Synopsys VC Formal Tutorial
Formal X-Propagation Verification (FXP)

Version 1.1 | 11-Mar-2023

Portland State University

Disclaimer:

Formal X-Propagation Verification in Synopsys VC Formal FXP App Youtube Tutorial

Link:

The video is unlisted, and is not to be shared outside of this class.

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

Introduction 3

About and Usage of FXP. 4

Design Files 5

TCL File 6

Application Setup 7

Invoking VC Formal GUI 8

User Interface Details 8

Loading TCL File 9

Invoking VC Formal Along with TCL File 10

Running Files 11

Detecting Errors. 12

Source Tracing 14

Resolving Errors 16

Restarting VC Formal 19

Appendix 20

Table 1.1. FXP Property Types. 20

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 “FXP_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 FXP analysis for us.

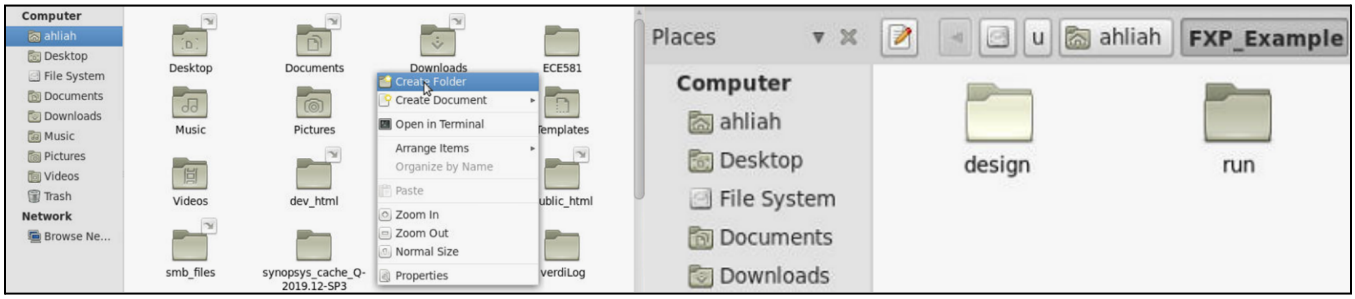
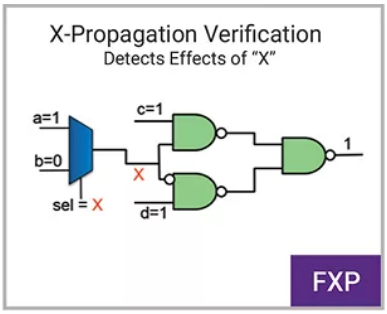


Figure 1. Creating folders to organize design and TCL files.

About and Usage of FXP



The Formal X-propagation (FXP) app in VC Formal is used to check for and trace back an unknown signal value (X). The app detects X propagation through a design and guides you to the failed property that is the source of this signal by using the Verdi schematic and waveform within VC Formal.

To perform Formal X-Propagation Verification, we will need a design file that is usually written in an HDL language, such as, Verilog, SystemVerilog, or VHDL.

The purpose of this app is to prevent unknown signals from getting into critical parts of the system. VC Formal determines what is critical by generating injection and observation points. A user is able to customize this and pinpoint the area/s, but for the tutorial, we will only use the default points. This will result in FPX checking all the following types of 'X' values to generate: abs, absin, absout, auto, bbin, bbout, in, oba, out, snip, snipdrv, undef, undriven, uninit, unresin, unresout, xassign (see *Table 1.1* in the Appendix for more details).

Since we will be looking at everything, there will be some designs that have a number of 'X' that won't necessarily be a major issue; we are not just looking at the critical parts of the system. Nonetheless, if your design fails certain FPX checks, it means it is vulnerable and most times the fundamental design is to blame.

Design Files

In the Design folder, you'll put in the SystemVerilog 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.

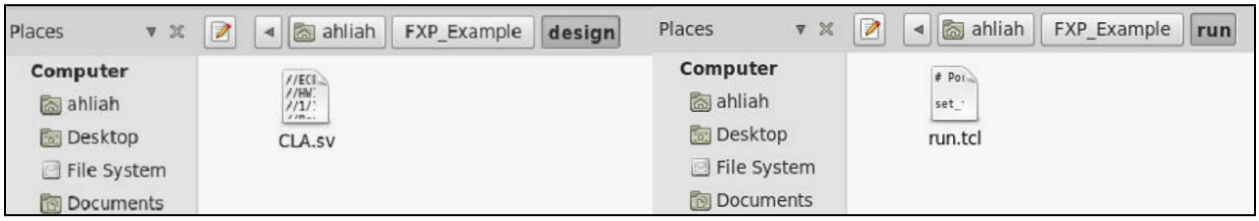


Figure 2. Showing the SystemVerilog and TCL files in the correct folder locations.

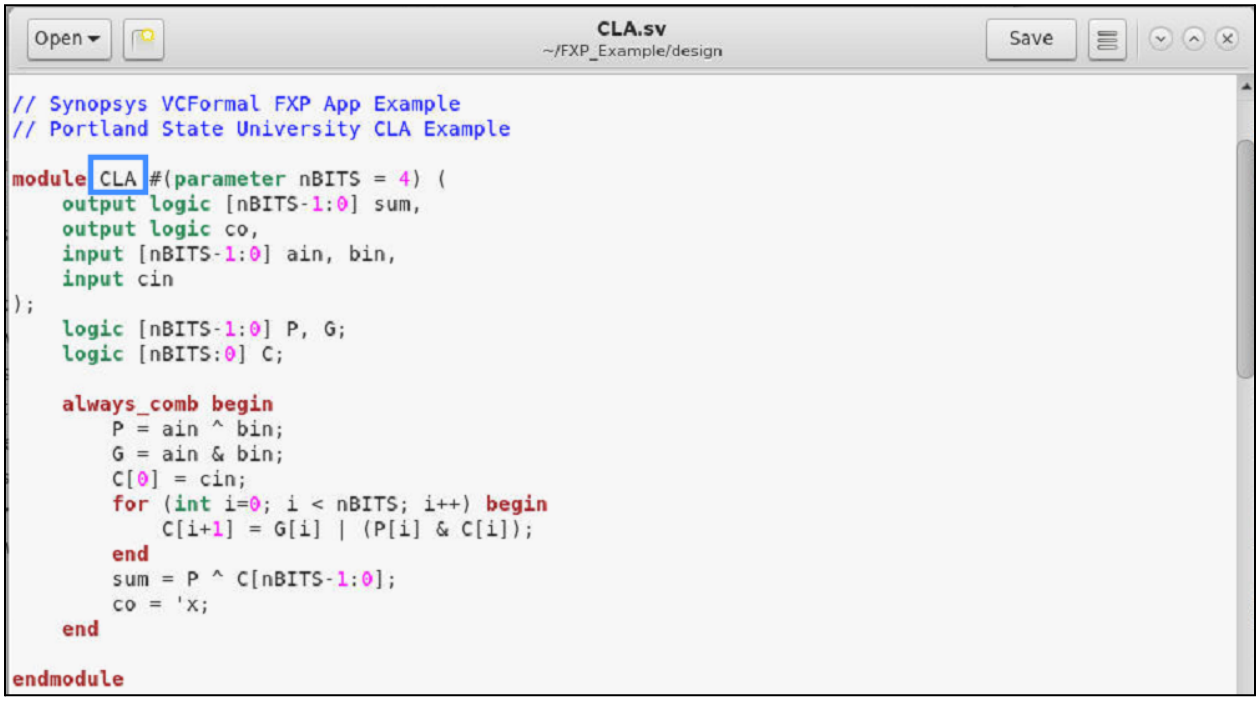


Figure 3. Example of the design we are using in this tutorial.

In order for VC Formal to map your design, you need to acknowledge and keep note of your exact module name, as highlighted in *Figure 3* above. This will come in handy when you create your TCL file. To make things simple, we suggest naming your design file after its corresponding module name (more on this in the TCL File section below).

TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 4)*, which you can use as a template for your functional checks on VC Formal.

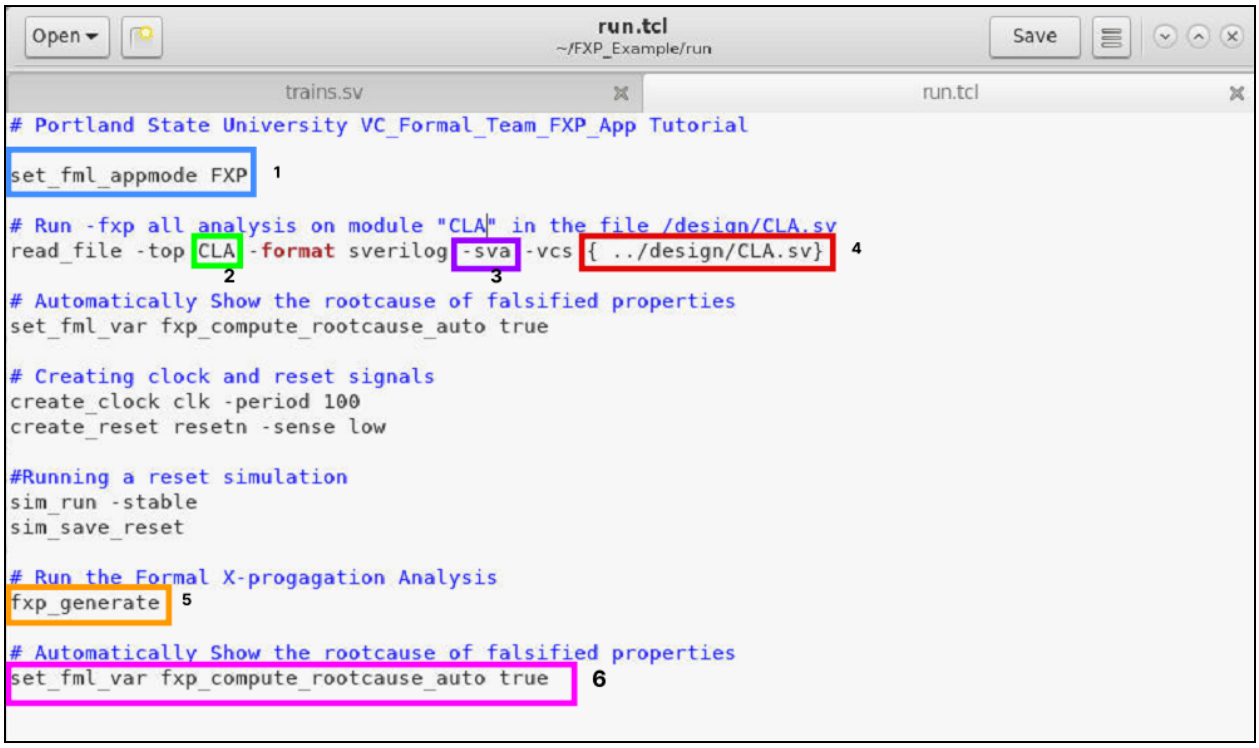


Figure 4. Annotated TCL template file.

(1) Instruction that sets the appmode to FXP in VC Formal.
(2) Name of the main module as established in the Design file.
(3) SystemVerilog assertions (or conditions - automatic for FXP app)
(4) Design file location so VC Formal knows where to find the file.
(5) Generates default points for FXP checking.
(6) This will help identify where a failed FPX check came from.

As mentioned before, VC Formal is case AND character sensitive, and in order for it to map your designs, you will need to use the same module name in the TCL script **(2)** and the module name in the design file. For example, we can see that our module name in the TCL script is the same as the module name in the design file in *Figure 3*.

The file name (CLA.sv) can be named however you want.

Application Setup

There are multiple ways to invoke the VC Formal GUI and load the TCL script.

In this tutorial, we will show two methods for doing so. When using either method, it is important that you open the terminal within the appropriate “Run” folder associated with the FXP app.

- Invoke VC Formal GUI, then manually load the TCL script in the application:

  $vcf -gui

  OR

- 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 both of these methods in the following sections.

Invoking VC Formal GUI

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 -gui

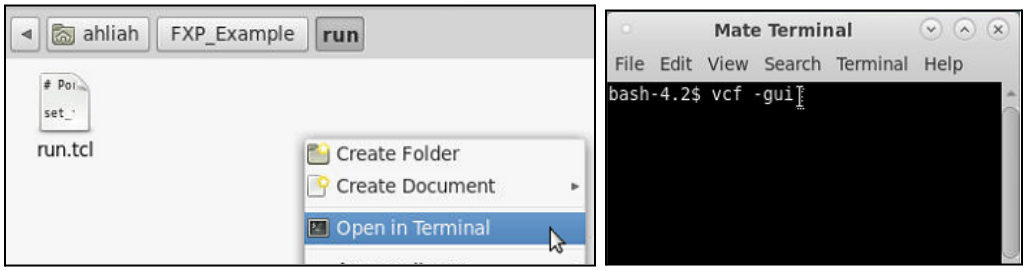


Figure 5. Invoking VC Formal in the terminal.

Note: You may have to wait a few seconds for the program to start up.

You should then see the VC Formal GUI as shown in *Figure 6* below. You can toggle between showing Targets, Constraints, or both Targets and Constraints window by clicking the blue box icon in the upper right-hand of the application **(1)**.

It may look like any of these three icons: 

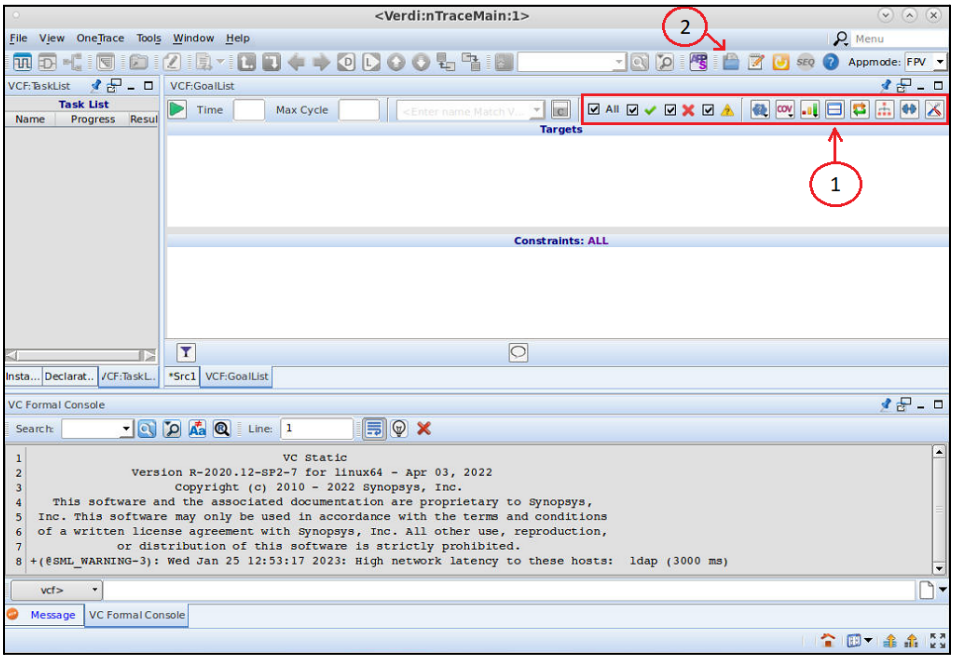


Figure 6. VC Formal GUI introductory screen.

Then load a TCL script by clicking on the  icon **(2)** as shown in *Figure 6*.

Next, select the “run.tcl” file we have in the “run” folder:

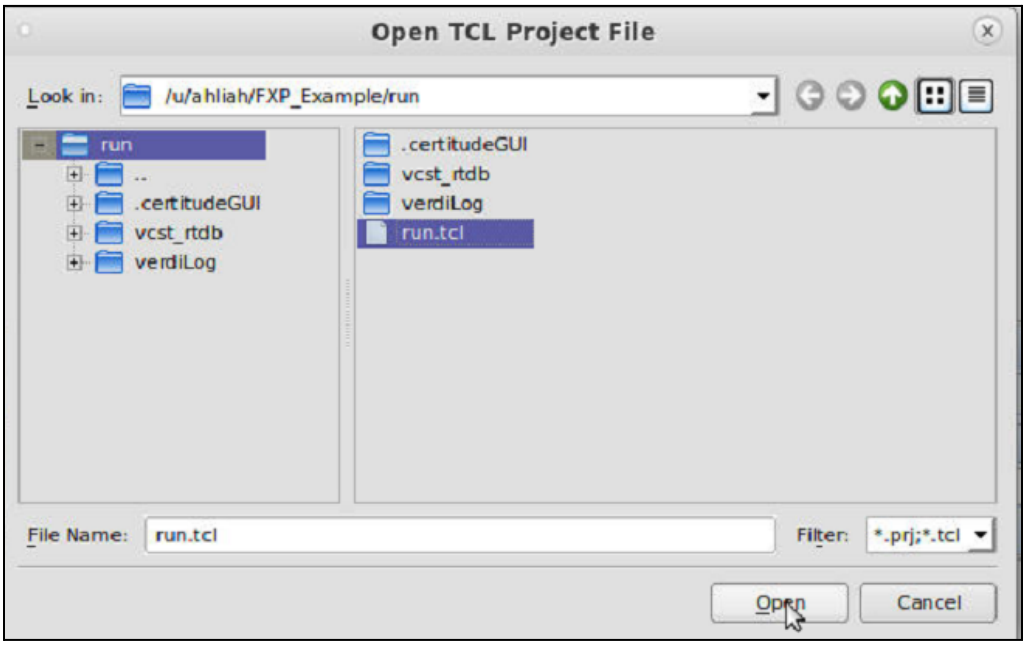


Figure 7. Selecting TCL file.

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

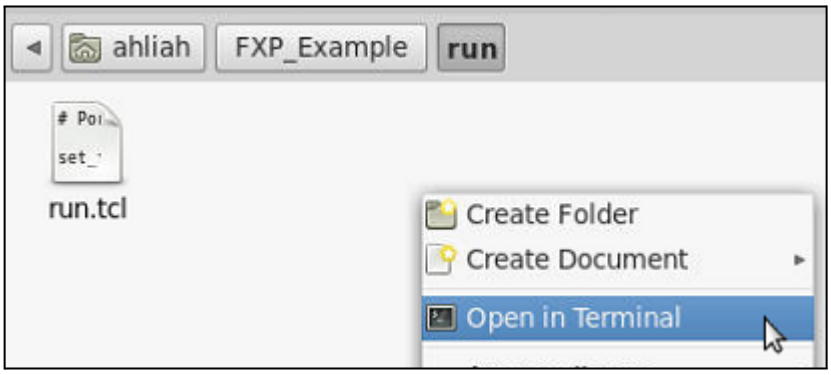


Figure 8. Opening terminal in the ‘run’ folder.

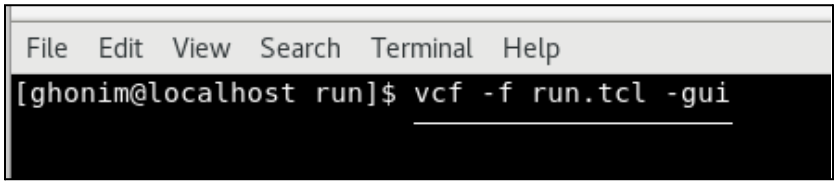


Figure 9. Invoking VC Formal and TCL script in the terminal.

And that’s it!

Running Files

After successfully invoking VC Formal GUI and loading the TCL script in the above methods, your screen should have contents in the *VCF:GoalList* tab and look something like this:

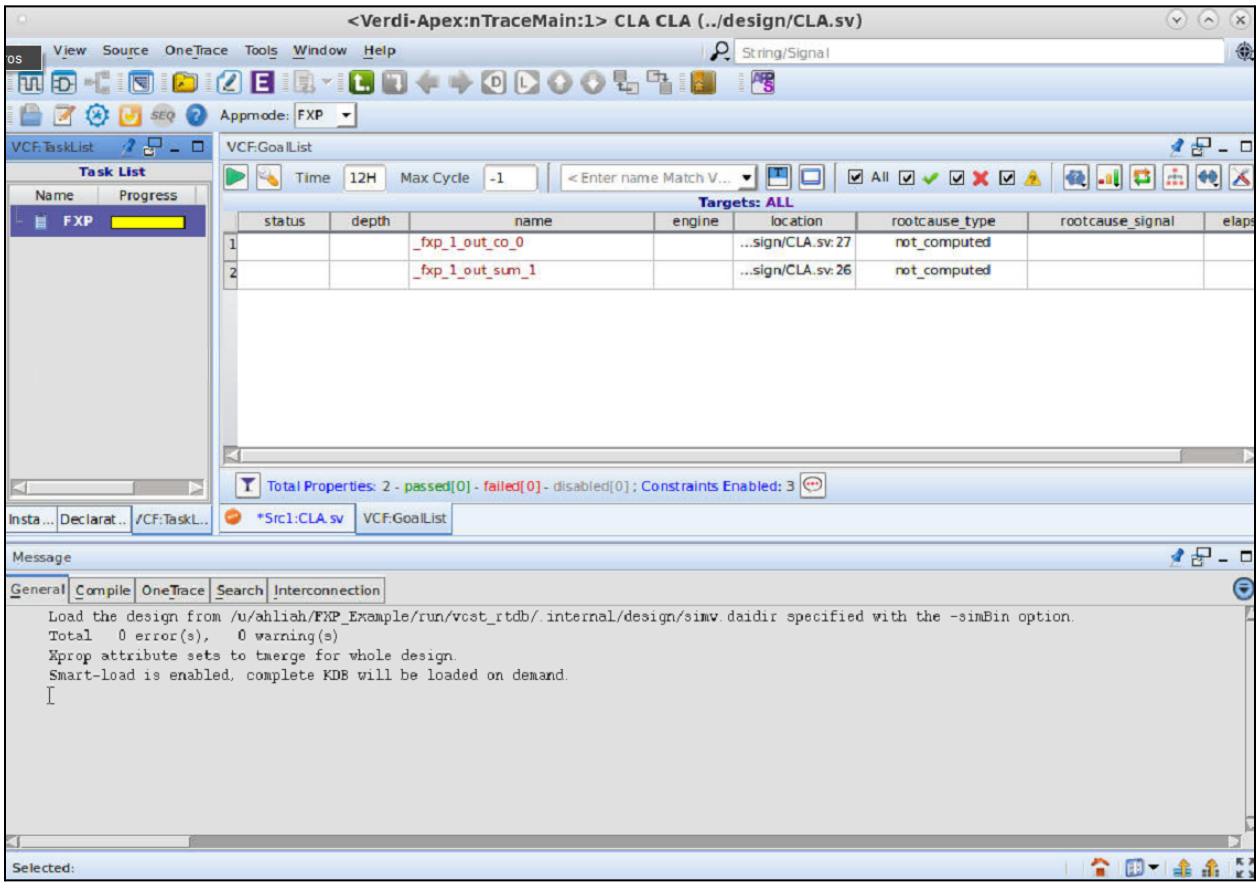


Figure 10. Screen after loading TCL script.

You should see all of your outputs listed here. Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the *VCF:GoalList* tab.

Detecting Errors

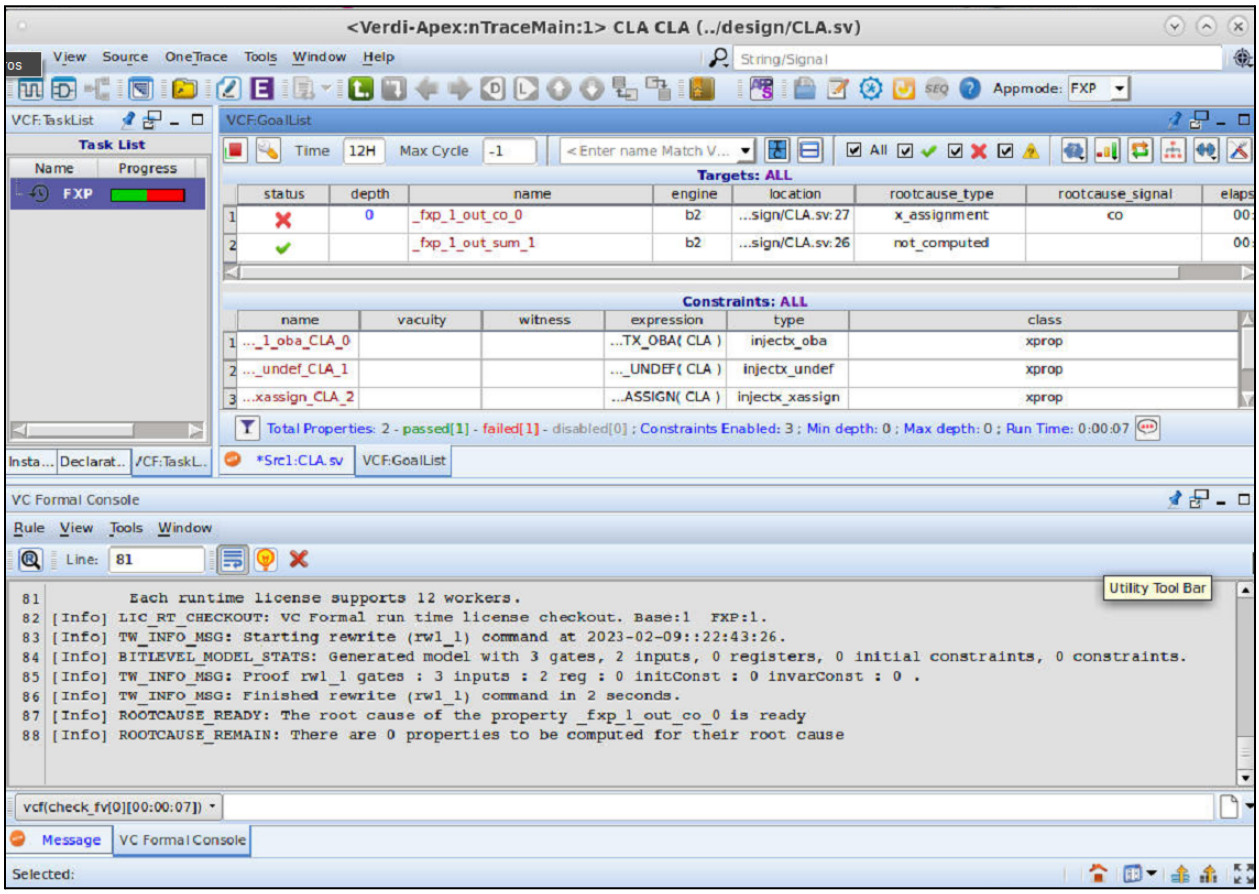


Figure 11. Screen after running TCL script and the status given = ✖.

In *Figure 11* above, we see one ☑ icon for not computed.

The ☑ sign here means that the 'sum' variable on line 26 was not computed and therefore passed FXP checking.

We also see one ✖ icon for x_assignment.

The ✖ sign here means that the 'co' variable on line 27 was assigned the value 'x'.

On the left under *Task List*, we can see that VC Formal was given one task by the FXP app. You can hover over the numbers under *Result* to see the results in detail. You may need to alter the tab size by dragging the side panels or scrolling to the left:

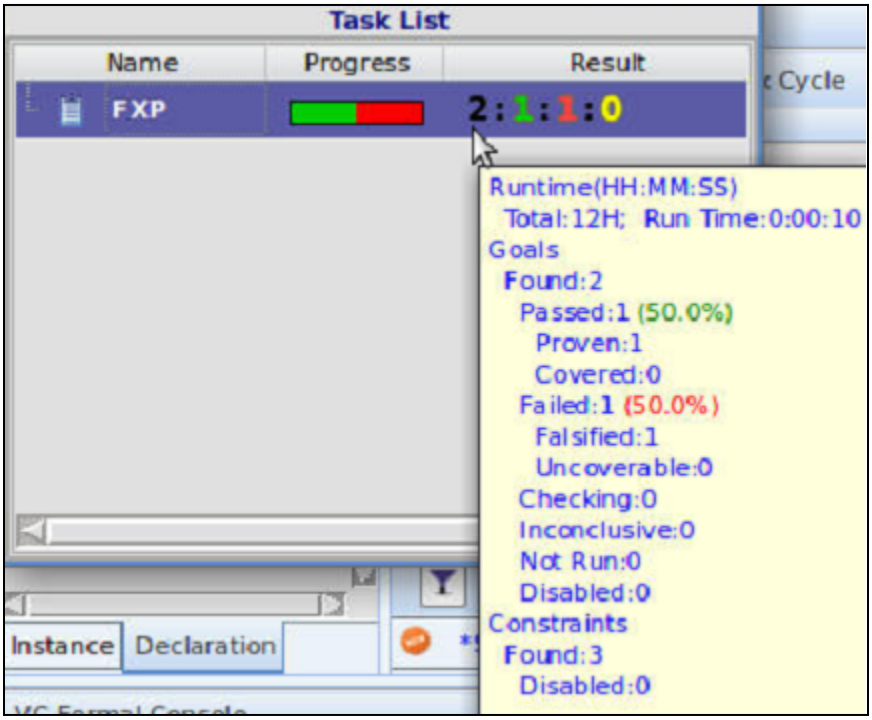


Figure 12. Results of the analyzed script.

Source Tracing

Source tracing is a great way to determine where exactly our errors lie and to get a deeper look into what the issue is. To start, go ahead and double-click on a  icon.

We are going to double-click on the first  (row 1) from *Figure 11*. You should then see a generated waveform as shown below:

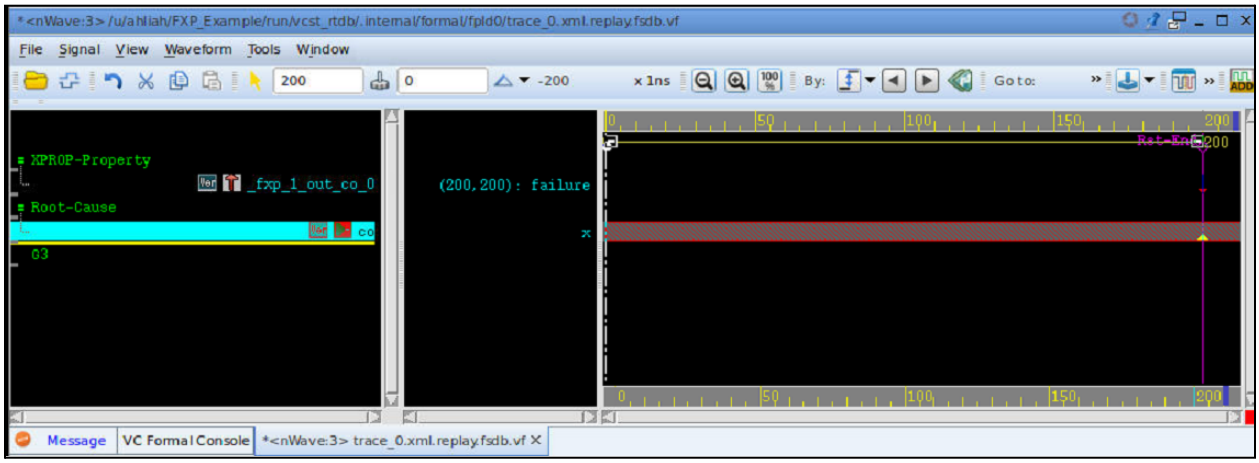


Figure 13. Examining the failed FXP check in our design.

On the left in *Figure 13* above, we see *Root-Cause -> co* signal in green text. Right-click on it the red signal associated with it.

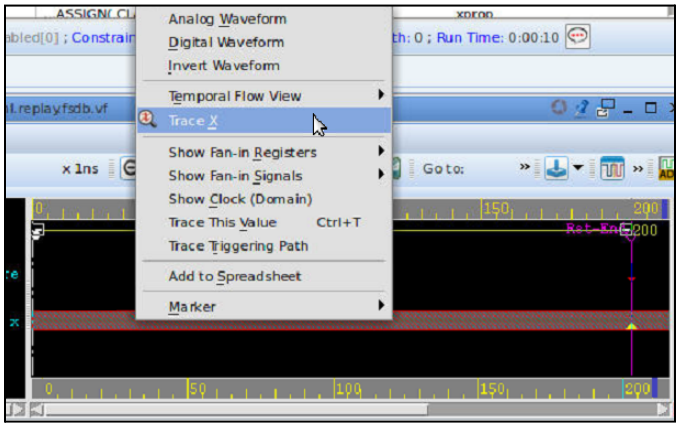


Figure 14. Pop-up from right-clicking on a signal.

As shown in Figure 14 above, this is the general method of source tracing:

Right-click the signal → Show OnceTrace Signals → Trace X

By tracing the source, we are essentially backtracking the X signal to its origin.

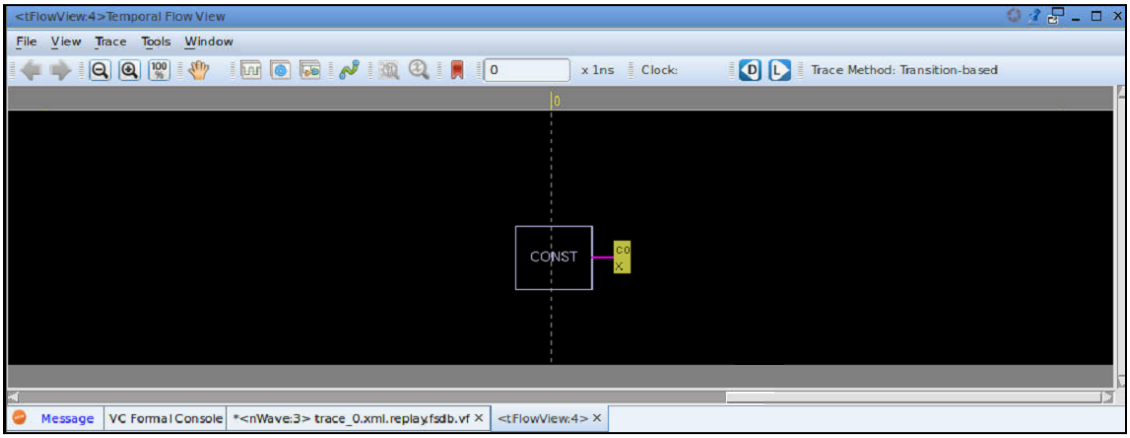


Figure 15. Generated temporal view for schematic.

Source tracing will also bring up a generated temporal view for the schematic of your design. This gives another visual for you to understand how the unknown signal is traveling within your design.

Due to our example driver being a constant, we are shown that the signal is not expandable at time 0 (*Figure 15*). For a better understanding of this temporal view tool, take a look at another example showcasing two latches below:

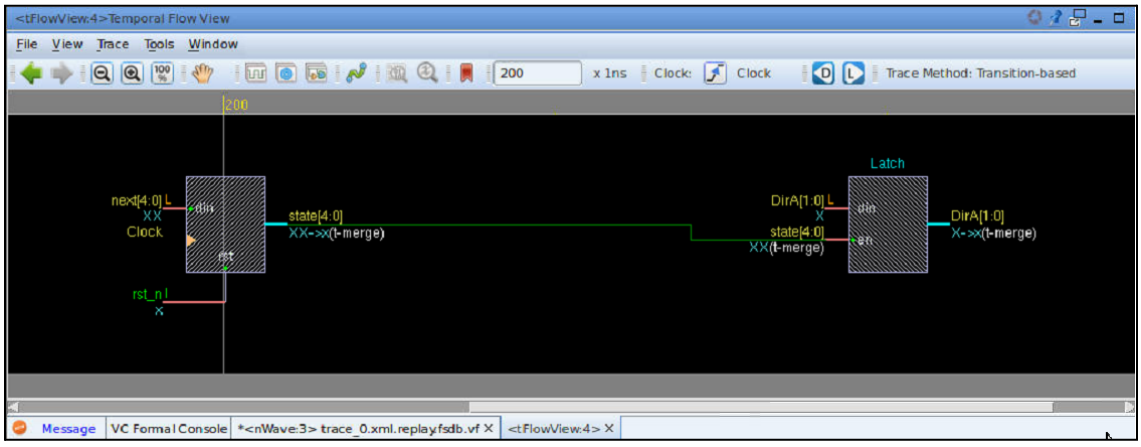


Figure 16. Generated temporal view for a different schematic.

Resolving Errors

To see the code/design where this fault is resulting, you can double-click on the red signal:

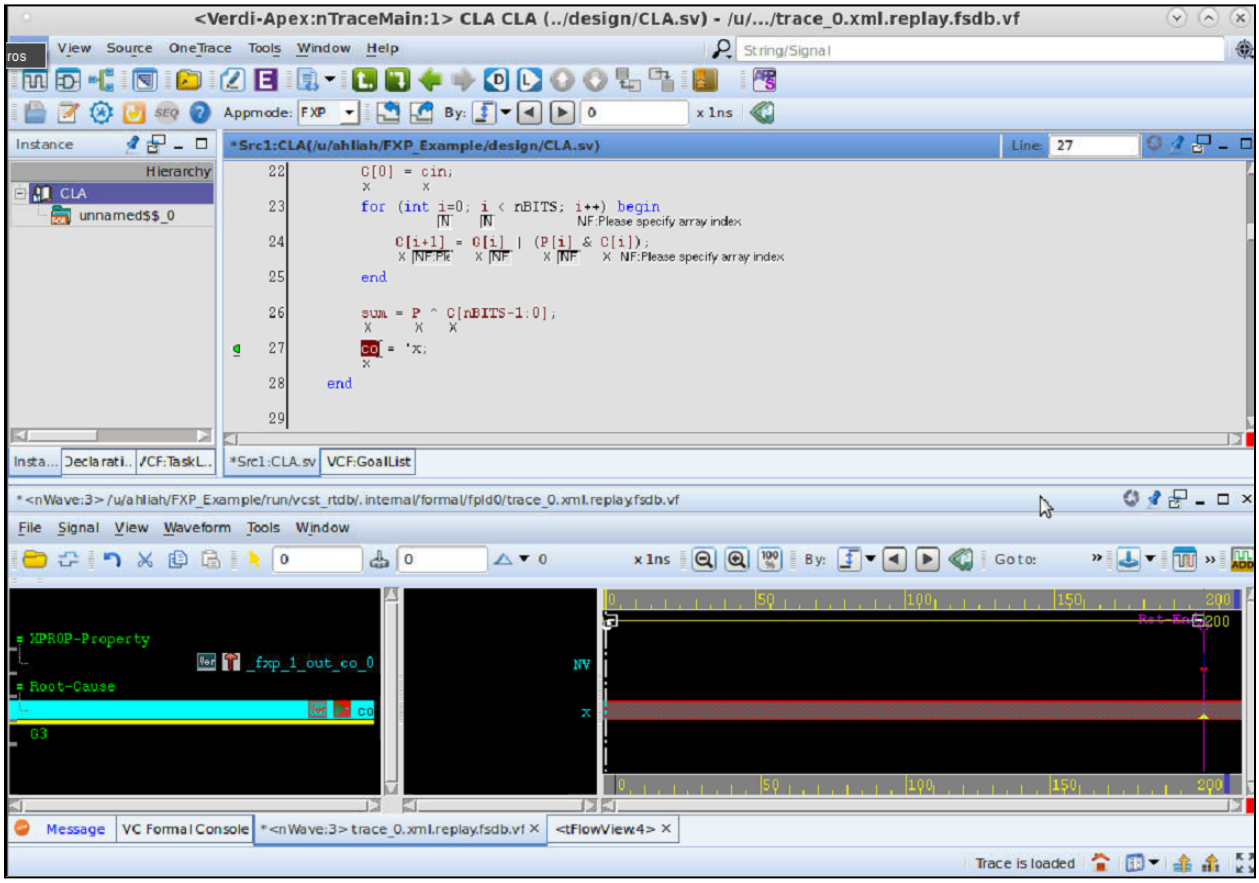


Figure 16. Identified errors within the Design file.

We are then taken to the part of the code that is causing this fault (line 27), with the signal highlighted in red. We have traced X to be from the operation performed on line 27. If we go back and look at the *rootcause_type and rootcause_signal* columns in the *VCF:GoalList* tab for this signal, we can determine the issue is caused by ‘co’ being assigned ‘x’.

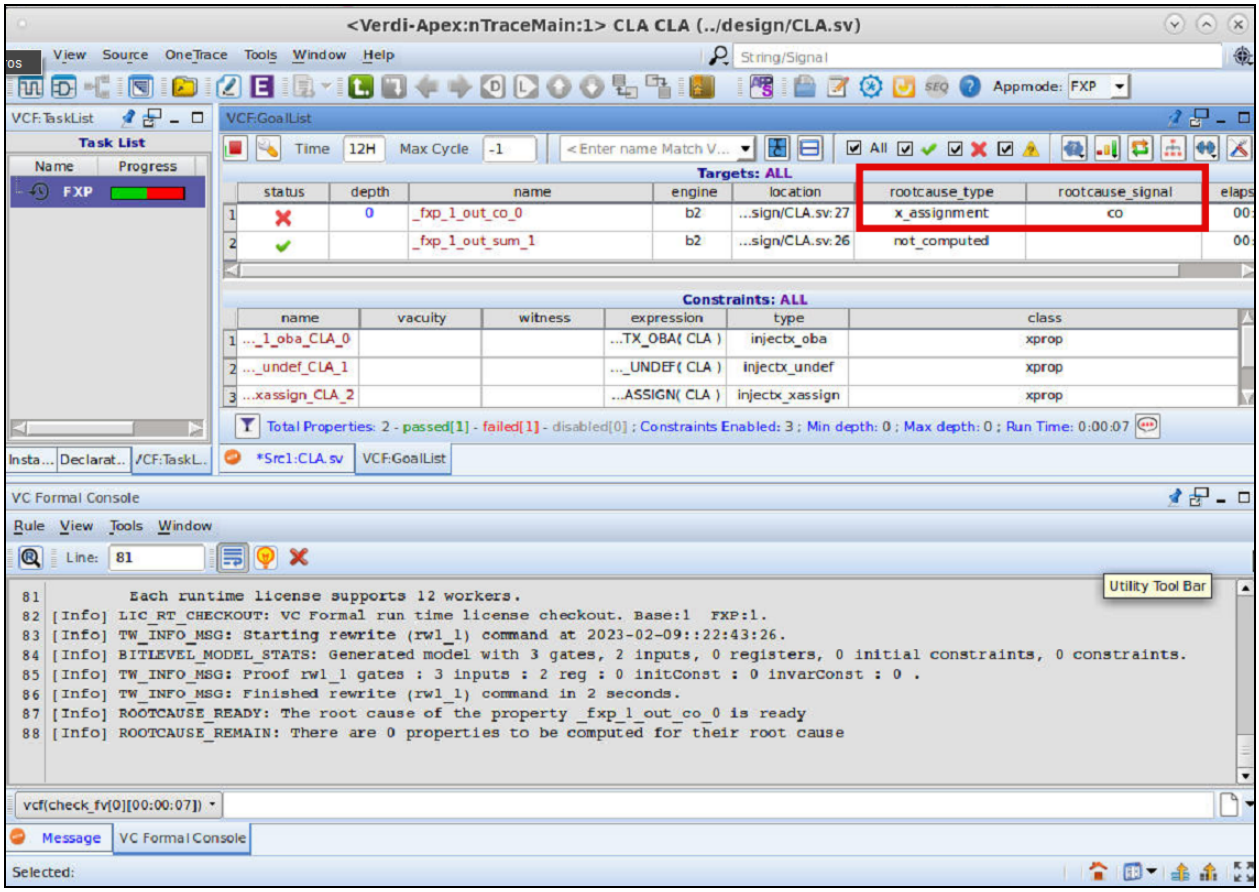


Figure 17. Under "rootcause_type" and "rootcause_signal" columns.

To fix this issue, we need to go alter our design file and make the following changes:

Figure 19. Altered design file to fix the error on line 27 (compare with Figure 3).

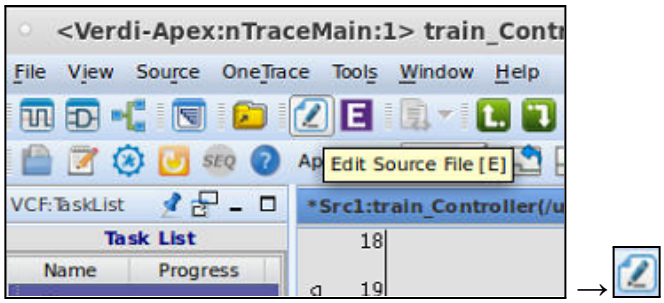


Figure 20. To make changes to the design file in VC Formal, click on "Edit Source File". Don't forget to save your changes!

Next, to complete our changes, follow the steps below to restart VC Formal.

Restarting VC Formal

We can restart VC Formal by clicking on  . If you invoked VC Formal along with the TCL script in the one-line command, then VC Formal will automatically load the same TCL file again. If you invoked VC Formal without the TCL script, then you will have to manually load the script again.

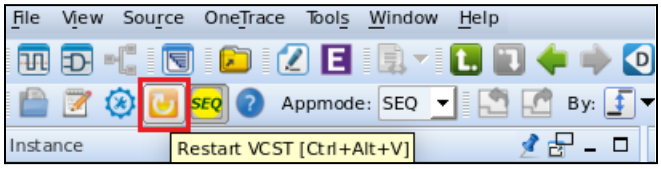


Figure 26. Location of the restart button in VC Formal window.

After the application and TCL file is loaded, click the green play button and we should see no errors.

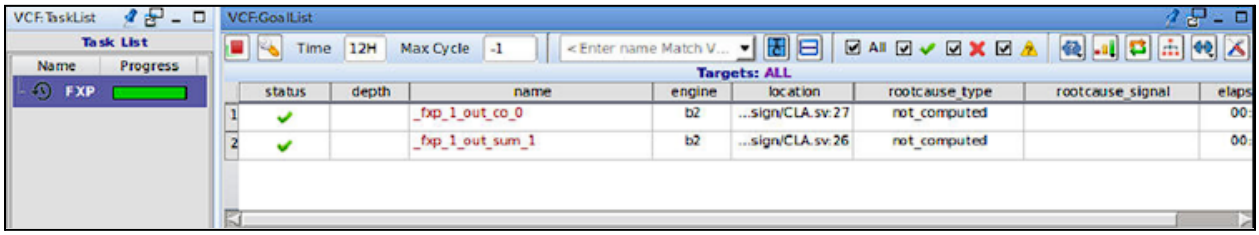


Figure 27. Results after fixing errors and restarting VC Formal and reloading TCL script.

Appendix

Name	Description	Type	Default
abs	Abstractions	Inject x; Observe x	Yes
bbin	Black box inputs	Observe x	Yes
bbout	Black box outputs	Inject x	No
in	Primary inputs	Inject x	No
oba	Array bound violations	Inject x	Yes
out	Primary outputs	Observe x	Yes
snip	Snips as inputs	Inject x	Yes
snip_drv	Snips for observations	Observe x	Yes
undef	Undefined behavior	Inject x	Yes
undriven	Undriven nets	Inject x	Yes
uninit	Uninitialized registers	Inject x	Yes
xassign	Explicit X-assignments	Inject x	Yes
unresin	Unresolved module inputs	Observe x	Yes
unresout	Unresolved module outputs	Inject x	No

Table 1.1. FXP Property Types and default options.