to full regular expressions. The `filter_collection` command also provides a `-regexp` option to enable full regular expressions.

8.3.4.5 Using the filter_collection Command

The `filter_collection` command is useful for filtering an existing collection. However, it is usually less efficient than using the `-filter` option.For more information on the -filter option, see the “Using the -filter Option”section.

The `filter_collection` command takes a collection and a filter expression as arguments. The result is a new collection, or an empty string if no objects match the criteria. The following two commands are equivalent:

```
get_cells * -filter "ref_name =~ A*P"
filter_collection [get_cells *] "ref_name =~ A*P"
```

The `filter_collection` command is useful in translating DC scripts that use the filter command. In addition, if you derive several collections from one larger collection, using the `filter_collection` command might be more efficient than using the `-filter` option, as shown in the following series of commands:

```
set acells [get_cells "*"]
{"u1", "u2", "u3", "u4"}
set ands [filter_collection $acells "ref_name =~ AN*"]
{"u1", "u2"}
set ors [filter_collection $acells "ref_name =~ OR*"]
```

```
{"u3", "u4"}
```

Optionally, the `filter_collection` command accepts the `-regexp` option, which changes =~ and !~ to perform real regular expression matching.

8.3.5 Iterating Over the Elements of a Collection

The `foreach_in_collection` command iterates over a collection.

```
foreach_in_collection itr_var collections body
```

For more information on the `foreach_in_collection` command, see the *VC Static PLatform Command Reference Guide*. You can nest the `foreach_in_collection` command within other control structures, including another `foreach_in_collection` command.

During each iteration, the `itr_var` variable is set to a collection of exactly one object. Any command that accepts one of the collections also accepts the `itr_var` variable because they are of the same type (a string representation of a collection).

To generate a separate report for each cell, use the following commands:

```
foreach_in_collection itr [get_cells o*] {
redirect [format "%s%d" $fbase $ndx]report_cell
}
```

Note

You can use the foreach command to iterate over a list, but not a collection. Attempting to do so will cause the collection to be deleted.

8.3.6 Removing From a Collection and Adding to a Collection

The VC Static Platform allows you to remove objects from a collection or add objects to a collection by using the `remove_from_collection`, `add_to_collection`, and `filter_collection` commands.

The following is the syntax:

```
remove_from_collection collection object_spec collection
add_to_collection collection object_spec [-unique]
filter_collection collection expression [-regexp] [-nocase]
```

You can remove objects using the `filter_collection` command or by filtering objects when you initially create the collection. However, there are some constructs, such as removing the elements in one collection from another collection, that you cannot accomplish with filtering.

Use the `remove_from_collection` and `add_to_collection` commands for this purpose. The arguments to both commands are a collection and a specification of the objects that you want to add or remove. The specification can be a list of names, wildcard strings, or other collections.

The result of these commands is a new collection, or an empty string if the operation results in zero elements in the case of the `remove_from_collection` and `filter_collection` commands. The first argument (the base collection) is a read-only argument.

Examples:

This command sets the ports variable for collection of all ports except for clock:

```
set ports [remove_from_collection [get_ports "*"] CLOCK]
```

This command results in a collection of all cells in the design except those in the top level of hierarchy:

```
set lcells [remove_from_collection [get_cells * -hier] [get_cells *]]
```

You can add objects to a collection with the add_to_collection command:

```
set isos [add_to_collection [get_cells i*] [get_cells o*]]
```

Most collections are homogeneous since commands suchas `get_ports`, `get_cells`, create homogeneous collections. Some commands, such as `all_connected`, create heterogeneous collections. You can also create heterogeneous collections using the `add_to_collection` command.

For example:

```
set a [get_ports PH*]
{"PH1", "PH2"}
set b [get_pins -regexp {reg(0|1)/CP}]
{"reg0/CP", "reg1/CP"}
query_objects -verbose [add_to_collection $a $b]
{"port:PH1", "port:PH2", "pin:reg0/CP", "pin:reg1/CP"}
```

For more information on the `remove_from_collection`, `add_to_collection`, and `filter_collection` commands, see the *VC Static PLatform Command Reference Guide*.

8.3.7 Sorting Collections

You can sort a collection using the `sort_collection` command. The following is the syntax:

```
sort_collection [-descending] collection criteria
```

The VC Static Platform sorts according to the list of attributes you specify and are ascending by default. You can reverse the order using the `-descending` option. In an ascending sort, the VC Static Platform first sorts boolean attributes with the objects that have the attribute set to false, then sorts the objects that have the attribute set to true. In the case of a sparse attribute, objects that do not have the attribute follow the objects that have the attribute.

For example, enter the following command to sort the ports by direction, then by full name:

```
sort_collection [get_ports *]  {direction full_name}
{"in1", "in2", "out1", "out2"}
```

To get a collection of hierarchical cells decreasing alphabetically, followed by leaf cells decreasing alphabetically, sort the collection of cells using the `is_hierarchical` and `full_name` attributes. In the following example, i1 and i2 are hierarchical, and u1 and u2 are leaf cells:

```
set c [get_cells {u2 i2 u1 i1}]
{"u2", "i2", "u1", "i1"}
set cs [sort_collection -descending $c {is_hierarchical full_name}]
{"i2", "il", "u2", "u1"}
```

The `is_hierarchical` attribute is boolean, so in a descending sort, cells that have the attribute set to true (i1 and i2) are first. To resolve the equality for the `is_hierarchical` attribute on cells i1 and i2, use the `full_name` attribute.

For more information on the `sort_collection` command, see the *VC Static PLatform Command Reference Guide*.

8.3.8 Using Collection Utility Commands

The VC Static Platform provides the following additional commands for manipulating collections:

- `sizeof_collection`
- `compare_collections`
- `copy_collection`

- `index_collection`

Note
Some collections cannot be indexed, copied, or compared.

For more information on these commands, see the *VC Static PLatform Command Reference Guide*.

8.3.8.1 Determining the Size of a Collection

The `sizeof_collection` command returns the number of elements in a collection. The following is the syntax:

```
sizeof_collection collection
```

You can use the empty string (the empty collection) as an argument. `sizeof_collection ""` is always 0.

Examples:

To gauge the size of a design, enter:

```
sizeof_collection [get_cells * -hier]
```

To determine whether a collection command yielded results, enter:

```
if {[sizeof_collection [get_cells U2/U2]] != 0}
```

8.3.9 Comparing Collections

The `compare_collections` command compares the contents of two collections, object for object. You can specify that the VC Static Platform compare the objects in order. The following is the syntax:

```
compare_collections collection1 collection2 [-order_dependent]
```

If the objects in both collections are the same, the result is 0 (like the result of string compare). If the objects are different, the result is non-zero.

8.3.10 Copying Collections

The `copy_collection` command duplicates a collection, resulting in a new collection. The base collection remains unchanged.

The `copy_collection` command is an efficient mechanism for duplicating an existing collection. However, copying a collection and having multiple references to the same collection are significantly different. Not all collections can be copied. The following is the syntax:

```
copy_collection collection
```

Examples:

If you create a collection and save a reference to it in variable c1, assigning the value of c1 to another variable c2 creates a second reference to the same collection:

```
set c1 [get_cells "U1*"]
{"U1", "U10"}
set c2 $c1
{"U1", "U10"}
echo $c1 $c2_sel3 _sel3
```

The previous commands, do not copy the collection; only `copy_collection` creates a new collection that is a duplicate of the original.

The following command sequence shows the result of copying a collection:

```
set collection1 [get_cells "U1*"]
```

```
{"U1", "U10"}
set collection2 [copy_collection $collection1]
{"U1", "U10"}
compare_collections $collection1 $collection20
```

8.3.11 Indexing Collections

The `index_collection` command creates a collection of one object that is the nth object in another collection. Following is the syntax:

```
index_collection collection index
```

Objects in a collection are numbered 0 through n-1. Not all collections can be indexed. Other collections which result from commands such as `get_cells` are not really ordered, but each has a predictable, repeatable order. The same command executed n times, for example, `get_cells *`, creates a collection of cells in the same order.

The index field of `index_collection` specifies the index into the collection.

Example:

To extract the first object in a collection, enter:

```
set c1 [get_cells {u1 u2}]
{"u1", "u2"}
index_collection $c1 0{"u1"}
```

8.4 Using Attributes

Attributes describe logical, electrical, physical, and other properties of objects in the design database. An attribute is attached to a design object and is saved with the design database. The VC Static Platform uses attributes on the following types of objects:

- Design objects, such as nets, pins, and ports.
- Cell instances within a design.
- Library components such as library cells and library cell pins.
- Objects related to low power.

An attribute has a name, a type, and a value. The following are the different types of attributes:

- String
- Numeric
- Logical (Boolean)

Some attributes are pre defined and are recognized by the VC Static Platform. These are termed *Application* attributes and are almost always read-only. Other attributes are user-defined and may be read or written by the user. Many attributes apply to one object type. For example, the `port_direction` attribute applies only to ports.

Some attributes apply to several object types. For example, the `full_name` attribute can apply to a net, cell, or port.

You can get detailed information about the pre defined attributes that apply to each object type by using the `list_attributes -application` command.

8.4.1 Attribute-Related Commands

The following are the VC Static Platform Tcl commands related to attributes:

- `define_user_attribute`
- `get_attribute`
- `list_attributes`
- `remove_attribute`
- `set_attribute`

8.4.2 Predefined Application Attributes

This section describes the different pre-defined application attributes which are available on various design objects. The attribute names and expected values are described here.

- Application attributes are all read-only.
- Attributes of type boolean return true or false.
- Attribute names and values are case sensitive.

8.4.2.1 The object_class Attribute

Many collections are heterogeneous in nature. The `object_class` attribute is the mechanism used at the Tcl level to differentiate various object types. The `object_class` attribute is a string with the following values and meanings:

- cell - Cells/Instances
- port - Ports
- pin - Pins
- net - Net
- design - Modules
- lib_cell - Library Cell
- lib_pin - Library Pin

8.4.2.2 Cell Attributes

Table 8-1 lists the various cell attributes.

Table 8-1 Cell Attributes

Attribute Name	Type	Description
base_name	String	The leaf name of a cell. For example, the base_name of cell U1/U2/U3 is U3.
full_name	String	The complete name of a cell. For example, the full name of cell U3 within cell U2 within cell U1 is U1/ U2/U3.
is_three_state	Boolean	This attribute is true if a cell is a three-state (tri-state) device.
is_hierarchical	Boolean	This attribute is true for any hierarchical cell. It is false for library cells, also known as leaf cells.

Attribute Name	Type	Description
is_sequential	Boolean	It is true only if a cell is a leaf-level cell and it is sequential.
is_combinational	Boolean	A cell is combinational if it is non-sequential or non-three-state and all of its outputs compute a combinational logic function. It is NOT black box.
is_black_box	Boolean	This attribute is true if the cell's reference is not linked to a library cell or design. This attribute is read-only. You cannot change the setting
is_mux	Boolean	This attribute is true if a cell is a multiplexer.
is_pad_cell	Boolean	This attribute is true if the library cell is a pad cell.
is_macro_switch	Boolean	This attribute is true if the reference library cell is defined as a fine-grain switch cell in the library.
number_of_pins	Integer	Number of pins on the cell.
ref_name	String	The name of the design or library cell of which the cell is (or will be) an instantiation. Also known as the reference name. The linker looks for a design or library cell by this name.
has_multi_ground_rails	Boolean	The attribute is true if a cell has multiple ground rails.
has_multi_power_rails	Boolean	The attribute is true if a cell has multiple power rails.
is_integrated_clock_gating_cell	Boolean	This attribute is true if a cell is defined in the library as an integrated clock-gating cell.

8.4.2.3 Pin Attributes

Table 8-2 lists the various pin attributes.

Table 8-2 Pin Attributes

Attribute Name	Type	Description
base_name	String	The leaf name of a pin. For example, the base name of pin i1/i1z1 is i1z1.
full_name	String	The complete name of a pin. For example, the full_name of pin i1z1 within cell i1 is i1/i1z1.
direction	String	The direction of a pin. Value can be in, out, inout, or internal.
pin_direction	string	Same as direction (for PT compatibility).
is_async_pin	Boolean	This attribute is true if a pin is an asynchronous preset/clear pin.
is_clock_pin	Boolean	This attribute is true if this is clock pin of a sequential cell instance.
is_clear_pin	Boolean	This attribute is true if a pin is an asynchronous clear pin.
is_clock_gating_pin	Boolean	This attribute is true if a pin for a clock-gating cell.
is_data_pin	Boolean	This attribute is true if a pin is a data pin of a sequential cell.

Attribute Name	Type	Description
is_hierarchical	Boolean	This attribute is true for any pins that are instantiations of another design and false for pins that are instantiations of a library pin (also known as leaf pins).
is_mux_select_pin	Boolean	This attribute is true if a pin is the select pin of a multiplexer device.
is_three_state	Boolean	This attribute is true if a pin is a three-state (tri-state) driver.
is_three_state_enable_pin	Boolean	This attribute is true if a pin is an enable pin of a three-state device.
is_three_state_output_pin	Boolean	This attribute is true if a pin could output a three-state signal.
constant_value	String	The logic value of a pin tied to logic constant zero or one in the netlist.
is_enable_pin	Boolean	This attribute will return true for enable pin of a cell and will return false for other pins.
user_case_value	String	The user-specified logic value of a pin or port.

8.4.2.4 Port Attributes

Table 8-3 lists the various port attributes.

Table 8-3 Port Attributes

Attribute Name	Type	Description
base_name	String	The leaf name of a pin. For example, the base name of pin i1/i1z1 is i1z1.
full_name	String	The complete name of a pin. For example, the full_name of pin i1z1 within cell i1 is i1/i1z1.
case_value	String	The user-specified logic value of a pin or port propagated from a case analysis or logic constant in the design.
constant_value	String	The logic value of a port tied to logic constant 0 or 1 in the netlist.
direction	String	The direction of a port. Value can be in, out, inout, or internal.
port_direction	String	Same as direction (PT compatibility).
user_case_value	String	The user-specified logic value of a pin or port.

8.4.2.5 Net Attributes

Table 8-4 lists the various net attributes.

Table 8-4 Net Attributes

Attribute Name	Type	Description
base_name	String	The leaf name of a net. For example, the base name of net i1/i1z1 is i1z1.
full_name	String	The complete name of a net. For example, the full_name of net i1z1 within cell i1 is i1/i1z1.

8.4.2.6 Design Attributes

Table 8-5 lists the various design attributes.

Table 8-5 Design Attributes

Attribute Name	Type	Description
full_name	String	The name of a design. For example, the full_name of design TOP read in from /u/user/simple.db is TOP. This name can be ambiguous because several designs of the same name can be read in from different files.
extended_name	String	The complete, unambiguous name of a design. The extended_name of the design is the source_file_name attribute followed by a colon (:) followed by the full_name attribute. For example, the extended_name of design TOP read in from /u/user/ simple.db is /u/user/simple.db:TOP.
is_current	Boolean	This attribute is true for the current design. This attribute changes for all designs when you use the current_design command.
source_file_name	String	The name of the file from which the design was read. For example, the source_file_name of design TOP read in from /u/user/simple.v is /u/ user/simple.v.

8.4.2.7 lib_cell Attributes

Table 8-6 lists the various `lib_cell` attributes.

Table 8-6 lib_cell Attributes

Attribute Name	Type	Description
base_name	String	The name of a library cell. For example, the base_name of library cell tech1/AN2 is AN2.
full_name	String	The fully qualified name of a library cell. This is the name of the library followed by the library cell name. For example, the full_name of library cell AN2 in library tech1 is tech1/AN2.
is_black_box	Boolean	This attribute is true if the cell's reference is not linked to a library cell or design.
is_sequential	Boolean	This attribute is true if the library cell is sequential.
is_three_state	Boolean	This attribute is true if a library cell is a three-state device.
is_combinational	Boolean	A cell is combinational if it is non-sequential or non-three-state and all of its outputs compute a combinational logic function. It is NOT black box.
is_integrated_ clock_gating_cell	Boolean	This attribute is true if a cell is defined in the library as an integrated clock-gating cell.
has_multi_ground _rails	Boolean	The attribute is true if a cell has multiple ground rails.
has_multi_power_ rails	Boolean	The attribute is true if a cell has multiple power rails.

Attribute Name	Type	Description
is_isolation	Boolean	This attribute is true if a library cell is an isolation cell.
is_level_shifter	Boolean	This attribute is true if the library cell is a level-shifter cell.
is_macro_switch	Boolean	This attribute is true if the library cell is defined as a fine-grain switch cell in the library.
is_retention	Boolean	This attribute is true if the library cell is a retention cell.
is_mux	Boolean	This attribute is true if a library cell is a multiplexer.
is_pad_cell	Boolean	This attribute is true if the library cell is a pad cell.
number_of_pins	Boolean	Number of pins on the cell.

8.4.2.8 lib_pin Attributes

Table 8-7 lists the various `lib_pin` attributes.

Table 8-7 lib_pin Attributes

Attribute Name	Type	Description
base_name	String	The name of a library cell. For example, the base_name of library cell tech1/AN2 is AN2.
full_name	String	The fully qualified name of a library cell. This is the name of the library followed by the library cell name. For example, the full_name of library cell AN2 in library tech1 is tech1/AN2.
direction	String	The direction of a pin. Value can be in, out, inout, or internal.
pin_direction	String	Same as direction (PT compatibility).
is_async_pin	Boolean	This attribute is true if a pin is an asynchronous preset/clear pin.
is_clock_pin	Boolean	This attribute is true if at least one instance of that clock pin exists in the design that has the is_clock_pin attribute equal to true.
is_clear_pin	Boolean	This attribute is true if a pin is an asynchronous clear pin.
is_data_pin	Boolean	This attribute is true if at least one instance of that data pin exists in the design that has the is_data_pin attribute equal to true.
is_mux_select_pin	Boolean	This attribute is true if a library pin is a select pin of a multiplexer device. is_padBooleanThis attribute is true if the library pin is a pad.
is_three_state	Boolean	This attribute is true if the library pin is a three-state driver.
is_three_state_enable_pin	Boolean	This attribute is true if a pin is an enable pin of a three-state device.
is_three_state_output_pin	Boolean	This attribute is true if a pin could output a three-state signal.