Efficient Graph Techniques for Partial Scan Pattern Debug and Bounded Model Checkers

Supratik Kumar Misra

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Michael S. Hsiao, Chair
Sandeep K. Shukla
A. Lynn Abbott

3rd February, 2012
Blacksburg, Virginia

Keywords: Directed Acyclic Graph, Partial Scan Design, Pattern Debugger
Implication Graphs

Copyright 2012, Supratik K Misra

Efficient Graph Techniques for Partial Scan Pattern Debug and Bounded Model Checkers

Supratik Kumar Misra

(ABSTRACT)

Continuous advances in VLSI technology have led to more complex digital designs and shrinking transistor sizes. Due to these developments, design verification and manufacturing test have gained more importance and 70 % of the design expenditure in on validation processes. Electronic Design Automation (EDA) tools play a huge role in the validation process with various verification and test tools. Their efficiency have a high impact in saving time and money in this competitive market. Direct Acyclic Graphs (DAGs) are the backbone for most of the EDA tools. DAG is the most efficient data structure to store circuit information and also have efficient backt traversing structure which help in developing reasoning/ debugging tools.

In this thesis, we focus on two such EDA tools using graphs as their underlying structure for circuit information storage

- Scan pattern Debugger for Partial Scan Designs
- Circuit SAT Bounded Model Checkers.

We developed a complete Interactive Scan Pattern Debugger Suite currently being used in the industry for next generation microprocessor design. The back end is an implication graph based sequential logic simulator which creates a Debug Implication Graph during the logic simulation of the failing patterns. An efficient node traversal mechanism across time frames, in the DIG, is used to perform the root-cause analysis for the failing scan-cells. In addition, the debugger provides visibility into the circuit internals to understand and fix the

root-cause. We integrated the proposed technique into the scan ATPG flow for industrial microprocessor designs. We were able to resolve the First Silicon logical pattern failures within hours, which would have otherwise taken a few days of manual effort for root-causing the failure, understanding the root-cause and fixing it.

For our circuit SAT implementation, we replace the internal implication graph used by the SAT solver with our debug implication graph (DIG). There is a high amount of circuit unrolling in circuit SAT/ BMC (Bounded Model Checking) problems which creates copies of the same combinational blocks in multiple time frames. This allows us to use the repetitive circuit structure and club it with the CNF database in the SAT solver. We propose a new data structure to store data in a circuit SAT solver which results up to 90% reduction in number of nodes.

To....

Maa & Baba

Acknowledgments

It gives me great pleasure to acknowledge all people who made this work possible. First, I owe my sincere gratitude to my advisor Prof. Michael S. Hsiao, for his continued support and academic guidance. He has been a source of inspiration on how to do applied research and learn from multiple sources in daily life. Interactions with him during our individual meetings and group meetings are the most valuable moments in my graduate life.

I thank Prof. Sandeep Shukla and Prof. Lynn Abbott for graciously agreeing to serve on my thesis committee. I also thank the administrative staff of the Graduate School and the Bradley Department of Electrical and Computer Engineering for their timely help and continued cooperation with paperwork and other administrative matters.

I would like to express my sincere gratitude to Sanjay Sengupta for giving me an opportunity to intern in his team at Intel Corporation and to work on a challenging project towards my thesis. I thank Kameshwar Chandrasekar for his continued mentorship during my eleven months stay at Intel. I learned a lot from my interactions with him during at period. He has been a friend, guide and a great teacher. I thank him for his guidance even after the internship and helping me with new ideas towards my thesis.

I am thankful to my friends in the PROACTIVE group for creating such a learning yet fun atmosphere in the lab - Maheshwar Chandrasekar, Mainak Banga, Saparya Krishnamoorthy,

Neha Goel, Min Li, Dhrumeel Bakshi, Huy Nguyen, Nikhil Rahagude, Chinmaye Limaye, Nevedetha Narayanan and Indira Munagani. Special thanks to Sarvesh Prabhu for the numerous problem solving ideas and continuous motivation throughout my graduate school.

I also thank my friends, Apoorv Naik, Anup Mandlekar, Shankha Banerjee and Sachin Hirve for all the fun time and cherish able memories during my years at Virginia Tech.

Finally, I would like to thank my parents Ganesh Misra and Surasree Misra for their unconditional love and support. I also thank God for showering His blessings on me.

Supratik K. Misra

February 03, 2012

Contents

1 Introduction **1**

1.0.1 Scan Pattern Debugger for Partial Scan Designs 2

1.0.2 Circuit SAT Solvers and Bounded Model Checkers 4

2 Background **7**

2.1 Design for Testability - Scan 7

2.2 Pattern Debug 9

2.3 Static Logic Implication 11

2.3.1 Direct Implications 12

2.3.2 Indirect Implications 14

2.3.3 Extended Backward Implications 14

2.3.4 Implication Graph 17

2.4 Boolean Satisfiability - SAT 18

2.5 Circuit Unrolling for SAT based Bounded Model Checking (BMC) 20

3 Debug Implication Graph **22**

3.1 DIG Construction 23

3.2 Backward Traversal 28

3.3 Summary 30

4 Interactive Scan Pattern Debugger **31**

4.1 Introduction 31

4.2 Motivation 33
4.3 Select a Silicon Failure 34
4.4 Debug the Silicon Failure 35
4.5 Pattern Debug Flow 37
4.6 Industrial Data 39
4.7 Summary 40

5 Circuit SAT Implementation with DIG **42**
5.1 Introduction 42
5.2 Implication Graph in SAT Solver 43
5.3 Key Idea 45
5.4 Implementation 45
5.4.1 Data Structure 46
5.4.2 Graph Construction 47
5.4.3 Conflict Analysis - Backward Traversal 48
5.5 Experimental Setup and Results 48

6 Conclusion **52**

Bibliography **55**

List of Figures

1.1 Verification Gap Vs Design Gap 2

2.1 Scan Cell and Scan Chain 9

2.2 Example Circuit 13

2.3 Example Circuit 14

2.4 AND gate and its Implication Graph 18

2.5 Unrolling and Monitor circuit for Bounded Model Checking 19

3.1 DIG Construction 23

3.2 Frame-wise DIG Construction 26

3.3 Event Driven Simulation does not Keep Track of the Latest Reason 27

3.4 Final Graph 28

4.1 Scan Pattern Delivery Flow 32

4.2 Interactive Pattern Debug Flow 38

5.1 Implication Graph 44

5.2 Data Structure 47

List of Tables

2.1 Controlling Values 13

4.1 Industrial Data for Pattern Debugger 37

4.2 DIG Run-time Overhead 40

5.1 Implication Graph in SAT Solver using DIG 51

List of Algorithms

3.1 Algorithm for Backward Traversal DIG 30
5.1 Algorithm for Backward Traversal 49

Chapter 1

Introduction

Integrated circuits (ICs) are indispensable in today's electronics systems. These systems range from small micro controllers to very complex processors. As digital circuits continue to follow Moore's law [1] and become more complex, engineers are facing huge challenges in bringing out a robust product within the Time To Market (TTM) goals. Here the Electronic Design Automation (EDA) tools play an indispensable role in reducing human effort, time and money. Complex systems invite a greater likelihood of manufacturing defects and more design errors. Since it is quintessential to deliver a bug-free, defect-free robust product, more resources are being allotted to validation in the IC design flow as shown in Figure 1.1 [2]. The current set of EDA tools which are responsible for validation at different levels in the IC design flow are lagging behind in performance issues due to large amount of circuit information they have to handle for the debug or backtrace operations.

Graph representations are frequently used in the field of Electronic Design Automation in modeling circuits. A graph is a mathematical structure that models relationships among items of a certain form. The abstraction of graphs often simplifies the formulation, analysis, and solution of a problem [3]. A graph primarily is a relation between two sets i) set of nodes,

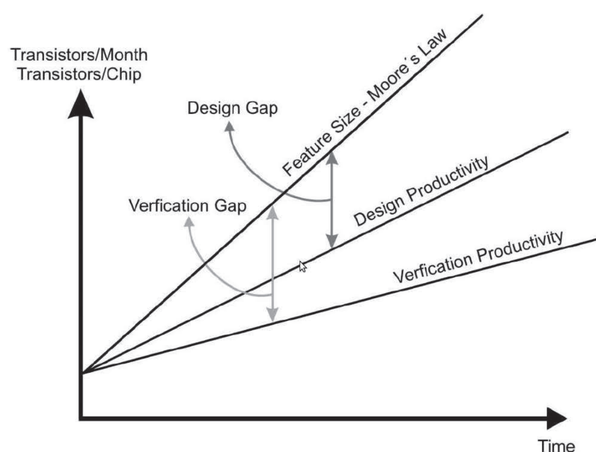


Figure 1.1: Verification Gap Vs Design Gap

and ii) set of edges. The nodes carry the circuit element information and the edges model the relationships among the nodes. This relationship can be structural or functional in nature. As we store the whole circuit in a graph structure, it usually serves as the backbone for most of the EDA tools specially validation/debug tools. Creating efficient graph structures and algorithms can play a huge part in covering the gap between the increasing design complexity and performance of EDA tools.

In this thesis, we pick two applications from pre- and post-silicon validation to exemplify our efficient graph techniques. They are as follows:

1.0.1 Scan Pattern Debugger for Partial Scan Designs

For the past several decades, Manufacturing Test has been a prominent methodology to screen and identify defective ICs. With shrinking sizes and increasing design complexity, defects have become harder to detect and diagnose, resulting in more test escapes. There are two main processes involved in manufacturing tests: The first is to generate high-quality ATPG (Automatic Test Pattern Generation) vectors as well as high-quality DTPG (Diag-

nostic Test Pattern Generation) vectors to achieve coverage and diagnostic goals [4]. Many state-of-the-art ATPG engines are available. The second is to enhance the testability of the CUT (Circuit Under Test) by inserting additional logic into it. Such techniques are referred to as DFT (Design For Testability) and usually aid in reducing the cost to detect those hard-to-detect faults. Scan DFT has been the most popular method as it enables a systematic methodology for making design changes to insert scan-cells, generate scan content, estimate/achieve test coverage goals, and recently diagnose process issues during Low Yield Analysis (LYA). This systematic methodology enables automation in the design flow, which is critical to meet the time to market (TTM) goals for any product.

Scan patterns are generated by an ATPG tool that operates on a gate level abstraction model. It is performed on smaller functional blocks to reduce to the complexity, but the interaction between various functional blocks is controlled by a user developed environment. The environment for the functional block – including clocks, interface signals, partially implemented blocks and initial sequential values – are typically specified as ATPG constraints. These ATPG constraints are based on design assumptions / specifications and are taken as inputs to the ATPG tool. Issues related to the ATPG gate model along with any other flow issues are tackled as soon as the First Silicon prototype of the IC is manufactured. It is critical to debug the Si failures and feed forward the fixes for the next Si re-spin.

We address the pattern cleaning issues in First Silicon, i.e., debugging the first-silicon failures and fixing the incorrect ATPG collateral assumptions made during pattern generation. We create an efficient *implication graph* based pattern debugger suite which analyzes the failures and help the debug engineer to root cause the failures. It should be noted that the goal is NOT to perform diagnosis on a large number of Silicon samples. The diagnosis is typically performed after the pattern debug step, i.e., after the patterns have been cleaned and stabilized. To perform First silicon debug of a pattern, we are given a set of silicon

failures (scan-cell values for a pattern that mismatch on silicon), the failing pattern and the ATPG environment used to generate the pattern. It should also be noted that the debug engineer's visibility is limited to the internal signal values in the ATPG model only and does not include all the corresponding values on silicon.

1.0.2 Circuit SAT Solvers and Bounded Model Checkers

Complexity of a verification task varies on the size of the functional block. Small functional blocks can be verified easily with existing state-of-the-art EDA tools. But as the design complexity and size increase, the time and memory complexities grow exponentially. Today's designs can consist of thousands to millions of state elements which give rise to a huge state space for a verification tool to search for an error. For example, a design with 30 state elements can already have more than one billion states, and checking for a certain property might require visiting each and every state. Handling the circuit and state information using reduced graph structures is one way to make verification tools more efficient.

We explore reducing the complexity of SAT solvers often used in formal verification. Unlike the Dynamic Simulation methods, where a verification engineer intuitively or randomly generates test stimuli and check for the functionality, this is a static method which uses formal methods. Dynamic functional simulation has been effective for decades and still covers a huge part of the verification process, but the increase in design complexity has enhances the chances of missing out corner cases and design errors whose detection requires traversing a huge state space. On the other hand, formal engines too have a huge bottleneck of managing memory and time resources. It has been seen that the SAT procedures and symbolic procedures (such as Binary Decision Diagram (BDD)) are prone to explosion in either run time or memory. But despite these inherent drawbacks, formal methods have

shown promise of verifying a design in its entirety which is unlike with dynamic simulation with today's complex designs.

In applications like circuit SAT and Bounded Model Checking where we unroll a circuit for a certain number of time frames, there is a lot of structural information that is repeated in each time frame. For formal applications using a SAT solver, the structural information of the circuit is converted to a CNF (conjunctive normal formula) formula. Hence, the CNF formula for an unrolled circuit contains a lot of redundant information. A SAT solver, which takes this CNF formula as its input, is oblivious to the circuit structure and hence operates on redundant information. Specially, in cases where memory explosion is a concern, managing the redundant information efficiently can be high priority.

A SAT solver inherently creates an implication graph [5] to help in its backtracking operation. In the second half of this thesis, we create a new graph structure for this implication graph which creates a more memory efficient circuit SAT solver and also give competitive run time results in case of big circuits.

Contributions of the thesis

1. We developed a complete Interactive Scan Pattern Debug Suite for industrial designs. The suite is currently being used for production of next generation micro-processors.

2. We propose a novel Debug Implication Graph (DIG) which is created dynamically during logic simulation of the failing scan patterns.

3. We propose a new failure analysis technique to root cause the Silicon failure using the DIG.

4. We investigate the use of DIG in SAT solvers with applications in Circuit SAT and

Bounded Model Checking. We propose a reduced graph data structure which helps in reducing the system memory usage with a minimal run time hit.

Organization of the thesis

The remainder of the thesis is organized as follows.

In Chapter 2, we go over the necessary basics and definitions that will be helpful to understand the rest of the thesis. It covers basic concepts and previous work for both our applications in scan pattern debugger and bounded model checkers.

In Chapter 3, we present our concept of a debug implication graph (DIG). We look into how it is generated dynamically during logic simulation. We look at an example from a partial scan setup. We also describe a new failure analysis technique to do failure analysis starting from any node in the graph.

In Chapter 4, we provide the details for our new Interactive Scan Pattern Debugger. We give a list of all the commands that were introduced and how do they integrate in the pattern debug flow. We provide with the results of the time saved for the debug engineer in cleaning the patterns for the first silicon in an industrial production environment.

In Chapter 5, we discuss the implementation of the DIG in a SAT solver setup being used for circuit SAT or Bounded Model Checking (BMC). We present a new data graph structure for the inherent implication graph in the SAT solver for circuit based problems. The results show a 90% reduction in the graph size and traversal has a very minimal run time overload.

Chapter 6 concludes the thesis.

Chapter 2

Background

This chapter introduces the preliminaries and related definitions used in context of our approach. It illustrates the terms partial scan, pattern debug, implication graph, conjunctive normal form (CNF) and unrolling of a circuit in detail, and summarizes the related previous works done in this area.

2.1 Design for Testability - Scan

Design for Testability (DFT) is an integral part of modern day integrated circuit (IC) production flow. Unlike decades ago where design and test were kept entirely separate and little thought was given on how to test the device during its design, the process of testing is considered early in the design flow today to make the final manufactured chips more testable. As a result, integration of design and test, referred to as design for testability (DFT), began in the 1970s [6].

The main challenge to structurally test a circuit was the ability to control and observe

internal lines in a circuit. In a sequential circuit, some nodes can be very difficult to control and observe due to the presence of state elements; for example, the most significant bit of an n-bit counter can only be controlled after 2^{n-1} clock cycles (assuming the counter starts in an all-0 initial state). The main idea in a scan design is to increase controllability and observability [7] for flip flops, which in turn can increase the testability in other nodes in the combinational portion of the circuit. This is achieved by replacing some or all of the state elements by scan cells [8] as shown in Figure 2.1. Additional I/O pins, including the scan-in (SI), scan-out (SO) and test mode (TM), are needed. TM acts as a select line for switching between the two modes: Test Modes and Functional/Normal Mode. In the functional mode, the scan cell behaves as the original storage element, while in the test mode, we can control each storage element by forcing external input and observing the output response (often via a scan shift operation). In large designs, the scan cells are arranged in one or more chains to reduce the number of extra I/O pins needed for each storage element. In a scan chain, scan cells are stitched together to form a chain by connecting the output of one cell to the input of next cell. This flow of controlled input stimuli along the scan chains is generated automatically by Automated Test Pattern Generator (ATPG) tools.

- Full Scan Design [9]: A design where all storage elements are selected for scan insertion. This makes all state elements completely controllable. Hence, a combinational ATPG engine is used to generate test patterns.

- Partial Scan Design [10][11]: A design where some storage elements are selected for scan insertion and sequential ATPG is applied.

A hardware overhead of about 10 to 15% is generally considered acceptable in ASIC chips today. In addition, there may be a performance penalty, which can be less than 5%. These costs are justified by the high quality of devices obtained in a short design time. Finally,

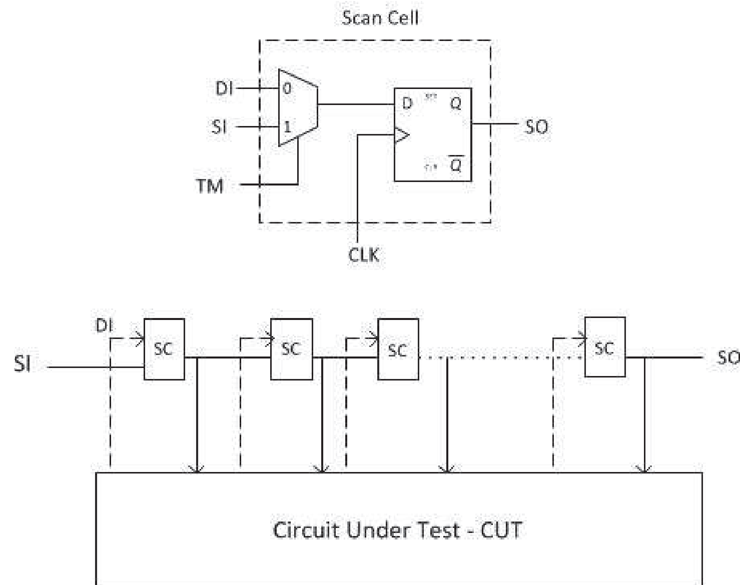


Figure 2.1: Scan Cell and Scan Chain

there is a test application time penalty, caused due to the slow scan shift clock in the scan chain. To overcome this, the concept of multiple parallel scan chains can be implemented. But there is tradeoff due to increase in SI and SO pin count.

2.2 Pattern Debug

A few well-established methods are used in the industry to address the flow issues encountered during First Silicon. This includes checking whether the device under test (DUT) is put in scan mode properly by checking the initialization sequence, using loop-back tests to check the content delivery path, using advanced tester features to probe the DUT and check if it is active, etc. To perform electrical validation and fine tune the voltage / frequency settings, the DUT is exercised at different voltage and frequency ranges on the tester and the Shmoo plots [12] are analyzed. We refer the reader to [13] for a comprehensive over-view of the flow validation techniques.

Initially, different flavors of chain/shift patterns are applied to the DUT. A series of bit values are shifted into the scan chain and then shifted out to validate the continuity of the chains. In case of chain test failures, d-port tests are used to identify the broken point in the scan chain. Further, several chain diagnosis techniques are available in literature[14][15] to debug the chain issues on silicon. Validating the flow and chain continuity gives us a high level of confidence to start applying the ATPG capture patterns to the DUT. Any silicon failures that are reported on the tester, after applying the capture patterns, need to be debugged.

In designs that are not full-scanned, checking the test flow is particularly important, since the initial states for the non-scan elements must be correctly assumed for the ATPG patterns to work properly. The first systematic debug approach for First silicon debug issues was proposed in [16] based on a fault injection approach. In this approach, the potential candidates for fault locations are identified using fan-in analysis from the failing scan-cells. A fault is injected on each of the candidates and the response is compared against the failing Si response. If the faulty response matches the failing Si response, then the score for the candidate is incremented. Finally, the debug engineer reviews the diagnosis scores for the potential candidates to fix the issue. The technique has been successful in identifying a set of potential candidates based on this analysis. It requires multiple fault simulations [17] and the performance is dependent on the number of potential candidates identified for analysis. It does not provide any transparency to the debug engineer into the internal signals for debug analysis to understand the issue and fix the failure.

Another approach to debug pattern mismatches was presented in [18], albeit, for debugging pattern mismatches during the pattern validation step and not silicon debug (please refer to Figure 1). The ATPG pattern is simulated with a timing model and the signals in the logic cone of the failing scan cell are dumped. Then the values of the signals in the fan-in cone of the failing scan cell are systematically compared between the ATPG simulation and the

timing simulator to determine the source of divergence. This approach is dependent on the internal signal values that are available only during the pattern validation step and not for silicon debug. It has significant potential to identify modeling and race issues pre-Si that can alleviate the First Silicon failures that might be otherwise encountered.

On the other hand, a scan pattern debugger is necessary to understand the root-cause and fix the failures in addition to identifying the potential candidates for Si failures / pattern mismatches. It can also serve complementary to the above approaches by providing visibility into the internal signals of the CUT in the ATPG model for fixing a Si failure. We use a logic simulation based method and construct an implication graph to build the scan pattern debugger. The event driven simulation technique is the most effective and efficient logic simulation due to the elegance of the selective trace approach (i.e., evaluating only the active components), together with its ability to easily handle asynchronous designs and timing analysis [19].

2.3 Static Logic Implication

When we assign a gate to a logic value '1' and '0', these values propagate throughout the circuit. These effects are called Static Logic Implications. Static logic implications relate two nodes of interest, and they can be divided into direct, indirect and extended backward implications. Indirect and extended backward implications are also called non-trivial [20] as they use logic simulation and contrapositive and transitive laws extensively[21].

- *transitive fanins:* All gates that drive gate N
- *transitive fanouts:* All gates that are driven by gate N

- *impl* [N, v, t]: Set of all implications resulting by assigning a value v to gate N in time frame t, where $v \in$ 0,1. For combinational circuits, t is equal to 0 and hence omitted.

- $(N, v) \rightarrow (M, w, t)$: Gate N when assigned a value v implies a value w on gate M in time frame t.

- *Contrapositive law:* If there exists $(N, v) \rightarrow (M, w, t)$, then contrapositive law states that $(M, w') \rightarrow (N, v', t)$ where w' and v' are complementary values of w and v respectively. The contrapositive law is applied repeatedly during extended backward implications.

- *Controlling value:* It is a logic value when at any fanin, is enough to determine the output of the gate. For example, in AND and NAND gates, when any one input is '0' then output is always '0' irrespective of the values at other inputs. Similarly for OR and NOR gates where the controlling value is '1'; any input is '1', the output will always be '1'.

- *Impossible/ constant nodes:* If there exists $(M, w) \rightarrow (N, v, t)$ and $(M, w) \rightarrow (N, v', t)$ or if $(M, w) \rightarrow (M, w')$ then (M, w) is impossible, i.e., gate M will never be able to acquire value w and would be a constant with value w'.

- *Transitive Law:* If there exists $(M, w) \rightarrow (N, v, t_1)$ AND $(N, v) \rightarrow (L, w, t_2)$, then the transitive law states $(M, w) \rightarrow (N, v, t_1 + t_2)$.

- *Conflicting Assignments:* If $(M, w) \rightarrow (N, v, t)$ AND $(M, w) \rightarrow (N, v', t)$, then (M, w) is an impossible setting. Therefore, M is a constant node holding the value w' permanently.

2.3.1 Direct Implications

Direct Implications of a Gate N are a set of implications to the gates that directly drive or are driven by N. They are based on controlling values of gates. The controlling values

h

Table 2.1: Controlling Values

Gate Type	Controlling Value	Non-Controlling Value	Controlling to
AND	0	1	0
NAND	0	1	1
OR	1	0	1
OR	1	0	0

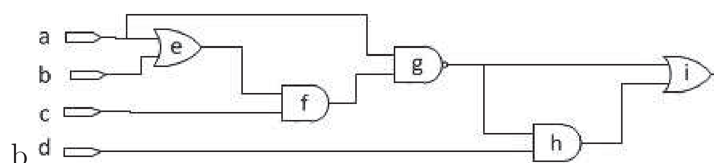


Figure 2.2: Example Circuit

for all 4 primary gates are shown in Table 2.1. Columns 1 and 2 show the controlling and non-controlling values which are by obvious nature complimentary to each other. The third column shows the output value when one of the inputs has a controlling value.

The direct implications are of two types: 1) forward implications, and 2) backward implications. Consider the example circuit in Figure.2.2. Here, e is an OR gate, g and h are NAND gates and f and i are AND gates. If we consider gate g and assert a value 0 on its output, the direct forward implications are $(i,\ 0)$ and $(h,\ 0)$. Similarly, the direct backward implications are $(f,\ 1)$ and $(a,\ 1)$. So combining forward and backward direct implications, the set of direct implications we have so far on gate g is $impl[g,0] = (g,\ 0),\ (i,\ 0),\ (h,\ 0),\ (f,\ 1),\ (a,\ 1)$.

Figure 2.3 shows an example how direct implications result into constant nodes. Here, $impl\ [c,0] = (c,\ 0),\ (a,\ 1),\ (b,\ 1), impl\ [b,\ 1] = (b,\ 1),\ (a,\ 0), impl\ [b,\ 0] = (b,\ 0),\ (a,\ 1),\ (c,\ 1)$, $impl\ [a,\ 1] = (a,\ 1),\ (b,\ 0), impl\ [a,\ 0] = (a,\ 0),\ (b,\ 1),\ (c,\ 1)$. If we take the transitive closure of implications of $(c\ ,0)$ we get $impl\ [c,\ 0] = (c,\ 0),\ (a,\ 1),\ (b,\ 0),\ (c,\ 1),\ (a,\ 0),\ (b,\ 1)$. Since $impl\ [c,\ 0]$ contains both $(a,\ 0)$ and $(a,\ 1)$, therefore $(c,\ 0)$ is an impossible assignment and c should be a constant with logic value 1.

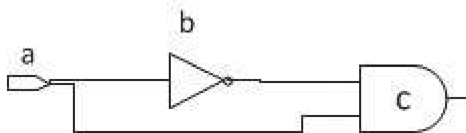


Figure 2.3: Example Circuit

2.3.2 Indirect Implications

In Figure 2.2, we see that $(g,\ 0)$ or $(h,\ 0)$ individually do not imply anything on gate i. However, together they imply $(i,\ 0)$. Since $(g,\ 0)$ and $(h,\ 0) \in impl\ [g,\ 0]$, therefore $(g,\ 0) \rightarrow (i,\ 0)$. This is an indirect implication and can be computed by logic simulating the direct implication list of $impl\ [g,\ 0]$. We can formulate the above statement mathematically as $impl\ [N,\ v] \equiv impl\ [N,\ v] \cup [LogicSimulate(impl\ [N,\ v])]$ where $LogicSimulate()$ refers to performing logic simulation with direct implications asserted on the gates. They are also termed as global and non-local implications in [22] [23] and used them in combinational test generation in a SAT framework.

2.3.3 Extended Backward Implications

The concept of extended backward implications was first introduced by Zhao *etal*. in [24]. The extended backward implications are computed by considering both the target gate N and the unjustified output specified gates in the implication list of the target gate. There are generally four cases depending on the gate type of the gate where the target signal is the output:

Let us consider gate N has p inputs, among which m inputs $(l_1, ...l_m)$ are unspecified.

- **AND Gate**

If N is an AND gate, then $impl\ [N,\ v] \equiv impl\ [N,\ v]$

$\cup\ [\cap_{i=1}^{m}\ logic\ simulate\ (impl\ [N, v] \cup impl[l_i, 0])]$

The above formulations states that if the implications set of gate N with value v contains a AND gate M which has a unjustified output specified(i.e. it has an output value 0 which is not determined by the value at its fanins), then the common set of implications obtained by logic simulation after setting every unspecified input of N to 0 and setting all other implications under $impl[N, 0]$. The common set of implications is the group of extended backward implications.

- **OR Gate**

  If N is an OR gate, then $impl\ [N,\ v] \equiv impl\ [N,\ v]$

  $\cup\ [\cap_{i=1}^{m}\ logic\ simulate\ (impl\ [N, v] \cup impl[l_i, 1])]$

  The above formulations states that if the implications set of gate N with value v contains a OR gate M which has a unjustified output specified(i.e. it has an output value 1 which is not determined by the value at its fanins), then the common set of implications obtained by logic simulation after setting every unspecified input of N to 1 and setting all other implications under $impl[N, 1]$. The common set of implications is the group of extended backward implications.

  Extended Backward Implications can be computed for NAND and NOR in the same way as shown above.

- **XOR Gate**

  If M is a 2-input XOR gate and $(M, 1) \in impl[N, v]$ then $impl[N, v] \equiv impl[N, v] \cup$ $logic\ simulate(impl[N, v] \cup impl[l_0, 1]$

  $\cup\ impl[l_1, 0]) \cap logic\ simulate(impl[N, v] \cup impl[l_0, 0] \cup impl[l_1, 1])$

  The above formulations states that if the implications set of gate N with value v contains

a XOR gate M which has a unjustified output specified(i.e. it has an output value 1 which is not determined by the value at its fanins), then the common set of implications obtained by logic simulation after setting its two fanins to 1 and 0 and then to 0 and 1, respectively and setting all other implications under $impl[N, v]$. The common set of implications is added to $impl[N, v]$ as extended backward implications.

If M is a 2-input XOR gate and $(M, 0) \in impl[N, v]$ then $impl[N, v] \equiv impl[N, v] \cup$ $logic\ simulate(impl[N, v] \cup impl[l_0, 1]$
$\cup\ impl[l_1, 1]) \cap logic\ simulate(impl[N, v] \cup impl[l_0, 0] \cup impl[l_1, 0])$

The above formulations states that if the implications set of gate N with value v contains a XOR gate M which has a unjustified output specified(i.e. it has an output value 1 which is not determined by the value at its fanins), then the common set of implications obtained by logic simulation after setting its two fanins to 1 and 1 and then to 0 and 0, respectively and setting all other implications under $impl[N, v]$. The common set of implications is added to $impl[N, v]$ as extended backward implications.

- **XNOR Gate**

  If M is a 2-input XNOR gate and $(M, 1) \in impl[N, v]$ then $impl[N, v] \equiv impl[N, v] \cup$ $logic\ simulate(impl[N, v] \cup impl[l_0, 0]$
  $\cup\ impl[l_1, 0]) \cap logic\ simulate(impl[N, v] \cup impl[l_0, 1] \cup impl[l_1, 1])$

  The above formulations states that if the implications set of gate N with value v contains a XNOR gate M which has a unjustified output specified(i.e. it has an output value 1 which is not determined by the value at its fanins), then the common set of implications obtained by logic simulation after setting its two fanins to 0 and 0 and then to 1 and 1, respectively and setting all other implications under $impl[N, v]$. The common set of implications is added to $impl[N, v]$ as extended backward implications.

If $(M, 0) \in impl[N, v]$ then $impl[N, v] \equiv impl[N, v] \cup logic\ simulate(impl[N, v] \cup impl[l_0, 0]$ $\cup\ impl[l_1, 1]) \cap logic\ simulate(impl[N, v] \cup impl[l_0, 1] \cup impl[l_1, 0])$

The above formulations states that if the implications set of gate N with value v contains a XNOR gate M which has a unjustified output specified(i.e. it has an output value 1 which is not determined by the value at its fanins), then the common set of implications obtained by logic simulation after setting its two fanins to 1 and 0 and then to 0 and 1, respectively and setting all other implications under $impl[N, v]$. The common set of implications is added to $impl[N, v]$ as extended backward implications.

2.3.4 Implication Graph

The logic implications are stored in a graph data structure using a directed graph $G(V, E)$ where $V(vertices) \in\ the\ set\ of\ 2 * N\ nodes$ (N = Number of gates in the circuit) corresponding to both value assignments (0 and 1) and $E(edges) \in\ single - node\ implications$. The weight associated with an edge represents the relative time frame associated with the implication. For example, consider an AND gate and its implication graph shown in Figure 2.4. The AND gate has 3 signals and therefore 6 nodes (for both 0 and 1). An edge in the graph showing the implication relation. The implicant and implication are stored as nodes with arrows pointing from implicant to the implication in the implication graph. We refer the reader to[25, 26] for details on the implication graph and some of its applications in the Test domain. The implication graph provides a very strong data model to keep track of the simulation values and root-cause.

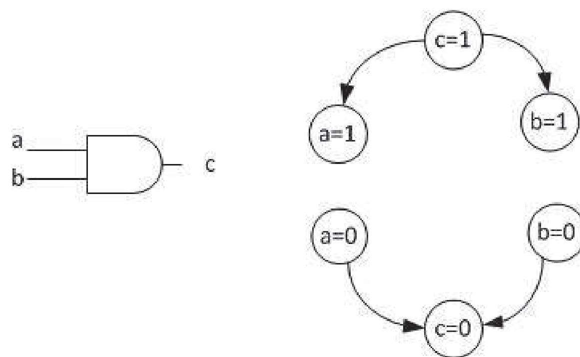


Figure 2.4: AND gate and its Implication Graph

2.4 Boolean Satisfiability - SAT

A Boolean Satisfiability (SAT) problem can be formulated as follows. Given a Boolean formula F where is F is a function of variables $x_1, x_1, x_1, ...x_1$, determine a variable set V for which F is true (1) or prove that no such V exists. It is the first NP- complete problem [27]. Most of the state of the art SAT solvers today are based on the Conjunctive Normal Form (CNF). This form consists of conjunction (AND) of one or more clauses. A *clause* is a disjunction (OR) of one or more literals. A *literal* is a Boolean variable of F in its true or complement form.

All modern day deterministic SAT solvers[28–30] are descendants of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [31]. The base algorithm picks a variable, take a decision of setting it either 0 or 1. Then propagate this information on all other clauses, also known as Binary Constraint Propagation (BCP). If any clause evaluates to 0 then backtrack the decision on the variable. Repeat this step until all the clauses get satisfied or a satisfying assignment is not found.

Boolean Constraint Propagation (BCP) [28] is a process in which we get the implications

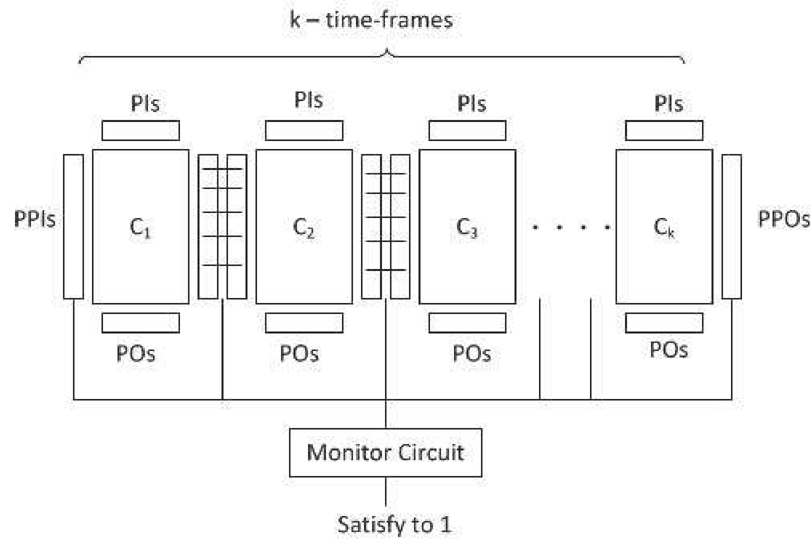


Figure 2.5: Unrolling and Monitor circuit for Bounded Model Checking

from the unit clauses (a clause with one unassigned literal). The one literal left have to be assigned true for the SAT solver to move forward. For example the CNF formula -

$(\neg a \vee b \vee \neg c)(a \vee \neg c)(a \vee d)(\neg a \vee b \vee \neg d)$.

If we set a = 0, then we get

$(1)(\neg c)(\vee d)(1)$.

We have two unit clauses which will force us to set c=0 and d=1. Hence, $a \rightarrow \neg c$ and $a \rightarrow d$ and they will added to the implication graph.

2.5 Circuit Unrolling for SAT based Bounded Model Checking (BMC)

Bounded Model Checking (BMC) refers to an algorithm which searches the state space of a finite transition system to determine if a certain property hold in the bounded time length of k.

To perform BMC in a bounded length k, the sequential circuit first needs to be unrolled into k time-frames. The unrolled circuit as shown in Figure 2.5. The initial and final states of the unrolled circuit consist of Pseudo Primary Inputs (PPIs) and Pseudo Primary Outputs (PPOs). The flip flop signals which serve as the current state (CS) elements are treated as PPIs and the output of the flip flops which serve as the next state (NS) elements. The rest of the circuit is just the copy of the original circuit C and connecting the nth frame's PPOs to the $(n+1)^{th}$ frame's PPIs. This repeating circuit structure is used extensively in Chapter 5 of this thesis. After unrolling sequentially, a *Monitor Circuit* [32] is created to carry out BMC of the property to be verified. A CNF database is created We have used of the unrolled circuit along with the monitor and the SAT solver is asked to satisfy the monitor circuit output to 1.

There are two kind of properties that are usually verified using Bounded Model Checking which belong to the CTL class [33]

- Safety Property: it specifies that *something bad will never happen*. In other words, this property should never be violated by a design during its proper functioning. In CTL formula, it is represented as $AG(p)$. It means that starting from an initial state there does not exist a path that leads to state where p is falsified. In usual BMC practice,

we check for $\neg EF(\neg p)$. If it is satisfied, a counter-example is generated, implying that the design has violated p.

- Liveness Property : it specifies that *something good will eventually happen*, i.e., the target property will eventually come true for the design. In CTL formula, it is represented as $AF(p)$. It means that starting from an initial state, all paths must hold p sometime in the future. In usual BMC practice, we denote, we check for $\neg EG(\neg p)$ which if true would imply that along some path for all future states p is not reached.

The efficiency of BMC depends on the property to be verified as well as the underlying SAT solver. In our work, we use the unrolling information and improve the efficiency of the SAT solver via our new implication constructs, thereby reducing the memory requirements.

Chapter 3

Debug Implication Graph

A Debug Implication Graph (DIG) is a dynamically constructed and optimized implication graph between multiple time frames. The primary purpose of an implication graph is to keep track of all functional relationships (implications) for a particular process [5]. This gives the power for the engineer to back trace any conflicting implications to the originating decisions.

In a partial scan circuit, only a subset of the storage elements is converted to scan cells. The resultant partial scan circuit can be unrolled in a similar manner as described in Section 2.5., but with only non-scan flip flops acting as connecting elements within adjoining time frames. This unrolled circuit is used extensively for both pattern diagnosis and BMC enhancements in the rest of the thesis.

We create a DIG for our pattern debugger during logic simulating the failing patterns. We use an event-driven logic simulation engine. An event driven logic simulation engine takes advantage of the functional redundancies due to the unrolling of a sequential circuit. For example: If the predecessors (immediate inputs) of a gate g keeps the same value from the previous time-frame, then g will not need to be re-scheduled for evaluation. This contributes

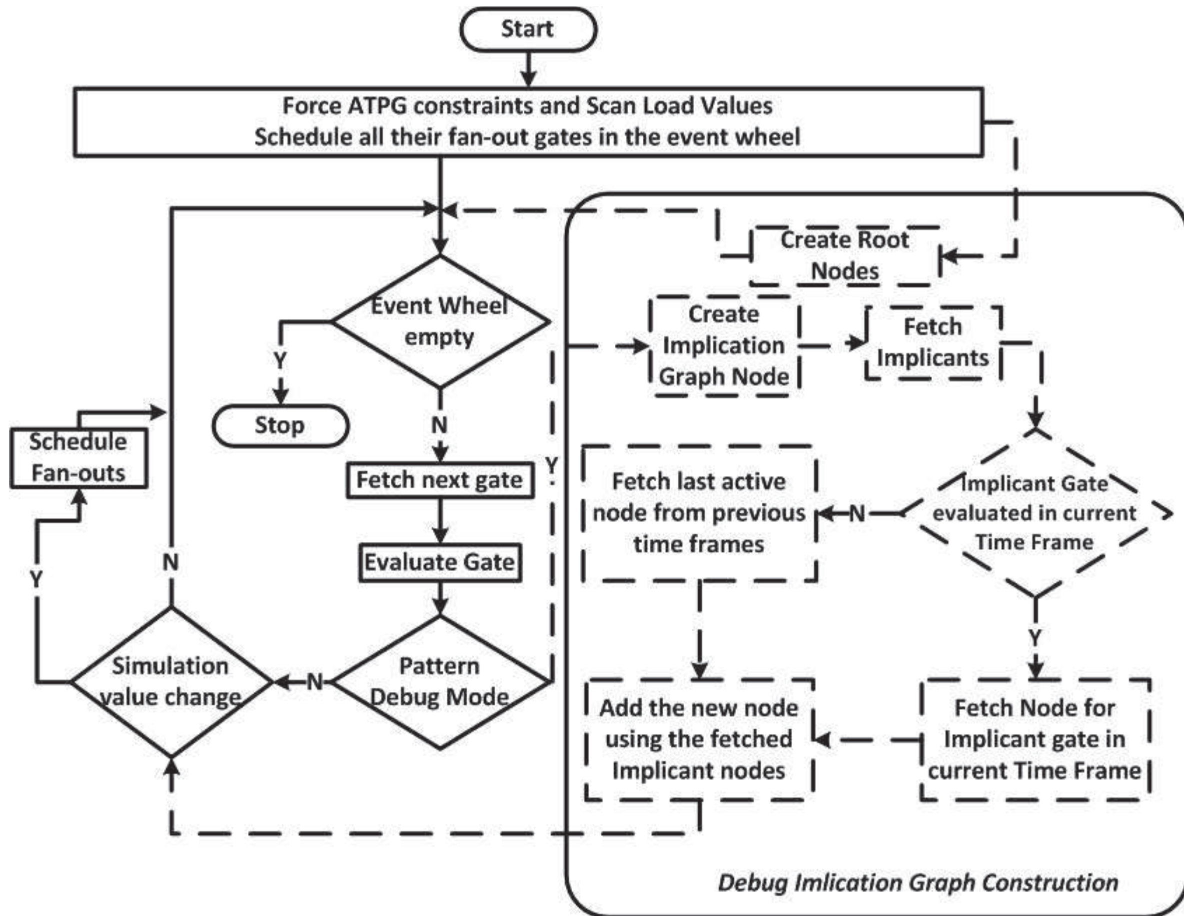


Figure 3.1: DIG Construction

to significant performance advantage for a logic simulator. This is a critical performance incentive as simulation consists of 70% of the validation time in the silicon bring up cycle mainly a featuring in functional verification[34].

3.1 DIG Construction

In Figure 3.1, the blocks outside the solid box correspond to the base line logic simulator. For each test vector, we first start with assigning the set of all scan load values, constrained pins, clocks, internal constraints and latch initialization values. We then schedule their

immediate successors in an event wheel. Each gate g in the event-wheel is evaluated and its successors are scheduled only if g's value changes. A subtle, but important, fact is that the logic simulator holds this property across time frames while simulating a pattern as well. We dynamically construct the DIG during logic simulation without disrupting these fundamental properties to keep the performance benefits of event driven simulation.

The construction of DIG during event-driven logic simulation is shown in dotted lines inside the box in Figure 3.1. We start the implication graph construction by creating the root nodes (terminal nodes with no input edges) for each of the scan load values, constrained pins, clocks, internal constraints and latch initialization values. After each gate is evaluated, we create an implication node for the gate. The gate ID and time frame for the test cycle are stored in each graph node. The implicant nodes, created in the appropriate time frames, are added as fan-ins to the implication node to represent the implication relation for the gate value. It should be noted that the implicant information can be fetched during gate evaluation depending on the gate type.

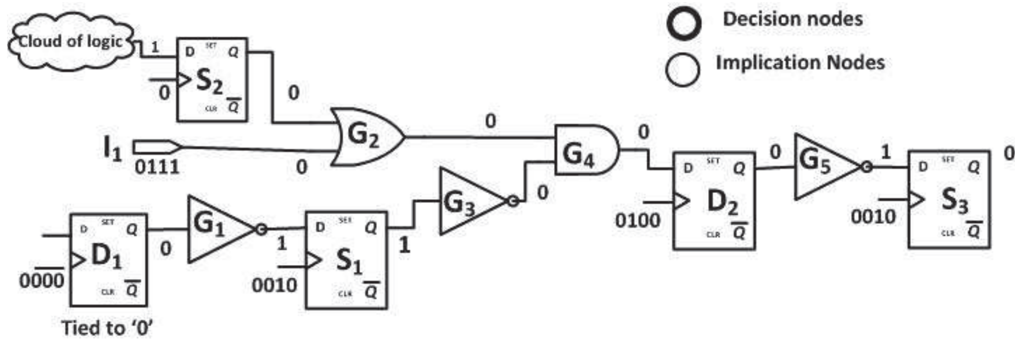
We illustrate the implication graph construction with a circuit fragment shown in Figure 3.2a, for a test vector with a test cycle width of 4 time frames. S_1, S_2 and S_3 are scan latches with a capture clock waveform 0010, indicating that they are captured only in the third time-frame. D_1 and D_2 are non-scan latches that are initialized to 0. D_1 and D_2 get capture clock waveform 0000 and 0100 respectively. D_1 get 0000 at clock as it is *tied to '1'* due to external ATPG constraint and is constant for the whole test procedure. The primary input I_1 is constrained to 0111 for the four time-frames. The values for all gates corresponding to the four Frames from 0 to 3 are shown in the Figure 3.2a from left to right. We use the following convention in the Debug Implication Graph for ease of explanation: (i) the root nodes are represented by thick circles and the implied nodes are represented by thinner circles. (ii) Each graph node has the gate name inside it and the time frame is

denoted by the super-script. The nodes are represented by node numbers in order of gate evaluation / node creation.

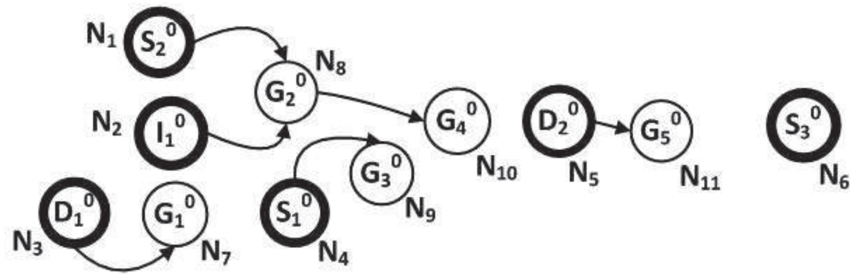
The Debug Implication Graph constructed in time-frame 0 is shown in Figure 3.2a. Initially, we create the root nodes for the scan load values at S_1, S_2 and S_3, non-scan initialization D_2, ATPG constraint *tied to '1'* at D_1 and, primary input value I_1. These are represented by nodes N_1 - N_6 in the graph. During logic simulation, we create the implication nodes for the remaining gates. For the sake of explanation, let us consider evaluation of gates G_2 and G_4. The value G_2=0 is implied by I_1=0 and S_2=0. We create a graph node N_8 (G_2=0) and add the implicant nodes N_1 and N_2 as fan-ins. At G_4^0, either G_3^0=0 or G_2^0=0 can imply the value since it is the controlling value of an AND gate. We can pick either of them without loss of generality. We pick G_2^0 in this case and construct N_{10}. In this way, all graph nodes are created for time-frame 0 as shown in Figure 3.2a.

In time-frame 1, the primary input I_1 changes from 0 to 1. Correspondingly, the graph nodes N_{12} and N_{13} are created in the DIG due to evaluation of I_1 and G_2, as shown in Figure 3.2b. The value at gate G_4 remains at 0, however, it is now implied by G_3 and not G_2 anymore. This evaluation of G_4 creates a new graph node, N_{14}, for G_4^1 and the graph node from the most recent frame for G_3, i.e. N_9, is added as its implicant. Next gate to be evaluated is D_2 and it is scheduled because of the capture clock value being triggered for D_2 in Frame 1 (and not because of G_4). D_2 was initialized to 0 with graph node N_5 as a root node initially. Nevertheless, a new graph node is created for D_2 in Frame 1 with G_4^1 as implicant to reflect the latest implicant (the clock graph node is skipped for brevity). It should be noted we include at-least one implicant as a fan-in to the graph node that is sufficient to explain the gate value in the circuit.

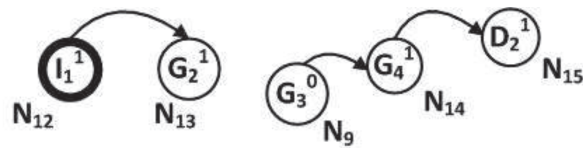
In time-frame 2, the capture clocks are fired to capture the functional values into the three scan cells, S_1, S_2, and S_3. First, the clock values for S_1, S_2 and S_3 change. Then, the scan



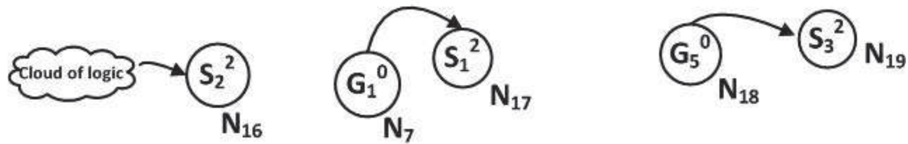
Circuit Diagram
-- Internal signal values shown are for the initial time frame 0.
-- Clocks and Primary Input values are shown for all time frames.



Time Frame 0



Time Frame 1



Time Frame 2

Figure 3.2: Frame-wise DIG Construction

Figure 3.3: Event Driven Simulation does not Keep Track of the Latest Reason

cells are scheduled and evaluated. The corresponding implication graph nodes are created in the DIG. It should be noted that the graph nodes for initial load values are created as root nodes in Frame 0 and the final capture values are created as implication nodes in the capture frame 2. The nodes created are shown in Figure 3.2b.

Frame 3 is the unload frame and there is no activity in that frame. None of the gates are evaluated and no new graph nodes are created. The final DIG is shown in Figure 3.4. It can be noted that there are discontinuities in the graph (shown in dotted circles) due to the evaluation of a gate to same value in different time frames. This is a characteristic of the event driven simulation to avoid re-simulating the fan-out cone of those gates. It will keep the performance efficiency of our framework and will have an increased impact on multiple capture or sequentially deep test vectors.

For DIG construction, the ATPG constraints and latch initialization will be the same across all test vectors and the corresponding graph nodes and its implications may be created only once. The scan-cell graph nodes and their implications will change dynamically for every test cycle, as the scan-cell value changes, and will be updated dynamically as we progress across time frames during logic simulation of the pattern.

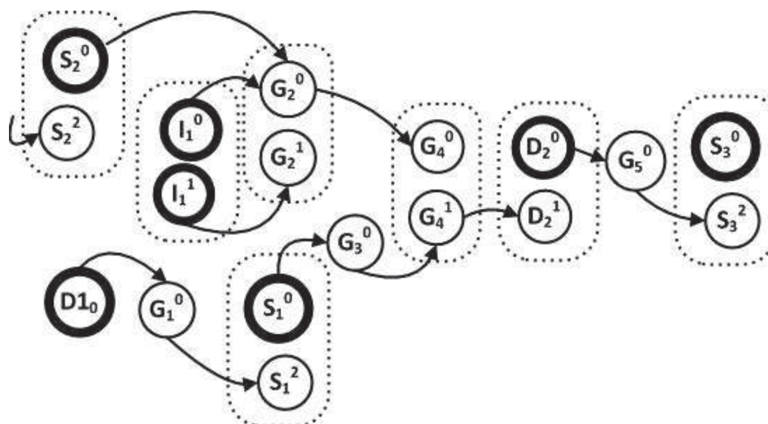


Figure 3.4: Final Graph

3.2 Backward Traversal

Consider the circuit shown in Figure 3.3. The inputs to the AND gate g change between time-frames 1 and 2 but the output remains constant. Event driven logic simulation during construction of DIG forces us to keep the reasons the same if there is no change in the gate's value. Hence, the NOT gate n in the fanout of g does not get evaluated in time-frame 2. If we try and find out the reason for the value '0' at n we get the a as the reason from time-frame 1, which is not correct. In our further discussion, we demonstrate a backtraversal algorithm to get correct reason for any gate value in our DIG setup.

To recall from our previous discussion, a scan-cell is termed as failing when the unload value on silicon is not the same as the expected one. There are a few special cases when we have 'X' (don't care) situations. If either one of the expected value or the real value is 'X', that scan cell can never fail for that particular unload frame / pattern. For a given gate (g) with value (v) in time frame (t), at least one graph node g_i s.t. i $\in$ (0,t) will be created in the DIG while evaluating the gate. Otherwise, the gate would not have been assigned a value to start with. It should be noted that a gate could have been evaluated multiple times in

different time frames and different nodes will be created as shown in the dotted circles in Figure 3.4.

We define a function R(x,y) which gives the reason graph node from the DIG for the logic value at gate x at time-frame y. So, the actual graph node responsible for the value at gate g in time-frame t, R(g,t), is g_i; s.t i = max(0,t) and $g_i \in$ DIG. For example: R(S1,0) = S_1^0 and R(S1,3) = S_1^2 represents the difference between scan load and unload values on S_1.

To find the root-cause of a gate value in a particular frame, we can perform a smart backward traversal from the corresponding implication graph node, R(g,t), to the root nodes in DIG as shown in Equation 3.1.

$$Root\ Cause\ (g,t) = \begin{cases} R(g,t), & \text{if R(g,t) is a root node.} \\ Root\ Cause\ of\ R(fanins(g,t)), & \text{otherwise.} \end{cases} \tag{3.1}$$

When we reach a graph node during backward traversal, we should update to the latest graph node, R(g,t) and the traversal time frame (T) for further traversal. These scenarios are shown by dotted circles in Figure 3.4. For example, let us say, we need to root-cause the value on S_3 in the unload frame. We will start with R (S3, 3) = S_3^2, and update the traversal time frame to 2. During backward traversal in DIG, we will do similar update at D_2 when we reach D_2^0 by updating R(D_2, 2) = D_2^1 and updating the traversal time frame to 1. In this way we can reach S_1^0 as the root-cause for unloading a value 1 on S_3. Note that we will not back-trace further since R(S1, 1) = S_1^0 is a root node. It should be noted that the DIG can be used to debug any gate value during simulation and is not limited to scan-cell unload values. Algorithm 3.1 describes the complete procedure.

Algorithm 3.1 Algorithm for Backward Traversal DIG

Initialize $failing_scan - cell = a$
$Traversal\ time - frame\ (T)\ \leftarrow$ Unload time-frame of a
$n\ \leftarrow$ last node created for the value at T
if $n.time - frame()\ <\ T$ **then**
  $T \leftarrow n.time - frame()$
end if
$DFS\ starting\ from\ n$
for every fanin node F_n in the fanin list of n **do**
  if $F_n = rootnode$ **then**
    add to the list of reasons : report the gate value and time-frame
  else
    search for the node that was last created till T
    $f_k \leftarrow$ *node that was last created till* T
  end if
  if $f_k \rightarrow time - frame < T$ **then**
    $T \leftarrow f_k \rightarrow time - frame$
  end if
end for

3.3 Summary

In this chapter, we explained our concept of a Debug Implication Graph. The graph is dynamically constructed in parallel with event driven logic simulation process. The graph creates the minimum number of nodes during its construction with the help of event driven properties of the logic simulation engine. During the backtraversal / reasoning process we keep a global time parameter which decides the latest (in terms of time frames) implication node for a gate that can be responsible for the value in one of its fanout gates. In the next chapter we discuss the complete implementation of the Pattern Debug Suite having the DIG as its backbone. It also provides us with the implementation results of the DIG on the debugger application in the industry production environment.

Chapter 4

Interactive Scan Pattern Debugger

4.1 Introduction

Scan patterns are typically generated by an Automatic Test Pattern Generation (ATPG) tool that operates on a gate level abstraction model of the design. ATPG is performed on smaller functional blocks, in the design, to alleviate the test generation complexity. The environment for the functional block, such as clocks, interface signals, partially implemented blocks and initial sequential values, are typically specified as ATPG constraints. These ATPG constraints are based on design assumptions / specifications and are taken as inputs to the ATPG tool.

The scan patterns are typically validated pre-Silicon by simulating them on a transistor level model that is supposed to be closer to Silicon behavior than the ATPG gate level model. Any issues related to ATPG modeling are most likely to be caught in this validation step. On the other hand, the constraints and sequential value initialization are usually the same as used during pattern generation to keep the environment in validation and pattern generation the

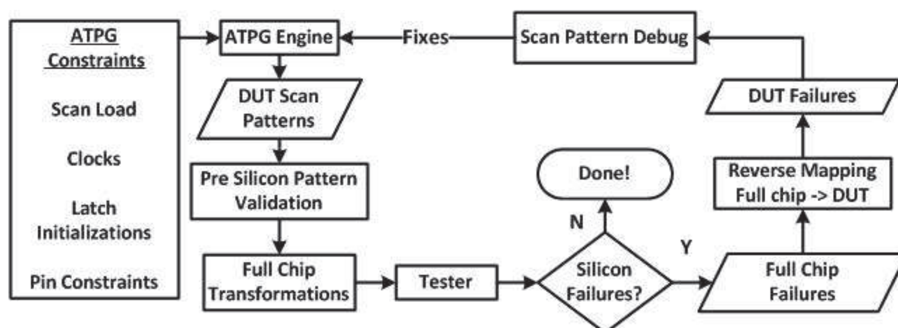


Figure 4.1: Scan Pattern Delivery Flow

same. Any scan-cell mismatches during pattern validation is debugged with the complete information of values of all internal signals in the gate level (comparison) and transistor level (reference) netlist. The scan pattern delivery/ debug flow is shown in Figure 4.1.

Once the design tapes out and First Silicon arrives, there is a rush to apply the scan patterns on the tester and debug any silicon (Si) mismatches/ failures as soon as possible. Scan Pattern Mismatches can be caused due to many reasons [35, 36]. Only a few samples are available during First Silicon to flush out the flow and clean any pattern issues. It is critical to debug the Silicon failures and feed forward the fixes for the next Silicon re-spin. First Silicon issues can be broadly classified into two categories: (i) Flow issues in applying the pattern on the tester and (ii) ATPG collateral issues in generating the pattern. The flow issues could arise from pattern conversion from Device Under Test (DUT) level to full chip level, voltage/frequency setting, content delivery path from full chip to DUT, etc. A few established methods exist in the industry to resolve the flow issues using tester utilities. The ATPG collateral issues could arise from incorrect clock/ constraint assumptions, incorrect sequential initialization, modeling issues, etc.

In this work, we are focus on addressing the pattern issues in First Silicon, i.e., debugging the Silicon failures and fixing the incorrect ATPG collateral assumptions made during pattern generation. It should be noted that the goal is NOT to perform diagnosis on a large number

of Silicon samples. The diagnosis is typically performed after the pattern debug step, i.e., after the patterns have been cleaned and stabilized. It should also be noted that the debug engineers visibility is limited to the internal signal values in the ATPG model only and does not include all the corresponding values on silicon.

4.2 Motivation

Presently in the industry, scan pattern debug is a manual process. The test engineer writes multiple handy scripts to analyze the mass failure information, then analyzes the fan-in cone of a selected failing scan cell and suggests a debug fix based on his design knowledge. After analyzing the failure, there is a post-processing step to confirm that the reasons listed are correct and the learning is used to fix the design constraints. Since the TTM (time to market) is very critical in today's economy, there is need for a tool to assist the test engineer in this whole process and reduce the throughput time for the whole pattern debug process.

We use the debug implication graph (DIG) to create an Interactive Pattern Debug Environment (IDE) for the debug engineer. We have developed a set of interactive commands that are suited to debug the First Silicon failures viz. root-cause the silicon failures, understand the pattern behavior in CUT and clean the patterns. Typically, First Silicon debug consists of two high-level steps:

1. Select a silicon failure
2. Debug the silicon failure

4.3 Select a Silicon Failure

The debug engineer needs to first select a failure, i.e., a scan-cell that fails on silicon and the corresponding test vector where it fails. We refer to the <scan-cell, test vector> pair as a silicon failure. As a rule of thumb, we should select a scan-cell that is representative of the majority of silicon failures and that has the least support-set / fan-in cone logic. This will enable fixing a majority of the issues when the selected scan-cell is debugged and make the debug easier. To select a test vector for debug, we need to pick the test vector where the selected scan-cell fails for the first time. This ensures that the chosen scan-cell remains un-affected by a majority of other failures. To select a failure, we developed a set of simple and effective interactive debug commands:

- *failure scan cell rank* : Rank failing scan-cells by number of times they fail. The best target are those scan cells, which are failing for most of the vectors and in every testing condition such as temperature, frequency, etc.

- *failure scan cell support set rank* : Rank failing scan-cells by size of fan-in logic cone (size = number of levels till the next sequential element) with preference for logic cones ending at scan-cells than non-scan cells. This is done to reduce the test engineers effort in verifying the reasons pointed out by the tool.

- *failure scan cell group* : Group scan-cells with similar names and rank them based on the above two criteria. The term similar names stands for same block or same bus, etc. This is because in the industry the signal and gate names have a directory like structure. For example : $function/module/block/sub-block/bus[1]$

- *failure scan cell pattern <scan cell id>* : List and sort all failing test vectors for a scan-cell. This is to know which patterns the scan cell is failing for which vector and

in which order. It also displays the load and unload values for the scan cell in each failing vector pattern. This helps the engineer to quickly analyze any patterns. For example, if the scan cell is failing only for the unload value '0'/'1'. There might be a port issue on the tester for that scan cell.

To select a failing scan-cell, we use a combination of group and rank metrics, with a heuristic to (i) select a failing scan-cell that has many other similar failures, (ii) has many failures cumulatively as a group and (iii) has a smaller support set. The first test vector for the selected scan-cell is picked using command 4 (*failure scan cell pattern <scan cell id>*). It has been our experience that fixing the selected failure, using the group and rank metrics, can resolve most of the issues related to the group. A common fanin-cone based grouping would also be a viable alternative in place of a name-based grouping mechanism. In some cases, it is possible that the test engineer gives inputs to debug specific failures based on design / tester feedback. In that case, we can by-pass this step and directly start with debugging the silicon failure.

4.4 Debug the Silicon Failure

The scan-cell failure debug is generally split into debugging the clock path and data path. We first start with clock path debug and then proceed to the data path debug.

1. Clock path failure debug: The clock path is expected to be the source of the failure; if all scan-cells driven by the same clock source and expecting different load / unload values fail on silicon [2]. The list of commands is below.

    - *failure clock path <scan cell id>* : This command fetches the clock source of the scan cell from the netlist information and lists out all the scan cells being fed

by the clock. It then analyses the failure information, and reports the number of failures for the four LOAD-UNLOAD(LU) combinations (00,01,11,10). If the readings are completely biased, i.e., all the failures are reported for LU = 01 and LU = 10, then there is a need to check if the capture clock is firing properly or not.

This analysis is done interactively on the selected scan cell. In addition to this analysis, the IDE enables us to query the DIG and analyze the expected clock values in the clock logic systematically to root-cause the potential cause of the clock not firing, understand the internal values of the clock network and fix any issue. We developed two interactive commands: (i) to analyze failing scan cells with similar clock sources and (ii) to query the internal values of CUT in DIG.

2. Data path failure debug: The data path debug is premised on two interactive commands:

   - *failure logic simulate <pattern number>* : The failing test pattern is simulated till the selected vector (pattern number). The DIG construction is triggered for the selected test vector, using the mechanism in Section 3.2, and skipped for the previous test vectors by default. Once the logic simulation is finished, the engineer can then step for whole test cycle or choose to step one clock cycle at a time. In this manner, the engineer can see the changes occurring at every step of the the failing vector by looking into the schematic view.
   - *failure debug <gate id, test cycle frame>* : This commands takes gate ID and the test cycle frame as its inputs. Since the engineer uses this command in conjunction with the *failure logic simulate command*, it can report the root-cause for values of any gate (not only scan cells) for any clock cycle. This flexibility is seen because

Table 4.1: Industrial Data for Pattern Debugger

D	Size	Root-Cause reported	Debug Fix	Debug Time
D1	711K	Hold Latches and Primary Inputs	Hold Latch from non-scan blocks	3.5
D2	3.4M	Latch Initialization	Initialization Sequence	1.5
D3	6.2M	Latch Initialization and Design Constraints	Incorrect Latch Initialization	3.5
D4	9.8M	Scan Loads and Latch Initialization	Incorrect Latch Initialization	6.5
D5	1.1M	Primary Inputs and Latch Initialization	Primary Input Values from different CUT instance	2.0

TV: Test Vector

of the DIG smart back traversal scheme presented in Section 3.2. The feature gives the user a chance to compare the reasons for value on any gate for two consecutive time frames in a test cycle.

4.5 Pattern Debug Flow

We typically start the debug with the failing scan-cell values in the unload frame. It is this unload value that is a mismatch on silicon. The tools allows us to traverse from the scan-cell node to the root nodes that caused the expected scan-cell value. These root-nodes indicate the constraints, scan load values or initialization latches that lead to the failure. These directly translate to incorrect ATPG constraints, modeling issues (scan load values are confirmed with chain tests earlier) or incorrect initialization sequence that have to be fixed to clean the patterns. The impact of a debugger, in addition to all previous techniques, enables us to understand the root-cause by querying the DIG for internal CUT values and fixing the actual issue. The complete flow is summarized in Figure 4.2.

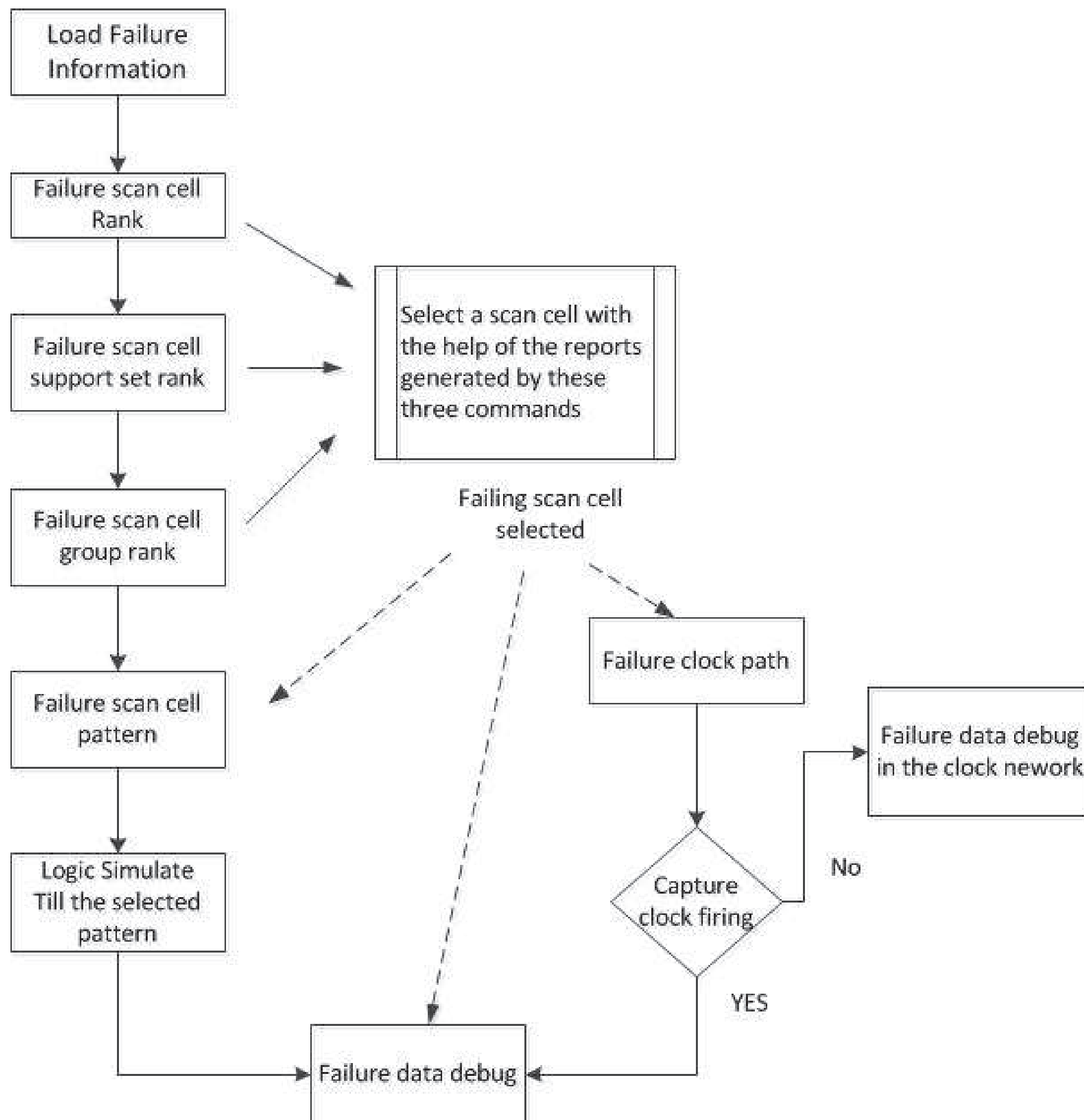


Figure 4.2: Interactive Pattern Debug Flow

4.6 Industrial Data

The proposed techniques for constructing a Debug Implication Graph (DIG) and the Interactive Pattern Debug Environment (IDE) were implemented in an industrial sequential logic simulator that is part of an ATPG tool. The scan pattern debugger was integrated into the scan ATPG flow for pattern debug in industrial micro-processor designs in a production environment. The data for these microprocessor designs and the impact of the scan pattern debugger is provided in Table 4.1. Since it was deployed in a production environment, we do not have any comparison against the other techniques, but it should be noted that the pattern debugger would be complimentary to any other root-cause identification techniques, or manual techniques, based on design knowledge or tester feedback, for understanding the CUT internal values and making the fix.

In Table 4.1, Column 1 lists the design, Column 2 shows the design size, Column 3 reports the root-cause, Column 4 shows the actual fix done with the debugger and Column 5 reports the debug time for fixing the cause. It should be noted that the root-cause reporting is very quick and the debug time involves debugging multiple scan cell failures. The DIG-driven debugger was successful in root-causing all logical silicon failures and particularly useful to understand the internals of the CUT while fixing the bug. For example, in D1 with 711K gates, the tool reported the primary input and hold-latches as root-causes. After carefully examining the internals of CUT, the hold latch initializations were fixed to clean the pattern. On previous products, First Silicon debug took days, in the absence of our scan pattern debugger. We have reduced the debug time to the order of a few hours from the order of days.

In Table 4.2, we performed a controlled experiment - by simulating 100 test vectors with and without the DIG construction - to determine the average performance and memory overhead of the DIG construction during logic simulation. We report the run-time and

Table 4.2: DIG Run-time Overhead

Design	Size	Run-Time Overhead		Memory Overhead	
	no. of gates	100 TVs	Avg/TV	100 TVs	Avg/TV
D1	711K	2X	2%	8.6X	8.6%
D2	3.4M	1.7X	1.7%	6.8X	6.8%
D3	6.2M	2X	2%	9.2X	9.2%
D4	9.8M	2.4X	2.4%	7.6X	7.6%
D5	1.1M	2X	2%	30X	30%

TV: Test Vector

memory overhead for constructing the DIG for all test vectors in Columns 3 and 5 and the average overhead per test vector in Columns 4 and 6. It should be noted that the debug is typically performed for the failing scan cell and test vector. The DIG construction need not be performed for all the 100 test vectors. The DIG construction can be triggered multiple times and the graph nodes re-used across different DIGs.

It is seen that the average run-time overhead is between 1 and 2.5% and memory overhead is between 8 and 30% for each test vector in the debug build of the logic simulator. The increased memory overhead, in D5, is attributed to higher activity in the CUT leading to larger number of graph nodes being created.

4.7 Summary

In this chapter, we illustrated the implementation details of our Scan Pattern Debugger currently being used for partial scan pattern cleaning activities for production of next generation microprocessors in the industry. A suite of interactive commands have been added complementing with each step of the pattern debug flow. The results show that the debug time for patterns have decreased from a days to a few hours. The DIG overhead is illustrated

with an experiment of 100 patterns and the average run time and memory overheads are found to be negligible.

Chapter 5

Circuit SAT Implementation with DIG

5.1 Introduction

In this chapter, we investigate the application of the Debug Implication Graph in SAT solvers for Bounded Model Checking (BMC) and SAT solvers for circuit applications. Both applications require us to unroll the circuit for multiple time frames. As discussed in Section 2.2.4, an unrolled circuit for k time-frames, contains copies of the same circuit with PPOs of $(n - 1^{th})$s frame connected to the PPIs of the nth frame. The redundancy in making copies of the combinational circuit for each unrolled frame gives us an opportunity to use these repetitive properties to create more efficient implementation for many processes. Further in the chapter, we discuss about the implication graph that is formed within a SAT solver and modify on the lines of our DIG concept to make it more efficient.

Most state of the art SAT solvers are based on the Davis-Putnam-Logemann-Loveland

(DPLL) algorithm which performs a branching search with backtracking. It prunes the search space by using a deductive procedure called Binary Constraint Propagation (BCP). To summarize, there are three main engines:

1. Decision Engine: for choosing which variable/value to branch on
2. Deduction/ search Engine: for performing BCP and checking conflicts
3. Diagnosis: for conflict analysis and backtracking

As explained in Section 2.1.1, every gate can be expressed in a CNF form which can be extended for a whole circuit. Any decision/implication on the variable or gate gets assigned a node. The major roadblock in a generic SAT solver with no circuit properties is that there are no structural constraints on the order of decisions. This is unlike logic simulation, where the gate assignments happen in a levelized fashion with the earlier time frame evaluated first. It is because of this property, the implication graph constructed in a generic SAT solver is not the leveled fashion. On the other hand, our implementation keeps in mind the generalized SAT solver and does not constrain the decision orders to simplify the graph.

5.2 Implication Graph in SAT Solver

An implication graph is a key component of the conflict analysis which captures the current state of the SAT solver. The key components of the graph are -

- Nodes: represent assignments to variables
- Edges: represent clauses, which cause implication from the assignment to implication node

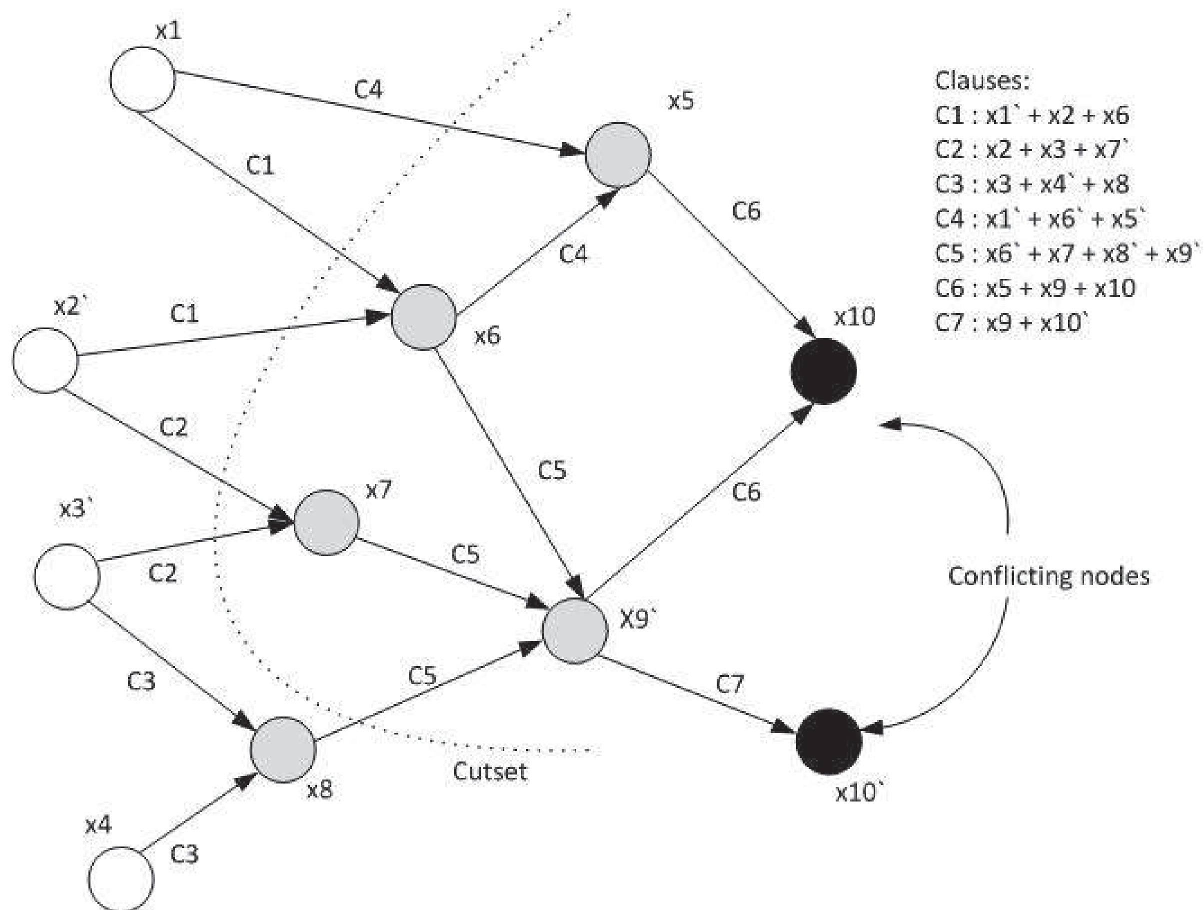


Figure 5.1: Implication Graph

- Decision Nodes: decision assignments, no incoming edges
- Conflict: when there are two nodes with opposite values assigned to the same variable.

Figure 5.1 shows an example how a implication graph is created in a SAT solver. The white circles are decision/ root nodes. $x1$ and $x2'$ are decisions which create a implication node for $x6$ through $C1$ clause. The gray nodes are implication nodes. We see that implication node $x9'$ implies $x10'$ through clause $c7$. It creates a conflict with the implication through clause $c6$. Conflicting nodes are marked with black circles. The solver then traverses back through the graph to find the cut set and add the conflict clause in the clause database.

5.3 Key Idea

When a circuit is unrolled in k time frames, the CNF formula for the new transformed circuit is just a k time extension of the basic circuit CNF representation. The number of clauses is k times the number of clauses from the base copy. For example, suppose originally the circuit had n clauses in its CNF formula. Then after unrolling for k time frames the total clause count will be $n * k$. Similar phenomenon is seen with the variable count. If originally there are N variables then after unrolling there will $N * k$ variables. So when a variable is assigned an implication graph node with a clause edge as its reason, this information can be reused in future time frames. Suppose if the $p^{th}(0 < p < k)$ copy of v say v_p is implied during BCP through clause c_p. At this moment we see that v_q already has a implication node in the graph with an edge from c_q. So instead of creating a new node element in the graph, we can use the available information and improve the memory efficiency.

5.4 Implementation

We use MiniSAT as our core SAT engine in our implementation. The MiniSAT takes a DIMACS format file ($.cnf$) file as its input. A $.cnf$ contains the CNF formula to be solved by the SAT solver. The variables are in form of positive integers with a '-' sign denoting NOT logic for the literal. Adapting to this, we have a complete unsigned integer based identification structure. All the clauses and variables are assigned positive integers as their IDs. Listed below are a few of definitions that will be useful to understand our implementation details

Definition 1. *$Variable\ Number\ (VN)$: It is the actual number associated with the variable*

in the .cnf file. $VN = v_i$ *where* $i \in 1 - n * k$ *and is associated for each and every variable exclusively.*

Definition 2. $Variable\ Id\ (VI)$ *: It is the Variable Number of the variable in the* O^{th} *time frame (base copy).* VI *for* v_i *where* $i \in 1 - n * k$ *is* v_j *where* $j \in 1 - n$*. Since we are unrolling the circuit the gate count in each frame will be the same. Therefore, we arrive to the relation* $v_i = v_j + t * n$*, where* t *is the time frame or the* t^{th} *copy of* v_j*.*

Definition 3. $Clause\ Number\ (CN)$ *: While reading in the* $.cnf$*, a clause database is created. Each clause is assigned a number to help in access its information from the database. The number associated in the database is the* CN *of the clause.*

Definition 4. $ClauseId\ (CI)$ *: It is the clause number of the clause in the* 0th *time frame (base copy). CI for* c_i *where* $i \in 1 - n * p$ *is* c_j *where* $j \in 1 - p$*.*

5.4.1 Data Structure

In our original SAT solver (miniSAT), every variable was assigned a node consisting of the reason clause information and the decision level. In our DIG implementation we try to store this information in a memory-efficient setup. We use a map structure with the VIs as the key information and a array of nodes mapped to it. Each node represents a unique representation of the variable by the unique CI of the clause from which it is implied. For example, consider a variable with VI as v which features in clause C_1, C_3 and C_3. Each node in the mapped array for key v either belongs to C_1, C_2 or C_3. Therefore, in this case there can be at most 3 nodes in the array. We manage the node creation for a specific variable by checking whether there is exists a node which has the same reason clause (same CI) or not. This takes care of the reason clause information from the original node structure. We store the decision level information in a decision stack for every node. Apart from being memory-

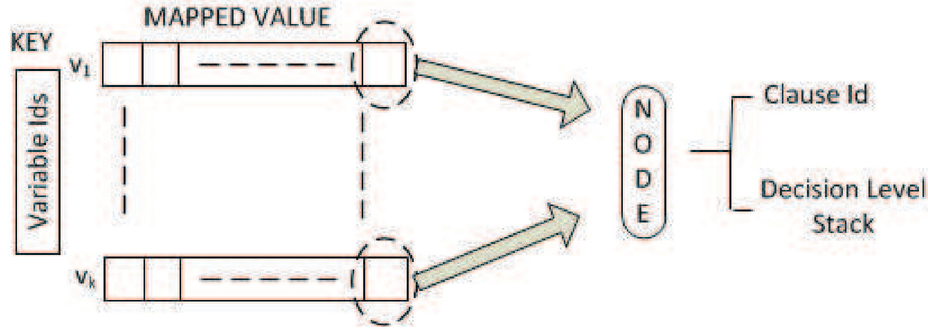


Figure 5.2: Data Structure

efficient, a stack serves as a perfect data structure for this purpose as during backtrack the node from a higher decision level will be eliminated before one from the lower decision level.

5.4.2 Graph Construction

We apply our DIG concept to the inherent implication graph that is created by the SAT solver to store the current state of the solver. As shown in Figure 5.1, the graph creation starts from assigning decision nodes and proceed to implication nodes by performing BCP after each decision. During BCP of assigning implication nodes, we check in our map if there is a node present for the particular VI having the same reason clause (same CI) as its edge. If such a node is found, then we simply push the decision level of the current implication to the decision level stack present in the node found; note that the decision level is different. If such a node is not present, we create a new node with the clause id as the reason and push the decision level in the stack. Let us consider a circuit C with k number of gates unrolled for t time frames. A gate g has VI as g_j where $i \in [1, k]$ and VN as g_i where $i \in [1, t * k]$. Suppose the order of decisions for g is $g_{j+5k},\ g_{j+4k},\ g_{j+2k},\ g_{j+3k}$. For the first assignment g_{j+5k} we create a node with a reason CI as c_i. Once we have this node for g_j, now if further assignments $g_{j+4k},\ g_{j+2k}\ or\ g_{j+3k}$ have reason clause CI as c_i, then we wont create a new separate node, but only add the decision level for the new assignment on the stack.

5.4.3 Conflict Analysis - Backward Traversal

In case of a conflict, conflict analysis is performed by following back the edges from the conflicting nodes, up to any edge which separates the conflicting nodes from the decision nodes. The graph is traversed in a breadth first fashion by picking up a reason clause and visiting all the variable nodes which form the clause.

A conflict is detected when we try creating a node for a variable id but it already exists with an inverse sign. At that moment the new node creation process is canceled and backtracking is initiated. For example, the SAT solver returns a variable with $VN = v_j$ for which conflict is occurring. It also provides us with reason clause with $CN = c_j$. We calculate the CI and the time frame from the CN and start back-traversing the graph in a breadth first search (BFS) manner. In Figure 5.1, suppose we are at node $x9'$ and we traverse in a breadth first manner, we will first visit nodes connected through $C5$ clause i.e. $x6$, $x7$ and $x8$ and then move deeper in the graph. While visiting each variable we record the reason clause in a queue. So, once we have visited $x6$, $x7$ and $x8$ we have $C1$, $C2$, and $C3$ in the queue. In our implementation, to get a reason clause for every VI and fetch the CI from the popped node. We calculate the CN of the popped clause reason by multiplying it with time frame of the CN. The corresponding graph node matching with the time frame is accessed and the CI is returned along with the decision level. The decision level is popped from the decision stack of the node accessed. The complete algorithm is shown below in Algorithm 5.1.

5.5 Experimental Setup and Results

In this section, we explain the experimental setup we used to implement our proposed graph technique in a Circuit-CNF SAT environment and discuss our results on ISCAS89 circuits.

Algorithm 5.1 Algorithm for Backward Traversal

Initialize $backtrack_level = \phi$
$conflict_variable \leftarrow$ Returned from BCP method
$reason_clause \leftarrow$ Clause
while $backtrack_level = \phi$ **do**
  for all variables v in reason_clause **do**
    $variable_id \leftarrow$ fetch variable id for v
    if $variable_id = v$ **then**
      fetch the $clause_id$ exclusive node information for $variable_id$
    else
      $time_frame \leftarrow$ difference between v and $variable_id$
      fetch the $clause_id$ from the database for $variable_id$ and the $time_frame$
      $decision_level \leftarrow$ pop decision stack from the node
      $reason_clause \leftarrow$ extending $clause_id$ by $time_frame$
    end if
  end for
  $backtrack_level \leftarrow$ process the learnt information
end while

We use MiniSAT as our underlying SAT solver for our experiments. The proposed tool takes as input the total number of gates and the number of time frames for unrolling. The tool has been completely written in C++ and executed on a dual-core 2.8GHz Intel Core 2 Duo machine with 2.9 GB of RAM. We keep the comparison confined to the number of nodes created and handled by the SAT solver (MiniSAT) in its original run against our graph implementation. We also record the memory saved in this process. Since this method is primarily targeted to large industrial circuits, where there is a abort because of memory explosion, we have taken the largest possible academic circuits and unrolled them to multiple time frames. The results are promising and show potential of creating an impact in any Hybrid (CNF-Circuit SAT) solver environment in the industry.

In Table 5.1, Column 1 lists the ISCAS89 circuits used; Column 2 shows the number of frames unrolled; Column 3 reports the number of variables in CNF formula for unrolled circuit; Column 4 lists the number of implication nodes created in the implication graph

of the original miniSAT solver; Column 5 lists the number of nodes created in our DIG implementation; Column 6 reports the reduction in the number of implication nodes. Column 7 report the runtime overhead due to reduced DIG implementation. But we note that this runtime overhead, when combined with a reduced memory, can show a reduction in final total runtime.

From Table 5.1 we see that our DIG-based solver creates only 2 to 5% of the total nodes created by the original miniSAT. The run time overhead almost averages around 23% which can be brought down to negligible amounts by putting in more information in the CNF file or creating a combined Circuit SAT environment. Our motive was to investigate the impact of DIG in SAT solvers being used for BMC applications. If we notice the 99.9% node reduction in column 6 for s15850, we can strongly defend that for industrial cases which have an abort case due to memory explosion at a very late stage of satisfiability process, our implementation will be highly beneficial and might skip the abort to present a complete solution.

Table 5.1: Implication Graph in SAT Solver using DIG

Ckt	# Frames	# variables	# Orig. impl nodes	# DIG impl nodes	node reduction	run-time overhead
s5378	200	608400	1451220	35837	97.5%	23.4%
s5378	300	912600	3601908	53434	98.5%	26.3%
s5378	500	1521000	10437138	88971	99.1%	28.3%
s13207	100	877200	3022753	49431	98.3%	23.8%
s13207	200	1754400	9388598	95514	98.9%	21.1%
s13207	700	6140400	81123690	356280	99.5%	28.2%
s15850	100	1047000	7463893	101066	98.6%	22%
s15850	200	2094000	56088456	221170	99.6%	20.1%
s15850	500	5235000	99622028	558409	99.9%	25.4%

Chapter 6

Conclusion

In this thesis, we proposed an optimized graph structure for sequential circuits in EDA tools such as a Scan Pattern Debugger and Bounded Model Checker. The whole motivation is to reduce the implication graph size, in particular the unnecessary nodes needed for the analysis. In huge industrial circuits, storing the complete graph can often cause insufficient memory problems. Using our proposed graph structure, we are able to control the memory problem and with a little (and sometimes no) run time overhead we are able to successfully backtrack to the correct decisions.

After the preliminaries and concepts are given in Chapter 2, in *Chapter* 3, we proposed the novel Debug Implication Graph (DIG) structure. We used a partial scan circuit example to explain how we create the graph along with event driven logic simulation. To avoid the node explosion, we stick to event driven properties and create new nodes only for gates which are being evaluated in the event wheel. The structure is closely interconnected through gate information from the netlist. During backtraversal we update our debug time frame as we visit each graph node. This way we are able to navigate through the broken reduced graph structure and get the correct reasons.

In *Chapter* 4, we applied DIG to Interactive Scan Pattern Debugger. The tool takes the failure information and the failing pattern as its input. It provides the debug engineer with handy commands to analyze the failures and debug them in a stepwise fashion. We explain how each command fit into the pattern debug flow. The current suite is in use for cleaning patterns for next generation microprocessor chips in the industry. The proposed approach provides insight and transparency into the gate values and their root-causes to the debug engineer for cleaning the pattern and fixing First Silicon failures. We were able to root-cause and clean all pattern related issues in a matter of few hours, as compared to a few days on previous products. The technique is also complementary to other manual and automated debug and diagnosis techniques to start with a minimal set of root-causes and debug the failures.

In *Chapter* 5, we modified the internal SAT solver with our DIG technique for a BMC application. In the SAT solver, we replaced the implication graph used for backtracking and conflict analyses with our modified graph structure based on DIG. We presented our data structure to hold the essential non redundant information and while backtracking help in fast information retrieval. Our implementation showed 98% node reduction compared to the original SAT solver. Though we had a runtime of around 35% we observed that only a few nodes were being reused more frequently than the others. Hence, we created an environment where our information can be easily fit into a cache compared to scattered and not optimized representation in the original SAT solver. We created this environment by running parallel instances of our DIG implementation and comparing to same number parallel instances of the original SAT solver. We were able to reduce our runtime from 35% to almost 4% and in some case our technique was faster.

Future Work

There are several opportunities for future work. Efficient backtracking algorithms having

multiple starting / conflict points can be very useful. For example, analyzing the fanin cone of many scan cells at one time and report the common reasons for debug. Commands for doing a reverse step and undo simulation can be very useful in the pattern debug suite. This will require creation of more efficient data structures for storing previous states in simulation processes or SAT solvers. For Circuit SAT application, using DIG for a hybrid CNF-circuit solver which has a common database for circuit elements and clauses will give us further insight on how our method can help on a commercial platform.

Bibliography

[1] G. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, pp. 114–117, 1965.

[2] A. Warren, "Introduction: Special issue on microprocessor verifications," *Formal Methods in System Design. USA*, pp. 135–137, 2002.

[3] C.-W. W. L-T. Wang and X. Wen, *VLSI Test Principles and Architectures*. Morgan Kaufmann Publishers, 2008.

[4] M. Chandrasekar and M. Hsiao, "Diagnostic test generation for silicon diagnosis with an incremental learning framework based on search state compatibility," in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pp. 68 –75, Nov 2009.

[5] W. Kunz and D. Pradhan, "Recursive learning: a new implication technique for efficient solutions to cad problems-test, verification, and optimization," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 13, pp. 1143 –1158, Sep 1994.

[6] Y.-W. C. L-T. Wang and K.-T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann Publishers, 2009.

[7] M. B. M. Abramovici and A. Friedman, *Digital Sysytems Testing and Testable Design*. IEEE Press, 1994.

[8] E. McCluskey, *Logic Design Principles: With Emphasis on Testable Semiconductor Circuits*. Prentice Hall, NJ, 1986.

[9] S. Reddy and R. Dandapani, "Scan design using standard flip-flops," *Design Test of Computers, IEEE*, vol. 4, pp. 52 –54, Feb 1987.

[10] V. Agrawal, K.-T. Cheng, D. Johnson, and T. Sheng Lin, "Designing circuits with partial scan," *Design Test of Computers, IEEE*, vol. 5, pp. 8 –15, apr 1988.

[11] V. Chickermane and J. Patel, "An optimization based approach to the partial scan design problem," in *Test Conference, 1990. Proceedings., International*, pp. 377 –386, Sep 1990.

[12] K. Baker and J. von Beers, "Shmoo plotting: the black art of ic testing," in *Test Conference, 1996. Proceedings., International*, pp. 932 –933, Oct 1996.

[13] D. Gizopoulos, *Advances in Electronic Testing Challenges and Methodologies*. Springer, 2006.

[14] S. Narayanan and A. Das, "An efficient scheme to diagnose scan chains," in *Test Conference, 1997. Proceedings., International*, pp. 704 –713, Nov 1997.

[15] S. Kundu, "Diagnosing scan chain faults," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 512 –516, Dec 1994.

[16] D. Nayak, S. Venkataraman, and P. Thadikaran, "Razor: a tool for post-silicon scan atpg pattern debug and its application," in *VLSI Test Symposium, 2004. Proceedings. 22nd IEEE*, pp. 97–102, Apr 2004.

[17] S. Venkataraman and S. Drummonds, "Poirot: a logic fault diagnosis tool and its applications," in *Test Conference, 2000. Proceedings. International*, pp. 253 –262, 2000.

[18] K.-H. Tsai, R. Guo, and W.-T. Cheng, "A robust automated scan pattern mismatch debugger," in *Asian Test Symposium, 2008. ATS '08. 17th*, pp. 309 –314, Nov 2008.

[19] Z. Wang and P. Maurer, "Lecsim: a levelized event-driven compiled logic simulator," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pp. 491 –496, Jun 1990.

[20] J.-K. Zhao, J. Newquist, and J. Patel, "A graph traversal based framework for sequential logic implication with an application to c-cycle redundancy identification," in *VLSI Design, 2001. Fourteenth International Conference on*, pp. 163 –169, 2001.

[21] M. Schulz, E. Trischler, and T. Sarfert, "Socrates: a highly efficient automatic test pattern generation system," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 7, pp. 126 –137, Jan 1988.

[22] T. Larrabee, "Test pattern generation using boolean satisfiability," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 11, pp. 4 –15, Jan 1992.

[23] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 15, pp. 1167 –1176, Sep 1996.

[24] J.-K. Zhao, E. Rudnick, and J. Patel, "Static logic implication with application to redundancy identification," in *VLSI Test Symposium, 1997., 15th IEEE*, pp. 288 –293, Apr-May 1997.

[25] P. Tafertshofer, A. Ganz, and M. Henftling, "A sat-based implication engine for efficient atpg, equivalence checking, and optimization of netlists," in *Computer-Aided Design, 1997. Digest of Technical Papers., 1997 IEEE/ACM International Conference on*, pp. 648 –655, Nov 1997.

[26] S. Bommu, K. Chandrasekar, R. Kundu, and S. Sengupta, "Concat: Conflict driven learning in atpg for industrial designs," in *Test Conference, 2008. ITC 2008. IEEE International*, pp. 1 –10, Oct 2008.

[27] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, (New York, NY, USA), pp. 151–158, ACM, 1971.

[28] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Design Automation Conference, 2001. Proceedings*, pp. 530 – 535, 2001.

[29] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pp. 142 –149, 2002.

[30] N. Een and N. Sorensson, "An extensible sat-solver," in *SAT*, pp. 502 –508, 2003.

[31] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[32] V. Boppana, S. P. Rajan, K. Takayama, and M. Fujita, "Model checking based on sequential atpg," in *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, (London, UK), pp. 418–430, Springer-Verlag, 1999.

[33] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 244–263, Apr 1986.

[34] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Design Automation Conference, 2003. Proceedings*, pp. 286 – 291, June 2003.

[35] H. Howe, "Pre- and postsynthesis simulation mismatches," in *Verilog HDL Conference, 1997., IEEE International*, pp. 24 –31, Mar-Apr 1997.

[36] U. Kumar, "Simulation mismatches can foul up test-pattern verification," *Electronic Design, ONLINE ID 10821*, Aug 2005.