Synopsys® VC Formal Tutorial
Automatically Extracted Property (AEP)

Portland State University

Disclaimer:

Functional auto checks in Synopsys VC Formal AEP 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

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 “AEP_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 AEP analysis for us.

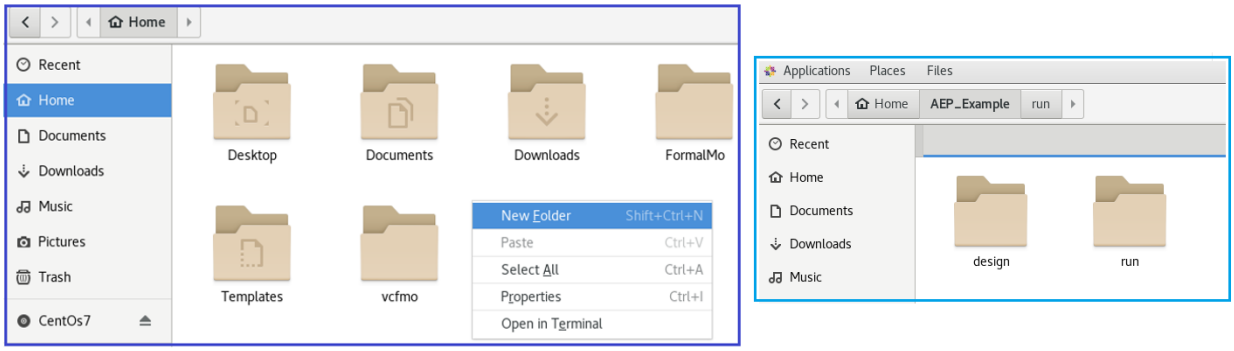
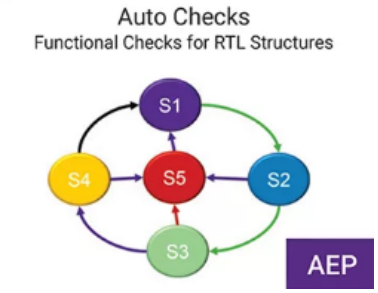


Figure 1. Creating folders to organize design and TCL files.

About and Usage of AEP



The “AEP” app in VC Formal is used as an analysis tool for auto-checking a design for out-of-bound arrays, arithmetic overflow, X-assignments, simultaneous set/reset, full case, parallel case, multi-driver/conflicting bus, and floating bus checks. Without the AEP app, these checks would each require their own dedicated simulation tests, consuming lots of time and energy.

The possible checks offered through the AEP app along with their descriptions are listed in ***Appendix Table 1***. This is a resource for you to know all the AEP checks that can be ran on your design. Labeled under *Switch* in the table are the commands you will use in your TCL file to reflect such analysis. To assign multiple switches, we can also use the **+** operator in such syntax:

```
-aep x_assign+set_reset
```

In this tutorial, we will simply be applying all the verifications on our design *(recommended)* with the switch command.

Property (Analysis) Type	Switch	Description
All AEP Checks	-aep all	All possible AEP checks

Table 1.2. Example of “All AEP Checks” property type derived from Table 1 in the Appendix.

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.

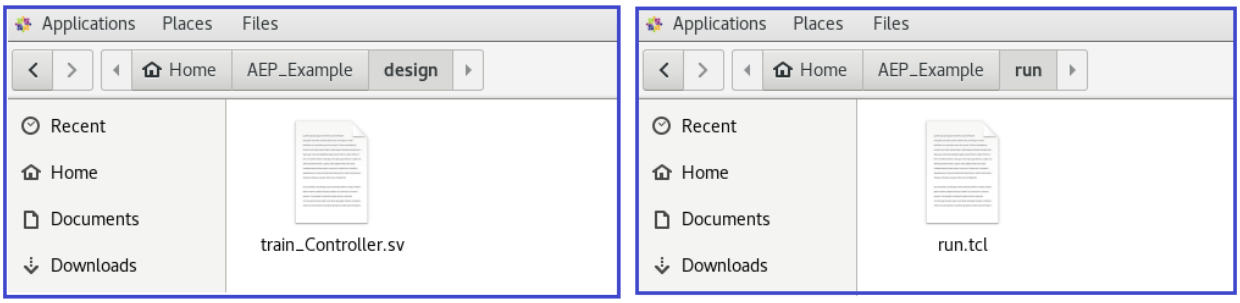


Figure 2. Showing the SystemVerilog and TCL files in the correct folder locations.

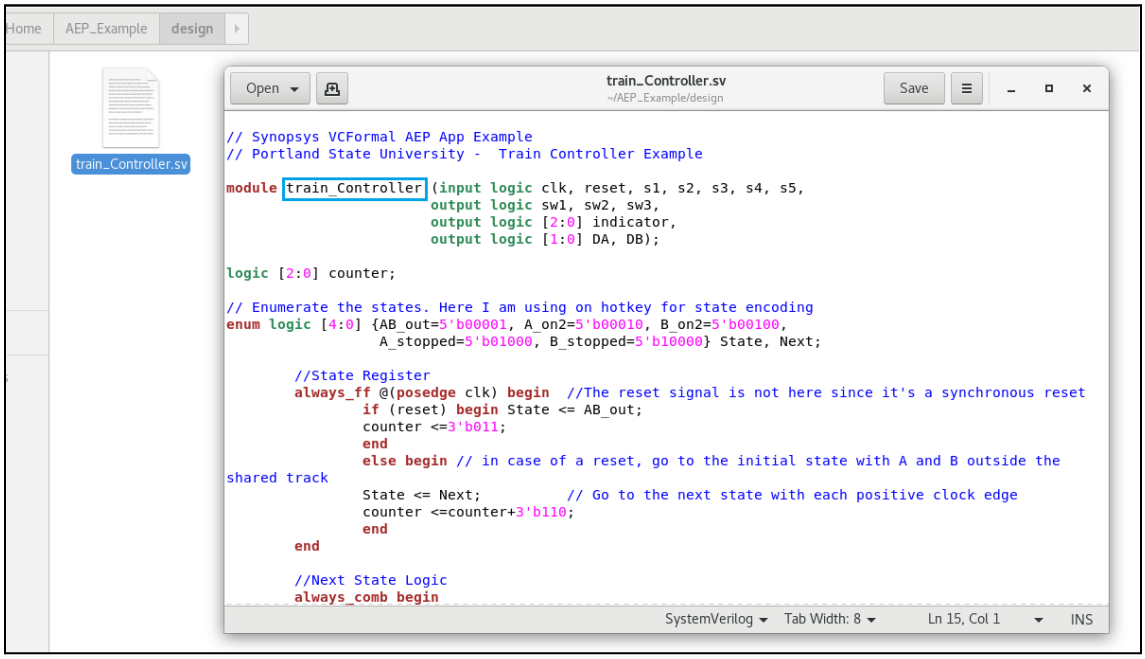


Figure 3. Example of the design we are using in this tutorial.

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.

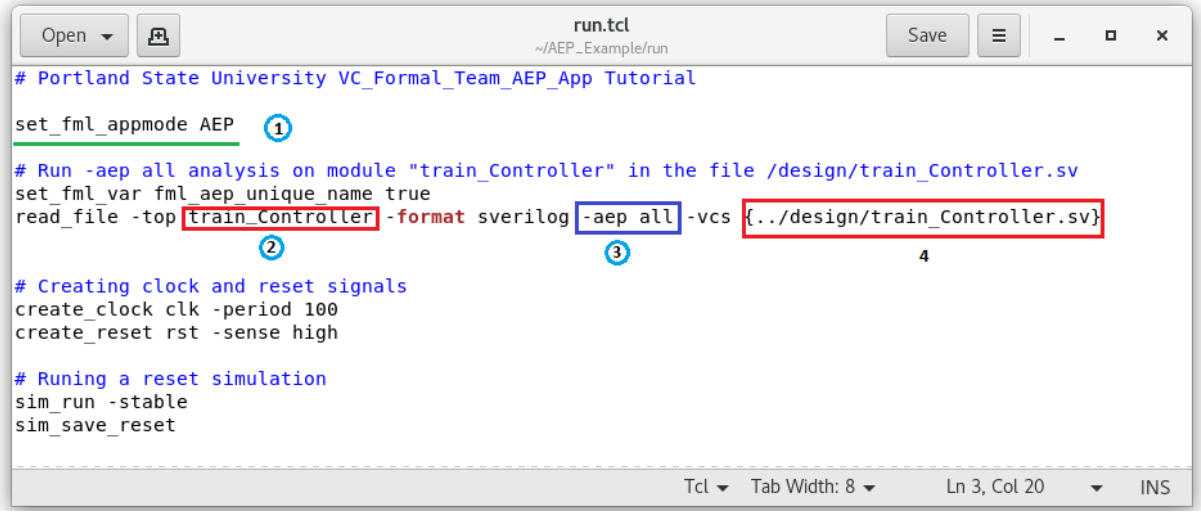


Figure 4. Annotated TCL template file.

(1) Instruction that sets the appmode to AEP in VC Formal.
(2) Name of the main module as established in the Design file.
(3) The property type identifier switch name for desired AEP analysis (see *Table 1* in Appendix).
(4) Design file location so VC Formal knows where to find the file.

As mentioned before, VC Formal is case 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 (train_Controller.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 AEP 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
```

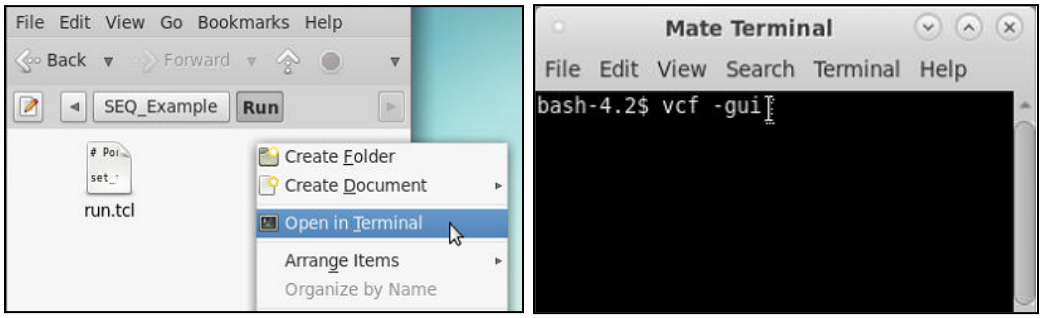


Figure 5. Invoking VC Formal in the terminal.

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: 

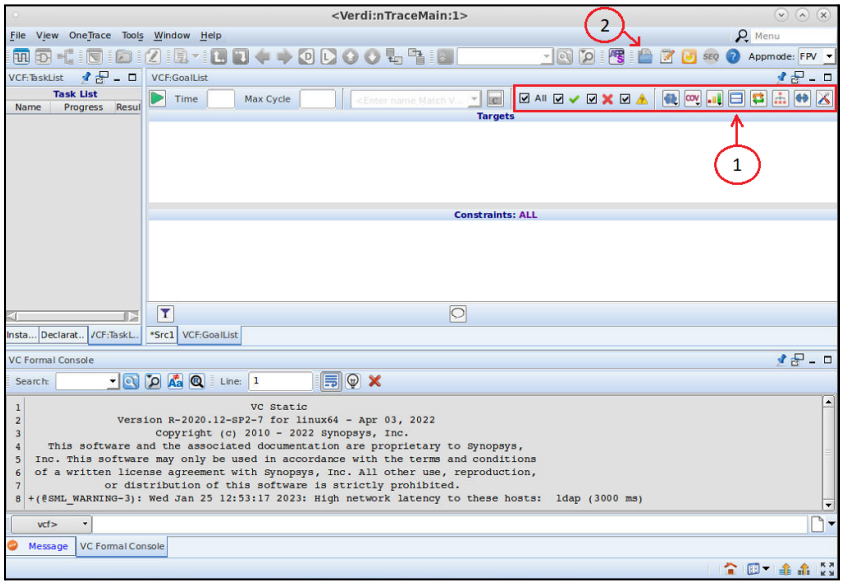


Figure 6. VC Formal GUI introductory screen.

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:

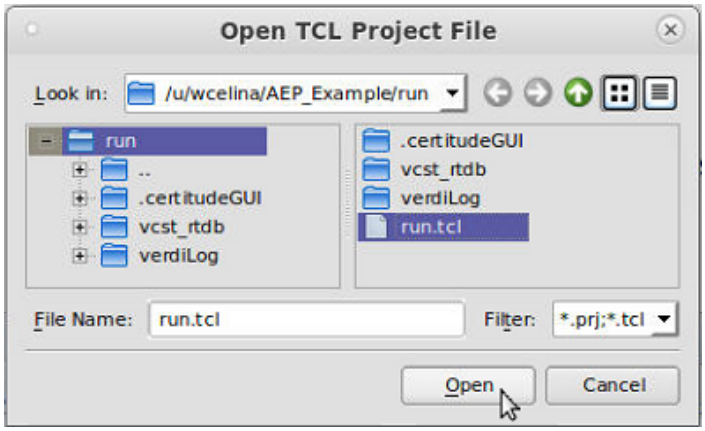


Figure 7. Selecting TCL file.

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
```

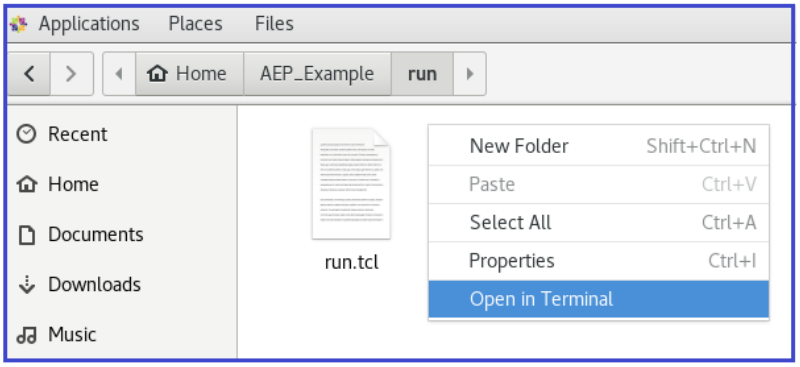


Figure 8. Opening terminal in the ‘run’ folder.

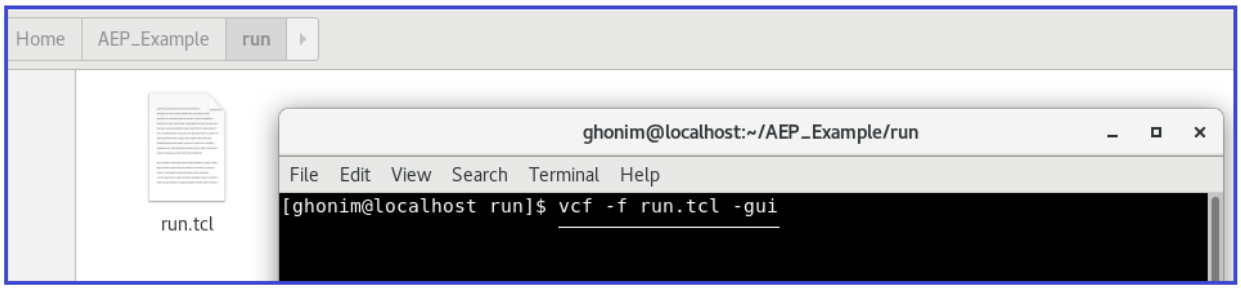


Figure 9. Invoking VC Formal and TCL script in the terminal.

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:

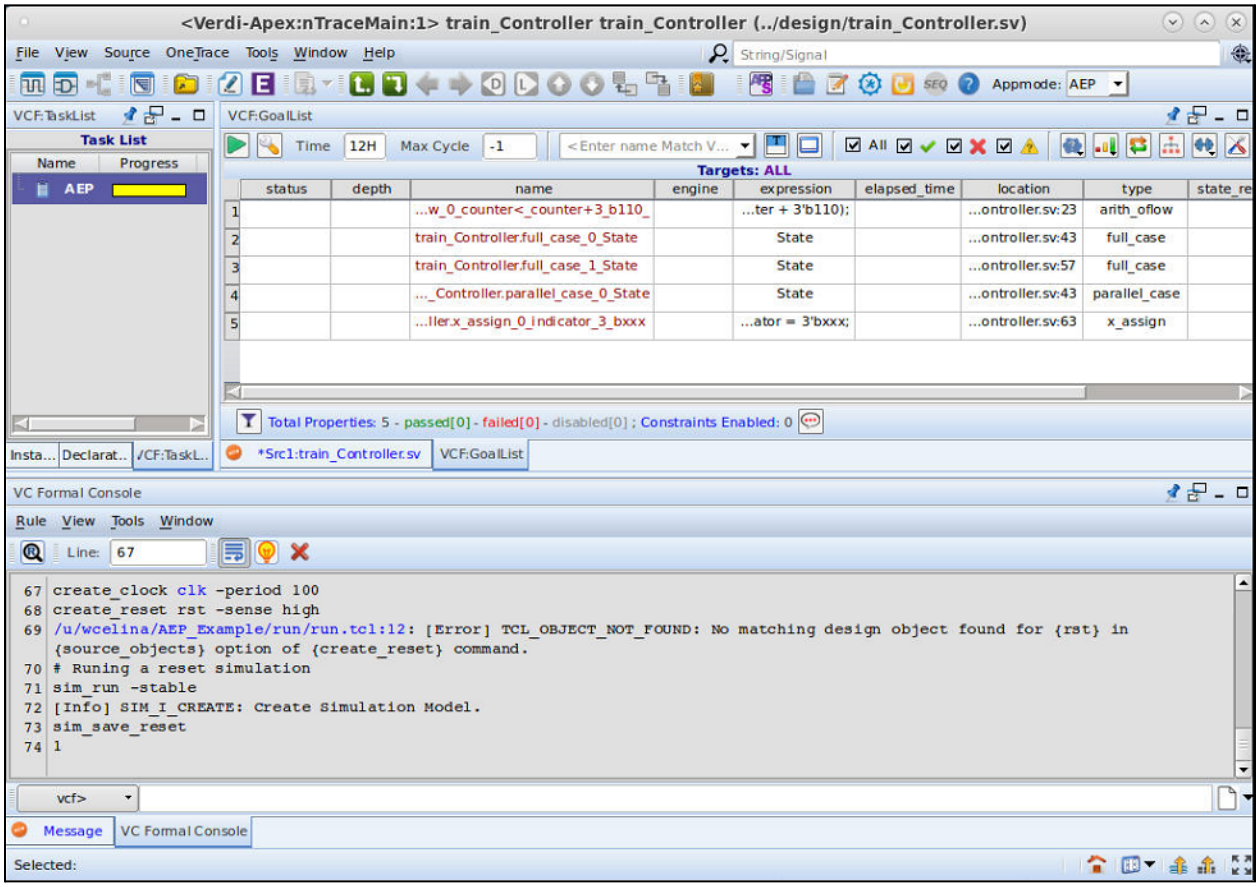


Figure 10. Screen after loading TCL script.

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

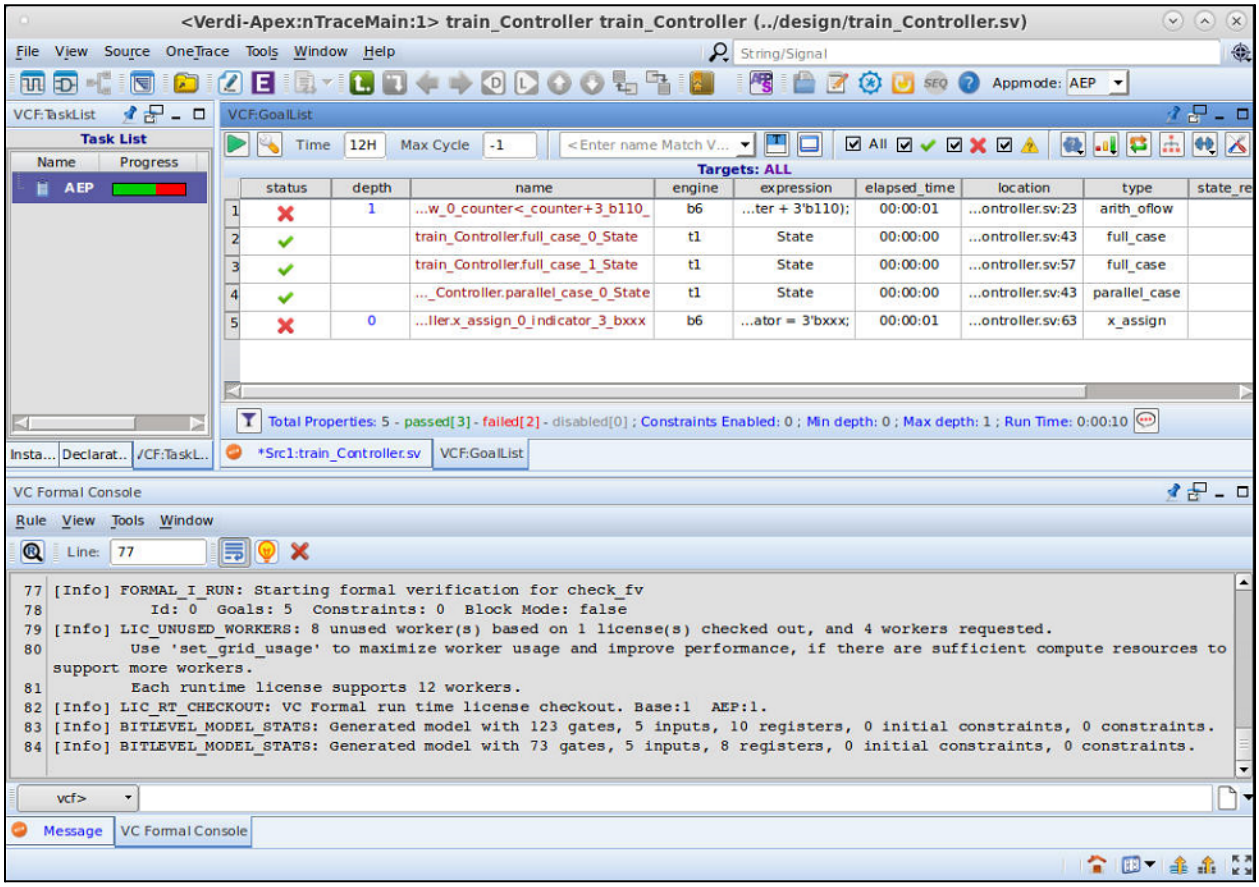


Figure 11. Screen after running TCL script and the status given = ✖.

In *Figure 11* above, we see three ✔ icons, one for parallel case and two for full case.

The ✔ signs here mean that the priority case on line 57 of our design has a full case and that the unique case on line 43 has a full case, and only one unique case, so it can be checked in a parallel fashion.

We also see two ✖ icons; one for arithmetic overflow and one for x_assignments.

The ✖ signs here mean that we have an arithmetic overflow on line 23 with our counter signal. There is also an active x assignment in the design on line 63.

On the left under *Task List*, we can see that VC Formal was given one task by the AEP 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:

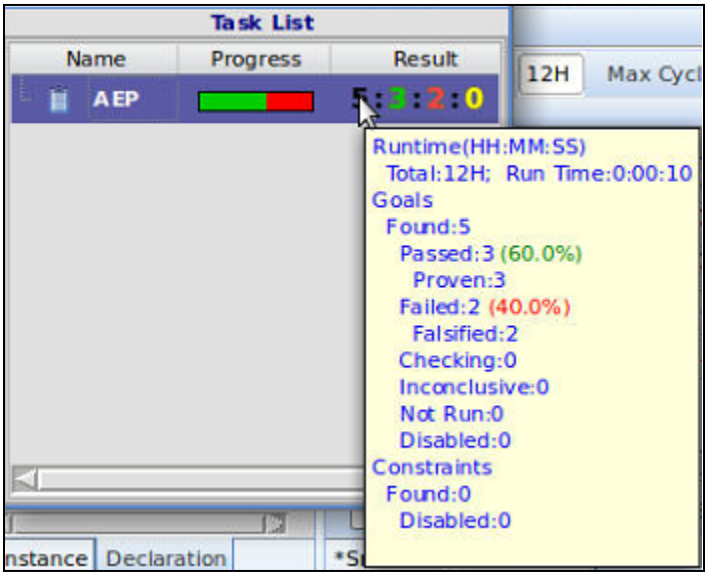


Figure 12. Results of the analyzed script.

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 an  icon.

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

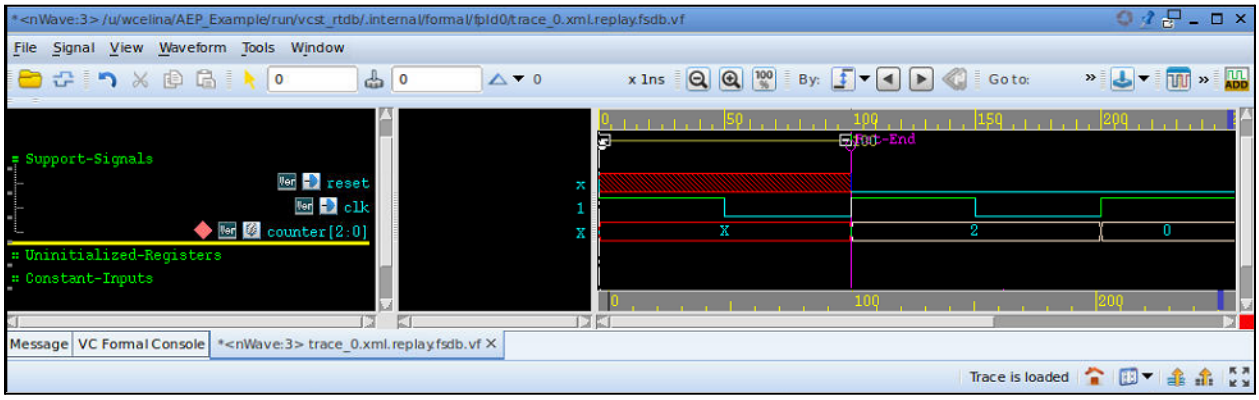


Figure 13. Examining the failed AEP check in our design.

On the left in *Figure 13* above, we see *Support-Signals* in green text. Go down three lines to the text that says *counter[2:0]* and right-click on it.

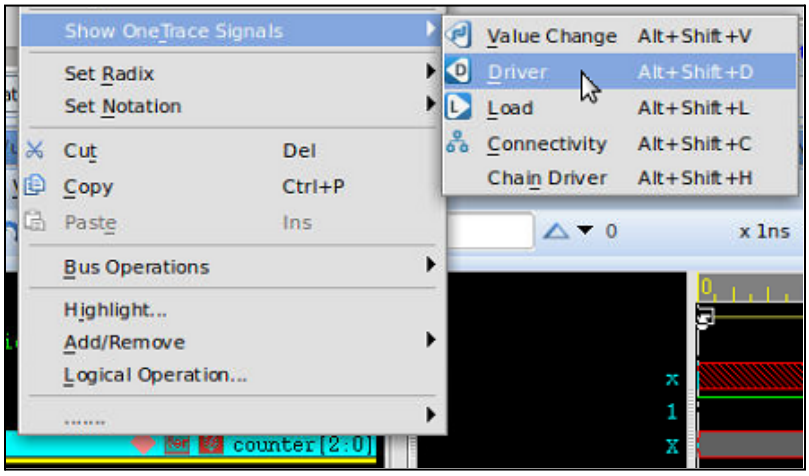


Figure 14. Pop-up from right-clicking on a signal.

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

Right-click the signal → Show OnceTrace Signals → Driver

You can also use the short-cut keys: **ALT + SHIFT + D**

By tracing the source, we are essentially backtracking it to the driving signal of the output in order to find the discrepancy. We then get these additional waveforms, showing the driving signal of this “counter” signal, which is the previous counter value.

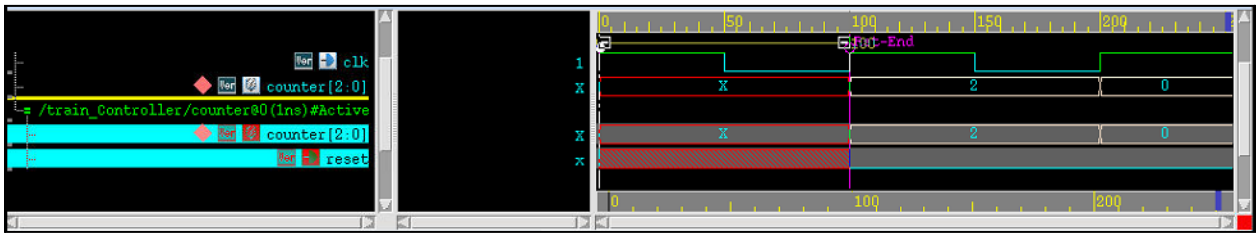


Figure 15. Tracing the source to find the discrepancy. After one trace.

Resolving Errors

To see the code/design where this fault is resulting, we go to the *type* column in the *VCF:GoalList* tab and double-click on *arith_oflow*.

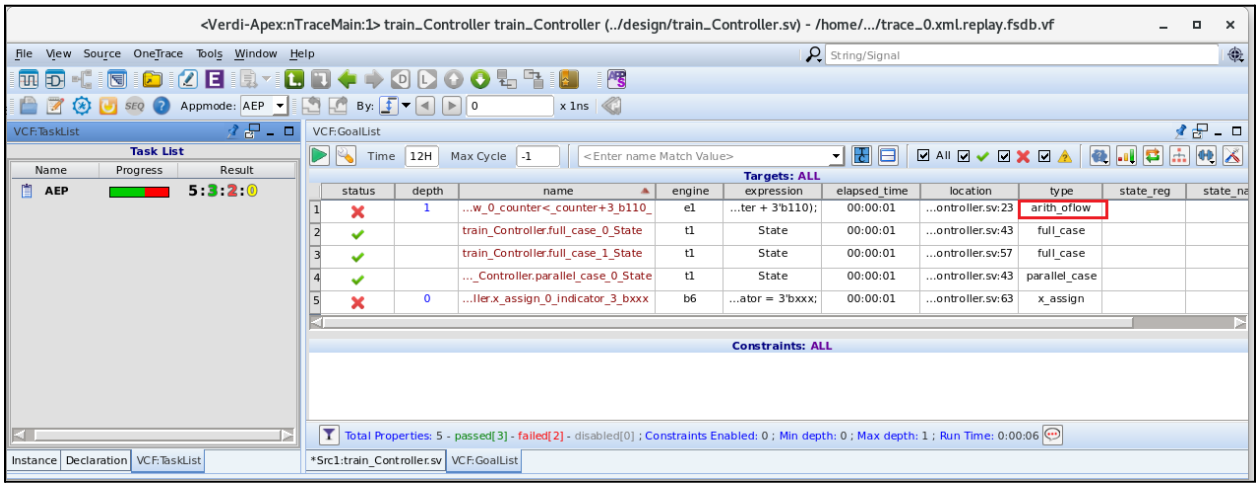


Figure 16. Under "type", double-click on "arith_oflow".

We are then taken to the part of the code that is causing this fault (arithmetic overflow), with the operation highlighted in blue, and the signal highlighted in red.

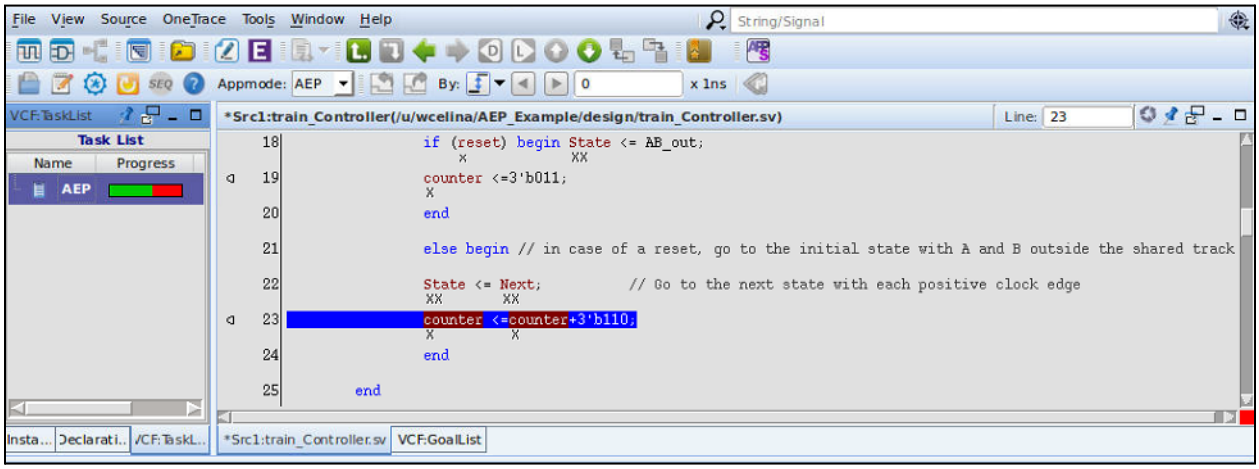


Figure 17. Identified errors within the Design file.

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

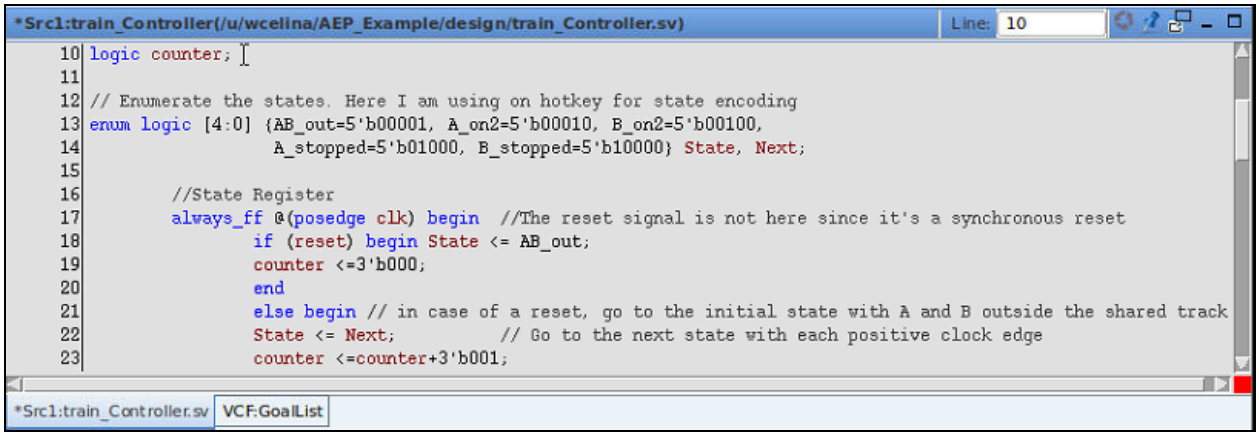


Figure 18. Altered design file to fix the error on lines 10, 19, and 23 (compare with Figure 3).

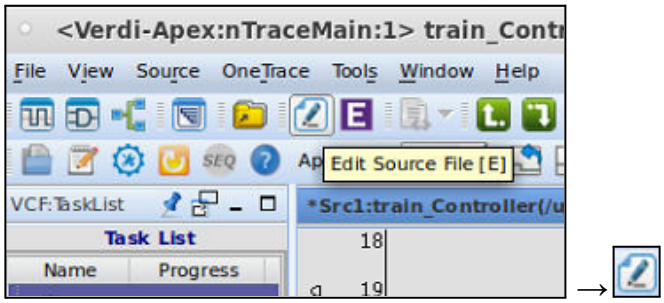


Figure 19. To make changes to the design file in VC Formal, click on "Edit Source File". Don't forget to save your changes.

Now let's go look at the other incomplete task on line 5:

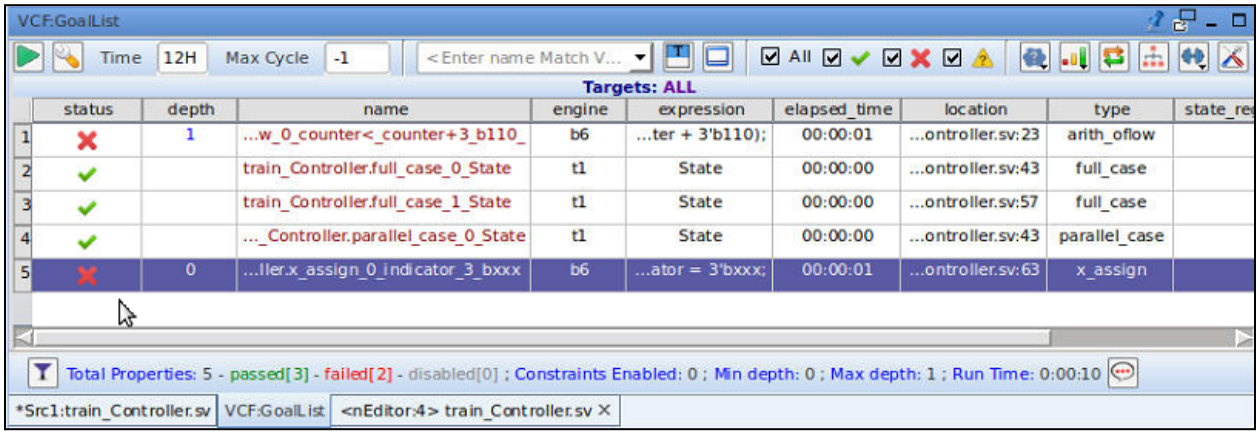


Figure 20. Targeting the next error to resolve.

We are going to follow the same steps above to locate this error. As before, we will trace the source back to the driving signal in order to find the discrepancy.

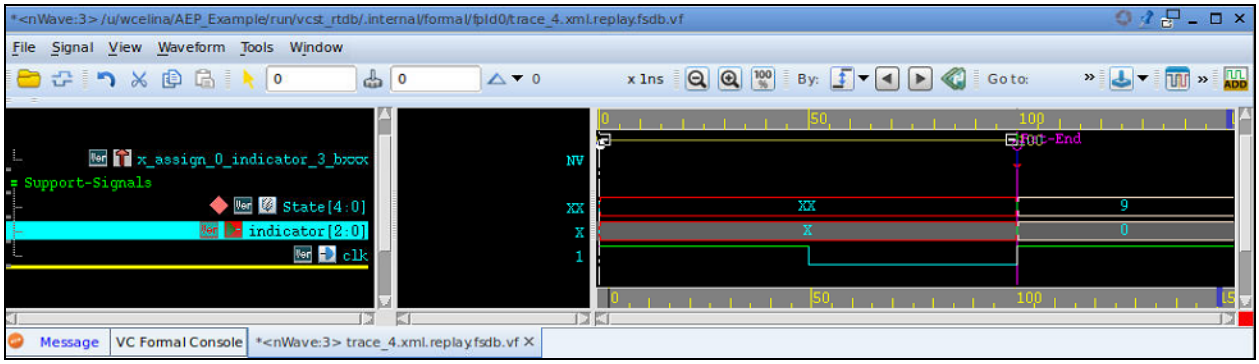


Figure 21. Looking into x_assign waveform.

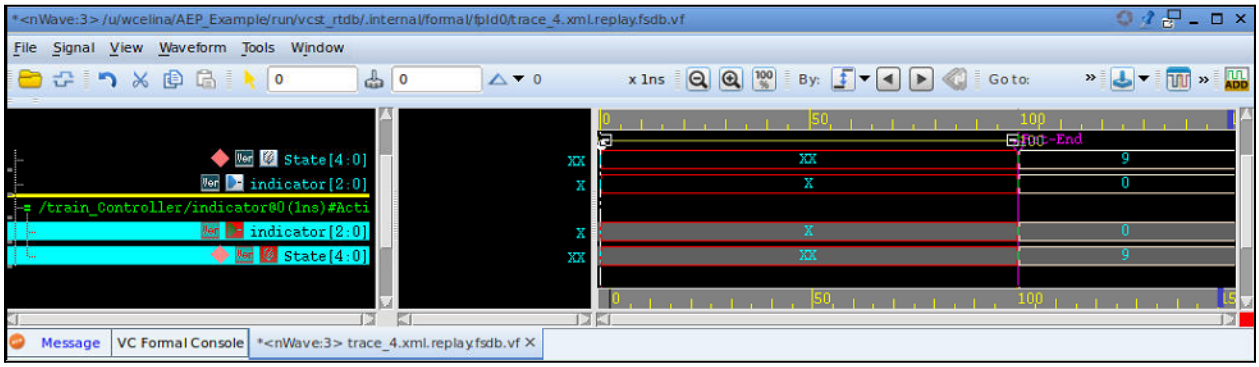


Figure 22. The waveform after tracing "indicator[2:0]" once.

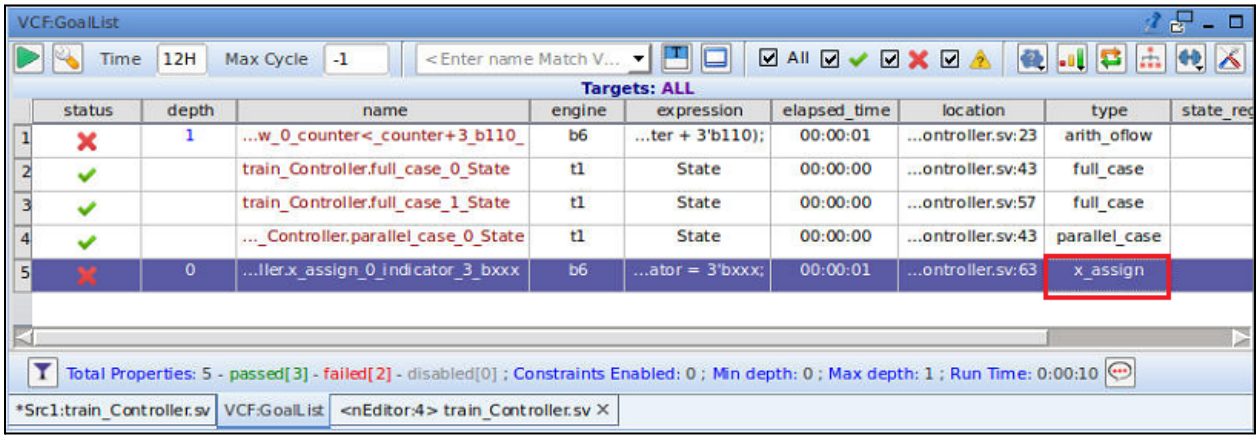


Figure 23. Double-clicking on the error type

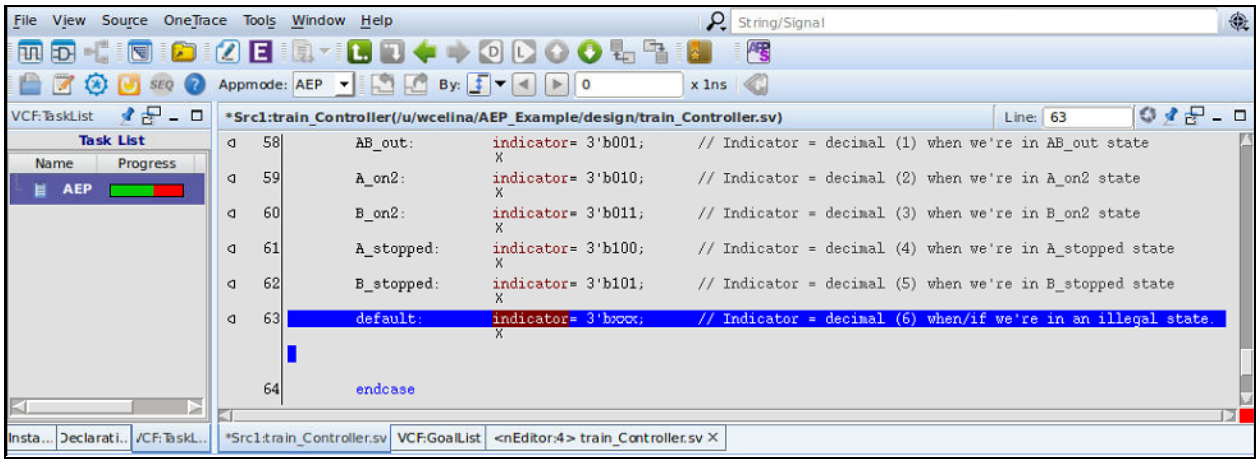


Figure 24. *Identified errors within the design file.*

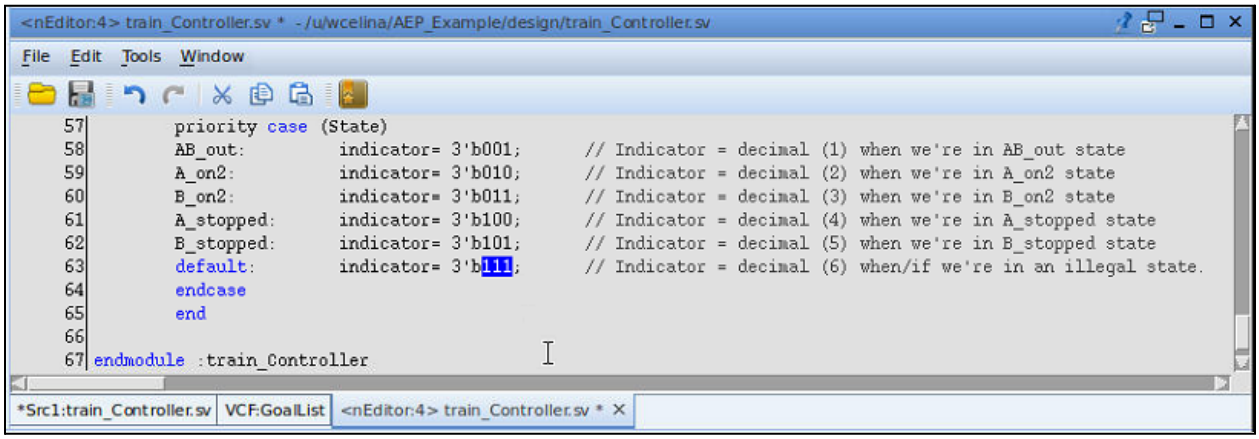


Figure 25. Saving changes made to the design file.

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.

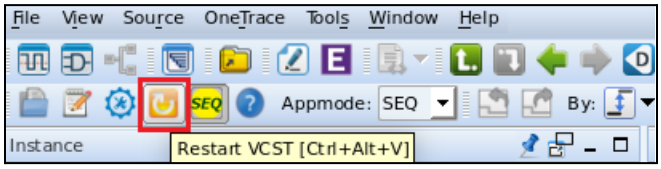


Figure 26. Location of the restart button in VC Formal window.

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

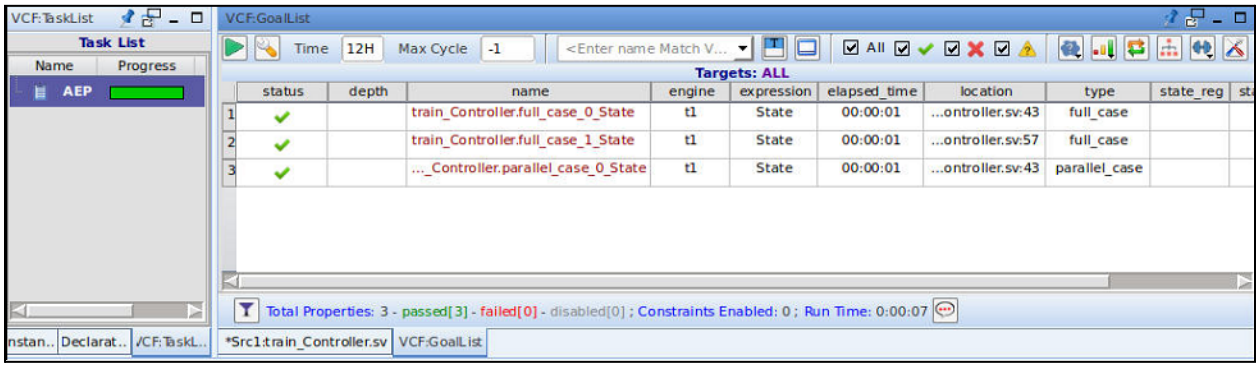


Figure 27. Results after fixing errors and restarting VC Formal and reloading TCL script.

Appendix

Property (Analysis) Type	Switch	Description
Arithmetic overflow	-aep arith_oflow	Checks for arithmetic overflow. For example, when no limit is set on a counter.
Conflict bus checks	-aep conflict_driver	Bus contention, conflicting driver, or x-propagating through a bus.
X_assignment reachability	-aep x_assign	Verifies that x assignments to signals are never activated.
Simultaneous set/reset	-aep set_reset	Verifies that set and reset signals are never asserted at the same time for asynchronous set and reset sequential circuits.
Array boundaries	-aep bounds_check	Makes sure we are not accessing data in an array outside the array boundaries.
Pragma Check Priority Case	-aep full_case/priority_case	Generated from full case synthesis directive. Case expression must match at least one case item at every cycle.
Pragma Check Unique Case	-aep parallel_case/unique_case	Generated from parallel case synthesis directive. Case expression matches only one case item at every cycle.
Floating bus	-aep floating_bus	Only for tri-state drivers. Verifies at least one driver is active at all times.
Multiple driver bus	-aep multi_driver	Only for tri-state drivers. Verifies only one driver is active at a time.
Single State Starvation (SSS)	-aep fsm_sss	Checks for infinite wait in a single state. FSMs can get stuck forever in a given state.
FSM Deadlock	-aep -fsm_deadlock	Checks for infinite wait in a single state. Each state has its own deadlock assertion
FSM Livelock	-aep -fsm_livelock	Checks for FSM stuck between multiple states. Each state has its own livelock assertion
FSM Unionlock	-aep -fsm_unionlock	Checks for any FSM locks, covering both deadlock and livelock. Each state has its own unionlock assertion
All AEP Checks	-aep all	All possible AEP checks

Table 1.1. AEP App Property Types. Identifiers and descriptions of all analyses offered.