Synopsys® VC Formal Tutorial
Data Path Validation (DPV)

Version 1.0 | 23-May-2023

Portland State University

The design example in this tutorial is not supplied. You need to use your own files to follow the tutorial steps and instructions

Table of Contents

1. Introduction

VC Formal uses TCL (Tool Command Language) scripts to tell it what to do. The script can define the app within VCFormal that we want to use, the files we are working on, how we are mapping them, the clock cycles, and more. Instead of diving into the details of TCL scripts, we will use templates. One would then simply need to modify those templates to interact with VC Formal as needed in their project.

To begin, you need to set up the VNCserver as shown in the previous tutorials and open the Linux GUI. For better practices and to keep everything organized, we create a main folder for the design we want to analyze, and in this case, I called mine “DPV_Example”. **Don’t use spaces when naming the files and folders**.

Inside that folder, we create two folders: one is Design, where you put the actual Verilog or SystemVerilog designs we want to analyze, and the other is Run, where we put the TCL that will run the VCFormal DPV analysis for us.

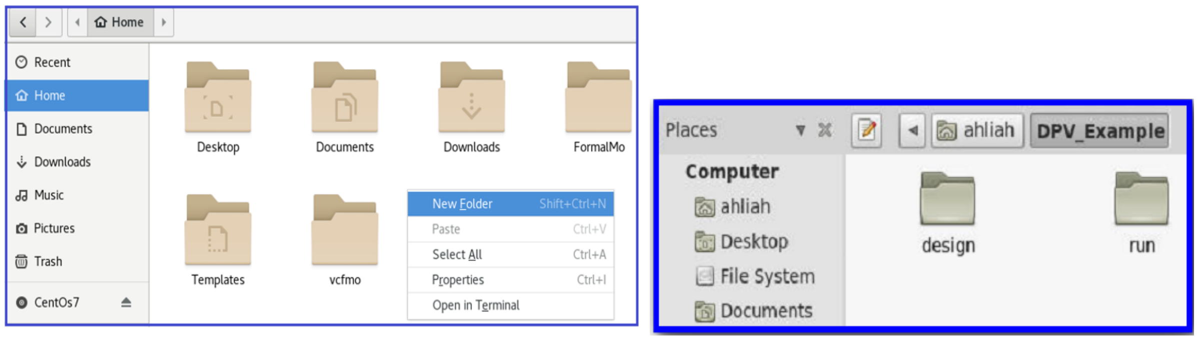
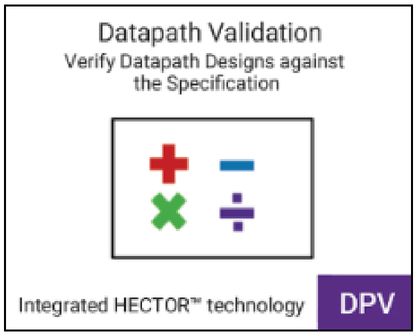


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of DPV


Datapath Validation
Verify Datapath Designs against the Specification
Integrated HECTOR™ technology
DPV

In the previous applications, we have dealt with FPV, which verifies control paths (FIFOs, FSMs, bus bridges, etc.).

Datapath Validation (DPV) verifies data transformation blocks through data manipulation and transformation (ALU, FPU, DSP, etc.), where these path blocks are floating point/integer adder, multiplier, divider, and more.

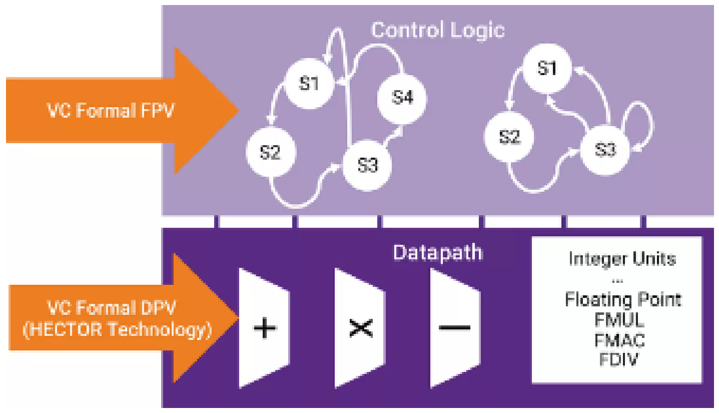


Figure 1.1.2. *A visual image of the difference between FPV and DPV.*

DPV also uses transactional equivalence to compare two versions of a design - one representing the design functionality at architecture level (mostly untimed C or C++ model), and the other representing the pipeline implementation (mostly RTL).

Transactional equivalence requires you to define a transaction for each design, which is a unit of computation that produces a specific set of output values from a specific set of input values.

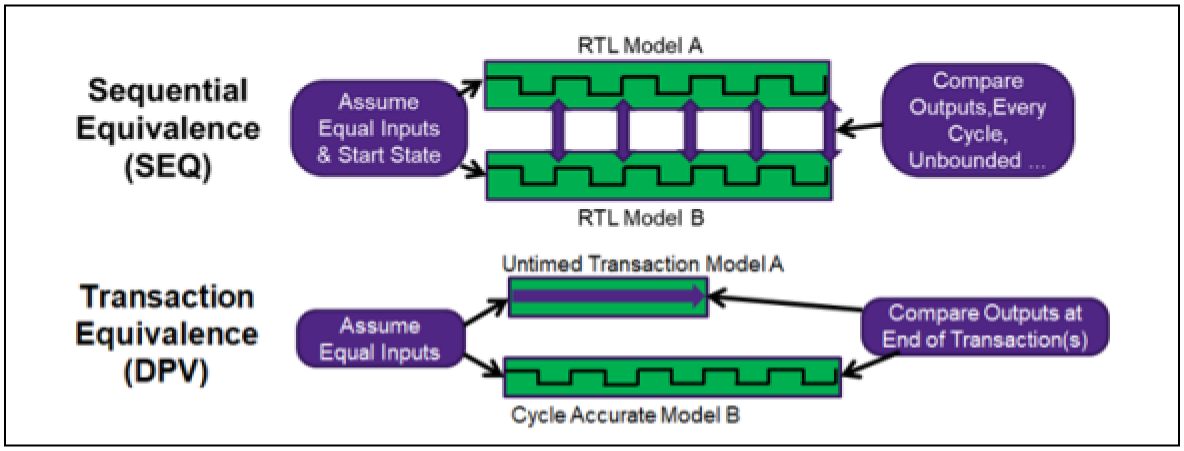

Sequential Equivalence (SEQ)
Assume Equal Inputs & Start State
RTL Model A
RTL Model B
Compare Outputs,Every Cycle, Unbounded ...
Transaction Equivalence (DPV)
Assume Equal Inputs
Untimed Transaction Model A
Cycle Accurate Model B
Compare Outputs at End of Transaction(s)

Figure 1.1.3. *A visual image of the comparison between SEQ and DPV.*

You can learn more about transactional equivalence in the “*VC Formal Datapath Validation User Guide”* under *Section 2: Methodology*.

The following are features that VC Formal DPV supports:

- ❖ Waveform and source code debugging of failed lemmas
- ❖ Powerful formal engines
- ❖ Automates checks for model correctness
- ❖ Support for multi-processing
- ❖ Assume-guarantee and case-splitting automation

The DPV application also supports a long list of commands that can be used in your TCL file. Below is an example of the *fvclear* command:

❖ **fvclear** *(example 1.1)*

Usage:

```
fvclear      # Clears the run status of selected properties
   [-class <list-of-class-attributes> ]
                    (property class selection:
                     Values: aep,  user)
   [-type <list-of-type-attributes> ]
                    (property type selection:
                     Values: aep, user, lemma, cover)

   [-usage <list-of-usage-attributes> ]
                    (property usage selection:
                     Values: lemma, cover)
   [-regexp]        (Use regular-expression instead of glob
                     filtering)
   [-exact]         (No pattern matching performed)
   [-subtype <list-of-subtypes> ]
                    (goal subtype selection:
                     Values: property, vacuity, witness)
   [<list-of-names-ids-or-collections-of-properties> ]
                    (List of property names, name patterns, or
                     property collections)
```

See *Table 1.1.* in the Appendix to see the complete list of commands. To view the usage of each command in-depth, go to “*VC Formal Datapath Validation User Guide”* under *Section 1.5: Supported VC Formal Commands*.

See *Tables 5.1* and *5.2* to familiarize yourself with other TCL commands.

1.2 Design Files

In the Design folder, you'll put in the RTL and C/C++ design file. The TCL file will be put in the Run folder. We suggest you name your folders with lowercase letters, as VC Formal is case-sensitive.

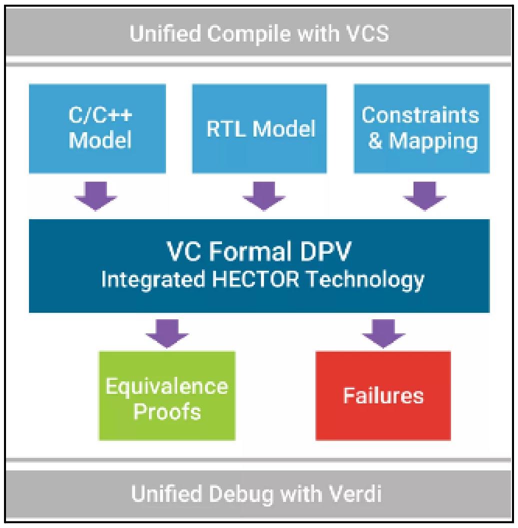


Figure 1.2.1. Showing the required files needed; C/C++, RTL, and TCL file that holds our constraints & mapping.

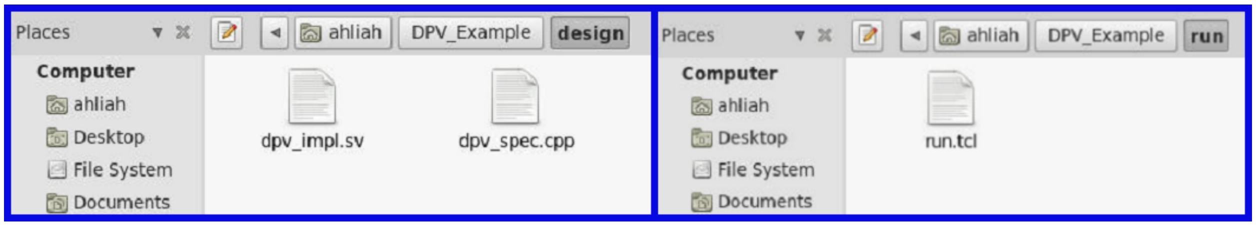


Figure 1.2.2. Showing the (Left) RTL/C++ and (Right) TCL files in the correct folder locations.

2 TCL File Syntax

Next, we will create a TCL script for DPV setup. This is where we compile our designs using various commands and into formal models represented in data-flow graphs (DFGs). These commands will be put in procedures, or *proc* and executed in the VC Formal GUI.

As with the other applications, make sure to set app mode to DPV:

```
set_fml_appmode DPV
```

To enable the C++11 front-end, the following command must be placed in the DPV setup file.

```
set _hector_comp_use_new_flow true
```

Now, we need to compile our designs. Steps to compile a design:

1. Create a design using create_design command.
2. Analyze the design (language specific step).
3. Generate the DFG by calling compile_design command.

Step 1. Below is the syntax for creating a design using the **create_design** command:

```
create_design -name <spec|impl> -top <topname>
  [-options <string>]
  [-clock <name>] [-reset <name>] [-negReset]
  [-lang <c|c++|scdt|systemc|verilog|vhdl|sverilog|mx>]
```

Example 2.1. *Syntax for the create_design command.*

Specifications for *Example 2.1*:

- `-name <designname>`: The name of the design. Possible values are *spec* or *impl*.
- `-top <topname>`: The top level module/function name for DPV analysis.
- `-options <string>`: Used to provide any additional compiler specific options.
- `-clock <name>`: The name of the clock port in a single clock design.
- `-reset <name>`: The name of the reset signal in a design with simple reset.
- `-negReset`: The polarity of the reset signal.
- `-lang <name>`: The language for the design. (i.e. c, c++, scdt, systemc, verilog, vhdl, mx, or sverilog).

See *Table 2.1* for compile design examples in different languages.

Step 2. To analyze C++ files, the command is:

```
cppan <options> <list of filenames>
```

- <options>: The g++ compiler options for compiling the files
- <list of filenames>: List of files to compile

File Language	Command
SystemC	vcs <options> <list of filenames>
Verilog	vlogan <options> <list of filenames>
VHDL	vhdlan <options> <list of filenames>

Table 2.2. *Command for analyzing certain languages.*

Step 3. Generating a data-flow graph (DFG) involves converting models into a visual representation. The command to create the .dfg file is:

```
compile_design name
```

The argument name with this command should match the name used in the create_design command in ***Example 2.1*** ('spec' or 'impl'). This .dfg file will be written in the current working directory (folder).

An existing spec.dfg and impl.dfg can be reused if needed. If reusing, the compile_spec/compile_impl commands are not needed.

To avoid compilation of spec or impl, spec.dfg or impl.dfg needs to be copied to the vcst_rtdb/.internal/hector folder.

```
exec cp spec.dfg vcst_rtdb/.internal/hector
exec cp impl.dfg vcst_rtdb/.internal/hector
compose
solveNB -ual myUAL myUAL
```

Example 2.2. *Reusing a dfg example.*

The next section provides some example templates you can use for your designs. Simply make the proper changes in file language, argument names, etc.

2.1 TCL File Examples

Example for compiling C/C++ design:

```
proc compile_spec { } {

   create_design -name -spec -top main
   cppan -Iinclude -DCHECKFP foo.cc
   compile_design spec
}
```

Example 2.1.1. *Compile example for C/C++ design.*

Example for compiling RTL design:

```
proc compile_impl {} {

   create_design -name -impl -top play \
           -clock clk -reset rst -negReset
   vcs play.v
   compile_design impl
}
```

Example 2.1.2. *Compile example for Verilog design.*

See ***Table 4.3*** to see the complete list of design examples.

Not defining procs and proof generation will result in an empty task list and goal list. Continue onto the next sections to see examples on how to curate these proofs for your TCL/setup file.

2.2 TCL: Assumes, Lemmas, and Covers

VC Formal DPV allows you to specify assumptions on RTL signals and/or C-variables and lemmas that need to be proven.

Assumptions syntax	→	assume <name> = <expression>
Proof obligations or lemmas syntax	→	lemma <name> = <expression>
Covers syntax	→	cover <name> = <expression>

These commands must be placed inside a TCL procedure as shown in the templates below, along with the command '**set_user_assumes_lemmas_procedure**'.

Template to define lemmas/assumes:

```
proc my_assumes_lemmas {} {

   assume a1 = .....
   lemma l1 = .....
   cover c1 = .....
}
set_user_assumes_lemmas_procedure "my_assumes_lemmas"
```

Example 2.2.1. *Template using assumptions, lemmas, and covers in a proof.*

Example defining lemmas/assumes:

```
proc my_assumes_lemmas {} {

   assume a1 = (spec.ain(1) == impl.ain(1))
   lemma l1 = (spec.cout(1) == impl.cout(4) [31:0])
   cover c1 = ( (impl.mode(3) == 2) )
   cover c2 = ( (impl.mode (3) == 2) && (impl.out1(5) == 3) )
}
set_user_assumes_lemmas_procedure "my_assumes_lemmas"
```

Example 2.2.2. *Example using assumptions, lemmas, and covers in a proof.*

VC Formal DPV uses a two-state logic system, so **X in assume/lemma expressions are not supported**. For example, *"assume (impl.mysignal_4a(0) == 1'bx)".*

2.3 TCL: Case Splitting

Case splitting is used for breaking a hard formal verification proof into subsections. Below is a visual reference for how this technique works:

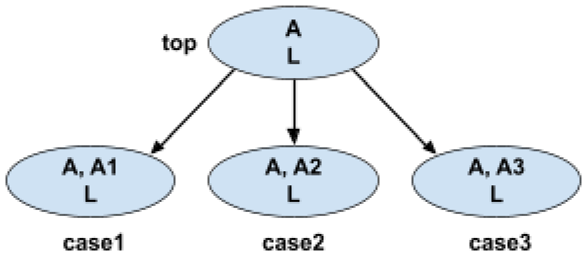


Figure 2.3.1. *Case split tree diagram. 'A' stands for assumptions, and 'L' stands for a lemma to be proven.*

Looking at the diagram above, we have split *top* into three cases by adding more assumptions to the original problem.

Case 1: Consists of proving L using assumptions A and A1
Case 2: Consists of proving L using assumptions A and A2
Case 3: Consists of proving L using assumptions A and A3

If all of the case splits are complete and the lemmas pass in each sub-proofs, then they pass in the original proof as well. VC Formal DPV checks the completeness of these case splits.

caseSplitStrategy command example:

```
caseSplitStrategy name [-script sname]
```

- -name: Specifies a name used to refer a collection of case splits
- -script sname: Optional. Specifies the name of a solve script to use for all case splits under this case split strategy. Individual case splits can override the solve script to use.

"This command is used to provide a name to a collection of case splits. You can create multiple case split strategies. You can also specify which case split strategy to use during the proof."

See ***Table 5.4.*** for the full list of commands for case splits.

You can also use the following commands to navigate between multiple case split strategies:

- listCaseSplitStrategies
- cdCaseSplitStrategy name
- listCaseSplitStrategy

These commands must also be placed inside a TCL procedure, as shown in the examples below, and be enabled by adding the following command in the TCL script before running:

```
set_hector_case_splitting_procedure <tcl proc>
```

Template for caseBegin and caseAssume commands:

```
proc case_split_template {} {

   caseSplitStrategy name
   caseBegin name
      caseAssume expr
   caseBegin name
      caseAssume expr
   caseBegin name
      caseAssume expr
}
set_hector_case_splitting_procedure <tcl proc>
```

Example 2.3.1. *Template using assumptions, lemmas, and covers in a proof.*

Template for caseEnumerate commands:

```
proc case_split_template {} {

   caseSplitStrategy name -script sname
   caseEnumerate pname -expr expr
   caseEnumerate pname -expr expr -parent pname
}
set_hector_case_splitting_procedure <tcl proc>
```

Example 2.3.2. *Template using assumptions, lemmas, and covers in a proof.*

Example diagram with proc commands:

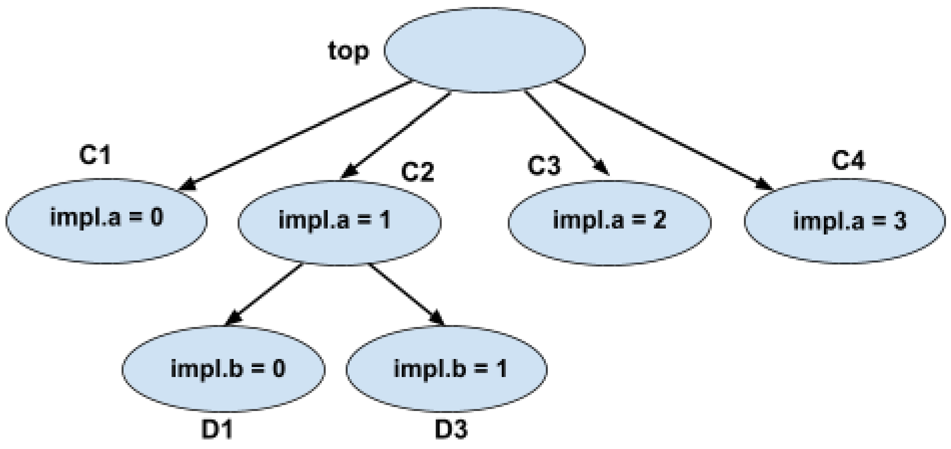


Figure 2.3.2. *Case splitting strategy diagram for example "bar"*

```
proc case_split_template {} {

   caseSplitStrategy bar
   caseBegin C1
      caseAssume (impl.a(1) == 0)
   caseBegin C2
      caseAssume (impl.a(1) == 1)
   caseBegin C3
      caseAssume (impl.a(1) == 2)
   caseBegin C4
      caseAssume (impl.a(1) == 3)
      caseEnumerate D -type FULL -expr “impl.b(1)” -parent C2
}
set_hector_case_splitting_procedure <tcl proc>
```

Example 2.3.3. *Example of case splitting for the diagram in Figure 2.3.2.*

2.4 GUI: Case Splitting

This section shows how to add, remove, or edit existing case splits in the VC Formal GUI. This can only be done after we have successfully run our code through the DPV application.

Navigate to the Case Split Strategy dialog by click on the proof button [ ] and selecting **Case Split Settings**

Figure 2.4.1. Finding proof button location on the GUI.

Figure 2.4.2. Customize Case Split Strategy window.

Section 9.4.8 - pg. 145/216
Maybe explain a little more? But not much.

3. Application Setup

The DPV application uses Hector technology, so currently we can only invoke DPV via the shell command prompt. There are mutiple ways to invoke the DPV app, we can open it using the following command, as you might have seen in the other tutorials.

- Invoke VC Formal GUI and TCL script in one command:

  `$vcf -f run.tcl -gui` or **`$vcf -f run.tcl -verdi`**

  'run.tcl' is the name of the TCL file we are using. If your file name differs from this, you will need to change it accordingly in this command.

  The "-gui" switch opens VC Formal in the GUI, and it's equivalent to the switch "-verdi".

We will go through this method in the following section.

3.1 Invoking VC Formal Along with TCL File:

To proceed with invoking VC Formal:

1) Inside the “run” folder, right-click the whitespace and choose “Open in Terminal”
2) In the terminal, type in the command:

vcf -f run.tcl -gui

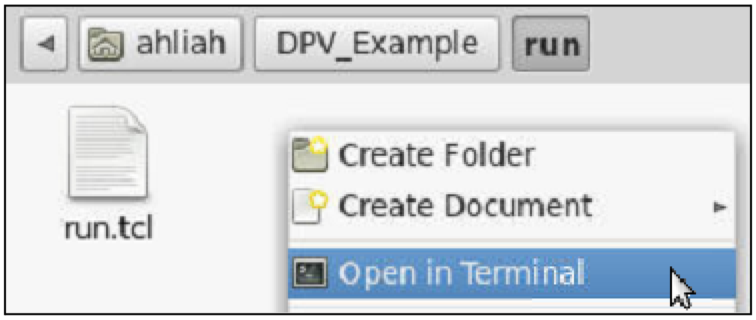
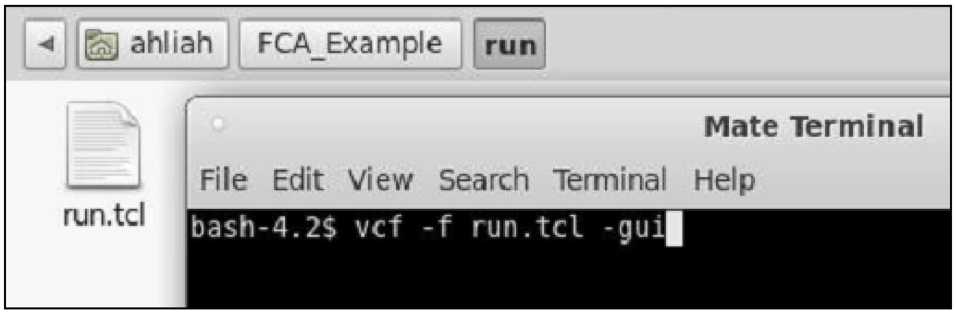


Figure 3.1.1. Opening terminal in the ‘run’ folder.



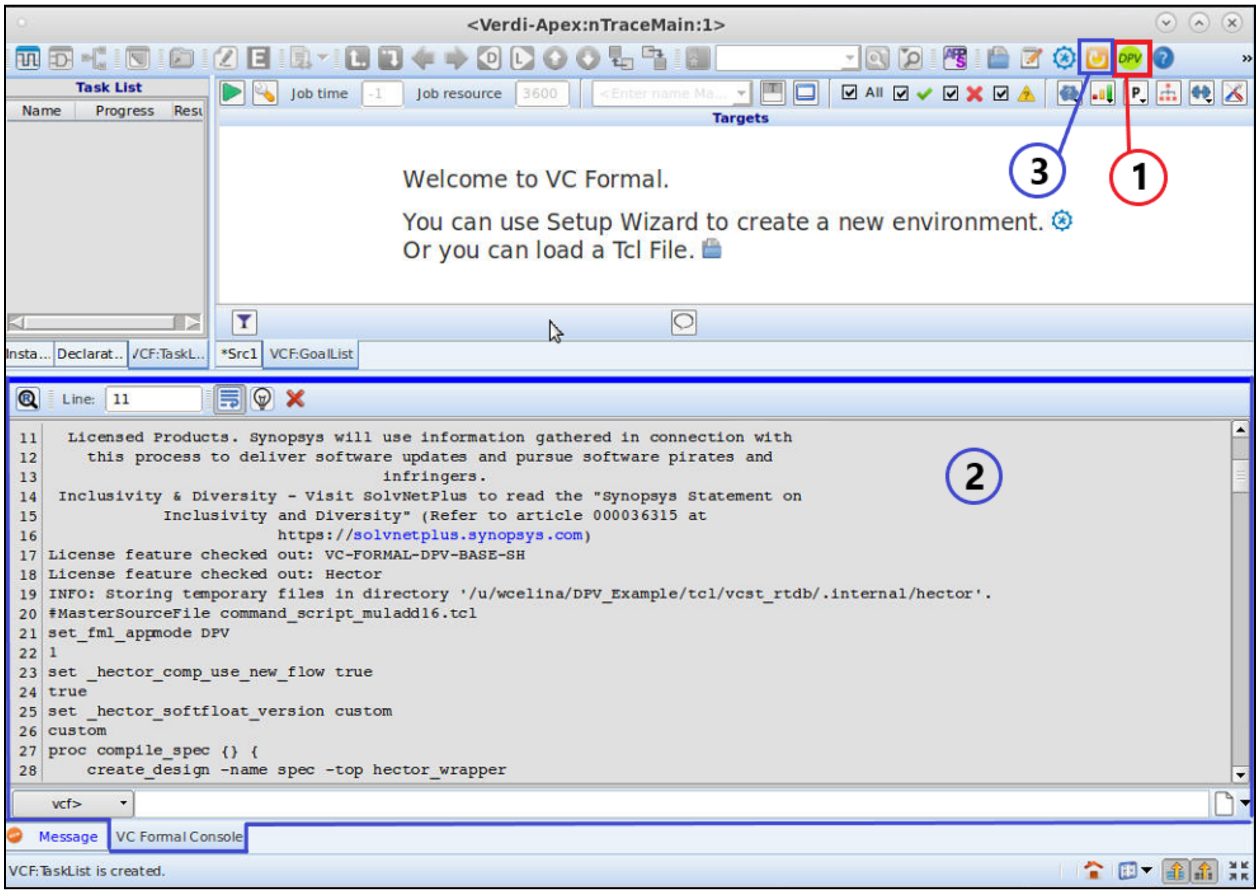
```
bash-4.2$ vcf -f run.tcl -gui
```

Figure 3.1.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

4 Application Usage

After successfully invoking VC Formal GUI and loading the TCL script, your screen should look something like this:



```
 Licensed Products. Synopsys will use information gathered in connection with
   this process to deliver software updates and pursue software pirates and
                             infringers.
Inclusivity & Diversity - Visit SolvNetPlus to read the "Synopsys Statement on
          Inclusivity and Diversity" (Refer to article 000036315 at
                    https://solvnetplus.synopsys.com)
License feature checked out: VC-FORMAL-DPV-BASE-SH
License feature checked out: Hector
INFO: Storing temporary files in directory '/u/wcelina/DPV_Example/tcl/vcst_rtdb/.internal/hector'.
#MasterSourceFile command_script_muladd16.tcl
set_fml_appmode DPV
1
set _hector_comp_use_new_flow true
true
set _hector_softfloat_version custom
custom
proc compile_spec {} {
    create_design -name spec -top hector_wrapper
```

Figure 4.1. *Screen after loading TCL script.*

1) DPV icon signifies that we are in the correct mode
2) VCF-Shell/Terminal Console
3) Restart VCF Icon

Go ahead and click the DPV icon to enable the mode. It should turn yellow.

Check that your source files and proof/lemmas are populated by looking in the console (2). Any errors in the analysis of your command script will be shown.

You may notice that there are no contents in the *VCF:GoalList* tab or anywhere else. Don't worry, this is normal! Continue on to the next topic, *4.1 Running DPV Formal Proofs*, for the next steps.

Remember, **not** defining procs and proof generation will result in an empty task list and goal list. Go to **Sections 2.2** and **2.3** to see examples on how to curate these proofs for your TCL/setup file.

Along with these proofs, make sure to include these commands to enable them:

```
>>  set_user_assumes_lemmas_procedure <tcl proc>

>>  set_hector_case_splitting_procedure <tcl proc>
```

❀ <tcl proc>: proc name, or function name (i.e. “proc adder{}”; proc name = adder)

4.1 Running DPV Formal Proofs

After properly running your TCL script, execute all of the proc names in your TCL script to the VCF-shell one by one. A quick example is provided below.

TCL Example:

```
proc make {} {
   compile_spec
   compile_impl
   compose
}

proc proc_name_here {} {
   create_design -name “impl” -top “demo” -clock clk
   vcs demo.v
   compile_design “impl”
}

proc split {} {
   caseBegin c1
      caseAssume (impl.a(1) = 0)
}

proc another_example {} {
  set_user_assumes_lemmas_procedure “proc_name_here”
  set_hector_case_splitting_procedure “split”

}

. . .
```

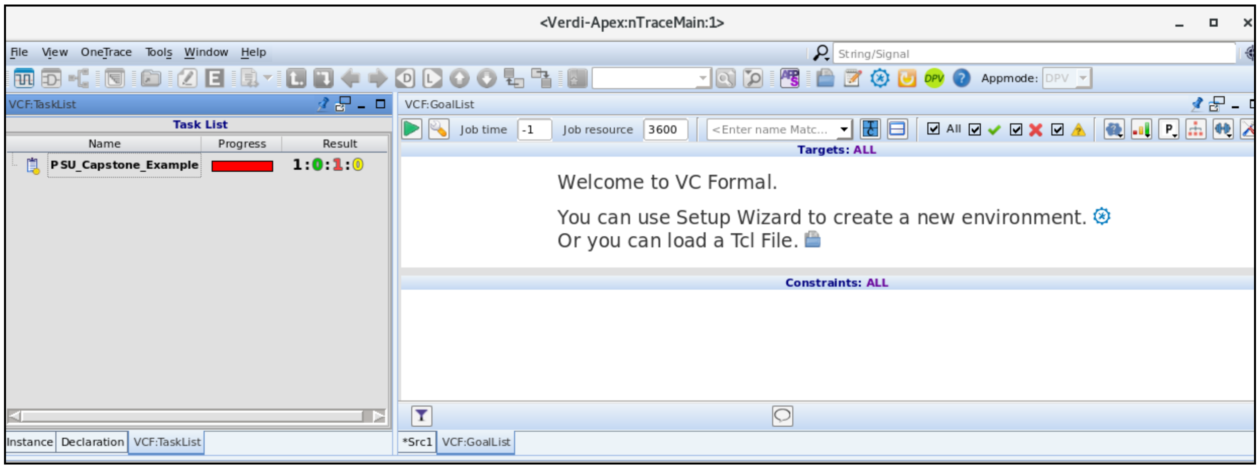
Example 4.1.1. *“Lorem Ipsum” example TCL script - gibberish code, only used to show how the formatting can be done and to use for visual explanation.*

Commands Example:

```
% make
% proc_name_here
% split
% another_example
```

Example 4.1.2. *Example for list of commands to execute in VCF-shell from Example 4.1.1.*

The screen you get will depend on the kind of analysis you need to run, including the different lemmas as specified in your TCL file. Below are two examples of what you can expect to see:


<Verdi-Apex:nTraceMain:1>
File View OneTrace Tools Window Help
String/Signal
Appmode: DPV
VCF:TaskList
Task List
Name Progress Result
PSU_Capstone_Example 1:0:1:0
VCF:GoalList
Job time -1 Job resource 3600
<Enter name Matc...
All
Targets: ALL
Constraints: ALL
Instance Declaration VCF:TaskList
*Src1 VCF:GoalList

Welcome to VC Formal.

You can use Setup Wizard to create a new environment.
Or you can load a Tcl File.

Figure 4.1.1. *Capture of GUI example 1. Targets/constraints are not populated, but there is an item in the task list.*

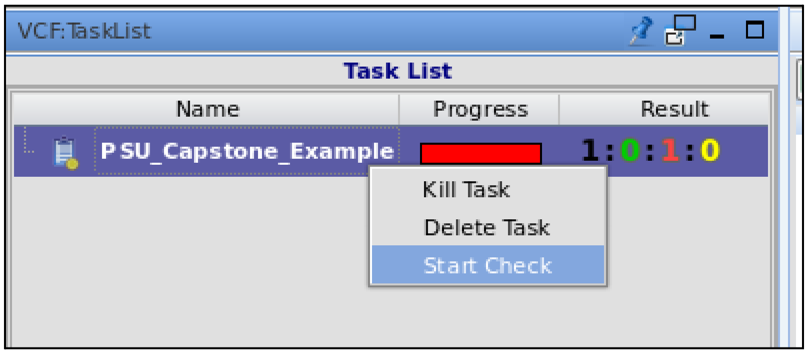
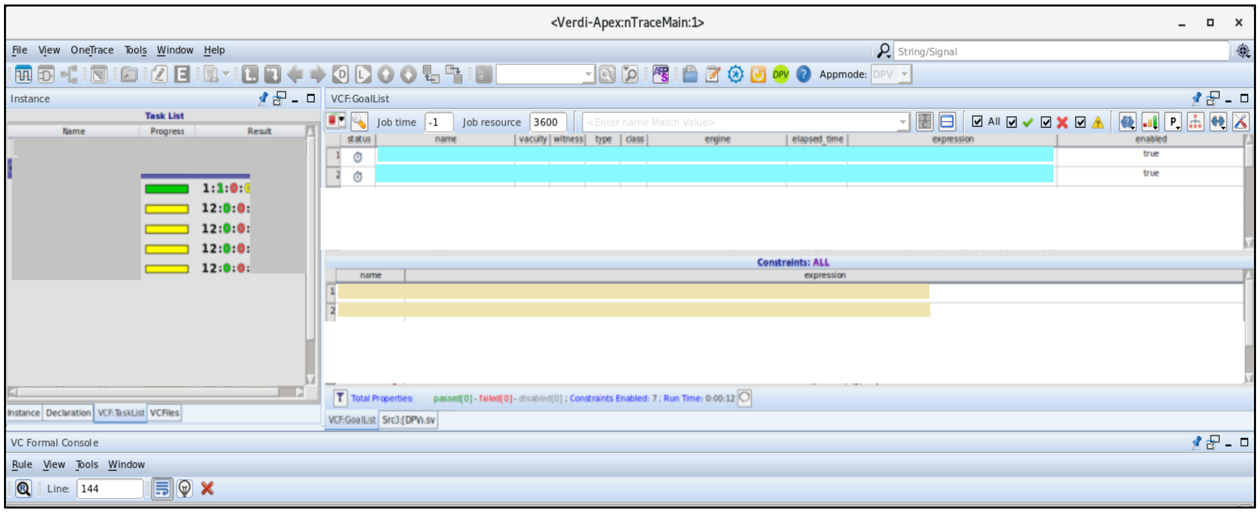


Figure 4.1.2. *Capture of GUI example 1. Start the check by right clicking on the task.*


<Verdi-Apex:nTraceMain:1>
File View OneTrace Tools Window Help
Appmode: DPV
Instance
Task List
Name Progress Result
Instance Declaration VCF:TaskList VCFiles
VCF:GoalList
status name vacuity witness type class engine elapsed_time expression enabled
Constraints: ALL
name expression
VCF:GoalList Src3:[DPV\.sv
VC Formal Console
Rule View Tools Window

Job time -1 Job resource 3600

	status	name	vacuity	witness	type	class	engine	elapsed_time	expression	enabled
1										true
2										true

Total Properties: passed[0] - failed[0] - disabled[0] ; Constraints Enabled: 7 ; Run Time: 0:00:12

Line: 144

Figure 4.1.3. *Capture of GUI example 2, after executing proc names sequentially in the VCF-shell. The targets, and constraints tables will be populated with the properties and constraints specific to your project. The task list on the left will also be specific to your project.*

4.2 Debugging - Waveforms

After running the analysis, the status icon will show whether the lemmas were verified ✓ or falsified ✗. In the case they are falsified, right-click on the lemma ⯑ “View trace” ⯑ “Property” to look at the trace-failure waveform.

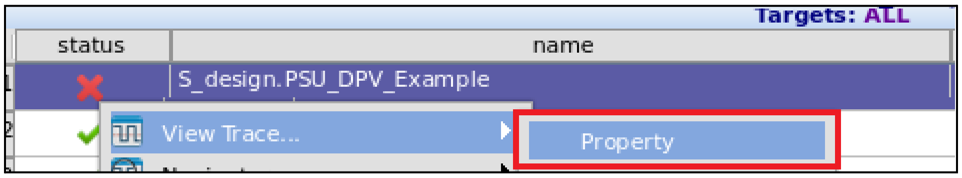


Figure 4.2.1. *Inspecting falsified lemmas.*

After clicking on property, a waveform window should open:

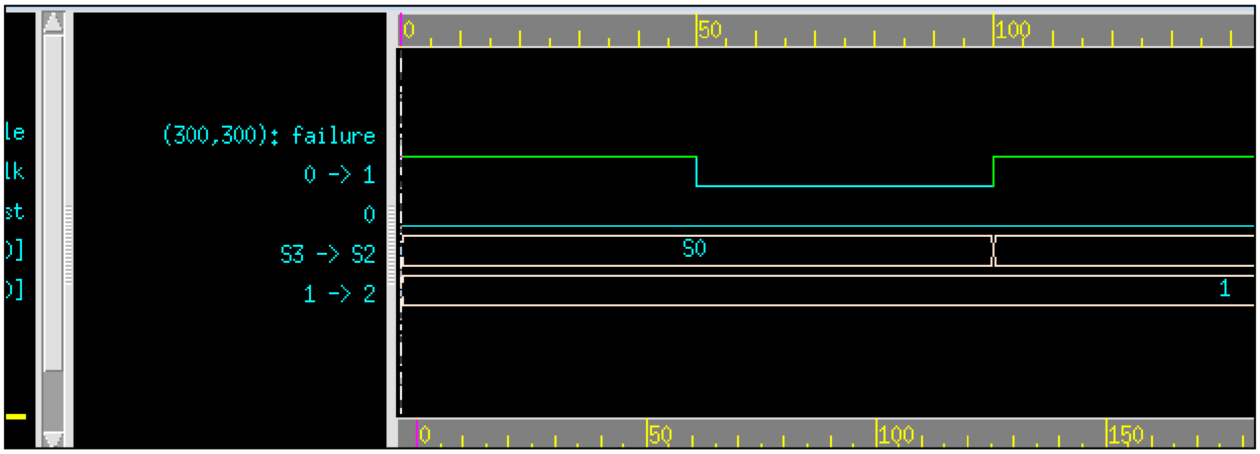


Figure 4.2.2. *Producing the waveform of a falsified lemma.*

The results will be specific to your design, but the waveform signals and transitions should aid you in understanding what caused the lemma to be falsified. You can fix this by altering your design accordingly and restarting VC Formal.

You can also add a debugging tool for C/C++ code designs to the TCL script:

```
simcex -gdb <failed lemma name>
```

4.4 Restarting VC Formal

Restart VC Formal by clicking on . VC Formal will automatically load the same TCL file again when we invoke VC Formal along with the TCL script in the terminal.

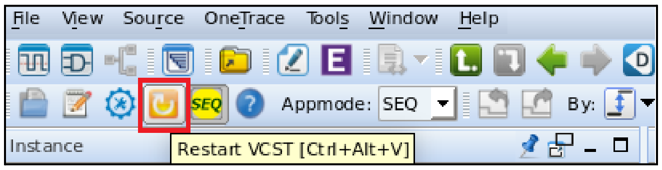
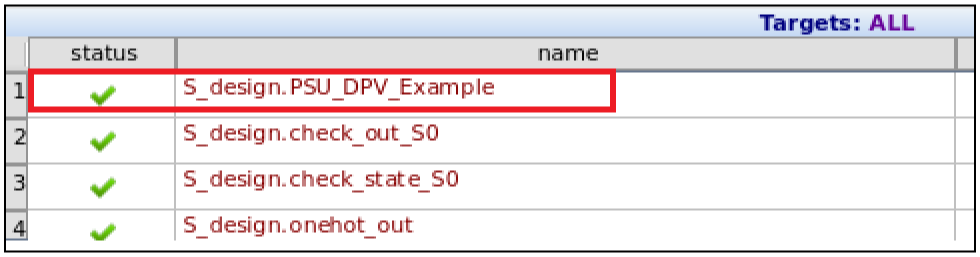


Figure 3.4.1. Location of the restart button in VC Formal window.

After the application and TCL file is loaded, we should see no falsified properties and that they all are proven:



Targets: ALL

	status	name
1	✓	S_design.PSU_DPV_Example
2	✓	S_design.check_out_S0
3	✓	S_design.check_state_S0
4	✓	S_design.onehot_out

Figure 3.4.2. Results after fixing errors and restarting VC Formal.

Appendix

Command	Description
fvclear	Clears the run status of selected properties
fvdelete	Deletes the selected properties
fvenable	Enables properties
fvdisable	Disables properties
view_coverage	Show the coverage view
get_props	Get properties
set_grid_usage	Set grid configuration for VC-Static Formal
start_gui	Show the Activity “Hybrid” View
snip_driver	Snip the driver of the net
report_fv	Reports formal information
check_fv	Run/stop solvers in a given proof
get_blackbox	Returns list of objects which are listed as blackbox
report_blackbox	Returns a report of blackboxed objects in the design
get_sequentials	Gets collection of sequentials of the design
report_app_var	Show application variables
set_app_var	Set the value of an application variable
get_app_var	Get the value of an application variable
set_fml_var	Sets task specific formal variables
get_fml_var	Gets task specific formal variables
report_fml_var	Reports task specific formal variables
set_fml_appmode	Set DPV Mode
fvassert	Creates an assertion
fvassume	Creates an assume

Table 1.1. *VC Formal commands supported by the DPV application (Part 1/2).*

Command	**Description**
fvcover	Creates a cover
report_fml_jobs	Reports information about solver jobs running locally
report_fv_complexity	Print a summary of the operators used in both spec and impl designs
save_session	Save DPV snapshot
restore_session	Restore DPV snapshot
get_snips	Get snip drivers
restart_vcf	Restart VC-Formal
start_verdi	Show Verdi with Docked Activity View
stp_extract -help	Extract the current testcase using STP
set_change_at	Specify legal transition time of the signals
get_license	Get licenses for the specified features

Table 1.1. *VC Formal commands supported by the DPV application (Part 2/2).*

Language	**Example**

Verilog	`proc compile_impl {} {`
	`   create_design -name -impl -top play \`
	`           -clock clk -reset rst -negReset`
	`   vcs play.v`
	`   compile_design impl`
	`}`

SystemVerilog	`proc compile_impl {} {`
	`   create_design -name -impl -top play \`
	`           -clock clk -reset rst -negReset`
	`   vcs -sverilog play.v`
	`   compile_design impl`
	`}`

VHDL (ver. 1)	`proc compile_impl {} {`
	`   create_design -name -impl -top DW01_add_cla.vhd`
	`   vhdlan wrapper.vhd DW01_add.vhd DW01_add_cla.vhd`
	`   compile_design impl`
	`}`

VHDL (ver. 2)	`proc compile_impl {} {`
	`   create_design -name -impl -top testmultipipe \`
	`           -reset rst -negReset -clock CLK`
	`   vhdlan mult_pipeline.vhd`
	`   compile_design impl`
	`}`

Mixed Verilog and VHDL (MX)	`proc compile_impl {} {`
	`   create_design -name -impl -top DW01_add_inst`
	`   vlogan wrapper.v`
	`   vhdlan DW01_add.vhd DW01_add_cla.vhd`
	`   compile_design impl`
	`}`

Table 2.1. *Examples for compile designs of different languages with “create_design” command.*

Case Split Commands
`caseSplitStrategy` `name [-script sname]` ❀ -name: Specifies a name used to refer a collection of case splits. ❀ -script sname: Optional. Specifies the name of a solve script to use for all case splits under this case split strategy. Individual case splits can override the solve script to use.
`caseBegin` `name [-parent pname] [-script sname]` ❀ name: Specifies a name for this case split. ❀ -parent pname: Optional. Specifies the name of the parent case split. Default: top level case splits. ❀ -script sname: Optional. Specifies the name of a solve script to use for this case split.

`1)` **`caseAssume`** `expr`
`OR`
`2) caseAssume name = expr`
❀ expr: An expression involving specification and/or implementation inputs in different places.
❀ name =: Provides a name to the assumption (2)

`caseEnumerate` `name -expr <expr> [parent pname] [-type tname] [-script sname]` ❀ name: Specifies the name to refer to all the cases in the enumeration. ❀ -parent pname: Optional. Specifies the name of the parent case split. Default: top level case splits. ❀ -type tname: Optional. Specifies the type of enumeration (full or leading1). Default = full. ❀ -expr expr: An expression involving specification and/or implementation inputs in different phases. ❀ -script sname: Optional. Specifies a solve script to use for all cases in the enumeration.

`1)` **`caseLemma`** `expr`
`OR`
`2) caseLemma name = expr`
❀ expr: An expression involving specification and/or implementation inputs in different phases.
❀ name =: Provides a name for the lemma (2)

Table 2.3.1. *Case split commands and option descriptions. Learn more about each case starting in Section 9.4.2 of "VC Formal Datapath Validation User Guide".*

TCL Set-up Commands	
Command	Definition
assume	Provides assumptions on inputs or variables in C++ and on signals and registers in RTL.
caseAssume	Specify an assumption that forms the part of the currently selected case split.
caseBegin	Start a case split.
caseConstraint	Specify a constraint that forms the part of the currently selected case split.
caseEnumerate	Creates a collection of case splits by performing a specified type of enumeration on a given expression.
caseSplitStrategy	Provide a name to a collection of case splits.
cutpoint	Enable a cutpoint signal.
lemma	Generates a lemma that will be proven by VC Formal DPV.
set_aep_selection	Controls the checking of automated lemmas.

Table 5.1. VC Formal DPV specific TCL set-up commands. These perform additional configuration of the DPV environment. Can be executed in an interactive session or placed in the TCL script.

TCL Runtime Commands	
Command	Definition
cdproof	Changes to a specific proof.
compile_design	Compiles the specified design in to a DFG.
compose	Creates the internal test framework.
cppan	Analyzes a C/C++ program.
create_design	Creates a specification or implementation design
getTaskDetails	Returns a TCL list with details of lemmas running in each task.
ignore_functions	Ignores a list of functions for C/C++/SystemC programs.
killTasks	Kills one or more scheduled or running tasks.
listassumes	Lists all the assumptions of the currently selected proof.
listproof	Lists the status of all the assumptions and lemmas of the currently selected proof.
listtask	Lists information (error messages) posted by the task.
listtasks	Lists the status of all existing tasks.
listTaskDetails	Lists information about each lemma in each task.
list_aep_selection	Lists the enable status of AEP lemmas.
proofstatus	Returns 1 if all lemmas in all proofs in the list were successful.
proofwait	Waits for each proof int he list provided to have a "conclusive" status.
reload_script	Sources the TCL command script again without exiting the VC Formal DPV shell.
scdtan	Analyzes a SystemC datatypes program.

Table 5.2. VC Formal DPV specific TCL runtime commands. These cause some action to be performed by DPV. Can be executed in an interaction session or placed in the TCL script.

TCL Runtime Commands (cont.)	
Command	Definition
simcex	Simulates a counter example, saves results in one or more forms.
solveNB	Proves the equivalence or in-equivalence of each corresponding output. (non-blocking command)
syscan	Analyzes a SystemC program.
vlogan	Analyzes a Verilog program.
vhdlan	Analyzes a VHDL program.
vcs	Analyzes and elaborates the Verilog only designs. Elaborates an RTL design for VHDL and MX designs.

Table 5.2. VC Formal DPV specific TCL runtime commands.