New Advances in Verification and Debugging of Hardware Systems

Debjyoti Bhattacharjee

July 2015

New Advances in Verification and Debugging of Hardware Systems

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Technology
in
Computer Science

by

Debjyoti Bhattacharjee
[Roll No: MTC-1322]

under the guidance of

Ansuman Banerjee
Associate Professor
Advanced Computing and Microelectronics Unit

Indian Statistical Institute
Kolkata-700108, India
July 2015

To my parents and my supervisor

CERTIFICATE

This is to certify that the dissertation titled **"New Advances in Verification and Debugging of Hardware Systems"** submitted by **Debjyoti Bhattacharjee** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Ansuman Banerjee
Associate Professor,
Advanced Computing and Microelectronics Unit,
Indian Statistical Institute,
Kolkata-700108, INDIA.

Acknowledgments

I would like to show my highest gratitude to my advisor, *Ansuman Banerjee*, Advanced Computing Microelectronics Unit, Indian Statistical Institute, Kolkata, for his guidance and continuous encouragement. He has taught me how to be a humble researcher and a vigorous learner, motivated me with great insights and always supported my pursuit of innovative ideas.

My deepest thanks to all the teachers of Indian Statistical Institute, for their valuable suggestions and discussions which added an important dimension to my research work. I would specially like to thank *Dr. Mandar Mitra*, Computer Vision and Pattern Recognition Unit, Indian Statistical Institute, Kolkata, for his teaching and support throughout the duration of my course that has inspired me.

I would also like to thank *Dr. Anupam Chattopadhyay*, Nanyang Technological University, Singapore, for his constructive comments and active help for crossing the hurdles in my research. I would also like to thank *Soumi Chattopadhyay* for her active efforts in helping me throughout my course.

Finally, I am very much thankful to my parents for their everlasting support.

Last but not the least, I would like to thank all of my friends for their help and support. I thank all those, whom I have missed out from the above list.

Debjyoti Bhattacharjee
Indian Statistical Institute
Kolkata - 700108 , India.

Abstract

Increasing design complexity, skyrocketing fabrication costs for modern digital systems coupled with an unacceptably large number of silicon respins led to growing importance of comprehensive and automated design verification. This thesis is an attempt to enhance the state of the art in a verification and debugging of hardware systems.

Assertions play a vital role in specifying and testing the expected behavior of the digital circuit designs. The current generation of hardware simulation tools evaluate each assertion separately by converting them into finite state automatons before simulation. In this dissertation, we propose an efficient technique for linear temporal logic (LTL) assertion evaluation. The proposed technique, EAST (Efficient Assertion Simulation Techniques), creates a shared data structure from the set of assertions using some simple rules, based on the operators during preprocessing. EAST infers the decision of the assertions during simulation without evaluating the assertion expressions. This approach is scalable for large designs.

Akin to software configuration management, it is becoming commonplace to maintain large hardware design code-bases with hardware configuration management tools. A missing piece of crucial technology in the approach of hardware configuration management tools is to manage design verification across evolving hardware designs. In this work, we propose an efficient methodology, EvoDeb for automatically localizing design errors across design variants. EvoDeb can be seamlessly integrated into existing hardware design flows. Experimental results exhibit the efficacy of our proposals.

Keywords: *Dynamic Assertion Based Verification, LTL, Debugging, Bug localization, Bug fix suggestion.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent times, with growing hardware design complexity, it is becoming extremely important to integrate assertion based property verification to verify the functionality of such designs. With such growth, it is common place to reuse existing designs and incrementally supplement their functionality, with the objective to cut down on overall design time. However, while debugging, designers are faced with the mammoth challenge of trying to localize the cause of errors by looking at long, failing execution traces. The designers treat the supplemented design as a fresh design while debugging and do not use the knowledge of the existing older stable version of the design. With these practical issues to solve, we set out to develop methodologies for fast and efficient assertion based verification and automated tool flow to reduce debugging efforts of the designer while debugging evolving designs.

Assertion-Based Verification (ABV) is assuming a significant role in the design validation flow of chip design companies. In recent times active participation from the design and EDA industries have led to the adoption of several formal languages for assertion specification. These include Forspec (of Intel) [8], Sugar/PSL (of IBM/Accellera) [9] and SVA (of Synopsys) [41]. Assertion specifications written in these languages are used to verify given implementations, either through formal property verification (FPV) techniques like model checking or through dynamic assertion-based verification (ABV) which is typically done by monitoring the properties over simulation runs. In this dissertation, we have used ABV as the background for our work with the input assertions written using Linear Temporal Logic(LTL).

Debugging denotes the process of detecting root causes of unexpected observable behaviour in design codes (e.g. an unexpected output value being produced or an assertion violation). Assertion violations give the developers a peek into what has gone wrong in during simulation, however the violated assertion might be affected by a large portion of the hardware design code, hence does not give a concise reason of the violation. Debugging errors is a difficult process, and often takes a significant fraction of the time in the development stage.

To ease the effort of manual debugging, of late there have been several attempts [17], [20], [27] to automate the debugging activity in the context of software programs by fully automated / semi-automated formal analysis of the program and the failed execution trace. These methods with rich theoretical foundations have found a moderate degree of acceptance in the software debug community. In the context of hardware pro- grams, research on automated debugging has been relatively scarce. In any programming community, it is a widely accepted reality in any large-scale development that a complex piece of program is never written from scratch. Usually a program evolves from one version to another. This is termed as program evolution. To allow management of diverse and complex hard- ware blocks along the evolution path, hardware management tools are gaining widespread industrial acceptance [12], [32]. The roots of these hardware management tools remain in the traditional software configuration management flows. It is natural to think at this juncture, whether the debugging of hardware designs can be automated for an evolving design, which forms one of the core motivations of this work.

1.1 Motivation of the dissertation

Given the fact, that the set of assertions for a design under test share multiple common signals, we focus on devising an efficient strategy for assertion evaluation in such a way that we are be able to infer the evaluation results of the assertions without actual evaluation of the assertions. Thereby in this dissertation, we propose a shared graph data structure based assertion evaluation strategy devised on the aforementioned ideal.

When we change a piece of code to produce a new version, we may introduce bugs. The change introduced may either be structural or a behavioural one, depending on the intent of the change and the original code. In particular, we study in this dissertation the effect of changes in some specific programming constructs in the Verilog programming language, and show how the presence of the earlier version can help in debugging the new one. The effect of a change varies depending on the semantics of the programming construct. A change in the sensitivity list of a programming statement (e.g. always / assign in Verilog, process in VHDL) may lead to new executions of the sensitised block as we explain in the Chapter 4. Similarly, changes in conditional statements may lead to different program paths being followed at simulation time. The effect of a change may percolate from one block to another as well. Bugs resulting out of such programming changes are extremely hard to debug, considering the fact that neither a textual difference of the source nor of the execution profile carries enough semantic meaning for which the change was initiated. Thereby we devise an efficient means for debugging change-induced bugs in the context of a hardware design flow.

1.2 Contribution of the dissertation

In this dissertation, we propose two methodologies for speeding up and at the same time reducing development efforts of hardware designs. One of the aspects in focus is regarding design of an efficient simulation strategy for Linear Temporal Logic (LTL) that leverages the presence of common signals across multiple assertions in an assertion suite to minimize the computational overhead of assertion evaluation during simulation. The second aspect of the dissertation focuses on efficient use of a stable version of source code of hardware designs for debugging an evolved version of the same design. Our proposed methodology can successfully pinpoint control flow errors, code missing errors and even incorrect data assignments. With rapid pace of evolution of existing hardware devices be it cheap sensors to high end flagship electronics products to keep up with the market demands, our proposed methodology is poised to play a crucial role in assisting developers in reducing debugging efforts during design of such evolved hardware systems.

1.3 Organization of the dissertation

The rest of the dissertation is organized into 6 chapters. A summary of the contents of the chapters is as follows:

Chapter 2 Detailed study of existing relevant research and a brief introduction to the semantics of LTL is presented here.

Chapter 3 The chapter presents a novel and efficient simulation strategy for LTL simulation based on shared graph data structures.

Chapter 4 The chapter presents an automated methodology for debugging change induced bugs in the source code of evolving hardware design, by leveraging the stable version of the hardware design.

Chapter 5 The chapter demonstrates the performance of our proposed methodologies on standard benchmarks.

Chapter6 The chapter concludes this thesis and presents avenues for future research based on the presented methodologies.

Chapter 2

Background and related work

In this chapter, we first present a few background concepts necessary for understanding our work. We also present an overview of different schemes proposed in literature in the field of verification and debugging of hardware systems, that are relevant to our context.

2.1 Background

In this section, we discuss a few background concepts.

2.1.1 Propositional Logic

Propositional logic is widely used in diverse areas such as database queries, in artificial intelligence, automated reasoning etc. A proposition is a sentence which is either true or false. If a proposition is true, then we say its truth value is true, and if a proposition is false, we say its truth value is false. The syntax of formulas in propositional logic is defined by the following grammar:

$$\begin{aligned} formula &= formula \wedge formula | \neg formula | (formula) | atom \\ atom &= BooleanIndicator | True | False \end{aligned}$$

Other Boolean operators such as OR ($\vee$) can be constructed using AND ($\wedge$) and NOT ($\neg$).

2.1.2 Satisfiability

In computer science, Boolean, or propositional, satisfiability (often written SATISFIABILITY or abbreviated SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it establishes if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to *TRUE*. If no such assignments exist, the function expressed by the formula is identically *FALSE* for all possible variable assignments. In this latter case, it is called unsatisfiable, otherwise satisfiable. SAT was the first known example of an *NP-complete* problem [1].

2.1.3 Linear Temporal Logic

The use of temporal logics [15] in verification was proposed by Pnueli in a seminal paper [35]. Since then several different logics have been proposed for specifying temporal properties. All these logics use the two basic temporal operators – *next* and *until*. Some of these logics also use additional temporal operators that can be derived out of the basic two. The logics differ in terms of how we are allowed to mix these operators to express the desired formula.

In this section, we introduce the popular temporal operators and the logics that are built around them. In this part we also introduce some formalisms in an intuitive way that show us how these logics are interpreted over time.

The basic temporal operators

The formal introduction to a language has two main parts, namely the *syntax* and the *semantics*. The syntax defines the *grammar* of the language – it tells us how we may construct properties using the basic set of signals and operators. The semantics define the *meaning* of the properties.

The semantics of the traditional temporal logics were defined over *closed systems*, which are finite state machines without any inputs. This tradition has been followed in languages such as SVA and PSL as well – there is no distinction between input and non-input variables in these languages. At this point we present the semantics of these languages in the traditional form over a non-deterministic finite state machine. Open systems (modules having input bits) can be modeled by treating the input bits also as state bits. This typically yield a non-deterministic state machine, since the choice of inputs in the next state lies with the environment, and is not a function of the present state.

Suppose J is a finite state machine having k state bits. Each of the 2^k valuations of these state bits represent a *state* of the machine. Let S denote the set of these states. Let R denote the state transition relation of J. R consists of pairs of states, (s_i, s_j), where it is possible to transit from state s_i to state s_j. Finally, J has a start state s. Formaly we say that J is a tuple $\langle S, s, R \rangle$.

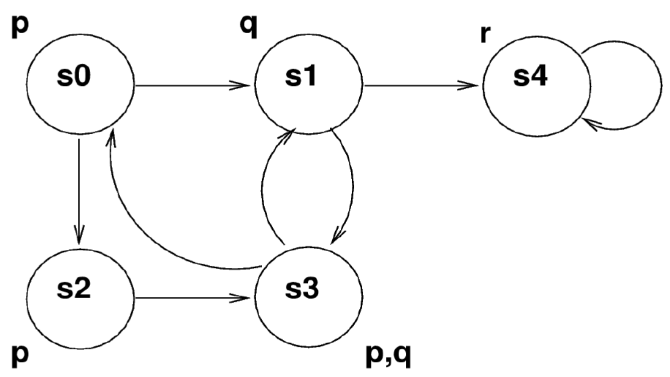


Figure 2.1: A sample finite state machine

Example 2.1 *Fig 2.1 shows a 3-bit finite state machine. Let the state bits be* n_0, n_1, n_2*. The state bits are shown on the nodes. The start state is* s_0*. Fig 2.1 shows 5 states – the remaining three states are not reachable from the start state and are not shown. The circuit has three outputs, which are functions of the state bits. These are:*

$$\begin{aligned} p &= n_0 \lor n_1 \\ q &= n_2 \\ r &= \neg n_0 \land \neg n_1 \land \neg n_2 \end{aligned}$$

The nodes of Fig 2.1 are labeled by the outputs that are true at that state. We shall use this toy example to demonstrate the meaning of various temporal properties. □

Intuitive explanation

To convey the semantics of the basic temporal operators, we first introduce the notion of a *run* (alternatively, a *path* or a *trace*). A run, π, of J is a sequence of states, $\nu_0, \nu_1, \ldots$, where $s = \nu_0$ is the start of the run, and for each i, ν_i represents a state in S, and R contains a transition from the state represented by ν_i to the state represented by ν_{i+1}. In other words, the run is a sequence of states representing a valid sequence of state transitions of J. For example the run, $\pi = s_0, s_1, s_3, s_1, s_4, \ldots$, is one run of the state machine shown in Fig 2.1. States of the machine may be revisited in the run – for example we have $\nu_1 = \nu_3 = s_1$ in π. The run, $\pi' = s_0, s_2, s_0, \ldots$, does not belong to this state machine, since it has no transition from s_2 to s_0.

Let us now consider the two fundamental temporal operators, namely *next*($\bigcirc$) and *until*($\mathcal{U}$), and a run $\pi = s_0, s_2, s_3, \ldots$.

Next operator($\bigcirc$): A property, $\bigcirc f$, is true at a state of a run iff the property f is true at the next state on the run. For example, $\bigcirc q$ is false at the state, s_0, of the run, $\pi = s_0, s_2, s_3, \ldots$, since q is false at the next state s_2. The property $\bigcirc \bigcirc q$ is true at s_0, of π, because q is true at s_3.

Until operator($\mathcal{U}$): A property, $f \mathcal{U} g$, is true at a state of a run iff the property g holds on some future state, z, of the run, and the property f holds on all states preceding z on the run. For example, the property, $p \mathcal{U} q$, is true at the start state of π, since q is true at the state s_3 and p is true at the states s_0, s_2 preceding s_3 in π. The property, $p \mathcal{U} r$, is false on all paths of Fig 2.1, because no r-labeled state can be reached along a p-labeled path starting from s_0.

We now define the two other operators namely, *always*($\Box$) and *eventually*($\Diamond$). To do this, we need the definitions of the propositions, *TRUE* and *FALSE*. We say that the proposition, *TRUE*, holds in all states, and the proposition, *FALSE*, is false in all states.

Eventually operator($\Diamond$): A property, $\Diamond f$, is true at a state of a run iff the property f holds on some future state in the run. Since the proposition, TRUE, holds on all states, we can express the $\Diamond$ operator using the $\mathcal{U}$ operator as:

$$\Diamond f \;=\; TRUE\, \mathcal{U}\, f$$

The property, $\mathcal{U}\ q$, holds on all runs starting from s_0 in Fig 2.1. The property, $\mathcal{U}\ r$, does not hold in the run which loops forever in the loop s_0, s_2, s_3, s_0.

Always operator ($\Box$): A property, $\Box\ f$, is true on a run iff the property f holds on all states of the run. This is the same as saying that $\neg f$ never holds on the run. In other words we may write:

$$\begin{aligned} \Box\ f &= \neg\Diamond\ \neg f \\ \Diamond\ f &= \neg\Box\ \neg f \end{aligned}$$

The first equation allows us to express the $\Box$ operator using the $\Diamond$ operator, and in turn, in terms of the $\mathcal{U}$ operator. The second, says: *sometimes is not never* – there is a seminal paper with this title by Leslie Lamport [33].

The property, $\Box\ p$ is true in the run which loops forever in the loop, s_0, s_2, s_3, s_0, in Fig 2.1. The property is false in all other runs of the same state machine.

The duality between the *always* and *eventually* operators is not surprising. In fact, it is a variant of DeMorgan's Laws when we interpret the properties over time. This is because:

$$\begin{aligned} \Diamond f &= f \vee \bigcirc f \vee \bigcirc \bigcirc f \vee \bigcirc \bigcirc \bigcirc f \ldots \\ &= \neg(\neg f \vee \bigcirc \neg f \vee \bigcirc \bigcirc \neg f \vee \bigcirc \bigcirc \bigcirc \neg f \ldots) \\ &= \neg(\Box \neg f) \end{aligned}$$

Linear Temporal Logic (LTL) is the most popular among linear time logics. We can define the syntax of linear temporal logic recursively as follows:

- All Boolean formulas over the state variables are LTL properties.
- If f and g are LTL properties, then so are: $\neg f$, $\bigcirc f$, and $f\mathcal{U}g$.

Formal semantics

It is very important to know the formal semantics of a formal property specification language. If the semantics is specified ambiguously, there may be a gap between the property that the designer intends to express and the formal property tool's interpretation of the property that she writes. Bugs may hide in this gap thereby defeating the whole purpose of *formal* property verification. Language lawyer volunteers who make up the working groups of the language standards committees spend years debating over the exact formal semantics of the languages that they standardize. The goal of standardization is to ensure that languages with precise definitions are made available to improve communcation within the industry.

The problem with understanding formal semantics is that they are replete with terse notations. It is widely suspected that the intimidating nature of the notations used in existing literature on formal property verification is one of the main deterrents to its wider adoption in practice.

Let $\pi = \nu_0, \nu_1, \ldots$ denote a run, and $\pi^k = \nu_k, \nu_{k+1}, \ldots$ denote the part of π starting from ν_k. We use the notation $\pi \models f$ to denote that the property f holds on the run π. Given a run π, we also use the notation $\nu_k \models f$ to denote $\pi^k \models f$. In other words, a property is said to be true at an intermediate state of the run iff the fragment of the run starting from that state satisfies the property. The formal semantics of the basic temporal operators are as follows:

- $\pi \models Xf$ iff $\pi_1 \models f$
- $\pi \models f\ \mathcal{U}g$ iff $\exists j$ such that $\pi_j \models g$ and $\forall i, 0 \le i < j$ we have $\pi_i \models f$.

Fg is a short-form for TRUE $U\ g$, and Gf is a short-form for $\neg F \neg f$.

Bounded temporal logics

The temporal operators discussed so far, namely *next* (X), *until* ($\mathcal{U}$), *always* ($\Box$), and *eventually* ($\Diamond$), are *temporal* because they can define sequences of events over time. Significantly, none of these operators with the exception of the *next* operator, attempt to *quantify* time. For example the property, $\Diamond f$, requires f to be true in future, but does not specify any time bound by which f needs to be true.

Real time temporal operators are intuitively simple extensions of the basic *untimed* temporal operators where we annotate the operator with a time bound. The real time extensions of CTL and LTL simply use these bounded operators (as well as the unbounded ones).

The bounded Until operator: The property $fU_{[a,b]}g$ is true on a run, $\pi = s_0, s_1, \ldots$, iff there exists a k, $a \leq k \leq b$ such that g is true at s_k on π, and f is true on all preceding states, $s_0, \ldots, s_{k-1}$. Formally,

$$\pi \models f\ U_{[a,b]}\ g \text{ iff } \exists k, a \leq k \leq b, \nu_k \models g \text{ and } \forall i, 0 \leq i < k \text{ we have } \nu_i \models f$$

The bounded LTL property $p\ U_{[1,3]}\ q$ is true at the state s_0 of Fig 2.1. The bounded CTL property:

$$A[p\ U_{[1,3]}\ E[q\ U_{[1,2]}\ r]]$$

is also true at s_0. This is because s_3 and s_1 satisfy $E[q\ U_{[1,2]}\ r]$ (since they can reach s_4 within the time bound $[1,2]$), and we reach s_1 or s_3 along all paths from s_0 within the time bound $[1,3]$.

The bounded Eventually operator: The property $\Diamond_{[a,b]}g$ is true on a run, $\pi = s_0, s_1, \ldots$, iff there exists a k, $a \leq k \leq b$ such that g is true at s_k on π. For example, the bounded LTL property $\Diamond_{[1,3]}q$ is true at the state s_0 of Fig 2.1.

The bounded Always operator: The property $\Box_{[a,b]}f$ is true on a run, $\pi = s_0, s_1, \ldots$, iff f is true in every state in the sequence, $s_a, \ldots, s_b$. The bounded LTL property $\Box_{[0,1]}\neg r$ is true at s_0 of Fig 2.1 no run can reach s_4 is less than 2 cycles.

Real time operators are extremely useful in practice. Most design properties have a well defined time bound, and must be satisfied within that time.

Since the real time operators deal with finite bounds, a and b, they can be expressed in terms of the X operator. For example, the property $\Diamond_{[2,4]}q$ can be rewritten as:

$$\Diamond_{[2,4]}\ q\ =\ \bigcirc\bigcirc(q\ \vee\ \bigcirc q\ \vee\ \bigcirc\bigcirc q)$$

and $p\ \mathcal{U}_{[3,4]}\ q$ can be rewritten as:

$$p\ \mathcal{U}_{[3,4]}\ q\ =\ (p\ \wedge\ \bigcirc p\ \wedge\ \bigcirc\bigcirc p)\ \wedge\ \bigcirc\bigcirc\bigcirc(q\ \vee\ (p\ \wedge\ \bigcirc q))$$

The first part of the property specifies that p be must be true in the present cycle and the next two cycles. The second part of the property specifies that q must be true in the third cycle, failing which, p must be true in the third cycle and q must be true in the fourth cycle.

Therefore, real time operators actually help us to succinctly express properties that would require too many $\bigcirc$ operators otherwise.

2.2 Related Work

In recent times, active participation from the design and EDA industries have led to the adoption of several formal languages for assertion specification. These include Forspec [7], Sugar/PSL [9] and SVA [44]. There is a rich body of literature ([8, 11, 6, 14]) for dynamic ABV of LTL [36] and other LTL based languages. Industrial simulators such as [5], [4] translates each assertion to a finite state automaton, and thereafter deploys a thread based simulation strategy for dynamic assertion evaluation. In [11], PSL assertions are translated to Verilog and thread based monitoring is undertaken for simulation based assertion checking. In [8], assertions are compiled to deterministic automata and simulated in a thread-less uniprocessor environment. In [41], a methodology has been proposed for development of temporal monitors for SystemC.

To the best of the our knowledge, there has not been any research work reported till date on automated bug localization for evolving HDL programs that take into account the evolution and version change information between the two program versions. However, some work has been done on automated bug localization for HDL programs in general. In this area, basically there are two approaches to fault localization [29]: simulation-based approaches [25, 32, 37, 45] and symbolic approaches [24]. Symbolic approaches are accurate but suffer from combinatorial explosion whereas simulation-based approaches, although scalable with design size, require numerous test vectors to be accurate enough.

There has been only a few articles on slicing [28] and its applications in the HDL context. [21] reports the application of static program slicing to VHDL. In [30, 31] the authors describe a diagnosis tool for VHDL that employs functional fault models and reason from first principles by means of constraint suspension. [26] discusses an application of algorithmic debugging to automatic fault localization in VLSI designs. In [18], a comprehensive overview of automated source-level fault localization techniques are given. Their work is based on modeling of abstract behavior and extracting functional or value-change dependencies for HDL programs. [19] presents the idea behind model-based diagnosis and its application to localizing faults in Verilog programs is discussed. In [22], a hierarchical approach for automated debugging is introduced. In [16], synchronization bugs are identified by applying a bug model to isolate a set of possible bug candidates. In addition to fault localization, [23] proposed an approach to explain the fault for improved automated diagnosis. There is a steady foray of automated debugging tools in commercial arena. A prime example of that is Synopsys Verdi automated debug system [40]. This tool does efficient tracing of behavior for code analysis, explores the interaction between design, assertions and testbench and provides an intuitive graphical front-end. Recently, [39] commercialized a tool for automated bug localization, explanation of the bug together with hints to fix it. The closest to our work is reported by [43], with a bug localization tool called *PinDown*. This tool provides a combined version management system for the design and the bugs. With every new design modification, an interactive diagnosis interface is provided to search through the revisions that might have caused a bug. However, no automated analysis of bug localization across revisions is done, which is exactly what we propose in this work. Our work can be looked at as an adaptation of the debugging efforts for evolving software programs proposed in [17] and the slice and WP construction in [28] in the evolving design debug context. The default concurrency semantics of HDLs along with several other programming constructs (sensitivity list, process etc) makes our approach novel.

Chapter 3

EAST : Efficient Assertion Evaluation Techniques

Assertion-Based Verification (ABV) plays a vital role in the design validation flow of chip design companies. With growing hardware design size, it is of utmost importance to have a low overhead and efficient ABV. LTL is a commonly used language for assertion specification. In this chapter, we present a novel methodology for LTL assertion simulation consisting of two stages namely preprocessing stage and simulation stage. The semantics of LTL has been already explained in detail in section 2.1.3. The preprocessing stage processes the assertion set to store the requisite information in a look up table format. This look up table can be visualized as a shared graph data structure. By using look up tables, instead of individual assertion for simulation in the simulation stage, we leverage the presence of common propositional variables across assertions to accelerate the simulation of assertions.

3.1 Existing approach for assertion evaluation

We present a small set of LTL assertions and explain briefly how they are simulated. Let the set of assertions be the following :-

$P_1 \ : \square(a \vee c)$

$P_2 \ : \square(a \wedge \bigcirc b)$

$P_3 \ : \Diamond(c \vee d \vee e)$

The common approach in the semiconductor industry is transaction-based monitoring, which in effect constructs the monitor dynamically. For example, in monitoring the property $\square(\neg p \vee \Diamond q)$, the simulator spawns a new thread waiting for q each time it observes

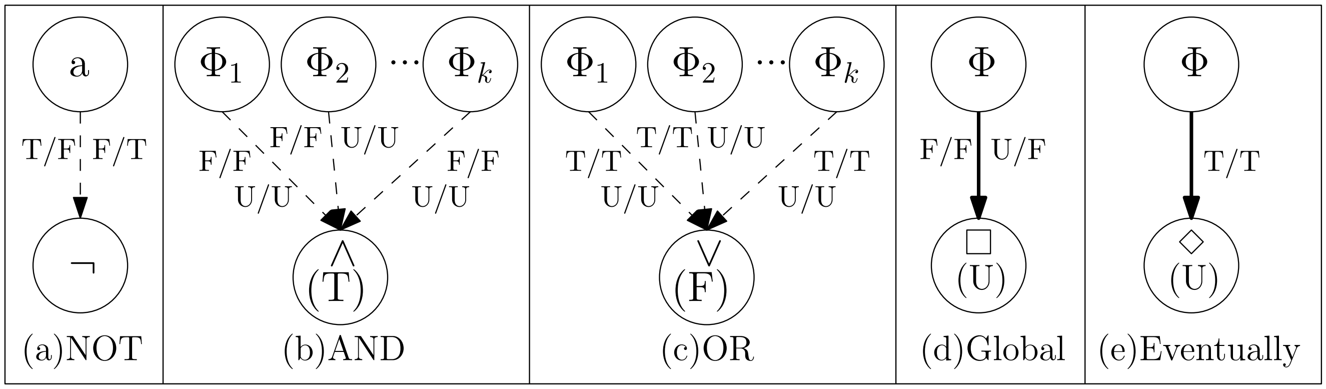


Figure 3.1: Graph data structures for various operators

$p \wedge \neg q$. Such a thread is called a transaction. With this approach the number of active transactions is potentially unbounded, resulting in degraded performance for long simulation runs. (The conventional methodology [10] advises users to prevent this problem by bounding the number of active transactions.) Each assertion is evaluated separately by the assertion evaluation engine, by reading the values of the propositional variables and thereafter computing the result of the assertion as per the LTL semantics by implementing transaction-based monitoring, with automata-based monitors.

However, from the above example,it is easy to note that if c is found to be $True$ during evaluation of the assertion P_1, we can conclude the expression $(c \vee d \vee e)$ in assertion P_3 is also $True$. Similarly if a is found to be $False$ using evaluation of assertion of P_1, we can say the expression $(a \wedge Xb)$ in assertion P_2 will evaluate to $False$. Thus, we can see the presence of common variables across assertions can be used to minimize evaluation efforts of the assertion set and thereby speedup simulation.

For ease of understanding of the readers, we proceed by explaining the building blocks of a shared graph data structure in the following section and then present the preprocessing and simulation steps in Section 3.3 and 3.4 respectively. Thereafter we present a couple of optimizations to speed up our methodology in Section 3.5. In addition, we give a detailed walk through of our methodology in Example 3.1.

3.2 Building blocks of the shared graph data structure

Our methodology uses a look up table for simulation of the assertions which can be equivalently represented as a shared graph data structure. We assume that the assertions are present in Negation Normal Form (NNF)[2]. Before we explain our methodology in detail, we present the building blocks of the graph, namely nodes, edges and basic rules of graph generation that will be used in the subsequent sections.

- Nodes : There are three types of nodes.
  - Input Node : Each propositional variable present in the set of assertions to be simulated is assigned an input node. Input nodes have zero in-degree.
  - Internal Node : Each internal node is associated with a subexpression, which consists of only a single type of operator and one or more operands. The internal node holds the evaluated value of the subexpression. Internal nodes have a non-zero in-degree as well as a non-zero out-degree.
  - Assertion Node : Each assertion node corresponds to a particular assertion and holds the result of that assertion across cycles. Assertion nodes have non-zero in-degree and zero out-degree.

  The level of a node is defined as follows.
  Level of input node = 0
  Level of other nodes = max(level of the immediate predecessors of the node) +1

- Edges : There are two types of edges.
  - Strong Edge : A directed edge between two nodes which causes the destination node to be assigned a fixed value that is not going to change in the future clock cycles is termed as a strong edge, marked by thick lines. Thereby, once a strong edge has been traversed, the destination node is never evaluated again in the future. Figure 3.1 (d) shows the example of a strong edge, where the source node contains ϕ and the destination node contains $\Box$, shown as thick line.
  - Ordinary Edge : A directed edge that connects two nodes, where the value of the destination node may change across cycles. Figure 3.1 (a) shows the example of an ordinary edge, shown as dashed line.

  Edges are marked using val_{in}/val_{out} notation where val_{in} and val_{out} take values from the set $\{True(T), False(F), Unknown(U)\}$. For example, in Figure 3.1 (a), the edge is annotated by T/F and F/T. The markings signify that if the source node has value val_{in}, then val_{out} is propagated to the destination node. There can be one or more markings corresponding to an edge.

We present simple rules for creating an equivalent graph representation of the LTL expressions below. We define path as a sequence of truth values spread across consecutive cycles, corresponding to one evaluation result of the assertions.

1. ***Operator NOT ($\neg$)***: For an expression, $\neg\phi$, where ϕ is a propositional variable, the corresponding graph is shown in Figure 3.1(a). If the source node is $True$ ($False$), then the destination node will be set $False$ ($True$). Hence the edge markings are T/F and F/T.

2. ***Operator AND ($\wedge$)***: For the expression $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$, the corresponding graph is shown in Figure 3.1(b). If any one of the operands is $False$, then the result is $False$, irrespective of the value of the other operands. Hence the edges from source to destination

are marked by F/F. If the source is $Unknown$ (which may happen for temporal expression), then the destination node can be $Unknown$ and hence the edges have the marking U/U as well. The default value of the destination node in this case is set to $True$ because it implies that none of the operands were $False$ / $Unknown$, otherwise the node would have been set to $False$ / $Unknown$ by the corresponding source node. The destination node will be set by a value only when it holds $True$ / $Unknown$ and a different value is propagated through the edge. For example, if the destination node is currently holding a value $True$, it will be overwritten with the value $False$ / $Unknown$ depending on what value appears on the edges. It is to be noted that the value $True$ is never propagated in case of the AND operator, as apparent from Figure 3.1(b) edge markings. These rules are summarized in Table 3.1 a) where EV is the Evaluated Value sent via the edge and NV is the source node's existing value.

EV/NV	T	F	U
F	F	F	F
U	U	F	U

a) AND

EV/NV	T	F	U
T	T	T	U
U	U	U	U

b) OR

Table 3.1: Rules for setting operator node values

3. ***Operator OR* ($\vee$)**: For the expression, $\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$, the corresponding graph is shown in Figure 3.1(c). If any one of the operands is $True$, then the result is $True$, as per the semantics of the OR operator. Hence the edges from the operator node to operand node are marked by T/T. If the operand is $Unknown$, then the operator node will be $Unknown$ and hence the edges have the marking U/U as well. The default value of the operator node is $False$ because it implies that none of the operands were $True$ / $Unknown$. The rules for evaluation of OR operator node are presented in Table 3.1 (b).

4. ***Operator GLOBAL* ($\Box$)**: For the expression $\Box\phi$, the corresponding graph structure is shown in Figure 3.1(d). The GLOBAL operator signifies that ϕ has to hold on the entire path from the current cycle onwards. Hence if ϕ is $False$ or $Unknown$ in a given clock cycle, the expression will be $False$ all through from that clock cycle. The source to destination is thereby connected by a strong edge marked F/F and U/F. The default value for the $\Box$ operator node is $Unknown$.

5. ***Operator EVENTUALLY* ($\Diamond$)**: For the expression $\Diamond\phi$, the corresponding graph structure is shown in Figure 3.1(e). The EVENTUALLY operator signifies that ϕ eventually has to hold. Once ϕ is $True$, the expression will be $True$ all through from that clock cycle, and hence the operator to operand is connected by a strong edge marked T/T. The default value for the $\Diamond$ operator node is $Unknown$.

6. ***Operator UNTIL* ($\mathcal{U}$)**:The graph structure corresponding to $\phi_1\ \mathcal{U}\ \phi_2$ is shown in Figure 3.2(a). UNTIL operator signifies ϕ_1 has to hold at least until ϕ_2, which holds at the current or a future position. If internal $\vee$ node is $False$, from the definition of UNTIL, the $\mathcal{U}$ operator node is set to $False$ from the current cycle. On the other hand, if ϕ_2 is $True$,

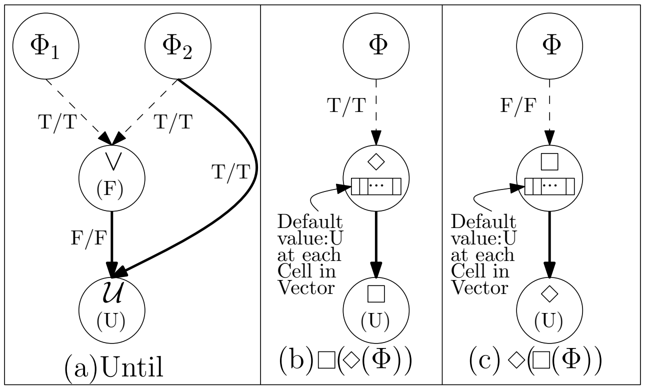

Φ_1
Φ_2
T/T
T/T
T/T
$\vee$
(F)
F/F
$\mathcal{U}$
(U)
(a)Until
Φ
T/T
$\Diamond$
Default value:U at each Cell in Vector
$\Box$
(U)
(b)$\Box(\Diamond(\Phi))$
Φ
F/F
$\Box$
Default value:U at each Cell in Vector
$\Diamond$
(U)
(c) $\Diamond(\Box(\Phi))$

Figure 3.2: *Graph Structure of Until and nested temporal operators*

the UNTIL operator node will be *True* from the current cycle onwards. The default value here is *Unknown*.

7. ***Operator NEXT ($\bigcirc$)***: We do not explicitly create a node for the NEXT operator, rather at each node, we store a delay t_d of that node as a property of that node. We use the following properties for the next operator, i.e., $\bigcirc(\Phi_1 \vee \Phi_2) = \bigcirc(\Phi_1) \vee \bigcirc(\Phi_2)$; $\bigcirc(\Phi_1 \wedge \Phi_2) = \bigcirc(\Phi_1) \wedge \bigcirc(\Phi_2)$; $\bigcirc(\neg(\Phi_1) = \neg(\bigcirc(\Phi_1))$. Therefore we can associate the next operator with the variable itself, instead of associating it with an expression. Each assertion is transformed into our desired form. We now introduce the notion of the delay corresponding to a node. The delay of an input node is the number of next operators preceding the corresponding variable present in this node. The delay of an internal node is the maximum delay of its predecessor nodes.

Nested temporal operators require special handling during simulation. We explain below two specific cases.

- $\Box(\Diamond(\phi))$: The corresponding graph structure is shown in Figure 3.2(b). Instead of a single value, the EVENTUALLY operator node stores a vector of truth values. Each value in the vector corresponds to the value of an instance of the EVENTUALLY operator starting at each new cycle. Whenever the propositional expression ϕ evaluates to *True*, the corresponding value in the vector is set to *True*. At the end of simulation, the value of GLOBAL operator node is set to *True* if all the values in the EVENTUALLY operator node value vector are *True*, otherwise to *False* since it implies that at least one instance of EVENTUALLY operator evaluated to either *Unknown* or *False*.
- $\Diamond(\Box(\phi))$: The treatment of $\Diamond(\Box(\phi))$ is similar. The corresponding graph structure is shown in Figure 3.2(c). At the end of simulation, the value of the GLOBAL operator node is set to *True* if at least one value in the value vector of the GLOBAL operator is not *False*.

3.3 Pre-processing assertions to generate Look Up Table

The preprocessing stage of the methodology is executed only once for a set of assertions and generates a look up table corresponding to the assertions. We explain below what a look up table is and how we obtain it from the graph structures presented earlier. For each source node we store a list of destination nodes that need to be considered for this source node with value annotations as appropriate. For an edge n_i to n_j with edge marking val_{in}/val_{out}, we store node n_j with val_{out} in the associated list (also called the val_{in}) list of node n_i. In addition, the type of the edge is also saved in the entry corresponding to n_j in the same list of n_i. Thus during simulation if node n_i has value val_{in}, we can set the value of the successor nodes of n_i by looking up the val_{in} list of node n_i that we created in the preprocessing step. We define this set list structures as the Look Up Table (LUT). It is to be noted that such a look up table captures all the elements of the corresponding graph data structures.

Initially the assertions are converted to postfix form using standard transformations [34]. and processing individually in a manner similar to postfix evaluation to generate the look up table corresponding to the set of assertions. We explain the preprocessing stage using a detailed example using the set of assertions presented earlier in the chapter.

Example 3.1 *Let the set of assertions be the following :-*

P_1 : $\Box(a \vee c)$

P_2 : $\Box(a \wedge \bigcirc b)$

P_3 : $\Diamond(c \vee d \vee e)$

The set of assertions converted to postfix is :-

P_1 : $ac \vee \Box$

P_2 : $ab \bigcirc \wedge \Box$

P_3 : $cd \vee e \vee \Diamond$

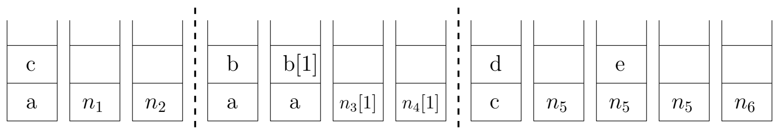


Figure 3.3: *Preprocessing the assertion set*

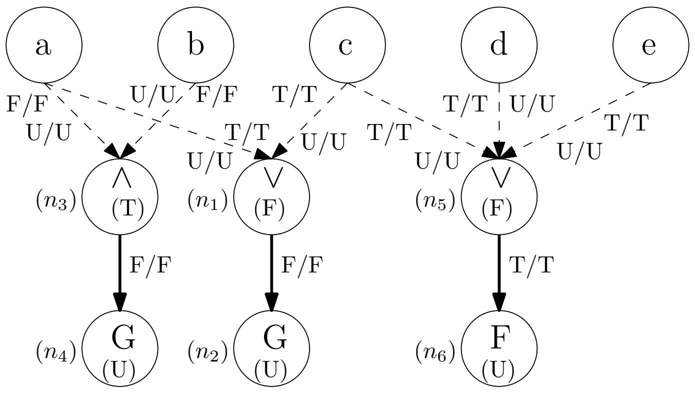


Figure 3.4: *Shared Graph Data Structure*

The input nodes a, b, c, d and e are at level 0 from the definition of level. For assertion P_1, a and c are pushed to the stack. Thereafter, the $\vee$ operator is encountered and hence a and c are popped from stack. We create a new operator node n_1 corresponding to this operator with default value $False$ as per the Rule 3. In addition, n_1 is added to $True$ and $Unknown$ list of a and c respectively as per Rule 3. Since a and c are at level 0, level of n_1 is set to 1 as per the definition of level. Then, n_1 is pushed into the stack. We encounter the GLOBAL operator and pop n_1 from the stack. A GLOBAL operator node n_2 is generated with default value $Unknown$ and n_2 is added to the $False$ list of n_1 as per Rule 4 with strong edge marking. This concludes preprocessing of assertion P_1. Assertion 2 presents an interesting case due to the presence of NEXT operator before operand b. Operands a and b are pushed to the stack. On encountering the NEXT operator, b is marked with delay equal to 1 as per Rule 8 without creating any additional node. For the AND operator, a new AND operator node n_3 with default value $True$ is created and it is added to the $False$ and $Unknown$ lists of a as well as that of c according to rule 2. The delay of n_3 is set to 1 since the maximum delay of its immediate predecessors is 1. The GLOBAL operator node n_4 is generated with default value $Unknown$ and n_4 is added to the $False$ list of n_3 as per Rule 4 with strong edge marking. n_4 is also assigned a delay 1 as its only predecessor n_3. For assertion 3, we proceed similarly. The only fact to note is that for the second $\vee$ operator in P_3 that we encounter, we do not create an additional node, rather we place in the $True$ and $Unknown$ list of input node e, the previously created $\vee$ operator node n_5. The look up table generated after completion of the preprocessing stage, is presented in Table 3.2. For ease of visualization, the shared graph structure is also presented in Figure 3.4.

The generated look up table is used as input to the simulation stage described in Subsection 3.4.

Level	**Type**	**Node**	**List type**	**List**
0	Input	a	True False Unknown	n_1 n_3 n_1,n_3
0	Input	b[1]	False	n_3
0	Input	c	True Unknown	n_1,n_5 n_1,n_5
0	Input	d	True Unknown	n_5 n_5
0	Input	e	True Unknown	n_5 n_5
1	$\vee$	n_1	False	n_2[s]
1	$\wedge$	n_3[1]	False	n_4[s]
1	$\vee$	n_5	True	n_6[s]
2	$\Box$	n_2	-	-
2	$\Box$	n_4[1]	-	-
2	$\Diamond$	n_6	-	-

Table 3.2: Look Up Table

3.4 Assertion simulation using Look Up Tables

In this section, we discuss our thread based assertion simulation strategy. Nodes that are in assertions without any temporal operators, are evaluated using value substitution at the start of simulation and results are reported. Otherwise, in each cycle, a new thread is created for the evaluation. If the evaluation of an assertion starts at clock cycle t_0, each node n_i at delay t_d is evaluated at the $(t_0 + t_d)^{th}$ cycle. The maximum number of threads that can be active in memory simultaneously is equal to the maximum delay of any node in the shared data structure. We now present a level-based assertion simulation strategy. This method arranges the nodes in order of the level to which it belongs. The algorithm proceeds by initializing the nodes to their default values. Thereafter, it starts a new thread for simulation, say at clock cycle t_0. In clock cycle $t_i(>= t_0)$, the thread reads the simulation inputs for input nodes with temporal delay $(t_i - t_0)$ and then processes nodes in increasing order of level. Processing involves checking the current value val_{curr} of the node n_i and setting the values of the nodes in the val_{curr} list of the node n_i as saved in the LUT. Once all the nodes in the current level have been processed, the nodes in the next level are taken up. If an internal node has not been set to a value, it stays at its default value. To take into account the delay of the nodes, the simulation thread only starts by processing the input nodes at the current delay and waits across cycles to process the other input nodes at greater delays. The assertion nodes are set to their default value at start of simulation and shared amongst all the threads so that there is a consistent visible result of assertion evaluation. To do so, assignment of truth value to assertion nodes by threads is done in a thread safe manner, keeping in mind the execution semantics of LTL operators. Algorithm 1 presents the detailed methodology of the work done by a single thread. Intuitively, each thread reads the simulation values and handles the evaluation mechanism based on the look up table up to a maximum delay of any assertion.

Algorithm 1 Algorithm for level based assertion simulation

```
Input: Look-up table, max_level, max_temporal_depth in
Output: out
1: Initialization : Set each node to default value according to the type of node.
2: Start a new thread for evaluation
3: for $time_{curr} = 0$ to $max\_delay$ do
4:     Read simulation inputs to input nodes with temporal delay $time_{curr}$
5:     for $level = 0$ to $max\_level$ do
6:         for $node_i \in level$ and ($node_i[time] == time_{curr}$ or $isEvaluated(node_i) == True$) do
7:             if ($node_i == True$) then
8:                 Set values to nodes in $True$ list of $node_i$
9:             else if ($node_i == False$) then
10:                Set values to nodes in $False$ list of $node_i$
11:            else
12:                Set values to nodes in $Unknown$ list of $node_i$
13:            end if
14:            Set evaluated flag for the new value assigned nodes to $True$
15:        end for
16:    end for
17:    Wait till next clock cycle
18: end for
```

We explain the working of our algorithm on Example 3.1. The order of evaluation can be visualized as shown in Table 3.3 (a).We explain the demonstration of simulation shown in Table 3.4, using simulation inputs stated in Table 3.3 (b),

Level	Nodes[0]				Nodes[1]
0	a	c	d	e	b
1	n1		n5		n3
2	n2		n6		n4

a) Nodes visualized level-wise

Time	a	b	c	d	e
cc_0	T	F	T	F	T
cc_1	T	T	T	T	T
cc_2	T	F	T	F	T
...	...				

b) Simulation inputs

Table 3.3: Nodes of LUT visualized level-wise and the simulation inputs

- In clock cycle 0 (cc_0), Thread 0 (T_0) starts execution. We note that the maximum delay is 1 (due to P_2), and hence each thread will be alive for two clock cycles. T_0

cc_0	cc_1	cc_2
T_0		
n_1 = True n_5 = True	n_3 = True	
n_2 = U n_6 = True	n_4 = U	
	T_1	
	n_1 = True n_5 = True	n_3 = False
	n_2 = U	n_4 = False

Table 3.4: Simulation of assertions

reads the inputs a, c, d, e needed for this clock. As per the semantics of our level order simulation strategy, b is not present at level 0 and hence not read. As a is $True$, n_1 is assigned $True$. Similarly, since c is $True$, n_5 is assigned $True$. Since a is $True$ but b is not known, no value is assigned to node n_3 which is at delay 1. Thereafter, as n_5 is True, n_6 is assigned $True$ via a strong edge and hence n_6 is eliminated and reported as $True$. n_2 stays at assigned default value U. Thus we can observe that the algorithm proceeds level wise to assign the values to nodes in the next level depending on their present value.

- In clock cycle 1 (cc_1), T_0 reads the value of b which is True and it does not assign a value to n_3. n_3 stays at the default value $True$. Similarly, n_4 stays at its default value U. T_0 terminates at the end of this cycle. In this clock cycle, a new thread T_1 starts execution. Similar to T_0 at cc_0, T_1 assigns both n_1 and n_5 to $True$ at this cycle. However, now n_5 does not assign any value to n_6 since the strong edge was already traversed by T_0 in cc_0. n_2 stays at its default value U.
- In clock cycle 2 (cc_2), T_1 reads b which is $False$ and thereby sets n_3 as $False$, which in turn sets n_4 to $False$. n_4 is eliminated and $False$ is reported as the final value of n_4. T_1 terminates. Another new thread will begin and proceed as mentioned above.

The values of the assertions after 3 cycles of simulation (cc_0, cc_1, cc_2) are available as the assigned values of the nodes n_2, n_4 and n_6 as done above.

3.5 Optimizations

We propose two simple optimizations that would help in speeding up our methodology.

- Once an assertion with a GLOBAL or EVENTUALLY operator that has been evaluated to False or True respectively, has been processed, these assertions will not be affected by the simulation inputs in the future clock cycles. Thus the inputs that drive these assertions need not be considered as well in the future clock cycles, provided these inputs are not driving other assertions. This can be achieved by using an *outdegree* field in the input nodes. In the assertion nodes that do not change values once evaluated, the list of inputs (*inputlist*) driving the assertion is stored. Once the assertion has been evaluated, we decrement the *outdegree* of the nodes in the *inputlist* by one. If any of the input node has *outdegree* zero, it is not considered in the future clock cycles.

- For common subexpressions in different assertions, we have duplicate nodes in the same level. Thereby, in the preprocessing stage, we can merge nodes with identical immediate predecessors and edge marking in the same level, to reduce the number of lookups by utilizing these common subexpressions.

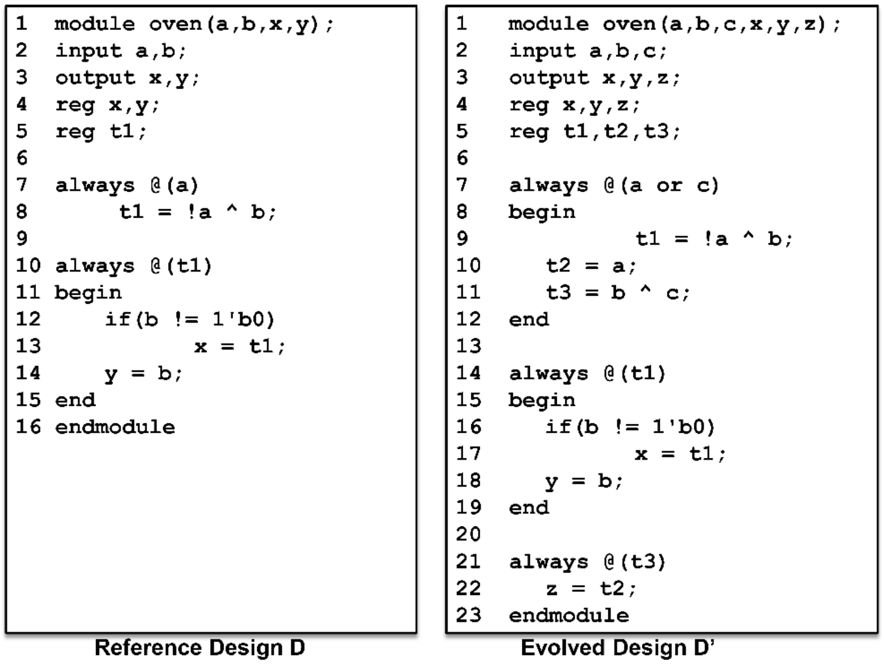
3.6 Conclusion

We presented the foundation of the LTL assertions simulation based on inferring the evaluation results of the assertions without actual evaluation by using a shared data structure implemented as a LUT. The developed framework can be easily integrated into existing standard simulation tool flows and the performance results are shown in Subsection 5.1.

Chapter 4

EvoDeb: Debugging Assertion failures in Hardware Designs using evolution information

Assertions provide a concise way of specifying expected behavior of a hardware design. Assertion violations present a brief idea of what has gone wrong in a hardware design to the developer. Current debugging methodologies do not leverage the knowledge of existing stable version of a hardware design for debugging an evolved version of the design. In this chapter, we study how the presence of a stable bug-free earlier hardware version can be effectively exploited for debugging bugs (assertion violations, structural bugs, etc) in an evolved buggy version. We employ a combination of dynamic program slicing and weakest precondition (as in [17]) in the hardware design context for effective bug localization. Our method involves simultaneously performing dynamic slicing and symbolic constraint analysis in both the programs - the stable version as well as the modified implementation. In conjunction with the dynamic slice, we perform a step-by-step weakest pre-condition computation along the dynamic slice. The constraints generated out of the weakest precondition computation in the two design versions are then compared to find new / missing constraints in the new version. There is a rich body of literature [6], [9] for localizing version change bugs in the software verification and debugging community. However, to the best of our knowledge, no such work has been proposed for debugging version change bugs in the context of designs written in Hardware Description Languages (HDL). As we show later in Chapter 5, this yields orders of magnitude reduction in the number of statements to be examined by the developer.


Reference Design D
Evolved Design D'

```
module oven(a,b,x,y);
input a,b;
output x,y;
reg x,y;
reg t1;

always @(a)
     t1 = !a ^ b;

always @(t1)
begin
   if(b != 1'b0)
          x = t1;
   y = b;
end
endmodule
```

```
module oven(a,b,c,x,y,z);
input a,b,c;
output x,y,z;
reg x,y,z;
reg t1,t2,t3;

always @(a or c)
begin
          t1 = !a ^ b;
   t2 = a;
   t3 = b ^ c;
end

always @(t1)
begin
   if(b != 1'b0)
          x = t1;
   y = b;
end

always @(t3)
   z = t2;
endmodule
```

Figure 4.1: *Source code of the modules*

4.1 Demonstrative Example

In this section, we demonstrate our methodology through a simple example. Consider a simple Verilog design $\mathcal{D}$ with ports a, b, x, y and its evolved version $\mathcal{D}$' with additional ports c and z, with some added functionality, as shown in Figure 4.1. The evolved design D' in Figure 4.1 is expected to implement the same functionality as the reference design D with respect to the variables x and y while adding a new feature z. To implement the feature z, we have new internal registers $t2$ and $t3$ and a change in the sensitivity list of the first always block with an inclusion of c in it. For the sake of simplicity and ease of illustration, we have considered a simple Verilog code that can be simulated by a standard simulator like VCS [5]. Both the designs are simulated with a common testbench and outputs are recorded. Examining the simulation waveforms in Figure 4.2, we notice at clock cycle 2, the values of the register y ($y_{D'}$) produced by the evolved design D' and the register y (y_D) produced by the reference design D are different. The execution statement dump of the simulation traces of the two designs are shown in Figure 4.3. Since design D' is an evolved variant with an expectation to preserve the functionality of the design D, this is undesirable.

The change in the behavior of y occurs due to an additional execution of the code inside the first always block of design D' triggered by changes in signal c in the sensitivity list. As

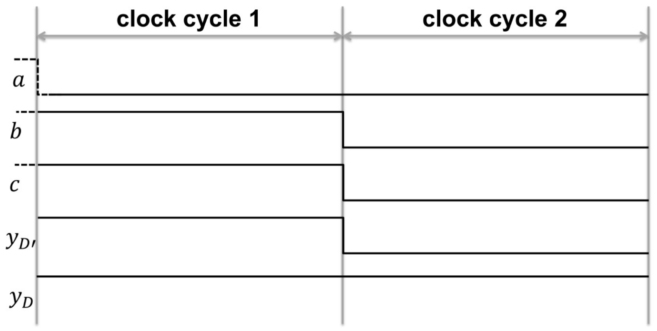


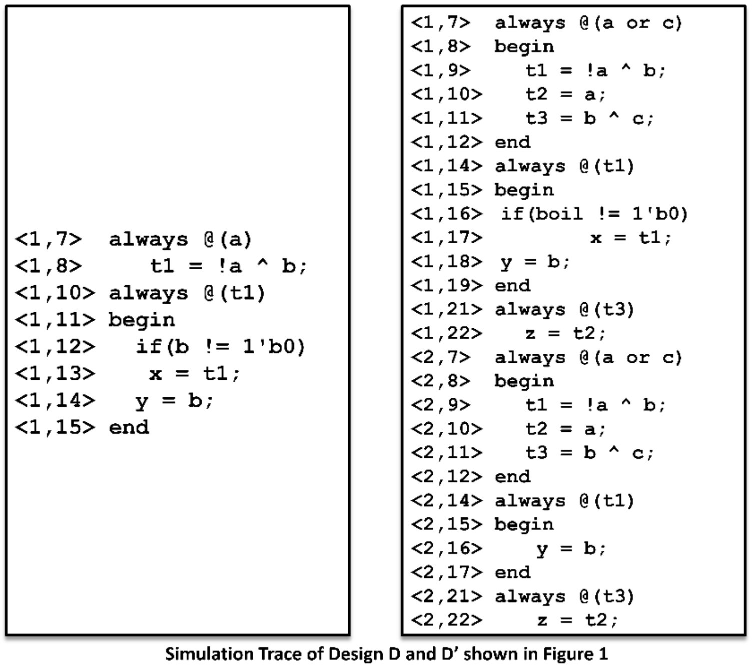
Figure 4.2: *Waveform of simulation output*

per the semantics of an always block, an execution is triggered whenever any signal in its sensitivity list changes. A change in a (from 1 to 0) as shown by dotted lines in Figure 4.2 triggers an execution of the first always block in both the designs in the first cycle. The value of $t1$, which was 1 earlier, is now changed to 0, triggering an execution of the second always block. As a result, the signal y is assigned a value 1 for both the designs. At clock cycle 2, the value of a does not change, and thus there are no executions of any of the always blocks in design D. However due to a change in c, the first always block in D' is triggered. This triggers an additional execution of the second always block, leading to the assignment of y to 0 (b is 0 in this clock) due to the blocking assignment. This explains the difference between y_D and $y_{D'}$.

We now discuss ways by which this bug can be localized. A simple comparison of the source files shows a number of lines as the difference, most of which do not have a role to play in the functionality of y and are related to z. The always block [lines 21 - 23] can be completely discarded since it has no role to play in the computation of y. The other two always blocks influence the computation of y in a direct / indirect way, however, some statements like 10, 11, 16 and 17 inside them are irrelevant. For a large design, distributed across multiple source files, it is difficult to isolate the conditions triggering the bug by pure textual difference analysis. Moreover, much of this effort is needless, since many of the lines appearing in the source difference may not have been executed at all.

A more effective approach would be to compare the execution statement dump of the simulation traces obtained for the two design variants. This will remove the statements which are not executed, thereby reducing the effort of the developer in understanding the bug. However, in this case as well, there may be too many statements in the trace dump that are completely unrelated to the bug and need not be examined.

To motivate our proposal, let us study the effect of the other statements on the bug in y. Looking back from line $< 2, 16 >$ in the execution trace of the design D' shown in Figure 4.3, we can see that this assignment is guarded by the always statement $< 2, 14 >$



```
<1,7>  always @(a)
<1,8>      t1 = !a ^ b;
<1,10> always @(t1)
<1,11> begin
<1,12>   if(b != 1'b0)
<1,13>     x = t1;
<1,14>   y = b;
<1,15> end
```

```
<1,7>  always @(a or c)
<1,8>  begin
<1,9>     t1 = !a ^ b;
<1,10>    t2 = a;
<1,11>    t3 = b ^ c;
<1,12> end
<1,14> always @(t1)
<1,15> begin
<1,16>  if(boil != 1'b0)
<1,17>          x = t1;
<1,18>  y = b;
<1,19> end
<1,21> always @(t3)
<1,22>    z = t2;
<2,7>  always @(a or c)
<2,8>  begin
<2,9>     t1 = !a ^ b;
<2,10>    t2 = a;
<2,11>    t3 = b ^ c;
<2,12> end
<2,14> always @(t1)
<2,15> begin
<2,16>     y = b;
<2,17> end
<2,21> always @(t3)
<2,22>     z = t2;
```

Simulation Trace of Design D and D' shown in Figure 1

Figure 4.3: *Execution Traces of the Designs from Figure 4.1*

triggered by $t1$. Changes in $t1$ that trigger this execution depend on the values of a and b which are primary inputs. This takes us to statements $< 1, 9 >$ and $< 2, 9 >$ which are guarded by an always block with a and c in the sensitivity list. This is evident in both the instantiations of the always block in the execution trace of design D'. Therefore it is quite obvious that the remaining statements do not contribute in the analysis of this bug. A similar construction on the reference design D with respect to the variable y at $< 1, 14 >$ yields the statements $< 1, 7 >, < 1, 8 >, < 1, 10 >$. A simple comparison of these statement sets can lead us to the bug. This method of statement filtering based on some condition is termed as *slicing* [42] in the software debugging literature.

We put forward the proposal of use of a functional program slice in the context of HDL designs for debugging. This is combined with a symbolic analysis of the slice with the construction of the weakest precondition that is easily amenable to automated analysis.

4.2 Detailed Methodology

Our methodology consists of four main steps.

- Bug scenario identification

- Backward dynamic time-domain slicing
- Weakest pre-condition Computation
- Source reverse mapping

In the following subsections, we explain the steps in detail.

4.2.1 Bug scenario identification

Both the reference design D and the evolved design D', are simulated with a common simulation input S, resulting in the execution traces λ and λ' respectively, as shown in Figure 4.3. Since HDL variables have different values across clock cycles, we represent the instance of each variable v at the clock cycle t_i by $< v, t_i >$. The first observable difference in the simulation behavior for the output variables which are expected to remain unchanged or an assertion violation, serve as our bug scenario.

Definition 4.1 *A bug scenario is a tuple $< v, t_s >$ where v is the value of interest and t_s is the clock instance where the first difference is observed or assertion is violated.* □

The value of interest can be an output signal with an incorrect value or the variables in the assertion which was violated. In our case, we have $< y, 2 >$ as the bug scenario, as can be seen from Figure 4.2.

4.2.2 Backward dynamic time-domain slicing

Backward dynamic time-domain slicing takes a simulation trace λ and a bug scenario $< v, t_s >$ as input and returns a subset of λ, called the *slice* with respect to $< v, t_s >$. The slice consists of the statement instances that influence the computation of v at clock cycle t_s, either through a direct / indirect assignment of value or through conditional statements guarding the statements that get executed. The former are called data dependencies and the latter control dependencies, which constitute the program slice.

To compute the slice, we proceed backwards along the trace from the clock cycle t_s, specified by the bug scenario. The slice computation algorithm is similar in spirit to the classical dynamic slicing paradigm as used in [17] with the additional task of propagating the computation along multiple cycles for each variable instance over time across multiple concurrent statements as outlined in [28]. We outline the philosophy below. The statements executed during simulation, beyond time t_s in the trace are ignored since they do not contribute to the error in simulation output which we are investigating. We initialize the *slice* to empty to start with. Therefore, initially we have the following: (i) dynamic slice *slice* set to empty (ii) set of variables θ whose dynamic data dependencies are unresolved, initialized to $< v, t_s >$ i.e. the bug scenario (iii) set of statement instances π whose dynamic control dependencies need to be resolved, initially set to empty. For each statement instance $< t, stmt >$ that we encounter while traversing the trace backwards, we can classify the statement as a data or control dependency, and proceed as follows.

Analyzing dynamic data dependencies: In Verilog, the semantics of execution of a data assignment statement varies depending on whether the assignment is blocking, non-blocking or a continuous assignment. The essential difference is when any change in the right hand side flows into the left hand side. In our framework, we handle all the variations unlike the proposal in [28]. Let v^t_{stmt} be the net or register assigned by $stmt$ in clock cycle t of simulation.

- For a blocking / assign assignment (denoted by = in Verilog), if the left hand side variable i.e. the one being assigned in $< t, stmt >$ is v, and $v^t_{stmt} \in \theta$, we have found the definition of v^t_{stmt}. v^t_{stmt} is removed from θ and for each of the variables w in the right hand side of $< t, stmt >$, w^t_{stmt} is added to θ.
- For a non-blocking assignment (denoted by <= in Verilog), if the left hand side variable i.e. the one being assigned in $< t, stmt >$ is v and $v^{t+1}_{stmt} \in \theta$, we have found the definition of v^{t+1}_{stmt}. Similarly, v^{t+1}_{stmt} is removed from θ and for each of the variables w in $< t, stmt >$, w^t_{stmt} is added to θ.

In both cases, $< t, stmt >$ is added to *slice* and π.

Analyzing dynamic control dependencies: All statement instances in π which are dynamically control dependent on $stmt$ at clock cycle t are removed from π. Additionally, for each variable v used by $stmt$, v^t_{stmt} is inserted into θ and $< t, stmt >$ is inserted into *slice* and π. HDL conditional constructs like if-else, while, case appear as control dependencies and are handled in this step. In addition, always statements also get examined as a conditional because of the sensitivity condition. The underlying execution semantics of an always statement demands a change in value of at least one of the members in the sensitivity list, which is interpreted as a conditional comparison between the instances of the variables.

The slice construction terminates when the start of the simulation trace is reached. The slice is reported as the final dynamic time-domain slice. Through this process, we are able to filter out variables and statements irrelevant to the bug scenario, since at each step we check for membership in θ and π, which are initialized to the bug scenario and empty respectively.

Example 4.1 *Consider our example in the previous section. As discussed, our bug scenario is set to $< y, 2 >$. Dynamic time-domain slicing on the simulation trace λ of D with the bug scenario $< y, 2 >$ yields the statements $\{< 1, 7 >, < 1, 8 >, < 1, 10 >, < 1, 14 >\}$,* *of which $< 1, 7 >$ and $< 1, 10 >$ are inserted as control dependencies and $< 1, 8 >$ as a data dependency. It is interesting to note that $< y, 2 >$ is not recorded in the trace for D (since we use a value change dump format) indicating the value of y is preserved from the previous clock cycle. Therefore, we find $< y, 1 >$ and since it exists, we proceed from there. A similar backward dynamic time-domain slice computation on the execution trace λ' for the evolved design D' for the slicing criteria $< y, 2 >$ yields the statement instances $\{< 1, 7 >, < 1, 9 >, < 2, 7 >, < 2, 9 >, < 2, 14 >, < 2, 16 >\}$.* □

It may be noted that the slice computation is quite different from the software context. As seen in the example above, the absence of a variable at a particular clock cycle is interpreted as the value being preserved from earlier clock cycles and therefore has to be considered carefully in this slicing step. There are more such differences in the HDL execution semantics that make the slice construction task different than the classical one.

4.2.3 Weakest pre-condition (WP) Computation

The weakest precondition is a symbolic representation of the dynamic slice that helps us in automating the comparison between the reference and the evolved versions. Intuitively this is a conjunction of the path condition obtained on the slice with respect to the bug scenario. In our approach, this is computed along with the dynamic time-domain slice computation. Hence, WP-computation finishes as soon as the slice computation terminates. WP computation needs us to set a post condition p with respect to which the WP is required to be computed. The post condition is the bug scenario $< v, t_s >$. Proceeding backwards along the trace λ from clock cycle t_s specified by the post condition, in a manner similar to dynamic time-domain slicing, we can encounter either data dependency statements or control dependency statements. We use the following data structures (i) α, to store the current WP, initialized to the post condition p (ii) *map*, a set of three tuples of the form $(stmt, t, lineno)$, initially set to empty, where *stmt* is a statement executed at simulation cycle t and occurs on *lineno* in the source code of the design. For each statement *stmt* encountered during computation of backward dynamic time-domain slicing, WP is updated as follows:

Analyzing Data Dependency (Assignment statements):

- For each blocking assignment / assign statement of the form $v = rhs$ executed at clock cycle t, we replace all occurrences of $< v, t' >$ for all $t' \geq t$, from WP with rhs, after annotating each variable w in rhs with the clock cycle t.
- For nonblocking assignments of the form $v <= rhs$ executed at clock cycle t, we replace all occurrences of $< v, t' >$ for all $t' \geq t + 1$, from WP with rhs, after annotating each variable w in rhs with the clock cycle t.

In both the cases, v is a net or register and rhs is a valid HDL expression. We substitute in this manner, since a net or register may be assigned a value, which is used in a later clock cycle t', with the value actually having been assigned in an earlier clock cycle t.

Analyzing Control Dependency (Branch statements): For each conditional statement with condition C, we conjoin C with the current WP α to update the WP i.e. $\alpha = \alpha \wedge C$. For an always statement, the conjunct is a disjunction of terms of the form $< v, t > \oplus < v, t-1 >$, where v is an element of the sensitivity list of the always statement.

In both the cases, the statement *stmt*, the simulation clock cycle t and the line number of the *stmt* is stored in *map*. The algorithm terminates on reaching the start of the trace. The WP present in C is the final WP. It should be noted that only those statements that are part of the backward time-domain slice are considered during WP computation. This computation is performed on both the designs D and D'. We explain this construction on our example below.

Example 4.2 *The WP α for trace λ is computed as follows. The conditional contributed by the guarding always triggered by $t1$ is added in WP as $(< t1, 0 > \neq < t1, 1 >)$ which in the next backward step elaborates as $(< t1, 0 > \neq (\neg < a, 1 > \oplus < b, 1 >)$. In this case, as noted earlier, $< y, 2 >$ is absent and we start with $< y, 1 >$. Similarly we have the term $(< a, 0 > \neq < a, 1 >)$ from the other always block. Thus we have the WP condition for α as $(< a, 0 > \neq < a, 1 >) \wedge (< t1, 0 > \neq (\neg < a, 1 > \oplus < b, 1 >))$ The corresponding WP α_1 for the post-condition $(< y, 2 >== 1)$ is $((< a, 0 > \neq < a, 1 >) \vee (< c, 0 > \neq < c, 1 >)) \wedge (< t1, 0 > \neq (\neg < a, 1 > \oplus < b, 1 >)) \wedge ((< a, 1 > \neq < a, 2 >) \vee (< c, 1 > \neq < c, 2 >)) \wedge ((\neg < a, 1 > \oplus < b, 1 >) \neq (\neg < a, 2 > \oplus < b, 2 >))\square$*

4.2.4 Source reverse mapping

Computing the weakest pre-condition of the reference design D and the evolved design D' by analyzing their trace λ and λ' respectively, we obtain the weakest preconditions α and α' respectively which are basically a conjunction of constraints in the following form:

$$\alpha = \phi_1 \wedge \phi_2 ... \wedge \phi_m$$

$$\alpha' = \phi'_1 \wedge \phi'_2 ... \wedge \phi'_m$$

Our objective is to find a constraint ϕ_i (or symmetrically ϕ'_j) such that $\alpha' \not\Rightarrow \phi_i$ (symmetrically $\alpha \not\Rightarrow \phi_j$). Such constraints are the reason for the difference in functionality between D and D'. Once we have found such unimplied constraints, we map back to the source code with the help of *map* that we had defined earlier in the WP-computation step. We find that the unimplied constraints are $((< a, 0 > \neq < a, 1 >) \vee (< c, 0 > \neq < c, 1 >))$, $((< a, 1 > \neq < a, 2 >) \vee (< c, 1 > \neq < c, 2 >))$, and $((\neg < a, 1 > \oplus < b, 1 >) \neq (\neg < a, 2 > \oplus < b, 2 >))$ which map to statement instances $< 1, 7 >, < 2, 7 >$ and $< 2, 14 >$ of design D' respectively.

These statements are reported in the bug report. We can notice that the additional signal c in the sensitivity list in line number 7 of design D' is the reason behind the always block getting executed for the second time in clock cycle 2 in design D', leading to register y being incorrectly set in clock cycle 2. Thus we are able to pinpoint the bug.

4.3 Conclusion

We believe the proposed methodology is expected to serve a pivotal role in debugging in evolving hardware designs. The methodology has been implemented as a framework and the performance has been evaluated on open source designs, as presented in Subsection 5.2.

Chapter 5

Implementation and Results

5.1 Implementation and performance of EAST

The proposed LTL simulation architecture [Chapter 3] is shown in Figure 5.1. LTL assertions in Negated Normal Form act as input to the preprocessing stage. The preprocessing stage parses individual assertions to check for syntactical validity and converts them into postfix form. The assertions in postfix form are passed on to the LUT Generator which generates the LUT representation of the assertions. The preprocessing stage has been im-

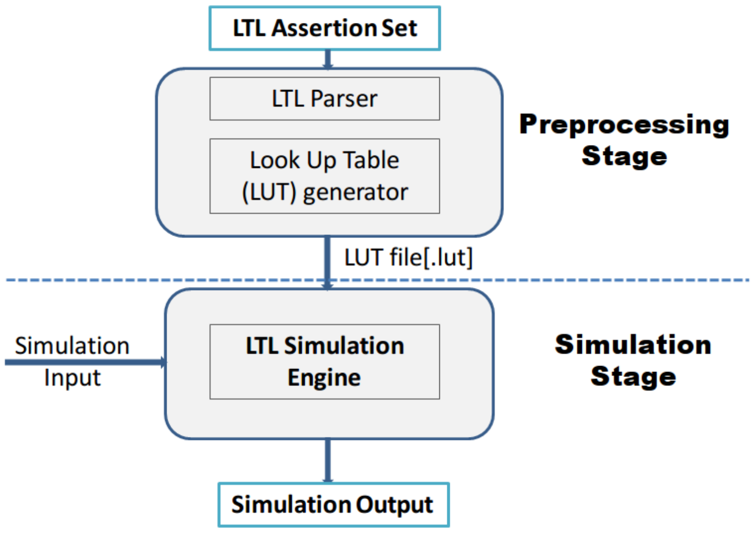


Figure 5.1: *Proposed LTL Simulation Architecture*

plemented using Python. For simulating the same assertion set, the preprocessing stage needs to be executed only once for generating the LUT. The LTL simulation engine, implemented in Java, takes the LUT generated from the preprocessing stage as input along with the simulation inputs. The simulation inputs for a given clock cycle are input as a hash table with the input variable name as key. For each clock cycle, the simulation engine needs to be provided with the corresponding simulation input. The simulation engine simulates the assertion using level based assertion evaluation algorithm stated in Subsection 3.4.

We put to test our proposed Level Based Simulator (LBS) with an in-house LTL simulator that implements transaction-based monitoring (TBM), with automata-based monitor using randomly generated simulations: simulation inputs, assertions and number of simulation cycles. The performances, shown in Table 5.1 demonstrate the considerable gains of LBS over TBM. Experimental setup was : Intel Core 2 Duo P8700 (2.53 GHz), 3 GB RAM, Ubuntu. LBS was also evaluated using the Open Cores Protocol (OCP) assertion suite [3].

Exp	**#Assertions**	**#signals**	**#cycles**	**LBS(secs)**	**TBM(secs)**
1	31	4	753	0.083	38.847
2	26	6	899	0.125	114.687
3	33	6	1159	0.154	148.99
4	15	18	1690	0.181	177.29

Table 5.1: Performance comparison of LBS vs TBM

The assertions manually written in LTL NNF, consist of 45 assertions and 46 signals. The results of simulation using random inputs are shown in Table 5.2 and visualized in Figure 5.2.

No of signals	46
No of assertions	45
Look up Table Generation Time	0.455s
No of cycles simulated	**Simulation time (in secs)**
100	0.0442
1000	0.1032
10000	0.5652
100000	5.1282

Table 5.2: LBS performance for OCP assertions

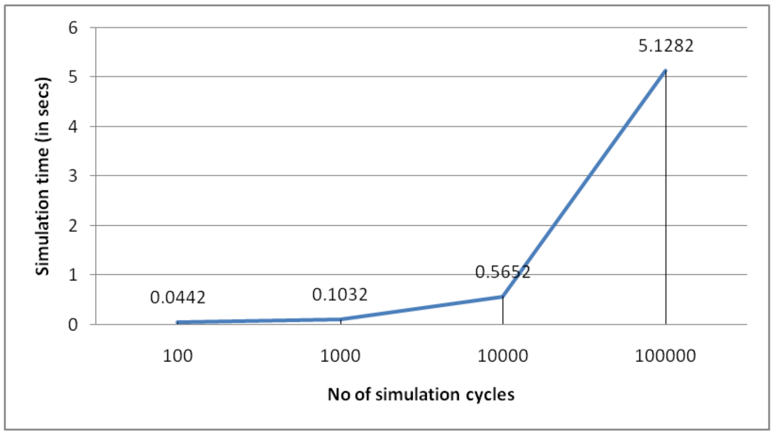


Figure 5.2: *LBS performance for OCP assertions*

5.2 Implementation and Results of EvoDeb

The entire work-flow of EvoDeb is depicted in Figure 5.3. To obtain the statement dump, we insert Verilog Procedural Interface (VPI) call statements in the original source code of the reference design D and evolved design D' to get the statement level execution dump, without affecting the functionality, using a Verilog parser. We record the simulation traces λ and λ' by simulation with a common test bench using the VCS [5] simulator. The algorithms for slicing and WP computation, comparison and source reverse mapping were implemented by us in Python. We used YICES2 [38] for constraint implication checking as needed in the final step of our approach. Our framework translates the WPs originally in HDL to YICES2's specification language and then proceeds to determine the unimplied constraints followed by source remapping. We put to test our framework for locating change induced bugs on two open-source designs using our in-house random test generator.

5.2.1 Experience with UART16550

UART16550 [13] defines the UART core WISHBONE interface. We used this design as the reference design. We modified the condition of an if statement (line no 203 in the original source file) *if* (*wb_stb_is wb_cyc_is*) to *if* (*wb_stb_is*) to test our framework as shown in Figure 5.4. We simulated the designs for 10000 clock cycles. Even though the designs differ by a single line, it is worth mentioning that simulation trace difference of the two designs for the same testbench, has 29279 lines, which shows how much a single change can affect the design. Due to the change in the design, the first difference in the output that we observe is the value of *wb_ack_o* in clock cycle 7751, and hence we chose $< wb\_ack\_o, 7751 >$ as the bug scenario. The WP consists of around 2400 constraints for each of the designs. Our framework determined the unexplained constraints correctly and returned the line number of the first statement in the source code that caused the bug in the changed design. Our

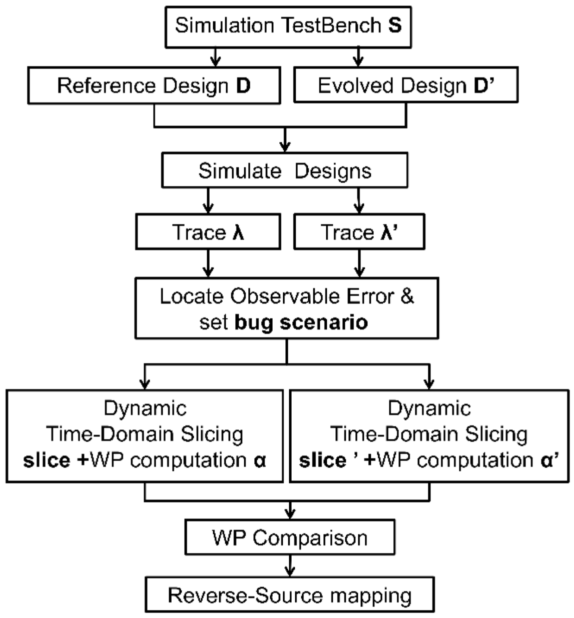
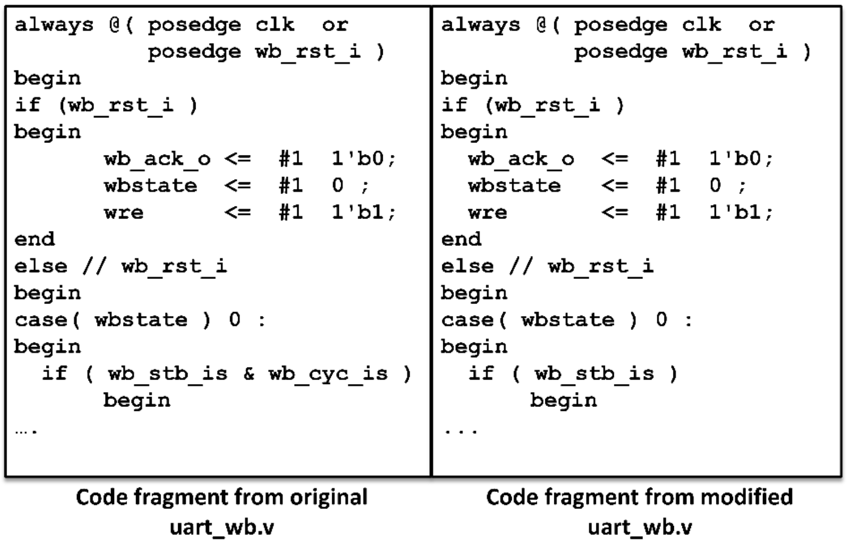

Simulation TestBench **S**
Reference Design **D**
Evolved Design **D'**
Simulate Designs
Trace **λ**
Trace **λ'**
Locate Observable Error & set **bug scenario**
Dynamic Time-Domain Slicing **slice +**WP computation **α**
Dynamic Time-Domain Slicing **slice ' +**WP computation **α'**
WP Comparison
Reverse-Source mapping

Figure 5.3: *Proposed Framework of EvoDeb*

framework returned the line corresponding to the else part of the if condition that we had changed in the new design. The runtime for our debug step was a few seconds.

5.2.2 Experience with PCI bridge

We used the *pci_wb_slave.v* as our second test case. This is part of the PCI bridge project [10]. We modified two sensitivity lists [line number 587 and 651 of the original file] to create a buggy version and considered the original file as the reference design. By reducing the number of signals in the two sensitivity lists, we created a code missing bug scenario. The modified sensitivity lists are shown in Figure 5.5. The traces obtained were of 183319 and 91091 lines for the original and the modified design respectively. The first difference in the output of the two designs that we observe is the value of *sample_address_out* to differ at clock cycle 2970. Thereby we consider $< sample\_address\_out, 2970 >$ as our bug scenario. The WPs were around 80000 constraints in each design. In this case as well, our framework found the presence of unexplained constraints and reported line 587 and 651 as bugs successfully, in addition to other lines that were executed as a result of the always blocks in the original design getting triggered a greater number of times as compared to the one in the modified design, due to more signals present in the sensitivity list.



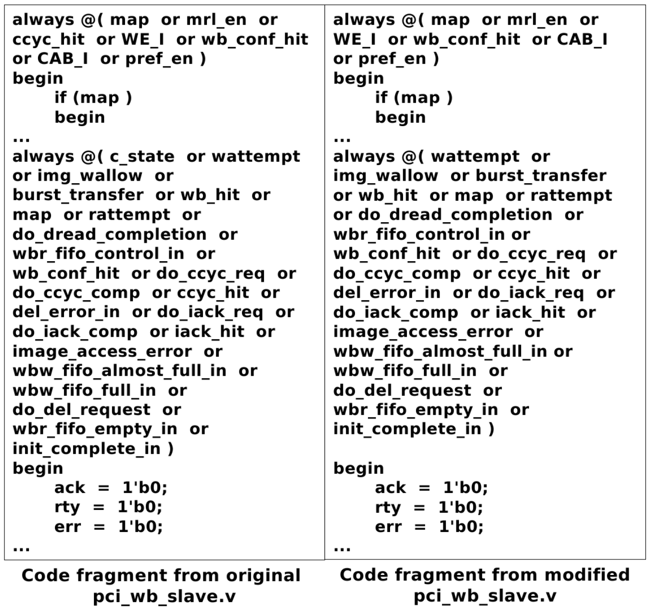
```
always @( posedge clk   or
          posedge wb_rst_i )
begin
if (wb_rst_i )
begin
       wb_ack_o <=  #1  1'b0;
       wbstate  <=  #1  0 ;
       wre      <=  #1  1'b1;
end
else // wb_rst_i
begin
case( wbstate ) 0 :
begin
  if ( wb_stb_is & wb_cyc_is )
       begin
….
```

Code fragment from original uart_wb.v

```
always @( posedge clk   or
          posedge wb_rst_i )
begin
if (wb_rst_i )
begin
  wb_ack_o  <=  #1  1'b0;
  wbstate   <=  #1  0 ;
  wre       <=  #1  1'b1;
end
else // wb_rst_i
begin
case( wbstate ) 0 :
begin
  if ( wb_stb_is )
       begin
...
```

Code fragment from modified uart_wb.v

Figure 5.4: *Source code fragment of uart_wb.v*



```
always @( map  or mrl_en  or
ccyc_hit  or WE_I  or wb_conf_hit
or CAB_I  or pref_en )
begin
     if (map )
     begin
...
always @( c_state  or wattempt
or img_wallow  or
burst_transfer  or wb_hit  or
map  or rattempt  or
do_dread_completion  or
wbr_fifo_control_in  or
wb_conf_hit  or do_ccyc_req  or
do_ccyc_comp  or ccyc_hit  or
del_error_in  or do_iack_req  or
do_iack_comp  or iack_hit  or
image_access_error  or
wbw_fifo_almost_full_in  or
wbw_fifo_full_in  or
do_del_request  or
wbr_fifo_empty_in  or
init_complete_in )
begin
     ack  =  1'b0;
     rty  =  1'b0;
     err  =  1'b0;
...
```

Code fragment from original pci_wb_slave.v

```
always @( map  or mrl_en  or
WE_I  or wb_conf_hit  or CAB_I
or pref_en )
begin
     if (map )
     begin
...
always @( wattempt  or
img_wallow  or burst_transfer
or wb_hit  or map  or rattempt
or do_dread_completion  or
wbr_fifo_control_in or
wb_conf_hit  or do_ccyc_req  or
do_ccyc_comp  or ccyc_hit  or
del_error_in  or do_iack_req  or
do_iack_comp  or iack_hit  or
image_access_error  or
wbw_fifo_almost_full_in or
wbw_fifo_full_in  or
do_del_request  or
wbr_fifo_empty_in  or
init_complete_in )

begin
     ack  =  1'b0;
     rty  =  1'b0;
     err  =  1'b0;
...
```

Code fragment from modified pci_wb_slave.v

Figure 5.5: *Source code of pci_wb_slave.v*

Chapter 6

Conclusion and Future Work

In this work, we presented an efficient methodology for simulation of LTL assertions. The foundation of the work is based on inferencing the evaluation results of the assertions with actual evaluation of the assertions by using a shared data structure. The methodology has been demonstrated to offer better performance in simulation of LTL assertions than transaction- based monitoring (TBM), with automata-based monitor on random assertions as well as on a standard benchmark assertion suite. The developed framework can be easily integrated into existing standard simulation tool flows.

The work on debugging evolving designs, EvoDeb presents a novel and efficient automated methodology for debugging change induced bugs. The application of classical program analysis in the context of HDL programs makes our proposal useful and attractive in practice. Results indicate the efficiency of our approach.

In our future work, we are interested in extending our research towards automated assertion generation for evolving designs. For an existing reference design, there usually exists a set of assertions to validate the functionality of the given design, along with a test suite. We aim at extending our approach presented in EvoDeb such that we could obtain an assertion suite for the evolved design from the reference assertion suite that would validate the evolved design. We plan to use the differences in source code of the designs to mutate the reference assertion suite to obtain the new assertion suite for the evolved design. We believe that our methods can be extended to reduce efforts in designing assertion suites for evolved designs with minimal designer intervention. .

Chapter 7

Disseminations out of this work

- D. Bhattacharjee, A. Banerjee, and A. Chattopadhyay, “EvoDeb: Debugging Evolving Hardware Designs”, in *VLSI Design (VLSID), 2015 28th International Conference on*, pp. 481-486. IEEE, 2015.
- D. Bhattacharjee, S. Chattopadhyay, and A. Banerjee, “EAST: Efficient Assertion Simulation Techniques”, under review *24th IEEE Asian Test Symposium*, IEEE, 2015.

Bibliography

[1] Boolean satisfiability problem. `http://en.wikipedia.org/wiki/Boolean_satisfiability_problem` .

[2] Negation Normal Form. `en.wikipedia.org/wiki/Linear_temporal_logic#Negation_normal_form` .

[3] Open Core Protocol. accellera.org/downloads/standards/ocp/.

[4] Questa Advanced Simulator. `www.mentor.com/products/fv/questa/`.

[5] VCS. `www.synopsys.com/tools/verification/functionalverification/\pages/vcs.aspx` .

[6] Ara, K., and Suzuki, K. A proposal for transaction-level verification with component wrapper language. In *Proceedings of the conference on Design, Automation and Test in Europe: Designers' Forum-Volume 2* (2003), IEEE Computer Society, p. 20082.

[7] Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., et al. The forspec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2002, pp. 296–311.

[8] Armoni, R., Korchemny, D., Tiemeyer, A., Vardi, M. Y., and Zbar, Y. Deterministic dynamic monitors for linear-time assertions. In *Formal Approaches to Software Testing and Runtime Verification*. Springer, 2006, pp. 163–177.

[9] Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., and Rodeh, Y. The temporal logic sugar. In *Computer Aided Verification* (2001), Springer, pp. 363–367.

[10] bridge, P. http://opencores.org/project,pci.

[11] Chang, K.-H., Tu, W.-T., Yeh, Y.-J., Kuo, S.-Y., et al. A simulation-based temporal assertion checker for psl. In *Proceedings of IEEE International Symposium on Micro-NanoMechatronics and Human Science* (2003), Citeseer, pp. 1528–1531.

[12] ClioSoft. Available at. `http://www.cliosoft.com/products/`.

[13] core, U. . http://opencores.org/project,uart16550.

[14] Daga, A. J., and Birmingham, W. P. A symbolic-simulation approach to the timing verification of interacting fsms. In *2012 IEEE 30th International Conference on Computer Design (ICCD)* (1995), IEEE Computer Society, pp. 584–584.

[15] Dasgupta, P. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[16] Dehbashi, M., and Fey, G. Debug automation for synchronization bugs at rtl. In *VLSI Design, 27th International Conference on* (Jan 2014), pp. 44–49.

[17] et al., A. B. *Golden implementation driven software debugging.* FSE, 2010.

[18] et al., B. P. Automated source level error localization in hardware designs. *IEEE Design & Test of Computers 23*, 1 (2006), 8–19.

[19] et al., B. P. Employing test suites for verilog fault localization. *Lecture Notes in Computer Science 5988* (2010), 1–10.

[20] et al., D. Q. *DARWIN: An approach for debugging evolving programs.* ESEC-FSE, 2009.

[21] et al., E. M. C. Program slicing for vhdl. *International Journal on Software Tools for Technology Transfer (STTT) 4*, 1 (2001), 125–137.

[22] et al., F. A. Post-verification debugging of hierarchical designs. In *MTV* (Nov 2005), pp. 42–47.

[23] et al., G. F. Towards unifying localization and explanation for automated debugging. In *MTV* (Dec 2010), pp. 3–8.

[24] et al., H.-T. L. Efficient automatic diagnosis of digital circuits. In *ICCAD* (1990), pp. 464–467.

[25] et al., J. C. M. Automating the diagnosis and rectification of the design errors with priam. In *Proc. of ICCAD* (1989), pp. 30–33.

[26] et al., K. K. Automated diagnosis of vlsi digital circuits using algorithmic debugging. In *Proc. of the 1st Intl. Workshop on Automated and Algorithmic Debugging* (1993), pp. 350–367.

[27] et al., R. S. *Test-suite augmentation for evolving software.* ASE, 2008.

[28] et al., S. M. Formal guarantees for localized bug fixes. *Proc. of IEEE TCAD 32*, 8 (2013), 1274–1287.

[29] et al., S.-Y. H. Fault-simulation based design error diagnosis for sequential circuits. In *Proc. of DAC* (1998), pp. 632–637.

[30] et al., V. P. A system for fault diagnosis and simulation of vhdl descriptions. In *Proc. of DAC* (1991), pp. 144–150.

[31] et al., V. P. A vhdl fault diagnosis tool using functional fault models. *IEEE Test and Design of Computers 9*, 2 (1992).

[32] Kuo, S.-Y. Locating logic design errors via test generation and don't-care propagation. In *Proc. of Euro-DAC* (1992), pp. 466–471.

[33] Lamport, L. "sometime" is sometimes "not never": On the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1980), pp. 174–185.

[34] Miller, B. N., and Ranum, D. L. *Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION*. Franklin, Beedle & Associates Inc., 2011.

[35] Pnueli, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science, Washington, DC, USA: IEEE Computer Society* (1977), pp. 46–57.

[36] Pnueli, A. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on* (1977), IEEE, pp. 46–57.

[37] Pomeranz, I., and Reddy, S. M. A method for diagnosing implementation errors in synchronous sequential circuits and its implications on synthesis. In *Proc. of Euro-DAC* (1993), pp. 252–258.

[38] Solver, T. Y. S. http://yices.csl.sri.com/.

[39] Solvertec. Solvertec. `http://www.solvertec.de/`, 2014.

[40] System, V. A. D. Available at:. www.synopsys.com/Tools/Verification/debug/Pages/Verdi-ds.aspx.

[41] Tabakov, D., Rozier, K. Y., and Vardi, M. Y. Optimized temporal monitors for systemc. *Formal Methods in System Design 41*, 3 (2012), 236–268.

[42] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[43] Verifyter. Verifyter. `http://www.verifyter.com/`, 2014.

[44] Vijayaraghavan, S., and Ramanathan, M. *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2006.

[45] Wahba, A. M., and Borrione, D. A method for automatic design error location and correction in combinational logic circuits. *J. Electron. Test. 8*, 2 (1996), 113–127.