



Portland State
UNIVERSITY

DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

SYNOPSYS VC FORMAL FINAL REPORT

INDUSTRY SPONSOR AND ACADEMIC ADVISOR
PROFESSOR XIAOYU SONG

ECE 413 CAPSTONE TEAM#7

MOHAMED GHONIM
AHLIAH NORDSTROM
DMYTRO PRYSTUPA
ALEXANDER KIM
CELINA WONG

ghonimpdx@gmail.com

ahliahnordstrom7@gmail.com

dimaprystupa12@gmail.com

kalex3030@gmail.com

wcelinapdx@gmail.com

VERSION:

1.4

REVISED:

06/15/2023

Table of Contents

Executive Summary	3
Background	4
Product Design Specification (Requirements and Stakeholders)	5
Objectives and Deliverables.....	7
Approach	8
Design	9
Test plan.....	18
Results	19
Challenges	20
Collaboration Tools	23
Summary	27
Appendices.....	30
3030	
Appendix 2: VC Formal Tutorials Table of Contents	34
Appendix 3: Sample ECE 582/682 Projects performed using our tutorials	
Appendix 4: Weekly Progress Reports	
References	(Last Pages)

Disclaimer:

This report includes figures and pictures obtained from Synopsys VC Formal and Cadence JasperGold tools, which are proprietary software products. The usage of these figures and pictures in this report is solely for the purpose of demonstrating the capabilities and functionalities of the tools in the context of our capstone project. The inclusion of these visuals does not imply any endorsement or sponsorship by Synopsys Inc. or Cadence Design Systems Inc. Furthermore, all intellectual property rights associated with the figures and pictures remain with their respective owners.

Executive Summary

The capstone project, *Formal Verification Using Synopsys VC Formal*, focuses on providing comprehensive tutorials for undergraduate and graduate students at Portland State University. Our objective is to equip students with practical knowledge and proficiency in formal verification techniques using Synopsys VC Formal, a powerful tool for silicon design and verification.

Synopsys is a renowned electronic design automation (EDA) company and developed VC Formal as a leading tool for formal verification. By leveraging logical and mathematical principles, formal verification ensures the correctness of complex RTL designs, particularly in critical applications such as airplane controller chips. VC Formal currently consists of 12 distinct applications, each serving specific verification purposes; in this project, we will focus on 11 of their applications.

Our team focuses on various essential components, including introducing VC Formal to users, simplifying the installation process, and creating comprehensive tutorials and videos. The tutorials cater to users with varying levels of familiarity with VC Formal, providing step-by-step guidance and practical examples for each of the apps.

In collaboration with our sponsor, Dr. Song, we are working to develop user-friendly tutorial documentation for the installation, utilization, and creation of examples using VC Formal. This project addresses the need for accessible resources in formal verification within academic settings.

Through this project, we aim to bridge the gap between academia and industry by empowering undergraduate and graduate students at Portland State University to apply formal verification techniques effectively. The tutorials and examples we provide will enhance students' understanding and proficiency in formal verification using Synopsys VC Formal, preparing them for future challenges in the field of silicon design and verification.

Background

Digital logic designs, especially RTL (Register Transfer Level), have become increasingly complex, posing challenges for effective verification methods. Traditional techniques like simulation are impractical for verifying large and intricate designs due to time constraints. Fuzz testing, a non-exhaustive simulation method, provides some confidence but falls short in critical aspects of design verification.

To address these limitations, formal verification has emerged as a crucial approach. By leveraging logic and mathematical principles, formal methods can establish the correctness or incorrectness of designs. While formal verification was once a niche tool reserved for complex designs, it has gained mainstream popularity since 2015. However, the industry faces a shortage of skilled engineers in this field, and academic instruction in formal verification is limited.

Our project aims to bridge this gap by acquiring, applying, and documenting the use of Synopsys VC Formal, one of the industry's most powerful commercial formal verification tools. Our goal is to teach formal verification within the engineering department at Portland State University.

Currently, VC Formal training is primarily available in the industry, with limited accessibility in academia. Existing resources, such as Synopsys' extensive user manual of over 1000 pages and introductory videos, cater to industry professionals and lack academic focus. Expecting students to read such lengthy manuals for a class project is unrealistic. Therefore, we intend to develop concise yet comprehensive tutorials, incorporating written and visual components specifically tailored for academic settings.

By providing accessible and easy-to-understand tutorials, we aim to empower students at Portland State University to grasp the concepts and practical implementation of formal verification using Synopsys VC Formal. Our focus is on delivering practical knowledge that enhances their understanding of this verification technique, enabling them to tackle complex digital designs with confidence.

Product Design Specification (Requirements and Stakeholders)

Concept of Operation / User stories:

The primary objective of our project is to provide PSU students, specifically those enrolled in Dr. Song's ECE 582 and other M.S./Ph.D. classes, with comprehensive tutorial documentation on using VC Formal. The tutorials will guide students in understanding and completing coursework related to formal verification. Dr. Song will grant his students access to these tutorials, ensuring their availability for educational purposes. The tutorial documentation should have a long lifecycle, with updates made only if major changes occur in the VC Formal program. The success of our project will be measured by the ability of individuals with little to no knowledge of VC Formal to navigate and utilize our tutorial documentation effectively.

Stakeholders:

The stakeholders involved in our project include:

- **PSU student body:** They will benefit from gaining practical knowledge and experience in using VC Formal, enhancing their employability.
- **Professor Xiaoyu Song:** He can integrate VC Formal into PSU classes, providing students with an opportunity to apply various verification techniques using a state-of-the-art tool.
- **PSU Electrical and Computer Engineering Department:** By teaching this specialized tool, the department can improve its ranking and enhance students' career prospects. Faculty members will also have access to VC Formal tutorials for research and publications.
- **Synopsys:** As a key supporter, Synopsys will have its VC Formal tool taught to a significant number of graduate students each year. This will increase the pool of potential employees who are already familiar with Synopsys tools, making it more convenient for employers to deploy these tools in the industry. The increased popularity and usage of VC Formal in academia may lead to wider adoption in the industry, further expanding Synopsys's customer base.

Requirements:

Must:

- Thoroughly understand the basic functions of VC Formal and its "apps" (in priority order).
- Install and configure an operational environment for VC Formal on a local PC at PSU.
- Document how to connect to PSU labs remotely using a VPN and a Linux client such as MobaXterm.
- Explore the functionalities and usages of VC Formal.
- Document sequential equivalence checking in the VC Formal (SEQ) app.
- Create a thorough tutorial documenting LTL model-checking in VC Formal using SystemVerilog Assertions (SVA).

Should:

- Develop at least one example per app to work with VC Formal.
- Create a step-by-step video demonstrating the installation and configuration of VC Formal on a local PC at PSU.

May:

- Develop multiple examples per app.
- Create videos demonstrating the configuration and usage of some or all of the apps.
- Conduct a case study involving one or more RTL designs, running formal verification using more than two VC apps.
- Provide TCL templates for apps utilizing TCL.

Objectives and Deliverables

The specific objective of this project is to provide comprehensive tutorial documentation on using Synopsys VC Formal, focusing on the various apps within the tool. The aim is to enable undergraduate and graduate students at Portland State University to gain practical knowledge and proficiency in formal verification techniques using VC Formal.

At the end of our project, Dr. Song wanted to have a set of clear and accessible tutorials that guide students through the installation, configuration, and utilization of VC Formal. The tutorials would cover the prioritized apps, such as Formal Property Verification (FPV), Formal Coverage Analyzer (FCA), Formal Connectivity Checking (CC), and others, as well as specific topics like sequential equivalence checking and LTL model-checking.

While the project's main focus remained on Synopsys VC Formal, there was a change in the deliverables. Our sponsor requested additional tutorials using Cadence's formal verification tool/solution, JasperGold. Initially challenging, we took on the task and successfully learned and worked with JasperGold, delivering a tutorial on its usage as an added deliverable.

Overall, the project objectives remained consistent with a primary emphasis on providing tutorials for VC Formal. However, the expanded scope to include JasperGold tutorials allowed us to offer a more comprehensive learning experience for students, covering multiple formal verification tools.

Approach

Given the unfamiliarity of the team with formal verification and VC Formal, the project initially appeared to be a significant challenge. The specialized nature of the topic and the expertise required to operate these tools and create tutorials made the project seem daunting. However, the team embraced the challenge and approached it methodically, along with using their prior experience in following introductory tutorials to formulate better ones.

The first phase involved building a foundation of knowledge by researching formal verification, both at a non-technical level and a deeper technical level. Additionally, the team refreshed their understanding of the Linux environment, focusing on the installation and utilization of complex software tools. Valuable resources provided by Synopsys, including tutorials and documentation, were leveraged to gain a better understanding of VC Formal.

Throughout the project, the team maintained regular communication with their sponsor and academic advisor, Dr. Song. They sought clarification and guidance whenever questions arose or confusion arose during the process. Dr. Song's expertise and support were instrumental in overcoming challenges and addressing concerns.

In our attempt to maximize productivity, the team decided to divide into smaller groups with two groups of two, and the remaining student coordinating the work and providing help when needed. This approach allowed for parallel work on different tutorials while facilitating collaboration, mutual assistance in debugging, and issue resolution. Fortunately, splitting up into subgroups proved to speed up the production process.

Despite the initial apprehensions and the complexity of the subject matter, the team's systematic approach, thorough research, and effective communication with their sponsor and advisor allowed them to tackle the project successfully. The division into smaller groups further enhanced productivity and collaboration, resulting in significant progress and achievement in creating tutorials for formal verification using VC Formal.

Design

The big picture:

The design process involved extensive learning, experimentation, and iterative refinement. Our primary focus was to gain proficiency in utilizing the various VC Formal apps and create tutorials that effectively explain their usage and offer practical examples.

To achieve this, we dedicated significant time to studying each app individually, understanding its functionalities, and exploring different techniques. We conducted thorough testing to ensure that our understanding of the apps was solid and accurate. Through this process, we built a strong foundation of knowledge and hands-on experience with VC Formal.

The creation of tutorials was a crucial aspect of our design. We wanted each tutorial to be independent and self-contained, allowing for easy sharing with students in relevant classes. This approach ensures that students can access tutorials specific to the app they are working with, without being overwhelmed by unnecessary information from other tutorials.

Throughout the project, we constantly revisited and refined our tutorials. As we gained more experience and knowledge, we identified areas where improvements and clarifications were necessary. For instance, we discovered techniques like tracing the driving signal in the waveform view, which prompted us to revisit earlier tutorials and incorporate these valuable insights.

By investing significant effort into learning, experimentation, and tutorial creation, our design provides a solid foundation for students to understand and utilize VC Formal. The tutorials are designed to be informative, user-friendly, and independent, offering practical examples that enhance the learning experience and promote effective utilization of VC Formal's apps.

Starting with the fundamentals

As we embarked on our project, we followed a structured approach by creating three fundamental tutorials before diving deep into VC Formal:

Tutorial 1: Installing Mobaxterm

- In this tutorial, we focused on installing Mobaxterm, a software necessary to access the remote PSU labs where VC Formal is installed.
- We provided a step-by-step guide, ensuring users could successfully install Mobaxterm and familiarize themselves with its functionalities.
- The tutorial aimed to establish a solid foundation for users to access the remote lab environment seamlessly.

Tutorial 2: Installing and Running Cisco VPN Client

- This tutorial addressed the installation and configuration of the VPN Client "Cisco VPN" to enable users to access the remote PSU lab securely.
- We provided detailed instructions, guiding users through the installation process and necessary configurations to establish a secure connection.
- The tutorial aimed to ensure users could connect to the remote lab environment without any complications.

Tutorial 3: Installing Synopsys VC Formal on Linux System

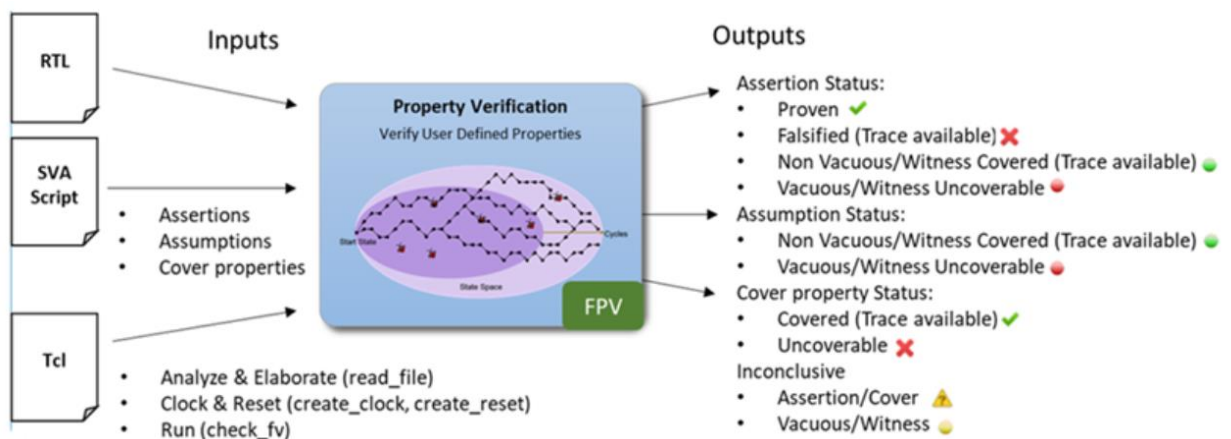
- In this tutorial, our focus was on installing Synopsys VC Formal on the Linux system used within Mobaxterm.
- We provided a comprehensive guide, walking users through the installation process and the setup of the environment for their first project.
- The tutorial aimed to equip users with the necessary tools and configurations to effectively work with VC Formal.

The tutorials laid the groundwork, ensuring users could access the remote lab environment, establish a secure connection, and have VC Formal installed and configured correctly. This step-by-step approach aimed to build users' confidence and prepare them for further exploration of VC Formal's functionalities and applications.

To provide users with a comprehensive understanding of VC Formal and its various applications, we developed tutorials for the VC Formal apps. Here is an overview of the VC Formal apps and a brief description of the corresponding tutorials:

1. Formal Property Verification (FPV) App:

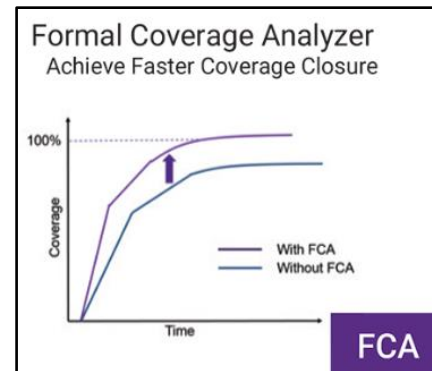
- Tutorial: In this tutorial, we dive into the FPV App and guide users on how to verify properties in their designs. We cover both user-defined and Assertion IP (AIP) properties, providing detailed examples and explanations.
- The FPV application is utilized for verifying a DUT by proving properties. These properties can be created by users or provided by commercial AIPs for interface protocol or function blocks. The app uses a range of powerful VC Formal engines to thoroughly prove or disprove a property. If an assertion fails, the FPV app will generate a counter-example to demonstrate a violating trace. However, there may be cases where it is impossible to definitively prove or disprove a property. In these cases, the app will provide a bounded proof result indicating that no falsification can be found up to a specific depth from the initial state. Proof results for a set of assertions mean that given the given constraints, it is impossible to falsify the properties.



This figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022 (Page 471)

2. Formal Coverage Analyzer (FCA) App:

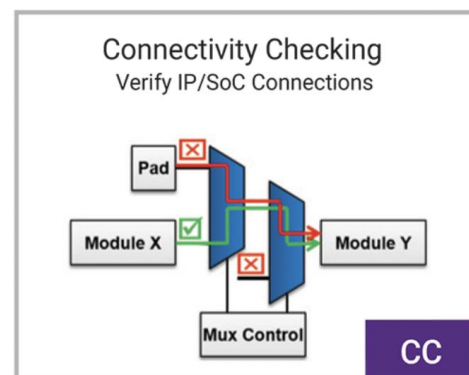
- Tutorial: Our tutorial focuses on the FCA App and its role in simulation-based verification coverage signoff. We demonstrate how to use FCA to qualify formal property verification with coverage signoff, ensuring comprehensive verification of the design.
- Code coverage closure is when a design reaches all of its target goals, and often is very difficult to achieve and time consuming, so coverage metrics are commonly used in simulation to measure progress and reachability analysis, such as providing information on what checks have passed and what has not. This provides proof on uncovered points in coverage goals that are indeed unreachable, allowing them to be removed from further analysis to save manual labor.



This FCA figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

3. Formal Connectivity Checking (CC) App:

- Tutorial: Here, we explore the CC App and provide step-by-step instructions on checking the structural and functional connectivity between source and destination. Users will learn how to ensure proper connectivity within their designs using this powerful app.
- The CC app in VC Formal is used as a connectivity verification tool to prove conditional wiring. CC app is a powerful debugging tool that includes automatic root-cause analysis of unconnected connectivity checks which saves significant debug time. It expedites SoC connectivity verification at the top level along with the connection between IP blocks.
- The CC application takes the design (RTL) and the connection specifications as inputs and verifies if the connection specifications are good or not. CC app is used to verify the correctness of a design's connectivity by analyzing the circuit's logic and ensuring that signals are properly connected between modules.



This FCA figure was taken from the CC Formal User Guide Version T-2022.06-SP2, December 2022

4. Automatically Extracted Properties (AEP) App:

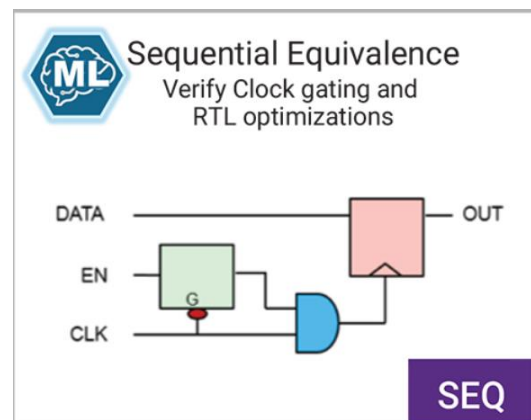
- Tutorial: In this tutorial, we explain the AEP App and its functionality in automatically extracting properties from the design. Users will learn how to verify these properties efficiently using the AEP App.
- The “AEP” app in VC Formal is used as an analysis tool for auto-checking a design for out-of-bound 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.



This AEP figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

5. Sequential Equivalence Checking (SEQ) App:

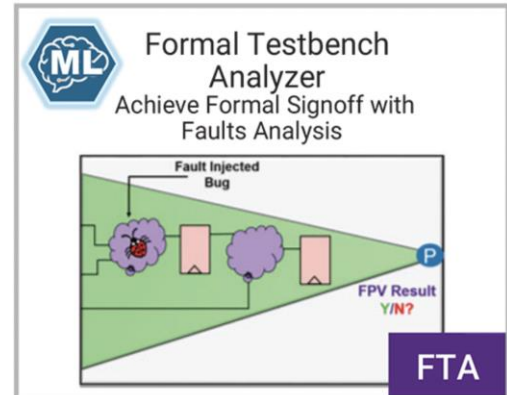
- Tutorial: Our tutorial on the SEQ App focuses on comparing two RTL designs to verify their functional equivalence. We provide detailed examples and discuss the significance of SEQ App in validating design modifications.
- The Sequential Equivalence Checking “SEQ” app is used to verify the equivalence between two sequential designs. It’s specifically used for sequential equivalence checking tasks. With this app users can verify the functional equivalence of sequential designs by comparing the behavior of two designs. It will perform deep analysis of the designs to ensure that their outputs are equivalent for all possible inputs. By using this app, you can effectively ensure that your design behaves the same and meets your verification goals.



This SEQ figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

6. Formal Testbench Analyzer (FTA) App:

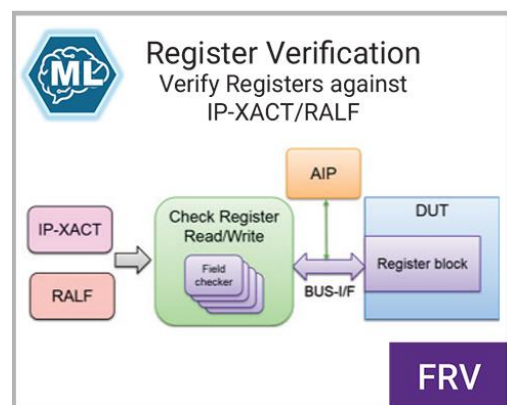
- Tutorial: Here, we guide users through the FTA App, which helps in assessing the quality of formal testbenches. The tutorial includes step-by-step instructions on using FTA and emphasizes the importance of fault detection analysis.
- The Synopsys VC Formal FTA (Formal Testbench Apps) is an innovative tool designed to enhance the efficiency and effectiveness of functional verification using formal methods. VC Formal FTA offers a comprehensive set of features and capabilities for creating advanced testbenches based on formal techniques. By leveraging its powerful automation capabilities and intelligent analysis, VC Formal FTA enables engineers to rapidly generate high-quality testbenches that thoroughly exercise their designs, ensuring robustness and identifying hard-to-find bugs. This application revolutionizes the traditional testbench creation process, significantly reducing the time and effort required for functional verification.



This FTA figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

7. Formal Register Verification (FRV) App:

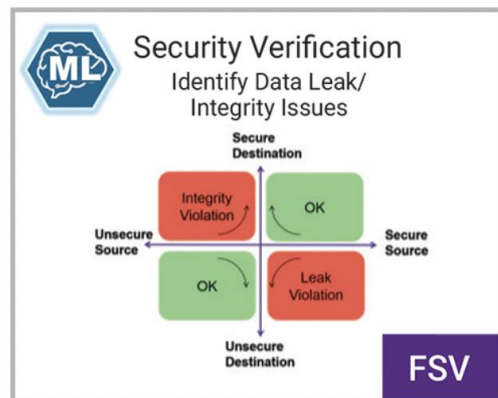
- Tutorial: Our tutorial for the FRV App showcases the creation and formal verification of checks for registers in the design. Users will gain insights into how FRV App can ensure the correctness of register implementations.
- Formal register verification is used to perform formal verification checks for registers in a design. It cuts down the time of performing directed simulation tests by formally verifying register behavior of a design configuration. Intended register behavior is provided by the user via IP-XACT (.xml file) or RALF (.ralf file) format. This register specification file will be a separate file that compliments your main design file(s) (SV or V file). It will be where you define all registers and/or reg-fields.



This FRV figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

8. Formal Security Verification (FSV) App:

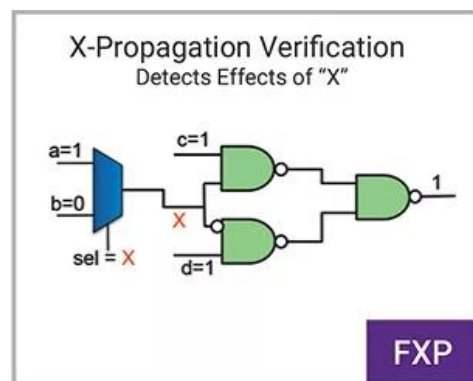
- Tutorial: This tutorial focuses on the FSV App, which is essential for preventing unintended data propagation between secure and non-secure areas. We explain how to use the FSV App effectively to enhance the security of the design.
- The “FSV” app in VC Formal is used as a verification application to ensure no illegal data propagations are in your design. VC Formal FSV app streamlines the verification process, ensuring the completeness and correctness of designs. By leveraging formal verification techniques, this application empowers designers to identify and resolve complex functional issues efficiently, minimizing the risk of bugs and improving overall design quality.



This FSV figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

9. X-Propagation Verification (FXP) App:

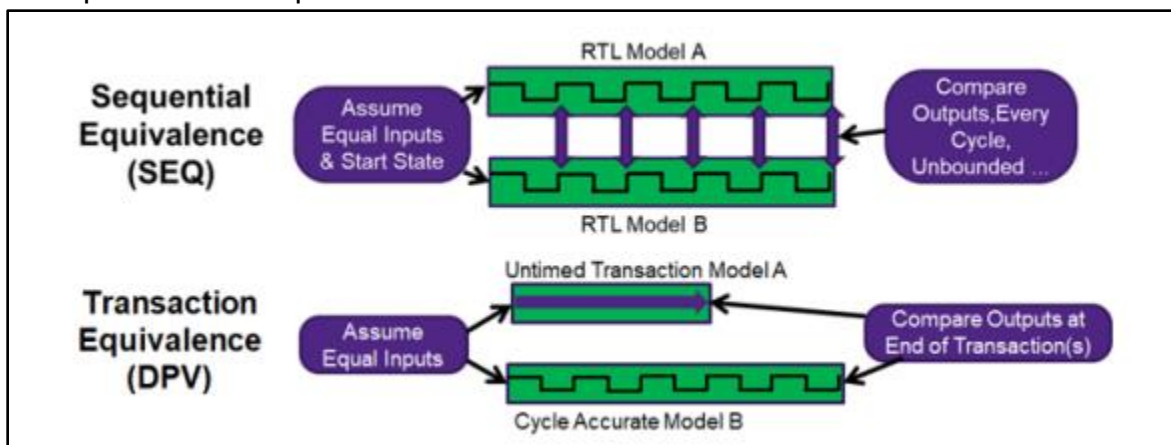
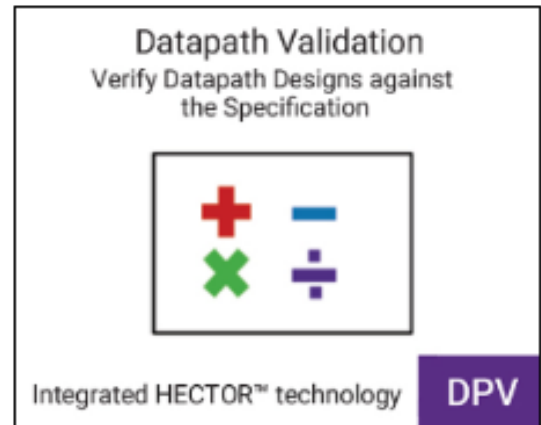
- Tutorial: Here, we delve into the FXP App and guide users on checking for the propagation of unknown signal values (X's) to dangerous points in the design. The tutorial includes practical examples and highlights the importance of X-propagation verification.
- The Formal X-propagation (FXP) app in VC 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.



This FXP figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

10. Data Path Validation (DPV) App:

- Tutorial: In this tutorial, we explore the DPV App, leveraging HECTOR technology to validate data transformation blocks between untimed C/C++ and RTL models. Users will learn how to effectively use DPV for data path verification.
- 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.
- 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.



This DPV figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

11. Functional Safety (FuSa) App:

- Tutorial: Our tutorial on the FuSa App provides insights into analyzing the controllability and observability of Z01X faults, enabling users to calculate FMEDA metrics for functional safety assessment.
- The Functional Safety Verification (FuSa) application is used as a safety tool to minimize risk caused by a malfunctioning system. A separate in-house tool that Synopsys uses, called Z01X, is a Verilog software fault injection simulator that

provides coverage results. VC Formal takes these results and determines whether the type of faults simulated and identified are “safe” or “dangerous”, via the FuSa application. Since the Z01X tool produces many types of faults, VC Formal serves as an extra step to help sift through and reduce the need for manual reviewing.

- The FuSa application performs the following types of analysis:

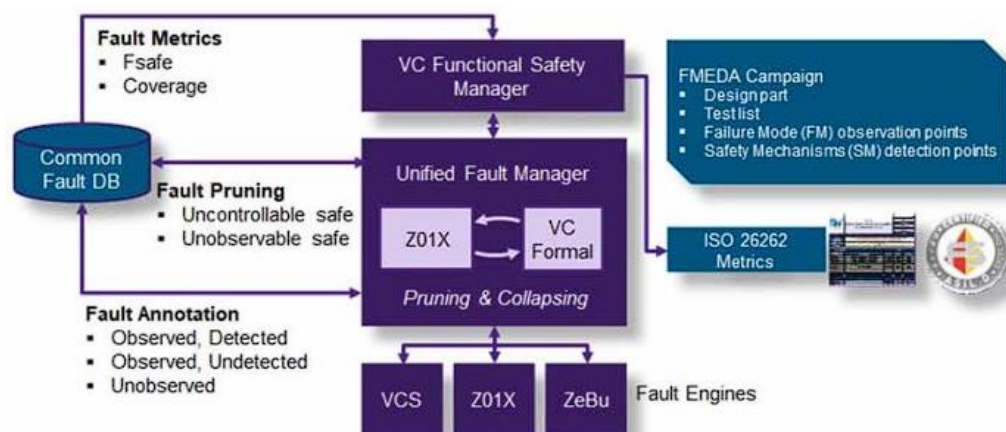
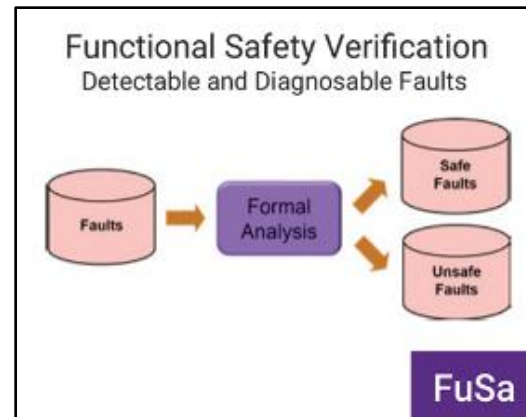
- ❖ Structural Analysis

- ❖ Controllability Analysis

- ❖ Observability Analysis

- ❖ Detection Analysis

- Structural Analysis is used to identify whether faults are in or not in the COI of observation/detection points. They are marked if they are not and those that are not are put through further analysis.
- Controllability Analysis is used to determine if a signal can transition from one established state to another at the location of the fault. Cover properties are set at fault locations and are marked non-controllable if unreachable or sent for further analysis if properties are covered/inconclusive.
- Observability and Detection Analysis is used to observe whether or not a fault is blocking a signal from propagating through to an observation/detection point. Faults that do not propagate are considered safe. Faults that propagate to observation/detection points are marked accessed further.



This FuSa figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

Through these tutorials, we aimed to cover each VC Formal app comprehensively, providing users with practical guidance and examples to understand their functionalities and leverage them effectively for formal verification.

12. Cadence JasperGold Superlint - Ensuring Design Integrity through Static Linting

- In this tutorial, we provide a comprehensive guide on using Cadence JasperGold Superlint, a powerful static linting tool for digital design verification. Learn how to identify potential coding errors, design rule violations, and improve design integrity. Our step-by-step instructions cover setting up the tool, configuring linting rules, running the process, and interpreting reports. Enhance the quality, reliability, and maintainability of your digital designs with Cadence JasperGold Superlint.

Test plan

The effectiveness of the tutorials was tested in a real-world academic setting, specifically in the ECE682/582 Verification of Hardware and Software Systems course at PSU for Winter 2023. Since this course was taught by Dr. Song, our faculty advisor and industry sponsor, tutorials could be introduced to students with ease, as with monitoring the validity of each tutorial.

Approximately 80 graduate students worked on five of the tutorials across two projects. The first project involved using the VC Formal SEQ app to design and verify the equivalence of two digital circuits. The second project involved using four VC Formal apps (AEP, FXP, FPV, and FCA) to design a specific circuit, with errors intentionally introduced, analyzed, and corrected using the formal apps. These testing scenarios provided the opportunity to receive valuable feedback and insights from the students, which were used to further enhance the quality and effectiveness of the tutorials. This iterative process of creation, testing, and refinement ensured that the tutorials were not only informative but also aligned with the needs of the students.

Not every document was created by the same individuals on the team - for the remainder of the applications, team members who were not involved in the process of certain apps had to act as a test subject and go through the tutorials to see if there are any room for improvement or if there were any setbacks.

Results

- **Student Feedback and Familiarization:**
The feedback received from students provided valuable insights into the effectiveness of the tutorials. Initially, students found VC Formal to be complex and intimidating. However, as they gradually worked through the tutorials, they became more familiar with the VC Formal workflow and graphical user interface (GUI). This familiarity enabled them to successfully run formal analyses on their designs.
- **Benefits of Automatic Apps:**
The automatic apps, such as AEP and FXP, proved to be particularly beneficial for students. These apps did not require students to write assertions, which lowered the entry barrier and allowed them to gain hands-on experience with VC Formal quickly. Students who started with the automated apps reported an easier transition to using the FPV app, which involved writing assertions. The sequential progression from automated to more involved analysis helped students become comfortable with the tool and build confidence in utilizing an industry-grade formal verification tool like VC Formal.
- **Successful Guidance in Running Formal Analyses:**
Overall, the tutorials effectively guided all students in running formal analyses of their SystemVerilog designs. The step-by-step approach and hands-on exercises equipped students with the necessary skills to utilize VC Formal for verifying their designs. The positive outcome underscores the effectiveness of the tutorials and highlights the potential benefits of incorporating industry-grade formal verification tools like VC Formal into academic curricula.

Challenges

Throughout the course of our project, we encountered several challenges that impacted our progress. Here is a list of challenges we faced:

- Connection Issues:
We encountered various challenges related to the connection setup using MobaXterm. The issues primarily arose from incorrect login credentials and misconfigured VPN settings. These connectivity hurdles impeded our ability to establish a seamless connection with the remote PSU labs where VC Formal was installed. We had to invest time in troubleshooting and rectifying these issues to ensure uninterrupted access.
- Platform Compatibility:
Another obstacle we faced was the disparity between using Mac and Windows operating systems. These variations introduced compatibility issues and discrepancies in the tools and software we were utilizing. We had to navigate through platform-specific challenges, adapt our workflows, and find workarounds to ensure smooth collaboration and compatibility across different platforms.
- Access and License Renewal:
Our project was dependent on accessing VC Formal through the PSU CAT team. However, we encountered delays in obtaining the necessary access to install VC Formal. Additionally, there were instances where the VC Formal license expired, leading to a temporary halt in our progress. We had to coordinate with the PSU CAT team to resolve these issues and ensure timely access and license renewal to continue our work.
- Lack of Formal Verification Examples:
One of the significant challenges we encountered was the scarcity of readily available formal verification examples suitable for our project. We struggled to find meaningful SystemVerilog examples that aligned with the specific formal analysis tasks we aimed to demonstrate. As a result, we had to dedicate significant effort and time to create our own examples that effectively showcased the functionalities of VC Formal, adding an additional layer of complexity to the project.
- DPV App Limitations:

The DPV app presented unique challenges due to the simplicity of our design. Since the app is primarily designed to verify complex datapaths, our relatively straightforward design did not cover a wide range of test cases provided by the DPV app. The VC Formal user guide emphasized server-based verification analysis, which introduced a level of complexity that surpassed the scope of our tutorials. We had to carefully adapt the DPV app to suit our project's objectives and provide meaningful insights to users without overwhelming them with unnecessary complexities.

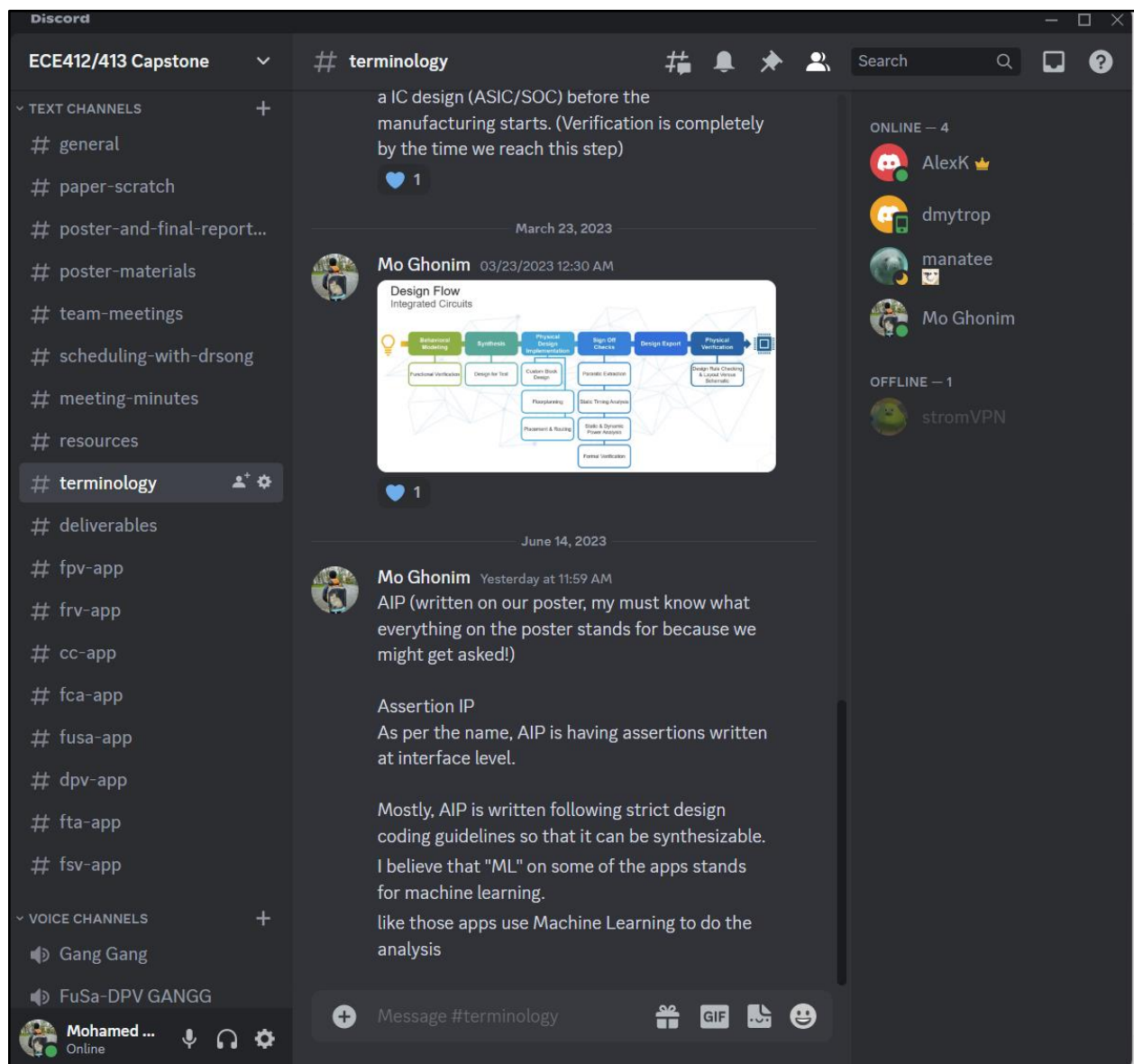
- Complex User Guide:
Navigating the VC Formal user guide proved to be a time-consuming and intricate task. The extensive documentation often lacked clarity and required significant effort to locate the necessary information. We had to invest additional time in thoroughly understanding the user guide and leveraging its insights to effectively utilize VC Formal in our tutorials. The complexity of the user guide added an extra layer of challenge in comprehending and implementing the tool's functionalities.
- Varied GUI Experience:
Each VC Formal app came with its own graphical user interface (GUI), which meant we had to adapt to different interface designs and functionalities. Transitioning between the different GUIs required us to familiarize ourselves with their unique features and workflows, which required additional effort and time investment. We had to ensure that our tutorials provided clear instructions and guidance to users for navigating and utilizing the specific GUI associated with each app.
- TCL File Structure:
The structure of the TCL (Tool Command Language) file varied across the VC Formal apps, adding complexity to our project. Since most of us were new to TCL, we had to invest time in learning the language and understanding its syntax and conventions. This allowed us to effectively utilize TCL for scripting and automation in our tutorials. The diverse TCL file structures across the apps demanded a comprehensive understanding of TCL to ensure smooth execution of the tutorials.

- Learning SystemVerilog Assertions (SVA):
Understanding and employing SystemVerilog Assertions (SVA) for the FPV app posed significant challenges due to the lack of accessible examples and educational materials. We had to navigate through unfamiliar territory, investing considerable time and effort to learn SVA and its application in formal analysis. Overcoming these hurdles required extensive research, problem-solving skills, and collaboration to incorporate SVA teachings into our tutorial as an appendix.

By acknowledging and addressing these challenges, we were able to navigate through them, finding solutions and adapting our approach to ensure the successful completion of our project.

Collaboration Tools

- **Discord:**
We utilized Discord as our primary communication platform throughout the capstone project. We organized the server into multiple channels to facilitate efficient collaboration. Administrative channels were used for scheduling meetings, discussing tasks, and assigning responsibilities, while technical channels were dedicated to each VC Formal app, allowing focused discussions on specific topics.



This is a screenshot of our Discord server

- GitHub:

We leveraged GitHub as a version control and file-sharing platform. It served as a central repository for our finalized tutorials, specifications, and reports. By using GitHub, we could easily track changes, manage different versions of our files, and collaborate seamlessly on shared documents.

<https://github.com/VCFormal>

<https://github.com/VCFormal/VC-Formal-Documentation-and-Examples>

The screenshot displays the GitHub repository interface for 'VCFormal / VC-Formal-Documentation-and-Examples'. The repository is public and has 14 commits. The file list on the left includes folders for various applications (e.g., AEP App, CC App, DPV App, FCA App, FPV App, FRV App, FSV App, FTA App, FXP App) and tutorials (e.g., AEP_Tutorial, FCA_Tutorial, FXP_Tutorial, SEQ_Tutorial), as well as a LICENSE file and a README.md file. The right sidebar provides additional information: 'Project deliverables' (Readme, GPL-3.0 license, Activity, 2 stars, 0 watching, 0 forks), 'Releases' (No releases published), 'Packages' (No packages published), and 'Contributors' (wcelina, anordstrom21, Ghonimo).

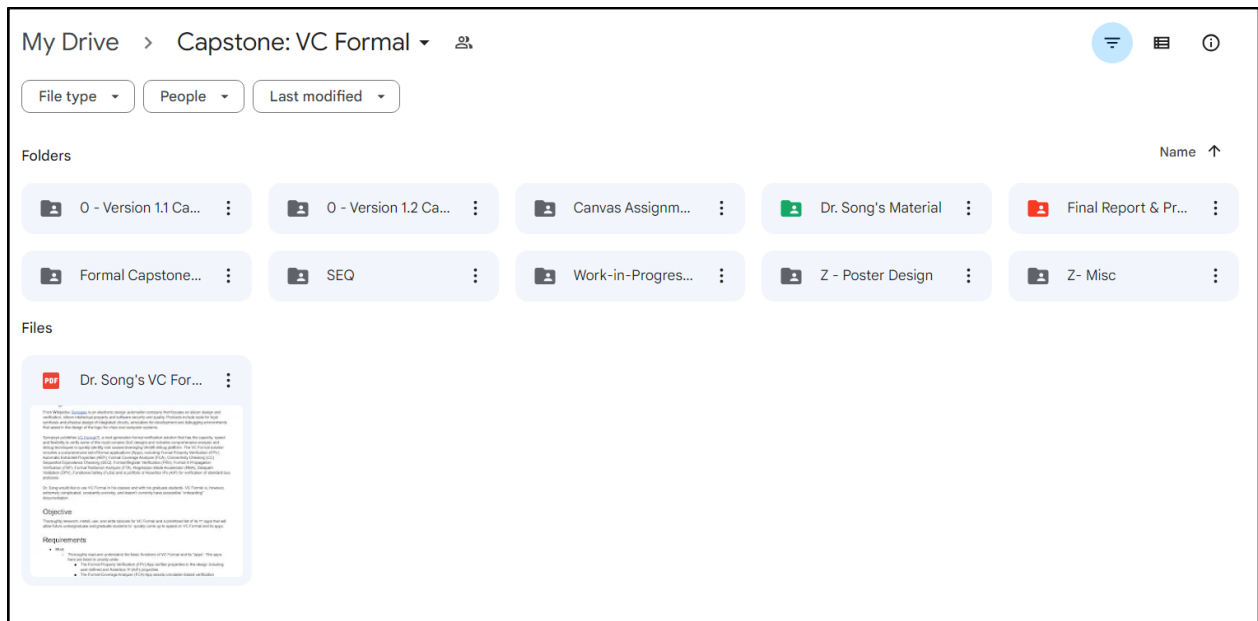
This is a screenshot of our Github repository

- Trello:

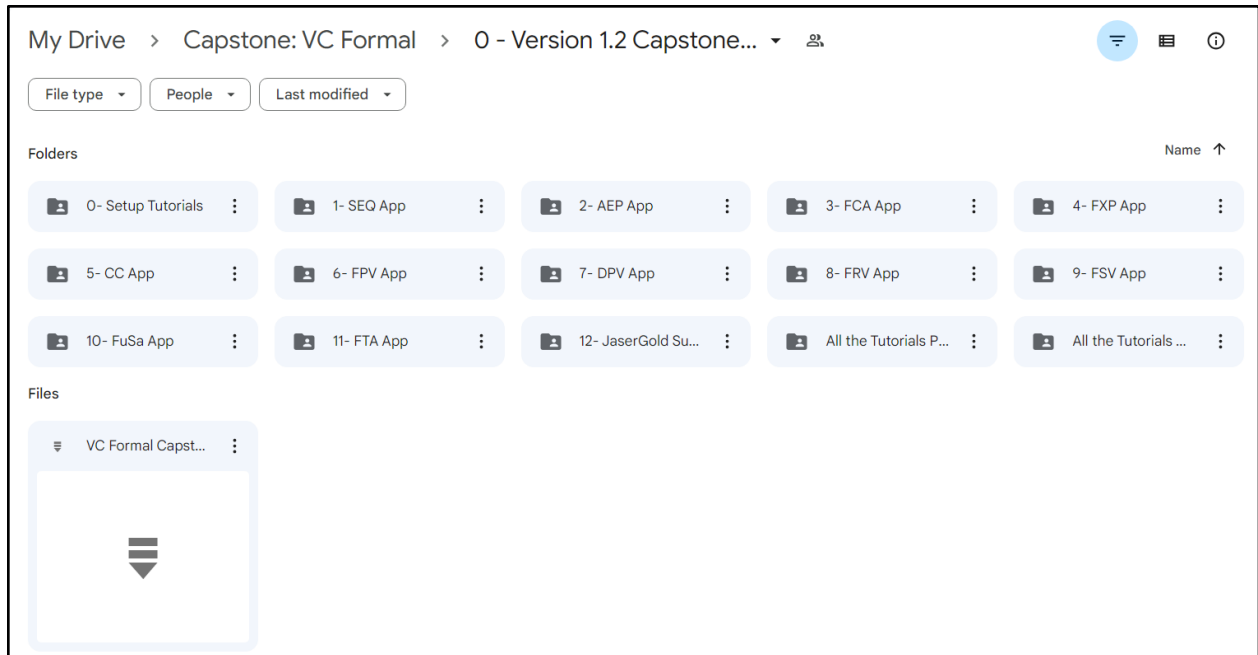
We adopted Trello as our project management tool to keep track of tasks, deadlines, and progress. It allowed us to create boards, lists, and cards to visualize our workflow and assign tasks to team members. Trello proved to be an effective tool for organizing and prioritizing our activities, ensuring that everyone was aware of their responsibilities and the overall project timeline.

- Google Drive:

We utilized Google Drive as a cloud storage and collaboration platform. It provided us with a convenient way to upload, store, and organize our project files. Additionally, we took advantage of the versioning feature in Google Drive to track document revisions and ensure that the latest versions were readily accessible. The chat and comments feature in Google Docs enabled real-time discussions and feedback exchange among team members, promoting effective collaboration and document refinement.



This is a screenshot of our Google Drive folder.



By leveraging these collaboration tools, we established a streamlined workflow, enhanced communication, and ensured efficient coordination among team members. These tools played a vital role in facilitating effective project management, version control, and document sharing, ultimately contributing to the overall success of our capstone project.

Summary

Our capstone project aimed to integrate industry-grade formal verification tools, specifically Synopsys VC Formal, into the academic curriculum at Portland State University (PSU). Throughout the project, we focused on creating comprehensive tutorials for various VC Formal apps, with the objective of enhancing students' understanding and practical experience in formal verification.

We started by developing a structured approach, beginning with tutorials on basic setup and installation, followed by tutorials on each VC Formal app. These tutorials covered essential topics such as formal property verification, formal coverage analysis, fault-tolerant analysis, and datapath validation. Each tutorial provided step-by-step instructions, relevant examples, and practical insights to guide students through the formal verification process.

Testing the effectiveness of our tutorials, we incorporated them into the ECE682/582 Verification of Hardware and Software Systems course at PSU. Approximately 80 students participated in projects that involved designing and verifying digital circuits using VC Formal. The feedback received from students proved valuable in assessing the tutorials' impact and making necessary refinements.

Throughout the project, we encountered several challenges, including connectivity issues, availability of formal verification examples, and the learning curve associated with the tools. However, we tackled these challenges by leveraging our collaborative efforts, research, and problem-solving skills. We refined the tutorials, addressing the complexities and ensuring they were accessible to students at various skill levels.

The success of our capstone project was evident in the positive student feedback and the effective use of the tutorials in coursework. Students initially found VC Formal to be complex and intimidating, but as they progressed through the tutorials, they gained confidence and familiarity with the tools. Our tutorials provided them with valuable insights, hands-on experience, and a solid foundation in formal verification.

The project not only enriched the educational experience for students but also paved the way for future initiatives in integrating industry-grade formal verification tools into academic curricula. Our lessons learned, best practices, and recommendations will serve as a valuable resource for future projects in this domain.

Overall, our capstone project showcased the potential of industry-academia collaboration and the impact of incorporating formal verification tools into the curriculum. It demonstrated the value of hands-on learning, real-world applications, and the bridging of academia and industry. Our collective efforts, dedication, and collaboration have

contributed to advancing the field of formal verification education and preparing students for the challenges and opportunities in the industry.

EXPLORING FORMAL VERIFICATION USING VC FORMAL

Team: Mohamed Ghonim, Ahliah Nordstrom, Dmytro Prystupa, Alexander Kim, Celina Wong
 Industry Sponsor/Academic Advisor: Dr. Xiaoyu Song | Supported by Dr. Jin Zhang (Synopsys)

Sequential Equivalence

Auto Checks

X-Propagation Verification

Property Verification

Connectivity Checking

Formal Testbench

Register Verification

Formal Coverage Analyzer

Security Verification

Data Path Validation

Functional Safety Verification

Background

Formal Verification:
The ultimate assurance of flawless digital systems, leaving no room for errors. Revolutionizing validation in critical domains like avionics and biology, it unlocks unparalleled reliability and safety. It unleashes the power of rigorous analysis, guaranteeing perfection and accuracy.

Synopsys VC Formal:
A powerful tool for rigorous verification of digital designs. It offers a suite of applications that provide comprehensive verification capabilities. With VC Formal, engineers can ensure the reliability and integrity of critical systems in various domains.

Due to how specialized, and possibly complex Formal Verification is, industry verification tools are rarely used in academia.

Approach

The tutorials we made:

Installation and Getting Started:

- Installing Mobaxterm
- Installing and Running Cisco VPN Client
- Installing Synopsys VC Formal on Linux System

Using the VC Formal Apps:

Sequential Equivalence Checking (SEQ): Comparing two RTL designs and verifying functional equivalence.

Automatically Extracted Properties (AEP): Automating property extraction from designs for efficient verification.

X-Propagation Verification (FXP): Checking for propagation of unknown signals (X's) to dangerous points in the design.

Formal Property Verification (FPV): Writing System Verilog Assertion (SVA) properties to verify RTL designs.

Formal Connectivity Checking (CC): Verifying structural and functional interconnection in a design.

Formal Testbench Analyzer (FTA): Checking the formal testbench quality and measuring fault detection capability.

Formal Register Verification (FRV): Create and formally verify checks on IP-XACT or RALF format for registers in designs.

Formal Coverage Analyzer (FCA): Qualifying formal property verification with coverage sign off using FCA.

Formal Security Verification (FSV): Prevent unexpected data propagation between secure and non-secure areas.

Data Path Validation (DPV): Verify data transformation blocks between untimed C/C++ and RTL models.

Functional Safety (FuSa): Analyzing the controllability and observability of faults.

Testing

- Real-world academic testing conducted in the ECE682/582 Verification of Hardware and Software Systems course at PSU.
- Approximately 80 students participated, working on five tutorials across two projects.
- Projects involved using VC Formal SEQ app for circuit equivalence verification and four VC Formal apps (AEP, FXP, FPV, and FCA) for error analysis and correction.

Objective

- Research, install, and learn Synopsys VC Formal.
- Develop step-by-step tutorials for using VC Formal.
- Create and test simple examples to demonstrate formal verification with VC Formal.
- Integrate VC Formal into PSU classes to provide students with hands-on experience.
- Introduce students to industry-grade formal verification tools for real-world applications.

Results

- Student feedback provided valuable insights into tutorial effectiveness.
- Initial complexity and intimidation of VC Formal gradually overcome as students progressed through tutorials.
- Automatic apps (AEP, FXP) highly beneficial for students without assertion writing requirements.
- Recommended tutorial order and automated analysis (AEP) improved navigation and understanding of FPV app.
- Overall success in guiding students to run formal analysis on SystemVerilog designs.
- Demonstrated effectiveness of tutorials and potential benefits of incorporating industry-grade formal verification tools into academic curricula.

Conclusion

- The integration of industry-grade formal verification tools into academic curricula has been successfully demonstrated in the capstone project.
- Positive student feedback and the effective use of the tutorials in coursework highlight the potential of such initiatives.
- The project contributes valuable insights to the teaching approach for formal verification tools and strengthens academia-industry collaboration.
- The project sets the stage for further advancements in formal verification education and prepares students for industry demands.

Department of Electrical and Computer Engineering

Maseeh College of Engineering and Computer Science
PORTLAND STATE UNIVERSITY

This is our final presentation poster.

Appendices

Appendix 1: Formal Verification Article

Bridging the Gap between Industry and Academia in Formal Verification: A Capstone Project at Portland State University

Abstract - This paper presents a capstone project at Portland State University (PSU) aimed at introducing students to industry-standard formal verification tools, with a primary focus on Synopsys VC Formal and a brief exposure to Cadence's JasperGold. The project involved the creation of 12 practical tutorials on various formal verification apps and three additional tutorials on tool installation, VPN setup for remote lab access, and Linux interface familiarization. These tutorials were successfully integrated into the PSU ECE 582/682 Verification of Hardware and Software Systems course, benefiting approximately 80 students. This paper documents the project's journey, including the challenges encountered and their solutions, the design and testing process, and advice for future similar initiatives. The project underscores the importance of bridging the gap between academia and industry, particularly in the area of formal verification.

I. Introduction

The rapidly evolving landscape of digital design and verification necessitates a strong foundation in formal verification techniques for emerging engineers. [3] Formal verification, a rigorous method of ensuring the correctness of digital systems, is a cornerstone of modern design processes. However, a noticeable disparity exists between the industry's reliance on these techniques and their representation in academic curricula. This gap, particularly evident in the limited exposure students receive to industry-grade formal verification tools, presents a challenge for academia and industry alike.[1]

In an effort to address this disparity, a unique capstone project was undertaken at Portland State University (PSU). Unlike typical capstone projects, this initiative focused on the exploration and utilization of industry-standard formal verification tools, primarily Synopsys VC Formal [1], and to a lesser extent, Cadence's JasperGold. The project's primary objective was to create a suite of practical tutorials that could be integrated into the academic curriculum, providing students with hands-on experience with these advanced tools.

II. Background

Formal verification is a rigorous method used in hardware and software design to verify the correctness of systems against a set of specifications. It employs mathematical techniques to prove or disprove the correctness of intended algorithms or protocols, ensuring the reliability of digital systems and eliminating the need for exhaustive testing.

Several industry-grade formal verification tools are available, each offering unique capabilities. These tools, including Synopsys VC Formal and Cadence's JasperGold, are designed to handle complex designs and large codebases typical in industrial settings. Synopsys VC Formal, in particular, is a comprehensive tool used extensively in the industry, providing a suite of applications for various formal verification tasks.

In academia, formal verification is often taught using academic tools such as SAT solvers and NuSMV, a symbolic model checker developed at Carnegie Mellon University. These tools provide students with a fundamental understanding of formal verification

techniques, including Computation Tree Logic (CTL) model checking and Linear Temporal Logic (LTL).

However, there is a noticeable gap between industry practice and academic instruction. Industry-level formal verification tools have been rarely used in academic settings, typically reserved for research purposes rather than educational ones. This disparity motivated our project at Portland State University, recognizing the need to incorporate industry-grade formal verification tools into our curriculum and provide students with practical training that aligns more closely with industry expectations.

III. The Capstone Project

The primary aim of our capstone project was to develop comprehensive and user-friendly tutorials on the usage of the VC Formal software application and its various verification methods. The overarching objective was to facilitate a deeper understanding and effective utilization of these industry-grade tools among students. The creation of these tutorials was a meticulous process, involving extensive research and self-study. The team delved into open-source materials available online and sifted through hundreds of pages of official documentation. This process required a keen eye for discerning the most relevant and useful information to help users build a foundational understanding of each application.

Each tutorial was designed to be comprehensive and practical. They included detailed explanations of how to interact with the tool's GUI, the specific use cases of the application, and a worked example. These examples were carefully crafted to demonstrate the workflow of formal verification, including the process of debugging and fixing errors, providing students with a hands-on experience of using these tools.

The journey of creating these tutorials was not without its challenges. Formal verification is a highly specialized area in digital electronics, and resources, especially worked out examples, are scarce. The team had to navigate this landscape, familiarizing themselves with the tools and their workflows. The creation of examples that effectively demonstrated the functionality of each app was a significant

challenge, particularly for complex apps like the DPV in VC Formal.

Debugging errors presented another hurdle due to the lack of readily available information about industry-grade formal verification tools. To overcome this, the team reached out to the Synopsys Formal team multiple times for assistance in resolving issues and understanding the different uses of each formal tool. Another challenge was to present the tutorials in a coherent and easy-to-follow manner, especially when introducing new terminologies to students who may not have encountered them before.

IV. Testing

The effectiveness of the tutorials was tested in a real-world academic setting, specifically in the ECE682/582 Verification of Hardware and Software Systems course at PSU. Approximately 80 students worked on five of the tutorials across two projects. The first project involved using the VC Formal SEQ app to design and verify the equivalence of two digital circuits. The second project involved using four VC Formal apps (AEP, FXP, FPV, and FCA) to design a specific circuit, with errors intentionally introduced, analyzed, and corrected using the formal apps. These testing scenarios provided valuable feedback and insights, which were used to further enhance the quality and effectiveness of the tutorials. This iterative process of creation, testing, and refinement ensured that the tutorials were not only informative but also aligned with the needs of the students.

V. Results

The feedback received from students provided valuable insights into the effectiveness of the tutorials. Initially, many students found VC Formal to be complex and intimidating. However, as they gradually worked through the tutorials, they became familiar with the VC Formal workflow and GUI, enabling them to successfully run formal analysis on their designs.

The automatic apps, such as the AEP and FXP, were particularly beneficial for students as they did not require writing assertions. Some students reported

difficulties when they initially tried to run the analysis on the FPV app due to the requirement of writing assertions. However, those who followed the recommended order of the tutorials and performed automated analysis such as AEP first found the FPV app easier to navigate. Running an automatic analysis first provided students with valuable insights into their design and helped them become comfortable with using an industry-grade tool like VC Formal.

Overall, the tutorials were successful in guiding all the students to run formal analysis on their SystemVerilog designs. This positive outcome underscores the effectiveness of the tutorials and the potential benefits of incorporating industry-grade formal verification tools into academic curricula.

VI. Discussion and future work

Looking ahead, there are several directions for teaching formal verification at PSU. One of the key lessons from our project is the effectiveness of starting with the GUI when teaching formal apps. While most of the formal apps can be run from and used exclusively in the terminal, starting with the GUI is often more intuitive for students and easier to follow.

We also recommend starting with automated apps like VC Formal's AEP and FXP apps, and JasperGold's Superlint app. These apps have a lower overhead to get started, and once students gain some momentum, introducing more involved apps like FPV would likely be more successful.

Another important finding from our project is the effectiveness of using small examples in our tutorials. These examples were easier for students to follow and allowed them to grasp the concepts more effectively.

For future similar projects, our advice is to keep these lessons in mind. Starting with the GUI, focusing on automated apps initially, and using small, manageable examples can significantly enhance the learning experience for students. Furthermore, continuous feedback from students can provide valuable insights for refining and improving the tutorials.

VII. Conclusion

In conclusion, the capstone project at Portland State University has demonstrated a successful model of integrating industry-grade formal verification tools into academic curricula. The project has not only enhanced the learning experience for students but also provided a blueprint for future initiatives in this direction.

The project's success, as evidenced by the positive student feedback and the effective use of the tutorials in coursework, underscores the potential of such initiatives in bridging the academia-industry gap. The insights gained from the project, particularly regarding the teaching approach for formal verification tools, are valuable contributions to the pedagogical strategies in this field.

Looking ahead, the project sets the stage for continued efforts in this direction, with the aim of further enriching the educational experience for students and preparing them for the demands of the industry. The project also serves as a testament to the potential of industry-academia collaboration in advancing formal verification education.

Acknowledgements

We would like to express our sincere gratitude to Synopsys for their invaluable support to Portland State University and our capstone team. Their generosity in providing us access to VC Formal and its documentation, as well as their assistance through support cases when we encountered difficulties, was instrumental to the success of our project.

We also acknowledge Cadence for their support. Their provision of access to JasperGold and its documentation.

These collaborations have not only enriched our project but also underscored the potential of industry-academia partnerships in advancing education and research.

References

- [1] M. Khazeev, M. Mazzara, H. Aslam, and D. de Carvalho, “Towards a broader acceptance of formal Verification Tools,” *Advances in Intelligent Systems and Computing*, pp. 188–200, 2020. doi:10.1007/978-3-030-40271-6_20

- [2] “VC formal: Formal verification solution: Synopsys verification,” *Formal Verification Solution* | Synopsys Verification, <https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html> (accessed Jun. 2, 2023).

- [3] M. Massoumi and A. Sagahyroon, “ASIC verification: Integrating formal verification with HDL-based courses,” *Computer Applications in Engineering Education*, 2010. doi:10.1002/cae.2025

Appendix 2: VC Formal Tutorials

Through our findings, the order in which these tutorials are recommended to be followed:

0. Getting Started with VC Formal:
 - [Part 1: Setting Up Cisco VPN Connection](#)
 - [Part 2: Setting Up MobaXterm](#)
 - [Part 3: Installing and Starting VC Formal](#)
1. [Sequential Equivalence \(SEQ\)](#)
2. [Automatic Extracted Properties \(AEP\)](#)
3. [Formal Coverage Analyzer \(FCA\)](#)
4. [X-Propagation Verification \(FXP\)](#)
5. [Connectivity Checking \(CC\)](#)
6. [Formal Property Verification \(FPV\)](#)
7. [Datapath Validation \(DPV\)](#)
8. [Formal Register Verification \(FRV\)](#)
9. [Security Verification \(FSV\)](#)
10. [Functional Safety \(FuSa\)](#)
11. [Formal Testbench Analyzer \(FTA\)](#)



Portland State
UNIVERSITY

DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

SYNOPSYS VC FORMAL TUTORIALS

ECE 413 CAPSTONE TEAM#7

MOHAMED GHONIM
AHLIAH NORDSTORM
DMYTRO PRYSTUPA
ALEXANDER KIM
CELINA WONG

Synopsys® VC Formal Tutorial

Getting Started: Part 1

**Downloading, Installing, and Making a VPN connection
using the Cisco VPN Client**



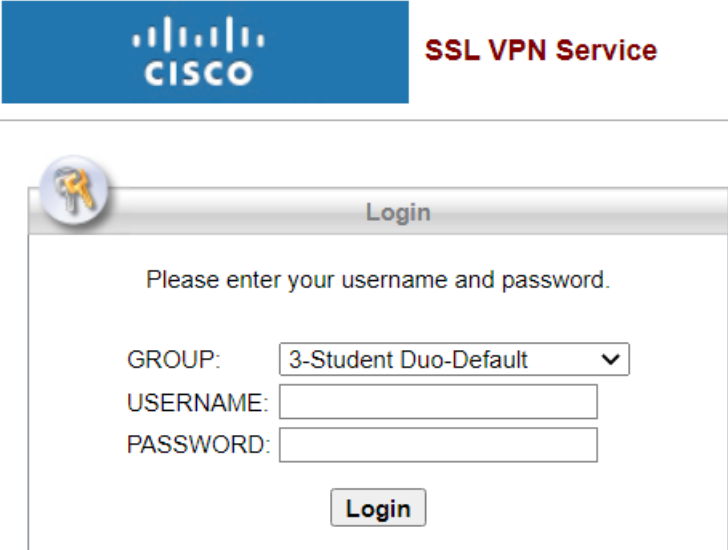
Version 1.1 | 02-June-2023

Portland State University

©2023

To access Synopsys VC Formal via the PSU remote lab, we need to use a VPN connection to the remote lab, we recommend using the cisco VPN client, which one can get using this link:

<https://vpn.pdx.edu/>



The image shows a Cisco SSL VPN Service login window. At the top, there is a blue header with the Cisco logo and the text "SSL VPN Service". Below the header, the window has a title bar with a key icon and the word "Login". The main content area contains the instruction "Please enter your username and password." followed by three input fields: "GROUP:" with a dropdown menu showing "3-Student Duo-Default", "USERNAME:" with a text box, and "PASSWORD:" with a text box. A "Login" button is located at the bottom center of the window.

Figure 1: Cisco VPN client login.

From the group drop-down list, choose 3-Student Duo-Default, then enter your PSU Odin username and password and click on Login. You might then receive an automated phone call, or a text message to confirm that you're indeed the one accessing the VPN.

Please note: If you click on Login and nothing happens, this probably means that you haven't activated the DUO security feature in your PSU Odin account, in this case, you'll need to do that first before continuing with downloading and using the cisco VPN client.

The next step would be to download and install the cisco VPN client.

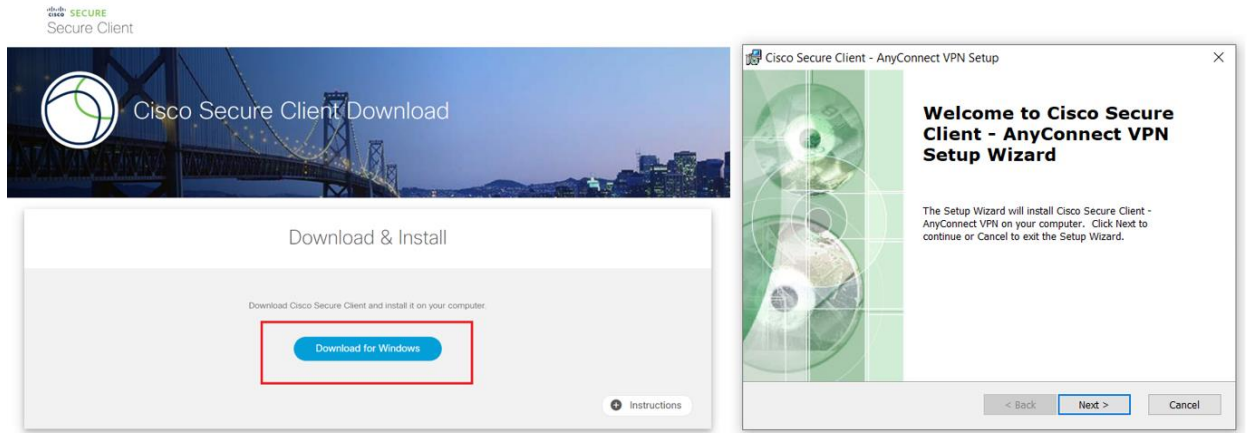


Figure 2: Cisco Secure Client setup windows.

Once the Cisco Secure Client is installed, you'll choose or enter "**vpn.pdx.edu**" and click on Connect as shown below.

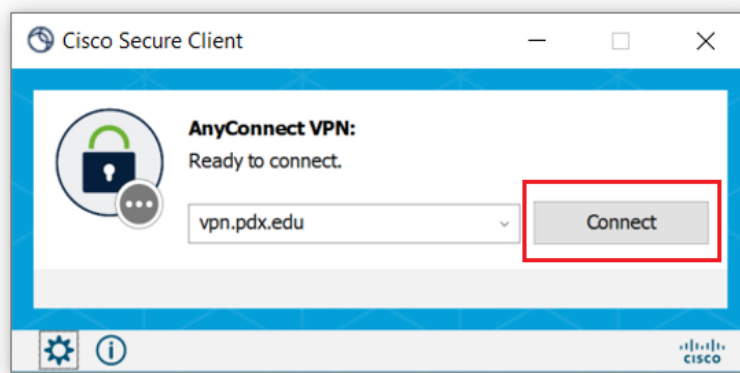


Figure 3: Cisco Secure Client connection window.

You will then be asked to enter your PSU Odin Username and password. Please do so and click "**Ok**". You might again receive an automated phone call or text message to confirm that you are the one making the connection.

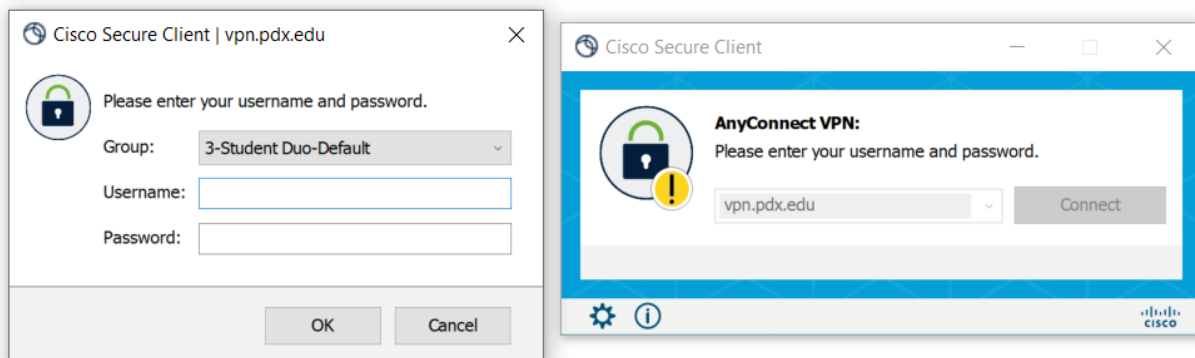


Figure 4: Cisco Secure Client login and connection windows.

You should now be connected to the **vpn.pdx.edu** server and ready to use MobaXterm to access Synopsys VC Formal.

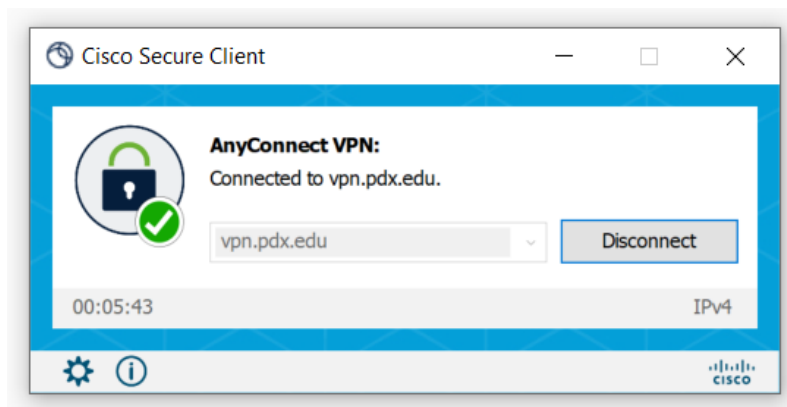


Figure 5: Cisco Secure Client window with a successful connection.

Synopsys® VC Formal Tutorial

Getting Started: Part 2

Downloading, Installing, and Setting up MobaXterm

Version 1.1 | 2-June-2023



Portland State University

©2023

We need to use a Linux client to access the Linux environment in which we run VC Formal, in this tutorial, we will download, install, and set up the MobaXterm client.

You can google “MobaXterm” to download MobaXterm, or follow the link:

<https://mobaxterm.mobatek.net/>

and click on the “Download” tab.

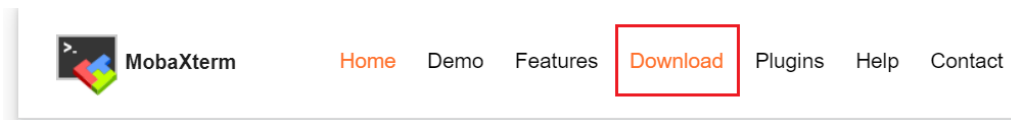


Figure 1: Download tab on MobaXterm website.

For our uses, the “Free Home Edition” is sufficient. Click “Download Now” to download and install MobaXterm. We recommend the “Installer” edition, but the “Portable Edition” should work too.

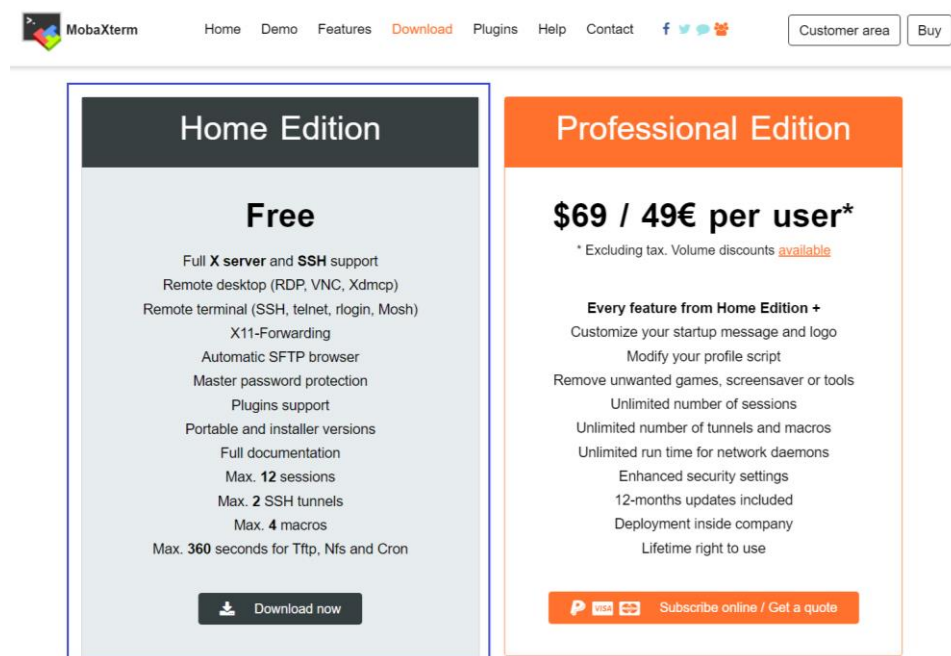


Figure 2: MobaXterm versions offered.

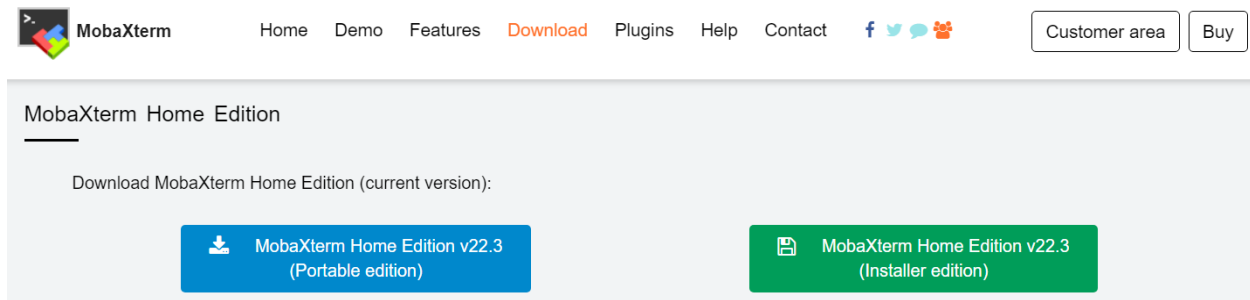


Figure 3: Current version of MobaXterm Home Edition.

Follow the prompts to install MobaXterm on your computer.

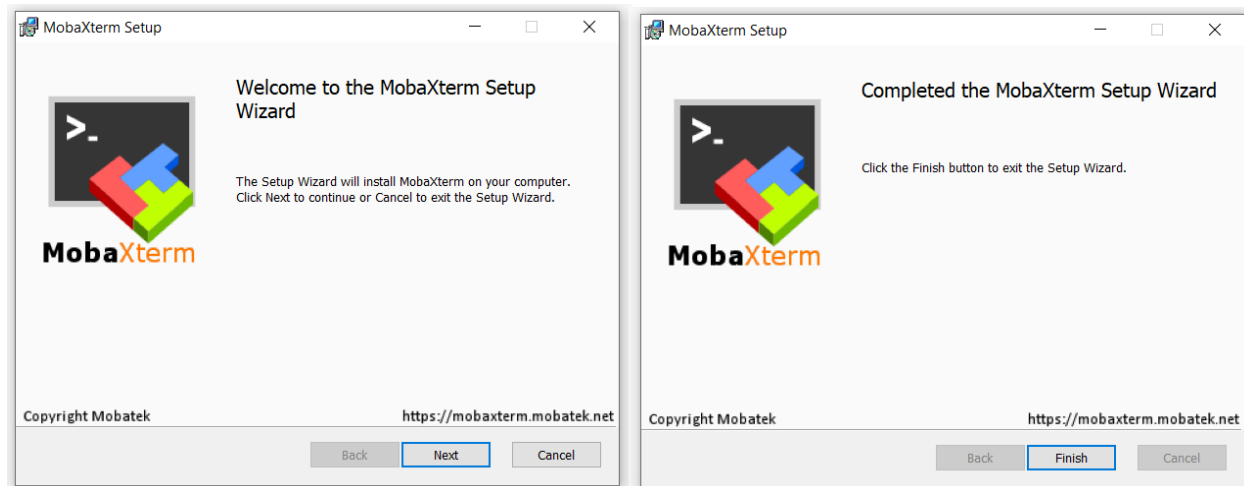


Figure 4: MobaXterm installation window prompts.

Once installed, open MobaXterm. It should look something like *Figure 5*.

Click on “*Start local terminal*”. **Make sure the VPN is running in the background before proceeding further.**

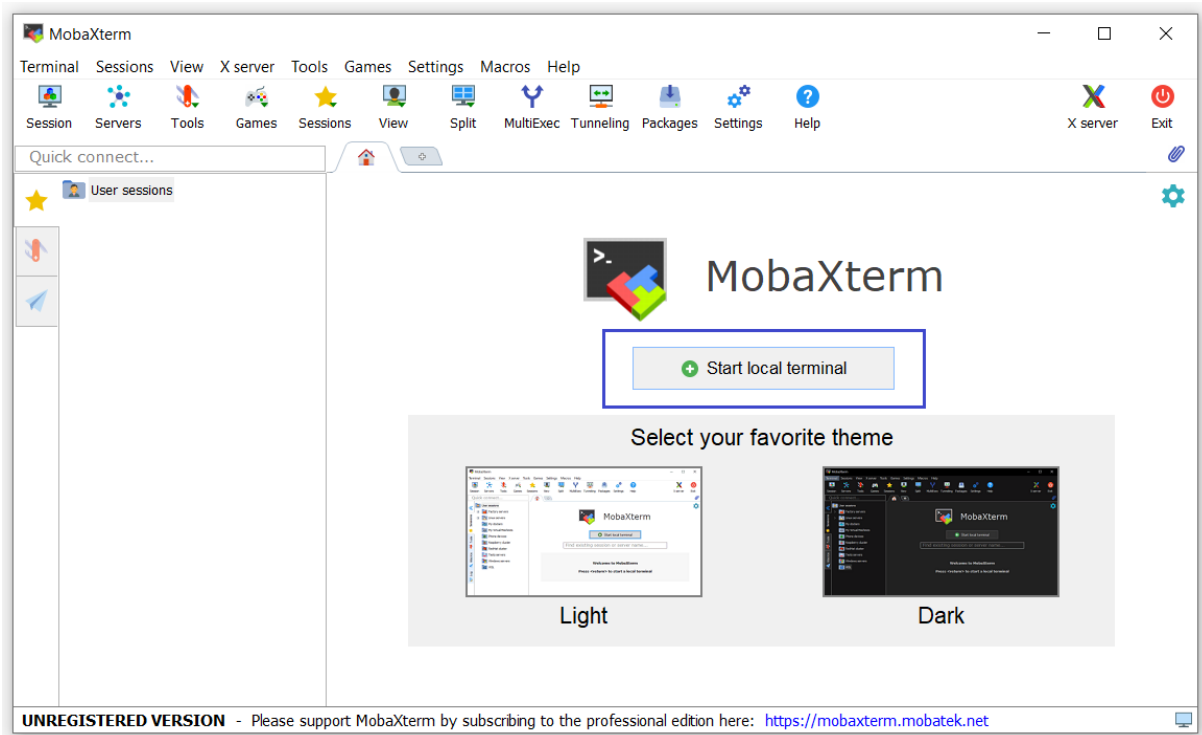


Figure 5: MobaXterm opening screen.

In this terminal, the first command should be:

```
ssh [yourPSUusername]@mo.ece.pdx.edu
```

For example, my Odin and PSU username is ghonim, so my command is (see Figure 6)

```
ssh ghonim@mo.ece.pdx.edu
```

If needed, you can also use

```
@auto.ece.pdx.edu rather than @mo.ece.pdx.edu
```

You must keep note of which you use, as it will be used again later in this Setup tutorial.

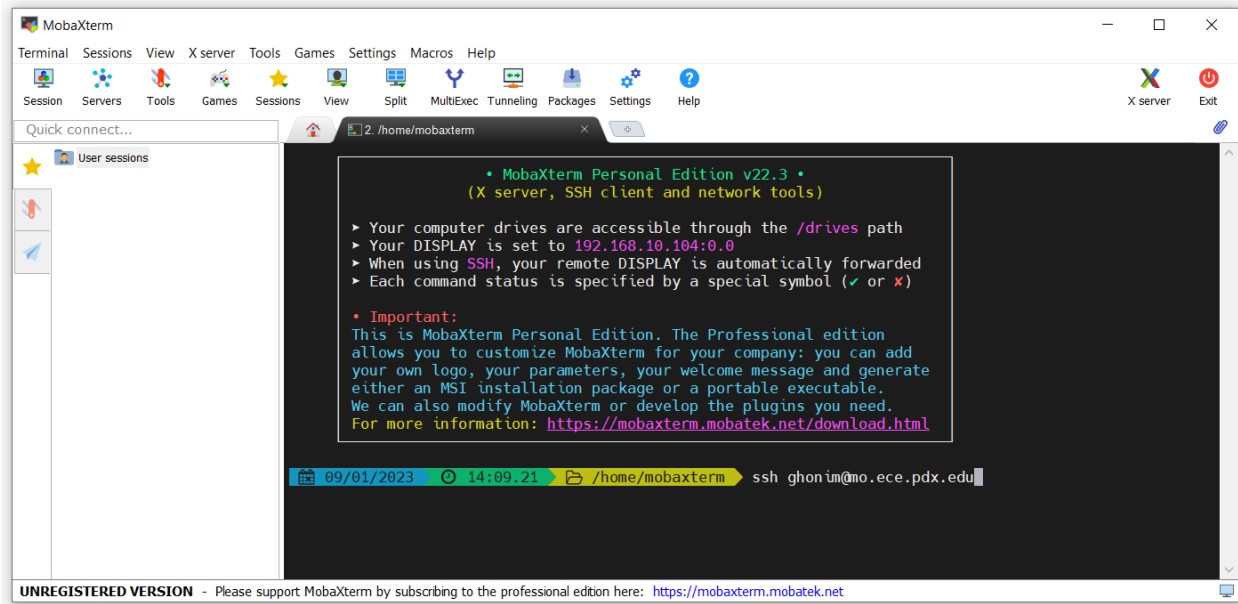


Figure 6: First command example in MobaXterm terminal.

You will then be promoted to enter your CAT PSU password.

After successfully connecting to the remote lab, you should get something like the screen below.

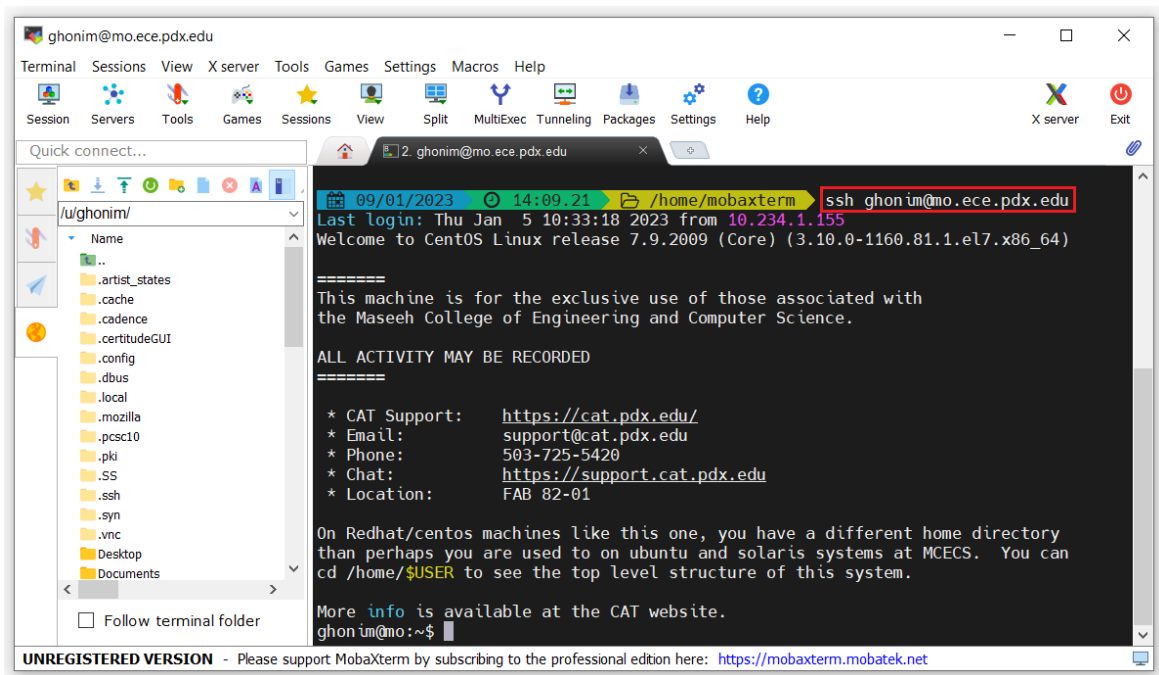


Figure 7: Successfully accessing remote lab.

Now to access a desktop-like graphical user interface (GUI) we will set up a VNC server. To do so, we type the command (also highlighted red in *Figure 8*)

```
vncserver
```

If prompted, please enter a password for the VNC Server connection. This is a new password you're setting up, you can use your PSU CAT password, but it's recommended that you use a different one for security reasons.

```
More info is available at the CAT website.
ghonim@mo:~$ vncserver
Please create a new password
This password will _only_ be used
for the vncserver.
This should be a unique password
separate from your MCECS password
Password:
```

Figure 8: Password prompt after vncserver command.

Now you should get a screen like the one shown in *Figure 9*. All we care about here is the port number, the one highlighted red in *Figure 9* (5902), and the localhost in purple. Please note that your port number could be different, but the localhost will most likely be the same "localhost".

```
3. ghonim@mo.ece.pdx.edu
Would you like to enter a view-only password (y/n)? n
A view-only password is not used
-----
You will need to create an SSH tunnel between your local machine and the host
On MacOS and Linux, run this command in Terminal:
ssh ghonim@mo.ece.pdx.edu -L 5902:localhost:5902
You will then connect your VNC viewer with the vncserver process
running on the host
On Linux, connect the VNC viewer with the host using
localhost:5902
On MacOS, use Finder and connect with the host using
vnc://localhost:5902
If you are using MobaXterm on Windows, this will all be handled by
the built-in VNC functionality and the SSH Gateway/jumphost sub-menu.
See the following guide for more details:
https://cat.pdx.edu/platforms/linux/remote-access/vnc-in-mcecs/
-----
ghonim@mo:~$
New 'mo.ece.pdx.edu:2 (ghonim)' desktop is mo.ece.pdx.edu:2
Starting applications specified in /u/ghonim/.vnc/xstartup
Log file is /u/ghonim/.vnc/mo.ece.pdx.edu:2.log
ghonim@mo:~$
```

Figure 9: Successful access to VNC server.

Once you have done that and know your part number, please don't close the terminal, keep it open, go to the top left side of the MobaXterm window and click on "Session":

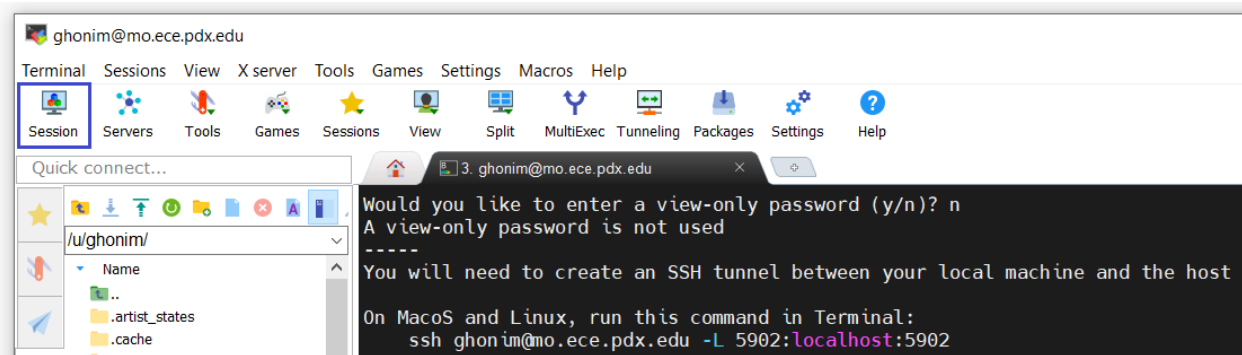


Figure 10: MobaXterm Session tab highlighted blue.

Now click on "VNC", and for the "Remote hostname or IP address *" Please enter

localhost

which was the hostname in purple text we got from the terminal.

Change the port to the one you got in the terminal as well. In my case, it was port "5902".

Once you have done that, click on "SSH gateway (jump host)".

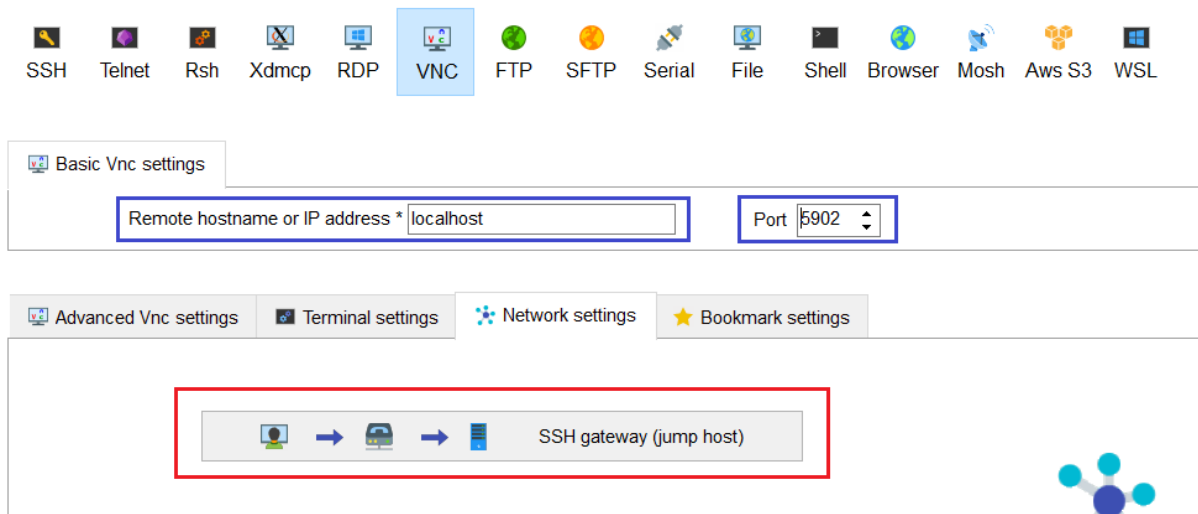


Figure 11: MobaXterm Session window.

In the new window that pops up, please enter

mo.ece.pdx.edu or auto.ece.pdx.edu

in the “Gateway host” field. Whichever you choose in previous steps, enter the corresponding one here to maintain consistency.

Enter your PSU CAT username in the “Username” field, and leave the “Port” field unchanged. Leave the “Use SSH key” field blank.

You should have something like:

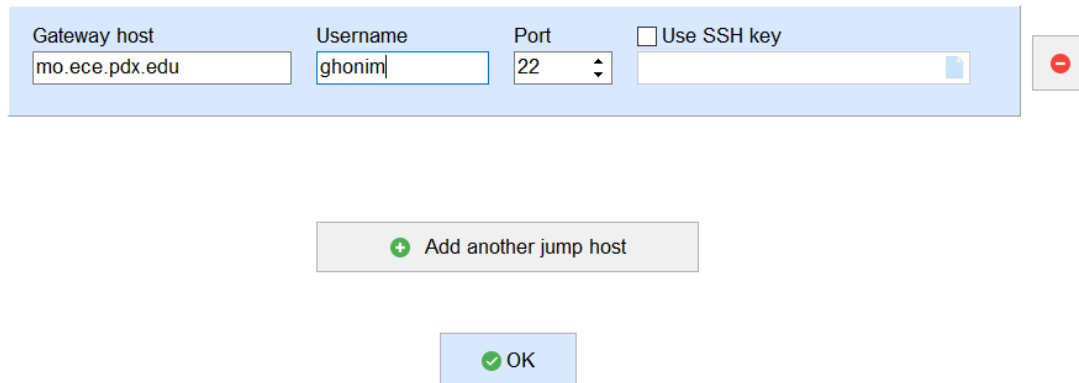


Figure 12: SSH Gateway Host window.

After you click on “OK” you might get a pop-up message like the one shown in *Figure 13*. In this case, you **must** choose “Accept” in order to continue and access the VNC server.

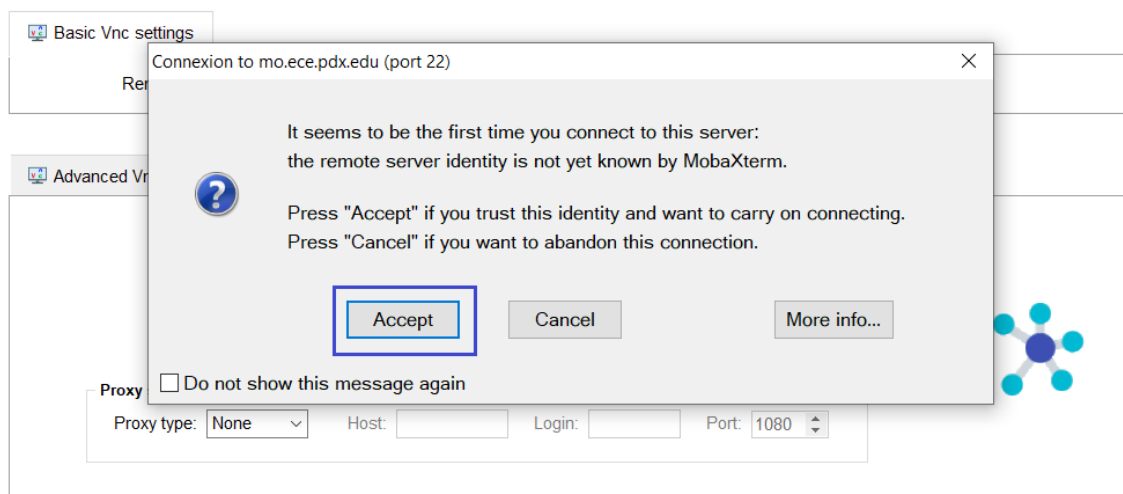


Figure 13: Possible pop-up connection window.

The, you may get a message like the one shown in *Figure 14*. I recommend choosing “Yes” for faster access in the future, but feel free to choose “No”.

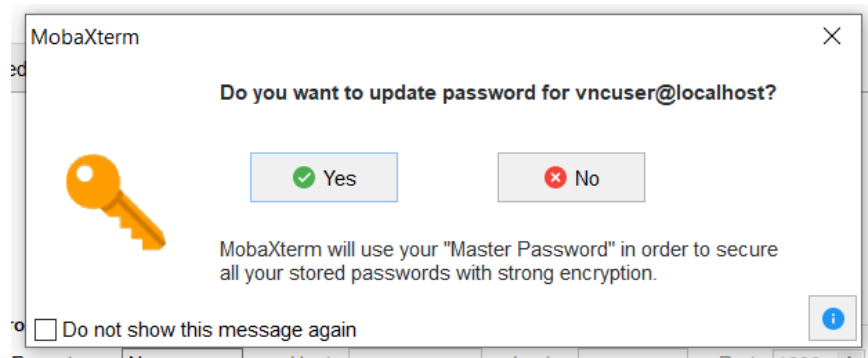


Figure 14: Possible pop-up password window

Now you will be back to the main MobaXterm page, with “localhost” now showing under “recent sessions” as shown in Figure 15 boxed in red. Please click on it.

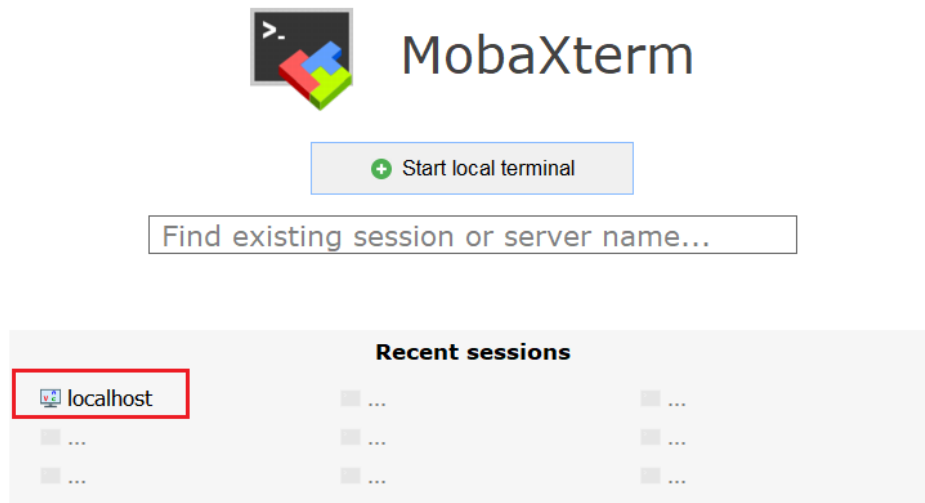


Figure 15: MobaXterm main page.

If asked to enter a localhost password, this should be the **VNC server password** you created above in the terminal. Depending on which password you entered there, **this might not be the same as your PSU CAT password unless you put your same password there.**

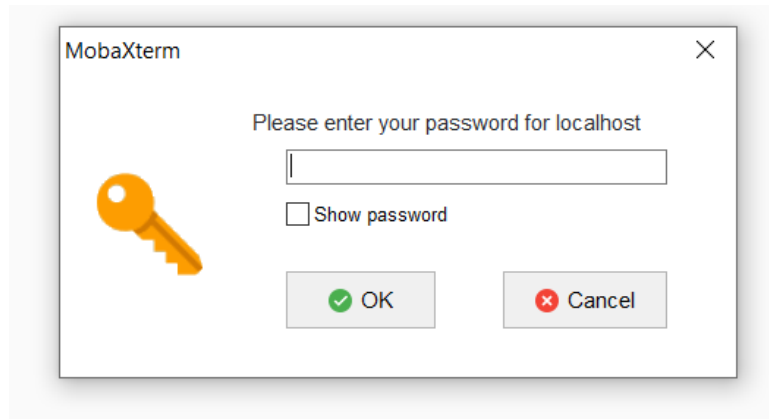


Figure 16: MobaXterm localhost password prompt window.

Now a Linux GUI screen will open in a new tab, and if you are asked to enter your password you will need to enter your **PSU CAT password** there and click “Unlock”.

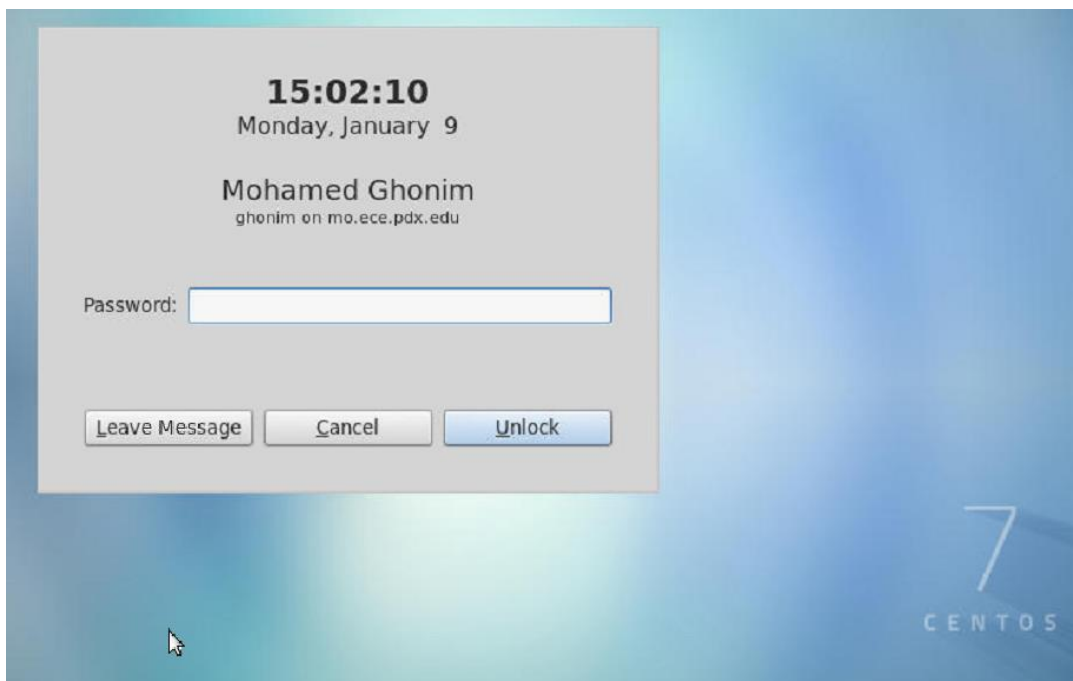


Figure 17: Linux GUI screen prompting password.

Now you should get the Linux desktop and you are ready to install VC Formal!

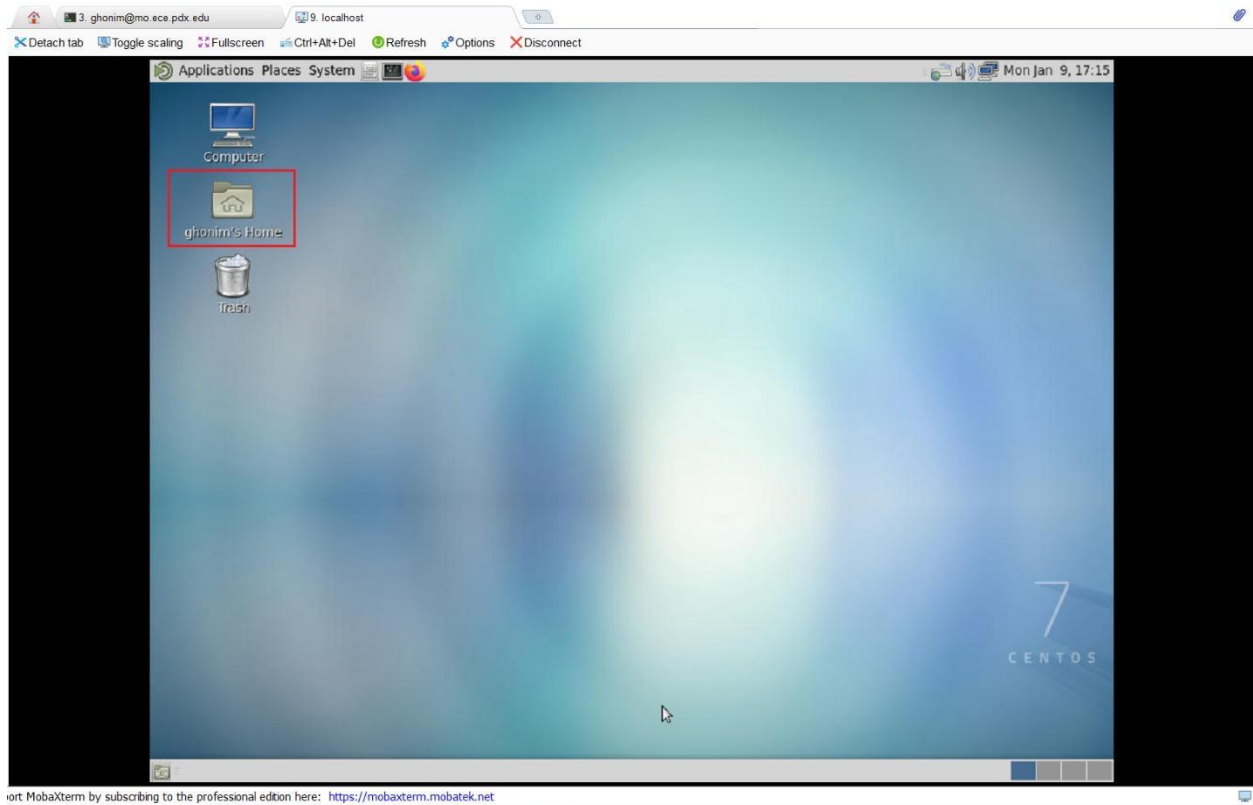


Figure 18: Linux desktop window.

One last, yet important step we need to do is to create a folder for our project. Once you open something like VC Formal, you'll be surprised with how many files get created randomly on your desktop or wherever you have opened VC Formal. We don't want that to happen.

Please click on the home folder (as shown in *Figure 18* highlighted red), it should have your CAT username and then the word home. For example, mine is "*ghonim's Home*".

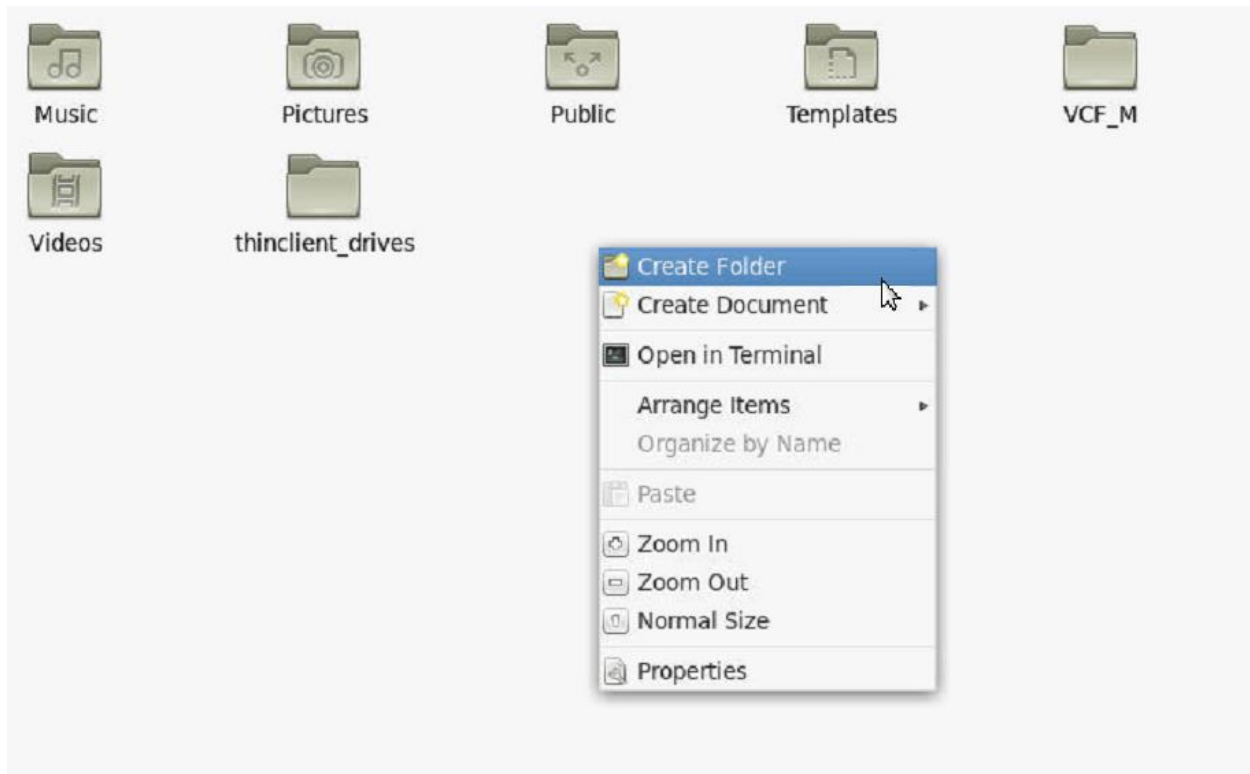


Figure 19: *Creating a new folder.*

You will then need to right-click on the mouse and choose “*Create Folder*” and name your folder anything you want! I created a folder and named it “*VCF_M*” for example.

Now that we have the environment setup, the next step is to install and open Synopsys VC Formal as we show next.

Synopsys® VC Formal Tutorial

Getting Started: Part 3

Installing and Starting Synopsys VC Formal

Version 1.1 | 02-June-2023



Portland State University

©2023

Now that we have the VPN running, and MobaXterm connected, we need to install and start VC Formal. There are multiple ways to do this, you can do it from the MobaXterm terminal, or from

within the Linux GUI by doing a right click anywhere, choosing “Open in Terminal” and continuing with the following steps.

If you have immediately finished the Getting Started: Part 2 tutorial: Click on the “Terminal” tab that was used to set-up and access the Linux GUI. This tab should be labeled as “[yourUsername]@mo.ece.pdx.edu”.

If not: On the MobaXterm home page, click on “Start local terminal”.

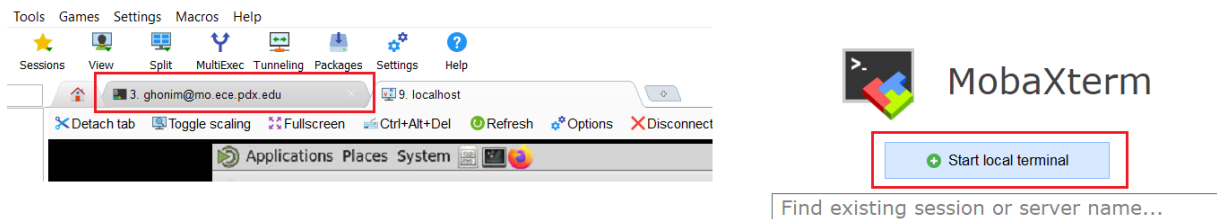


Figure 1: Highlighting the corresponding tabs suggested in bold above.

If you are not already connected, you will need to enter the

```
ssh [yourUsername]@mo.ece.pdx.edu
```

command, click enter, and if prompted, enter your password.

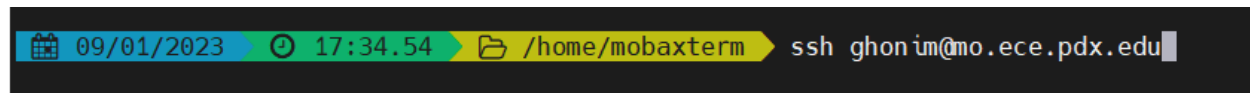


Figure 2: SSH command in MobaXterm terminal.

Once you are connected, you will now enter the

```
addpkg
```

command, and press “Enter” on your keyboard.

```
* CAT Support: https://cat.pdx.edu/
* Email: support@cat.pdx.edu
* Phone: 503-725-5420
* Chat: https://support.cat.pdx.edu
* Location: FAB 82-01

On Redhat/centos machines like this one, you have a different home directory
than perhaps you are used to on ubuntu and solaris systems at MCECS. You can
cd /home/$USER to see the top level structure of this system.

More info is available at the CAT website.
ghonim@mo:~$ addpkg
```

Figure 3: Command after successful access to ssh server.

Your screen should then look like what is shown in *Figure 4*, with the CAT Package Selector prompt. Scroll down until you get to “synopsys-vc_static”. This is Synopsys’s Verification Continuum (VC) package and VC Formal is included within it.

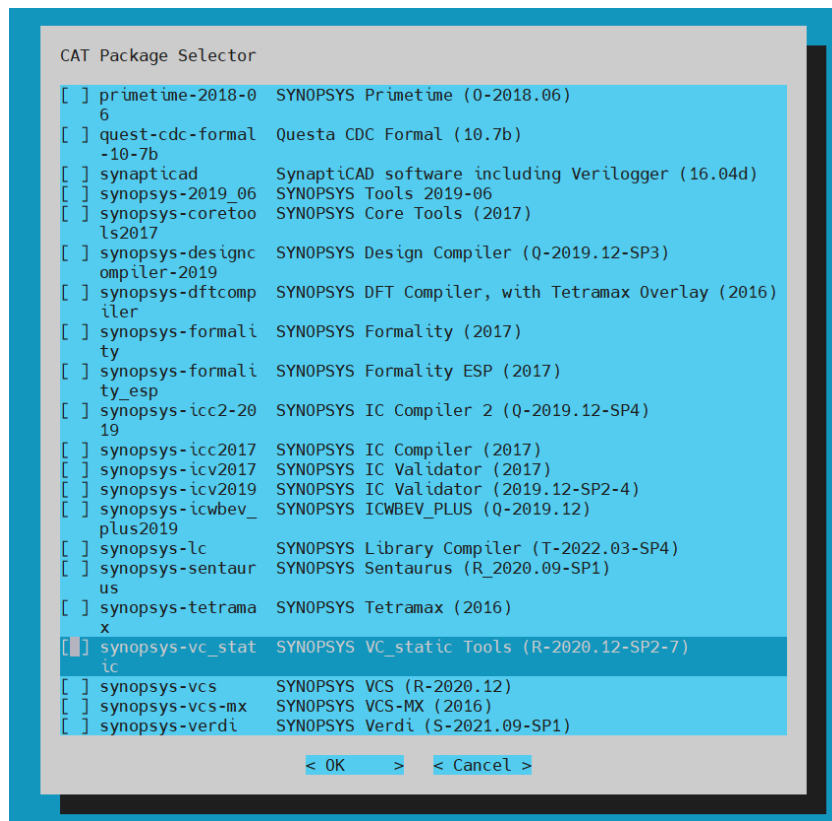


Figure 4: CAT package selector window.

Press “Enter” or “Space” on your keyboard to select the box, then click “OK”

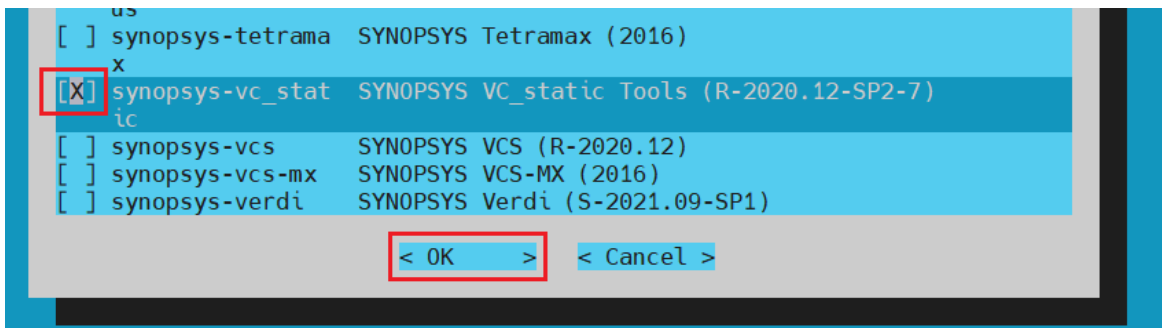


Figure 5: Box selected for correct package.

You must now completely exit and close MobaXterm and open it again for VC Formal to get installed.

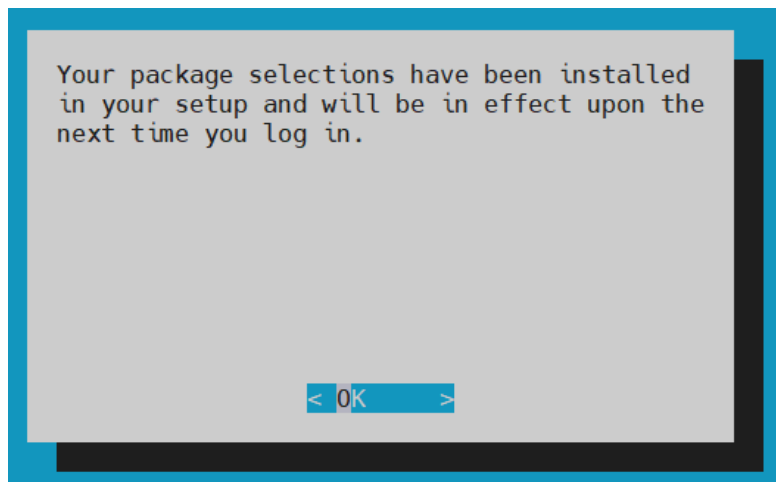


Figure 6: Pop-up window after successfully selecting package.

Once you open MobaXterm again, you will need to

- (1) Start a new local terminal**
- (2) Repeat the steps mentioned before to open Linux GUI**

Once you have opened the Linux GUI, access the empty folder for random VC Formal files. If you followed the last tutorial, this is what you created last time inside your home folder. If you have any issues with this step, please follow the previous steps again from the **Getting Started Part 2** tutorial.

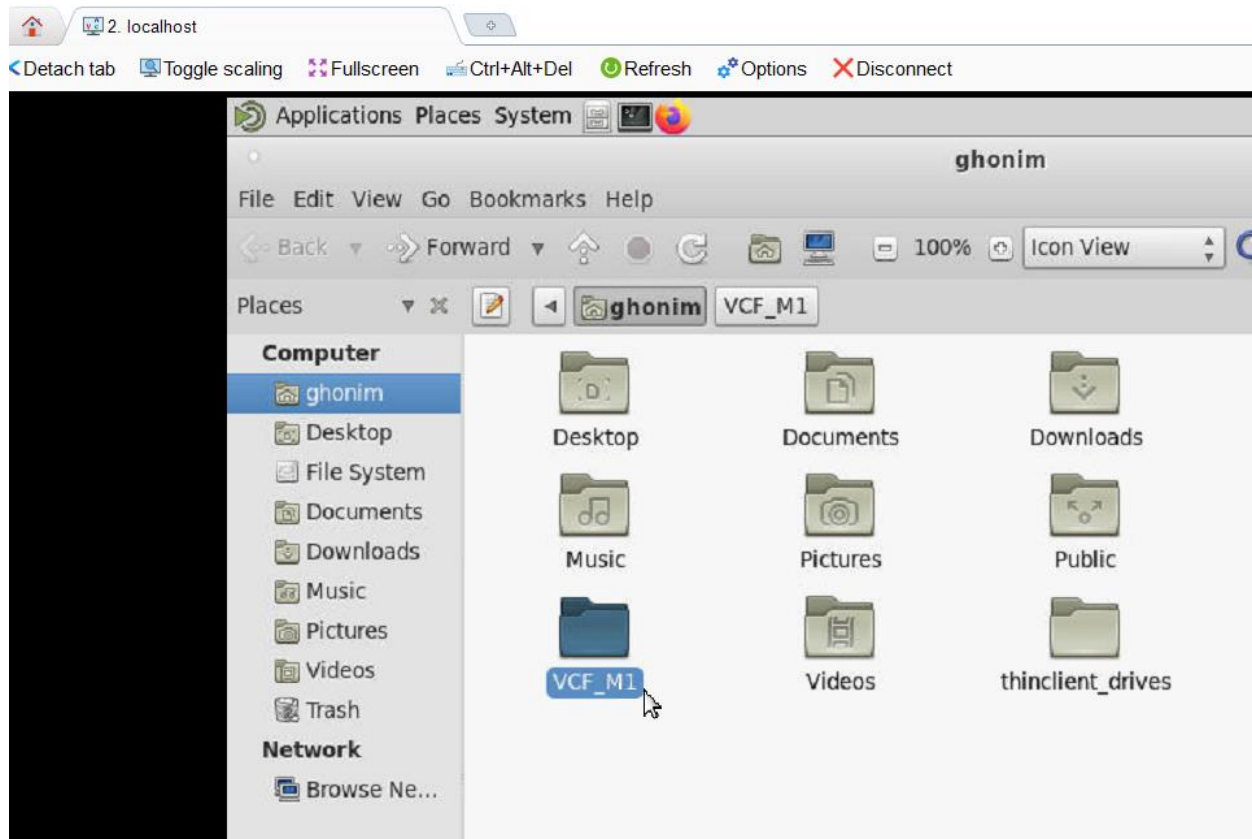


Figure 7: Home folder window highlighting empty folder for VC Formal.

Right click and choose open in terminal:

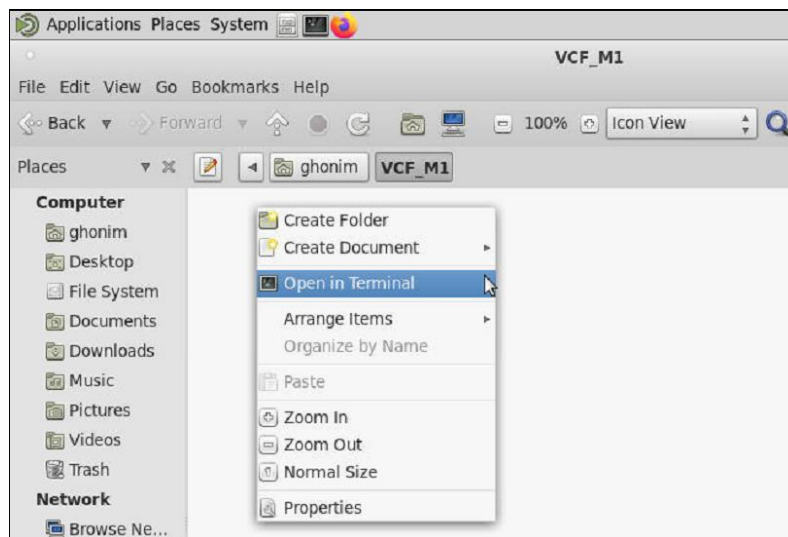


Figure 8: Right click within the empty VC Formal folder.

Once the terminal pops up, please enter the command

vcf -gui

as shown highlighted red in *Figure 9*.

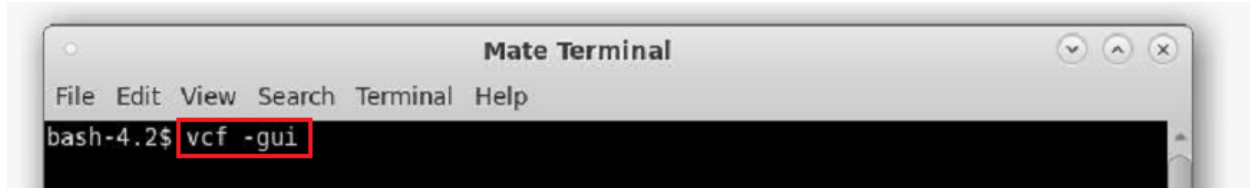


Figure 9: Command in terminal.

It might take a minute or two for VC Formal to load and open. Upon completion, you'll notice some files automatically created in the folder, please do not delete them. (It gets quite messy, that is why we created a new folder, to begin with).

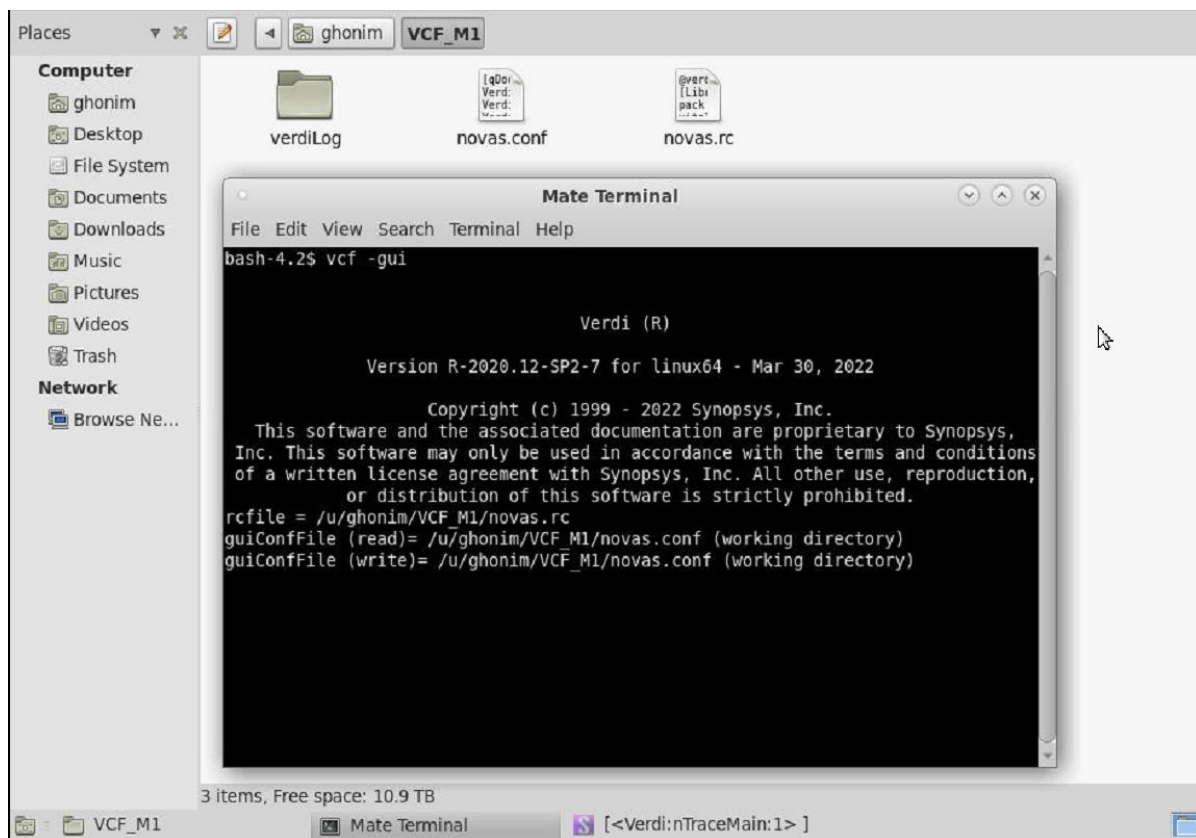


Figure 10: Random files created upon opening VC Formal.

At this point, VC Formal should be up and running.

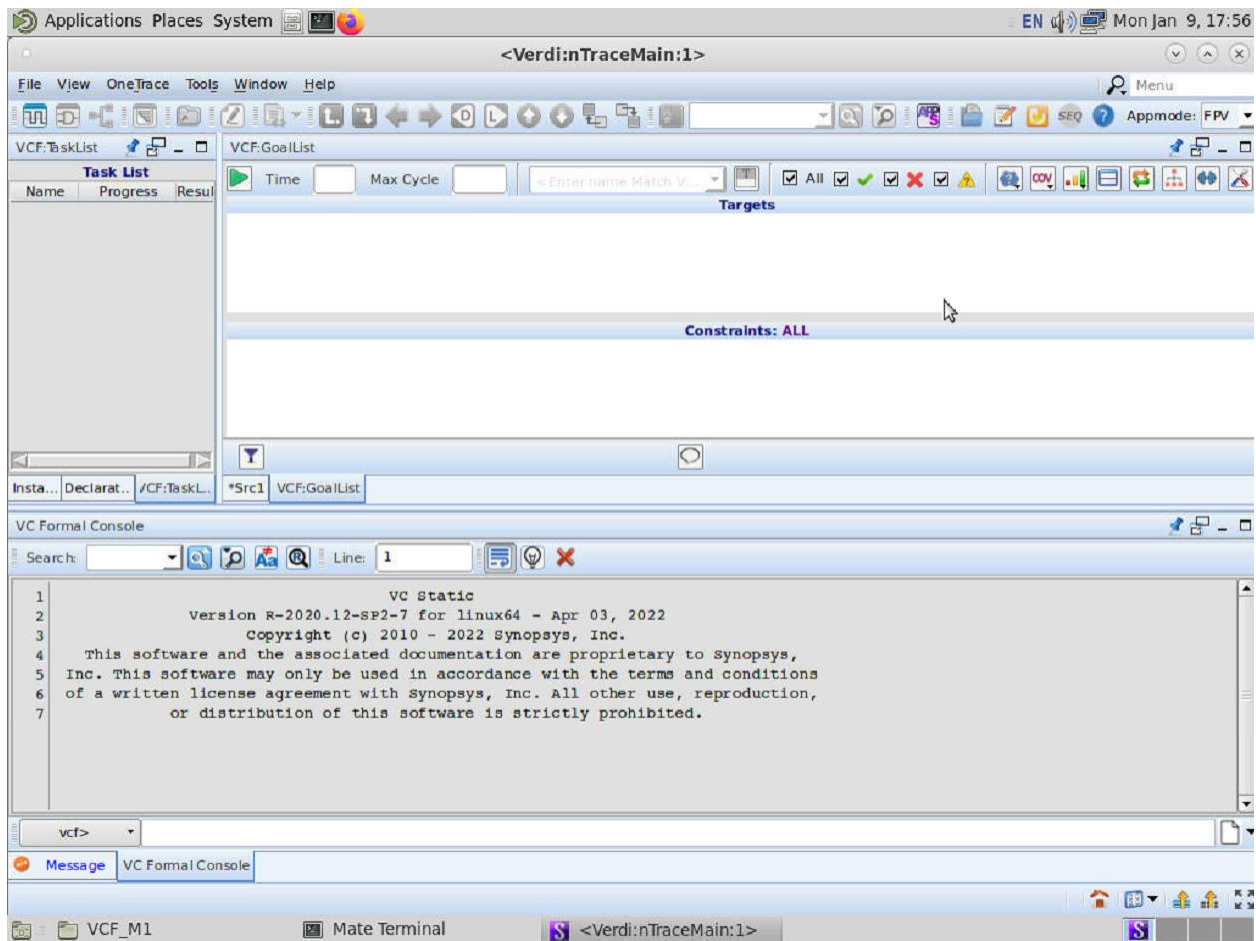


Figure 11: VC Formal starting window. (This screenshot was taken from the VC Formal GUI)

Please note: you can use VC Formal using commands, but some functionalities might be confusing, and might not work at all in the command terminal. So, we use the GUI version of VC Formal (hence the use of “*vcf -gui*” command).

VC Formal has many apps within it, and changing between those apps will change what the interface looks like.

To change between the apps you click on the active app next to “*Appmode:*” and choose the app you would like to work with. For example, I will select “*SEQ*” for Sequential Equivalence Checking.



Figure 12: Changing of appmodes in VC Formal. (This screenshot was taken from the VC Formal GUI)

We will work with TCL files a lot when it comes to using VC Formal and utilizing its powerful abilities. Thus, it's worth noting where the "Load TCL Project File" button is:



Figure 13: Loading TCL file button highlighted red. (This screenshot was taken from the VC Formal GUI)

Now that we have Synopsys VC Formal running, we are ready to start working with its apps and starting our first project!

Synopsys VC Formal Tutorial

Sequential Equivalence Checking App (SEQ)

Version 1.1 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged and only for the information of the intended recipient and may not be published or redistributed.

Equivalence Checking in Synopsys VC Formal SEQ App Youtube Tutorial

Link: <https://youtu.be/Dlci5naLkaY>

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

1. Introduction	3
1.1 About and Usage of SEQ.....	4
1.2 Design Files	5
1.3 TCL File	6
2. Application Setup	8
2.1 Invoking VC Formal GUI.....	9
2.2 Invoking VC Formal Along with TCL File	11
3 Application Usage.....	12
3.1 Detecting Errors.....	13
3.2 Resolving Errors	17
3.3 Restarting VC Formal	18

1. Introduction

The “SEQ” app in VC Formal is used to verify the equivalence of two designs using formal methods. One is called the specification, and the other is the implementation. So, we need designs, or files (usually written in an HDL language like Verilog, SystemVerilog, VHDL, etc.).

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 “SEQ_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 SEQ analysis for us.

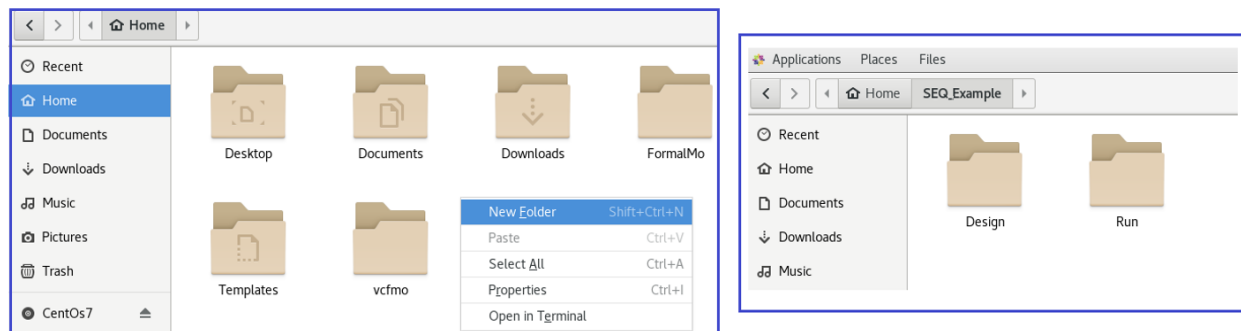
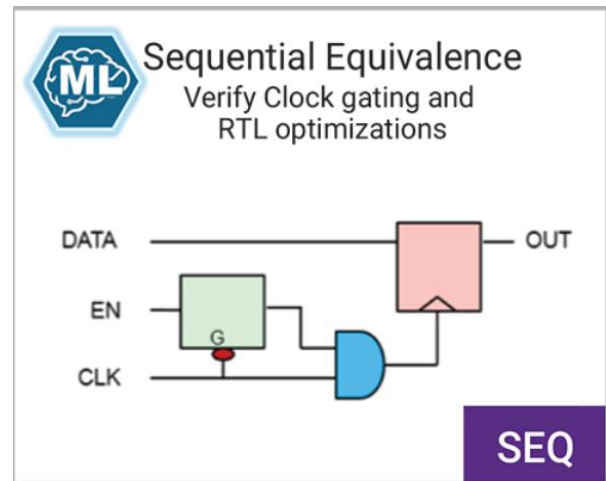


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of SEQ

The Sequential Equivalence Checking “SEQ” app is used to verify the equivalence between two sequential designs. It’s specifically used for sequential equivalence checking tasks. With this app users can verify the functional equivalence of sequential designs by comparing the behavior of two designs. It will perform deep analysis of the designs to ensure that their outputs are equivalent for all possible inputs. By using this app, you can effectively ensure that your design behaves the same and meets your verification goals.



This SEQ figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

1.2 Design Files

In the Design folder, you'll put the two SystemVerilog design files. We are going to tell VC Formal to automatically map the two designs by name, so it's crucial to have the same input and output names in both the designs (don't use indices for example). The module names can be different, however.

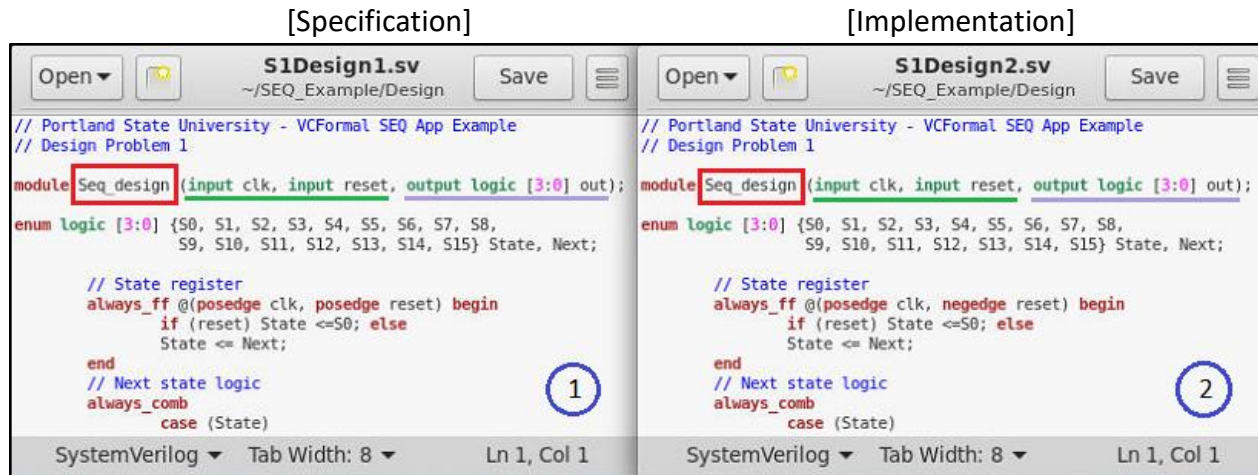


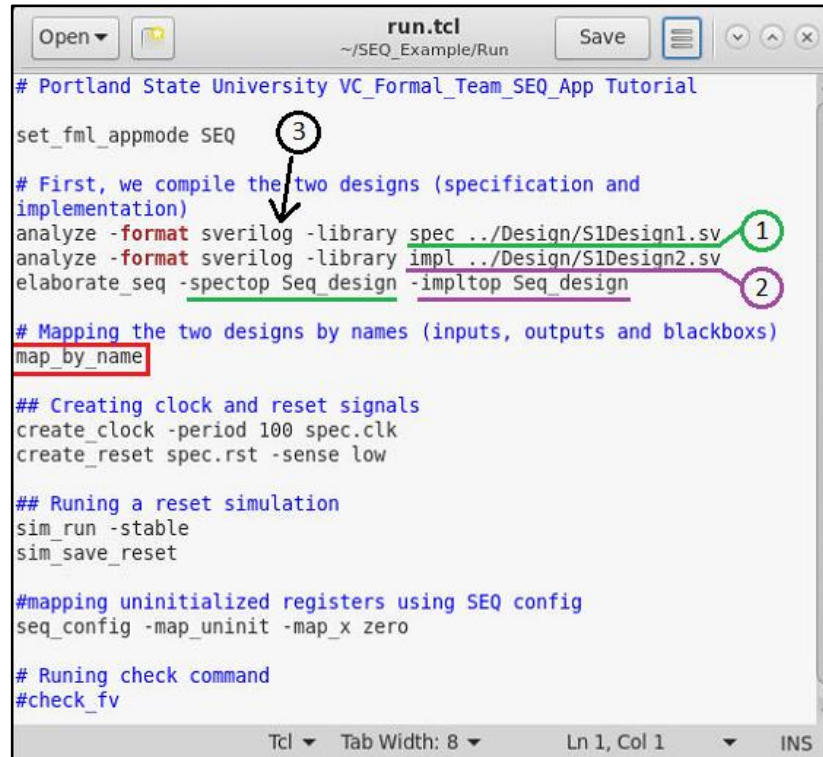
Figure 1.2.1. Showing the two example designs with same input and output module names.

- (1) S1Design1.sv (Specification file)
- (2) S1Design2.sv (Implementation file)

The two designs in *Figure 1.2.1* above are identical, besides the “negative edge triggered reset” in the ***always_ff*** block in the “implementation design” (2). Later, you will see the effects this will have in our simulation.

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.1*), which you can use as a template for your equivalence checking on VC Formal.



```
run.tcl
~/SEQ_Example/Run
Save

# Portland State University VC_Formal_Team_SEQ_App Tutorial

set_fml_appmode SEQ (3)

# First, we compile the two designs (specification and implementation)
analyze -format sverilog -library spec ../Design/S1Design1.sv (1)
analyze -format sverilog -library impl ../Design/S1Design2.sv (2)
elaborate_seq -spectop Seq_design -impltop Seq_design

# Mapping the two designs by names (inputs, outputs and blackboxes)
map_by_name

## Creating clock and reset signals
create_clock -period 100 spec.clk
create_reset spec.rst -sense low

## Running a reset simulation
sim_run -stable
sim_save_reset

#mapping uninitialized registers using SEQ config
seq_config -map_uninit -map_x zero

# Running check command
#check_fv
```

Figure 1.3.1. Annotated TCL file.

- ❖ The “**spec**” command indicates the specification file; represented by (1).
- ❖ The “**impl**” command indicates the implementation file; represented by (2).
- ❖ The “**sverilog**” command means that the designs are written in SystemVerilog; represented by (3).
- ❖ The “**set_fml_appmode SEQ**” line is an instruction that sets the appmode to SEQ in VC Formal.
- ❖ The two dots, “**../**” before the file location tells VC Formal to go to the main folder, then open the folder “Design” and look for the files “**S1Design1.sv**” and “**S1Design2.sv**”.
- ❖ When you are running your own designs, you would need to change “**S1Design1.sv**” and “**S1Design2.sv**” to your file names.
- ❖ The “**elaborate_seq -spectop Seq_design -impltop Seq_design**” highlighted line shows that the module name in both the specification and implementation designs are both named “Seq_design”. **You will need to change this when running your own designs to its appropriate module names.**

You can explore the other options in the TCL script if you like, but most importantly, when running your own designs, you need to change the file names in lines **(1)** and **(2)** from *Figure 1.3.1* and the module names in the **elaborate_seq -spectop** line.

2. 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 SEQ 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.tc*” 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.

2.1 Invoking VC Formal GUI

Now, we can 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

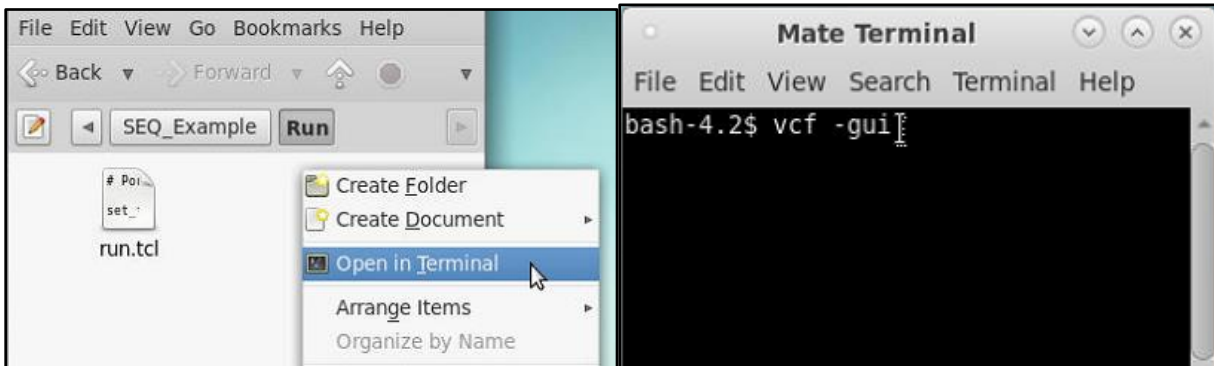


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2* 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: 

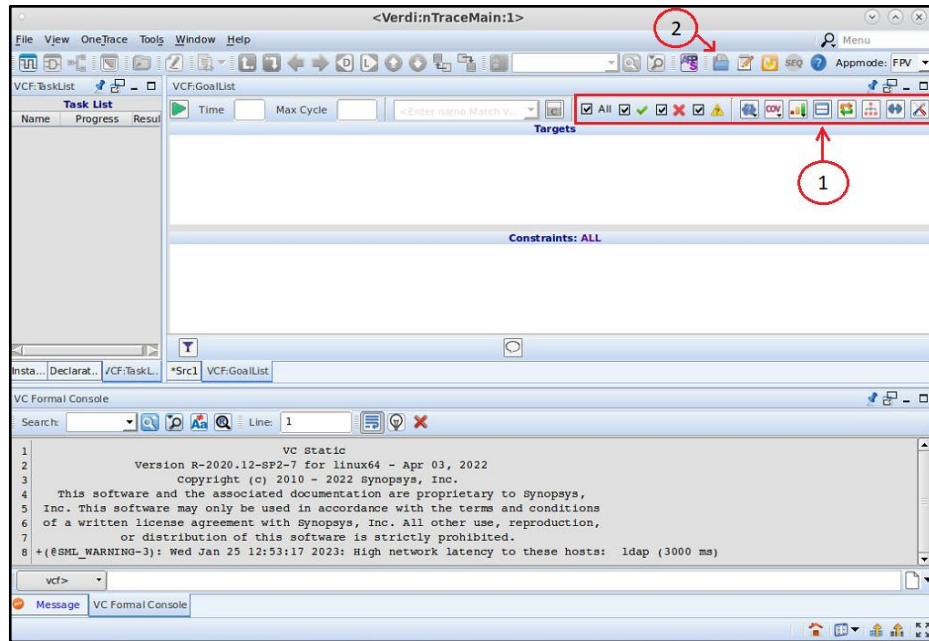


Figure 2.1.2. VC Formal GUI introductory screen. (This screenshot was taken from the VC Formal GUI)

Then load a TCL script by clicking on the  icon (2) as shown in Figure 2.1.2.

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

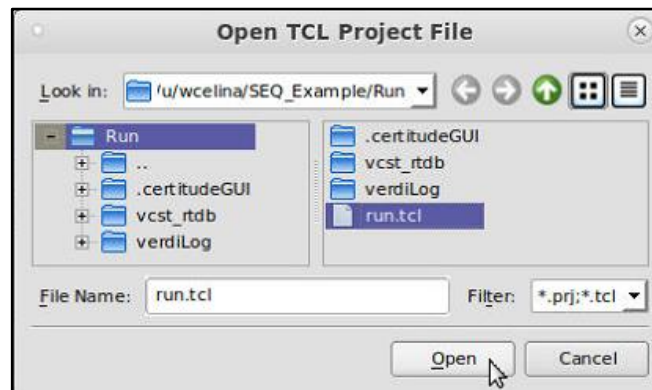


Figure 2.1.3. Selecting TCL file.

2.2 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
```

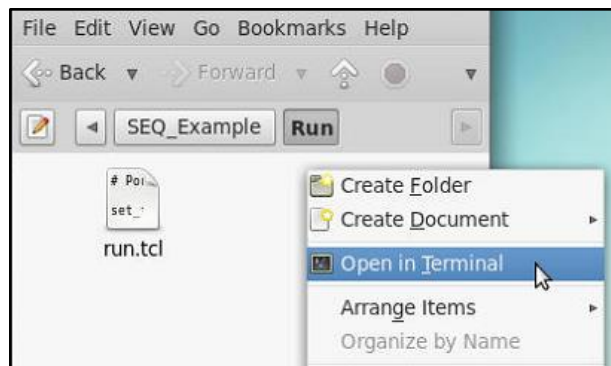


Figure 2.2.1. Opening terminal in the ‘run’ folder.

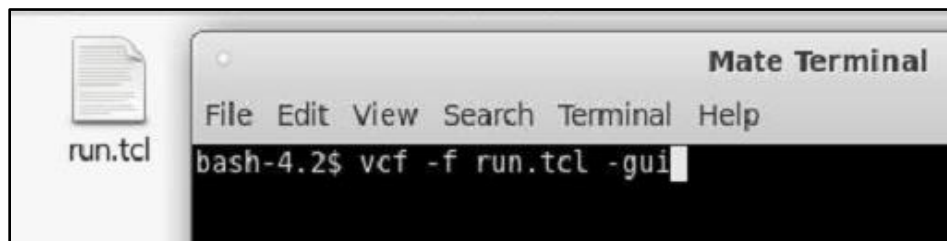


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

3. Application Usage

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:

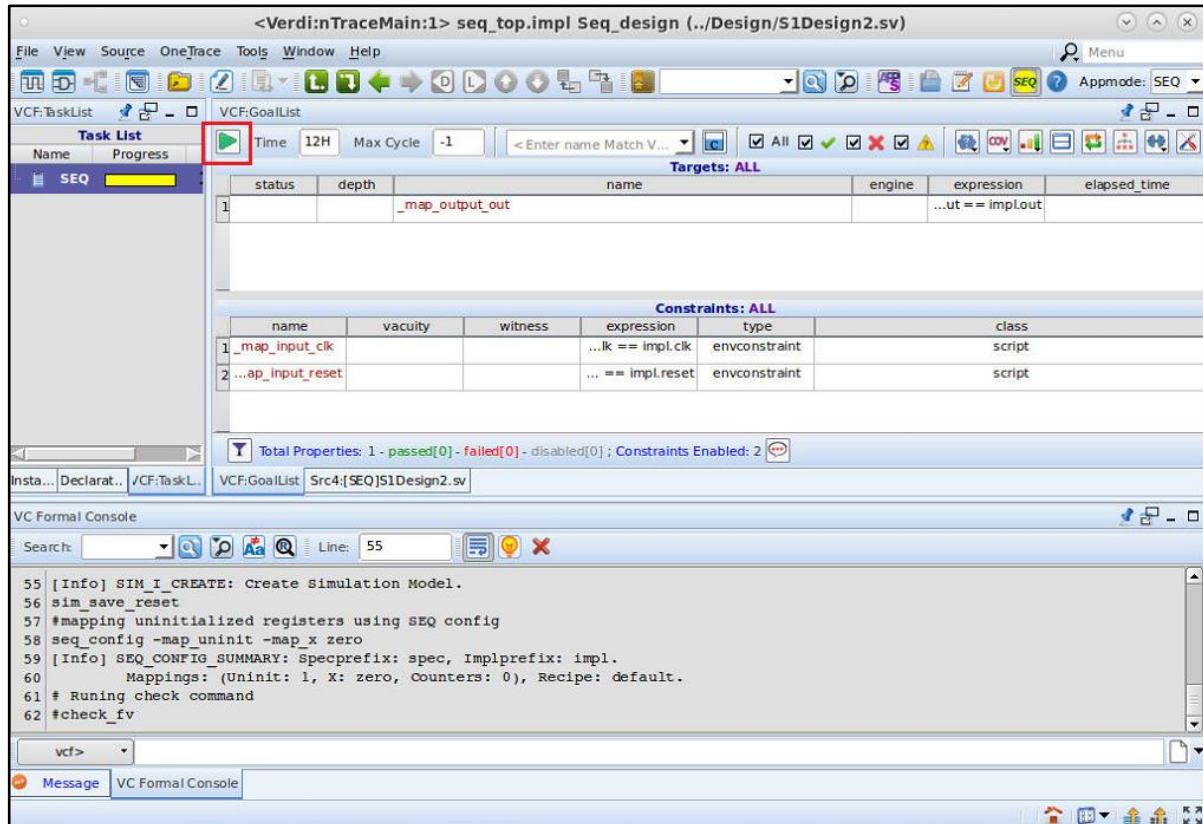


Figure 3.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the “VCF:GoalList” tab.

3.1 Detecting Errors

In the “Task List” on the left, hover over the results to see that VC Formal was given 1 task (one target, an output) to verify its equivalence, and it was shown that the two designs were not equivalent: “Failed/Falsified”. The other two outcomes would have been “Passed/Proved” and “Inconclusive” along with the additional options shown below.

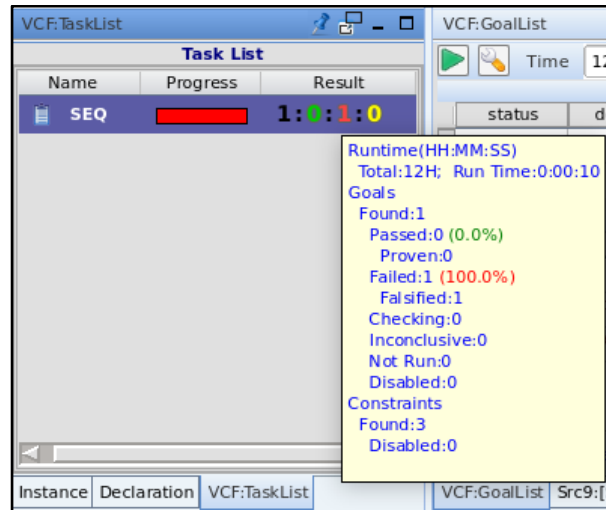


Figure 3.1.1. A closer look into the details of the analysis. (This screenshot was taken from the VC Formal GUI)

Now, we check the “Status” icon. Here, we see the  icon under “status”, which means that the two designs are not equivalent.

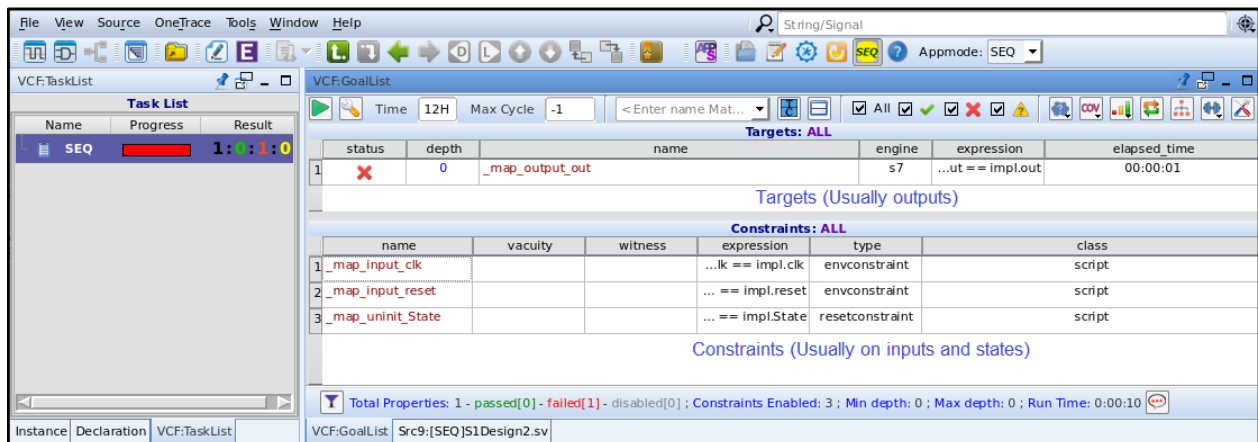


Figure 3.1.2. After running the script. Status given = . (This screenshot was taken from the VC Formal GUI)

To check for the inequivalent part of the design, click on the  icon. We should then get the screen below:

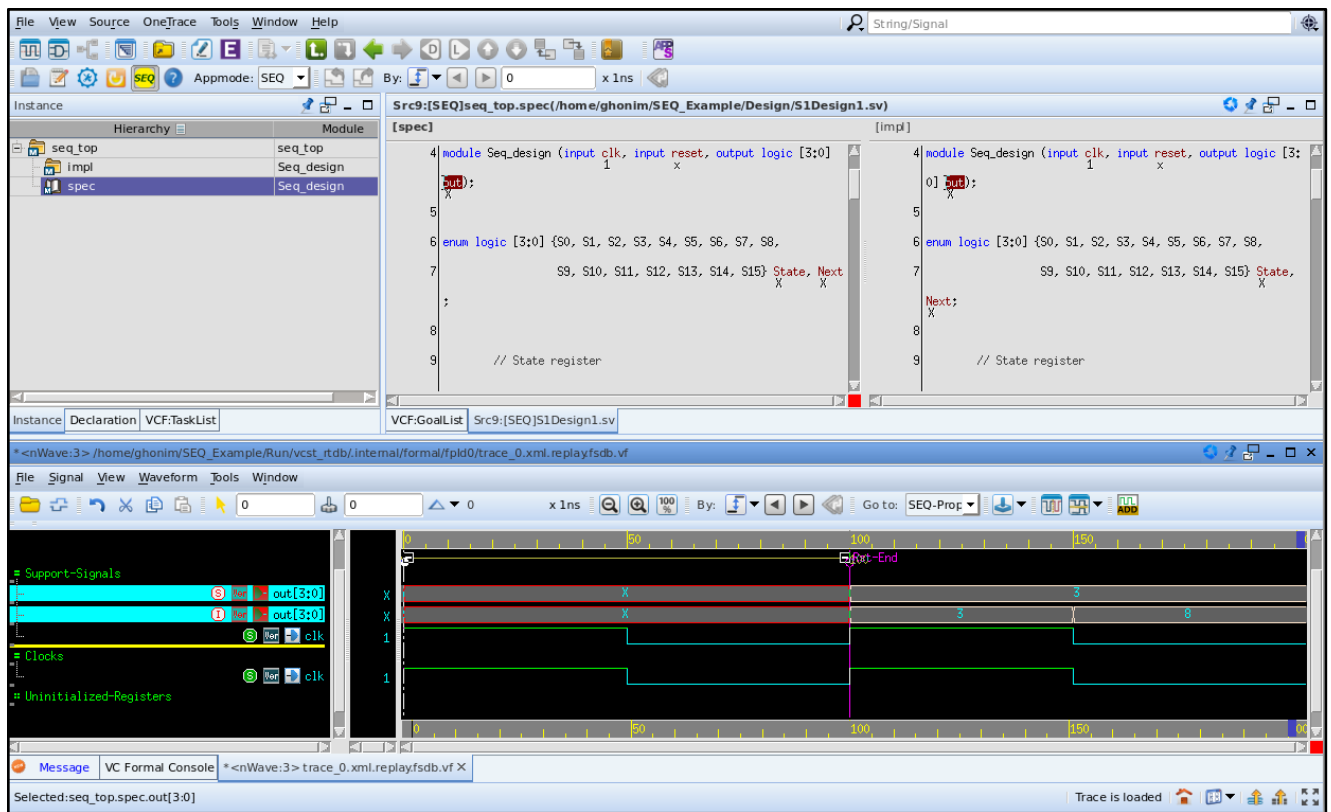


Figure 3.1.3. Examining the inequivalent parts of the design. (This screenshot was taken from the VC Formal GUI)

In *Figure 3.1.3*, in the black box on the lower left corner, we see “Support Signals” in green text, showing the output signals of the specification design (S) and the implementation design (I). The red color indicates the mismatch between the designs. We can also see that the outputs are not equivalent. To see where this is coming from, we can trace it back to its source by going back to the driving signal to that output:

As shown in *Figure 3.1.4* below, under “Support Signals”:

Right-click the implementation design (I) → Show OnceTrace Signals → Driver

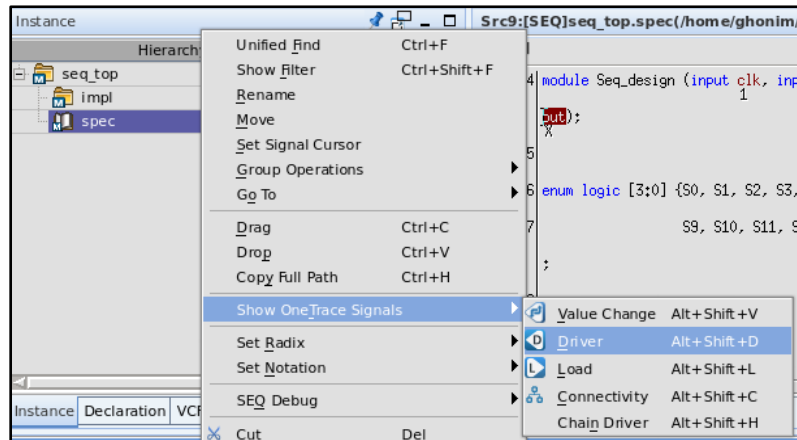


Figure 3.1.4. Tracing the source to find the discrepancy. (This screenshot was taken from the VC Formal GUI)

After selecting “Driver”, the Wave window should look like this:

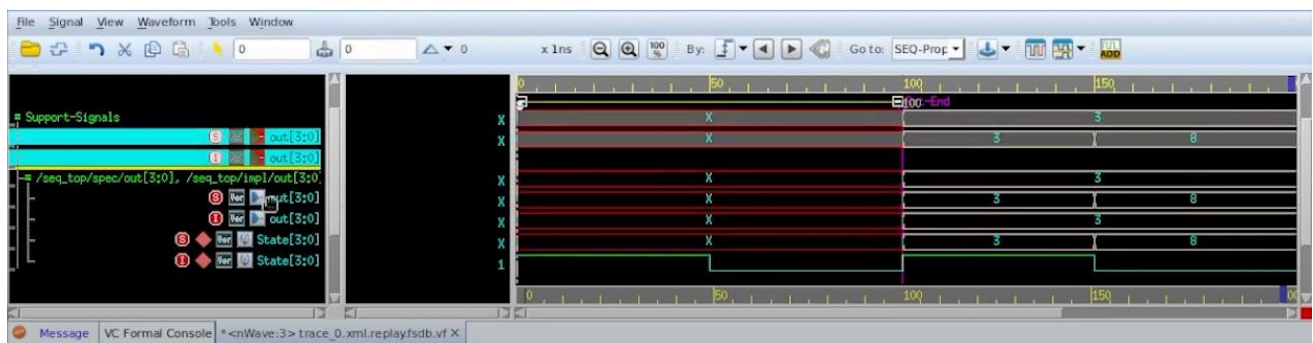


Figure 3.1.5. After tracing once, this is the waveform we get. (This screenshot was taken from the VC Formal GUI)

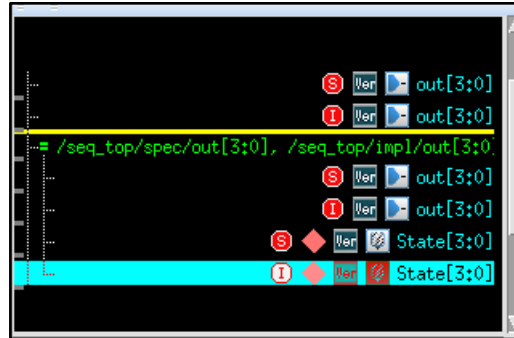


Figure 3.1.6. The states can also be seen as inequivalent. (This screenshot was taken from the VC Formal GUI)

We still need to trace this back more. Keep repeating the procedure above multiple times, or use the shortcut

Alt+Shift+D

until we see a more specific source of the inequivalence.

Here are the manual instructions again:

Right-click the implementation design (I) → Show OnceTrace Signals → Driver

Eventually, we will see that the mismatch is in the reset signal. There is also an “x” sign under it on the elaborated spec and impl codes as shown below.

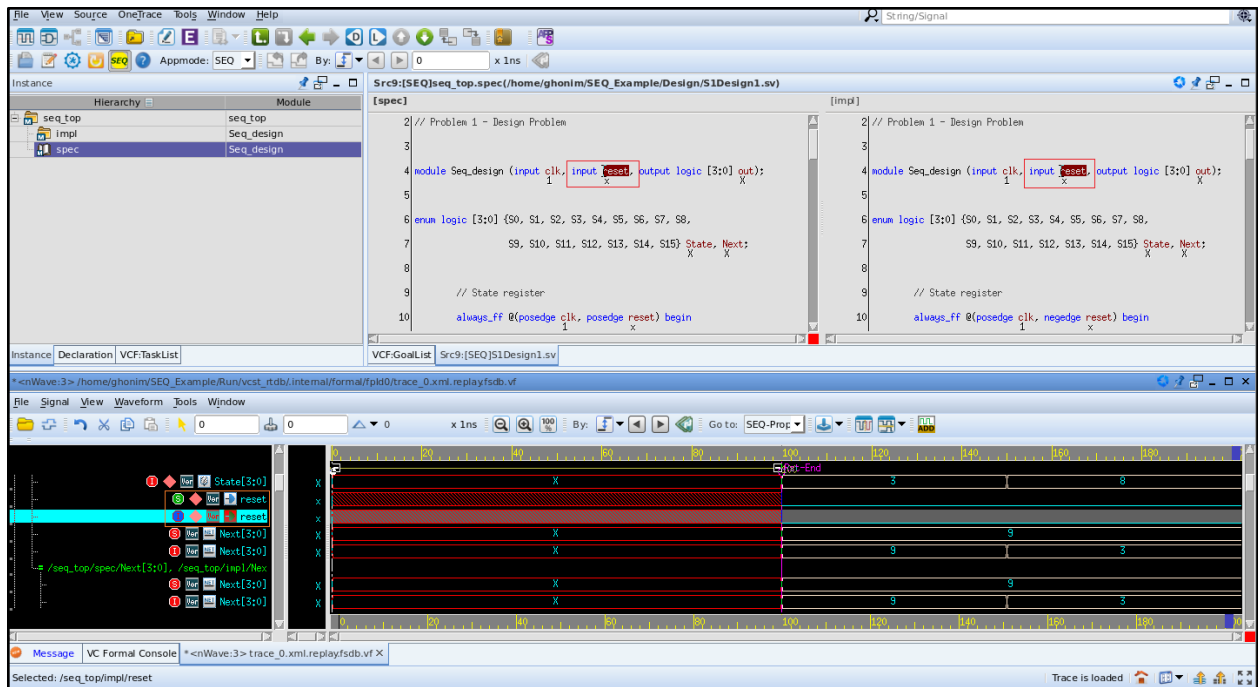


Figure 3.1.7. After multiple repeats of tracing. We can also see exactly where the errors are in the design files in VC Formal. (This screenshot was taken from the VC Formal GUI)

3.2 Resolving Errors

To fix this issue, we check the two design files and look at the [reset signal](#) in the Finite State Machine (FSM) implementation. Remember in the beginning of this tutorial, we noted that the implementation design has a negative edge-triggered reset, while the specification design has a positive edge-triggered reset.

Alter the implementation design and switch the **negedge reset** to a **posedge reset**. Now both the specification and implementation files should have the same reset.

Save the file, and go back to VC Formal.

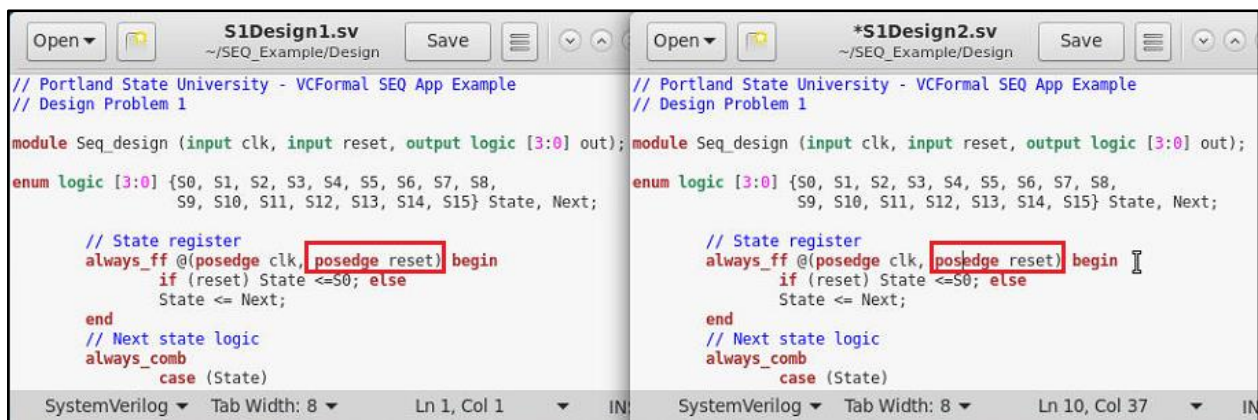


Figure 3.2.1. Altering the S1Design2.sv implementation file to have a posedge reset.

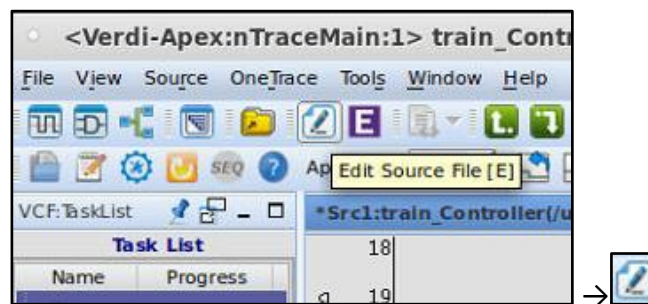


Figure 3.2.2. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes. (This screenshot was taken from the VC Formal GUI)

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

3.3 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 3.3.1. Restarting VC Formal. (This screenshot was taken from the VC Formal GUI)

Now load the TCL script and run the simulation as we did before. This time, we can see that the specification and implementation designs are equivalent through the  icon.

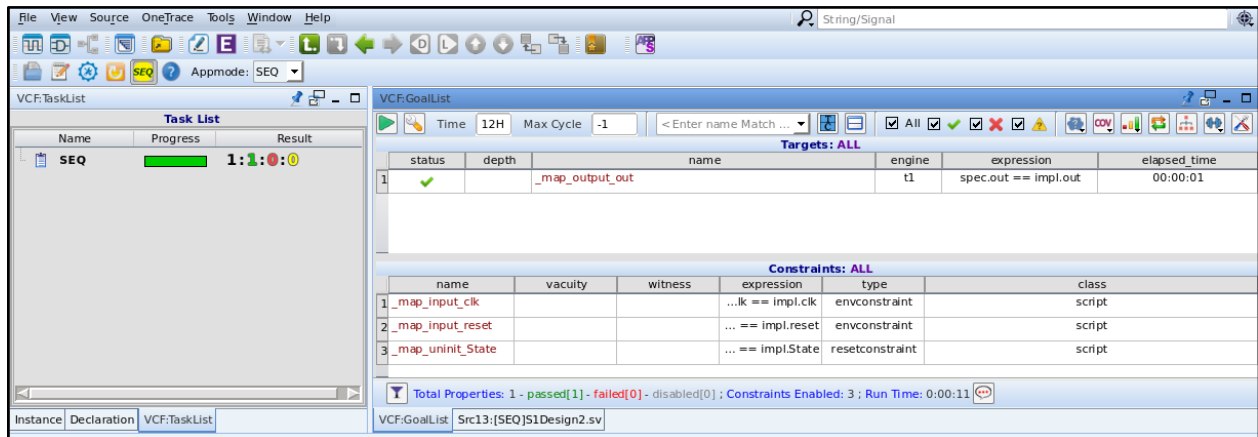


Figure 3.3.2. Equivalence checking is successful. (This screenshot was taken from the VC Formal GUI)

Synopsys® VC Formal Tutorial

Automatically Extracted Property (AEP)

Version 1.2 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of AEP	4
1.2 Design Files	5
1.3 TCL File	6
2. Application Setup	7
2.1 Invoking VC Formal GUI	8
2.2 Invoking VC Formal Along with TCL File:	10
3. Application Usage	11
3.1 Detecting Errors	12
3.2 Source Tracing	14
3.3 Resolving Errors	16
3.4 Restarting VC Formal	20
Appendix	21

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 “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.

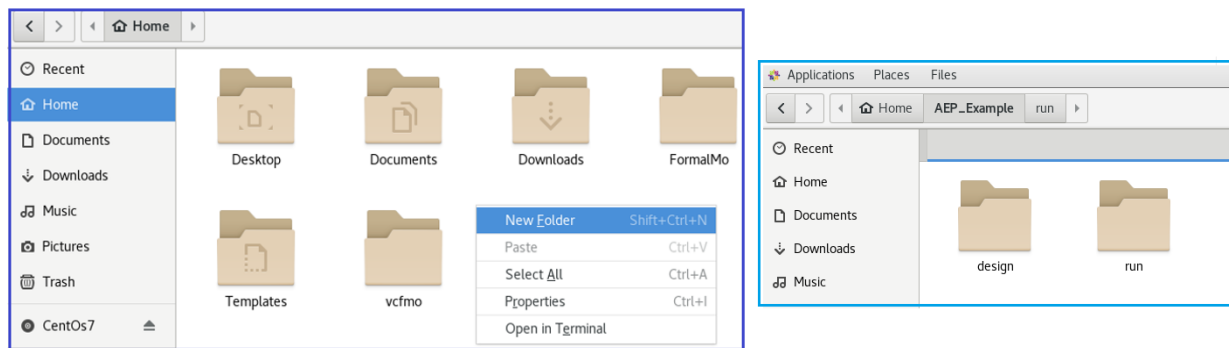
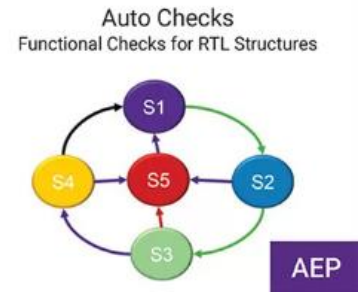


Figure 1.1. Creating folders to organize design and TCL files.

1.1 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.



This AEP figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

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

1.2 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.

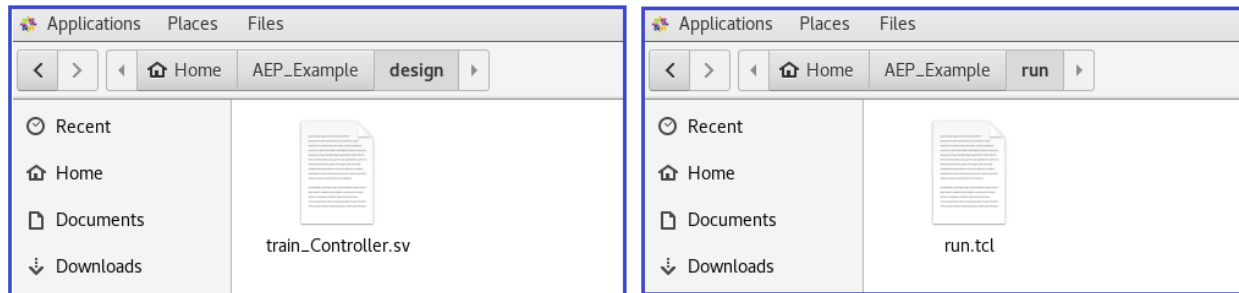


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

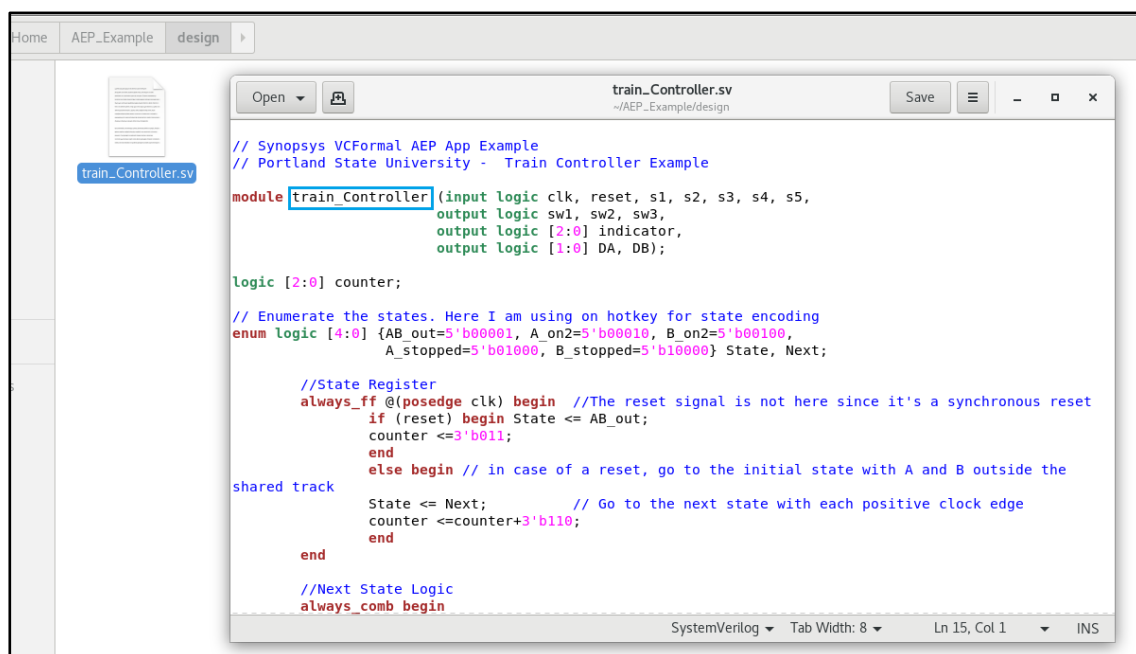
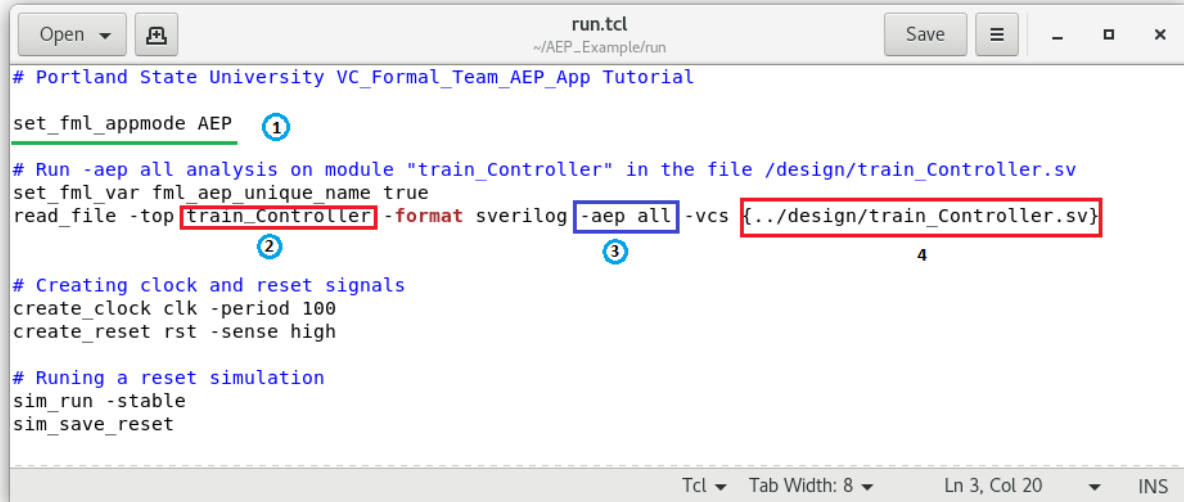


Figure 1.2.2. Example of the design we are using in this tutorial.

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 1.2.2* 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).

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.1*), which you can use as a template for your functional checks on VC Formal.



```
# Portland State University VC_Formal_Team_AEP_App Tutorial

set_fml_appmode AEP ①

# Run -aep all analysis on module "train_Controller" in the file /design/train_Controller.sv
set_fml_var fml_aep_unique_name true
read_file -top ② train_Controller ③ -format sverilog -aep all -vcs ④ {../design/train_Controller.sv}

# Creating clock and reset signals
create_clock clk -period 100
create_reset rst -sense high

# Running a reset simulation
sim_run -stable
sim_save_reset
```

Figure 1.3.1. 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 2.2.2*.

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

2. 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.

2.1 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
```

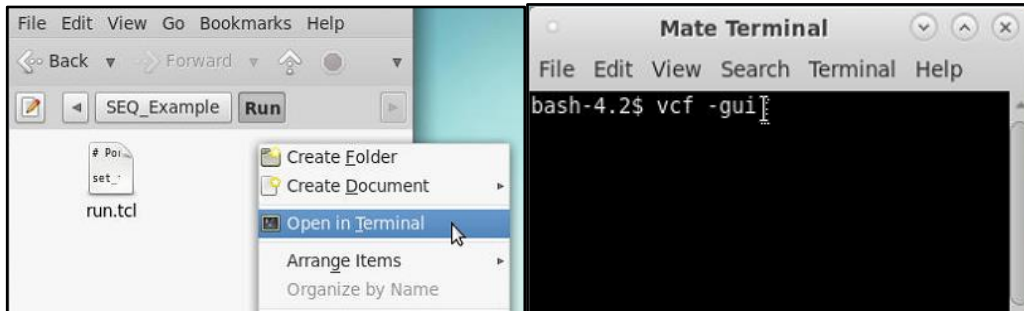


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2* 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: 

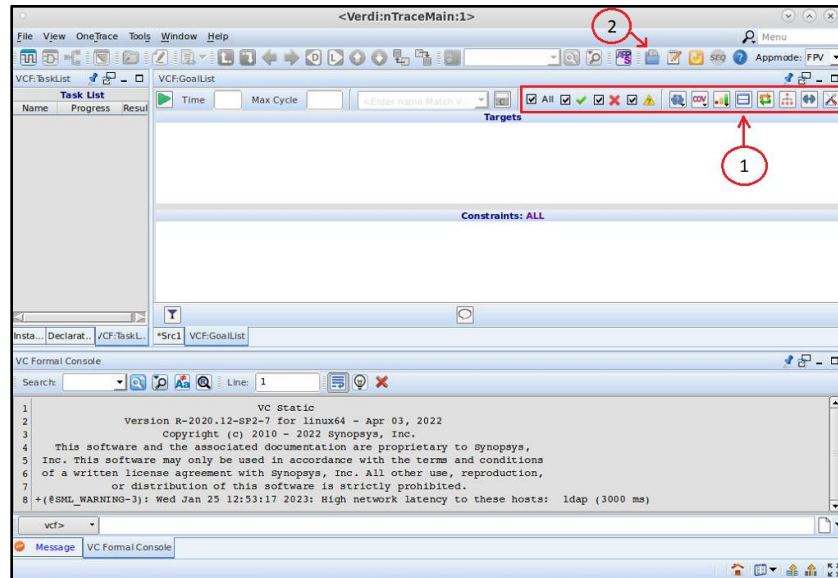


Figure 2.1.2. VC Formal GUI introductory screen. (This screenshot was taken from the VC Formal GUI)

Then load a TCL script by clicking on the  icon **(2)** as shown in *Figure 2.1.2*.

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

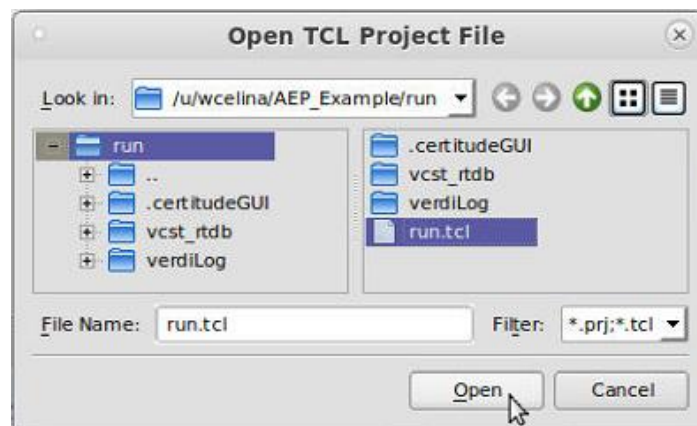


Figure 2.1.3. Selecting TCL file.

2.2 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
```

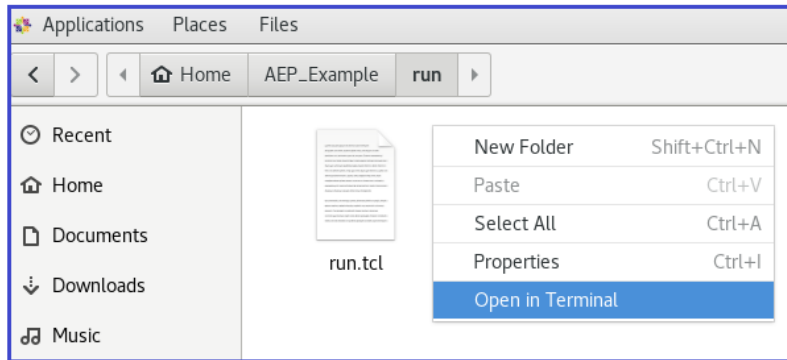


Figure 2.2.1. Opening terminal in the ‘run’ folder.

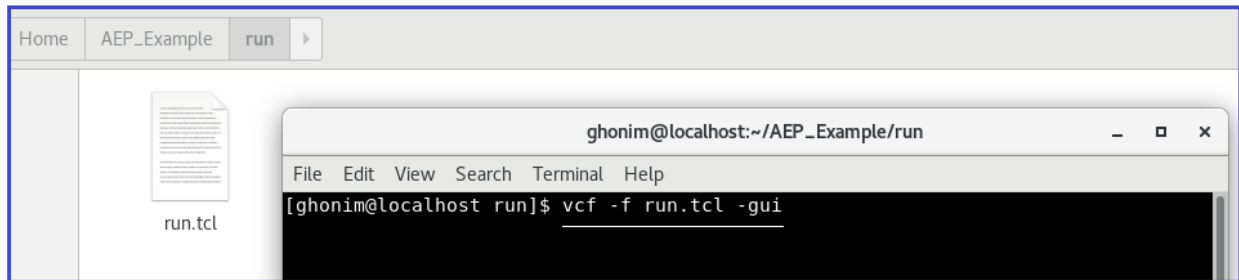


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

3. Application Usage

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:

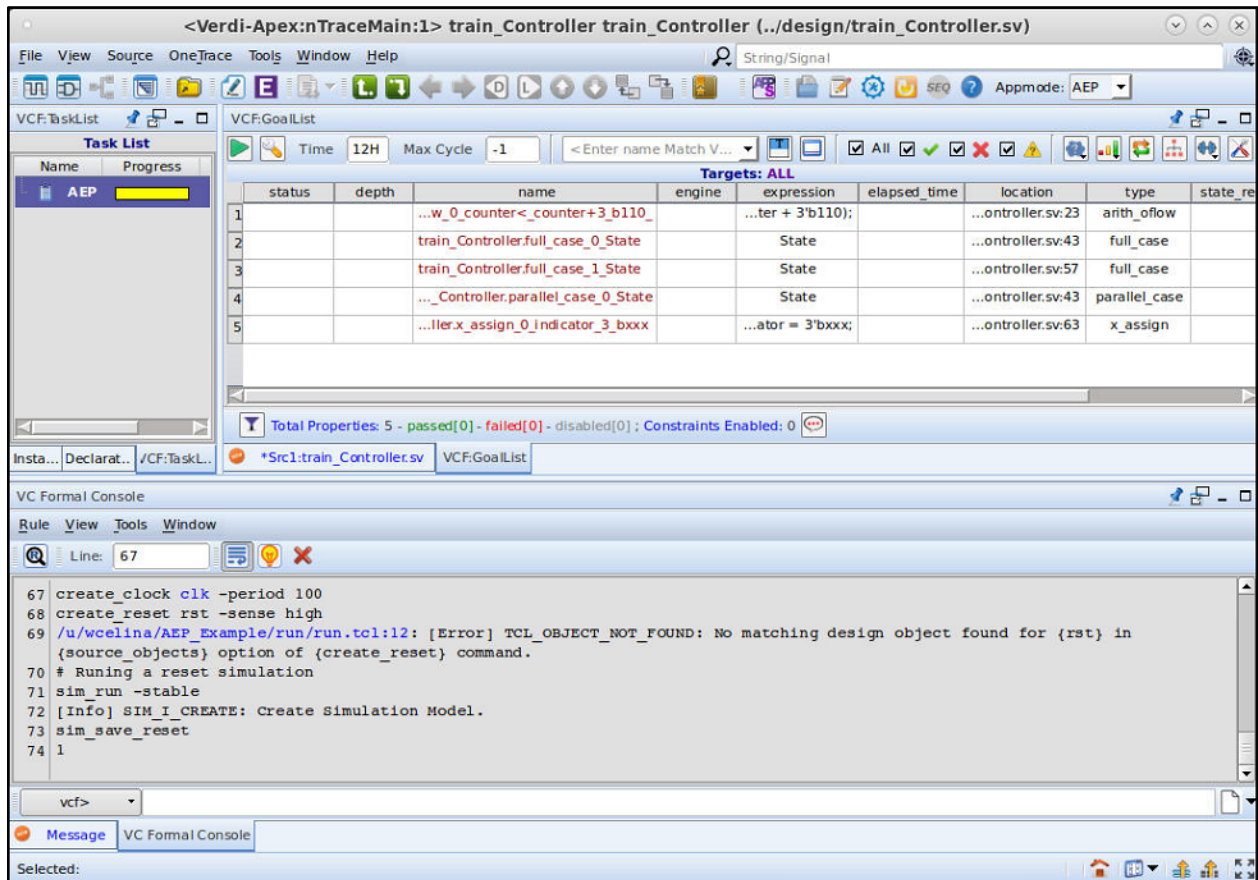


Figure 3.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the “VCF:GoalList” tab.

3.1 Detecting Errors

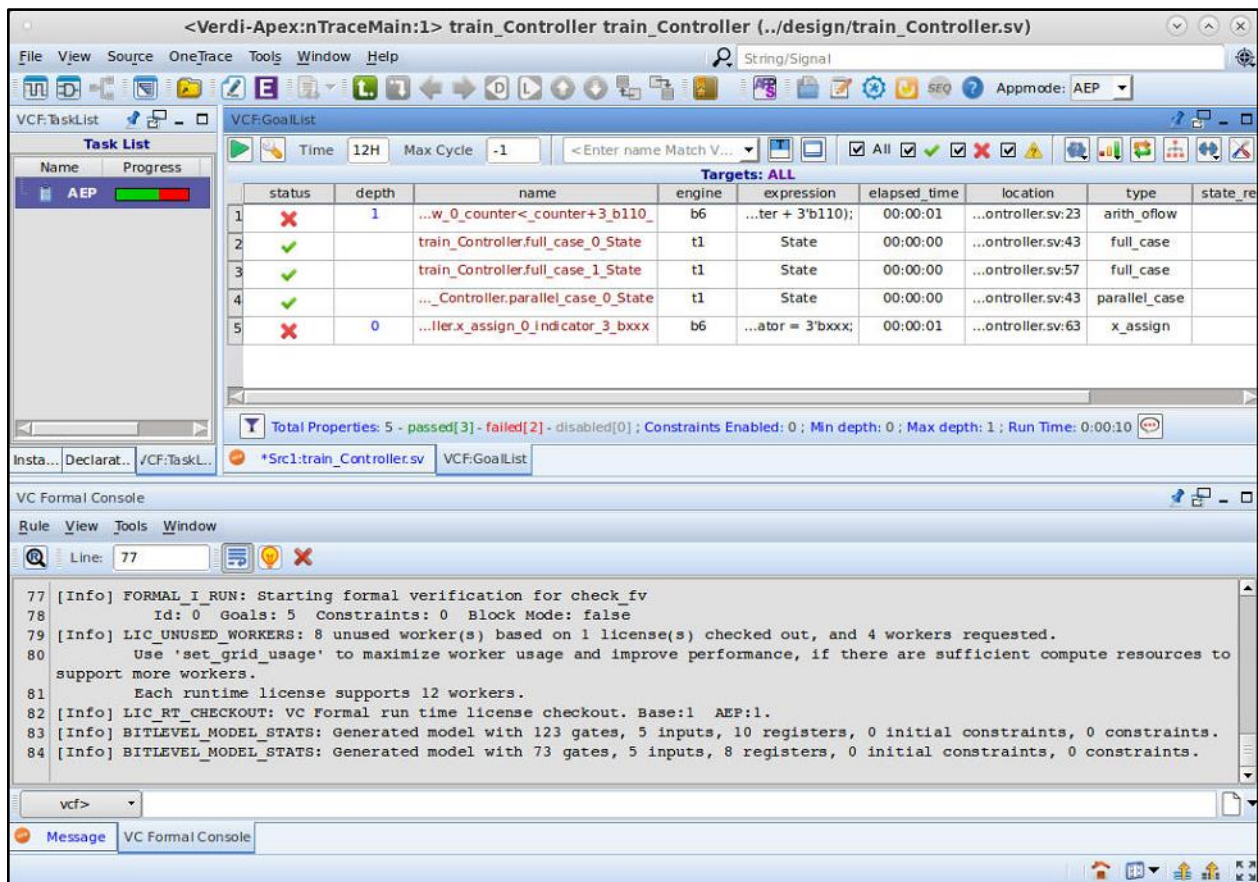


Figure 3.1.1. Screen after running TCL script and the status given = ✗. (This screenshot was taken from the VC Formal GUI)

In Figure 3.1.1 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 3.1.2. Results of the analyzed script. (This screenshot was taken from the VC Formal GUI)

3.2 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 3.1.1*. You should then see a generated waveform as shown below:

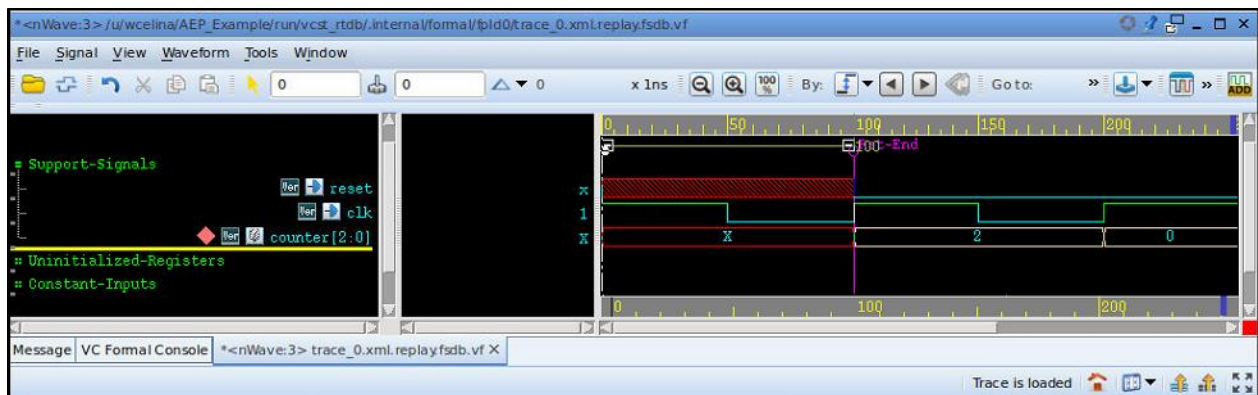
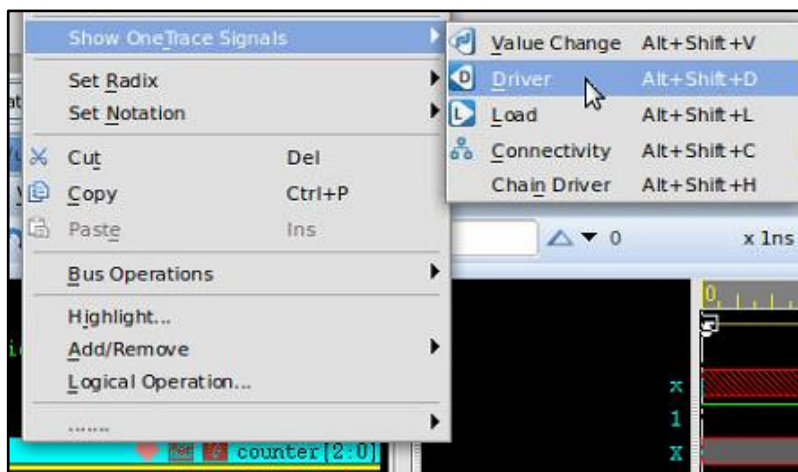


Figure 3.2.1. Examining the failed AEP check in our design. (This screenshot was taken from the VC Formal GUI)

On the left in *Figure 3.2.1* 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.



3.2.2. Pop-up from right-clicking on a signal. (This screenshot was taken from the VC Formal GUI)

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.

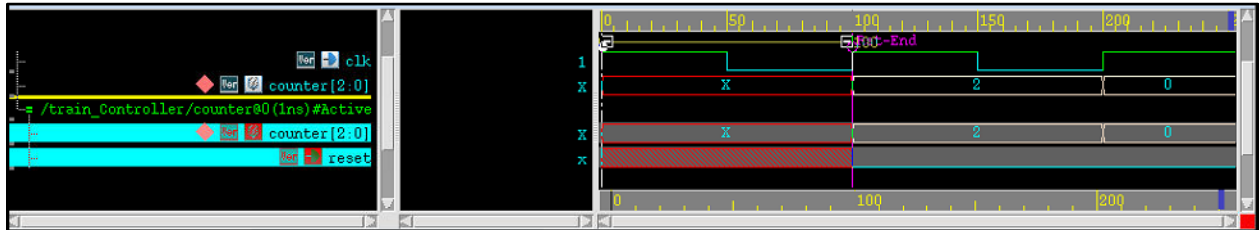


Figure 3.2.3. Tracing the source to find the discrepancy. After one trace. (This screenshot was taken from the VC Formal GUI)

3.3 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”.

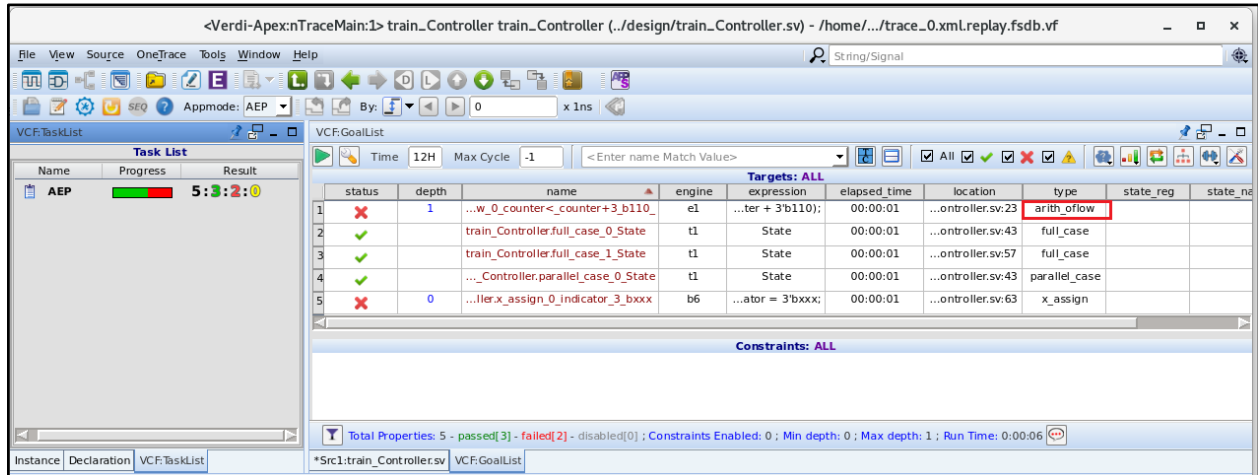


Figure 3.3.1. Under “type”, double-click on “arith_oflow”. (This screenshot was taken from the VC Formal GUI)

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.

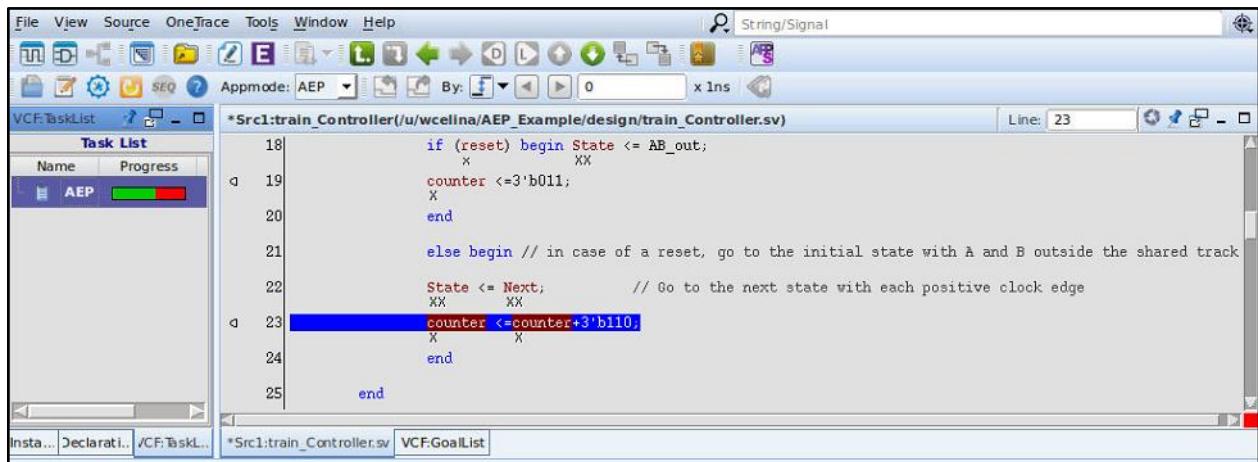


Figure 3.3.2. Identified errors within the Design file. (This screenshot was taken from the VC Formal GUI)

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


```

*Src1:train_Controller(/u/wcelina/AEP_Example/design/train_Controller.sv)
10 logic counter;
11
12 // Enumerate the states. Here I am using on hotkey for state encoding
13 enum logic [4:0] {AB_out=5'b00001, A_on2=5'b00010, B_on2=5'b00100,
14                  A_stopped=5'b01000, B_stopped=5'b10000} State, Next;
15
16 //State Register
17 always_ff @(posedge clk) begin //The reset signal is not here since it's a synchronous reset
18     if (reset) begin State <= AB_out;
19         counter <= 3'b000;
20     end
21     else begin // in case of a reset, go to the initial state with A and B outside the shared track
22         State <= Next; // Go to the next state with each positive clock edge
23         counter <= counter+3'b001;

```

Figure 3.3.3. Altered design file to fix the error on lines 10, 19, and 23 (compare with Figure 1.2.1). (This screenshot was taken from the VC Formal GUI)

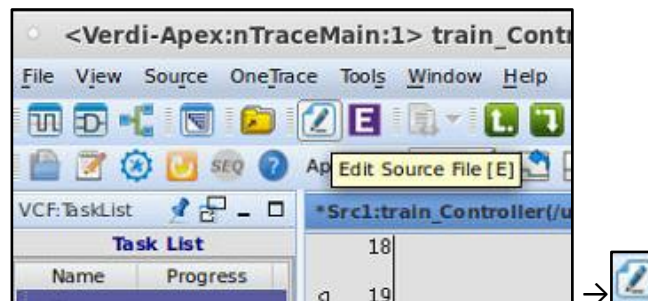


Figure 3.3.4. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes. (This screenshot was taken from the VC Formal GUI)

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

VCF:GoalList

Time 12H Max Cycle -1 < Enter name Match V... [Icons]

Targets: ALL

	status	depth	name	engine	expression	elapsed_time	location	type	state_re
1	✗	1	...w_0_counter<_counter+3_b110_	b6	...ter + 3'b110);	00:00:01	...ontroller.sv:23	arith_oflow	
2	✓		train_Controller.full_case_0_State	t1	State	00:00:00	...ontroller.sv:43	full_case	
3	✓		train_Controller.full_case_1_State	t1	State	00:00:00	...ontroller.sv:57	full_case	
4	✓		..._Controller.parallel_case_0_State	t1	State	00:00:00	...ontroller.sv:43	parallel_case	
5	✗	0	...ller.x_assign_0_indicator_3_bxxx	b6	...ator = 3'bxxx;	00:00:01	...ontroller.sv:63	x_assign	

Total Properties: 5 - passed[3] - failed[2] - disabled[0]; Constraints Enabled: 0; Min depth: 0; Max depth: 1; Run Time: 0:00:10

*Src1:train_Controller.sv VCF:GoalList <nEditor:4> train_Controller.sv x

Figure 3.3.5. Targeting the next error to resolve. (This is a screenshot from VC Formal)

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.

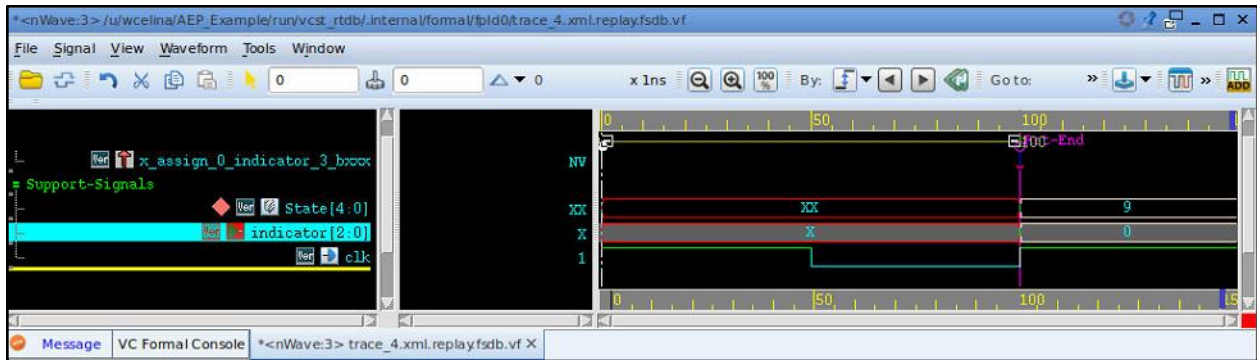


Figure 3.3.6. Looking into `x_assign` waveform. (This screenshot was taken from the VC Formal GUI)

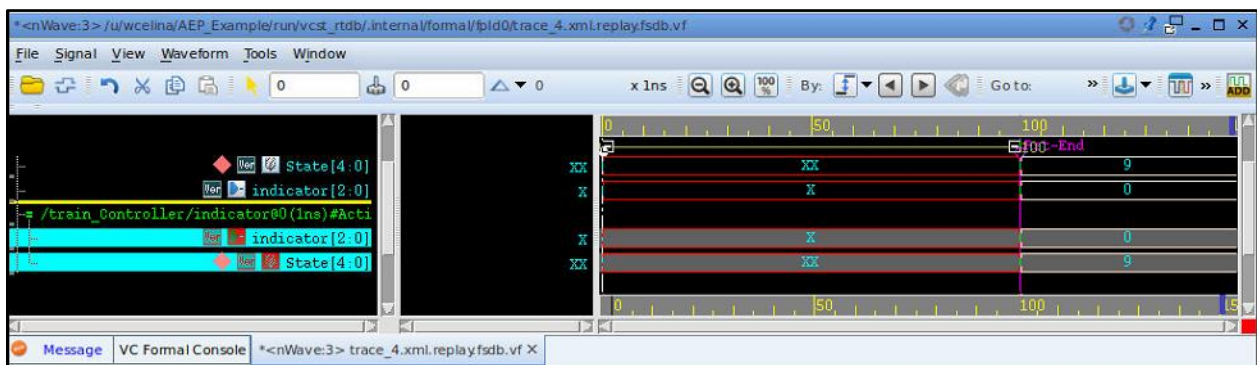


Figure 3.3.7. The waveform after tracing “`indicator[2:0]`” once. (This screenshot was taken from the VC Formal GUI)

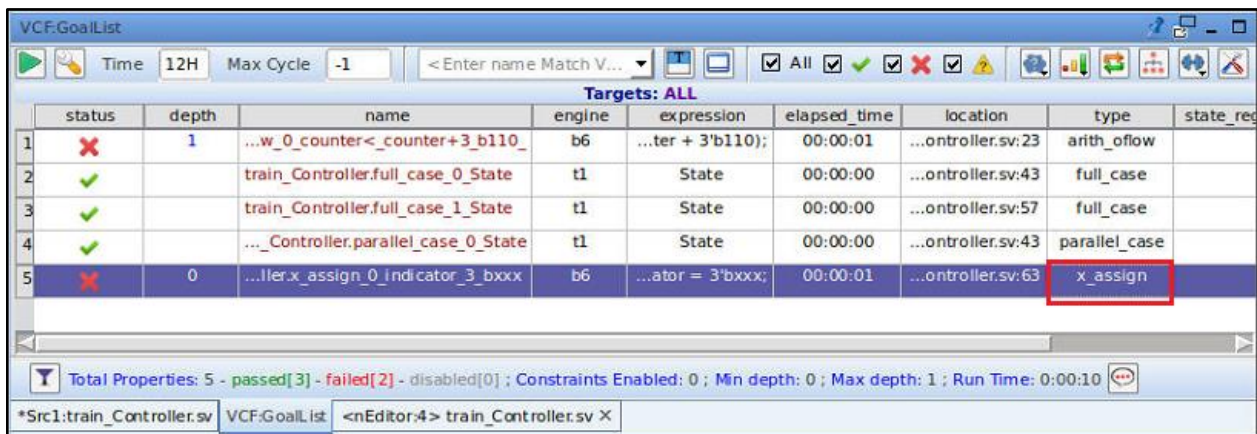


Figure 3.3.8. Double-clicking on the error type. (This screenshot was taken from the VC Formal GUI)

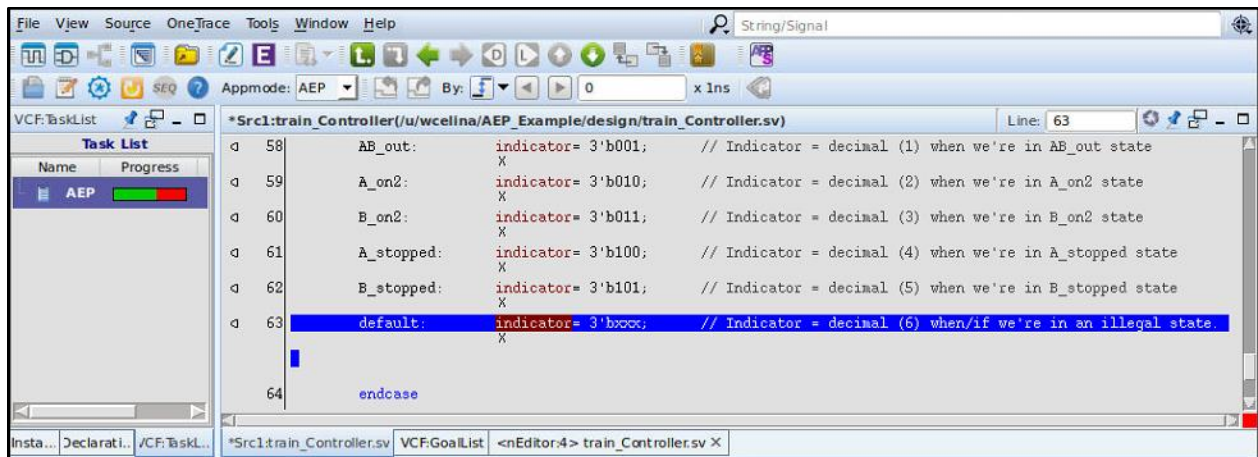


Figure 3.3.9. Identified errors within the design file. (This screenshot was taken from the VC Formal GUI)

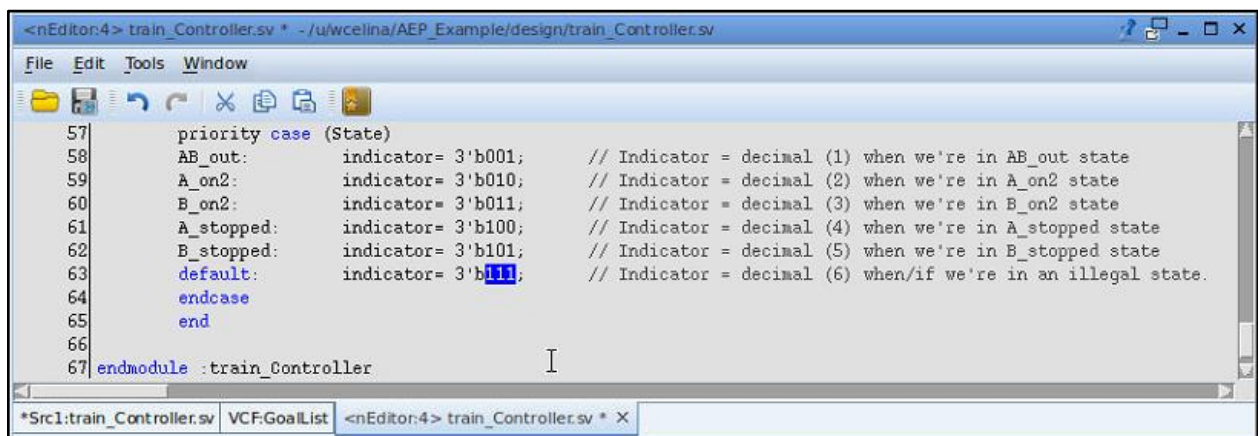


Figure 3.3.10. Saving changes made to the design file. (This screenshot was taken from the VC Formal GUI)

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

3.4 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 3.4.1. Location of the restart button in VC Formal window. (This screenshot was taken from the VC Formal GUI)

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

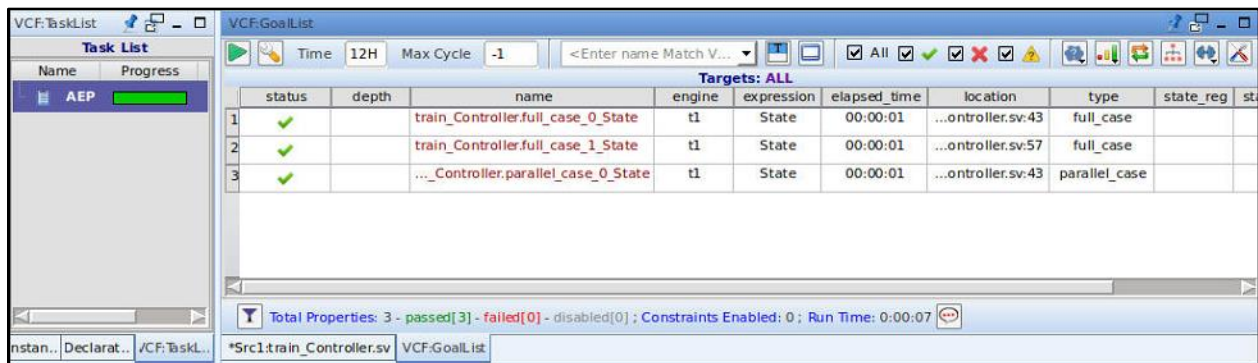


Figure 3.4.1. Results after fixing errors and restarting VC Formal and reloading TCL script. (This screenshot was taken from the VC Formal GUI)

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 <u>must match at least one case</u> item at every cycle.
Pragma Check Unique Case	-aep parallel_case/unique_case	Generated from parallel case synthesis directive. Case expression <u>matches only one case</u> item at every cycle.
Floating bus	-aep floating_bus	<u>Only for tri-state drivers.</u> Verifies at least one driver is active at all times.
Multiple driver bus	-aep multi_driver	<u>Only for tri-state drivers.</u> 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. AEP App Property Types. Identifiers and descriptions of all analyses offered.

Synopsys® VC Formal Tutorial

Formal Coverage Analyzer (FCA)

Version 1.1 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of FCA	4
1.2 Design Files.....	5
1.3 TCL File	6
2. Application Setup.....	8
2.1 Invoking VC Formal GUI	9
2.2 Invoking VC Formal Along with TCL File:.....	11
3. Application Usage	12
3.1 Detecting Errors	14
3.2 Generating Waveforms	17
3.3 Resolving Errors	18
3.4 Restarting VC Formal	20
Appendix.....	21

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 “FCA_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 FCA analysis for us.

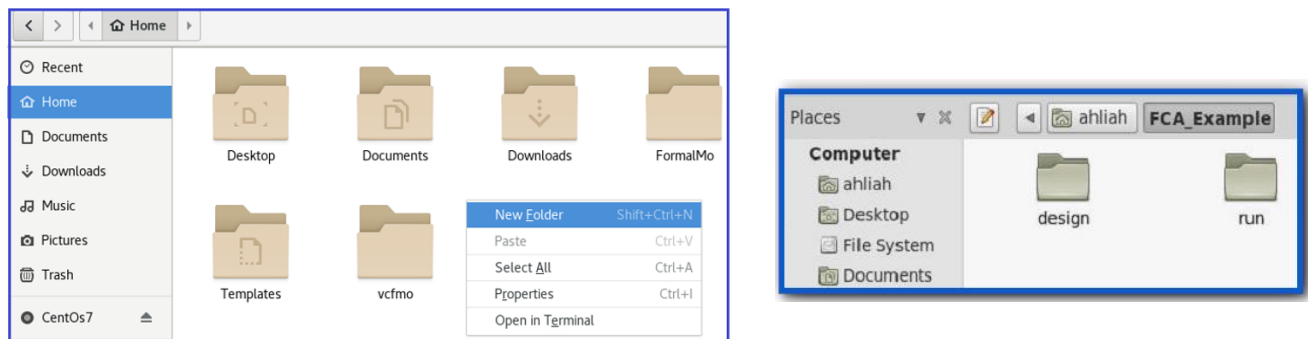
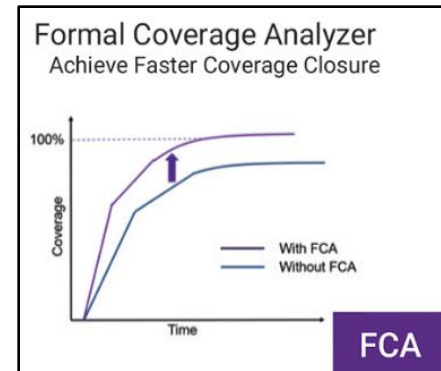


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of FCA

Code coverage closure is when a design reaches all of its target goals, and often is very difficult to achieve and time consuming, so coverage metrics are commonly used in simulation to measure progress and reachability analysis, such as providing information on what checks have passed and what has not. This provides proof on uncovered points in coverage goals that are indeed unreachable, allowing them to be removed from further analysis to save manual labor.



This FCA figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

VC Formal's Formal Coverage Analyzer application provides:

- Assisted simulation-based verification coverage signoff
 - Leverage the simulation coverage database
 - Perform unreachability analysis on goals not covered by simulation
 - Provide exclusion list for goals uncoverable for review and waive
- Formal property verification coverage signoff
 - Over-constraints - unreachable goals due to constraints
 - Property density - structural coverage of the design code in the COI of all properties
 - Bound depth - identify whether the required design code is covered within specified proof depths
 - Formal core - provides minimal design code that is required for the formal engines to reach the full proof or specific proof depths
 - Fault coverages - provides design code covered by the Formal Testbench Analyzer (FTA) application

The full list of specific coverages and their description provided by the FCA application is listed in *Table 1* in the Appendix and it is recommended you go through it to gain a better understanding of what this app truly provides.

1.2 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.

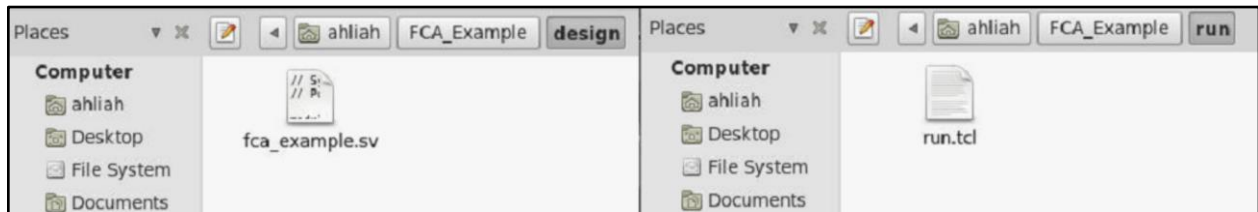


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

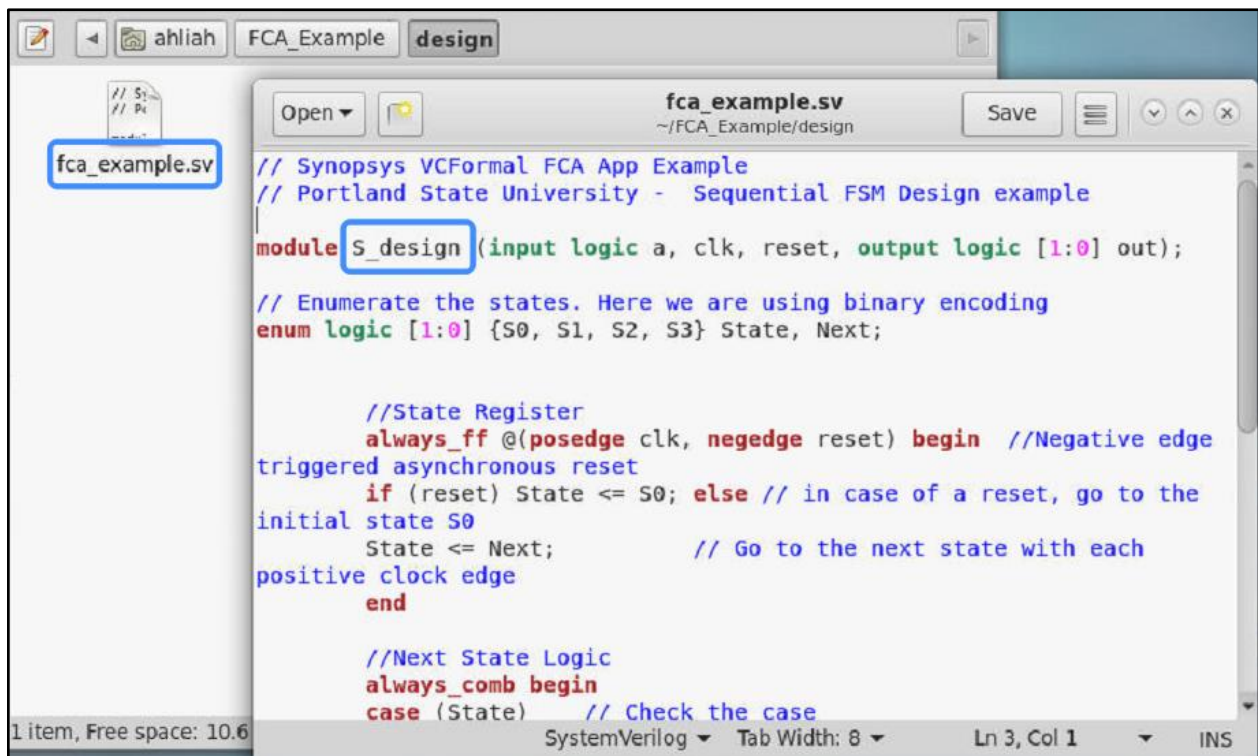


Figure 1.2.2. Example of the design we are using in this tutorial.

In order for VC Formal to map your design, you need to acknowledge and **keep note of your exact module name AND your file name**, as highlighted in Figure 1.2.2 above. This will come in handy when you create your TCL file.

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (full script in *Figure 1.3.2*), which you can use as a template for your functional checks on VC Formal.

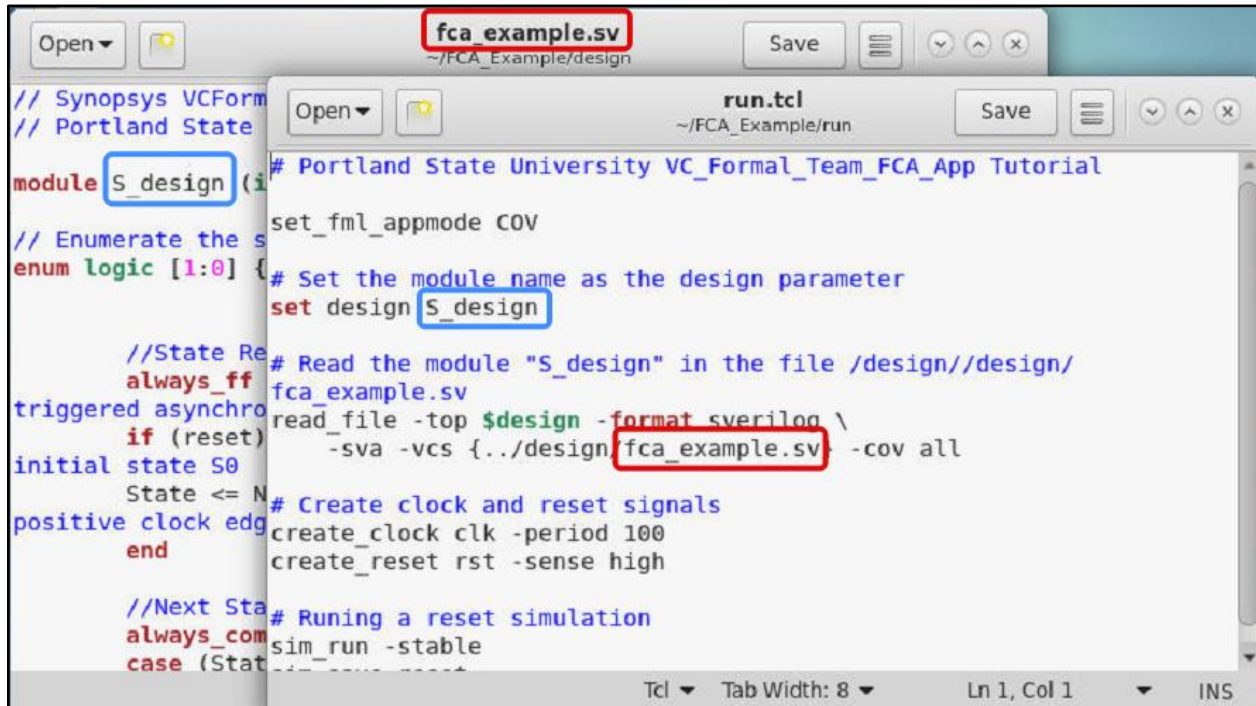


Figure 1.3.1. Difference between the design file name and module name.

In *Figure 1.3.1*, both the Design file and TCL files are shown side by side.

- ❖ The blue highlights are to demonstrate the **module name**.
- ❖ The red highlights are to demonstrate the **design file name**.

These will **NOT** be the same for every user. You will need to keep track of your own module and design file names in order for your TCL file to work correctly when you try to run in VC Formal.

```
run.tcl
~/FCA_Example/run

# Portland State University VC_Formal_Team_FCA_App Tutorial

set_fml_appmode COV 1

# Set the module name as the design parameter
set design S_design 2

# Read the module "S_design" in the file /design//design/fca_example.sv
read_file -top $design -format sverilog \
  -sva -vcs {../design/fca_example.sv} -cov all 4

# Create clock and reset signals 3
create_clock clk -period 100
create_reset rst -sense high

# Running a reset simulation
sim_run -stable
sim_save_reset
```

Figure 1.3.2.. Annotated TCL template file.

- (1) Instruction that sets the appmode to FCA in VC Formal.
- (2) Name of the main module as established in the Design file.
- (3) Design file location so VC Formal knows where to find the file.
- (4) Identifier for FCA coverage metrics. Here we will enable all.

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 1.2.2*.

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

2. 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 FCA 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.

2.1 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

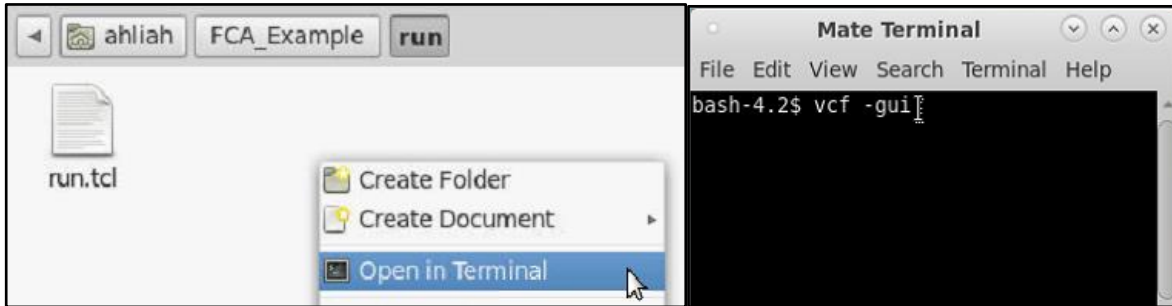


Figure 2.1.1. Invoking VC Formal in the terminal. (This screenshot was taken from the VC Formal GUI)

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 2.1.2* 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: 

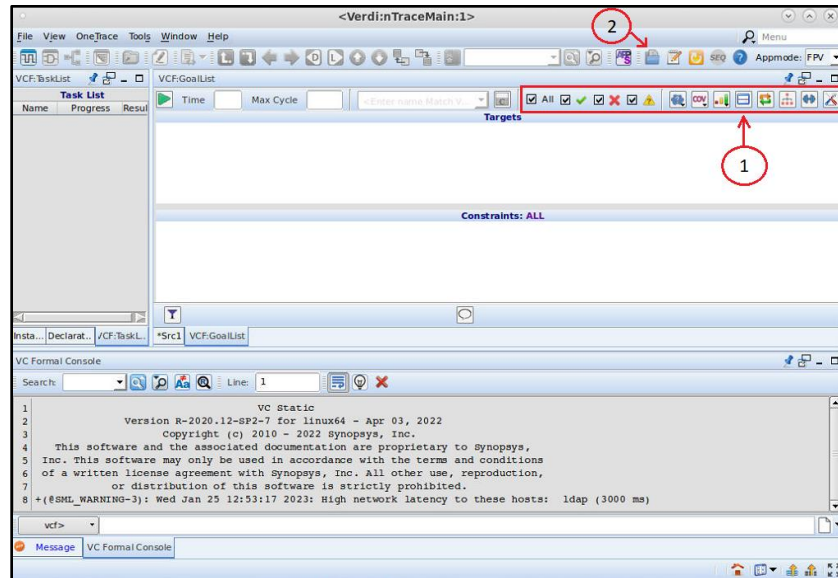


Figure 2.1.2. VC Formal GUI introductory screen. (This screenshot was taken from the VC Formal GUI)

Then load a TCL script by clicking on the  icon **(2)** as shown in *Figure 2.1.2*.

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

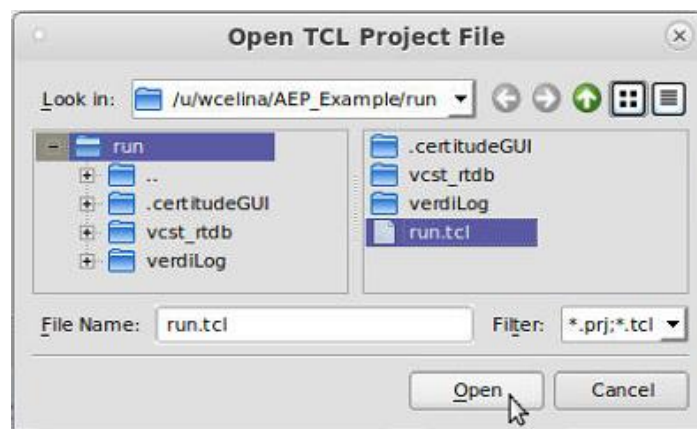


Figure 2.1.3. Selecting TCL file.

2.2 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
```

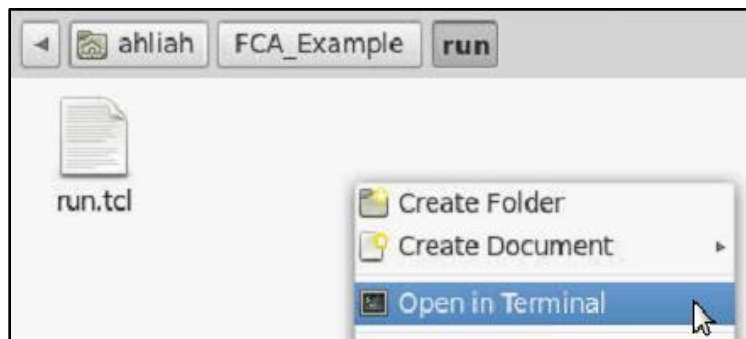


Figure 2.2.1. Opening terminal in the “run” folder.

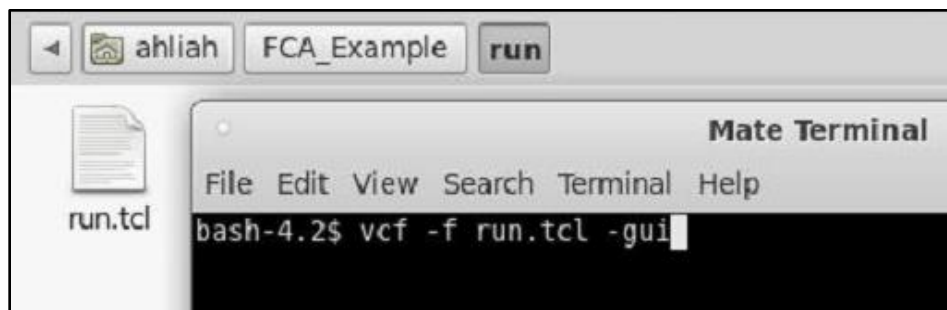


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that's it!

3. Application Usage

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:

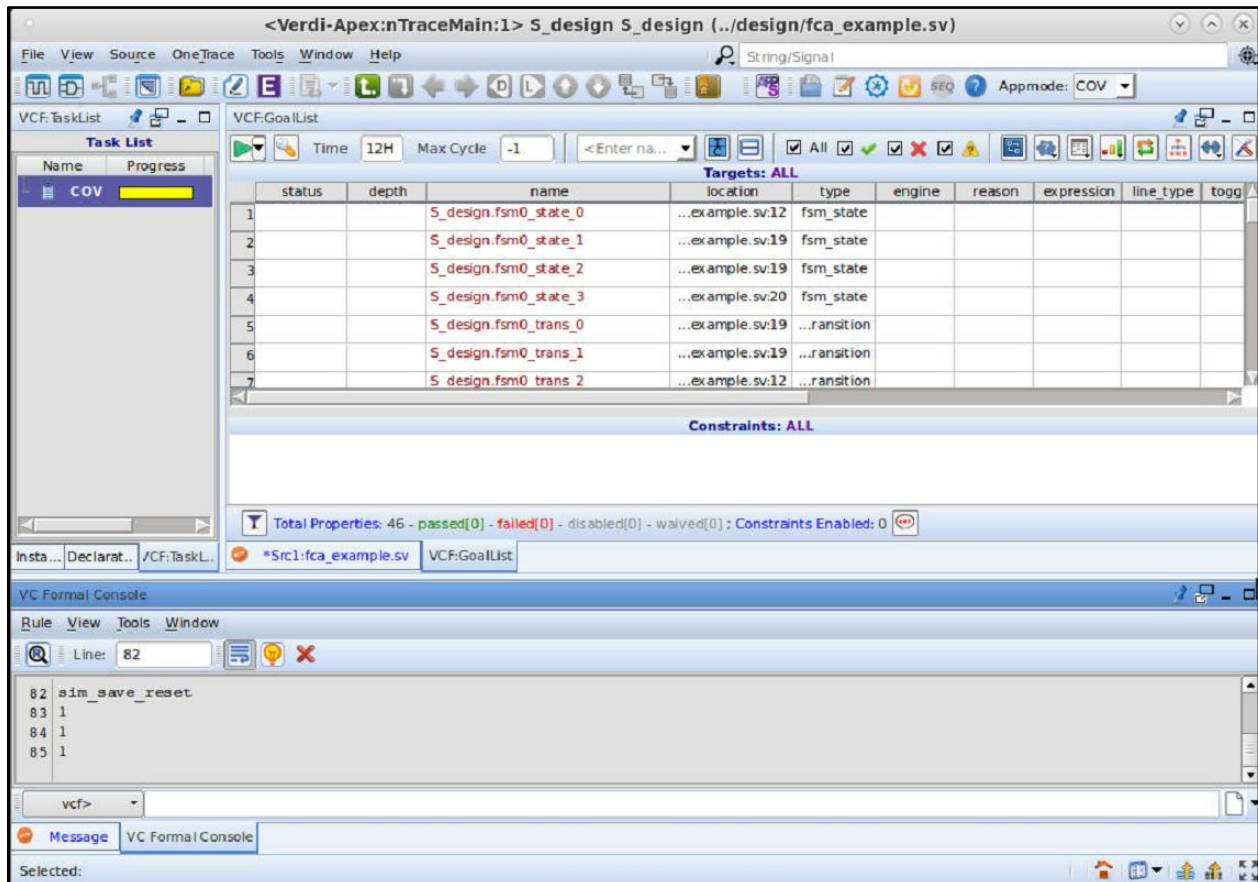


Figure 3.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the “VCF:GoalList” tab.

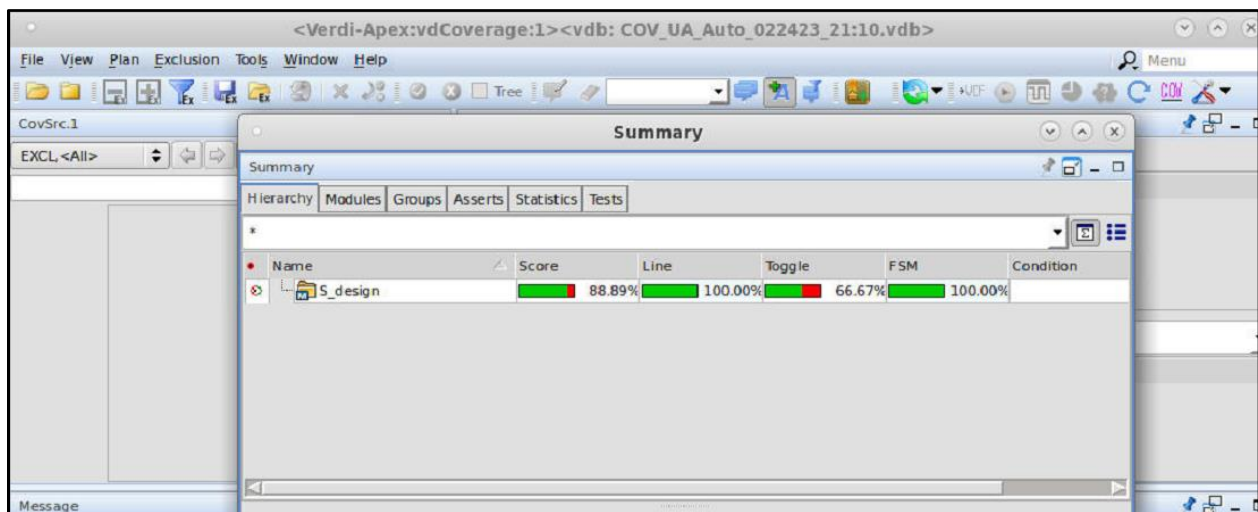


Figure 3.2. Coverage window pop-up after clicking the play button. (This screenshot was taken from the VC Formal GUI)

When you run your design, a separate window will appear. Here we want to take notice of the Hierarchy tab within the Summary window. This will tell us our FCA score for our design; in *Figure 3.2* above, we got 88.99% for our “fca_example” design file. Our score is made up of the following checks: Line, Toggle, and FSM.

For the rest of the tutorial, we will not be using this window so go ahead and minimize it. The following steps will take place in the main VC Formal window.

3.1 Detecting Errors

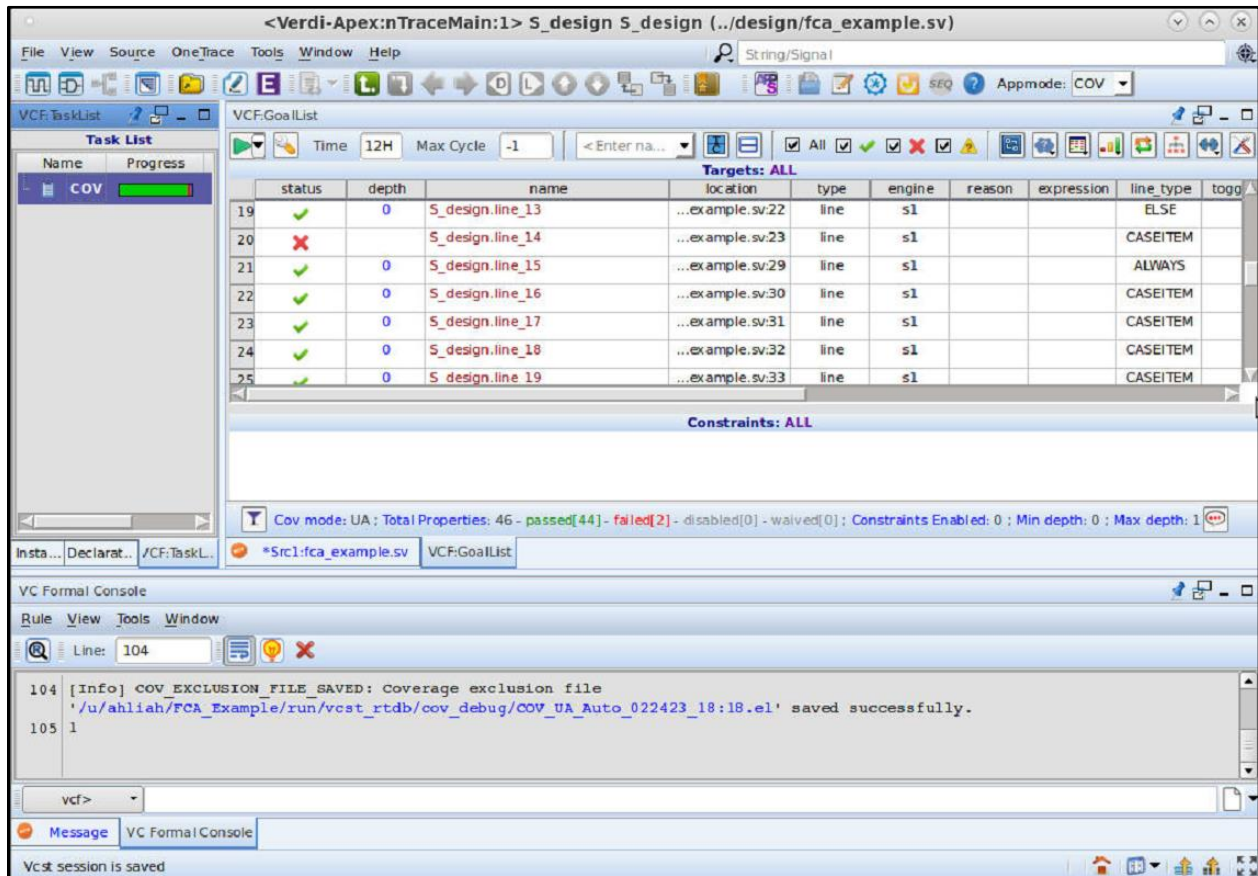


Figure 3.1.1. Screen after running TCL script and the status given = ✗. (This screenshot was taken from the VC Formal GUI)

In Figure 3.1.1 above, we see multiple icons. VC Formal gives this “Passed” status icon for the part of our design that was “covered” by FCA.

We also see one icon; VC Formal gives this “Fail” status icon for the part of our design that was either “uncoverable” or “inconclusive”.

For example, in the figure above, the sign means that there is a “CASEITEM” that did not pass the “Line” coverage check on line 23 (see location column in “Targets” tab).

On the left under “Task List”, we can see that VC Formal was given one task by the FCA 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:

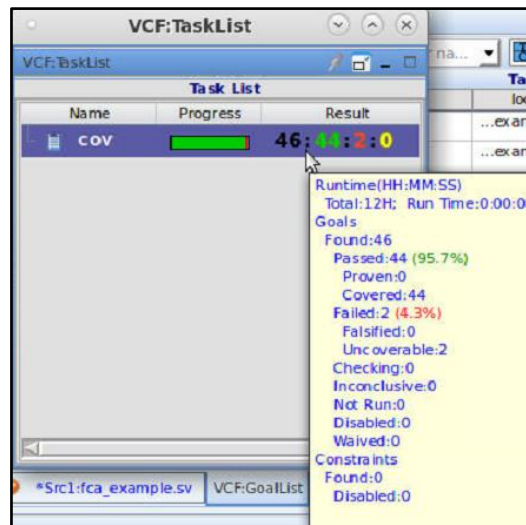


Figure 3.1.2. Results of the analyzed script. (This screenshot was taken from the VC Formal GUI)

This does not align exactly with our FCA score (as shown in *Figure 3.2*) because these results are specifically VC Formal tasks.

We can start by looking at the “Summary Report”. Use the command below in the VC Formal Console window to see the report:

report_fv

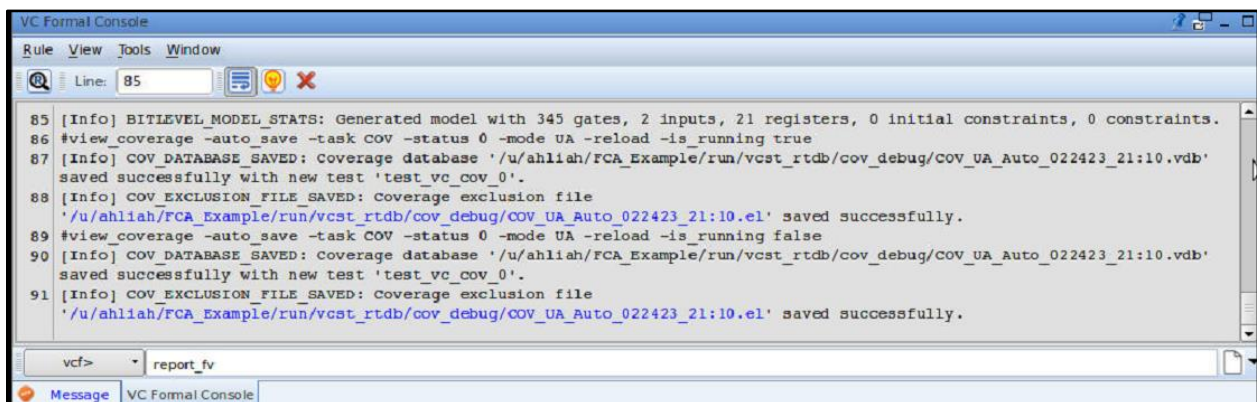


Figure 3.1.3. VC Formal Console window. (This screenshot was taken from the VC Formal GUI)

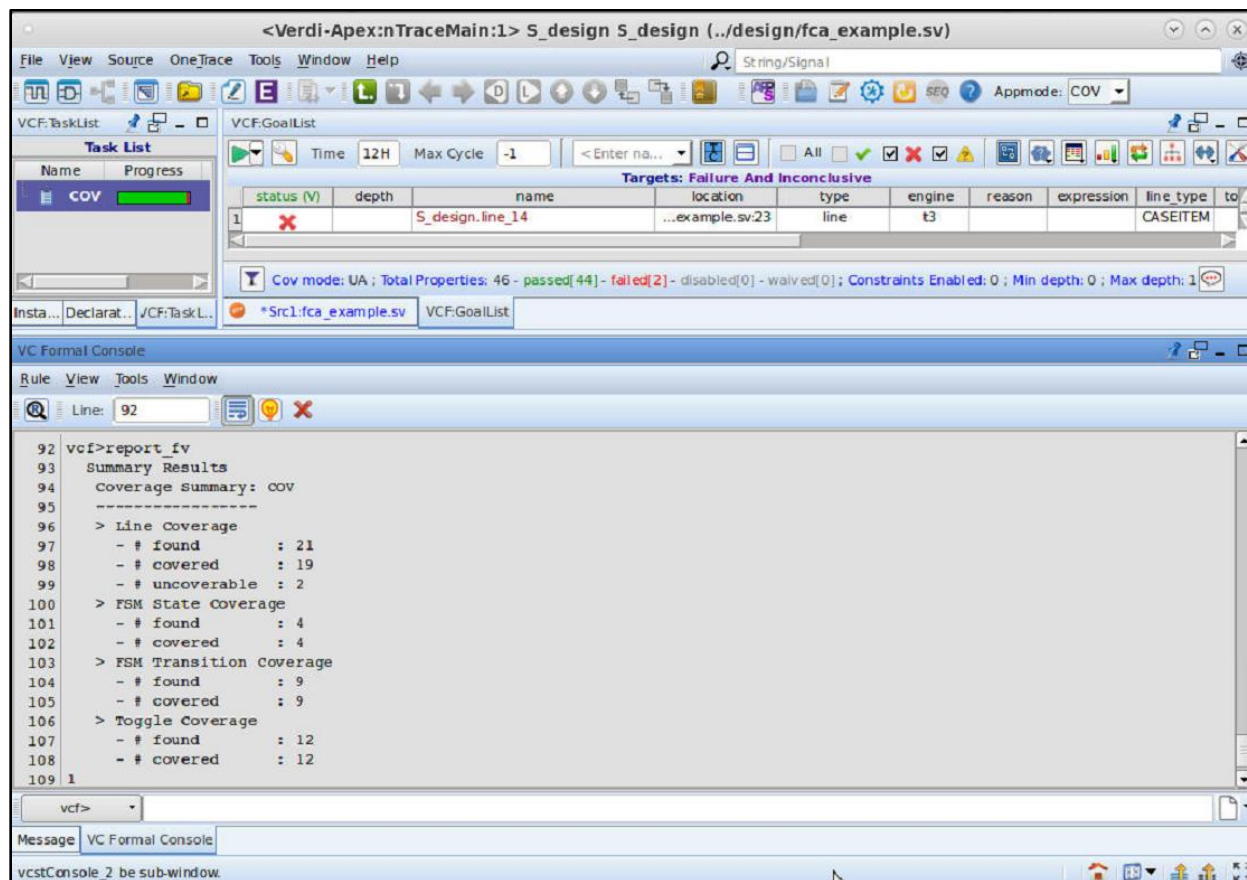


Figure 3.1.4. VC Formal Console showing Summary Results. (This screenshot was taken from the VC Formal GUI)

The “Summary Results” will show our “Coverage Summary” for the types of checks applicable to our design: “Line”, “FSM Stage”, “FSM Transition”, and “Toggle”. Ultimately, we want the number of “found” statuses to match the “covered” statuses.

Observed in Figure 3.1.4, the only time where this does not happen for our design is when FCA checks for Line Coverage, where it states that we have 2 uncoverable areas.

3.2 Generating Waveforms

In previous tutorials, we generated waveforms to examine and source trace our errors. In FCA, or COV mode, we will not be using this method, and waveforms can only be generated for covered conditions - the items with a green check . The overall purpose of this is for the user to be able to review how a condition was covered.

To enable this feature, add this line to your tcl file:

```
set_fml_var fml_cov_gen_trace on
```

Make sure to save, restart/reload VC Formal after any changes to the script, and when you double-click on a check , the waveform should be generated for that covered task.

```
# Portland State University VC_Formal_Team_FCA_App Tutorial
set_fml_appmode COV

# Set the module name as the design parameter
set design S_design

# Read the module "S_design" in the file /design/fca_example.sv
read_file -top $design -format sverilog \
    -sva -vcs {../design/fca_example.sv} -cov all

# Create clock and reset signals
create_clock clk -period 100
create_reset rst -sense high

# Running a reset simulation
sim_run -stable
sim_save_reset

# Enable trace for covered goals
set_fml_var fml_cov_gen_trace on
```

Figure 3.2.1. Adding command to tcl script to enable trace for covered goals.

3.3 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 “CASEITEM”.

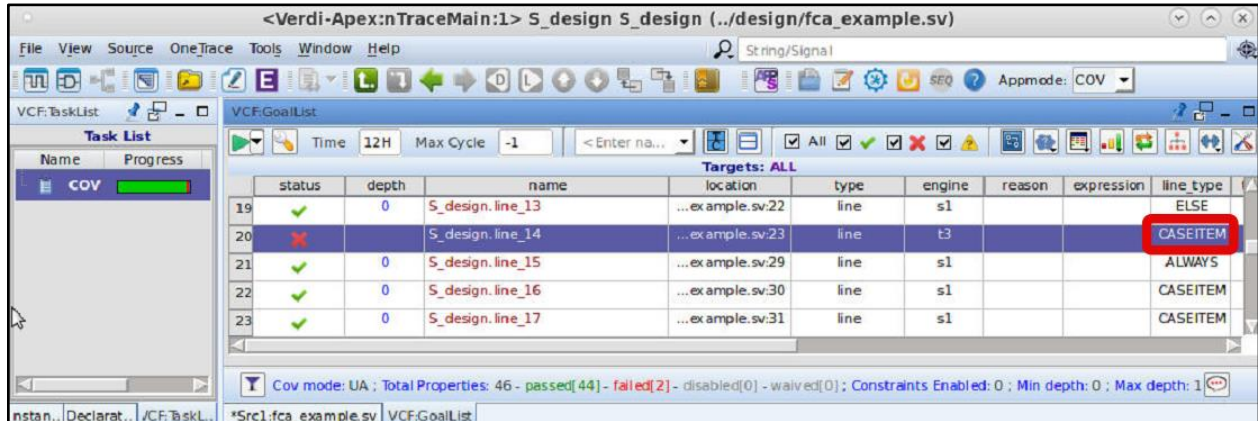


Figure 3.3.1. Under line_type, double-click on CASEITEM. (This screenshot was taken from the VC Formal GUI)

We are then taken to the part of the code that is causing this fault “CASEITEM”, with the operation highlighted in blue, and the signal highlighted in red.

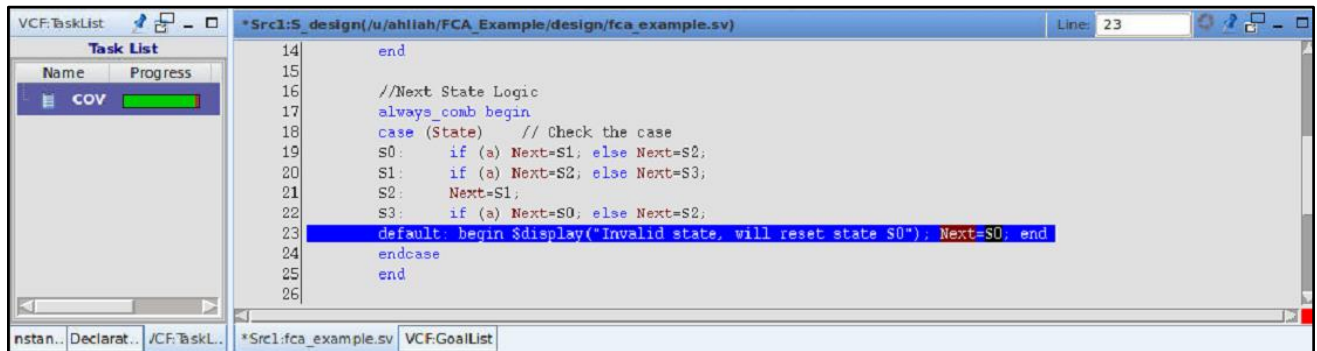


Figure 3.3.2. Identified errors within the Design file. (This screenshot was taken from the VC Formal GUI)

Looking at the FCA app Coverage Metrics table in the Appendix, the “Line Coverage” check looks at which lines of code are not exercised. In my code, the default case was not utilized during the simulation.

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

→ Comment out line 23

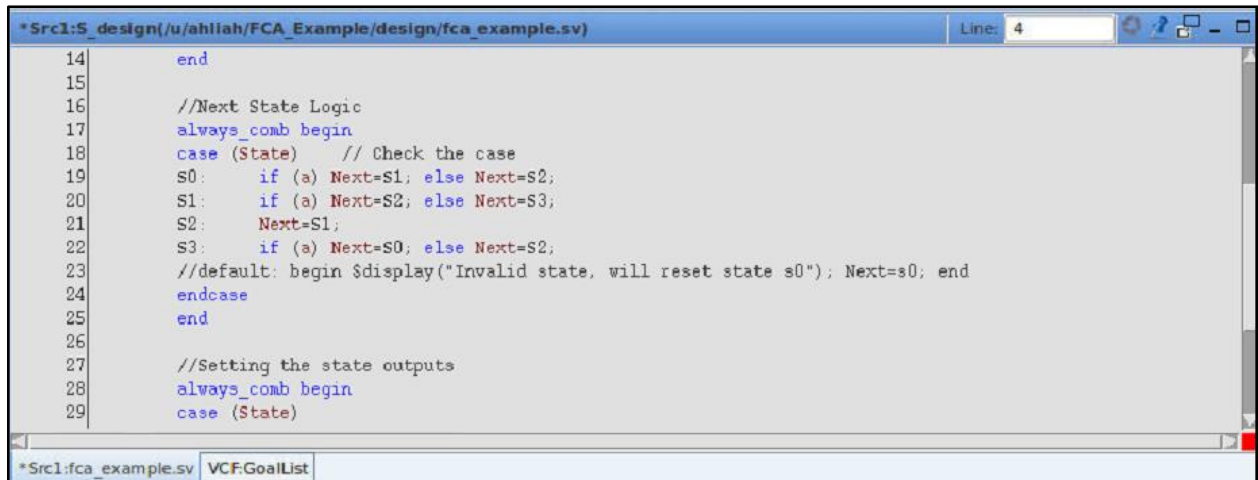


Figure 3.3.3. Altered design file to fix the error on line 23 (compare with Figure 3.3.2). (This screenshot was taken from the VC Formal GUI)

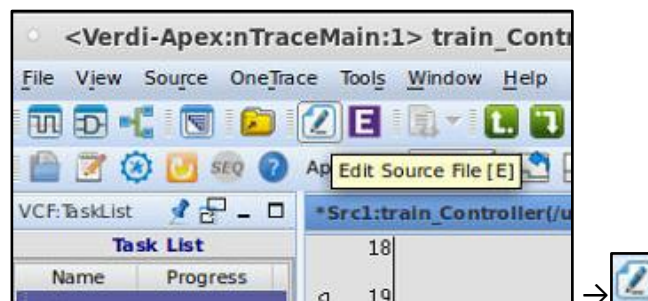


Figure 3.3.4. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes. (This screenshot was taken from the VC Formal GUI)

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

3.4 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 3.4.1. Location of the restart button in VC Formal window. (This screenshot was taken from the VC Formal GUI)

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

	status
5	✓
6	✓
7	✓

Figure 3.4.2. Results after fixing errors and restarting VC Formal and reloading TCL script.

Appendix

Coverage Type	Description
Line Coverage	Line coverage is applied to signal and variable assignments in HDL code. Shows which lines of code are exercised and which ones are not, by your testbench during a simulation run. A zero execution count, pin points a line of code that has not been exercised. This could be the source of a potential design error.
Condition Coverage	Monitors whether certain expressions and sub-expressions in your code evaluate to true or false. <i>Note: By condition coverage is not supported inside functions and tasks. You must use the -cm_cond tf option in VCS to enable the same.</i>
Toggle Coverage	Monitors value changes on signal bits in the design. When toggle coverage reaches 100%, that means every bit of monitored signals has changed its value from 0 to 1 and from 1 to 0. Missing transitions of values provide definitive conclusions about inactive elements and unexercised portions of the design. <i>Note: Toggle coverage is not supported inside functions and tasks.</i>
Branch Coverage	Monitors the execution of conditional statements such as if/else and case statements, and the ternary operator ?: in a design. By default, branch coverage analyzes which alternative of these conditional statements is covered. <i>Note: This is needed either -cov branch of set_fml_var fml_cov_enable_branch_cov true or with -cov all.</i>
FSM Coverage	Identifies a group of statements in the source code to be a finite state machine (FSM) and tracks the states and transitions that occur in the FSM during simulation.
SV Covergroups Coverage	Monitors values, transitions, and crosses for variables and signals. There are a list of limitations regarding SV covergroups; refer to the VC Formal User Guide on Formal Coverage Analyzer Application for more details (pg267).

Table 1. FCA App Coverage Metrics. Identifiers and descriptions of all coverages offered.

Synopsys VC Formal Tutorial

Formal X-Propagation Verification (FXP)

Version 1.2 | 02-June-2023



Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of FXP	4
1.2 Design Files.....	5
1.3 TCL File	6
2. Application Setup.....	7
2.1 Invoking VC Formal GUI	8
2.2 Invoking VC Formal Along with TCL File.....	10
3. Application Usage	11
3.1 Detecting Errors	12
3.2 Source Tracing.....	14
3.3 Resolving Errors	16
3.4 Restarting VC Formal	19
Appendix.....	20

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 “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.

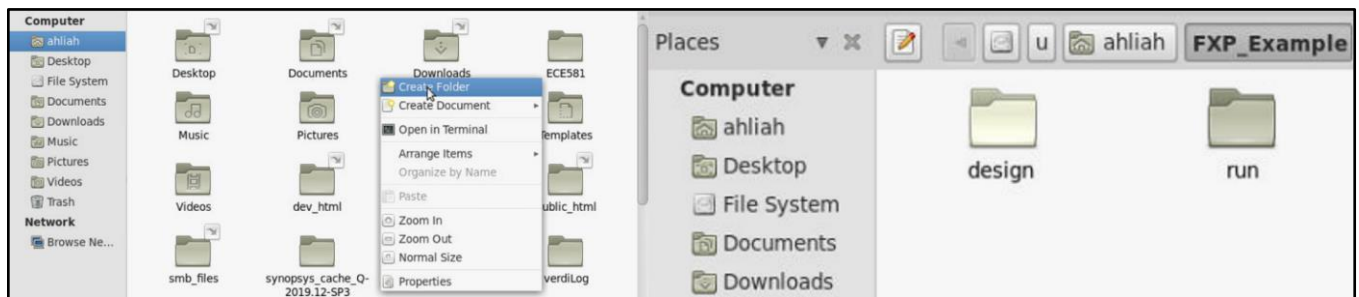
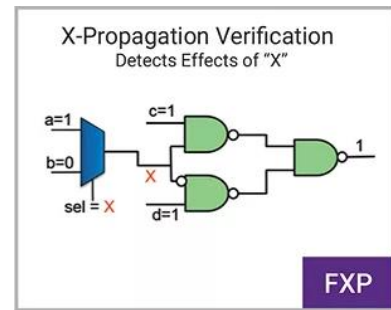


Figure 1.1. Creating folders to organize design and TCL files.

1.1 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.



This FXP figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

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 FXP checking all the following types of 'X' values to generate: (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 FXP checks, it means it is vulnerable and most times the fundamental design is to blame.

1.2 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.

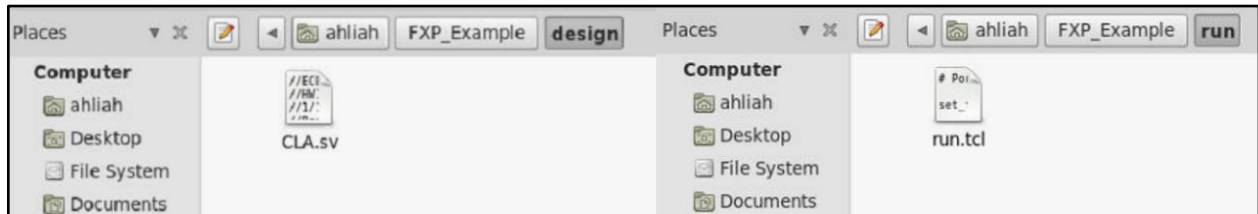


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

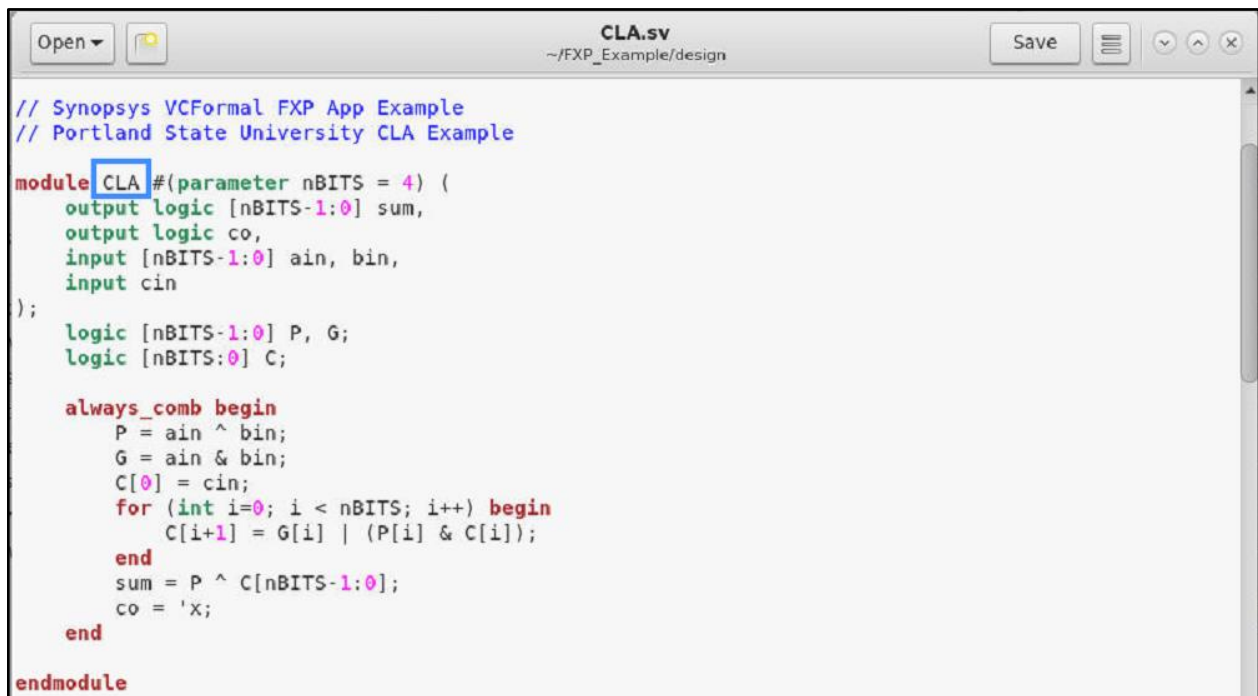


Figure 1.2.2. Example of the design we are using in this tutorial.

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 1.2.2* 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).

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.1*), which you can use as a template for your functional checks on VC Formal.

```
# Portland State University VC_Formal_Team_FXP_App Tutorial
set_fml_appmode FXP 1

# Run -fxp all analysis on module "CLA" in the file /design/CLA.sv
read_file -top CLA -format sverilog -sva -vcs { ../design/CLA.sv} 4

# Automatically Show the rootcause of falsified properties
set_fml_var fxp_compute_rootcause_auto true

# Creating clock and reset signals
create_clock clk -period 100
create_reset resetn -sense low

#Running a reset simulation
sim_run -stable
sim_save_reset

# Run the Formal X-propagation Analysis
fxp_generate 5

# Automatically Show the rootcause of falsified properties
set_fml_var fxp_compute_rootcause_auto true 6
```

Figure 1.3.1. 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 1.2.2*.

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

2. 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.

2.1 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

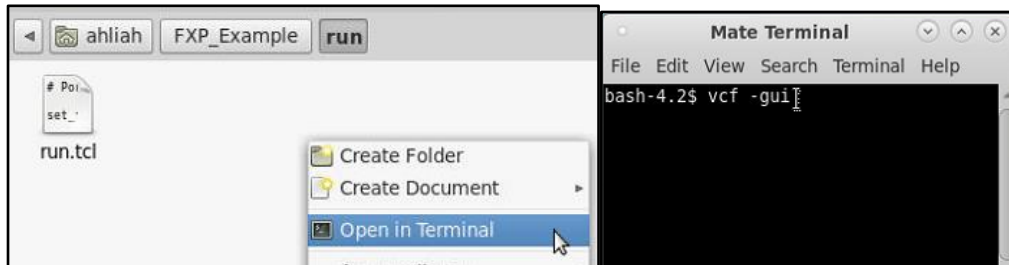


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2* 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:

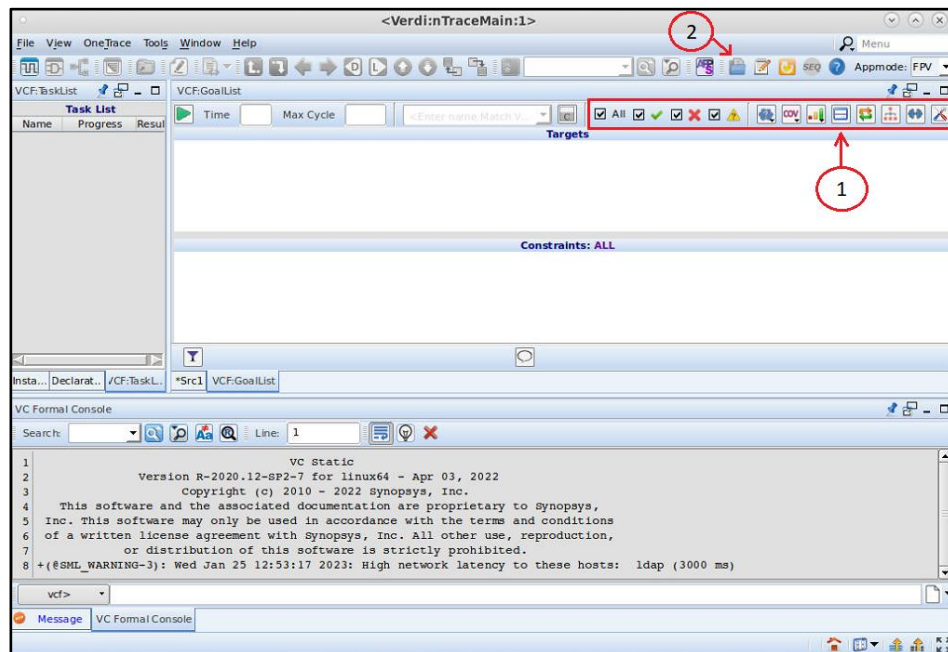


Figure 2.1.2. VC Formal GUI introductory screen. (This screenshot was taken from the VC Formal GUI)

Then load a TCL script by clicking on the  icon **(2)** as shown in *Figure 2.1.2*.

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

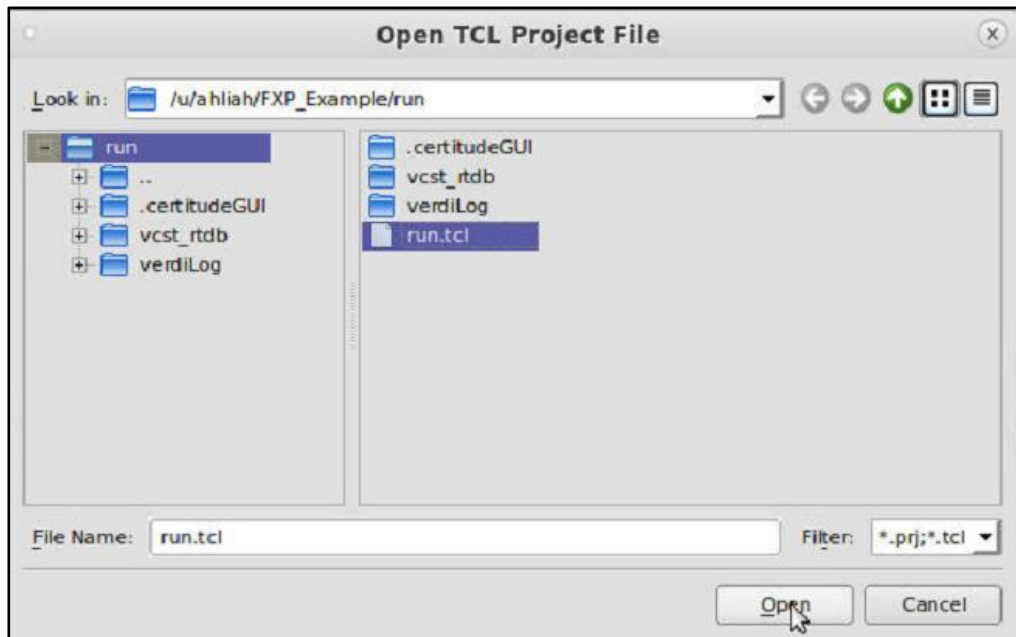


Figure 2.1.3. Selecting TCL file.

2.2 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
```

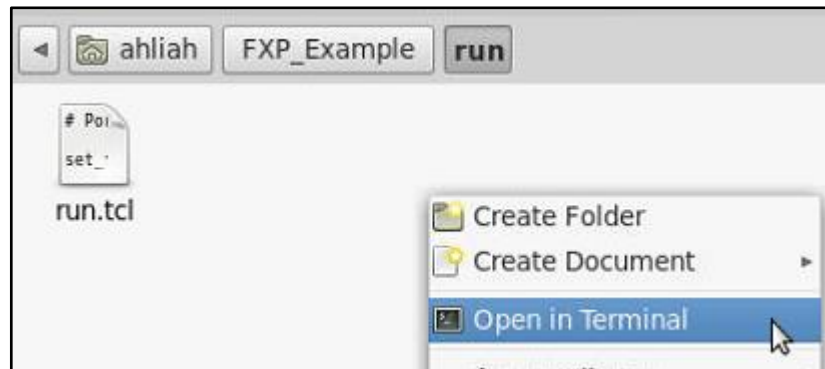


Figure 2.2.1. Opening terminal in the “run” folder.

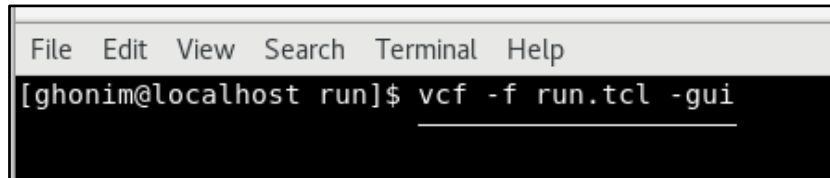


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that's it!

3. Application Usage

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:

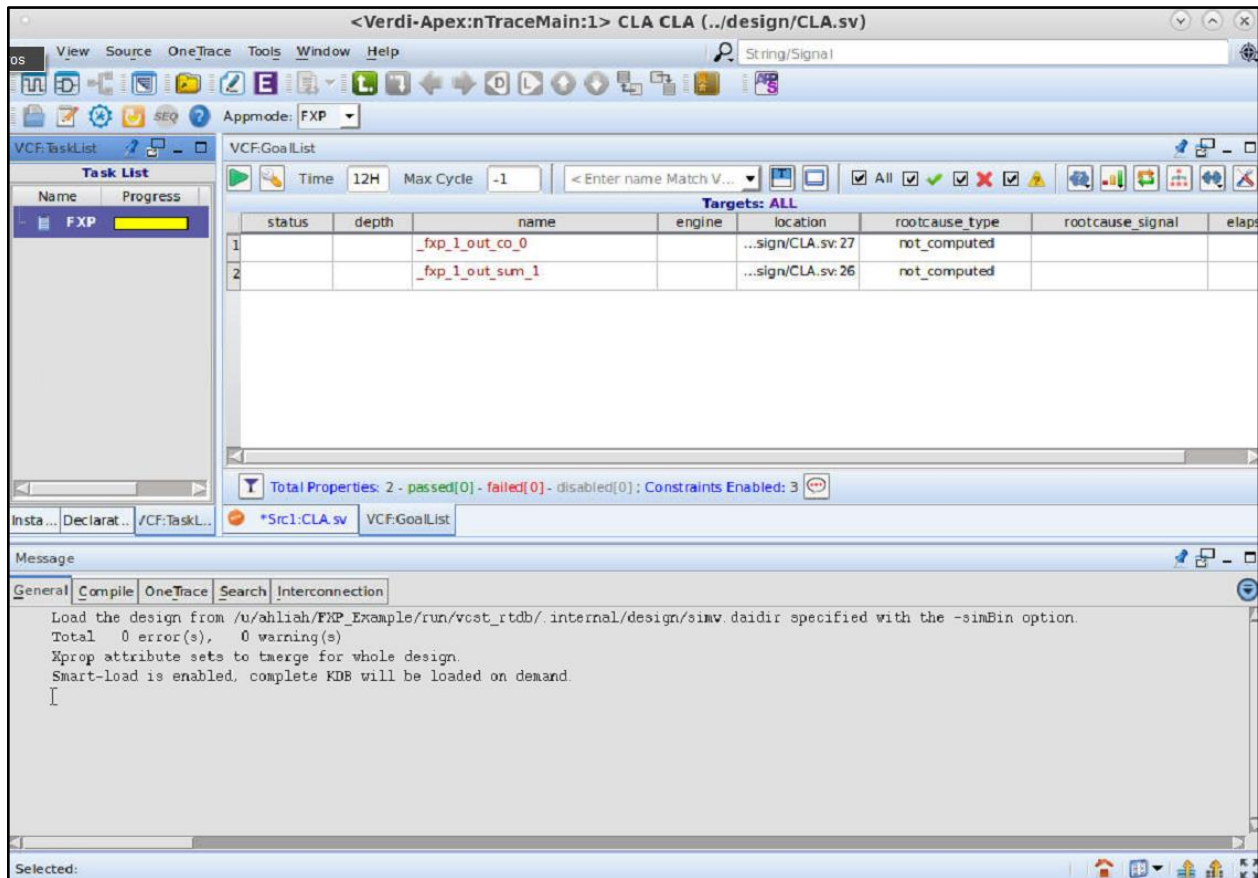


Figure 3.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

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.

3.1 Detecting Errors

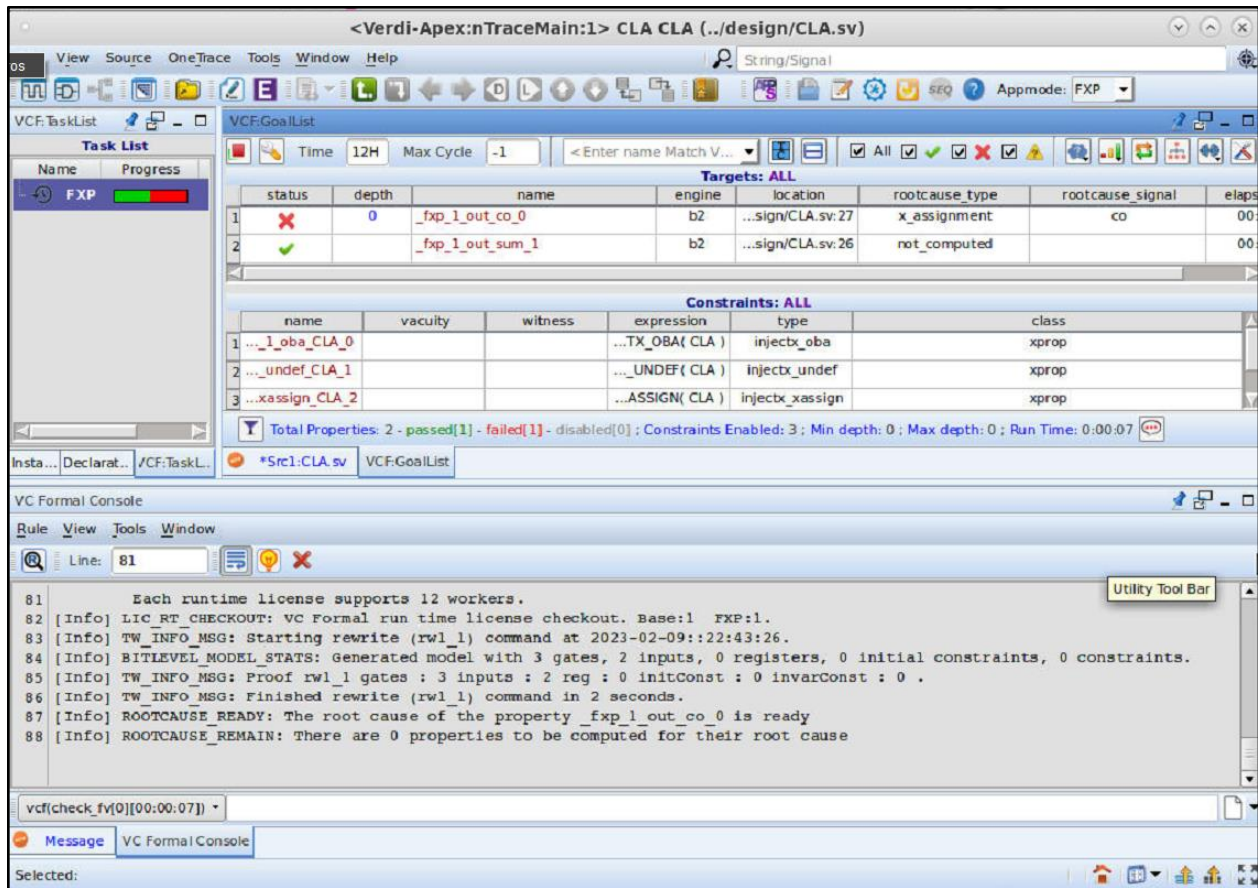


Figure 3.1.1. Screen after running TCL script and the status given = ✗. (This screenshot was taken from the VC Formal GUI)

In Figure 3.1.1 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:

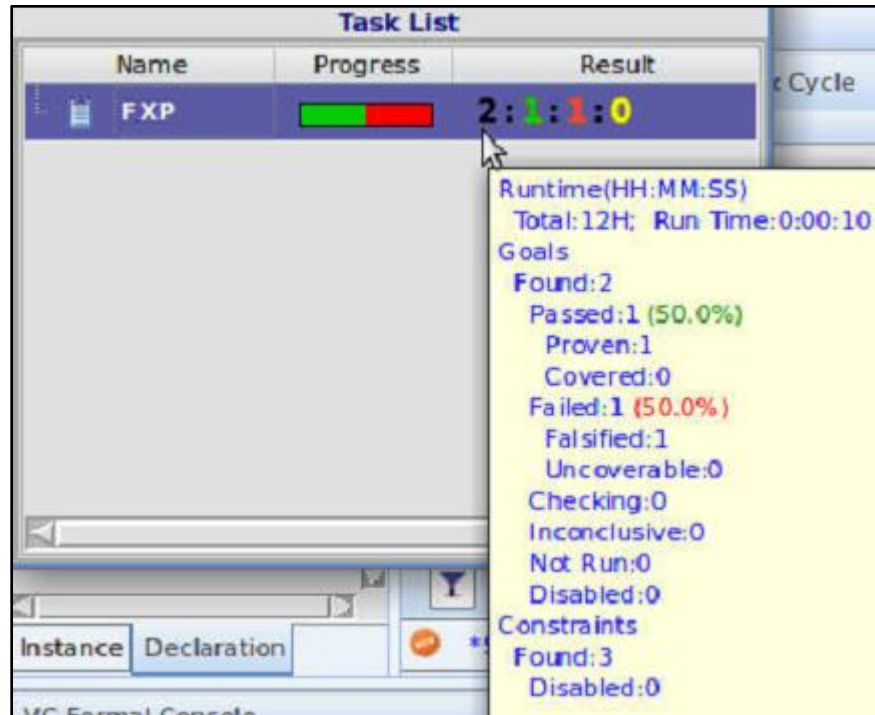


Figure 3.1.2. Results of the analyzed script. (This screenshot was taken from the VC Formal GUI)

3.2 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 3.1.1*. You should then see a generated waveform as shown below:

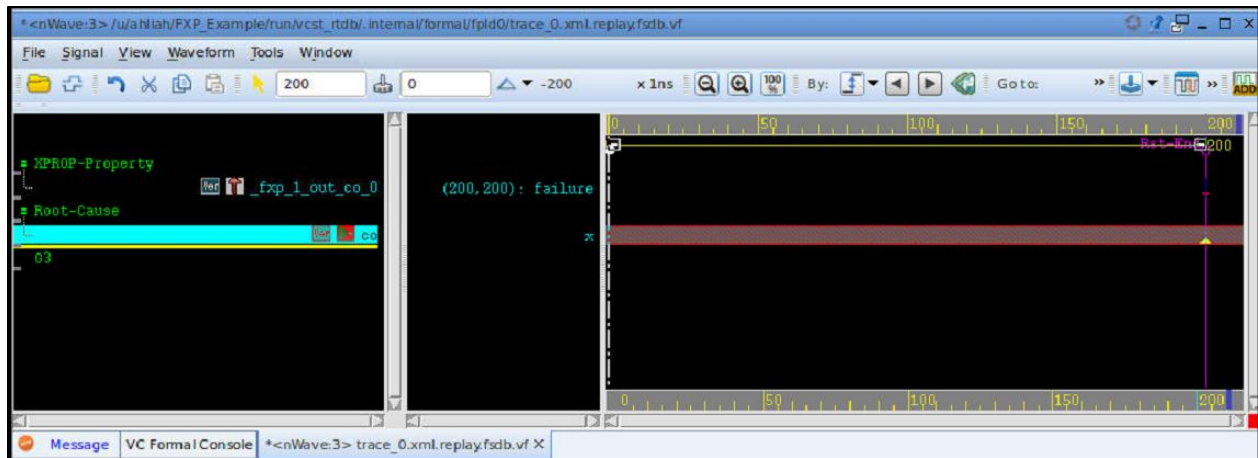


Figure 3.2.1. Examining the failed FXP check in our design. (This screenshot was taken from the VC Formal GUI)

On the left in *Figure 3.2.1* above, we see “Root-Cause -> co” signal in green text. Right-click on it the red signal associated with it.

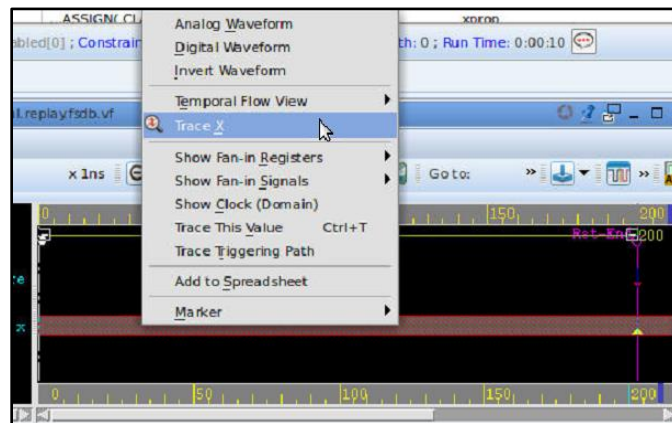


Figure 3.2.2. Pop-up from right-clicking on a signal. (This screenshot was taken from the VC Formal GUI)

As shown in Figure 3.2.2 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.

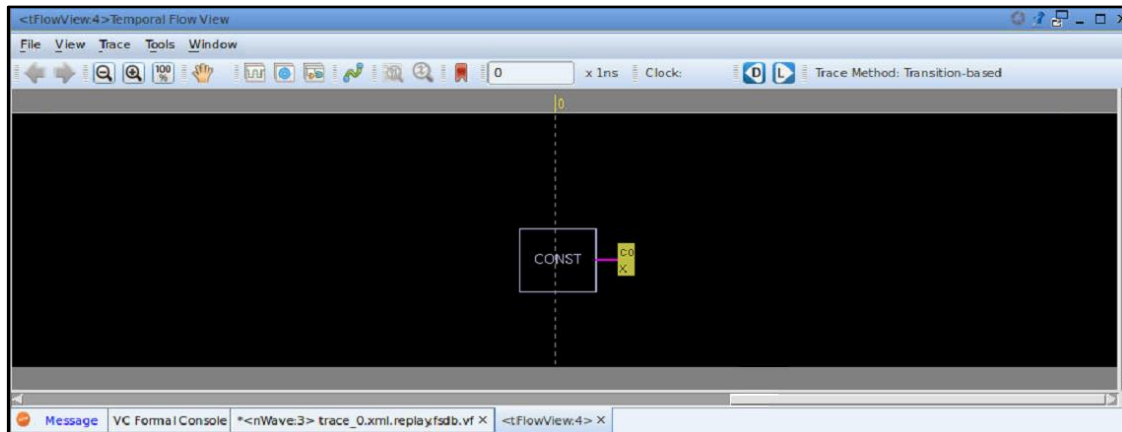


Figure 3.2.3. Generated temporal view for schematic. (This screenshot was taken from the VC Formal GUI)

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 3.2.3). For a better understanding of this temporal view tool, take a look at another example showcasing two latches below:

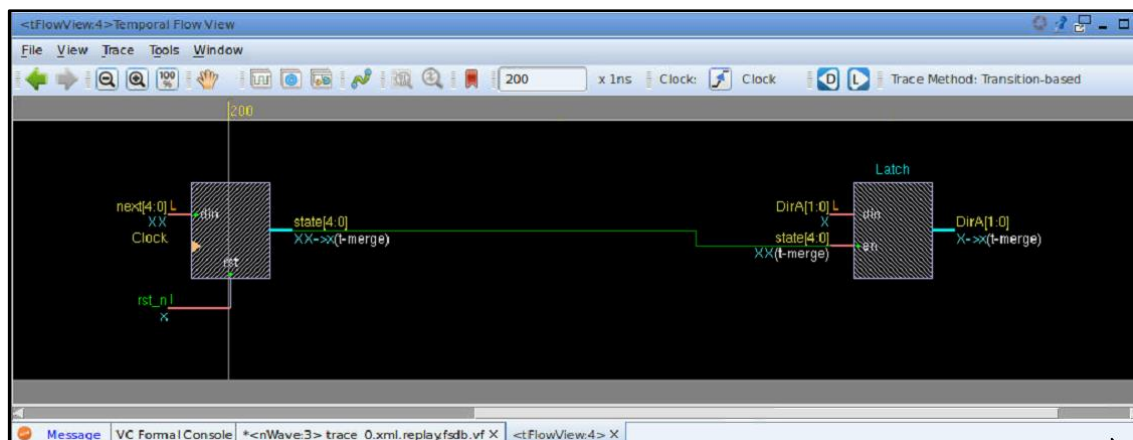


Figure 3.2.4. Generated temporal view for a different schematic. (This screenshot was taken from the VC Formal GUI)

3.3 Resolving Errors

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

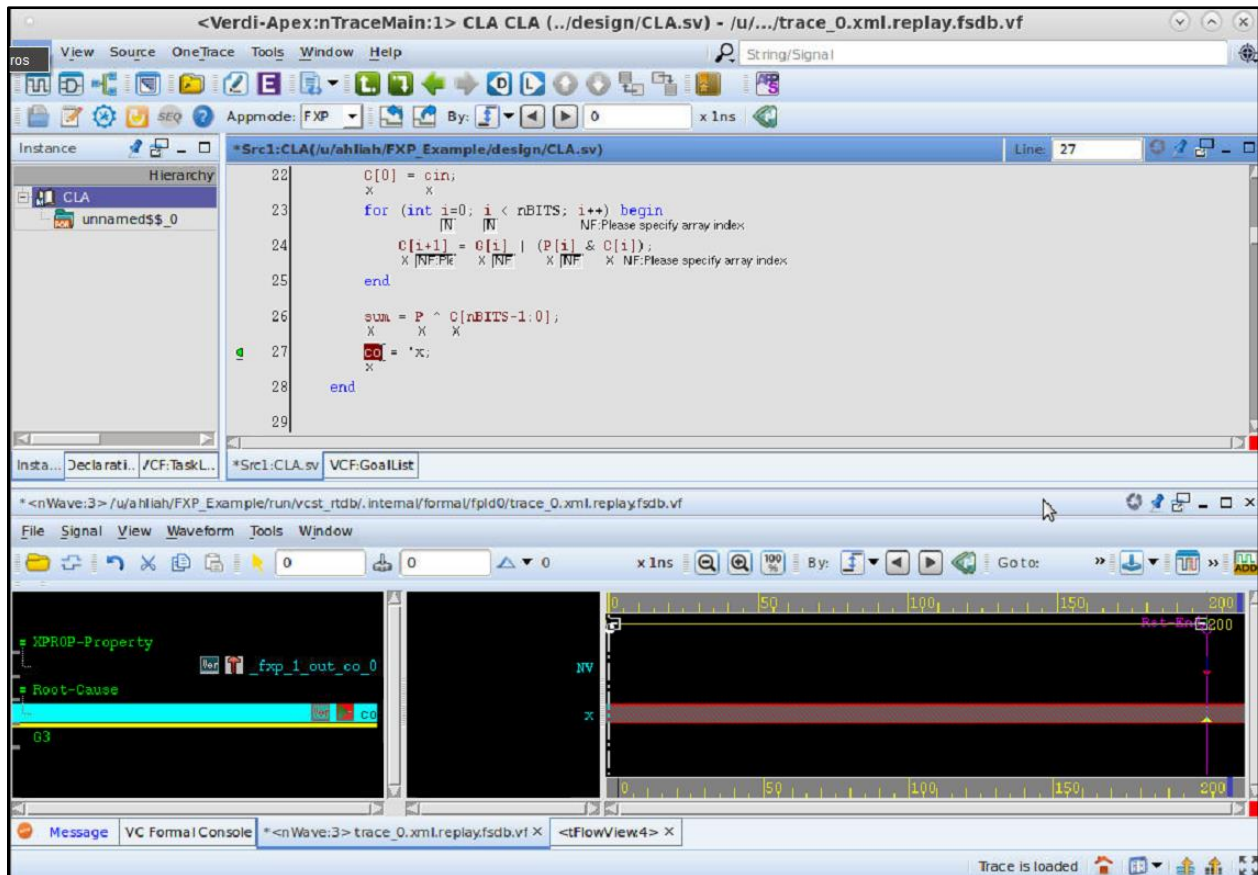


Figure 3.3.1. Identified errors within the Design file. (This screenshot was taken from the VC Formal GUI)

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".

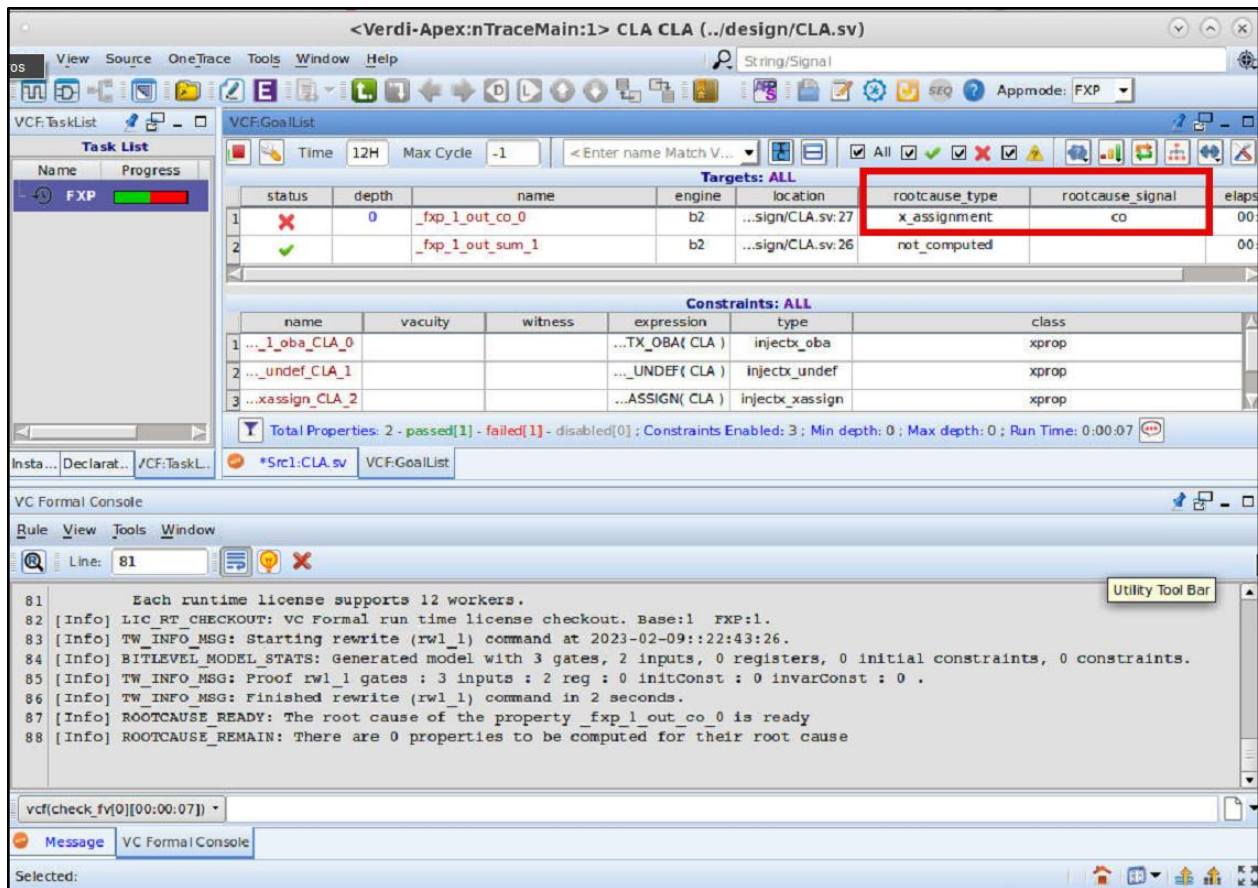


Figure 3.3.2. Under “rootcause_type” and “rootcause_signal” columns. (This screenshot was taken from the VC Formal GUI)

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

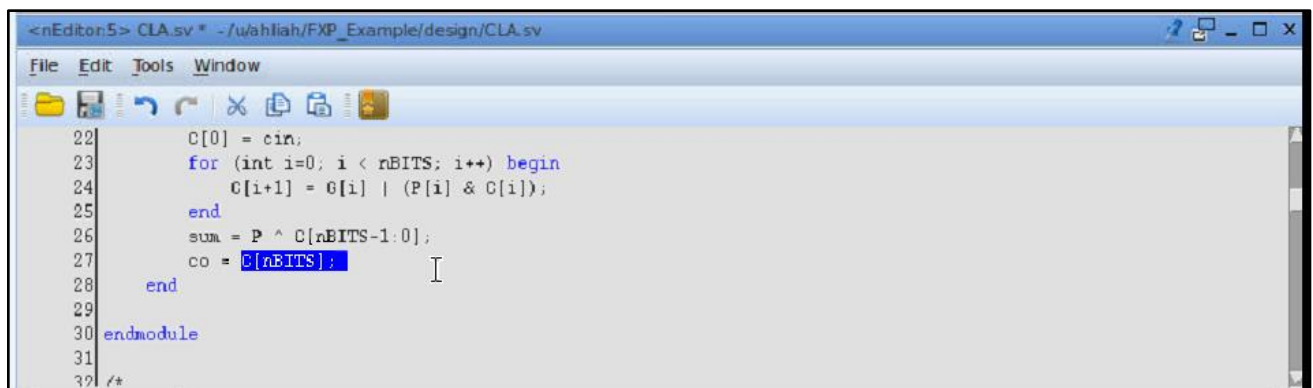


Figure 3.3.3. Altered design file to fix the error on line 27 (compare with Figure 1.2.2). (This screenshot was taken from the VC Formal GUI)

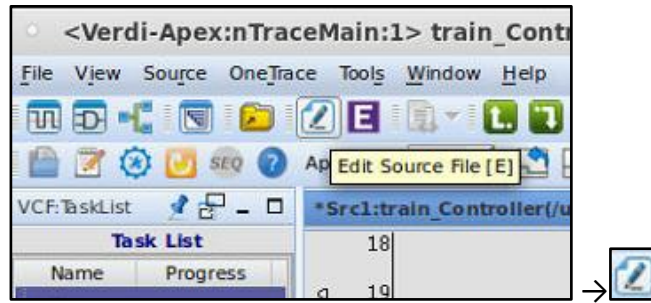


Figure 3.3.4. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes! (This screenshot was taken from the VC Formal GUI)

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

3.4 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 3.4.1. Location of the restart button in VC Formal window. (This screenshot was taken from the VC Formal GUI)

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



Figure 3.4.2. Results after fixing errors and restarting VC Formal and reloading TCL script. (This screenshot was taken from the VC Formal GUI)

Appendix

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

Table 1. FXP Property Types and default options.

Synopsys® VC Formal Tutorial

Connectivity Checking Application (CC)

Version 1.1 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of CC	4
1.2 Design Files.....	5
1.3 TCL File	7
2. Application Setup.....	8
2.1 Invoking VC Formal GUI	9
2.2 Invoking VC Formal Along with TCL File:.....	11
3. Application Usage	12
3.1 Detecting Errors	13
3.2 Debugging Failures.....	15
3.3 Resolving Errors	20
3.4 Restarting VC Formal	23
Appendix.....	24

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 “CC_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 CC analysis for us.

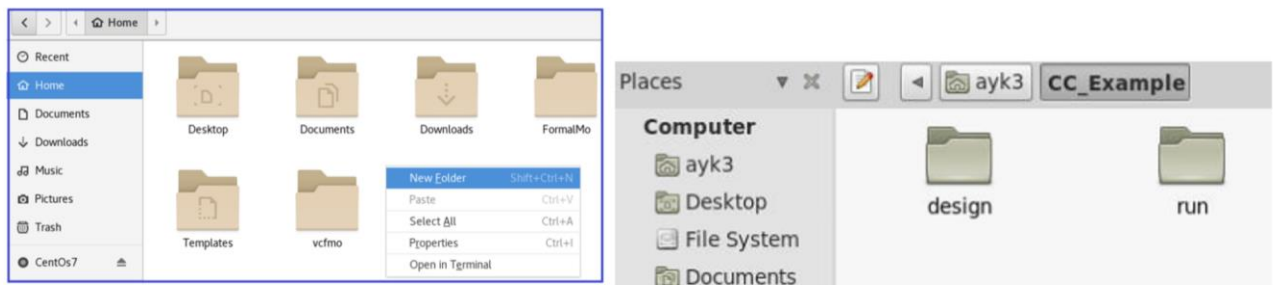
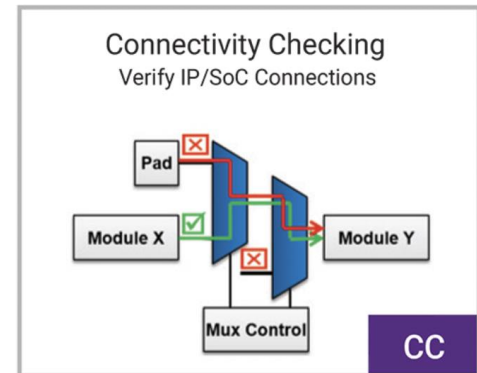


Figure 1. Creating folders to organize design and TCL files.

1.1 About and Usage of CC

The CC app in VC Formal is used as a connectivity verification tool to prove conditional wiring. CC app is a powerful debugging tool that includes automatic root-cause analysis of unconnected connectivity checks which saves significant debug time. It expedites SoC connectivity verification at the top level along with the connection between IP blocks.

The CC application takes the design (RTL) and the connection specifications as inputs, and verifies if the connection specifications are good or not. CC app is used to verify the correctness of a design's connectivity by analyzing the circuit's logic and ensuring that signals are properly connected between modules.



This CC figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

1.2 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.

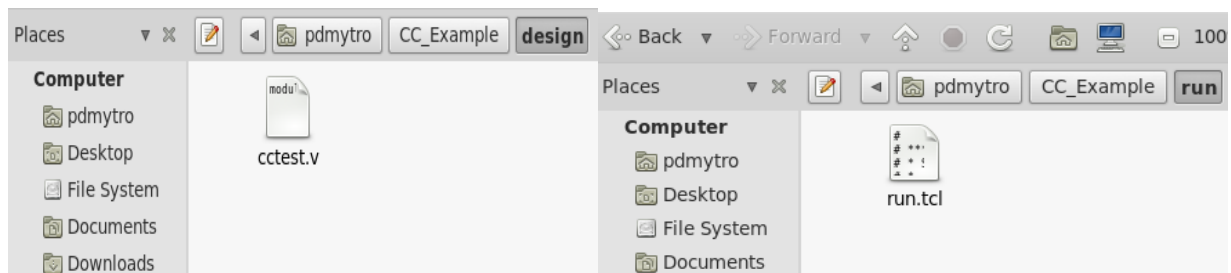


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

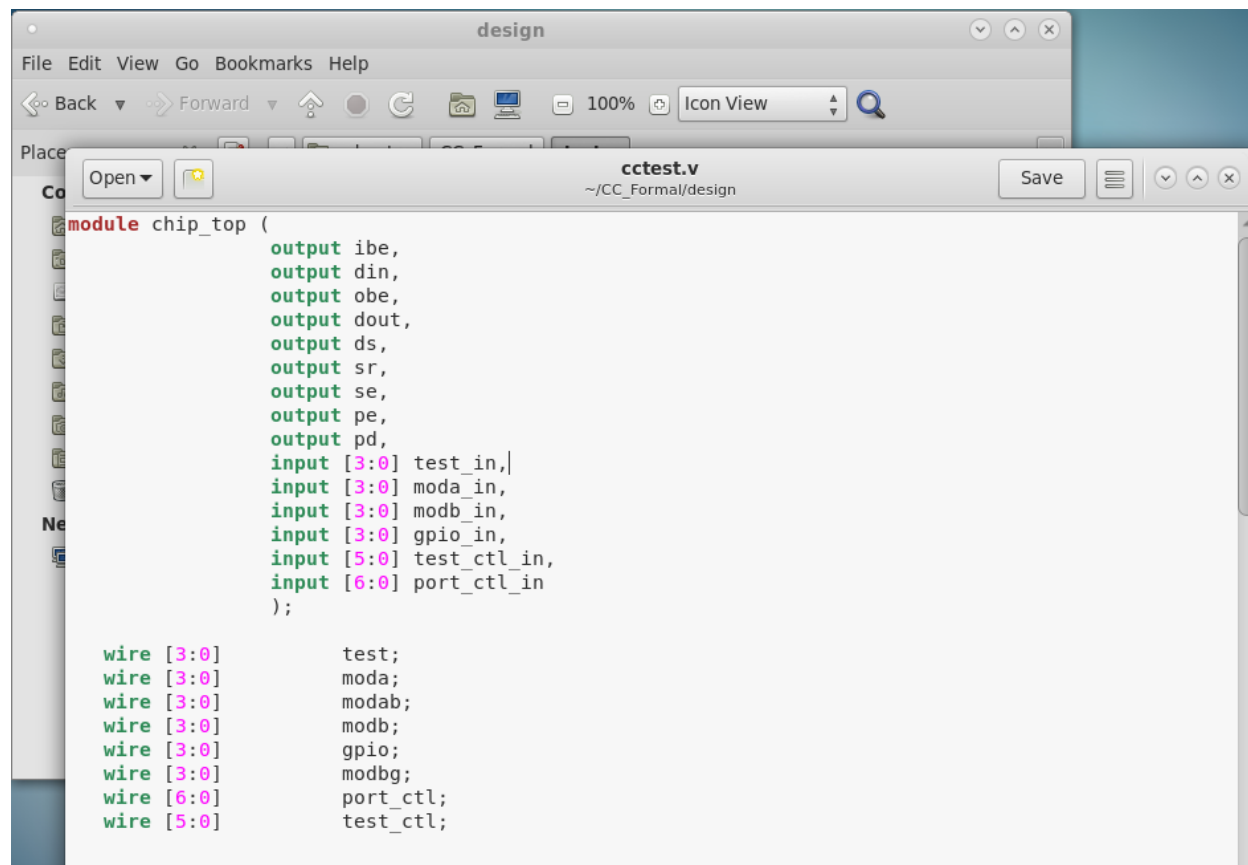


Figure 1.2.2. Example of the design we are using in this tutorial.

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 1.2.2* 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).

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.1*), which you can use as a template for your functional checks on VC Formal.

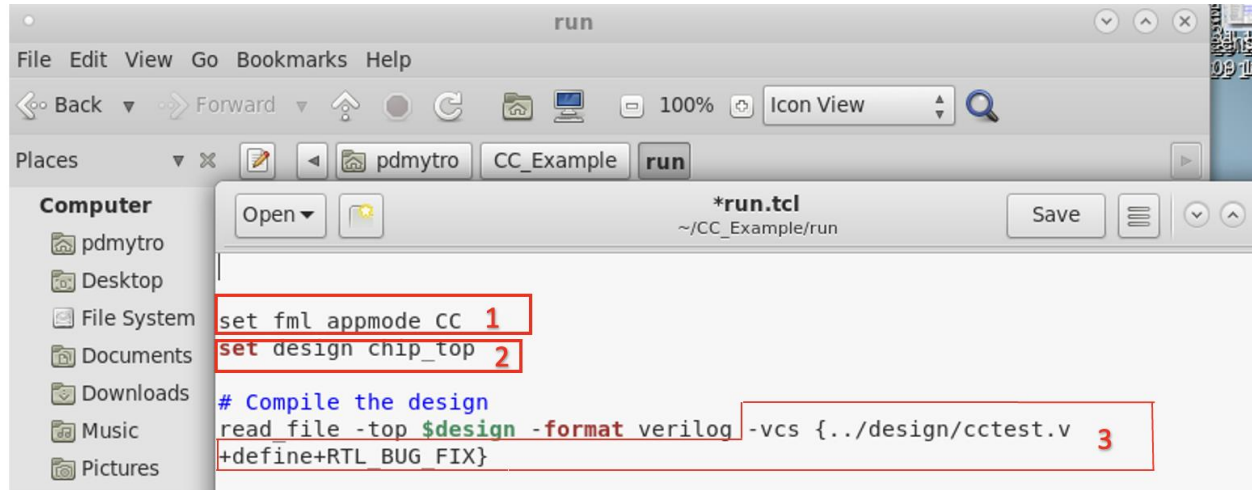


Figure 1.3.1. Annotated TCL template file.

- (1) Instruction that sets the appmode to CC in VC Formal.
- (2) Name of the main module as established in the Design file.
- (3) The property type identifier switch name for desired CC analysis.

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 1.2.2*.

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

2. 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 CC 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.

2.1 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
```

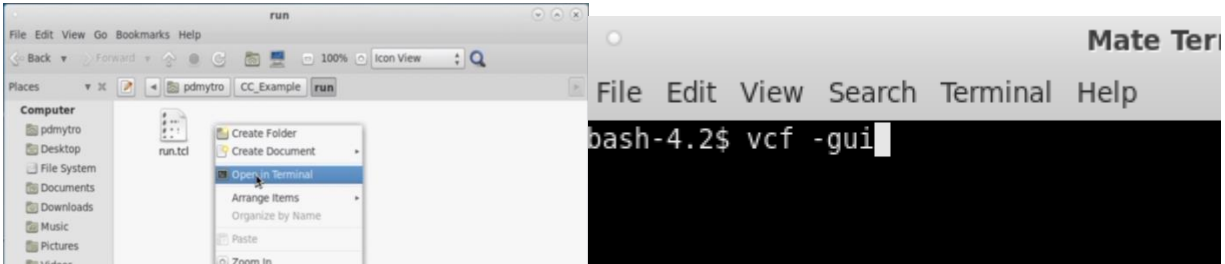


Figure 2.1.1. Invoking VC Formal in the terminal.

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: 

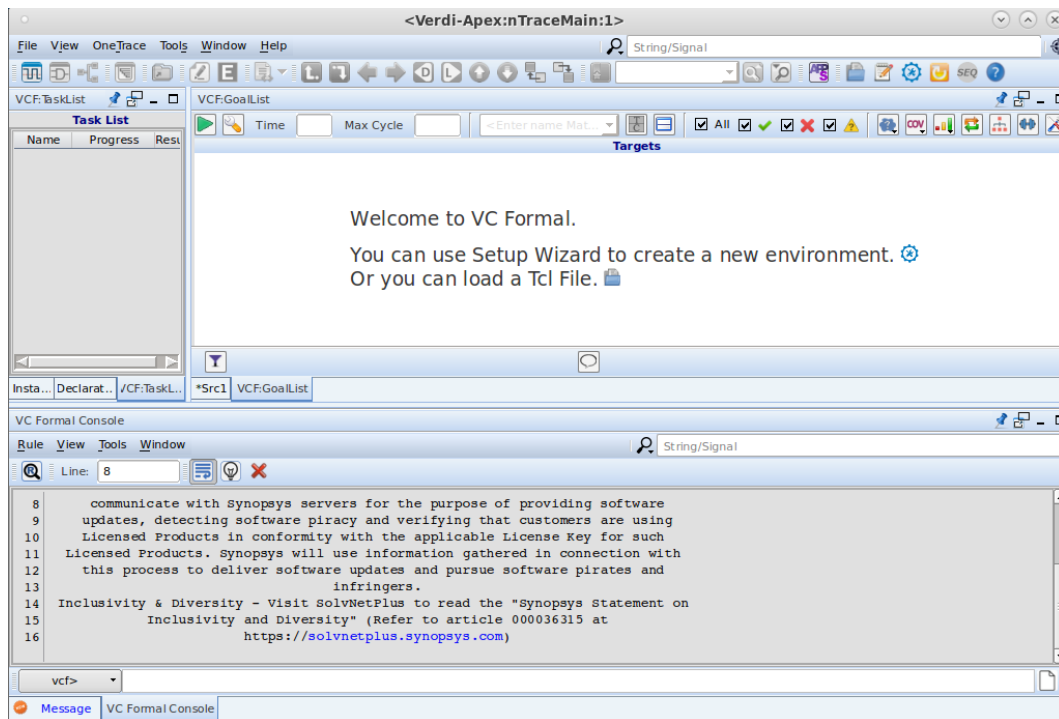


Figure 2.1.2. VC Formal GUI introductory screen. (This screenshot was taken from the VC Formal GUI)

Then load a TCL script by clicking on the  icon **(2)** as shown in Figure 2.1.2.

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

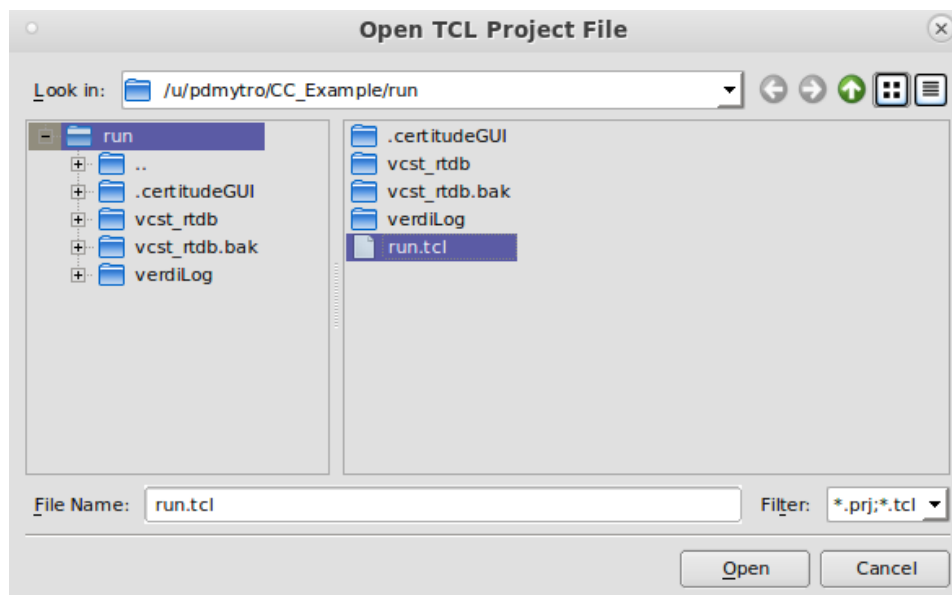


Figure 2.1.3. Selecting TCL file.

2.2 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
```

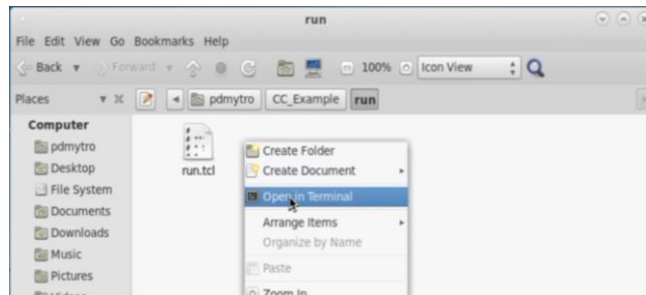


Figure 2.2.1. Opening terminal in the ‘run’ folder.

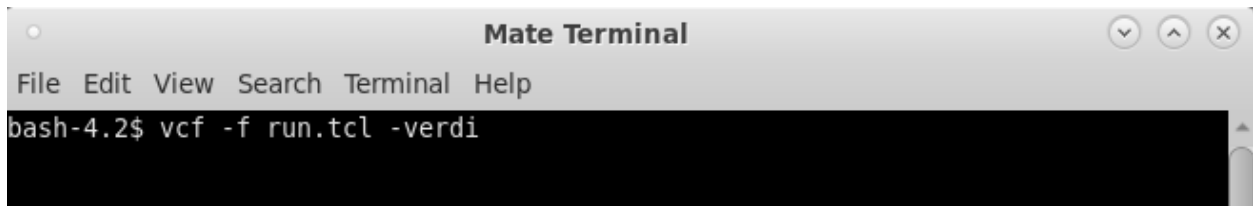


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

3. Application Usage

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:

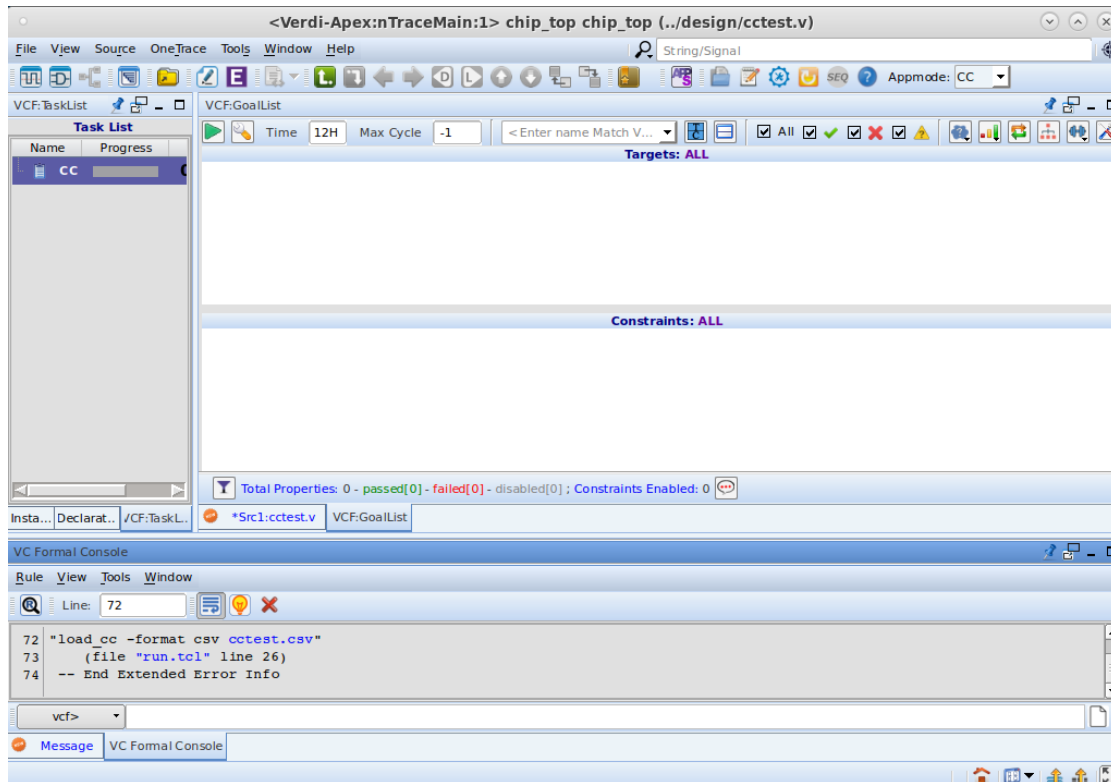


Figure 3.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the “VCF:GoalList” tab.

3.1 Detecting Errors

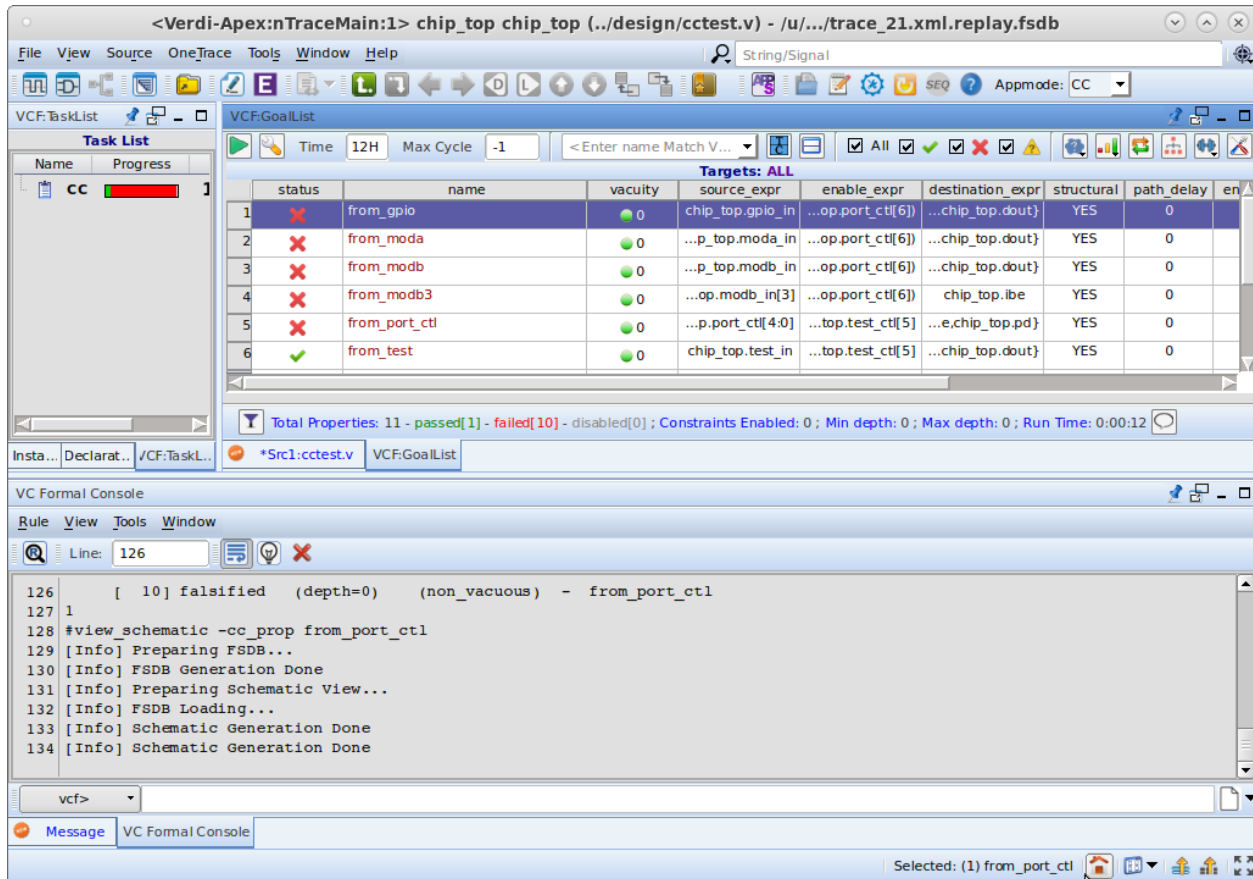


Figure 3.1.1. Screen after running TCL script and the status given = ✗. (This screenshot was taken from the VC Formal GUI)

In Figure 3.1.1 above, we see one  icon, this shows that the connection on that line has been proven and passes.

We also see ten  icons; indicating that the connection on these lines cannot be proven. The  mean that the application is detecting an error in the connection from the source to the destination.

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

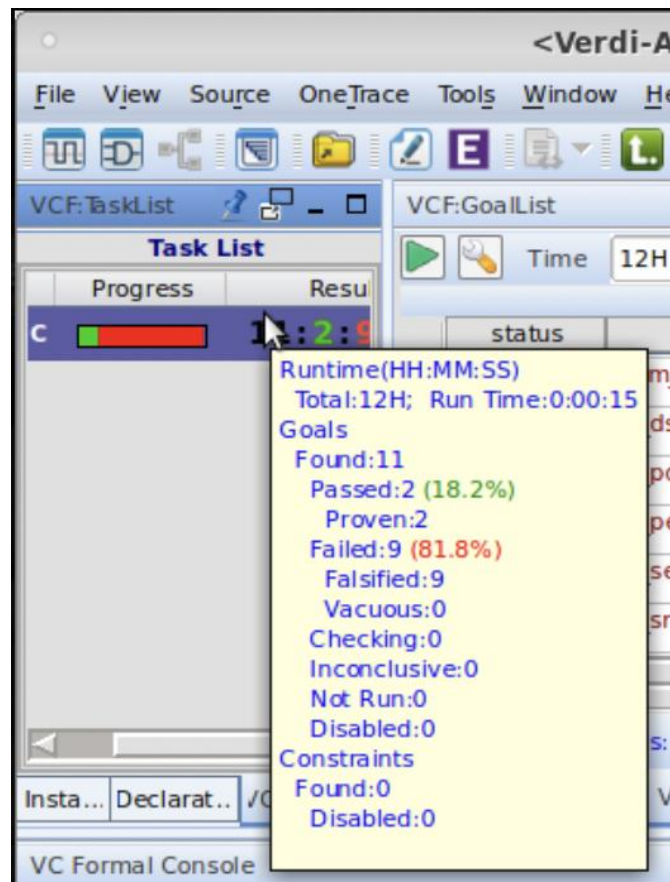


Figure 3.1.2. Results of the analyzed script. (This screenshot was taken from the VC Formal GUI)

3.2 Debugging Failures

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 right-click on the fifth  (line 5) from *Figure 3.1.1*. Select New Schematic Path.

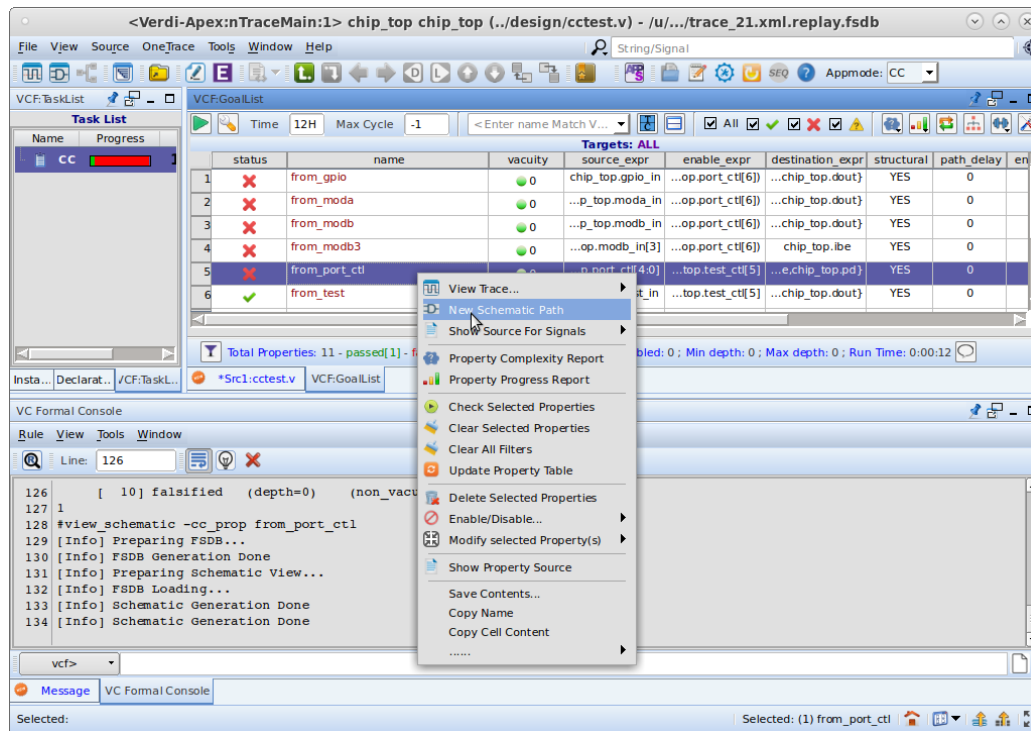


Figure 3.2.1. Examining the failed connectivity check in our design. (This screenshot was taken from the VC Formal GUI)

You should then see a generated schematic as shown below from *Figure 3.2.1*:

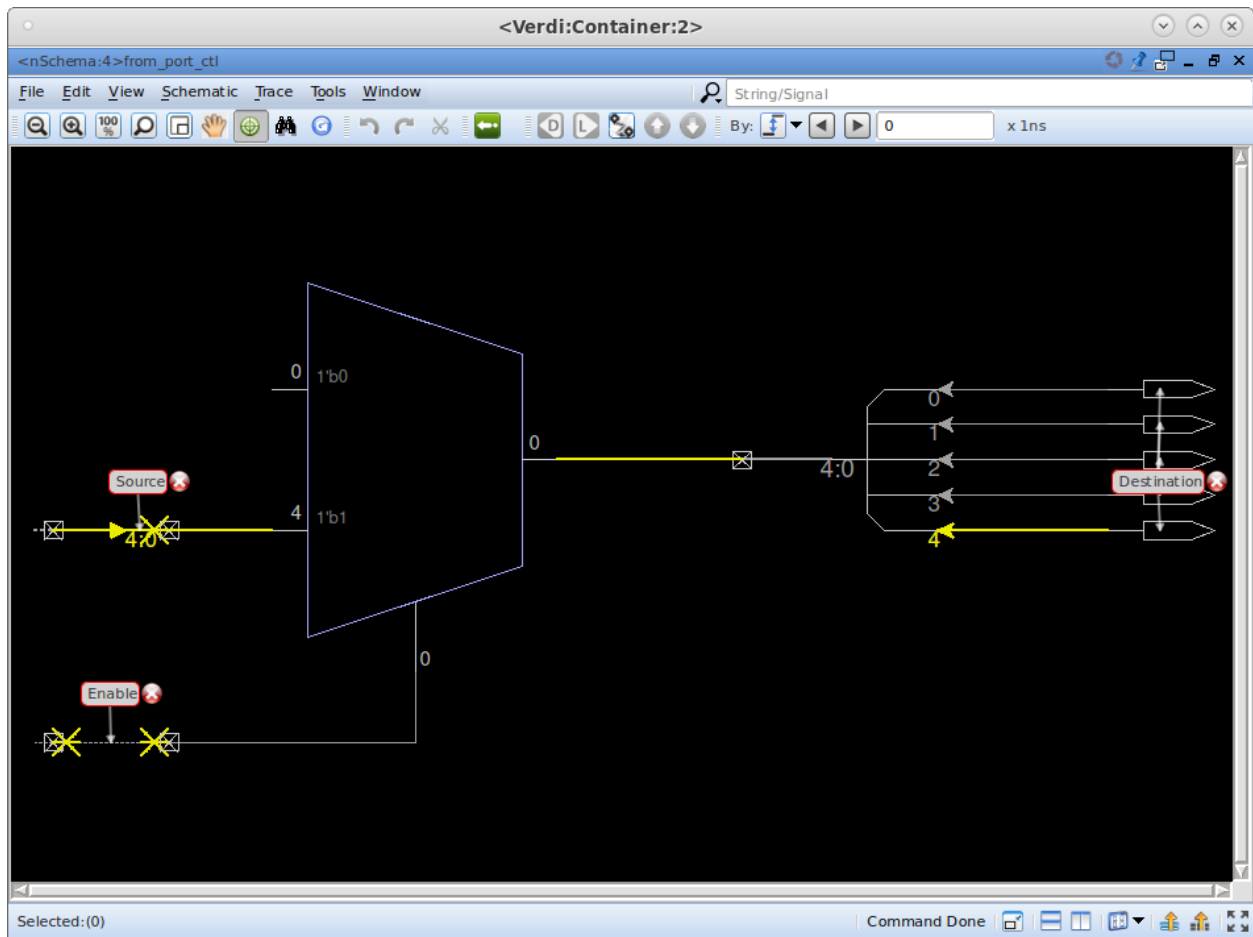


Figure 3.2.2. Examining the schematic (This screenshot was taken from the VC Formal GUI)

As shown in *Figure 3.2.2* 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.

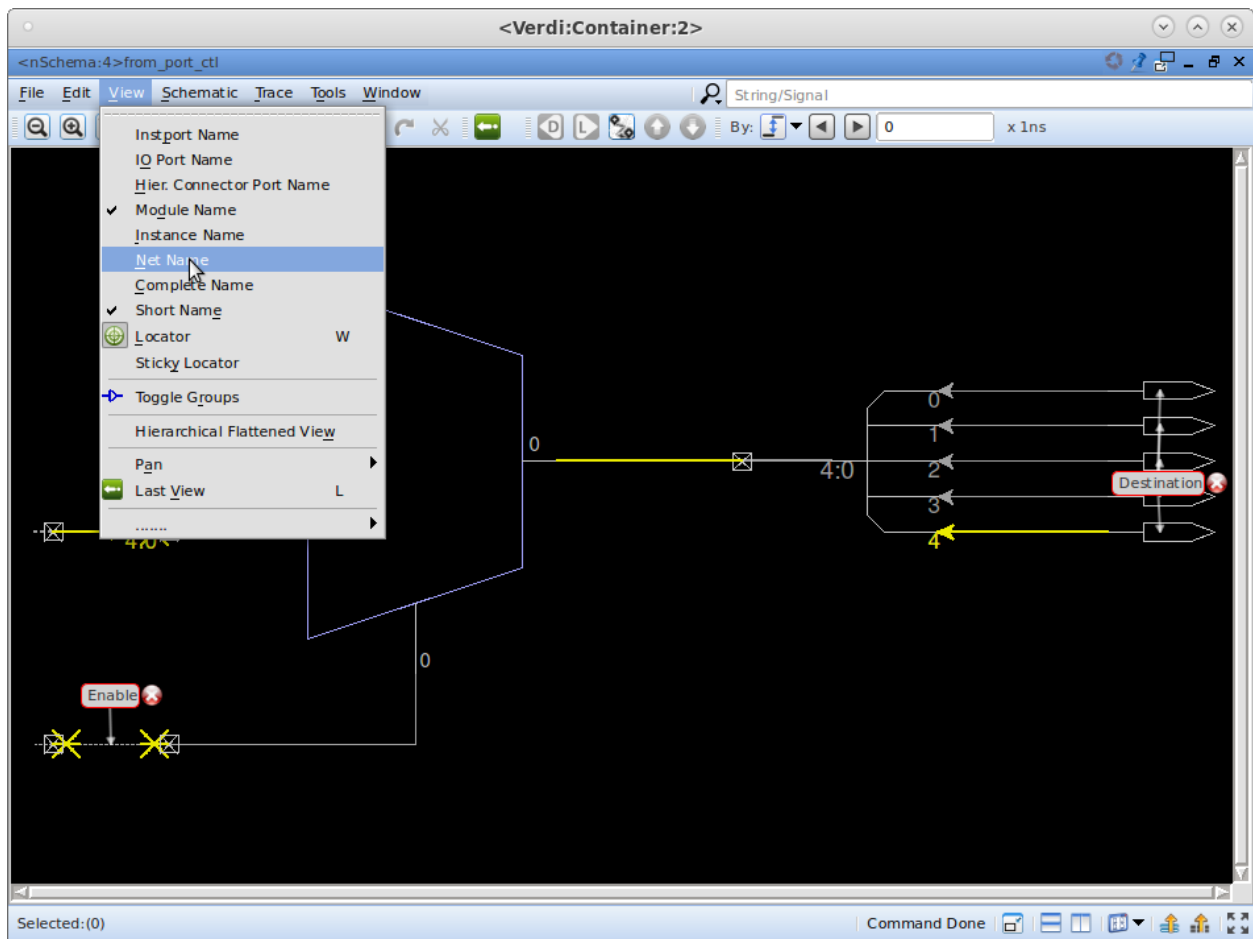


Figure 3.2.3. Showing schematic with how to enable net name view (This screenshot was taken from the VC Formal GUI)

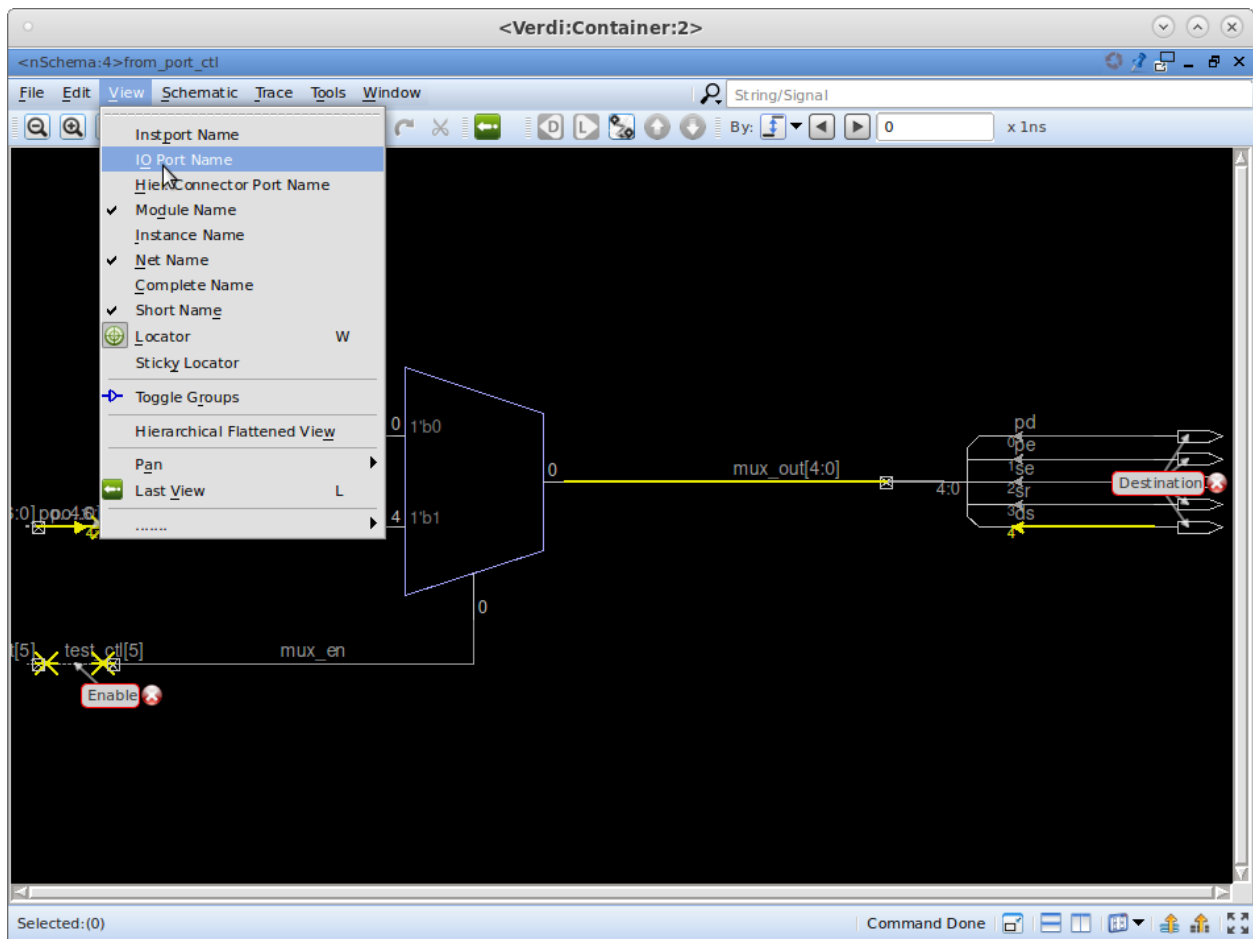


Figure 3.2.4. Showing schematic with how to enable IO Port name view (This screenshot was taken from the VC Formal GUI)

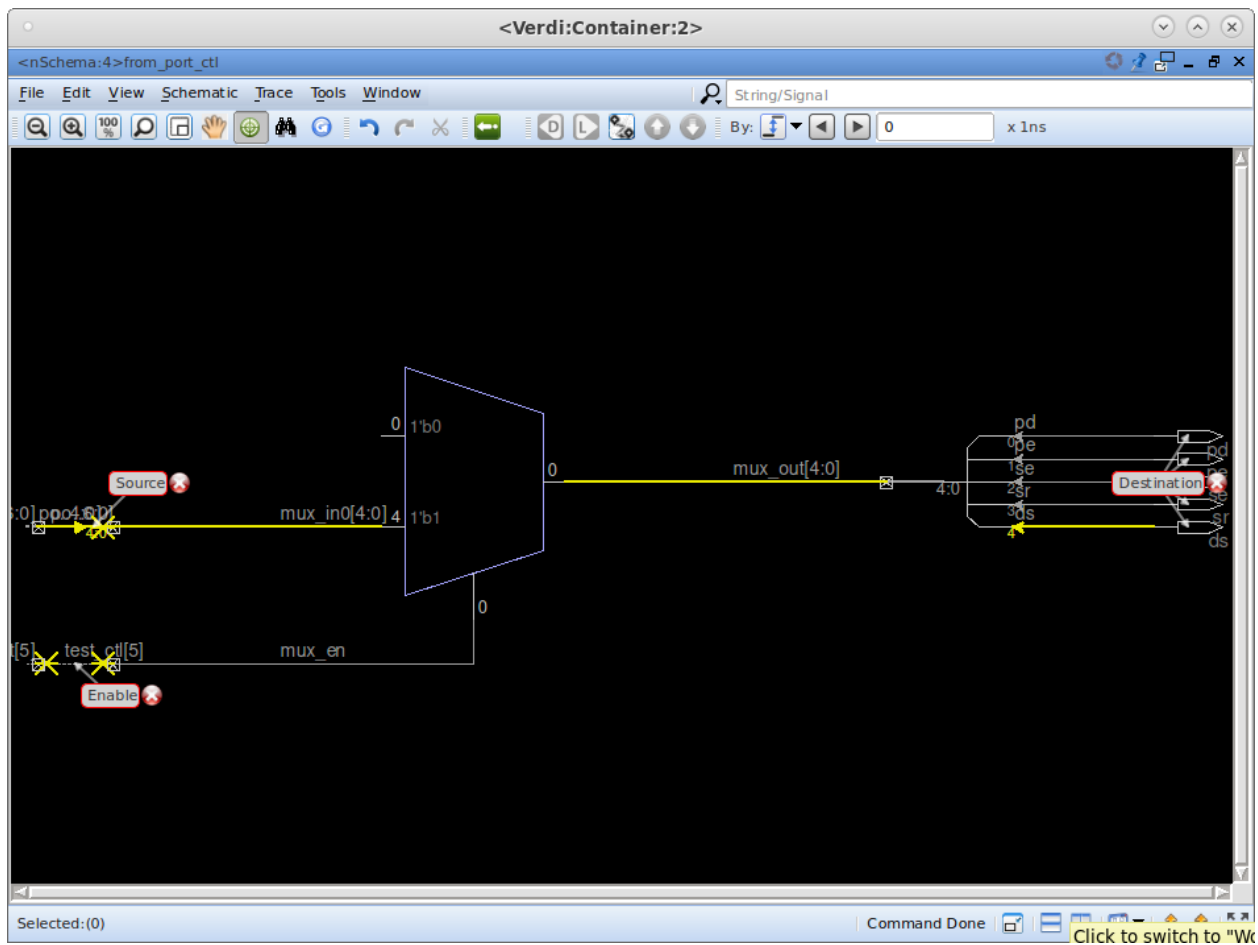


Figure 3.2.5. Net name and IO port name enabled helping to identify source of error (This screenshot was taken from the VC Formal GUI)

3.3 Resolving Errors

Upon debugging the failure on line 5 the schematic shows that the source does not equal to the destination therefore causing an error.

5	✗	from_port_ctl	0	...p.port_ctl[4:0]	...top.test_ctl[5]	...e.chip_top.pd}	YES	0
---	---	---------------	---	--------------------	--------------------	-------------------	-----	---

Figure 3.3.1. Identified Error on line 5

The schematic showing the cause of the error. The source is “*chip_top.port_ctl[4:0]*”, the destination is “*{ds,sr,se,pe,pd}*”, and the enable condition is “*~chip_top.test_ctl[5]*”. In the schematic it can be seen that the mux enable signal is zero, therefore the source is not equal to the destination causing the error.

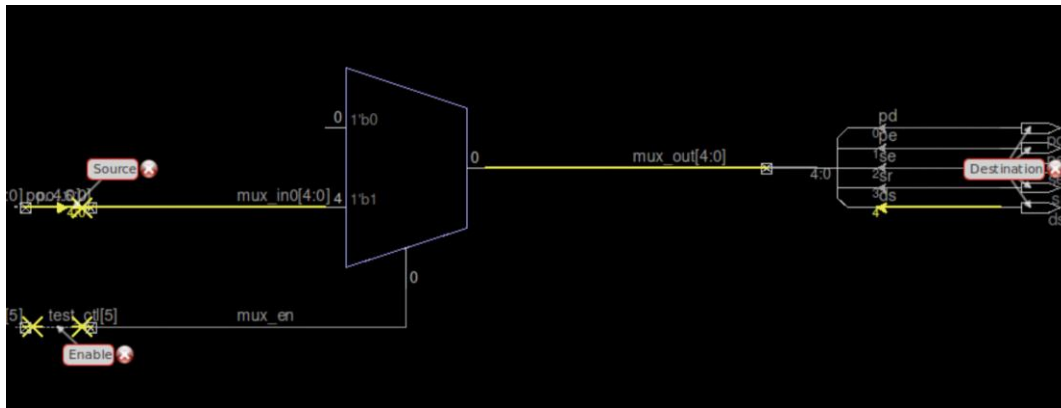


Figure 3.3.2. Identified errors in schematic (This screenshot was taken from the VC Formal GUI)

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

```

cctest.csv
~/CC_Example/run

ENABLE, SOURCE, DESTINATION, NAME
chip_top.test_ctl[5],chip_top.test_ctl[4],chip_top.ds, to_ds
chip_top.test_ctl[5],chip_top.test_ctl[3],chip_top.sr, to_sr
chip_top.test_ctl[5],chip_top.test_ctl[2],chip_top.se, to_se
chip_top.test_ctl[5],chip_top.test_ctl[1],chip_top.pe, to_pe
chip_top.test_ctl[5],chip_top.test_ctl[0],chip_top.pd, to_pd
chip_top.test_ctl[5],chip_top.test_in,{chip_top.ibe,chip_top.din,chip_top.obe,chip_top.dout},
from_test
(chip_top.test_ctl[5] && ~chip_top.port_ctl[6]),chip_top.moda_in,
{chip_top.ibe,chip_top.din,chip_top.obe,chip_top.dout}, from_moda
(chip_top.test_ctl[5] && chip_top.port_ctl[5] && ~chip_top.port_ctl[6]),chip_top.modb_in,
{chip_top.ibe,chip_top.din,chip_top.obe,chip_top.dout}, from_modb
(chip_top.test_ctl[5] && chip_top.port_ctl[5] &&
~chip_top.port_ctl[6]),chip_top.modb_in[3],chip_top.ibe, from_modb3
(chip_top.test_ctl[5] && chip_top.port_ctl[5] && chip_top.port_ctl[6]),chip_top.gpio_in,
{chip_top.ibe,chip_top.din,chip_top.obe,chip_top.dout}, from_gpio
~chip_top.test_ctl[5],chip_top.port_ctl[4:0],
{chip_top.ds,chip_top.sr,chip_top.se,chip_top.pe,chip_top.pd}, from_port_ctl

```

Figure 3.3.3. Alter design file to fix the error on the line highlighted in blue.

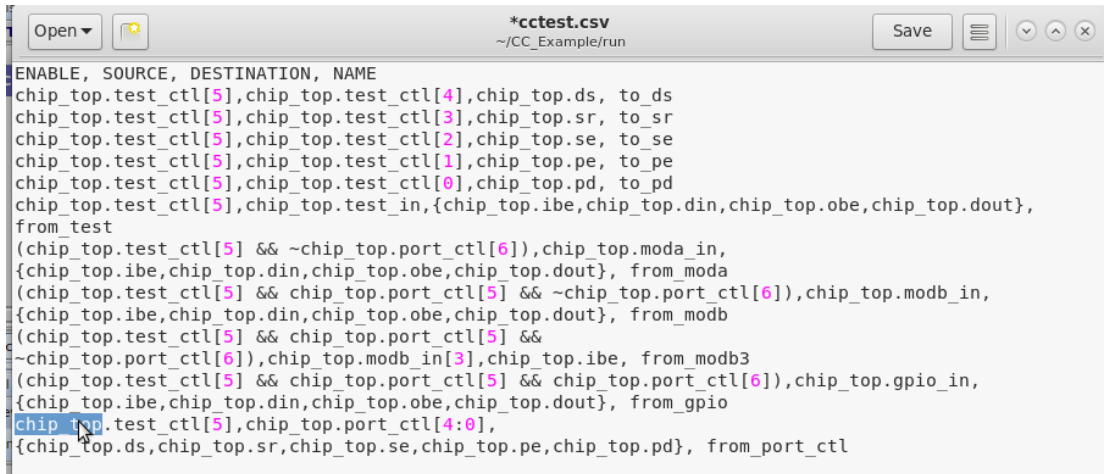


Figure 3.3.4. Remove the “~” from the last line and save the file.

Now return to the VC Formal Application.

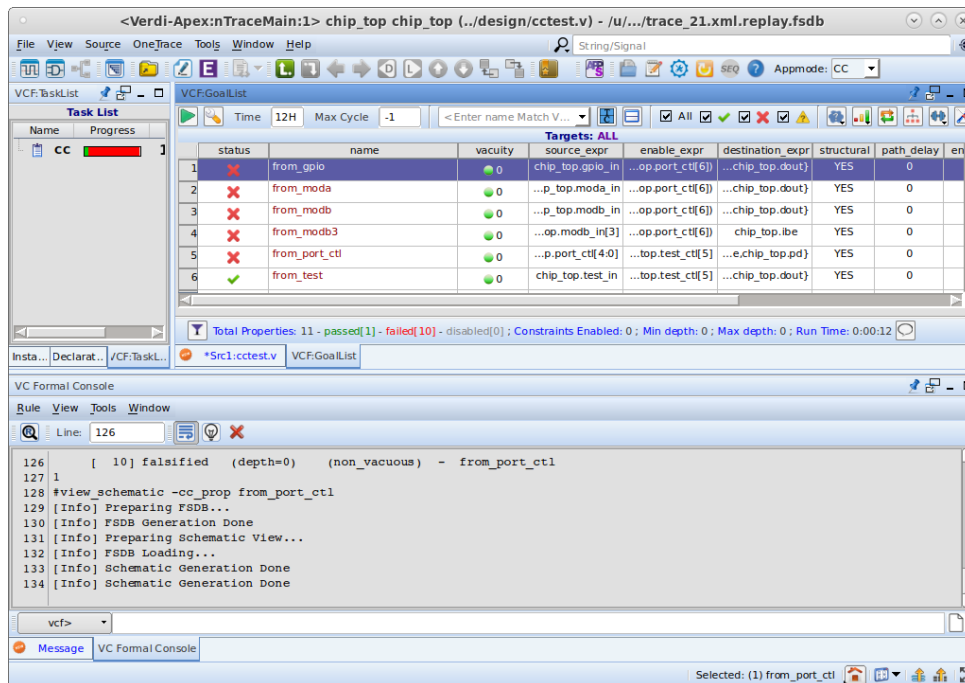


Figure 3.3.5. Restart VC Formal with the updated file by clicking the yellow arrow button in the top right corner next to “Appmode:” (This screenshot was taken from the VC Formal GUI)

The connection now passes.

3.4 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 3.4.1. Location of the restart button in VC Formal window. (This screenshot was taken from the VC Formal GUI)

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

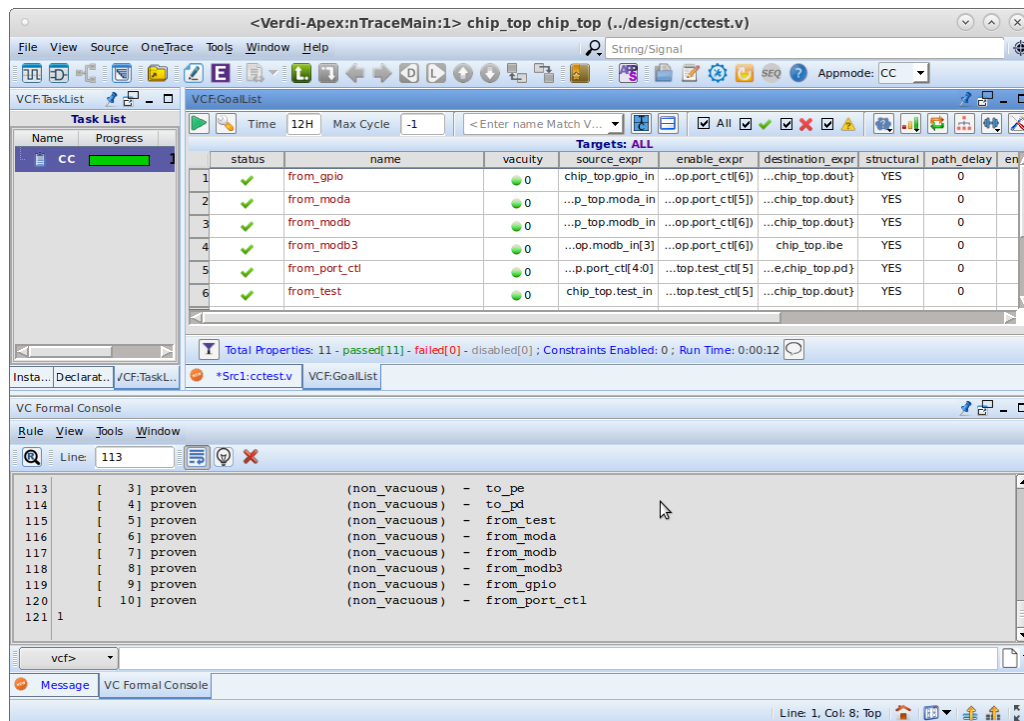


Figure 3.4.2. Results after fixing errors and restarting VC Formal and reloading TCL script. (This screenshot was taken from the VC Formal GUI)

All of the connections now pass.

Appendix

Function	Code	Description
Set App mode	set_fml_appmode CC	VC Formal App mode command for CC
Format	-format <format>	The format of the file to be loaded. The two formats are csv or table. The format option is optional, and default is csv.
Results Summary	-quiet	Option to hide the progress message and summary results.
CSV File Path	<filename>	Path of the properly formatted csv file to read.
Non-applicable Properties to CC	-not_applicable	Shows properties for which structural check is not applicable (disabled/constant src).
Disconnected Properties	-disconnected	Shows structurally disconnected properties only.
Report of Warnings	-warning	Specify this option to get a report of warnings. You can view the warnings file-wise. Appending the -warning switch with -list and -verbose, displays all warnings from the list and verbose.
Detailed Report Information	-verbose	Displays the summary of results and then provides detailed information about the report such as the status, connection name, line number and message of each property file-wise.
enable_delay	<enable_delay>	Enable time relative to source, that is, how long before src does enable need to be asserted (default 0).
offset_delay	<offset_delay>	Number of cycles after reset to start connectivity check.
clock	<clock-expr>	Optional clocking expression for the CC. The default is to check on the posedge of the clock declared with create_clock.
dest	<destination-signals>	Provide an ordered list of signals to use as the destination of a connection to check. Wild card characters are supported

Table 1. CC App important functions and commands

Synopsys VC Formal Tutorial

Formal Property Verification App (FPV)

Version 1.2 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed

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	3
1.1 About and Usage of the FPV App.....	4
1.2 Design Files.....	5
1.3 TCL File	7
1.4 Writing Assertions.....	8
2. Application Setup.....	9
2.1 Invoking VC Formal GUI	10
2.2 Invoking VC Formal Along with TCL File:.....	12
3. Application Usage	13
3.1 Detecting Errors	14
3.2 Exploring Falsified Properties	16
3.3 Resolving Errors	19
3.4 Restarting VC Formal	21
Appendix: SystemVerilog Assertions (SVA)	22

1. Introduction

VC Formal utilizes TCL (Tool Command Language) scripts to direct its actions. These scripts can specify various aspects of the analysis, such as the app to be used, the files to be analyzed, clock cycles, and more. However, instead of delving into the intricacies of writing TCL scripts, we can utilize pre-made templates. By modifying these templates, we can interact with VC Formal in a manner tailored to our specific project needs.

To begin, it is necessary to set up the VNCserver, as outlined in the previous tutorials, and open the Linux GUI. For organizational purposes, it is recommended to create a main folder for the design to be analyzed. In this example, the folder is named "FPV_Example", but it is important to avoid using spaces when naming files and folders. Within this folder, two subfolders are created: "Design" for storing the Verilog or SystemVerilog designs to be analyzed, and "Run" for storing the TCL script that will run the VC Formal FPV analysis, and "sva" for the .sva assertion file.

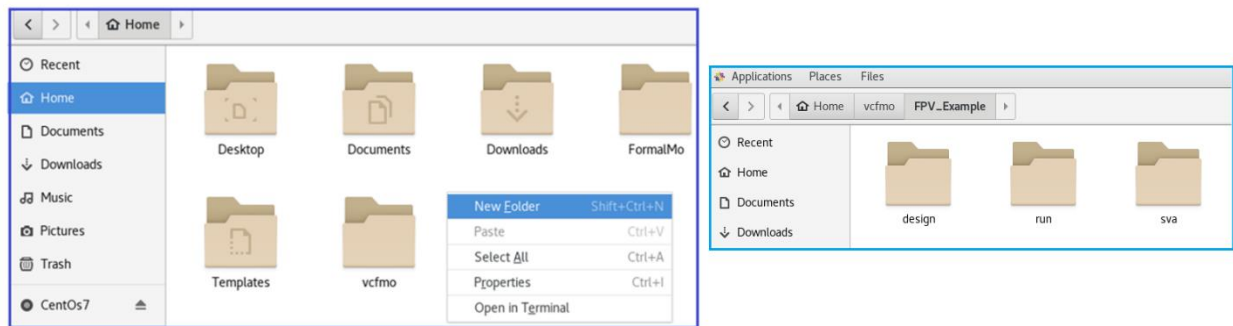


Figure 1.1 Creating folders to organize design and TCL files.

1.1 About and Usage of the FPV App

The FPV application is utilized for verifying a DUT by proving properties. These properties can be created by users or provided by commercial AIPs for interface protocol or function blocks. The app uses a range of powerful VC Formal engines to thoroughly prove or disprove a property. If an assertion fails, the FPV app will generate a counter-example to demonstrate a violating trace. However, there may be cases where it is impossible to definitively prove or disprove a property. In these cases, the app will provide a bounded proof result indicating that no falsification can be found up to a specific depth from the initial state. Proof results for a set of assertions mean that given the given constraints, it is impossible to falsify the properties.

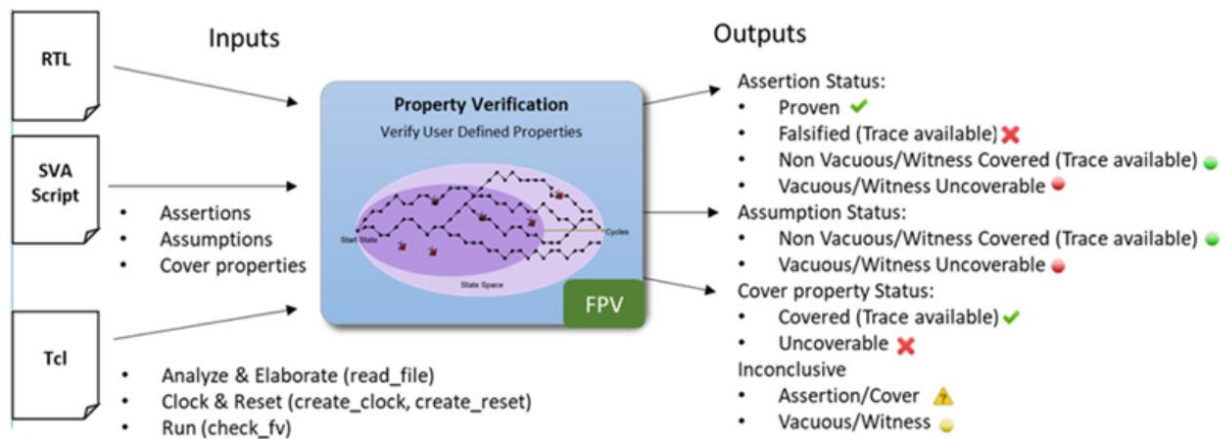


Figure 1.1.1. Inputs and outputs of the FPV app.

This FPV figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

The FPV application requires several inputs, including:

- ❖ RTL design files in Verilog/SystemVerilog/VHDL formats
- ❖ Assertions, Assumptions, and cover properties written in an SVA file, or an inline SVA in the SystemVerilog design file.
- ❖ TCL file

The FPV app would then output the status of properties such as assertion status, assumption status, and cover status. The assertion status could be proven, falsified, vacuous, witness-coverable, uncoverable, or inconclusive.

1.2 Design Files

The SystemVerilog design file should be placed in the Design folder, while the TCL file belongs in the Run folder. It's recommended to name the folders with lowercase letters since VC Formal is case-sensitive. Additionally, ensure that there are no spaces in the names of the folders or files.

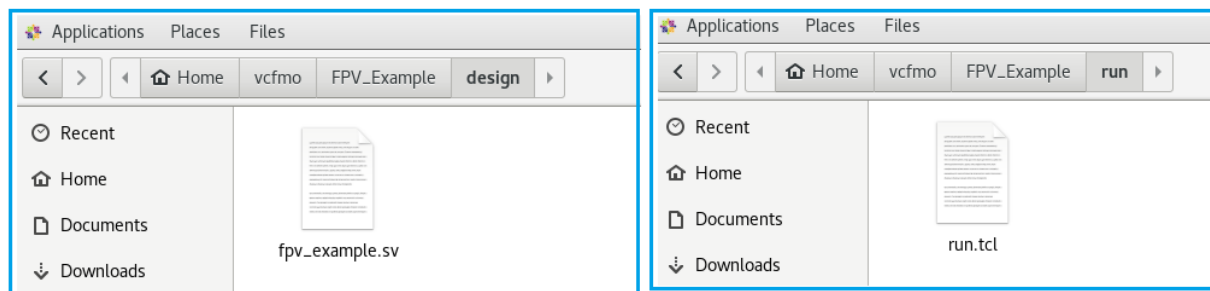


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations. Two places to write the SystemVerilog Assertions

The first way is to create a .sva file in the SVA folder, create a module, and write the SVA assertions in it. You'll then need to create another binder file and bind your SV design and the SVA assertion module together.

The second way is to write inline SVA assertions inside your SV code. This is usually much faster, and possibly easier to do, though it's not the most practical when writing many assertions.

For simplicity we will go through how to write inline assertions within your SV design itself. In this case, you won't need to put anything inside the "sva" folder.

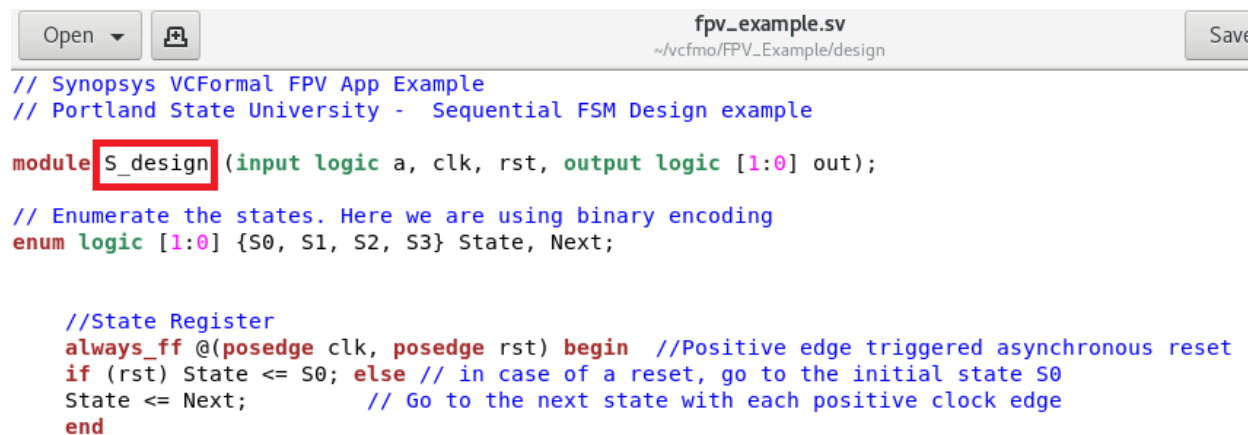


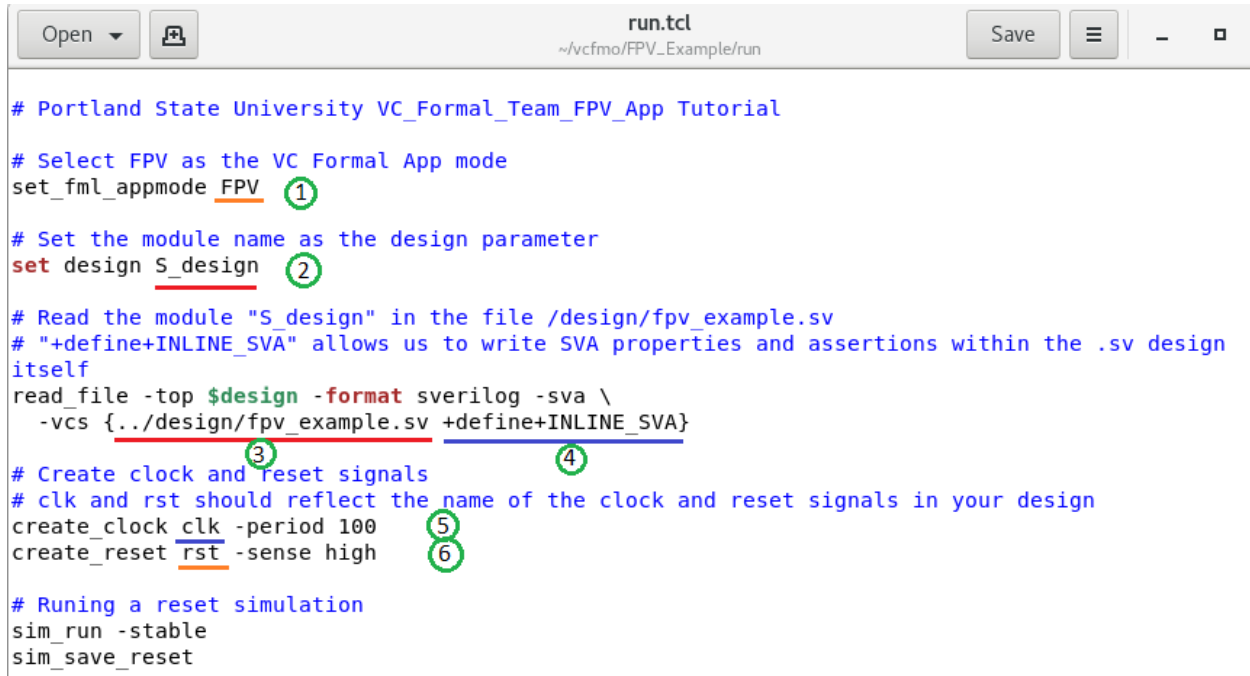
Figure 1.2.2. Example of the design we are using in this tutorial.

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 1.2.2 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).

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.1*), which you can use as a template for your functional checks on VC Formal.



```
# Portland State University VC_Formal_Team_FPV_App Tutorial

# Select FPV as the VC Formal App mode
set_fml_appmode FPV (1)

# Set the module name as the design parameter
set design S_design (2)

# Read the module "S_design" in the file /design/fpv_example.sv
# "+define+INLINE_SVA" allows us to write SVA properties and assertions within the .sv design itself
read_file -top $design -format sverilog -sva \
  -vcs {./design/fpv_example.sv +define+INLINE_SVA} (3) (4)

# Create clock and reset signals
# clk and rst should reflect the name of the clock and reset signals in your design
create_clock clk -period 100 (5)
create_reset rst -sense high (6)

# Running a reset simulation
sim_run -stable
sim_save_reset
```

Figure 1.3.1. Annotated TCL template file.

- (1) Instruction that sets the appmode to FPV in VC Formal.
- (2) Name of the main module as established in the Design file.
- (3) Location of the design file should be specified for VC Formal to locate it.
- (4) “+define+INLINE_SVA” lets VC Formal know that we will be using inline SVA in our design.
- (5) Name of the clock signal. You can specify the period you’d like to use. (You can keep it as 100)
- (6) Name of the reset signal in your design, and whether it’s active high or low.

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 1.3.1*.

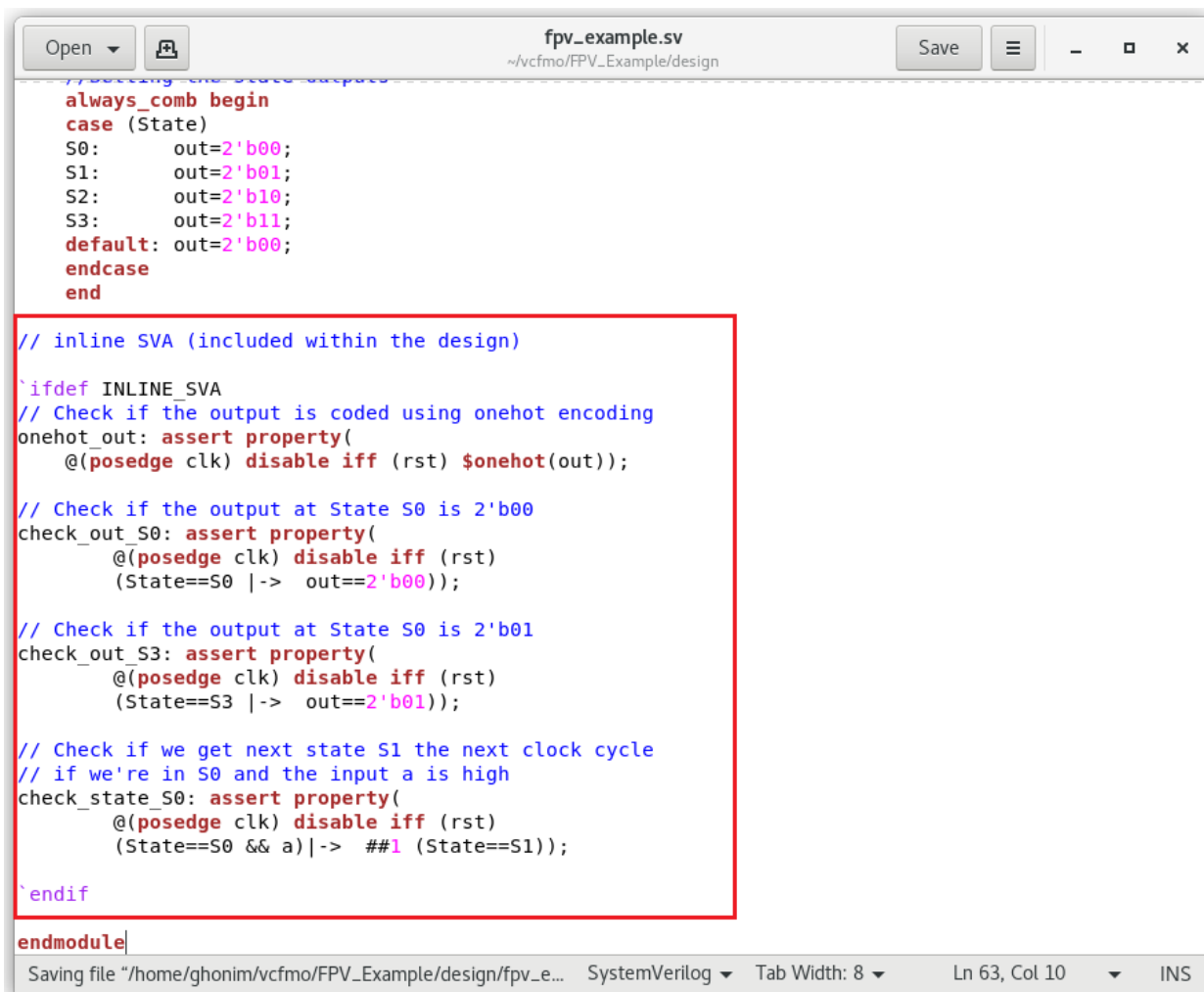
The file name (fpv_example.sv) can be named however you want.

1.4 Writing Assertions

Now to use the FPV app, we need to write the assertions we want to test in our design. The assertions are written in SVA (SystemVerilog Assertions). In this tutorial , we will write inline SVA assertions.

In your sv design file, scroll down just before endmodule to write the assertions.

You'll need to use “``ifdef INLINE_SVA`” before you write the assertions, and “``endif`” once you're done writing the assertion. Below is an example of 4 Inline assertions written at the bottom of my “fpv\_example.sv” file, just before “`endmodule`”

A screenshot of a text editor window titled "fpv_example.sv" with a file path of "~/.vcfmo/FPV_Example/design". The editor shows Verilog code. A red rectangular box highlights a section of code containing four inline SVA assertions. The code includes a case statement for state outputs, followed by the SVA assertions, and ends with an endmodule statement. The status bar at the bottom indicates the file is being saved, the language is SystemVerilog, tab width is 8, and the cursor is at line 63, column 10.

```
Open [icon] fpv_example.sv Save [icon] [icon] [icon] x
~/vcfmo/FPV_Example/design

//Setting the state outputs
always_comb begin
  case (State)
    S0:      out=2'b00;
    S1:      out=2'b01;
    S2:      out=2'b10;
    S3:      out=2'b11;
    default: out=2'b00;
  endcase
end

// inline SVA (included within the design)
`ifdef INLINE_SVA
// Check if the output is coded using onehot encoding
onehot_out: assert property(
  @(posedge clk) disable iff (rst) $onehot(out));

// Check if the output at State S0 is 2'b00
check_out_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 |-> out==2'b00));

// Check if the output at State S0 is 2'b01
check_out_S3: assert property(
  @(posedge clk) disable iff (rst)
  (State==S3 |-> out==2'b01));

// Check if we get next state S1 the next clock cycle
// if we're in S0 and the input a is high
check_state_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 && a)|-> ##1 (State==S1));

`endif

endmodule

Saving file "/home/ghonim/vcfmo/FPV_Example/design/fpv_e... SystemVerilog Tab Width: 8 Ln 63, Col 10 INS
```

Figure 1.4.1. Inline SVA assertions in the design sv file

Check with the Appendix to help you write basic SVA assertions.

2. 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 FPV 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.

2.1 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
```

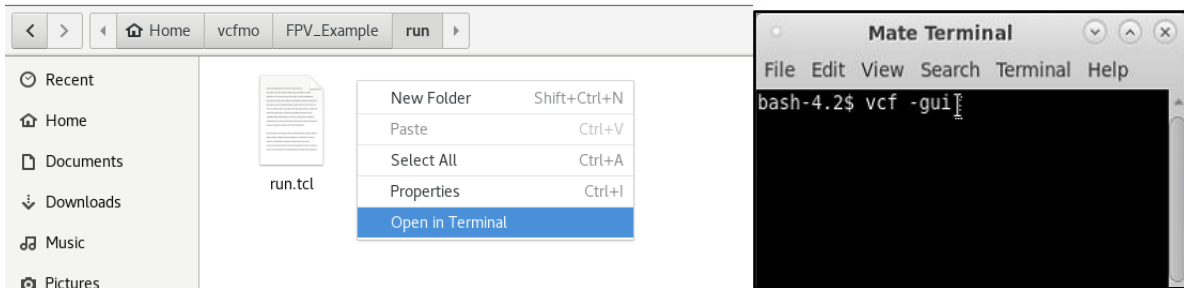


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2 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:

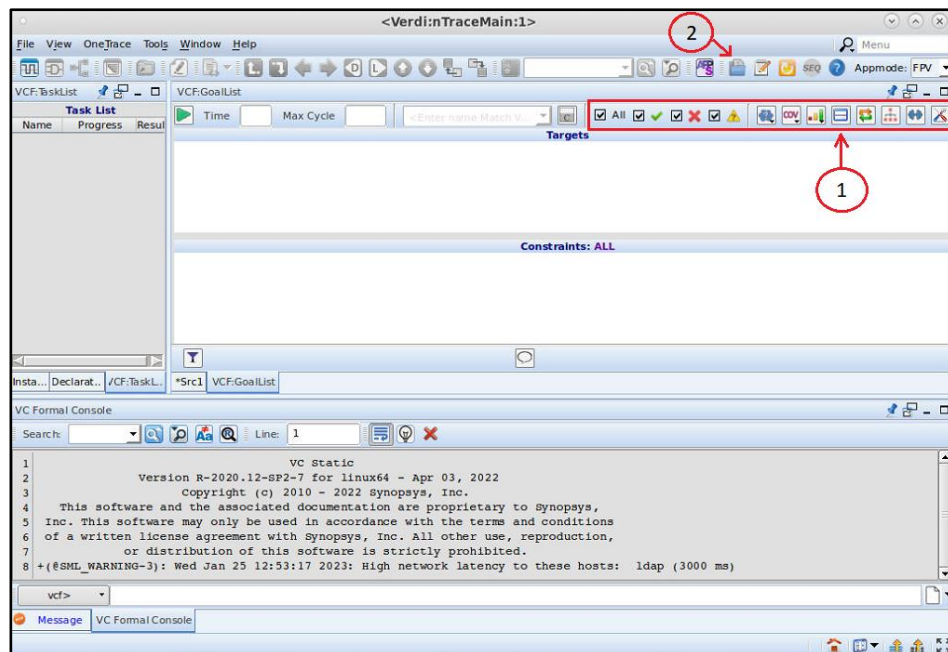


Figure 2.1.2. VC Formal GUI introductory screen. (VC Formal GUI)

Then load a TCL script by clicking on the  icon **(2)** as shown in *Figure 2.1.2*.

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

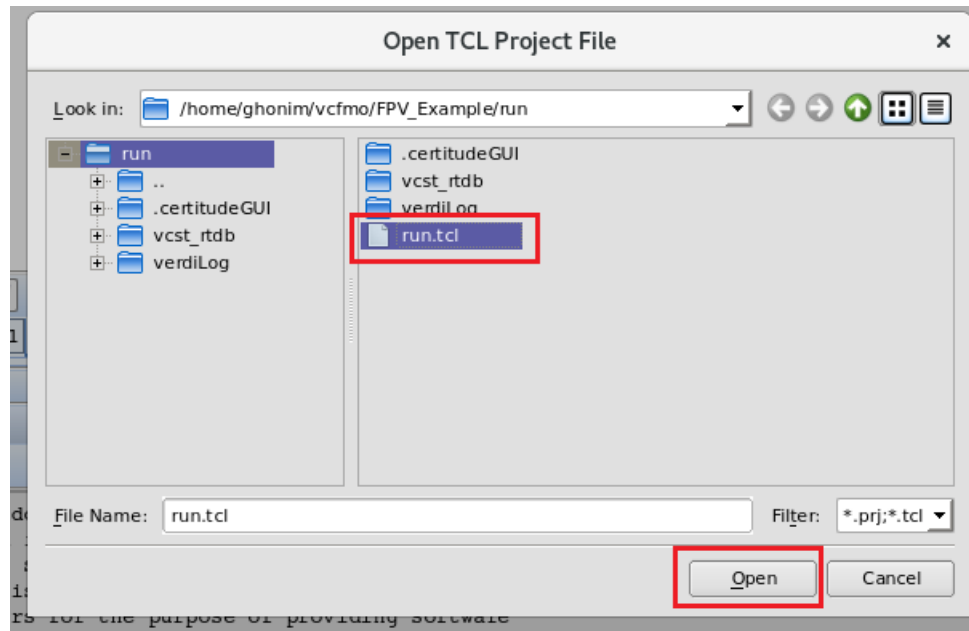


Figure 2.1.3. Selecting TCL file.

2.2 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:

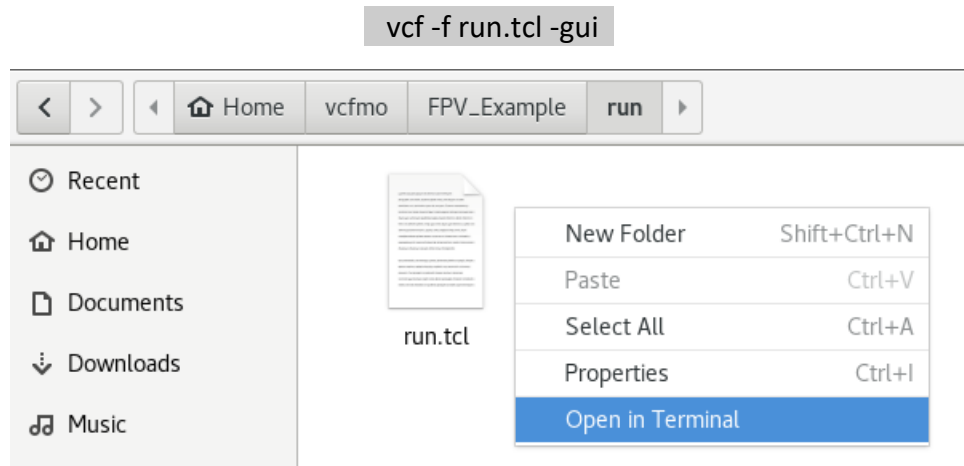


Figure 2.2.1. Opening terminal in the ‘run’ folder.

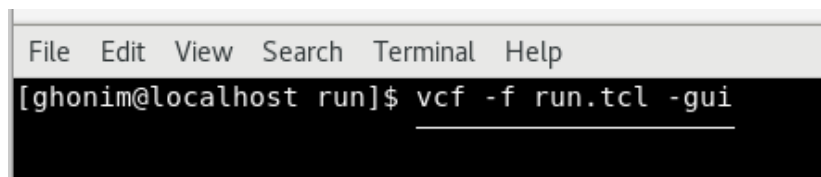


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

3. Application Usage

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:

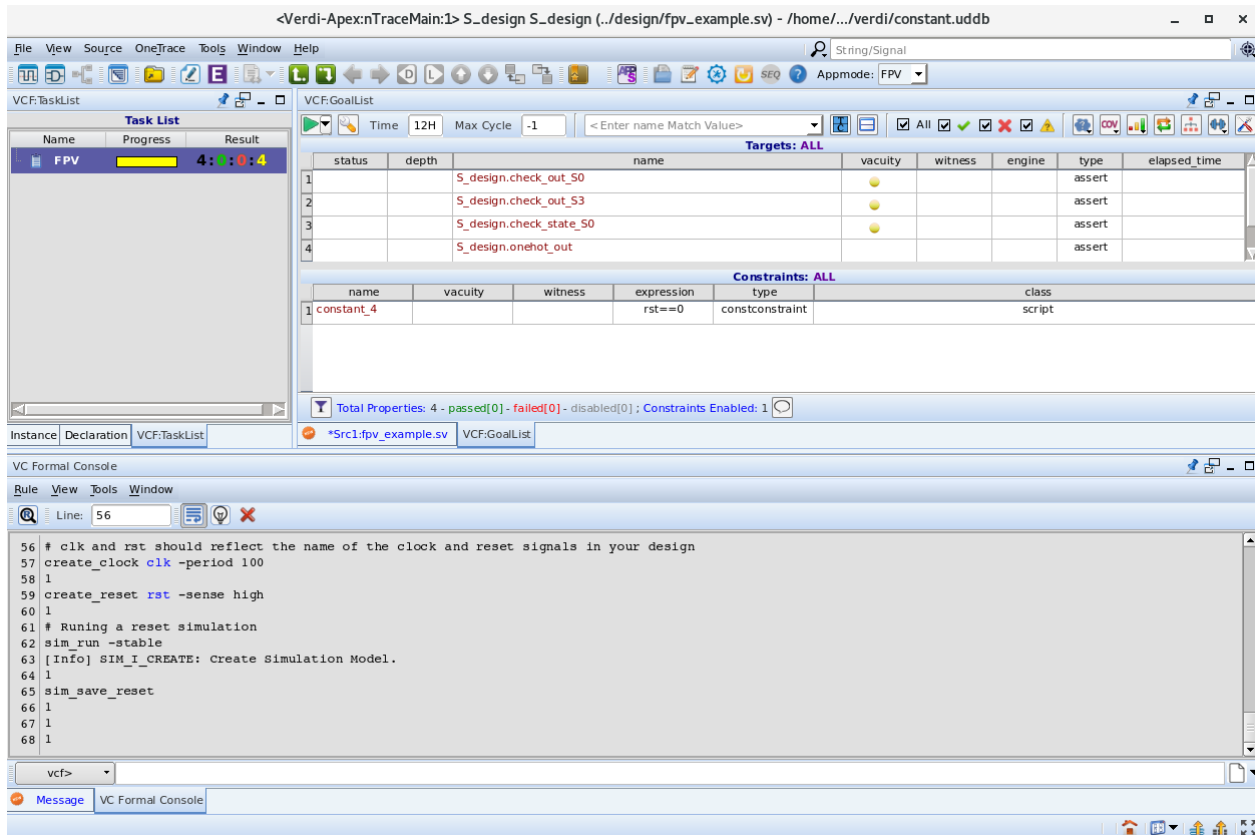


Figure 3.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

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.

3.1 Detecting Errors

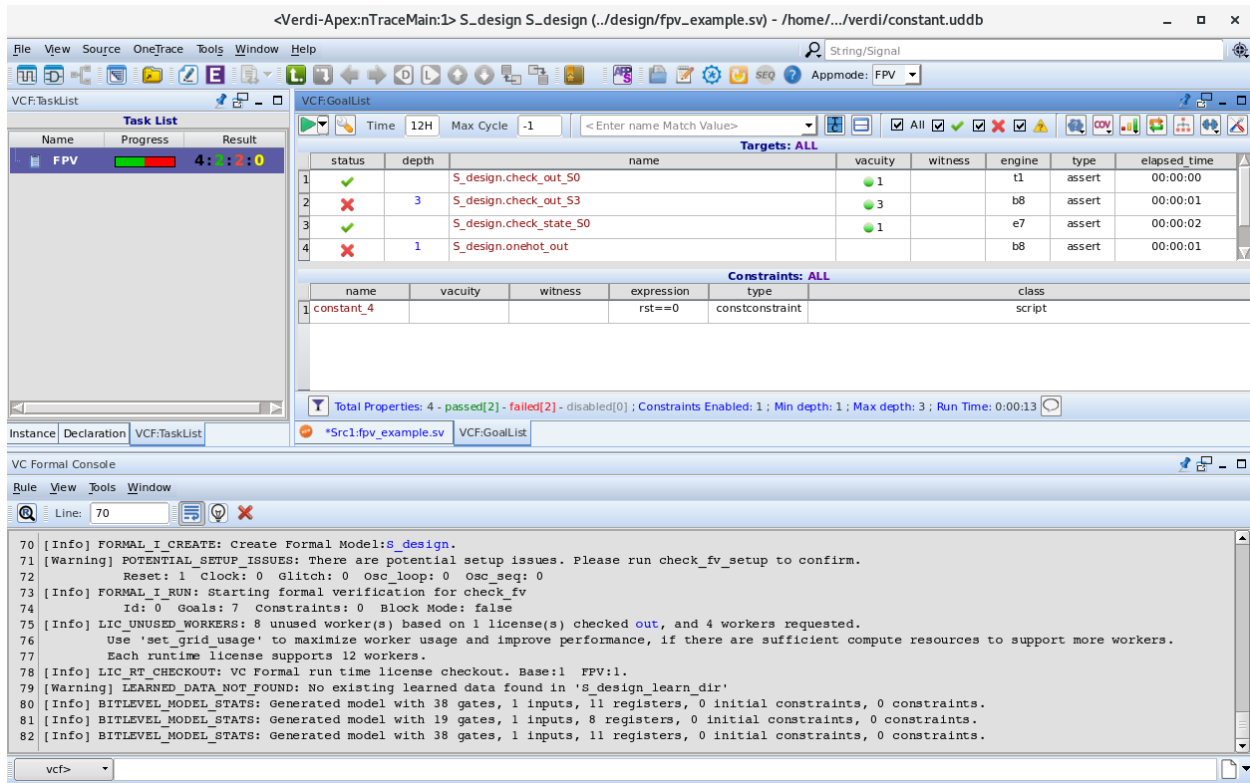


Figure 3.1.1. Screen after running TCL script and the status given = ✗. (This screenshot was taken from the VC Formal GUI)

In Figure 3.1.1 above, we see two  icons, meaning that two of the assertions are proven. In our design, the “check_out_S0” property means that the output of S0 is indeed 2’b00, and the “check_state_S0” assertion means that indeed when we’re in State S0, and the input a is high, the next state is going to be S1 as we specified in our inline assertions.

We also have two  icons, meaning that two of the assertions were falsified. In our design, the “onehot_out” property means that signal “out” has onehot encoding, meaning that we only have exactly one bit high “1” at a time, which clearly false in our design since we’re using binary encoding, and the signal out can be 2’b11, and 2’b00. VC Formal falsified this property, as expected.

The “check_out_S3” property indicates that the output signal “out” of state S3 is 2’b01, which is clearly incorrect since we’re setting that output to be 2’b11 in our design, hence the property is falsified.

On the left under “Task List”, we can see that VC Formal was given one task by the FPV 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:

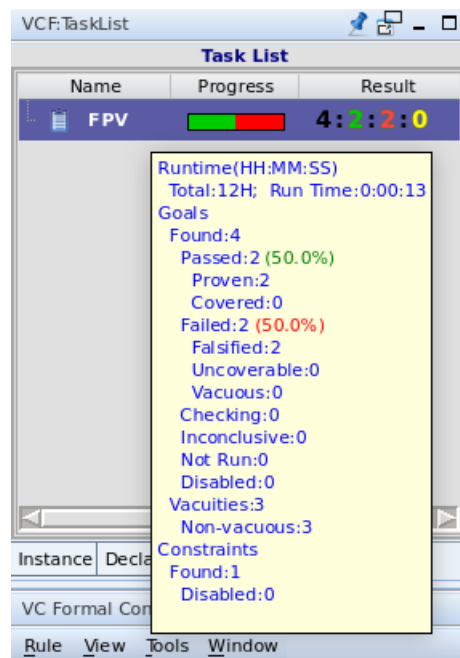


Figure 3.1.2. Results of the analyzed script. (This screenshot was taken from the VC Formal GUI)

3.2 Exploring Falsified Properties

To see what caused the falsified properties to be falsified, double-click on the first  (row 2) from *Figure 3.1.1*. You should then see a generated waveform like the one shown below:

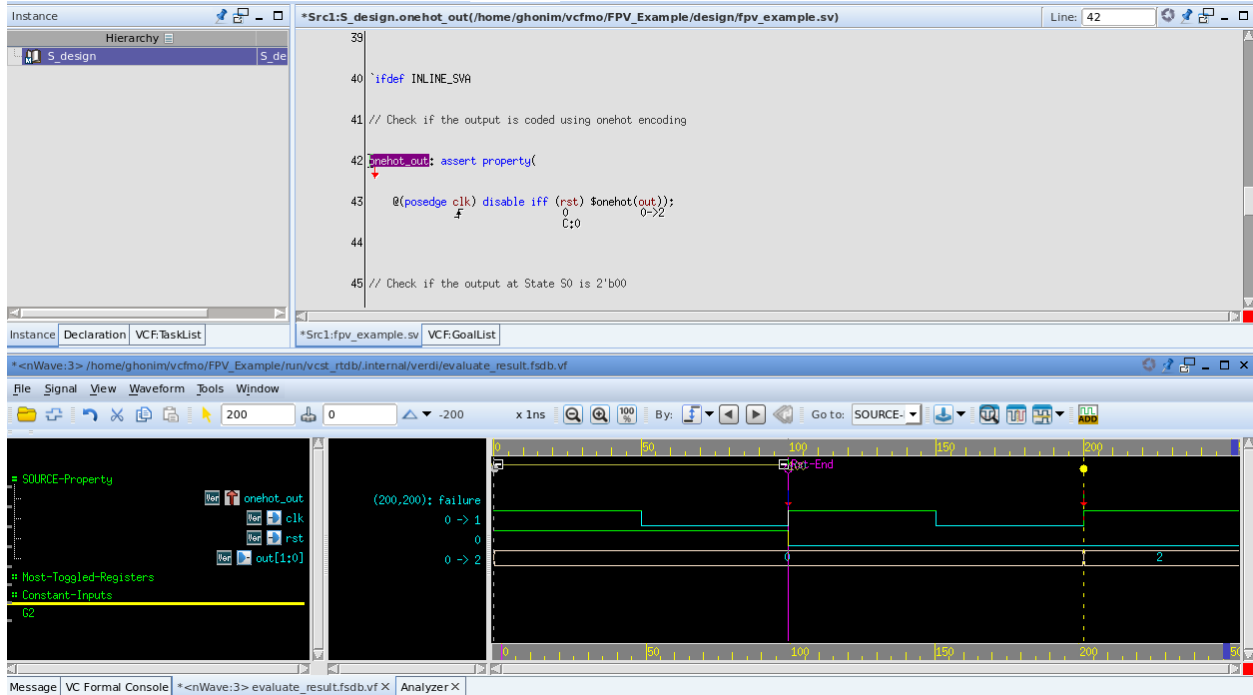


Figure 3.2.1. Examining the failed FPV check in our design. (This screenshot was taken from the VC Formal GUI)

If you double click on the “out[1:0]” signal, you’ll see instances where both out[1] and out[0] are 0’s or 1’s, which is not correct onehot encoding.

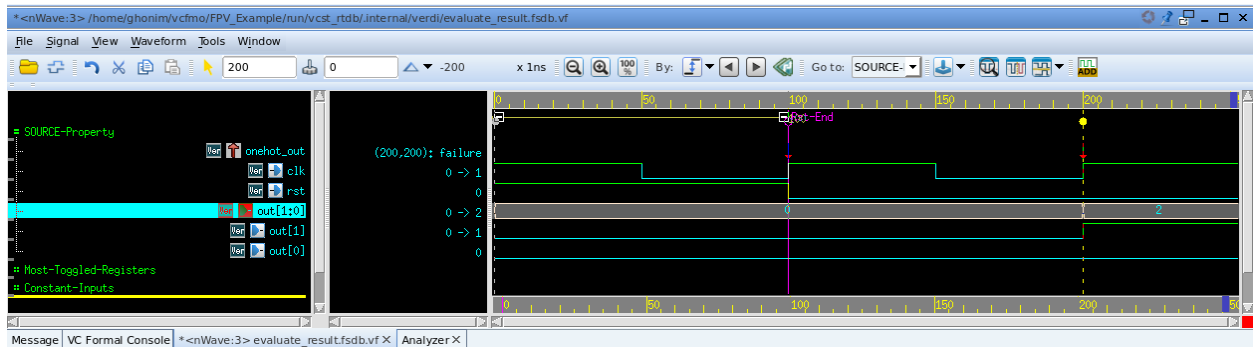


Figure 3.2.2. The out[1:0] signal expanded. (This screenshot was taken from the VC Formal GUI)

To check the other falsified property, we can double-click on the second  (row 4) from *Figure 3.1.1*. You should then see a generated waveform like the one shown below:

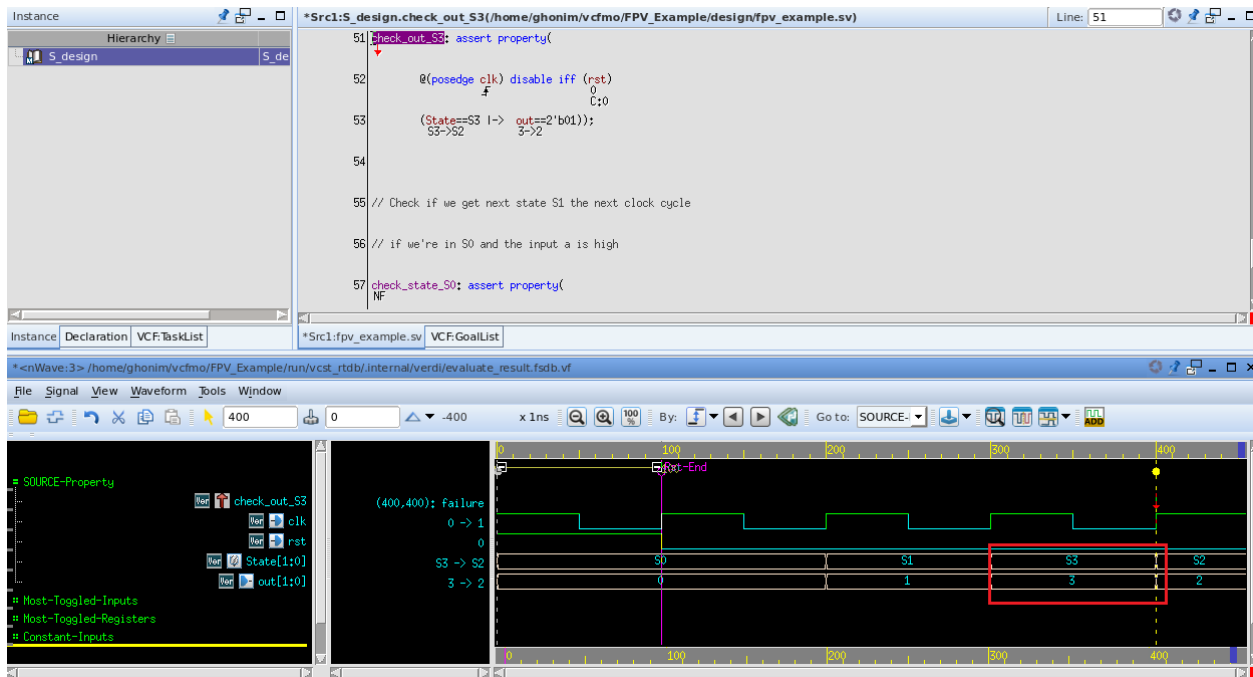


Figure 3.2.3. We see here that State S3 has an output of 3 “11”, and not 1 “01” as specified in our assertion, hence it’s falsified. (This screenshot was taken from the VC Formal GUI)

To gain better understanding of those proved, or falsified signals we can look at the COI “Cone of Influence” digital diagrams.

Right-click on the property which you want to see its COI, and click on “Show COI Schematic”.

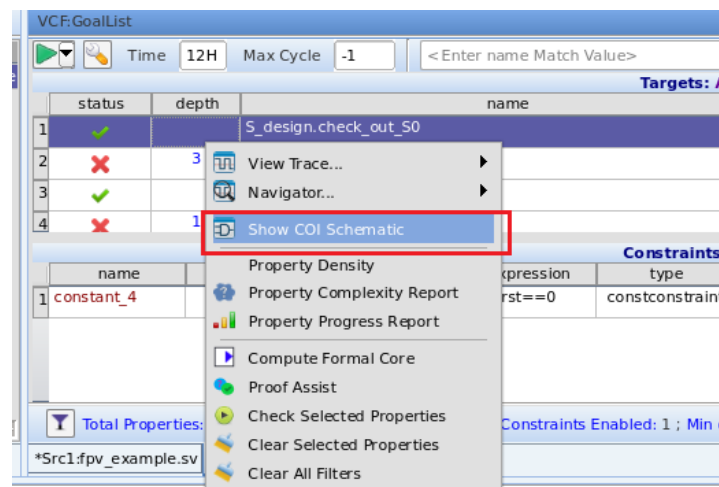


Figure 3.2.4. Showing the Cone Of Influence Schematic (This screenshot was taken from the VC Formal GUI)

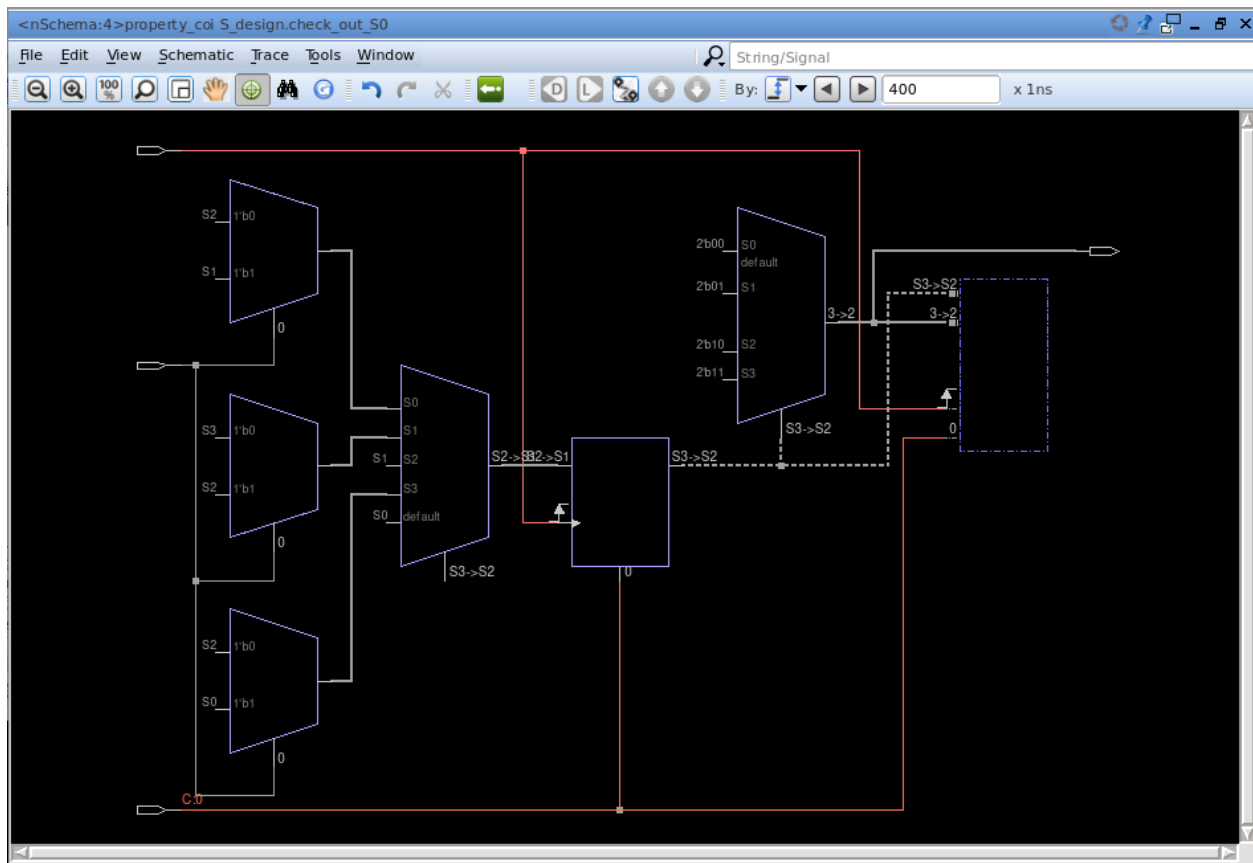


Figure 3.2.5. COI Schematic. (This screenshot was taken from the VC Formal GUI)

And you can do this with the different assertions to help you see the signals and factors affecting the property we're verifying.

3.3 Resolving Errors

To resolve the falsified properties, we need to modify our code such that it behaves as the properties specify, otherwise, we need to rewrite the assertions such that they work for our design.

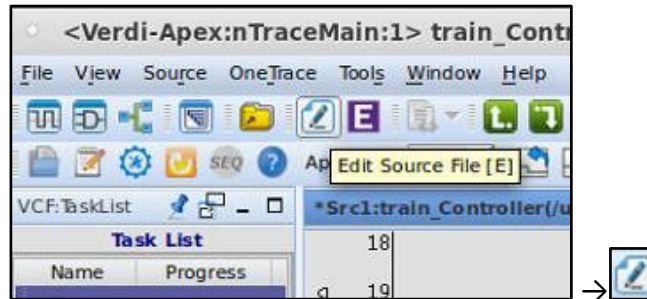


Figure 3.3.1. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes. (This screenshot was taken from the VC Formal GUI)

You can also open the file in any text editor to make the changes, and restart VC Formal to re-run the analysis.

In our case, we modified the output signal such that we have onehot encoding, and the output of state S3 is 2'b01, and we also modified the assertion for the output of S0 to be 2'b01 to maintain onehot encoding.

```

//Setting the state outputs
always_comb begin
  case (State)
    S0: out=2'b01;
    S1: out=2'b01;
    S2: out=2'b10;
    S3: out=2'b01;
    default: out=2'b01;
  endcase
end

// inline SVA (included within the design)

`ifdef INLINE_SVA
// Check if the output is coded using onehot encoding
onehot_out: assert property(
  @(posedge clk) disable iff (rst) $onehot(out));

// Check if the output at State S0 is 2'b00
check_out_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 |-> out==2'b01));

// Check if the output at State S0 is 2'b01
check_out_S3: assert property(
  @(posedge clk) disable iff (rst)
  (State==S3 |-> out==2'b01));

// Check if we get next state S1 the next clock cycle
// if we're in S0 and the input a is high
check_state_S0: assert property(
  @(posedge clk) disable iff (rst)
  (State==S0 && a) |-> ##1 (State==S1));

`endif
endmodule

```

Figure 3.3.2. Updated *fpv_example.sv* file. (this is our own design)

Now we run the VC Formal FPV app analysis again, we can do this by closing VC Formal and running it again in the same way, or by clicking the restart button after updating the SV source file.

3.4 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.

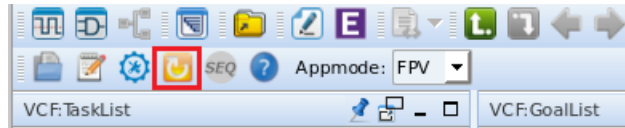


Figure 3.4.1. Location of the restart button in VC Formal window. (This screenshot was taken from the VC Formal GUI)

Click on the green run button again to get the formal property verification results:

status	depth	name	vacuity	witness	engine	type	elapsed_time
1	✓	S_design.check_out_S0	1		t1	assert	00:00:01
2	✓	S_design.check_out_S3	3		t1	assert	00:00:01
3	✓	S_design.check_state_S0	1		rp1	assert	00:00:01
4	✓	S_design.onehot_out			t1	assert	00:00:01

name	vacuity	witness	expression	type	class
1 constant_4			rst==0	constconstraint	script

Total Properties: 4 - passed[4] - failed[0] - disabled[0] ; Constraints Enabled: 1 ; Run Time: 0:00:07

Figure 3.4.2. Now we see that all the properties have been proved! (This screenshot was taken from the VC Formal GUI)

Appendix: SystemVerilog Assertions (SVA)

Assertions are an important tool used to verify the behavior of a design and provide functional coverage information. This involves determining whether the design is working correctly and assessing the quality of the test. Assertions can be checked either dynamically through simulation or statically using a separate tool known as a property checker. This tool is used to confirm if the design meets its specification but may require certain assumptions about the design's behavior to be defined.

In SystemVerilog, there are two types of assertions: immediate (`assert`) and concurrent (`assert property`). Concurrent assertions include coverage statements (`cover property`) and assume property statements, which have the same syntax as concurrent assertions. Another statement called `expect` is utilized in testbenches and checks that some specified activity occurs. All these statement types use sequences and properties to describe the design's temporal behavior over time, defined by one or more clocks.

Example:

```
// Check if the output is coded using onehot encoding (this is just a comment)
```

```
    onehot_out: assert property(  
        @(posedge clk) disable iff (rst) $onehot(out));
```

The first thing you do is to name your property or assertion, you do this by writing that name with no spaces, when putting a colon.

```
Name_of_the_property: assert property( ....);
```

If you are writing an immediate assertion, you use “`assert`”, otherwise, if it’s a concurrent assertion, you should use “`assert property`” and open the brackets where the whole assertion goes. After the brackets, you must put a semicolon.

Now, inside the assertion, if you're writing an immediate assertion, there’s no need to use a clock or reset signals, you write a statement as if it’s an if statement in a `always_comb` block. Immediate assertions are rarely used.

The most common assertions are concurrent assertions, and below are some examples:

```
// Check if the output at State S0 is 2'b00
```

```
check_out_S0: assert property(  
    @(posedge clk) disable iff (rst)  
    (State==S0 |-> out==2'b01));
```

This property takes place at every positive clock edge, and it's disabled (won't happen) if and only if the reset signal (rst) is high, this assertion is disabled (we're not expecting it to hold if we have a reset signal) we're assuming this reset signal is positive edge triggered, as we specified in our design.

If reset is not asserted, at every clock edge the FPV app will check which state we're in, and if we're in State S0, this implies that the output is 2'b01. We're specifying that this is a property that needs to hold. If it doesn't hold, then something is wrong with our design, or the way we wrote this assertion.

|-> Overlapping Implication operator. The RHS is evaluated at/from the same cycle the LHS is true.

|=> Non-overlapping Implication operator. The RHS is evaluated one clock cycle after the LHS is true.

For example:

```
// if both a and b are high, then ack is high after 2 clock cycles from this cycle.
```

```
Check_ack: assert property (@posedge (clk) disable iff (rst)  
    (a && b ) |-> ##2 ack);
```

```
// if both a and b are high, then ack is high after 2 clock cycles from this cycle.
```

```
Check_ack: assert property (@posedge (clk) disable iff (rst)  
    (a && b ) |=> ##1 ack);
```

Those expressions above are equivalent. There is no need to use both the |-> and |=> operators, as one of them would be sufficient to convey the assertion meaning as long as we specify the correct number of cycles. To maintain consistency, we recommend using the |-> Overlapping operator.

Some SVA operators

Operator	Semantics
\$onehot	Returns true if the expression has exactly one bit set to 1. Otherwise, it returns false.
\$onehot0	Returns true if there are zero or one bits set to 1 in the expression.
\$isunknown	Returns true if any bit in the expression is 'X' or 'Z'.
\$stable	Returns true if the expression's value remained unchanged; false otherwise.
\$rose	Returns true if the LSB of the expression changes to 1, otherwise false.
\$fell	Returns true if the LSB of the expression changes to 0, otherwise false.
\$past(expression, number of cycles)	This function returns the value of an expression from a specified number of cycles ago.

Some SVA operators

##n Delay operator, delay n number of cycles.

[m:n] Delay this fixed time interval from m to n

!, |, && Boolean operators

not, or, and Property operators.

More SVA examples used in our design:

// Check if the output at State S0 is 2'b00

```
check_out_S0: assert property(  
    @(posedge clk) disable iff (rst)  
    (State==S0 |-> out==2'b01));
```

// Check if the output at State S0 is 2'b01

```
check_out_S3: assert property(  
    @(posedge clk) disable iff (rst)  
    (State==S3 |-> out==2'b01));
```

// Check if we get next state S1 the next clock cycle

// if we're in S0 and the input a is high

```
check_state_S0: assert property(  
    @(posedge clk) disable iff (rst)  
    (State==S0 && a) |-> ##1 (State==S1)
```

Synopsys® VC Formal Tutorial

Data Path Validation (DPV)

Version 1.2 | 15-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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

Table of Contents

1. Introduction.....	4
1.1 About and Usage of DPV	5
1.2 Design Files.....	7
2. TCL File Syntax	8
2.1 TCL File Examples	10
2.2 TCL: Assumes, Lemmas, and Covers	11
2.3 TCL: Case Splitting	12
3. Application Setup.....	16
3.1 Invoking VC Formal Along with TCL File:	17
4. Application Usage.....	18
4.1 Running DPV Formal Proofs	20
4.2 Debugging - Waveforms.....	23
4.4 Restarting VC Formal.....	24
Appendix	25

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.

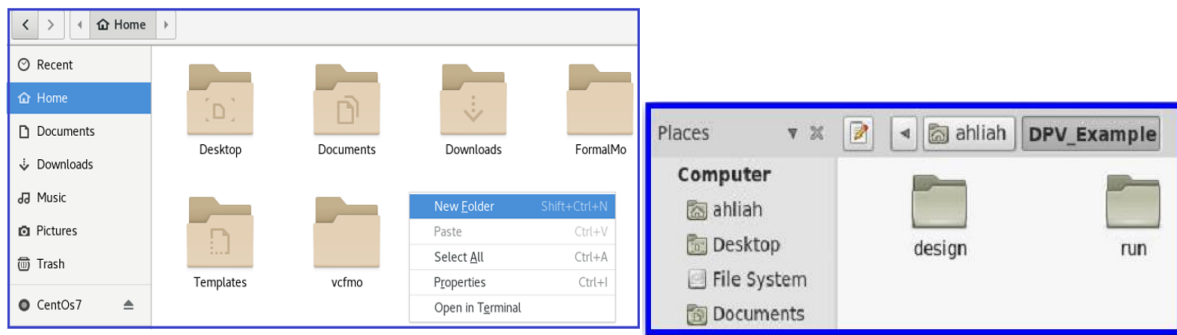


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of DPV

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.

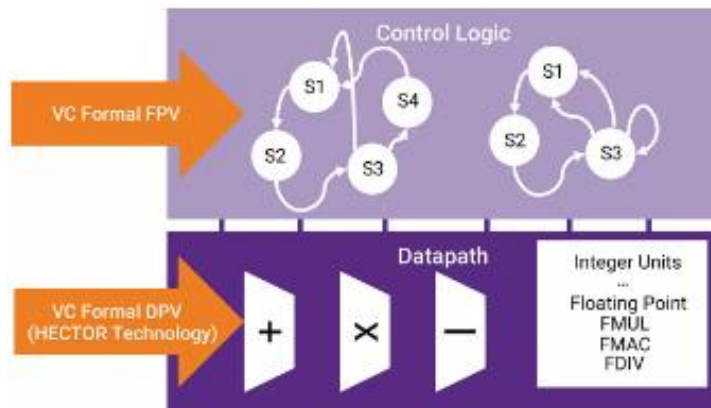
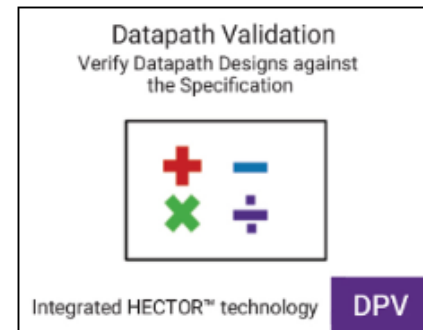


Figure 1.1.2. A visual image of the difference between FPV and DPV.

These DPV figures were taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

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.

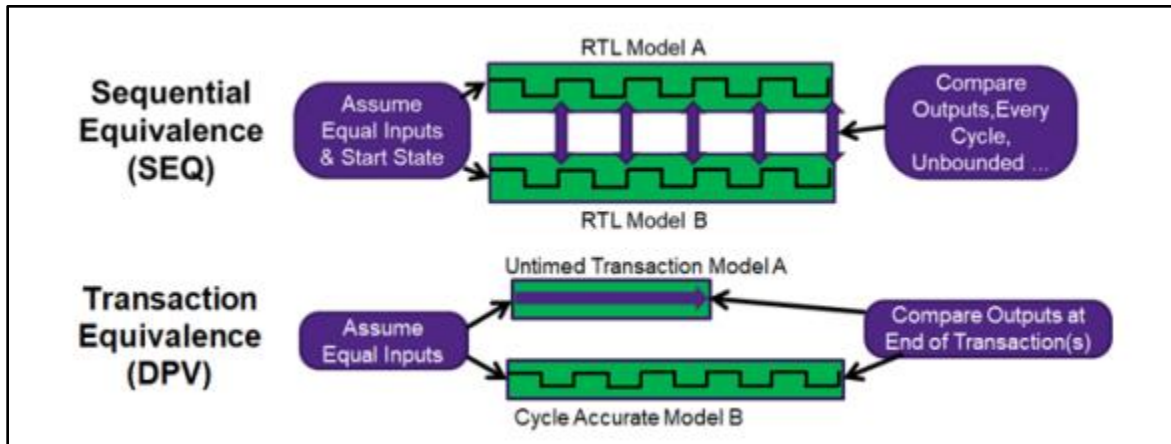


Figure 1.1.3. A visual image of the comparison between SEQ and DPV.

These DPV and SEQ figures were taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

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.

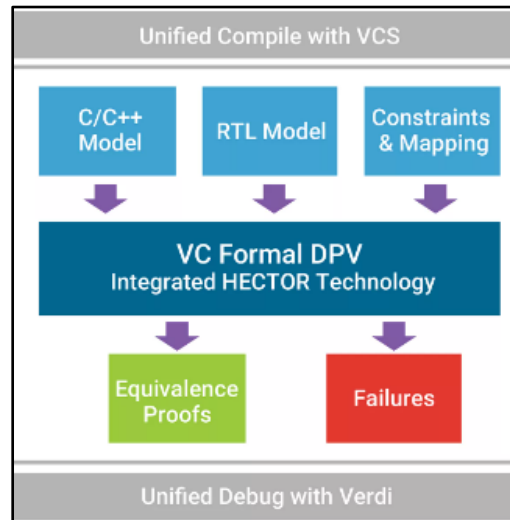


Figure 1.2.1. Showing the required files needed; C/C++, RTL, and TCL file that holds our constraints & mapping.

This figure was taken from the Synopsys (<https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html>)

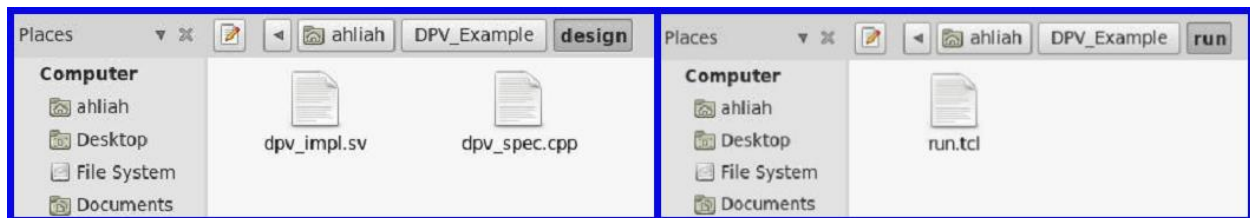


Figure 1.2.2. Showing the (Left) RTL/C++ and (Right) TCL files in the correct folder locations.

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:

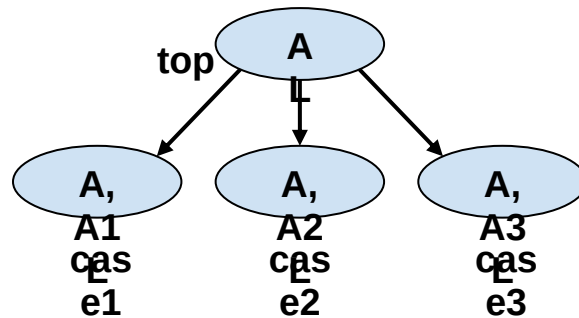


Figure 2.3.1. Case split tree diagram. ‘A’ stands for assumptions, and ‘L’ stands for a lemma to be proven.

This figure was taken from the VC Formal User Guide DPV App Version T-2022.06-SP2, December 2022

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:

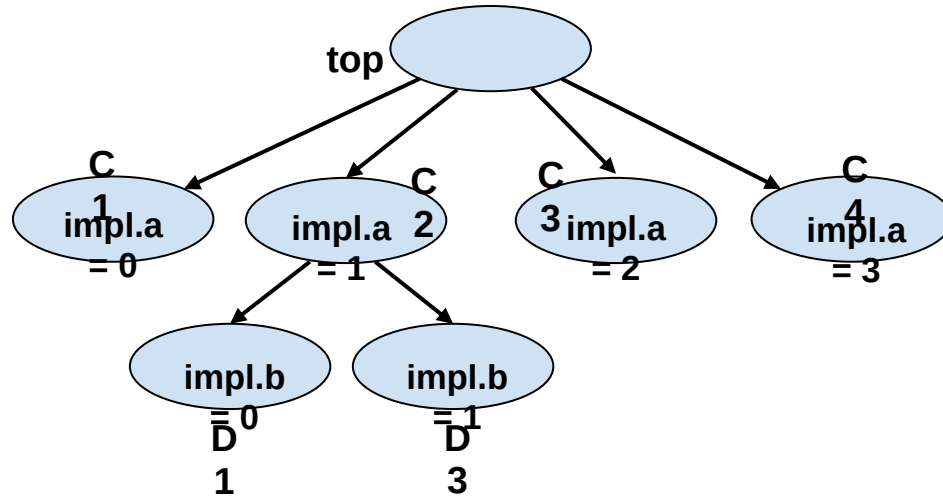


Figure 2.3.2. Case splitting strategy diagram for example “bar”

This figure was taken from the VC Formal User Guide DPV App Version T-2022.06-SP2, December 2022

```

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.

3. Application Setup

The DPV application uses Hector technology, so currently we can only invoke DPV via the shell command prompt. There are multiple 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
```

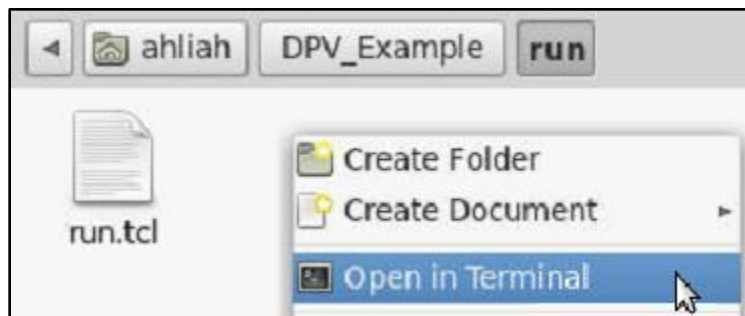


Figure 3.1.1. Opening terminal in the ‘run’ folder.

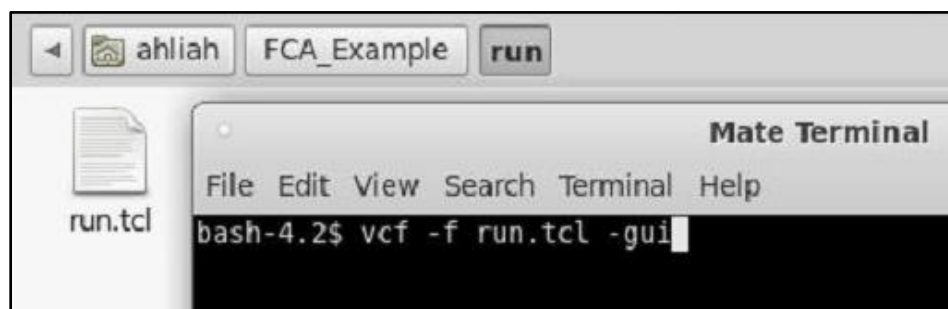


Figure 3.1.2. Invoking VC Formal and TCL script in the terminal.

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:

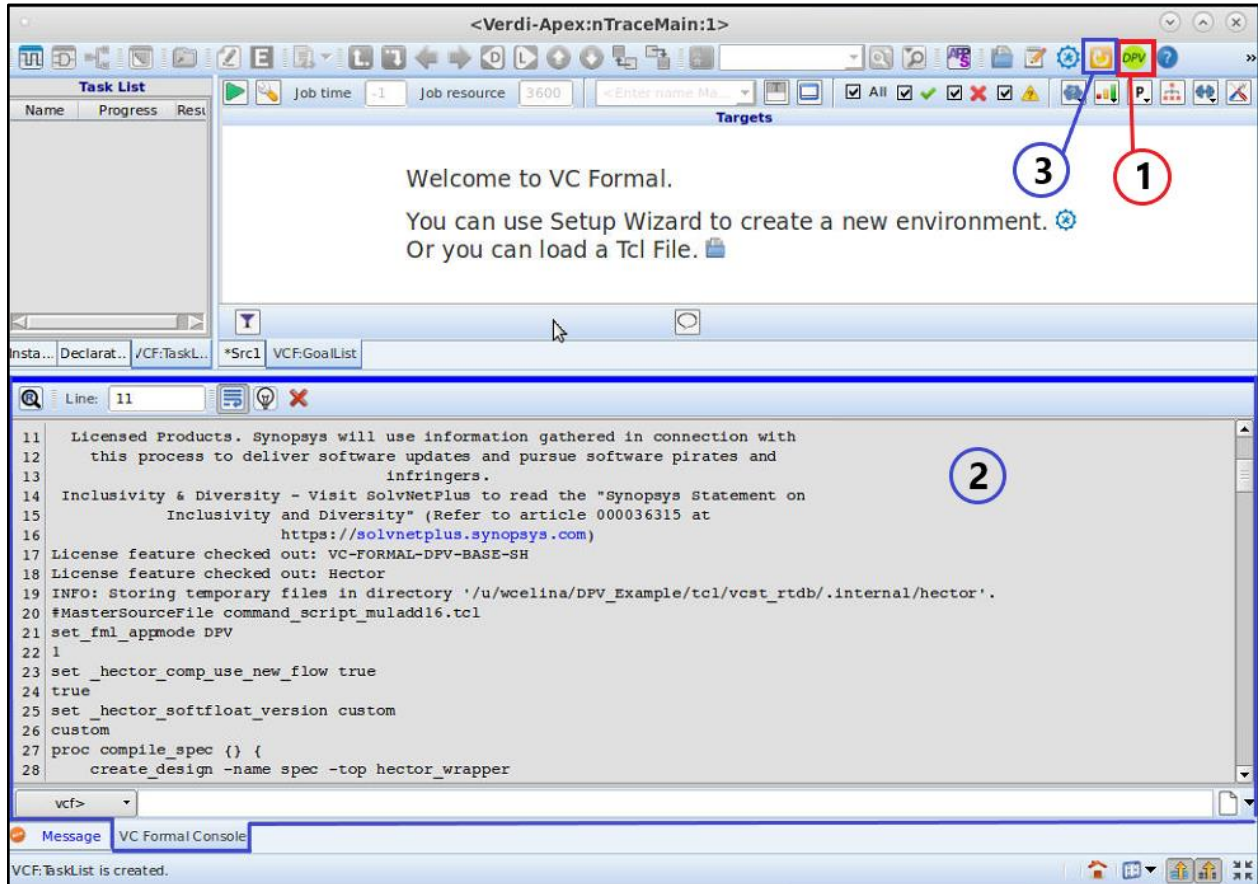


Figure 4.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

- 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:

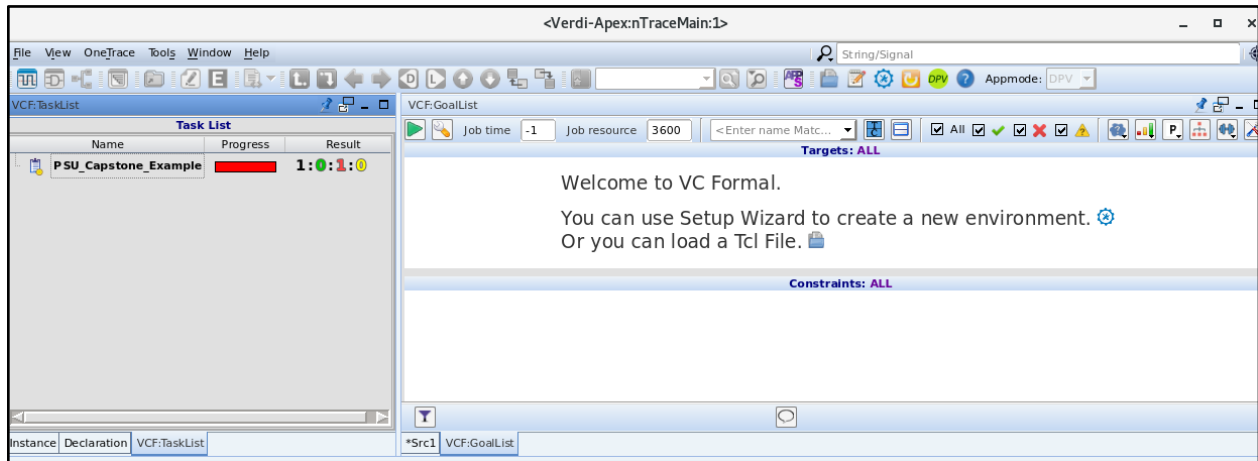


Figure 4.1.1. Capture of GUI example 1. Targets/constraints are not populated, but there is an item in the task list. (This screenshot was taken from the VC Formal GUI)

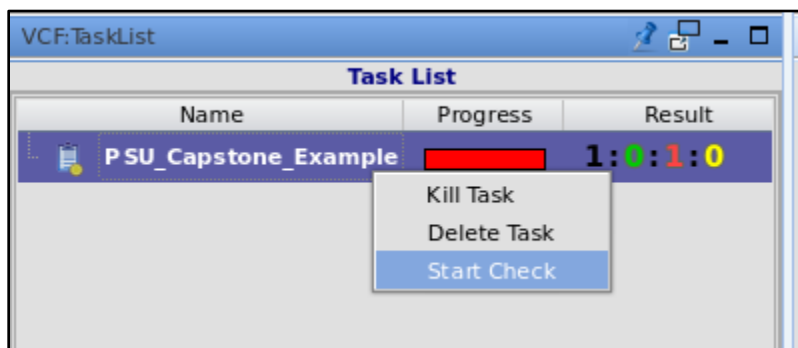


Figure 4.1.2. Capture of GUI example 1. Start the check by right clicking on the task. (This screenshot was taken from the VC Formal GUI)

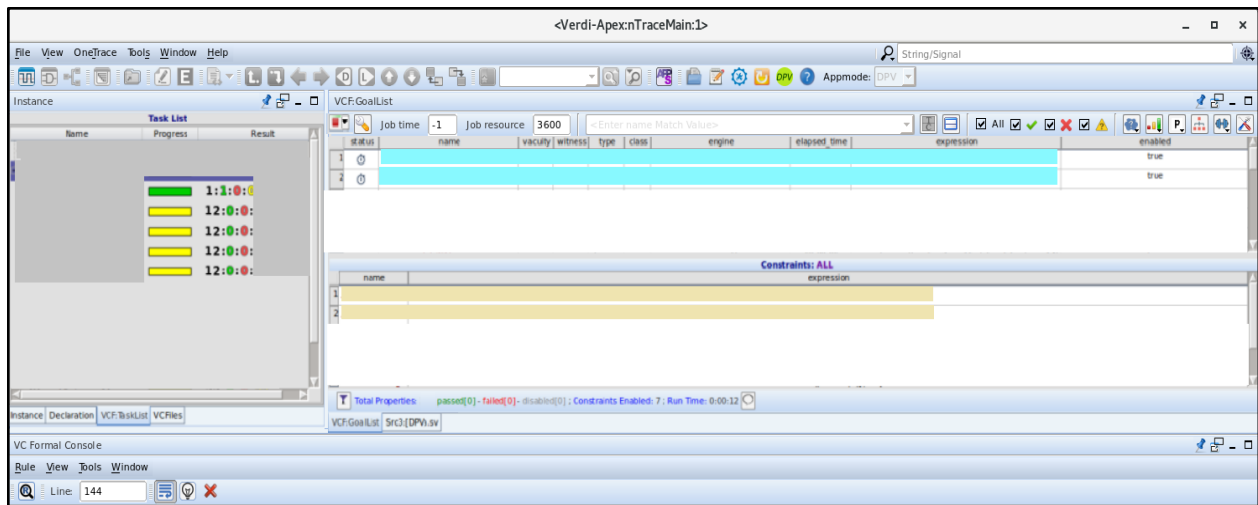


Figure 4.1.3. Capture of GUI example 2, after executing proc names sequentially in the VCF-shell. The targets, and constraints tables will be populated with the properties and constraints specific to your project. The task list on the left will also be specific to your project. (This screenshot was taken from the VC Formal GUI)

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

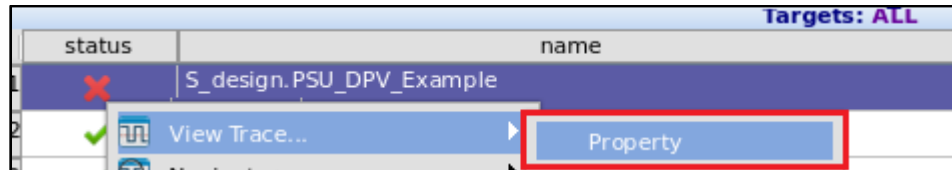


Figure 4.2.1. Inspecting falsified lemmas. (This screenshot was taken from the VC Formal GUI)

After clicking on property, a waveform window should open:

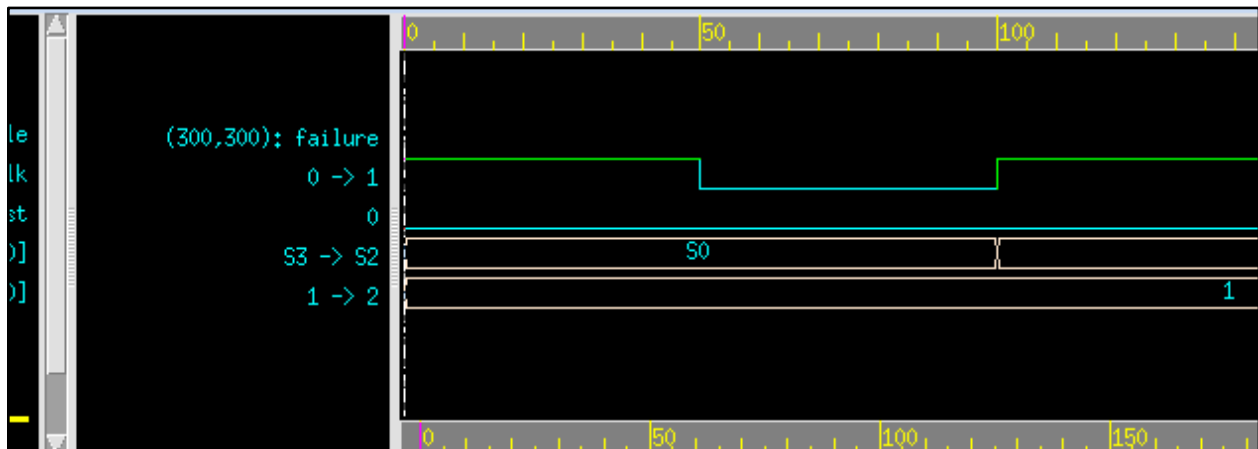


Figure 4.2.2. Producing the waveform of a falsified lemma. (This screenshot was taken from the VC Formal GUI)

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. (This screenshot was taken from the VC Formal GUI)

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. (This screenshot was taken from the VC Formal GUI)

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_sequential	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	<pre> proc compile_impl {} { create_design -name -impl -top play \ -clock clk -reset rst -negReset vcs play.v compile_design impl } </pre>
SystemVerilog	<pre> proc compile_impl {} { create_design -name -impl -top play \ -clock clk -reset rst -negReset vcs -sverilog play.v compile_design impl } </pre>
VHDL (ver. 1)	<pre> 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 } </pre>
VHDL (ver. 2)	<pre> proc compile_impl {} { create_design -name -impl -top testmultipipe \ -reset rst -negReset -clock CLK vhdlan mult_pipeline.vhd compile_design impl } </pre>
Mixed Verilog and VHDL (MX)	<pre> 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 } </pre>

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 in the 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.

Synopsys® VC Formal Tutorial

Formal Register Verification (FRV)

Version 1.1 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of FRV	4
1.2 Design Files.....	5
1.3 TCL File	7
2. Application Setup	9
2.1 Invoking VC Formal GUI	10
2.2 Invoking VC Formal Along with TCL File:.....	12
3. Application Usage	13
3.1 Detecting Errors	14
3.2 Resolving Errors	17
3.3 Restarting VC Formal	20
Appendix.....	21

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 “FRV_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 FRV analysis for us.

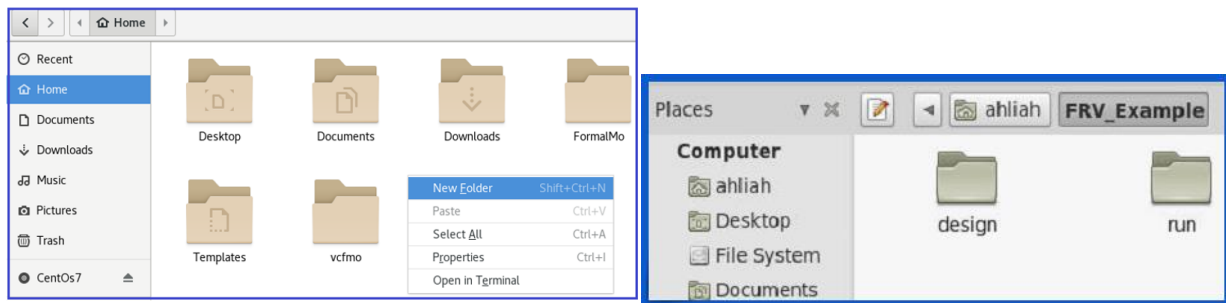
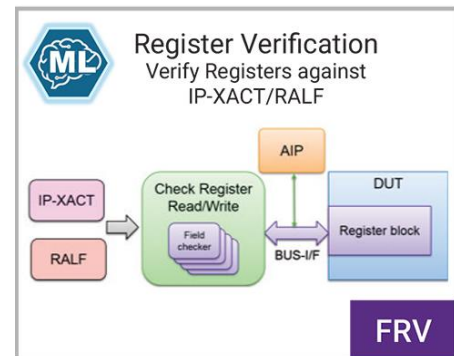


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of FRV

Formal register verification is used to perform formal verification checks for registers in a design. It cuts down the time of performing directed simulation tests by formally verifying register behavior of a design configuration. Intended register behavior is provided by the user via IP-XACT (.xml file) or RALF (.ralf file) format. This register specification file will be a separate file that compliments your main design file(s) (SV or V file). It will be where you define all registers and/or reg-fields.



This FRV figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

During the verification process, FRV will determine if register intent is consistent by checking read/write transactions. Once FRV is initiated, a checker file will be produced. The checker file is what VC Formal uses in order to check on your specified registers. To ensure FRV performance, an additional file is needed to “bind” (SV file) this checker file and your register specification file.

1.2 Design Files

In the Design folder, you'll put the design files. 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.

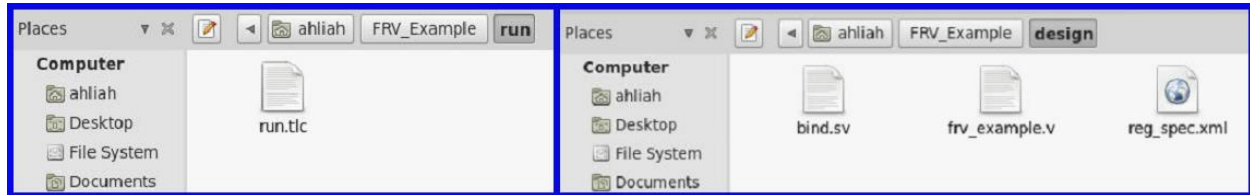


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

Shown left of *Figure 1.2.1*, the design folder should hold **3 files**:

- ❖ A SystemVerilog or Verilog file containing your main module
- ❖ A .xml or .ralf file containing your register specifications
- ❖ A SystemVerilog file that will “bind” main module file to checker file



Figure 1.2.2. Example of the main design module 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 AND your file name**, as highlighted in *Figure 1.2.2* above. Although I chose to keep the same name for both in this example, it is important to know the difference. This will come in handy when you create your TCL file.

The register specification file is where you define all registers in your main design file in XML or RALF format.

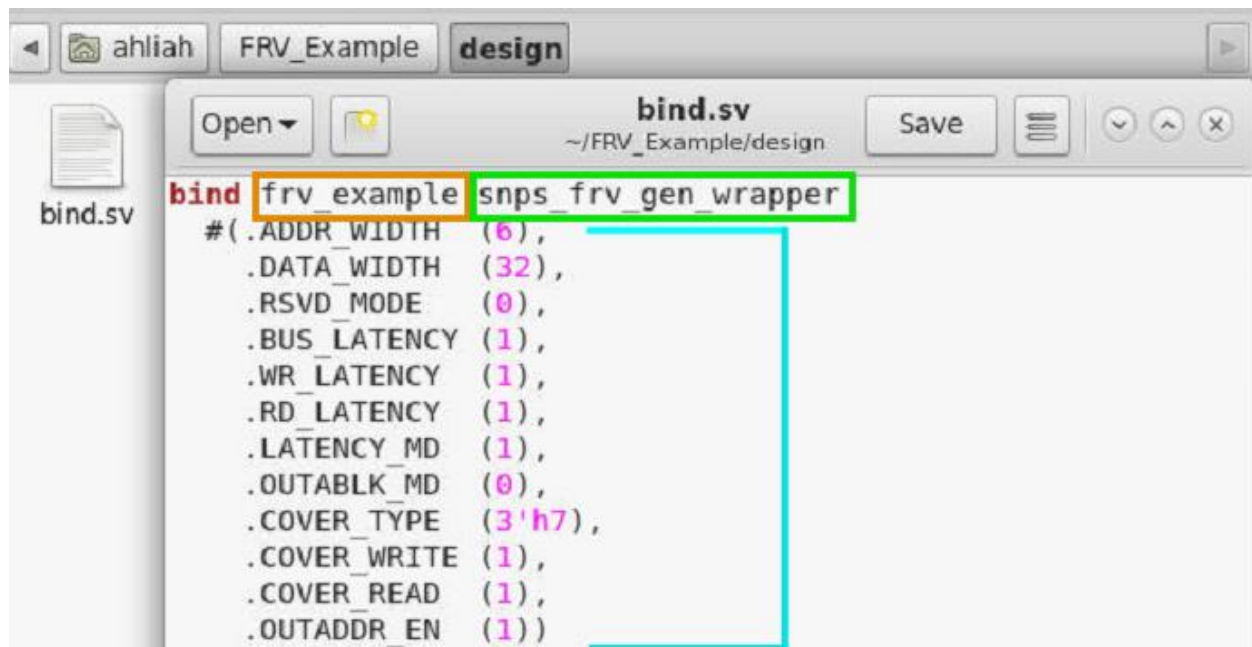


Figure 1.2.3. Example of the *bind* file we are using in this tutorial.

The “*bind*” file will bind the checker file (generated after running TCL file) to your design.

Things to note from *Figure 1.2.3*:

- ❖ The orange highlight is our design module name
- ❖ The green highlight is a generic bus wrapper (recommended)
- ❖ The blue highlight encapsulates the checkers parameters (see Appendix Table 1.1)

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (full script in *Figure 1.3.2*), which you can use as a template for your functional checks on VC Formal.

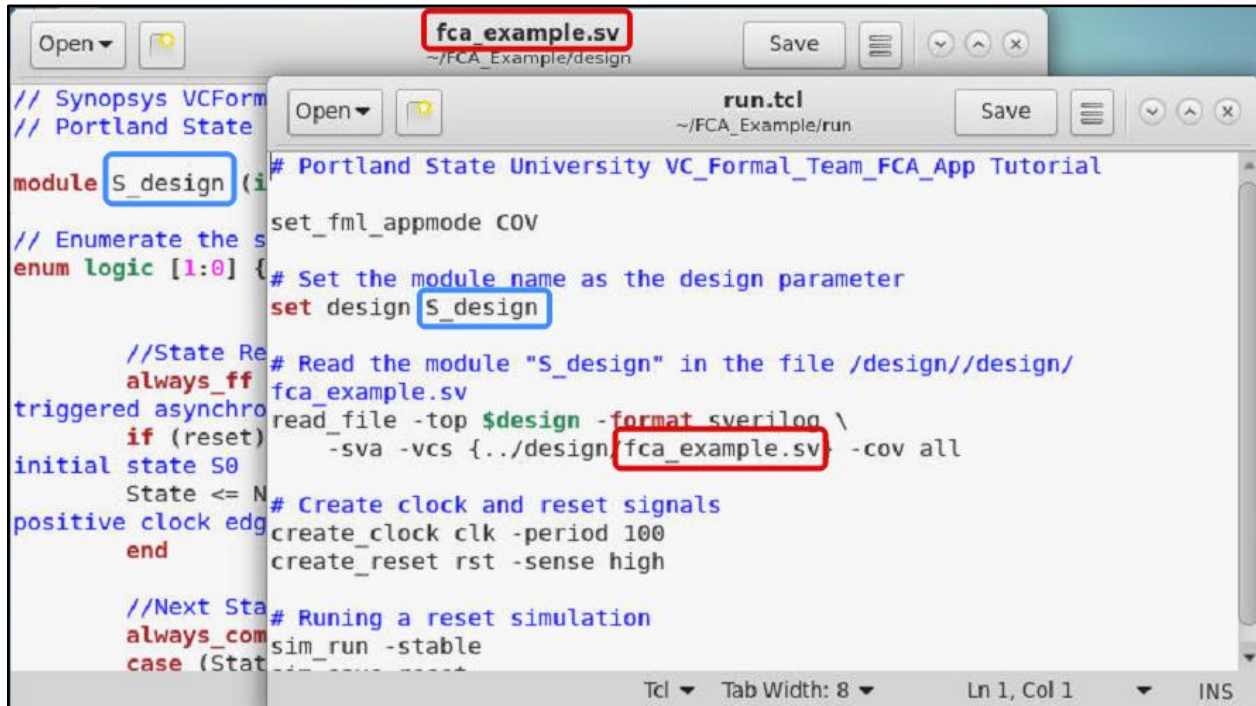


Figure 1.3.1. Difference between the design file name and module name.

In *Figure 1.3.1*, both the Design file and TCL files are shown side by side. This is an example from another VC Formal application; this is **NOT** a TCL file for the FRV app.

- ❖ The blue highlights are to demonstrate the **module name**.
- ❖ The red highlights are to demonstrate the **design file name**.

These will **NOT** be the same for every user. You will need to keep track of your own module and design file names in order for your TCL file to work correctly when you try to run in VC Formal.



```
# Portland State University VC_Formal_Team_FRV_App_Tutorial

set_fml_appmode FRV

# Set the module name as the design parameter
set design frv_example 1

# Read register specification file then generate checker file
frv_load -ipxact ../design/reg_spec.xml -auto_load 2

# Read the module "FRV_design" and bind it to checker file
read_file -top $design -format sverilog \
    -sva -vcs "../design/frv_example.v" ../design/bind.sv 3 4

# Create clock and reset signals
create_clock clk -period 100
create_reset rst -sense high

# Running a reset simulation
sim_run -stable
sim_save_reset
```

Figure 1.3.2. Annotated TCL template file.

- (1) Instruction that sets the appmode to FRV in VC Formal.
- (2) “frv_load” identifier that perform FRV by taking in the register specification file and verifying specified registers in the file.
- (3) Design file location so VC Formal knows where to find the file.
- (4) Bind 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 (3) and the module name in the design file.

Any file name (ex. frv_example.sv) can be named however you want.

2. 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 FRV 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.

2.1 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:

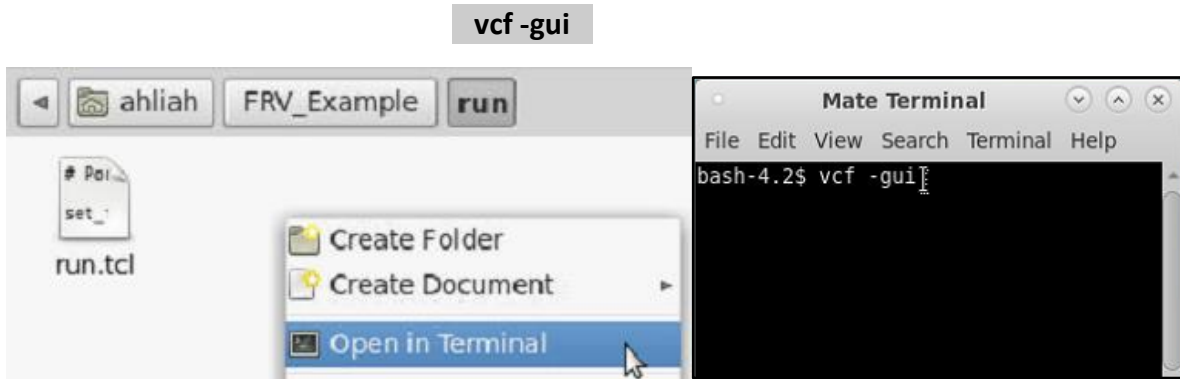


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2* 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: 

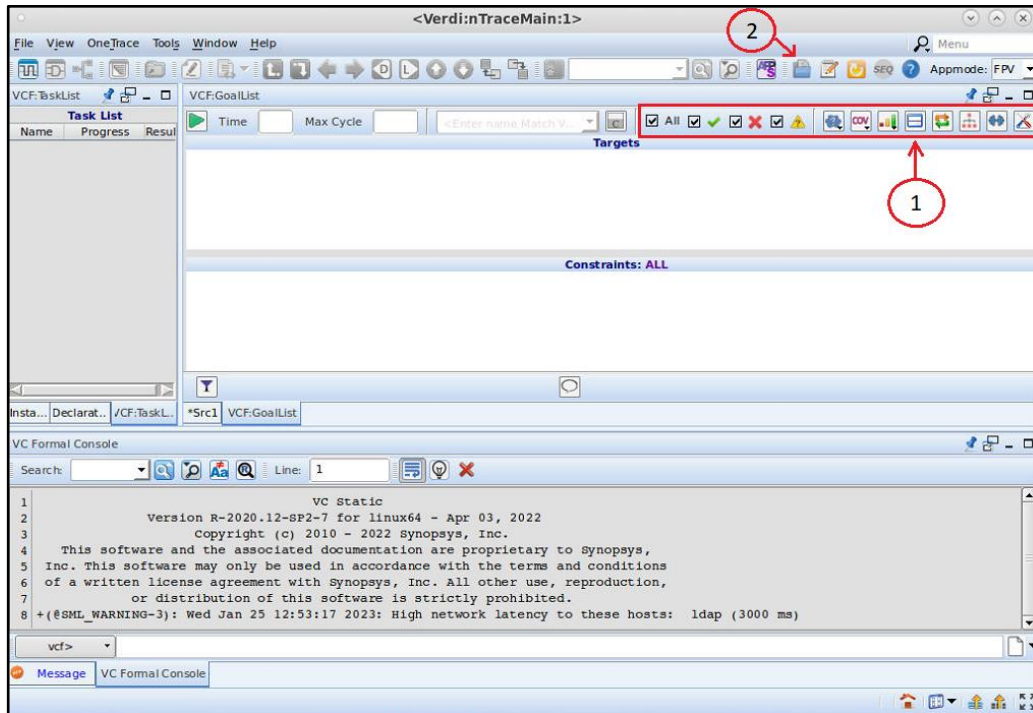


Figure 2.1.2. VC Formal GUI introductory screen. (This screenshot was taken from the VC Formal GUI)

Then load a TCL script by clicking on the  icon **(2)** as shown in Figure 2.1.2.

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

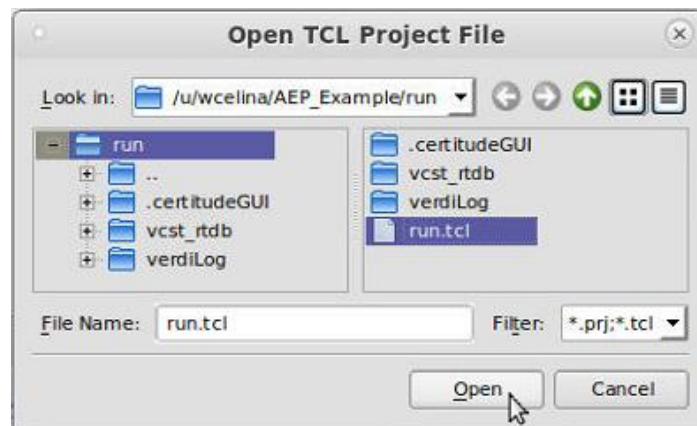


Figure 2.1.3. Selecting TCL file.

2.2 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
```

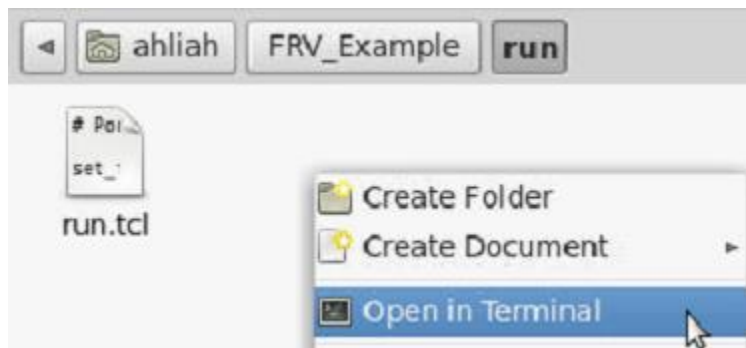


Figure 2.2.1. Opening terminal in the ‘run’ folder.

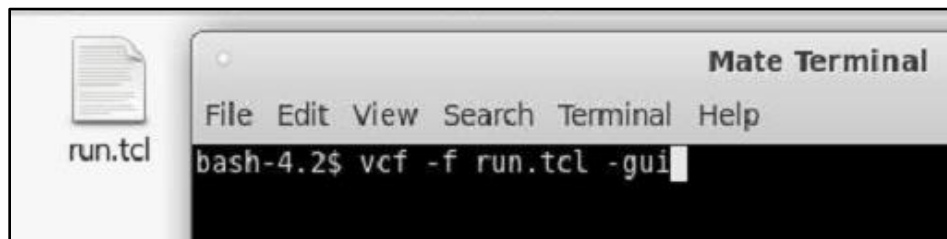


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

3. Application Usage

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:

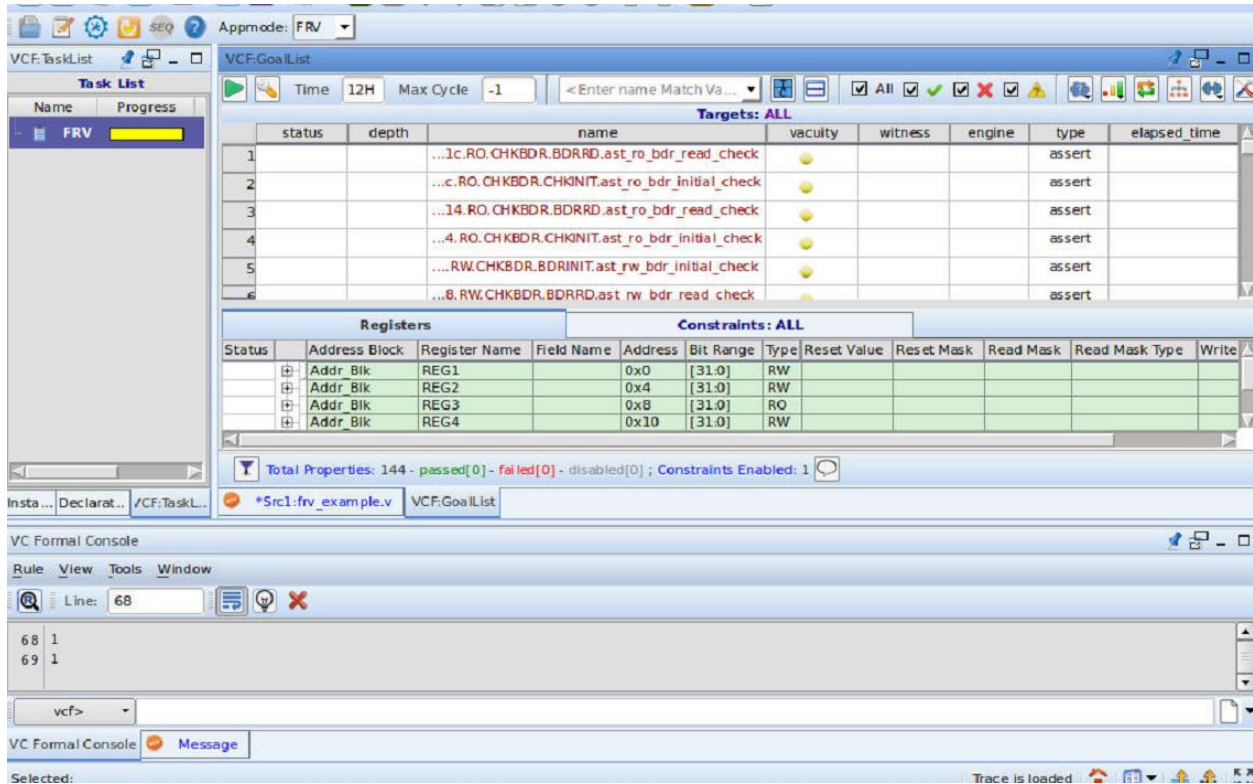


Figure 3.1. Screen after loading TCL script. (This screenshot was taken from the VC Formal GUI)

Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the “VCF:GoalList” tab.

3.1 Detecting Errors

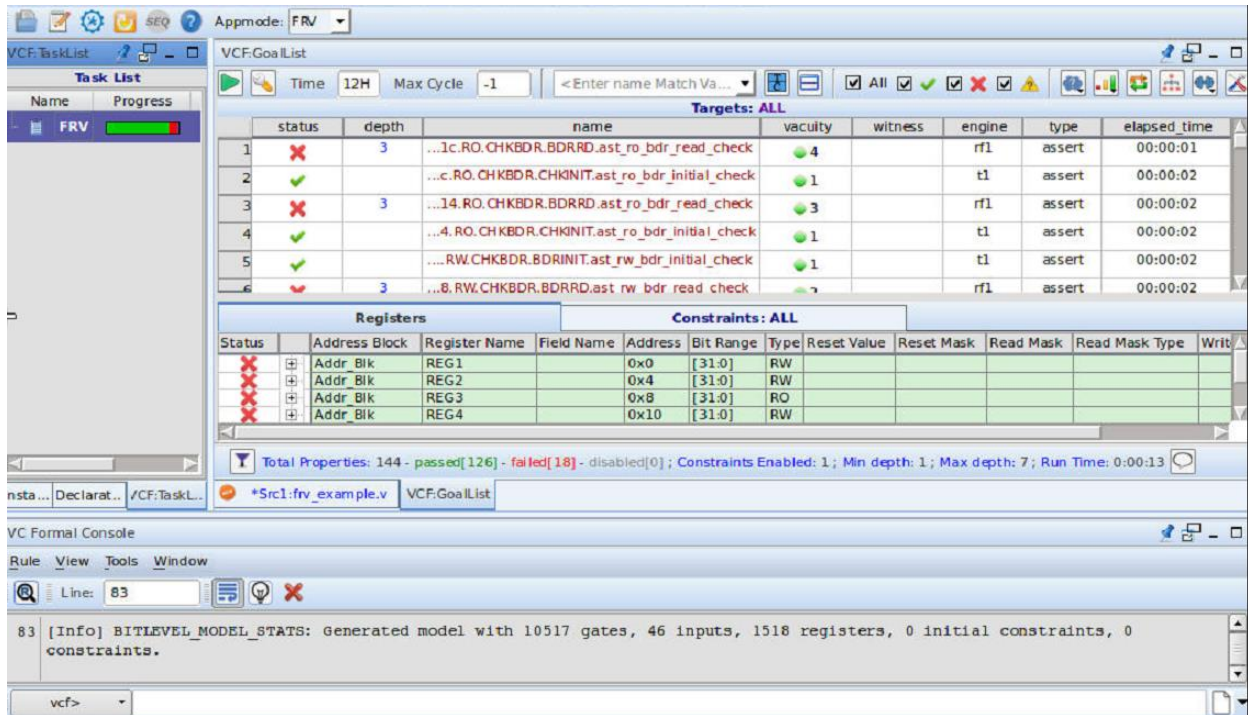


Figure 3.1.1. Screen after running TCL script and the status given = ✗. (This screenshot was taken from the VC Formal GUI)

In Figure 3.1.1 above, under the VCF: GoalList tab, we see icons. VC Formal gives this “Proven” status icon for the part of our design that ran through FPV successfully.

We also see one icon; VC Formal gives this “Falsified” status icon for the part of our design that failed FRV.

Under the Registers tab, the same statuses can be seen. Other information such as “Address block”, “Register Name”, “Field Name”, “Address”, “Bit Range”, and “Type” display the default register attributes from this analysis. The other columns can provide further information but it must be declared in your register specification file. These include:

- ❖ Reset Value
- ❖ Reset Mask
- ❖ Read Mask (and type)
- ❖ Write Mask (and type)
- ❖ Backdoor

On the left under “Task List”, we can see that VC Formal was given one task by the FRV 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:

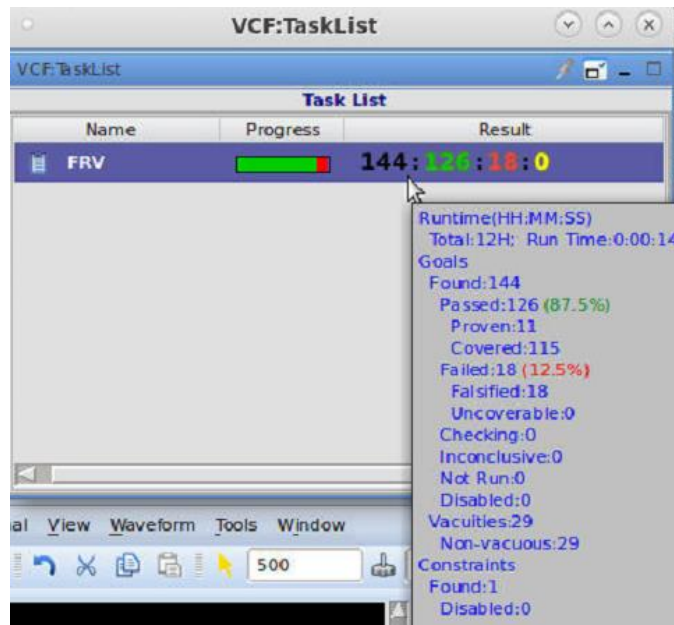


Figure 3.1.2. Results of the analyzed script. (This screenshot was taken from the VC Formal GUI)

We can start by looking at the register data in the FRV report. Use the command below in the VC Formal Console window to see the report:

frv_report

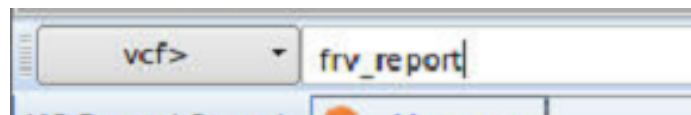


Figure 3.1.3. VC Formal Console window.


```

86 vcf>frv_report
87 Register Summary:
88   Component      : frv_example
89   MemoryMap/System: Example_Code
90   Address Block   : Addr_Blk
91   Base Address    : 0x0
92   Register Count  : 7
93   Assertions      : 29/11/18/0
94   Covers          : 115/115/0/0
95 Register List: 0x0 : frv_example/Example_Code/Addr_Blk
96   ID      Register/field Name      Address/bit offset      S:
97   Asserts/Covers
98 -----
99 4      REG1      0x00      3:
100 16/16/0/0

```

Figure 3.1.4. VC Formal Console showing a report of register data.

The FRV report will show the Register Summary of your design.

- ❖ “*Component*” is the design module under evaluation.
- ❖ “*MemoryMap/System*”, “*Address Block*”, “*Base Address*”, and “*Register Count*” are all provided in the register specification file.
- ❖ “*Assertions*” are in the order of *Total Asserts - Proven Asserts - Falsified Asserts and Unknown Asserts*.
- ❖ “*Covers*” are in the order of *Total Covers - Covered Covers - Uncoverable Covers and Unknown Covers*.

Under the “*Register Summary*” will consist of a “*Register List*” with a table of identified registers.

3.2 Resolving Errors

To see the register where this fault is resulting, we go to the column in between “*Status*” and “*Address Block*” within the “*Registers*” tab and click on the **+** icon

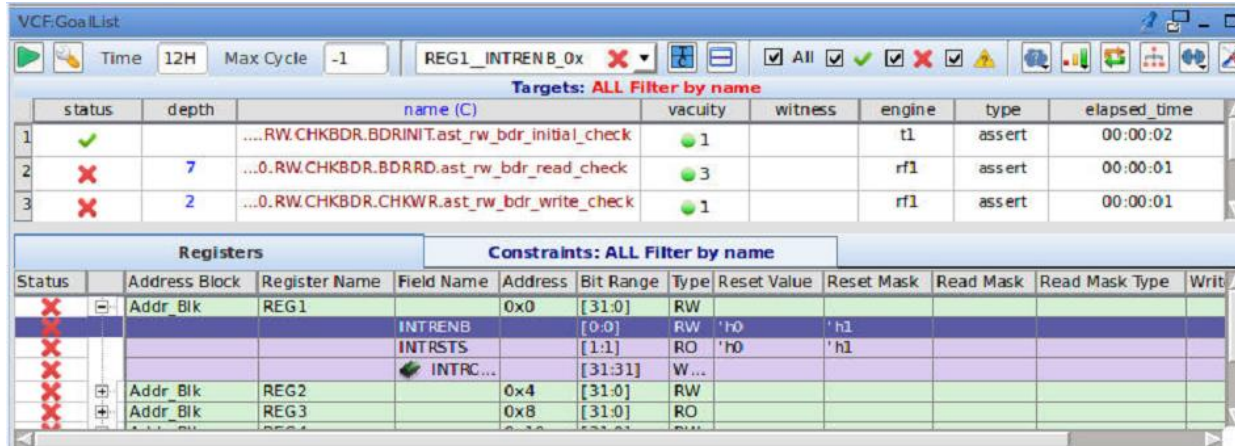


Figure 3.2.1. Under “Registers”, click on **+** icon.

We will then be given the field names within the address block that is failing in our design.

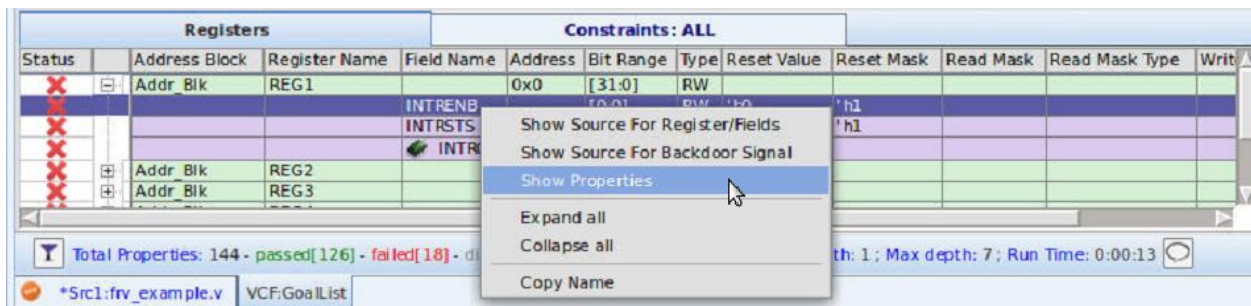


Figure 3.2.2. Field names under the address block that is causing failures.

Right-click on the “*Field Name*” of choice that failed FRV and click “*Show Properties*”. This will show/filter all targets that are associated with the field name you selected (shown in Figure 3.2.1).

Double-click on a failed status icon associated with the filtered “*Targets*”. This will generate a waveform tab.

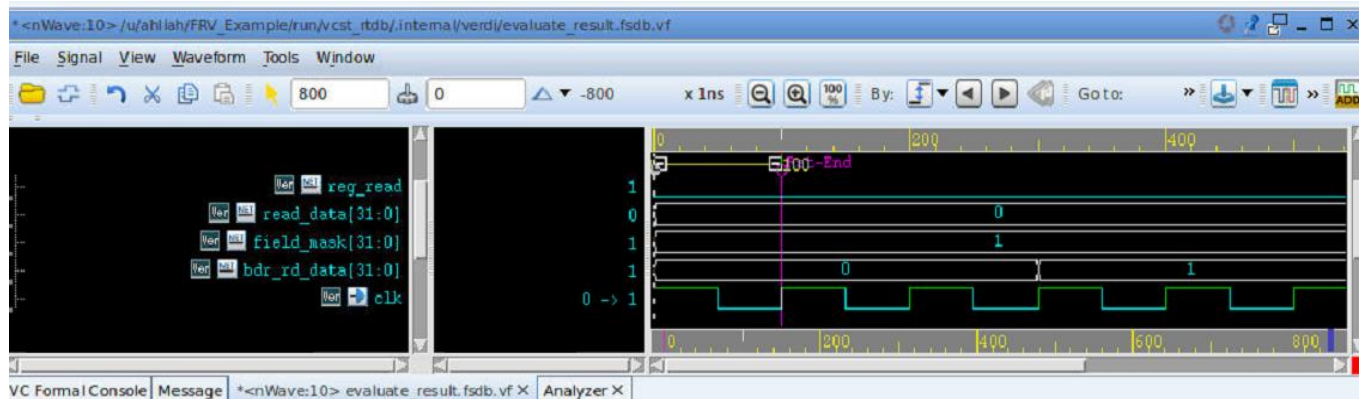


Figure 3.2.3. Waveform window with general traces provided by FRV.

General traces provided by FRV:

- ❖ “**reg_read**” - Goes “high” to indicate register read is complete.
- ❖ “**read_data**” - Indicates read data driven by your design.
- ❖ “**field_mask**” - Indicates the data bits that correspond to the register field that is checked
- ❖ “**bdr_rd_data**” - Indicates the expected data within this register field (unbolded part will be different name for everybody). **If this trace shows a different assertion than reg_data, it will fail.**

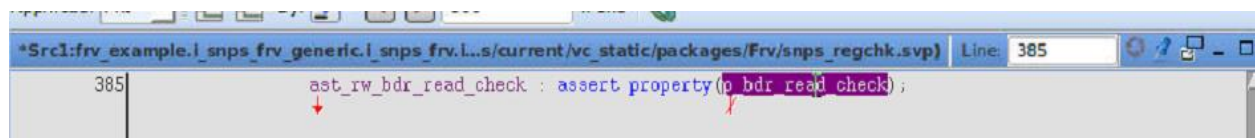


Figure 3.2.3. Source of failure corresponding with property being investigated.

Double-click on a property name and it will take you to the source within the design (shown in Figure 3.2.3). You can then begin back-tracing. This will eventually take you to the line where the signal is being driven and you can identify whether or not this logic is that to be expected; revealing the root-cause. This will follow similar tracing techniques laid out in other VC Formal tutorials.

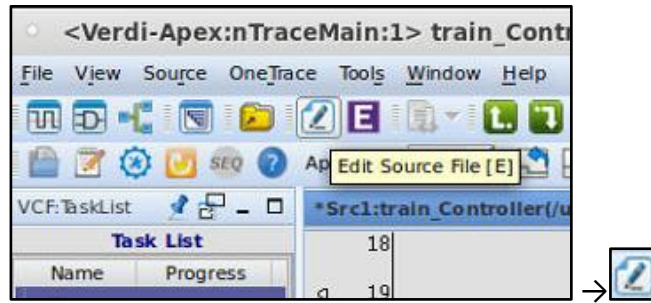


Figure 3.2.4. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes.

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

3.3 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 3.3.1. 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, every status within the GoalList and Registers windows embodying the green checkmark.

Appendix

Parameter Name	Description	Default Value
ADDR_WIDTH	Address bit width.	32
DATA_WIDTH	Data bit width.	32
RSVD_MODE	Specify how to check reserved and/or unmapped registers and/or fields • 0: no check when ACCESS_TYPE==ACS_RSVD . • 1: 0 should be read when ACCESS_TYPE==ACS_RSVD and CHECK_INIT==0 . • 2: 1 should be read when ACCESS_TYPE==ACS_RSVD and CHECK_INIT==0.	1
COMPOSITE	Specify type of assertions per field for initial value and after write • 1: Generate one common assertion (better convergence). • 0: Generate two individual assertions.	1
BUS_LATENCY	Effective only in generic wrapper: Latency from read enable signal to read data valid on bus.	1
WR_LATENCY	Write Latency from bus write data propagates to internal field.	1
RD_LATENCY	Read Latency from internal field to bus read data.	1
COVER_TYPE	When COVER_WRITE==1 • 3'bx1: cover write between write to read window • 3'bx1x: cover write before first write. • 3'b1xx: cover write before first read. When COVER_READ==1 • 3'bx1: cover read between write to read window • 3'bx1x: cover read before first write. • 3'b1xx: cover read before first read.	3'h7
COVER_WRITE	Specify enable or disable write access related cover properties • 0: Disable all write access related cover properties. • 1: Enable all write access related cover properties.	1
COVER_READ	Specify enable or disable read access related cover properties • 0: Disable all read access related cover properties. • 1: Enable all read access related cover properties.	1
OUTADDR_EN	Specify enable or disable out of address range access related cover properties • 0: Disable out of address range access related cover properties. • 1: Enable out of address range access related cover properties.	1

LATENCY_MD	Specify Latency handling • 0: RD_LATENCY parameters are used only in backdoor checks. • 1: RD_LATENCY parameters are used in all checks.	0
OUTABLK_MD	Specify how to check out of address range accesses • 0: check read data should be 0 for out of addressBlock range. • 1: check read data should be 1 for out of addressBlock range. • other: no check for out of addressBlock range.	2
SNPS_AS_ABS	Synopsys Non-Deterministic Abstractions • 0: check all possible scenarios. • 1: check only specific scenario in each step.	0

Table 1. FRV App Checker Parameters. Names, default values, and descriptions of all offered.

Synopsys® VC Formal Tutorial

Formal Security Verification (FSV)

Version 1.2 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of FSV	4
1.2 Design Files.....	5
1.3 TCL File	6
2. Application Setup	7
2.1 Invoking VC Formal GUI	8
2.2 Invoking VC Formal Along with TCL File:.....	10
3. Application Usage	11
3.1 Detecting Errors	12
3.2 Debugging Failures.....	13
3.3 Resolving Errors	16
3.4 Restarting VC Formal	17
Appendix.....	18

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 “FSV_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 VC Formal FSV analysis for us.

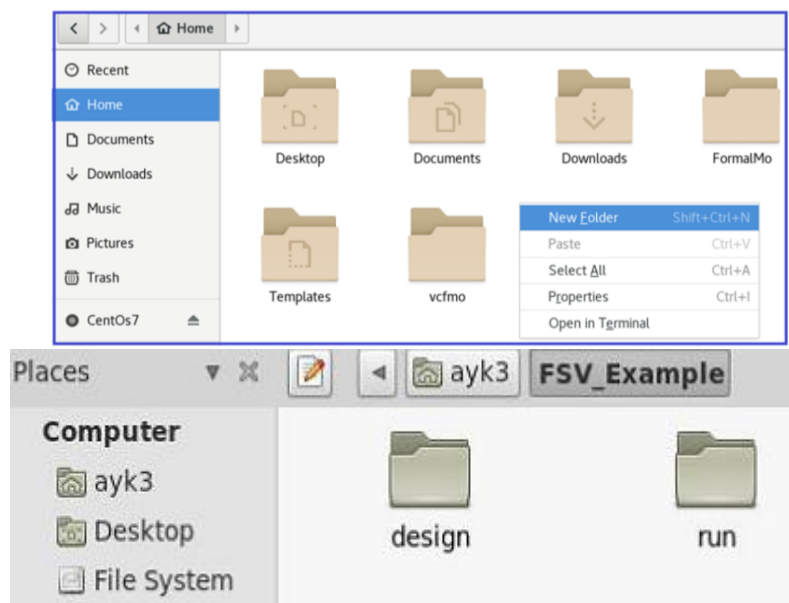
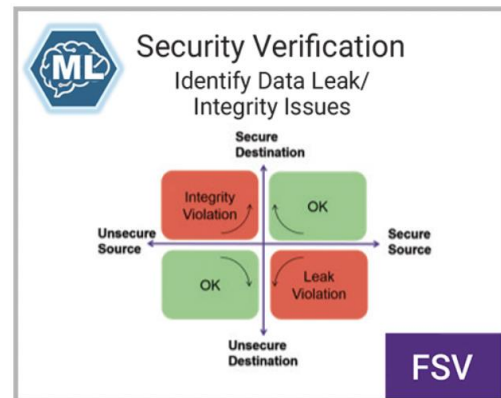


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of FSV

The “FSV” app in VC Formal is used as a verification application to ensure no illegal data propagations are in your design. VC Formal FSV app streamlines the verification process, ensuring the completeness and correctness of designs. By leveraging formal verification techniques, this application empowers designers to identify and resolve complex functional issues efficiently, minimizing the risk of bugs and improving overall design quality.



This FSV figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

1.2 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.

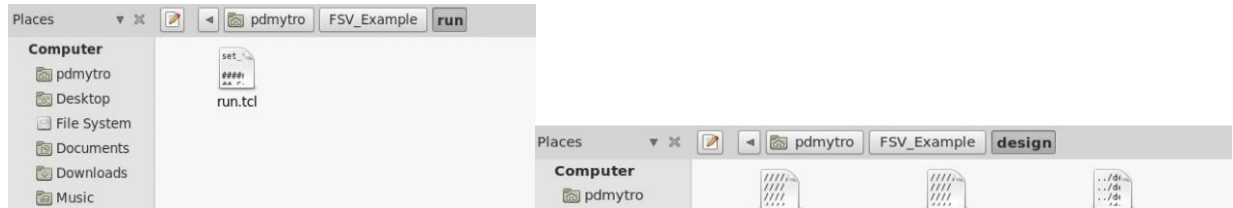


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

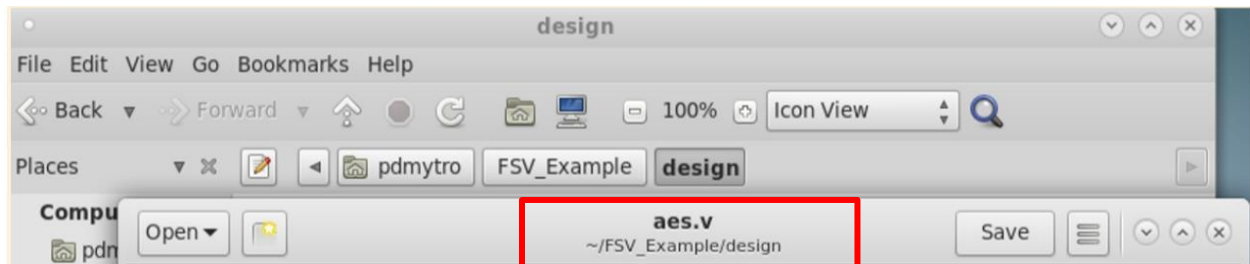


Figure 1.2.2. Example of the design we are using in this tutorial.

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 1.2.2* 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).

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.1*), which you can use as a template for your functional checks on VC Formal.

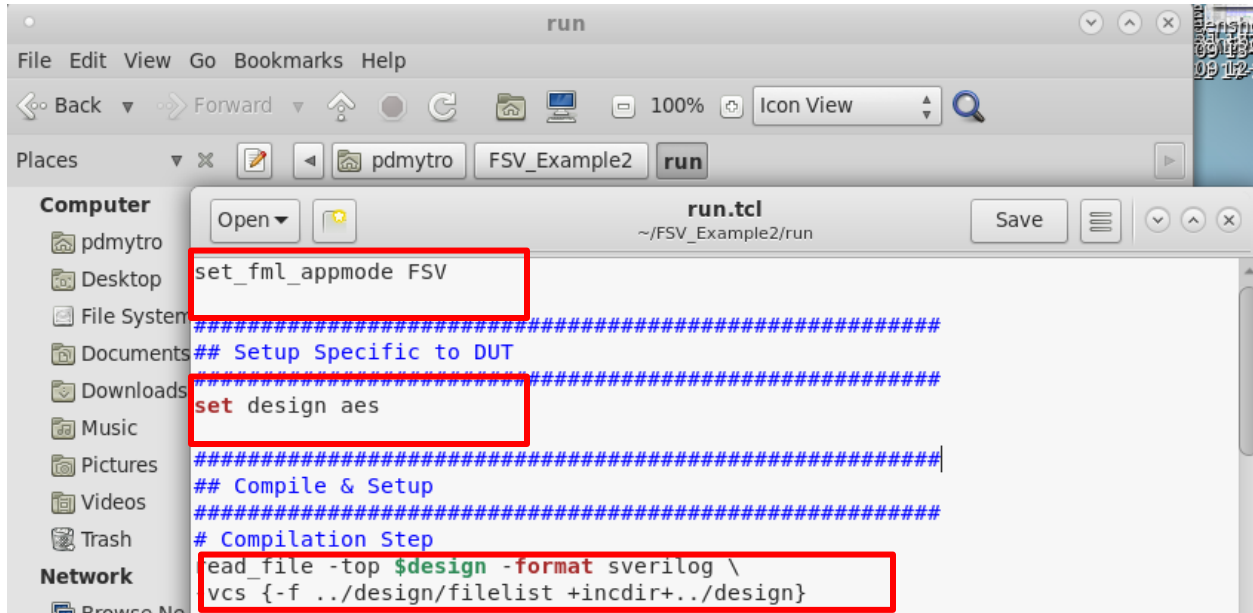


Figure 1.3.1. Annotated TCL template file.

- (1) Instruction that sets the appmode to FSV in VC Formal.
- (2) Name of the main module as established in the Design file.
- (3) The property type identifier switch name for desired FSV analysis.

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 1.2.2*.

The file name (aes.v) can be named however you want.

2. 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 FSV 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.

2.1 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
```

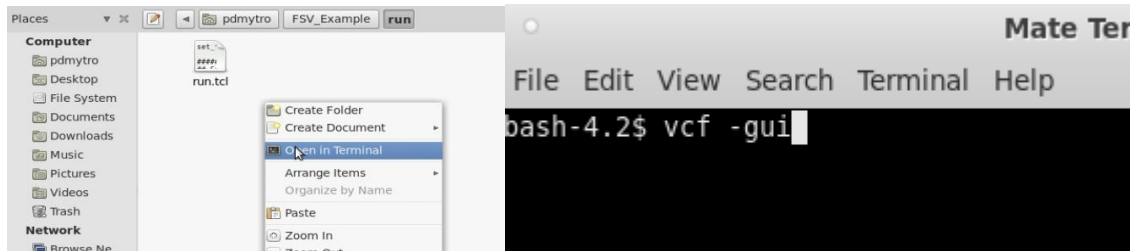


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2* 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: 

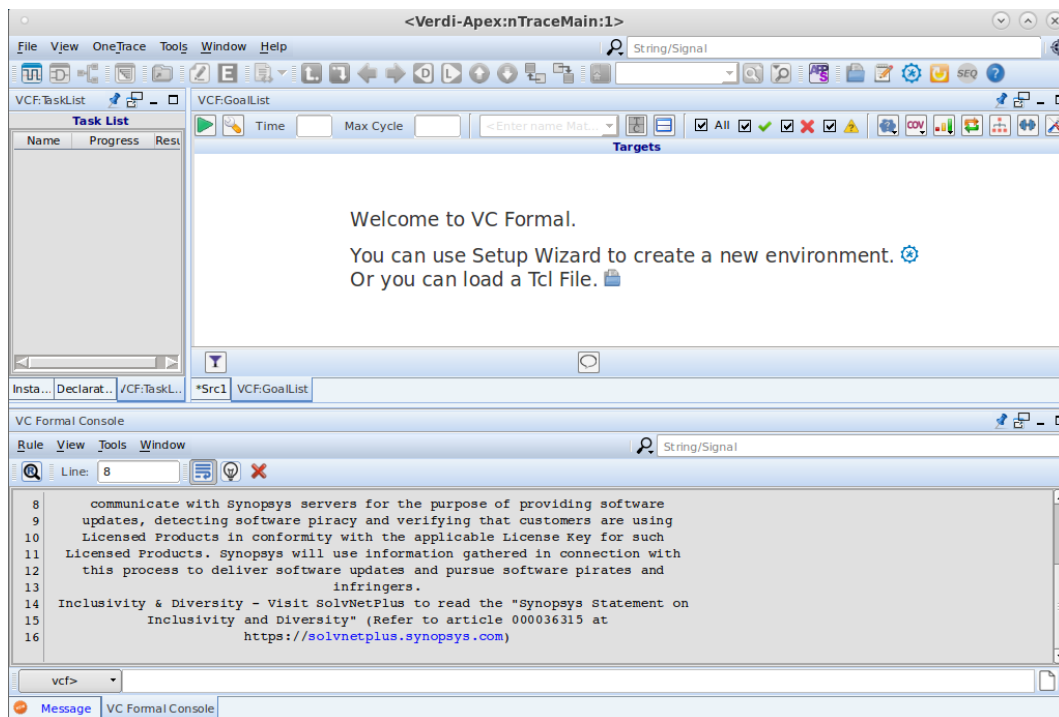


Figure 2.1.2. VC Formal GUI introductory screen.

Then load a TCL script by clicking on the  icon **(2)** as shown in *Figure 2.1.2*.

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

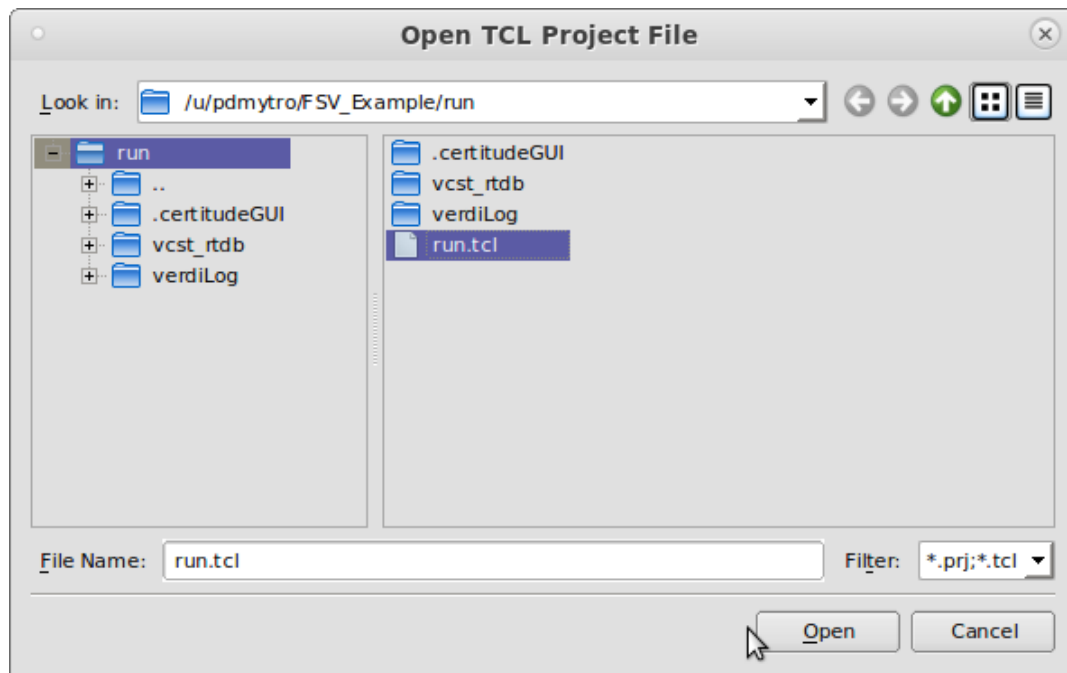


Figure 2.1.3. Selecting TCL file.

2.2 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
```

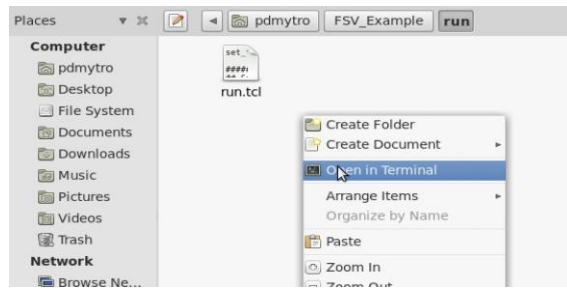


Figure 2.2.1. Opening terminal in the ‘run’ folder.

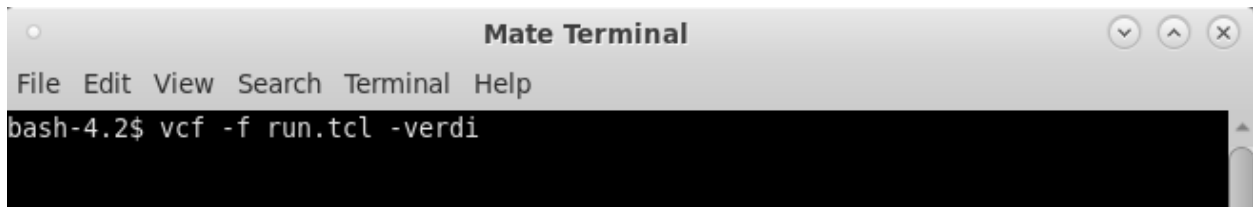


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

3. Application Usage

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:

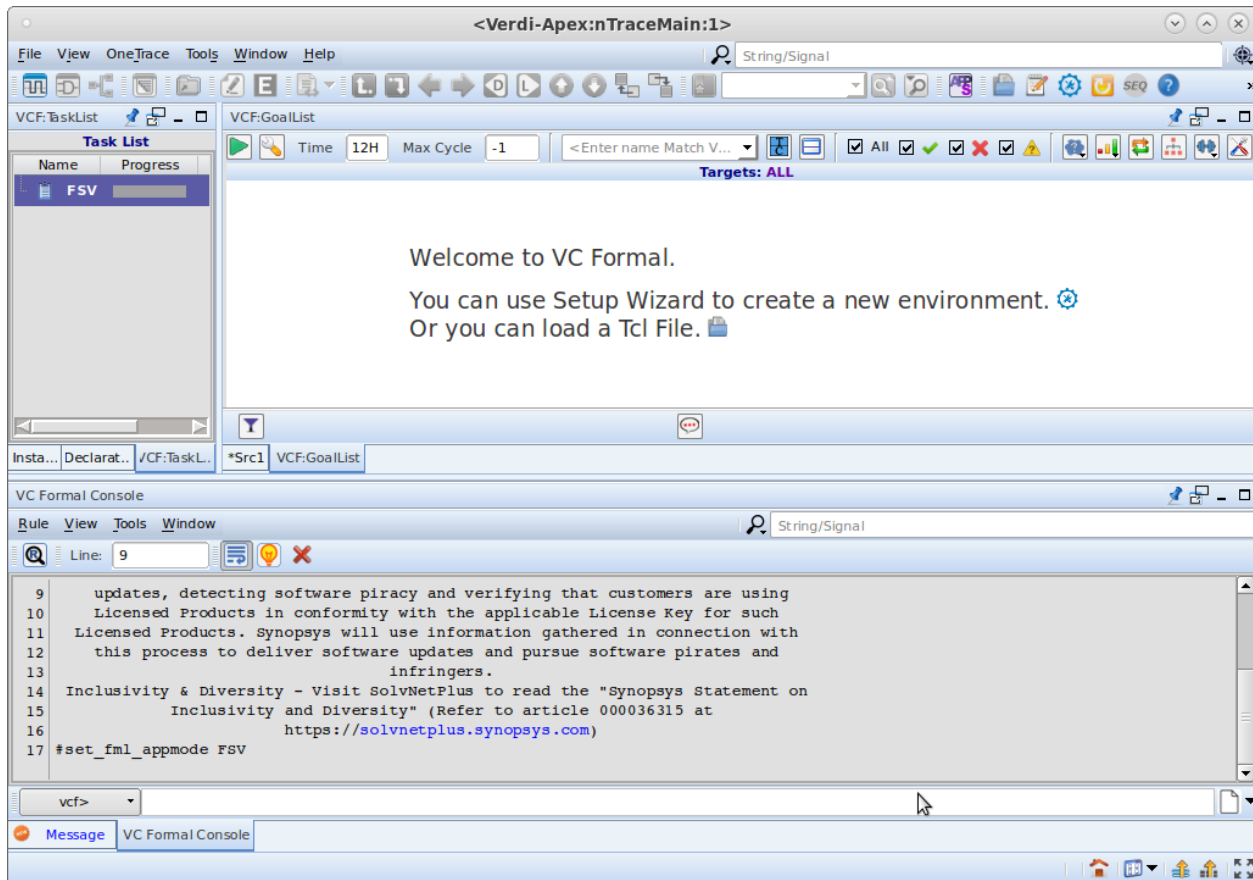


Figure 3.1. Screen after loading TCL script.

Now, go ahead and run the verification analysis by clicking on the play  icon in the upper left corner of the “VCF:GoalList” tab.

3.1 Detecting Errors

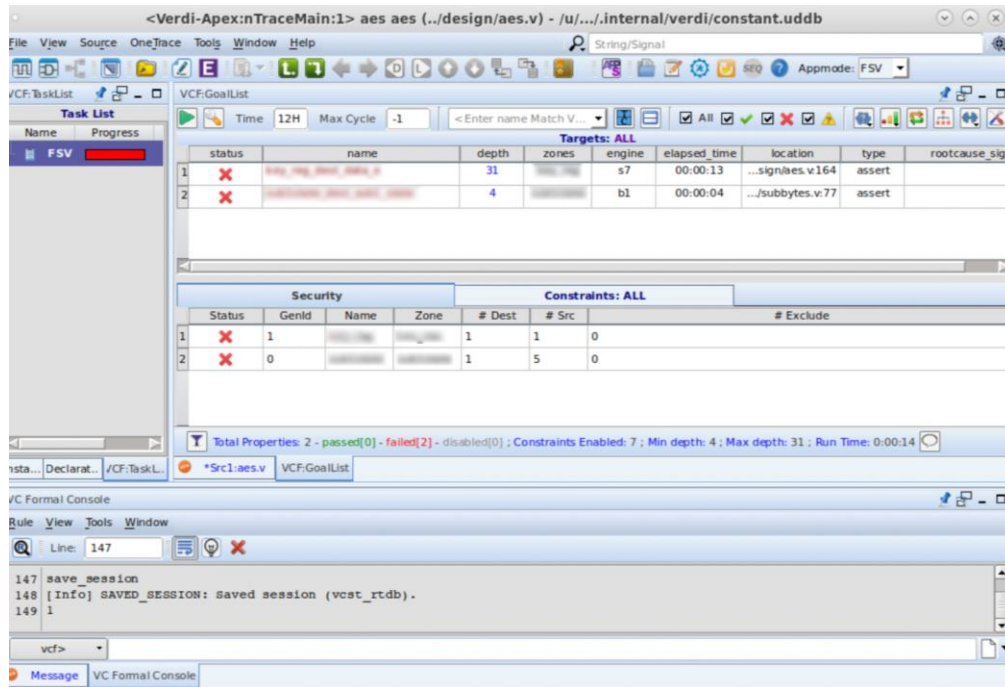


Figure 3.1.1. Screen after running TCL script and the status given = X.

We see two  icons; indicating that there are security properties that are falsified. Debugging falsified properties can help determine which input and outputs are involved.

On the left, under “Task List”, we can see that VC Formal was given one task by the FSV 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:

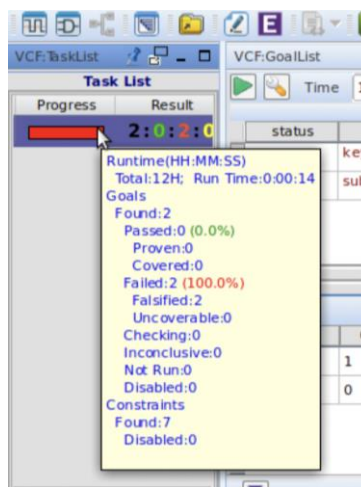


Figure 3.1.2. Results of the analyzed script.

3.2 Debugging Failures

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 right-click on the first  (line 1) from *Figure 3.1.1*. Select View Trace and then Property.

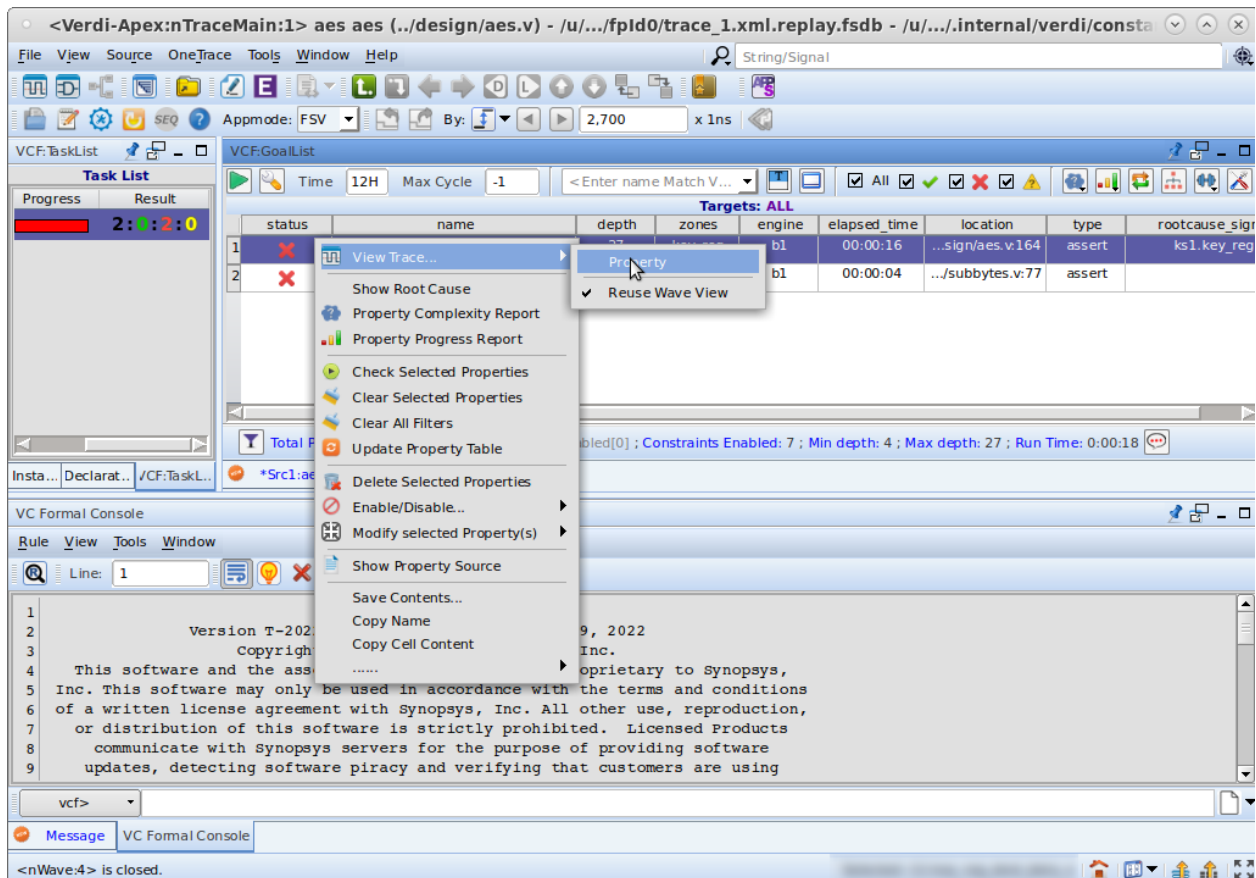


Figure 3.2.1. Examining the falsified security properties in our design.

You should then see a generated counter-example waveform as shown below from *Figure 3.2.2*:

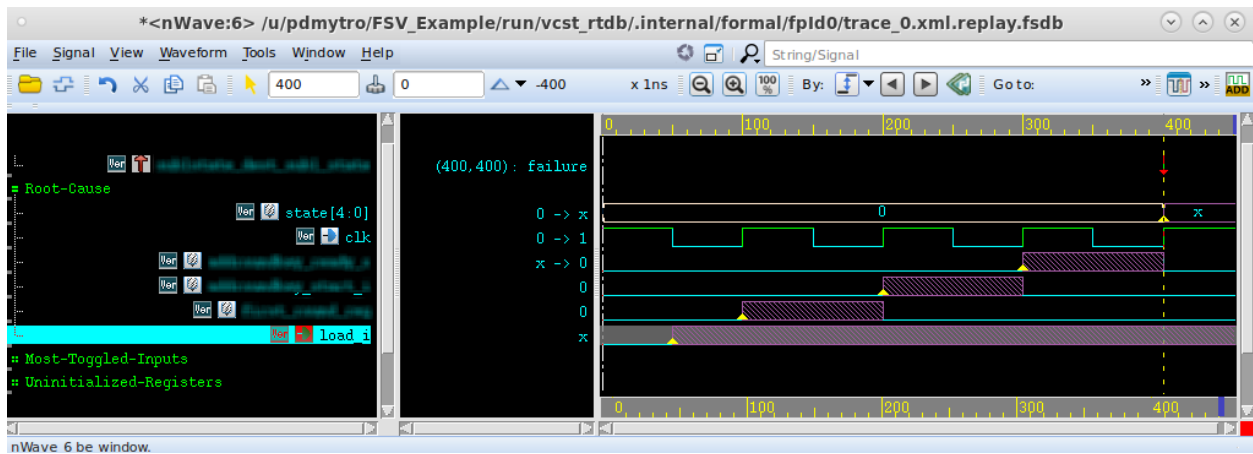


Figure 3.2.2. Examining the waveform

In *Figure 3.2.2* on the left side we see all of the registers in the propagation path from the source to the destination.

As seen above the waveform shows that the given input “load_i” can be affecting the value of the register state which is a potential security issue within the DUT.

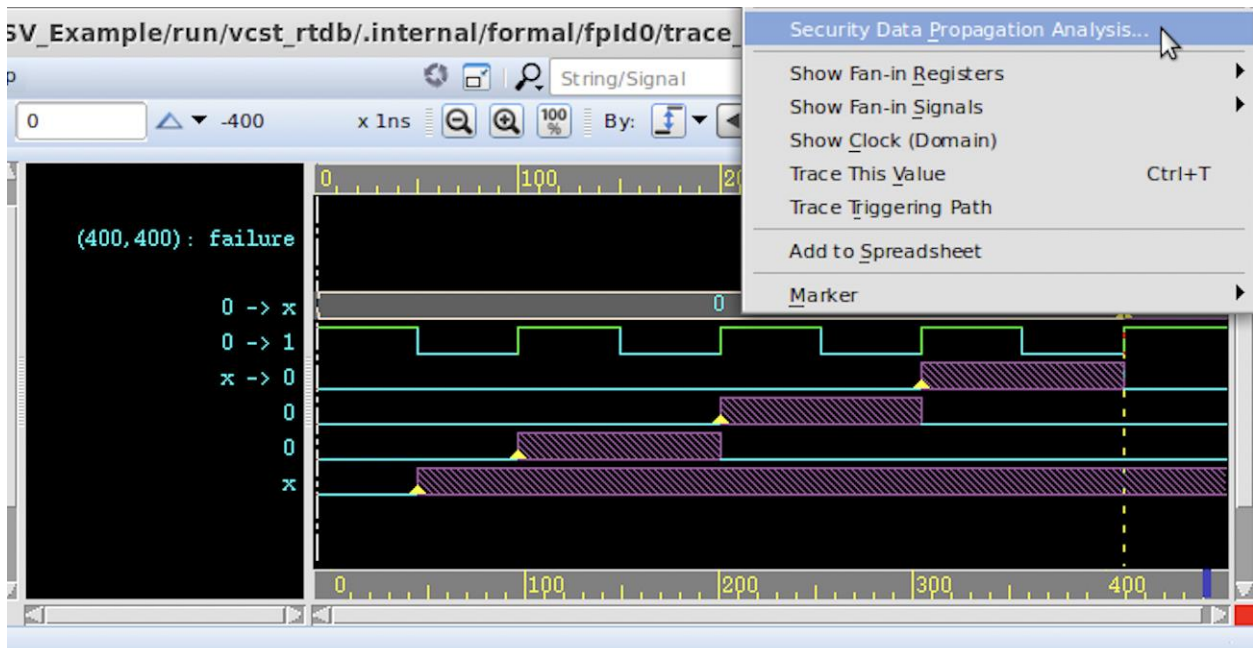


Figure 3.2.3. Further examining the propagation between the input and the secure register

For further analysis right click on the location of the failure and select “Security Data Propagation Analysis”. (see *Figure 3.2.3* above)

This will highlight how the propagation happens between the input and the secure register. (see *Figure 3.2.4* below)

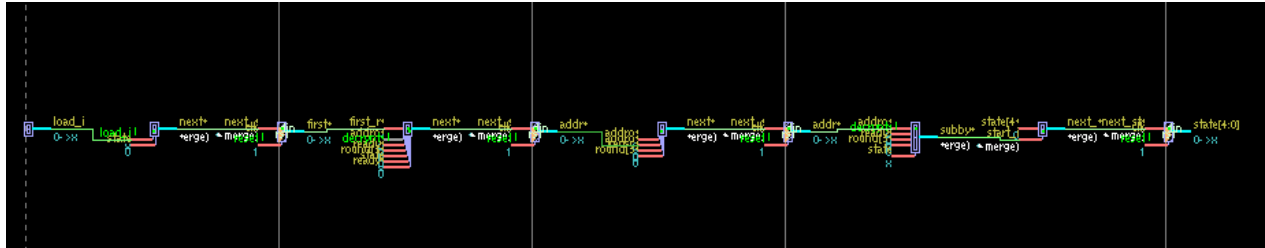


Figure 3.2.4. Data propagation of the source & destination of the security in temporal flow view

Repeat the debugging step for each falsified property.

3.3 Resolving Errors

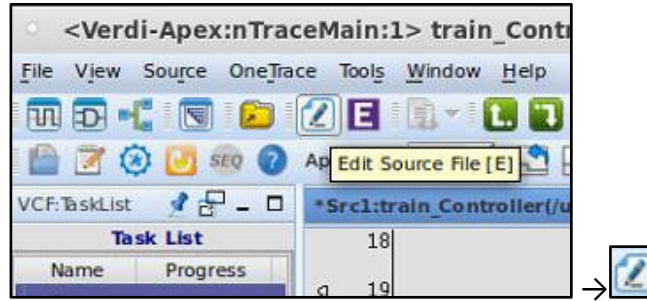


Figure 3.3.1. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes.

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

3.4 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 3.4.1. 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:

	status
5	✓
6	✓
7	✓

Figure 3.4.2. Results after fixing errors and restarting VC Formal and reloading TCL script.

Appendix

Function	Code	Description
Set App mode	set_fml_appmode FSV	VC Formal App mode command for FSV
Format	-format <format>	The format of the file to be loaded. The two formats are csv or table. The format option is optional, and default is csv.
Results Summary	-quiet	Option to hide the progress message and summary results.
CSV File Path	<filename>	Path of the properly formatted csv file to read.
Formal Compile	formal_compile	This command is used to compile the design and the formal properties specified in the input files, generating a formal model for verification.
Formal Verify	formal_verify	Use this command to initiate the formal verification process. It checks the specified properties against the formal model generated during compilation.
Formal Debug	formal_debug	Use this command to initiate the formal debugging process. It provides interactive debugging capabilities to help analyze and resolve issues found during verification.
Report of Warnings	-warning	Specify this option to get a report of warnings. You can view the warnings file-wise. Appending the -warning switch with -list and -verbose, displays all warnings from the list and verbose.
Summary OFF State	-no_summary	Specify this option if you do not want to see the summary information for any other switch.
Detailed Report Information	-verbose	Displays the summary of results and then provides detailed information about the report such as the status, connection name, line number and message of each property file-wise.
Save initial state	sim_save_reset	Saves initial state
Check setup	check_fv_setup	Checks setup for errors

Table 1. FSV App important functions and commands.

Synopsys® VC Formal Tutorial

Functional Safety Application (FuSa)

Version 1.1 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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.....	3
1.1 About and Usage of FuSa	4
1.2 Design Files.....	5
1.3 TCL File Syntax.....	6
2. Application Setup.....	8
2.1 Invoking VC Formal GUI	9
2.2 Invoking VC Formal Along with TCL File:	11
3. Application Usage.....	12
3.1 Detecting Errors	13
3.2 CEX Waveform and Source Tracing	14
3.3 Resolving Errors.....	17
3.4 Restarting VC Formal.....	18
Appendix	19

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 “FuSa_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 FuSa analysis for us.

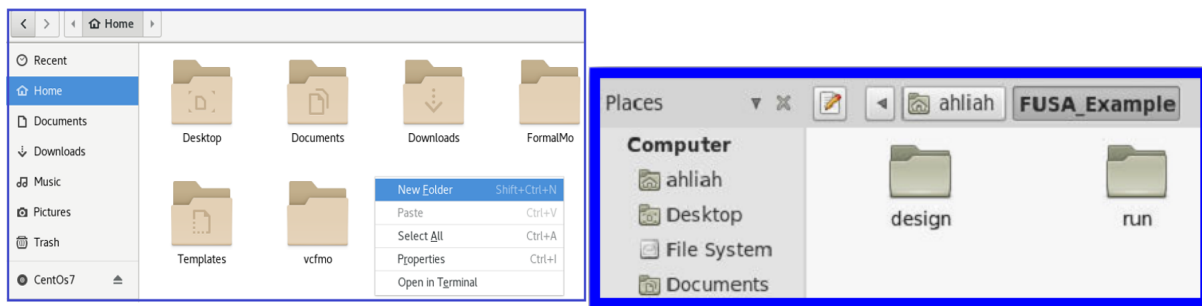
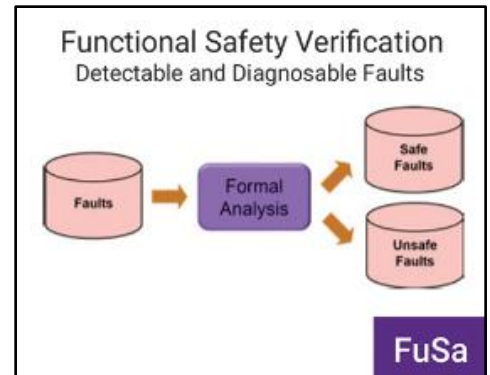


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of FuSa

The Functional Safety Verification (FuSa) application is used as a safety tool to minimize risk caused by a malfunctioning system. A separate in-house tool that Synopsys uses, called Z01X, is a Verilog software fault injection simulator that provides coverage results. VC Formal takes these results and determines whether the type of faults simulated and identified are “safe” or “dangerous”, via the FuSa application. Since the Z01X tool produces many types of faults, VC Formal serves as an extra step to help sift through and reduce the need for manual reviewing.



This FuSa figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

The FuSa application performs the following types of analysis:

- ❖ Structural Analysis
- ❖ Controllability Analysis
- ❖ Observability Analysis
- ❖ Detection Analysis

Structural Analysis is used to identify whether faults are in or not in the COI of observation/detection points. They are marked if they are not and those that are not are put through further analysis.

Controllability Analysis is used to determine if a signal can transition from one established state to another at the location of the fault. Cover properties are set at fault locations and are marked non-controllable if unreachable or sent for further analysis if properties are covered/inconclusive.

Observability and Detection Analysis is used to observe whether or not a fault is blocking a signal from propagating through to an observation/detection point. Faults that do not propagate are considered safe. Faults that propagate to observation/detection points are marked accessed further.

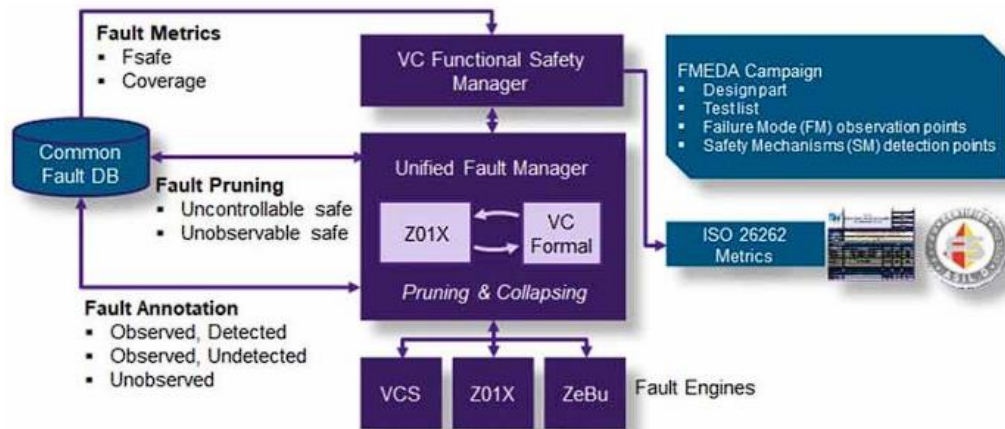


Figure 1.1.1. FuSa Verification Flow Map

This FuSa figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

1.2 Design Files

In the Design folder, you'll put in your RTL design file(s) and SFF file. Your design file should include all necessary clocks, reset, and constraints needed to function, while also being free from any errors or violations. If your design includes a testbench and/or multiple .sv files, make sure they are also free from any violations, then house them in the design folder. For this tutorial, there will only be one design file (fusa_design.sv).

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.

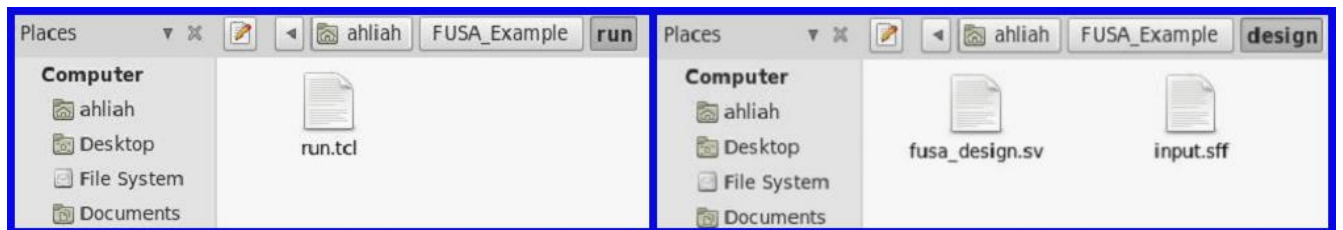


Figure 1.2.1. Showing the (Right) RTL/SFF and (Left) TCL file in the correct folder locations.

1.3 TCL File Syntax

Next, we will create a TCL command script for FuSa setup. This is where we compile our designs using various commands and perform each Functional Safety Analysis type respectively.

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

```
set_fml_appmode DPV
```

Then, you need to load the fault list:

```
Fusa_config -sff <sff filename>
```

Importing faults from SFF means that faults are defined in Standard Fault Format.

```
FaultGenerate FMI
{
  # Create faults on all ports in hierarchy
  NA [0,1] {PORT "top.***" }
  NA [0,1] {WIRE "top.***" }
  NA [0,1] {VARI "top.***" }
  NA [0,1] {ARRY "top.***" }
```



Figure 1.3.1. Faults defined in SFF, within a .sff file.

This figure was taken from the VC Formal User Guide FuSa App Version T-2022.06-SP2, December 2022

Once the fault list has been imported, compile your error-free design and define “clock” and “reset” as needed for your design

```
read_file -top test -format verilog -vcs <RTL filename>
```

and define “clock” and “reset” as needed for your design following these commands:

```
create_clock and create_reset
```

Then initialize VCF setup:

```
sim_run -stable and sim_save_reset
```

If Observation/Detection points are not already specified in the SFF file like this:

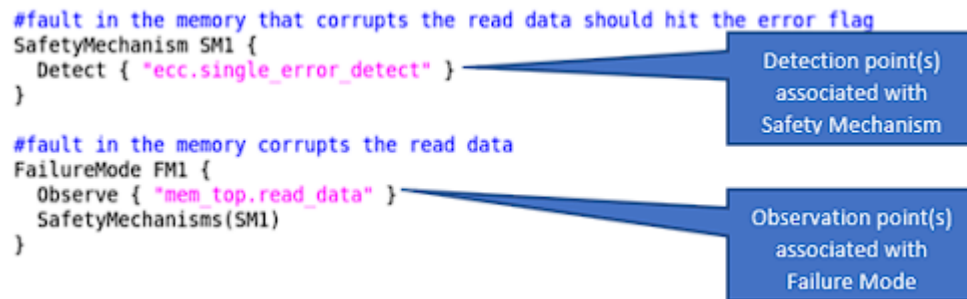


Figure 1.3.2. Observation/Detection points defined within a .sff file.

This figure was taken from the VC Formal User Guide FuSa App Version T-2022.06-SP2, December 2022

You can identify them in the TCL file with these commands:

```
fusa_observation -add {<filename>.<list of signal names>}
```

and

```
fusa_detection -add {<filename>.<list of signal names>}
```

To ensure that false proofs are avoided, this command allows data to propagate through a safe black box:

```
fusa_blackbox -all -all_auto_path
```

And now to generate FuSa properties:

```
fusa_generate
```

From here we can run Structural, Controllability, Observability, or Detectability analysis:

```
set_fml_var fusa_run_mode <structural/control/observe/detect>
```

Following this, you want to run the FuSa check and report:

```
check_fv and fusa_report
```

The end of the TCL file will be for saving the results to SFF:

```
fusa_save -sff < sff filename>
```

2. 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 FuSa 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.

2.1 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
```

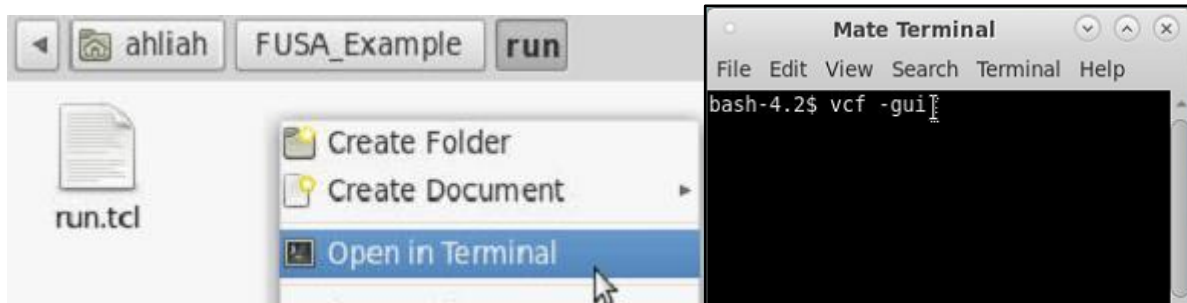


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2 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:

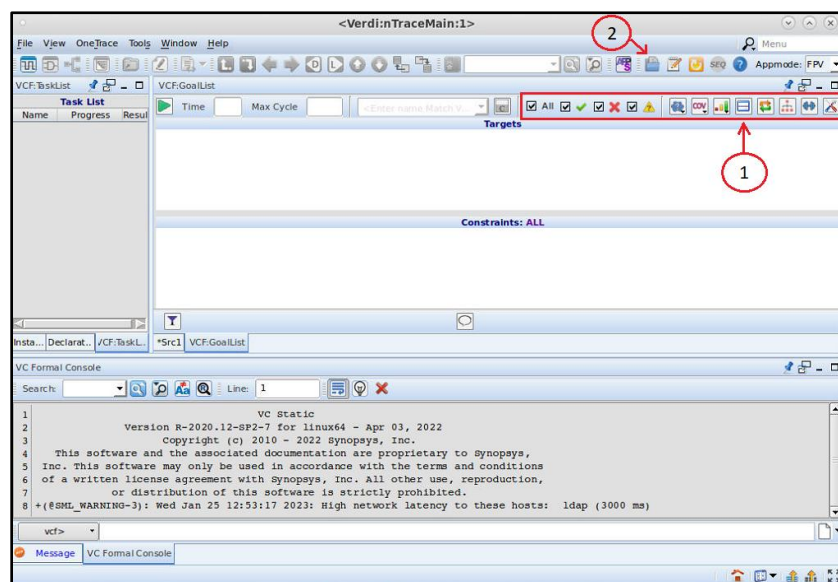


Figure 2.1.2. VC Formal GUI introductory screen.

Then load a TCL script by clicking on the  icon (2) as shown in Figure 2.1.2.

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

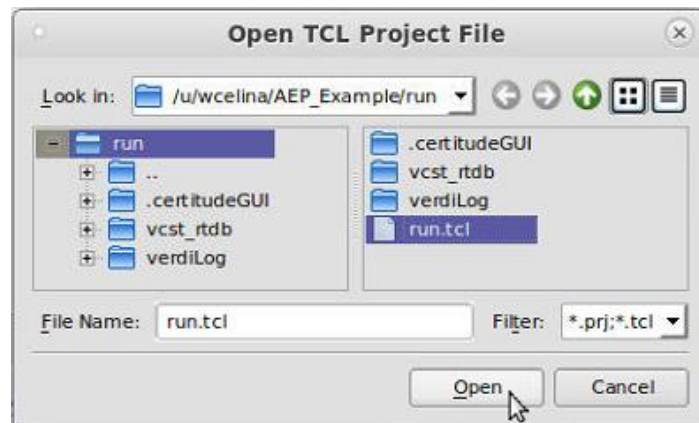


Figure 2.1.3. Selecting TCL file.

2.2 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:

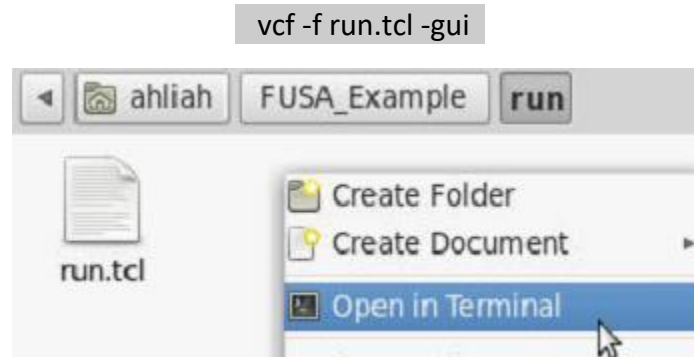


Figure 2.2.1. Opening terminal in the ‘run’ folder.

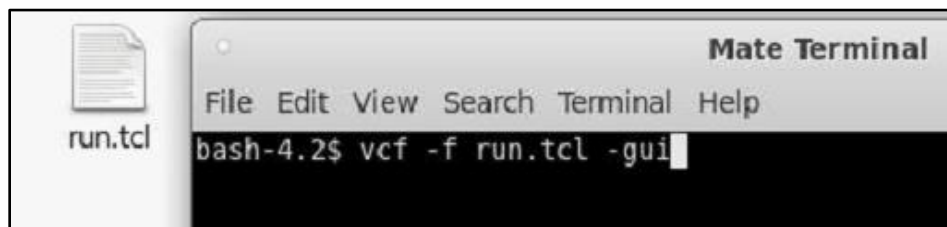


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

3. Application Usage

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:

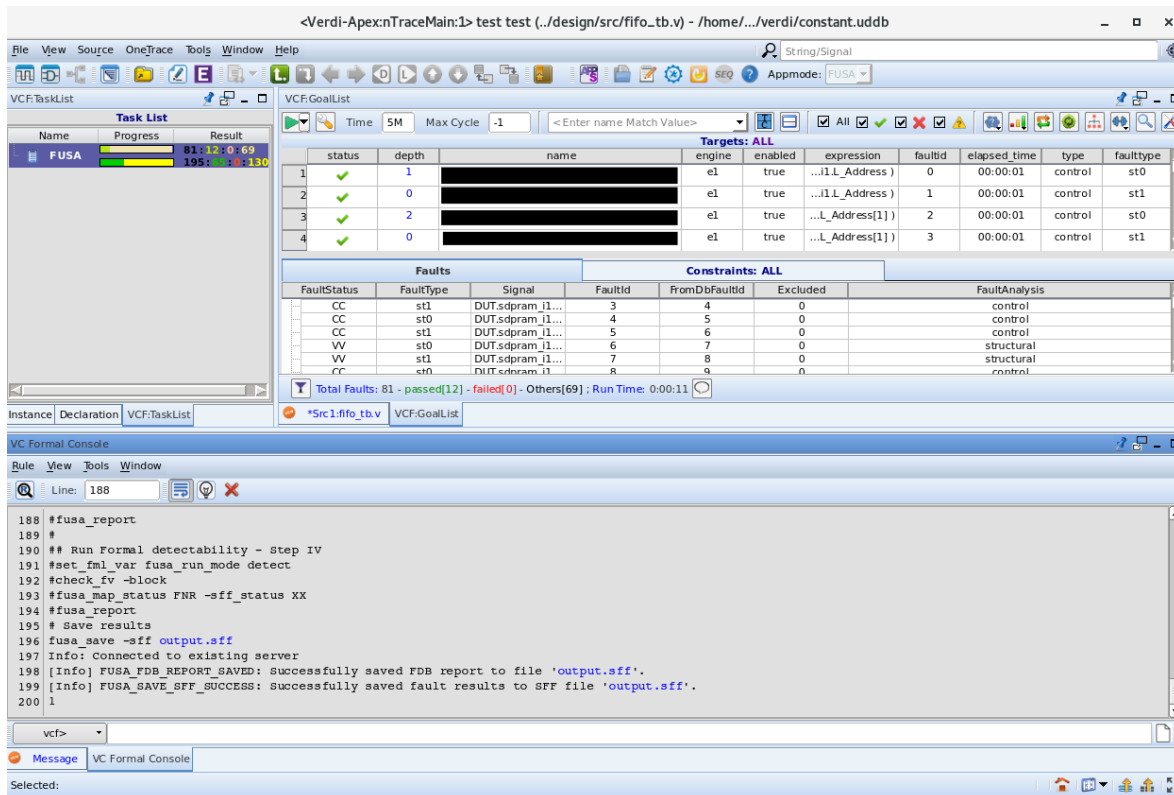


Figure 3.1. Screen after loading TCL script.

An initial Verification analysis will run automatically after invoking VC Formal. However, you can choose one of the Functional Safety Analysis types by clicking on the down arrow next to the play icon in the upper left corner of the “VCF:GoalList” tab.

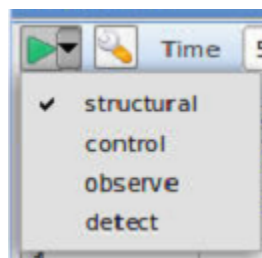


Figure 3.2. Functional Safety Analysis types from drop arrow menu.

3.2 CEX Waveform and Source Tracing

VC Formal uses an CEX waveform as a debugging tool for the FuSa app. It is used to show a signal value in a “good machine” and a “faulty machine”. The analysis will fail if there is an instance where these values differ.

To start, go ahead and double-click on an  icon. You should then see a generated waveform similarly shown below:

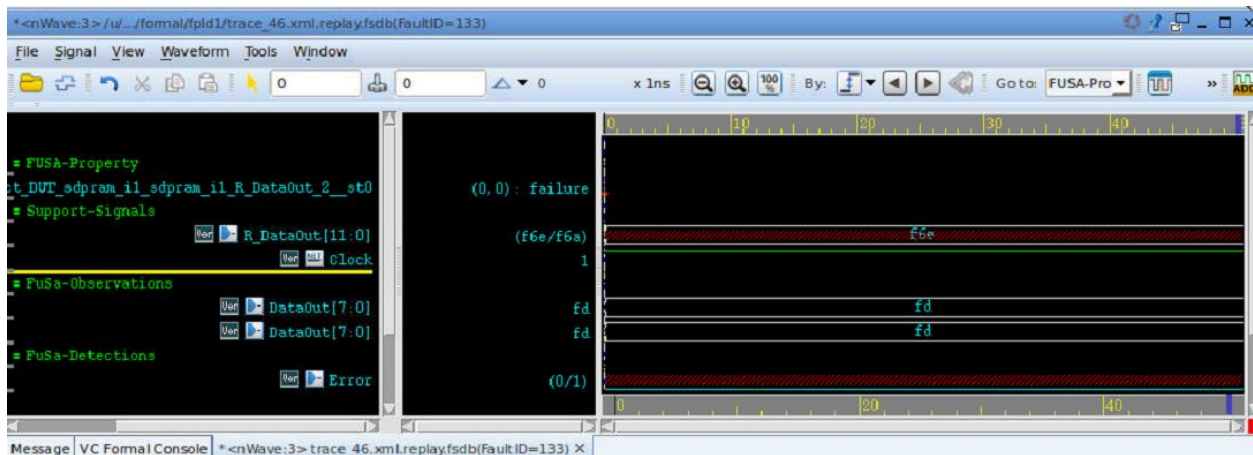


Figure 3.2.1. Examining the failed FuSa fault in our design.

On the left in *Figure 3.2.1* above, we see “Support-Signals”, “FuSa-Observations”, and “FuSa-Detections” in green text. You can choose to look at where this fault was observed and/or detected by selecting a signal with a shaded portion. Right-click on the chosen signal in the waveform and select “Add Fault Waveform”; shown below.

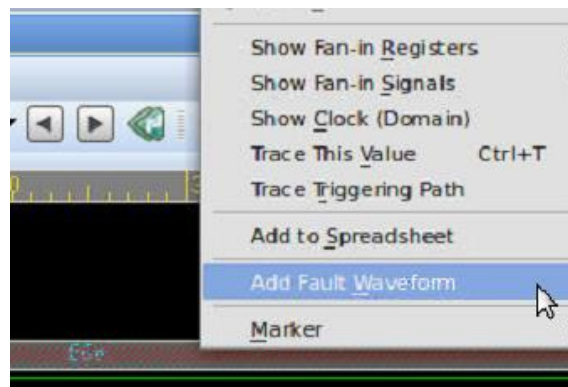


Figure 3.2.2. Pop-up from right-clicking on a signal.

This will add a faulty signal in order to exhibit the value causing the fault:

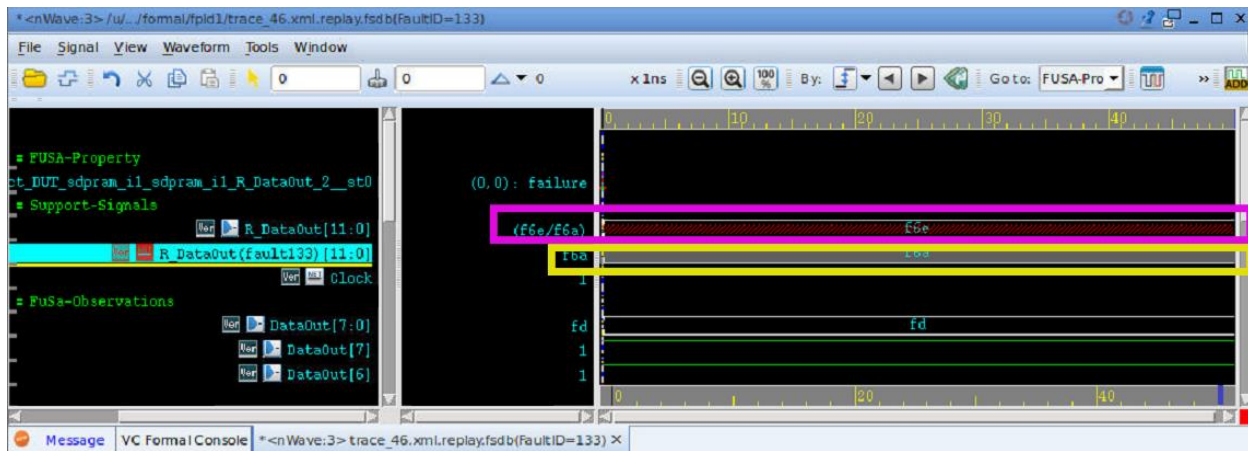


Figure 3.2.3. Faulty signal added to waveform.

By tracing the source and adding this faulty signal, we are essentially showing the faulty machine signal values and comparing them to the good machine signal values. The red shaded signals are an indication of a deviation from the good machine; the initial generated signals are all good machine values (one highlighted in pink). When you add a “*fault waveform*” signal (highlighted in yellow), it shows the values associated with the faulty machine.

Another way to trace the source of a fault and exhibit where it was observed/detected is through a temporal flow view.

Right-click on the chosen signal in the waveform and select “*Auto Trace*” under “*Temporal Flow View*”; shown below.

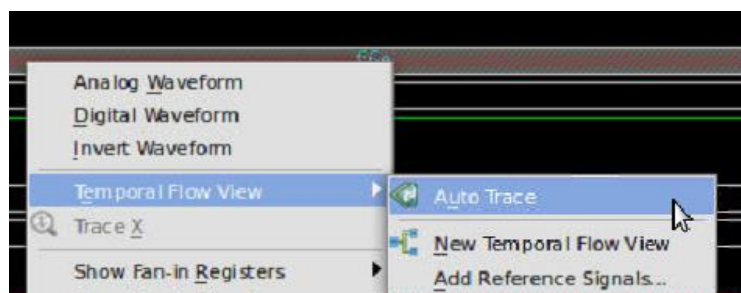


Figure 3.2.4. Pop-up from right-clicking on a signal.

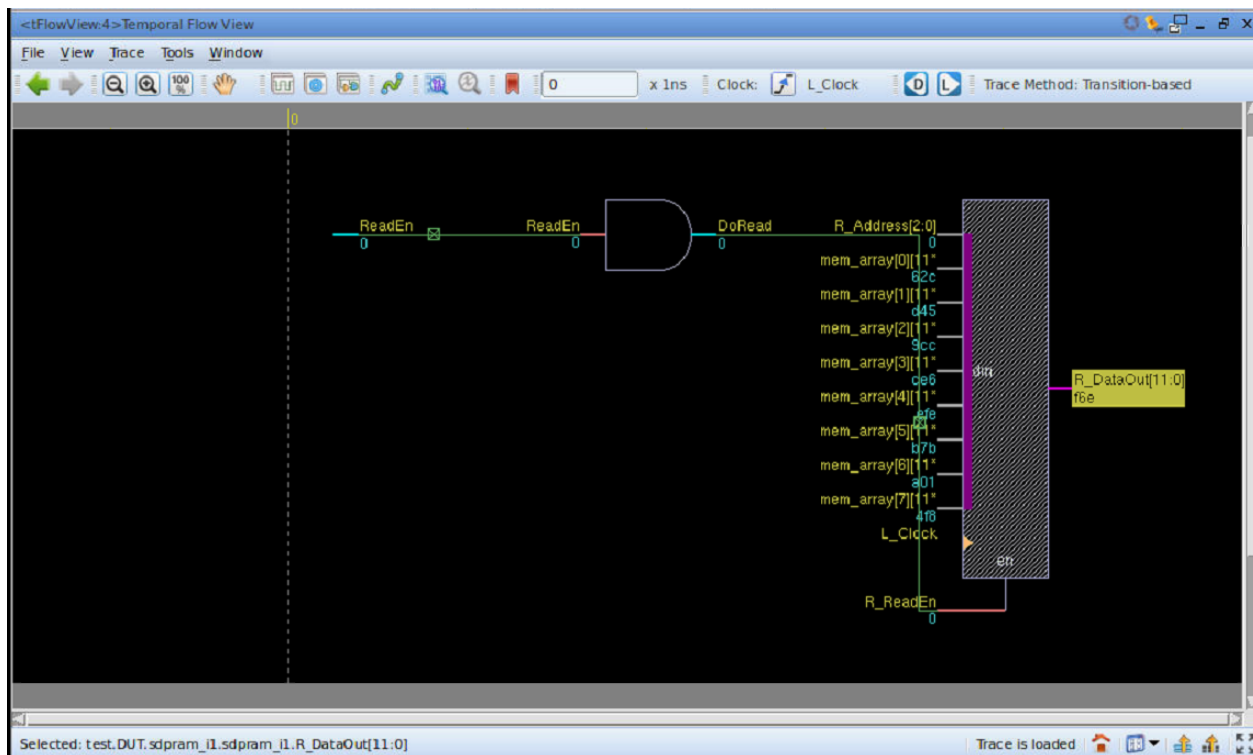


Figure 3.2.5. Temporal Flow View of design schematic.

By tracing the source, we are essentially backtracking it to the driving signal of the output in order to find the discrepancy. Here you can examine the root-cause by double-clicking on a connection on the RTL and It will take you to the part of your code where the fault was observed/detected.

3.3 Resolving Errors

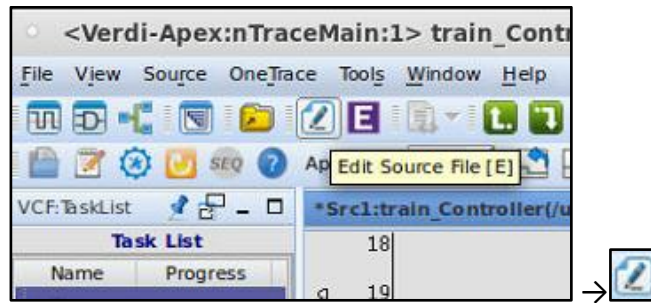


Figure 3.3.1. To make changes to the design file in VC Formal, click on “Edit Source File”. Don’t forget to save your changes.

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

3.4 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 3.4.1. 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:

	status
5	✓
6	✓
7	✓

Figure 3.4.2. Results after fixing errors and restarting VC Formal and reloading TCL script.

Appendix

Description	Internal Formal Status	Status Codes
Formal No Result	FNR	NA
Structural Inconclusive	SIN	VV
Controllability Inconclusive	CIN	VV
Controllability Controllable	CCC	CC
Observability Inconclusive	OIN	XX
Detectability Inconclusive	DIN	XX
Observability Observed	OOB	OX
Detectability Not Detected	DND	XF
Detectability Detected	DDT	XD
Observable and Not-Detected	OND	OF
Structurally Not Observed Not Detected	SSF	UU
Structurally Not Observed	SNO	IV
Structurally Not Detected	SND	VI
Controllability Non-Controllable	CNC	UT
Observability Non-Observable	ONO	UB
Non-Observable and Detected	NOD	UB
Observable and Detected	ODT	OD
Non-Observable and Not Detected	NON	UB

Table 1. *FuSa Fault Status Mapping.*

Synopsys® VC Formal Tutorial

Formal Testbench Analyzer (FTA)

Version 1.2 | 2-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of FTA.....	4
1.2 Design Files.....	5
1.3 TCL File	6
2. Application Setup	7
2.1 Invoking VC Formal GUI	8
2.2 Invoking VC Formal Along with TCL File:.....	10
3. Application Usage	11
3.1 Switching App Modes	13
3.2 Debugging Non-Detected Faults	15
3.3 Debugging Non-Activated Faults	17
3.4 Restart FTA and Verify Faults Resolved	19
3.5 Exporting Faults to FPV	20
3.6 Clustering Faults.....	21
Appendix.....	23

1. Introduction

VC Formal uses TCL (Tool Command Language) scripts to tell it what to do. The script can define the app within VC Formal 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 “FTA_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 VC Formal FTA analysis for us.

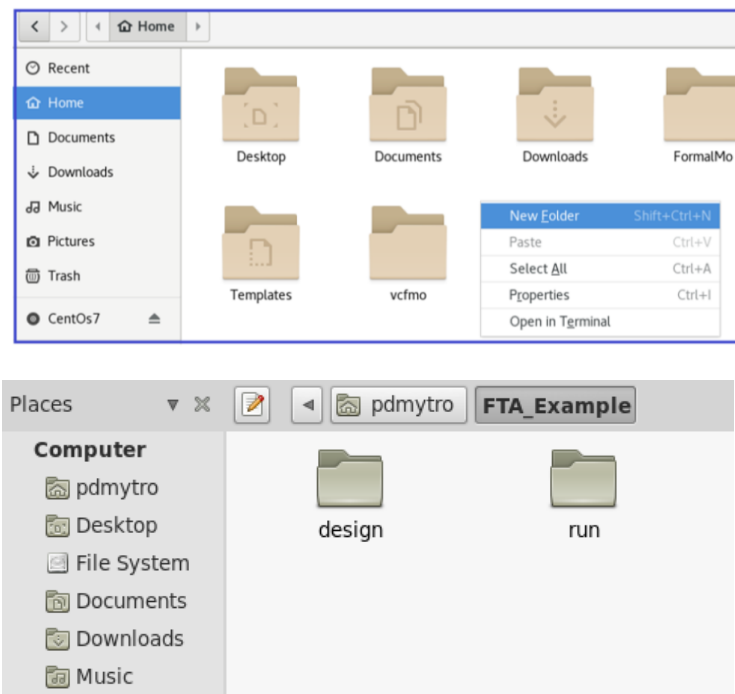
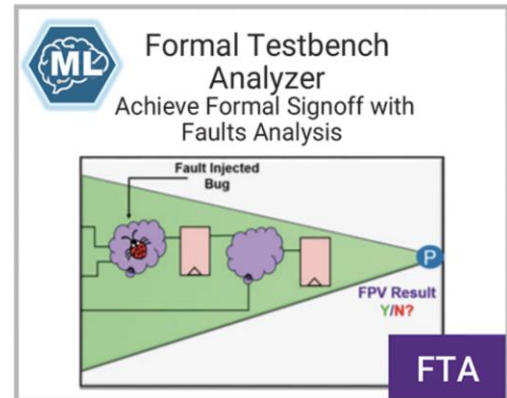


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of FTA

The Synopsys VC Formal FTA (Formal Testbench Apps) is an innovative tool designed to enhance the efficiency and effectiveness of functional verification using formal methods. VC Formal FTA offers a comprehensive set of features and capabilities for creating advanced testbenches based on formal techniques. By leveraging its powerful automation capabilities and intelligent analysis, VC Formal FTA enables engineers to rapidly generate high-quality testbenches that thoroughly exercise their designs, ensuring robustness and identifying hard-to-find bugs. This application revolutionizes the traditional testbench creation process, significantly reducing the time and effort required for functional verification.



This FTA figure was taken from the VC Formal User Guide Version T-2022.06-SP2, December 2022

1.2 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.

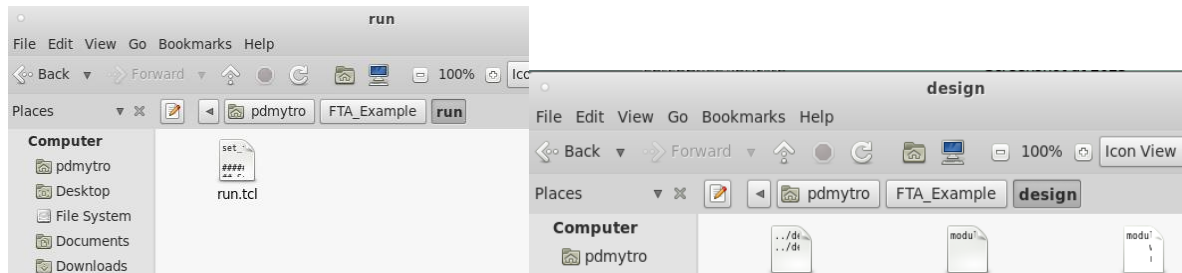


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

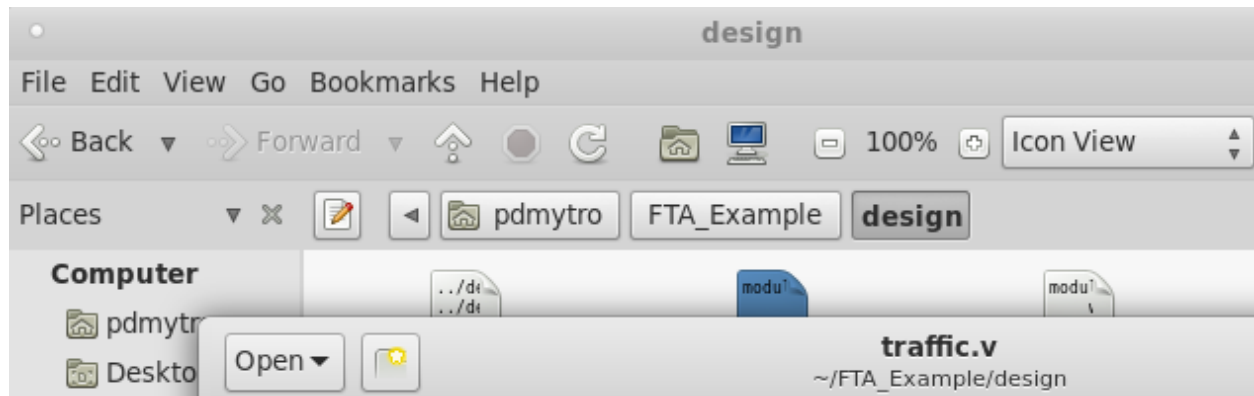


Figure 1.2.2. Example of the design we are using in this tutorial.

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 1.2.2* 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).

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.1*), which you can use as a template for your functional checks on VC Formal.

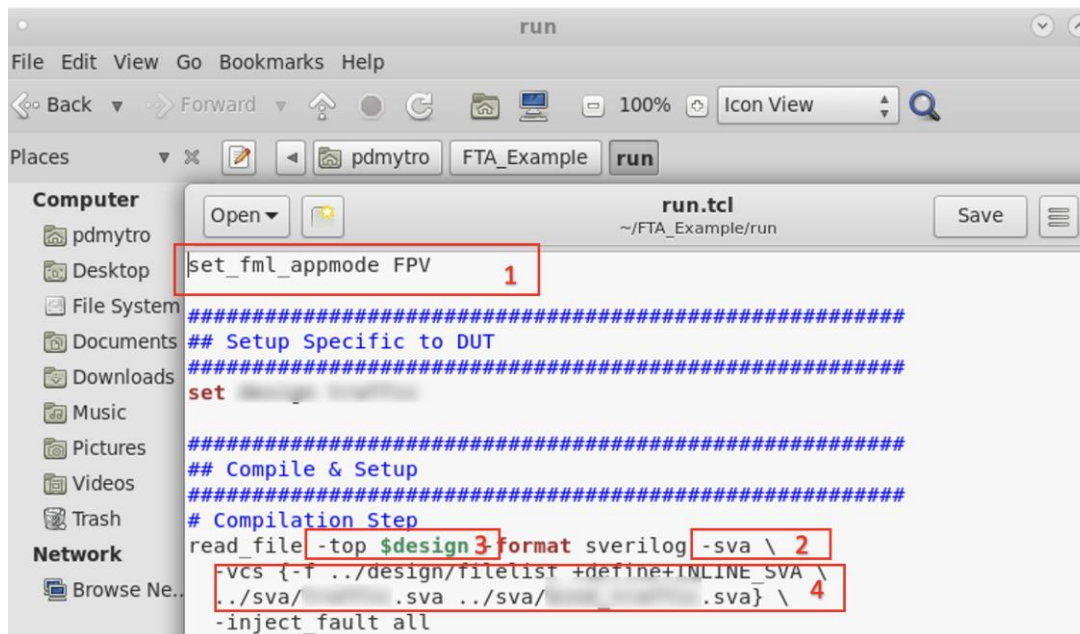


Figure 1.3.1. Annotated TCL template file.

- (1) Instruction that sets the appmode to FPV in VC Formal.
- (2) Name of the main module as established in the Design file.
- (3) The property type identifier switch name for desired FSV analysis.
- (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 1.3.1*.

The file name (aes.v) can be named however you want.

2. 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 FTA 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.

2.1 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
```

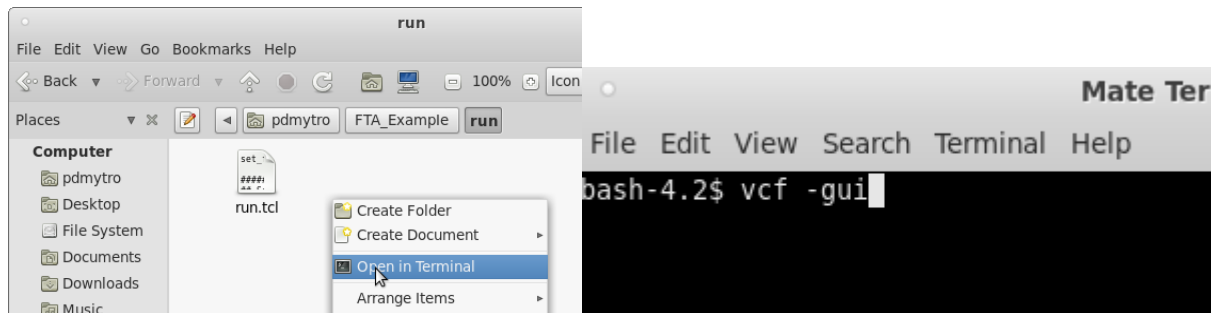


Figure 2.1.1. Invoking VC Formal in the terminal.

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 2.1.2* 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: 

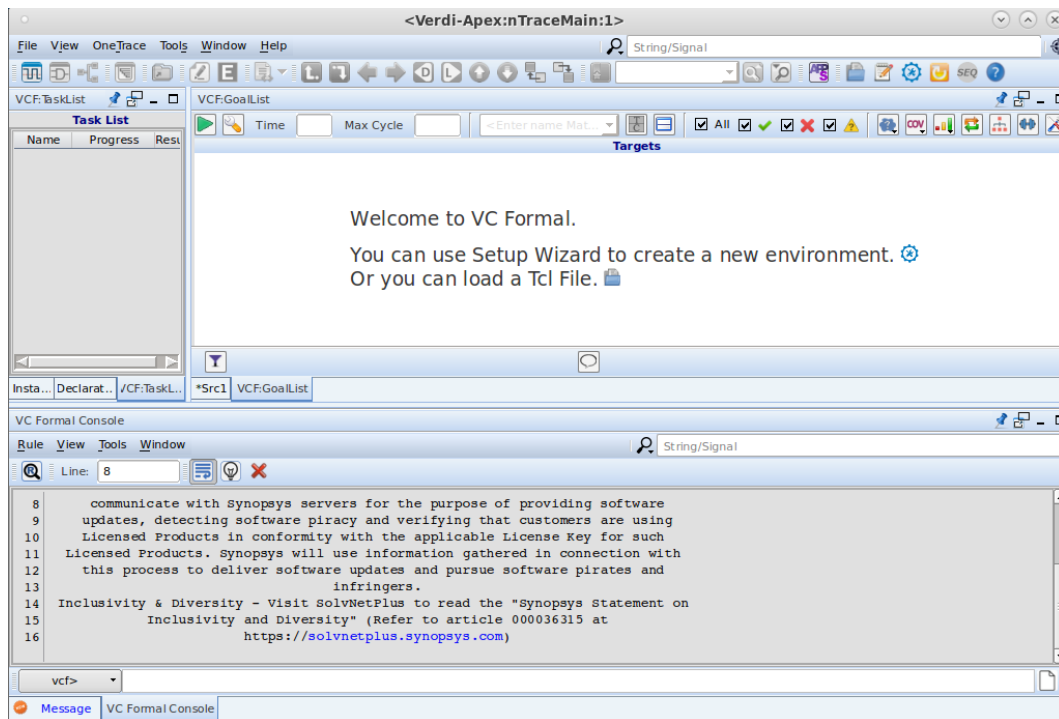


Figure 2.1.2. VC Formal GUI introductory screen.

Then load a TCL script by clicking on the  icon as shown in *Figure 2.1.2*.

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

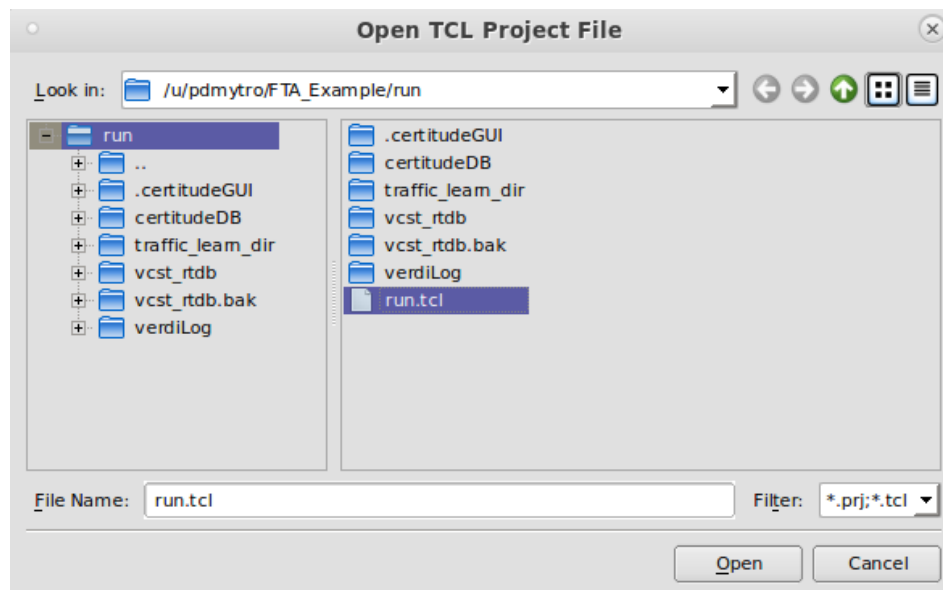


Figure 2.1.3. Selecting TCL file.

2.2 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
```

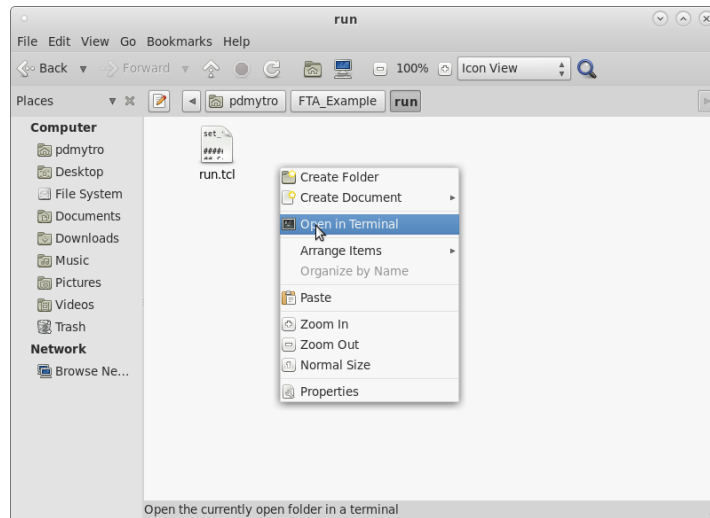


Figure 2.2.1. Opening terminal in the ‘run’ folder.

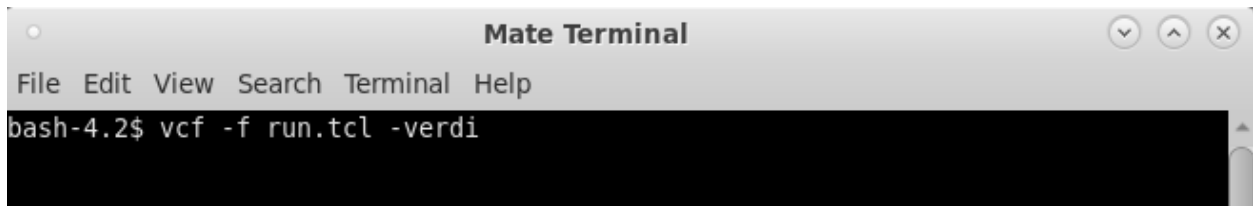


Figure 2.2.2. Invoking VC Formal and TCL script in the terminal.

And that’s it!

3. Application Usage

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:

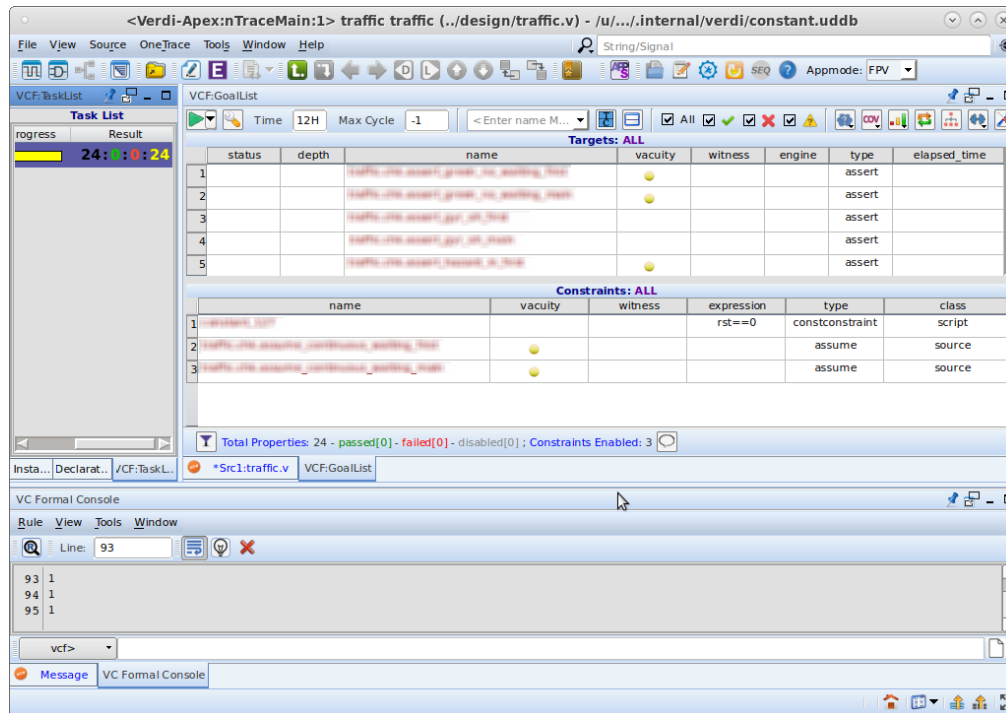


Figure 3.1. Screen after loading TCL script.

Now, go ahead and run the property verification analysis by clicking on the play  icon in the upper left corner of the “VCF:GoalList” tab.

Once the property verification analysis is completed your screen should look like this:

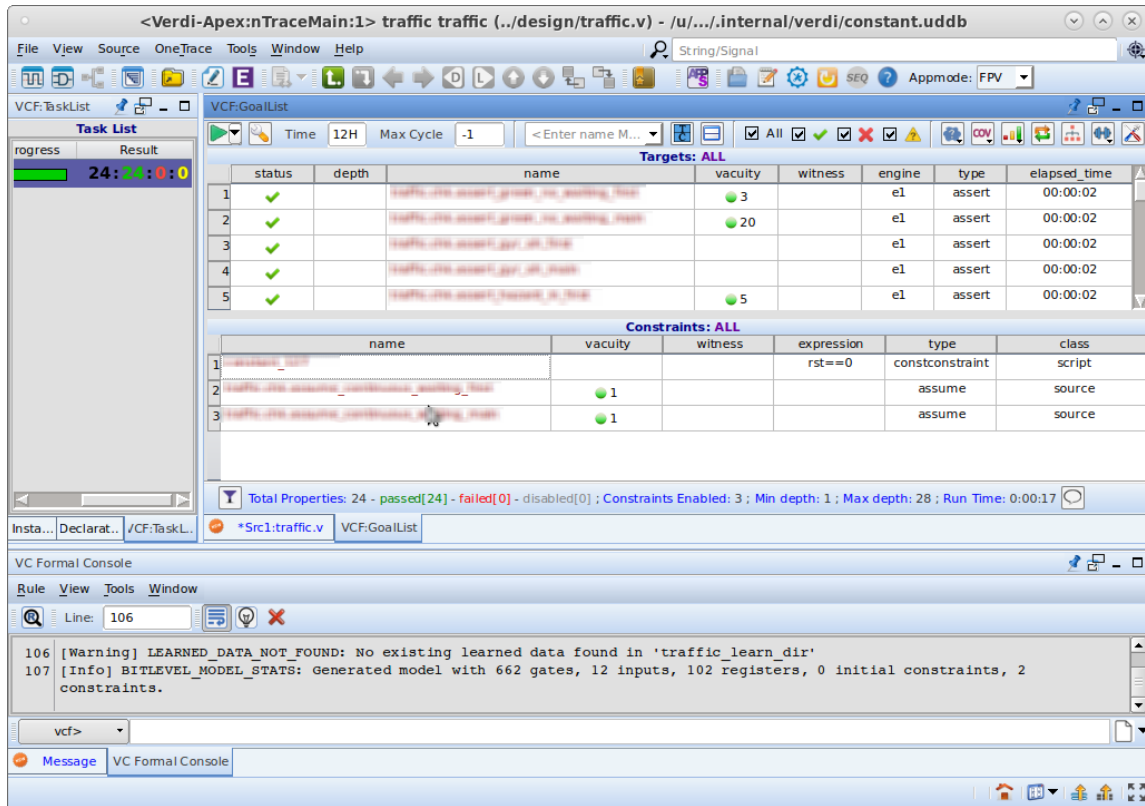


Figure 3.2. After running property verification.

This step is necessary because the proven properties from the FPV run will be further used in the FTA flow to detect faults.

3.1 Switching App Modes

After running property verification, switch to the FTA app mode. This is done by finding the appmode icon and clicking on the drop down menu:

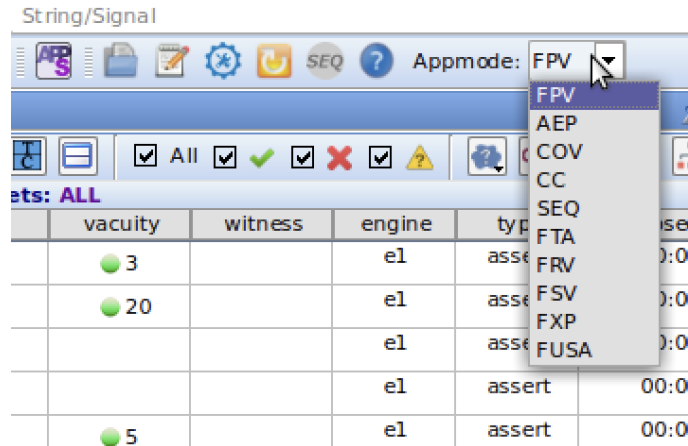


Figure 3.1.1. Switching between applications.

Once the switch from the FPV app to the FTA app has been invoked your screen should look like this:

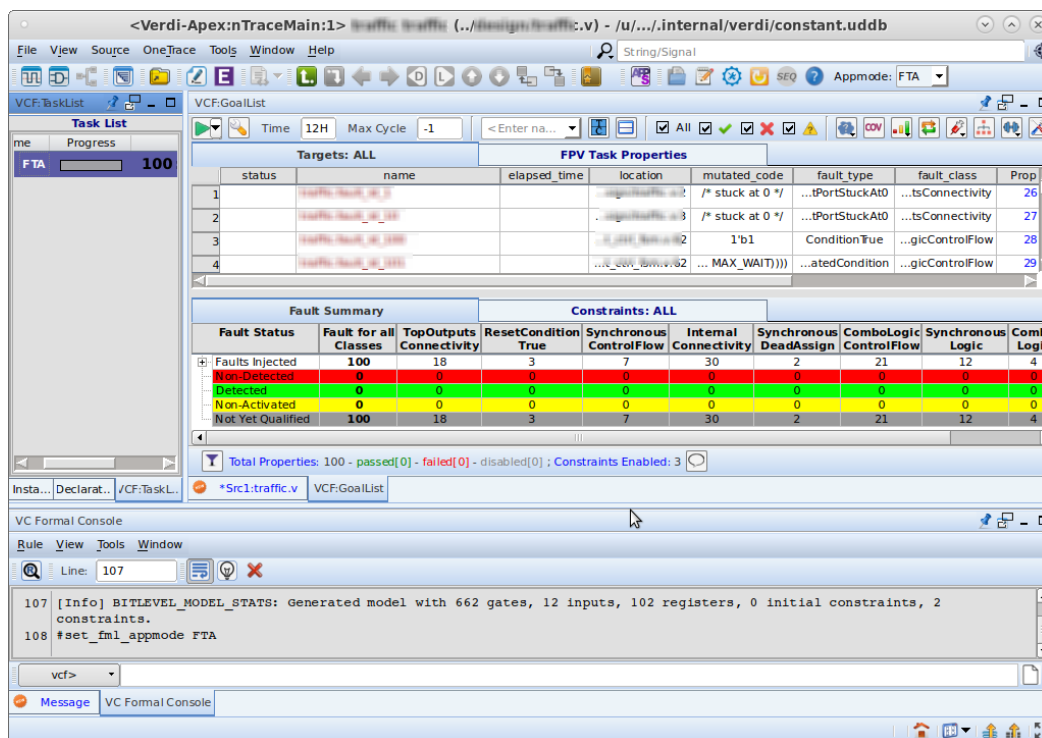


Figure 3.1.2. Loaded into FTA appmode after running property verification.

Start property verification in the FTA appmode by clicking on the  play icon in the upper left corner. The play  command will create an FPV_FTA task and start the verification process.



Figure 3.1.3. New task created once the start icon is clicked in the FTA appmode.

Once the check is complete a fault summary will be displayed.

Fault Summary			Constraints: ALL						
Fault Status	Fault for all Classes	TopOutputs Connectivity	ResetCondition True	Synchronous ControlFlow	Internal Connectivity	Synchronous DeadAssign	ComboLogic ControlFlow	Synchronous Logic	Combo Logic
⊕ Faults Injected	100	18	3	7	30	2	21	12	4
Non-Detected	22	2	0	1	6	0	10	2	1
Detected	75	16	2	6	24	1	11	10	3
Non-Activated	3	0	1	0	0	1	0	0	0
Not Yet Qualified	0	0	0	0	0	0	0	0	0

 Total Properties: 100 - passed[75] - failed[22] - disabled[3] ; Constraints Enabled: 3 ; Min depth: 2 ; Max depth: 22 ; Run Time: 0:06:18 

Figure 3.1.4. Fault Summary display once FTA check is complete.

3.2 Debugging Non-Detected Faults

Debugging non-detected faults is done through the fault summary that is generated when the FTA appmode is done running.

The non-detected faults will be displayed in the faults summary and they will be highlighted in red labeled “Non-detected”.

Non-Detected	22	2	0	1	6	0	10	2	1
--------------	----	---	---	---	---	---	----	---	---

Figure 3.2.1. Non-Detected faults tab in the faults summary.

Double click on the TopOutputsConnectivity cell that corresponds to the Non-Detected (the cell with the number 2) cell to filter the “Targets: All” table to show the “Targets: Failure” table.

Fault Summary		
Fault Status	Fault for all Classes	TopOutputs Connectivity
Faults Injected	100	18
Non-Detected	22	2

Figure 3.2.2. TopOutputsConnectivity X Non-Detected Faults (2).

The displayed “Targets: Failure” table will look like the following table:

Targets: Failure Filter by fault_class, status			FPV Task Properties					
status (V)	name	elapsed_time	location	mutated_code	fault_type	fault_class (V)	Prop ID	
1	...	00:05:23	...	/* stuck at 0 */	...tPortStuckAt0	...tsConnectivity	43	
2	...	00:05:36	...	/* stuck at 0 */	...tPortStuckAt0	...tsConnectivity	72	

Fault Summary			Constraints: ALL						
Fault Status	Fault for all Classes	TopOutputs Connectivity	ResetCondition True	Synchronous ControlFlow	Internal Connectivity	Synchronous DeadAssign	ComboLogic ControlFlow	Synchronous Logic	Combo Logic
Faults Injected	100	18	3	7	30	2	21	12	4
Non-Detected	22	2	0	1	6	0	10	2	1
Detected	75	16	2	6	24	1	11	10	3
Non-Activated	3	0	1	0	0	1	0	0	0
Not Yet Qualified	0	0	0	0	0	0	0	0	0

Total Properties: 100 - passed[75] - failed[22] - disabled[3] ; Constraints Enabled: 3 ; Min depth: 2 ; Max depth: 22 ; Run Time: 0:06:18

Figure 3.2.3. Targets:Failure table showing the undetected faults.

The isolated undetected faults will be labeled with an  in the status cell. Once the non-detected faults are identified and isolated, double click on the fault in the name cell to open up a highlighted source code window. This window will open the source code and identify the undetected faults with the color red.



Figure 3.2.4. Non-detected faults identified in the source code highlighted in red.

Modify the property by clicking on the “edit source file” icon . Once modified save the new source code.

3.3 Debugging Non-Activated Faults

Debugging non-activated faults is also done through the fault summary that is generated when the FTA appmode is done running.

The non-activated faults will be displayed in the faults summary and they will be highlighted in yellow labeled “Non-activated”.

Non-Activated	3	0	1	0	0	1	0	0	0
---------------	---	---	---	---	---	---	---	---	---

Figure 3.3.1. Non-activated faults tab in the faults summary (highlighted in yellow)

Double-click on the “ResetCondition True” cell that corresponds to the “Non-activated” cell (the cell with the number 1) to filter the “Targets: All” table to show the “Targets: Failure” table.

The displayed “Targets: Failure” table will look like the following table:

Targets: Failure Filter by fault_class, status		FPV Task Properties						
	status (V)	name	elapsed_time	location	mutated_code	fault_type	fault_class (V)	Prop ID
1	✘	traffic.fault_id_50		...ign/traffic.v:45	1'b1	ConditionTrue	...tConditionTrue	83

Fault Summary			Constraints: ALL						
Fault Status	Fault for all Classes	TopOutputs Connectivity	ResetCondition True	Synchronous ControlFlow	Internal Connectivity	Synchronous DeadAssign	ComboLogic ControlFlow	Synchronous Logic	Combo Logic
Faults Injected	100	18	3	7	30	2	21	12	4
Non-Detected	22	2	0	1	6	0	10	2	1
Detected	75	16	2	6	24	1	11	10	3
Non-Activated	3	0	1	0	0	1	0	0	0
Not Yet Qualified	0	0	0	0	0	0	0	0	0

Figure 3.3.2. Targets:Failure table showing the non-activated faults.

The isolated non-activated faults will be labeled with an ✘ in the status cell. Once the non-activated faults are identified and isolated, double click on the fault in the name cell to open up a highlighted source code window. This window will open the source code and identify the non-activated faults with the color yellow.

If the non-activated faults indicate source code that is outside of the current set properties of the current used FTA flow, these properties will need to be added. Adding properties that cover those signals will eliminate non-activated faults. Since these properties are set in the .tcl script, the current script will need to be edited.

To edit and add properties to the Tcl script find the “*Edit TCL Project File*” icon. () This can also be done by opening up the Tcl script from the run folder in your project with a text editor application.

Add the commands to cover the signals outside your script's current cone of influence and save the edited Tcl script by pressing the save button.

3.4 Restart FTA and Verify Faults Resolved

Restart VC Formal by clicking on the Restart VCST icon. ()

Once VC Formal is restarted after making changes to the .tcl/source files & adding modified assertions, we can go back in and verify whether or not the faults have been resolved.

Once VC formal is restarted with the updated parameters, observe that the non-activated and non-detected faults are now detected.

Fault Summary			Constraints: ALL						
Fault Status	Fault for all Classes	TopOutputs Connectivity	ResetCondition True	Synchronous ControlFlow	Internal Connectivity	Synchronous DeadAssign	ComboLogic ControlFlow	Synchronous Logic	Comb Logic
Faults Injected	100	18	3	7	30	2	21	12	4
Non-Detected	20	0	0	1	6	0	10	2	1
Detected	80	18	3	6	24	2	11	10	3
Non-Activated	0	0	0	0	0	0	0	0	0
Not Yet Qualified	0	0	0	0	0	0	0	0	0

Total Properties: 100 - passed[80] - failed[20] - disabled[0] ; Constraints Enabled: 3 ; Min depth: 2 ; Max depth: 22 ; Run Time: 0:03:05

Figure 3.4.1. Observe the non-activated faults and non-detected faults are now detected.

3.5 Exporting Faults to FPV

The faults can be exported as an FPV task.

Right-click on a fault status category and choose “Export Fault”:

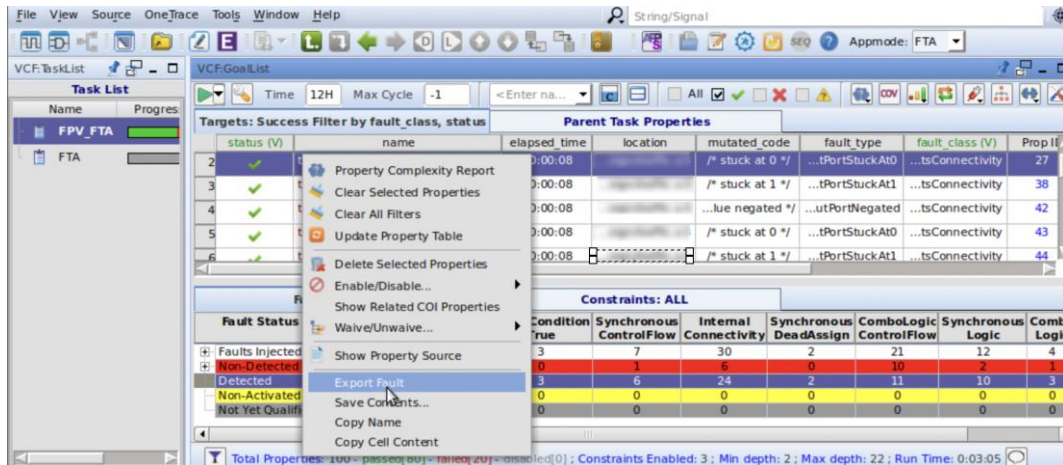


Figure 3.5.1. Exporting Faults to FPV.

The target fault is now exported to the FPV task where its impact on proven properties can be checked by running the proof on the design with the fault inserted.

Once faults are exported to FPV, we can start the property verification by pressing the green play button/Start check icon. ()

Once completed check that the properties are falsified due to the exported fault.

3.6 Clustering Faults

Clustering faults can be a tool used for easier and more efficient analysis. Clusters are faults grouped together based on their classification and their locations in the DUT.

Cluster the faults by clicking the “FTA Property and Fault Selection” icon () and selecting the “non-detected” fault status cluster option.

Return to the fault summary.

Expand the non-detected faults highlighted in red, by clicking the “+”, to see the faults clustered in the same fault class.

Fault Summary			Constraints: ALL						
Fault Status	Fault for all Classes	TopOutputs Connectivity	ResetCondition True	Synchronous ControlFlow	Internal Connectivity	Synchronous DeadAssign	ComboLogic ControlFlow	Synchronous Logic	Com Log
Faults Injected	100	18	3	7	30	2	21	12	4
Non-Detected	20	0	0	1	6	0	10	2	1
Cluster 0	1	0	0	0	0	0	0	0	1
Cluster 1	3	0	0	0	0	0	3	0	0
Cluster 2	2	0	0	0	2	0	0	0	0
Cluster 3	1	0	0	1	0	0	0	0	0
Cluster 4	2	0	0	0	2	0	0	0	0
Cluster 5	7	0	0	0	0	0	7	0	0
Cluster 6	1	0	0	0	1	0	0	0	0
Cluster 7	2	0	0	0	0	0	0	2	0
Cluster 8	1	0	0	0	1	0	0	0	0

Total Properties: 100 - passed[80] - failed[20] - disabled[0] ; Constraints Enabled: 3 ; Min depth: 2 ; Max depth: 22 ; Run Time: 0:03:05

Figure 3.6.1. Clustered faults by all of the faults in the same fault classification

Analyze the faults by double clicking on the cluster number. The clustered faults will appear in the “Targets:Failure” table. They will be labeled with a  status.

Targets: Failure Filter by fault_class, status		FPV Task Properties						
status (V)	name	elapsed_time	location	mutated_code	fault_type	fault_class (V)	Prop ID	
1		00:05:23	...	/* stuck at 0 */	...tPortStuckAt0	...tsConnectivity	43	
2		00:05:36	...	/* stuck at 0 */	...tPortStuckAt0	...tsConnectivity	72	

Fault Summary			Constraints: ALL						
Fault Status	Fault for all Classes	TopOutputs Connectivity	ResetCondition True	Synchronous ControlFlow	Internal Connectivity	Synchronous DeadAssign	ComboLogic ControlFlow	Synchronous Logic	Comb Logic
 Faults Injected	100	18	3	7	30	2	21	12	4
 Non-Detected	22	2	0	1	6	0	10	2	1
 Detected	75	16	2	6	24	1	11	10	3
 Non-Activated	3	0	1	0	0	1	0	0	0
 Not Yet Qualified	0	0	0	0	0	0	0	0	0

Total Properties: 100 - passed[75] - failed[22] - disabled[3] ; Constraints Enabled: 3 ; Min depth: 2 ; Max depth: 22 ; Run Time: 0:06:18

Figure 3.6.2. Clustered Faults Table

Double-click on the name of the clustered property to open a highlighted source code file showing the clustered non-detected faults highlighted in the color red.

Appendix

Function	Code	Description
Set App mode	set_fml_appmode FSV	VC Formal App mode command for FSV
Format	-format <format>	The format of the file to be loaded. The two formats are csv or table. The format option is optional, and default is csv.
Results Summary	-quiet	Option to hide the progress message and summary results.
CSV File Path	<filename>	Path of the properly formatted csv file to read.
FTA create	fta_create	Creates a new formal testbench project. It initializes the project structure and sets up the necessary files and directories.
FTA Add Property	fta_add_property	This command allows you to add properties to the formal testbench. You can specify assertions, cover properties, or assume-guarantee properties to capture the desired behavior of the design.
FTA Generate Testbench	fta_generate_testbench	Automatically generate a formal testbench based on the properties specified. The tool will synthesize test vectors and stimulus patterns to exercise the design and verify the properties.
FTA Analyze Coverage	fta_analyze_coverage	Enables you to analyze the coverage achieved by the formal testbench. It provides insights into the completeness of the testbench and identifies areas that require additional coverage.
FTA Debug	fta_debug	Initiate the formal debugging process for the testbench. It provides interactive capabilities to help analyze the behavior of the design and investigate any issues encountered during verification.
Detailed Report Information	-verbose	Displays the summary of results and then provides detailed information about the report such as the status, connection name, line number and message of each property file-wise.
Save initial state	sim_save_reset	Saves initial state
Check setup	check_fv_setup	Checks setup for errors

Table 1. FTA App important functions and commands.

Cadence JasperGold Tutorial

The Superlint App

Version 1.2 | 02-June-2023



Portland State University

©2023

Disclaimer:

The contents of this document are confidential, privileged, and only for the information of the intended recipient and may not be published or redistributed.

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	3
1.1 About and Usage of the Superlint App	4
1.2 Design File	5
1.3 TCL File	6
2. Application Setup	7
2.1 Invoking JasperGold GUI	8
2.2 Invoking the JasperGold Superlint GUI:	12

1. Introduction

While JasperGold (JG) can be used in GUI mode directly without the need for a TCL (Tool Command Language) file, this approach can be time-consuming and inefficient. Therefore, we recommend using TCL scripts to guide JG's actions. These scripts can define various aspects of the analysis, such as the app to be used, the files to be analyzed, and the clock cycles. However, instead of creating TCL scripts from scratch, we can take advantage of pre-made templates. By modifying these templates, we can interact with JG in a way that suits our project's specific requirements.

To get started, we need to set up the VNCserver as described in the previous tutorials and launch the Linux GUI. To keep things organized, we should create a top-level folder for the design we want to analyze. In this example, we'll call it "Superlint_Example," but it's important to avoid using spaces in file and folder names. Within this top-level folder, we can create an optional subfolder named "design" to store the Verilog or SystemVerilog designs we want to analyze. Additionally, we'll create a TCL file in the top-level folder (not inside the "design" subfolder) to store any TCL commands we want to execute.

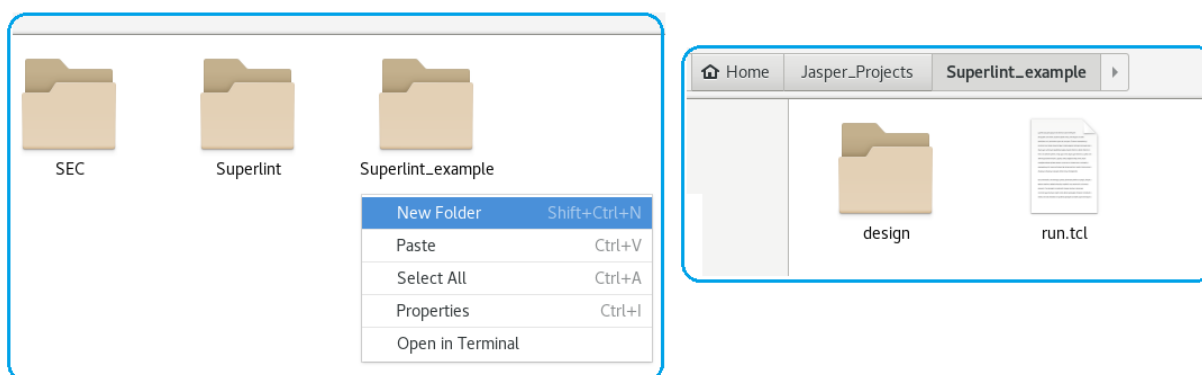


Figure 1.1. Creating folders to organize design and TCL files.

1.1 About and Usage of the Superlint App

The Superlint App is a tool that helps engineers verify complex designs and catch potential issues early in the development process. It does this by running different kinds of checks on the design, such as:

Lint checks, which find common coding errors.

DFT (Design for testability) checks, which ensure that the design is testable, and

Automatic formal checks, which verify that the design meets certain specifications.

By using Superlint, engineers can fix issues before validation begins, which saves time and money. Additionally, Superlint can be used as a way to continually check the design as it evolves, helping to catch any new issues that may arise.

Lint	Superlint
Basic linting typically involves analyzing the RTL code for syntax errors, naming conventions, and coding style. This technique can detect simple errors such as incorrect usage of keywords, missing semicolons, and undefined variables. Basic linting can be performed quickly and is a useful tool for ensuring code quality and consistency.	Superlinting goes beyond basic linting and provides more comprehensive analysis of the RTL code. Superlinting includes DFT (Design-for-Testability) checks, which ensure that the design is testable and can be easily validated during the testing phase. It also includes automatic formal checks, which use mathematical methods to verify that the design meets its specification. Superlinting can detect complex errors such as race conditions, deadlocks, and incorrect use of registers.

In summary, while basic linting is a useful tool for ensuring code quality and consistency, superlinting provides additional features and checks that can identify and fix more complex issues in the RTL code. By using superlinting, engineers can improve the overall quality of their designs and reduce the risk of errors during testing and validation.

We need two inputs for the Superlint app:

- ❖ RTL design files in Verilog/SystemVerilog/VHDL formats
- ❖ TCL file

1.2 Design File

The SystemVerilog design file should be placed in the Design folder, while the TCL file belongs in the main folder. It's recommended to name the folders with lowercase letters since JG is case-sensitive. Additionally, ensure that there are no spaces in the names of the folders or files.

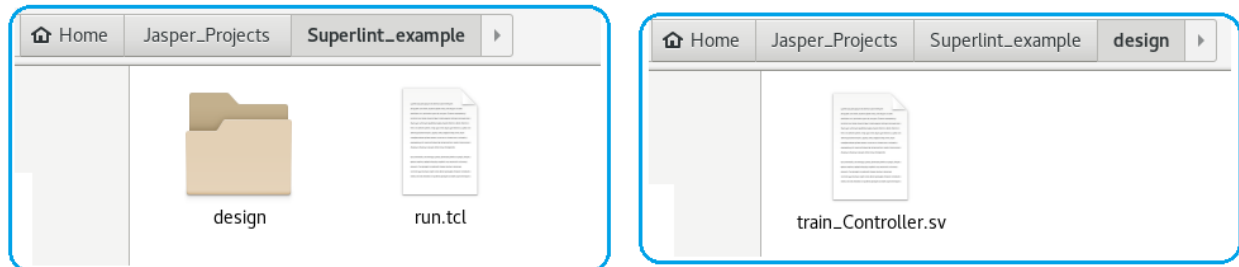
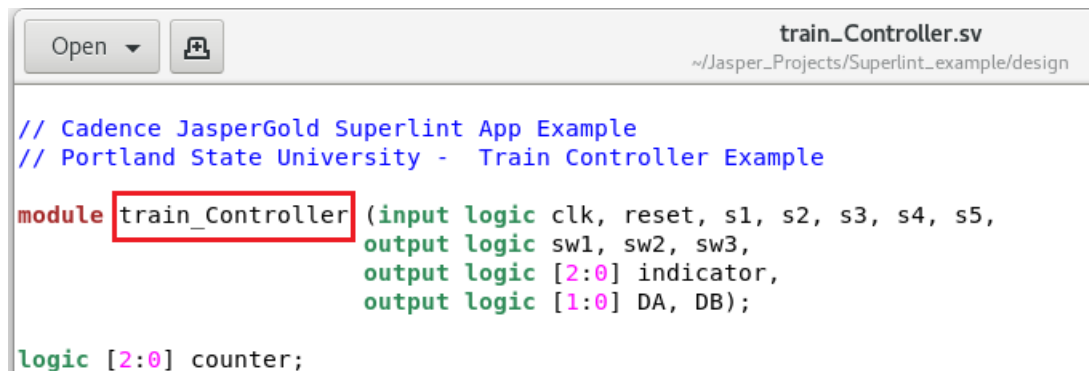


Figure 1.2.1. Showing the SystemVerilog and TCL files in the correct folder locations.

The Superlint App runs automated checks, therefore, writing assertions is not required for this app.

The image shows a code editor window with the title 'train_Controller.sv' and the path '~Jasper_Projects/Superlint_example/design'. The code is as follows:

```
// Cadence JasperGold Superlint App Example
// Portland State University - Train Controller Example

module train_Controller (input logic clk, reset, s1, s2, s3, s4, s5,
                        output logic sw1, sw2, sw3,
                        output logic [2:0] indicator,
                        output logic [1:0] DA, DB);

logic [2:0] counter;
```

The word 'module' is in red, and 'train_Controller' is highlighted with a red rectangular box.

Figure 1.2.2. Example of the design we are using in this tutorial.

When performing formal analysis in general, it's important to keep track of your top module names. In our example here our module name is train_Controller.

1.3 TCL File

Next, we will set up the TCL file. Below is the TCL file (*Figure 1.3.2*), which you can use as a template for your functional checks on VC Formal.



```
*run.tcl
~/Jasper_Projects/Superlint_example

# Cadence JasperGold Superlint App Example
# Portland State University - Formal Verification team

# Initialize the Superlint App
check_superlint -init

# Clear any previous analysis, and read the SystemVerilog Design file
analyze -clear
analyze -sv design/train_Controller.sv (1)

# Elaborate the design
elaborate -bbox_a 1024

# Setup clocks and reset. In our design, the clock signal is "clk"
# and we have a positive edge-triggered reset signal called "reset"
clock clk (2)
reset reset (3)

# Extract the superlint checks
check_superlint -extract

# Generate a task specifically for design assumptions and duplicate the assumptions from the embedded task
task -create design_assumptions -copy_assumes -copy_related_covers -source_task <embedded>

# prove the extracted properties
set_max_trace_length 50
check_superlint -prove -task {<SL_*}
```

Figure 1.3.1. Annotated TCL template file.

- (1) Location of the design file should be specified for JG to locate it.
- (2) Name of the clock signal. If your design uses a clock, you need to type the exact name of your clock. You can also use the `-infer` switch to have JG automatically infer the clock from the design. If you're not using a clock (if it's a combinational logic design), you can use the switch `none`.
- (3) Name of the reset signal in your design. If your design uses a reset signal, you need to type the exact name of your reset. If you're not using a reset, you can use the switch `none`.

2. Application Setup

There are multiple ways to invoke the JasperGold 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 in the “*Superlint_example*” folder.

- [Invoke JasperGold GUI](#), then manually choose the app and load the TCL script in the application:

```
$jg
```

OR

- [Invoke JasperGold GUI in the superlint app](#), then manually load the TCL script in the application:

```
$jg -superlint
```

OR

- [Invoke JasperGold GUI in the superlint app and TCL script](#) in one command:

```
$jg -superlint run.tcl
```

“*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.

We will go through both of these methods in the following sections.

2.1 Invoking JasperGold GUI

To proceed with invoking VC Formal:

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

jg

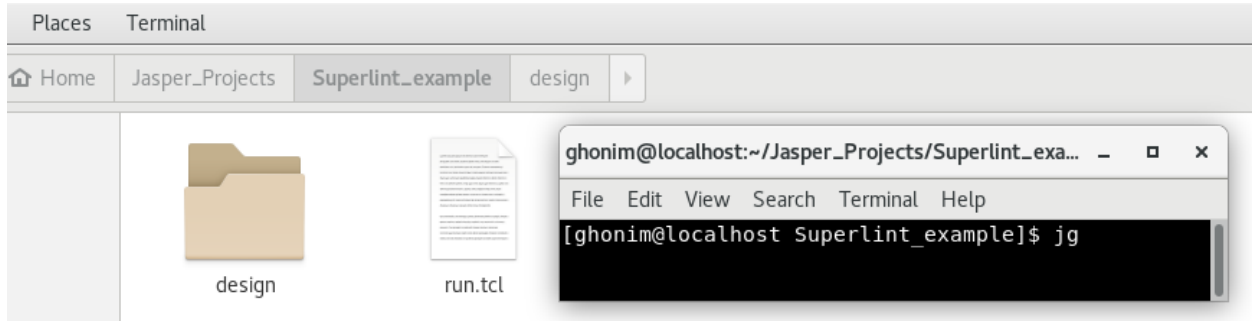


Figure 2.1.1. Invoking JasperGold in the terminal.

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

You should then see the JasperGold GUI as shown in *Figure 2.1.2* below. JasperGold will always open in the Formal Property Verification app unless instructed otherwise in the terminal or the tcl file. To change/add other modes or apps, we can click on the + sign, the one boxed in blue in the figure below. . The GUI interface will look different depending on the app being used.

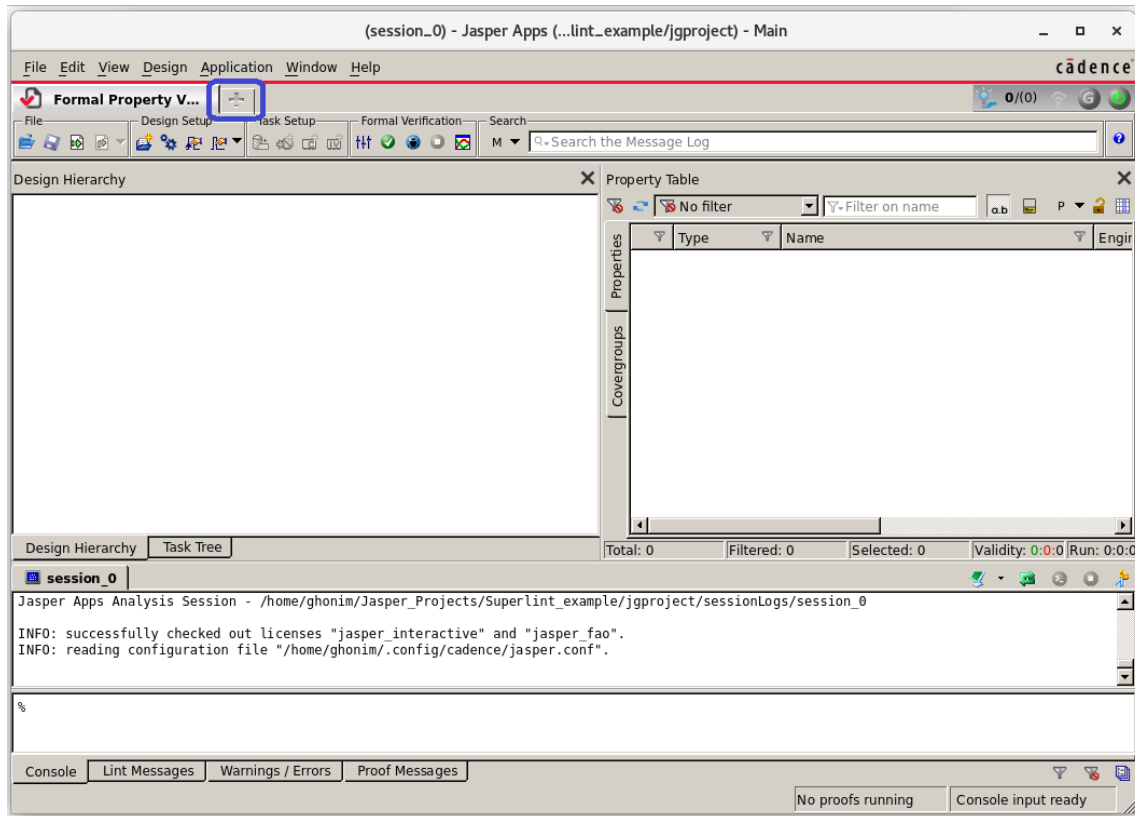


Figure 2.1.2. JasperGold introductory screen in the Formal Property Verification app.

After clicking on the “+” icon, we can then choose the JG app in which we want to work. Here we will choose the Superlint App as shown below.

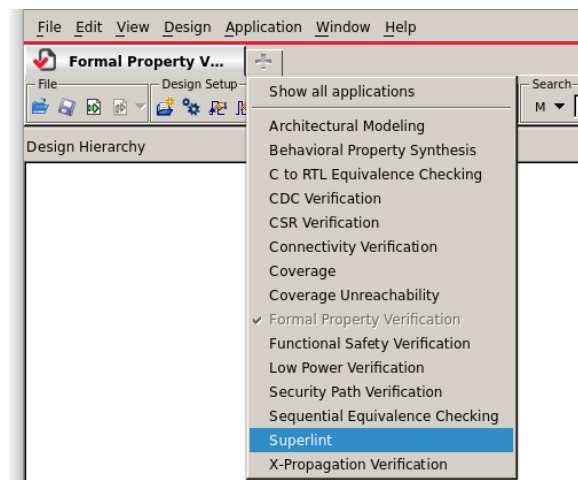


Figure 2.1.3. Selecting the Superlint app

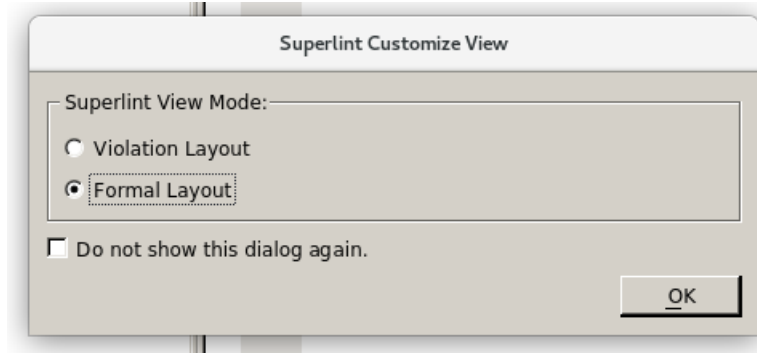


Figure 2.1.4. *Selecting the superlint view mode the first time you run it*

You'll then be asked to choose the Superlint View mode, you can choose either of them depending on your focus of the analysis. You can easily switch between them later. Here we recommend choosing the Formal Layout as a start.

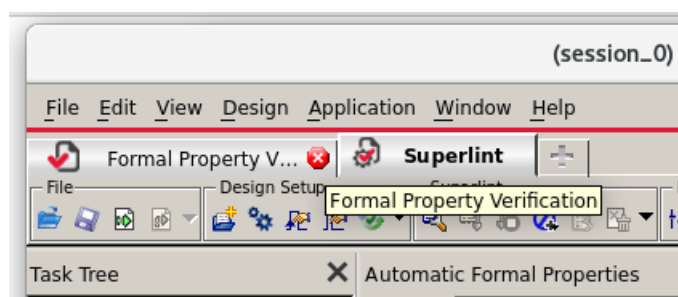


Figure 2.1.5. *Choose the right mode you want to work with “Superlint” in our case. Close any other apps you don’t need.*

If not needed anymore, you can hover over the Formal Property Verification tab and click on “x” to close it.

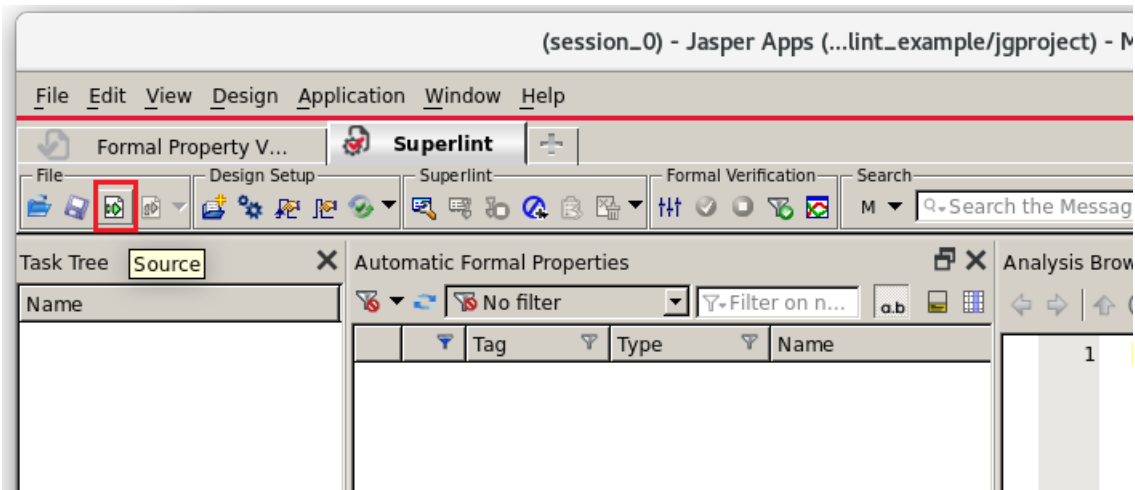


Figure 2.1.6. Superlint App interface. We can click on “Source” to choose and load the TCL file.

To load the TCL file, click on the source icon, which can be found in the File section in the top left, the one boxed in red in the figure above.

2.2 Invoking the JasperGold Superlint GUI:

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

```
hg -superlint
```

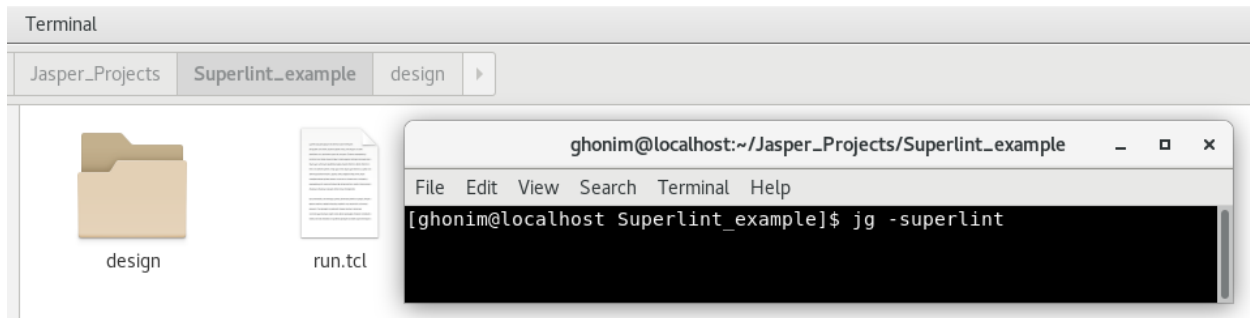


Figure 2.2.1. Invoking JG and choosing the superlint app from the terminal

The last, and most efficient option is to load JasperGold in the app you want “*superlint*” in our case and load the TCL file in the same step as shown below:

```
hg -superlint run.tcl
```

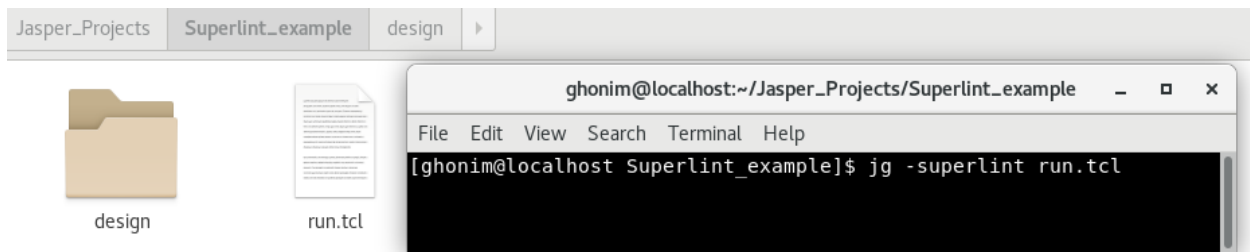


Figure 2.2.2. Invoking JG, Choosing the superlint app, and loading the TCL file from the terminal

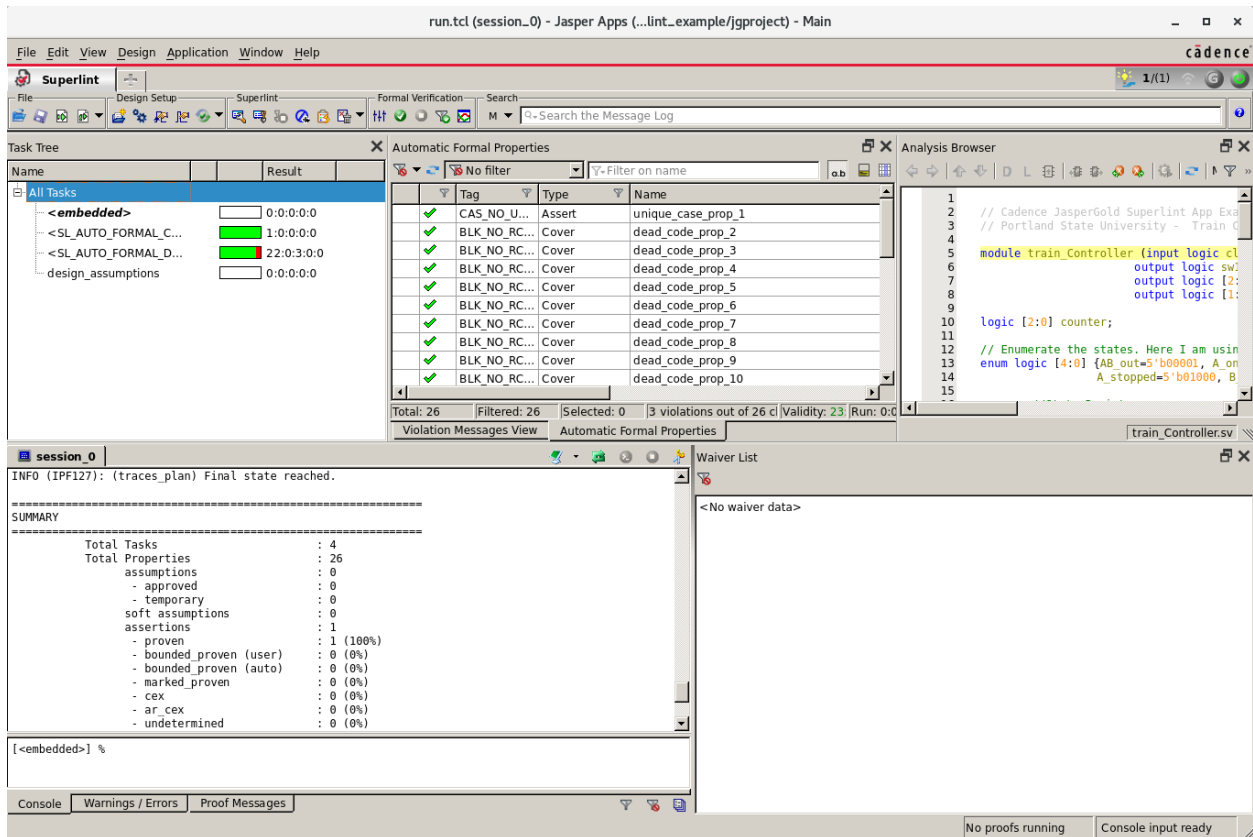


Figure 2.2.3. GUI of the JasperGold Superlint App

This is the interface we get after running JG in the Superlint app and loading the TCL file. Some information and analysis will run automatically given our TCL file, and to run the formal engines, we can click on the green tick icon under formal verification.

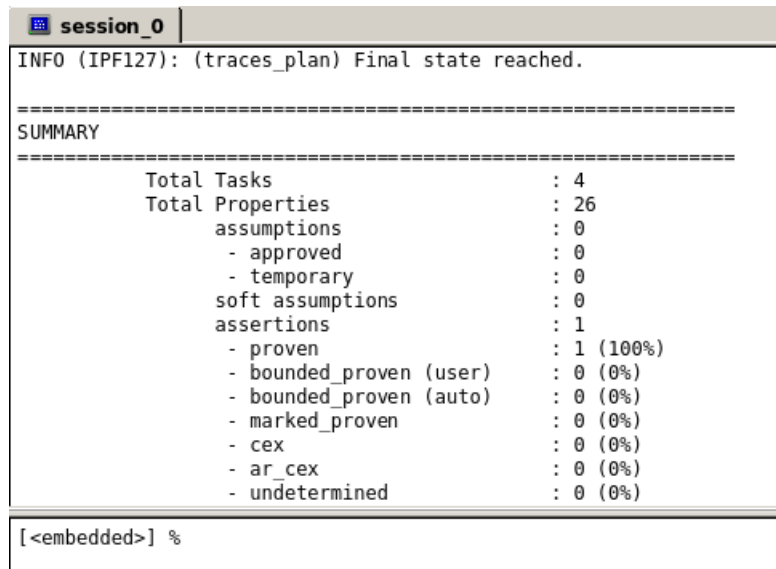


Figure 2.2.4. JG Superlint Session Summary information

We also have a session window of the screen with a summary of the analysis. This summary gives the tutorial number of properties checked, 26 in our case, how many assertions were placed, and so on.

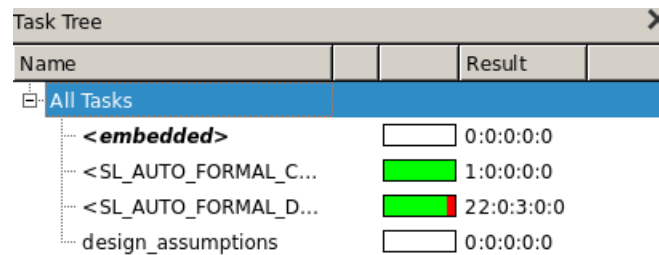
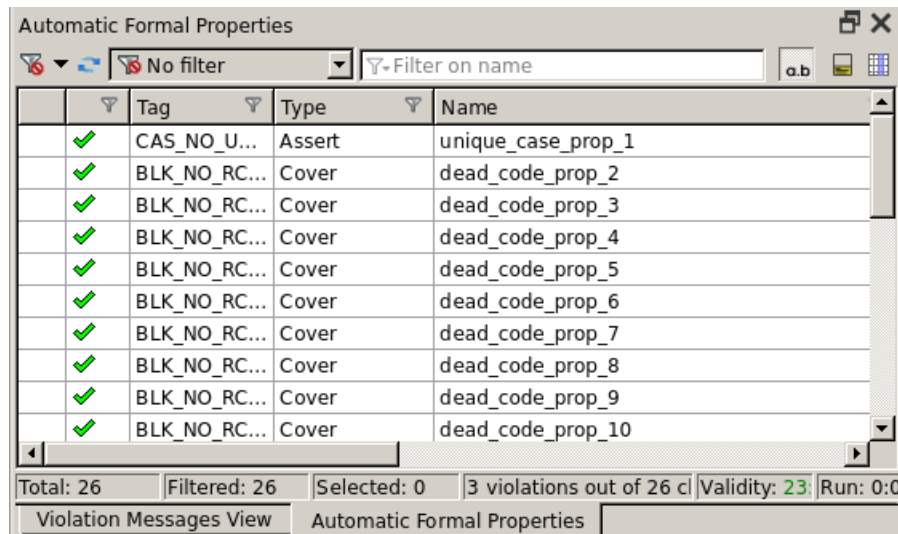


Figure 2.2.5. JG Superlint Task tree progress and results

When we run the analysis, we see that the



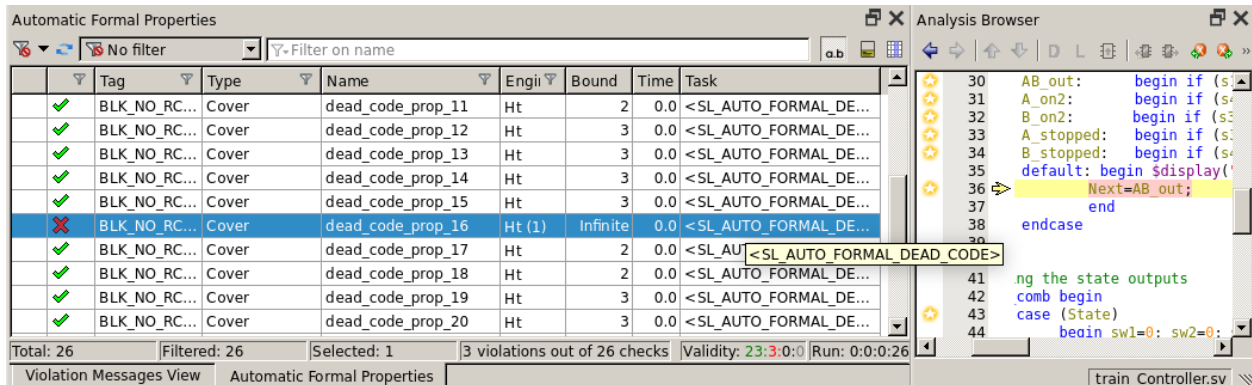
	Tag	Type	Name
✓	CAS_NO_U...	Assert	unique_case_prop_1
✓	BLK_NO_RC...	Cover	dead_code_prop_2
✓	BLK_NO_RC...	Cover	dead_code_prop_3
✓	BLK_NO_RC...	Cover	dead_code_prop_4
✓	BLK_NO_RC...	Cover	dead_code_prop_5
✓	BLK_NO_RC...	Cover	dead_code_prop_6
✓	BLK_NO_RC...	Cover	dead_code_prop_7
✓	BLK_NO_RC...	Cover	dead_code_prop_8
✓	BLK_NO_RC...	Cover	dead_code_prop_9
✓	BLK_NO_RC...	Cover	dead_code_prop_10

Total: 26 Filtered: 26 Selected: 0 3 violations out of 26 c Validity: 23 Run: 0:0

Violation Messages View Automatic Formal Properties

Figure 2.2.6. JG Superlint Automatic Formal Properties view

We can also see in the screen the automatic formal properties that were generated and are being analyzed in this code. Those covers, assertions and properties in general can be verified or falsified.



	Tag	Type	Name	Engin	Bound	Time	Task
✓	BLK_NO_RC...	Cover	dead_code_prop_11	Ht	2	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_12	Ht	3	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_13	Ht	3	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_14	Ht	3	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_15	Ht	3	0.0	<SL_AUTO_FORMAL_DE...
✗	BLK_NO_RC...	Cover	dead_code_prop_16	Ht (1)	Infinite	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_17	Ht	2	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_18	Ht	2	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_19	Ht	3	0.0	<SL_AUTO_FORMAL_DE...
✓	BLK_NO_RC...	Cover	dead_code_prop_20	Ht	3	0.0	<SL_AUTO_FORMAL_DE...

Total: 26 Filtered: 26 Selected: 1 3 violations out of 26 checks Validity: 23:3:0:0 Run: 0:0:0:26

Violation Messages View Automatic Formal Properties

```

30 AB_out: begin if (s
31 A_on2: begin if (s
32 B_on2: begin if (s
33 A_stopped: begin if (s
34 B_stopped: begin if (s
35 default: begin $display(
36     Next=AB_out;
37 end
38 endcase
39
41 ng the state outputs
42 comb begin
43 case (State)
44     begin sw1=0: sw2=0:

```

train_Controller.sv

Figure 2.2.7. JG Superlint Automatic Formal Properties and code line of the falsified cover

In case of a falsified property, we can click on the x sign, or that whole row in general to see the code in the Analysis Brower on the right, with the line causing this cover to be falsified highlighted in red. We can also see the tasks under which these covers are generated, which engines were used for the analysis, and how long it took those engines to complete the analysis.

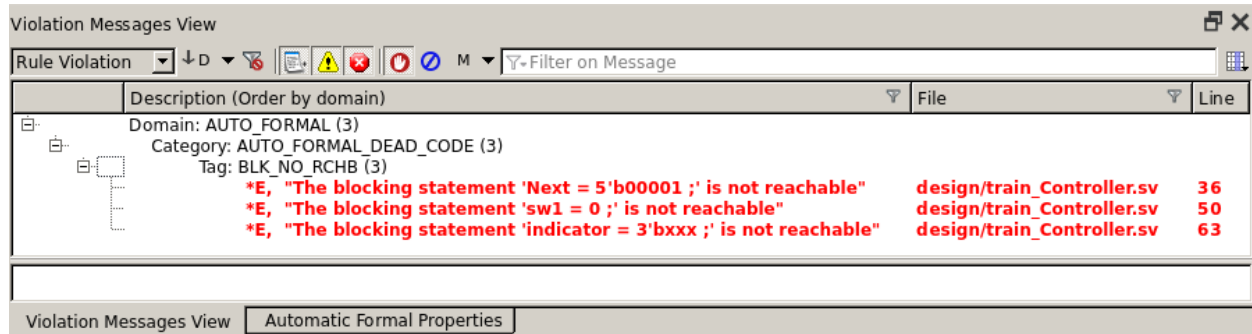


Figure 2.2.8. Violation messages window

In the bottom part of the screen, we also have the Violation messages view, which gives us more insight on the issues causing this specific cover to be falsified. In our case here we have 3 violation messages. There's also a reference to the sv file and line where the issue is found. This is essential for debugging and fixing the code.

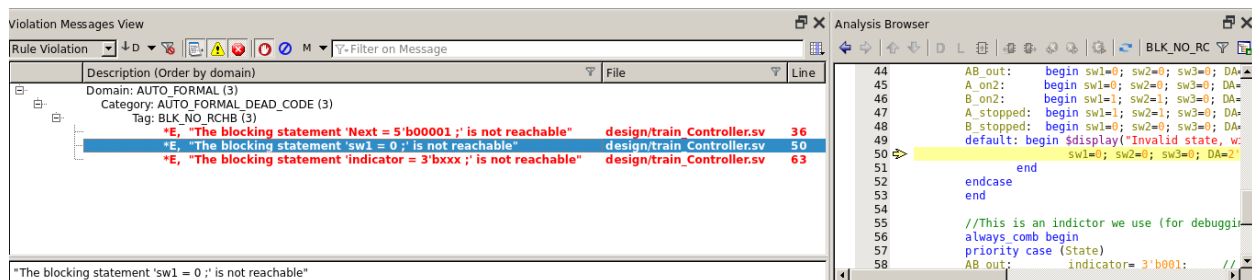


Figure 2.2.9. Violation messages window with reference to the code line

Once the issues are fixed, we can rerun the analysis again to confirm that we are getting any covers, or assertions falsified. Please note that sometimes the falsified properties may not be very meaningful to our design. Remember, these properties are all generated automatically. For example, some falsified covers may not be truly errors or issues in the design

Appendix 3: Sample ECE 582/682 Projects performed using our tutorials.
ECE 582/682 Formal Verification of Hardware and Software Systems

Project 2

Mohamed Ghonim

Alexander Maso

02/18/2023

Honor Pledge

On my honor, I have neither given nor received unauthorized aid on this project.

Mohamed Ghonim.

On my honor, I have neither given nor received unauthorized aid on this project.

Alexander Maso

A red handwritten signature in cursive script that reads "Mohamed Ghonim".A black handwritten signature in cursive script, likely reading "Alexander Maso".

Portland State University

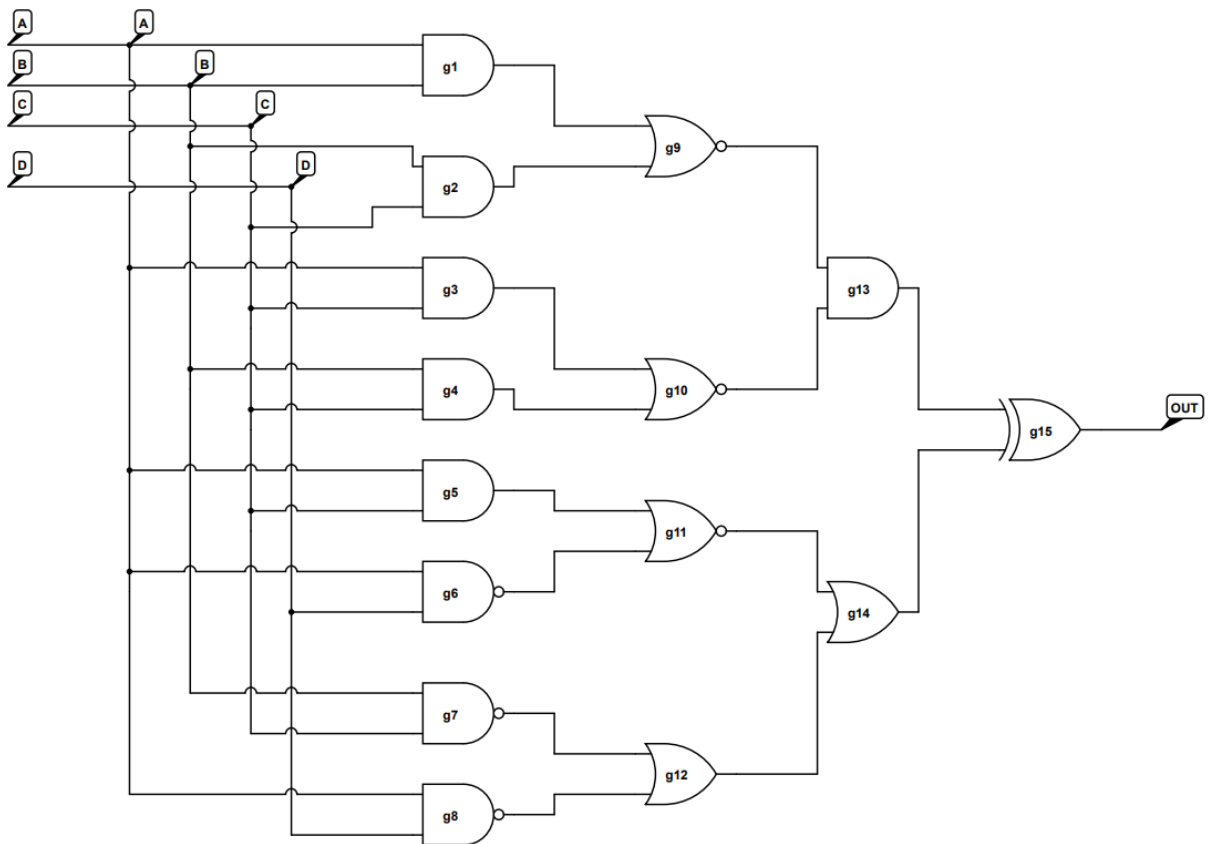
Problem 1:

You design a combinational circuit C1 with the following requirements. The combinational logic C1 should contain at least 15 gates which include at least 3 NOR gates, 3 NAND gates, and one XOR gate. The inputs of a gate cannot be tied together as a single input. Inverters are not counted for the 15 gates. Make sure that your design is different from those used by other students. If your design is identical to the one used by other students, further investigation will be conducted.

Solution

- 1) Draw your circuit C1.
- 2) You make an identical copy of C1 and name it C2. Keep the input and output names as well as the module names identical.

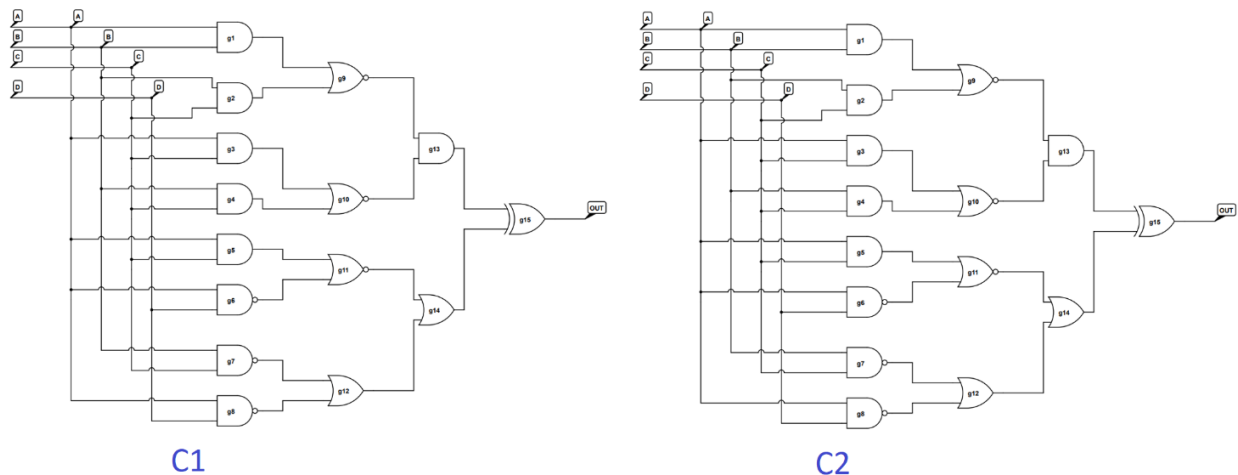
C1 and identical C2 drawing:



SystemVerilog code for C1 (and identical C2) in both continuous assignment and always comb block.

Ln#	C1: C:/IntelFPGA_pro/22.3/C1_circuit.sv	C2: C:/IntelFPGA_pro/22.3/C1_circuit_always_block.sv - Default
1	// Mohamed Ghonim - Alexander Maso	// Mohamed Ghonim - Alexander Maso
2	// ECE 582/682 Verification of Hardware and Software Systems	// ECE 582/682 Verification of Hardware and Software Systems
3	// Project 2 Combinational Circuit	// Project 2 Combinational Circuit
4	// This is the original circuit C1, in continuous assignment	// This is the original circuit C1, in an always_comb block
5		
6	module comb_mode (input bit a, b, c, d, output bit out);	module comb_mode (input bit a, b, c, d, output bit out);
7		
8	bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,	bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
9	g11, g12, g13, g14, g15;	g11, g12, g13, g14, g15;
10		
11	assign g1 = a & b;	always_comb begin
12	assign g2 = b & c;	g1 = a & b;
13	assign g3 = a & c;	g2 = b & c;
14	assign g4 = b & c;	g3 = a & c;
15	assign g5 = a & c;	g4 = b & c;
16	assign g6 = ~(a & d);	g5 = a & c;
17	assign g7 = ~(b & c);	g6 = ~(a & d);
18	assign g8 = ~(a & d);	g7 = ~(b & c);
19		g8 = ~(a & d);
20	assign g9 = ~(g1 g2);	
21	assign g10 = ~(g4 g4);	g9 = ~(g1 g2);
22	assign g11 = ~(g5 g6);	g10 = ~(g4 g4);
23	assign g12 = (g7 g8);	g11 = ~(g5 g6);
24		g12 = (g7 g8);
25	assign g13 = g9 & g10;	
26	assign g14 = g11 g12;	g13 = g9 & g10;
27		g14 = g11 g12;
28	assign g15 = g13 ^ g14;	
29	assign out = g15;	g15 = g13 ^ g14;
30		out = g15;
31	endmodule: comb_mode	end
32		endmodule: comb_mode
33		
34		

C1 and identical C2 next to each other:



C1 and identical C2 next to each other (code)

```
// Mohamed Ghonim - Alexander Maso
// ECE 582/682 Verification of Hardware and Software Systems
// Project 2 Combinational Circuit
// This is circuit C2, in an always_comb block

module comb_mode (input bit a, b, c, d, output bit out);

    bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
        g11, g12, g13, g14, g15;

    always_comb begin
        g1 = a & b;
        g2 = b & c;
        g3 = a & c;
        g4 = b & c;
        g5 = a & c;
        g6 = ~(a & d);
        g7 = ~(b & c);
        g8 = ~(a & d);

        g9 = ~(g1 | g2);
        g10 = ~(g4 | g4);
        g11 = ~(g5 | g6);
        g12 = (g7 | g8);

        g13 = g9 & g10;
        g14 = g11 | g12;

        g15 = g13 ^ g14;
        out = g15;
    end
endmodule: comb_mode
```

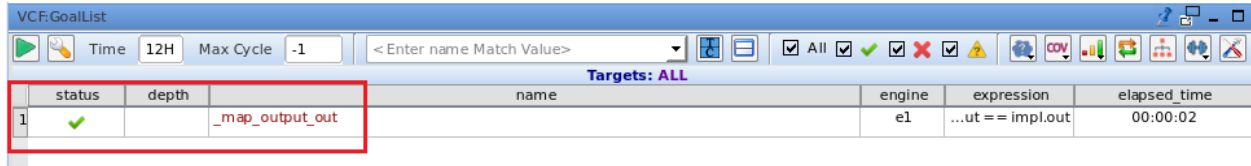
```
1
2
3 // Mohamed Ghonim - Alexander Maso
4 // ECE 582/682 Verification of Hardware and Software Systems
5 // Project 2 Combinational Circuit
6 // This is the original circuit C1, in an always_comb block
7
8 module comb_mode (input bit a, b, c, d, output bit out);
9
10     bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
11         g11, g12, g13, g14, g15;
12
13     always_comb begin
14         g1 = a & b;
15         g2 = b & c;
16         g3 = a & c;
17         g4 = b & c;
18         g5 = a & c;
19         g6 = ~(a & d);
20         g7 = ~(b & c);
21         g8 = ~(a & d);
22
23         g9 = ~(g1 | g2);
24         g10 = ~(g4 | g4);
25         g11 = ~(g5 | g6);
26         g12 = (g7 | g8);
27
28         g13 = g9 & g10;
29         g14 = g11 | g12;
30
31         g15 = g13 ^ g14;
32         out = g15;
33     end
34 endmodule: comb_mode
```

Tasks:

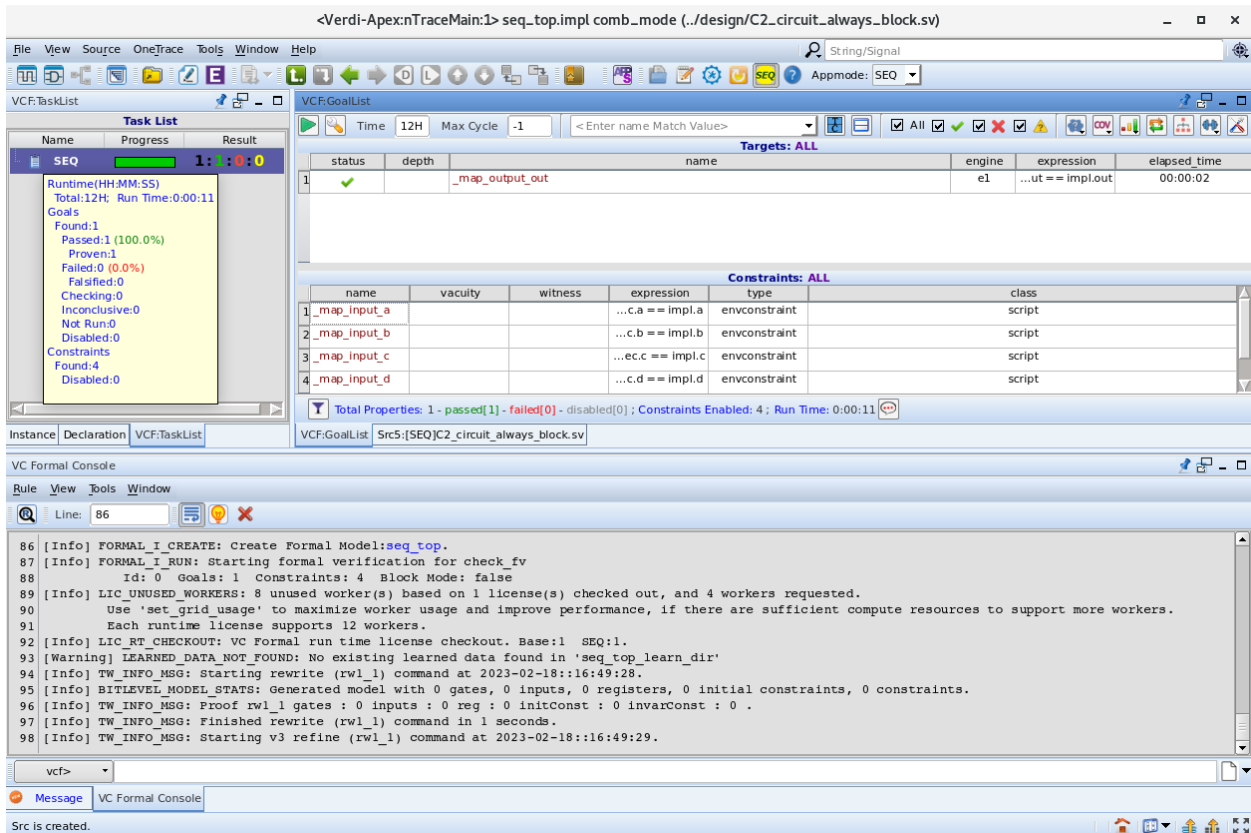
- 1) Perform the equivalence check on C1 and C2 with the SEQ App in VC Formal.

After following the tutorial steps and getting VC Formal running in the SEQ app, we ran the analysis and found that the two designs were equivalent as show below.

The SEQ analysis has passed. (Since we have this green tick sign under status)



status	depth	name	engine	expression	elapsed_time
1		_map_output_out	e1	...ut == impl.out	00:00:02



Task List

Name	Progress	Result
SEQ	100%	1:0:0

Runtime(HH:MM:SS)
Total:12H Run Time:0:00:11
Goals
Found:1
Passed:1 (100.0%)
Proven:1
Failed:0 (0.0%)
Falsified:0
Checking:0
Inconclusive:0
Not Run:0
Disabled:0
Constraints
Found:4
Disabled:0

VCF:GoalList

status	depth	name	engine	expression	elapsed_time
1		_map_output_out	e1	...ut == impl.out	00:00:02

Constraints: ALL

name	vacuity	witness	expression	type	class
1_map_input_a			...c.a == impl.a	envconstraint	script
2_map_input_b			...c.b == impl.b	envconstraint	script
3_map_input_c			...c.c == impl.c	envconstraint	script
4_map_input_d			...c.d == impl.d	envconstraint	script

Total Properties: 1 - passed[1] - failed[0] - disabled[0]; Constraints Enabled: 4; Run Time: 0:00:11

VCF Formal Console

```

86 [Info] FORMAL_I_CREATE: Create Formal Model:seq_top.
87 [Info] FORMAL_I_RUN: Starting formal verification for check_fv
88     Id: 0 Goals: 1 Constraints: 4 Block Mode: false
89 [Info] LIC_UNUSED_WORKERS: 8 unused worker(s) based on 1 license(s) checked out, and 4 workers requested.
90     Use 'set_grid_usage' to maximize worker usage and improve performance, if there are sufficient compute resources to support more workers.
91     Each runtime license supports 12 workers.
92 [Info] LIC_RT_CHECKOUT: VC Formal run time license checkout. Base:1 SEQ:1.
93 [Warning] LEARNED_DATA_NOT_FOUND: No existing learned data found in 'seq_top_learn_dir'
94 [Info] TW_INFO_MSG: Starting rewrite (rw1_1) command at 2023-02-18:16:49:28.
95 [Info] BITLEVEL_MODEL_STATS: Generated model with 0 gates, 0 inputs, 0 registers, 0 initial constraints, 0 constraints.
96 [Info] TW_INFO_MSG: Proof rw1_1 gates : 0 inputs : 0 reg : 0 initConst : 0 invarConst : 0 .
97 [Info] TW_INFO_MSG: Finished rewrite (rw1_1) command in 1 seconds.
98 [Info] TW_INFO_MSG: Starting v3 refine (rw1_1) command at 2023-02-18:16:49:29.
  
```

vcf>

Message VC Formal Console

Src is created.

As the two designs are equivalent, there's not much more that the SEQ App can tell us, whereas if the two designs are not equivalent, we can trace the signal back to its driving signals to try to detect the gate, or the point at which the inequivalence was first introduced.

<Verdi-Apex:nTraceMain:1> seq_top.spec comb_mode (../design/C1_circuit_always_block.sv)

File View Source OneTrace Tools Window Help

String/Signal

Appmode: SEQ

VC:TaskList

Task List

Name	Progress	Result
SEQ		1:0:0

Runtime(HH:MM:SS)
Total:12H Run Time:0:00:11
Goals
Found:1
Passed:1 (100.0%)
Proven:1
Failed:0 (0.0%)
Falsified:0
Checking:0
Inconclusive:0
Not Run:0
Disabled:0
Constraints
Found:4
Disabled:0

Src5:[SEQ]seq_top.spec(/home/ghonim/vcfmo/582C1C2/design/C1_circuit_always_block.sv)

[spec]

```

2 // Mohamed Ghonim - Alexander Maso
3 // ECE 582/682 Verification of Hardware and Software Systems
4 // Project 2 Combinational Circuit
5 // This is the original circuit C1, in an always_comb block
6
7 module comb_mode (input bit a, b, c, d, output bit out);
8
9 bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12, g13,
10  g14, g15;
11
12 always_comb begin
13     g1 = a & b;
14     g2 = b & c;
15     g3 = a & c;
16     g4 = b & c;
17     g5 = a & c;
18     g6 = ~(a & d);
19     g7 = ~(b & c);
20     g8 = ~(a & d);
21
22     g9 = ~(g1 | g2);
23     g10 = ~(g4 | g4);
24     g11 = ~(g5 | g6);
25     g12 = (g7 | g8);
26
27     g13 = g9 & g10;
28     g14 = g11 & g12;
29
30     g15 = g13 ^ g14;
31     out = g15;
32 end
endmodule: comb_mode

```

[impl]

```

2 // Mohamed Ghonim - Alexander Maso
3 // ECE 582/682 Verification of Hardware and Software System
4 // Project 2 Combinational Circuit
5 // This is the original circuit C1, in an always_comb block
6
7 module comb_mode (input bit a, b, c, d, output bit out);
8
9 bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12, g13,
10  g14, g15;
11
12 always_comb begin
13     g1 = a & b;
14     g2 = b & c;
15     g3 = a & c;
16     g4 = b & c;
17     g5 = a & c;
18     g6 = ~(a & d);
19     g7 = ~(b & c);
20     g8 = ~(a & d);
21
22     g9 = ~(g1 | g2);
23     g10 = ~(g4 | g4);
24     g11 = ~(g5 | g6);
25     g12 = (g7 | g8);
26
27     g13 = g9 & g10;
28     g14 = g11 & g12;
29
30     g15 = g13 ^ g14;
31     out = g15;
32 end
endmodule: comb_mode

```

Instance Declaration VCF:TaskList

VCF:GoalList Src5:[SEQ]C1_circuit_always_block.sv

VC Formal Console

Rule View Tools Window

Line: 97

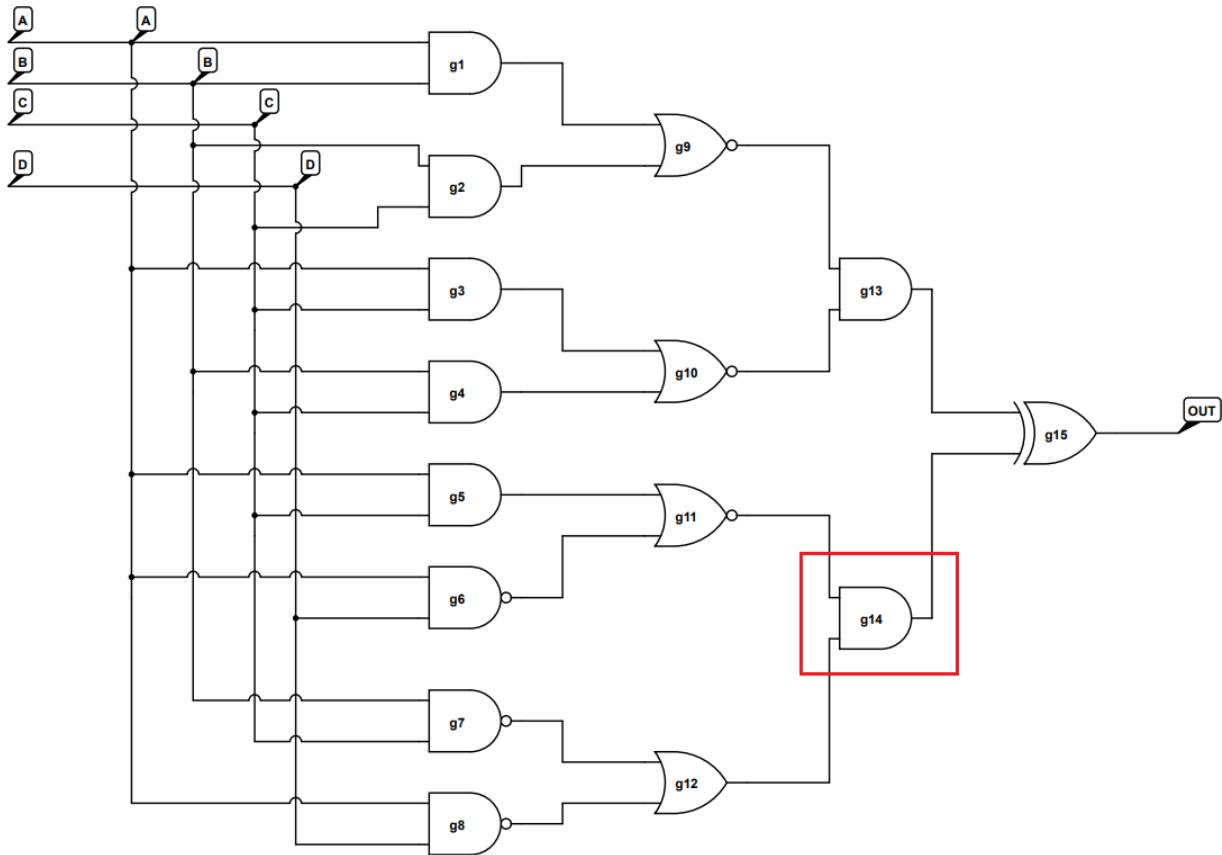
97 [Info] TW_INFO_MSG: Finished rewrite (rw1_1) command in 1 seconds.
98 [Info] TW_INFO_MSG: Starting v3 refine (rw1_1) command at 2023-02-18::16:49:29.

vcf>

Message VC Formal Console

- 2) Replace one gate of C1 with a different gate so that C1 is different from C2. Perform the equivalence check on C1 and C2 with the SEQ App in VC Formal.
-

Modified C2 drawing:



SystemVerilog code for the modified C2 Circuit in both continuous assignment and
always comb block.

C:/intelFPGA_pro/22.3/C2_circuit_always_block.sv - Default

```

Ln#
1 // Mohamed Ghonim - Alexander Maso
2 // ECE 582/682 Verification of Hardware and Software Systems
3 // Project 2 Combinational Circuit
4 // This is the modified circuit C2, in an always_comb block
5
6 module comb_mode (input bit a, b, c, d, output bit out);
7
8     bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
9       g11, g12, g13, g14, g15;
10
11     always_comb begin
12         g1 = a & b;
13         g2 = b & c;
14         g3 = a & c;
15         g4 = b & c;
16         g5 = a & c;
17         g6 = ~(a & d);
18         g7 = ~(b & c);
19         g8 = ~(a & d);
20
21         g9 = ~(g1 | g2);
22         g10 = ~(g4 | g4);
23         g11 = ~(g5 | g6);
24         g12 = (g7 | g8);
25
26         g13 = g9 & g10;
27         g14 = g11 & g12;
28         // We changed this g14 gate from OR to AND
29         g15 = g13 ^ g14;
30         out = g15;
31     end
32 endmodule: comb_mode
33

```

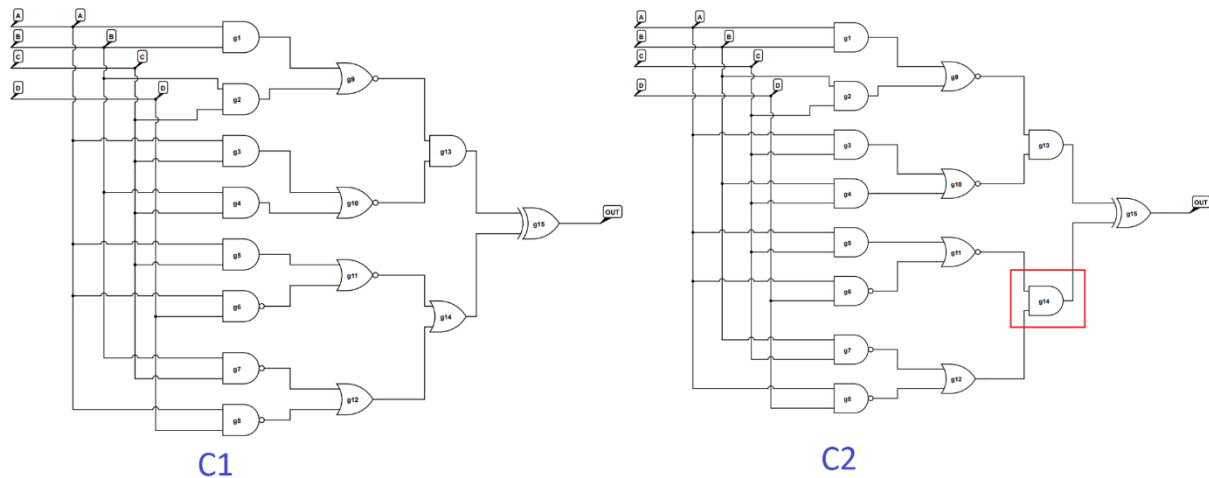
C:/intelFPGA_pro/22.3/C2_circuit.sv

```

Ln#
1 // Mohamed Ghonim - Alexander Maso
2 // ECE 582/682 Verification of Hardware and Software Systems
3 // Project 2 Combinational Circuit
4 // This is the modified circuit C2, in continuous assignment
5
6 module comb_mode (input bit a, b, c, d, output bit out);
7
8     bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
9       g11, g12, g13, g14, g15;
10
11     assign g1 = a & b;
12     assign g2 = b & c;
13     assign g3 = a & c;
14     assign g4 = b & c;
15     assign g5 = a & c;
16     assign g6 = ~(a & d);
17     assign g7 = ~(b & c);
18     assign g8 = ~(a & d);
19
20     assign g9 = ~(g1 | g2);
21     assign g10 = ~(g4 | g4);
22     assign g11 = ~(g5 | g6);
23     assign g12 = (g7 | g8);
24
25     assign g13 = g9 & g10;
26     assign g14 = g11 & g12;
27     // We changed this g14 gate from OR to AND
28     assign g15 = g13 ^ g14;
29     assign out = g15;
30
31 endmodule: comb_mode
32

```

C1 and modified C2 next to each other:



C1 and modified C2 next to each other (code)

The image displays two side-by-side screenshots of the IntelFPGA Pro IDE, showing Verilog code for a combinational circuit. The left window shows the original circuit C2, and the right window shows the modified circuit C1.

Left Window (C2):

```
// Mohamed Ghoniem - Alexander Maso
// ECE 582/682 Verification of Hardware and Software Systems
// Project 2 Combinational Circuit
// This is the modified circuit C2, in an always_comb block

module comb_mode (input bit a, b, c, d, output bit out);
    bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
        g11, g12, g13, g14, g15;

    always_comb begin
        g1 = a & b;
        g2 = b & c;
        g3 = a & c;
        g4 = b & c;
        g5 = a & c;
        g6 = ~(a & d);
        g7 = ~(b & c);
        g8 = ~(a & d);

        g9 = ~(g1 | g2);
        g10 = ~(g4 | g4);
        g11 = ~(g5 | g6);
        g12 = (g7 | g8);

        g13 = g9 & g10;
        g14 = g11 & g12;
        // We changed this g14 gate from OR to AND
        g15 = g13 ^ g14;
        out = g15;
    end
endmodule: comb_mode
```

Right Window (C1):

```
// Mohamed Ghoniem - Alexander Maso
// ECE 582/682 Verification of Hardware and Software Systems
// Project 2 Combinational Circuit
// This is the original circuit C1, in an always_comb block

module comb_mode (input bit a, b, c, d, output bit out);
    bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
        g11, g12, g13, g14, g15;

    always_comb begin
        g1 = a & b;
        g2 = b & c;
        g3 = a & c;
        g4 = b & c;
        g5 = a & c;
        g6 = ~(a & d);
        g7 = ~(b & c);
        g8 = ~(a & d);

        g9 = ~(g1 | g2);
        g10 = ~(g4 | g4);
        g11 = ~(g5 | g6);
        g12 = (g7 | g8);

        g13 = g9 & g10;
        g14 = g11 | g12;
        g15 = g13 ^ g14;
        out = g15;
    end
endmodule: comb_mode
```

VC Formal SEQ Checking on C1 and C2

VCF:GoalList

Time 12H Max Cycle -1

< Enter name Match ...

Targets: ALL

	status	depth	name	engine	expression	elapsed_time
1	X	0	_map_output_out	e1	...ut == impl.out	00:00:01

We see here that the SEQ analysis has failed, meaning that the two designs are NOT equivalent.

<Verdi-Apex:nTraceMain:1> seq_top.impl comb_mode (../design/C2_circuit_always_block.sv)

File View Source OneTrace Tools Window Help

String/Signal

Appmode: SEQ

VCF:TaskList

Name	Progress	Result
SEQ		1:0:1:0

Runtime(HH:MM:SS)
Total:12H: Run Time:0:00:09
Goals
Found:1
Passed:0 (0.0%)
Proven:0
Failed:1 (100.0%)
Falsified:1
Checking:0
Inconclusive:0
Not Run:0
Disabled:0
Constraints
Found:4
Disabled:0

VCF:GoalList

Time 12H Max Cycle -1

Targets: ALL

status	depth	name	engine	expression	elapsed_time
✗	0	_map_output_out	e1	...ut == impl.out	00:00:01

Constraints: ALL

name	vacuity	witness	expression	type	class
1 _map_input_a			...c.a == impl.a	envconstraint	script
2 _map_input_b			...c.b == impl.b	envconstraint	script
3 _map_input_c			...ec.c == impl.c	envconstraint	script
4 _map_input_d			...c.d == impl.d	envconstraint	script

Total Properties: 1 - passed[0] - failed[1] - disabled[0] ; Constraints Enabled: 4 ; Min depth: 0 ; Max depth: 0 ; Run Time: 0:00:09

Instance Declaration VCF:TaskList

VCF:GoalList Src9:[SEQ]C2_circuit_always_block.sv

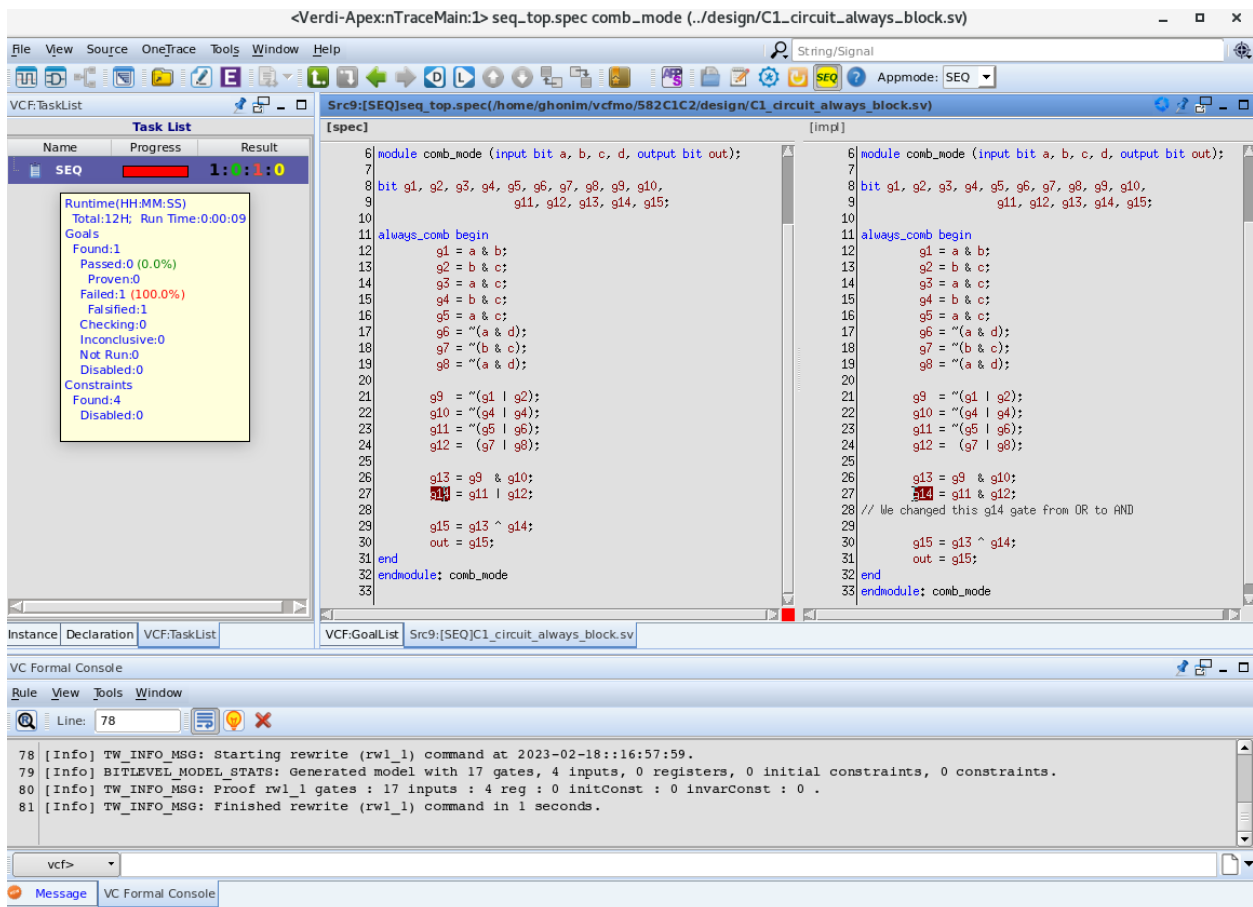
VC Formal Console

Rule View Tools Window

Line: 80

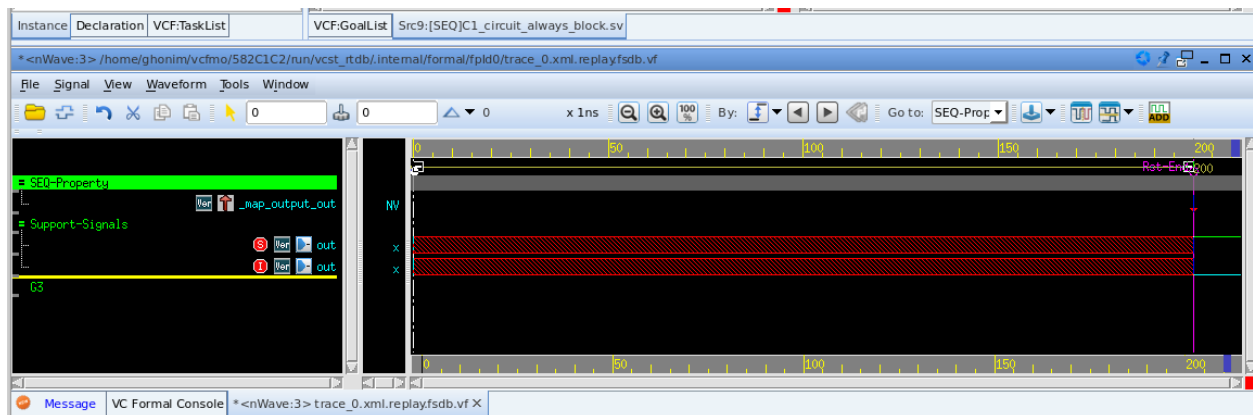
80 [Info] TW_INFO_MSG: Proof rw1_1 gates : 17 inputs : 4 reg : 0 initConst : 0 invarConst : 0 .
81 [Info] TW_INFO_MSG: Finished rewrite (rw1_1) command in 1 seconds.

vcf>



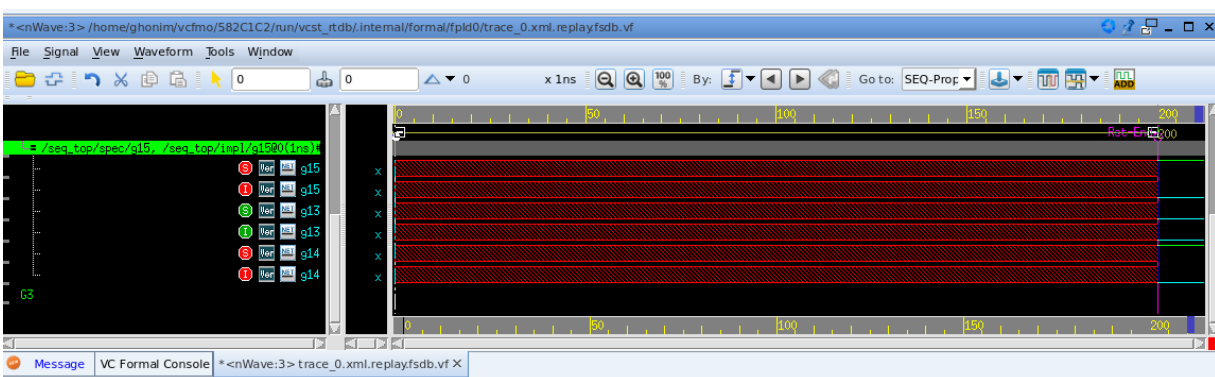
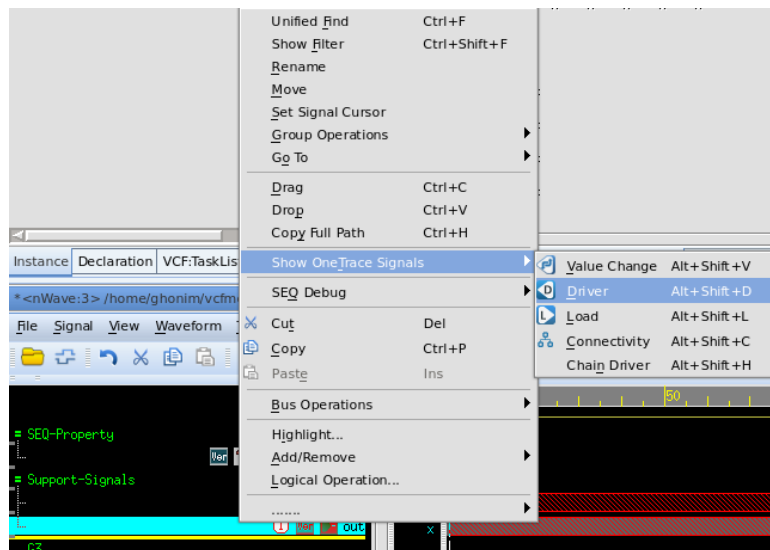
(This is additional/not required in the submission)

Furthermore, as shown in the tutorial, the VC Formal SEQ app can trace and detect the source of the inequivalence. To do this, we can double-click on the X sign under status to get the waveforms on the button left.

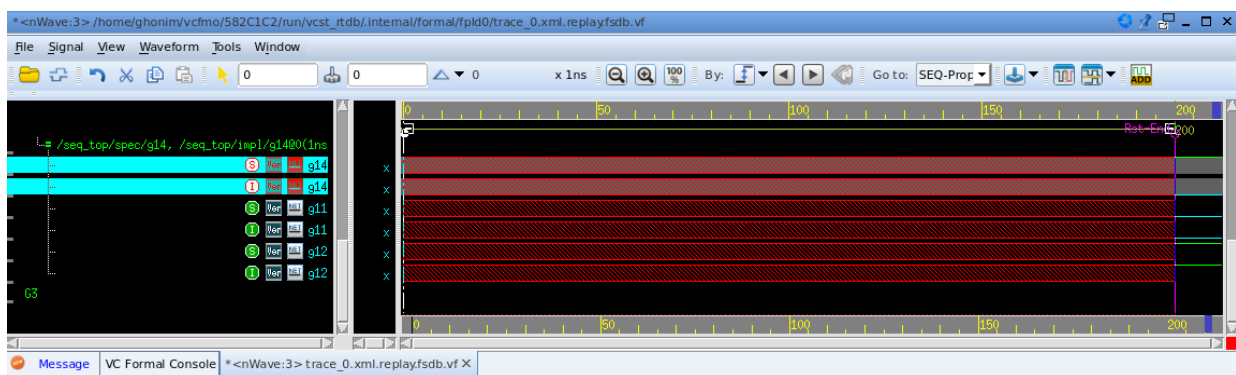


The red S and I indicate the inequivalence. We will now right-click on the I “implementation design” and choose “Show OneTrace Signals=> Driver” to trace the output back to the

signal that caused the inequivalence. We can repeat this process a few times to get to this screen.



Here we see that the output signal is driven by g15, as we trace g15 we find that it's driven by g13 and g14. G13 is green in the S "specification" and the I "Implementation" designs, which means those are equivalent, g14 however is not equivalent in the two designs. We can trace back g14 to find that g11 and g12 are equivalent as shown below:



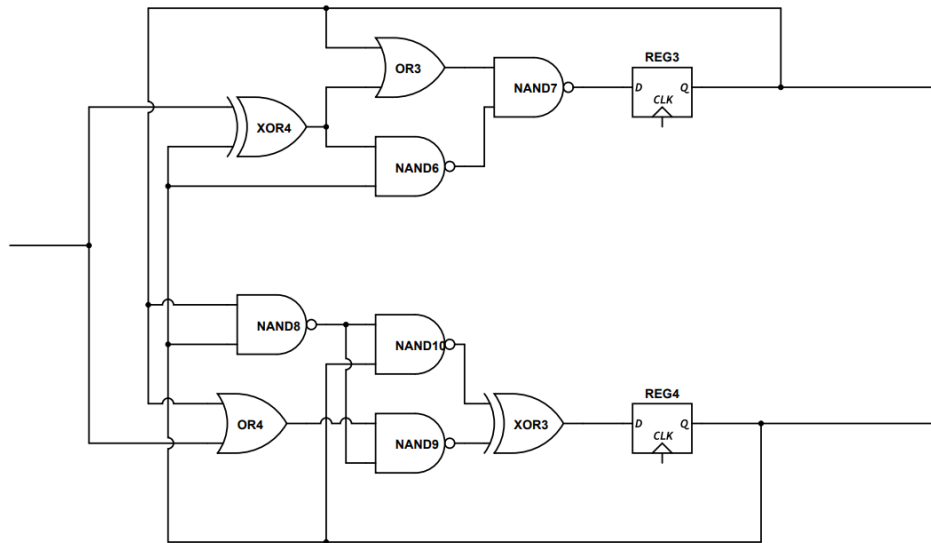
This means that the inequivalence happened or was introduced at g14.

Problem 2:

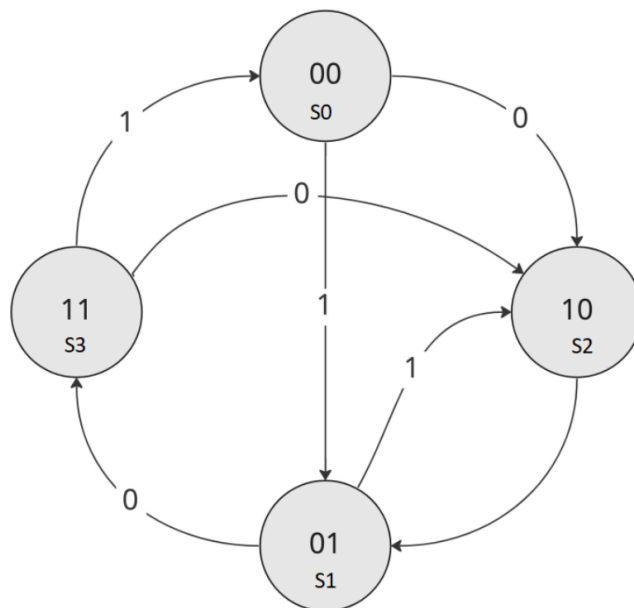
Consider the sequential circuits S1 and S2 in your Project 1.

Solution

S1 Circuit Drawing/Structure



FSM representing S1



All the codes compiled successfully

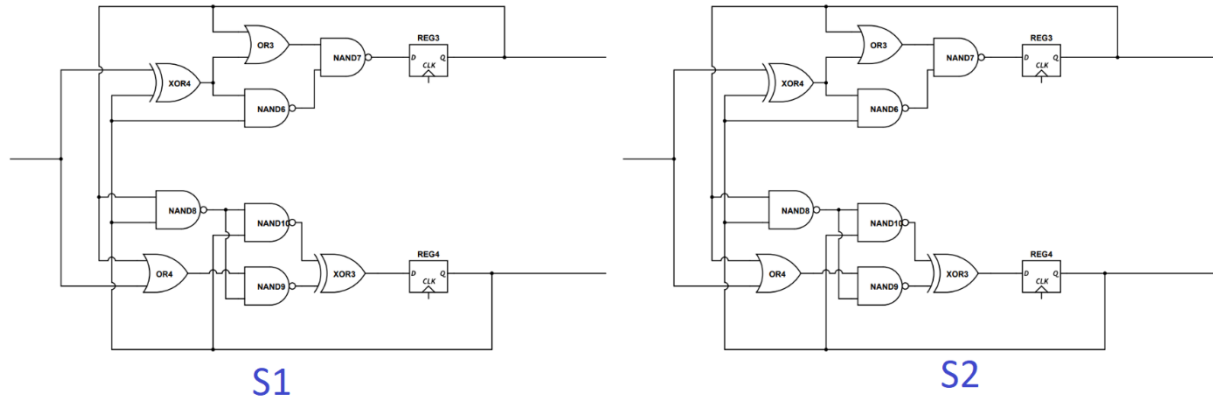
Project - C:\intelFPGA_pro\22.3\Project2					
Name	Status	Type	Order	Modified	
C1_circuit.sv	✓	SystemVerilog	0	02/17/2023 06:51:27 ...	
C1_circuit_always_block.sv	✓	SystemVerilog	2	02/17/2023 06:50:25 ...	
C2_circuit.sv	✓	SystemVerilog	1	02/17/2023 06:52:55 ...	
C2_circuit_always_block.sv	✓	SystemVerilog	3	02/17/2023 06:54:56 ...	
S1.sv	✓	SystemVerilog	4	02/17/2023 10:13:39 ...	
S2.sv	✓	SystemVerilog	5	02/17/2023 10:15:41 ...	

SV Code/RTL model represeting S1 (FSM behaviorial model)

```
C:\intelFPGA_pro\22.3\S1.sv - Default
Ln#
1
2 // Mohamed Ghonim - Alexander Maso
3 // ECE 582/682 Verification of Hardware and Software Systems
4 // Project 2 Sequential Circuit
5 // This is the original circuit S1
6
7 module S_design (input logic a, clk, reset, output logic [1:0] out);
8
9 // Enumerate the states. Here we are using binary encoding
10 enum logic [1:0] {S0, S1, S2, S3} State, Next;
11
12
13 //State Register
14 always_ff @(posedge clk, negedge reset) begin //Negative edge triggered asynchronous reset
15 if (reset) State <= S0; else // in case of a reset, go to the initial state S0
16 State <= Next; // Go to the next state with each positive clock edge
17 end
18
19 //Next State Logic
20 always_comb begin
21 case (State) // Check the case
22 S0: if (a) Next=S1; else Next=S2;
23 S1: if (a) Next=S2; else Next=S3;
24 S2: Next=S1;
25 S3: if (a) Next=S0; else Next=S2;
26 default: begin $display("Invalid state, will reset state S0"); Next=S0; end
27 endcase
28 end
29
30 //Setting the state outputs
31 always_comb begin
32 case (State)
33 S0: out=2'b00;
34 S1: out=2'b01;
35 S2: out=2'b10;
36 S3: out=2'b11;
37 default: out=2'b00;
38 endcase
39 end
40
41 endmodule: S_design
```


Task 2.1: Perform the equivalence check on S1 and S2 with the SEQ App in VC Formal.

S1 and S2 next to each other



When we run the VC Formal SEQ analysis on S1 and S2, the verification passes as shown below:

VC:TaskList

Name	Progress	Result
SEQ	1:1:0:0	1:1:0:0

Runtime(HH:MM:SS)
Total:12H; Run Time:0:00:10
Goals
Found:1
Passed:1 (100.0%)
Proven:1
Failed:0 (0.0%)
Falsified:0
Checking:0
Inconclusive:0
Not Run:0
Disabled:0
Constraints
Found:4
Disabled:0

VC:GoalList

status	depth	name	engine	expression	elapsed_time
1	✓	_map_output_out	t1	...ut == impl.out	00:00:01

Constraints: ALL

name	vacuity	witness	expression	type	class
1 _map_input_a			...c.a == impl.a	envconstraint	script
2 _map_input_clk			...lk == impl.clk	envconstraint	script
3 ...ap_input_reset			... == impl.reset	envconstraint	script
4 ...p_uninit_State			... == impl.State	resetconstraint	script

Total Properties: 1 - passed[1] - failed[0] - disabled[0]; Constraints Enabled: 4; Run Time: 0:00:10

VC:TaskList

Name	Progress	Result
SEQ	1:1:0:0	1:1:0:0

Runtime(HH:MM:SS)
Total:12H; Run Time:0:00:10
Goals
Found:1
Passed:1 (100.0%)
Proven:1
Failed:0 (0.0%)
Falsified:0
Checking:0
Inconclusive:0
Not Run:0
Disabled:0
Constraints
Found:4
Disabled:0

Src5:[SEQ]seq_top.spec/home/ghonim/vcfmo/5825152/design/S1.sv

```

[spec]
14 if (reset) State <= S0; else
15 // in case of a reset, go to the initial state S0
16 State <= Next;
17 // Go to the next state with each positive clock edge
18 end
19 //Next State Logic
20 always_comb begin
21 case (State) // Check the case
22 S0: if (a) Next=S1; else Next=S2;
23 S1: if (a) Next=S2; else Next=S3;
24 S2: Next=S1;
25 S3: if (a) Next=S0; else Next=S2;
26 default: begin $display(
27 "Invalid state, will reset state S0"); Next=S0; end
28 endcase
29 end
30 //Setting the state outputs
31 always_comb begin
32 case (State)
33 S0: out=2'b00;
34 S1: out=2'b01;

```

Src5:[SEQ]S2.sv

```

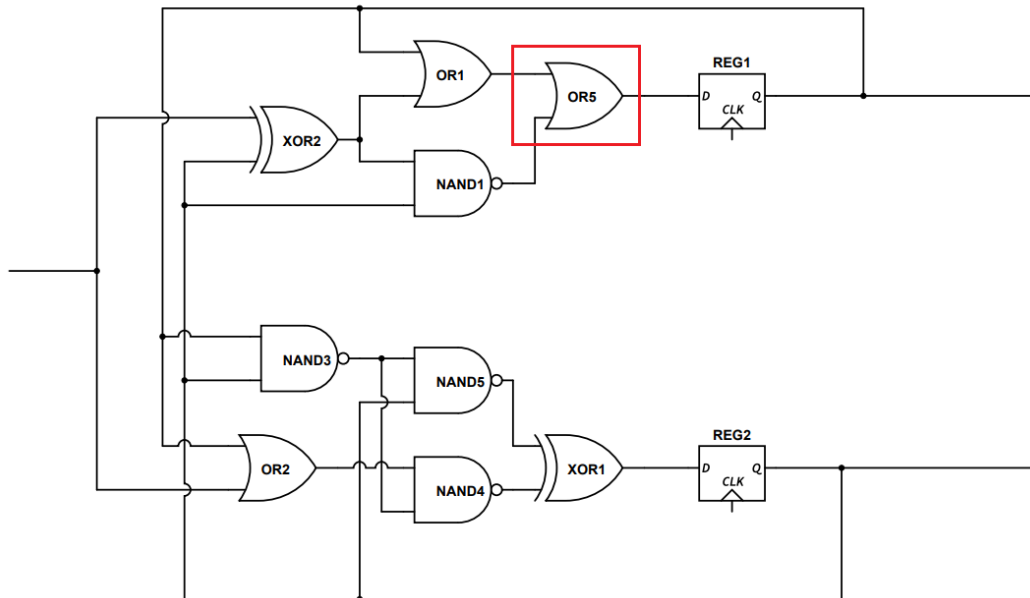
17 //Next State Logic
18 always_comb begin
19 case (State) // Check the case
20 S0: if (a) Next=S1; else Next=S2;
21 S1: if (a) Next=S2; else Next=S3;
22 S2: Next=S1;
23 S3: if (a) Next=S0; else Next=S2;
24 default: begin $display(
25 "Invalid state, will reset state S0"); Next=S0; end
26 endcase
27 end
28 //Setting the state outputs
29 always_comb begin
30 case (State)
31 S0: out=2'b00;
32 S1: out=2'b01;

```

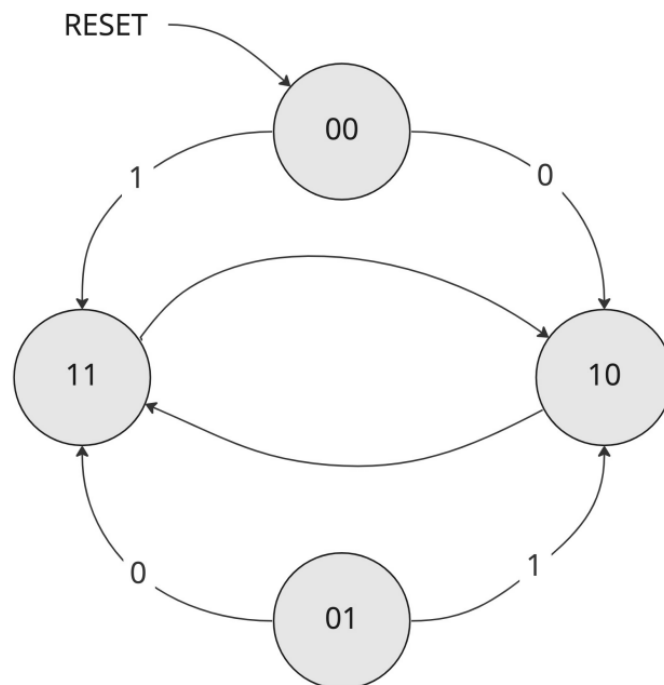
Which means that the two designs S1 and S2 are equivalent.

Task 2.2: Replace one gate of S1 with a different gate so that S1 is different from S2.
Perform the equivalence check on S1 and S2 with the SEQ App in VC Formal.

Modified S2 Circuit Drawing/Structure



FSM representing the behavior of the modified S2 above

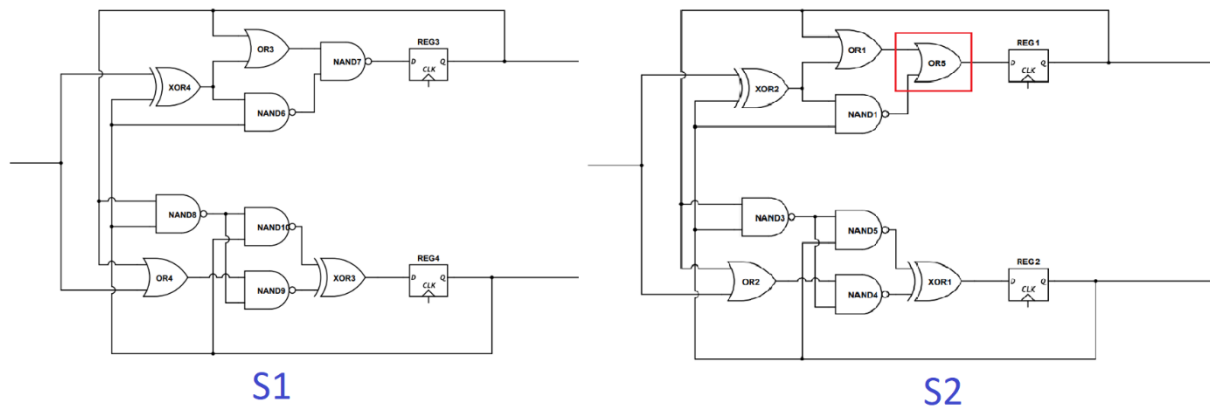


SV Code/RTL model represeting S2 (FSM behavioral model)

```
C:\intelFPGA_pro\22.3\S2.sv - Default

Ln#
1
2 // Mohamed Ghonim - Alexander Maso
3 // ECE 582/682 Verification of Hardware and Software Systems
4 // Project 2 Sequential Circuit
5 // This is the modified circuit S2
6
7 module S_design (input logic a, clk, reset, output logic [1:0] out);
8
9 // Enumerate the states. Here we are using binary encoding
10 enum logic [1:0] {S0, S1, S2, S3} State, Next;
11
12
13 //State Register
14 always_ff @(posedge clk, negedge reset) begin //Negative edge triggered asynchronous reset
15 if (reset) State <= S0; else // in case of a reset, go to the initial state S0
16 State <= Next; // Go to the next state with each positive clock edge
17 end
18
19 //Next State Logic
20 always_comb begin
21 case (State) // Check the case
22 S0: if (a) Next=S3; else Next=S2;
23 S1: if (a) Next=S2; else Next=S3;
24 S2: Next=S3;
25 S3: Next=S2;
26 default: begin $display("Invalid state, will reset state S0"); Next=S0; end
27 endcase
28 end
29
30 //Setting the state outputs
31 always_comb begin
32 case (State)
33 S0: out=2'b00;
34 S1: out=2'b01;
35 S2: out=2'b10;
36 S3: out=2'b11;
37 default: out=2'b00;
38 endcase
39 end
40
41 endmodule: S_design
```

S1 and S2 next to each other



We will then run the VC Formal SEQ app analysis on S1 and S2 (modified version), we get a status (X) which means that the two designs are inequivalent. If we hover the mouse of the SEQ app under the task list we see that this analysis/test was failed and falsified.

VCF:TaskList

Name	Progress	Result
SEQ		1:0:1:0

Runtime(HH:MM:SS)
Total:12H: Run Time:0:00:10
Goals
Found:1
Passed:0 (0.0%)
Proven:0
Failed:1 (100.0%)
Falsified:1
Checking:0
Inconclusive:0
Not Run:0
Disabled:0
Constraints
Found:4
Disabled:0

VCF:GoalList

Time 12H Max Cycle -1

Targets: ALL

status	depth	name	engine	expression	elapsed time
1	0	_map_output_out	b1	...ut == impl.out	00:00:00

Constraints: ALL

name	vacuity	witness	expression	type	class
1 _map_input_a			...c.a == impl.a	envconstraint	script
2 _map_input_clk			...lk == impl.clk	envconstraint	script
3 _ap_input_reset			... == impl.reset	envconstraint	script
4 _p_uninit_State			... == impl.State	resetconstraint	script

Total Properties: 1 - passed[0] - failed[1] - disabled[0]; Constraints Enabled: 4; Min depth: 0; Max depth: 0; Run Time: 0:00:10

If we double click on the (X) under status, we can see the waveforms, the red (S) and (I) here indicate the inequivalence on the outputs, and therefore on the whole design. The code shows the inequivalence as well.

VCF:TaskList

Name	Progress	Result
SEQ		1:0:1:0

VCF:GoalList

Src9:[SEQ]seq_top.spec(/home/ghonim/vcfmo/5825152/design/S1.sv)

src spec [impl]

```

18 //Next State Logic
19 always_comb begin
20 case (State) // Check the case
21 S0: if (a) Next=S1; else Next=S2;
22 S1: if (a) Next=S2; else Next=S3;
23 S2: Next=S1;
24 S3: if (a) Next=S0; else Next=S2;
25 default: begin $display(
"Invalid state, will reset state S0"); Next=S0; end

```

18 //Next State Logic
19 always_comb begin
20 case (State) // Check the case
21 S0: if (a) Next=S3; else Next=S2;
22 S1: if (a) Next=S2; else Next=S3;
23 S2: Next=S3;
24 S3: Next=S2;
25 default: begin \$display(
"Invalid state, will reset state S0"); Next=S0; end

File Signal View Waveform Tools Window

0 0 0 x 1ns 100 By: SEQ-Prop

Support-Signals

- out[1:0]
- clk

Clocks

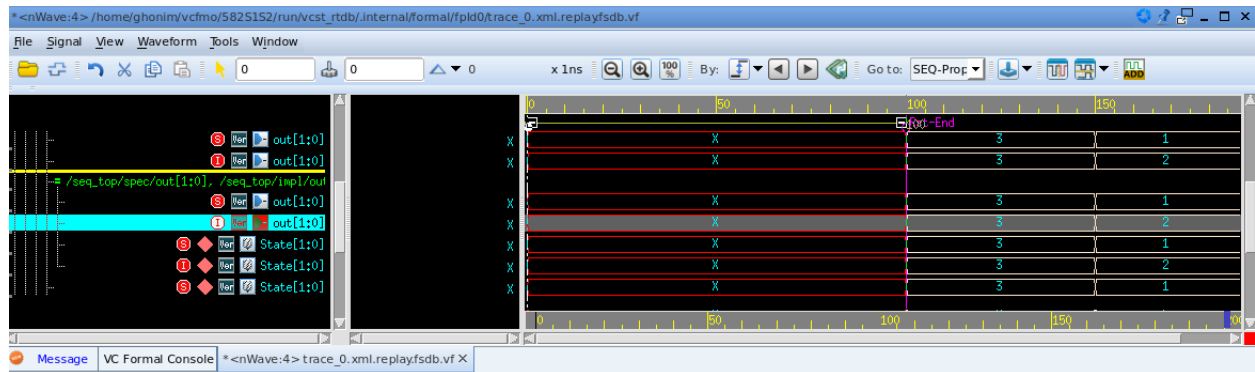
- clk

Uninitialized-Registers

We can further trace the output signal like we did before by right-clicking on the I “implementation design” and choosing “Show OneTrace Signals=> Driver”.

In this specific design, the signals will remain inequivalent regardless of how many times we trace them back, since the output was set incorrectly by the case block in the always_comb block within the first cycle. We can look at the code however and trace the

(X's) on the states and parameters to realize that the FSM machine is different and therefore the two designs are not equivalent.



The conclusion here is that we are able to very quickly tell that the two designs are inequivalent using the VC Formal SEQ app.

Appendix (SystemVerilog Codes)

C1 (and identical C2) in continuous assignment

```
// Mohamed Ghonim - Alexander Maso
// ECE 582/682 Verification of Hardware and Software Systems
// Project 2 Combinational Circuit
// This is the original circuit C1, in continuous assignment
```

```
module comb_mode (input bit a, b, c, d, output bit out);
```

```
    bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
        g11, g12, g13, g14, g15;
```

```
    assign g1 = a & b;
    assign g2 = b & c;
    assign g3 = a & c;
    assign g4 = b & c;
    assign g5 = a & c;
    assign g6 = ~(a & d);
    assign g7 = ~(b & c);
    assign g8 = ~(a & d);
```

```
    assign g9 = ~(g1 | g2);
    assign g10 = ~(g4 | g4);
    assign g11 = ~(g5 | g6);
    assign g12 = (g7 | g8);
```

```
    assign g13 = g9 & g10;
    assign g14 = g11 | g12;
```

```
assign g15 = g13 ^ g14;  
assign out = g15;
```

```
endmodule: comb_mode
```

C1 (and identical C2) in always_comb block

```
// Mohamed Ghonim - Alexander Maso  
// ECE 582/682 Verification of Hardware and Software Systems  
// Project 2 Combinational Circuit  
// This is the original circuit C1, in an always_comb block
```

```
module comb_mode (input bit a, b, c, d, output bit out);
```

```
    bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,  
        g11, g12, g13, g14, g15;
```

```
    always_comb begin
```

```
        g1 = a & b;  
        g2 = b & c;  
        g3 = a & c;  
        g4 = b & c;  
        g5 = a & c;  
        g6 = ~(a & d);  
        g7 = ~(b & c);  
        g8 = ~(a & d);
```

```

    g9 = ~(g1 | g2);
    g10 = ~(g4 | g4);
    g11 = ~(g5 | g6);
    g12 = (g7 | g8);

    g13 = g9 & g10;
    g14 = g11 | g12;

    g15 = g13 ^ g14;
    out = g15;
end
endmodule: comb_mode

```

modified C2 in continuous assignment.

```

// Mohamed Ghonim - Alexander Maso
// ECE 582/682 Verification of Hardware and Software Systems
// Project 2 Combinational Circuit
// This is the modified circuit C2, in continuous assignment

module comb_mode (input bit a, b, c, d, output bit out);

    bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
        g11, g12, g13, g14, g15;

    assign g1 = a & b;
    assign g2 = b & c;
    assign g3 = a & c;

```



```

assign g4 = b & c;
assign g5 = a & c;
assign g6 = ~(a & d);
assign g7 = ~(b & c);
assign g8 = ~(a & d);

assign g9 = ~(g1 | g2);
assign g10 = ~(g4 | g4);
assign g11 = ~(g5 | g6);
assign g12 = (g7 | g8);

assign g13 = g9 & g10;
assign g14 = g11 & g12;
// We changed this g14 gate from OR to AND
assign g15 = g13 ^ g14;
assign out = g15;

endmodule: comb_mode

```

modified C2 in always_comb block.

```

// Mohamed Ghonim - Alexander Maso
// ECE 582/682 Verification of Hardware and Software Systems
// Project 2 Combinational Circuit
// This is the modified circuit C2, in an always_comb block

module comb_mode (input bit a, b, c, d, output bit out);

```

```

bit g1, g2, g3, g4, g5, g6, g7, g8, g9, g10,
    g11, g12, g13, g14, g15;

always_comb begin
    g1 = a & b;
    g2 = b & c;
    g3 = a & c;
    g4 = b & c;
    g5 = a & c;
    g6 = ~(a & d);
    g7 = ~(b & c);
    g8 = ~(a & d);

    g9 = ~(g1 | g2);
    g10 = ~(g4 | g4);
    g11 = ~(g5 | g6);
    g12 = (g7 | g8);

    g13 = g9 & g10;
    g14 = g11 & g12;
    // We changed this g14 gate from OR to AND
    g15 = g13 ^ g14;
    out = g15;
end
endmodule: comb_mode

```

ECE 582/682 Formal Verification of Hardware and Software Systems

Project 4

Mohamed Ghonim

Alexander Maso

03/16/2023

Honor Pledge

On my honor, I have neither given nor received unauthorized aid on this project.

Mohamed Ghonim.

On my honor, I have neither given nor received unauthorized aid on this project.

Alexander Maso

A red handwritten signature in cursive script, reading "Mohamed Ghonim".A black handwritten signature in cursive script, reading "Alexander Maso".

Portland State University

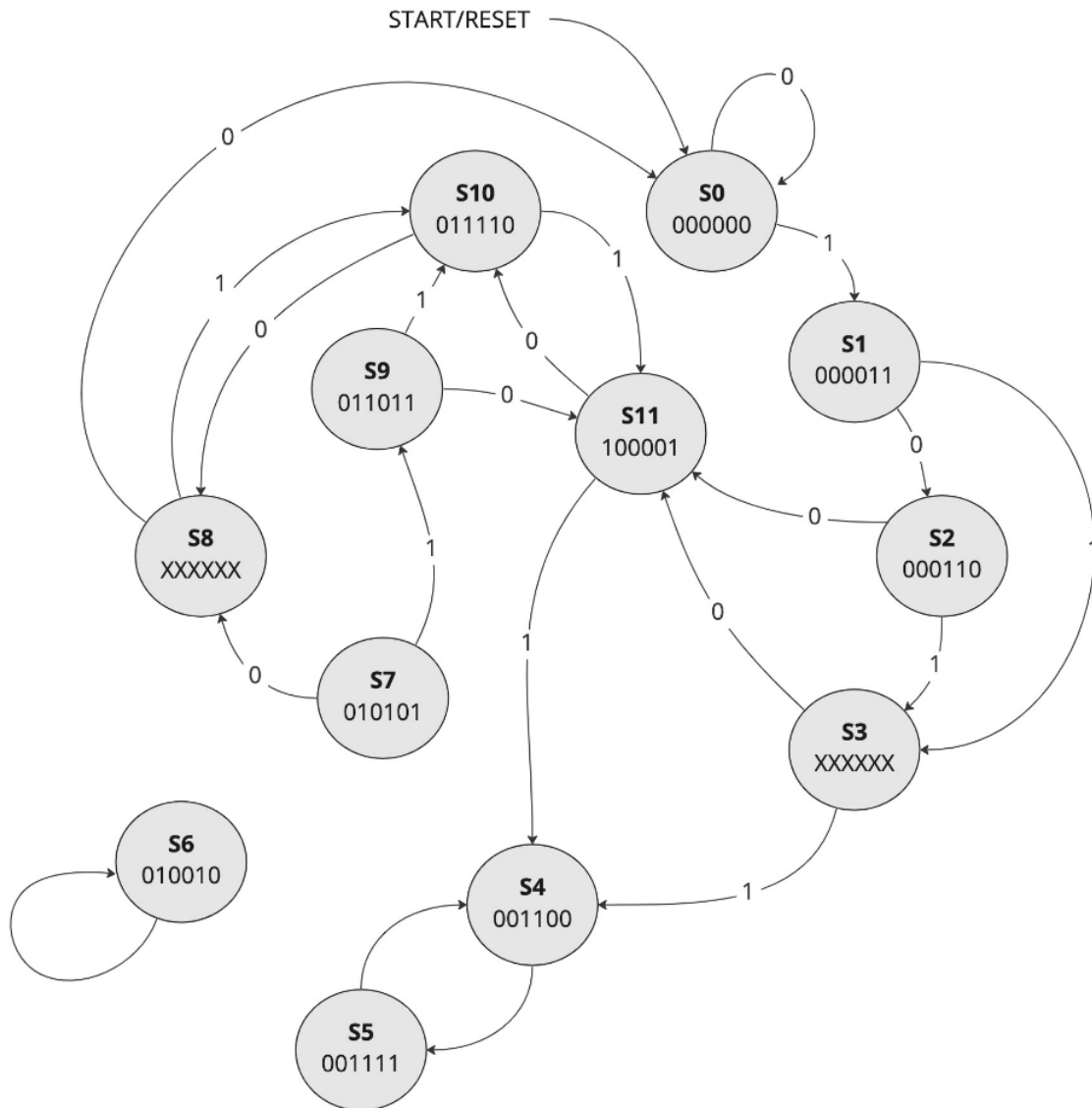
Problem 1:

Design a Finite State Machine (FSM) with at least 10 states with the following specifications:

- 1) The FSM must have at least one input.
- 2) Two of the ten states should cycle together, without any way to exit the pair, regardless of the input.
- 3) Each state must have an output. Please number the states from 0 to 10, or to the last state, and set the state output to be the state number multiplied by 3. For instance, state 5 must output 15.
- 4) Two of the states should have "x" as their outputs.
- 5) Create an additional state above the 10 states that is not connected to any other state, and any input applied to it must cycle through itself. If the FSM enters this state, there is no way out of it.
- 6) Use a negative edge triggered asynchronous reset that takes the FSM back to the starting state.

Tasks and Solution

- **Task 1.1:** Create a diagram of the FSM, indicating the state numbers, inputs, and outputs clearly.



Per the requirements, we have the output representing 3 x the state number:

State	Output	Binary Output	State	Output	Binary Output
S0	0	000000	S6	18	010010
S1	3	000011	S7	21	010101
S2	6	000110	S8	24	011000
S3	9	001001	S9	27	011011
S4	12	001100	S10	30	011110
S5	15	001111	S11	33	100001

Furthermore, we replaced the output of two states (S3 and S8) with don't cares. Xxxxxx.

S6 is a state that is not connected to other states, and gets stuck in S6 forever if somehow the system gets into it, as required. (Requirement 5).

S4 and S5 form a state pair which gets stuck together as required (requirement 2)

- **Task 1.2:** Write an SV code for the design, using two separates always_comb blocks - one for the state machines and one for the outputs. Use a case statement in one of the blocks and use "x" as the default case. For the other block, use either priority case or unique case.

Below is a picture of our design. the code is in our appendix as well:

```

// ECE 582/682 Formal Verification of Hardware and Software Systems
// Project 4 - Alexander Maso and Mohamed Ghoniim
// March 13th, 2023

module p4fsm(input logic clk, in, rst, output logic [5:0] out);

//Enumerate the states. We are using binary encoding here
enum logic[3:0] {S0=4'b0000, S1=4'b0001, S2=4'b0010, S3=4'b0011, S4=4'b0100, S5=4'b0101,
    S6=4'b0110, S7=4'b0111, S8=4'b1000, S9=4'b1001, S10=4'b1010, S11=4'b1011} state, next;

// //State Register - Sequential bBock
always_ff @ (posedge clk or negedge rst)
begin
    if (!rst) // if reset is high, go to the next state
    begin
        state <= next;
    end

    else // if reset is low, state = S0
    begin
        state <= S0;
    end
end

// Next State Logic
always_comb
begin
    case(state)
        S0: next = in ? S1 : S0;
        S1: next = in ? S3 : S2;
        S2: next = in ? S3 : S11;
        S3: next = in ? S4 : S11;
        S4: next = S5;
        S5: next = S4;
        S6: next = S6;
        S7: next = in ? S9 : S8;
        S8: next = in ? S10 : S0;
        S9: next = in ? S10 : S11;
        S10: next = in ? S11 : S8;
        S11: next = in ? S4 : S10;
    endcase
end

// Output State Logic
always_comb
begin
    unique case(state)
        S0: out = 6'b000000;
        S1: out = 6'b000011;
        S2: out = 6'b000110;
        S3: out = 6'bxxxxxx;
        S4: out = 6'b001100;
        S5: out = 6'b001111;
        S6: out = 6'b010010;
        S7: out = 6'b010101;
        S8: out = 6'bxxxxxx;
        S9: out = 6'b011011;
        S10: out = 6'b011110;
        S11: out = 6'b100001;
        default: out = 6'bxxxxxx;
    endcase
end

endmodule: p4fsm

```

VC Formal AEP app

- **Task 1.3:** Analyze the design using the VC Formal AEP app. Correct any false properties that are discovered during the AEP analysis by modifying the code or design. If a false property cannot be corrected, explain why it cannot be fixed and what is causing it.

The TCL File we configured to use VC Formal in the AEP app.

```
# ECE 582/682 Formal Verification of Hardware and Software Systems
# Project 4 - Alexander Maso and Mohamed Ghonim
# March 14th, 2023

set_fml_appmode AEP

# Run -aep all analysis on module "p4fsm" in the file /design/p4fsm.sv
set_fml_var fml_aep_unique_name true
read_file -top p4fsm -format sverilog -aep all -vcs {../design/p4fsm.sv}

# Creating clock and reset signals
create_clock clk -period 100
create_reset rst -sense low

# Running a reset simulation
sim_run -stable
sim_save_reset
```

AEP Analysis

The screenshot displays the Veri-Apex.nTraceMain:1> p4fsm p4fsm (../design/p4fsm.sv) - /home/.../verdi/constant.uddb interface. The interface includes a menu bar (File, View, Source, OneTrace, Tools, Window, Help), a toolbar, and a main workspace. The left sidebar shows the 'Task List' with a 'Name' column (AEP) and a 'Result' column (5:2:3:0). The main workspace displays the 'VC:GoalList' table, which lists analysis goals with columns for status, depth, name, engine, expression, elapsed_time, location, type, state_reg, and state_name. The table shows 5 goals, with 2 passed, 3 failed, and 0 disabled. The 'Targets: ALL' section is visible, and the 'Constraints: ALL' section shows a single constraint 'constant_5' with a vacuity of 1 and a witness of 1. The bottom status bar indicates 'Total Properties: 5 - passed[2] - failed[3] - disabled[0]; Constraints Enabled: 1; Min depth: 0; Max depth: 0; Run Time: 0:00:07'.

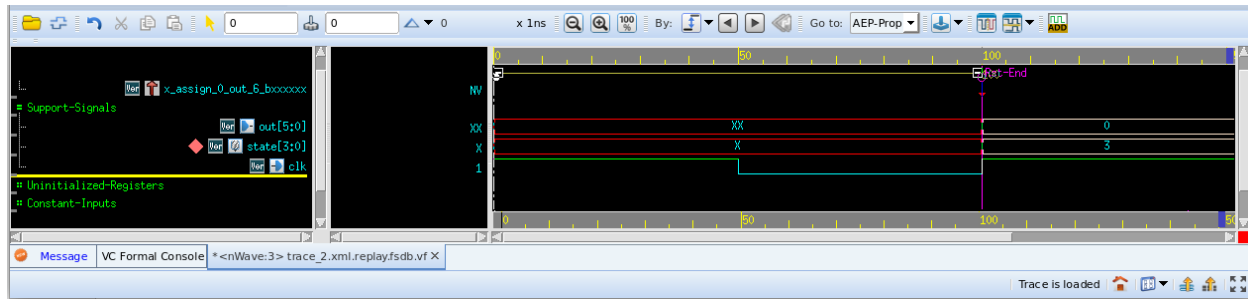
status	depth	name	engine	expression	elapsed_time	location	type	state_reg	state_name
✓		p4fsm.full_case_0_state	t1	state	00:00:00	...n/p4fsm.sv:48	full_case		
✓		p4fsm.parallel_case_0_state	t1	state	00:00:00	...n/p4fsm.sv:48	parallel_case		
✗	0	p4fsm.x_assign_0_out_6_bxxxxxx	b6	...t = 6'bxxxxxx;	00:00:01	...n/p4fsm.sv:52	x_assign		
✗	0	p4fsm.x_assign_1_out_6_bxxxxxx	b6	...t = 6'bxxxxxx;	00:00:01	...n/p4fsm.sv:57	x_assign		
✗	0	p4fsm.x_assign_2_out_6_bxxxxxx	b6	...t = 6'bxxxxxx;	00:00:01	...n/p4fsm.sv:61	x_assign		

name	vacuity	witness	expression	type	class
constant_5	1	1	rst==1	constconstraint	script

Total Properties: 5 - passed[2] - failed[3] - disabled[0]; Constraints Enabled: 1; Min depth: 0; Max depth: 0; Run Time: 0:00:07

The AEP app automatically extracted 5 properties, two of them have passed the verification, one for the full_case, and one for the parallel case. The 3 other properties are of the type x_assign, pertaining the don't cares in the output of our design.

To gain a better understanding of the falsified properties, we can double click on the "X" signs to see the waveforms and the annotated source code.



The issue here is in the output having don't cares, we have 3 falsified properties for the outputs, one for S3, one for S8, and one for the default case.

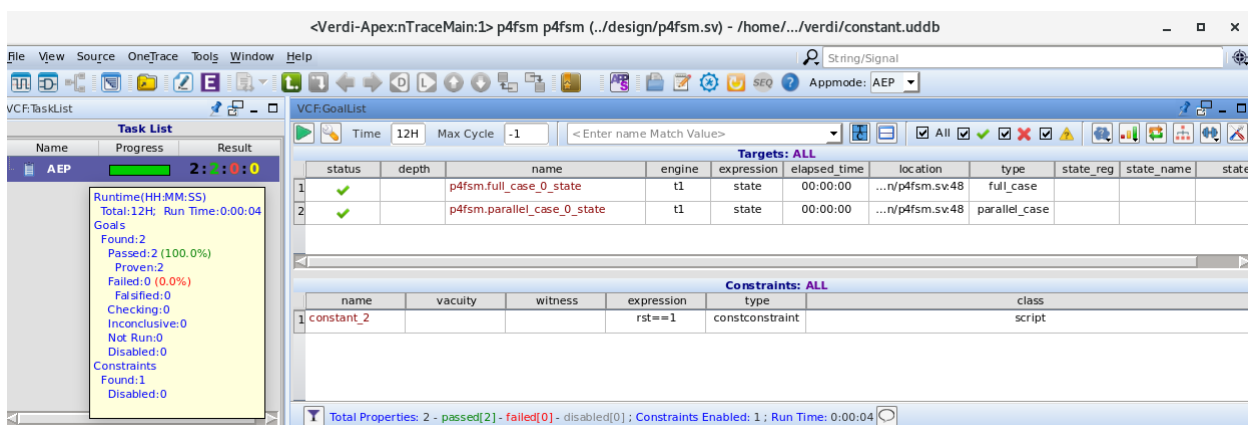
To resolve those issues, we need to remove the don't cares in our design, and replacing them with the true output. So for S3, S8 and the default case, instead of X's, now we will have:

S3 = 9 = 001001

S8 = 24 = 011000

Default = 0 = 000000

When we run the analysis again, this is the result we get:



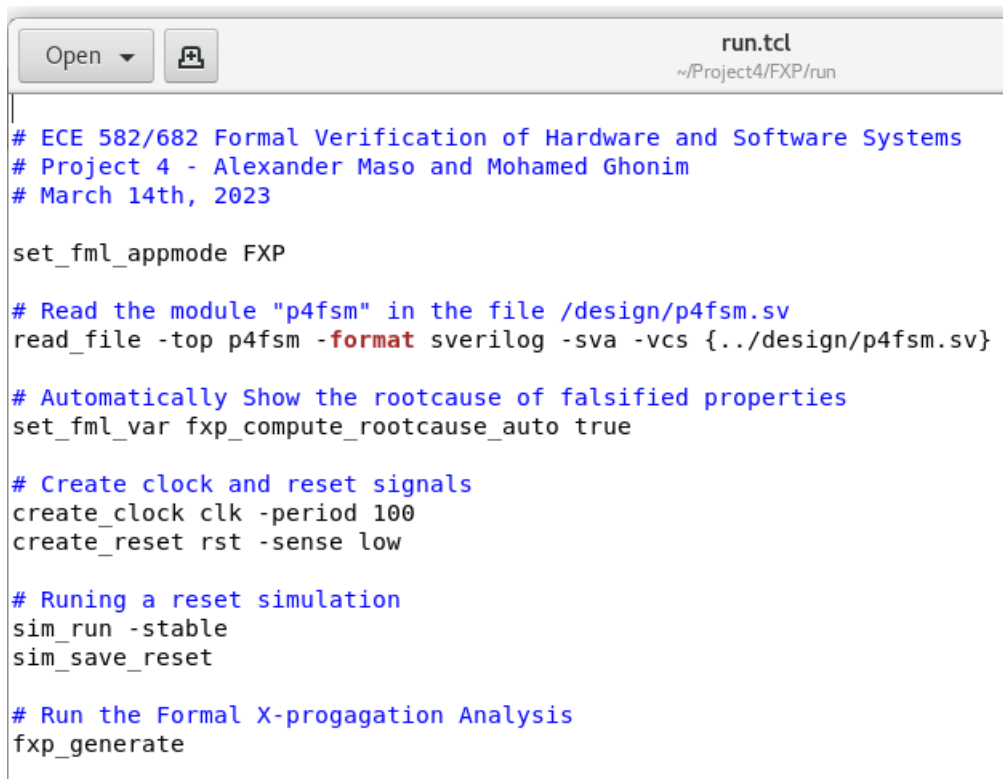
Since the AEP extracts the properties automatically, the 3 x-assign properties don't get extracted in this analysis now that the design doesn't include any x's "don't cares". We still have the full_case and parallel case properties, and they're passing in this updated analysis.

The TCL and updated SV code are attached in our appendix.

VC Formal FXP app

- **Task 1.4:** Analyze the original design (not the corrected one resulting from the previous task) in the VC Formal FXP app. Modify the code or design to fix any falsified properties that are identified during the FXP analysis. If a falsified property cannot be corrected, explain what is causing the issue.

TCL file:



```
Open [icon] run.tcl
~/Project4/FXP/run

# ECE 582/682 Formal Verification of Hardware and Software Systems
# Project 4 - Alexander Maso and Mohamed Ghonim
# March 14th, 2023

set_fml_appmode FXP

# Read the module "p4fsm" in the file /design/p4fsm.sv
read_file -top p4fsm -format sverilog -sva -vcs {../design/p4fsm.sv}

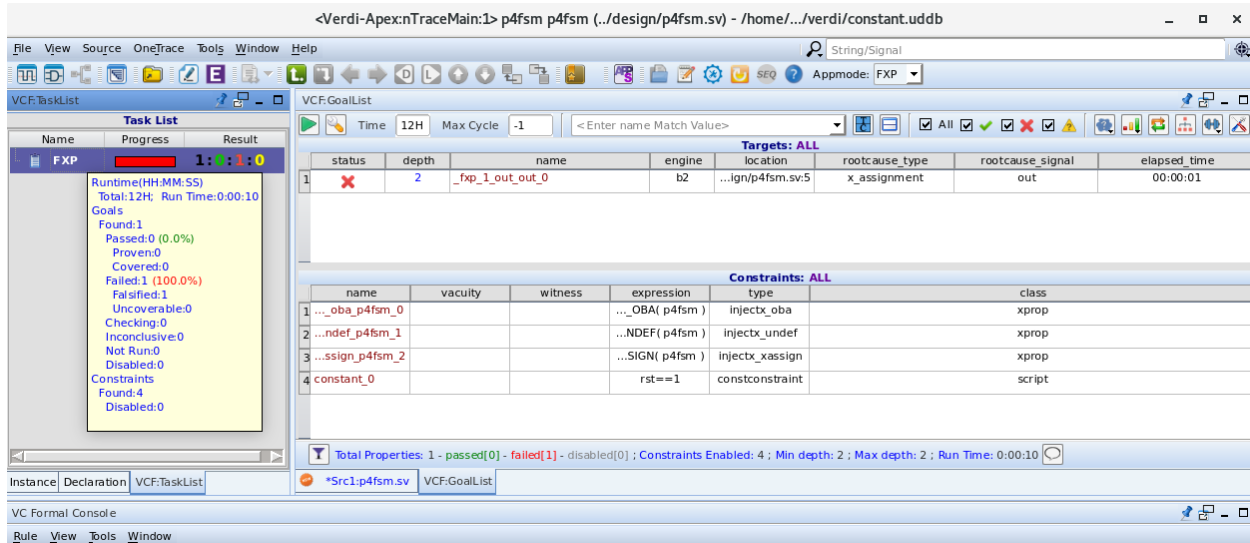
# Automatically Show the rootcause of falsified properties
set_fml_var fxp_compute_rootcause_auto true

# Create clock and reset signals
create_clock clk -period 100
create_reset rst -sense low

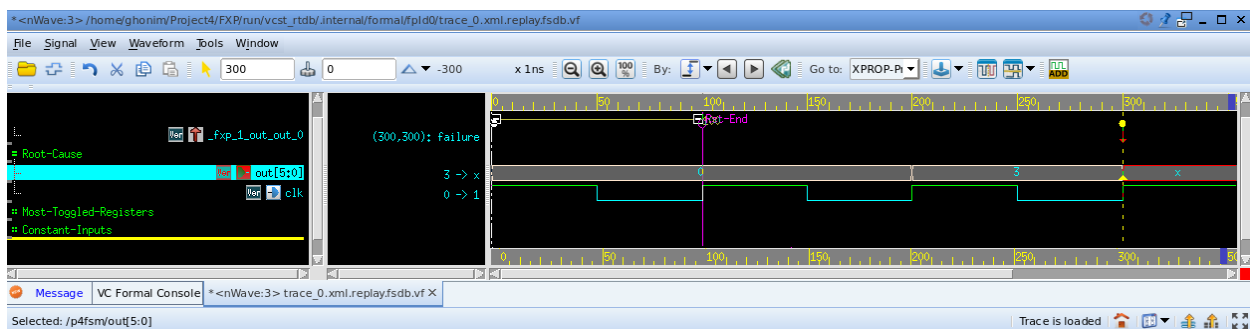
# Running a reset simulation
sim_run -stable
sim_save_reset

# Run the Formal X-propagation Analysis
fxp_generate
```

Analysis ran in the FXP app:



We see here that the FXP app injected 4 don't cares in our system "Constraints" and detected the don't care we have in our output signal as a failed property. To further investigate this (though not required since this design is relatively simple) we can click on the "X" sign under status to look at the waveforms.



In the waveform we see that there's an instance where the output signal is = x "don't care", this happens on the transition from output = 3 to output = x.

To resolve those issues, we need to remove the don't cares in our design, and replacing them with the true output. So for S3, S8 and the default case, instead of X's, now we will have:

S3 = 9 = 001001

S8 = 24 = 011000

Default = 0 = 000000

When we run the analysis again, this is the result we get:

<Verdi-Apex:nTraceMain:1> p4fsm p4fsm (../design/p4fsm.sv) - /home/.../verdi/constant.uddb

File View Source OneTrace Tools Window Help

String/Signal

VCF:TaskList

Task List

Name	Progress	Result
FXP		1:0:0

Runtime(HH:MM:SS)
Total:12H; Run Time:0:00:08

Goals

Found:1
Passed:1 (100.0%)
Proven:1
Covered:0
Failed:0 (0.0%)
Falsified:0
Uncoverable:0
Checking:0
Inconclusive:0
Not Run:0
Disabled:0
Constraints
Found:4
Disabled:0

VCF:GoalList

Time: 12H Max Cycle: -1 <Enter name Match Value>

Targets: ALL

status	depth	name	engine	location	rootcause_type	rootcause_signal	elapsed_time
1	✓	_fxp_1_out_out_0	t1	...ign/p4fsm.sv:5	not_computed		00:00:01

Constraints: ALL

name	vacuity	witness	expression	type	class
1 ..._oba_p4fsm_0			..._OBA(p4fsm)	injectx_oba	xprop
2 ...ndef_p4fsm_1			...NDEF(p4fsm)	injectx_undef	xprop
3 ...ssign_p4fsm_2			...SIGN(p4fsm)	injectx_xassign	xprop
4 constant_0			rst==1	constconstraint	script

Total Properties: 1 - passed[1] - failed[0] - disabled[0] ; Constraints Enabled: 4 ; Run Time: 0:00:09

Instance Declaration VCF:TaskList

*Src1:p4fsm.sv VCF:GoalList

We see here that using the same constraints “injections” VC Formal was not able to detect any falsified properties! The output property passed in this analysis, meaning that there’s no x “don’t care” propagation issues in our design now.

VC Formal FCA app

- **Task 1.5:** Use the VC Formal FCA app to analyze the original design and display the FCA Design hierarchy score, as well as the verified or falsified properties.

[08]

TCL File for the FCA app:

```
# ECE 582/682 Formal Verification of Hardware and Software Systems
# Project 4 - Alexander Maso and Mohamed Ghoni
# March 14th, 2023

# Select FCA (COV) as the VC Formal App mode
set_fm1_appmode COV

# Set the module name as the design parameter
set design p4fsm

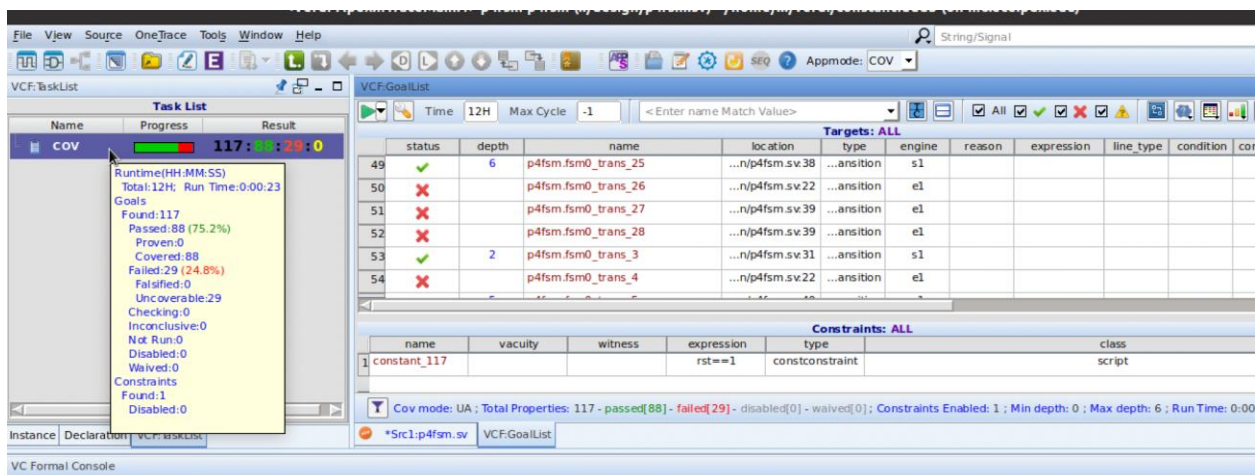
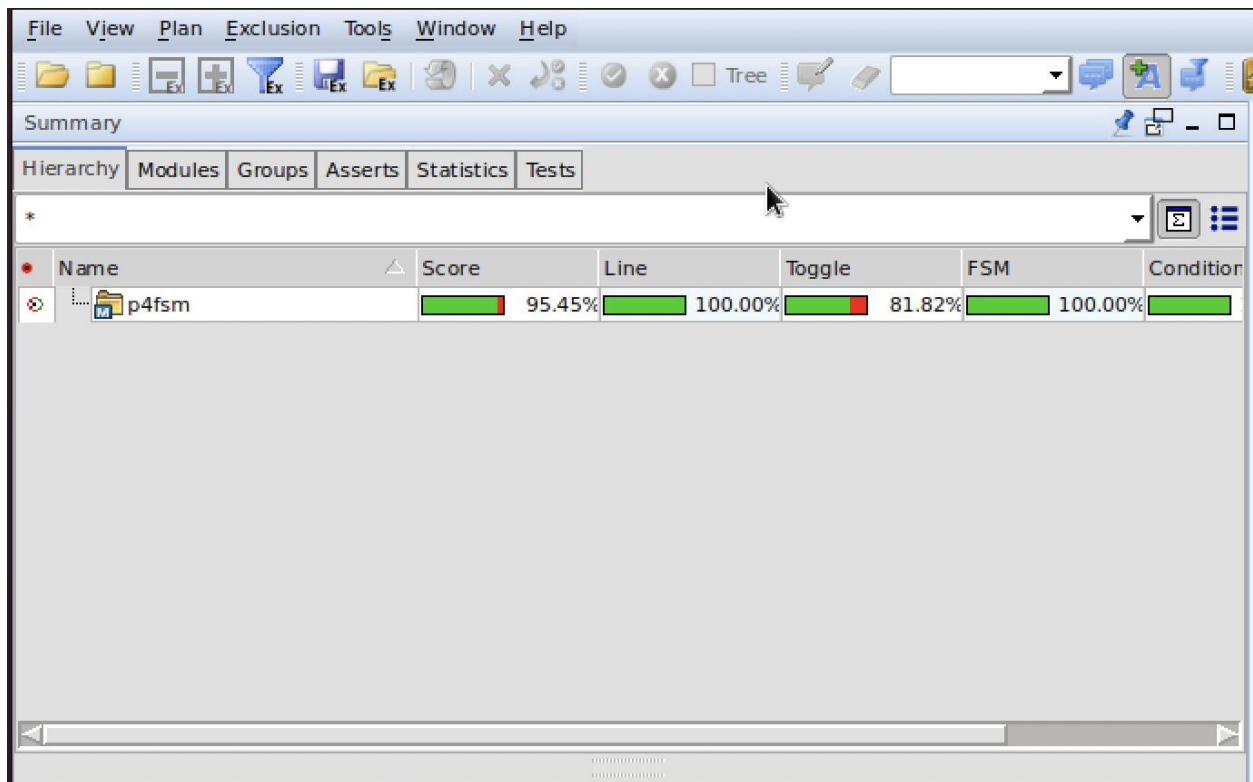
# Read the module "p4fsm" in the file /design/p4fsm.sv
# "+define+INLINE_SVA" allows us to write SVA properties and assertions within the .sv design itself
read_file -top $design -format sverilog -sva \
  -vcs {../design/p4fsm.sv +define+INLINE_SVA} -cov all

# Create clock and reset signals
# clk and rst should reflect the name of the clock and reset signals in your design
create_clock clk -period 100
create_reset rst -sense low

# Runing a reset simulation
sim_run -stable
sim_save_reset
```

Analysis ran in the FCA app:

We ran our design through VC Formal using the FCA app and received a coverage score of approximately 95%. Upon further inspection we are able to see that our model passed 88 of the 117 Goals and that we failed the remaining 29. Looking through the Goal List itself we are able to determine that many of the false tests are related to the the transition between states (as shown below)



In industry, we'd need to get a coverage score of 100% before signoff, and we do this by writing enough assertions to cover the whole design with all the corner cases.

The FCA app works by injecting bugs into the systems and traces their propagation. VC formal inserted 117 types of bugs into our system and 88 of them passed, and 29 of them affected our design and there were not enough assertions to cover those cases.

- **Task 1.6:** Use the VC Formal FPV app to analyze the original design. At least two properties or assertions in SVA must be written for this section. Identify and correct any falsified properties that are detected during the analysis. If a false property cannot be corrected, explain why it cannot be fixed and what is causing it.

Inline SVA Assertions

```
// Inline SVA Assertions
`ifndef INLINE_SVA

check_trans_S0: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S0) |-> ##1 (state==S1));

check_trans_S0_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S0) |-> ##1 (state==S0));

check_trans_S1: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S1) |-> ##1 (state==S3));

check_trans_S1_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S1) |-> ##1 (state==S2));

check_trans_S2: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S2) |-> ##1 (state==S3));

check_trans_S2_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S2) |-> ##1 (state==S11));

check_trans_S3: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S3) |-> ##1 (state==S4));

check_trans_S3_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S3) |-> ##1 (state==S11));

check_cycle_S4_S5: assert property(
    @(posedge clk) disable iff (!rst)
    (state==S4) |-> ##1 (state==S5));

check_cycle_S5_S4: assert property(
    @(posedge clk) disable iff (!rst)
    (state==S5) |-> ##1 (state==S4));

check_trans_S7: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S7) |-> ##1 (state==S9));

check_trans_S7_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S7) |-> ##1 (state==S8));

check_trans_S8: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S8) |-> ##1 (state==S10));

check_trans_S8_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S8) |-> ##1 (state==S0));

check_trans_S9: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S9) |-> ##1 (state==S10));

check_trans_S9_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S9) |-> ##1 (state==S11));

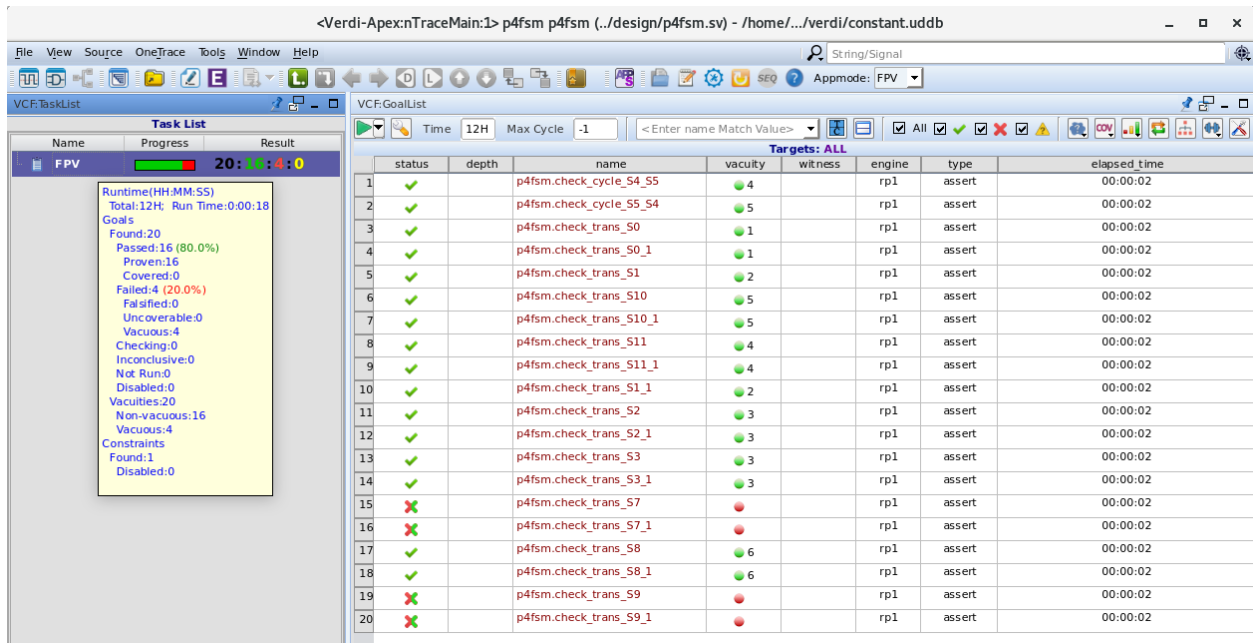
check_trans_S10: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S10) |-> ##1 (state==S11));

check_trans_S10_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S10) |-> ##1 (state==S8));

check_trans_S11: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S11) |-> ##1 (state==S4));

check_trans_S11_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S11) |-> ##1 (state==S10));

`endif
```



We wrote 20 SystemVerilog Assertion (SVA) statements, even though only 2 were required. Out of the 20 assertions, 16 passed while 4 were vacuous. Our assertions were focused on testing FSM transitions. We also wrote additional assertions regarding the output of each state, which were not included in the picture above. The vacuous results occurred on states with no input, indicating that the assertion was true, but there was no way to reach that state due to the absence of input going into it or its connected state.

Overall, the Formal Property Verification (FPV) application provided us with a better understanding of our design and helped us validate the transitions of our FSM. To convert the vacuous assertions into passed assertions, we would need to connect inputs to those states. However, this would result in a change to our design. It is important to note that the vacuous result does not indicate a falsification of the design, but rather that the assertion is trivial.

Appendix

Original RTL Design for the specified FSM:

// ECE 582/682 Formal Verification of Hardware and Software Systems

// Project 4 - Alexander Maso and Mohamed Ghonim

// March 13th, 2023

```
module p4fsm(input logic clk, in, rst, output logic [5:0] out);
```

```
//Enumerate the states. We are using binary encoding here
```

```
enum logic[3:0] {S0=4'b0000, S1=4'b0001, S2=4'b0010, S3=4'b0011, S4=4'b0100,  
S5=4'b0101, S6=4'b0110, S7=4'b0111, S8=4'b1000, S9=4'b1001, S10=4'b1010,  
S11=4'b1011} state, next;
```

```
// //State Register - Sequential bBock
```

```
always_ff @ (posedge clk or negedge rst)
```

```
begin
```

```
    if (!rst) // if reset is high, go to the next state
```

```
    begin
```

```
        state <= next;
```

```
    end
```

```
    else // if reset is low, state = S0
```

```
    begin
```

```
        state <= S0;
```

```
    end
```

```
end
```

```
// Next State Logic
```

```
always_comb
```

```
begin
```

```

case(state)
  S0:  next = in ? S1 : S0;
  S1:  next = in ? S3 : S2;
  S2:  next = in ? S3 : S11;
  S3:  next = in ? S4 : S11;
  S4:  next = S5;
  S5:  next = S4;
  S6:  next = S6;
  S7:  next = in ? S9 : S8;
  S8:  next = in ? S10 : S0;
  S9:  next = in ? S10 : S11;
  S10: next = in ? S11 : S8;
  S11: next = in ? S4 : S10;
endcase
end

```

```
// Output State Logic
```

```
always_comb
```

```
begin
```

```

  unique case(state)
    S0:  out = 6'b000000;
    S1:  out = 6'b000011;
    S2:  out = 6'b000110;
    S3:  out = 6'bxxxxxx;
    S4:  out = 6'b001100;
    S5:  out = 6'b001111;
    S6:  out = 6'b010010;

```

```

S7:  out = 6'b010101;
S8:  out = 6'bxxxxxx;
S9:  out = 6'b011011;
S10: out = 6'b011110;
S11: out = 6'b100001;
default: out = 6'bxxxxxx;
endcase
end
endmodule: p4fsm

```

AEP App

TCL File

```

# ECE 582/682 Formal Verification of Hardware and Software Systems
# Project 4 - Alexander Maso and Mohamed Ghonim
# March 14th, 2023

```

```
set_fml_appmode AEP
```

```

# Run -aep all analysis on module "p4fsm" in the file /design/p4fsm.sv
set_fml_var fml_aep_unique_name true
read_file -top p4fsm -format sverilog -aep all -vcs {../design/p4fsm.sv}

```

```

# Creating clock and reset signals
create_clock clk -period 100
create_reset rst -sense low

```

```
# Running a reset simulation
```

```
sim_run -stable
```

```
sim_save_reset
```

Fixed RTL Design for the AEP app:

```
module p4fsm(input logic clk, in, rst, output logic [5:0] out);
```

```
//Enumerate the states. We are using binary encoding here
```

```
enum logic[3:0] {S0=4'b0000, S1=4'b0001, S2=4'b0010, S3=4'b0011, S4=4'b0100,  
S5=4'b0101, S6=4'b0110, S7=4'b0111, S8=4'b1000, S9=4'b1001, S10=4'b1010,  
S11=4'b1011} state, next;
```

```
// //State Register - Sequential bBock
```

```
always_ff @ (posedge clk or negedge rst)
```

```
begin
```

```
    if (!rst) // if reset is high, go to the next state
```

```
    begin
```

```
        state <= next;
```

```
    end
```

```
    else // if reset is low, state = S0
```

```
    begin
```

```
        state <= S0;
```

```
    end
```

```
end
```

```
// Next State Logic
```

```
always_comb
```

```

begin
  case(state)
    S0:  next = in ? S1 : S0;
    S1:  next = in ? S3 : S2;
    S2:  next = in ? S3 : S11;
    S3:  next = in ? S4 : S11;
    S4:  next = S5;
    S5:  next = S4;
    S6:  next = S6;
    S7:  next = in ? S9 : S8;
    S8:  next = in ? S10 : S0;
    S9:  next = in ? S10 : S11;
    S10: next = in ? S11 : S8;
    S11: next = in ? S4 : S10;
  endcase
end

```

```

// Output State Logic

```

```

always_comb

```

```

begin
  unique case(state)
    S0:  out = 6'b000000;
    S1:  out = 6'b000011;
    S2:  out = 6'b000110;
    S3:  out = 6'b001001;
    S4:  out = 6'b001100;
    S5:  out = 6'b001111;

```

```
S6:  out = 6'b010010;
S7:  out = 6'b010101;
S8:  out = 6'b011000;
S9:  out = 6'b011011;
S10: out = 6'b011110;
S11: out = 6'b100001;
default: out = 6'b000000;
endcase
end
endmodule: p4fsm
```

FXP App

TCL File

```
# ECE 582/682 Formal Verification of Hardware and Software Systems
```

```
# Project 4 - Alexander Maso and Mohamed Ghonim
```

```
# March 14th, 2023
```

```
set_fml_appmode FXP
```

```
# Read the module "p4fsm" in the file /design/p4fsm.sv
```

```
read_file -top p4fsm -format sverilog -sva -vcs {../design/p4fsm.sv}
```

```
# Automatically Show the rootcause of falsified properties
```

```
set_fml_var fxp_compute_rootcause_auto true
```

```
# Create clock and reset signals
```

```
create_clock clk -period 100
```

```
create_reset rst -sense low
```

```
# Running a reset simulation
```

```
sim_run -stable
```

```
sim_save_reset
```

```
# Run the Formal X-propagation Analysis
```

```
fxp_generate
```

Fixed RTL Design for the FXP app

```
// ECE 582/682 Formal Verification of Hardware and Software Systems
```

```
// Project 4 - Alexander Maso and Mohamed Ghonim
```

```
// March 13th, 2023
```

```
module p4fsm(input logic clk, in, rst, output logic [5:0] out);
```

```
//Enumerate the states. We are using binary encoding here
```

```
enum logic[3:0] {S0=4'b0000, S1=4'b0001, S2=4'b0010, S3=4'b0011, S4=4'b0100,  
S5=4'b0101,
```

```
    S6=4'b0110, S7=4'b0111, S8=4'b1000, S9=4'b1001, S10=4'b1010, S11=4'b1011} state,  
next;
```

```
//State Register - Sequential block
```

```
always_ff @ (posedge clk or negedge rst)
```

```
begin
```

```
    if (rst) // if reset is high, go to the next state
```

```

begin
state <= next;
end
else          // if reset is low, state = S0
begin
state <= S0;
end
end

```

```

// Next State Logic

```

```

always_comb

```

```

begin

```

```

    next = S0;

```

```

case(state)

```

```

    S0:  next = in ? S1 : S0;

```

```

    S1:  next = in ? S3 : S2;

```

```

    S2:  next = in ? S3 : S11;

```

```

    S3:  next = in ? S4 : S11;

```

```

    S4:  next = S5;

```

```

    S5:  next = S4;

```

```

    S6:  next = S6;

```

```

    S7:  next = in ? S9 : S8;

```

```

    S8:  next = in ? S10 : S0;

```

```

    S9:  next = in ? S10 : S11;

```

```

    S10: next = in ? S11 : S8;

```

```

    S11: next = in ? S4 : S10;

```



```

        endcase
    end

    // Output State Logic
    always_comb
    begin

        out = 6'b000000;

        unique case(state)
            S0:  out = 6'b000000;
            S1:  out = 6'b000011;
            S2:  out = 6'b000110;
            S3:  out = 6'b001001;
            S4:  out = 6'b001100;
            S5:  out = 6'b001111;
            S6:  out = 6'b010010;
            S7:  out = 6'b010101;
            S8:  out = 6'b011000;
            S9:  out = 6'b011011;
            S10: out = 6'b011110;
            S11: out = 6'b100001;
            default: out = 6'b000000;
        endcase
    end

endmodule: p4fsm

```

FCA App.

TCL file

ECE 582/682 Formal Verification of Hardware and Software Systems

Project 4 - Alexander Maso and Mohamed Ghonim

March 14th, 2023

Select FCA (COV) as the VC Formal App mode

set_fml_appmode COV

Set the module name as the design parameter

set design p4fsm

Read the module "p4fsm" in the file /design/p4fsm.sv

"+define+INLINE_SVA" allows us to write SVA properties and assertions within the .sv design itself

read_file -top \$design -format sverilog -sva \

-vcs {../design/p4fsm.sv +define+INLINE_SVA} -cov all

Create clock and reset signals

clk and rst should reflect the name of the clock and reset signals in your design

create_clock clk -period 100

create_reset rst -sense low

Running a reset simulation

sim_run -stable

sim_save_reset

FPV App

TCL file:

ECE 582/682 Formal Verification of Hardware and Software Systems

Project 4 - Alexander Maso and Mohamed Ghonim

March 14th, 2023

Select FPV as the VC Formal App mode

set_fml_appmode FPV

Set the module name as the design parameter

set design p4fsm

Read the module "S_design" in the file /design/fpv_example.sv

"+define+INLINE_SVA" allows us to write SVA properties and assertions within the .sv design itself

read_file -top \$design -format sverilog -sva \
-vcs {../design/p4fsm.sv +define+INLINE_SVA}

Create clock and reset signals

clk and rst should reflect the name of the clock and reset signals in your design

```
create_clock clk -period 100
```

```
create_reset rst -sense low
```

```
# Running a reset simulation
```

```
sim_run -stable
```

```
sim_save_reset
```

Code with assertion:

```
// ECE 582/682 Formal Verification of Hardware and Software Systems
```

```
// Project 4 - Alexander Maso and Mohamed Ghonim
```

```
// March 13th, 2023
```

```
module p4fsm(input logic clk, in, rst, output logic [5:0] out);
```

```
//Enumerate the states. We are using binary encoding here
```

```
enum logic[3:0] {S0=4'b0000, S1=4'b0001, S2=4'b0010, S3=4'b0011, S4=4'b0100,  
S5=4'b0101, S6=4'b0110, S7=4'b0111, S8=4'b1000, S9=4'b1001, S10=4'b1010,  
S11=4'b1011} state, next;
```

```
//State Register - Sequential block
```

```
always_ff @ (posedge clk or negedge rst)
```

```
begin
```

```
    if (rst) // if reset is high, go to the next state
```

```
    begin
```

```
        state <= next;
```

```
    end
```

```
    else // if reset is low, state = S0
```

```
    begin
```

```
    state <= S0;
end
end
```

```
// Next State Logic
```

```
always_comb
```

```
begin
```

```
    next = S0;
```

```
    case(state)
```

```
        S0: next = in ? S1 : S0;
```

```
        S1: next = in ? S3 : S2;
```

```
        S2: next = in ? S3 : S11;
```

```
        S3: next = in ? S4 : S11;
```

```
        S4: next = S5;
```

```
        S5: next = S4;
```

```
        S6: next = S6;
```

```
        S7: next = in ? S9 : S8;
```

```
        S8: next = in ? S10 : S0;
```

```
        S9: next = in ? S10 : S11;
```

```
        S10: next = in ? S11 : S8;
```

```
        S11: next = in ? S4 : S10;
```

```
    endcase
```

```
end
```

```
// Output State Logic
```

```
always_comb
```

```

begin
  unique case(state)
    S0:  out = 6'b000000;
    S1:  out = 6'b000011;
    S2:  out = 6'b000110;
    S3:  out = 6'b000110;
    S4:  out = 6'b001100;
    S5:  out = 6'b001111;
    S6:  out = 6'b010010;
    S7:  out = 6'b010101;
    S8:  out = 6'b010101;
    S9:  out = 6'b011011;
    S10: out = 6'b011110;
    S11: out = 6'b100001;
    default: out = 6'b100001;
  endcase
end

// Inline SVA Assertions
`ifdef INLINE_SVA

check_trans_S0: assert property(
  @(posedge clk) disable iff (!rst)
  (in && state==S0) |-> ##1 (state==S1));

check_trans_S0_1: assert property(
  @(posedge clk) disable iff (!rst)

```

```
(!(in) && state==S0) |-> ##1 (state==S0));
```

```
check_trans_S1: assert property(  
    @(posedge clk) disable iff (!rst)  
    (in && state==S1) |-> ##1 (state==S3));
```

```
check_trans_S1_1: assert property(  
    @(posedge clk) disable iff (!rst)  
    (!(in) && state==S1) |-> ##1 (state==S2));
```

```
check_trans_S2: assert property(  
    @(posedge clk) disable iff (!rst)  
    (in && state==S2) |-> ##1 (state==S3));
```

```
check_trans_S2_1: assert property(  
    @(posedge clk) disable iff (!rst)  
    (!(in) && state==S2) |-> ##1 (state==S11));
```

```
check_trans_S3: assert property(  
    @(posedge clk) disable iff (!rst)  
    (in && state==S3) |-> ##1 (state==S4));
```

```
check_trans_S3_1: assert property(  
    @(posedge clk) disable iff (!rst)  
    (!(in) && state==S3) |-> ##1 (state==S11));
```

```
check_cycle_S4_S5: assert property(  
    @(posedge clk) disable iff (!rst)  
    (in && state==S4) |-> ##1 (state==S5));
```

```
@(posedge clk) disable iff (!rst)
(state==S4) |-> ##1 (state==S5));
```

```
check_cycle_S5_S4: assert property(
    @(posedge clk) disable iff (!rst)
    (state==S5) |-> ##1 (state==S4));
```

```
check_trans_S7: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S7) |-> ##1 (state==S9));
```

```
check_trans_S7_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S7) |-> ##1 (state==S8));
```

```
check_trans_S8: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S8) |-> ##1 (state==S10));
```

```
check_trans_S8_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S8) |-> ##1 (state==S0));
```

```
check_trans_S9: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S9) |-> ##1 (state==S10));
```



```

check_trans_S9_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S9) |-> ##1 (state==S11));

check_trans_S10: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S10) |-> ##1 (state==S11));

check_trans_S10_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S10) |-> ##1 (state==S8));

check_trans_S11: assert property(
    @(posedge clk) disable iff (!rst)
    (in && state==S11) |-> ##1 (state==S4));

check_trans_S11_1: assert property(
    @(posedge clk) disable iff (!rst)
    (!(in) && state==S11) |-> ##1 (state==S10));
`endif
endmodule: p4fsm

```

Appendix 4: Weekly Progress Reports

Synopsys VC Formal Project

Team 7 Weekly Progress Reports 1
Winter 2023

Mohamed Ghonim
Alex Kim
Dmytro Prystupa
Ahliah Nordstrom
Celina Wong

Start Date: 2023-01-09
End Date: 2023-03-19

❖ Team Review

During the initial week, we will establish our SolvNetPlus accounts, verify the accessibility of VC Formal, and peruse the application's documentation. Additionally, the team had a meeting with the advisor to discuss the project's objectives and expectations.

- Mohamed Ghonim (the 3 weeks before the term began + this week).
 - Installed VC Formal on my own computer using a linux virtual machine [~ 20]
 - Begin this two weeks before the term began
 - Added VC formal on the PSU lab machine remotely [2]
 - Read through the VC Formal usermanual [3]
 - Read the “finding your way through formal verification book” [5]
 - Setup the VCFormal.com webpage. [2]
 - Worked on the documentation of how to install and setup the VPN, Linux Client and VCFormal. [4]
 - Make a video on the documentation of how to install and set up the VPN, Linux Client and VCFormal. [2]
- Alex Kim
 - Reviewed setup tutorials for VC Formal [1.5]
 - Installed/accessed VC Formal using PSU Server remotely through MobaXterm [2]
 - Reviewed VC Formal User guide & VC Formal quick reference guide [2]
- Dmytro Prystupa
 - Installed/accessed VC Formal using PSU Server remotely through MobaXterm [3]
 - Reviewed VC Formal User guide [1]
- Ahliah Nordstrom
 - Successfully added/accessed VC Formal remotely [3]
 - Had trouble at first with VNC connection
 - Made team availability chart to make scheduling for meetings throughout the term easier [.5]
 - Reviewed VC Formal User guide [1]
- Celina Wong
 - Installed/accessed VC Formal using PSU Server remotely through MobaXterm [2]
 - Reviewed VC Formal User Guide [1]

Next Week

❖ Team Plan

The team should begin working on the rough draft of the project proposal and continue to familiarize themselves with VC Formal. We want to be as comfortable as possible working in the VC Formal environment before working with any of the apps; exploring shortcuts, toolbars, etc.

END OF WEEK 1 REPORT

❖ Team Review

At the start of the week, the team held a meeting to begin working on the rough draft proposal in Google Docs. This task was relatively easy as Dr. Song served as both our faculty advisor and industry sponsor, minimizing any potential for miscommunication or disagreements between us and external parties. Additionally, our proficiency in using VC Formal rapidly improved, prompting us to initiate work on our first application, SEQ.

- Mohamed Ghonim
 - I have thoroughly reviewed the SEQ application in the VC Formal User Guide. [2]
 - Revised the necessary TCL files for executing VC Formal on the SEQ app and loading the appropriate designs.
 - Executed the SEQ application on multiple designs.
 - Initiated the process of creating a tutorial draft for the utilization of the SEQ app.[4]
- Alex Kim
 - Started on rough draft proposal [2]
 - VC Formal setup verification and review [1]
 - VC Formal research to get an idea of what to add/do for our project/proposal (user guides, and VC Formal website resources) [2]
- Dmytro Prystupa
 - Successfully added/accessed VC Formal on a macOS system [3]
 - Make a video of how to install and set up the VPN, Linux Client and run VCFormal on a macOS system exploration [5]
 - VC Formal Application Research [1]
- Ahliah Nordstrom
 - Set up Trello site and added due dates to upcoming assignments and meetings [1]
 - Started on rough draft proposal [1]
 - VC Formal walkthrough/exploring program [2]
- Celina Wong
 - Overviewed VC Formal Applications on Synopsis website to learn more about their purpose [1.5]

Next Week

❖ Team Plan

Our main priority is to finalize the rough draft project proposal. In addition, we plan to schedule a team meeting to review and discuss the SEQ tutorial to ensure that everyone is familiar with its mechanics.

END OF WEEK 2 REPORT

❖ Team Review

We completed the rough draft proposal and submitted it for review on Canvas. During a team meeting, we reviewed the SEQ application and spent dedicated time going through the first draft of its tutorial.

● Mohamed Ghonim

- Finalized the SEQ tutorial
- Created a Youtube video on the SEQ app [3]
- Started reading about the AEP app on the VC Formal user guide and ran it on a few designs
- Took an introductory Synopsys course on SVA (SystemVerilog Assertions) for model checking [4]

● Alex Kim

- Reviewed SEQ tutorial first draft document [2]
- Tested/followed SEQ tutorial in VC Formal [1.5]
- Finalize draft proposal [2]

● Dmytro Prystupa

- Reviewed the SEQ tutorial and documentation and accessed VC Formal SEQ Application. [1.5]
- Followed SEQ tutorial and documentation to access SEQ application and verified that the application worked [2]
- Draft Proposal Review [1]

● Ahliah Nordstrom

- Followed SEQ tutorial steps in VC Formal, making sure to be mindful of new users [2]
- Reviewed SEQ tutorial first draft document, providing constructive criticism for final draft [1.5]
- Made project timeline using GanttProject [1.5]
- Reviewed Draft Proposal [.5]

● Celina Wong

- Followed SEQ steps in VC Formal to make sure it's easy for first time users to understand [2]
- Reviewed SEQ tutorial documentation [2]
- Crafted a tutorial template for documentation from here on out [4]

Next Week

❖ Team Plan

We are prioritizing the upcoming public VC Formal Workshops hosted by Synopsis next week. These workshops will give us the opportunity to ask questions about VC Formal in general as well as the SEQ app. We believe it is important to take advantage of this opportunity to gain more knowledge and insight about VC Formal.

END OF WEEK 3 REPORT

❖ Team Review

This week, Synopsis started their public VC Formal Workshops, which had a somewhat challenging schedule from 9am to 3pm. Therefore, we decided to make team attendance optional for the entire duration of the workshop. Moving forward, we plan to focus on exploring the next app for creating a tutorial, which could be either AEP or FXP.

- **Mohamed Ghonim**
 - Attended VC Formal workshop on DPV [4]
 - Worked on the rough draft for the AEP tutorial [2]
 - Started experimenting with the FXP app and continued debugging the AEP app.
 - “Tailor designs” a project for Dr. Song utilizing the SEQ app for his ECE 682/582 class. [2]
- **Alex Kim**
 - I participated in a workshop on DPV in VC Formal and completed the lab exercises. [4]
 - Read the VC Formal user guide for the AEP app and learn about its use cases and benefits to become more familiar with the tool. [1]
 - Began drafting appropriate tutorial steps for AEP app [1]
 - Invoking GUI, TCL file
- **Dmytro Prystupa**
 - Synopsys VC Formal Workshop on DPV [3]
 - AEP Tutorial Steps Verification, making sure initial steps were easy to follow [1]
- **Ahliah Nordstrom**
 - Followed AEP first draft tutorial steps in VC Formal [1]
 - Updated AEP tutorial documentation to reflect new figures and steps with source tracing, ready for final revisions [3]
 - Attended Synopsis VC Formal workshop on DPV [2]
- **Celina Wong**
 - Attended Synopsis VC Formal workshop DPV [4]
 - Updated Trello [.5]

Next Week

❖ Team Plan

We plan to finalize the AEP app tutorial and start working on the first draft of the FXP app tutorial. There is an optional Synopsys VC Formal workshop scheduled for next Wednesday, which will be a continuation of the DPV app lab from this week. Additionally, we are anticipating receiving feedback on our draft proposal early next week.

END OF WEEK 4 REPORT

❖ Team Review

Two applications have been implemented by our team: SEQ and AEP. AEP has been completed and is ready to be submitted to Dr. Song for his 582/682 students. We received feedback from the students regarding the SEQ tutorial, and we plan to revise it accordingly. In addition to this, we have allocated time to work on our final proposal.

● Mohamed Ghonim

- I participated in a workshop on DPV in VC Formal and completed the lab exercises. [4]
- Attempting to fix issues in the FXP application through debugging. [3]
- Began creating a video tutorial for the AEP and FXP applications. [1]
- Consulted with three Synopsys engineers regarding the feasibility of utilizing CTL in VC Formal, but a definitive answer on whether or not it's achievable has not been provided yet. [2]

● Alex Kim

- VC Formal workshop on DPV [4]
- Adapt rough draft project proposal to new feedback given from lecture [1]
- Begin work on final draft of project proposal[1]
- Helped finalize/edit AEP tutorial documentation [2]
 - Provided comments and formatting help

● Dmytro Prystupa

- VC Formal workshop on DPV [4]
- Project Proposal work, implementing feedback from lecture [2]
- AEP Tutorial edits, comments, and revisions [2]

● Ahliah Nordstrom

- Completed first draft of FXP tutorial steps in VC Formal and necessary documentation [5]
- Debugging FXP design file errors and warnings [2]
- Created FXP design and TCL file [2]

● Celina Wong

- Finalized AEP tutorial documentation [4]
 - Revisions, comments, formatting
- Created an updated version of document template/added more details based on user feedback[3]

Next Week

❖ Team Plan

We are shifting our focus to the FPV app, as Dr. Song has expressed a keen interest in using it for his class this term. We anticipate receiving feedback on our draft proposal early next week, as the submission was postponed this week. Once we receive feedback, we will make the necessary revisions to the final proposal and submit it to Dr. Song. We expect progress to slow down next week due to midterms.

END OF WEEK 5 REPORT

❖ Team Review

Progress was slower this week due to midterms, as expected. The team agreed to shift their focus to other priorities but still met with Dr. Song to finalize the project proposal. We incorporated feedback from our first draft and made appropriate adjustments. Mohamed, who is also taking the formal verification class with Dr. Song, was able to contribute to the project while studying for midterms.

● Mohamed Ghonim

- I had a discussion with engineers from Synopsys regarding the implementation of "CTL-style" verification in VC Formal, as this feature is not directly supported by the tool. [1]
- Started a project on VC Formal's FPV app, focusing on a very simple design with only two SVA assertions. [2]
- Started exploring LTL as a potential alternative to CTL for model checking in VC Formal. This involved researching the differences between LTL and CTL, as well as studying how to write LTL assertions in SVA. While this topic is time-consuming and very design-specific, we believe it has the potential to improve our formal verification processes in the long term. We plan to continue our exploration of LTL in the coming weeks. [4]

● Alex Kim

- Editing the rough draft project proposal based off feedback given by Andrew [1.5]
- Began researching VC Formal CC application [1]

● Dmytro Prystupa

- Rough draft proposal feedback editing [1]

● Ahliah Nordstrom

- Did not spend any time in VC Formal this week.
- Updated trello [.5]
- Read rough draft project proposal feedback [.5]
- Updated project timeline based on feedback given from first draft [1]

● Celina Wong

- Did not spend any time on VC Formal this week
- Added table of contents to project proposal and assisted in final revision [2]

Next Week

❖ Team Plan

Our team has made the decision to assign specific apps to each member for development. Mohamed will be working on the FPV app, Alex and Dmytro will be working on the CC app, and Ahliah and Celina will be focusing on the FCA app. This is a departure from our previous approach of passing one app around the team at a time. We believe that this new approach will be more efficient, as we are now more experienced with VC Formal and will require less time to become familiar with each app. Although we will still review each other's work, we anticipate that this process will be expedited.

END OF WEEK 6 REPORT

❖ Team Review

The team's plan to divide the tutorial work was effective and productive this week. We made progress on the FPV, CC, and FCA apps, with the FCA tutorial now complete. However, the CC app in VC Formal presented many challenges and required significant debugging, while the FPV app proved to be more difficult due to its "High Value" classification compared to the "Productivity" apps.

● Mohamed Ghonim

- Extracted the relevant chapters from the VC Formal User Guide for the FCA, CC, FPV, and FRV apps. [0.5]
- Edited and tested the TCL file for the FCA app [0.5]
- Edited and tested the TCL file for the CC app [0.5]
- Conducted research and studied the process of writing Linear Temporal Logic (LTL) assertions in SystemVerilog Assertions (SVA) to replicate Computational Tree Logic (CTL) behavior. Created simple but not particularly meaningful assertions as part of this effort. This topic is both time-consuming and highly design-specific, requiring significant effort to master. [5]

● Alex Kim

- CC Application Documentation Research- includes reviewing the VC Formal quick reference guide & user guide. Also, reviewed a Synopsys provided lab on CC app [4]
- Began CC Application Tutorial [2]
 - TCL file

● Dmytro Prystupa

- CC Application Documentation Research [3]
- Began CC Application Tutorial [1.5]
 - TCL file

● Ahliah Nordstrom

- Updated Trello [.25]
- Researching on Synopsis site and reading FCA app chapters [1]
- Familiarizing myself with FCA app in VC Formal [1]
- Began/Finished FCA tutorial steps and supplied screenshots for documentation [4]
 - Detecting errors, Generating waveforms, resolving errors

- Celina Wong
 - Read through FCA app user guide [0.5]
 - Implemented information about FCA on its tutorial document; finished [3]
 - Invoking GUI, loading TCL
 - Formatted FCA tutorial app document to make it user friendly [2]

Next Week

❖ Team Plan

Next week, our plan is to complete the CC and FPV tutorials, which will be implemented by Dr. Song in his upcoming class project. With the FCA app now complete, Ahliah and Celina will begin exploring the final "Productivity" app in the set, FRV.

END OF WEEK 7 REPORT

❖ Team Review

Our team's focus is on designated tutorials such as FPV, CC, and FRV. Due to half of our team having limited SystemVerilog experience, we encountered some unexpected design file errors while debugging with CC and FRV. This led to team members being pulled away from their designated tasks to provide assistance. Additionally, the lack of information provided by Synopsys for the CC app, compared to other apps, made it difficult to understand this form of design analysis, causing delays.

● Mohamed Ghonim

- Developed and verified a basic SystemVerilog (SV) example consisting of multiple modules intended for utilization in the CC application. [1]
- Collaborated with Dr. Song to develop Project 4, a unique formal verification project assigned to over 70 students enrolled in his class. The project involves the utilization of 4 VC Formal apps for which we provided tutorials. This project is likely unprecedented in the academic realm. [4]
- Developed the FPV tutorial, and wrote over 30 SystemVerilog Assertions while testing it. Developed the example to be used in that tutorial as well and deliberately introduced bugs in it for VC Formal FPV app to find. [7]

● Alex Kim

- Continued research on CC app and its functions [2]
- Run CC app in VC Formal and ensure setup leading up to it was correct [2]
 - Invoking GUI, loading TCL file
- Formatting and working on CC app tutorial steps [2]

● Dmytro Prystupa

- Successfully ran CC application and checked connectivity [2.5]
- Continued work on CC Tutorial steps, making sure user friendly [2]
 - Screenshots, highlights, introduction

● Ahliah Nordstrom

- Researching and familiarizing myself with FRV app in VC Formal [3]
- Began working on design and TCL file for FRV app [2]

● Celina Wong

- Finalized and pushed SEQ and Getting Started tutorials onto GitHub [1]
- Updated Trello [.25]
- Read FRV app chapter in the VC Formal user guide [1.5]

Next Week

❖ Team Plan

Make sure all “Productivity” apps are finalized and polished, and complete the first set of tutorials before diving into the more difficult “High Value” app set. Hopefully, wrap up the FPV High Value app so Dr. Song can have it for his class this term; this is the only “High Value” app exception. Dr. Song knows how difficult this app is and understands the long hours we've put into it so far.

END OF WEEK 8 REPORT

❖ Team Review

We focused on wrapping up all “Productivity” apps and the FPV app, as it will be used for Dr. Song’s class for their final class project. Heading into the last couple weeks of the term, we are right on the mark with our project timeline. Although there are some existing and new road bumps with the last few “Productivity” apps, this week gave us some confidence. We heard some new feedback from users and it was positive.

● Mohamed Ghonim

- I generated two examples to evaluate different scenarios of the FPV app, testing its capabilities in Case Splitting, Counter Abstraction, Memory Abstraction, and Symbolic Variable. [4]
- Dedicated 2 hours to complete Synopsys tutorials and comprehend their user manual regarding the signoff process. Also, sketched out a plan to present the entire workflow in our tutorials, keeping in mind the broader perspective. [2]
- Explored the topic of Unbounded Linear Temporal Operators (LTL) in SVA and practiced writing expressions through generating additional examples. [1]
- Searched for open-source Register Abstraction Layer (RAL) models to be utilized in the Functional Register Verification (FRV) application. [0.5]

● Alex Kim

- Continued on CC Application Tutorial steps and translated them over to document [3]
 - Resolving errors
- Researching debugging function in CC app [1.5]
- Trello/Github check [.25]

● Dmytro Prystupa

- Continued on CC Application Tutorial steps and translated them over to document [3]
 - Source tracing

● Ahliah Nordstrom

- Continued working/debugging FRV design file [3]
 - Getting it to compile
- Began working on the initial FRV Tutorial steps, creating folders, running TCL [2]

- Celina Wong
 - Minor revisions/additions to FXP app based on user feedback [1.5]
 - Finalized and pushed AEP, FXP, and FCA docs onto GitHub [1]
 - Updated Trello [.25]

Next Week

- ❖ Team Plan

Finalize “Productivity” apps! Begin working on “High Value” apps and shoot tutorial videos for finished tutorials. We also want to make and release a quick final survey to Dr. Song’s students for feedback on the apps they have been using for their projects.

END OF WEEK 9 REPORT

❖ Team Review

We slowed down again this week for final exams. Big progress was made with the CC tutorial and it is now complete. As we took time to focus on final exams, we made sure to round back to the tutorials that we haven't already reviewed. Since Dr. Song wanted these tutorials ASAP, we didn't have the chance to go through each tutorial individually; this was the week to do so.

● Mohamed Ghonim

- I watched tutorials on how to create sequences in SVA and spent time exploring various ways to implement them in our upcoming tutorials. This is a complex topic that requires significant effort to fully grasp. [2]
- Coordinated with Cadence Design Systems to arrange additional tutorials and facilitate a comparison between Cadence's Formal Verification solution, Jaspergold, and Synopsys's VC Formal. Contact was made with the PSU CAT team, as well as our sponsor, Dr. Song, to ensure the smooth execution of this endeavor. [3]
- Searched for open-source System-on-Chip (SoC) designs to be utilized in the tutorials, particularly in reproducing a sample scenario in the CC application. [0.5]
- Acquired additional knowledge on the concept of "coverage" in verification and its significance in the context of formal verification compared to simulation. [1]

● Alex Kim

- Finished final draft of CC Application Tutorial [4.5]
 - Formatting, resolving errors
- Discussed with team members on progress thus far (including CC app) [0.5]
- Reviewed FPV app, going through tutorial steps in VC Formal [1]

● Dmytro Prystupa

- Finished final draft of CC Application Tutorial [5]
 - Formatting, source tracing
- Reviewed FPV app, going through tutorial steps in VC Formal [1]

● Ahliah Nordstrom

- Reviewed CC app, going through tutorial steps in VC Formal [1.5]
- Reviewed FPV app, going through tutorial steps in VC Formal [1]

- Celina Wong
 - Reviewed CC app, going through tutorial steps in VC Formal [1.5]
 - Reviewed FPV app, going through tutorial steps in VC Formal [1]
 - Updated Trello [.5]

END OF WEEK 10 REPORT

❖ Term Review

Our team has completed 5 tutorials (SEQ, AEP, CC, FCA, and FPV) in addition to the first 3 getting started tutorials on how to use VPN to connect to the PSU remote labs, installing Mobaxterm and starting a VNC connection, and installing the Synopsys VC Formal package. Although we initially planned to focus on all "Productivity" apps this term as they are easier to produce and would take about one week each, with the guidance of our faculty/industry advisor, we developed a tutorial for a "High Value" app: FPV. As a result, this took the place of the fifth and final "Productivity" app, FRV. Everyone contributed their agreed-upon part to the best of their ability, and we are currently on schedule. We plan to carry this momentum into next term.

Synopsys VC Formal Project

Team 7 Weekly Progress Reports 2
Spring 2023

Mohamed Ghonim
Alex Kim
Dmytro Prystupa
Ahliah Nordstrom
Celina Wong

Start Date: 2023-04-03
End Date: 2023-06-16

❖ Team Review

For the first week of the second quarter, we went over the specifications in the PDS to see which tasks there are left. We compiled and split those tasks into mini-teams to work on.

- Mohamed Ghonim (Spring break + first week).
 - Installed Cadence JasperGold and took online classes on its apps during the break. (Got certified!). This is an added feature that Dr. Song requested. [many break hours! +20]
 - Tested small examples in JasperGold's Superlint App and Formal Property Verification app. [3]
- Alex Kim
 - Reviewed which apps still need tutorials and begin to formulate gameplan with the team [2]
 - Begin to focus on FSV and FTA apps by looking at user guides [3.5]
- Dmytro Prystupa
 - Review FSV and FTA applications [2]
 - Test FSV and FTA applications [3]
- Ahliah Nordstrom
 - Continue reading FRV user guide [1]
 - Update Trello [.5]
 - Drafting first FRV tutorial [2]
- Celina Wong
 - Fix CC app and formatting [blocked]
 - Review FPV [blocked]

Next Week

❖ Team Plan

We should create a formal survey for our test plan subjects from the previous term. Mini-teams should get started on their assigned apps.

END OF WEEK 1 REPORT

❖ Team Review

We made progress in various areas of the project, including developing and debugging an example in the JasperGold Superlint app with assistance from tutorials and Cadence support. We studied the VC Formal FuSa app documentation and worked on compiling a TCL file. Additionally, we reviewed the FSV app user guide and started creating the corresponding tutorial. We also drafted the FRV tutorial and researched the DPV and FuSa user guides, contributing to the DPV tutorial documentation. Our teamwork demonstrates dedication and advancement in our tasks.

● Mohamed Ghonim

- Worked on developing and debugging an example in the JasperGold Superlint app, resolving issues by consulting tutorials and reaching out to Cadence support [4].
- Studied the documentation for the VC Formal FuSa app and attempted to compile a TCL file, which initially encountered issues [2].

● Alex Kim

- Reviewed FSV app user guide to begin tutorial [2]
- Begin tutorial for FSV app [4]

● Dmytro Prystupa

- FSV application Tutorial [4]
- FSV application user guide and tutorial review [2]

● Ahliah Nordstrom

- Finish drafting FRV tutorial [3]
- Begin researching and reading DPV user guide [2]
- Begin researching and reading FuSa user guide [1]

● Celina Wong

- Go through DPV User Guide [2]
- Add some information in DPV tutorial documentation [2]

Next Week**❖ Team Plan**

Start diving deeper into the applications.

END OF WEEK 2 REPORT

❖ Team Review

We worked on various aspects of the project, including exploring the functionalities of the JasperGold GUI interface in the Superlint app, reviewing the FSV app user guide, troubleshooting and debugging the FSV application, researching and reading the DPV and FuSa user guides, and making progress on the corresponding tutorials. Our efforts were focused on understanding the tools, resolving issues, and creating comprehensive documentation for future reference.

- Mohamed Ghonim

- Explored the various functionalities of the JasperGold GUI interface in the Superlint app, researching the three different modes of operation to determine the most suitable one for our tutorials [2].
- Explored the process of performing Formal Property Verification (FPV) in JasperGold, initially expecting similarity to VC Formal but found that the VC Formal methodology did not apply, prompting further exploration of alternative methodologies [3].

- Alex Kim

- FSV app user guide review [1]
- Continue tutorial for FSV app [3.5]

- Dmytro Prystupa

- Troubleshooting and debugging FSV application [4]
- FSV Tutorial Work [3]

- Ahliah Nordstrom

- Continue researching and reading DPV user guide [1]
- Continue researching and reading FuSa user guide [3]
- Update Trello [.25]

- Celina Wong

- Go through FuSa documentation [1]
- Continue building DPV tutorial [2]

Next Week

❖ Team Plan

We plan to conduct a progress update meeting with our sponsor to discuss our project's advancements and seek guidance on sourcing relevant examples for our tutorials. We will collaborate on revising and creating a commented TCL file for the FuSa app, based on a tutorial video. Our focus will be on troubleshooting and debugging the FSV app tutorial, while also exploring the FuSa application in VC Formal and starting the documentation for it. Additionally, we will work on outlining the code for the FuSa TCL file and continue building the DPV tutorial.

END OF WEEK 3 REPORT

❖ **Team Review**

We conducted a progress update meeting with our sponsor to provide an overview of our project's advancements and seek guidance on sourcing relevant examples for our tutorials. We assisted each other in revising and creating a commented TCL file for the FuSa app, compiling a sample file based on a tutorial video. We continued working on the tutorial for the FSV app, troubleshooting and debugging as needed. Additionally, we explored the FuSa application in VC Formal, wrote the background and introduction for the tutorial, and began outlining the code for the FuSa TCL file. We also made progress on building the DPV tutorial.

● **Mohamed Ghonim**

- Conducted a progress update meeting with our sponsor Professor Song, providing an overview of our project's advancements and seeking guidance on sourcing relevant examples for our tutorials [1.5].
- Assisted Ahliah and Celina in revising and creating a commented TCL file for the FuSa app, compiling a sample file based on a tutorial video from the Synopsys Portal [2].

● **Alex Kim**

- Continue tutorial for FSV app [4.5]
- Troubleshooting .TCL file for FSV [1.5]
- Begin to discuss gameplan for FTA app [1]

● **Dmytro Prystupa**

- FSV app tutorial work [5]
- FSV troubleshooting and debugging [2]
- FTA app review [1]

● **Ahliah Nordstrom**

- Explore FuSa application in VC Formal [1]
- Write FuSa tutorial background and introduction [2]
- Begin code outline of FuSa TCL file [1.5]

● **Celina Wong**

- Continue building DPV tutorial [2]

Next Week

❖ Team Plan

For the upcoming week, our plan is to hold a team meeting to discuss the final report, presentation, and establish individual deadlines. We will ensure everyone is informed about the progress by providing an update to Professor Song. Furthermore, we will dedicate time to review the FTA and FSA tutorials, seeking opportunities for improvements. As we continue exploring the JasperGold apps, we anticipate uncovering distinct methodologies in comparison to VC Formal.

END OF WEEK 4 REPORT

❖ Team Review

We attended a team meeting to discuss the final report, presentation, and set individual deadlines. We provided an update on our progress to Professor Song. Additionally, we reviewed the FTA and FSA tutorials for potential improvements. In our exploration of the JasperGold apps, we discovered that many of them have different methodologies compared to VC Formal.

● Mohamed Ghonim

- Continued refining and finalizing the JasperGold Superlint app tutorial, making further progress towards its completion. Spent additional time interpreting the analysis results. [5]

● Alex Kim

- Finalize FSV app tutorial by proofreading, checking tables/figures, etc. [5]
- Review FTA user guide & begin FTA tutorial [2]

● Dmytro Prystupa

- FTA application review and user guide review [2]
- Finish and review FSV application [5]
- FTA application tutorial [2]

● Ahlia Nordstrom

- Running and debugging FuSa TCL file [3]
- Create code outline of FuSa example code [2]

● Celina Wong

- Continue building DPV tutorial, add more tables and simplified explanations [6]

Next Week**❖ Team Plan**

Next week, our plan is to progress with the final report by outlining and writing sections such as the Executive Summary, Background, and Product Design Specification. Additionally, we will explore the Pulpino SoC code and design, review the FTA user guide, start the FTA app tutorial, document the FTA application tutorial, run and debug FuSa example code, and create an appendix table for FuSa fault types. We will also include screenshots from running the DPV tutorial example code.

END OF WEEK 5 REPORT

❖ **Team Review**

We made progress on the final report by outlining and writing sections such as the Executive Summary, Background, and Product Design Specification. Additionally, we explored the Pulpino SoC code and design, reviewed the FTA user guide, started the FTA app tutorial, documented the FTA application tutorial, ran and debugged FuSa example code, and added an appendix table for FuSa fault types. We also included screenshots from running the DPV tutorial example code.

● **Mohamed Ghonim**

- Created the outline for the final report, including the introduction and initial content for each section. Developed approximately one page of content for the following sections: Executive Summary, Background, and Product Design Specification (Requirements and Stakeholders). [4]
- Explored the Pulpino SoC code and design for possible examples for our project. 2 of my tests failed. The Pulpino SoC design files are nested in a way that may not be suitable for our project [3]

● **Alex Kim**

- Review FTA user guide for tutorial [1]
- Begin FTA app tutorial [4]

● **Dmytro Prystupa**

- FTA application tutorial documentation [5]

● **Ahliah Nordstrom**

- Running and debugging FuSa example code [3]
- Create appendix table for FuSa fault types [1]

● **Celina Wong**

- Run user guide example code and put screenshots in DPV tutorial [3]

Next Week

❖ Team Plan

We plan to make progress on the final report, with a focus on sections such as Objectives and Deliverables and Approach. Our plan is to develop a concise example for the DPV (Datapath Validation) app, ensuring its inclusion in the tutorial documentation with relevant screenshots. We will share insights with the team regarding the complexity of C++ and SV designs in DPV and emphasize the importance of presenting simpler examples in our tutorials. Additionally, we will continue working on the FTA app tutorial and troubleshooting, format the tutorial specifically for FTA, and work on formatting screenshots and steps for the FuSa tutorial. Finally, we will finalize the DPV document.

END OF WEEK 6 REPORT

❖ **Team Review**

We made progress on the final report, focusing on sections like Objectives and Deliverables and Approach. We also worked on developing a concise example for the DPV (Datapath Validation) app, ensuring its inclusion in the tutorial documentation with relevant screenshots. We shared insights with the team about the complexity of C++ and SV designs in DPV and the importance of presenting simpler examples in our tutorials. Additionally, we continued the FTA app tutorial and troubleshooting, formatted the tutorial specifically for FTA, and worked on formatting screenshots and steps for the FuSa tutorial. Lastly, we finalized the DPV document.

● **Mohamed Ghonim**

- Continued working on the the final report, wrote about a page for those sections Objectives and Deliverables and Approach (2.5)
- Developed a concise and illustrative example for the DPV (Datapath Validation) app, ensuring its inclusion in the tutorial documentation. Incorporated relevant screenshots to enhance the clarity of the tutorial. Shared insights with the team regarding the complexity of C++ and SV designs typically tested in DPV, emphasizing the significance of presenting simpler examples in our tutorials.(4)

● **Alex Kim**

- Continue FTA app tutorial and troubleshooting with TCL and app use [5]
- Format tutorial specific for FTA [3]

● **Dmytro Prystupa**

- FTA App tutorial work [5]
- FTA app troubleshooting and debugging [3]

● **Ahliah Nordstrom**

- Formatting screenshots/steps for FuSa tutorial [3]

● **Celina Wong**

- Finalize DPV document [1]

Next Week

❖ Team Plan

We have a team meeting scheduled to discuss the final report, presentation, and establish deadlines. We will provide an update on our progress to Professor Song and review the FTA and FSA tutorials for potential enhancements. During our exploration of the JasperGold apps, we will note the differences in methodologies compared to VC Formal. Our plan includes finalizing the FTA and FSV tutorials, updating the FSV tutorial, and working on the initial draft of the FuSa tutorial. We will also update our project management tool, Trello, and initiate the drafting process for the final report.

END OF WEEK 7 REPORT

❖ Team Review

We attended a team meeting to discuss the final report, communicated our progress to Professor Song, reviewed the FTA and FSA tutorials, and explored alternative implementations of JasperGold apps. We are finalizing the FTA and FSV tutorials, updating the FSV tutorial, working on the first draft of the FuSa tutorial, updating Trello, and starting the final report draft.

● Mohamed Ghonim

- Attended a team meeting discussing the final report, presentation, and setting our own deadlines. [0.5]
- Communicated our progress with prof. Song. [0.5]
- Reviewed the FTA and FSA tutorials for possible improvement [0.5]
- Continued to explore further possible implementations of JasperGold apps in our capstone. Many of the apps ended up having quite different methodologies than VC Formal. [2.5]

● Alex Kim

- Complete FTA app and proofread table of contents, figures, tables, etc. [6]
- Review other team members tutorials [1]

● Dmytro Prystupa

- Finalizing FTA and FSV tutorials [4]
- Updating FSV tutorial [2]

● Ahliah Nordstrom

- Finalize first draft of FuSa tutorial [2]
- Update Trello [.5]
- Contributed to the final report draft background and design sections and rewrote the outline [1]

● Celina Wong

- Go through FSV document [1]
- Go through FTA document [1]

END OF WEEK 8 REPORT

- Mohamed Ghonim
 - Continued to work on Finalizing, and indexing the tutorials and sent the current version to Dr. Song [2]
 - Worked on writing and finishing the final draft of our academic paper [5]
- Alex Kim
 - Worked on academic article [1]
 - Worked on final report [3]
 - Begin discussion about poster board with the group [1]
- Dmytro Prystupa
 - Academic article review [1]
 - Final report review/work [2]
- Ahliah Nordstrom
 - Perform final revisions to all tutorials and making sure they all look alike/coherent. [6]
 - Reviewed the academic article for VC Formal capstone. [1]
- Celina Wong
 - Work on academic article [2]
 - Work on final report [2]

END OF WEEK 9 REPORT

- Mohamed Ghonim
 - Attended a group team meeting to work on the poster outline and putting things together. [1]
 - Designed draft 1 and 2 of the poster in Photoshop, and discussed them with the team and the sponsor Dr. Song, to improve our final version. [5]
 - Indexed and organized the final tutorials (version 1.2) and sent them to Dr. Song [1]
 - Met and Discussed with Dr. Song our capstone project and ways to improve the academic paper [1]
- Alex Kim
 - Worked on poster board draft design in Google document as well as creating poster board design using Indesign [5]
 - Group meeting [1]
- Dmytro Prystupa
 - Reviewed FTA, FSV, CC application tutorials. Edited and added missed sections onto each tutorial [4]
 - Group meeting for poster outline [1]
 - Poster work [1]
- Ahliah Nordstrom
 - Continued working on the final revision of all tutorials, editing and making note of missed sections, and updating all table of contents' [5]
 - Worked/reviewed poster board design [1]
- Celina Wong
 - Worked on poster board design in shared Google document

END OF WEEK 10 REPORT

❖ Team Review

Had our last team meeting to discuss presentation preparation, finalized Canvas documents that are due on Friday (6/16).

● Mohamed Ghonim

- Reviewed and revised the final report, incorporating the challenges encountered by our team and including an appendix with two project reports developed using our tutorials. [3]
- Participated in a team meeting to discuss the final report review, finalize details, and plan for the upcoming poster session. [1]
- Completed the CATME survey to provide valuable feedback on the team dynamics and collaboration throughout the capstone project. [1]

● Alex Kim

- Final review of final report and adding missing or revised material since the rough draft submission [2]
- Poster board review for presentation [1]
- CATME survey [1]

● Dmytro Prystupa

- Reviewed all tutorials for missing content [2]
- Team meeting, planning final details of capstone [1]
- CATME [1]

● Ahliah Nordstrom

- Fill out CATME survey [1]
- Planning final details in team meeting [1]

● Celina Wong

- Completed CATme survey
- Fix up the final report with spacing, page numbers, etc.

END OF WEEK 11 REPORT

❖ Term Review

Our team successfully completed 11 VC Formal applications in total.

The team would meet up every week for the first couple of weeks, and then we would split up tasks and go our separate ways for the next couple of weeks. We would also check on each other every other week or so to see where the progress lies for each member. This method enabled us to work at our own pace (very important since we had other classes to worry about), and every member did their assigned tasks. We never really had issues with communication; if anyone ever had any concerns or questions, we were all readily available on Discord.

References

[1] Synopsys Inc. (2022). VC Formal User Guide Version T-2022.06-SP2, December 2022. This comprehensive user guide provided detailed documentation and step-by-step instructions for utilizing VC Formal. It served as a valuable reference throughout our project, offering insights into the features, methodologies, and best practices of VC Formal.

[2] Cadence Design Systems Inc. (n.d.). JasperGold UserGuide. The JasperGold UserGuide served as an essential resource, providing extensive documentation and guidance on effectively using the Cadence JasperGold Verification Platform. It offered detailed explanations of the platform's features, workflows, and advanced verification techniques, enabling us to leverage the full potential of JasperGold in our capstone project.

[3] Synopsys Inc. (n.d.). VC Formal. Retrieved from <https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html>. We referenced the official Synopsys VC Formal webpage, which provided a comprehensive overview of VC Formal's capabilities, use cases, and benefits. This webpage offered valuable insights into the tool's industry significance and helped us gain a deeper understanding of its applications in static and formal verification.

[4] Cadence Design Systems Inc. (n.d.). JasperGold Verification Platform. Retrieved from https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html. The Cadence JasperGold Verification Platform webpage served as a valuable reference, providing an in-depth overview of the platform's features, capabilities, and its role in formal and static verification. It contributed to our understanding of JasperGold's functionalities and guided our exploration of the platform throughout our capstone project.

These references were instrumental in our research and understanding of VC Formal and JasperGold. They provided valuable insights, technical guidance, and industry perspectives, which greatly influenced the successful application of these tools in our capstone project.