New Paradigms in Functional Verification and Test

A THESIS SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

by

Vineesh. V. S

Roll No. 123079040

Advisor: **Prof. Virendra Singh**, Department of EE, IIT Bombay

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Mumbai 400076, INDIA.

April 2021

Abstract

In the present era of pervasive computing, system-on-chips (SoCs) play a very important role. Given the nature of the complexity of these designs, significant verification efforts are required to ensure their correctness. It has been observed that design debugging has become one of the most important steps in the development cycle of these SoCs. Simulation-based verification is never sufficient for ensuring design correctness because of its incomplete nature. Formal techniques like model checking promise to solve this issue through a complete state space traversal approach. However, because of increasing design complexity, such methods suffer from scalability issues. Guidance-based state space traversal techniques have been proposed in the past to assist the model checkers in overcoming the complexity bottleneck. Automatically identifying these guidance hints turns out to be relatively difficult and requires heuristic-based reasoning procedures. Additionally, to come up with quick fixes during the debug stage, an effective bug localization strategy is needed. In this thesis, we revisit the paradigm of guidance-based model checking and propose a methodology to improve these guidance generation mechanisms for achieving fine-grained bug localization. The proposed technique involves the mining of invariant-like properties from simulation traces. The mined properties act as probable guidance candidates for the model checking exercise. To identify useful guidance hints out of possible ones, we use Bayesian networks that explore conditional dependence between the various hints at different levels and the target property. These guidance hints are utilized for obtaining possible buggy regions, which are then analyzed via an iterative model checking methodology for fine-grained bug localization. The proposed bug localization methodology involves counterexample based reasoning for hypothesizing the bug in terms of related signals. We present different case studies to illustrate the benefits of the proposed methodology. We also propose an alternative methodology of bug localization through mining of assertions and assistance from static analysis. In this methodology, we obtain multiple error traces based on high level functional failure. Starting from an initial error trace, we employ model checking based reasoning

to generate more error traces. We utilize these error traces to extract common conditions which are utilized for bug localization. We employ static analysis of the design to obtain fine-grained error localization.

Efficient methods to do online testing of faults in the chips is as important as fixing bugs in pre-silicon. With the technology scaling and the rise of complex designs, it is becoming difficult to achieve required faults coverage in manufactured chips. That stresses the need for online testing approaches which enables testing of the chips even after shipment. Online testing can be done in two ways: using a Built-In-Self-Test hardware or using a set of instructions run in functional mode. Since the first approach leads to area and power overhead due to the additional hardware, the second approach is becoming more popular. But, generic instruction based testing methods does not always give good fault coverage for the hidden portions like Forwarding unit. Hence it becomes important to identify test generation methods which is best for different parts of these units. In the last part of this thesis, we discuss an approach for testing stuck-at faults in a processor using an auto-generated instruction template. This work combines constrained random instruction template generation and an approach to convert test vector to instruction sequences for generating a set of instruction sequences which will detect all stuck-at faults in the selected module of the processor. The fault simulation using the instruction sequence generated using the above process delivered a coverage of 97.33%. A detailed analysis of the non-detected faults revealed the fact that they cannot be applied in functionally using the existing instruction set of the processor used for test. It proves that the proposed method is able to generate test for all stuck-at faults, which can be functionally tested, in the forwarding unit of the MIPS 5-stage pipelined processor.

Keywords - Model Checking, Guided state space traversal, Guideposts, Bayesian modelling, Property extraction.

List of Publications

Included in Thesis

1. **V. S. Vineesh**, B. Kumar, R. Shinde, N. Sharma, M. Fujita and V. Singh, "Enhanced Design Debugging with Assistance from Guidance-based Model Checking," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020.
2. B. Kumar^{+}, **V. S. Vineesh**$^{+}$, P. Nemade, N. Sharma, M. Fujita and V. Singh, "A Semi-formal Technique for Effective Bug Localization in Hardware Designs," [under submission to IEEE Transactions on Computers (TC), April 2021. [$^{+}$ means equal contribution, *names given in alphabetical order*]
3. **V. S. Vineesh**, B. Kumar, R. Shinde , A. Jaiswal, H. Bhargava, and V. Singh, "Orion: A technique to prune state-space search directions for guidance-based formal verification," in *2019 IEEE 28th Asian Test Symposium (ATS)*, Kolkata, Dec 2019
4. **V. S. Vineesh**, B. Kumar, and J. Adhaduk, "Identification of effective guidance hints for better design debugging by formal methods," in *23rd International Symposium on VLSI Design and Test (VDAT)*, Indore, India, July 2019
5. **V. S. Vineesh**, N. Hage, B. Karthik and V. Singh, "Achieving full functional coverage for the forwarding unit of pipelined processors," *2017 IEEE East-West Design & Test Symposium (EWDTS)*, Novi Sad, Serbia, 2017
6. R. Shinde, B. Kumar, **V. S. Vineesh**, and V. Singh, "Aquila: A Methodology for Achieving Fine-grained Bug Localization during Design Verifi-

cation", *20th IEEE Workshop on RTL and High Level Testing (WRTLT)*, Kolkata, Dec 2019

Not included in Thesis

1. G. U. Vinod, **V. S. Vineesh**, J. Tudu, M. Fujita and V. Singh, "LUT-based Circuit Approximation with Targeted Error Guarantees," in *2020 IEEE 29th Asian Test Symposium (ATS)*, Malaysia, Nov 2020

2. B. Kumar, A. K. Jaiswal, **V. S. Vineesh** and R. Shinde, "Analyzing Hardware Security Properties of Processors through Model Checking", in *33rd International conference on VLSI Design (VLSID)*, Bangalore, Jan 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern electronic systems have created revolutionary transformation in the domain of information and communication technology. These systems essentially contain a large number of components known as System-on-Chips(SoCs). Persistent increase in the number of Intellectual properties (IPs) cores such as embedded processors has led to the enormous complexity of SoCs. This is largely attributed to the growing integration of functionality on a single chip due to the demand of miniature size, low cost and power efficient devices. The usage of these devices in safety critical applications such as those related to space, aircrafts, biomedical and automotive applications raises serious reliability concerns. Due to the tremendous complexity of these designs, traditional verification and testing methodologies employed to ensure customer satisfaction fail to guarantee product quality compliant with various standards. Given the increasing size of modern designs and the embedding of rich functionalities, complete verification requires huge efforts and elongated implementation schedules. In a competitive market space, meeting time-to-market (TTM) deadlines becomes highly challenging.

Figure 1.1 shows the amount of time spent by design houses in verification

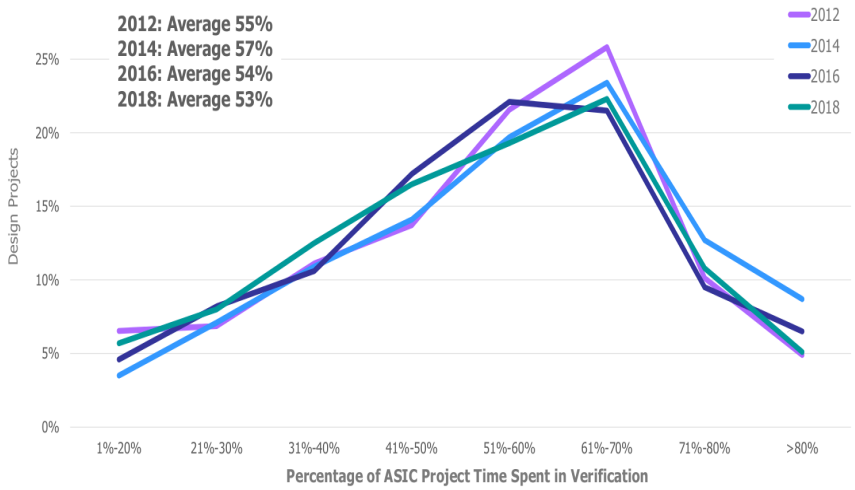


Figure 1.1: Time spent in verification

activities. It can be inferred that a larger chunk (more than 50 %) of design development time is related to verification planning and efforts. Therefore, speeding up the verification process becomes an interesting problem to investigate. It is worth to note that analyzing verification results becomes the second most important step. Even after the accelerated verification efforts, bugs stand probable to escape to silicon. Figure 1.2 shows the different causes behind such bug escapes. It can be observed that design errors are the largest contributor to bug escapes.

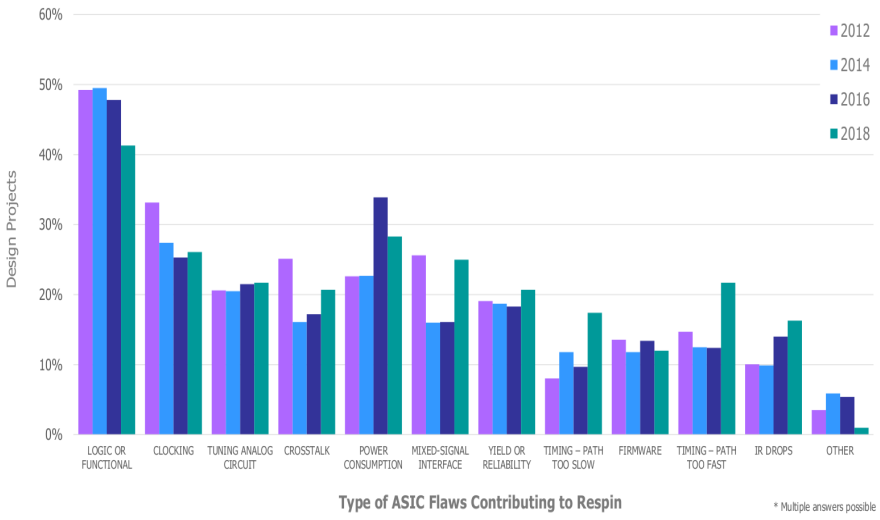


Figure 1.2: Reasons behind bug escapes

Therefore, it can be clearly understood that enhanced verification need to be researched upon with a focus on automatic bug localization. One of the most important aspects of the the bug localization problem is to analyze high-level failures from a low-level design perspective. Expert knowledge of the design internals is one option to obtain fine-grained bug identification. However, this manual intervention adds up to the elongation of the bug localization process. With the help of automated localization of bugs, the amount of time spent in verification process can be significantly brought down.

Bentley [6] described following major reasons of different bug types observed during pre/post-silicon validation:

- Inconsistent Specifications: these types of bugs fall into several categories: SoC architects not communicating their expectations clearly to the designers, misunderstandings between the specifications and design as well as between different parts of design (e.g. incorrect assumptions about what another unit was doing).

- Logic Changes: these kind of bugs are caused when the design gets changed, usually to fix bugs or timing problems, and the designer did not take into account all the places that would be impacted by the change.

- Random Initialization: these types of bugs may be caused by state not being properly cleared or initialized at reset. These are related to corner cases mostly or clock/power gating issues.

- Improper Documentation: something (algorithm, micro-instruction, protocol) may not be documented properly leading to disconnect between the final specification and implementation.

- Late Feature Definition: these kind of bugs are related to features that are defined very late into the project. Given the complexity of the designs, integration of new features lead to introduction of unintentional bugs.

Apart from the above scenarios, there are cases when the designer is not fully confident of the developed design and relies on the verification step to uncover bugs. Therefore, the verification step becomes very essential for the overall success of the design implementation methodology. This thesis address the verification (and subsequent bug localization) problem from a bug agnostic view point.

1.1 Need for Enhanced Functional Verification

Due to state space explosion, complete formal verification of large designs is a prohibitive exercise [7]. Different techniques such as abstraction, design (RTL) slicing assist to solve this problem to some extent. The primary problem with the generalization of such techniques is their suitability to certain kind of designs only. For example, bit slicing is very successful usually in datapath-intensive kind of designs. Semi-formal methods utilize a mixture of formal and simulation methods to discover design bugs [7, 8]. Furthermore, because of increasing third party intellectual property (IP) integration in the modern designs, achieving complete functional verification by formal methods also remains a daunting goal to date. With advancements in model checkers and other formal techniques, large designs can be verified in a partial or semi-formal manner. However, it is well known that exhaustive exploration of design state space is still prohibitive. For formal verification of designs, guided state space traversal [9, 10] methodologies have often been utilized to varying degree of success.

In this thesis, we revisit the concept of guided state space exploration which holds the promise of complete formal verification. Similar to the concept of partitioned model checking, guided traversal is a viable alternative for solving the scalability problems in model checking. Since it is not trivial to devise guidance strategies in an automatic manner, identification of the guidance hints becomes very crucial for a directed traversal of the state space. This directed traversal can ultimately reduce the time spent in formal verification and also assist in better design debugging. We propose a methodology for identification of such guideposts and utilize them for debugging purpose. Our goal is to achieve faster counterexample generation by the usage of guideposts. Experiments on a complex design show that guidance hints identified with the proposed methodology provide significant gains during model checking for different error traces.

1.2 Bug Localization and Coverage Estimation during Functional Verification

The usage of random testbenches or constrained-random testbenches enhances the quality of the verification process. With the help of proper constraints (based on design and specifications), comparatively more bugs can be uncovered leading to observation of higher number of failures in the verification runs [11]. One of the most important steps after observing a functional failure is to find the root cause behind it. With help of this root cause analysis (RCA), the corresponding fix can be derived. Generally, by observing high-level failures during pre-silicon verification simulations can hint towards a major block (or region) in the design. However, such bigger design portions fail to assist in localizing at a fine-grained level. Therefore, achieving bug localization at low -level

design description (i.e., at RTL) proves very useful. Additionally, lack of automation during the bug localization process becomes an obstacle in obtaining the required design correction.

Figure 1.3 shows a design bug in tlu_tcl module of OPENSPARC processor. The lines numbered as 1106 and 1107 are the actual RTL code whereas

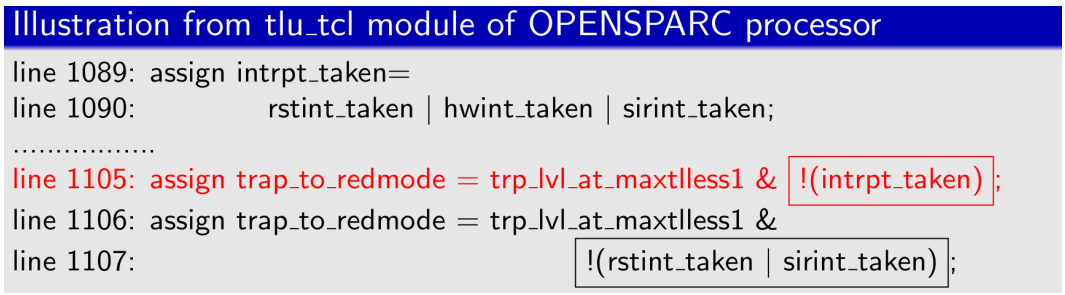


Figure 1.3: Design bug illustration at RTL [1]

line 1105 (shown in red color) denote the buggy RTL code. It is obvious that a high-level failure can not directly point towards the exact line (of the design description) relating to the bug.

There are many different types of coverage used in the verification procedure. For example, the dominant verification coverage metric in industrial practice is functional coverage, where the designers and verification engineers devise a set of functional coverage points that the validation is supposed to hit. There are even academic proposals for coverage metrics specifically designed for the lack of observability in post-silicon validation. Code coverage is a useful metric for finding out the bugs which are there in our RTL design. Code coverage includes two parts:

- Statement Coverage: The fraction of the RTL statements exercised by the testbenches.
- Branch Coverage: The fraction of all RTL branch directions exercised by the testbenches.

Similarly, FSM coverage measures how many finite state machine (FSM) states have been visited and how many transitions occur during execution of test-benches. It is worth to note that the above coverage metrics have some potential in assisting towards bug localization. However, these metrics are not explicitly related to bug localization efficacy and a linear correlation between them is not expected. This motivates us to come up with dedicated methods for effective bug localization. Therefore, in this thesis, we have tried to utilize these coverage metrics for assisting in the goal of automatic bug localization.

1.3 Functional Test Generation

Test solutions based on Design-for-Testability (DfT) methodologies, such as scan or built-in self-test (BIST), fail to meet the requirements in various scenarios, such as generating test for deeply embedded SoC components. Further, small delay effects and new type of defects such as those related to device aging or wear-out as well as process variability are aggravating the situation when we move down to lower technology nodes. Generating tests to account for these effects needs looking into new directions and techniques. Software-Based Self Test (SBST) helps to test inner and less accessible SoC modules through in-field testing. It does not need any extra hardware and allows at-speed testing thereby enabling screening of delay defects. However, the challenge here is to address issues such as the storage of self-test programs, triggering the processor for its execution along with retrieval and checking of obtained results. Another major issue is to achieve a high value of effectiveness of such test programs. In many cases, DfT structures are deliberately made inaccessible thereby terminating the possibility of their re-use for in-field testing. Under such scenarios, functional

tests are the only feasible solution. Application of functional tests can help overcome the limitations imposed by structural fault models and thus it has the potential to cater to unmodeled defects as well. The complete process of test application needs to be automated which can lead to reduction in overall test effort and time when compared to manual generation of functional tests. This leads to a growing demand of generation of tests which can guarantee detection of all kind of defects which are appearing more frequently in modern designs as the technology scaling continues. Development of efficient on-chip testing mechanisms can help ensure checking the critical features of operation of the SoC each time before its use.

1.4 Thesis Organization & Contribution

The organization of this thesis (outlined in Fig. 1.4) is as follows.

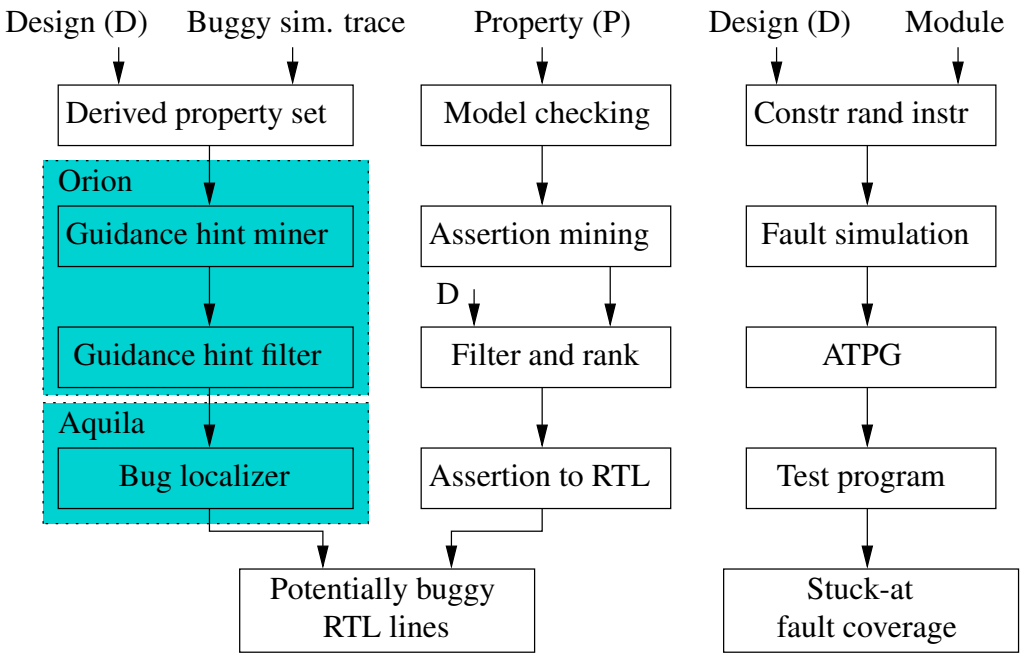


Figure 1.4: Thesis overview

- Chapter 1 provides an introduction to the thesis.

- Chapter 2 discuss the prior art. The limitations of the previous work are also briefly discussed in relevant sections of this chapter.
- Chapter 3 elaborates on guidance hint generation based on static approaches and their application in model checking.
- Chapter 4 discusses three broader sections of the thesis overview diagram shown in Figure 1.4. It presents the bug localization approach using guidance hints.
- Chapter 5 discusses bug localization using semi-formal approaches that utilize assertion mining. It also elaborates on a genetic algorithm-based framework for intelligent testbench generation.
- In Chapter 6, instruction-based self test approaches targeting structural faults is discussed.

Finally, the thesis is concluded in Chapter 7 with a listing of contributions and future scope arising out of the work done in this thesis.

Chapter 2

Literature Survey

Functional verification is a step of utmost importance in modern design development cycle [12]. Given that exhaustive design simulation is rarely possible, formal methods hold promise in discovering bugs at an early stage. With the help of techniques like model checking, a mathematical reasoning of the design correctness can be performed and abnormal behavior can be classified as belonging to buggy regions in the RTL. However, whenever a model checker provides a complex and tedious counterexample, the subsequent design debug process becomes elongated. This results in delay of the bug-fixing step which in turn delays the overall design development cycle. If the verification engineer succeeds in localizing the bug behavior to any specific region (say, in range of certain lines of the RTL code) of the design, the design team can quickly revise the design and the subsequent steps can begin without any hindrance.

Typically, for large designs, formal methods hit the scalability limits thereby requiring assistance of heuristics like bit-width slicing [13–15]. These heuristics can reduce the computational efforts required for state space traversal [10, 13, 14]. In a similar vein, with a mix of simulation and formal methods, design bug detection becomes relatively easier and has been reported to be successful [7,8].

However, for guaranteeing completed design correctness, formal methods are the only techniques available to verification engineers. To tame the scalability issues in model checking, guided state space traversal [9, 10, 16, 17] offers some assistance. This guidance (which is in the form of internal signal values) helps in partitioning the model checking problem. We utilize this partitioned model checking technique for fine-grained design bug localization. Abstraction is also one of the methods to handle the scalability issues of formal methods in general (and model checking in particular). The obtained counterexamples have often been utilized to refine the design abstraction or find other complex bugs [18–20].

2.1 Guided state space traversal in functional verification

Guided state-space traversal was first proposed in [10] and then refined in [9]. Since model checking turns out to be very useful in the process of bug finding, guidance-assistance helps in quicker bug detection too. A case study on designer provided guidance has been elaborately presented in [21]. However, the automatic generation of effective guidance hints has been an open research problem [22]. A semi-automatic methodology for gathering useful guidance hints through structural dependency graphs was proposed in [16]. Apart from this, guidance-assisted formal methods have also been explored in the area of semi-formal bug detection techniques [7]. Along the lines of [16], the proposed methodology strives to cut down the manual effort by taking the assistance of Bayesian modeling. An alternative approach of Bayesian modeling could be to add more nodes (corresponding to designer provided guidance hints) to the network, along with the nodes obtained from the mining process. Through a few

heuristics, techniques have been described in [23] to decompose large properties into smaller ones. This work is related to the part where we decompose properties to get nodes for the Bayesian network. Counterexample guided abstraction refinement is discussed in [19] and [20] for model checking the properties during design verification. In the proposed methodology, we do not utilize abstractions for refining and getting better counterexamples subsequently, since obtaining abstractions turn out to be an equally challenging problem. The prior work most closely resembling our approach (specifically the $Aquila$, i.e., bug localization part presented in Chapter 4) is that of [24]. In this work, the authors have attempted localization of bugs in the SVA (SystemVerilog Assertion) representations assuming that the design is correct. However, we assume that the properties are correct, while the design is buggy. The approach in [24] is SAT-based and aims to bring out the inconsistency between a property and the correctness of the design. While a direct comparison of [24] and the proposed technique is not trivial, we believe that finding the bug in the design is a more difficult problem since the probable inconsistency (which is the reason for the bug) is not very evident from a high-level simulation error trace. A SAT-based methodology for localizing mismatch between gate-level netlist and design specifications has been proposed in [25] utilizing time-frame expansion and the knowledge of design hierarchy. However, the scalability of this diagnostic technique is limited by the time spent in SAT solving and multiple diagnostic runs with obtained counterexamples. Verification of circuits by modeling them into trace structures is discussed by the authors in [26] for asynchronous designs.

Hanna et al. [27] proposed ways to identify intermediate states for reaching the target state from an initial state with less memory requirements and time

required for the state space traversal. The intermediate states act as starting states for the model checkers to traverse to the immediate next intermediate state. Multiple such intermediate states are tried out in parallel to reduce the possibility of bypassing any essential intermediate state. While the concept of the intermediate states (or referred to as guide posts in our work) is similar to this idea, the generation of the same is different in our work. Zamfir et al. [28] provided methods to generate programs which cause bugs to manifest itself. It uses statistically found intermediate goals to reach the failure point of program execution. These intermediate goals help in dividing the search space required to reach a target into several smaller search spaces. The intermediate goals discussed in this work is a software equivalent of the guidance hints discussed in our methodologies. Chopra et al. [29] proposed a method and system to partition integrated circuits for the purpose of verification. The authors discuss methods to get partitions so that they are contiguous in the design. These partitions are either verified using model checking or through simulation. Our work proposes a method for debug using guidance hints. The use of guidance hints in model checking produces speed ups similar to the scenarios where one uses partitioned model checking. The difference is that we use a different approach to achieve that speed up. Du et al. [30] discussed a method for the property verification of different partitions in a circuit. The coverage outcome is selected from a set of possible outcomes for each partition. The coverage of each element in the design is calculated using that outcome. Cutler [31] introduced new methods for software debug that reproduces the program execution flow which led to the error. This is done by storing the function call information in the program. This execution flow which is reproduced can be used to localize the cause of error in the software program. Altekar [32] proposed a cost effec-

tive reproduction of the program execution flow in the microprocessor and the related software. The objective is to use that flow for reproducing the errors in the device functions such as a software bug. The key differentiator here is that it does low-overhead recording of the program execution with high precision replay. This replay mechanism is applicable for microprocessors and network software. Our techniques produces a more efficient debug/error localization (in hardware designs) mechanism compared to the full reproduction of the program execution flow (hardware simulation is the equivalent term in this work). Lam et al. [33] simulates the Verilog design with a testbench with bug detection setup in it. It consists of a rapid bug detection tool (an assertion which catches the bug in a cycle) and a bug isolation tool (mechanism to store desired values in memory for further analysis). It uses checkpoints to rollback simulations when a bug is detected. It helps in bug isolation. The authors propose one of the methods for hardware debug. In our work, we have proposed a faster way for bug localization using guidance hints. Chang et al. [34] checked for the errors in the outputs corresponding to a given test vector in the RTL. This is done by comparing the output with the expected value of the output. When an error is detected, the tool returns a signal path which might have led to the error. It also gives suggestions for fixing the error. Similar to our techniques, this work is also operating in the RTL-level, instead of the gate-level. This proposal requires the debug engineer to change the RTL design to an enriched RTL form which contains necessary conditional assignments which helps in identifying problematic signals in the RTL. With different design practices in writing RTL, it is difficult to generalize this method. In our experiments, the RTL in the Verilog format can be used as it is. The authors in [35] obtain the counter example from another source and annotate different types of values of the signals based on its relationship

with the failing property. Rahman [36] correlates the (simulation) databases of different tools corresponding to certain signals. It enables faster fault isolation in circuits. Safarpour et al. [37] proposed four methods for debugging hardware designs. The first one is time abstraction, second using design abstraction and refinement, third using QBF formulations and the forth one using max-SAT debugging formulation. This is discussed as a prior art since the end goal of our methodologies is also to do debugging and error localization. The authors in [38] discusses RTL static analysis using the grammar for the Hardware Description Language (HDL). Static analysis similar to this is used for generating the SDG in our methodologies. Martensson [39] proposed a way to visualize the signals in the counterexample of a failed property. It helps the user in easier debugging. In our methodologies, we utilize the selection of signals generated by the Jaspergold tool which generates counterexample using a similar concept as discussed in [39]. Singhal et al. [40] discussed a mechanism to highlight signal waveforms in the simulator. The source code corresponding to this highlighted region can be accessed using the automation proposed by the authors. This approach can be used for reaching the signal in the hardware which is buggy. Vasudevan et al. [41] proposed a decision tree logic-based data miner for mining assertions from the RTL simulation traces. It is similar to the miner in our work. Continuing along similar lines, the authors in [42] proposes a mechanism to evaluate the importance score of hardware assertions. It is done by combining the individual importance score of the variables present in the assertions using certain metrics-based approaches. A method to automatically generate guidance hints after analysis of signal dependencies of the design through Control Data Flow Graph (CDFG) construction has been proposed in [16]. However, this method fails to filter the generated guidance hints resulting in a set of inter-

mediate signals (and their values) which require multiple iterations to achieve success during model checking. It may also happen that the design engineer has provided a set of guidance hints to the verification engineer [21] for guided formal verification. However, since the designer knowledge is limited only to specifications or a high-level overview of the design, these sets of guidance hints have higher chances to be spurious.

2.2 Semi-formal methods of bug localization

Logic simulation remains the highly popular technique to identify design bugs, due to its scalability. However, simulation-based techniques suffer from insufficient coverage, hence most often fail to identify all the design bugs. Formal verification (FV) is an alternative technique to overcome the coverage limitations of simulation-based techniques, due to its exhaustiveness. Therefore, FV methods can enable verification engineers to identify intricate design flaws too complex to find using simulation-based methodologies. However, FV techniques have scalability drawbacks that restrict the size of design components that can be formally verified. Note that one of the key strengths of FV techniques is their use of symbolic reasoning, to efficiently explore a huge number of individual scenarios that would have not been covered using simulation [7, 43]. As mentioned before, localizing functional bugs becomes a daunting task in the design development cycle [44]. Typically, industrial practices involve usage of waveform-based debugging tools like Synopsys Verdi [45]. However, this requires lot of manual intervention thereby leading to elongation of the overall development process. This is primarily because unless the complete debug is over, the respective fix/correction can not be embedded into the design. In

those scenarios, automatic bug localization becomes very important. A few approaches in the literature have proposed methods of automatic bug localization. Pal et al. [46] analyzed simulation traces for the purpose of identifying bugs in the design description based on the commonality of symptoms between them. However, simulation-based methodologies typically suffer from dependence on the employed testbenches. In this regard, exploration of semi-formal methods is an alternative for bug localization during functional verification [7]. During formal verification, if the property (that we are checking for a given design) fails, we obtain a counterexample. As counterexamples can turn out to be tedious for the subsequent analysis, we need a methodology for the automatic analysis of these counterexamples. Note that this problem has been acknowledged in the area of software bug localization [47]. Apart from the application of semi-formal techniques in automatic bug localization, semi-formal methods have found wider acceptability in speeding up the simulation tasks [48]. Shyam et al. [49] proposed distance metric-based hybrid verification methodology targeting improvement in testbench quality. Chang et al. [50] proposed methods to minimize longer error traces into smaller ones through redundancy removal and a mix of other hybrid methods. SAT-based error trace minimization has been proposed in various approaches [51–53].

2.3 Instruction-based test generation

One of the very first attempts for automatic test instruction generation was by Thatte et al. [54]. They used a graph theoretical approach for modelling any processor by a System graph (S-graph) at the RTL level. Another technique proposed by Parvathala et al. [55] focussed on the testing-based on random

instruction generation. The motive behind this work was to do automated test instruction generation to reduce the test cost. A DFT feature enabled a low cost tester to load the test patterns using less number of pins, into the on-chip memory.

A series of different graph modelling approaches have also become popular in the test generation of processors. Some of the works in this track include those by Singh et al. [56] and Shaneen et al. [57]. Generation of pseudorandom test patterns was proposed by Chen et al. [58]. C.H. Chen et al. [59] generated constrained test patterns using ATPG tools with the information gathered from the architectural model, RTL description and the gate level netlist. Template-based test generation was another approach followed by Chen et al. [60]. Another method proposed by Guramurthy et al. [61] did the mapping of module level test vectors into test instructions. Here a bounded model checker was used for generating test instructions corresponding to the test vectors. Additionally, researchers have identified that there is no one-stop solution for testing the full processor. Hence the works like *Hybrid-SBST* [62] has emerged. Contributions from Kranitis et al. [62] combines the strengths of deterministic structural SBST methods (using constrained APTG for e.g.) with Random Test Program Generation. Work done by Chen et al. [58] discusses the concept of three types of programs required for testing the faults: Test generation program to generate test patterns from self-test signatures, test application program and test response analysis program. These programs along with the signatures can be loaded into the cache using a low-cost tester. Some of the other works focus on modelling the processor fully or in parts in graphs and decision diagrams. One such method propsoed by Shaheen et al. [57] uses SAT solvers on the ADD (Assignment Decision Diagram) for test generation. Another method proposed

by Singh et al. [56] introduced the concept of IE (Instruction Execution) graphs for processor modelling and test. But these methods are yet to be automated and in some cases really difficult. Some other techniques suggests the test generation of independent modules inside the processor using powerful commercial tools and map them to test instructions. One such method was proposed by S. Guramurthy et al. [61]. The work done by Bernardi et al. [63] makes use of the theorem propsed by Makar et al., [64] which says that you require only 2n test vectors to test a generic n-to-1 mux. This concept eliminates the requirement of introducing all possible data in the instruction sequence At the same time this concept is applicable only to muxes and cannot be applied to other blocks. Hence this method will be too specific to the case where only muxes are present at the input of the module under test. To make the process more generalized and applicable to other parts of the processors also, we chose to use constrained test generation method for the remaining faults. Instead of using constrained ATPG we designed a generic wrapper in Verilog which can be synthesized upon specifying the constraints. The method of using wrapper modules is discussed in the work done by Tupuri et al. [65]. When these test vectors are applied at the input of the wrapper, output of the wrapper (which is the input of the forwarding unit) is assured to be a functional test vector.

Chapter 3

Guidance Hints for Design Verification

3.1 Introduction

Functional verification is one of the most challenging tasks in the modern design development cycle. While considerable progress has been made in the area of model checking to handle the state space explosion problem, developing automatic methodologies for achieving complete design verification still needs great attention [9, 49]. By utilizing simulation, formal techniques like model checking can be guided and made more usable for successfully verifying larger designs. However, discovering such guidance strategies in an effective manner is not a trivial exercise [7,21]. Therefore, two important problems emerge out of this methodology for design verification by formal methods. First, generation of some guidance for assisting the model checking process. Second, developing techniques for further pruning out the state space with the help of obtained traversal (reachability) guidance. The second step is required because the guidance generation step is never a complete one and leads to a large number of possibilities remaining in the state space which serve as a hindrance in achieving higher speed-ups during model checking. This chapter proposes a Signal Dependency Graph (SDG)-based methodology for solving the second challenge.

With the help of Signal Dependency Graph built from the RTL (Register Transfer Level) design, the guidance (i.e., the list of internal signals and their values which act as guidance hints) help us to perform a partially directed analysis of the design state space. This directed state space search leads to significant gains in CPU time during model checking for properties. As a result, we succeed in generating counterexamples in lesser time.

The remainder of the chapter is organized as follows. The preliminary concepts on waypoints are covered in Section-3.2 and the proposed methodology is elaborately explained in Section-3.3 with a detailed discussion on a case study. Experimental results and observations are presented in Section-3.6. The related work is briefly mentioned in Section-3.7. Finally, the chapter is concluded in Section-3.8 with a discussion regarding the limitations of the proposed waypoint identification methodology and its possible extensions.

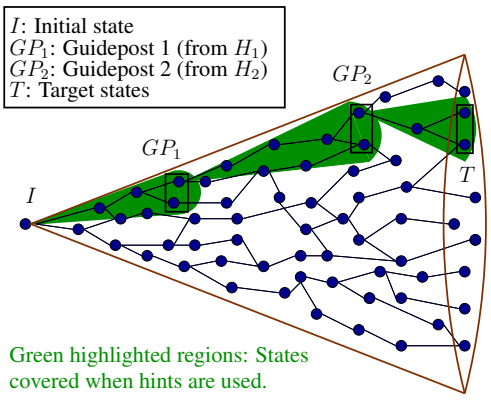


Figure 3.1: Guidance-based state space traversal (For illustration purposes only).

3.2 Illustration of guidance-based state-space traversal

We refer to the guidance hints as "waypoints" hereafter in this chapter. For the sake of brevity, we put forward the following definition:

Let S be the set of reachable states in design. A waypoint WP_k is a set of fully

or partially specified signals $(s_1, s_2, ..., s_n)$ *that correspond to a set of states* $S^K = \{S_1, S_2, ..., S_n\} \epsilon S$.

S^K is also referred to as guidepost (GP) in this thesis. Let us consider the diagram given in Figure 3.1 which illustrates guidance-based state space traversal. The set of states in the blue boxes (except for the leftmost and rightmost ones which are the initial and target states) are examples of guideposts (S^K) for reaching the target states *T* which satisfy the property ϕ. Along with identifying the waypoints, it is also important to select one state from the set of states represented by the guideposts (S^K) for the traversal to the subsequent guideposts. This concept is explained next with an example.

3.2.1 Concept of waypoints

We have designed a system of two counters for explaining the identification of the starting state for the traversal to the next waypoint. The first counter counts in steps of 15 and the second one in steps of 5. The maximum counts allowed for counters 1 and 2 are 150 and 90 respectively. We have another feature in the design according to which the output of the counter 2 will not go beyond 80 if counter 1 is full. The target state (ϕ) is any state which satisfies the following property.

$$EF(count2(c2) == 90)$$

The fully specified hints (i.e., signal with their values) selected for debugging are as follows:

- Initial state (I): count2 = 0
- Waypoint 1 (WP1): count2 = 45

- Waypoint 2 (WP2): count2 = 60

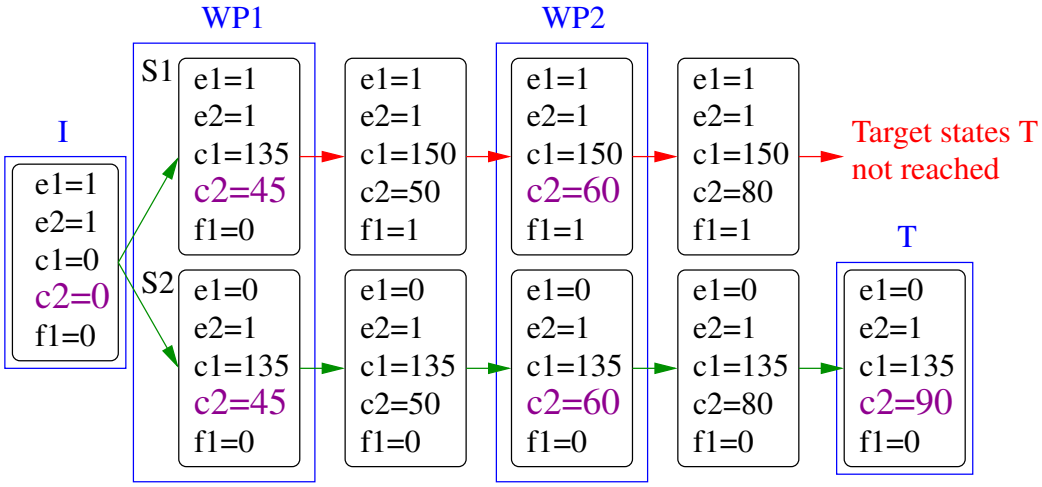
As per our assumption, all the states with $count2 = 45$ will be included in GP1. Let's say there are two states *S1* and *S2* in GP1 (there can be many more states in reality). Note that we did not use any particular criteria in selecting the starting state while traversing to GP2 and selected *S1* arbitrarily. As shown in the Figure 3.2, f1 (status full signal of counter 1) will become 1 just after GP1 and hence *c2* will be stuck at the value 80 (because of the particular nature of the design due to which *c2* will not count above 80 if *f1=1*). Hence, we would never reach any of the target states T (state where *c2=90*) if such a state space traversal strategy is applied.

This problem could be addressed by selecting the right set of signals which can help us identify the starting state from waypoint for further traversals, using Signal Dependency Graphs (SDG) (discussed in Section. 3.3.1). It is clear from the SDG generated for the signal *count2* (Figure 3.4) that the signal *count2* depends on *full_1* (i.e., *f1*) which in turn depends on *en1*. In short, we should have considered the value of the signal *en1* also while picking the states from *S1* and *S2*. This example is just for illustration purposes. However in practice, we should consider all the dependent signals and it's values for better SDG pruning.

3.2.2 Utility of waypoints

An example design with four parallel counters

To further elucidate the advantages in using waypoints, we have designed a system with multi 13-bit counters which has the same clock. The four counters numbered as 1, 2, 3 and 4 count in steps of 1, 2, 3 and 4 respectively. The target property (ϕ) is,



e1, e2: Count enable signals for counters 1 and 2.
c1, c2: Outputs of the counters 1 and 2.
f1: Status full ouput of counter 1.

Figure 3.2: FSM showing Waypoints for the design *counters*

$$EF(output_{any\ counter} >= 8188)$$

We performed model checking for the above design for the property specified above. In the first iteration, we did not apply any waypoint. We have performed model checking using the open source model checker, Yosys [66]. As mentioned in the Table 3.1, the BMC (Bounded Model Checker) tool took 224 seconds to check the property. We experimented with different number of waypoints. Initially, we have selected one waypoint ($output_{any\ counter}$= *8188/2*), which reduced the time to check the property by almost half (115 s). More experiments were carried out with two, three and four number of waypoints. Waypoints for these cases were the nearest integer multiples of 8188/3, 8188/4 and 8188/5 respectively. The reduction in time can be explained from the fact that at each stage of model checking, the state space is getting reduced successively.

Table 3.1: BMC time for different number of waypoints (Counters)

Waypoints	0	1	2	3	4	BMC time (s)
0	224					224
1	58	57				115
2	26	27	27			80
3	16	16	15	15		62
4	10	10	11	10	10	51

3.3 Proposed methodology

The proposed methodology of waypoint identification involves a SDG-based filtering of the internal signals of the design. The primary motive behind this filtering is to obtain a list of signals (used to form waypoints) which would guide the model checking process for counterexample generation in lesser time. Given an error trace, we formulate the property which is to be formally checked. From this property, we obtain the Signal Dependency Graph for the target variables which are further analyzed to select the waypoints. Figure 3.3 shows a high level overview of the proposed methodology which is illustrated in Section 3.5.2.

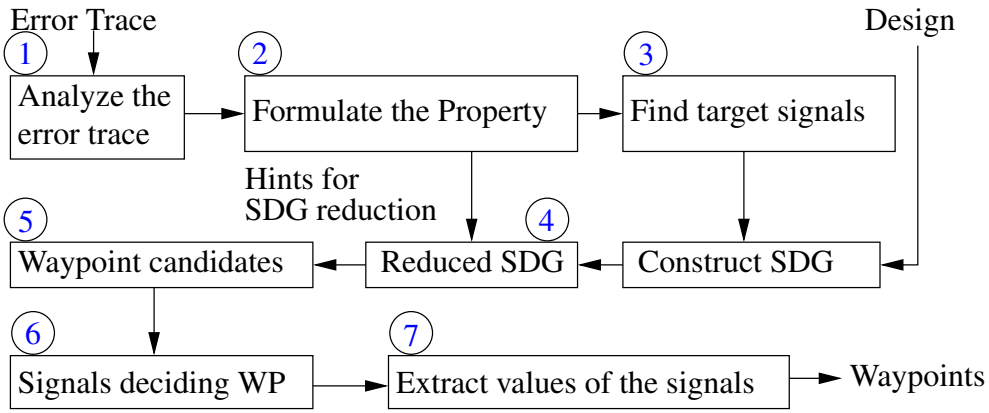


Figure 3.3: proposed methodology of waypoint identification

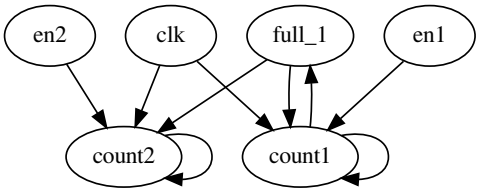


Figure 3.4: Signal Dependency Graph for the signal "count2" (count value of the counter 2) of the design discussed in Section 3.2.1

3.3.1 Signal Dependency Graph (SDG)

SDG is defined as a directed graph (G) with signals (in the RTL description) as Vertices (V) and the dependence between nodes as directed edges (E). If there is a directed edge $v_i \rightarrow v_j$*, it means that the signal* v_j *depends on the signal* v_i*, where* $v_i, v_j \epsilon V$ *and* $(v_i, v_j) \epsilon E$.

To generate the SDG, we first generate the CDFG (Control Data Flow Graph) of all the modules in the design. After that the intra-module dependency is derived by parsing the CDFGs and the inter-module dependencies are extracted by parsing the RTL description. We derive the final SDG by combining both.

3.3.2 Proposed Methodology for Identifying Waypoints

The concept of waypoint is represented in Figure 3.1 in accordance with the terminology adopted in [21]. As it is clear from this diagram, the goal is to identify waypoints (the blue boxes in the Figure 3.1, except for the leftmost and rightmost one which are the initial and target states) which assist in reaching from the reset state (or, otherwise a predefined state) to the target state which satisfies the property.

The proposed methodology takes as input an error trace (ETr) and the RTL design to find the waypoints. These waypoints assist us in finding a counterexample within lesser CPU time as compared to conventional model check-

ing. After construction of SDG for target variables involved in the property, we need to reduce these graphs to prune out the unnecessary regions (paths) in the graphs. From the reduced Signal Dependency Graphs, we identify waypoints based on a heuristic which is basically a path clustering procedure. Similarly, other heuristics which can be explored are the number of input nodes to each one of waypoint candidates ($WPcand$) and the number of toggles in signal values of each one of $WPcand$ in any simulation trace($SiTr$). The proposed methodology is presented here as Algorithm 1. Note that step-6 (which is concerned with selection of effective waypoints) is not fully automated and we repeat the process for few iterations until we achieve the least BMC time. Furthermore, it can be argued that multiple error traces are possible which if not accounted for, may affect this algorithm. However, we observed that multiple error traces lead to the same property formulation in most of the cases as the final error state remains same. It is worth to note the practical usage of the proposed methodology lies in obtaining counterexamples in lesser CPU time.

Algorithm 1: $FindWaypoints$

Input: $ErrorTrace(ETr)$,$Design(D)$

Output: $Waypoints(WP)$

1 analyze the error behavior(ETr) observed in the design(D) simulation;

2 formulate properties(P) related to step1 which are to be given to model checking;

3 select signal(s) from P and construct their Signal Dependency Graph (SDG) using D;

4 reduce the SDG in step3 to generate the reduced SDG (SDG′) with the help of abstraction techniques(modular similarity, bit-width minimization and slicing);

5 identify waypoint candidates ($WPcand$) from SDG′;

6 from $WPcand$, select WP using structural distance-based clustering of signals in the longest *K* paths of SDG′. ;

7 for waypoints of the form sig_i=m, $sig_i \neq n$ or sig_i in the numerical/bit-width range of q to r, D is analyzed to obtain the values of m,n,q,r etc.

3.4 Design descriptions utilized in experiments

Let's discuss all the designs used for the case studies in the subsequent chpaters here.

3.4.1 MESI controller

The first case study in the Chapter 4 is based on a bug in the MESI Coherency Intersection Controller [2] given in Figure 3.5. The MESI-ISC supports MESI coherence protocol [67] which is widely employed for performing coherency maintenance tasks in systems with multiple masters having local caches. Coherence systems are used to ensure consistency between different copies of data with the same memory address when multiple cores with local caches try to access the same memory location. The cache line in any core can have four different states: *M*odified, *E*xclusive, *S*hared, and *I*nvalid.

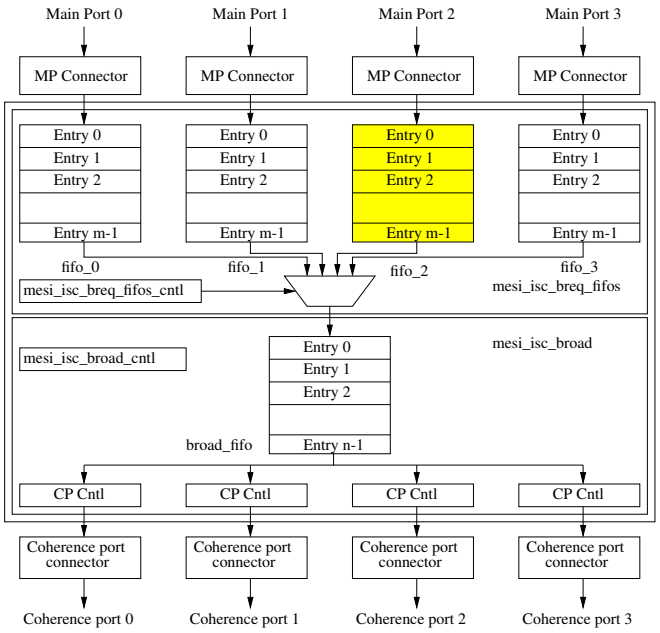


Figure 3.5: MESI-ISC design illustration [2].

There are two buses called the main bus and coherency bus in this imple-

mentation. The transactions on the main bus are *write access*, *read access*, *write broadcast*, and *read broadcast*. They are initiated and driven by the masters. The main bus communicates with the main memory, and the coherency controller (i.e., cache controller). The coherency bus is driven by the coherency controller. The transactions done here are *write snoop*, *read snoop*, *enable write* and *enable read*. There is also a priority logic which decides the order in which the requests from different CPUs are handled. In Chapter 3, we will discuss a bug which we have introduced into this priority logic which results in the miss of all requests from the CPU2. The cache coherence process is initiated with the master requesting memory access. Before that, the master sends a broadcast request to the main memory. Once the request is received, the coherence controller communicates with the other masters and collects their responses. Then the controller allows the initiator to perform the memory access.

Note that the verification of cache coherence protocols is a challenging task [68]. This means that verifying the implementation of such coherence protocols is equally difficult which necessitates verification by formal methods for completeness. Moreover, the subtle complexity of cache coherence protocols makes MESI-ISC a good candidate for case study involving model checking experiments.

An interesting anecdote regarding the wide-spread incompleteness in checking/verification of cache coherence protocol implementations is reported by Komuravelli et al. [69]. The authors discovered many bugs in the high level implementation of MESI coherence protocol in the popular architectural simulator (GEM5) even though the implementation was actively used by a large community of architectural researchers all across the globe for a time period of more than six years. Thus, given that the corresponding RTL implementation is more

detailed and complex in nature, it is highly probable that subtle bugs escape into the RTL design owing to the incomplete verification.

3.4.2 USB 2.0

Universal serial bus (USB) enables serial communication between PC, and it's peripherals. The block diagram of USB 2.0 is given in Figure 3.6. This design

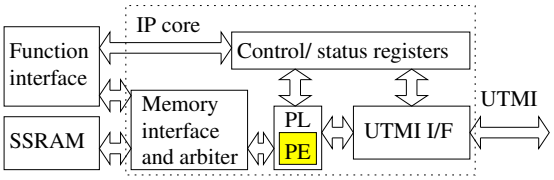


Figure 3.6: USB design illustration [3].

has the following major blocks. Control/status registers: contains all control and status registers required to realize the SIE (Serial Interface Engine) function, PL: handles USB data packets and transactions, UTMI I/F: communicates with external USB 2.0 transceiver, Memory interface and arbiter: provides arbitration over the access to external SSRAM between the SIE and FI (Function Interface). In the second case study given in Section 4.4.2 we have discussed a bug in the PE (Protocol Engine) inside the PL (Protocol Layer).

3.4.3 PCI

The PCI bridge core has two units namely PCI target unit and WISHBONE slave unit each capable of supporting necessary bridging operations from PCI to WISHBONE and vice versa by acting as master and slave. It is equipped with precompiler directives like HOST and GUEST, which is used to configure the bridge as either master or slave. The bug we discuss in the Chapter 4 is inside one module (shown in yellow) in the parity-check block where the parity on the

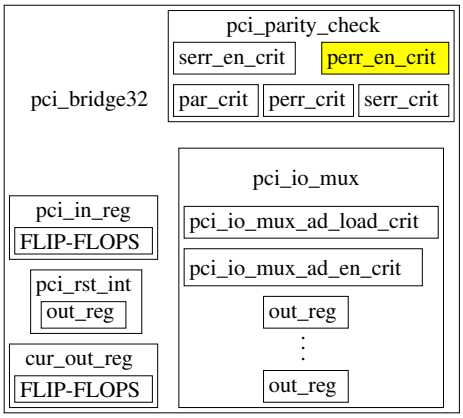


Figure 3.7: PCI design illustration [4].

data is calculated. The bugs in all the case studies are chosen such that they are deep inside the design, thereby making their localization sufficiently difficult.

3.5 Case Study: Bugs and Properties

3.5.1 Properties selected for model checking

We have identified two difficult properties for our experiments. To generate counterexamples, we intentionally injected one subtle design bug (separately during each property checking experiment) so that design functionality deviates from the desired specifications. For bringing out the differences between model checking with and without waypoints, we perform model checking, first by just providing the design and property to the model checker tool and then repeating the same with waypoints.

Bug 1: We have introduced a bug in the *status_full_o*[1] signal of FIFO_2. Because of this bug, the *status_full_o* never becomes high and FIFO_2 will keep accepting new entries even if it is full. The new entries will replace the last stored entry. For this reason, there is a chance that the requests coming from CPU 2 (which is stored in FIFO_2) will get replaced with subsequent requests.

Manifestation of bug: This bug can be noticed by the occasional misses of the requests from CPU2.

[1]The signal which goes high when the FIFO is full.

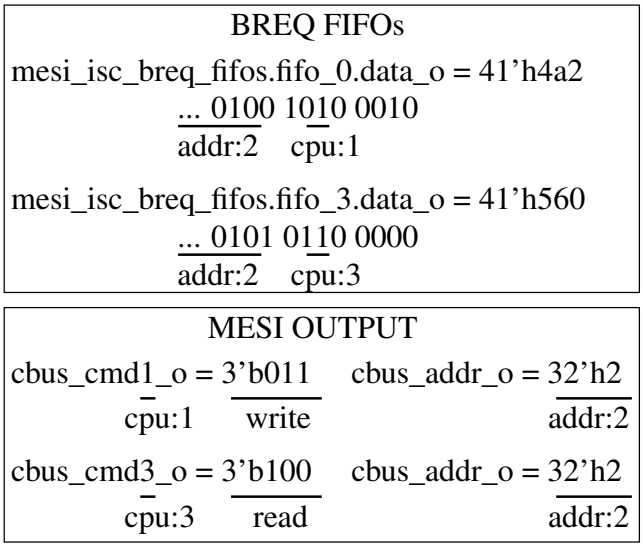


Figure 3.8: Encoding of the signals used in Property 2

Property 1: We have designed a property (as shown in Listing. 3.1) which will expose this bug. In this property we have a request from CPU0 followed by another request from CPU2. After sometime CPU0 sends another request. This request from CPU2 does not reach the BROAD FIFO and hence will be missed at the MESI OUTPUT also. As shown in Tables 3.2 and 3.3 later in this chapter, this property takes few hours for model checking.

Listing 3.1: property1

```
assert property (
not(
// ------------------BREQ FIFOs------------------
// FIFO_0 contains WR req. from CPU0 to addr 0
(mesi_isc_breq_fifos.fifo_0.data_o==41'h83
// FIFO_2 contains WR req. from CPU2 to addr 2
and mesi_isc_breq_fifos.fifo_2.data_o==41'h4c1)
// FIFO_0 contains WR req. from CPU0 to addr 2
[*1:$]##1 mesi_isc_breq_fifos.fifo_0.data_o==41'h483

// ------------------BROAD FIFO------------------
// WR req. from CPU0 to addr 0
##[1:$](mesi_isc_broad.broad_fifo.data_o==41'h83)
// WR req. from CPU0 to addr 2
[*1:$]##1 (mesi_isc_broad.broad_fifo.data_o==41'h483)

// ------------------MESI OUTPUT------------------
// WR req. from CPU0 to addr 0
##[1:$] (cbus_cmd0_o==3'b011 and cbus_addr_o==32'h0)
// WR req. from CPU0 to addr 2
[*1:$]##[1:$] (cbus_cmd0_o==3'b011 and cbus_addr_o==32'h2))
);
```

3.5.2 Identification of Waypoints using the proposed algorithm

We have followed the steps mentioned in the Algorithm 1 to find the appropriate signals which decide waypoints.

1. The *Error Trace* obtained by simulating the design reveals that there could be some problems with the requests from CPU 2. Note that this *Error Trace* is provided by simulators like Modelsim and not the BMC tool.

2. Out of many possibilities where the bug could be, we have shortlisted the area between BROAD FIFO and the primary inputs as the buggy region. A property was designed (Listing. 3.1) to check the possible miss of requests from CPU 2 targeting this area. As per the proposal, other possible buggy regions could be explored if this assumption turned out to be wrong (i.e., if no counterexample is generated for the property by the end of this iteration). It's important to note that if we fail to cover all possible buggy areas for designing the property at this step, we might miss out a region where the actual bug is and end up not catching the bug.

3. We have selected the signal *cbus_cmd0_o* and *cbus_addr_o* as the target variables since they are the ones which are closest to the primary output and hence the SDG (Signal Dependency Graph) corresponding to that will include all the potential signals for waypoints. Selection of these signals from the property is done by analyzing the RTL and identifying the signals which are structurally near to the primary output. An SDG for illustration is given in Figure 3.9. The actual SDGs used for this work are not shown here since they are very large.

4. We used slicing techniques for reducing the size of the SDG generated in the previous step. This step required extra set of hints which are to be used as the slicing criteria. The hint which we used from the *ErrorTrace* is the following: the regions in the design which doesn't deal with the data from CPU 2 could be discarded from the SDG. The slicing of the SDG is done resulting in a modified SDG (SDG′) which does not have the signals related to BREQ FIFO 1 and BREQ FIFO 3.

5. All signals (except for the elementary ones like clock, reset etc.) in the reduced SDG are selected as the potential signals for the waypoints.

6. To identify the candidate signals for the waypoints, we have grouped the signals obtained in the previous step based on the *structural distance* in SDG from the root signal. The *structural distance* is determined by the number of nodes in the path from one node to another (note that we are not assigning weights to the edges in the SDG). We have used each of those signals with their allowed range of values (discussed as the next point) in the property and compared the speedup in the Model Checker. Higher speed-up in model checking time was obtained for the following set of signals (i.e., waypoints):

   - BREQ (Broadcast Request) FIFO becomes full (*status_full*).
   - Data output of the BROAD (Broadcast) FIFO (*broad_fifo.data_o*).

   As mentioned previously, the selection of the signal which decides waypoint is very important. For example, the model checking when done using *status_full* signal, the obtained speedup in CPU time was significant. However, when it was done with *ptr_wr* (the write pointer for the BREQ FIFO 2, which is also there in *WPcand*) the Model checker went for a time out (T.O.).

7. The data values assigned to each signal is chosen using the design knowledge and the property. The final waypoints are as follows: *status_full == 1*, *broad_fifo.data_o == 41'h83* [2]. Here the Guidepost 1 (GP1) is a set of states where the signals *status_full* is equal to *1* and Guidepost 2 (GP2) is a set of states where the signal *broad_fifo.data_o* is equal to *41'h83*. Please note that obtaining these data values itself is a hard problem and we have not addressed it in this chapter. Devising techniques to solve this problem is an open area for future research.

For comparative evaluation, Property1 is checked via the following two ways. The same is repeated for property 2 (corresponding to Bug 2) as well.

1. Without any waypoint.
2. With waypoints (Property is broken down into 3 parts).
   (a) Reach the case where *status_full* of BREQ FIFO 2 is *1*.

[2]For this particular case, WP2 happens to be there in the property as well. We had to use *broad_fifo.data_o* in property 1 to ensure that the final output signals are generated by the requests 41'h83 and 41'h483, and not one of the many other possible requests which might give the same output values for *cbus_cmd0_o* and *cbus_addr_o* at the primary output of the MESI design.

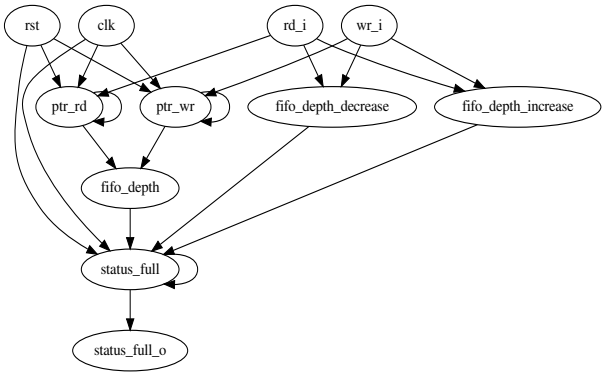


Figure 3.9: Signal Dependency Graph for the signal *status_full_o* in the design *mesi_isc_basic_fifo* of MESI-ISC design [5]

(b) Initialize the signal *status_full* of BREQ FIFO 2 to *1* in the design and check if the data is correctly transferred from all BREQ FIFOs to BROAD FIFO by specifying the value of *broad_fifo.data_o*.

(c) Check for the transfer of data from BROAD FIFO to the primary output of MESI-ISC.

Bug 2 : We introduced another bug in the priority logic of the BROAD FIFO controller of MESI-ISC design. It decides the priority at which the requests from the BREQ FIFOs are passed to the BROAD FIFO. Accordingly, the requests from BREQ FIFO 2 will always be neglected. In terms of implementation, it is done by introducing a simple *AND* gate with one of it's inputs tied to *4'b1011* and the other one connected to the output of the existing priority logic.

Manifestation of bug: Due to this bug, the requests from CPU 2 never reach the ouptut of MESI-ISC resulting in unexpected results (for example, if CPU 2 writes some data to a particular address and another CPU reads it, the data which is read will be the one which existed previous to the write by CPU 2. The effects can be in many other ways.

Listing 3.2: property2

```
assert property(
not(
// ------------------BREQ FIFOs-------------------
// fifo_1 contains WR req. from CPU1 to addr 2
(mesi_isc_breq_fifos.fifo_1.data_o==41'h4a2
// fifo_2 contains WR req. from CPU2 to addr 2
and mesi_isc_breq_fifos.fifo_2.data_o==41'h4c1
// fifo_3 contains RD req. from CPU3 to addr 2
```

```
and mesi_isc_breq_fifos.fifo_3.data_o==41'h560)

// ------------------BROAD FIFO------------------
// WR req. from CPU1 to addr 2
[*1:$]##[1:$](mesi_isc_broad.broad_fifo.data_o==41'h4a2
// RD req. from CPU3 to addr 2
##1 mesi_isc_broad.broad_fifo.data_o==41'h560)

// ------------------MESI OUTPUT------------------
// WR req. from CPU1 to addr 2
[*1:$]##[1:$](cbus_cmd1_o==3'b011 and cbus_addr_o==32'h2)
// RD req. from CPU3 to addr 2
[*1:$]##[1:$] (cbus_cmd3_o==3'b100 and cbus_addr_o==32'h2))
);
```

Property 2: FIFO_1 will write at address 2 and then BREQ FIFO 2 will also write at the same address. Then BREQ FIFO 3 will read the value from the address 2. Since there is a bug in the priority logic, BREQ FIFO 3 will read the value which is written by BREQ FIFO 1 instead of BREQ FIFO 2. As in the case with property 1, there is a significant improvement in time when we use waypoints.

The waypoint used for checking property 2 is following:

- Data from BREQ FIFO reaches BROAD (Broadcast) FIFO. (WP1: *broad_fifo.data_o* with the values *41'h4a2* and *41'h560*).

3.6 Experimental results

We utilized JasperGold model checker (from Cadence) for our experiments with the MESI-ISC design. All the model checking experiments are carried out on Intel-i7 machine running at 3.4GHz with 16 GB RAM and booting on CentOS. The Tables 3.2 and 3.3 discuss the improvements in BMC time and memory usage for the property 1. Similarly Table 3.4 and Table 3.5 discuss the results for the property 2. For property 1, we get a speed up of 11.44x in time. The reduction of memory usage is 2.34x. Moreover, it is important to note that the property 2 goes for a timeout (after running for more than 10 hours) without waypoints (Table 3.4). Using waypoints it got completed in 215.9 seconds. Another set of results for property 2 is available in Table 3.5.

Comparison for bound, time and memory usage is omitted in Table 3.4 since the Model checker goes for timeout (T.O.) for the case without waypoints, leaving us no data to compare against. This case strongly shows the benefits of using waypoints as compared to the original

Table 3.2: Improvements in bound, time and memory usage for property 1 (with BREQ FIFO size 2 and BROAD FIFO size 4)

Method		Bound	Time (s)	Memory (MB)	Bound (reduction)	Time (reduction)	Memory (reduction)
Without Waypoint		53	7947.6	365.28	0	1x	1x
With Way-Point	FIFO_Full	5	0.02	4.25	11	11.44x	2.34x
	BREQ FIFO to BROAD	38	515.6	49.45			
	BROAD FIFO to Final o/p	42	178.9	102.37			
	Total	42	694.52	156.07			

model checking experiment (i.e., model checking without using waypoints). Note that the experimental results presented here consider only the model checking time and not the time taken by the algorithm for the generation of waypoints. Also, the manual effort required in the proposed semi-automatic method of effective waypoint identification is not accounted for in these results.

Table 3.3: Improvements in bound, time and memory usage for property 1 (with BREQ FIFO size 2 and BROAD FIFO size 8)

Method		Bound	Time (s)	Memory (MB)	Bound (reduction)	Time (reduction)	Memory (reduction)
Without Waypoint		50	1370.4	151	0	1x	1x
With Way-Point	FIFO_Full	5	0.03	3.93	4	3.01x	0.81x
	BREQ FIFO to BROAD	40	249.3	124.58			
	BROAD FIFO to Final o/p	46	204.5	56.71			
	Total	46	453.83	185.22			

Table 3.4: Improvements in Bound, Time and Memory usage for property 2 (with BREQ FIFO size 2 and BROAD FIFO size 4)

Method		Bound	Time (s)	Memory (MB)
Without Waypoint		T.O.	T.O.	-
With Way-Point	BREQ FIFO to BROAD	32	215.8	102.30
	BROAD FIFO to Final o/p	18	0.1	16.08
	Total	18	215.9	118.38

Table 3.5: Improvements in Bound, Time and Memory usage for property 2 (with BREQ FIFO size 2 and BROAD FIFO size 8)

Method		Bound	Time (s)	Memory (MB)	Bound (reduction)	Time (reduction)	Memory (reduction)
Without Waypoint		24	2	28.48	0	1x	1x
With Way-Point	BREQ FIFO to BROAD	13	0.4	14.77	6	3.33x	0.86x
	BROAD FIFO to Final o/p	18	0.2	18.12			
	Total	18	0.6	32.89			

3.7 Discussion

The usage of semi-formal methods can ease out the challenges of state space exploration of the designs in many ways. One of them is generating high quality inputs used for functional verification. Plethora of work has been reported in this direction which utilize abstraction techniques in addition to guiding the random simulation-based state space search [17, 49, 70, 71]. Similarly, supplying guidance assists in faster state space traversal also. Yang et al. [9] proposed the idea of a guidance-based search strategy for invariant proving and bug finding. Typically, this scheme is highly successful in discovering subtle design bugs which are missed out by random or constrained-random simulation [7, 14, 72]. Various strategies for generation of guidance for a successful verification methodology based on semi-formal techniques are summarized in [7, 10, 73]. Abstraction is one of the most common ways to glean some hints which can assist in directed state space traversal. Techniques of abstraction are also useful for the minimization of the length of error traces. However, abstraction techniques are generally very useful for circuits dominated by data paths such as arithmetic circuits only. In this regard, guidance-based verification becomes essential for semi-formal/formal verification of circuits dominated by control paths. Note that abstraction can still be used in conjunction with guidance-based methodologies [7, 14, 74].

3.8 Conclusion

This chapter proposed a methodology for identifying guidance hints, known as waypoints for better design debugging by model checking. With the help of waypoints, the state space traversal becomes faster leading to significant speed-up while property checking. With the proposed semi-automatic methodology for the identification of waypoints, we achieved significant speedup during counterexample generation. For large designs, a larger number of waypoints can significantly reduce the achievable gain in model checking time (i.e., CPU time). An extension of this work is a simulation-based strategy which assists the Signal Dependency Graph-based analysis. The methodology involves generation of constrained-random simulation traces from the initial state to the first waypoint, from the first to second waypoint and so on. After careful analysis of these simulation traces, the subset of states corresponding to the intermediate waypoints can be identified automatically.

Chapter 4

Design Debugging with Guidance-based Model Checking

4.1 Introduction

As mentioned in the previous chapters, functional verification continues to occupy a key position in the design implementation process [7, 8, 44]. With significant advancements in verification strategies and emulation tactics to overcome the bottleneck of simulation speed, bug localization is still challenging and needs significant manual expertise [7]. To cope up with time-to-market targets for an integrated circuit (IC) design, automatic bug localization becomes essential so that the design fixes/corrections can be quickly carried out. Given that exhaustive design simulation is rarely possible, formal methods like model checking hold promise in discovering bugs at an early stage. Typically, a counterexample, CEX (which is the output of model checker), provides opportunities to expose the bug more explicitly (in terms of a few signals) compared to a lengthy simulation trace. However, automatic bug localization from tedious counterexamples is not trivial and leads to elongation of the debug step. In the proposed bug localization methodology, we begin with a buggy simulation trace (consisting of only system-level behavior failure) and perform guided model checking of the design with the help of properties written on the assumption of probable broader buggy regions. Model checking of large designs is a challenging task because of different scalability issues requiring assistance of heuristics like bit-width slicing [13, 14]. These heuristics can reduce the computational efforts required for state space traversal [10,13,14], which in turn enhances the feasibility of the model checking exercise. Guided state space traversal [9,10] methodologies have often been utilized to alleviate

some of the obstacles due to scalability issues. We revisit this paradigm of guidance-assisted model checking for automatic bug localization at RTL.

An intelligent selection of the guidance hints/mechanisms in an automatic manner turns out to be an equally challenging problem [14, 16, 21, 75]. Therefore, we propose a methodology to automatically obtain guidance hints such that after the application of these hints, the model checking problem becomes easier. The proposed technique involves usage of simulation-based mining of properties of the design and their subsequent incorporation into Bayesian reasoning for filtering of the most profitable ones. The guidance hints, essentially, act as assistance for bug localization into smaller regions of the design through iterative model checking of properties targeting these regions. These smaller model checking instances turn out to be easier than the original model checking problem. Therefore, with the obtained guidance hints, probable buggy sub-regions can be obtained as candidates for further analysis. After that, through static analysis, RTL lines of the design responsible for the buggy behavior can be identified by utilizing a technique similar to counterexample guided abstraction and refinement (CEGAR) analysis [19]. We do not attempt the bug coverage assessment with the derived properties (written from the buggy region assumptions after observing the high-level system behavior); however, we show in our experiments that the proposed approach indeed succeeds in effective bug localization for different case studies. In practice, writing properties from the assumption of buggy regions of the design is akin to modeling of bugs or faults for the purpose of automatic test generation [23].

The rest of the chapter is organized as follows. Section 4.2 describes in detail the proposed technique (*Orion*) for guidance hint generation along with preliminaries (concepts and associated terminologies) for the proposed methodology. The fine-grained bug localization methodology (*Aquila*) is presented elaborately in Section 4.3. Designs that have been utilized in our experiments are presented in Section 3.4. Section 4.4 and 4.5 describe the case studies (of different designs) on fine-grained bug localization with the proposed methodology. Finally, the chapter is concluded with a few directions on future research in Section 4.6.

4.2 Orion: Proposed methodology for effective guidance hints identification

4.2.1 Guidance hint mining

4.2.1.1 Basic procedure of mining

Similar to the assertion mining techniques described in [76, 77], we utilize simulation traces for mining the assertions. The simulation traces are obtained with the help of either constrained-random or designer-provided testbenches. The mining step can yield a large number of assertions, out of which many may not be relevant to the guided state space traversal problem. So, we supply structural constraints to the miner framework for generating assertions ($\phi_1, \phi_2 ..... \phi_n$). Specifically, our miner framework implements a mining procedure with the help of classification and regression trees. The miner framework generates assertions with the following considerations:

1. From the simulation traces, breadth-first method is used to obtain classification and regression trees. Based on whether the target signal (for assertion ϕ_k) is discrete-valued (i.e., 0 or 1) or continuous- valued, either decision/classification tree or a regression tree is generated, respectively.

2. For regression tree, an additional argument (a threshold on the number of leaves), which forms the stopping criterion for regression tree is required. This hyper-parameter is calculated by a cross-validation method.

3. Assertions are generated by taking the previous value of target signal as features for previous 'T' clock cycles.

4. To calculate the range of temporal information in assertions, it sums up the execution depth (which is defined as RTL static analysis-based distance between nodes in CDFG of the respective modules of the design) of all the signals that appear in the assertion (taking execution depth of previous value of the target feature as zero).

4.2.1.2 Toy example for illustrating assertion mining

Consider a circular queue of capacity 4, which gives the simulation trace for a few clocks as shown in Table 4.1.

Table 4.1: Sample simulation trace for the toy example.

cycle	read/$\overline{\text{write}}$	full	empty	front	rear
1	0	0	1	5	5
2	1	0	0	1	1
3	0	0	1	5	5
4	0	0	0	1	1
5	0	0	0	1	2
6	0	0	0	1	3
7	0	0	0	1	4
8	0	1	0	1	4
9	0	1	0	1	4
10	1	1	0	1	4
11	1	0	0	2	4
12	1	0	0	3	4
13	1	0	0	4	4
14	0	0	1	5	5

Suppose, $rear$ is the target feature for which assertions have to be generated. We take the following assumptions into account.

1. Execution depth of *read/write*, *full*, *empty* and *front* is 1. Past values of $rear$ (for e.g., $rear[t-1]$, $rear[t-2]$), etc. have an execution depth of 0.

2. We will consider only two past values (T = 2) of $rear$ i.e., $rear[t-1]$, $rear[t-2]$.

For the above simulation trace the modified trace for generating temporal assertion becomes as shown in Table 4.2. The parameter fitting function gets 3 as the optimum number of leaves as the threshold in the classification and regression tree shown in the Figure 4.1. This threshold forms the base criterion to stop further splitting.
In the above tree, solid nodes represent the nodes where splitting is done using some feature and hollow nodes represent the leaf nodes, to which some label is assigned. An assertion is formed by traversing the path from root upto leaves and merging the clauses if the same splitting feature appears in the path more than once. For e.g., consider the path with the following edges,

$$(rear[t-1] \leq 4), (front \leq 4), (rear[t-1] \leq 2)$$

Table 4.2: Modified simulation trace for the toy example.

read/$\overline{\text{write}}$	**full**	**empty**	**front**	**rear[t-1]**	**rear[t-2]**	**rear**
0	0	1	5	1	5	5
0	0	0	1	5	1	1
0	0	0	1	1	5	2
0	0	0	1	2	1	3
0	0	0	1	3	2	4
0	1	0	1	4	3	4
0	1	0	1	4	4	4
1	1	0	1	4	4	4
1	1	0	2	4	4	4
1	0	0	3	4	4	4
1	0	0	4	4	4	4
0	0	1	5	4	4	5

It generates the following assertion (ϕ)

$$(front \leq 4)\,\&\&\,(rear[t-1] \leq 2)\,|->\#\#[0:1]\,(rear[t] == 2)$$

where [0:1] indicates that the assertion can be true with a 0 to 1 cycle delay. This 1-cycle delay is calculated by adding up the execution depths of all the unique atomic clauses that appear in the path of an assertion from root to leaf. So, execution depth of only $front$ is added because $rear[t-1]$ has an execution depth of 0 with respect to $rear[t]$. After the mining of assertions, they are fed to a model checker for removing the spurious ones.

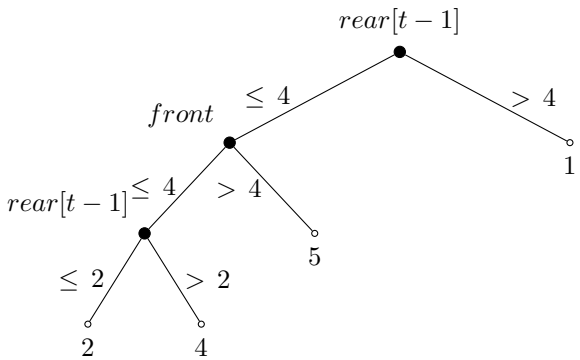


Figure 4.1: Classification and Regression Tree (CART) illustration.

4.2.2 Application of Bayesian analysis

A Bayesian model can be utilized to reason about the conditional occurrence of a series of events [78–80]. This type of modeling is ideally suited for the problem of guidance hint selection because it is very difficult to provide guarantees on the usefulness of guidance hints (H_{ij}) from the mining step alone since a large number of hints given by the miner might not guide the model checker in the right direction. Hence, we aim to capture the usefulness guarantee of the guidance hints in a probabilistic manner and rank them for the final selection to assist the model checking of the property P and subsequent bug localization procedure. The Bayesian analysis part in our framework is motivated by the techniques in [78] and [80].

4.2.3 Toy example for Bayesian Modeling

To explain the application of Bayesian method in the proposed methodology, we consider a design with four independent 13-bit counters $counter_1$, $counter_2$, $counter_3$, and $counter_4$, which count in the steps of 1, 2, 3 and 4 respectively. This example is similar to the one discussed in the Section 3.2.2 used for explaining the utility of waypoints. The counters have enable signals depicted by en_1, en_2, en_3, and en_4, respectively. The target property under consideration is: $P = EF(output_{counter_i} >= 16'h1ff8)$ for i ∈ {1,2,3,4}. Let's consider that a part of this property P_α represented by $output_{counter_4} >= 16'h1ff8$. P_α is shown as S_4 in Figure 4.2, which shows the Bayesian network for experiments on this design. The guidance hints used in this example are shown in Table 4.3. Although we performed analysis with a few other guidance hints as well, we present here the interesting case of two such hints: $output_{counter_4}$ and en_4. The former is the output of the fastest counter (i.e., counter 4) while the latter is the enable signal (en_4) for the same counter.

Table 4.3: Nodes in the Bayesian network for the toy example.

Node	H	Signal with value
S_0	H_{11}	$output_{counter_4}$ = 16'h1f90
S_1	H_{12}	en_4 = 1
S_2	H_{21}	$output_{counter_4}$ = 16'h1fb0
S_3	H_{22}	$output_{counter_4}$ = 16'h1fd0
S_4	P_α	$output_{counter_4}$ = 16'h1ff8

By simulating the design with a constrained-random testbench, we calculate the conditional probabilities for the edges (i.e., $S_j \rightarrow S_i$). We obtained the conditional probability values and subsequently fed it to the Bayesian network (shown in Figure 4.2) for analysis.

For illustrating the calculation of conditional probability values from the simulation trace, let's take the example of the node S_2. From the structure of the Bayesian network, we extract the incoming nodes to S_2 (which are S_4 and S_3) and construct the Table 4.4 with the input nodes in topological order. From the table, we find out the input combinations which need to be traced from the simulation trace. The first two columns of Table 4.4, respectively, represent whether S_4 and S_3 are true (1) or not (0). The remaining column depicts the conditional dependency on S_4 and S_3 in case of S_2 becoming false (0) and true (1) respectively[1]. The conditional probability entries in the column S_2 of Table 4.4 (calculated using the method which will be discussed in Section 4.2.4) are fed to the Bayesian network ($BModel$) as inputs.

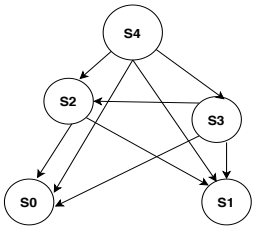


Figure 4.2: Bayesian model ($BModel$) for the toy example.

Table 4.4: Conditional probabilities for the node S_2.

S_4	S_3	S_2	
		0	1
0	0	1	0
0	1	0.004	0.996
1	0	0.009	0.991
1	1	0.004	0.996

The first column of Table 4.5 denotes different log-likelihood values obtained as a result of Bayesian analysis.

From the respective final queries,[2] S_0 is selected as a more useful guidance hint than S_1. Note that this toy example discusses only step-1 of the two-step Bayesian filtering discussed in

[1]Values in rows 4 and 6 are the same since S_4 and S_3 occur too close to each other in this particular design.

[2]The process of calculating a particular probability distribution from the network is referred to as querying.

Table 4.5: Queries from the Bayesian model for the toy example.

Likelihood	Value
ll (S_0,S_2,S_3,S_4)	1.6307
ll (S_1,S_2,S_3,S_4)	1.6203

the first part of our proposal (Section 4.2) for simplicity.

4.2.4 CPD calculation details

As mentioned earlier, the conditional probabilities for each of the edges in the Bayesian network is calculated from the simulation trace. The simulation trace is first divided into frames of size *frame_size*. It is a parameter called that can be modified. For our experiments we used a value of 1000. The toy example shown in Section 4.2.3 uses a signal-value combination as the node. It is important to note that in the proposed framework, all the nodes of the Bayesian Network are assertions and not signal-value combinations. The CPD calculation is done using an in-house simulation trace analyzer, which looks for the assertions corresponding to the start and the end nodes of the particular edge of the network. The time interval between them is calculated for all occurrences of this edge in all the frames of the simulation trace ($SiTr$). After that, we calculate the minimum time interval among all those and divide it by the *frame_size* to get the conditional probability corresponding to that edge. Instead of the minimum time interval, it could have been the average time interval also. For Bayesian model implementation, we utilized the BNT toolbox [81] in MATLAB.

4.2.5 Effective Guidance Hints Generation

As stated previously, we develop a methodology (referred to as $Orion$) for the identification of effective guidance hints.[3] An overview of the complete framework is given in Figure 4.3. With this methodology, we obtain filtered guidance hints which is to be utilized during the bug localization step.

The identification of better quality guidance hints begins with designing a property (P) assuming a broader buggy region which can include multiple modules or regions in the design.

[3]We derive this name from the fact that Orion is a prominent constellation in the night sky, often acting as a guide during sky-walks.

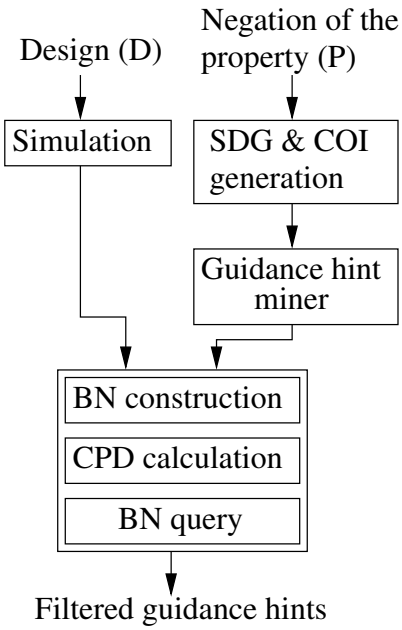


Figure 4.3: Overall flow of proposed methodology for hint generation ($Orion$).

The negation of this property will generate a counterexample if the assumption on broader buggy region is correct. This region is shortlisted based on the preliminary analysis of the erroneous simulation trace[4]. The steps in Figure 4.3 are outlined below.

- Constraint random simulation of the design targeting the regions, which are considered to be the potential broader buggy regions while writing the property.
- Extraction of the COI (Cone Of Influence) signals for the signal in the property, which is closest to the primary output of the design. This signal is referred to as root signal in the algorithm. It is extracted from a Signal Dependency Graph (SDG), an example of which is given in Figure 3.9. Feed the signals in the COI as constraints to the *guidance hint miner* so that it can mine assertions (ϕ) for those signals from the simulation trace.
- Filtering the mined hints using Bayesian analysis.

In accordance with the illustration in Section 4.2.2, Bayesian modeling can be done by constructing a network where nodes depict events of interest, and edges represent the conditional dependency among them. In our framework, these nodes are assertions (either the mined ones, ϕ_k, or parts[5] of property, P_α). Bayesian analysis involves the following steps for pruning

[4]The erroneous simulation traces are identified based on the deviation from the expected behavior mentioned in the design specification.

[5]These are parts of the property obtained by decomposing P into smaller properties/assertions without altering its temporal meaning.

out (rejecting) the ineffective guidance hints.

Step-1: Bayesian network with good guidance hints at the next level.

- At any current level, construct a Bayesian network with the guidance hints at that level and the selected/shortlisted hints (also referred to as good hints) for the next level (the filtering for which has already been done using a similar process) along with parts of the main property.[6]

- Obtain the *ll* (log-likelihood) values for each of the guidance hints at the current level as shown below.

  ll (H_{ij}, $H_{(i+1)j}$,P_α): log-likelihood of a guidance hint H_{ij} (j^{th} guidance hint of i^{th} level) along the good guidance hint(s) of the next level $H_{(i+1)j}$ (j^{th} guidance hint of $(i+1)^{th}$ level) and P_α becoming true. The ll value actually gives the likelihood of the occurrence of a path from the current guidance hint to the main property through the immediate next guidance hint. The guidance hint(s) with the higher *ll* values are shortlisted for further processing in the step-2.[7]

The objective of step-2 is to eliminate those hints which might be taking a path to the target state, which is not through the good guidance hints at the next level.

Step-2: Bayesian network with bad guidance hints at the next level.

- At any current level, construct a Bayesian network with the guidance hints at that level and the rejected hints (also referred to as bad hints) for the next level along with parts of the main property.

- Obtain the *ll* (log-likelihood) values for each of the guidance hints at the current level (which were selected in step-1) as shown below.

  ll (H_{ij}, $H'_{(i+1)j}$): log-likelihood of a guidance hint (H_{ij}: j^{th} guidance hint of i^{th} level) along the bad guidance hint of the next level ($H'_{(i+1)j}$: j^{th} guidance hint of $(i+1)^{th}$ level) becoming true. We select the hints with lower *ll* values in this step as the good candidates for guidance hints at the current level. This is because the hints with higher *ll* values will mislead the model checker to pass through the bad hints at the next level, which is against our expectations.

[6]Henceforth *main property* is referred to as the property, which is written assuming a broader buggy region for $Orion$. Negation of this property gives a CEX if the assumption about the broader buggy region is correct.

[7]We might require more than one guidance hint at each level based on the size of the design.

The methodology is presented in a high-level manner as Algorithm 2. This algorithm internally utilizes two minor algorithms, $GetStructuralConstr(P, D)$, for the computation of structural constraints and $BuildRegClassTree$ for the construction of the tree used in assertion mining. Step 4 utilizes model checking for the removal of probable spurious assertions obtained as a result of the mining step.

Algorithm 2: $Orion$: Guidance hint generation.

Input: P $(Property)$, D $(Design)$, $SiTr$ $(Errortrace)$

Output: $GuidanceHints$

1 $Constr \leftarrow GetStructuralConstr(P, D)$;

2 $RegClassTree \leftarrow BuildRegClassTree$ $(SiTr, Constr)$;

3 $MineProp \leftarrow$ Mine $RegClassTree$;

/* Get assertions in SVA format. */

4 $FinalProp \leftarrow SpuriousRemoval(MineProp)$;

5 $BNodes \leftarrow FinalProp$ and P;

6 $EdgeVal \leftarrow getCPD$ $(BNodes)$;

/* Compute conditional probability distribution (CPD) values for edges constituted by *nodes* from $BNodes$. */

7 $BNet \leftarrow ConstructBNet(BNodes, EdgeVal)$;

/* Construct Bayesian network (BN) for the current level. */

8 $GuidanceHints \leftarrow BNqueryStep2(BNqueryStep1(BNet))$;

Note that the guidance hints (H_{ij}) essentially correspond to the mined assertions (ϕ_k) obtained from our guidance hint generation strategy.

4.3 Aquila: Proposed methodology for fine-grained bug localization

4.3.1 Description of methodology steps

We utilize the guidance hints for effective bug localization in an automatic manner. Specifically, we aim to localize the buggy RTL line(s) in the design description. An overview of the

proposed methodology, $Aquila$ is presented in Figure 4.4.[8] In this technique, we use iterative model checking by modifying the target property (P) using certain selected signals from the counterexample traces of the failing properties.

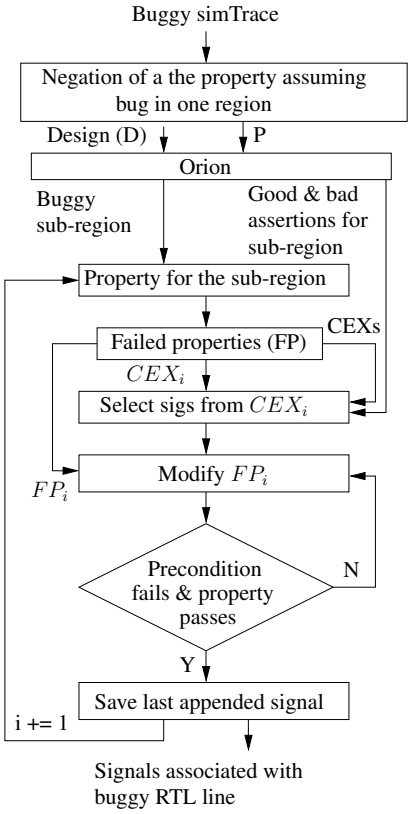


Figure 4.4: Overview of the bug localization methodology ($Aquila$).

In the guidance hint generation methodology ($Orion$) discussed in Section 4.2 we consider a large region (typically consisting of a few modules/a part of one module in some cases) as the potential broader buggy region. Then we write the properties (i.e., set of P) which would pass (i.e., the negation of it would generate a counterexample) if the assumption about the broader buggy region is correct. It is worth noting that in $Orion$, we do not consider each possible bug in the design and write property targeting that. It is not practically possible. Instead, we consider broader regions in the design, the outputs from where become buggy due to many potential/probable bugs in that region. In summary, we consider the manifestation of many possible bugs in one large region in a combined manner. Once we obtain a counterexample for the negation of such a property using the guidance hints at different levels, we proceed to

[8]We name our methodology as "Aquila" since this constellation is surrounded by a large number of dim clusters and nebulae, yet it retains its presence in a prominent way. Analogously, the particular buggy sub-region is surrounded by a large number of unimportant regions (i.e., lines in the RTL description).

$Aquila$ portion of our framework for localizing the RTL line(s) where the bug is present.

Through the usage of $Orion$ methodology, we obtain the module/blocks of RTL lines containing those guidance hints as the sub-region on which $Aquila$ works for localizing the buggy RTL line(s). Therefore, the inputs to $Aquila$ technique are the following:

- The sub-region of the design (in our case studies, we have considered the modules), which contains the potential buggy RTL line(s).
- The good and bad[9] guidance hints in that sub-region.

There will be many sub-regions in the design corresponding to the partitions in the main property decided by the guidance hints shortlisted by the $Orion$ algorithm[10]. To prioritize the sub-regions which $Aquila$ should look at, we consider speedup in the CEX generation time when the hints at different levels are used in $Orion$. The sub-regions corresponding to those hints with maximum speedup is given initial preference for $Aquila$ algorithm. The rationale behind this selection process is that if the CEX generation is faster in $Orion$ using certain hints, then those hints will be closer to the bug and the sub-region where the hint belongs to might have the bug.

The first step in $Aquila$ methodology is to use the standard Formal Property Verification approach and write properties for the good design targeting the above-mentioned sub-region. Some of those properties would fail because of the bug. If no property fails, then we were either wrong about the assumption of the potential buggy sub-region given to $Aquila$ or did not write enough properties targeting this sub-region. If it is the first case, we need to go back to the $Orion$ step and select another sub-region corresponding to guidance hints at a different level. Our implementation requires the properties to be written in the sequence form (Eg: $(p == 1)|=> (q == 2)|=> (r == 3)$). We select one of those properties which failed and modify it with the negation of a signal-value combination selected from the CEX (for example, ppend with ~$(t == 1)$ if the CEX trace has $(t == 1)$ in it) at the particular cycle where that signal value combination occurs. The choice of signal from the CEX trace is discussed in Section 4.3.2. The appending of the signals is done backward from the last

[9]The notion of good and bad is linked with the possibility of the guidance hint speeding up/down the model checking process.

[10]The sub-regions corresponding to the partitions in the property are currently selected based on the location of signals in the RTL, present in the partition.

cycle where the sequence fails (i.e., where the model checker gives *all preconditions*[11] *pass, property failed* scenario). If we consider the case where $(t == 1)$ appeared one clock before, the modified property will be $(p == 1)| => ((q == 2)\&\& \sim(t == 1))| => (r == 3)$. If, during model checking of this property, the precondition 2 fails, it means that the bug/effect of the bug appears in the signal t.

With the knowledge that precondition 2 fails, we can succeed in fine-grained bug localization compared to the case where we had only the original property, when we just knew that the effect of the bug appears in the signal r. For many designs, it is very much probable that there will be a large number of clock cycles between the clocks at which the precondition 1 passes, and the last condition $(r == 3)$ fails. In those scenarios, it is not practical to attempt appending signals for all those intervals. Hence, it is important to decide the number of cycles backward we should go before moving to the next signal from the CEX trace. It can be decided based on the nature of the design. For our first case study, MESI-ISC design, we have fixed this as 5 clocks considering the depth of the FIFO in the design along with other parameters.

The $Aquila$ algorithm in Figure 4.4 has a parameter *i* which denotes the number of properties tried out for the sub-region. Note that, in our case studies given in Section 4.4, we have shown only one property for the selected sub-region.

4.3.2 Selection of the signals from CEX trace

The following steps are to be adopted for selecting the signals from the CEX trace. Let us consider the same failed property sequence $(p == 1)| => (q == 2)| => (r == 3)$ discussed above for illustration purposes.

- Select the signals within $max\text{-}COI\text{-}level$[12] distance (from the signal r) in the COI of the signal r.
- Remove the signals present only in the assertions corresponding to bad hints. This step ensures that we do not waste time on signals which are not aiding the model checking process in $Orion$. These hints are supposed to be not influenced by the bug.
- Find the common patterns in signal-value combinations along with time-stamps across all

[11] precondition 1: $(p == 1)$, precondition 2: $(q == 2)$ for this example.

[12] It's a parameter which indicates how deep in the Cone-of-Influence COI we should explore for signals to be added to the failing property.

those CEXs corresponding to the other failed properties for the current sub-region. Then give first preference to signals in those patterns.

The *Aquila* methodology is presented as Algorithm 3. Comments are added whenever important steps are mentioned.

4.3.3 Toy example for illustrating bug localization methodology

Let's consider the design description provided in Listing 4.1.

Listing 4.1: Example module for the illustration of *Aquila*.

```
module top(clk, in1, s, out);
   input clk, in1, s; output out;
   reg out, r1; wire w1;
   mod1 m1(.clk(clk),.a(in1),.b(w1));
   always @(posedge clk) if (s) out = w1; else out = 1'b0;
   always @(posedge clk) r1 = ~w1 ;
endmodule
module mod1(clk, a, b);
   input clk, a; output b; reg b, d;
   always @(posedge clk) b = a ; d = ~a;
   //original line: always @(posedge clk) b = ~a ; d = a;
endmodule
```

This design has a bug and the equivalent non-buggy line is shown as a comment (beginning with "//"). Using the specification used to write this code, we write the following property (P), which should pass if there is no bug.

$$(in1 == 1)\&\&(s == 1)| => \#\#1(out == 0)$$

When we attempt model checking P, it gives us the counterexample with a trace spanned over 3 cycles giving $in1 == 1$ and $s == 1$ in the first cycle and in third cycle signal $out == 1$. Clearly, this is a violation of the property (P) described above.

Using steps discussed in Section4.3.2 we have selected the signals to be used to modify the preconditions in the property. The different values of the signals in CEX are the following

- Cycle 1: in1=1, s=1, a=1
- Cycle 2: b=1, w1=1
- Cycle 3: out=1

Algorithm 3: *Aquila*: Bug localization.

Input: D, P, $sub\text{-}region$, $good\text{-}hints$, $bad\text{-}hints$, $max\text{-}COI\text{-}level$.

Output: $suspectSigs$.

$FP \leftarrow$ Failed property set;
for *each* FP_i *in* FP **do**
    /* CEX analysis to find common sig-value combinations. */
    $FP_i \leftarrow i^{th}$ failing property;
end
/* Find signals related to buggy RTL line */
for *each* FP_i *in* FP **do**
    $CEX_i \leftarrow$ Counterexample for FP_i;
    /* Select signals from CEX_i */
    $sigList_i = ()$
    for *each* s *in* CEX_i **do**
        $s \leftarrow$ A signal present in CEX_i;
        if s *not only in* $bad\text{-}hints$ **then**
            $sigList_i.append$(s);
        end
    end
    /* Consider signals which are at a distance $max\text{-}COI\text{-}level$ from the root signal in P used to construct the COI. */
    for $0 <$ *sigLevel* $< max\text{-}COI\text{-}level$ **do**
        for *each* s *in* $sigList_i$ **do**
            $FP_i' \leftarrow FP_i$ modified using s;
            modelCheck(FP_i');
            if *precondition fails* **then**
                $suspectSigs$.append(s);
            end
        end
    end
end

In the next step we augmented original property with the selected signals (a, $w1$, etc.) from CEX one by one and checked for the cases where the precondition fails. The modified property for the case when the signal a is considered is,

$$(in1 == 1)\&\&(s == 1)| => \tilde{}(a == 1)| => \#\#1(out == 0)$$

Since the precondition fails and the property passes for this modified property, we conclude that the bug is related to a or the effect of the bug appears in a. We investigate the RTL lines where a is present and localize the buggy RTL line. If the bug was not found, we would have to perform a secondary analysis on the signals which decide the value of a and so on. The first case study given in Section3.4.1 is an example where such a secondary analysis is required to find the buggy RTL line, whereas the other two case studies (Section 3.4.2 and 3.4.3) expose the buggy RTL line in the primary analysis itself.

4.4 Case studies

4.4.1 MESI-ISC

4.4.1.1 $Orion$ (guidance hint mining)

The bug: The bug we discuss here is in $fifo\_2$, which stores the requests from the CPU-2. The original (non-buggy) behavior of the design is such that if the FIFO is full, then the new input data along with the write enable signals (wr_i) will not be replacing any of the already stored entries. Due to the bug, the new $data\_i$ values which come along with wr_i would replace the entry at the top of the FIFO, leading to a loss of the stored data.

Buggy RTL line: $else\ if\ (wr\_i)$

Original RTL line: $else\ if\ (wr\_i\ \&\ !status\_full)$

The manifestation of the bug: In the simulation trace it is observed that the signals corresponding to certain requests from the CPU-2 *occasionally*[13] are not generated at the output of the MESI-ISC design.

The assumption about the broader buggy region: The debug engineer who is not aware of the bug has a lot of assumptions on what could have gone wrong in the design. The bug could be in the $breq\_fifo$ control logic, $broad\_fifo$ (inside the request broadcast logic), the priority logic (which prioritizes the requests), etc. By following proposed approach, we will consider

[13]It does not always happen since this anomaly appears only in those cases where $fifo\_2$ is full.

one such region at a time. The broader buggy region considered here includes $fifo\_2$ and $broad\_fifo$. Based on the assumption that this region is buggy, we have designed the property given in Listing 4.2. For this property we consider two *levels* of guidance hints.[14] In this case study we will show how to identify the guidance hints for the first level. The guidance hints are calculated in the reverse order. First, the hints corresponding to the higher levels, which are closer to the target state where the property fails, are calculated and then backward. We assume that the hint for the second level is already found using a similar process and is incorporated into the property, as shown in Listing 4.3. All further steps refer to that property.

Listing 4.2: Original property for MESI-ISC.

```
assert property(not(
(fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
##[1:8](fifo_2.data_i==41'h4c5 and fifo_2.wr_i==1)
##[1:8]((fifo_2.data_i==41'h8c9 and fifo_2.wr_i
and fifo_2.data_o==41'h4c1)
##[1:4](fifo_2.data_o==41'h8c9))
##[1:16]((broad_fifo.data_o==41'h4c1)
and (cbus_addr_o ==32'h2)##2(cbus_cmd2_o==3'b100))
##[3:20]((broad_fifo.data_o==41'h8c9)
and(cbus_addr_o==32'h4)##2(cbus_cmd2_o==3'b100))));
```

Listing 4.3: Original property for MESI-ISC with waypoint assertions at level-2.

```
assert property(not(
(fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
##[1:8](fifo_2.data_i==41'h4c5 and fifo_2.wr_i==1)
##[1:8]((fifo_2.data_i==41'h8c9 and fifo_2.wr_i
and fifo_2.data_o==41'h4c1)
##[1:4](fifo_2.data_o==41'h8c9))
(broad_fifo.entry[1]=41'h4c1)
##[1:16](broad_fifo.entry[0]==41'h8c9)
##[1:16]((broad_fifo.data_o==41'h4c1)
and(cbus_addr_o==32'h2)##2(cbus_cmd2_o==3'b100))
##[3:20]((broad_fifo.data_o==41'h8c9)
and(cbus_addr_o==32'h4)##2(cbus_cmd2_o==3'b100))));
```

Structural constraints generation: From the property, the signals closer to the primary output in the design are selected for deriving the constraints. These are a set of signals which are supplied as input constraints to the guidance hint miner. The signal selected for this example is $cbus\_cmd2\_o$. We have generated the COI (Cone Of Influence) signals from the SDG (Signal Dependency Graph) for this signal and the signals present in the COI are given as constraints to the miner.

[14]This decision on the number of levels is based on the size of the broader buggy region.

Some of the level-1 waypoint assertions (guidance hints) mined from the simulation trace are shown in Table 4.6.

Table 4.6: Level-1 assertions mined from simulation trace (MESI-ISC).

No:	Assertion
H_{11}	`fifo_2.wr_i == 1 and fifo_2.ptr_wr == 0 ##1 fifo_2.ptr_wr == 0`
H_{12}	`fifo_2.wr_i == 1 and fifo_2.ptr_wr == 1 ##1 fifo_2.ptr_wr == 0`
H_{13}	`fifo_2.wr_i == 1 and fifo_2.ptr_wr == 1 ##1 fifo_2.ptr_wr == 1`
H_{14}	`fifo_2.rd_i == 1 and fifo_2.ptr_rd == 0 ##1 fifo_2.ptr_rd == 1`
H_{15}	`fifo_0.wr_i == 1 and fifo_0.ptr_wr == 0 ##1 fifo_0.ptr_wr == 0`
H_{16}	`fifo_0.wr_i == 1 and fifo_0.ptr_wr == 1 ##1 fifo_0.ptr_wr == 0`
H_{17}	`fifo_0.wr_i == 1 and fifo_0.ptr_wr == 1 ##1 fifo_0.ptr_wr == 1`
H_{18}	`fifo_0.rd_i == 1 and fifo_0.ptr_rd == 0 ##1 fifo_0.ptr_rd == 1`

The good and bad level-2 guidance hints are shown in Table 4.7.

Table 4.7: Level-2 assertions used in filtering process.

Type	Assertion
Good (H_{21})	`broad_fifo.data_o == 41'h4c1 ##2 broad_fifo.data_o == 41'h4c9`
Bad (H'_{21})	`broad_fifo.data_o == 41'h4c1 ##2 broad_fifo.data_o == 41'h4c5`

The parts of the main property, converted into the form of an assertion[15], that is used in the Bayesian analysis is the following:

```
cbus_addr_o == 32'h2 ##0 cbus_addr_o == 32'h2
```

The results of the Bayesian filtering process is shown in Table 4.8. The selected assertion (for level-1) when embedded into the property mentioned in Listing 4.3 is given in Listing 4.4.

Listing 4.4: Property for MESI-ISC with waypoint assertions at level-1 and level-2.

```
assert property(not(
(fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
##[1:8]((fifo_2.data_i==41'h4c5 and fifo_2.wr_i==1)
and(fifo_2.wr_i==1 and fifo_2.ptr_wr==0
##1 fifo_2.ptr_wr==0))
##[1:8]((fifo_2.data_i==41'h8c9 and fifo_2.wr_i
```

[15]Since every node in the Bayesian network is an assertion.

Table 4.8: Likelihood values for two-step filtering of waypoints.

Guidance Hints (H)	ll at step-1	ll at step-2	Selected/rejected
H_{11}	0.2738	0	*Selected*
H_{12}	0.2109	0.2109	Rejected in step-2
H_{13}	0.2479	0.2368	Rejected in step-2
H_{14}	0.2738	0.2738	Rejected in step-2
H_{15}	0	NA	Rejected in step-1
H_{16}	0	NA	Rejected in step-1
H_{17}	0	NA	Rejected in step-1
H_{18}	0	NA	Rejected in step-1

```
and fifo_2.data_o==41'h4c1)
##[1:4](fifo_2.data_o==41'h8c9))
(broad_fifo.entry[1]==41'h4c1)
##[1:16](broad_fifo.entry[0]==41'h8c9)
##[1:16]((broad_fifo.data_o==41'h4c1)
and (cbus_addr_o==32'h2)##2(cbus_cmd2_o==3'b100))
##[3:20]((broad_fifo.data_o==41'h8c9)
and(cbus_addr_o==32'h4)##2(cbus_cmd2_o==3'b100))));
```

The model checking results for the guidance hints selected during the Bayesian filtering along with some of the hints which are not selected are shown in Table 4.9. These results are generated using Jaspergold model checker from Cadence in a PC with Intel Core i7-7700 CPU @3.60GHz x 8 processor running on Ubuntu 18.04 LTS with a RAM of 8GB.

Table 4.9: Model checking results for MESI-ISC with different hints.

Guidance Hints (H)	Time for model checking (s)
Original property without any hints	7303
H_{11}	5844
H_{12}	T.O. (Terminated at 15,544 s)
H_{13}	T.O. (Terminated at 17,748 s)
H_{14}	T.O. (Terminated at 16,8888 s)

Note that with the hint H_{11}, we are able to complete the model checking in lesser time compared to the time taken by the original property. Though it's desirable, it is not always guaranteed that we obtain gain in CPU time during model checking the property with assistance

of guidance hints. We observed in our experiments for a wide range of properties that CPU gain is a strong function of the properties under consideration and the guidance hints obtained with our methodology. In the other two case studies, we observed a slight increase in CPU time when model checking of properties is attempted with assistance from the guidance hints. Since the next part ($Aquila$) requires a comparison of model checking time between different hints at a level and not with the original property; this is not a concern for the current work.

4.4.1.2 $Aquila$ (bug localization)

Using the steps to select signals from the CEX we have arrived at the signals given in Table 4.10 for this property. For those signals, we find the values present in various clock cycles. Then use those signal-value combinations at the appropriate places in the property sequence and see whether we obtain a case in which preconditions fail. The model checking results for the properties modified using these signals are also shown in Table 4.10. The Listings 4.5 to 4.11 show the original property written for good design and various versions of it after appending the signals with IDs 1-6. Note that as per the CEX signal selection methodology discussed in Section 4.3.2 the signal ptr_rd (with signal ID 3) should not be considered for the bug localization process. However, we have shown it in Listing 4.8 just to illustrate that it does not help in localizing the bug.

Table 4.10: Model checking results for MESI-ISC with different signals from CEX.

ID	Selected signal-value combinations	Model checking status
1	`fifo_2.fifo_depth_decrease==1`	Preconditions fail, property passes. Selected to find buggy lines.
2	`fifo_2.fifo_depth_increase==0`	Preconditions fail, property passes. Selected to find buggy lines.
3	`fifo_2.ptr_rd==0`	Preconditions pass, property fails.
4	`fifo_2.ptr_wr==0`	Preconditions pass, property fails.
5	`fifo_2.status_empty==0`	Preconditions fail, property passes. Selected to find buggy lines.
6	`fifo_2.status_full==0`	Preconditions fail, property passes. Selected to find buggy lines.

Listing 4.5: Original property written for good design (MESI-ISC).

```
assert property((
fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
|=>(fifo_2.rd_i==1)
|=>(fifo_2.data_i==41'h4c5 and fifo_2.wr_i==1
and fifo_2.rd_i==0)
|=>##[0:1](fifo_2.data_i==41'h4c9 and fifo_2.wr_i==1
and fifo_2.data_o==41'h4c1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_o==41'h4c5));
```

Listing 4.6: Appending signal ID 1 from CEX (MESI-ISC).

```
assert property((
fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
|=>(fifo_2.rd_i==1)
|=>(fifo_2.data_i==41'h4c5 and fifo_2.wr_i==1
and fifo_2.rd_i==0 and fifo_2.fifo_depth_decrease==1)
|=>##[0:1](fifo_2.data_i==41'h4c9 and fifo_2.wr_i==1
and fifo_2.data_o==41'h4c1)|=>(fifo_2.rd_i==1)
|=>(fifo_2.data_o==41'h4c5));
```

Listing 4.7: Appending signal ID 2 from CEX (MESI-ISC).

```
assert property((
fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_i==41'h4c5
and fifo_2.wr_i==1 and fifo_2.rd_i==0
and fifo_2.fifo_depth_increase==0)
|=>##[0:1](fifo_2.data_i==41'h4c9
and fifo_2.wr_i==1 and fifo_2.data_o==41'h4c1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_o==41'h4c5));
```

Listing 4.8: Appending signal ID 3 from CEX (MESI-ISC).

```
assert property((
fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_i==41'h4c5
and fifo_2.wr_i==1 and fifo_2.rd_i==0 and fifo_2.ptr_rd==0)
|=>##[0:1](fifo_2.data_i==41'h4c9 and fifo_2.wr_i==1
and fifo_2.data_o==41'h4c1)|=>(fifo_2.rd_i==1)
|=>(fifo_2.data_o==41'h4c5));
```

Listing 4.9: Appending signal ID 4 from CEX (MESI-ISC).

```
assert property((
fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_i==41'h4c5
and fifo_2.wr_i==1 and fifo_2.rd_i==0 and fifo_2.ptr_wr==0)
|=>##[0:1](fifo_2.data_i==41'h4c9 and fifo_2.wr_i==1
and fifo_2.data_o==41'h4c1)|=>(fifo_2.rd_i==1)
|=>(fifo_2.data_o==41'h4c5));
```

Listing 4.10: Appending signal ID 5 from CEX (MESI-ISC).

```
assert property((
fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_i==41'h4c5
and fifo_2.wr_i==1
and fifo_2.rd_i==0 and fifo_2.status_empty==0)
|=>##[0:1](fifo_2.data_i==41'h4c9 and fifo_2.wr_i==1
and fifo_2.data_o==41'h4c1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_o==41'h4c5));
```

Listing 4.11: Appending signal ID 6 from CEX (MESI-ISC).

```
assert property((
fifo_2.data_i==41'h4c1 and fifo_2.wr_i==1)
|=>(fifo_2.rd_i==1)|=>(fifo_2.data_i==41'h4c5
and fifo_2.wr_i==1 and fifo_2.rd_i==0
and fifo_2.status_full==0)
|=>##[0:1](fifo_2.data_i==41'h4c9 and fifo_2.wr_i==1
and fifo_2.data_o==41'h4c1)|=>(fifo_2.rd_i==1)
|=>(fifo_2.data_o==41'h4c5));
```

The RTL lines in the design which contain the signals in Table 4.10 for which preconditions fail are presented later in Listing 4.25 in the Section 4.5. These lines are investigated for the bug using the knowledge from the design specification. In this case study, the bug was not found in any of those lines. Hence we do a secondary analysis on the lines which contain the signals which influences the signals considered in Table 4.10 in Listing 4.25. Such an analysis on wr_i takes us to the buggy RTL line shown in the secondary analysis section of the Listing 4.25.

The values of the signals given in Table 4.10 also helps in the primary and secondary analysis as shown below.

- The use of signal ID 1 indicates that ~$(fifo\_depth\_decrease == 1)$ is a must for the property to fail. The following line from $fifo\_2$ says that either $wr\_i = 1$ or $rd\_i = 0$ should happen.

  215 $assign\ fifo\_depth\_decrease = !wr\_i\ \&\ rd\_i;$

- The use of signal ID 2 indicates that ~$(fifo\_depth\_increase == 0)$ is a must for the property to fail. The following line from $fifo\_2$ says that $wr\_i = 1$ and $rd\_i = 0$ should happen.

  211 $assign\ fifo\_depth\_increase = wr\_i\ \&\ !rd\_i;$

- From previous steps it can be concluded that wr_i has to be one and rd_i has to be zero for the bug to appear.
- Signal ID 6 indicates that ~$(status\_full == 0)$ is essential for the bug to appear.

4.4.2 USB 2.0

4.4.2.1 $Orion$ (guidance hint mining)

The bug: The bug is in $usbf\_pe$ (Protocol Engine inside the Protocol Layer) module of USB design.

Buggy RTL line:

$IDLE : if((!match\_r)\&\&(!ep\_disabled)\&\&(!pid\_SOF))$

Original RTL line:

$IDLE : if((match\_r)\&\&(!ep\_disabled)\&\&(!pid\_SOF))$

The bug relates to the signal $match\_r$ being negated by a design mistake. When the logic inside the *if* condition is true, then the state can transition to states other than *IDLE*.

The manifestation of the bug: The register files were getting updated with wrong values. The reason for this behavior is the following. When the signal $match\_r$ is zero, the FSM in the protocol engine is supposed to stay in the state $IDLE$. But because of the bug, it moves out of the $IDLE$ state and proceeds to $UPDATE2$ state in situations where other signals are in favor of the state transition. This bug won't be exposed if the other signals are not in favor of this transition. That is the reason why this bug is excited only occasionally. In the $UPDATE2$ state, a signal $uc\_sta\_set\_d$ is set to 1, which enables the writes to register files.

The assumption about the broader buggy region: The debug engineer considers the possibilities for register updates and shortlists the modules *Protocol Engine* and *Packet Disassembler* as the potential broader buggy region. After that, they designed a property that ensures signal flows which eventually enables one possible way of register file update. As in the previous case study, there can be other possibilities which the engineer would consider if this assumption proves out to be wrong; i.e., the negation of the property written with this assumption on the bug passes. The property is shown in Listing 4.12. Following the approach similar to that of the earlier case study on MESI-ISC, we have the modified property with level-2 guidance hint as shown in Listing 4.13.

Listing 4.12: Original property for USB 2.0.

```
assert property(not(
(~u1.u2.tx_dma_en_r)throughout(u1.u2.size==14'h8
##[1:$]u1.u2.sizd_c==14'h20
##[1:$](u1.u2.sizd_is_zero_d==14'h1
##1 u1.u2.idma_done==1))
##[1:$]u1.u3.state==10'b000000_0001
&&(~u1.u3.match||u1.u3.pid_SOF)
##0 u1.u3.uc_bsel_set==1'b0 ##1 u1.u3.uc_bsel_set==1'b1));
```

Listing 4.13: Original property for USB 2.0 with guidance hint assertion at level-2.

```
assert property(not(
(~u1.u2.tx_dma_en_r)throughout(u1.u2.size==14'h8
##[1:$]u1.u2.sizd_c==14'h20
##[1:$](u1.u2.sizd_is_zero_d==14'h1 ##1 u1.u2.idma_done==1))
##[1:$]u1.u3.state==10'b000000_0001
&&(~u1.u3.match||u1.u3.pid_SOF)
##[0:$](u1.u3.state==10'b010000_0000
##[1:$]u1.u3.state==10'b100000_0000)
##0 u1.u3.uc_bsel_set==1'b0 ##1 u1.u3.uc_bsel_set==1'b1));
```

Structural constraint generation: The signal selected for the generation of the COI signals in this example is $u1.u3.uc\_bsel\_set$. Similar to the previous example, the signals present in the COI are given as inputs to the guidance hint miner.

Some of the level-1 hints mined from the simulation trace are shown in Table 4.11.

Table 4.11: Level-1 assertions mined from the simulation trace (USB 2.0).

No:	Assertion
H_{11}	(~u1.u3.ep_disabled && u1.u3.ep_stall ##1 u1.u3.state==10'b000000_0010)
H_{12}	(~u1.u3.ep_disabled ##10 u1.u3.rx_ack_to_clr_d==0)
H_{13}	(u1.u3.token_valid && u1.u3.pid_ACK ##1 u1.u3.state==10'b010000_0000)
H_{14}	(u1.u3.rx_active ##1 ~u1.u3.tx_data_to_cnt)
H_{15}	(u1.u3.uc_stat_set_d ##1 u1.u3.uc_dpd_set)

The good and bad level-2 guidance hints are shown in Table 4.12.

Table 4.12: Level-2 assertions used in the filtering process (USB 2.0).

Type	Assertion
Good (H_{21})	(u1.u3.state==10'b010000_0000 ##[1:$] u1.u3.state==10'b100000_0000)
Bad (H'_{21})	(u1.u3.state==10'b000000_0100 ##[1:$] u1.u3.state==10'b000000_1000)

The parts of the main property, which is used in the Bayesian analysis, is the following.[16]

```
u1.u3.uc_bsel_set == 1'b0 ##1 u1.u3.uc_bsel_set == 1'b1
```

The results of the Bayesian filtering process is shown in Table 4.13.

Table 4.13: Likelihood values for two-step filtering of waypoints.

Guidance Hints (H)	ll at step-1	ll at step-2	Selected/rejected
H_{11}	0.4955	0	*Selected*
H_{12}	0	NA	Rejected in step-1
H_{13}	0	NA	Rejected in step-1
H_{14}	0.497	0.497	Rejected in step-2
H_{15}	0	NA	Rejected in step-1

The selected assertion (for level-1) when embedded into the property mentioned in Listing 4.13 is given in Listing 4.14.

Listing 4.14: Property for USB 2.0 with waypoint assertions at level-1 and level-2.

```
assert property(not(
(~u1.u2.tx_dma_en_r)throughout(u1.u2.size==14'h8
##[1:$]u1.u2.sizd_c==14'h20
##[1:$](u1.u2.sizd_is_zero_d==14'h1 ##1 u1.u2.idma_done==1))
##[1:$]u1.u3.state==10'b000000_0001
&&(~u1.u3.match||u1.u3.pid_SOF)
##[0:$]( u1.u3.ep_disabled && u1.u3.ep_stall
##1 u1.u3.state==10'b000000_0010)
##[0:$](u1.u3.state==10'b010000_0000
##[1:$]u1.u3.state==10'b100000_0000)
##0 u1.u3.uc_bsel_set==1'b0 ##1 u1.u3.uc_bsel_set==1'b1));
```

The model checking results for the waypoint selected during the Bayesian filtering along with some which are not selected is shown in Table 4.14.

4.4.2.2 *Aquila* (bug localization)

The CEX given by Model Checker, with the signals of interest, is the following.

- Cycle 1: $(match = 1)$.

[16]Unlike the example of MESI-ISC, we have a part of the main property available in an assertion format itself in the case of USB 2.0.

Table 4.14: Model checking results for USB 2.0 with different hints.

Guidance Hints (H)	Time for model checking (s)
Original property without any hints	121.4
H_{11}	123.3
H_{12}	Property passes
H_{13}	Property passes
H_{14}	140
H_{15}	220.4

- Cycle 2: $(csr[22] = 0)$, $(state = IDLE)$, $(match\_r = 1)$, $(ep\_disabled = 0)$, $(pid\_SOF = 0)$, $(send\_token\_d = 0)$.
- Cycle 3 $(send\_token = 0)$.

Listing 4.15: Original property written for good design (USB 2.0).

```
assert property(
(match==1)##1(pid_SOF==0)&&(pid_PING==1)
&&(mode_hs==1)&&(csr[22]==0)
|=>(send_token==1));
```

Listing 4.16: Appending signal ID 1 from CEX (USB 2.0).

```
assert property(
(match==1)##1(pid_SOF==0)&&(pid_PING==1)
&&(mode_hs==1)&&(csr[22]==0)
|->(ep_disabled==0)
|=>(send_token==1));
```

Listing 4.17: Appending signal ID 2 from CEX (USB 2.0).

```
assert property(
(match==1)##1(pid_SOF==0)&&(pid_PING==1)
&&(mode_hs==1)&&(csr[22]==0)
|->(ep_disabled==0)&&(State==IDLE)
|=>(send_token==1));
```

Listing 4.18: Appending signal ID 3 from CEX (USB 2.0).

```
assert property(
(match==1)##1(pid_SOF==0)&&(pid_PING==1)
&&(mode_hs==1)&&(csr[22]==0)
|->(ep_disabled==0)&&(State==IDLE)&&(match_r==1)
|=>(send_token==1));
```

Listing 4.19: Appending signal ID 4 from CEX (USB 2.0).

```
assert property(
(match==1)##1(pid_SOF==0)&&(pid_PING==1)
&&(mode_hs==1)&&(csr[22]==0)
|->(ep_disabled==0)&&(State==IDLE)&&(match_r==0)
|=>(send_token==1));
```

The precondition[17] fails for the property shown in Listing 4.19. Using the last appended signal $match\_r$ we perform the fine-grained bug localization to arrive at the suspicious lines shown in Listing 4.26 in the Section 4.5.

4.4.3 PCI

4.4.3.1 $Orion$ (guidance hint mining)

The bug: The bug for PCI design is in the module $pci\_perr\_en\_crit$ block inside the pci_parity_check (parity checker block).

Buggy RTL line:

$perr = (par\_err\_response\_in||perr\_generate\_in)\ \&\&\ (non\_critical\_par\_in \wedge pci\_par\_in);$

Original RTL line:

$wire\ perr = par\_err\_response\_in \&\& perr\_generate\_in\ \&\&(non\_critical\_par\_in \wedge pci\_par\_in);$

The manifestation of the bug: Due to this bug, the signal $pci\_perr\_oe\_o$ goes high, indicating that there is a parity error, when it's not supposed to do so.

The assumption about the broader buggy region: The erroneous signal $pci\_perr\_oe\_o$ is a primary output of the PCI design. It comes from the module pci_io_mux, which gets some of its inputs from the pci_parity_check block. Using this information the verification engineer selects the region containing the modules pci_parity_check and pci_io_mux as the broader buggy region. The property designed considering bug in this region is given in Listing 4.20. Also, the modified version of this property with the level-2 guidance hint incorporated in it is shown in Listing 4.21[18].

Listing 4.20: Original property for PCI.

```
assert property(not(
```

[17]The precondition referred here is part of the property before $send\_token == 1$ in Listing 4.19.

[18]The redundant $\#\#0$ in the level-2 guidance hint present in Listing 4.21 is due to the particular output format of the assertion miner.

```
parity_checker.pci_par_en_in##1~pci_perr_oe_o;));
```

Listing 4.21: Original property for PCI with waypoint assertions at level-2.

```
assert property(not(
parity_checker.pci_par_en_in
##0(parity_checker.pci_perr_en_out ##0 pci_io_mux.perr_en_in)
##1~pci_perr_oe_o;));
```

Structural constraint generation: The signal selected for the generation of the COI signals in this example is $pci\_perr\_oe\_o$. Similar to the previous example, the signals present in the COI are given as inputs to the waypoint miner.

Some of the level-1 waypoint assertions/guidance hints mined from the simulation trace are shown in Table 4.15.

Table 4.15: Level-1 assertions mined from the simulation trace (PCI).

No:	Assertion
H_{11}	(parity_checker.par_err_response_in && parity_checker.perr_generate ##1 parity_checker.pci_perr_en_out)
H_{12}	(~parity_checker.non_critical_par && ~parity_checker.pci_par_in ##1 ~parity_checker.pci_perr_en_out)
H_{13}	(parity_checker.non_critical_par && ~parity_checker.check_for_serr ##0 ~parity_checker.pci_serr_out)
H_{14}	(parity_checker.pci_frame_reg_in && parity_checker.frame_dec2 ##[1:$] parity_checker.check_for_serr_on_second)
H_{15}	(parity_checker.pci_frame_en_in && parity_checker.frame_dec2 ##[1:$] parity_checker.check_for_serr_on_second)

Good and bad level-2 guidance hints are shown in Table 4.16.

The parts of the main property which is used in the Bayesian analysis is the following:

```
pci_perr_oe_o == 0 ##0 pci_perr_oe_o == 0
```

Table 4.16: Level-2 assertions used in the previous filtering process (PCI).

Type	Assertion
Good (H_{21})	`(parity_checker.pci_perr_en_out ##0 pci_io_mux.perr_en_in)`
Bad (H'_{21})	`(parity_checker.pci_serr_en_in ##1 parity_checker.pci_serr_en_out)`

Table 4.17: Likelihood values for two-step filtering of waypoints.

Guidance Hints (H)	ll at step-1	ll at step-2	Selected/rejected
H_{11}	0.498	0	*Selected*
H_{12}	0	NA	Rejected in step-1
H_{13}	0	NA	Rejected in step-1
H_{14}	0.4965	0.4775	Rejected in step-2
H_{15}	0.4965	0.4775	Rejected in step-2

The results of the Bayesian filtering process is shown in Table 4.17. The selected assertion for level-1 when embedded into the property mentioned in Listing 4.21 is given in Listing 4.22.

Listing 4.22: Property for PCI with waypoint assertions at level-1 and level-2.

```
assert property(not(
parity_checker.pci_par_en_in
##[0:$](parity_checker.par_err_response_in
&& parity_checker.perr_generate
##1 parity_checker.pci_perr_en_out)
##[0:$](parity_checker.pci_perr_en_out
##0 pci_io_mux.perr_en_in)
##1 ~pci_perr_oe_o);
```

The model checking results for the guidance hints selected during the Bayesian filtering along with some which are not selected is shown in Table 4.18.

4.4.3.2 *Aquila* (bug localization)

The original property written assuming the design is good is given in Listing 4.23. After following a sequence of modifications to the property we get one version for which the preconditions fail and the property passes. That version of the property is given in Listing 4.24. Identifying the last appended signal and following similar steps as earlier case studies, we arrive at the buggy RTL line highlighted in Listing 4.27.

Listing 4.23: Original property written for good design (PCI).

Table 4.18: Model checking results for PCI with different guidance hints.

Guidance Hints (H)	Time for model checking (s)
Original property without any hints	0.7
H_{11}	0.8
H_{12}	1.6
H_{13}	Property passes
H_{14}	17.8
H_{15}	16.5

```
assert property(
(pci_par_en_in==1)##1(par_err_response_in==0)
|->(perr_mas_detect_out==0));
```

Listing 4.24: After appending all shortlisted signals from CEX (PCI).

```
assert property(
(pci_par_en_in==1)
|->((perr_en_crit_gen.non_critical_par_in==1)
&&(perr_en_crit_gen.par_generate_in==0)
&&(perr_en_crit_gen.pci_par_in==0)&&
(perr_en_crit_gen.par_err_response_in==1))
|->(perr_en_crit_gen.perr==0)##1(par_err_response_in==0)
|->(perr_mas_detect_out==0));
```

4.5 Summary of Experimental results

The output of the entire framework is a set of lines in the RTL code, which we can say contains the bug or can lead us to it. The results from our case studies are summarized in Table 4.19. The RTL lines which are shortlisted for each of the designs is available in Listings 4.25 to 4.27. It is to be noted that the number of RTL lines shortlisted depict those lines where the selected signals[19] are present. In other words, these lines need to be investigated for fine-grained bug localization. For example, if the particular signal which we have identified is getting a particular value assigned using some other signals, the bug could be in any of those signals also. One will have to investigate those signals as well to catch the bug. Our method saves time by assisting the verification engineer that they need to look only backward from a particular signal. Therefore, with the proposed technique, we can easily arrive at a set of lines in the design responsible for

[19]for which the preconditions fail in $Aquila$.

the buggy behavior. Table 4.19 summarizes the results for all three case studies where it can be observed that the number of RTL lines (shown in the third column) is within 1% of total signals in the design (shown in the fourth column).

Table 4.19: Summary of bug localization results.

Design	# Signals related to the bug	# of RTL lines shortlisted	# Total lines in the RTL
MESI-ISC	3	7	990
USB 2.0	1	5	4,062
PCI	1	3	16,693

Listing 4.25: The shortlisted RTL lines (MESI-ISC).

```
709 else if(fifo_depth==1&fifo_depth_decrease)
716 else if(fifo_depth==0&status_full&fifo_depth_decrease)
  status_full<=0;
718 assign fifo_depth_decrease=!wr_i&rd_i;
711 else if(fifo_depth==0&status_empty&fifo_depth_increase)
715 else if(fifo_depth==FIFO_SIZE-1&fifo_depth_increase)
  status_full<=1;
717 assign fifo_depth_increase=wr_i&!rd_i;
714 if(rst)status_full<=0;
------------------------------------------------------------
Secondary analysis from wr_i
120 else if (wr_i)
```

Listing 4.26: The shortlisted RTL lines (USB 2.0).

```
1864 always@(posedge clk)match_r<=match;
2005 always@(posedge clk)abort<=buffer_overflow|
(match&(state!=IDLE))|(match_r&to_large);
2024 always@(posedge clk)int_upid_set<=match_r&!pid_SOF&
((OUT_ep&!(pid_OUT_r|pid_PING_r))|(IN_ep&!pid_IN_r)
|(CTRL_ep&!(pid_IN_r|pid_OUT_r|pid_PING_r|pid_SETUP_r)));
2036 always@(state or ep_stall or buf0_na or buf1_na
or pid_seq_err or idma_done or token_valid or pid_ACK
or rx_data_done or tx_data_to or crc16_err or ep_disabled
or no_bufs or mode_hs or dma_en or rx_ack_to or pid_PING
or txfr_iso or to_small or to_large or CTRL_ep
or pid_IN or pid_OUT or IN_ep or OUT_ep or pid_SETUP
or pid_SOF or match_r or abort or buffer_done or no_buf0_dma
or max_pl_sz)begin
2050 IDLE:if(match_r&&!ep_disabled&&!pid_SOF)begin
```

Listing 4.27: The shortlisted RTL lines (PCI).

```
12503 wire perr=(par_err_response_in||perr_generate_in)
&&(non_critical_par_inp̂ci_par_in);
12510 perr_en_reg_out<=#1 perr;
12512 assign perr_en_out=perr||perr_en_reg_out;
```

The prior work most closely resembling to our approach (specifically the *Aquila* i.e., bug localization part of the work) is that of [24]. In this work, the authors have attempted localization of bugs in the SVA representations assuming that the design is correct. However, we assume that the properties are correct while the design is buggy.

4.6 Conclusion

This work proposed a methodology for design debugging with the assistance of the model checking technique. While formal methods like model checking are useful in exhaustive functional verification, they are hindered by scalability issues. This can be tackled to a significant extent through guidance-based strategies. However, providing guidance hints in an automatic manner is a non-trivial exercise. In one case study, the proposed technique of guidance filtering provides suitable hints to the model checker such that CPU time is reduced. During debugging by formal methods, efficient bug localization is often exacerbated by tedious counterexamples. The proposed method of counterexample analysis assists in bug localization with the support of guidance hints. The proposed work can be extended in multiple directions. First, enhancing the quality of guidance hints generation with the help of directed testbenches during the mining step. Second, the selection of signals from the counterexample trace (for $Aquila$) can also be significantly improved to obtain a lesser number of RTL lines as suspect candidates during the bug localization step. Third, automatically obtaining the broader buggy regions (for $Orion$) needs to be investigated to ensure the minimization of false positives during the debug step. Fourth, the guidance hint generation methodology needs to be further improved for obtaining speed up in model checking for all designs.

Chapter 5

Bug localization with Semi-formal Techniques

5.1 Introduction

Bug localization is one of the most challenging steps in achieving the time-to-market deadlines during integrated circuits development [82] [44] [83]. Automatic bug localization from simulation traces is generally very tedious and requires multiple iterations [46, 84]. Effective design bug localization offers significant benefits in quick design fixes [85, 86]. However, for localization to succeed, we require testbenches that can trigger interesting execution scenarios in the design that can probably hint towards the bug. We refer to them as *intelligent testbenches*. This approach's rationale is that the testbenches can excite some design paths and lead the execution to some internal states that are in an explicit/implicit relationship with the bug. We utilize a genetic algorithm (GA)-based framework to obtain such testbenches.

As stated in the previous chapters, despite the wide usage of simulation-based verification strategies, the completeness of the correctness can not be guaranteed [84, 87]. This problem of completeness can be significantly solved by formal verification methodologies such as model checking. However, automatic bug identification from the model checking process turns out to be a tedious exercise [16, 25, 47, 75]. As a result of the growing design complexity, analyzing generated counterexamples requires manual intervention for fine-grained bug localization. We address this problem in this work through an automated methodological approach. Typically, semi-formal methods have been found to be useful in bug hunting [7] and accelerating the verification tasks [10,48]. However, straightforward usage of semi-formal techniques for automatic

bug localization is not trivial. We employ a mixture of formal and semi-formal methodologies to address the bug localization in RTL design descriptions given a failing property.

5.2 Intelligent Testbench Design using Genetic Algorithm

Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection [88]. It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a relatively huge amount of time to solve. Note that it is a probabilistic approach, thereby giving one or more than one solution depending on the particular problem formulation. It includes the process of selection, crossover, and mutation.

5.2.1 Basic Terminology

In any genetic algorithm-based framework, the basic terminology involves gene, population, chromosome, individual, fitness, and selection. We briefly explain these terms here to elucidate their usage.

5.2.1.1 Gene, Chromosome & Population

Fig. 5.1 represents a pictorial view of a population matrix. There are four arrays named A1, A2, A3 and A4, each of them, is a chromosome. A chromosome is also called an individual. One signal element of an individual is called Gene. The complete matrix is referred to as population. Thus gene is the smallest set and population is the largest set.

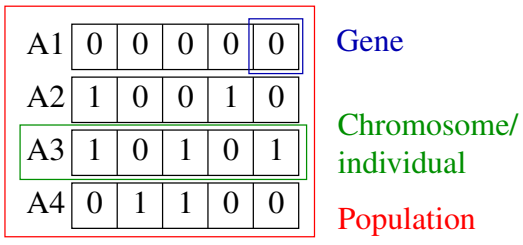


Figure 5.1: The concept of gene, chromosome and population in GA

5.2.1.2 Fitness Function

The notion of fitness denotes the quality of a chromosome. In this way, the fitness function evaluates the fitness of a chromosome in the context of the particular problem getting formulated. As it varies from one usage scenario to another, we explain in detail the chosen fitness function in the experimental section.

5.2.1.3 Selection

After calculating the fitness, the mating parents are selected, which will give a new crossover child. The selection of the mating parents are made on various strategies like rank-based selection, tournament selection, roulette wheel selection, etc.

Table 5.1: Rank-Based Selection

Chromosome	Fitness Value	Rank
A	10.06	3
B	15.01	1
C	13.02	2
D	9.67	4
E	1.97	5

Table 5.1 represents the fitness score of the chromosomes and their corresponding rank. Rank is decided based on the fitness value. The higher the fitness higher will be the rank. Thus if for mating, three parents need to be selected on the basis of rank, then parents B, C, and A will be selected for the crossover.

5.2.1.4 Crossover & Mutation

In the process of crossover, some genes of the mating parent get swapped based on the crossover point. The above Fig. 5.2 represents the one-point crossover strategy in which a crossover point is chosen at the midpoint and half of the genes get swapped. By this process, new offspring created, which works as the parent for the next generation. There are other types of crossovers called the two-point and uniform crossovers. The generalization of one point and the two-point crossover is referred to as uniform crossover.

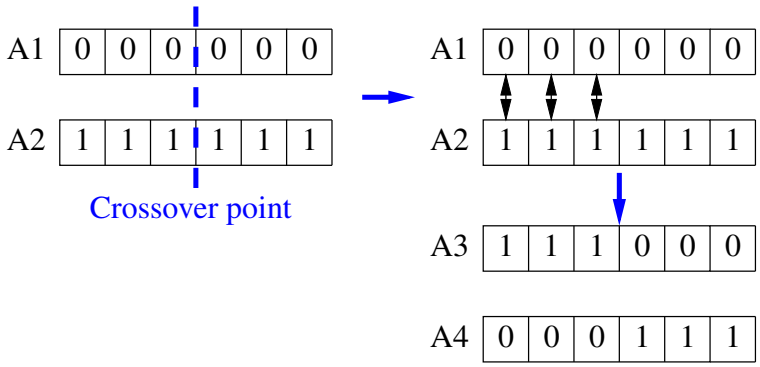


Figure 5.2: One Point Crossover

By the process of mutation, some bits in the chromosome get flipped. The process of mutation happens very rarely, and it is performed after crossover. It is done to maintain diversity in the offspring. In the below Fig. 5.3, three bits are flipped in a chromosome.

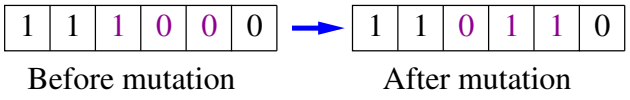


Figure 5.3: Mutation

.

5.2.2 Proposed GA-based Methodology

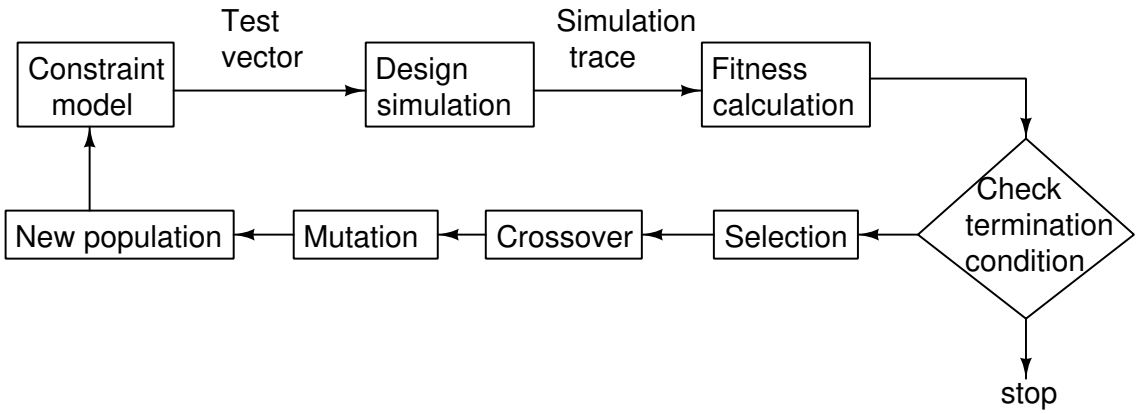


Figure 5.4: Diagram of GA-based Methodology

Fig. 5.4 represents a general GA framework. In the first run, the new population will be the initial population, which is randomly generated by the GA framework. Based on the decoding logic for the population, using the constraint model we will generate a file that contains the test

vectors for the Design-under-Verification (DUV). Then, the design is simulated, and simulation traces are obtained. Based on the Finite State Machine (FSM) coverage numbers, the fitness is calculated. After that, we check for the termination conditions.

We have provided two termination conditions, 1^{st} if the target reached and 2^{nd} if the upper limit of the generation, which is taken as 100. If any one of these two conditions is met, then we stop the simulation. If the condition is not met then, we continue with the process of crossover, mutation and get the new set of parents. The mutation is not done in each generation. We have specified the mutation frequency for each generation that determines if the mutation is to be performed or not.

5.3 Experiments and Results of GA-based Methodology

To run the experiments, we have used Modelsim simulator, beginning with a constrained-random testbench. The fitness values are calculated by parsing the branch and FSM coverage reports produced by Modelsim. The Genetic algorithm framework is a modified version of the code available in GitHub.

This section discusses a Proof-of-concept example of a FIFO and two other experiments on USB 2.0 [3] and PCI [4] designs from Opencores.

5.3.1 First In First Out (FIFO) of Various Sizes

FIFO of different sizes like 16, 64, and 256 are taken for the first experiment. We have set the objective to make FIFO full, as buffer overflow is one of the trigger points of bugs, and it is an important part of FIFO verification. The population size has been chosen as 8×6 i.e. there are eight chromosomes, and each chromosome has six genes.

5.3.1.1 Chromosome Encoding

We constructed a constrained-random testbench that executes read (RD), write (WR), and reset (RS) operations on FIFO. All the operations will be done consecutively, i.e., if there are six read operations, then all the read operations are performed consecutively. We refer to specific values of these operations/parameters as input directives. Let the sequence {RD, WR, RS, RD-6, WR-5, RS-7} be an input directive. In this directive, the initial three words of RD, WR, RS represent

the order in which the operation will perform, and later three words represent the frequency of operation. So this directive means there will be six reads, five writes, and seven reset operations.

Table 5.2: Chromosome Encoding in FIFO

14	20	11	5	6	7

Table 5.2 represents decimal encoded directive of FIFO, here initial three genes, i.e., 14, 20, 11 are the weights of WR, RD, and RS sequence respectively, thus sequence of operation will be {RD, WR, RS} and the last three genes, i.e., 5, 6, 7, are the frequency of operation of WR, RD and RS operations, respectively. Thus, this chromosome will be decoded as the generated test vector for {RD-6, WR-5, RS-7} operation sequence.

5.3.1.2 FIFO Design Bug Scenarios

One case of FIFO bug is if buffer_full signal comes one cycle later. Due to this bug, data will be overwritten. The fitness function for this case is defined as below: eq. 5.1

$$Fitness = number\ of\ (write - read)\ operations \tag{5.1}$$

Another case of buggy scenario in FIFO is if buffer_empty signal delayed by one cycle. Due to this issue, the wrong data will be read by the user. Fitness is evaluated by the below eq. 5.2

$$Fitness = number\ of\ (read - write)\ operations \tag{5.2}$$

5.3.1.3 General Chromosome Encoding

In the previous experiment, we have taken a tightly constrained testbench, which does not include all the operations. So for all other modules of the design, we decided to proceed with an unconstrained test setup. We formally define a term called sequence mentioned below in eq. 5.3

$$Sequence = all\ the\ signals\ in\ a\ design\ module\ except\ clock\ +\ no.\ of\ cycles \tag{5.3}$$

Table 5.3: General Structure of a Chromosome

Sequence 1	Sequence 2	...	Sequence N

Table 5.3 represents a pictorial representation of a general unconstrained chromosome. The parameter N will be chosen based on the design knowledge. The minimum no. of cycles is equal to the total no. of states. The binary encoding scheme is selected for all the next experiment as it gives more flexibility and the range of signals can be specified properly.

5.3.2 IDMA module of USB 2.0

5.3.2.1 The bug

In this experiment, we have selected a bug in the IDMA module of the USB 2.0 design. Due to the bug, the signal $idma\_done$ goes high. The objective of the GA experiment is to get a simulation trace that will contain the root cause of this bug. We follow two approaches to get the simulation trace. In the first approach, we used branch coverage/hits to selected branches for the fitness calculation. In the second approach, FSM state coverage is used for the same purpose. Even though in terms of implementation, FSM coverage is a sub-set of branch coverage, FSM coverage helps in faster convergence of the GA operation in some cases.

5.3.2.2 FSM coverage

The first step is to identify the state at which the $idma\_done$ signal is set to 1. We can see that MEM_RD3 is the state we are looking for. Once the target state is identified, we need to identify a possible set of states which can lead to the target state. Those states are referred to as *desired states*. The states which are left out in the FSM after this process are grouped in the category of *undesired states*.

In this example, the target state is MEM_RD3, and after the target state is identified, we select a path from the initial state to the target state. Table 5.4 represents the desired and undesired states for this example.

Table 5.4: Desired and Undesired States in the IDMA Module of USB 2.0

Desired states	**Undesired states**
IDLE, MEM_RD1, MEM_RD2, MEM_RD3	WAIT_MRD, MEM_WR1, MEM_WR2

5.3.2.3 Fitness function

Fitness is the most important parameter for the GA framework. It should be a progressive parameter, not a binary parameter. As in the case of the binary, we will not be able to decide whether we are moving in the right direction or not. The fitness function should be designed such that there should be an indication of whether we are moving in the right direction or not. In other words, the right direction would mean more hits to the *desired states*. The underlying thought here is that the more the hits to the states in the paths leading to the target state, the more is the chance for eventually reaching the target state. While this need not always be true for all designs, this approach gave us promising results.

A simple version of the fitness function is given in Equation 5.4. We have also performed experiments with fitness functions with varying weights given to hits to different states. In cases where the simple fitness function didn't help the GA converge, fitness functions with varying weights seem to help.

$$Fitness = total\ hits\ to\ desired\ states \tag{5.4}$$

The GA run converged for some runs and did not converge for others. For the runs which converged, it took a maximum of 100 generations. For the runs which did not converge, we experimented with different possible *desired states*. For example, the state *IDLE* was avoided from the set of *desired states* in some runs. We observed that the high fitness value was sometimes due to the hits to only the *IDLE* states, misleading the GA framework giving it a false feeling of progressing towards the target state. Manual interventions like these are required while running GA for the new designs.

5.3.2.4 Branch coverage

The second approach for fitness calculation is using hits to selected branches during RTL simulation. Following the same approach as that of the FSM coverage, we need to identify branches in the RTL which can potentially lead to the target condition. Here, the target condition is the signal $idma\_done$ becoming 1. The branches which were identified for this purpose are shown in Table 5.5. To demonstrate the use of branch coverage, we have taken the branches (case statements) corresponding to the states considered in Section above.

Table 5.5: Desired branches in the IDMA module of USB 2.0

Desired branches
line 332: MEM_RD1:begin
line 338: MEM_RD2:begin
line 342: MEM_RD3: begin

5.3.2.5 Fitness function

The fitness function used in this experiment is shown in Equation below. The *weight* is an integer number with one assigned to the lowest line number. The number is incremented by one for the next higher line number and so on.

$$Fitness = \sum_{i=1}^{num_desired_states} weight_i * hits_{state_i} \tag{5.5}$$

The GA run converged within 60 generations providing us a simulation trace where the target branches are hit.

5.3.3 PE (Protocol Engine) module of USB 2.0

In the second experiment with USB 2.0 we have chosen the protocol engine (PE) module inside the protocol layer (PL).

5.3.3.1 The bug

Here, the signal $match\_r$ is inverted by a design mistake. The buggy and correct RTL lines are shown below.

Buggy RTL line :

$$IDLE : if((\textbf{!match\_r})\&\&(!ep\_disabled)\&\&(!pid\_SOF))$$

Original RTL line:

$$IDLE : if((\textbf{match\_r})\&\&(!ep\_disabled)\&\&(!pid\_SOF))$$

When the match_r signal is zero, the FSM should remain in the IDLE state, but due to this bug, it can traverse from the IDLE to the UPDATE2 state. In UPDATE2 state, uc_stat_set_d

signal becomes 1, which enables the register write operation. This leads to an invalid register write, causing data corruption in the register file.

5.3.3.2 FSM coverage

It is probable that the debug engineer can think that in UPDATE2 state resister write signal is enabled. Thus UPDATE2 state is selected as the target state. Table 5.6 shows good states and bad states. Considering various paths which lead to the target state from the IDLE state, a set of *desired* and *undesired states* is identified. It is shown in Table 5.6. This table shows two cases for two different experiments we have run.

Table 5.6: *desired* and *undesired states* of USB_PE module

Case	Desired states	Undesired States
1	IDLE, IN, IN2, UPDATE, UPDATE2	OUT, OUT2A, UPDATEW, TOKEN
2	IDLE, OUT, OUT2A, UPDATE, UPDATE2, UPDATEW	IN, IN2, TOKEN

The fitness function used in this experiment is shown in Equation 5.6. We have chosen a *weight* of 5 in this experiment.

$$Fitness = (fitness\ value\ 2\ ) + (weight) * (fitness\ value\ 2\ ) \tag{5.6}$$

The hits to the target states were achieved within 100 generations.

5.3.3.3 Branch coverage

The branches identified as *desired branches* for this experiment are shown in Table 5.7.

Table 5.7: Desired branches in the PE module of USB 2.0

Desired branches
line 732: 10'b000001_0000:
line 742: 10'b000010_0000: // This is a delay State
line 761: 10'b010000_0000:begin // Interrupts
line 768: 10'b100000_0000: begin // Update Register File

The fitness function used is a weighted sum of the hits to the *desired branches*, as discussed in one of the previous sections. We were able to get hits to the target states in more than half of the experiments.

5.3.4 Wishbone module of PCI

The Wishbone master module (wb_master) of the Peripheral Component Interconnect (PCI) [4] design module is selected for the experiment.

5.3.4.1 The bug

The bug is related to states assigned inside if and else conditions are swapped by designer mistake.

Buggy RTL line :

$$if(last\_data\_from\_pciw\_fifo\_reg)$$
$$n\_state = \textbf{S\_TURN\_ARROUND};$$
$$else$$
$$n\_state = \textbf{S\_WRITE\_ERR\_RTY};$$

Original RTL line:

$$if(last\_data\_from\_pciw\_fifo\_reg)$$
$$n\_state = \textbf{S\_WRITE\_ERR\_RTY};$$
$$else$$
$$n\_state = \textbf{S\_TURN\_ARROUND};$$

Due to this bug, FSM can traverse to S_TURN_ARROUND state while the last transaction is not completed while it was supposed to go to S_WRITE_ERR_RTY state and cleanup the previous transaction. Because of this issue, FIFO might not be cleared from the last event before starting the next transaction. This would lead to data corruption root cause of which would be difficult to detect at a later stage.

5.3.4.2 FSM coverage

Once the erroneous data is observed, the debug engineer can come to the conclusion that the data corruption might have happened in the current cycles or any cycle before that. It becomes difficult to identify the good states only based on that assumption. Hence they need to identify

different states which will be covered to finish one full write operation. They can also refer to the simulation trace with the erroneous behavior present in it, to reach a conclusion on *desired states*. Out of many possible *desired states*, we will discuss the set of states which would reproduce the bug in the simulation trace. It is given in Table 5.8.

Table 5.8: *Desired* and *undesired states* in the pci_wb_master module of PCI

Desired states	Undesired States
S_WRITE, S_TURN_AROUND	$S\_IDLE, S\_WRITE\_ERR\_RTY, S\_READ, S\_READ\_RTY$

The fitness function used in this case is the same as the one given in Equation 5.5. After adjusting the weights to the hits given to each of the desired states, we were able to convergence in the GA experiment within 30 generations. As the end results, we obtained simulation traces traversing through different paths, eventually reaching S_TURN_AROUND. Some of them contain the state transitions related to the bug.

5.3.4.3 Branch coverage

Following the approaches mentioned above, we arrive at the set of *desired branches* shown in Table 5.9.

Table 5.9: Desired branches in the pci_wb_master module of PCI

Desired branches
line 821: S_WRITE: // WRITE from PCIW_FIFO to WB bus
line 864: 3'b010 : // If writing of one data is terminated with ERROR
line 879: else // if there wasn't last data of transfer
line 1107: default: // S_TURN_ARROUND:

The fitness function used in this experiment is similar to the one given in Equation 5.5. In this experiment, we were able to produce a simulation trace, which can expose the buggy scenario within 50 generations. To give a sense of timing, on average, it took 70 minutes to run GA for around 100 generations. Again, this will depend on the complexity of the design.

5.3.5 Discussion

In the light of running different experiments on various designs, we can conclude that Genetic An algorithm-based framework can be used for generating targeted simulation traces which can

reproduce a buggy scenario. However, we observed during experiments that if the bug is too deep inside the state space, it will require many runs and fine-tuning of the parameters of the GA. Especially, the way we design the fitness function can play an important role. Additionally, tuning the mutation parameter can help in getting faster convergence in cases where the scenario we want to reproduce in the RTL simulation is not straightforward.

5.4 Assertion-based bug localization

Given the complexity of designing perfectly tuned testbenches for bug localization (as is clear from the previous sections), we propose an alternative automated fine-grained bug localization method. This method does not have strict requirements of a high-quality testbench. The specific objective of this method is to get a set of potential buggy RTL lines which contain the root cause of the failure of a property. The input of this RTL bug localization flow is a failing property and the buggy design. Once we have the failing property, the first step is to generate more failing properties. To achieve this, we modify the original failing property and run model checking on them using the Jaspergold model checker [89]. The exact methodology of modifying the original property is discussed later. The traces of the modified properties along with the trace generated by the original property are used by the assertion miner (similar to the one presented in Chapter 4) to generate assertions. Mined assertions are checked for their correctness using a model checker. Those assertions that pass this step are analyzed in the subsequent steps. The assertions are then filtered to avoid redundant assertions. The filtered assertions are then ranked based on their potential to point towards the buggy RTL lines. In the last step, the RTL lines pointed to by the high-ranked assertions are identified. These lines are the ones that potentially contain the root cause of the failure of the original property. The overview of the proposed assertion-based bug localization approach is given in Figure 5.5.

5.4.1 Steps of Proposed Methodology

Following are the major steps in the proposed assertion-based bug localization methodology:

1. Multiple failure trace generation
2. Assertion mining from the failure trace
3. Assertion filtering

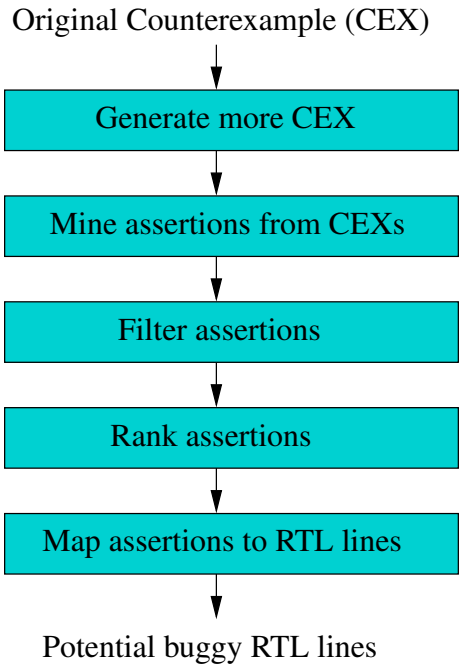


Figure 5.5: Assertion-based bug localization

4. Assertion ranking

5. Assertion to RTL line mapping

5.4.2 Multiple Counterexample(CEX) Generation

As said before, we have a failing property from where the proposed methodology begins. Our approach requires multiple traces with the same failure observed in each trace. We generate a collection of such counterexample traces by iteratively modifying the property with the signal-value combinations selected from the original counterexample. Since there is a large number of signals in real designs, we need to identify a set of *target signals* that we need to look for in the original counterexample for modifying the property. We will discuss that in the next section.

5.4.2.1 Target Signals

The outputs of all the modules to which the signals in the original failing property belong are identified first. To cut down the number of signals further, we consider only the control signals among them. The next step is to find the control signals in the Cone of Influence(COI) of each of the signals identified above. A union of all these signals is considered as *target signals*.

5.4.2.2 Property modification and counterexample generation

We take the original property and modify it multiple times by negating the signal-value combination of the control signals present in the COI of target signals. The idea here is to find different ways of exposing the bug. If the modified property is failing, we get a counterexample to excite the bug. Hence, finally, we get multiple simulation traces from the failing original property as well as the modified properties that failed.

When we modify the property with the signal-value combinations selected from the counterexample trace, it is possible that the condition mentioned in the modified property is not feasible in the design. Those properties will lead to cases where pre-condition failure happens in the model checker. Such properties will be categorized by the tool as *vacuously proven* properties. We ignore such properties in our approach. On the contrary, when a property fails, there is a chance that it leads to a different path to the bug. Many such counterexample traces ensure that we cover a large number of scenarios where the bug is exposed. Mining assertions from such traces will give us the relations between signals which are potentially pointing us towards the bug. The procedure of assertion mining is discussed next.

5.4.3 Assertion Mining

The miner used in this section is a modified version of the one discussed in Section 4.2.1 of Chapter 4. The key changes are given below.

5.4.3.1 Support to mine from counterexamples

The previous version of the miner was used to generate assertions from the RTL simulation trace provided by the simulators like Modelsim (in the list format). This version of the miner accepts the counterexample traces provided by the model checkers like Jaspergold (in the VCD format). In short, this is a more generic version which that accepts the VCD files from simulators, model checkers, etc.

5.4.3.2 Only single-bit signals

The modified version of assertion miner generates assertions with only single-bit signals. In the previous version, the output was an assertion with $>$ and $<$ in it. Since we were considering

multi-bit signals, the possibility of getting false assertions mined from the simulation trace was very high. The modified version does not use $>$ or $<$ operators, instead use only $==$ symbol.

5.4.3.3 Assertions with only selected signals

The miner involves building a decision tree using the signals present in the simulation trace. When the designs become large, the size of the tree will increase, making the process of mining time-consuming. Hence, we introduced a feature to mine the relations between only selected signals. These signals can be stored in a file where the miner can access them. We generate a list of control signals in the Cone of Influence(COI) of the target signals. These signals, in addition to target signals stored in the above-mentioned file. The list of signals also has temporal signals. Hence it also tells us if the value of a signal some clock cycles before the current cycle affects the value of the target signal or not. For example, if $[3]sig\_a$ is present in the COI of a target signal, it means that the value of the signal $[2]sig\_a$ before three cycles affects the value of the target signal in current cycle.

5.4.3.4 Improved approach for temporal assertion generation

In the previous version of the miner, we used to feed a file with delay associated with each signal with respect to the consequent signal. This was generated through structural approaches. This approach led to more invalid assertions. Hence we eliminated the need for such an input file and used the counterexample trace itself to generate the temporal information, as discussed above.

The signal-value table will have entries with and without the temporal information present in it. The decision tree will be constructed with all those signals, and assertions will be generated with the temporal value of signals still present at the start of the signal. This temporal information present at the beginning of each signal is later converted to standard delays denoted by $\#\#temporal\_value$ accepted by the SystemVerilog Assertions (SVA) format. The algorithm used to convert begins with first finding the highest delay present in all the signals present in the antecedent of the assertion. Then we divide the antecedent signals into different groups corresponding to the same delays. Finally, we start writing the signals group-wise, starting with the group with the highest delay and going in decreasing order of the delays while using the difference between consecutive groups to join two adjoining groups with sign ##. Let's consider an example where the CART decision tree generates the following assertion.

[3]a && [2]b && [1]c |-> d

where a, b, c and d are the names of the signals in the design. It will be converted to a standard SVA format, as shown below.

a ##1 b ##1 c |-> ##1 d

As discussed in Section 4.2.1, we pass the mined assertions through a model checker and remove those which fail the model checking process.

5.4.4 Assertion Filtering

This step involves removing the redundant assertions. It is possible that multiple assertions lead to the same path while trying to expose the bug. We need to take only one of these assertions, and the other is declared redundant/common assertions.

Following are the rules to determine whether two assertions are common (similar to the methodology presented in [46]).

- The consequent of both assertions have the same signal-value pair.
- The temporal delay between successive conjunctions in the antecedent of both assertions are identical.
- Each of the conjunctions in both assertions should consist of the same set of propositions.

Here, conjunction is a set of signal-value pairs in the antecedent which have the same temporal delay. If all three rules are followed, those two assertions are declared as common. Finally, we get a list of unique assertions which we will be using for the next process. During the filtering process, we also note the number of times an assertion was common with the other. The assertion which has come common more times is more likely to point towards the bug.

5.4.5 Assertion Ranking

Ranking of the assertions is done to ensure that we give high priority to those assertions which have more potential to localize the bug. The overview of the assertion ranking approach is given in Figure 5.6.

The four parts of the assertion ranking step are discussed next.

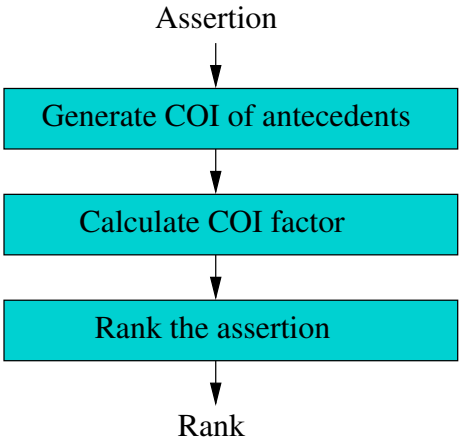


Figure 5.6: Assertion Ranking flow

5.4.5.1 Cone of Influence(COI) calculation of assertions

We calculate the combined cone of influence (COI) of all the signals present in the antecedent of all assertions. As done in the mining, we consider only the control signals in the COI for the final list of signals corresponding to an assertion. This is primarily because control signals cover a relatively large portion of the design.

5.4.5.2 Intersection with original property COI

The next step is to calculate the control signals in COI of signals present in the antecedent of the original failing property. For each assertion, we calculate its intersection with the original property COI that gives us common signals in COI of both. Afterward, we normalize these number of common signals for each assertions using the following formula to get the COI factor.

$$COI\ factor = \frac{Number\ of\ common\ signals}{Maximum\ number\ of\ common\ signals\ for\ any\ assertion}$$

The final ranking is done on the basis of the *COI factor*. The assertions which have higher *COI factor* are ranked higher.

5.4.5.3 Guarantee parameter

We calculate a guarantee parameter which means that we give a number on the basis of our analysis which says that these number of top assertions out of the total ranked assertions are enough for consideration. This is needed because the number of assertions can still be high after filtering. Thus, we do not want the less important ones to hamper our final localization results.

For calculating this parameter, we take the sum of the helpfulness factor of all assertions. Now we start taking cumulative sum of the helpfulness factors from the top and when the cumulative sum gets greater than a specific percent (e.g., 50%, variable) of total sum, we stop and say that only the assertions from the top till the one where we stopped should be used for next process.

5.4.6 RTL Mapping

This is the final step of the proposed flow which maps the assertions to RTL lines. The overview of the approach is given in Figure 5.7.

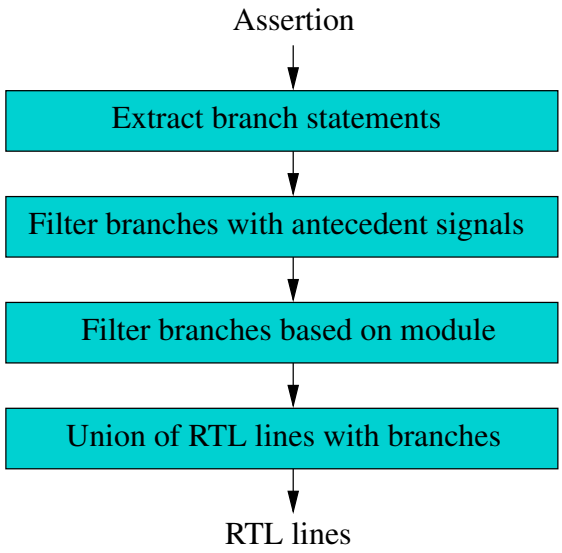


Figure 5.7: RTL Mapping flow

5.4.6.1 Extraction of branch RTL lines data

As discussed earlier, we utilize static analysis to get CFG for our design. The CFG also contains the lines where the control signals are present. By extracting these line numbers, we can know all the branches present in our design since the control signals are present in the conditional statement of branches. Thus we extract RTL line numbers of branches of types if-else, case, and always statements.

We make two sections according to the type of branches. One section has information about the case and if-else statements, while the other has information about always statements. We remove the information of *always* statements, which only have a clock in their arguments as it misleads us in terms of bug localization most of the time.

5.4.6.2 Mapping RTL lines according to antecedent signals

For each assertion, we have determined the antecedent signals, and we also have information about branch line number, branch type, and signals present in the conditional statement of that branch. Now, for every assertion, we determine all the branches which have at least one antecedent signal in their conditional statement. Again, we keep the tally of branches of each line separated into two sections based on branch types, as stated before.

5.4.6.3 Module wise RTL Line filtering

After the last step, we get a high number of mapped RTL lines for each assertion. We determine the union of modules to which all the antecedent and consequent signals belong to each assertion. For each assertion, out of all the mapped RTL lines, we consider only those branches which belong to the union of the modules determined for that assertion. This gives us the RTL lines mapped by each of the assertions.

5.4.6.4 RTL line ranking

In this step, we take the union of all the mapped RTL lines obtained after module-wise mapping for all assertions. With this step, the assertion to RTL line mapping is completed. These mapped RTL lines can be passed over to the verification engineer so that he/she can prioritize these RTL regions for bug localization.

5.5 Experimental Results of Assertion-based Bug Localization

An overview of the bug localization results is given in Table 5.10. Additional results in terms of the percentage of the localized code can also be generated.

Table 5.10: Bug localization results

Design	Bug number	# Mined assertions	# Filtered assertions	# Ranked assertions	Rank of the buggy *always* block	Rank of the buggy *case/if* block
USB 2.0	1	2269	646	100	1	3
USB 2.0	2	101	60	58	1	2
PCI	1	1357	592	202	3	∞

The rank is shown as ∞ in the table if the desired *always* or *case/if* block is not found in the ranked assertion set.

5.5.1 USB 2.0

5.5.1.1 Bug 1

The first bug in the USB 2.0 design is given below.

Line 3117: IDLE: if(match_r && !ep_disabled && !pid_SOF) begin // correct
Line 3117: IDLE: if(!match_r && !ep_disabled && !pid_SOF) begin //buggy

The property which failed due to this bug is as shown below.

$$(!u1.u3.match\_r)\&\&(u1.u3.state == 1)\#\#1(!u1.u3.match\_r)$$

$$[*1 : \$]|- > (u1.u3.uc\_dpd\_set == 0)$$

In this experiment, the rank of line 3100 (always block containing bug) as per *always ranking* is 1. Also, the rank of line 3117 (case block containing bug) as per *case+if ranking* is 3. To localize the bug, one should consider high-ranked *if/case* statements inside the high ranked *always* statements first. If the bug is not found, look for lower-ranked *if/case* blocks inside the high-ranked assertions and so on. Once all the *if/case* statements inside an *always* block, move to the lower-ranked *always* blocks and repeat the process.

5.5.1.2 Bug 2

The first bug in the USB 2.0 design is given below.

Line 1772: else if(pid_TOKEN && rx_valid && rx_active && !rx_err) begin // correct
Line 1772: else if(pid_TOKEN && rx_valid && rx_active) begin // buggy

The property which failed due to this bug is as shown below.

$$(u1.u0.state == 2)\&\&(u1.u0.rx\_err)|- > \#\#1u1.u0.state! = 4;$$

In this experiment, the rank of line 1753 (always block containing bug) as per *always ranking* is 1. But the rank of line 1772 (if block containing bug) as per *case+if ranking* is same as remaining 19. So, we need to give equal priority to all the lines inside the *always* block while looking for the root cause of the bug.

5.5.2 PCI

5.5.2.1 Bug 1

The first bug in the PCI design is given below.

Line 6839: $n\_state = S\_IDLE$; // Correct

Line 6839: $n\_state = S\_READ\_RTY$; // Buggy

The property which failed due to this bug is as shown below.

$$(pci\_target\_unit.wishbone\_master.first\_wb\_data\_access == 1)\&\&$$

$$(pci\_target\_unit.wishbone\_master.rty\_counter\_almost\_max\_value == 0)\&\&$$

$$(pci\_target\_unit.wishbone\_master.c\_state == `WB\_FSM\_BITS'h3)|-> \#\#1$$

$$(pci\_target\_unit.wishbone\_master.c\_state != `WB\_FSM\_BITS'h4);$$

In this experiment, the rank of line 1753 (always block containing bug) as per *always ranking* is 3. The case statement(line 6832) with the buggy RTL (line 6839) is present in the mapped RTL Lines. The rank of *case/if* block is shown as ∞ for this experiment since its was not in one of the top ranked assertions.

5.5.3 Discussion

We present some of the subtle differences between iterative model checking-based bug localization (presented in Chapter-4) and assertion-based bug localization:

- In the former approach, we begin with a failing simulation error trace, while in the latter approach, we start the localization procedure with a failing counterexample.
- Guidance hints are needed in the former approach (the hints assist in speeding the model checking process), whereas these hints are not utilized in the latter approach. Given the complexity of obtaining effective guidance hints, we believe that the latter approach would find wider applicability. Although mining is used in both approaches, the former technique requires mining from simulation traces to obtain assertions that are used as guidance hints, whereas in the latter approach, assertion (used directly for bug localization) are mined from a set of failing counterexample traces.

- In the former approach, we need to construct a set of properties (targeting specific buggy scenarios) that aid in the process of localization, while in the latter approach, the set of properties is not required. Because of this reason, the former approach can not be fully automated (as construction of property set is a manual step and error-prone), while the proposed assertion-based bug localization can be fully automated without any manual intervention.

5.6 Conclusion

Effective bug localization during verification is a challenging step in the development cycle of complex hardware designs. While meeting different coverage goals/metrics is possible in the verification process, yet bug localization can not be easily related to such goals. We proposed a two-step methodology to achieve fine-grained design bug localization. In the first step, we generate assertions from counterexample trace(s) through a mining exercise. As the second step, the mapping of the assertions leads to specific lines (in RTL descriptions), which are the root cause of the design bug. Experimental case studies substantiate the efficacy of the proposed methodology. We also proposed a genetic algorithm-based testbench generation technique that strives to overcome the inefficacy of random or constrained-random testbenches.

Chapter 6

Instruction-Based Test

6.1 Introduction

Owing to the constraints for test time and cost, the full testing of chip is not a feasible option. Hence, we require methods which will enable the chip to test for itself once manufactured. These methods can also enable us to do the testing even after the chip is shipped (in-field testing). There are two ways by which you can do the self testing, also termed as Built In Self Test (BIST): hardware and software-based BIST (SBST). The traditional hardware-based SBST leads to area overhead and additional power consumption. Hence, software-based self testing becomes more attractive. Once test vectors are generated they are mapped to test instructions using the information available in RTL and ISA. This ensures that all possible structural faults are tested using the instructions. The main difficulty of this method is the generation of constraints required for the mapping process. One needs to have good knowledge of the RTL to generate those constraints. The difficulty in generating the test instructions from the test vectors also depends on the component under test. However, for the modules like *Forwarding unit* the mapping process might take a good amount of time. Also the constraint extraction for all the paths from the inputs of the forwarding unit to the primary input of the processor will cover a good percentage of the design. With different coding styles followed by designers, it becomes even more difficult to automate the process, which in turn would require manual intervention at some stages.

We have identified drawbacks of both the template-based instruction sequence generation method as well as the method of mapping of test vectors into test instructions. Very often in the first case, the size of the test instruction sequence explodes and in the second case, the

constrained test vector generation as well as mapping of test vectors into instructions are the challenges. Broadly we can say that none of these methods are ideal for testing the full processor. Some methods work really well for certain parts of the processor and performs poorly for some other parts. Hence, the real challenge is to map different sections of the processor with test generation methods ideal for it. To understand the real difficulties, we had to select a block for which test instruction generation will not be straight forward. We have chosen the forwarding unit of the MIPS pipelined processor [90] for that reason. The first step is to identify the blocks in the forwarding unit and understand which test method which best suits them. It became clear that for testing the control logic of the forwarding unit, we can easily generate test instruction templates deliberately introducing dependencies. However, it is not guaranteed that all the faults in the forward control logic will be detected using the above method, because the faults should also propagate through the forward muxes which has 32-bit data inputs. These faults along with the remaining faults in the left over parts of the forwarding unit (which contains 6 muxes) need to be tested with appropriate method. Generating all possible combinations of the data input is not required and also is not practical.

We chose to design the wrapper instead of specifying the constraints in the ATPG tool because of two reasons. First one is the limitation of the Tetramax ATPG tool in accepting certain constraints. Secondly, we wanted to keep the possibility of automating the constraint-based test generation process. These test vectors are then converted into test instructions. Out of multiple combination of test instructions which will serve the purpose of applying the test vectors at the input of the forwarding unit, we have identified two major templates which will cover all possible functional test vectors for the forwarding unit The test application is verified by running Modelsim simulation using the testbench generated. In brief, the main contributions of this chapter are the following.

- Validates the concept of mapping different design blocks with the optimum SBST testing method [91] with forwarding unit testing
- Concept of generic wrapper design using constraints
- Proposes two instruction templates which can apply all possible functional test vectors in the forwarding unit

It is also important to note that the generic methods might not give good fault converage for all sections of the processor. Research by Gizopoulos et al. [92] demonstrating poor coverage

for the pipelined logic is relevant in that sense. Results from their work show that customization in standard SBST programs which targets functional blocks is needed to get good fault coverage for special blocks like the pipelined logic. Since we are targeting maximum fault coverage for the forwarding unit in this chapter, outputs from Bernardi et al. [63]'s research is also relevant. They were successful in generating an instruction sequence template for testing the comparator and muxes inside the forwaring unit. But it is to be noted that manual effort was involved in generating the test data inputs needed for testing the MUXes.

6.2 Proposed methodology of Functional Test Generation

6.2.1 Forwarding unit operation

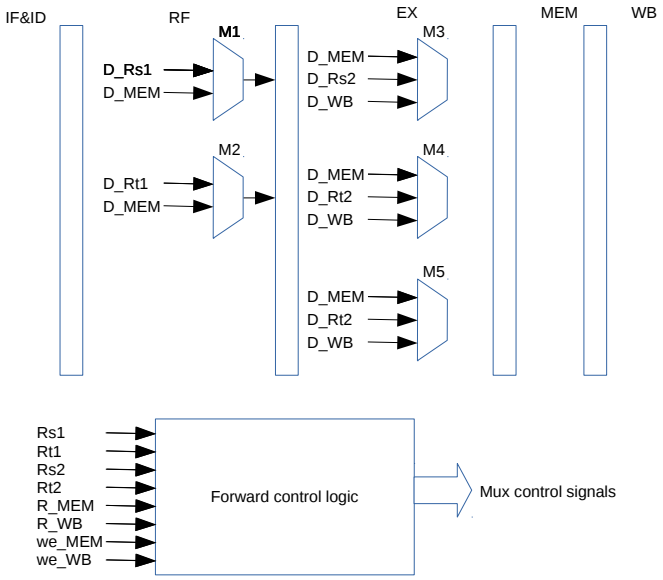


Figure 6.1: Forwarding unit of MIPS 5-Stage pipelined processor

We are using the Forwarding unit of 5-stage MIPS pipelined processor available in Opencores [90] with minor modifications. It has six muxes (2 in the Register Fetch stage and 3 in the Execution stage) and a forward control logic. D_Rs1, D_Rs2, D_Rt1, D_Rt2 are the 32-bit data which will be passed through the registers if there is no data dependency. R_MEM and R_WB are the destination registers of the instructions at the MEM and WB stages respectively. Also D_MEM and D_WB are the data forwarded from the Memory and Write-Back stages respectively. The 1-bit control signals we_MEM and we_WB are the write enable signals corresponding to the instructions at the MEM and WB stages.

Table 6.1: Forwarding cases

R_MEM = D_Rs1 && we_WB = 1
R_MEM = D_Rt1 && we_WB = 1
R_WB = D_Rs2 && we_WB = 1
R_WB = D_Rt2 && we_WB = 1
R_MEM = D_Rs2 && we_MEM = 1
R_MEM = D_Rt2 && we_MEM = 1
R_MEM = R_WB = D_Rs2 && we_WB = we_MEM = 1
R_MEM = R_WB = D_Rt2 && we_WB = we_MEM = 1

The objective of this work is to generate a test instruction sequence which can detect all possible functionally excitable stuck-at faults in the forwarding unit of the MIPS 5-stage pipelined processor. We followed the steps mentioned below.

6.2.2 Work flow

The first step in this process is to identify the test instruction generation methods which best suits each block inside the forwarding unit. Once that is done, test instruction sequences are generated using those methods. These are then combined them to make a single testbench for the processor. The synthesized gate-level netlist of the processor is then simulated in Modelsim to generate the output VCD file which is used to do the fault simulation of the processor in Tetramax ATPG tool. As shown in the Figure 6.2, we first do the full process with the testbench which has only the instructions for testing the forward comparator logic as given in Section 6.2.3. In further iterations, it picks the test vectors generated using the method mentioned in 6.2.4 and use the instructions generated from them using the method mentioned in Section 6.2.5. The relevant blocks for this step is highlighted in the Figure 6.3. After each iteration of fault simulation, the faults which are detected by the simulation are stored in a table for the further analysis.

6.2.3 Instruction template to test forward comparator logic

A detailed analysis will tell you that we can generate a test instruction template which will emulate all the data dependencies given in Table 6.1. We use a combination of four instructions

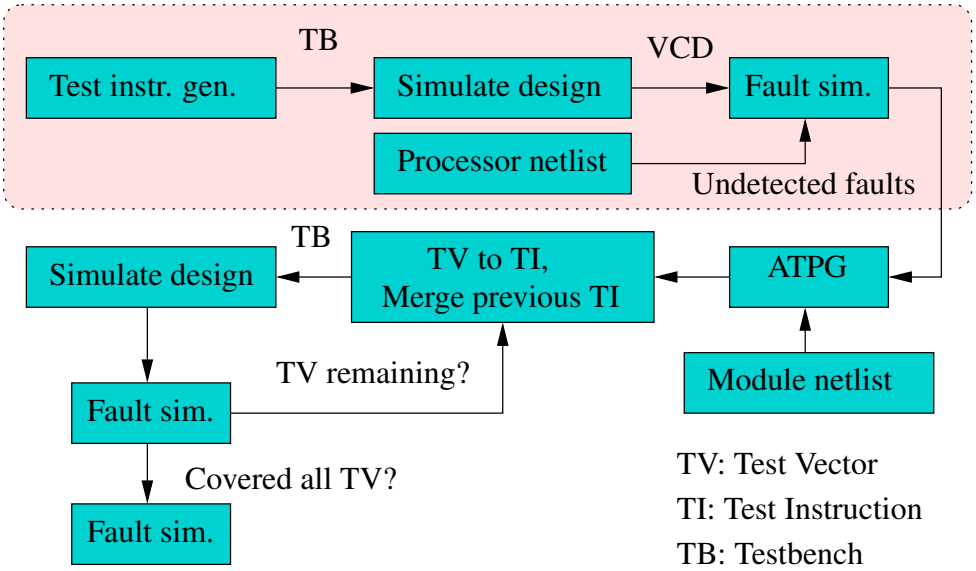


Figure 6.2: Constrained random instruction sequence generation

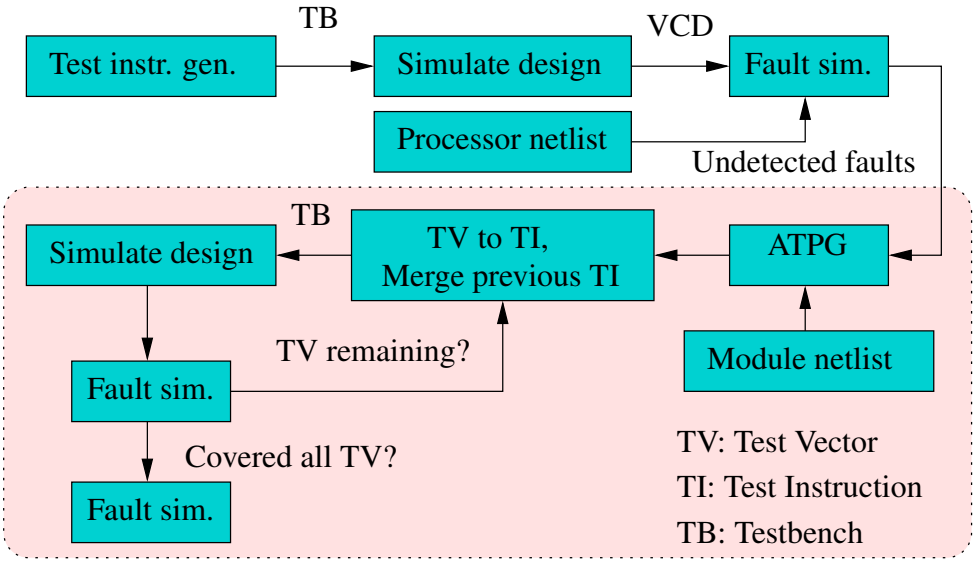


Figure 6.3: Test instruction generation from the test vectors

for each dependency as given in Table 6.2.

Table 6.2: Instruction template to test the forward comparator logic

ADD R_WB, Rx1, Rx2
ADD R_MEM, Rx3, Rx4
ADD Rx5, Rs2, Rt2
ADD Rx6, Rs1, Rt1

6.2.4 Constraint-based wrapper design and test vector generation

Fault simulation is done using a VCD file. This file is generated from the simulation of the processor netlist using the testbench which contains the a sequence of instructions as given in the Section 6.2.3. The faults which are not detected using this simulation are used for generating test patterns. We give them as the input fault list to the ATPG along with the extracted netlist of the forwarding unit. To make sure that the test vectors are functionally applicable, we have designed a wrapper module in Verilog which is connected at the input of the forwarding unit and generated test for the combined module. The functional test vectors for the forwarding unit are extrated from the gate level simulation results (in the form of a list file from Modelsim) of the block given in Figure 6.4. The test vectors are then extracted from the simulation results which contains the inputs to the forwarding unit. The constraints which we have used for designing the wrapper are given in Table 6.3.

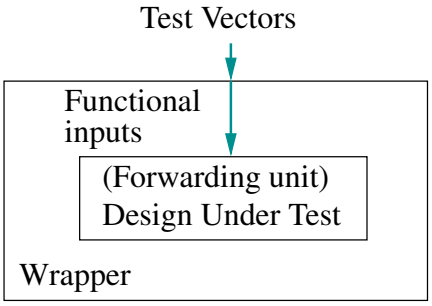


Figure 6.4: Wrapper and the forwarding unit

Table 6.3: Constraints used for wrapper design

Case	Constraint
A	If (we_MEM = 0) then R_MEM = 5'h0
B	If (we_WB =0) then R_WB = 5'h0
C	If (Rt1 = 5'h0) then D_Rt1 = 32'h0 else if (Rt1 = Rs1) then D_Rt1 = D_Rs1
D	If (Rs1 = 5'h0) then D_Rs1 = 32'h0 else if (Rs1 = Rs2) then D_Rs1 = D_Rs2 1
E	If (Rs2 = 5'h0) then D_Rs2 = 32'h0 else if (Rs2 = Rt2) then D_Rs2 = D_Rt2
F	If (Rt2 = 5'h0) then D_Rt2 = 32'h0 else if (Rt2 = Rt1) then D_Rt2 = D_Rt1

6.2.5 Test vector to test instruction conversion

We have identified two cases of test instruction templates which can be used for applying all possible functional test patterns at the input of the forwarding unit. Table 6.4 and Table 6.5 contains the sample test patterns and the instruction templates which will apply that test pattern at the input of the forwarding unit.

Store (SW) instructions are appended at the end of each set of instructions to ensure observability of the results.

Table 6.4: Sample test pattern and the instruction template corresponding to that (Template 1)

we_MEM	1
we_WB	1
Rs1	00001 (R1)
Rt1	00010 (R2)
Rs2	00011 (R3)
Rt2	00100 (R4)
R_MEM	00101 (R5)
R_WB	00110 (R6)
D_Rs1	000A00AA
D_Rt1	000B00BB
D_Rs2	000C00CC
D_Rt2	000D00DD
D_MEM	000E00EE
D_WB	000F00FF

Loading registers

I1: LUI R9, 000C

I2: XORI R3, R9, 00CC

I3: LUI R10, 000D

I4: XORI R4, R10, 00DD

I5: LUI R11, 000A

I6: XORI R1, R11, 00AA

I7: LUI R12, 000B

I8: XORI R2, R12, 00BB

I9: LUI R13, 000E

I10: LUI R14, 000F

Buffer instructions

I11: SW R15, 0(R16)

I12: SW R17, 0(R18)

Instructions for test application

I13: XORI R5, R13, 00EE

I14: XORI R6, R14, 00FF

I15: ADD R7, R3, R4

I16: ADD R8, R1, R2

Table 6.5: Sample test pattern and the instruction template corresponding to that (Template 2)

we_MEM	0
we_WB	0
Rs1	00001 (R1)
Rt1	00010 (R2)
Rs2	00011 (R3)
Rt2	00100 (R4)
R_MEM	00000 (R0)
R_WB	00000 (R0)
D_Rs1	000A00AA
D_Rt1	000B00BB
D_Rs2	000C00CC
D_Rt2	000D00DD
D_MEM	000E00EE
D_WB	000F00FF

Loading registers

I1: LUI R9, 000C

I2: XORI R3, R9, 00CC

I3: LUI R10, 000D

I4: XORI R4, R10, 00DD

I5: LUI R11, 000A

I6: XORI R1, R11, 00AA

I7: LUI R12, 000B

I8: XORI R2, R12, 00BB

I9: LUI R13, 000E

I10: XORI R5, R13, 00EE

I11: LUI R14, 000F

I12: XORI R6, R14, 00FF

Buffer instructions

I13: SW R15, 0(R16)

I14: SW R17, 0(R18)

Instructions for test application

I15: SW R5, 0(R0)

I16: SW R6, 0(R0)

I17: ADD R7, R3, R4

I18: ADD R8, R1, R2

6.3 Experimental Result

The processor is simulated using the testbench with the instructions generated according to the template mentioned in Section 6.2.3. The fault simulation of the full processor with only the faults in the forwarding unit added to the fault list resulted in a fault coverage of 77.35% [Total faults: 2060, Detected (DT): 1556, Possibly Detected (PT): 75, Undetectable (UD): 36, Not Detected (ND): 393]. We used the faults from PT, UD and ND (Total: 504) as the input fault list to the ATPG to generate test for the forwarding unit with wrapper (Section 6.2.4). Fault simulation using the instructions generated by the methods discussed in 6.2.5 detected 449 faults out of the 504 faults. There are 55 faults which are *Not Detected* (*Not Controlled: 4, Not Observed: 51*). Out of these 55 Not-Detected faults, 36 were anyway Undetected as per the first simulation. In short, there are 19 faults which are structurally applicable but functionally not applicable. Using the full test instruction set which combined the instructions from both methods gives the result as given in Table 6.6.

Table 6.6: Fault coverage

Detected (DT)	2005
Not Detected (ND)	55
Fault Coverage (FC)	97.33%

6.4 Conclusion

Software-based self test is an attractive alternative to structural test. In this chapter, we attempted to solve the automatic functional test generation problem. We have identified the need for selecting optimum test generation methods for different blocks inside the processor and validated it using the forwarding unit as an example. Subsequently, we need to generalize the classification process of various other components (like hazard detection unit) for validation of the proposed methodology. Additionally, the constraint extraction portion of the proposed methodology needs to be fully automated to improve the quality of functional tests generated.

Chapter 7

Conclusion and Future Work

Functional verification and test is an important problem in the overall design development and execution cycle. Traditional approaches suffer from a range of problems. In this thesis, few of these problems have been addressed. While formal methods like model checking are useful in exhaustive functional verification, they are hindered by scalability issues. This can be tackled to a significant extent through guidance-based strategies. For functional verification, we addressed the model checking problem and bug localization issue. We have also addressed the functional test pattern generation through mapping from structural faults-based test patterns to instructions of a MIPS processor.

7.1 Conclusion

For formal verification of designs, guided state space traversal [9, 10] methodologies have often been utilized to varying degree of success. In this thesis, we revisited this concept with special focus on automatically discovering effective guidance strategies. However, the complexity of model checking problem does not get significant reduction unless suitable guidance is provided. To address this issue, we have proposed a Bayesian model-based methodology for guidance identification which also utilizes design static analysis and simulation traces based on high level functional failure. With the help of a Bayesian modeling-based reasoning technique, we succeed in significantly reducing the number of probable candidate states which can act as guideposts. With the help of these guideposts, we achieve reduction in CPU time during the model checking exercise. Additionally, in few cases, we are able to solve the time out problem thereby taming the complexity of the design (and the particular property).

During debugging by formal methods, efficient bug localization is often exacerbated by tedious counterexamples. To address this, we proposed a methodology for design debugging with the assistance of the model checking technique. We also proposed a two-step methodology for bug localization using assertions and the help of static analysis of the design description. First, we obtain multiple error traces from the failing counterexample. Starting from the initial counterexample error trace, we employ iterative model checking to generate different error traces that are utilized to mine important assertions. In the second step, we utilize these assertions for fine-grained design bug localization. In experimental case studies, we observed that the proposed technique is able to obtain fine-grained bug localization at RTL.

Due to manufacturing at lower technology nodes, various kinds of defects can escape to the manufactured silicon. Although structural tests are widely successful in solving manufacturing tets, complex functionalities (for instance, forwarding) of processor-based systems can not be sufficiently tested. To solve this, functional tests can be applied. However, there are challenges to ensure the fault detection with these type of tests. We proposed instruction templates for mapping the obtained functional tests to the forwarding unit of the MIPS processor. Through usage of this technique, we observed that quality of functional tests can be significantly enhanced.

7.2 Future Work

The proposed work can be extended in multiple directions.

- As mentioned before, providing guidance hints in an automatic manner is a non-trivial exercise. In one case study, we observed that the proposed technique of guidance filtering provides suitable hints to the model checker such that CPU time is reduced. Enhancing the quality of guidance hints generation with the help of directed testbenches during the mining step can be an important direction of this work. First, the Bayesian model can be improved through better consideration of simulation traces. Second, the completeness of sub-properties can be enhanced by writing more temporal properties. The enhanced guidance hint generation methodology can assist us in obtaining speed up in model checking for all designs.

- The selection of signals from the counterexample trace (for $Aquila$) can also be significantly improved to obtain a lesser number of RTL lines as suspect candidates during

the bug localization step. Also, automatically obtaining the broader buggy regions (for $Orion$) needs to be investigated to ensure the minimization of false positives during the debug step. Hierarchical analysis of the design can be helpful in this step.

- The generated guidance hints can be analyzed for probable usage in the design abstraction. With the help of such abstraction, design approximation can also be carried out. Since, the guidance hints can assist us in identifying the important design regions, we believe that the proposed methodologies can be helpful in this direction also. Moreover, guidance hints can help in the assessment of sensitive areas for probable information leakage during design execution.

- Scalability issues of the proposed assertion-based bug localization technique can be investigated through experiments on large open-source designs like OPENSPARC processor modules. Typically, such complex designs have very large number of modules which would require many interactions of model checking the partitioned instances of the main property. Therefore, a second level refinement of internal signals corresponding to these partitions is necessary for reducing the probable buggy region candidates.

- The proposed functional test generation method needs to be tested for other hidden areas in the design like hazard detection blocks. Moreover, the constraint extraction technique needs to be automated. Improving the design of the generic wrapper can also be another direction of extension of this work.

- Usage of deep learning approaches in different aspects of the addressed challenges is also an important direction. For example, selection of guidance posts based on a training-based model can be a viable alternative to the identification (and ranking) methodologies outlined in this thesis. Similarly, improving the quality of testbenches based on deep learning techniques (like neural networks-based training) can be attempted. This has enormous opportunities given the decades of product development/academic research done so far in this area. This can enable us to a database of design bugs from published errata documents and other publications.

Bibliography

[1] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 282–293, Nov 2008.

[2] Y. Amitay, "Mesi coherency intersection controller." `http://www.opencores.org/projects/mesi\_isc`, 2013.

[3] "Usb 2.0 function core." `https://opencores.org/projects/usb`, Jun-2019.

[4] "PCI." `https://opencores.org/projects/pci`. Accessed: 2020-06-06.

[5] `http://www.opencores.org/projects/mesi\_isc`.

[6] B. Bentley, "Validating the intel pentium 4 microprocessor," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, (New York, NY, USA), p. 244–248, Association for Computing Machinery, 2001.

[7] P. K. Nalla, R. K. Gajavelly, J. Baumgartner, H. Mony, R. Kanzelman, and A. Ivrii, "The art of semi-formal bug hunting," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, Nov 2016.

[8] S. Agbaria, D. Carmi, O. Cohen, D. Korchemny, M. Lifshits, and A. Nadel, "Sat-based semiformal verification of hardware," in *Formal Methods in Computer Aided Design*, pp. 25–32, Oct 2010.

[9] C. H. Yang and D. L. Dill, "Validation with guided search of the state space," in *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, pp. 599–604, June 1998.

[10] J. Yuan, J. Shen, J. A. Abraham, and A. Aziz, "On combining formal and informal verification," in *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pp. 376–387, 1997.

[11] J. Bergeron, *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media, 2012.

[12] J. Bhadra, M. S. Abadir, L. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *IEEE Design Test of Computers*, vol. 24, pp. 112–122, March 2007.

[13] A. Sen, J. Bhadra, V. K. Garg, and J. A. Abraham, "Formal verification of a system-on-chip using computation slicing," in *2004 International Conference on Test*, pp. 810–819, Oct 2004.

[14] M. K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing simulation with bdds and atpg," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pp. 385–390, June 1999.

[15] J.-C. Ou, D. G. Saab, Q. Qiang, and J. A. Abraham, "Reducing verification overhead with rtl slicing," in *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, GLSVLSI '07, (New York, NY, USA), p. 399–404, Association for Computing Machinery, 2007.

[16] V. V S, B. Kumar, and J. Adhaduk, "Identification of effective guidance hints for better design debugging by formal methods," in *VLSI Design and Test* (B. K. Kaushik, ed.), (Singapore), Springer Singapore, 2019.

[17] F. M. D. Paula and A. J. Hu, "An effective guidance strategy for abstraction-guided simulation," in *2007 44th ACM/IEEE Design Automation Conference*, pp. 63–68, June 2007.

[18] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 156–161 Vol.1, Feb 2004.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification* (E. A. Emerson and A. P. Sistla, eds.), Springer Berlin Heidelberg, 2000.

[20] D. Kroening, A. Groce, and E. Clarke, "Counterexample guided abstraction refinement via program execution," in *Formal Methods and Software Engineering* (J. Davies, W. Schulte, and M. Barnett, eds.), Springer Berlin Heidelberg, 2004.

[21] C. R. Ho, M. Theobald, B. Batson, J. P. Grossman, S. C. Wang, J. Gagliardo, M. M. Deneroff, R. O. Dror, and D. E. Shaw, "Post-silicon debug using formal verification waypoints," in *DVCon*, 2009.

[22] M. Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal, "SIVA: A system for coverage-directed state space search," *Journal of Electronic Testing*, vol. 17, pp. 11–27, Feb 2001.

[23] M. Chen and P. Mishra, "Decision ordering based property decomposition for functional test generation," in *Design, Automation and Test in Europe, DATE, Grenoble, France, March 14-18,*, pp. 167–172, 2011.

[24] B. Keng, S. Safarpour, and A. Veneris, "Automated debugging of SystemVerilog assertions," in *2011 Design, Automation Test in Europe*, pp. 1–6, March 2011.

[25] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, pp. 1138–1149, June 2008.

[26] S. M. Nowick and D. L. Dill, "Practicality of state-machine verification of speed-independent circuits," in *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pp. 266–269, Nov 1989.

[27] Z. Hanna, F. Deaton, K. Kranen, B. Hjort, and L. Lundgren, "Guided exploration of circuit design states," *US Patent US9372949B1*, pp. 1–16, Jun. 2016.

[28] C. Zamfir and G. Candea, "Automatic generation of program execution that reaches a given failure point," *US Patent US8966453B1*, pp. 1–26, Feb. 2015.

[29] M. Chopra, X. Du, R. Hardin, A. Jain, R. Kurshan, P. Mahajan, R. Prakash, and K. Ravi, "Method and system for partitioning an integrated circuit design," *US Patent US00704751OB1*, pp. 1–21, May 2006.

[30] X. Du, R. Kurshan, and K. Ravi, "Coverage metric and coverage computation for verification based on design partitions," *US Patent US007712059B1*, pp. 1–15, May 2010.

[31] J. Cutler, "System and method for repeating program flow for debugging and testing," *US Patent US20090199161A1*, pp. 1–7, Aug. 2009.

[32] G. Altekar, "System and method for reproducing device program execution," *US Patent US20110078666A1*, pp. 1–57.

[33] W. Lam and M. Souti, "Method and apparatus for detection and isolation during large scale circuit verification," *US Patent US7051303B1*, pp. 1–9, May 2006.

[34] K. Chang, I. Wagner, I. Markov, and V. Bertaco, "Automatic error diagnosis and correction for rtl designs," *US Patent US20080295043*, pp. 1–22, Nov. 2008.

[35] "System and method to analyze vlsi designs," *US Patent US20040049371A1*, pp. 1–12, Mar. 2004.

[36] A. Rahman, "Method and apparatus for integrated circuit fault isolation and failure analysis using linked tools cockpit," *US Patent US20070118827A1*, pp. 1–8, May 2007.

[37] S. Safarpour and A. Veneris, "Method, system and computer program for hardware design debugging," *US Patent US20090125766*, pp. 1–15, May 2009.

[38] "Static analysis-based method and system for detecting rtl (resistor transistor logic) design errors," *CN Patent CN102054100B*, Dec 2010.

[39] J. Martensson, "Debugging of counterexamples in formal verification," *US Patent US8103999B1*, pp. 1–20, Jun. 2012.

[40] V. Singhal, J. Higgins, and A. Singh, "Trace based method for design navigation," *US Patent US007137078B2*, pp. 1–22, Nov. 2006.

[41] S. Vasudevan, D. Sheridan, L. Liu, and H. Kim, "Integration of data mining and static analysis for hardware design verification," *US Patent US20130019216*, pp. 1–62, Jan. 2013.

[42] S. Vasudevan and S. Hertz, "Merit-based characterization of assertions in hardware design verification," *US Patent US20150082263*, pp. 1–49, Mar. 2015.

[43] B. Chen, M. Yamazaki, and M. Fujita, "Bug identification of a real chip design by symbolic model checking," in *Proceedings of European Design and Test Conference EDAC-ETC-EUROASIC*, pp. 132–136, Feb 1994.

[44] B. Kumar, K. Basu, M. Fujita, and V. Singh, "Post-silicon gate-level error localization with effective and combined trace signal selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 248–261, Jan 2020.

[45] "Synopsys verdi." `https://www.synopsys.com/verification/debug/verdi.html`. Accessed: 2020-06-06.

[46] D. Pal and S. Vasudevan, "Symptomatic bug localization for functional debug of hardware designs," in *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID 2016, Kolkata, India, January 4-8, 2016*, pp. 517–522, IEEE Computer Society, 2016.

[47] G. Barbon, V. Leroy, and G. Salaun, "Debugging of behavioural models using counterexample analysis," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[48] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD-2000. IEEE/ACM Digest of Technical Papers (Cat. No. 00CH37140)*, pp. 120–126, IEEE, 2000.

[49] S. Shyam and V. Bertacco, "Distance-guided hybrid verification with guido," in *Proceedings of the Design Automation Test in Europe Conference*, vol. 1, pp. 1–6, March 2006.

[50] K. Chang, V. Bertacco, and I. L. Markov, "Simulation-based bug trace minimization with bmc-based refinement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 1, pp. 152–165, 2007.

[51] Y.-A. Chen and F.-S. Chen, "Algorithms for compacting error traces," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pp. 99–103, 2003.

[52] S. Shen, Y. Qin, and S. Li, "A fast counterexample minimization approach with refutation analysis and incremental sat," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pp. 451–454, 2005.

[53] Sung-Jui Pan, Kwang-Ting Cheng, J. Moondanos, and Z. Hanna, "Generation of shorter sequences for high resolution error diagnosis using sequential sat," in *Asia and South Pacific Conference on Design Automation, 2006.*, pp. 5 pp.–, 2006.

[54] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Transactions on Computers*, vol. C-29, pp. 429–441, June 1980.

[55] P. Parvathala, K. Maneparambil, and W. Lindsay, "Frits - a microprocessor functional bist method," in *Proceedings. International Test Conference*, pp. 590–598, 2002.

[56] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based self-testing of delay faults in pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 1203–1215, Nov 2006.

[57] A. U. R. Shaheen, F. A. Hussin, N. H. Hamid, and N. B. Z. Ali, "Automatic generation of test instructions for structural faults in processor cores using satisfiability," in *2013 International SoC Design Conference (ISOCC)*, pp. 388–391, Nov 2013.

[58] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 369–380, Mar 2001.

[59] C. H. Chen, C. K. Wei, T. H. Lu, and H. W. Gao, "Software-based self-testing with multiple-level abstractions for soft processor cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, pp. 505–517, May 2007.

[60] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pp. 548–553, June 2003.

[61] S. Guramurthy, S. Vasudevan, and J. A. Abraham, "Automated mapping of pre-computed module-level test sequences to processor instructions," in *IEEE International Conference on Test, 2005.*, pp. 10 pp.–303, Nov 2005.

[62] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos, "Hybrid-sbst methodology for efficient testing of processor cores," *IEEE Design Test of Computers*, vol. 25, pp. 64–75, Jan 2008.

[63] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. S. Reorda, M. Grosso, and O. Ballan, "On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors," in *2013 14th International Workshop on Microprocessor Test and Verification*, pp. 52–57, Dec 2013.

[64] S. R. Makar and E. J. McCluskey, "On the testing of multiplexers," in *International Test Conference 1988*, pp. 669–679, Sep 1988.

[65] R. S. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial atpg," in *Proceedings International Test Conference 1997*, pp. 743–752, Nov 1997.

[66] C. Wolf, "Yosys open synthesis suite." `http://www.clifford.at/yosys/`.

[67] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, pp. 348–354, 1984.

[68] F. Pong and M. Dubois, "The verification of cache coherence protocols," in *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pp. 11–20, 1993.

[69] R. Komuravelli, S. V. Adve, and C. Chou, "Revisiting the complexity of hardware cache coherence and some implications," *TACO*, vol. 11, no. 4, pp. 37:1–37:22, 2014.

[70] P. Yalagandula, V. Singhal, and A. Aziz, "Automatic lighthouse generation for directed state space search," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pp. 237–242, March 2000.

[71] R. Fraer, G. Kamhi, B. Ziv, M. Y. Vardi, and L. Fix, "Prioritized traversal: Efficient reachability analysis for verification and falsification," in *Computer Aided Verification* (E. A. Emerson and A. P. Sistla, eds.), (Berlin, Heidelberg), pp. 389–402, Springer Berlin Heidelberg, 2000.

[72] Pei-Hsin Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and Jiang Long, "Smart simulation using collaborative formal and simulation engines," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD - 2000. IEEE/ACM Digest of Technical Papers (Cat. No.00CH37140)*, pp. 120–126, Nov 2000.

[73] H. Choi, B.-W. Yun, and Y.-T. Lee, "Simulation strategy after model checking: experience in industrial soc design," in *Proceedings IEEE International High-Level Design Validation and Test Workshop (Cat. No.PR00786)*, pp. 77–79, Nov 2000.

[74] K. Nanshi and F. Somenzi, "Guiding simulation with increasingly refined abstract traces," in *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, (New York, NY, USA), pp. 737–742, ACM, 2006.

[75] V. S. Vineesh, B. Kumar, R. Shinde, A. Jaiswal, H. Bhargava, and V. Singh, "Orion: A technique to prune state space search directions for guidance-based formal verification," in *2019 IEEE 28th Asian Test Symposium (ATS)*, pp. 123–1235, Dec 2019.

[76] L. Liu, D. Sheridan, W. Tuohy, and S. Vasudevan, "A technique for test coverage closure using goldmine," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 5, pp. 790–803, 2012.

[77] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 67–72, March 2015.

[78] S. Mitra, A. Banerjee, P. Dasgupta, and H. Kumar, "Counterexample ranking using mined invariants," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 1978–1991, Dec 2013.

[79] H. Chipman, E. I. George, and R. E. McCulloch, *The Practical Implementation of Bayesian Model Selection*, vol. Volume 38 of *Lecture Notes–Monograph Series*, pp. 65–116. Beachwood, OH: Institute of Mathematical Statistics, 2001.

[80] R. O. Gallardo, A. J. Huy, A. Ivanov, and M. S. Mirian, "Reducing post-silicon coverage monitoring overhead with emulation and bayesian feature selection," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pp. 816–823, Nov 2015.

[81] K. Murphy *et al.*, "The bayes net toolbox for matlab," *Computing science and statistics*, vol. 33, no. 2, pp. 1024–1034, 2001.

[82] V. V. S, B. Kumar, R. Shinde, N. Sharma, M. Fujita, and V. Singh, "Enhanced design debugging with assistance from guidance-based model checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

[83] H. Foster, "2018 FPGA functional verification trends," in *19th International Workshop on Microprocessor and SOC Test and Verification, MTV 2018, Austin, TX, USA, December 9-10, 2018*, pp. 40–45, IEEE, 2018.

[84] M. Abadir, J. Bhadra, W. Chen, and L. Wang, "Guest editors' introduction: Emerging challenges and solutions in soc verification," *IEEE Design Test*, vol. 34, pp. 5–6, Oct 2017.

[85] M. Dehbashi, A. Sülflow, and G. Fey, "Automated design debugging in a testbench-based verification environment," *Microprocess. Microsystems*, vol. 37, no. 2, pp. 206–217, 2013.

[86] K. Chang, I. Wagner, V. Bertacco, and I. L. Markov, "Automatic error diagnosis and correction for RTL designs," in *IEEE International High Level Design Validation and Test Workshop, HLDVT 2007, Irvine, CA, USA, November 7-9, 2007*, pp. 65–72, IEEE Computer Society, 2007.

[87] S. Vasudevan, "Still a fight to get it right: Verification in the era of machine learning," in *IEEE International Conference on Rebooting Computing, ICRC 2017, Washington, DC, USA, November 8-9, 2017*, pp. 1–8, IEEE, 2017.

[88] `https://www.tutorialspoint.com/genetic\_algorithms/genetic\_algorithms\_introduction.htm`.

[89] "JasperGold Formal Verification Platform." `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html`. Accessed: 2020-06-06.

[90] "MIPS Opencores processor." `https://opencores.org/project,mips789`. [Online; accessed 12-Aug-2017].

[91] D. Gizopoulos, A. Paschalis, and Y. Zorian, "An effective built-in self-test scheme for parallel multipliers," *IEEE Transactions on Computers*, vol. 48, pp. 936–950, Sep 1999.

[92] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, pp. 1441–1453, Nov 2008.