Formal Methods For Design Debugging From Traces

by

Anvesh Komuravelli
(Roll No.: 05CS1031)

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology (*Hons.*)
in
Computer Science and Engineering

under the guidance of
Dr. Pallab Dasgupta

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India - 721302
May 2009

This document was typeset using LaTeX 2_ε.

Certificate

This is to certify that the thesis titled **Formal Methods For Design Debugging From Traces** submitted by **Anvesh Komuravelli** to the Department of Computer Science and Engineering in partial fulfillment for the award of the degree of **Bachelor of Technology (Hons.)** is a bonafide record of work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this Institute, the Indian Institute of Technology Kharagpur, and in our opinion, has reached the standard needed for submission.

Dr. Pallab Dasgupta
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India - 721302
May 2009

To my parents,
To the eternal mystery of the universe,
To Science,
To the humankind

Acknowledgements

I am greatly thankful to my advisor and Professor Pallab Dasgupta for firstly letting me work with the Formal-V group at Kharagpur. I thank him for all the time he spared listening to my ideas and for all the motivation he gave, even out of his busy schedule. Finally, I thank him for all the patience I learnt from him. I am really grateful to have met such a fantastic person.

Then, I would like to thank the Formal-V group and especially Srobona Mitra and Priyankar Ghosh for their initial inputs for my project and to Subrat Kumar Panda for all the fun time and fun talk we had and for always being ready to help me out be it with a SAT solver or be it with a problem in my code! I would like to thank Rajdeep Mukhopadhyay for letting me work with him on an earlier project on Verification which initiated my interests in the area. I also thank Aritra Hazra for being always ready for any help!

Finally, I thank my parents and brother for all their inspiration and motivation throughout.

Anvesh Komuravelli
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India - 721302
May 2009

Abstract

Verification of large integrated SOCs is still a big challenge. Simulation based Pre-Silicon verification is insufficient. Pre-Silicon formal verification is not scalable. We propose two new and contrasting formal methods for this problem which make use of the huge Post-Silicon traces to help in design debugging. The first method makes use of the trace along with the design while the second method makes use of the trace along with a property suite where the properties can also be on the invisible internal signals. The objective of the first method is to obtain a shorter bug trace which is feasible to be simulated on the design. The proposed approach shows significant improvement over the obvious reachability algorithm for small circuits while it faces the scalability problem for large circuits. The objective of the second method is to see if the trace refutes the property suite. The proposed approach is new of its kind and is shown to be very much scalable and feasible in industrial practice. It is also shown to report a refutation whenever one exists.

Contents

1 Introduction **1**

1.1 How about Post-Silicon Verification? 2

1.2 The Problem and the Issues Involved 3

1.3 Our Contribution 3

1.3.1 Debugging using the Trace and the Design 4

1.3.2 Debugging using the Trace and the Properties 5

1.4 Thesis Organization 5

2 Background and Related Work **7**

2.1 Shortening the trace 7

2.1.1 Related Work 7

2.1.2 SAT and BDD 8

2.2 Reasoning a property refutation 9

2.2.1 LTL 9

2.2.2 Types of Properties 10

2.2.3 Related Work 10

3 Design Assisted Debugging Approach **11**

3.1 The Approach 12

3.1.1 Time Frame Expansion 12

3.1.2 Bidirectional Search 13

3.2 Results 16

3.3 Conclusion 20

4 Property Assisted Debugging Approach **21**
4.1 A Motivating Example 23
4.2 The Sniffer Dog Approach 25
4.2.1 The backward debug algorithm 26
4.2.2 Algorithm Analysis 31
4.3 Experimental Results 33
4.3.1 Demonstration using AHB 33
4.3.2 Experiments with arbitrary safety properties 36
4.4 Conclusion 45

5 Conclusions and Future Work **48**
5.1 Design Assisted Debug 48
5.2 Property Assisted Debug 49
5.3 Combining both approaches 49

Bibliography **50**

List of Figures

1.1 A typical top-down design flow of a digital system. It begins with Specifications and ends after Layout and taping out of the chip. . . . 2

3.1 The combinational block of a sequential circuit. I, O, PS and NS stand for *inputs*, *outputs*, *present state* and *next state* respectively. . . 13

3.2 The time frame expansion where NS of previous (ith) frame are connected with PS of the next ($i+1$ th) frame. The inputs and outputs of each frame are separate. 13

3.3 (a) The forward and backward cones without using trace. A path obtained from S_{init} to S_{final} may be not quite short as it may go through unnecessary states. (b) The backward cone narrowed down by the trace and the path from S_{init} to S_{final} is more focused to the goal, going along the trace. 14

3.4 1-binding between the frames expanded in forward search and backward search. The PS of the first frame of the backward search away from the one corresponding to the final state can be matched with NS of any of the forward frames. 16

4.1 A module having an input a and an output b with c, d, e and f as internal signals and a trace for 20 cycles with the value of a followed by the value of b. 24

4.2 An architecture of two masters, M_0 and M_1, and two slaves, S_0 and S_1, using an $AMBA^{\text{TM}} AHB$ compliant bus. A is the arbiter. 34

4.3 The graph of inter-relations of three properties whose annotated parse trees, after neglecting the unary temporal operators, are also shown inside the circles representing the vertices. 42

List of Tables

3.1 Experimental results for pure forward search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. *tl* stands for *trace length*, *t* for *time needed for the search, in seconds* and *l* for *the length of the trace found*. *Linear* and *Exponential* denote the mode of addition of time frames. We use a timeout of 1000 seconds and *l* in that case is the length explored till then. 17

3.2 Experimental results for 1-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. *w* denotes *with trace* where the backward search uses the given trace and *wo* denotes *without trace*. *tl* stands for *trace length*, *t* for *time needed for the search, in seconds* and *l* for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and *l* in that case is the length explored till then. 18

3.3 Experimental results for 10-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. *w* denotes *with trace* where the backward search uses the given trace and *wo* denotes *without trace*. *tl* stands for *trace length*, *t* for *time needed for the search, in seconds* and *l* for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and *l* in that case is the length explored till then. 18

3.4 Experimental results for $b/2$-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. w denotes *with trace* where the backward search uses the given trace and wo denotes *without trace*. tl stands for *trace length*, t for *time needed for the search, in seconds* and l for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and l in that case is the length explored till then. 19

3.5 Experimental results for b-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. w denotes *with trace* where the backward search uses the given trace and wo denotes *without trace*. tl stands for *trace length*, t for *time needed for the search, in seconds* and l for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and l in that case is the length explored till then. 19

4.1 The experimental results of *Sniffer Dog* approach for various parameters. For each set of parameters, 10 properties are randomly generated. vis stands for *visible signals*, tot stands for *all signals*, l stands for *length of the counter-example trace*, t stands for *running time in seconds*, v stands for *SAT variables per time step*, c stands for *SAT clauses per time step*. The number in the brackets in each column header indicates the number of steps after which a new satisfiability check is made. . . 39

4.2 The values of various parameters of some property sets used in Table 4.1. As in that table, *vis* stands for *visible signals* and *tot* for *all signals*. l stands for *length of trace*. The column heads correspond to the value of l of the order of 10, 100 and 1000 respectively. c stands for *components* in the undirected graph corresponding to the digraph of the property set, p for *properties* in any *component* having at least one *visible* signal, e for *edges in that component*, v for *visible signals in that component* and n for *properties on a longest path in that component*. Entries indicated by '-' show that there is no component with any *visible* signal appearing in any property. All comma separated entries in a row are corresponding. 44

4.3 The experimental results of *Sniffer Dog* approach for various parameters for sets of 50 properties built upon those of Table 4.1. r stands for the *ratio* of the *visible* and *total* number of signals corresponding to that of Table 4.1 with $\#tot$ (in that table) multiplied by 5. l stands for *length of the counter-example trace*, t stands for *running time in seconds*, v stands for *SAT variables per time step*, c stands for *SAT clauses per time step*. The number in the brackets in each column header indicates the number of steps after which a new satisfiability check is made. 47

Chapter 1

Introduction

Digital systems are becoming increasingly complex. Validating the design in large integrated SOCs is now a highly non-trivial task. From a high architectural level, an integrated SOC consists of several design blocks interconnected among themselves. While validating those individual blocks using currently available verification techniques may be feasible, validating the integrated system is not practical using the same techniques due to the its huge size. Currently, design verification is done in two broad ways - *Formal Verification* and *Simulation Based Verification*. While Formal Verification aims to verify all possible behaviors of the input system, Simulation Based Verification only verifies those behaviors which are driven by the test bench. On one hand, running a formal verification technique on a huge integrated system is highly infeasible. On the other hand, simulating such a huge system takes exorbitant time and is again not practical. Thus, the current verification techniques can only verify *some* behaviors based on limited amount of simulation. But this is not at all sufficient. Thus there is a dire need for innovation in verifying such huge systems. This thesis is a small attempt towards that.

A typical design flow of a digital system is given in Fig. 1.1. Specification of the system to be built is at the top level. Going down the flow, more and more details are added and specifications are refined accordingly. Verification is required at every level to check the consistency of the specifications at that level and also to check the consistency with the previous levels. RTL is the most detailed design with levels further down corresponding to the building of the actual physical system. Thus, Design Verification is concerned with the levels before and including RTL. If this verification is done in the sequence specified by the design flow (after every level), it is also called Pre-Silicon Verification as we only verify the design based on the high

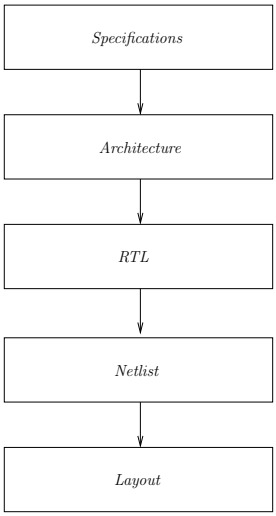


Figure 1.1: A typical top-down design flow of a digital system. It begins with Specifications and ends after Layout and taping out of the chip.

level RTL available. As mentioned in the previous paragraph, Pre-Silicon Verification techniques are inefficient for huge systems.

1.1 How about Post-Silicon Verification?

Typically, Post-Silicon Verification (also known as Post-Silicon Validation) is the last step in the design. Once the chip is taped out, it is run on several inputs and the outputs are compared with some golden behavior and after running sufficient tests, the chip is validated. We ask the question - Can we use Post-Silicon Verification for Design Debugging? This has the following advantages.

1. It complements the lack of coverage in Pre-Silicon Simulation Based Verification. Now that we can run the chip at-speed which is orders of magnitude faster than simulation of the RTL, we can take huge traces in considerably smaller time which was not possible with RTL.

2. It is possible to cover more behaviors using constrained random testing which may help uncover a bug which we couldn't using only RTL.

Note that the bugs we find Post-Silicon also include fabrication defects and others specific to the layout and the actual synthesis. This can make the verification challenging as we have to now differentiate between logical bugs and bugs arising out of

synthesis defects. In this thesis, we are only concerned with logical bugs which are also the actual defects in the design.

1.2 The Problem and the Issues Involved

We are given a huge Post-Silicon trace which is just a very long sequence of input-output pairs. It is given that this trace has an undesired behavior and thus serves as a counter-example trace for *some* unknown defect in the design. (It may as well be a counter-example for some physical defect but as said earlier, we only concentrate on logical defects. If no logical defect is found, then it may be further investigated for any physical defects.) The problem is to assist in locating such a defect in the design. We assume that the Post-Silicon execution stops as soon as some thing unexpected is reported in the trace. This can be best achieved if we have some golden sequence of input-output pairs corresponding to the given trace, where the execution stops just after some pair mismatch is found. But this is not always possible. Alternative ways like having a golden *behavior* of the trace can be used in reporting something unexpected.

Before moving forward, we look into some issues involved in Post-Silicon Verification. Firstly, the counter-example trace is very huge. This is not immediately useful as we cannot simulate the entire trace on the RTL. We need techniques to innovatively use this trace for design debug. Secondly, it has very limited *controllability*. The only control points available are the input pins which correspond to a tiny fraction of all the lines in the chip. Thirdly, it has very limited *observability*. The only lines we can observe are the output pins. We can't simply probe some internal lines to obtain their values and reason what went wrong. Thus, if we find that a trace has a buggy behavior the defect is not immediately apparent. Debugging the design from the trace is a non-trivial task.

1.3 Our Contribution

Based on the previous discussion, it makes sense to say that the undesired behavior in the counter-example trace is somewhere near the end of the trace. And it is more likely to be some peculiar behavior which was not caught in the Pre-Silicon Verification. One important inference that can be made at this point is that debugging the design

might be greatly eased if we can find a way to make a good use of the tail portion of the trace.

We describe two contrasting approaches for Post-Silicon Verification. In the first approach, we use the trace and the RTL design to obtain a considerably shorter trace which can be simulated to assist in design debug. In the second approach, we use the trace and the LTL properties which the design should ideally satisfy and try to find if the property suite is refuted on the trace which will again help in design debug as we can now create similar scenarios using the RTL and do a simulation.

1.3.1 Debugging using the Trace and the Design

Let us see what the state sequence corresponding to the trace looks like. We can't expect to get the complete state sequence (even if the system is tailored to have scan chains among the flip-flops) as that slows down the execution drastically. Thus, we only assume we have the initial and final states. As it is a counter-example trace, some state in this state sequence should have been unreachable because it is not a valid state (or fails some specification). This need not be the final state, but we expect it to be somewhere near the end of the sequence. Thus, if we can find a considerably shorter trace from the initial to the final state, using the tail portion of the given huge trace, we hope to simulate this behavior on the RTL using the shorter trace assisting in the design debug. As this simulation is now on RTL, we have the full observability and controllability of the system, and is lot better than simulating the original trace blindly as we now have a much shorter trace. Note that we are not using the properties here.

Our approach in utilizing the tail portion is to start from the end and go backward on the trace while also searching forwarding from the initial state till we find a shorter trace. As this trace has some peculiar behavior mainly concentrated towards the end, we want to keep this part of the trace as it is in the shorter trace. But we do not know how far to go back on the trace. We start with some small length and use some heuristics to iteratively increase this. For the head portion of the shorter trace, we use some forward state search techniques. This approach is discussed in Chapter 3.

1.3.2 Debugging using the Trace and the Properties

With any digital design, we also have some properties which the design should ideally satisfy. We assume that these are written in LTL. The idea is to find if the set of properties of the design is refuted on the trace. This has several implications. If we really find that the property suite is refuted, we will also obtain the tail portion of the trace corresponding to that. We can then focus the simulation of the RTL in obtaining the same refutation by trying to create similar scenarios as in the tail. As said in the previous paragraph, we now have the complete controllability and observability of the RTL which helps in identifying the actual design defect. On the other hand, if we cannot obtain a property refutation it might either be the case that there is a physical bug or it may be the case that the property suite is incomplete. In the latter case, the entire behavior is not specified and the unexpected behavior is not covered by the existing suite and hence does not refute the existing suite. But the verification engineer can now guess that it might be the latter case and try to add new properties into the suite thus help refining the specification as well as help uncover the bug in a second iteration.

In this approach, we again work backward beginning from the end of the trace. But LTL properties talk about the future given the present. That is, given the values of some signals in the present time, they dictate the values of some signals in future. So, we need to reason the properties backward into past while we go back along the trace from the end. For this, we develop an algorithm which uses the semantics of the LTL properties to help them unfold from the present to the past. This we do for all the properties and whenever we find that the trace refutes the properties, we report the refutation. This approach is discussed in Chapter 4.

1.4 Thesis Organization

This thesis is organized as follows. We discuss the relevant background and earlier work in the area of Post-Silicon Verification in general and corresponding to our approaches in particular in Chapter 2. Then we discuss the approach of Post-Silicon Verification using the trace and the RTL design, give the experimental results and discuss the scalability issues in Chapter 3. We discuss the approach of Post-Silicon Verification using the trace and the LTL properties, give and discuss the experimental

results and discuss the scalability issues in Chapter 4. Finally we conclude and discuss possible future directions in Chapter 5.

Chapter 2

Background and Related Work

In this chapter we discuss the relevant background and earlier related works for both of our approaches.

Post-Silicon debugging has been studied before in various flavors. One of them is shortening the length of the Post-Silicon traces. Chang et al. [1] proposed an approach for automatically debugging the silicon for functional or physical errors and also to attempt to fix them, Post-Silicon. They employ the *Butramin* technique proposed in their other paper [2]. In a completely different approach, Huang et al. [3] propose embedding an Infrastructure IP block in an SOC along with other blocks for transaction based verification, also Post-Silicon. Directed studies for particular classes of problems have also been taken up. DeOrio et al. [4] proposed a Post-Silicon methodology for verifying cache coherence protocols in multi-core systems. Kapoor [5] discuss the challenges in the Post-Silicon validation of IBM Cell/B.E. and Game processors.

2.1 Shortening the trace

In this section, we first discuss the earlier work in shortening a given long trace. Then, we discuss why we use a SAT Solver in our approach.

2.1.1 Related Work

Post-Silicon traces are very huge and as such they are not of much use. Many people have earlier tried to shorten these traces under different names such as *trace minimization*, *trace compaction*, etc. Chang et al. [2] have recently proposed a *Bug Trace*

Minimization technique which has the exact objective as we have. But their algorithms intelligently use the silicon die itself to repeatedly do some simulations and iteratively decrease the length of the trace. They use techniques like *single-cycle elimination*, *input-event elimination*, *alternative paths to bugs*, etc. Apart from the simulation-based reductions, they also use a BMC-based refinement. Though our objective is the same, we try to use static approaches. Thus, we only work on the input trace and the RTL design. Shortening the traces has been studied in contexts different from Post-Silicon also. Safarpour et al. [6] have suggested *Trace Compaction* techniques where they take a long state sequence and try to obtain a shorter state sequence from the initial to the final state. But this is not very much applicable to Post-Silicon traces as we execute the inputs on the silicon die itself and due to the limited observability we cannot obtain a complete state sequence corresponding to the trace. Pan et al. [7] have also studied generation of shorter state sequences. In a contrasting approach, Wagner et al. [8] propose *Reversi*, a randomized program generation for identifying the correct final state before hand and the objective remains just to match the final state of the Post-Silicon trace with this.

SAT-based reachability has been studied in Chauhan et al. [9] where instead of keeping the expanded time-frames during a forward search for a state, they try to efficiently store the state space so generated. But that approach was not very much scalable. Moreover, they tried to solve the general reachability problem without using any trace. Our approach of using the tail of the trace is partially influenced by the Structural Target Enlargement of Baumgartner et al. [10].

2.1.2 SAT and BDD

As we discuss in Chapter 3, we make use of zChaff SAT Solver [11] for checking the satisfiability of some boolean expressions we build in the course of shortening the trace. In the verification community there is a challenging alternative for SAT solving techniques, namely the BDD based techniques. BDDs efficiently store the state space which is not explicitly present in SAT based techniques. It is implicitly involved in the time taken to solve a SAT expression. So, there is a time-space tradeoff between between the two techniques. It has been shown in Cabodi et al. [12] that SAT based methods outperform when compared to BDD based techniques in a closely related area of BMC which is also concerned about expanding the states (typically forward) till the goal is reached or a bound is reached. Also, in such areas including ours, we

need to solve several SAT expressions each of which is an *increment* over a previous SAT expression, called *incremental SAT-solving*. A better SAT solver especially for incremental solving and for Sequential Circuits using a 3-valued logic (`true`, `false` and `don't care`) called *Sequential SAT* has been used in [13], [14] and [15].

2.2 Reasoning a property refutation

In this section, we first give a brief background on LTL and properties and then discuss the earlier work related to the backward reasoning of properties which inspired our approach.

2.2.1 LTL

In this section, we discuss the syntax and semantics of LTL properties.

We use these operators in LTL

$$\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \mathbf{X}, \mathbf{F}, \mathbf{U}, \mathbf{G}$$

The syntax is given by the following Backus-Naur form

$$\phi : \neg\phi | \phi \vee \phi | \phi \wedge \phi | \phi \Rightarrow \phi | \phi \Leftrightarrow \phi | \mathbf{X}\phi | \mathbf{F}\phi | \phi \mathbf{U} \phi | \mathbf{G}\phi | \top | \bot$$

The boolean operators have their usual meaning. An LTL property has semantics with respect to a state sequence (called a path). Given a path $\pi = \langle s_1, s_2, \ldots, \rangle$ beginning at the initial state s_1, let π^t be the sub-path beginning at s_t, that is, $\langle s_t, s_{t+1}, \ldots, \rangle$. Thus, $\pi = \pi^1$. The semantics of the temporal operators is as follows

$$\pi \models \mathbf{X}\varphi \Leftrightarrow \pi^2 \models \varphi$$
$$\pi \models \mathbf{F}\varphi \Leftrightarrow \pi^t \models \varphi \text{ for some } t \geq 1$$
$$\pi \models \varphi \mathbf{U} \psi \Leftrightarrow \pi^t \models \psi \text{ and } \pi^{t'} \models \varphi \text{ for all } 1 \leq t' < t$$
$$\pi \models \mathbf{G}\varphi \Leftrightarrow \pi^t \models \varphi \text{ for all } t \geq 1$$

2.2.2 Types of Properties

Safety, Liveness and Fairness are the three types of properties commonly used for digital designs. Safety properties are used to specify that something happens always, while Liveness properties are used to specify that something is live, i.e., something happens infinitely often. Fairness properties, on the other hand, are like assume properties which assume that the environment is fair. We assume that all the properties of our system are Safety properties. If some property ϕ is not of that type, we can always rewrite it as $\mathbf{G}(start \Rightarrow \phi)$ which is a safety property.

2.2.3 Related Work

As said earlier, we would like to go back along the trace and reason the past given the present from the LTL properties. A similar notion is present in a variant of LTL called Past LTL or PLTL ([16], [17]). The syntax of LTL is enhanced by introducing new past operators. Though the power itself is not enhanced ([18]), it has got increasing popularity because properties can be expressed more succinctly in PLTL than LTL. People have also extended Bounded Model Checking for properties in PLTL ([19], [20], [18]). Our approach to unfold the LTL properties backward is partly motivated by the notion of past in PLTL. Using the LTL properties to reason backwards has earlier been studied to model check finite traces based on the finite path semantics of LTL using Alternating Automata in Finkbeiner et al. [21]. Firstly, they use a finite *state sequence* which we do not have. Secondly, we need to reason about the values of some signals due to limited observability. Thus, Alternating Automata are not such a great alternative for us as we have to any way use a SAT solver. Similar approaches have been studied in Havelund et al. [22] where they use the Term Rewriting System Maude for checking finite traces. An approach more similar to the one we propose in Chapter 4 has been briefly discussed in Markey et al. [23] where they discuss algorithms and various bounds for model checking a finite state sequence. They use *dynamic programming* to recursively obtain the value of subformulae of a PLTL formula. One difference in our approach is the limited observability and we do not have the entire state sequence. Secondly, our approach utilizes properties on internal signals which are not visible, to reason about their values and we try to obtain a collective refutation for the whole property suite. Our approach is also shown to be practical feasible, in Chapter 4.

Chapter 3

Design Assisted Debugging Approach

In this chapter, we will discuss our first approach towards debugging the Post-Silicon trace. As discussed in earlier chapters, we are concerned with very long input-output traces. This is because of the assumption that enough Pre-Silicon verification has already been done to uncover most of the common errors. Also, we assume that we have some golden behavior corresponding to the given trace. We are not very concerned with the exact behavior. It can be the golden input-output pairs which is very exhaustive and most of the times very difficult to have. In that case, the input-output pairs can be compared to the golden ones regularly so that the trace is reported as soon as a mismatch is found. Alternatively, it can be a set of assertions which are monitored periodically as the input-output pairs are obtained from the hardware execution. Again, we are not concerned with how this is done. We only assume that the trace has some unexpected behavior towards the end. It can be the very last state which should have been unreachable. It can also be the case that a state which should have been unreachable has already been visited and further states have also been visited before reporting the trace.

Now that we have this trace which is known to have some unexpected behavior, our objective is to debug the trace and to assist in debugging the RTL of the system. One naïve approach is to apply the inputs to the system as given by this trace and monitor the behavior of the RTL. This has two major flaws :

1. It defeats the sole purpose of Post-Silicon Verification which is to achieve a significant speed-up by executing directly on the hardware. Repeating the trace by simulation on the RTL is not a good idea.

2. Given that the trace is very long and the unexpected behavior is towards the end, it makes little sense to simulate the whole trace right from the initial state.

Then, how do we use the trace for debugging the RTL? The only way out is to *shorten* the trace. That is, given this long trace, our first objective is to get a smaller trace which also depicts the buggy scenario. Once this is done, this smaller trace can then be simulated on the RTL. It is to be understood that this particular buggy scenario depicted by the original trace is not a common one for otherwise, it would have been caught in the Pre-Silicon Verification itself. So, whatever smaller trace we expect to get, it is always helpful if it is *in the lines* of the original trace. For this, we work on the given trace to obtain a smaller one. Our goal is not to obtain *the smallest* possible trace but to obtain *comparably smaller* trace.

To simplify things, we assume that the *last state* reached along the given trace should have been *unreachable* and our goal is to try to *reach* that state using a smaller trace. As noted above, it will be very useful if we can make good use of the last portion of the trace which is where the unexpected behavior is spotted.

3.1 The Approach

We first describe the idea of *time frame expansion* which we make use of in the approach.

3.1.1 Time Frame Expansion

Here, we describe the technique of *time frame expansion* in our context. First, we view the entire digital system we are working on as a single sequential circuit. Considering the whole combinational block as a black box the only signals left out are the *inputs*, *outputs*, *present state bits* and *next state bits* as shown in Fig. 3.1. Now, the next state bits become the present state bits for the next clock cycle. Removing the flip-flops that connect next state bits to the present state bits, what we get is called a *time frame*. In *time frame expansion*, we use multiple such frames put in order by directly connecting the next state bits of one frame with the present state bits of the next frame. We thus *expand* the combinational block multiple times which amounts to replicating the whole logic and all the signals those many times. See Fig. 3.2 for an illustration. Note that the inputs and outputs for each frame are separate

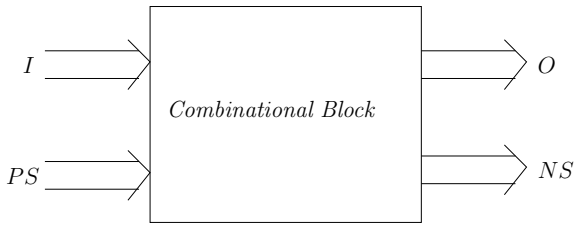


Figure 3.1: The combinational block of a sequential circuit. I, O, PS and NS stand for *inputs*, *outputs*, *present state* and *next state* respectively.

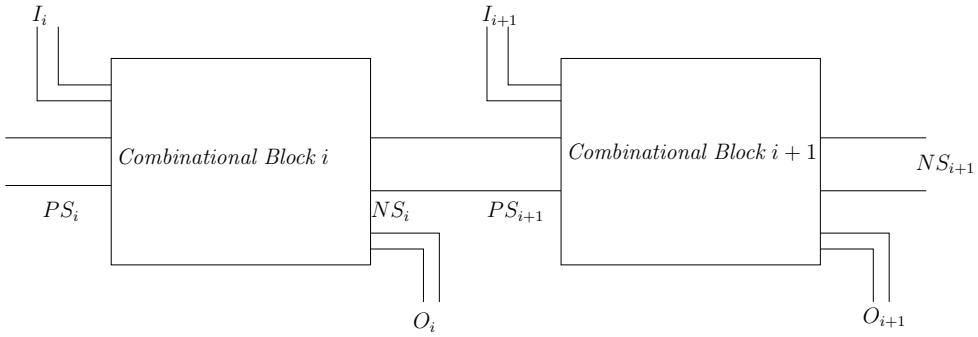


Figure 3.2: The time frame expansion where NS of previous (ith) frame are connected with PS of the next ($i+1$ th) frame. The inputs and outputs of each frame are separate.

and the advantage is that we can apply all the inputs and observe all the outputs simultaneously.

Note that this is only a *logical* expansion technique which we use to obtain a smaller trace.

3.1.2 Bidirectional Search

Here, we describe the complete approach for obtaining a smaller trace. As the name suggests, we do a bidirectional search for a smaller trace using the given trace to reach the final state on the trace from the initial state. To begin, we know the initial state S_{init} and there are techniques to obtain the final state S_{final}. In fact, these techniques can be employed to obtain the state of the sequential circuit (the values stored in the flip-flops) at any given point. Forming a *Scan Chain* with the flip-flops is a common technique. So, we assume that we are given with the initial and final states.

If we follow a naïve approach for searching for a path from S_{init} to S_{final} this is how it looks like. If we do a forward search, we initially have just S_{init}. At the

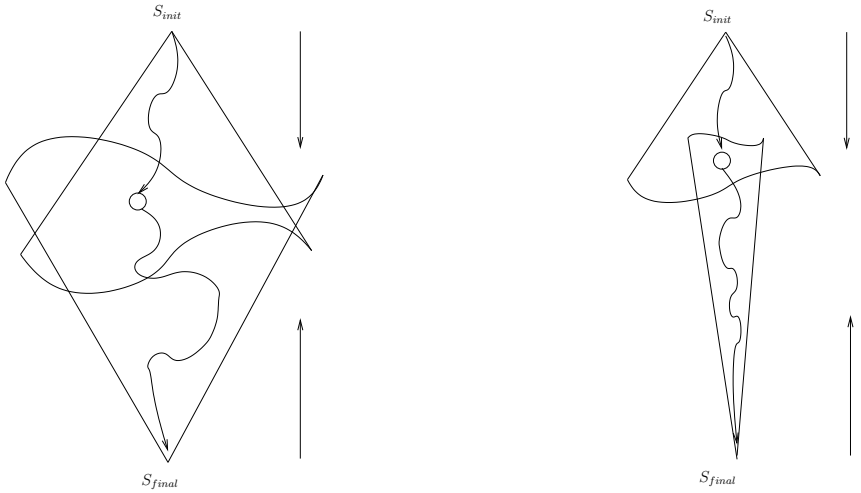


Figure 3.3: (a) The forward and backward cones without using trace. A path obtained from S_{init} to S_{final} may be not quite short as it may go through unnecessary states. (b) The backward cone narrowed down by the trace and the path from S_{init} to S_{final} is more focused to the goal, going along the trace.

next step, we have all the states that can reached from S_{init} in one step. Then, at every subsequent step, we obtain all the states reachable from the previously obtained states in one step. If at any point S_{final} is encountered, we found a path! Pictorially, this looks like a cone with S_{init} as root and we call it *forward cone*. Similarly, we can have a *backward cone* rooted at S_{final}. We can combine both cones to do a bidirectional search. The goal then becomes to have *some* common state in both cones. See Fig. 3.3(a) for illustration.

As already mentioned in the previous section, we make use of the given trace in our search. We have the following observations :

1. The unexpected behavior is concentrated towards the end of the trace.

2. The trace being very long, many states on the trace might have been revisited on the trace many times.

3. Many of these multiply visited states might be easily reachable from the initial state (may be through a different path).

Going by the first observation above, we first note that the tail part of the trace has to be properly utilized. Let us assume that we consider the last l input-output pairs. Let S_l be the state reached on the trace just before applying the lth input

(from the last). Note that we need not know what S_l is. Now going by the second and third observations above we try to reach S_l from the initial state but now, it need not be guided by the trace. Thus, in the backward search, we just go along with the trace. This helps in narrowing down the backward cone. This results in a small set of states that can be reached backward from S_{final}. We now try to hit one of these states in forward search, now unguided. By the second observation above, we expect *some* multiply visited state to be near the end of the trace and by the third observation, hope to hit it in the forward search *soon*. See Fig. 3.3(b).

We use time frame expansion for both directions. In the forward direction it is exactly as described before with the present state bits of the first frame now given the values of the initial state. Let the number of frames expanded be f. We follow a similar expansion for backward direction now that the next state bits of the previous frame have to be connected with the present state bits of the next frame, as the frames get added in the reverse order. We use the values of the final state for the next state bits of the last frame. For each frame added in the backward search, we also have the values for inputs and outputs from the trace. Let the number of frames expanded backward be b. Now we have to connect these two expanded lists of frames, somehow, to obtain a path. We call this process, *binding*. If any of the states S_i $1 \leq i \leq b$ reached along the trace before applying the ith input (from the last) of the trace is reached from the initial state we have a complete path from the initial state to the final state! We search for this by varying the values of b and f. We call the binding a *k-binding* if the present state bits of each of the first k frames of the backward search away from the frame corresponding to the final state are connected to the next state bits of each of the frames of the forward search. Thus, we have three parameters b, f and k. Fig. 3.4 shows 1-binding. For a given combination, we search for a path as follows.

We form the following SAT expression

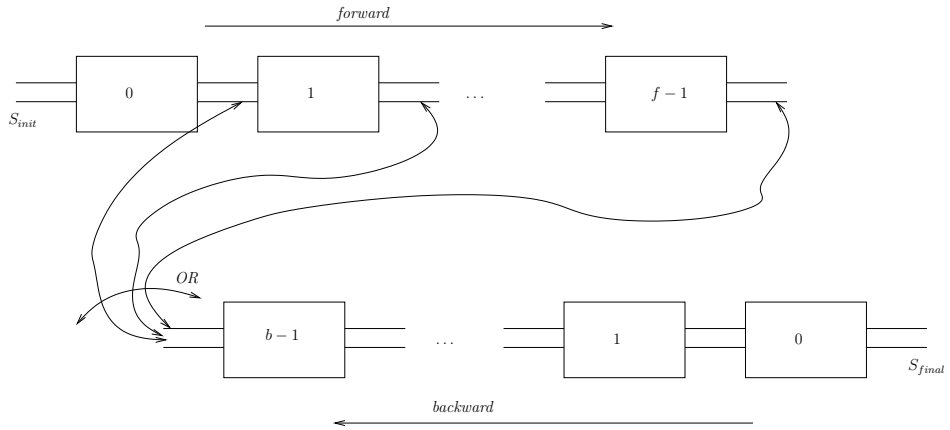


Figure 3.4: 1-binding between the frames expanded in forward search and backward search. The PS of the first frame of the backward search away from the one corresponding to the final state can be matched with NS of any of the forward frames.

$$
\begin{aligned}
&(PS_0 = S_{init}) \wedge \left(\bigwedge_{m=0}^{f-2} (trf(m) \wedge (NS_m = PS_{m+1})) \right) \wedge trf(f-1) \\
&\wedge \left(\bigwedge_{n=0}^{b-2} (trf(n) \wedge (NS_n = PS_{n+1})) \right) \wedge trf(b-1) \wedge (NS_{b-1} = S_{final}) \\
&\wedge \bigwedge_{n=0}^{k-1} \left(\bigvee_{m=0}^{f-1} (NS_m = PS_n) \right)
\end{aligned}
\tag{3.1}
$$

where PS, NS and trf stand for *present state*, *next state* and *transition function* (of the combinational block), respectively and the subscripts m and n represent the corresponding variables for *forward* and *backward* frames, respectively. We get additional clauses if we bind the inputs and outputs with the values from the trace.

If the above SAT expression evaluates to true, we obtain an assignment of the inputs and outputs and we thus have a smaller trace from the initial state to the final state.

3.2 Results

We did several experiments involving either pure forward or bidirectional searches and both using the trace or otherwise. If we do not use the trace for backward search, we just keep the inputs and outputs unassigned.

Table 3.1: Experimental results for pure forward search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. *tl* stands for *trace length*, *t* for *time needed for the search, in seconds* and *l* for *the length of the trace found*. *Linear* and *Exponential* denote the mode of addition of time frames. We use a timeout of 1000 seconds and *l* in that case is the length explored till then.

tl	*t*	*l*
$s208.1$		
10	0.1	7
500	429.7	257
1000	345.9	257
$s526$		
10	0.1	1
500	50.3	62
1000	666.1	125

To keep things simple, we always keep $f = b$. We start with an initial value, say 5. Then each iteration, we increment the values exponentially, doubling the previous values. We do this to avoid too many iterations before obtaining a path. After each iteration, the SAT expression so formed is checked for satisfiability. We made use of zChaff SAT solver [11]. For each possible mode of increment we choose either of the four values for k : 1, 10, $b/2$ or b. We experimented with some of the ISCAS 89 benchmark circuits. We first generated random traces of different lengths and various flavors of search are experimented. Table 3.1 shows the results for pure forward search. Tables 3.2-3.5 show the results for bidirectional search for various *bindings* of the forward and backward searches. We took these results on a quad Intel® Xeon™ 2.80 GHz CPU machine and a 4 GB RAM running the Linux ELsmp kernel.

It is evident from the tables that this approach needs exorbitant time for even practically small circuits like $s208.1$ and $s526$. We have also tried the approach for larger circuits like $s13207$ and $s38417$ but the program got aborted at very early stages due to memory problems. Time frame expansion for such huge circuits is thus not a very good idea. Nevertheless, we can draw some conclusions from the above results.

Firstly, whenever we found a path, pure forward search gives near optimal length traces, as expected. If we consider the trace lengths of 500 and 1000, this length is around 250 in both cases for $s208.1$ whereas it is around 50 and 125 respectively, for $s526$. If the trace is made use of, these lengths are around 400 for $s208.1$ and

Table 3.2: Experimental results for 1-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. w denotes *with trace* where the backward search uses the given trace and wo denotes *without trace*. tl stands for *trace length*, t for *time needed for the search, in seconds* and l for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and l in that case is the length explored till then.

	w		wo	
tl	t	l	t	l
$s208.1$				
10	0.1	$8(3+5)$	0.1	$8(3+5)$
500	79.7	$420(163+257)$	126.2	$258(129+129)$
1000	101.1	$452(195+257)$	169.7	$258(129+129)$
$s526$				
10	0.1	$3(1+2)$	0.1	$3(1+2)$
500	42.4	$125(60+65)$	16.4	$66(33+33)$
1000	97.5	$223(94+129)$	178.1	$129(64+65)$

Table 3.3: Experimental results for 10-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. w denotes *with trace* where the backward search uses the given trace and wo denotes *without trace*. tl stands for *trace length*, t for *time needed for the search, in seconds* and l for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and l in that case is the length explored till then.

	w		wo	
tl	t	l	t	l
$s208.1$				
10	0.1	$8(3+5)$	0.1	$9(4+5)$
500	137.1	$440(183+257)$	199.1	$257(128+129)$
1000	179.8	$464(207+257)$	218.6	$254(125+129)$
$s526$				
10	0.1	$3(1+2)$	0.1	$3(1+2)$
500	243.4	$128(65+63)$	27.2	$57(24+33)$
1000	1000	222	482.2	$104(44+60)$

Table 3.4: Experimental results for $b/2$-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. w denotes *with trace* where the backward search uses the given trace and wo denotes *without trace*. tl stands for *trace length*, t for *time needed for the search, in seconds* and l for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and l in that case is the length explored till then.

	w		wo	
tl	t	l	t	l
$s208.1$				
10	0.1	$8(3+5)$	0.1	$9(4+5)$
500	802.0	$446(189+257)$	705.7	$257(128+129)$
1000	937.3	$414(157+257)$	613.3	$258(129+129)$
$s526$				
10	0.1	$3(1+2)$	0.1	$3(1+2)$
500	358.2	$116(51+65)$	43.8	$56(23+33)$
1000	1000	132	723.7	$102(38+64)$

Table 3.5: Experimental results for b-*binding* in bidirectional search of the final state to obtain a smaller trace from a given very large trace for ISCAS 89 benchmark circuits $s208.1$ and $s526$. w denotes *with trace* where the backward search uses the given trace and wo denotes *without trace*. tl stands for *trace length*, t for *time needed for the search, in seconds* and l for *the length of the trace found* with the figures in the brackets meaning *length of trace in forward search* plus *length of trace in backward search*. We use a timeout of 1000 seconds and l in that case is the length explored till then.

	w		wo	
tl	t	l	t	l
$s208.1$				
10	0.1	$9(4+5)$	0.1	$9(4+5)$
500	1000	130	1000	258
1000	1000	130	1000	132
$s526$				
10	0.1	$3(1+2)$	0.1	$3(1+2)$
500	962.7	$118(58+60)$	44.2	$65(33+32)$
1000	1000	68	1000	107

around 125 and 200 respectively, for $s526$. This is just about double the optimal length which is not bad. What is more important is the time taken to reach this. A good observation of the tables shows that the best times are when the forward and backward searches are 1-*bound* or 10-*bound* when the trace is made use of apart from some exceptions. The tables also show that using the trace for the search saves as much as 40% time.

3.3 Conclusion

In this chapter, we discussed a formal approach for debugging the design using Post-Silicon traces along with the RTL. We utilized the idea of time frame expansion for searching for a path of a shorter length than that of the given trace from the initial state to the final state. As the trace is reported to have a bug and since this bug was not covered by the Pre-Silicon verification, this trace is some peculiar one and our approach tries to make good use of the trace. As the bug is reported at the end of the trace, it is expected that the cause of the bug is near the end and hence, the tail is the most important part of the trace. So, we do a time frame expansion in both directions, forward from the initial state and backward from the final state but in the backward expansion, we also make use of the signal values available from the trace. We combine both the expanded lists of frames and it is shown that for small circuits, 1-binding and 10-binding (refer to the previous sections) achieve the smallest times. This is also shown to be an improvement of about 40% over pure forward search. But, a major drawback of this approach is its scalability. Time frame expansion for large circuits turns out to be too costly. Thus, this approach, as is, may not be very useful in practice.

Chapter 4

Property Assisted Debugging Approach

Assertions have become ubiquitous in Pre-Silicon validation, because they add significant value towards exposing bugs and debugging the design. Today, in a complex digital design, assertions are everywhere, from small modules to large integrated SOCs. Not all of these assertions are checked using formal methods – in fact the capacity limitations of model checking tools have restricted formal property verification to modules of modest size. However assertions have proved to be useful in identifying bugs over simulation runs and vastly reducing the debugging effort.

The logical bugs which escape Pre-Silicon validation may be broadly classified into two categories:

1. *Type-1 Bugs.* Bugs for which properties were not specified at all. This typically happens if the architectural specification is misinterpreted and if enough properties are not written to cover the design intent.

2. *Type-2 Bugs.* Bugs for which properties were specified, but simulation coverage was not adequate enough to hit those bugs.

The presence of a bug is manifested in a Post-Silicon trace when the silicon output differs from the golden output. The available test structure (scan chains, etc) makes it possible to capture the state of the circuit when this happens, but it is hard to debug the cause for the mismatch. This is because the illegal behavior may not be a function of the present state, but may be the outcome of some logical bug which was sensitized many cycles ago.

In other words, when a logical bug is sensitized, it may take several cycles for the bug to propagate to the design interface and manifest itself in the Post-Silicon trace. The goal of the verification team is to identify the logical bug which is the cause of the mismatch. At this level the size of the integrated chip is so large that a static analysis of the design is virtually impossible. Also the speed of Pre-Silicon simulation is many orders of magnitude slower than the speed of the silicon, and hence running the whole trace through simulation is typically a costly affair.

This work is motivated by the fact that today a whole lot of assertions are available in the design. *Can we use these assertions to debug the trace and identify the cause of the mismatch?* There are several incentives in taking this approach.

1. Assertions are significantly lighter than RTL. Hence the analysis is expected to be scalable.

2. This approach allows the verification engineer to plug in more properties in retrospection. In other words, it enables the verification engineer to add properties in anticipation of Type-1 bugs after seeing the mismatch and *guessing* its cause.

There are several fundamental challenges in the proposed approach. These are:

1. The method must be able to use not only the assertions over the design interface, but also those assertions which are defined over the internal signals of the design. Assertions which express properties of the internal components of the design may assist debugging by propagating a bug towards the interface. Since the internal signals are not visible in the trace, the task of using the internal assertions in debugging the trace is a non-trivial problem.

2. The debugging method must be able to work backwards in the trace, starting from the state in which the mismatch was detected. There is a fundamental challenge in this task since temporal operators used in assertion languages express present and future behaviors, where as we need to use these properties to reason backwards, that is, from our present state in the trace towards the past states.

Since we do not know how many cycles were required by the bug to propagate to the design interface, there is a definite advantage in working backwards while debugging the trace.

It may be appropriate to point out at this stage that by using assertions in debugging as opposed to using the RTL, we strike a tradeoff between performance and the completeness of our approach. In other words, the RTL is a functionally complete model of the design. The design model we choose to use in this analysis is the module connectivity[1] and the assertions, which is lighter than the RTL but functionally incomplete. Therefore we may fail in some cases where simulation (given time) succeeds in finding an assertion failure. However, we consciously make this choice because the industry lacks a scalable approach for this problem, and a fast bug-hunting method is a useful precursor to a more time consuming search.

We discuss the following aspects in this chapter.

1. We highlight the role of assertions on internal components in isolating a bug which has manifested itself in a Post-Silicon trace.

2. We present a formal method for debugging a trace backwards using the assertions of the components. We present a formal proof establishing the soundness of our method.

3. We present experimental results to demonstrate the scalability of the approach and its utility in industrial practice.

4.1 A Motivating Example

Consider the module and the corresponding trace shown in Fig. 4.1 which has an input a, an output b and internal signals c, d, e and f. Let the following LTL properties be given on the signals

$$\mathbf{G}(a \Rightarrow e) \tag{4.1}$$

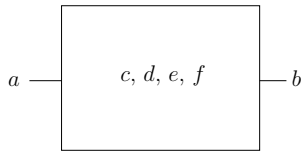
$$\mathbf{G}(e \Rightarrow \neg \mathbf{X} f) \tag{4.2}$$

$$\mathbf{G}(f \Leftrightarrow \mathbf{F}_{[3,3]} c) \tag{4.3}$$

$$\mathbf{G}(c \vee \mathbf{XX} d) \tag{4.4}$$

$$\mathbf{G}(d \Rightarrow \mathbf{XXXX} b) \tag{4.5}$$

[1] We assume that a signal name shared by two or more components represents an interconnection between them.



01. 00	11. 11
02. 01	12. 11
03. 01	13. 11
04. 11	14. 01
05. 10	15. 01
06. 01	16. 01
07. 11	17. 01
08. 00	18. 01
09. 10	19. 01
10. 10	20. 00

Figure 4.1: A module having an input a and an output b with c, d, e and f as internal signals and a trace for 20 cycles with the value of a followed by the value of b.

It is not difficult to reason a refutation in this simple example. As $\neg b$ is true at 20th step on the trace, (4.5) above implies $\neg d$ is true at 16th step. This and (4.4) above in turn imply c is true at 14th step. (4.3) now implies f is true at 12th step (note that **F** includes the current time step too) which in turn implies $\neg e$ is true at 11th step, from (4.2). Now (4.1) implies that $\neg a$ is true at 11th step. But according to the trace a is true at 11th step. And we have a refutation. It is important to note that we did not need to reason back till the initial step.

We now consider an example which shows that our approach is not complete. Consider the Verilog RTL shown below. The memory bit m is inverted every cycle only when g_1 is granted and the bit value is output only when g_2 is granted (the output is 0 otherwise). The request r (for g_1) has higher priority and g_1 is granted the next cycle while g_2 should be granted by default. The bug in the RTL is the **xnor** instead of **xor**, as indicated. r is the input and o is the output.

```
module ckt(r,clk,o);
input r,clk;
output o;
reg m, g1;
wire m1, g2;
 dff(g1,r,clk);
 dff(m,m1,clk);
 xnor(m1,g1,m); //the bug!
 and(o,g2,m);
 not(g2,g1);
endmodule
```

The engineer writes the following properties forgetting to write the property for the default grant of g_2.

$$\mathbf{G}(r \Leftrightarrow \mathbf{X}g_1) \text{ (higher priority)} \tag{4.6}$$

$$\mathbf{G}(\neg g_1 \vee \neg g_2) \text{ (mutual exclusion)} \tag{4.7}$$

$$\mathbf{G}((g_1 \wedge \neg m) \vee (\neg g_1 \wedge m) \Leftrightarrow \mathbf{X}m) \text{ (xor)} \tag{4.8}$$

$$\mathbf{G}(g_2 \wedge m \Leftrightarrow o) \text{ (output)} \tag{4.9}$$

Now consider the following valuations of the signals for six cycles for some input pattern. We use o and m for the bug free (golden) values and o_b and m_b for the buggy values.

r	g_1	g_2	m	o	m_b	o_b
1	0	1	0	0	0	0
1	1	0	0	0	1	0
1	1	0	1	0	1	0
0	1	0	0	0	1	0
0	0	1	1	1	1	1
0	0	1	1	$\underline{1}$	0	$\underline{0}$

Note that o differs from the golden output at the last step (0 for 1). The trace available Post-Silicon contains only the columns of r and o_b. If $\neg g_2$ is true at that step instead of g_2, $\neg o$ is also true (as o is the *and* of g_2 and m) and there would be no difference between the columns of o and o_b. Also, this violates none of the properties. Thus, the property set is not sufficient for our approach to catch the bug.

We will discuss more on these two examples in the following section.

4.2 The Sniffer Dog Approach

Let $\mathcal{M}$ be the given *model* which is a sequential digital circuit to be verified. Let $\mathcal{S}$ be the set of all *signals* of $\mathcal{M}$ which is the union of the set of *input* signals $\mathcal{I}$, the set of *output* signals $\mathcal{O}$ and the set of *internal* lines $\mathcal{N}$. Let $\mathcal{V} = \mathcal{I} \cup \mathcal{O}$, where $\mathcal{V}$ stands for *visible* as these are the only visible signals, Post-Silicon. In other words these are the only accessible signals whose boolean values can be tested. It follows that $\mathcal{N}$ is the set of all *invisible* signals.

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ be the set of all LTL *safety* properties on $\mathcal{S}$ such that it is desired to have $\mathcal{M} \models p_i$ for each i. As each p_i is a safety property, it is of the

form $\mathbf{G}\phi_i$, where ϕ_i is any LTL formula. We assume that not every p_i is only on $\mathcal{N}$. Otherwise it beats the whole purpose as the properties are useless. The operators we allow are from the set $\{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \mathbf{X}, \mathbf{F}, \mathbf{U}, \mathbf{G}\}$ which have their usual meaning and the following bounded variants of the temporal operators in the above set. Let $\pi = \langle s_0, s_1, \dots \rangle$ be the infinite state sequence corresponding to an infinite execution of $\mathcal{M}$ beginning at the state s_0. We denote $\langle s_k, s_{k+1}, \dots \rangle$ by π^k.

$$
\begin{aligned}
\pi \models \mathbf{X}_a\phi \quad & \iff \pi^a \models \phi \\
\pi \models \mathbf{F}_{[a,b]}\phi \quad & \iff \pi^{k-1} \models \phi \text{ for some } k, a \leq k \leq b \\
\pi \models \phi\mathbf{U}_{[a,b]}\psi \quad & \iff \pi^{k-1} \models \psi \text{ for some } k, a \leq k \leq b \text{ and} \\
& \qquad \pi^l \models \phi \text{ for all } l, 0 \leq l < k-1 \\
\pi \models \mathbf{G}_{[a,b]}\phi \quad & \iff \pi^{k-1} \models \phi \text{ for all } k, a \leq k \leq b
\end{aligned}
$$

Let $\mathcal{T}$ be the given finite input-output trace of $\mathcal{M}$ of length m obtained from the initial state of $\mathcal{M}$ and $\tau = \{t_0, t_1, \dots, t_m\}$, the corresponding state sequence (which is unknown). It is known that $\mathcal{T}$ has some unexpected behavior which is identified at the mth input-output pair. The problem is to see if this unexpected behavior is due to any property refutation on τ. Now, the only signal valuations we are provided with are those of $\mathcal{V}$ given by $\mathcal{T}$. So, we plan to use all the properties in $\mathcal{P}$ together (to gain more visibility) to see if there is any refutation.

4.2.1 The backward debug algorithm

The main advantage of Post-Silicon simulation is that we can obtain huge input-output traces which was not possible with Pre-Silicon simulation. Now that $\mathcal{T}$ is one such trace and the unexpected behavior is towards the end of $\mathcal{T}$, it is more likely that a property refutation (if any) occurred at a state t_i, i much closer to m than 0. This suggests to reason a possible refutation backwards, or *sniff* beginning with the mth input-output pair of $\mathcal{T}$. For this purpose, we propose the following *Sniffer Dog* approach.

Consider the set of all *subformulae* $sf(\phi)$ of an LTL formula ϕ which is recursively defined as follows

$$
\begin{aligned}
&sf(s) = \{s\},\ \text{if } s \text{ is a signal}\\
&sf(\neg\phi) = sf(\phi)\\
&sf(\phi \otimes \psi) = sf(\phi) \cup sf(\psi), \otimes \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}\\
&sf(\mathbf{X}\phi) = sf(\phi) \cup \{\mathbf{X}\phi\}\\
&sf(\mathbf{X}_a\phi) = \begin{cases} sf(\mathbf{X}_{a-1}\phi) \cup \{\mathbf{X}_a\phi\},\ \text{if } a > 1 \\ sf(\phi) \cup \{\mathbf{X}\phi\},\ \text{if } a = 1 \end{cases}\\
&sf(\mathbf{F}\phi) = sf(\phi) \cup \{\mathbf{F}\phi\}\\
&sf(\mathbf{F}_{[a,b]}\phi) = \begin{cases} sf(\mathbf{F}_{[a-1,b-1]}\phi) \cup \{\mathbf{F}_{[a,b]}\phi\},\ \text{if } a > 1 \\ sf(\mathbf{F}_{[a,b-1]}\phi) \cup \{\mathbf{F}_{[a,b]}\phi\},\ \text{if } a = 1, b > 1 \\ sf(\phi) \cup \{\mathbf{F}_{[a,b]}\phi\},\ \text{if } a = 1, b = 1 \end{cases}\\
&sf(\phi\mathbf{U}\psi) = sf(\phi) \cup sf(\psi) \cup \{\phi\mathbf{U}\psi\}\\
&sf(\phi\mathbf{U}_{[a,b]}\psi) = \begin{cases} sf(\phi) \cup sf(\phi\mathbf{U}_{[a-1,b-1]}\psi) \cup \\ \qquad \{\phi\mathbf{U}_{[a,b]}\psi\},\ \text{if } a > 1 \\ sf(\phi) \cup sf(\psi) \cup sf(\phi\mathbf{U}_{[a,b-1]}\psi) \cup \\ \qquad \{\phi\mathbf{U}_{[a,b]}\psi\},\ \text{if } a = 1, b > 1 \\ sf(\psi) \cup \{\phi\mathbf{U}_{[a,b]}\psi\},\ \text{if } a = 1, b = 1 \end{cases}\\
&sf(\mathbf{G}\phi) = sf(\phi) \cup \{\mathbf{G}\phi\}\\
&sf(\mathbf{G}_{[a,b]}\phi) = \begin{cases} sf(\mathbf{G}_{[a-1,b-1]}\phi) \cup \{\mathbf{G}_{[a,b]}\phi\},\ \text{if } a > 1 \\ sf(\mathbf{G}_{[a,b-1]}\phi) \cup \{\mathbf{G}_{[a,b]}\phi\},\ \text{if } a = 1, b > 1 \\ sf(\phi) \cup \{\mathbf{G}_{[a,b]}\phi\},\ \text{if } a = 1, b = 1 \end{cases}
\end{aligned}
\tag{4.10}
$$

The idea of splitting a formula into *subformulae* is imported from Clarke et al. [24].

We proceed by first finding $sf(\phi_i)$ where $p_i \equiv \mathbf{G}\phi_i$ for each $p_i \in \mathcal{P}$. While going back on the trace $\mathcal{T}$, at every step, we try to obtain the truth values of each *subformula* in each $sf(\phi_i)$. Now, consider the following rewriting of the temporal formulae

$$
\begin{aligned}
\mathbf{F}\phi &\equiv \phi \vee \mathbf{X}\mathbf{F}\phi\\
\phi\mathbf{U}\psi &\equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\mathbf{U}\psi))\\
\mathbf{G}\phi &\equiv \phi \wedge \mathbf{X}\mathbf{G}\phi
\end{aligned}
\tag{4.11}
$$

We can rewrite similarly for formulae with bounded operators as

$$
\begin{aligned}
\mathbf{X}_a\phi &\equiv \begin{cases} \mathbf{X}\mathbf{X}_{a-1}\phi, \text{ if } a > 1 \\ \mathbf{X}\phi, \text{ if } a = 1 \end{cases} \\
\mathbf{F}_{[a,b]}\phi &\equiv \begin{cases} \mathbf{X}\mathbf{F}_{[a-1,b-1]}\phi, \text{ if } a > 1 \\ \phi \vee \mathbf{X}\mathbf{F}_{[a,b-1]}\phi, \text{ if } a = 1, b > 1 \\ \phi, \text{ if } a = 1, b = 1 \end{cases} \\
\phi\mathbf{U}_{[a,b]}\psi &\equiv \begin{cases} \phi \wedge \mathbf{X}(\phi\mathbf{U}_{[a-1,b-1]}\psi), \text{ if } a > 1 \\ \psi \vee (\phi \wedge \mathbf{X}(\phi\mathbf{U}_{[a,b-1]}\psi)), \text{ if } a = 1, b > 1 \\ \psi, \text{ if } a = 1, b = 1 \end{cases} \\
\mathbf{G}_{[a,b]}\phi &\equiv \begin{cases} \mathbf{X}\mathbf{G}_{[a-1,b-1]}\phi, \text{ if } a > 1 \\ \phi \wedge \mathbf{X}\mathbf{G}_{[a,b-1]}\phi, \text{ if } a = 1, b > 1 \\ \phi, \text{ if } a = 1, b = 1 \end{cases}
\end{aligned}
\tag{4.12}
$$

It is evident from the rewriting of LTL formulae in (4.11) and (4.12) above that to evaluate the right hand side expressions we only need the valuations of some *subformulae* in the current step and that of some other *subformulae* in the immediate future step (which we would have already evaluated in the previous step as we are going backwards). We further note that each *subformula* whose value to be determined at the current step can again be recursively rewritten using (4.11) and (4.12) above until it boils down to obtaining valuations of some *signals* at the current step and some *subformulae* at the immediate future step (which are obtained as explained above). For the valuations of signals, the only ones available are those in $\mathcal{V}$ given by $\mathcal{T}$ at the current step. For all those *subformulae* whose valuations could not be determined, obviously, we leave them unassigned. Having obtained the valuations of all *subformulae* in each $sf(\phi_i)$ at the current step, we try to obtain the value of $\bigwedge_{i=1}^{n} p_i$. At any step, if this evaluates to $\perp$, as we shortly prove, we can conclude that we have found a property refutation on $\mathcal{T}$!

The detailed algorithm is given below as Algorithm 1 which is the top level routine and Algorithm 2 which is a function used by the top level routine. We use $f_{sf(\varphi)}$ to denote the formula of φ in terms of *subformulae* in $sf(\varphi)$ using boolean operators. For example, for $\phi \equiv a \vee \mathbf{X}b$ where a and b are signals, $sf(\phi) = \{a, b, \mathbf{X}b\}$ and $f_{sf(\phi)} = \phi_1 \vee \phi_2$ where $\phi_1 \equiv a$ and $\phi_2 \equiv \mathbf{X}b$. Also, we use φ^t to denote a boolean variable which has the value of φ at step t on $\mathcal{T}$, $1 \leq t \leq m$ and φ^{m+1} to denote the variable for the value at the immediate future step beyond the trace (which is unknown). Going with this notation, we mean by $f^t_{sf(\varphi)}$ the value of the formula

$f_{sf(\varphi)}$ at step t which is clearly obtained from the corresponding valuations of the *subformulae* in $sf(\varphi)$. Continuing with the above example, we have $f_{sf(\phi)} = \phi_1 \vee \phi_2$, $f_{sf(\phi_1)} = \phi_1$ and $f_{sf(\phi_3)} = \phi_3$ where $\phi_3 \equiv b$ and hence, $f^t_{sf(\phi)} = \phi^t_1 \vee \phi^t_2$, $f^t_{sf(\phi_1)} = \phi^t_1$ and $f^t_{sf(\phi_3)} = \phi^t_3$. Now, we use ϕ^{m+1}_1, ϕ^{m+1}_2 and ϕ^{m+1}_3 for the variables having the values of the corresponding *subformulae* at step $m+1$ which is immediate future beyond the trace. These values are unknown and hence are not directly useful. But they might be helpful in reasoning a refutation as we use them in the conjuncts we add to the formula $\mathcal{B}$ built in the algorithm. Beginning with $t = m$, we add the biimplication $\phi^t_2 \leftrightarrow \phi^{t+1}_3$ for every t down to 1. The other conjuncts we add to $\mathcal{B}$ are corresponding to $f^t_{sf(\phi)}$ which is $\phi^t_1 \vee \phi^t_2$ and to the signal values at time t which are ϕ^t_1 and ϕ^t_3 (assuming both a and b are in $\mathcal{V}$).

Algorithm 1 Sniffer Dog Approach on input the set $\{\mathbf{G}p_i,\ 1 \leq i \leq n\}$ (top level routine)

compute $\mathcal{E} = \bigcup_{i=1}^{n} sf(\phi_i)$
$\mathcal{B} \leftarrow \top$
for each *subformula* $\varphi \in \mathcal{E}$ such that $\varphi \equiv \mathbf{F}_{[1,1]}\phi$ or $\varphi \equiv \mathbf{G}_{[1,1]}\phi$ **do**
  $\mathcal{B} \leftarrow \mathcal{B} \wedge \{\varphi^{m+1} \Leftrightarrow f^{m+1}_{sf(\phi)}\}$
end for
for each *subformula* $\varphi \in \mathcal{E}$ such that $\varphi \equiv \phi \mathbf{U}_{[1,1]}\psi$ **do**
  $\mathcal{B} \leftarrow \mathcal{B} \wedge \{\varphi^{m+1} \Leftrightarrow f^{m+1}_{sf(\psi)}\}$
end for
for $t = m$ down to 1 **do**
  for each *subformula* $\varphi \in \mathcal{E}$ **do**
    $\mathcal{B}' \leftarrow$ `add_biimpl(`φ`,`t`)`
    $\mathcal{B} \leftarrow \mathcal{B} \wedge \mathcal{B}'$
  end for
  $\mathcal{B} \leftarrow \mathcal{B} \wedge \bigwedge_{i=1}^{n} \{f^t_{sf(\phi_i)}\}$, where $p_i \equiv \mathbf{G}\phi_i$
  for each signal $s \in \mathcal{E}$ **do**
    if s is either an input or output signal **then**
      $\mathcal{B} \leftarrow \mathcal{B} \wedge s^t$
    end if
  end for
  if $\mathcal{B}$ evaluates to $\perp$ **then**
    exit with failure
  end if
end for
exit with success

Algorithm 2 `add_biimpl(`φ`,`t`)` (adds all the biimplications corresponding to φ at step t)

$\mathcal{B}' \leftarrow \top$
if $\varphi \equiv \mathbf{X}\phi$ **then**
  $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t+1}_{sf(\phi)}\}$
else if $\varphi \equiv \mathbf{X}_a\phi$ **then**
  if $a > 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow (\mathbf{X}_{a-1}\phi)^{t+1}\}$
  else if $a = 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t+1}_{sf(\phi)}\}$
  end if
else if $\varphi \equiv \mathbf{F}\phi$ **then**
  $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\phi)} \vee \varphi^{t+1}\}$
else if $\varphi \equiv \mathbf{F}_{[a,b]}\phi$ **then**
  if $a > 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow (\mathbf{F}_{[a-1,b-1]}\phi)^{t+1}\}$
  else if $a = 1$ and $b > 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\phi)} \vee (\mathbf{F}_{[a,b-1]}\phi)^{t+1}\}$
  else if $a = 1$ and $b = 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\phi)}\}$
  end if
else if $\varphi \equiv \phi\mathbf{U}\psi$ **then**
  $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\psi)} \vee (f^{t}_{sf(\phi)} \wedge \varphi^{t+1})\}$
else if $\varphi \equiv \phi\mathbf{U}_{[a,b]}\psi$ **then**
  if $a > 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\phi)} \wedge (\phi\mathbf{U}_{[a-1,b-1]}\psi)^{t+1}\}$
  else if $a = 1$ and $b > 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\psi)} \vee (f^{t}_{sf(\phi)} \wedge (\phi\mathbf{U}_{[a,b-1]}\psi)^{t+1})\}$
  else if $a = 1$ and $b = 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\psi)}\}$
  end if
else if $\varphi \equiv \mathbf{G}\phi$ **then**
  $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\phi)} \wedge \varphi^{t+1}\}$
else if $\varphi \equiv \mathbf{G}_{[a,b]}\phi$ **then**
  if $a > 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow (\mathbf{G}_{[a-1,b-1]}\phi)^{t+1}\}$
  else if $a = 1$ and $b > 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\phi)} \wedge (\mathbf{G}_{[a,b-1]}\phi)^{t+1}\}$
  else if $a = 1$ and $b = 1$ **then**
    $\mathcal{B}' \leftarrow \mathcal{B}' \wedge \{\varphi^t \Leftrightarrow f^{t}_{sf(\phi)}\}$
  end if
end if
return $\mathcal{B}'$

4.2.2 Algorithm Analysis

We first look at the running of Algorithm 1 on the two examples considered in Section 4.1.

For the first example, $\mathcal{E} =$

$$\{a,\ b,\ c,\ d,\ e,\ f,\ \mathbf{X}b,\ \mathbf{XX}b,\ \mathbf{XXX}b,\ \mathbf{XXXX}b,$$
$$\mathbf{F}_{[1,1]}c,\ \mathbf{F}_{[2,2]}c,\ \mathbf{F}_{[3,3]}c,\ \mathbf{X}d,\ \mathbf{XX}d,\ \mathbf{X}f\}$$

Denote by v_i, $1 \leq i \leq 16$ the ith element of the above set in the given order so that v_i^t then denotes the value of that element at time step t. Add the conjunct $v_{11}^{21} \Leftrightarrow v_3^{21}$ corresponding to $\mathbf{F}_{[1,1]}c$. Now, for every t beginning with $t = 20$, first add the biimplications given in the function `add_biimpl` of the Algorithm to $\mathcal{B}$, whichever are applicable for the above $\mathcal{E}$ and then add the boolean formulae corresponding to each of the properties (4.1)-(4.5). For example, for $t = 15$ add $v_8^{15} \Leftrightarrow v_7^{16}$ for the *subformula* $\mathbf{XX}b$ and add $v_{11}^{15} \Leftrightarrow v_3^{15}$ for the *subformula* $\mathbf{F}_{[1,1]}c$. As an example for boolean formulae for properties, for the above t and for (4.5) add $\neg v_4^{15} \vee v_{10}^{15}$.

As $\neg v_2^{20}$ is true (from the trace), following the biimplications corresponding to v_7, v_8, v_9 and v_{10} it can be inferred that $\neg v_{10}^{16}$ is true. From the conjunct $\neg v_4^{16} \vee v_{10}^{16}$ of $\mathcal{B}$ corresponding to (4.5) it can be inferred that $\neg v_4^{16}$ is true. It can be inferred from the conjunct corresponding to (4.4) and biimplications of v_{14} and v_{15} that v_3^{14} is true. Similarly, v_6^{12} is true and v_5^{11} is true. Finally, $\neg v_1^{11}$ is true and v_1^{11} is true (from the trace) which clearly implies that at this point $\mathcal{B}$ evaluates to $\perp$.

Proposition 1 *The* Sniffer Dog *approach reports a failure if and only if there is a property refutation on* $\mathcal{T}$*.*

Proof. From the semantics of an LTL property over an infinite state sequence, it follows that all the biimplications added in Algorithm 1 (as given in the function `add_biimpl`) are valid. And as we add them every time step t, we also have the values of all the *subformulae* at t which can be inferred given the trace from t to m and hence the values of all ϕ_is at t which can be inferred. Let the algorithm report failure at t'. Now, it wouldn't have made a difference if the algorithm instead worked with the trace from t' to m. In other words, w.l.o.g. assume that $\mathcal{B}$ is completely built from m down to 1. What we then have is a conjunction of all ϕ_is over all time steps together with the inferred values of *subformulae* (through the biimplications and values of $\mathcal{V}$

on $\mathcal{T}$). As each p_i is a safety property, it readily implies that $\mathcal{B}$ evaluates to $\perp$ if and only if there is a property refutation on $\mathcal{T}$. □

Theorem 1 (*Soundness*) *Whenever the* Sniffer Dog *approach reports failure, there is a logical bug in the digital system.*

Proof. Follows directly from Proposition 1. □

Now consider the second example discussed in Section 4.1. As already mentioned there if $\neg g_2$ were made true at time step 6 the bug would go undetected and this is very well permitted by the properties at hand. Note that the bug in the RTL implementation is not the incorrect implementation of granting g_2 by default. In fact, that is correctly implemented. But since no property has been written to that effect it is masking the actual bug of using **xnor** instead of **xor**. This is w.r.t. the given implementation. On the other hand, as mentioned above, it could as well be a bug in the default grant of g_2. We never know. Now, if the engineer guesses that there is a lack of specification and that might be the reason for the approach not to report any refutation (which is exactly the case in this example), assume that she adds the following property.

$$\mathbf{G}(g_1 \vee g_2) \text{ (default grant)} \tag{4.13}$$

Is this sufficient for the bug to be detected? Yes. We have $\mathcal{E} = \{v_1 = r,$

$$v_2 = g_1, v_3 = g_2, v_4 = m, v_5 = o, v_6 = \mathbf{X}g_1, v_7 = \mathbf{X}m\}$$

After adding all the relevant conjuncts of $\mathcal{B}$ at time step 6, it is observed from the trace that $\neg v_5^6$ is true which implies $\neg v_3^6$ or $\neg v_4^6$ or both are true. After adding the conjuncts of time step 5, from $v_1^5 \Leftrightarrow v_6^5$ and $v_6^5 \Leftrightarrow v_2^6$ and from $\neg v_1^5$ (from the trace) it can be inferred that $\neg v_2^6$ is true. Now, from $v_2^6 \vee v_3^6$ (of the new property) it can be inferred that v_3^6 is true and hence $\neg v_4^6$ is true (from the above discussion on time step 6). This with the conjunct $v_7^5 \Leftrightarrow v_4^6$ implies that $\neg v_7^5$ is true. This combined with the conjunct of (4.8) at time step 5 implies that v_2^5 and v_4^5 have same values. It can be similarly inferred that $\neg v_2^5$ is true as $\neg v_1^4$ is true which implies $\neg v_4^5$ is true. But then from the conjunct of (4.9) $\neg v_5^5$ is true and v_5^5 is true from the trace. Thus $\mathcal{B}$ evaluates to $\perp$ at time step 4.

4.3 Experimental Results

We first demonstrate the use of the algorithm in detecting some violations of the $AMBA^{TM}AHB$ (*A*dvanced *H*igh-performance *B*us) protocol ([25]). Then, we show the results we obtained on arbitrary safety properties with various parameters. The algorithm is implemented in C++ and we made use of zChaff SAT solver [11] to check the satisfiability of the clauses so formed.

4.3.1 Demonstration using AHB

Consider the architecture in Fig. 4.2. It shows two masters M_0 and M_1 and two slaves S_0 and S_1 along with an arbiter A using an $AMBA^{TM}AHB$ compliant bus. (The Decoder is not shown in the figure.) As also shown, only the interface signals of S_1 are visible. For the examples we consider for demonstration, S_1 is not active in any transaction and in this mode, the signals at its interface which are also active in any current transaction with S_0 are *hreset_n*, *hmastlock*, *hmaster*, *hburst*, *hsize*, *htrans*, *hwrite*, *haddr* and *hwdata* (for readers who are not familiar with the protocol, please refer to the Specification document). As usual, *hreset_n* is active low. The four possibilities for *htrans* are 00($IDLE$), 01($BUSY$), 10($NONSEQ$) and 11(SEQ). For simplicity of presentation, we assume *hmaster* to be a one-bit vector and *haddr* to be a two-bit vector. Now we present two scenarios where multiple properties are needed to reason a non-trivial refutation and demonstrate how *Sniffer Dog* works it out. The type of bugs we are concerned with here are of protocol breaches by either a master or a slave.

As a first scenario, consider the following. M_0 starts a $WRAP4$ burst mode transfer to S_0. In response to the first beat, S_0 splits the transfer. But M_0 continues with its transaction with second and third beats, where ideally it is expected to abort and go into $IDLE$ mode. Meanwhile, A sees that the transaction has been split and parks the grant on the next master waiting, which happens to be M_1. Thus, the incorrect behavior observed from the visible signals is *a master change without completing the current transaction*. At this point a bug is reported and the following trace given for debugging.

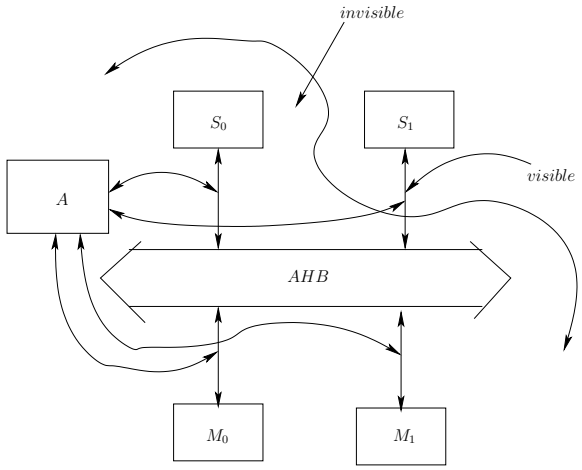


Figure 4.2: An architecture of two masters, M_0 and M_1, and two slaves, S_0 and S_1, using an $AMBA^{\mathrm{TM}}AHB$ compliant bus. A is the arbiter.

$hreset\_n$	$hmaster$	$hburst$	$haddr$	$htrans$
1	0	010	01	10
1	0	010	10	11
1	0	010	11	11
1	1	001	10	10

The following set of 12 properties can be written for a correct behavior of the system. For an n-bit vector v, let $v[i]$ denote the ith least significant bit, $0 \leq i \leq n-1$. Let b_0, b_1, b_2 and b_3 denote the 4 beats of a $WRAP4$ transfer. Also, we make use of *equality* relation in the properties, again to make the presentation simple. Define $wrap4 := \neg hburst[2] \wedge hburst[1] \wedge \neg hburst[0]$. Note that these properties have been tailored for the above scenario and the system. For example, we assume (in the second property) that b_0 is true at the beginning of the transaction and use a new boolean *start* which is true only at the first cycle to express that. Otherwise, these may turn more complex covering other scenarios to make the properties general.

$$\mathbf{G}(\neg(hresp[1] \wedge \mathbf{X}hresp[1]) \wedge b_0 \wedge b_1 \wedge wrap4 \Rightarrow$$
$$\mathbf{X}(hmaster = \mathbf{X}hmaster))$$
$$\mathbf{G}(start \Rightarrow b_0)$$
$$\mathbf{G}(b_0 \wedge (haddr \neq \mathbf{X}(haddr)) \Rightarrow \mathbf{X}(b_1))$$
$$\mathbf{G}(b_1 \wedge (haddr \neq \mathbf{X}(haddr)) \Rightarrow \mathbf{X}(b_2))$$
$$\mathbf{G}(b_2 \wedge (haddr \neq \mathbf{X}(haddr)) \Rightarrow \mathbf{X}(b_3))$$
$$\mathbf{G}(b_0 \Leftrightarrow \neg(b_1 \vee b_2 \vee b_3))$$
$$\mathbf{G}(b_1 \Leftrightarrow \neg(b_0 \vee b_2 \vee b_3))$$
$$\mathbf{G}(b_2 \Leftrightarrow \neg(b_0 \vee b_1 \vee b_3))$$
$$\mathbf{G}(b_3 \Leftrightarrow \neg(b_0 \vee b_1 \vee b_2))$$
$$\mathbf{G}(b_3 \Rightarrow \mathbf{X}(b_0))$$
$$\mathbf{G}(hresp[1] \wedge \neg hrdy \Rightarrow \mathbf{X}(\neg htrans[1] \wedge \neg htrans[0]))$$
$$\mathbf{G}(hresp[1] \wedge hrdy \Rightarrow \neg htrans[1] \wedge \neg htrans[0])$$

We only outline how the refutation can be reasoned. The first property and the change of $hmaster$ at the 4th cycle along with the properties on b_is imply that $hresp[1]$ is true at 2nd and 3rd cycles. But this along with the last two properties implies that $htrans[1]$ and $htrans[0]$ are low at 2nd or 3rd or 4th cycles (which is nothing but $IDLE$ mode in response to a $SPLIT$) none of which is true and hence the refutation. Note that this cannot be found until the initial cycle is reached as that is where b_0 is asserted by the second property. Our implementation of the algorithm uses 94 SAT variables and 233 SAT clauses per time step and finds the refutation after 4 time steps, as expected.

As a second scenario, we consider a bug on the part of S_0. Ideally, whenever a master floats a $BUSY$ mode, the slave should respond $OKAY$ in the next cycle. We consider a situation where it is violated. The master M_0 floats $BUSY$ in the middle of a transaction with S_0. But the slave responds with a $RETRY$ for the next two cycles upon which M_0 floats $IDLE$ (the correct behavior for the previous scenario) and the arbiter A sees a higher priority master M_1 waiting and parks the grant on M_1. Again, for an outside observer, this is an unexpected behavior and reports a bug giving the following trace (which can be the last part of a longer trace).

$hreset\_n$	$hmaster$	$hburst$	$htrans$
1	0	001	10
1	0	001	01
1	0	001	11
1	0	001	00
1	1	000	10

Following a similar notation as the first scenario, we have the following 7 properties. Define $incr := \neg hburst[2] \wedge \neg hburst[1] \wedge hburst[0]$.

$$\mathbf{G}(\neg(hresp[1] \wedge \mathbf{X}hresp[1]) \wedge incr \wedge \mathbf{X}(hbusreq_0) \Rightarrow \mathbf{X}(hmaster = \mathbf{X}(hmaster)))$$

$$\mathbf{G}(start \Rightarrow hbusreq_0)$$

$$\mathbf{G}(hresp[1] \wedge \neg hrdy \Rightarrow \mathbf{X}(\neg htrans[1] \wedge \neg htrans[0]))$$

$$\mathbf{G}(hresp[1] \wedge hrdy \Rightarrow \neg htrans[1] \wedge \neg htrans[0])$$

$$\mathbf{G}(\neg htrans[1] \wedge htrans[0] \Rightarrow \mathbf{X}(\neg hresp[0] \wedge \neg hresp[1] \wedge hrdy))$$

$$\mathbf{G}((hmaster = \mathbf{X}hmaster) \wedge \neg hmaster \Rightarrow hbusreq_0)$$

$$\mathbf{G}((hmaster = \mathbf{X}hmaster) \wedge (htrans[1] \vee htrans[0]) \wedge \neg hresp[1] \Rightarrow \mathbf{X}(htrans[1] \vee htrans[0]))$$

Here, only two properties are actually used for the refutation, namely the 5th and 7th, which say that *$BUSY$ should be followed by a high hrdy and $OKAY$ response from the slave* and *during a normal transaction without $RETRY/SPLIT$ non-$IDLE$ modes should be followed by non-$IDLE$ modes only*, respectively. After going back till 2nd cycle, the algorithm reports a refutation. As $htrans$ is $BUSY$ at that cycle, fifth property implies that $\neg hresp[1]$ is true at the 3rd cycle which along with 7th property and $htrans$ at that cycle give the refutation. Our implementation uses 56 SAT variables and 131 SAT clauses per time step and reports the refutation after 4 time steps.

4.3.2 Experiments with arbitrary safety properties

The parameters considered while randomly generating properties are the average length of a property (which is taken as the total number of temporal and boolean

operators), total number of signals, number of visible signals, length of the counter-example trace (which is the suffix of the given trace which actually contributes in finding the refutation). The visible signals are equally divided among the input and output signals. The lower bound for bounded temporal operators is a random number less than or equal to 2 and the upper bound (if applicable) is at most 3 more than the lower bound. As for the number of properties, it is initially fixed to 10. As we didn't start with any specific *scenarios* which correspond to some counter-example trace, we fixed the length of an expected counter-example trace, say to 500, and then generated properties and a random trace by unfolding the properties forward and giving random values for the input signals repeatedly, till we obtained a trace of the above length without any property refutation. Thus, the trace so obtained, in fact, corresponds to some counter-example trace of length *at least* the above length. But this just serves our purpose and henceforth, we use *the length of the counter-example trace* to mean the length fixed apriori the property generation. The following is such a set of properties generated for an average length of 10, total number of 100 signals out of which 50 are visible and for a counter-example trace of length 500. The nth ($0 \leq n \leq 99$) signal is denoted as $\_n$.

$$\mathbf{G}((\neg((\mathbf{F}(\neg(\_87)) \Rightarrow \_12)) \Rightarrow \_8))$$
$$\mathbf{G}(\neg(\mathbf{G}(\mathbf{X}_{[1]}(\neg(\mathbf{F}_{[1,1]}(\mathbf{X}_{[2]}(\_9)))))))$$
$$\mathbf{G}(\neg(\mathbf{X}(((\_90 \wedge (\_50 \Leftrightarrow \_58)) \wedge (\_65\mathbf{U}_{[1,2]}\mathbf{G}(\_84))))))$$
$$\mathbf{G}((\mathbf{G}_{[2,2]}(\mathbf{F}_{[1,2]}((\_44 \Rightarrow (\_26 \vee \_10))))\mathbf{U}_{[2,3]}\mathbf{F}(\mathbf{G}(\mathbf{G}_{[2,2]}(\_94)))))$$
$$\mathbf{G}((\mathbf{G}(\mathbf{X}(\mathbf{X}_{[2]}((\mathbf{G}(\mathbf{X}(\mathbf{G}(\mathbf{X}(\_9)))) \Rightarrow \_68)))) \wedge \_36))$$
$$\mathbf{G}(\mathbf{X}_{[1]}((\mathbf{F}(\mathbf{F}(\mathbf{X}_{[2]}((\mathbf{G}(\mathbf{F}((\_12 \wedge \mathbf{X}_{[1]}(\_82)))) \Rightarrow \_42))))\mathbf{U}\_33)))$$
$$\mathbf{G}(\neg(\mathbf{G}_{[1,2]}(\mathbf{F}((\mathbf{F}(\mathbf{F}((\_77\mathbf{U}_{[2,2]}\mathbf{F}((\_37\mathbf{U}\_38)))))\vee \mathbf{G}(\mathbf{X}(\_96)))))))$$
$$\mathbf{G}(\mathbf{X}(\mathbf{F}((\neg(\mathbf{F}(((\_27 \Rightarrow \_22) \Leftrightarrow (\mathbf{X}(\neg((\neg(\_28)\mathbf{U}_{[1,2]}\_72))) \wedge \_57)))) \wedge \_50))))$$
$$\mathbf{G}((\mathbf{X}_{[2]}((\mathbf{X}_{[1]}(\mathbf{F}_{[2,4]}(((\_38\mathbf{U}_{[1,2]}((\_87 \Leftrightarrow \mathbf{F}_{[2,2]}(\_97))\wedge \_88)) \Leftrightarrow (\_61\mathbf{U}\_60))))\mathbf{U}\_8)) \wedge (\_5 \Rightarrow \neg(\_37))))$$
$$\mathbf{G}(\mathbf{G}_{[2,3]}(\neg((\mathbf{X}_{[2]}(\mathbf{G}((((\_0 \Rightarrow \_34) \vee (\mathbf{G}(\mathbf{F}_{[2,4]}(((\mathbf{G}_{[1,3]}(\_59) \Rightarrow \_57)\mathbf{U}\_97))) \vee \_77))\mathbf{U}\_11))) \Rightarrow \_80))))$$

See Table 4.1 for the experimental results of the implementation on a quad Intel® Xeon™ 2.80 GHz CPU machine and a 4 GB RAM running the Linux ELsmp kernel. Typically, properties of digital systems do not have a length of more than 20. Thus, we experimented with four different lengths - 6, 10, 15 and 20. For each of these, and for each length of counter-example trace (corresponding to three consecutive columns) visibility of signals is varied as shown along the rows. Also, we decreased the frequency at which satisfiability of the intermediate formula $\mathcal{B}$ is checked (number of steps after which a new satisfiability check is made is increased) with the length of trace. In fact, this is also an important parameter affecting the total time taken in finding the refutation. Too many satisfiability checks will increase the time. We took the counter-example traces of lengths 50, 100, 500, 1000, 2000 and 3000 and the number of steps after which a new satisfiability check is made for each of these is 1, 1, 100, 500, 500 and 500 respectively.

Table 4.1: The experimental results of *Sniffer Dog* approach for various parameters. For each set of parameters, 10 properties are randomly generated. *vis* stands for *visible signals*, *tot* stands for *all signals*, l stands for *length of the counter-example trace*, t stands for *running time in seconds*, v stands for *SAT variables per time step*, c stands for *SAT clauses per time step*. The number in the brackets in each column header indicates the number of steps after which a new satisfiability check is made.

#vis/ #tot	l = 50 (1)			l = 100 (1)			l = 500 (100)			l = 1000 (500)			l = 2000 (500)			l = 3000 (500)		
	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$
average property length = 6																		
10/20	0.4	89	211	2.5	88	214	1.4	79	196	2.3	103	256	6.4	100	245	10.5	84	214
50/100	0.8	106	274	2.5	108	280	1.9	90	228	3.3	113	297	6.8	92	238	11.8	100	246
500/1000	0.8	100	688	2.5	100	695	2.4	120	740	3.1	101	703	9.1	122	746	12.9	92	676
5/20	0.5	89	206	1.8	88	209	1.5	79	191	2.6	103	251	6.4	100	240	10.8	84	209
25/100	0.6	106	249	2.5	108	255	1.4	90	203	2.9	113	272	6.1	92	213	10.3	100	221
250/1000	0.6	100	438	2.1	100	445	2.0	120	490	2.9	101	453	9.2	122	496	10.8	92	426
3/20	0.4	89	204	2.4	88	207	1.3	79	189	3.1	103	249	6.2	100	238	10.3	84	207
15/100	0.8	106	239	3.1	108	245	1.8	90	193	3.0	113	262	6.7	92	203	12.6	100	211
150/1000	0.7	100	338	2.5	100	345	1.9	120	390	2.6	101	353	8.6	122	396	11.4	92	326
average property length = 10																		
10/20	0.8	160	402	4.4	162	413	4.4	154	393	5.4	178	449	10.9	148	378	19.7	155	392
50/100	0.8	147	364	5.3	180	449	2.9	165	402	5.4	195	478	13.8	182	461	23.4	199	490
500/1000	0.1	207	948	4.4	183	878	3.6	191	880	4.8	177	854	15.8	194	901	30.7	212	931
5/20	0.9	160	397	5.0	162	408	10.8	154	388	6.3	178	444	10.6	148	373	21.4	155	387
25/100	0.7	147	339	5.5	180	424	3.6	165	377	6.2	195	453	12.4	182	436	25.3	199	465
250/1000	1.5	182	622	3.4	114	479	7.5	159	570	5.1	177	604	15.2	185	627	30.2	196	658
3/20	1.1	160	395	4.2	162	406	10.3	154	386	5.6	178	442	11.1	148	371	20.1	155	385
15/100	1.0	147	329	5.1	180	414	3.1	165	367	6.4	195	443	12.6	182	426	30.3	199	455
150/1000	1.4	207	598	5.1	183	528	3.7	191	530	5.0	177	504	14.2	194	551	27.3	212	581
average property length = 15																		
10/20	1.8	247	651	7.2	247	632	4.6	241	608	7.8	246	632	17.4	235	598	27.6	216	555
50/100	1.2	217	521	8.4	287	687	5.0	263	632	6.5	256	606	18.4	255	626	36.3	289	701
500/1000	2.1	281	1104	7.6	255	1031	6.1	283	1090	7.8	254	1017	19.4	253	1037	36.7	293	1109
5/20	1.6	247	646	8.9	247	627	5.0	241	603	8.4	246	627	19.9	235	593	27.2	216	550
25/100	1.5	217	496	7.8	287	662	4.7	263	607	7.0	256	581	17.6	255	601	37.5	289	676
250/1000	2.6	291	881	9.5	305	887	6.8	297	861	8.4	240	760	26.4	317	914	34.3	233	728
3/20	1.8	247	644	7.4	247	625	26.8	241	601	7.6	246	625	17.2	235	591	26.9	216	548
15/100	1.5	217	486	8.5	287	652	5.6	263	597	6.9	256	571	18.4	255	591	38.4	289	666
150/1000	2.2	281	754	7.4	255	681	5.3	283	740	7.6	254	667	18.3	253	687	41.7	293	759
average property length = 20																		
10/20	2.4	276	735	6.6	251	641	7.0	320	845	10.0	312	815	21.9	287	756	47.6	335	885
50/100	2.6	310	742	9.4	311	763	7.0	368	910	11.1	341	812	27.9	360	882	44.7	336	789
500/1000	2.6	332	1202	11.4	367	1262	7.8	391	1338	12.5	373	1290	28.0	376	1284	62.8	387	1306
5/20	2.1	276	730	6.8	251	636	6.6	320	840	10.0	312	810	25.3	287	751	46.2	335	880
25/100	2.5	310	717	9.2	311	738	10.0	368	885	12.8	341	787	28.8	360	857	44.9	336	764
250/1000	2.6	301	871	13.1	374	1037	8.3	348	997	17.0	340	997	32.7	376	1034	53.3	341	960
3/20	1.6	276	728	6.7	251	634	6.4	320	838	10.4	312	808	27.5	287	749	47.3	335	878
15/100	2.4	310	707	9.4	311	728	10.9	368	875	12.2	341	777	29.8	360	847	46.4	336	754
150/1000	2.8	332	852	12.6	367	912	9.8	391	988	15.5	411	1023	31.9	376	934	62.7	387	956

To analyze the results, it is first necessary to analyze the nature of properties generated above. One aspect is to see how useful each property is in a given set to infer the required refutation. For example, if a property is on some *invisible* signals none of which appear in any other property, then that property is useless for the refutation. The other aspects include how *inter-related* the properties are in a set, that is, how does a property assist other properties in finding the refutation. For example, in the first example in Section 4.1 the refutation found was such that (4.5) assisted (4.4) in inferring the value of the signal c (at some time step) which in turn assisted (4.3) in inferring the value of f and so on. Below we describe one possible way to achieve this analysis which we think is a reasonable one.

Build the parse tree of each property neglecting all the unary temporal operators and treating '**U**' similar to '$\wedge$'. Do a pre-order traversal noting down what truth values of the current node (assuming it to be a boolean value, forgetting the subtree rooted by that node for the time being) can be possibly inferred from that of its parent. The root is assumed to be true. (As each property has to be true we do not consider the falsity of the property.) Ultimately we are only interested in the truth values of the signals that appear in each property which can be inferred from the truth value of the property. The following table shows which values of the children can be inferred from the truth values of their parents. a, b, c and d are assumed to be boolean values.

parent	$\top$	$\bot$
$\neg a$	a	$\neg a$
$a \wedge b$	a, b	$\neg a, \neg b$
$a \vee b$	a, b	$\neg a, \neg b$
$a \Rightarrow b$	$\neg a, b$	$a, \neg b$
$a \Leftrightarrow b$	$a, b, \neg a, \neg b$	$a, b, \neg a, \neg b$

The first row is very easy to see. If $a \wedge b$ is true, we can only infer that a and b both are true. If it is false, $\neg a \vee \neg b$ is true which is treated as in the case of the third row. If $a \vee b$ is true we can only infer that b is true or a is true (corresponding to $\neg a \Rightarrow b$ and $\neg b \Rightarrow a$). It is never the case that we infer $\neg a$. Note that we are only concerned with what *new* the property infers. Now, if $a \vee b$ is false we infer that both $\neg a$ and $\neg b$ are true. Similarly we get the other entries of the table.

After the traversal of the parse tree, we obtain the set of values of each signal that can possibly be inferred from the truth value of the property. Now draw a digraph

with the properties as vertices and add an edge from property i to property j if and only if

1. i and j have common signals, and
2. for any common signal s, the truth value of s which can possibly be inferred by i implies (upon substituting that value for s in j) a *non-trivial* inference in j

where a *trivial* inference is either $\top$ or $\bot$. We are excluding the *trivial* inferences for then i is playing no role in assisting j!

The graph so obtained has all the information regarding the inter-relation among the properties and the use of each property in the refutation. Let us take an example to understand the construction of the graph. Consider the three properties where a, b, c and d are signals.

1. $\mathbf{G}(a \Rightarrow \mathbf{X}(b \wedge \mathbf{F}c))$
2. $\mathbf{G}(\neg b)$
3. $\mathbf{G}(c \Leftrightarrow \mathbf{X}d)$

The parse trees (after stripping off the unary temporal operators) annotated with the truth values that can possibly be inferred at the nodes and the final digraph are shown in Fig. 4.3. There is an edge from property 1 to property 3 as c being true infers a *non-trivial* d. Similarly c being false infers the *non-trivial* $\neg a$ and c being true infers the *non-trivial* $a \Rightarrow b$ and hence an edge from 3 to 1. There is an edge from 2 to 1 as b being true infers a *non-trivial* $a \Rightarrow c$ to be true. There is no edge from 1 to 2 as b being true infers a trivial $\bot$. Note that at this point we might have actually found a refutation. But 2 always asserts that $\neg b$ is true. As such 1 is not *assisting* 2 in inferring anything new about the values of the signals that appear in 2. When the satisfiability of the clauses at this point is checked, this refutation is automatically found. There are no edges between 2 and 3 as they have no common signals.

We are interested in the following parameters of the digraph obtained as above.

1. The number of components of the undirected graph obtained by removing the directions of edges from the above digraph. (This gives a measure of how

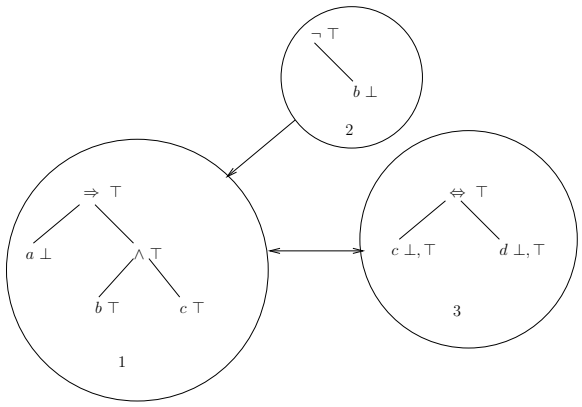


Figure 4.3: The graph of inter-relations of three properties whose annotated parse trees, after neglecting the unary temporal operators, are also shown inside the circles representing the vertices.

disconnected the property set is in reasoning a refutation. If it is known beforehand that the graph is disconnected and if there are significantly many components, an alternative would be to use the algorithm on each of these components separately and then combine the inferences at the end. This might give an improvement in the running time. This is just an idea which we haven't implemented.)

2. The number of properties in a component (formed as described above) which have at least one *visible* signal. (If no property in a component has any *visible* signal, then that component is useless in inferring anything new about the signals.)

3. Number of directed edges in each of the above components (whose properties have at least one *visible* signal). (This gives a measure of how inter-related the properties are.)

4. Number of *visible* signals which actually appear in any property in each of the above components. (This gives a measure of how well the properties can make use of the trace.)

5. Number of vertices (properties) on a longest path in each of the above components. (This gives a measure of how many properties are needed for a refutation in the worst case.)

See Table 4.2 for the values of the above parameters for some representative property sets, whose results are shown in Table 4.1.

From the table it can be seen that for a given average length of the properties, $\#c$ increases significantly with $\#tot$. And all of $\#p$, $\#e$, $\#v/\#vis$ and $\#n$ decrease with $\#tot$. This is expected as the more the number of signals available, the less is the chance of them being common. But this is also typically the case in systems with large number of signals, if we consider properties on all components of such a system.

Another important observation from the table is that for any given pair of $\langle \#vis, \#tot \rangle$ and any given value of l, as the average length of properties is increased, the inter-relation also increases. For example, for $\langle 50, 100 \rangle$ and $l \approx 10$, $\#v$ varies as 11, 22, 24 and 35 and $\#n$ varies as 5, 5, 10 and 10 as average length varies as 6, 10, 15 and 20.

Despite of the wide range of the above variations, we can observe from Table 4.1 that the significant factors affecting the running time, number of SAT variables per time step or number of SAT clauses per time step are only the number of properties in the set and the length of the counter-example! We can observe from Table 4.1 that for any given average length of a property and the length of a counter-example trace, the above three parameters have values of the same order except for some glitches. For example, for an average length of 10 of properties and the length of counter-example trace 500, running time is around 4 seconds, number of SAT variables is around 150 and number of SAT clauses is around 400 except for two rows where the running time is around 10 seconds. But it is to be noted that if $\#vis$ is decreased for the same $\#tot$ the counter-example of a possible refutation may increase in general and in that case, we need to look for the values in the table corresponding to a larger length of the counter-example trace (towards the right), which as shown in the table are greater than those towards the left.

Thus, the points to note are

1. Working with only those properties which are necessary can help reduce the running time,

2. Number of properties and the length of the counter-example trace (which is what to be found) are the important factors affecting the running time.

To see how scalable this approach is we also experimented with sets of 50 properties each. To help do a comparison with the results in Table 4.1 we first created sets of

Table 4.2: The values of various parameters of some property sets used in Table 4.1. As in that table, *vis* stands for *visible signals* and *tot* for *all signals*. l stands for *length of trace*. The column heads correspond to the value of l of the order of 10, 100 and 1000 respectively. c stands for *components* in the undirected graph corresponding to the digraph of the property set, p for *properties* in any *component* having at least one *visible* signal, e for *edges in that component*, v for *visible signals in that component* and n for *properties on a longest path in that component*. Entries indicated by '-' show that there is no component with any *visible* signal appearing in any property. All comma separated entries in a row are corresponding.

#vis/	$l \approx 10$					$l \approx 100$					$l \approx 1000$				
#tot	#c	#p	#e	#v	#n	#c	#p	#e	#v	#n	#c	#p	#e	#v	#n
average property length = 6															
10/20	1	10	30	9	10	2	9	29	8	8	1	10	33	6	9
50/100	4	5, 3	12, 4	11, 3	5, 3	8	3	3	4	3	8	2, 2	2, 2	1, 5	2, 2
500/1000	10	–	–	–	–	10	–	–	–	–	10	–	–	–	–
5/20	1	10	30	5	10	2	9	29	4	9	1	10	33	4	8
25/100	3	5, 3	12, 6	7, 1	5, 3	8	–	–	–	–	3	2	2	3	2
250/1000	10	–	–	–	–	10	–	–	–	–	10	–	–	–	–
3/20	1	10	30	3	10	2	9	29	2	8	1	10	33	2	9
15/100	4	5, 3	12, 4	5, 1	5, 3	8	–	–	–	–	8	2	2	2	2
150/1000	10	–	–	–	–	10	–	–	–	–	10	–	–	–	–
average property length = 10															
10/20	1	10	58	10	10	1	10	48	9	10	1	10	66	9	10
50/100	1	10	19	22	5	3	7, 2	14, 1	13, 2	6, 2	3	5, 4	12, 7	10, 11	5, 4
500/1000	10	–	–	–	–	7	3, 2	4, 2	9, 7	3, 2	9	2	2	7	2
5/20	1	10	58	5	10	1	10	48	5	10	1	10	66	5	10
25/100	1	10	19	8	6	3	7, 2	14, 1	10, 2	7, 2	3	5, 4	12, 7	3, 5	5, 4
250/1000	7	4	6	2	3	10	–	–	–	–	9	2	2	5	2
3/20	1	10	58	3	10	1	10	48	3	10	1	10	66	3	10
15/100	1	10	19	5	5	3	7	14	4	6	3	5, 4	12, 7	2, 3	5, 4
150/1000	10	–	–	–	–	7	2	2	3	2	9	2	2	1	2
average property length = 15															
10/20	1	10	82	10	10	1	10	37	9	10	1	10	85	9	10
50/100	1	10	37	24	10	1	10	29	29	10	1	10	40	32	10
500/1000	4	7	14	21	6	6	5	8	25	4	9	2	2	10	2
5/20	1	10	83	5	10	1	10	87	5	10	1	10	84	4	10
25/100	1	10	37	7	10	1	10	29	10	10	1	10	40	15	10
250/1000	6	5	8	8	5	6	4, 2	6, 2	14, 5	3, 2	8	2, 2	2, 2	3, 2	2, 2
3/20	1	10	82	3	10	1	10	37	3	10	1	10	85	2	10
15/100	1	10	37	7	10	1	10	29	7	10	1	10	40	9	10
150/1000	4	7	4	6	6	6	5	8	7	4	9	2	2	4	2
average property length = 20															
10/20	1	10	90	10	10	1	10	90	10	10	1	10	84	10	10
50/100	1	10	38	35	10	1	10	60	42	10	1	10	47	37	10
500/1000	8	3	4	14	3	9	2	2	7	2	3	6, 3	12, 4	39, 11	5, 3
5/20	1	10	90	5	10	1	10	90	5	10	1	10	84	5	10
25/100	1	10	38	18	10	1	10	60	20	10	1	10	47	22	10
250/1000	6	3, 2, 2	4, 2, 2	5, 6, 8	3, 2, 2	5	5, 2	8, 2	11, 4	5, 2	7	4	5	18	4
3/20	1	10	90	3	10	1	10	90	3	10	1	10	84	3	10
15/100	1	10	38	10	10	1	10	60	15	10	1	10	47	14	10
150/1000	8	3	4	2	3	9	2	2	2	2	3	6, 3	12, 4	6, 6	5, 3

10 properties each and obtained counter-example traces just as before. But now, we added 40 *stray* properties to each set. By *stray* we mean that none of the new properties is added to any existing component of the graph for 10 properties. They form a set of new components and are basically useless in any refutation as they are written on totally new signals. For each property of the set of 10 properties, four new properties are added to the set which differ only in the signals used. Thus $\#tot$ is increased by four times. The same counter-example traces are used and results are taken similar to that in Table 4.1. They are tabulated in Table 4.3. It can be observed that the results show similar variations as in Table 4.1. But the running times are significantly higher. This shows that it is always better to prune the set of properties based on external knowledge either manually or based on some heuristics. Nevertheless, the results in Table 4.3 show that the running times are not exorbitant and can be afforded for, thus showing the scalability of the *Sniffer Dog* approach.

4.4 Conclusion

In this chapter, we discussed a new formal approach for utilizing the Post-Silicon traces in the debugging of the design. We used a light-weight model of the module connectivity and the properties of the system (in LTL) and proposed an algorithm for checking if the trace is indeed a counter-example trace for this model, i.e., if the trace refutes the property suite. The algorithm works backwards on the trace for two reasons. One reason is that the cause of the bug is expected to be near the end of the trace and the second reason, which follows from the first, is that it makes little sense to start off from the beginning of a huge Post-Silicon trace when we know that the bug is near the tail. For this, the properties (formulae) are broken down to subformulae and the truth values of these subformulae are reasoned at every step while going backwards. The algorithm is shown to report a property refutation whenever one exists. The approach has two advantages. If the algorithm indeed reports a property refutation, we have also found the exact cycle (step) where the cause of the bug began. This is especially possible because we worked on the properties backward and this wouldn't have been possible with a forward search. Now, if we can also obtain the state information at this cycle, we can simulate the tail from this cycle on the RTL with full controllability and observability and know more about the bug and possibly repair it. On the other hand, if the algorithm doesn't report a property refutation

(note that the algorithm can be terminated before the beginning of the trace is reached if necessary and report that there is *possibly* no refutation, after a sufficient length of the tail has been used), new properties can be added in retrospection refining the specification and also help finding a property refutation in a second iteration. This approach has also been shown to be scalable and useful in industrial practice.

Table 4.3: The experimental results of *Sniffer Dog* approach for various parameters for sets of 50 properties built upon those of Table 4.1. r stands for the *ratio* of the *visible* and *total* number of signals corresponding to that of Table 4.1 with *#tot* (in that table) multiplied by 5. l stands for *length of the counter-example trace*, t stands for *running time in seconds*, v stands for *SAT variables per time step*, c stands for *SAT clauses per time step*. The number in the brackets in each column header indicates the number of steps after which a new satisfiability check is made.

	$l = 50$ (1)			$l = 100$ (1)			$l = 500$ (100)			$l = 1000$ (500)			$l = 2000$ (500)			$l = 3000$ (500)		
r	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$	t	$\#v$	$\#c$
average property length = 6																		
1	2.9	445	1015	12.0	440	1030	7.6	395	940	47.1	515	1240	32.6	500	1185	49.1	420	1030
2	5.0	530	1170	20.0	540	1200	11.4	450	940	24.9	565	1285	29.6	495	1075	73.3	500	1030
3	4.8	500	1440	18.3	500	1475	12.9	600	1700	20.1	505	1515	50.0	610	1730	66.5	460	1380
4	3.9	445	1010	15.3	440	1025	9.5	395	935	53.8	515	1235	40.6	500	1180	61.6	420	1025
5	4.8	530	1145	18.8	540	1175	10.4	450	915	20.1	565	1260	47.5	460	965	68.9	500	1005
6	3.2	550	1365	13.2	565	1400	7.6	560	1375	11.2	585	1425	38.4	585	1400	51.2	545	1340
7	4.0	445	1008	15.3	440	1023	9.0	395	933	61.3	515	1233	40.7	500	1178	61.5	420	1023
8	3.1	530	1135	14.9	540	1165	8.7	450	905	16.8	565	1250	38.6	460	955	54.4	500	995
9	2.9	500	1090	11.8	500	1125	8.1	600	1350	12.3	505	1165	29.8	610	1380	42.1	460	1030
average property length = 10																		
1	5.7	800	1970	25.8	810	2025	113.8	770	1925	54.3	890	2205	46.5	740	1850	102.5	775	1920
2	7.4	735	1620	31.3	900	2045	21.9	825	1810	59.8	975	2190	86.4	910	2105	285.3	995	2250
3	10.4	1035	2740	35.4	915	2340	26.9	955	2400	28.3	885	2270	82.9	970	2505	158.3	1060	2655
4	7.5	800	1965	29.3	810	2020	153.6	770	1920	70.2	890	2200	59.6	740	1845	114.2	775	1915
5	6.9	735	1595	32.8	900	2020	19.9	825	1785	57.8	975	2165	86.8	910	2080	242.5	995	2225
6	4.92	785	1830	24.9	1020	2335	14.5	1040	2415	15.3	775	1805	62.9	765	1830	84.6	835	1945
7	7.6	800	1963	29.4	810	2018	156.5	770	1918	69.6	890	2198	59.9	740	1843	113.9	775	1913
8	4.5	735	1585	22.2	900	2010	15.5	825	1775	41.0	975	2155	54.7	910	2070	194.3	995	2215
9	6.6	1035	2390	23.2	915	2040	19.3	955	2050	17.8	885	1920	50.0	970	2155	100.5	1060	2305
average property length = 15																		
1	12.3	1235	3215	58.7	1235	3120	140.6	1205	3000	963.1	1230	3120	125.5	1175	2950	76.5	1080	2735
2	8.0	1085	2405	56.3	1435	3235	41.2	1315	2960	43.2	1280	2830	116.9	1275	2930	273.6	1445	3305
3	14.6	1405	3520	46.7	1275	3155	45.7	1415	3450	75.8	1270	3085	121.4	1265	3185	256.2	1465	3545
4	15.4	1235	3210	73.8	1235	3115	170.6	1205	2995	1326.0	1230	3115	154.2	1175	2945	166.1	1080	2730
5	10.2	1085	2380	53.0	1435	3210	38.8	1315	2935	41.2	1280	2805	112.1	1275	2905	258.3	1445	3280
6	9.7	1450	3305	44.5	1520	3455	21.8	1215	2850	62.6	1440	3215	133.5	1380	3090	152.4	1360	3100
7	15.4	1235	3208	74.1	1235	3113	170.4	1205	2993	59.6	1230	3113	155.2	1175	2943	165.8	1080	2728
8	7.6	1085	2370	38.23	1435	3200	25.5	1315	2925	27.8	1280	2795	73.0	1275	2895	172.8	1445	3270
9	9.2	1405	3170	31.6	1275	2805	31.4	1415	3100	46.7	1270	2735	76.8	1265	2835	165.7	1465	3195
average property length = 20																		
1	10.7	1380	3635	37.8	1255	3165	55.3	1600	4185	201.3	1560	4035	208.9	1435	3740	235.0	1675	4385
2	16.3	1550	3510	57.8	1555	3615	74.9	1840	4350	158.8	1705	3860	264.1	1800	4210	256.1	1680	3745
3	16.7	1660	4010	69.3	1835	4310	75.1	1955	4690	217.1	2055	4865	219.0	1880	4420	430.8	1935	4530
4	13.6	1380	3630	45.8	1255	3160	66.5	1600	4180	397.1	1560	4030	260.0	1435	3735	336.4	1675	4380
5	15.5	1550	3485	57.7	1555	3590	76.0	1840	4325	159.4	1705	3835	257.0	1800	4185	420.7	1680	3720
6	12.9	1900	4155	50.8	1990	4410	47.1	1930	4320	104.8	1940	4280	113.6	1550	3510	328.2	1730	3910
7	13.6	1255	3158	45.9	1255	3158	55.3	1600	4178	2989.4	1560	4028	183.2	1435	3733	225.9	1675	4378
8	10.3	1550	3475	39.0	1555	3580	57.9	1840	4315	102.4	1705	3825	173.7	1800	4175	309.8	1680	3710
9	11.3	1660	3660	47.4	1835	3960	52.7	1955	4340	156.4	2055	4515	156.0	1880	4070	295.2	1935	4180

Chapter 5

Conclusions and Future Work

In this thesis, we proposed two contrasting approaches for Post-Silicon verification, i.e., utilizing Post-Silicon traces for design debugging. In the first approach discussed in Chapter 3, we made use of the RTL design along with the trace to assist in the debugging. Though this was shown to be a significant improvement over the naïve approach, this was not shown to be a scalable approach. In the second approach discussed in Chapter 4, we used a lighter model of the module connectivity and properties along with the trace to assist in the debugging. The goal of this approach is to see if the trace refutes the property suite which can then be used for design debug. This was shown to be a very scalable approach.

Finally, we would like to suggest some future directions in the lines of the above approaches.

5.1 Design Assisted Debug

The main problem with this approach was the scalability to large circuits. One way to attack this problem is to use abstractions of the design very much in the lines of CEGAR [26]. Start off with the most abstract design and keep refining it based on some heuristics. One can also obtain the state information at multiple points on the trace, instead of just the initial and final states thus increasing the observability and solve multiple reachability problems, then combining them in the end.

5.2 Property Assisted Debug

Though this approach was shown to be quite scalable, several improvements are possible. As suggested while discussing the nature of the results in Table 4.1, the set of properties can be first studied to see how inter-related they are. If there are too many disconnected components, one may run the algorithm on each of these components and combine the inferences at the end. Best, this can also be parallelized boosting the performance. Another aspect which can be researched is refining the end goal of the approach itself. Instead of just saying that the property suite is refuted, if there is some way to pin point a subset of properties which is refuted by the trace, future jobs of bug hunting and removal can be more directed. As of now, we do not know of any good way to do this. We strongly hope that this line of approach finds interest in the verification community and more innovation comes in the long run.

5.3 Combining both approaches

The light-weight model used in the property assisted debug can also be combined with the RTL design in innovative ways to obtain better results. The design gives a complete model of the system while using the entire system blindly can lead to scalability problems. Thus, there is a scope of utilizing the design along with properties leading to better algorithms and better objectives.

Bibliography

[1] K.-h. Chang, I. L. Markov, and V. Bertacco, Automating post-silicon debugging and repair, in *ICCAD '07: Proceedings of the 2007 IEEE/ACM International conference on Computer-Aided Design*, pp. 91–98, 2007.

[2] K.-h. Chang, V. Bertacco, and I. L. Markov, Simulation-based bug trace minimization with BMC-based refinement, in *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-Aided Design*, pp. 1045–1051, Washington, DC, USA, 2005, IEEE Computer Society.

[3] Y. Huang and W.-T. Cheng, Using embedded infrastructure IP for SOC post-silicon verification, in *DAC '03: Proceedings of the 40th conference on Design automation*, pp. 674–677, New York, NY, USA, 2003, ACM.

[4] A. DeOrio, A. Bauserman, and V. Bertacco, Post-silicon verification for cache coherence, in *ICCD '08: Proceedings of the 2008 IEEE International conference on Computer Design*, pp. 348–355, 2008.

[5] S. Kapoor, Challenges in post-silicon verification of IBMs Cell/B.E. and other game processors, in *Proceedings of High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pp. 48–52, 2007.

[6] S. Safarpour, A. Veneris, and H. Mangassarian, Trace Compaction using SAT-based Reachability Analysis, in *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation*, pp. 932–937, Washington, DC, USA, 2007, IEEE Computer Society.

[7] S.-J. S.-R. Pan, K.-T. Cheng, J. Moondanos, and Z. Hanna, Generation of shorter sequences for high resolution error diagnosis using sequential SAT, in *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pp. 25–29, Piscataway, NJ, USA, 2006, IEEE Press.

[8] I. Wagner and V. Bertacco, Reversi: Post-silicon validation system for modern microprocessors, in *ICCD '08: Proceedings of the 2008 IEEE International conference on Computer Design*, pp. 307–314, 2008.

[9] P. Chauhan, E. M. Clarke, and D. Kroening, Using SAT based Image Computation for Reachability Analysis, in *SCS CMU Technical Report CMU-CS-03-151*, Pittsburgh, PA, USA, 2003.

[10] J. Baumgartner, A. Kuehlmann, and J. A. Abraham, Property Checking via Structural Analysis, in *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pp. 151–165, London, UK, 2002, Springer-Verlag.

[11] zChaff SAT Solver, www.princeton.edu/∼chaff/zchaff.html.

[12] G. Cabodi, P. Camurati, and S. Quer, Can BDDs compete with SAT solvers on bounded model checking?, in *DAC '02: Proceedings of the 39th conference on Design automation*, pp. 117–122, New York, NY, USA, 2002, ACM.

[13] F. Lu *et al.*, An Efficient Sequential SAT Solver With Improved Search Strategies, in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 1102–1107, Washington, DC, USA, 2005, IEEE Computer Society.

[14] G. Parthasarathy, M. K. Iyer, K.-T. T. Cheng, and L.-C. Wang, Safety Property Verification Using Sequential SAT and Bounded Model Checking, in *Design and Test of Computers, IEEE* Vol. 21, pp. 132–143, Los Alamitos, CA, USA, 2004, IEEE Computer Society Press.

[15] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L. C. Wang, Efficient reachability checking using sequential SAT, in *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pp. 418–423, Piscataway, NJ, USA, 2004, IEEE Press.

[16] O. Lichtenstein, A. Pnueli, and L. D. Zuck, The Glory of the Past, in *Proceedings of the Conference on Logic of Programs*, pp. 196–218, London, UK, 1985, Springer-Verlag.

[17] D. M. Gabbay, The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems, in *Temporal Logic in Specification*, pp. 409–448, London, UK, 1987, Springer-Verlag.

[18] T. Latvala, K. Heljanko, and T. Junttila, Simple is better: Efficient bounded model checking for past LTL, in *Proceedings of Verification, Model Checking and Abstract Interpretation. Volume 3385 of LNCS*, pp. 380–395, Paris, France, 2005, Springer-Verlag.

[19] M. Benedetti and A. Cimatti, Bounded Model Checking for Past LTL, in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems. Volume 2619 of LNCS*, pp. 18–33, Warsaw, Poland, 2003, Springer-Verlag.

[20] A. Cimatti, M. Roveri, and D. Sheridan, Bounded Verification of Past LTL, in *Proceedings of Formal Methods in Computer-Aided Design. Volume 3312 of LNCS*, pp. 245–259, Austin, Texas, USA, 2004, Springer-Verlag.

[21] B. Finkbeiner and H. Sipma, Checking Finite Traces Using Alternating Automata, in *Formal Methods in System Design* Vol. 24, pp. 101–127, Hingham, MA, USA, 2004, Kluwer Academic Publishers.

[22] K. Havelund and G. Rosu, Testing Linear Temporal Logic Formulae on Finite Execution Traces, in *Technical Report*.

[23] N. Markey and P. Schnoebelen, Model Checking a Path, in *CONCUR 2003 - Concurrency Theory. Volume 2761 of LNCS*, pp. 251–265, Springer-Verlag, 2003.

[24] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking* (MIT Press, Cambridge, Massachusetts, 2001).

[25] AMBA Specification Rev 2.0, http://www.arm.com/products/solutions/AMBA_Spec.html.

[26] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, Counterexample-Guided Abstraction Refinement, in *Computer Aided Verification. Volume 1855 of LNCS*, pp. 154–169, Chicago, IL, USA, 2000, Springer-Verlag.