VERIFICATION OF RTL Design Using Formal Technique

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI DESIGN

Submitted By

Archit Gupta

602362005

Under Supervision of

Dr. Anil Singh

(Associate Professor)

Dr. Chandramohan Dhasarathan

(Assistant Professor)

Mr. Paras Gupta
(Formal Verification Manager, Intel)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITIUTE OF ENGINEERING AND TECHNOLOGY

(A DEEMED TO BEUNIVERSITY), PATIALA, PUNJAB

JUNE, 2025

DECLARATION

I, **Archit Gupta** hereby declare that the work presented in this thesis entitled **" Verification of RTL Design using Formal Technique"** in partial fulfilment of the requirement for the award of degree of **Master of Technology (VLSI Design**) submitted at **Electronics and Communication Engineering Department**, Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under supervision of Industry Mentor **Paras Gupta (Formal Verification Engineering Manager, FVCTO , Intel Technology India Private Limited)** and **Dr. Anil Singh (Associate Professor) & Dr. Chandramohan Dhasarathan** (**Assistant Professor, Electronics and Communication Engineering Department, Thapar Institute of Engineering & Technology**) from **January, 2025** to **June, 2025**. The matter presented in this has not been submitted either in part or full to any other university or institute for the award of any other degree.

Date: 30 June 2025

Archit Gupta
(602362005)

(29-6-25)

Paras Gupta
(Formal Verification Engineering Manager)

Dr. Anil Singh
(Associate Professor)

Dr. Chandramohan Dhasarathan
(Assistant Professor)

Regd. Office:
Intel Technology India Private Limited
23-56P, Outer Ring Road,
Devarabeesanahalli, Varthur Hobli website: www.intel.in
Bellandur Post
Bangalore 560 103, India
CIN-U85110KA1997PTC021606

Tel: +91-80-2605 3000
Fax: +91-80-2605 6190

To Whomsoever It May Concern

WWID: 12253966

Employee Name: Archit Gupta

Internship Dates: 24/06/2024 to 23/06/2025

The letter is to confirm the mentioned above has undergone internship at Intel Technology India Private Limited.

We wish you all the best for your future assignments.

Yours Sincerely

Gowre Saseedaran

Date: June 27, 2025

Place: **Bangalore**

ACKNOWLEDGEMENT

This thesis would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost, my sincere gratitude to **Dr. Anil Singh & Dr. Chandramohan Dhasarathan** for his continuous support,motivation, immense knowledge. His guidance has helped me in all the time of research andwriting of the thesis.

Mr. Paras Gupta, FVCTO, Intel Pvt Ltd for his patience and steadfast encouragement, his constructive criticisms at different stages of my work were thought-provoking and he helped me focus on my ideas. I am deeply grateful to him for all discussions that helped me in understanding the technical details of my work.

Last but not the least, my family and dear friends. None of this would have been possible without their love. And the one above all of us, the omnipresent God for giving me the strengtht plod on, thank you so much everyone.

ABSTRACT

As SOC designs have become more complex, traditional verification methods are increasingly difficult and expensive to implement effectively. To tackle this challenge, current research is exploring the integration of formal verification with changes in design methodology. Early formalization of abstract models has shown potential in identifying design flaws sooner, which can reduce the cost of fixing bugs. Recognizing that different verification challenges require tailored approaches is crucial for effectively applying formal verification in the early design stages. By using various models and tools, different perspectives on the design can be explored, helping to capture the design's essence in a way that allows quick validation and modification. Identifying corner-case issues early in the design process can significantly reduce the costs of re-designing later on.

While traditional formal verification, which begins from the very start of the design, is useful for initial verification, it often falls short when it comes to uncovering complex bugs that result from rare or intricate interactions between events. In this thesis, introduce a methodology that utilizes functional simulation traces to guide formal verification. Instead of starting the formal verification process from the beginning, we focus on "interesting spots" in the simulation that are likely to reveal complex bugs. By using formal engine health to prioritize these critical areas, we can make the bug-hunting process more efficient and targeted. This approach not only improves the chances of detecting complex bugs but also helps reduce the cost of fixing them.

TABLE OF CONTENTS

DESCRIPTION……..……..…….PAGE NUMBER

LIST OF FIGURES

DESCRIPTION……..……..…….PAGE NUMBER

CHAPTER 1

INTRODUCTION

1.1 Introduction to the area of work

As design complexity continues to grow, driven by advancements in fabrication technologies, the challenge of verification becomes increasingly significant. Traditionally, simulation techniques have been the primary method for verifying design correctness, and ongoing improvements in simulation technologies—such as coverage models, guided test generation, and faster simulation speeds—have made it more efficient at identifying bugs. Meanwhile, formal methods, particularly through the use of model checkers, have shown success in achieving complete coverage for small modules or conceptual protocols. These advances have also facilitated the development of hybrid verification approaches, where formal methods are used to guide simulation or even replace it in certain areas of the verification process [1].

However, both simulation and formal verification still face challenges as design complexity increases, particularly with the rise of parallel processing units. As designs become more intricate, the likelihood of missing corner cases increases, and simulation tools often struggle to analyze these complex scenarios. While formal verification has proven useful in identifying corner-case issues, its current capabilities are primarily limited to small designs. The consistent application of formal methods to large-scale, multi-module systems remains an ongoing challenge, and although formal verification has been successfully used for high-level designs, these efforts are often time-consuming and design-specific [1].

Formal verification is also valuable for quickly validating protocol concepts before the actual design phase. However, connecting validated protocol models to the final implementation has yet to be fully explored. Although challenges exist, formal verification remains a reliable method for validating intricate system on chips safety sensitive designs. The methodology encompasses three core elements: assurance (to prove correctness), bug hunting (to find known and unknown bugs), and coverage closure (to identify reachable or unreachable design elements) [2].

Emerging approaches, such as waypoints, coverpoints, and goal-posting, are becoming increasingly popular for uncovering deep, complex bugs that traditional methods might miss. As traditional formal verification approaches start from time zero and explore designs in their entirety, they become less efficient as the design space expands. Complex bugs often arise from rare events, and sophisticated bug-hunting techniques are now being employed to identify these issues more effectively [2].

1.2 Motivation of the work

The traditional approach to programming has largely been centered around starting with an abstract model and progressively incorporating more specific details. This method was the dominant paradigm until the emergence of object-oriented programming. The concept behind this approach is that a program can become more flexible and responsive to changes, whether they are extensions or corrections, by incrementally refining its architecture. In this process, only the final design decisions

and choices may need to be revisited and potentially re-implemented, as these modifications or additions often do not necessitate changes to the initial abstract framework. This approach is commonly referred to as the top-down design methodology, emphasizing a structured progression from general concepts to specific implementations. Formal verification appears to be a good fit for this methodology because it allows for the refinement layers to differ little, making it feasible to use formal methods to prove that these two layers coincide. Before moving on to the next level of refinement, each design layer can be made 100% coverage verified by formal verification, ensuring that no bugs remain. By doing this, it is certain that errors won't spread during the refinement process. Formal verification presents a challenge to the design process since, in the process of designing a project, designers must consider a multitude of factors in addition to the design's correctness. Because implementation efficiency and performance are taken into consideration, the design frequently evolves as it moves forward. Because it takes a lot of effort to formalize an abstract design, formalize the features of interest, and identify methods for reducing the design to a manageable size, this makes it difficult to undertake formal verification using refinement approaches during the design process. Moreover, modifications to the abstract design have an impact on formal verification efforts. Therefore, the labor-intensive parts of formal verification must be abandoned and reanalyzed when the actual design calls for modifications to the abstract model. Therefore, it is ineffective to utilize standard formal verification approaches in conjunction with design refinement since formal verification cannot keep up with the rapid advancement of designs due to modifications made to the abstract level model during the design process [3].

The ability to assess the design early in the process, when changes are less expensive to make, is the driving force behind the refining methodology. It makes it possible to identify and address design flaws early on from the standpoint of verification. Additionally, using an abstract design for bug hunting makes it simpler to track down the cause of design flaws and to troubleshoot using the abstract model. The benefits of early design verification are so great that most design approaches aim to achieve early design verification, whether explicitly or implicitly [3].

According to several functional verification surveys, two spins are necessary before manufacturing for at least 50% of design projects. The study also demonstrates that functional defects in the RTL design are the primary cause of respins. Respins can have a significant financial impact on the product and are very costly in terms of time and money. Fig. 1.2 shows a chart that illustrates the underlying causes of functional defects in design projects. The y-axis displays the proportion of design projects impacted by each sort of functional fault, while the x-axis displays various underlying causes. Design mistakes were the source of functional defects in seventy percent of the design initiatives. These kinds of design flaws can be linked to the design engineers' improper interpretation of the specifications. Additionally, the graphic indicates that 50% of the projects are impacted by defects brought about by modifications to the requirements, and 50% of the projects are impacted by inaccurate or incomplete specifications. The figures in Fig. 1.1 indicate a significant flaw in the design flows, namely the use of unofficial requirements to characterize a design's functions. Functional defects are mostly caused by insufficient and confusing informal specifications. Creating attributes based on loose specifications may result in an inaccurate or partial set of properties, which ultimately affects the quality of the verification process.

Furthermore, when the DUV's parameters change frequently, it becomes challenging to sustain the workflow since manual property coding necessitates a significant amount of rework.

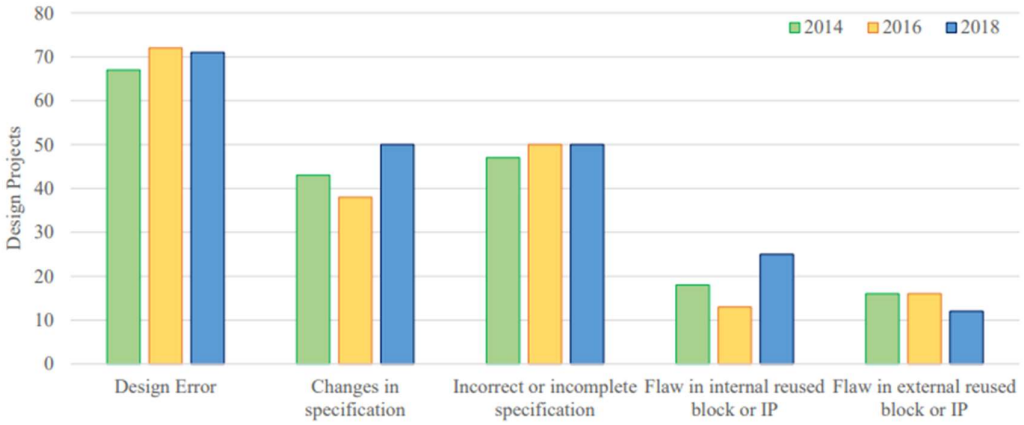


Figure. 1.1 Root cause of functional flaws [4]

The promise of formal verification (FV) is to achieve stronger, true verification of the DUV instead of merely testing with simulation-based methods. FV can provide full verification for at least part of a DUV, full checking of properties for all DUV states, in contrast to the limited. During the last decade, FV has established itself as a productive additional weapon in the arsenal of the verification team. Many engineers still misunderstand FV as an academic approach. However, rapid improvements and much practical use of formal methods yielded advances in areas like specification and assertion languages. These advances not only improved productivity of verification methods in the formal field, but also have influenced simulation methods [4].

1.3 Objective

To ensure converged results in the design verification process, we aim to address the issue where formal proofs do not converge due to bounded proof limitations. While the formal proofs confirm that the design holds within certain bounds, uncertainty remains beyond those bounds. To achieve a more definitive and converged result, we propose the application of the Bug Hunt technique, which is typically employed during the sign-off phase in formal verification. This approach not only helps to identify potential bugs but also enhances coverage, thereby improving the overall reliability and confidence in the design verification process.

CHAPTER 2

LITERATURE REVIEW

2.1 Design Verification

The design process is a systematic approach that converts a set of specifications into a tangible implementation. At the specification level, these specifications define the functionality that the design is intended to perform, but they do not detail the methods by which this functionality will be achieved. The implementation phase, on the other hand, provides a comprehensive explanation of how the specified functionality is realized.

Design verification serves as the counterpart to the design process. It begins with the execution of the implementation and involves ensuring that this implementation meets the established requirements. Therefore, each stage of the design process is accompanied by a corresponding verification phase to confirm its correctness. For example, when a design phase translates a functional specification into an algorithmic implementation, a verification phase is necessary to ensure that the algorithm performs according to the features outlined in the specification. Similarly, when a physical layout is created from a gate netlist, verification is required to ensure that the layout accurately reflects the gate netlist.

Design verification covers a broad spectrum of areas, including functional verification, timing verification, and layout verification, among others. Figure 2.1 visually represents the relationship between the design process and the verification process, highlighting how each phase of design is paired with a verification step to ensure accuracy and compliance with the original specifications [6].

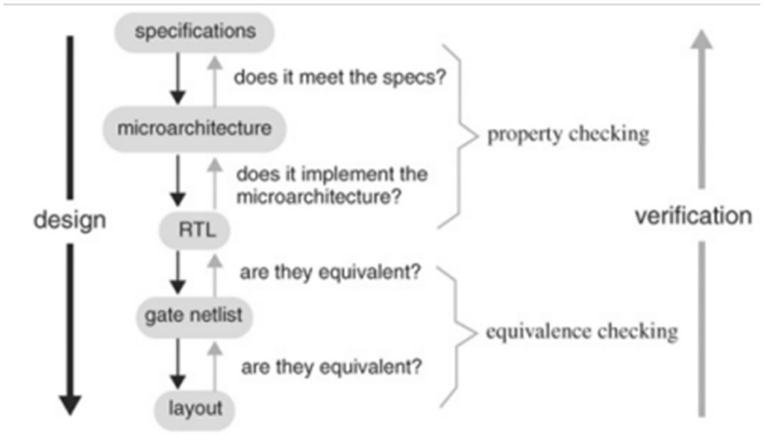


Figure. 2.1 Correlation between design verification and design processing [5]

Design verification is a systematic approach aimed at ensuring that a design meets its specified requirements and complies with established design rules. This phase is a critical component of the product development process, as it seeks to identify and address design issues early on, thereby avoiding costly and time-consuming modifications later in the development cycle. By conducting thorough design verification, the reliability and proper functioning of the final product—whether it is an

integrated circuit (IC), a system-on-chip (SoC), or any other electronic system—are assured.

In particular, the verification of System-on-Chip (SoC) and Application-Specific Integrated Circuit (ASIC) designs is vital for confirming the reliability and optimal performance of these complex integrated circuits. This process involves rigorous testing and analysis to ensure that every aspect of the design operates correctly and efficiently, ultimately leading to a dependable and high-quality end product.

2.2 Verification methodology

Functional verification in design projects is a best-effort process to meet goals in the verification plan, which are based on requirements and insights from architects, developers, and users. The process relies on an approximate verification strategy, which is fundamentally imperfect. An effective verification approach starts with a comprehensive test strategy that outlines the functionality to be verified, ensuring all standards are met. A "scoreboard" technique is used to monitor progress and mark test plan items as completed. Functional coverage and code coverage are commonly used metrics to quantify the quality of verification. A verification methodology must also determine the specific language or languages to be utilized during the verification process. Verilog and VHDL are widely used design languages, while verification code typically requires its own language. An optimal verification language should resemble a software language rather than a hardware language [6].

2.3 Serializer

A **serializer** is a digital circuit or module used to convert parallel data into a serial stream. It is commonly used in communication systems to transmit data over long distances or in high-speed circuits where sending data in parallel would be inefficient. A serializer takes a block of parallel data (usually a set of bits) and sends them one bit at a time over a single channel or line. This design is crucial in reducing the number of physical connections required for data transmission.

Key Concepts

1. **Parallel to Serial Conversion**:
    - A parallel data bus typically consists of multiple lines (e.g., 8, 16, 32 bits wide), and the serializer converts this wide parallel data into a sequence of bits sent one after the other (serially) over a single line.
    - For example, if the input data is 8 bits wide, the serializer will output one bit at a time, starting from the most significant bit (MSB) or least significant bit (LSB), depending on the design.
2. **Clocking**:
    - Serial data transmission often requires synchronization between the transmitter and receiver. To achieve this, the serializer uses a clock signal to control the timing of data shifts.
    - Each clock cycle, a new bit is transferred from the parallel input register to the serial output.

3. **Shift Register**:
    - A **shift register** is the core component of a serializer. It stores the parallel input data temporarily before shifting out each bit serially.
    - The shift register can be configured to work with a specific number of bits (depending on the input width) and is typically clocked to sequentially move the bits from the parallel input to the serial output.

CHAPTER 3

FORMAL VERIFICATION

3.1 Introduction

Formal verification can be defined as a process of checking the correctness of a design implementation against the design specification using mathematical theories. A temporal formula that captures a particular behavior of the design is checked thoroughly on the mathematical model of the design for all permissible input values. Commercial formal verification tools facilitate the use of languages with expanded syntax in industrial practice, which makes it easier to define temporal formulae. The temporal formulas, also known as qualities, are manually derived from specifications. An appropriate error trace, commonly referred to as a counterexample, is supplied in cases where the temporal formula's mathematical model is not supported. It is demonstrated that the design behaves as specified by the specifications when the model holds true for the formula. The three main categories of formal verification procedures are Model Checking, Equivalency Checking, and Theorem Proving. The formal techniques are categorized as shown in Fig. 3.1 [5].

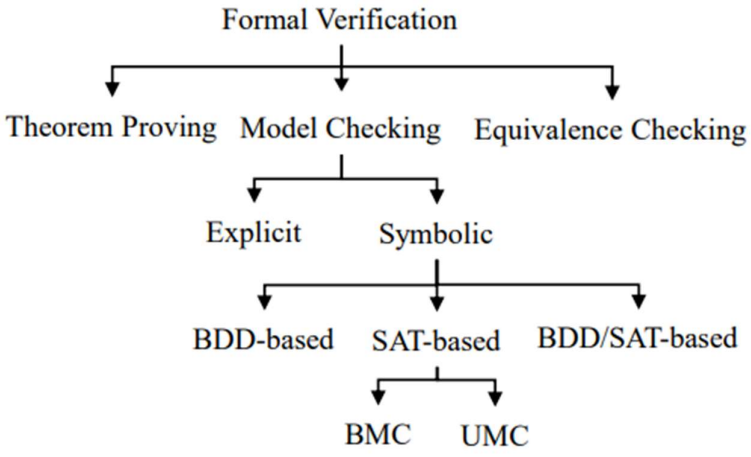


Figure. 3.1 Formal verification techniques [5]

3.2 Functional verification process

Once design specifications are developed for a processor design project, hardware designers proceed to construct a comprehensive, functional representation of the design using a hardware description language (HDL), such as Verilog or VHDL. Verification engineers construct a verification environment in order to accomplish the objectives outlined in a verification plan. The numerous design blocks are allocated to teams of design engineers who construct according to specifications, and verification engineers who conduct unit-level functional verification. Once design blocks are accessible, they are incorporated into subsystems and subjected to verification. The process persists until all the design blocks are finalized and incorporated into the system at the highest level. Design and verification

engineers identify and resolve bugs that are found during the verification process. Pre-silicon verification refers to the functional verification process that takes place prior to the construction of a silicon prototype. During the pre-silicon verification process, the hardware description language (HDL) representation of the design is systematically compared to the specifications using the verification environment. Formal verification, and to a larger extent, simulation-based verification, are the main methods used for pre-silicon verification [5].

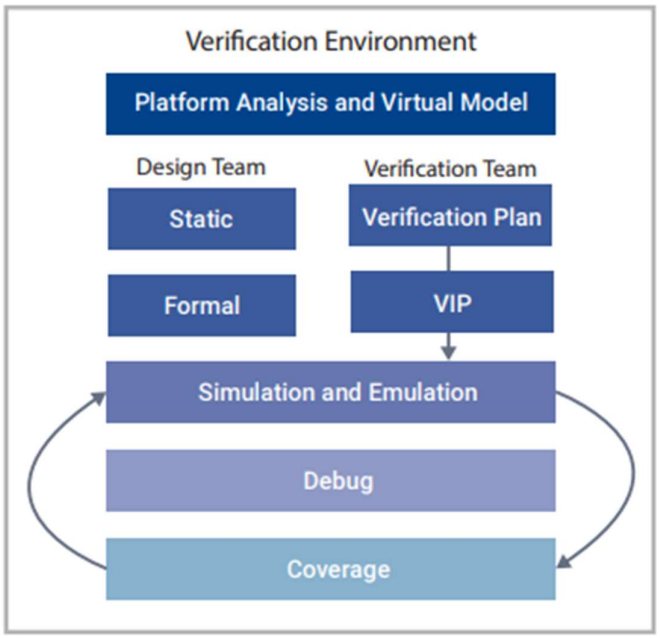


Figure. 3.2 High level view of the chip verification flow [7]

Formal verification tools transform the Hardware Description Language (HDL) representation of the design into Boolean functions. These functions are then compared to attributes that are obtained from the specifications. If engineers can express design specifications as accurate Boolean properties, the tools can mathematically verify or refute the accuracy of the design. The robust assurance that formal verification offers when the design is accurate, and the counterexamples it produces when the design is flawed, renders it an exceedingly desirable instrument for verification. Regrettably, the process of fully capturing a design's specifications as formally provable attributes is a formidable one. Furthermore, the computational requirements of formal verification tools are so high that they only allow for the analysis of tiny design units and subsystems. Hence, the efficacy of formal verification is constrained to diminutive design blocks. Given that the objective of this dissertation is to confirm the accuracy of intricate designs, we will not delve into the topic of formal verification any further.

Simulation-based functional verification encompasses most of the verification work done before the physical implementation of a chip. Software-based simulation systems transform the hardware description of the design into an executable file that is compatible with any computer. The verification testbench, a software that runs parallel to the simulated design, directly interacts with the simulation and has unrestricted access to all design signals. The versatility of software simulation and the extensive

visibility into design signals allows for robust verification and troubleshooting capabilities. Engineers have the ability to create advanced checkers, analyze the design using debuggers, and capture waveforms, among other tasks. Regrettably, software simulators exhibit inadequate speed when dealing with intricate designs, typically replicating a maximum of hundreds of design cycles per second. Simulating a second of a design's operation at these rates might require several months [6].

Simulation based Formal based

Fig. 3.3 Approaches to verification [5]

3.2.1 Simulation-Based Verification

Simulation-based verification is a widely used method of verifying designs in software development. It involves subjecting the design to a test bench, applying input stimuli, and comparing the output with a reference output. The test bench can be pre-produced or generated during the simulation process, and the reference output can be generated either beforehand or in real-time.

A design undergoes a linter program before simulation to identify static mistakes, possible errors, and breaches of code style guidelines. A project implements its own coding style rules to avoid design flaws and enhance simulation performance. Vectors representing test plan elements are created.

Directed tests are input vectors designed to test specific functionality and features, but they tend to be biased towards areas where designers are uninformed. To avoid this, pseudorandom tests are used alongside directed tests. Simulators are selected to conduct simulations, either event-driven or cycle-based.

The efficacy of modeling a test on a design is determined by the extent of coverage the test offers. Coverage tools provide reports on code or functional coverage, allowing designers to identify untested components and generate tests to address those areas.

When a bug is identified, it is essential to promptly inform the designer and address the problem. This can be effectively managed by logging the bug into a bug tracking system. Such a system automatically sends notifications to the design owner, alerting them to the issue. The bug tracking system then oversees the bug's journey through various stages, starting with its initial discovery and entry into the system. The stages typically include opening the bug report, validating the issue to confirm its existence, implementing a fix to resolve the problem, and finally closing the bug once it has been successfully addressed. This structured approach ensures that bugs are systematically managed and resolved,

minimizing disruptions to the design process.

The typical flow of simulation-based verification is summarized in Fig 3.4

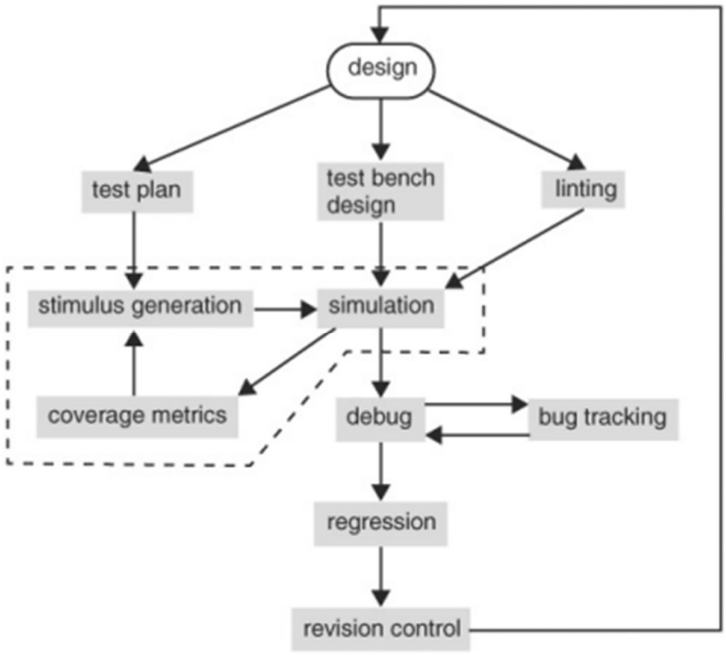


Figure. 3.4 Flow of simulation-based verification [5]

3.2.2. Formal Method-Based Verification

Formal verification provides properties on mathematical models representing the system implementation. In particular, the identification of the design errors is address by means of model checking tools. In the context temporal logic properties are defined to formally check the correctness of a design implementation with respect to the specification [1].

The formal method-based verification methodology offers a distinct approach compared to the simulation-based methodology, primarily because it does not require the creation of test vectors. Despite this difference, both methodologies share similarities in their overall objectives. Formal verification is categorized into two main types: equivalence checking and property verification.

Equivalence checking involves assessing whether two implementations are functionally identical. Attempting to verify equivalence using a simulation-based approach is impractical due to the vast number of possible input vectors, which makes exhaustive testing infeasible. In contrast, formal verification provides a clear and definitive outcome when using equivalency checkers. However, in practice, industrial equivalency checkers have not yet reached the point of being fully automated solutions. They often require users to identify corresponding nodes in the two circuits to focus the checkers' search area. These identified nodes, deemed equivalent by users, are known as cut points. Occasionally, a checker can deduce equivalent nodes based on the names of the nodes, which can

streamline the verification process [8].

The other type of formal verification is property checking. Property verification involves analyzing a given design and a property, which serves as a partial specification of the design, in order to determine whether the design satisfies the property. A property serves as a redundant explanation of the design, effectively confirming its validity. A program that verifies a property is commonly known as a model checker, as it pertains to the representation of a real circuit in the form of a computer model.

The typical flow of formal-based verification is summarized in Fig 3.5 [8].

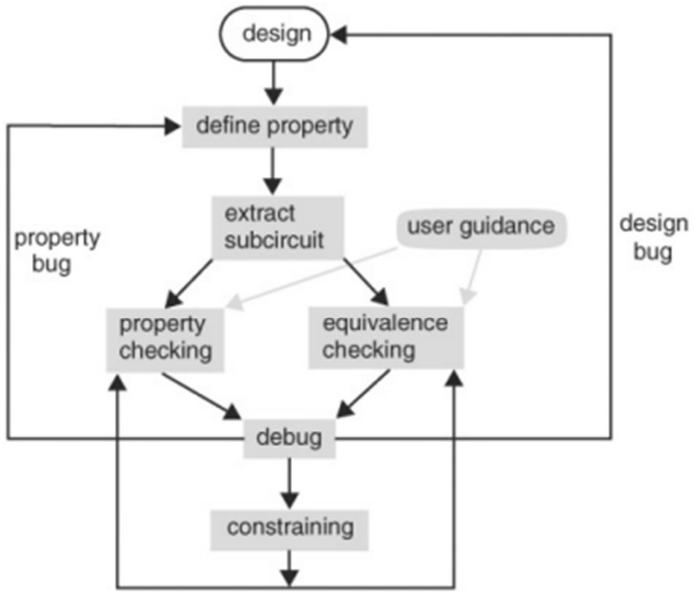


Figure. 3.5 Flow of formal based verification [5]

3.3 Simulation-Based Verification versus Formal Verification

The primary difference between formal verification and simulation-based verification is that the latter does not require input vectors while the former does. In simulation-based verification, producing input vectors first and then deriving reference outputs is the mindset. The formal verification procedure involves thinking in the opposite direction. The user first specifies the desired output behavior, which the formal checker is then left to confirm or refute. Users don't give input stimuli any thought at all. The formal technique is output driven, whereas the simulation-based methodology is input driven. The tendency to think input-driven is more common and is mirrored in the perceived difficulty of utilizing a formal checker. Completeness—the absence of any gaps in the input space—is another selling factor for formal verification, while simulation-based verification struggles with this issue. This formal verification's power, meanwhile, might occasionally give rise to the false belief that a design is 100% bug-free after formal verification. To find out if formal verification is accurately understood, let's compare simulation-based verification with formal verification. Conceptually, simulating a vector is like confirming a point in the input space. According to this perspective, input space sampling can be used for verification in simulation-based methods. If every point is not sampled, there is a chance that

a mistake will evade verification. Formal verification operates at the property level as opposed to the point level. Formal verification, given a property, looks for failures in every possible input and state circumstance. From an output standpoint, formal verification verifies a set of output points at a time (a collection of output points constitutes a property); simulation-based verification verifies a single output point at a time. This comparison between formal verification and simulation-based verification is shown in Figure 3.6. From this angle, the formal verification approach is different from the simulation-based approach in that it verifies groupings of points in the design space rather than individual points. Therefore, it must be further demonstrated that the collection of attributes that have been formally validated as a whole forms the specifications in order to fully verify that a design satisfies its specifications using formal methods. Formal verification software is less user-friendly and more difficult to use because it verifies a set of points at a time [5].

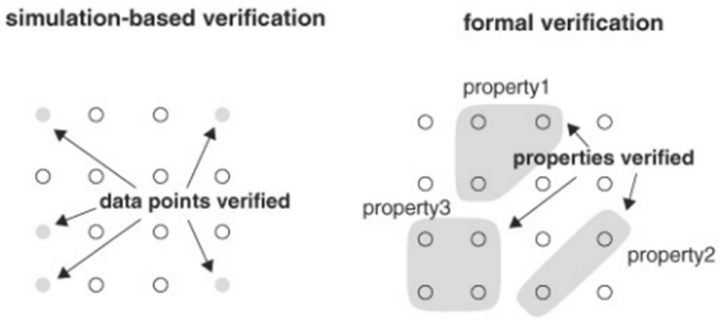


Figure. 3.6 An output space perspective of simulation-based verification versus formal verification [12]

3.4 Design assertions

Formal verification is a process that ensures a design adheres to its specified requirements. To accomplish this, the initial step involves defining what constitutes correctness for the design. This is typically done by formulating assertions using the System Verilog Assertions (SVA) language. SVA provides a structured way to express the expected behavior and properties of a design, allowing for precise verification.

The SVA language can be conceptualized as comprising multiple layers of complexity, each offering progressively advanced capabilities for specifying and checking design properties. These layers enable designers to articulate simple conditions as well as more intricate temporal sequences and relationships within the design. Figure 3.7 illustrates these layers, highlighting how SVA facilitates a comprehensive approach to defining and verifying design correctness. By leveraging SVA, designers can systematically ensure that their designs meet the intended specifications and perform reliably [9].

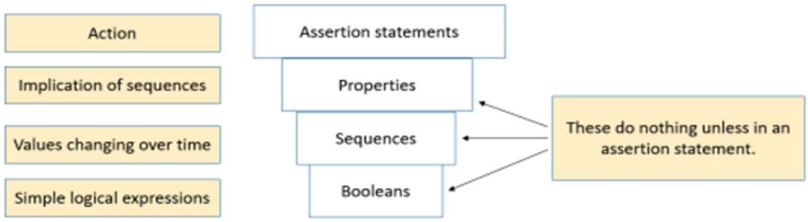


Figure. 3.7 Layers of SVA assertion language [9]

Booleans

Assertions in formal verification can be expressed using straightforward logical Boolean expressions. These expressions might consist of a single logic variable or a more complex Boolean formula, such as "a & b." These Boolean expressions are integral to defining sequences or properties, as depicted in figure 3.7, and are evaluated based on the sampled values of all relevant variables.

The sampled values refer to the state of each variable at the conclusion of each preceding simulation time step. By evaluating these expressions over sampled values, designers can verify that the design behaves as expected throughout its operation. This approach allows for precise monitoring and validation of the design's logical conditions, ensuring that it consistently meets its specified requirements [9].

Sequences

These are the statements with boolean expressions in it that are happening over time. The simplest of sequences are linear in order meaning that they are just a list of finite boolean expressions that occur over linear order to increasing time. The increasing passage of time is defined with a clocking event. Sequences are composed by concatenation which specifies the time delay, using ## operator, from the end of first sequence to beginning of second sequence as shown below

a ##N b It means that signal b shall be true on the Nth clock tick after signal a was true.

Properties

A property in formal verification is a construct that integrates sequences with additional operators to encapsulate the expected behavior of a design, as outlined in the design specifications. Properties serve various roles, such as assumptions, assertions, or coverage specifications, but they do not independently generate results. Instead, they define conditions and behaviors that need to be verified against the design.

```
property reqgnt;
        req |-> s_eventually gnt;
endproperty
```

It states that if there is a request req which is an antecedent then eventually a grant must come for it gnt which is the consequent. The antecedent and consequent in the property are connected via the

implication operators |-> or |=>. The first one (|->) is overlapping meaning that, if there is a match for antecedent then the end point for match is start point for the evaluation of consequent expression. While the second one (|=>) is non-overlapping meaning that, the start point for the evaluation of consequent is one tick after the match for antecedent [9].

Assertion statements

An assertion statement is a tool used to validate the behavior of a system by checking whether certain conditions or properties hold true during its operation. While properties themselves do not produce results, they become actionable when incorporated into assertion statements. These statements utilize specific keywords—assert, assume, or cover—to define the context and purpose of the verification.

Assert: This keyword is used to actively check that a property holds true during the execution of the design. If the property fails, it typically triggers an error or alert, indicating a deviation from the expected behavior.

assert property (req |-> gnt);

Assume: This keyword is used to specify conditions that are assumed to be true for the verification process. It helps in constraining the environment or inputs under which the design is verified, ensuring that the verification focuses on relevant scenarios.

assume property (!req |->! gnt);

Cover: This keyword is used to track whether certain conditions or sequences occur during the execution of the design. It is primarily used for coverage analysis, helping to ensure that all specified scenarios are exercised during testing.

cover property (req |-> gnt);

Assertions in formal verification can be categorized into two main types: immediate and concurrent assertions, each serving distinct purposes and operating under different semantics.

Immediate Assertion Statements: These are straightforward assertions that adhere to simulation event semantics and are executed within procedural blocks. Immediate assertions do not involve clocking or resetting mechanisms and lack support for advanced property operators. Consequently, they are unable to verify conditions that involve the passage of time. Immediate assertions are defined using the assert keyword alone, without the property keyword. An example of an immediate assertion might be:

immediate1: assert (!req && !gnt);

This type of assertion is primarily used for simulation purposes, checking conditions at specific points in time without considering temporal sequences.

Concurrent Assertion Statements: These assertions follow clock semantics, allowing them to describe behaviors that include the passage of time. Concurrent assertions support advanced property statements

that incorporate time intervals, making them suitable for verifying complex logical implementations over time. They utilize both the assert and property keywords in their statements. An example of a concurrent assertion is:

conc1: assert property (a ##2 req |=> gnt);

This statement checks that if condition "a" holds, then after two clock cycles, "req" implies "gnt."

In the context of this thesis work, only concurrent assertion statements were employed to verify the design behavior. This choice reflects the need to capture and validate temporal sequences and conditions that are integral to the design's operation, ensuring comprehensive verification beyond static checks. [8]

3.5 Bug Hunt

Formal verification has been used successfully by a lot of companies to verify complex SOCs and safety-critical designs. The ABCs of formal have been used extensively as described in:

- Assurance: to prove and confirm the correctness of design behavior.
- Bug hunting: to find known bugs or explore unknown bugs in the design.
- Coverage closure: to determine if a coverage statement/bin/element is reachable or unreachable.

Using formal verification to uncover new bugs is emerging as an efficient verification approach when functional simulation regression is stabilized and is not finding as many bugs as before. Traditional formal verification, which starts to explore the design from time 0, is good for early design verification. As more blocks are being integrated, the state space of the design increases exponentially, making traditional formal verification approaches inefficient for bug hunting. As we have observed, complex bugs happen under combinations of events and scenarios [10].

Sophisticated approaches such as waypoints, coverpoints and goal-posting have been proposed and used successfully to find bugs buried deep in the design, especially complex bugs.

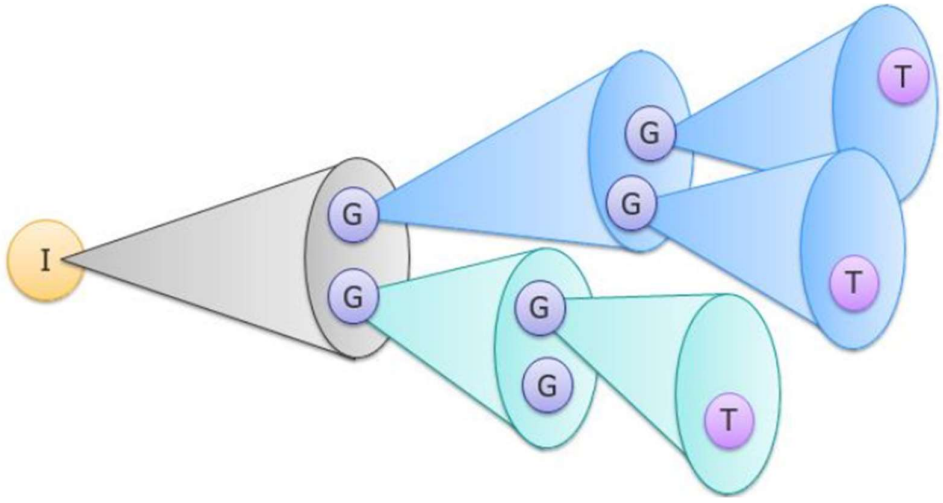


Figure 3.8. Waypoints, coverpoints and goal-posting methodology [10]

The essence of these approaches is that instead of focusing on the targets that are hard to reach, formal verification targets coverpoints, goals, or waypoints that are found deep in formal analysis. Once those

coverpoints are hit, they are used as initial states to explore further into the design, as illustrated in above figure. Using this approach, formal verification can leap from goal to goal, exploring deeper and deeper until the final targets are achieved [10]

In formal verification, rather than beginning with a single initial state, this approach involves selecting intriguing states from simulation traces to initiate the verification process. This method differs from goal-posting, which typically focuses on reaching specific coverpoints or goals to delve deeply into the design. Instead, by utilizing simulation traces, we can identify and start from interesting initial states that are near the final targets.

If functional simulations have already covered the desired coverpoints or goals, these simulation traces provide initial states that are as effective as those used in goal-posting. Additionally, if certain targets require specific preconditions to be met beforehand, this approach allows us to leverage various functional tests to pinpoint different promising starting points for those targets.

Essentially, this strategy capitalizes on the work already accomplished through standard formal property verification (FPV) and initiates the search for bugs as close to the targets as possible. By doing so, it enhances the efficiency of the verification process, focusing efforts on areas most likely to yield valuable insights into the design's behavior and potential issues.

As users of formal verification become more experienced, they increasingly seek greater transparency to better understand how the verification process is performing on their designs. One of the most effective indicators of this performance is formal engine health. This metric serves as a valuable tool for assessing the progress achieved through formal verification.

Formal engine health acts as a matrix that helps in estimating, prioritizing, and monitoring formal verification runs. By evaluating this metric, users can determine the effectiveness of various starting points, identify and eliminate low-quality ones, and prioritize the remaining spots for further investigation and bug hunting.

Continuously monitoring formal engine health during subsequent verification runs allows users to efficiently manage resources. If a particular run is deemed unfruitful based on its engine health, it can be terminated early, freeing up resources for new and potentially more productive runs. This approach not only optimizes the verification process but also ensures that efforts are concentrated on areas most likely to yield meaningful results, enhancing the overall efficiency and effectiveness of formal verification.

The concept of engine health helps us determine if, and to what degree, formal engines are contributing to the progress made on a given set of targets. We define formal engine health with a set of parameters:

1. Formal targets concluded (proven/fired/covered/uncoverable)
2. Sequential depth explored or cone of influence analysed
3. Formal knowledge and engine setting acquired

Targets concluded and sequential depth explored are two well-established metrics that have been used

regularly to determine the overall progress of a formal run.

On the other hand, if we want to understand how formal is doing on an individual property or target, we have to go deeper to examine the cone of influence, and the formal knowledge acquired on those targets. A formal setup may work extremely well on a few targets, but not so well on the others. Cone of influence provides more information about the depth explored per target. Cone of influence provides detailed information on the fan-in logic and their dependence concerning the targets. It highlights the "rocks" in the fan-in logic where formal gets hung up, spending a lot of time analysing. These difficult design elements can be identified, and users can determine whether it is necessary to intervene. Adding cutpoints, abstracting the design elements, or enabling some special engines are a few approaches to enable the "rock" to be analysed more efficiently [10].

A quick and comprehensive two-tier approach is employed to identify intriguing spots for formal verification, focusing on two primary criteria:

1. **Interface Interactions**: Inter-module communication and standard protocol interfaces are notorious for being sources of design issues. Protocol monitors are often used to instrument these on-chip buses and common interfaces, allowing us to pinpoint interesting spots based on activities occurring on these interfaces. Functional simulations generate a substantial number of bus transactions, which formal verification can leverage to explore new and potentially overlooked corner cases. By focusing on these interactions, formal verification can target areas where design flaws are more likely to occur, enhancing the thoroughness of the verification process.
2. **Control and Interrupts**: This criterion encompasses finite state machines (FSMs), bus controllers, memory controllers, flow charts, algorithmic controls, and other elements that represent critical control logic within a design. Due to the complexity of these components, it can be challenging for formal verification to traverse all possible or deeply nested state sequences. Therefore, each new state or transition exercised by simulation presents a potentially interesting spot for formal verification. By examining these areas, formal verification can uncover issues related to control logic that might not be easily accessible through traditional methods, ensuring a more comprehensive evaluation of the design's behavior.
3. **Concurrent events:** These are the arbiters, schedulers, switches, multiplexing logics, etc in a design. The timing and sequence of events are important. Functional simulation may exercise the same sequence of events repetitively. Instead, formal verification can leverage the setup but change the ordering of the events to ensure the design is robust.
4. **Feedback, loops, and counts**: These are the FIFOs, timers, counters, and so forth handling the data transfers, bursting, and computations in a design. Functional simulation exercises the non-stressful situations well. By leveraging the simulation activities, formal verification can explore the stressful corner case scenarios, such as starting, stopping, overflow, underflow, and stalling

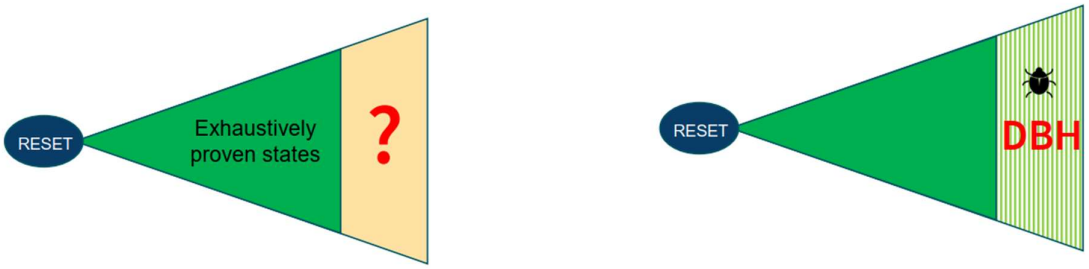


Figure 3.9: Deep Bug Representation [11]

Benefits

1. Can explore states beyond the traditional proof bound, potentially finding new counterexamples or covered results
2. Might produce counterexamples or covered results much faster than the traditional proof
3. Analysis is non-exhaustive, so design complexity becomes less of a problem

Limitations

1. Cannot deliver proven, bound, or unreachable results (except in trivial cases)
2. Analysis is non-exhaustive, so detection of bugs is not guaranteed
3. Cannot guarantee shortest traces; most traces will be non-minimal

3.6 Bug Hunt Modes

1. There are 9 bug hunt modes available, each mode has different technique to explore the given state space to gave results.

2. Each mode can be customized with settings to meet the needs of different designs.

3. Each modes have their defaults engines to run but they can also be configured as per the property types and complexity of RTL.

4. Two or more than two different modes can be used to prove certain sets of properties to get better results**.**

5. Some modes let each engine to be assigned a different value, that's promotes diversity between engines, so that it can start from different initial state point and can explore different spaces, leading to better results.

Figure 3.10: Different Bug Hunt Modes [11]

3.6.1 Formal Mode

1. Regular formal engine might not reach deeper design states due to design complexity.
2. Eliminating some legal behaviours by using over-constraints can help reduce complexity. This might let deeper states to be reached more easily.
3. Only cex/covered results are valid under these conditions, so proven result is not guaranteed.
4. Local over-constraint can be applied so that it did not contaminate other properties state spaces.

3.6.2 Cycle Swarm

BMC Engines are natural bug-hunting engines which perform exhaustive analysis unrolling the design cycle by cycle. By relaxing the settings of these BMC engines, we can reach deeper cycles. This results in a non-exhaustive analysis of each cycle allowing deeper bugs to be detected. Engines can be configured to start on any specific cycle (*First Trace Attempt*) and timeout (*Trace Attempt Time Limit*) can be applied to the cycles for progressing to the next cycle. Cycle Swarm uses BMC engines with relaxed settings for performing non-exhaustive analysis allowing the engines to reach beyond the bound reached by formal. By configuring Cycle Swarm to run with different first trace attempts for more than one engine, it allows each engine to work on independent cycles thereby accelerating the pace to reach deeper cycles.

As the design complexity varies on a cycle-by-cycle basis. Cycle Swarm may have difficulty in exposing bugs in high complexity cycles or reaching deeper cycles. For Example, some early cycles may have high complexity while deeper cycles have low complexity

.

▪ Running Cycle Swarm with a low *Trace Attempt Time Limit* may expose bugs in deep, low complexity cycles but miss closer bugs in high complexity cycles.

▪ Running with a high *Trace Attempt Time Limit* can expose bugs in early, high complexity cycles but may never advance to the deep cycles before the overall time limit expires [11].

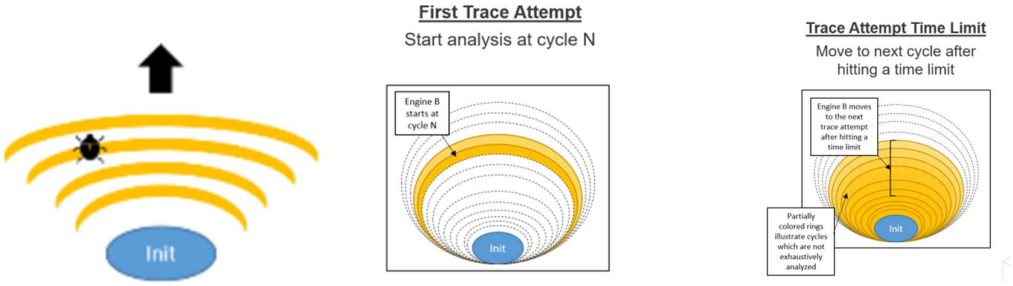


Figure 3.11: Cycle Swarm Mode [11, 12]

3.6.3 Bound Swarm

Design complexity varies on a cycle-by-cycle basis, Sometimes early or initials cycles might have high complexity with deeper cycles have low complexity and vice versa. So if we run it using cycle swarm, then there is high chances that with low trace attempt time limit it might expose bugs in deep, low-complexity cycles but miss closer bugs in high-complexity cycles. Likewise, running with a high time limit can expose bugs in early cycles, but can't in deeper with high complexity cycles.

Bound Swarm is an iterative analysis technique that runs Cycle Swarm over a bounded range of cycles starting from *First Trace Attempt* cycle to *Max Trace Length* cycle. By increasing the *Trace Attempt Time Limit* by a set factor (*Trace Attempt Time Limit Factor*) for each iteration, allowing BMC engines to run iterations starting with low effort to high effort at incremental iterations. This way bugs in low complexity cycles can be identified first and then bugs residing in the complex cycle can be found next [11].

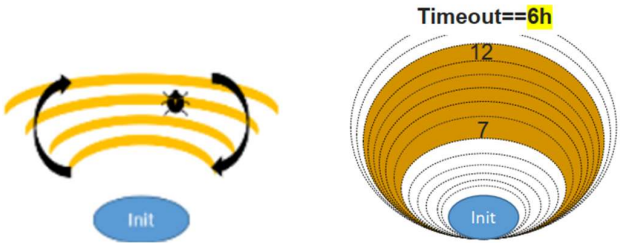


Figure 3.12: Bound Swarm Mode [11]

3.6.4 State Swarm

State Swarm is a bug hunting mode which makes use of engine L to perform formal search from the initial state until a cex/cover trace is found. Next, it starts a new search using a cycle along the previous cex/cover trace as a new initial state for performing formal search to find a new trace. This way the operation continues with different paths being explored by multiple engines. The efficiency of State Swarm relies on good set of helper covers [11].
But there are certain problems with State Swarm,

1. State swarm cannot analyze liveness asserts.
2. State swarm can miss bugs due to certain engine L settings such as first trace attempt, max segment length/time etc.

Without an appropriate set of helper covers, state swarm will not be effective, Covers for state swarm need to be:
Interesting: To expose interesting design behaviors
Not too close to each other: So that engine L can spent time analyzing asserts at each stage
Not too far from each other: So that engine L can make the transition before time/depth limits
Diverse: So that the same scenario can be hit in multiple ways

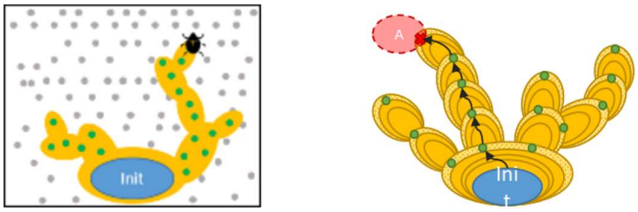


Figure 3.13: State Swarm Mode [11, 12]

3.6.5 Trace Swarm

Background: Existing trace can be a good basis for bug hunting, cover traces might already reached deeper states space, cex traces often leads to new cex for other property. And also to overcome demerits of state swarm, trace swarm is used.

Trace Swarm is a bug hunting mode which runs formal from any trace using any engine. Traces which have already reached deeper states can be used for finding deep hidden bugs [11].

Supports two run modes:
Static :- from the existing traces in the property table
Dynamic :- from traces as they are produced by another hunt mode as it runs

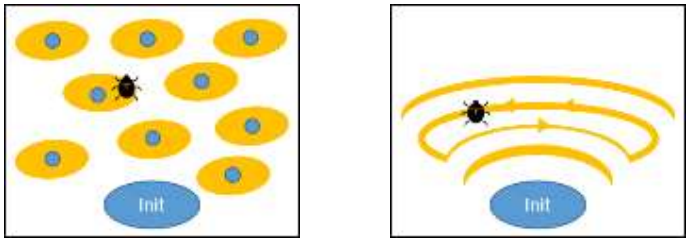


Figure 3.14: Trace Swarm Mode and Loop Swarm Mode [12]

3.6.6 Loop Swarm

Liveness properties are complex problems where bugs can be difficult to detect. A liveness cex contains a fixed stem sequence followed by a looping sequence – both can be of arbitrary length. During regular formal analysis, engines simultaneously consider all possible stem & loop lengths. Restricting the loop length to a fixed constant typically enables a cex with that length (if it exists) to be detected much faster. However, doing this prevents cex with other loop lengths from being detected. Loop Swarm uses multiple engine jobs to simultaneously consider different loop lengths. By increasing the number of engine jobs, the probability of bug detection is much faster compared to traditional formal proof [12].

3.6.7 Simulation Mode

In Simulation, it generated random seed value as initial point and explore state by constructing traces. Trace length can be configured on the user need, as with lower traces length it frequent restarts thus, more diversity in cycles close to reset. With higher trace length it will explore more deeper state.

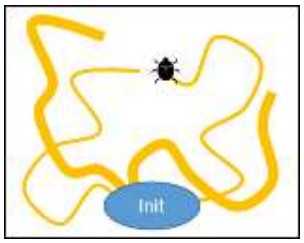


Figure 3.15: Simulation Mode [12]

3.6.8 Trace Search

Existing traces (Cover/Assert) can be a good basis for bug hunting. It's possible there may be bugs in the local vicinity of a trace, even if the trace itself is bug free. With trace swarm, proofs can be run from multiple cycles (search points) along a trace to provide a uniform search to a specified depth around the trace. Trace Search helps in finding bugs which are in the vicinity of the trace.

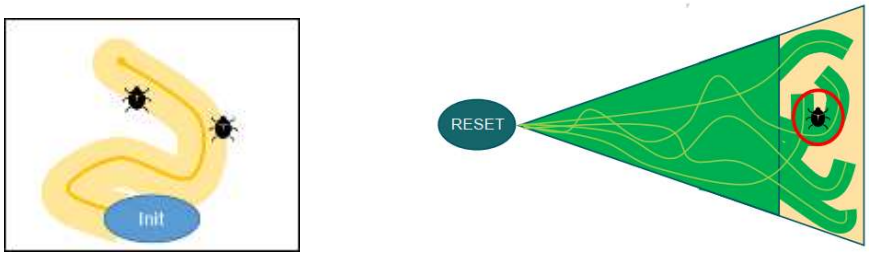


Figure 3.16: Trace Search Mode [11]

3.6.9 Guide Point

Design knowledge can be used to write a set of ordered cover properties that lead to a desired design state :
– Given an ordered set of covers, formal can be used to find paths that link them
– Complexity can be kept low if covers are spaced accordingly
When the desired design state is reached, formal analysis of target properties can be done.
Provide an automated way to construct a trace that joins a specified sequence of covers. Cover properties must be predefined and specified in an exact order o Covers are expected to take the design to an interesting state. The main target properties will only be analyzed once in the final state [12].

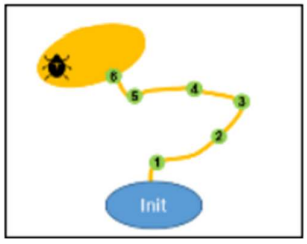


Figure 3.17: Guide Point Mode [12]

CHAPTER 4

Use of Bug Hunt Technique on DUT

4.1 Bug Hunt Planning

For the Bug Hunt, first we find out the last reachable point to get the proven result or counterexample. To do this these are the steps :

- Find the properties which have bounded proof or did not converge even after some time.
- Create the new task by taking same types of properties and also include Constraints related to them in that task.
- Generate covers for all counters for each count value.
- Prove the covers
- Define hunt mode with different strategies that suits your specifications like memory etc.
- Run the Hunt command.
- Initially run it for 4 hours and checks it proof converges or not. If not, then increase the run time.
- Try for different modes with different engines

4.2 Bug Hunt Implementation

A new task has been created by name bug_hunt in which those properties are added which do not converge using normal Formal verification.

The bug hunt is fired from the last covered counter max value. The property gave result in limited time means they converged.

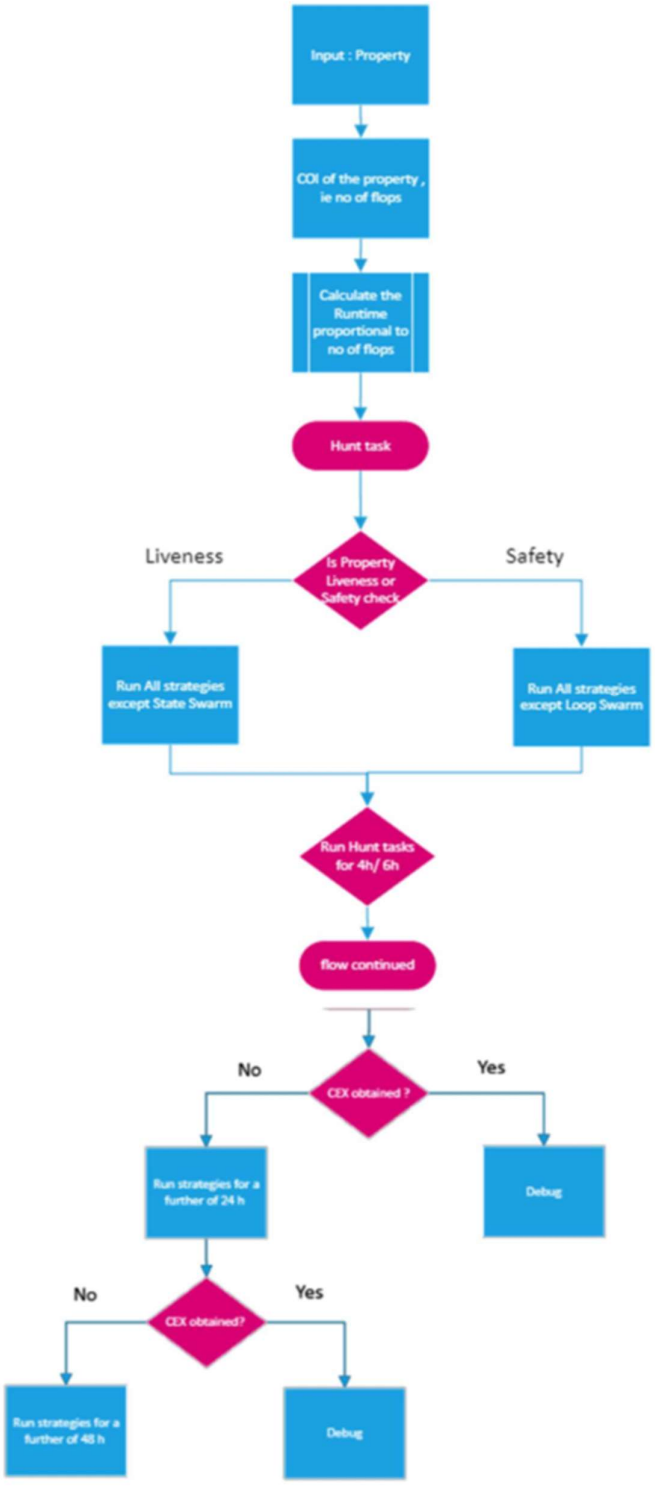
The implementation of serializer design using normal FPV. In this some properties did not converges. We will take this property and add them into new task and run bug hunt on task

Status	Type	Name	Bound
Proven	Assert	pp.thr_mem.mem.no_simultaneous_writes_to_same_address	Infinite
Proven	Assert	pp.fv_env.route_agent.AST_M_auto	Infinite
Proven	Assert	pp.fv_env.intr_agent.intr_list_init_done.AST_status_fell_implies_clear	Infinite
Proven	Assert	pp.fv_env.intr_agent.intr_list_init_done.AST_interrupt_stable	Infinite
Proven	Assert	pp.fv_env.intr_agent.intr_list_empty.AST_clear_implies_status_low	Infinite
Proven	Assert	pp.fv_env.intr_agent.AST_interrupt	Infinite
Undetermined	Assert	pp.payload_mem.mem.no_simultaneous_writes_to_same_address	31
Undetermined	Assert	pp.fv_env.eg_agent.AST_S_sop_ok	33
Undetermined	Assert	pp.fv_env.eg_agent.AST_S_resp_eventually	31
Undetermined	Assert	pp.fv_env.intr_agent.intr_list_almost_full.genblk1.AST_trigger_implies_status_rose	33
Undetermined	Assert	pp.fv_env.intr_agent.intr_thr_empty.AST_status_rose_implies_trigger	34
Undetermined	Assert	pp.fv_env.intr_agent.intr_thr_empty.genblk1.AST_trigger_implies_status_rose	45
Undetermined	Assert	pp.fv_env.intr_agent.intr_thr_almost_empty.AST_interrupt_stable	36
Undetermined	Assert	pp.fv_env.fl_chk.AST_full_ok	33
Undetermined	Assert	pp.fv_env.ig_route_sb.sb.genblk6.core.genblk5.genblk3.genblk1.latency_check	33
Undetermined	Assert	pp.fv_env.ig_route_sb.sb.genblk6.core.genblk5.genblk1.data_integrity	33

Figure 4.1: List of Undetermined Properties [FV Tools]

Flow Chart :

Flow chart of Bug hunt flow. The script has been written in this flow.



4.3 Outputs

There are some common settings which we used to define the mode of run.

Tag: Each hunt run has a unique tag associated with it. If we don't specify the tag manually, tool will auto generate the tag. Tags can be used for reporting or searching in the log. For example, to search the mode settings.

Strategy: Specifies the Strategy name associated with the hunt run. A strategy is a specific configuration of a general bug hunting mode. This name will match one of the strategies which are defined under the Hunt Manager list of strategies.

Mode: The bug hunting mode that the strategy is based on.

Time Limit: Specifies the total run time for the hunt run.

Engine Mode: The selected engine is Bm, a multi-property engine that can run proofs on multiple properties in parallel.

Max Jobs: The maximum number of instances of engine that can be run in parallel to distribute the workload across multiple cycles.

First Trace Attempt: The first trace attempt setting specifies the starting cycle from which an engine begins searching for a trace. If the first trace attempt value is not explicitly specified, the tool defaults to using the lowest minimum bound of all undetermined properties as the starting value for first trace attempt and used for all the target properties.

Max Trace Length: The maximum bound the trace can be reached from its starting point.

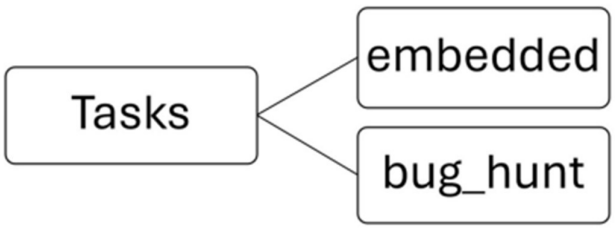
Seed: A hexadecimal value randomly selected by the tool. Based on this seed value, parameters specified as distributions (e.g., trace attempt time limit) take random values.

No Covers Traces: Specifies that no cover traces will be generated. Use this switch to avoid the overhead of generating and storing cover traces. This will save the memory space.

Trace Id: To specify a trace other than the default. The default property trace is the last shortest trace.

These are the common switch that are used. Define hunt mode with different strategies that suits specifications like memory, complexity of design, and the type of property etc. Some modes can only be used for liveness property.

A new task can only be created by consisting only those property which are unproven, while creating task keep the design constraints as in original task.



Cycle Swarm

```
INFO (IHT002): Settings used for this hunt:
    tag                          = <cycle_swarm>:0
    strategy                     = <cycle_swarm>
    mode                         = cycle_swarm
    time_limit                   = 1200s
    max_jobs                     = 6
    engine_mode                  = Bm
    first_trace_attempt          = 31 32 33 34 35 36
    max_trace_length             = 0 0 0 0 0 0
    deeper_cycles_earlier        = 0
    trace_attempt_time_limit     = 585 469 352 422 541 338
    seed                         = 67d398ec
```

Figure 4.2: Strategy for Cycle swarm [FV Tools]

Jasper FPV runs formal proof for a definite time (2 mins). Once the time limit is reached the proof stops. At this point, some properties may remain undetermined. In a practical scenario, FPV proof is run for a longer duration. When the proof times out there could be properties which remain undetermined. Bug hunting is then run to identify potential bugs at deeper cycles.

Trace Attempt Time Limit: Each engine has a maximum time limit (defined by trace_attempt_time_limit) to work on a particular cycle. For example, the first engine job searches for a trace starting from cycle 31 with a time limit of 401 second. Once the time limit is reached or if the cycle is completely analyzed within the time limit, the engine advances to the next cycle by incrementing the current cycle number by the value of max_jobs (e.g., 31 + 6 = 37). This continues until all cycles in the range have been analyzed. The engines stop processing once the total time limit gets expired.

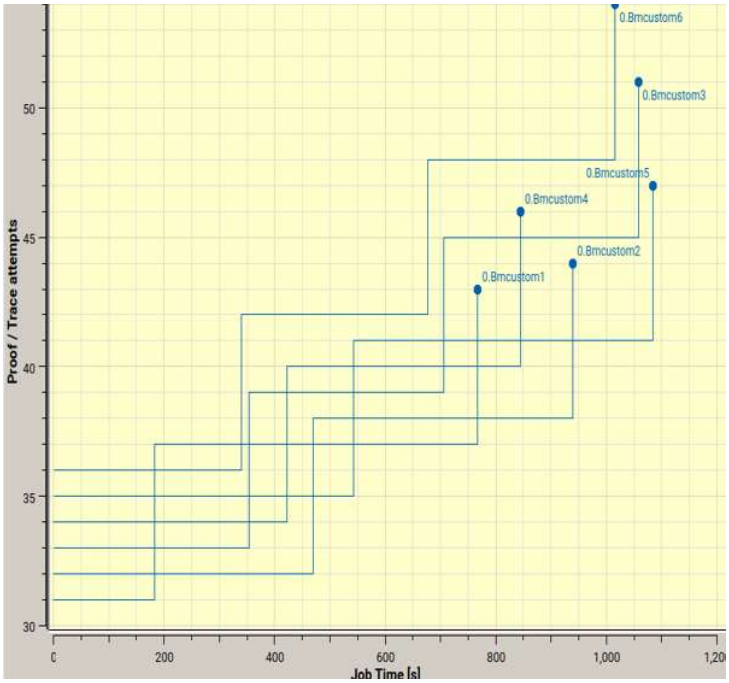


Figure 4.3: Proof grid Manager of task [FV Tools]

In the above image, the engine successfully analyzes the first attempt cycle 31 within the time limit and then moves to the next cycle, determined by adding (*max_jobs = 6*) to the current cycle (31 + 6 = 37). In this cycle the time limit expires, the engine advances to the next cycle (37 + 6 = 43). This behavior is consistent across all engine jobs.

Bound Swarm

```
INFO (IHT002): Settings used for this hunt:
    tag                          = <bound_swarm>:0
    strategy                     = <bound_swarm>
    mode                         = bound_swarm
    time_limit                   = 1200s
    max_jobs                     = 6
    engine_mode                  = Bm
    first_trace_attempt          = 31 48 65 82 99 116
    max_trace_length             = 47 64 81 98 115 131
    deeper_cycles_earlier        = 1
    trace_attempt_time_limit     = 1 1 1 1 1 1
    seed                         = 1
```

Figure 4.4: Strategy of Bound Swarm

Deeper cycles earlier: "deeper_cycles_earlier" divides trace attempts into smaller groups allowing each engine to work on cycles within the specific group. This helps in exploring deeper cycles earlier compared to the default behavior. The division into small groups is done by considering the start cycle (First Trace Attempt), end cycle (Max Trace Length) and Max Jobs. For Example, If Max Jobs is 6, First Trace Attempt is 31 and Max Trace Length is 100 (relative end cycle will be 31+ 100 =131), the range of cycles between start (31) and end cycles (131) will be divided into smaller groups of 16 cycles. Now, each engine instance will work on the smaller group with first trace attempts starting at deeper cycles.

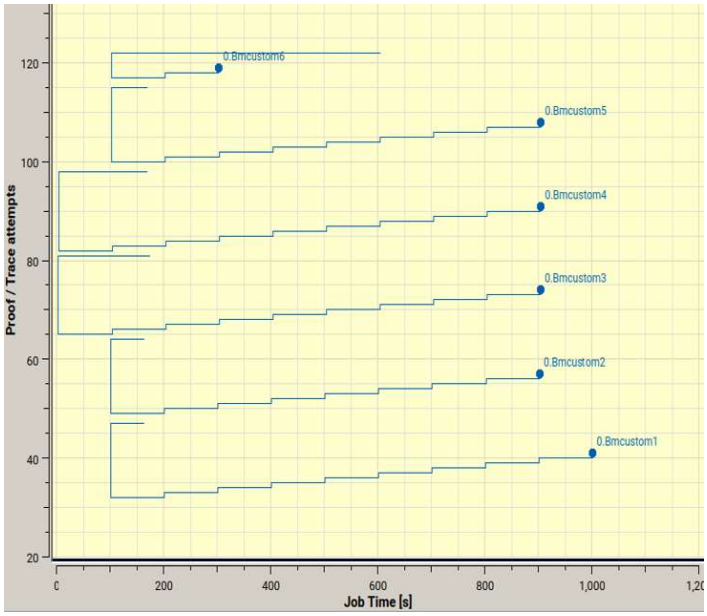


Figure 4.5: Proof Grid Manager of Task [FV Tools]

During the iteration #1, the engine can spend a maximum of *trace_attempt_time_limit*, set to 1 second (1 x 10^0), to attempt tracing on each cycle, starting from the first attempt at cycle 31 (*first_trace_attempt* = 31) and continuing up to a maximum trace length of 47 cycles (*max_trace_length* = 47)

During the iteration #2, the engine can spend a maximum of *trace_attempt_time_limit* seconds, set to 10 seconds (1 x 10^1), on each cycle.

During the iteration #3, the engine can spend a maximum of *trace_attempt_time_limit* seconds, set to 100 seconds (1 x 10^2), on each cycle.

This way the iteration continues till the total time limit (20 minutes) expires.

State Swarm

State Swarm do formal search from initial state until a CEX/cover is found using L engine. Then it used that point as a starting point for further exploration. Goods results depend on effective set of helper covers.

```
INFO (IHT002): Settings used for this hunt:
    tag                           = <state_swarm>:0
    strategy                      = <state_swarm>
    mode                          = state_swarm
    time_limit                    = 300s
    max_jobs                      = 20
    max_trace_length              = 0
    engine_mode                   = L
    no_cover_traces               = 1
    start_state_count             = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    tail_length                   = 2 1 2 1 1 1 2 4 1 2 2 1 3 1 1 5 2 2 1 1
    first_trace_attempt           = 13 12 10 15 4 9 3 7 8 7 8 6 5 9 5 11 13 6 4 14
    segment_time_limit            = 367s 346s 288s 414s 462s 589s 444s 221s 572s 535s 236s 193s 482s 259s 154s 324s 123s 500s 148s 394s
    max_segment_length            = 172 161 145 205 228 278 214 105 276 260 116 91 235 133 80 149 238 247 64 187
    random_diversification_factor = 0.354 0.042 0.476 0.035 0.094 0.071 0.304 0.18 0.125 0.393 0.341 0.139 0.535 0.137 0.105 0.212 0.253 0.434 0.081 0.063
    seed                          = 685248eb
```

Figure 4.6.1: Strategy of State Swarm

Max Trace Length: To specify the limit for the length of traces the tool searches for. specify a natural value for state swarm.

First Trace Attempt: To specify the cycle at which the engine will make its first trace attempt. We can specify a list of values or a distribution as the argument.

Start State Count: To specify how many parallel start states each engine L job will use to search for new traces. Each different helper cover trace is a new start state. It can take a single value, a list of values, or a distribution as the argument. The tool uses the default single value 1, that is, 1 start state for each engine L job.

Tail length: To specify the starting point for a new search that is N cycles before the end of found traces. We can specify a list of values or a distribution as the argument. If you do not specify this switch, the default distribution is {50%:[1] 30%:[2] 20%:[3..5]}.

Segment Time Limit: To specify the amount of time the tool spends searching from each start state.

Max Segment Length: To specify the number of trace attempts the tool makes from each start state. It can take a list of values or a distribution as the argument.

Random Diversification Factor: To specify the random diversification value used by engine L. A high value gives more diversification, but potentially, much slower progress. It can take a list of values or a distribution as the argument. The default distribution is {50%:[0.03..0.15] 45%:[0.15..0.50] 5%:[0.50..0.90]}.

```
INFO (IPF031): Settings used for proof thread 1:
    orchestration                 = off
    time_limit                    = 300s
    per_property_time_limit       = 1s * 10 ^ scan
    engine_mode                   = L
    proofgrid_per_engine_max_jobs = 20
    proofgrid_max_jobs            = 20
    proofgrid_engineL_max_jobs    = 20
    max engine jobs               = 20*L, total 20 (max 20)
    proofgrid_mode                = local
    proofgrid_restarts            = 10
    from_property                 = <embedded>::pp.fv_env.intr_agent.intr_list_almost_full.AST_status_fell_implies_clear:witness1
    from_trace                    = 1
    from_cycle                    = 28
```

Figure 4.6.2: Strategy of State Swarm from the helper covers provided

If no trace is detected within limits, the search continues from a previous start state. That's why covers should be diverse and interesting means exploring corner stage scenarios.

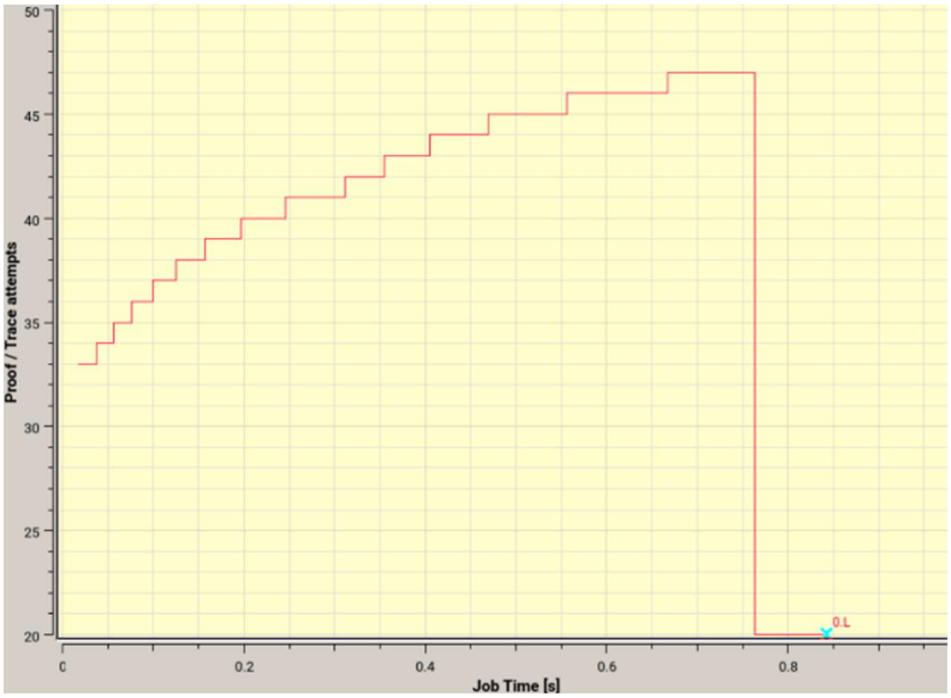


Figure 4.7: Proof Grid Manager of Task [FV Tools]

In the Proof Grid Manager, A counter example is found at cycle 47 by engine L. In the waveform it is clearly visible that the proof start from helper covers bound not from initial point.

Trace Swarm

Trace Swarm is used where state swarm gives limitation like states swarm cannot analyze liveness asserts and it can also miss bugs due to engine setting also.
It can run in two ways either from existing trace or from trace generated by it while proving property.

```
INFO (IHT002): Settings used for this hunt:
    tag                 = <trace_swarm>:1
    strategy            = <trace_swarm>
    mode                = trace_swarm
    time_limit          = 300s
    max_jobs            = 20
    max_trace_length    = 0
    engine_mode         = B Ht
    per_trace_time_limit = 60s
```

Figure 4.8: Strategy of Trace Swarm

Max Trace Length: To specify the limit for the length of traces the tool searches for. specify a natural value for trace swarm.

Per Trace Time Limit: To specify the time limit for hunting from each trace. If value is not specify for this switch, the default value is the "time limit" divided by 10, assuming 1m as the lower limit and 20m as the upper limit

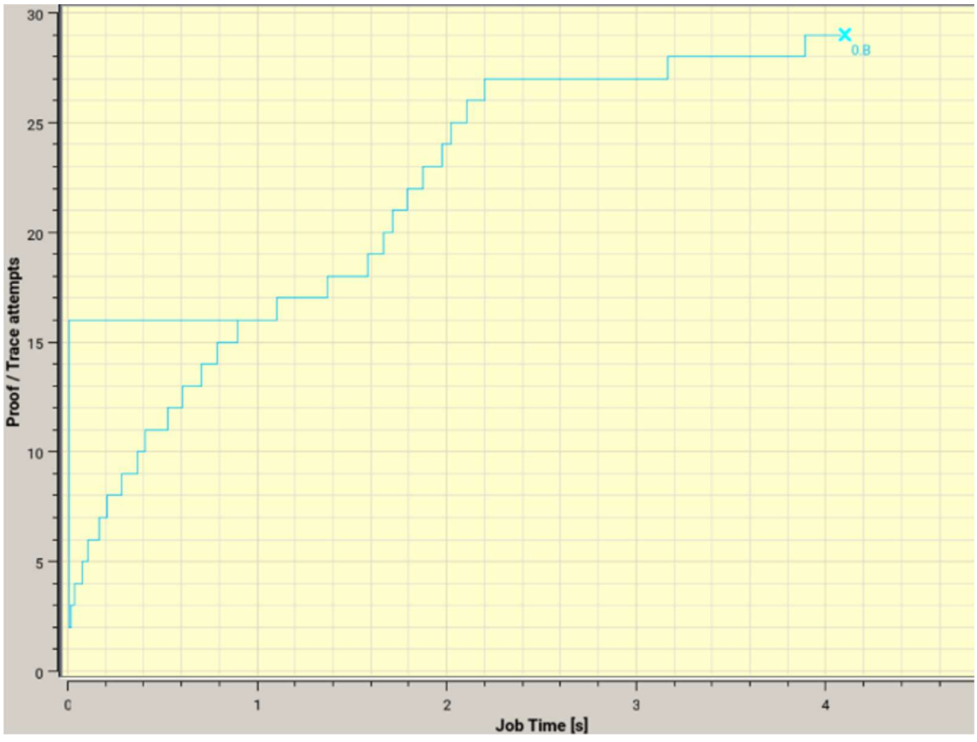


Figure 4.9: Proof Grid Manager of Task [FV Tools]

From the waveform, the proof start from last trace provided in strategy not from initial point. As this property is liveness, so engine tries in two directions simultaneously and after some point it converges and explore till counter example is found.

Loop Swarm

```
INFO (IHT002): Settings used for this hunt:
    tag                 = <loop_swarm>:0
    strategy            = <loop_swarm>
    mode                = loop_swarm
    time_limit          = 300s
    max_jobs            = 6
    per_engine_max_jobs = 1
    max_trace_length    = 0
    engine_mode         = B
    per_engine_max_jobs = 1 1 1 1 1 1
    loop_length         = 1 2 3 4 5 6
    seed                = 6850db93
```

Figure 4.10: Strategy of Loop Swarm

Loop length: Specifies the loop length each engine job would be running. By default, loop length is distributed in increments of 1 across different engine jobs.

Per engine max jobs: Specifies the number of properties for which proof can be run in parallel. By default, it is set to 1.

With max jobs equal to 6, there are 6 engine jobs that run in parallel, each working on different loop length.

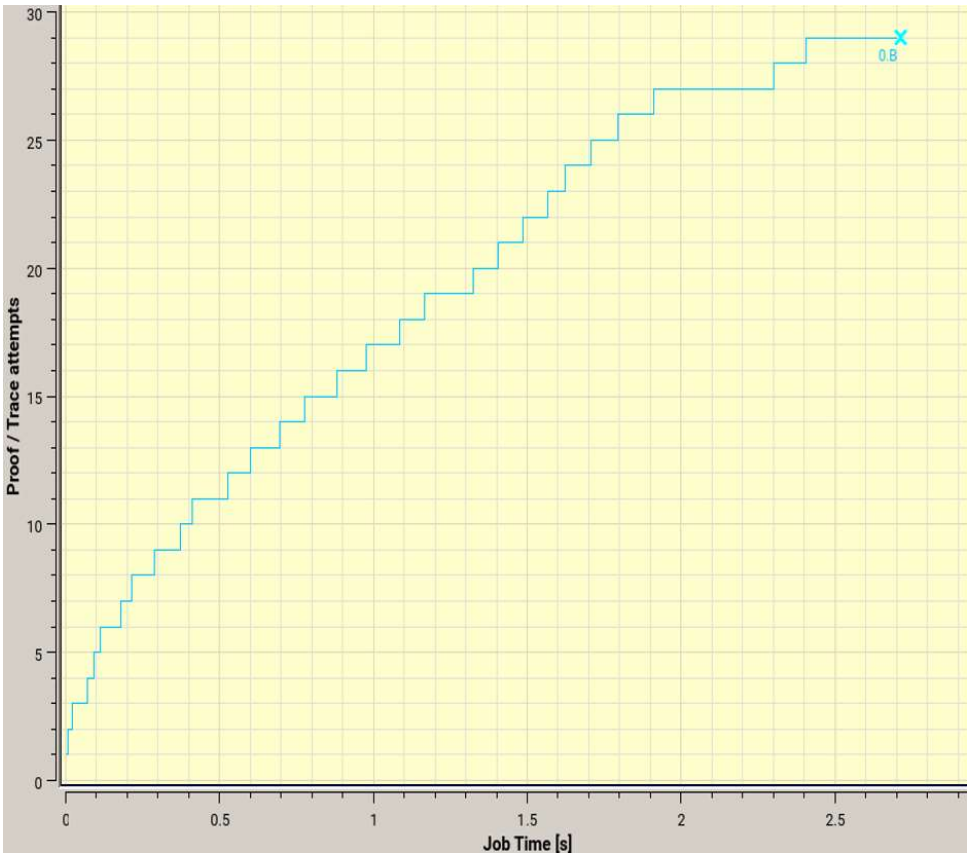


Figure 4.11: Proof Grid Manager of Task [FV Tools]

This proof thread will run Loop Swarm on a specific loop length of 2. From the proof messages, that Engine B is running formal analysis to find the trace. The property in red mark shows that this property (liveness) fails at 29 bound (cycles).

Trace Search

In this mode, a existing trace that can either be proven assert or reached cover used as a start point for run. In this way a bugs in the local vicinity of a trace used as starting point can be found easily and fast.

```
[NFO (IHT002): Settings used for this hunt:
    tag                 = <trace_search>:0
    strategy            = <trace_search>
    mode                = trace_search
    time_limit          = 120s
    max_jobs            = 20
    max_trace_length    = 20
    engine_mode         = B Bm
    interval_cycles     = 10
    target_depth        = 10
```

Figure 4.12.1: Strategy of Trace Search

Interval Cycles: Specifies the number of cycles between subsequent search points. By default, tool sets it equal to target depth.

Target Depth: Specifies the target depth to search from any point along the trace. By default, tool sets it to 10.

Previous analysis revealed a minimum bound of 33 for the undetermined assert property. Notably, the cover trace we provided to Trace Search mode had a bound of 45. When we execute Trace Search on this trace, it will launch proofs at various search points along the trace, specifically at intervals of 10 cycles as determined by interval_cycle setting and explore up to target depth.

This approach also enabled the proof to run beyond the proof bound attained by regular proof, effectively scanning the vicinity of the trace for potential bugs.

1) 1st proof starts at cycle 31, covering target depth of 10 for cycles 31-40.

```
INFO (IPF031): Settings used for proof thread 2:
    orchestration                   = on
    time_limit                      = 120s
    per_property_time_limit         = 1s * 10 ^ scan
    max_trace_length                = 20
    engine_mode                     = B Bm
    proofgrid_per_engine_max_jobs   = 1
    proofgrid_max_jobs              = 2
    max engine jobs                 = auto (max 2)
    proofgrid_mode                  = local
    proofgrid_restarts              = 10
    from_property                   = <embedded>::pp.fv_env.intr_agent.intr_list_almost_empty.AST_status_rose_implies_trigger:witness1
    from_trace                      = 94
    from_cycle                      = 31
```

Figure 4.12.2: Strategy of Trace Search for cycle 31-40

2) 2nd proof starts at cycle 41, covering target depth of 10 for cycles 41-50.

```
INFO (IPF031): Settings used for proof thread 3:
    orchestration                   = on
    time_limit                      = 120s
    per_property_time_limit         = 1s * 10 ^ scan
    max_trace_length                = 20
    engine_mode                     = B Bm
    proofgrid_per_engine_max_jobs   = 1
    proofgrid_max_jobs              = 2
    max engine jobs                 = auto (max 2)
    proofgrid_mode                  = local
    proofgrid_restarts              = 10
    from_property                   = <embedded>::pp.fv_env.intr_agent.intr_list_almost_empty.AST_status_rose_implies_trigger:witness1
    from_trace                      = 94
    from_cycle                      = 41
```

Figure 4.12.3: Strategy of Trace Search for cycle 31-50

3) 3rd proof starts at cycle 45 (last cycle of trace), covering a target depth of 10 for cycles 45 -54.

```
INFO (IPF031): Settings used for proof thread 4:
    orchestration                       = on
    time_limit                          = 120s
    per_property_time_limit             = 1s * 10 ^ scan
    max_trace_length                    = 10
    engine_mode                         = B Bm
    proofgrid_per_engine_max_jobs       = 1
    proofgrid_max_jobs                  = 2
    max engine jobs                     = auto (max 2)
    proofgrid_mode                      = local
    proofgrid_restarts                  = 10
    from_property                       = <embedded>::pp.fv_env.intr_agent.intr_list_almost_empty.AST_status_rose_implies_trigger:witnessl
    from_trace                          = 94
    from_cycle                          = 45
```

Figure 4.12.4: Strategy of Trace Search for cycle 45-50

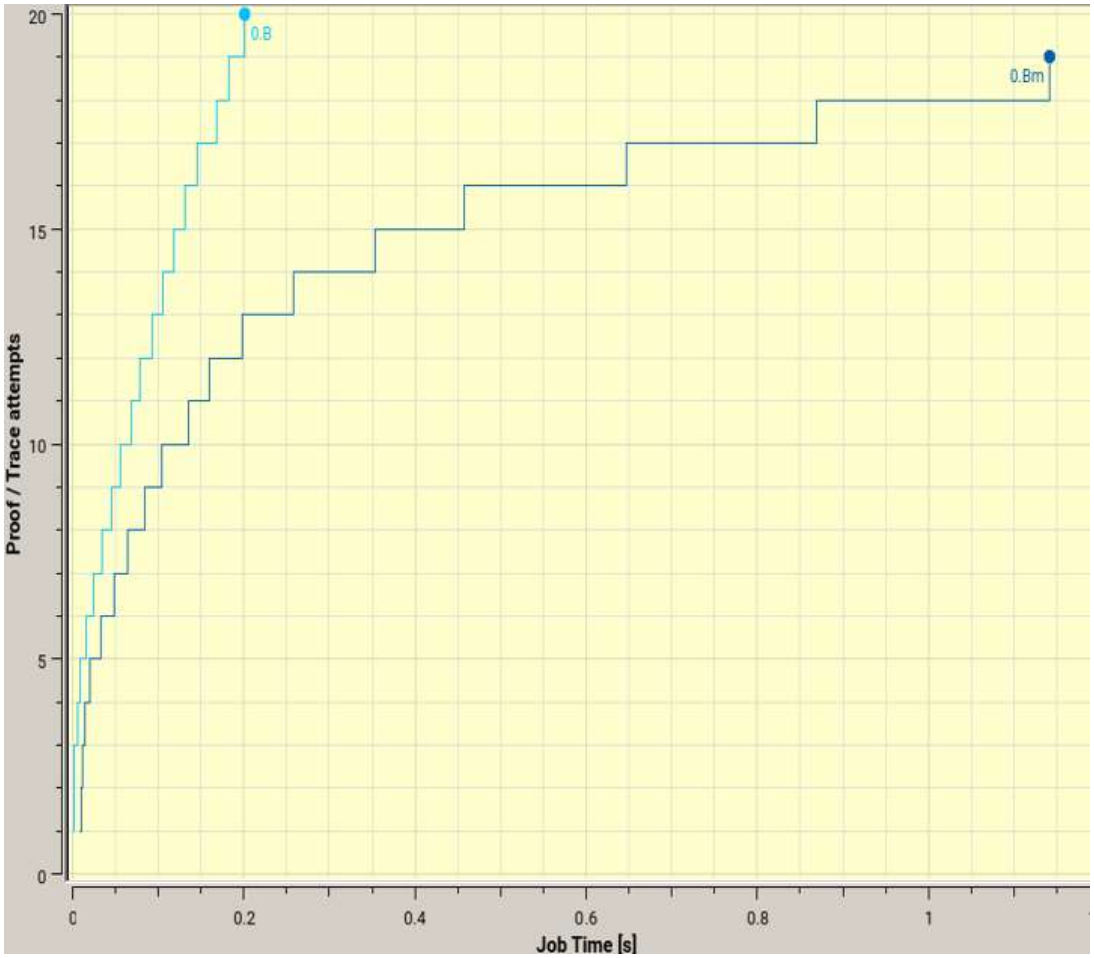


Figure 4.13: Proof Grid Manager of Task [FV Tools]

On dive deeper into proof grid manager, In the Proof Manager (PGM), locate the proof thread that produced a result. Identify and select the specific engine job that successfully found the result. Upon selection, examine the proof messages displayed on the right-hand side. These messages indicate that Trace Search is running formal up to a target depth and has successfully found a cex. A counter example is found at 45 cycles in thread 8 by engine B. Also other engines defined in strategy of mode are also running parallel to prove.

Guide Point

For Guide Pointing analysis, we have created a target assertion to check for the absence of FIFO underflow. We have written a set of cover properties that needs to be hit in a specific order. This order has been defined to ensure that all necessary conditions are met before the target assertion is checked, allowing for directed bug hunting.

The order is as follows:

a. **Initial condition**: pp.fv_env.ig_agent.AST_M_sop_fell:witness1
b. **Initialization completion**
   pp.fv_env.intr_agent.intr_list_init_done.AST_clear_implies_status_low:witness1
c. **Almost full trigger**: pp.fv_env.intr_agent.intr_list_almost_full.COV_trigger_then_clear
d. **Status update**:
   pp.fv_env.intr_agent.intr_list_almost_full.AST_status_fell_implies_clear:witness1

```
INFO (IHT002): Settings used for this hunt:
    tag             = GP:0
    strategy        = GP
    mode            = guidepoint
    time_limit      = 300s
    max_jobs        = 40
    max_trace_length= 0
    engine_mode     = Hp Ht B L
    path            = pp.fv_env.ig_agent.AST_M_sop_fell:witness1 pp.fv_env.intr_agent.intr_list_init_done.AST_clear_implies_status_low:witness1 pp.fv_env.intr_agent.intr_list_almost_full.COV_trigger_then_clear
pp.fv_env.intr_agent.intr_list_almost_full.AST_status_fell_implies_clear:witness1
```

Figure 4.14.1: Strategy of Guide Point

Path: Specifies the ordered set of cover paths that needs to be hit before exploring the target.

The first proof starts from initial state and hits a first cover.

```
INFO (IPF031): Settings used for proof thread 2:
    orchestration                       = off
    time_limit                          = 300s
    per_property_time_limit             = 1s * 10 ^ scan
    engine_mode                         = Hp Ht B L
    proofgrid_per_engine_max_jobs       = 1
    proofgrid_max_jobs                  = 40
    max engine jobs                     = Hp Ht B L, total 4 (max 40)
    proofgrid_mode                      = local
    proofgrid_restarts                  = 10
    from_property                       = <embedded>::pp.fv_env.ig_agent.AST_M_sop_fell:witness1
    from_trace                          = 1
    from_cycle                          = 2
```

Figure 4.14.2: Strategy of Guide Point Covers

The second proof starts from the 1st cover state to hit the second cover.

```
INFO (IPF031): Settings used for proof thread 3:
    orchestration                       = off
    time_limit                          = 300s
    per_property_time_limit             = 1s * 10 ^ scan
    engine_mode                         = Hp Ht B L
    proofgrid_per_engine_max_jobs       = 1
    proofgrid_max_jobs                  = 40
    max engine jobs                     = Hp Ht B L, total 4 (max 40)
    proofgrid_mode                      = local
    proofgrid_restarts                  = 10
    from_property                       = <embedded>::pp.fv_env.intr_agent.intr_list_init_done.AST_clear_implies_status_low:witness1
    from_trace                          = 3
    from_cycle                          = 10
```

Figure 4.14.3: Strategy of Guide Point Covers

The third proof starts from the 2nd cover state to hit the final cover.

```
INFO (IPF031): Settings used for proof thread 18:
    orchestration                       = off
    time_limit                          = 300s
    per_property_time_limit             = 1s * 10 ^ scan
    engine_mode                         = Hp Ht B L
    proofgrid_per_engine_max_jobs       = 1
    proofgrid_max_jobs                  = 40
    max engine jobs                     = Hp Ht B L, total 4 (max 40)
    proofgrid_mode                      = local
    proofgrid_restarts                  = 10
    from_property                       = <embedded>::pp.fv_env.intr_agent.intr_list_almost_full.COV_trigger_then_clear
    from_trace                          = 16
    from_cycle                          = 23
```

Figure 4.14.4: Strategy of Guide Point Covers

The final proof then starts from the final cover state to find bugs around the target. Once the counter example is found it will stop for that property.

```
INFO (IPF031): Settings used for proof thread 11:
    orchestration                       = off
    time_limit                          = 300s
    per_property_time_limit             = 1s * 10 ^ scan
    engine_mode                         = Hp Ht B L
    proofgrid_per_engine_max_jobs       = 1
    proofgrid_max_jobs                  = 40
    max engine jobs                     = Hp Ht B L, total 4 (max 40)
    proofgrid_mode                      = local
    proofgrid_restarts                  = 10
    from_property                       = <embedded>::pp.fv_env.intr_agent.intr_list_almost_full.AST_status_fell_implies_clear:witness1
    from_trace                          = 10
    from_cycle                          = 28
```

Figure 4.14.5: Strategy of Guide Point Covers

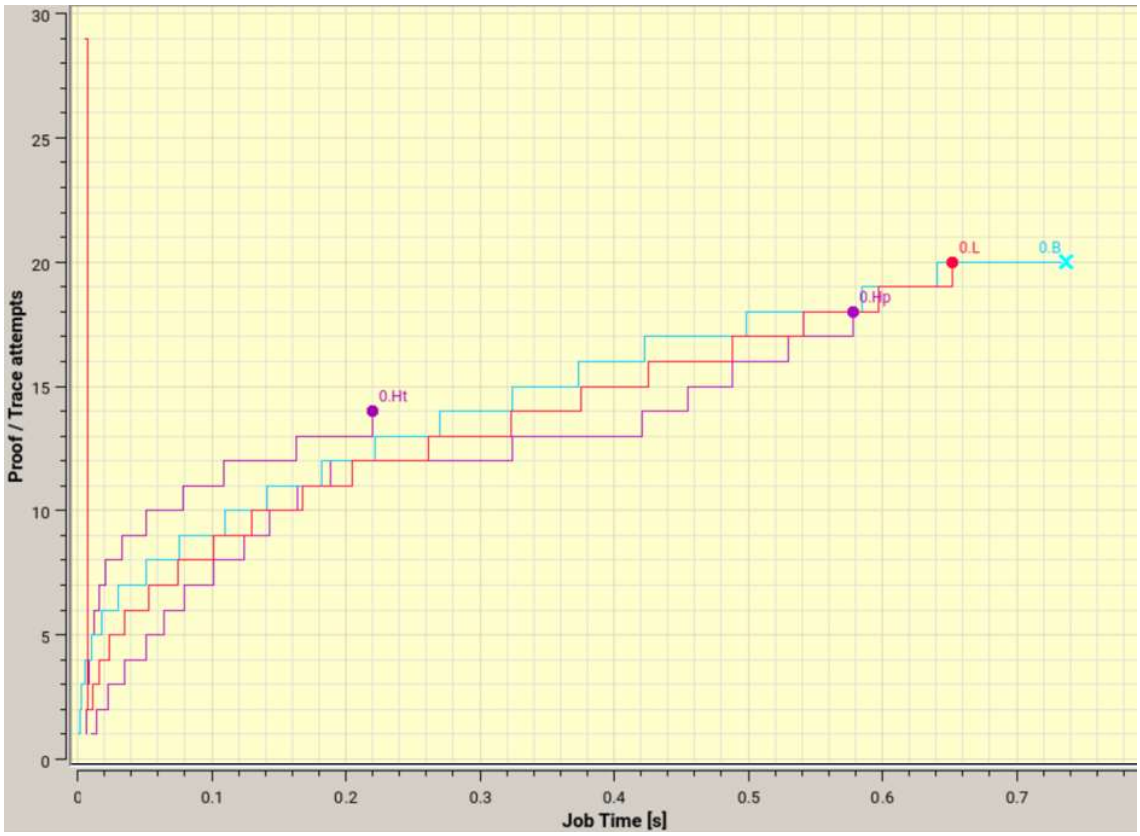


Figure 4.15: Proof Grid Manager of Task [FV Tools]

On proof grid waveform, it is reflected that in thread 11 a counter example is found at cycle 47 by engine. It is also noted that there are many engines run in parallel, the first engine which found a counter example reflected on property table with its name

CHAPTER 5

Conclusion and Future Scope

5.1 Conclusion

We strongly advocate for the integration of the Bug Hunt technique with formal methodologies to expedite the verification process of complex designs. The core concept is to utilize the insights and achievements gained from one methodology as foundational elements to enhance the effectiveness of the other.
By combining these approaches, verification teams can capitalize on the strengths of each method. The Bug Hunt technique, with its focus on identifying and addressing specific issues, can provide detailed information and uncover intricate bugs that might be missed by broader verification strategies. Meanwhile, formal methodologies offer a rigorous framework for systematically proving design correctness and exploring deep state sequences.

Leveraging the findings from Bug Hunts, formal verification can be more precisely targeted, using the identified issues as starting points for deeper exploration. Conversely, the comprehensive coverage and formal proofs obtained through formal methodologies can inform and refine the Bug Hunt process, ensuring that it focuses on the most critical areas.
This synergistic approach not only accelerates the verification process but also enhances the overall quality and reliability of the design, ensuring that intricate and complex systems are thoroughly vetted and robustly validated. In this report, bug hunt technique is described. It leverages the previous reached traces for converging the property using as initial start point to explore further state space. As described, it consists of three major steps:

1. Identify and extract a set of good interesting spots from the last covered traces
2. select one of the point from these traces.
3. Launch and monitor multiple bug hunt runs on the computing servers

To find initial starting point, we used several ways, including interface interactions, control and interrupts, concurrent events, feedback loops and counts, user-defined assertions, and coverage properties. Then these starting points are filtered on the basis of design complexity, types of mode and engine health. One should have a good knowledge of design to use some modes where we want to use different starting point so that property conclude fast. If initial points are not good or not in the COI of property, then engines will explore state space in different direction and will not get counter example. Engines should be selected according to type of property.

The results of Hunt run can be non-deterministic due to several factors like parallel synthesis, previous proof results, proofs running in parallel, compute environment (machine loads), time limits etc. This is why, we may need to run the setup for extended period, potentially exceeding the lab's time limit, to achieve deterministic results. Variability of results such as different number or different length of cex's are expected.

Formal engine health is characterized by several key parameters: the number of formal targets that have been proven or triggered, the sequential depth explored, and the formal knowledge acquired during the verification process. These parameters provide a comprehensive view of the progress and effectiveness of formal verification efforts.
Once formal engine health is assessed, a set of traces is utilized to initiate multiple bug hunt runs. These runs are designed to systematically explore the design, leveraging the insights gained from the initial traces to focus on areas most likely to yield complex bugs. By analyzing the results and identifying intricate bugs, it becomes evident that the bug hunt technique significantly enhances the quality of outcomes in a formal regression environment.
This approach not only improves the thoroughness of the verification process but also ensures that potential issues are identified and addressed early, contributing to a more robust and reliable design. The combination of formal engine health assessment and targeted bug hunts provides a powerful framework for optimizing formal verification efforts and achieving high-quality results.

5.2 Future Scope

Exploring formal verification using bug hunt techniques to rigorously prove the correctness of the design. Bug Hunt methods can provide mathematical assurances of the design and identify the potential issue in the design specification. Also, there are many types of hunt modes available with different engines, it has to be researched which mode gives best result in less time. This can help discover unique scenarios that might not be apparent through traditional testing methodologies.

REFERENCES

[1] E. Seligman, T. Schubert, M.V. Achutha kiran Kumar. Formal Verification: *An essential toolkit for modern VLSI design*, section Foreword

[2] Richard Ho, et al., "Post-Silicon Debug Using Formal Verification Waypoints," in Proc. *DVCon* 2009

[3] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method Based Approaches*, section 1.3.1.

[4] Verification Academy, "2018 Functional Verification Study," *Verification Academy*. [Online]. Available: https://verificationacademy.com/seminars/2018-functional-verification-study.

[5] Lam, W. K. (2009, January 15). *Hardware Design Verification: Simulation and Formal Method-Based Approaches*

[6] Verification Academy, "2020 Functional Verification Study," *Verification Academy*. [Online]. Available: https://verificationacademy.com/seminars/2020-functional-verification-study.

[7] Synopsys, "Delivering Functional Verification Engagements," *Synopsys*. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/services/whitepapers/delivering-functional-verification-engagements.pdf

[8] C. Kern and M. R. Greenstreet, "Formal Verification in Hardware Design: A Survey," *Tech. Rep.*, Univ. of British Columbia, Vancouver, 1999.

[9] Nishant Gupta," Exploration of formal verification in GPU hardware IP," Oct. 10, 2019

[10] Mandar Munishwar, Vigyan Singhal, et al., "Architectural Formal Verification of System-Level Deadlocks," in Proc. *DVCon* 2018

[11] Cadence Design Systems, "Bug Hunt Technique," *Cadence*. [Online]. Available: https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1OPP0000011VQy2AM&pageName=ArticleContent

[12] Cadence Design Systems, "Bug Hunt Technique Rak & Engines," *Cadence*. [Online]. Available: https://support.cadence.com/apex/ArticleAttachmentPortal?id=a1O3w000009l4u6EAA&pageName=ArticleContent

[13] V. Bertacco. Ph.D. Dissertation, Stanford University, August 2003. [Online]. Available: http://web.eecs.umich.edu/~valeria/research/thesis/ thesis2.pdf

[14] A. Gupta. "Formal hardware verification methods: a Survey*". Formal Methods in System Designs*, Vol. 1

[15] R. Stuber, *Formal Verification: Not Just for Control Paths*, Mentor, June 2017

[16] J. Bromley, J. Sprott. Verilog, 2016. [Online]. Available: https://www. verilab.com/files/dvcon_eu_2016_fv_tutorial.pdf

[17] C.E. Cummings, *System Verilog Assertions Design Tricks and SVA Bind Files*, SNUG, San Jose, 2009

[18] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, 1st ed., Springer, 2003. [Online]. Available: https://doi.org/10.1007/978-1-4615-0302-6

Plagiarism Check Report

turnitin

Digital Receipt

This receipt acknowledges that Turnitin received your paper. Below you will find the receipt information regarding your submission.

The first page of your submissions is displayed below.

Submission author:	Archit Gupta
Assignment title:	Thesis_Paper
Submission title:	MTech Thesis
File name:	Plagiarism_Archit_report-for_Plag.pdf
File size:	3.28M
Page count:	37
Word count:	10,568
Character count:	57,223
Submission date:	29-Jun-2025 11:15AM (UTC+0530)
Submission ID:	2707604882

CHAPTER 1

INTRODUCTION

1.1 Introduction to the area of work

As design complexity continues to grow, driven by advancements in fabrication technologies, the challenge of verification becomes increasingly significant. Traditionally, simulation techniques have been the primary method for verifying design correctness, and ongoing improvements in simulation technologies—such as coverage models, guided test generation, and faster simulation speeds—have made it more efficient at identifying bugs. Meanwhile, formal methods, particularly through the use of model checkers, have shown success in achieving complete coverage for small modules or conceptual protocols. These advances have also facilitated the development of hybrid verification approaches, where formal methods are used to guide simulation or even replace it in certain areas of the verification process. [1]

However, both simulation and formal verification still face challenges as design complexity increases, particularly with the rise of parallel processing units. As designs become more intricate, the likelihood of missing corner cases increases, and simulation tools often struggle to analyze these complex scenarios. While formal verification has proven useful in identifying corner-case issues, its current capabilities are primarily limited to small designs. The consistent application of formal methods to large-scale, multi-module systems remains an ongoing challenge, and although formal verification has been successfully used for high-level designs, these efforts are often time-consuming and design-specific. [1]

Formal verification is also valuable for quickly validating protocol concepts before the actual design phase. However, connecting validated protocol models to the final implementation has yet to be fully explored. Although challenges exist, formal verification remains a reliable method for validating intricate systems-on-chips safety-sensitive designs. This methodology encompasses three core elements: reasoning (to prove correctness), bug hunting (to find known and unknown bugs), and coverage closure (to identify reachable or unreachable design elements). [2]

Emerging approaches, such as waypoints, cex-points, and goal-posting, are becoming increasingly popular for uncovering deep, complex bugs that traditional methods might miss. As traditional formal verification approaches start from time zero and explore designs in their entirety, they become less efficient as the design space expands. Complex bugs often arise from rare combinations of events, and sophisticated bug-hunting techniques are now being employed to identify these issues more effectively.[2]

1.2 Motivation of the work

The traditional approach to programming has largely been centered around starting with an abstract model and progressively incorporating more specific details. This method was the dominant paradigm until the emergence of object-oriented programming. The concept behind this approach is that a program can become more flexible and responsive to changes, whether they are extensions or

1

MTech Thesis

ORIGINALITY REPORT

5%	%	4%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS