Search-space Aware Learning Techniques for Unbounded Model Checking and Path Delay Testing

Kameshwar Chandrasekar

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

Doctor of Philosophy
in
Computer Engineering

Dr. Michael S. Hsiao, Chair
Dr. Dong S. Ha
Dr. Sandeep K. Shukla
Dr. Allen B. MacKenzie
Dr. Ezra Brown

14 April, 2006

Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

Keywords: Model Checking, Delay Fault Testing, Learning, ATPG, SAT, BDD

Search-space Aware Learning Techniques for Unbounded Model Checking and Path Delay Testing

Kameshwar Chandrasekar

Abstract

The increasing complexity of VLSI designs, in recent years, poses serious challenges while ensuring the correctness of large designs for functionality and timing. In this dissertation, we target two related problems in Design Verification and Testing: Unbounded Model Checking and Path Delay Fault Testing, that commonly suffer from extremely large memory requirements. We propose efficient representations and intelligent learning techniques that reason on the problem structure and take advantage of the repeated search space, thereby alleviating the memory required and time taken to solve these problems.

In this dissertation, we exploit Automatic Test Pattern Generation (ATPG) for Unbounded Model Checking (UMC). In order to perform unbounded model checking, we need the core image / preimage computation engines that perform forward / backward reachability analysis. First, we develop an ATPG engine, with search-space aware learning, that computes "all solutions" for a given target objective and stores it as a decision diagram. We propose efficient decision selection heuristics and derive a suitable cut-set metric to quickly obtain a compact solution set. The solution set that is obtained, with the initial state set as the objective, represents the one-cycle preimage. In order to use the preimage state set as the objective in the subsequent iterations, we propose efficient techniques to convert a decision diagram into clauses/circuit. We propose a node-based conversion scheme that derives the functionality of each node in the decision diagram. The proposed scheme contains the size of the state set and helps to iteratively compute the preimage for many cycles until a fixed point / desired state is reached.

Further, we gear the ATPG engine to directly compute the circuit cofactors, rather than individual solutions. The circuit cofactors contain a large number of solutions and hence capture a larger solution space. We also propose efficient learning techniques to prune the cofactor space and accelerate preimage computation. Then, we develop an exclusive image computation procedure that branches on the combinational inputs of the circuit and projects the values on the next state flip-flops as the image. We perform learning on the input solution space and incrementally store the image obtained as a decision diagram. We consistently show, with our experimental results, that our techniques are better than the existing techniques in terms of both performance and capacity.

In the case of delay testing, we consider the test generation for path delay fault (PDF) model, which is the most accurate in characterizing the cumulative effect of distributed delays along each path in a circuit. The main bottle-neck in the ATPG for PDFs is the exponential number of paths in a circuit. In this work, we use the circuit information to analyze the common segments shared by different paths in a circuit. Based on the common sensitization constraints, we propose to identify the "untestable core of segments" that cannot be sensitized together. We use these segments to identify the conflict search space for a huge number of untestable path delay faults apriori and prune them on-the-fly during test generation. Experimental results show that a huge number of untestable path delay faults are identified and it helps to accelerate test generation.

To my parents, my brother and Rajani

Acknowledgements

It is a pleasure to acknowledge all the people who made this work possible. I would like to express my sincere thanks to my advisor Dr. Michael S. Hsiao, for his inspiration and support throughout my graduate program. I would like to thank Dr. Ha, Dr. Shukla, Dr. Brown and Dr. Mackenzie for serving on my thesis committee. Last but not the least, I thank my friends and relatives for their emotional support and encouragement.

Kameshwar Chandrasekar
April 2006

Contents

Table of Contents vi

List of Figures xi

List of Tables xiii

1 Introduction **1**

1.1 Design Verification 1

1.1.1 Symbolic Model Checking 4

1.2 Delay Testing 6

1.2.1 Path Delay Fault Model 7

1.3 Outline of the Dissertation 8

2 Background **11**

2.1 Model Checking 11

2.1.1 Bounded Model Checking (BMC) 13

2.1.2 Unbounded Model Checking (UMC) 14

2.2 Reachability Analysis 15

2.3 Techniques to perform Reachability Analysis 17

2.3.1 Binary Decision Diagrams (BDD) 18

2.3.2 Boolean Satisfiability (SAT) 20
2.3.3 Automatic Test Pattern Generation (ATPG) 22
2.3.4 Other Methods 26
2.4 Path Delay Testing 27
2.4.1 Robust path delay fault model 27
2.4.2 Non-robust path delay fault model 28
2.5 ATPG for a given path delay fault model 28

3 All Solutions Automatic Test Pattern Generation (ATPG) 30
3.1 Introduction 31
3.2 Related work on BDD Variable Ordering & Learning 33
3.2.1 Success-driven learning 35
3.3 New Decision Selection Heuristic 37
3.4 Search-state based symmetry 42
3.5 Cut-set Occurring Probability 44
3.6 Experimental Results 47
3.7 Summary 52

4 State Set Management 53
4.1 Introduction 53
4.1.1 Previous Work 55
4.2 Preliminaries 57
4.2.1 SAT-based UMC 58
4.2.2 State-set Decision diagram 60

4.3 State-set Management 62
4.3.1 Basic Idea 62
4.3.2 Algorithm 64
4.3.3 An Example 67
4.3.4 Complexity Analysis 68
4.3.5 FBDD State-set Management 70
4.4 Experimental Results 71
4.5 Summary 74

5 ATPG based Preimage Computation 75

5.1 Introduction 75
5.1.1 Previous Work 76
5.2 Preliminaries 78
5.2.1 Review of SAT-based Preimage Computation 79
5.2.2 Review of Circuit-based Preimage Computation 80
5.2.3 Motivation 81
5.3 Basic Circuit Cofactor Expansion 82
5.3.1 Circuit Search-state Relations 84
5.4 Search-state aware Circuit Cofactor Expansion 88
5.4.1 Search-state induced clauses 88
5.4.2 Non-chronological backtracking 90
5.5 Experimental Evaluation 92
5.6 Summary 94

6 ATPG based Image Computation **95**

6.1 Introduction 96

6.2 Preliminaries 99

6.2.1 Image computation 99

6.2.2 Zero-suppressed BDD (ZBDD) 100

6.3 Basic Image Computation 101

6.4 Efficient Image Computation 104

6.4.1 Decision selection heuristics 105

6.4.2 Search-state based learning 106

6.4.3 On the fly state-set minimization 107

6.5 Experimental Evaluation 108

6.6 Summary 110

7 Path Delay Fault Testing **112**

7.1 Introduction 112

7.2 Preliminaries 114

7.2.1 Incremental Satisfiability (ISAT) 115

7.2.2 Path Delay Fault Testing 116

7.3 Static Learning 117

7.3.1 Static Logic Implications 118

7.3.2 Segment-specific learned clauses 118

7.4 Dynamic Learning from Unsatisfiable Cores of untestable partial PDFs 120

7.5 Experimental Evaluation 123

7.6 Summary 125

8 Conclusion **126**

Bibliography 129

Vita **141**

List of Figures

1.1 Miter circuit for combinational equivalence checking 3

1.2 State Space 4

2.1 Sequential Model for computation 12

2.2 Bounded Model Checking 14

2.3 Image/Preimage computation for UMC 15

2.4 Example FSM 17

2.5 BDD next state functions 18

2.6 BDD Preimage computation 19

2.7 ATPG Preimage computation circuit 24

2.8 ATPG Preimage Decision Tree 25

3.1 Cut-sets in the Search Space 36

3.2 Update Dynamic Connectivity measures 38

3.3 Dynamic & static connectivity 39

3.4 An Example to Illustrate the Effect of Different Testability Measures 41

3.5 Counter-example for Remark 43

3.6 Symmetric Input Assignments 45

4.1 Zero-suppressed Binary Decision Diagram (ZBDD) 61
4.2 Free Binary Decision Diagram (ZBDD) 62
4.3 ZBDD Node 63
4.4 Preimage ZBDD 67
4.5 Preimage computation for b10 73

5.1 Preimage computation model 79
5.2 Decision Tree Search for Cofactor Enumeration 83
5.3 Cut-sets in the Search Space. 84
5.4 Proof for Lemma2 87
5.5 Proof for Theorem 88
5.6 Search-state induced learning 89
5.7 Non-chronological backtracking 91

6.1 Image computation for s27 - 000 state 100
6.2 ZBDD 101
6.3 Partial Image ZBDD 104
6.4 Minimization of ZBDD nodes 108

7.1 Incremental SAT for Non-Robust PDF Testing 116
7.2 Prefix & Postfix partial PDFs 119
7.3 Unsatisfiable core of segments 121

List of Tables

3.1 All Solutions ATPG for ISCAS ’89 circuits 48

3.2 All Solutions ATPG for ITC ’99 circuits 49

3.3 Efficiency of Our Cut-set metric 50

4.1 Cases for a ZBDD node 64

4.2 Path based ZBDD to CNF 69

4.3 Node based ZBDD to CNF 69

4.4 Preimage Computation: Solution cube enlargement using Blocking Clauses 71

4.5 Preimage Computation: Solution cube enlargement using Circuit Cofactoring . . . 72

5.1 Preimage Computation until convergence 93

6.1 Image computation for ISCAS ’89 & ITC ’99 circuits 109

7.1 PDF ATPG for ISCAS ’85 & ’89 circuits 124

Chapter 1

Introduction

The growing advances in VLSI technology have led to an increased complexity in hardware systems. The complexity of large designs pose serious challenges to pre-silicon verification and post-silicon testing. Bugs that are not found in the early stages of the design can be expensive, and bugs that remain in the chip even after tape-out can be even more expensive. The discrepancies in a manufactured chip may be due to the design bugs or manufacturing defects. In order to handle these discrepancies, it is necessary to verify the design and generate test cases that can detect the manufacturing defects.

1.1 Design Verification

Design verification is the process of checking, if a design implementation conforms to its specifications of functionality, timing, testability and power dissipation. In most of the industrial designs, more than 70% of the effort is spent on design verification. As mentioned in [1], functional verification can be broadly classified into: Software simulation, Hardware emulation and Formal Verification.

In most of the industrial designs, simulation has been the most common way of verifying the designs. A test vector is applied at the inputs of the design under verification and the software model of the design is simulated. The response that is thus obtained is analyzed, against the expected response, to check the correctness of the design. In order to completely verify a design, all the test patterns need to be generated, but the number of test patterns increases exponentially with the size of the design. For an n input combinational circuit, there are 2^n possible input vectors and hence it becomes impractical to completely verify large circuits. Generally, a tractable set of input patterns are pseudo-randomly/intellegently generated and the design is verified against those patterns. Therefore, simulation based methods may miss some corner case errors as was seen in the case of infamous Pentium bug. Another drawback of simulation based methods is the huge amount of time taken to simulate the design for every test pattern.

On the other hand, emulation based methods, using Field Programmable Gate Array (FPGA) chips, offer a potential alternative to simulation based methods. They help to simulate a huge number of test cases since they speed up simulation by several orders of magnitude. However, the fundamental drawbacks of this approach are the expensive hardware emulators and long time requirements to map the design under verification to the emulator.

A comparatively recent alternative to simulation has been formal verification. It is analogous to a mathematical proof, where the correctness of a formally verified hardware design holds regardless of the input values that are applied. The consideration of all test cases is *implicit* in the case of formal verification. The representation of a hardware design plays an important role in the techniques that are used to verify the design. Some of the common representations include a network of gates, decision diagrams, logic descriptions, etc. The exact choice usually depends on the aspects of the design one wishes to model and verify. In general, the formal methods for verifying hardware designs can be broadly classified into: Equivalence checking and Model Checking.

In the case of equivalence checking, we attempt to see if the formal description of the implementation conforms to the "golden model" or specification. A common premise for equivalence checking is to see if the optimized version of a circuit conforms to its original version. The basic idea for combinational equivalence checking is to construct a miter circuit [2], as shown in Figure 1.1 and prove that the output of the miter circuit is a tautology zero. Basically, we tie the inputs of both the circuits to apply same input patterns. Then, we check if the outputs of both the circuits are same. If the output of the miter circuit is proved to be a tautology zero, then it implies that the outputs of the implementation circuit and the specification circuit are the same for all possible input patterns. Therefore, we can be 100% sure that the functionality of the implementation circuit is equivalent to the functionality of the golden model.

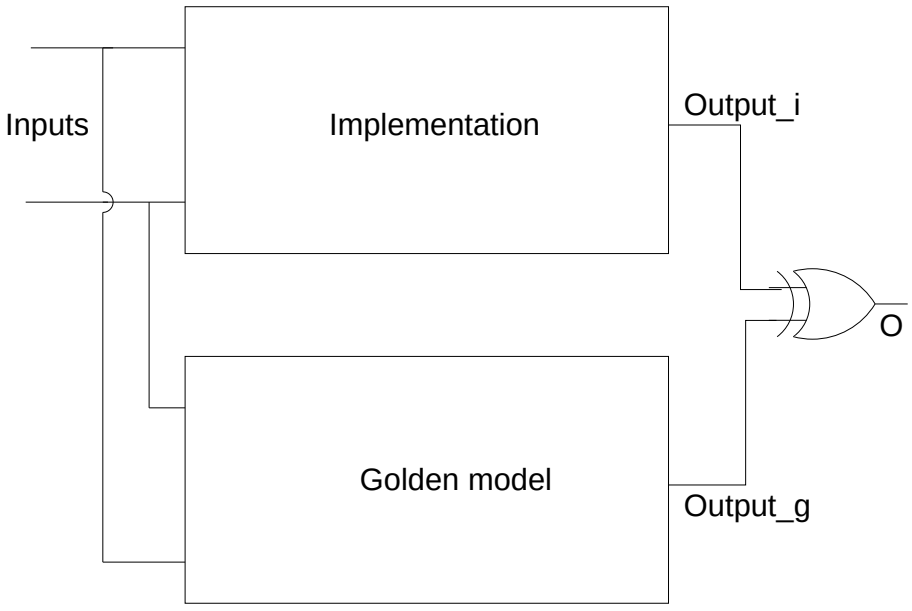


Figure 1.1. Miter circuit for combinational equivalence checking

In the case of model checking, we need to verify if the design satisfies a property or not. The specification is usually a set of properties that need to be verified for the circuit. For example, in a Traffic Light Controller, we need to verify the property that two perpendicular roads should

not get green light at the same time. Model Checking is usually performed on Kripke structures or Finite State Machine models of the hardware design, where analytical techniques are used to verify the properties. It relies on exploring the state space of the sequential design. This is called 'State space traversal', where we traverse the state space in the State Transition Graph (STG) to prove a property. The construction of the reachable state space is called Reachability Analysis. We need to compute the set all next/previous states that can be reached from a given set of states for Model Checking. In the State Transition Graph (STG), hypothetically shown for a Traffic Light Controller (TLC) in Figure 1.2, let node *I* refer to the initial state and node *GG* refer to a state where two perpendicular lights are green. It can be seen that we can perform backward / forward state space traversal to verify the above property and perform model checking.

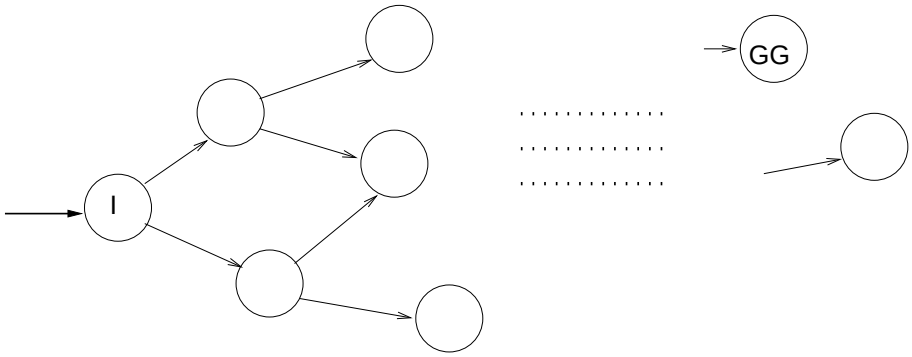


Figure 1.2. State Space

1.1.1 Symbolic Model Checking

Model Checking largely relies on state-space traversal of the sequential circuit. Explicit methods for state space traversal need a complete representation of the State Transition Graph as seen in the above example. Since the number of states is exponential to the number of state elements, explicit model checking methods are applicable only to small designs. With the increase in size of hardware designs, the state space increases exponentially for most of the them, thus making

explicit methods infeasible. Therefore symbolic methods were introduced more than a decade ago, where the transition relation and set of states are represented as Boolean formulas and state space traversal is done formally using mathematical computations. At the core of these formal methods, there is an image/pre-image computation step that performs state space traversal. In essence, computing image/pre-image requires the computation of the set of all next/previous states that can be reached from a given set of states in one cycle. Traditionally, Reduced Ordered Binary Decision Diagrams (ROBDD) were used [3, 4, 5], since they are canonical and can be efficiently manipulated. However, these methods suffer from potential memory explosion for present day circuits and are generally not suitable all circuits (ex. BDDs explode in size for multiplier circuits). Due to the increasing complexity and growing size of designs, several alternatives to ROBDD based methods have received attention in recent years. Recently, there is a growing interest in using non-canonical methods such as Boolean Expression Diagrams (BED), Reduced Boolean Circuits (RBC), Boolean Satisfiability (SAT), Automatic Test Pattern Generation (ATPG), etc. as a substitute for BDD based methods to trade-off space with time.

In this work, we exploit an ATPG engine to perform state space traversal that can be directly used for Symbolic Model Checking. We develop an 'all-solutions' ATPG engine that can be used as a substitute for BDD based methods in Symbolic Model Checking. To alleviate the inherent temporal explosion for ATPG, we propose search-space aware learning techniques that identify similar search-states and avoids re-searching them during solution enumeration. In order to perform backward reachability analysis, we directly compute the cofactors in the circuit and accelerate preimage computation. We analyze the relation among search-states and extend the search-space aware learning for preimage computation. On the other hand, for forward reachability analysis, we use the backtrace based ATPG to compute the image and store it in a specialized decision diagram for search-space aware learning. Experimental results on benchmark circuits show that our technique performs significantly better than existing BDD/SAT based methods.

1.2 Delay Testing

The increasing clock frequencies and reduced feature sizes have made delay testing a necessity. The purpose of delay testing is to verify that the circuit operates correctly at the desired clock speed. In the case of delay testing, the goal is to generate test vectors that can identify the timing defects in the circuit. Many fault models, such as transition fault model [6], gate delay fault model [7, 8], segment delay fault model [9], path delay fault model [10], etc. were proposed to model the timing defects. Detection of a delay fault normally requires the application of a pair of test vectors. The first vector, called the initialization vector, initializes the targeted faulty circuit line to a specific value. The second vector, called the launch vector, launches a transition at the circuit line and propagates the faulty effect to a primary output or a scan flip-flop.

In general, transition and gate delay fault models characterize the lumped delay defects at gates, while path and segment delay fault models characterize the distributed delay defects over several gates. The transition fault model characterizes an increased lump delay at a node in the circuit. It assumes that the rising/falling transition at a node in the circuit does not reach a latch or a primary output within the given clock period. The gate delay fault model assumes delays of different sizes (interval) at each gate in the circuit. The segment delay fault model assumes a distributed delay across several gates in a local region of occurence that forms a segment in the circuit. The path delay fault model assumes a small delay at each gate across a path. If the total delay exceeds the slack for the path, then the chip fails. It may be noted that the number of transitions faults, gate delay faults and segment delay faults is polynomial to the size of the circuit, whereas the number of path delay faults is exponential to the size of the circuit. As a result, ATPG for path delay faults takes exorbitant amount of time as compared to the other fault models. However, it considers a larger range of timing defects while generating the test vectors and is considered to be closest to ideal model for delay defects.

1.2.1 Path Delay Fault Model

The path delay fault (PDF) model is the most accurate in characterizing the cumulative effect of distributed delays along each path in a circuit. Based on the various sensitization criteria, path delay faults are classified into several classes: single-path sensitizable, robust, non-robust, functional sensitizable and functional unsensitizable. These classes have different testability conditions based on the specific fault detection conditions.

Irrespective of the sensitization criteria, the main bottle-neck in the PDF model is the exponential number of paths in a circuit. As a result, Automatic Test Pattern Generation (ATPG) suffers from temporal explosion if each path delay fault in the circuit is targeted explicitly. In order to accelerate test generation, the following two methods have been suggested in the literature: (1) Identify the untestable PDFs *apriori* and skip them for test generation. (2) Identify a subset of PDFs and perform ATPG on them. The first method does not violate the completenes of delay testing. The second technique does not consider all the PDFs in the circuit and hence, may miss some of the PDFs that are deemed necessary to ensure timing.

In our work, we target the problem of generating a complete test suite for all the path delay faults in a circuit. We propose static and dynamic learning techniques to improvise an Incremental Satisfiability (ISAT) framework for Path Delay Fault Testing. The identification of untestable path delay faults and pruning of path delay faults is inherent in our framework. We perform certain experiments to generate a complete test-suite for the non-robust path-delay fault model and the experimental results show that we are able to exclude a huge number of untestable path delay faults, on the fly during ATPG.

1.3 Outline of the Dissertation

In the next chapter, we provide some background on the recent techniques that have been investigated for Symbolic Model Checking. We start with some basic explanations on Unbounded and Bounded Model Checking. Then we explain the different methods used for Unbounded Model Checking such as: BED, RBC, SAT, ATPG, etc. In the latter part of the chapter, we introduce the sensitization critera for the different path delay fault models. Then, we explain ATPG for a given path delay fault model by recursively sensitizing the segments for all paths in the circuit.

In chapter 3, we introduce the new all-solutions ATPG engine. In contrast to the existing ATPG, which generates a test vector for a particular fault, we attempt to compute all solutions for a given objective. We explain the algorithm to generate all the tests and a brief overview of Success Driven Learning. Then, we target two problems that are tackled to improve the generation of all-solutions. We provide a decision selection heuristic that helps to improve the construction of the final solution set, both in terms of performance and capacity. We analyze the search-states that are reached during solution generation and prove some interesting relations between them. Then, we introduce a probabilistic metric to alleviate the memory cost of storing a huge number of search states. We evaluate the necessity of the decision selection scheme and the probabilistic metric on ISCAS ’89 and ITC’99 circuits and report the experimental results. Experimental results show that the new decision selection scheme and the use of our probabilistic metric significantly accelerates the all-solutions ATPG engine and also leads to memory savings.

In chapter 4, we explore the state-set management problem that deals with representing the sequential states in a decision diagram, across iterations, in an Unbounded Model Checking framework. We derive the functionality of each node in a decision diagram and show that it is suitable to represent the complete functionality of the state-set, instead of representing each state one-by-one. We demonstrate the effectiveness of our technique on a SAT-based Unbounded Model Checking framework to compute the preimage for ISCAS ’89 and ITC ’99 benchmark circuits. Experimen-

tal results show that our node based state-set representation alleviates both temporal and spatial complexity as compared to the existing explicit state-set representation.

In chapter 5, we use the ATPG-engine to compute the preimage i.e. to compute the set of all previous states that can be reached from a given set of target states in a sequential circuit. We introduce the novel idea of computing the circuit cofactors directly during the decision tree search, instead of computing the solutions using ATPG. We analyze the search-states for the circuit cofactors and introduce the search-state based learning techniques for pre-image computation. We evaluate the results on an iterative framework for ISCAS '89 and ITC '99 circuits and report the experimental results. We infer that the direct computation of circuit cofactors in an ATPG engine using search-space aware learning performs significantly better than existing SAT/BDD based techniques.

In chapter 6, we present the image computation problem from an ATPG perspective. We develop a novel backtracing-based ATPG technique for forward image computation. We carefully alter the ATPG engine to compute the image cubes and perform search-space aware learning. In order to improve the efficiency of image computation, we propose three heuristics: (i) gate-observability based decision selection heuristics to accelerate ATPG, (ii) search-state based learning techniques supported with a proof for correctness, and (iii) on-the-fly state-set minimization techniques to reduce the size of computed image set. Experimental results on ISCAS '89 and ITC '99 benchmark circuits show that we can achieve orders of magnitude improvement over OBDD-based and SAT-based techniques.

In chapter 7, we deal with ATPG for path delay faults in a circuit. We explain the basics of an Incremental Satisfiability (ISAT) framework for path delay testing. We introduce static and dynamic learning techniques to improve the ISAT framework. We use the static logic implications and segment specific learned clauses to constraint the search space for the path delay faults statically. Then, we identify the "untestable core of segments" genenerated from "untestable core of clauses" to identify the conflict spaces apriori. We use these segments to prune many untestable

faults on the fly. Experimental results on ISCAS ’85 and ISCAS ’89 circuits show that a huge number of untestable faults can be identified dynamically using our technique and this can potentially accelerate test generation.

Finally, we conclude the dissertation in chapter 8.

Chapter 2

Background

In this chapter, we provide the necessary background in Unbounded Model Checking and Path Delay Testing to understand our contribution in the area. In the first part of the chapter, we explain the different techniques to perform reachability analysis for Unbounded Model Checking. In particular, we explain the basics of an Automatic Test Pattern Generation (ATPG) engine with its application for preimage computation. In the later part of the chapter, we discuss the sensitization criteria for the different path delay fault (PDF) models. Then, we explain a recursive technique to find the complete test suite for a given path delay fault model.

2.1 Model Checking

Model Checking is an automatic technique for verifying finite state concurrent systems. Although the concept of model checking is generic to all finite state concurrent systems, in this work, we consider only hardware verification. The fundamental tasks to perform Model Checking are modeling, specification and verification.

In [11], the concurrent system is modeled as a Kripke structure. A Kripke structure, M is a four

tuple $M = (S, S_0, R, L)$, where

S is a finite set of states

$S_0 \subset S$ is the set of initial states

$R \subset S \times S$ is a total transition relation

$L\,S \mapsto 2^{AP}$ is a label for each state

A path in M is an infinite sequence of states $\pi = s_0, s_1, s_2 \ldots$ such that $s_0 = s_i$ and $R(s_i, s_{i+1})$ holds $\forall_i > 0$. We are generally interested in a *Boolean circuit description*, i.e. gate level representation of the hardware design. A pictorial representation of our model is a synchronous sequential design shown in Figure 2.1 (A). It is a mealy machine and a k-frame expansion of the Iterative Logic Array (ILA) model is also shown in the Figure 2.1 (B).

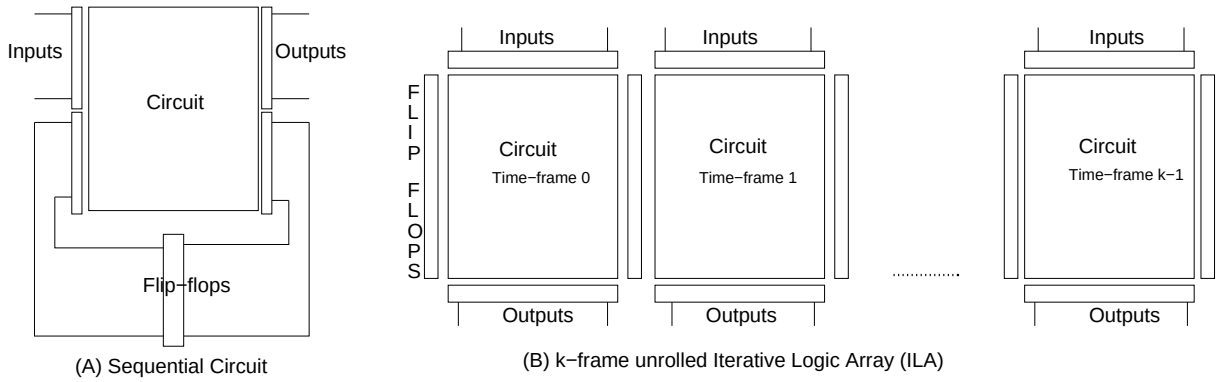


Figure 2.1. Sequential Model for computation

As mentioned in [11], temporal logic can be used to specify the properties and it is widely used in both the research community and the industry now-a-days. Temporal logic is a formalism for describing sequences of transitions between states in the hardware system. Some of the general properties that are verified using temporal logic are: safety, liveness and fairness. In this work, we

are primarily interested in the safety property verification, which verifies *if we can reach the buggy state from the initial state in a sequential design*. We refer the reader to [11] for details on model checking using the Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) and further information, since they are not necessary to understand the rest of the dissertation.

It should be noted that the evaluation of these properties relies on state space traversal, which requires a core image/preimage computation engine. Further, the main challenge in model checking is the *state space explosion problem*, which refers to the huge number of states that occur in large hardware systems.

2.1.1 Bounded Model Checking (BMC)

In most of the industrial designs, where bugs are profound in the initial stages of the design, it may be easier to generate counter-examples than to prove the correctness of a property. Bounded Model Checking (BMC) is geared towards finding these bugs faster in the design using a Boolean Satisfiability (SAT) framework. The concept of Bounded Model Checking was first introduced by Biere et al. in [12]. This concept generally works on sequential state paths of a bounded length, say k. In a BMC_k instance, we look at traces/paths of a bounded length, say k, that violate the property. We unroll the circuit to k-frames and assert the property in each frame. Then we attempt to see if we can find a consistent assignment to all gates in the design. The SAT problem is formulated such that a bug exists in the k-time-frame unrolled circuit if the propositional formula for BMC_k is satisfiable. If it is unsatisfiable, then the property holds for k-frames only and, we have to proceed with BMC_{k+1} iteratively until k reaches a *completeness bound* to prove that the property holds. This completeness bound could be very large and can lead to large CNF formulas for large values of k. Induction-based methods have been proposed in [13] [14] to obtain proofs earlier than the completeness bound. In general, BMC is more efficient in finding bugs/counter-examples in a (buggy) sequential design, rather than proving the correctness of a property.

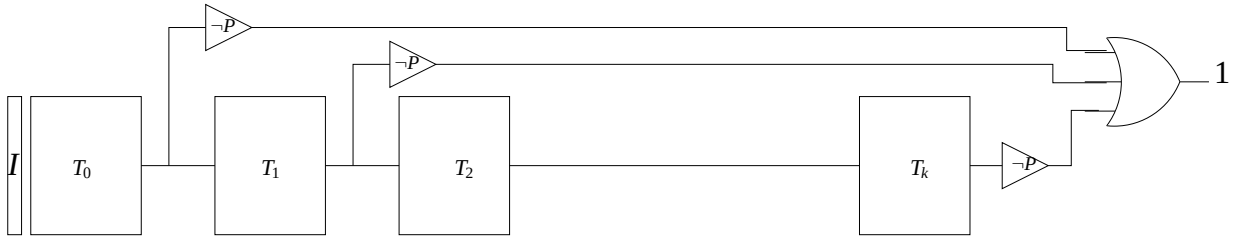


Figure 2.2. Bounded Model Checking

The Bounded Model Checking problem is formulated by constructing the following formula as shown in Figure 2.2:

$$BMC_k : \; I \wedge \bigwedge_{i=0}^{k-1} T_i \wedge \neg P$$

I is the characteristic function of the initial state set

T_i is the transition relation for the i^{th} frame

P is the property

The above formulation represents all the traces of length k or less, starting from an initial state. $\neg P$ represents the condition that the property is violated in any of the first k frames. Hence, if BMC_k is satisfiable, then we find a trace of length k or less that violates the property. Otherwise, we can guarantee that the property holds upto the first k frames of execution.

2.1.2 Unbounded Model Checking (UMC)

In the case of unbounded model checking, we explore the state space completely to prove the correctness of a property. We use the core image/preimage computation step to compute the set of reachable states iteratively as shown in Figure 2.3. We can start from the initial state and iteratively

compute the image, i.e. the set of next states, as shown in Figure 2.3(A). If the buggy state is reached in any iteration, then we stop and report the trace/path as a counter-example. Another alternative is to start from the buggy state and compute the preimage, i.e. the set of previous states, iteratively as shown in Figure 2.3(B). If the initial state is reached in any iteration, then we stop and report the trace as a counter-example. In either case, we attempt to compute the complete image/preimage iteratively in each step until either the buggy/initial state or a fixed point is reached.

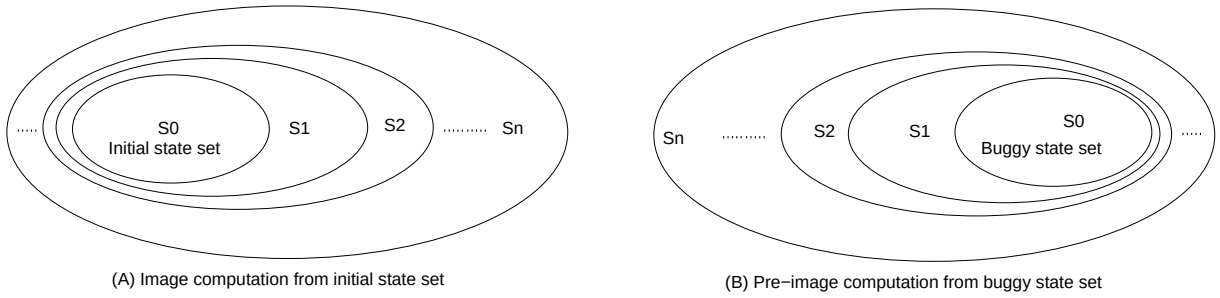


Figure 2.3. Image/Preimage computation for UMC

2.2 Reachability Analysis

It should be noted that the evaluation of all the model checking properties relies on state space traversal, which requires a core image/preimage computation engine. As mentioned earlier, image/preimage computation deals with finding the set of all next/previous states, starting from an initial state set.

Let us consider a completely specified, deterministic Finite State Machine (FSM) of Mealy type that is represented by the 6-tuple:

$$< I, O, S, \delta, \lambda, S_0 >$$

I represents the primary inputs

O represents the primary outputs

S represents the state set

$\delta : S \times I \mapsto S$ represents the next state function with partitions $\delta_i ... \delta_n$

$\lambda : S \times I \mapsto O$ represents the output function

$S_0 \subset S$ represents the initial state set

The corresponding transition relation $T(X,I,X')$ is defined by:

$$T(X,I,X') = \bigwedge_{i=1}^{i=n} (X_i' \equiv \delta_i(X,I))$$

X represents the present state variables

X' represents the next state variables

Symbolically, the image for a given set of present states, $S(X)$, is given by:

$$Image(X') : \ \exists_{I,X} \ T(X,I,X') \ \wedge \ S(X)$$

Similarly, the preimage for a given set of target states, $S(X')$, is given by:

$$Preimage(X) : \ \exists_{I,X'} \ T(X,I,X') \ \wedge \ S(X')$$

Basically, we first compute the conjunction of the transition relation $T(X,I,X')$ and the state set $S(X)$ / $S(X')$. Then, we quantify away all the input variables and present state variables to compute the image. Similarly, we have to quantify away the input variables and next state variables to compute the preimage.

2.3 Techniques to perform Reachability Analysis

Several techniques have been proposed to compute the image/preimage, depending on the representation of the transition relation and the state-set. We review some of the prominent symbolic techniques to compute preimage in the sequel. We use the FSM shown in 2.4 as a running example for all the techniques. It may be noted that the FSM has 1 input (i_1), 2 flip-flops (x_1, x_2) and 1 output(o_1). We will symbolically compute the preimage for the state 01, which can be explicitly seen to be $\{ 00, 01 \}$ in the Figure 2.4.

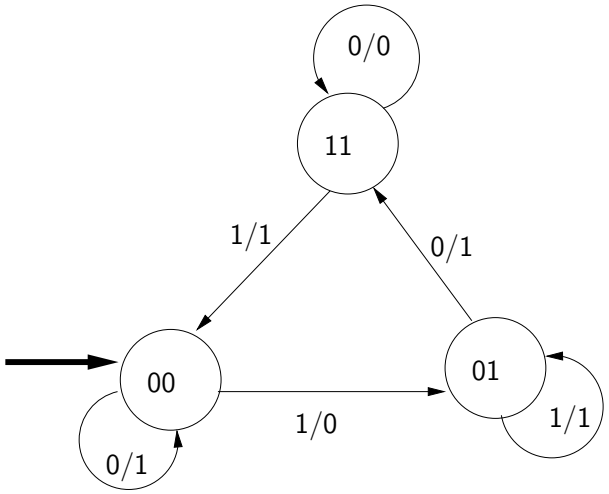


Figure 2.4. Example FSM

The next state functions and output function are given by:

$$x_1' = x_2 \wedge i_1 \tag{2.1}$$

$$x_2' = x_2 \wedge i_1 \vee \neg x_1 \wedge i_1 \tag{2.2}$$

$$o_1 = x_1 \wedge x_2 \wedge i_1 \vee \neg x_1 \neg x_2 \neg i_1 \vee \neg x_1 x_2 \tag{2.3}$$

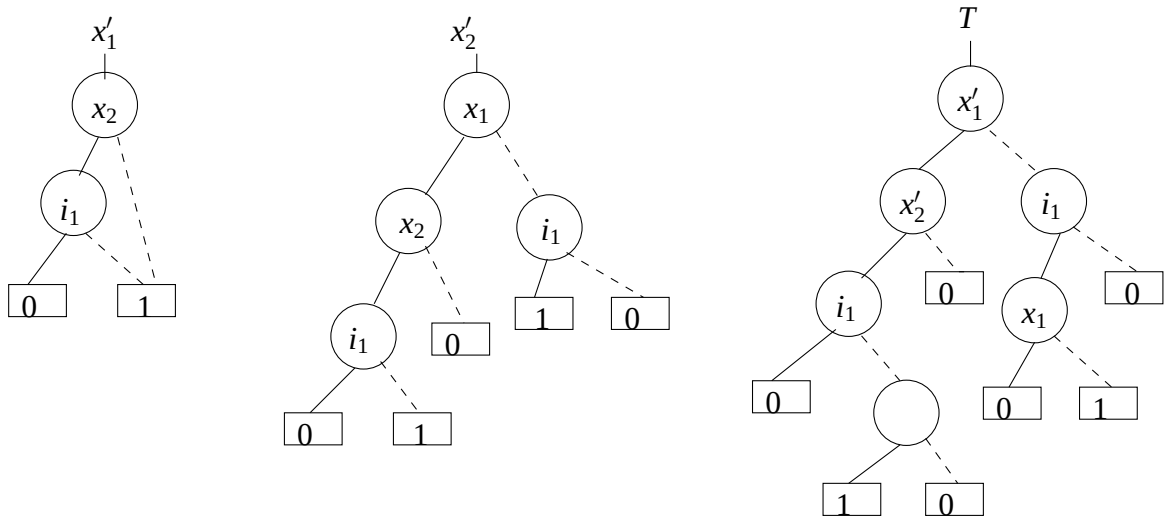
Therefore, the transition relation is given by:

$$T = (x_1' \equiv x_2 \wedge i_1) \wedge (x_2' \equiv x_2 \wedge i_1 \vee \neg x_1 \wedge i_1) \tag{2.4}$$

2.3.1 Binary Decision Diagrams (BDD)

Binary Decision Diagrams (BDD) were first introduced by Akers in [15] and then popularised by Bryant in [16]. It is basically a representation for the truth table of a circuit. It is a rooted directed acyclic graph that consists of two types of vertices: terminal and non-terminal vertices. Each non-terminal vertex, *v*, is labeled $var(v)$ and has two children: low(v) and high(v). Each terminal vertex is labeled either 0 or 1 and has no children. The root node of a BDD represents a Boolean function recursively, by defining the Boolean function of each node as follows:

- If v is a terminal node, $F_v = label(v)$
- If v is a non-terminal node, $F_v = (\neg v \wedge F_{low(v)}) \vee (v \wedge F_{high(v)})$



(A)$\delta_1(x) : x_1'$ (B)$\delta_2(x) : x_2'$ (C)$TransitionRelation : T(X, I, X')$

Figure 2.5. BDD next state functions

Bryant [16] proposed to obtain a canonical representation for the Boolean function using the following two restrictions and construct a Reduced Ordered BDD (ROBDD).

1. The variables should occur in the same order along each path from the root to terminal.
2. There should be no isomorphic subtrees or redundant vertices in the BDD

The ROBDDs for the next state functions and the transition relation for the running example are shown in Figure 2.5. The variable order is $x'_1, x'_2, i_1, x_1, x_2$. To compute the preimage, we first need to build the ROBDDs for the transition relation (Figure 2.5 (C)) and the initial state set (2.6 (A)). Then, we construct the BDD for conjunction of the transition relation and the initial state set as shown in Figure 2.6(B). Finally, we quantify away the next state variables and the primary inputs from the BDD in Figure 2.6 (B) to obtain the final preimage BDD as shown in Figure 2.6 (C). It can be verified that the final preimage BDD obtained is $\neg x_1$, which represents the state set $\{ 00, 01 \}$, as seen in the Figure 2.4.

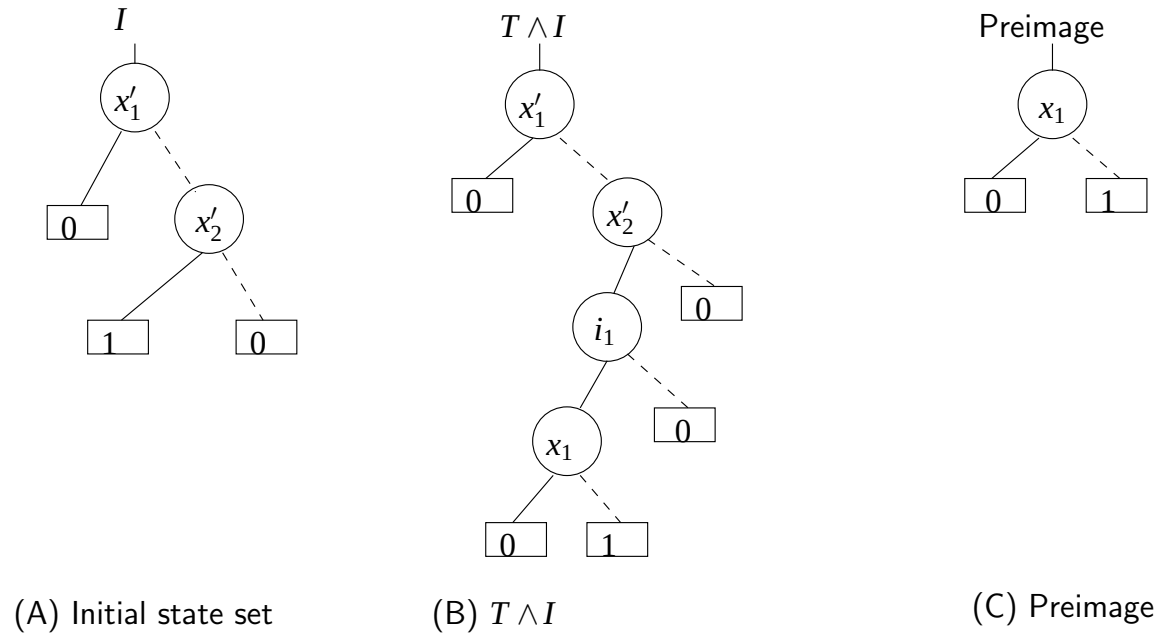


Figure 2.6. BDD Preimage computation

Efficient construction of ROBDDs was later proposed in [17] and was implemented in several publicly available packages, such as [18]. It was shown that BDDs can be used for representation

of Boolean functions and formula manipulation. They can be used for representing the transition relation and state-set for an FSM and then to compute the image/preimage. BDDs were used for image and preimage computation in [3, 4, 5] and we refer the reader to [5] for further details on ROBDD based Symbolic Model Checking.

2.3.2 Boolean Satisfiability (SAT)

Boolean Satisfiability (SAT) is a well-known constraint satisfaction problem, with many applications in VLSI CAD and Artificial Intelligence. Given a propositional formula ϕ that depends on a set of variables V, the SAT problem is to find if there exists a satisfying assignment for V that evaluates ϕ to true. Checking the satisfiability of general Boolean formulas in Conjunctive Normal Form (CNF) was shown to be an NP-complete problem, a few decades ago, in [19]. However, specific SAT problems such as 2-CNF, Horn-SAT can be solved in polynomial time. Most modern SAT solvers work on the Conjunctive Normal Form (CNF) representation of a formula. In a CNF, ϕ is represented as a set of clauses, where each clause is a disjunction of literals. They rely on the DPLL procedure [20, 21], and perform a backtracking decision tree search on the variables, to solve the SAT problem. The introduction of conflict analysis [22, 23] and optimized Boolean Constraint Propagation (BCP) with two-literal watching [24], has significantly accelerated the performance of present day SAT solvers.

The modern DPLL procedure to perform SAT solving is shown in Algorithm 1. Basically, we pick an unassigned decision variable and assert its implications. If there is a conflict, we analyze the conflict and derive a clause that constraints the conflict space and backtrack to the decision level that was responsible for the conflict. We proceed in this manner until all the variables have been assigned a value. If we backtrack to the root decision level, due to a conflict, then the formula is declared to be unsatisfiable. Otherwise, we will find an assignment to all the variables in the formula and declare the formula to be satisfiable.

Various variable ordering heuristics were investigated to pick a free decision variable, depending on the application of SAT solving. It was shown that the time taken to solve the problem depends significantly on the variable order chosen. We refer the reader to [25], for a survey on SAT solving and further information on SAT solvers. We also point out that a number of SAT solvers, such as [26, 27], are available in the public domain for research purpose.

Algorithm 1: DPLL-procedure for SAT solving

```
1  dpll_solve(node){
2      while (true) do
         /*Pick an unassigned variable as a decision */
3        if (var = decide_next_branch()) == false then
4            return SAT ;
         end
         /*Find its implications and see if there is a conflict */
5        while (deduce() == CONFLICT) do
           /*Add conflict clause & backtrack to that level */
6          blevel = analyze_conflict() ;
7          if (blevel == 0) then
8              return UNSAT ;
           end
9          backtrack(blevel) ;
         end
     end
  }
```

In order to perform preimage computation, we need an *all-solutions SAT solver*. Basically, we have to find all the solutions for the CNF, by backtracking even after a solution is found, and then quantify/eliminate the next state and primary input variables. In the case of our running example in Figure 2.4, the CNF for transition relation, T, is:

$$(\neg x_1' \vee x_2)) \wedge (\neg x_1' \vee \neg i_1) \wedge (\neg x_2 \vee i_1 \vee x_1') \wedge$$

$$(\neg x_2' \vee V_1 \vee V_2) \wedge (x_2' \vee \neg V_1) \wedge (x_2' \vee \neg V_2) \wedge$$

$$(\neg V_1 \vee x_2) \wedge (\neg V_1 \vee \neg i_1) \wedge (\neg x_2 \vee i_1 \vee V_1) \wedge$$

$$(\neg V_2 \vee \neg x_1) \wedge (\neg V_2 \vee i_1) \wedge (x_1 \vee \neg i_1 \vee V_2) \wedge$$

It may be noted that we have introduced intermediate variables, $V_1 = x2 \wedge i_1$ and $V_2 = \neg x_1 \wedge i_1$ to facilitate CNF conversion. The initial state set is represented by the CNF: $\neg x_1 \wedge x_2$. If we perform a simple all-solutions SAT solving, then the solutions obtained are:

$$(\neg x'_1, x'_2, i_1, \neg x_1, \neg x_2, \neg V_1, \neg V_2), (\neg x'_1, x'_2, i_1, \neg x_1, x_2, \neg V_1, \neg V_2)$$

After quantifying away the next state, primary input and intermediate variables, by simple elimination, the preimage is given by: $\{(\neg x_1, \neg x_2), (\neg x_1, x_2)\}$, which is the same preimage obtained in the BDD approach (Figure 2.6 (C)) and explicit state space traversal (Figure 2.4).

In recent years, significant amount of research work has been done on using SAT solvers for Model Checking. We discuss further details to perform reachability analysis using SAT solvers in the subsequent chapters. Further, we refer the reader to [28] for a basic survey on the recent advances on using SAT solvers for Model Checking.

2.3.3 Automatic Test Pattern Generation (ATPG)

Automatic Test Pattern Generation (ATPG) algorithms were originally designed to generate test vectors for different fault models in the circuit. The basic step is to compute an assignment to the primary inputs of a circuit that satisfy certain objectives. Some of the well-known ATPG algorithms are Roth's D-Algorithm [29], PODEM [30], FAN [31], SOCRATES [32], etc. We refer the reader to [33, 34, 35] for detailed information on ATPG and related concepts.

In our work, we use the modified PODEM based ATPG algorithm to generate a test vector that satisfies a given objective in the circuit. The basic PODEM algorithm is given in Algorithm 2. In PODEM, we implicitly explore all the 2^n possible input combinations to generate an input test vector, if it exists. It is basically a decision-tree search with the decision on the combinational inputs of the circuit only. We backtrace from the objective to find a value assignment to an input that is heuristically consistent with the objective. Then, we assert the implications of the new

assignment in the circuit. If the objective is satisfied, we stop and report the test vector. If the objective is violated, we backtrack in the decision tree. Finally, if there is no possible input vector, the algorithm reports failure. A number of heuristics are available in literature to accelerate the ATPG, by choosing the appropriate decisions and adding intelligence to the basic algorithm.

Algorithm 2: PODEM

```
1  PODEM(O, O_v){
       /*Objective satisfied */
2      if objSat() then
3          return SUCCESS ;
       end
       /*Objective cannot be satisified */
4      if objUnsat() then
5          return FAILURE ;
       end
       /*Backtrace to a primary input / flip-flop */
6      (p, v_p) := backtrace(O, O_v) ;
7      imply(p, v_p) ;
       /*Branch recursively */
8      if PODEM(O, O_v) == SUCCESS then
9          return SUCCESS ;
       end
       /*Backtrack on the value */
10     imply(p, ¬v_p) ;
11     if PODEM(O, O_v) == SUCCESS then
12         return SUCCESS ;
       end
13     imply(p, x) ;
14     return FAILURE ;
   }
```

It should be noted that this algorithm corresponds to a single-solution ATPG, i.e. it attempts to generate one solution that satisfies the objective. The above algorithm can be altered to compute *all* possible solutions, by exploring the complete decision tree, irrespective of solution / failure. Basically, we have to backtrack even after a solution is found to explore the remaining part of the

search space. This will remove the ‘if’ conditions on lines 8 and 11 in the algorithm. Finally, the decision tree will present itself as a free BDD that contains all the solutions that satisfy the given objective.

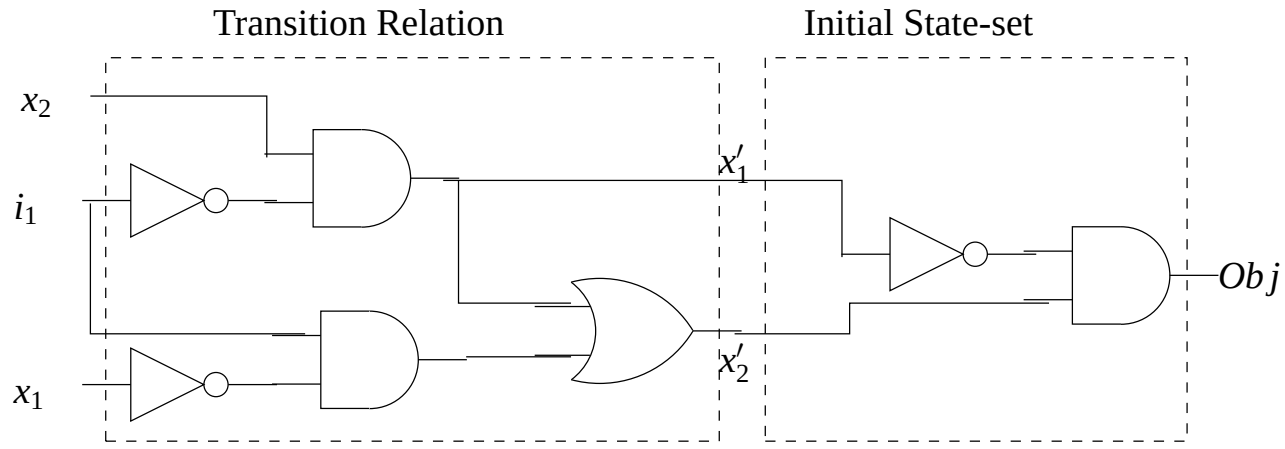


Figure 2.7. ATPG Preimage computation circuit

For preimage computation, we need an ‘all-solutions ATPG’. Similar to earlier techniques, we have to first construct the Boolean formula for the conjunction of the transition relation and the initial state-set. For ATPG, we use the combinational part of the sequential circuit for Transition Relation, T. This circuit has the primary inputs and present state latches as inputs, and next state latches as the outputs. The remaining portions of the sequential circuit can be abstracted away. For the initial state set, I, we again construct a circuit with the next state flip-flops as the inputs and a single output. Then, the conjunctive formula is obtained by appending the transition relation circuit to the initial state circuit. Let us call it, the monitor circuit for preimage computation.

Then, we compute all solutions for the monitor circuit, with the output set to 1 as the objective, using an ‘all-solutions ATPG’ engine. The decision tree obtained during the solution search is basically a free BDD and contains all the solutions. The next state variables are already abstracted away and we are left with the present state variables and primary inputs in the decision tree. The primary input variables can then be quantified, by elimination, from the solution cubes in the

decision tree.

For our running example in Figure 2.4, we first construct a monitor circuit as shown in Figure 2.7 to perform 'all-solutions' ATPG. As mentioned earlier, we construct the initial state-set circuit and append it to the transition relation to form the monitor circuit. Then, we attempt to find all-solutions, with the output of the monitor circuit set to 1 as the objective.

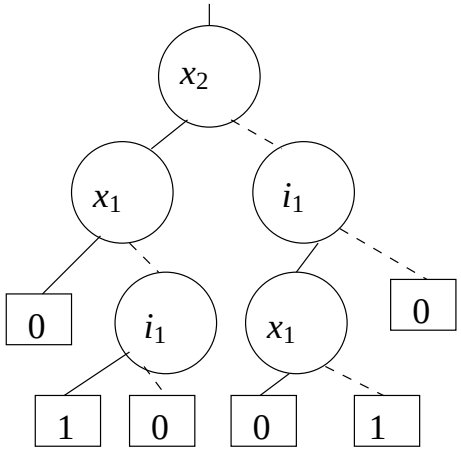


Figure 2.8. ATPG Preimage Decision Tree

The Decision Tree obtained during 'all-solutions ATPG' is shown in Figure 2.8. This Decision Tree is a free BDD and contains all the solutions that satisfy the objective. It may be noted that the variable order differs across different paths in the free BDD. From the Figure 2.8, the solutions obtained are $\{ (x_2, \neg x_1, i_1), (\neg x_2, i_1, \neg x_1) \}$. After we quantify away the inputs, by elimination, the preimage is obtained as $\{ (\neg x_1, x_2), (\neg x_1, \neg x_2) \}$. Again, it can be verified that this is the same preimage obtained by the earlier techniques in the previous BDD and SAT based approaches.

In this work, we make use of the basic PODEM algorithm for image/preimage computation in Unbounded Model Checking. We consider only the combinational part of the sequential circuit. In order to compute preimage, we need to compute the set of all-solutions, with the target state set as objective. Image computation presents itself as a different problem for the ATPG tech-

nique. We tweak the problem to compute the image and explain the details in chapter 6 for image computation. A straight-forward advantage of using an ATPG engine is the availability of circuit information to the algorithm. We can make use of this information to accelerate the reachability analysis.

2.3.4 Other Methods

Several other methods were proposed in the literature that combine the strengths of the above techniques or were alternatives to them. In [36], BDDs and SAT were interleaved for preimage computation. SAT was used to provide a disjunctive decomposition of the problem and later BDDs were used to solve the smaller end problems. Since BDDs are suitable to solve smaller problems, this technique provided a significant improvement over pure BDD based techniques in handling large instances.

In [37], the transition relation and the state-set is represented as specialized structures called Boolean Expression Diagrams (BED). They used elegant quantifier moving in their diagrams to simplify the problem. Then, the simplified BED structures are converted into BDDs. This method was suitable for equivalence checking problems that share a significant amount of circuit structure. In the case of model checking problems, the BED to BDD conversion could still blow up due to the space complexity.

In [38], the authors used Reduced Boolean Circuits (RBC), to represent the transition relation and state-set. They introduced simple rewriting rules, such as inlining and scope reduction, for quantifier elimination. However, in the worst case, they use formula disjunctions, i.e. $\exists_x F = F_x \vee F_{\neg x}$, for quantification. This can also potentially blow up the RBC size for large instances.

2.4 Path Delay Testing

As mentioned in chapter 1, path delay faults characterize the cumulative effect of distributed delays along a path in the circuit. Depending on the sensitization criteria, path delay faults are classified into: single-path sensitizable, robust, non-robust, validatable non-robust, functional sensitizable and functional unsensitizable faults. We review two of the popular fault models: non-robust and robust. We refer the reader to [39] for further information on Path Delay Testing.

For each path in the circuit from the primary input to the primary output, we can associate two path delay faults - rising PDF and falling PDF, which have the corresponding (rising / falling) transition at the beginning of the path. In order to detect a path delay fault, we need two vectors $< t_1, t_2 >$ to launch and observe the transition.

2.4.1 Robust path delay fault model

The robust sensitization criteria allows unconditional detection of a path delay fault. A test vector that detects a path delay fault under robust sensitization criteria will detect the fault independent of the delays in other paths i.e. it detects the fault under multi-fault assumption.

A path delay fault for which there exists an input vector pair such that it activates the required transitions on the path and all off-inputs in the path satisfy the following conditions is called a *robust path delay fault*:

1. there is controlling value to non-controlling value transition or a stable non-controlling value on the off-input, when the on-input has a controlling value to non-controlling value transition
2. there is a non-controlling value on the off inputs when the on-input has a non-controlling value to controlling value transition.

2.4.2 Non-robust path delay fault model

The non-robust test criterion is less stringent than the robust criterion. The detection criteria for non-robust PDF depends on the delays of certain signals outside the path as well, i.e., it detects the fault under the single fault assumption. A path delay fault for which there exists an input vector pair such that it activates the required transitions on the path and all the off-inputs settle down to their non-controlling values is called a *non-robust path delay fault*.

2.5 ATPG for a given path delay fault model

In this work, we target to generate the test vectors for all path delay faults in the circuit. We demonstrate a simple recursive algorithm [40, 41] to generate test vectors in Algorithm 3.

Algorithm 3: ATPG for path delay faults

```
1  PDF_ATPG(){
       /*Sensitize the PDF starting from each primary input (PI) */
2      foreach PI p do
3          recurseSegment(p, rising) ;
4          recurseSegment(p, falling) ;
       end
   }
5  recurseSegment(s, dir){
6      foreach y := fanout(s) do
7          if y is a primary output then
               /*Sensitizable PDF found */
8              storeTestVector() ;
           else
               /*Sensitize segment */
9              if sensitizeSegment(y, dir) == SUCCESS then
                   recurseSegment(y, dir ⊕ inverse(y));
               end
           end
       end
   }
```

In the algorithm, we start from a primary input and sensitize each segment incrementally until it reaches a primary output. After it reaches a primary output, we have a test vector that detects that path delay fault. If there is a conflict at a particular segment, then all path delay faults that are pre-fixed by the current sub-path are untestable. Therefore, we can backtrack at the segment and choose the next fanout segment for test generation. This is done for all the primary inputs in the circuit. It should be noted that we have to call the procedure twice for rising and falling path delay fault. At the end of the procedure, we have the set of all sensitizable path delay faults and the test vectors that sensitize them. A significant amount of research has been done to identify the untestable faults *apriori* and speed up test generation. We will review some of these techniques in chapter 7, when dealing with an Incremental Satisfiability (ISAT) framework for generating test vectors using Algorithm 3.

Chapter 3

All Solutions Automatic Test Pattern Generation (ATPG)

In this chapter, we will explain our intended approach to tackle Symbolic Model Checking problems using an all-solutions ATPG engine [42, 43]. In general, ‘All-solutions ATPG’ based methods have found applications in Model Checking sequential circuits, and they can also improve the defect coverage of a test-suite, by generating distinct multiple-detect patterns. Conventional decision selection heuristics and learning techniques for an ATPG engine were originally developed to ‘quickly’ find any available (single) solution. Such decision selection heuristics may not be the best for an ‘all-solutions ATPG’ engine, where all the solutions need to be found. In this chapter, we explore new techniques to guide an ‘all-solutions ATPG engine’. We first present a new decision selection heuristic that makes use of the ‘connectivity of gates’ in the circuit in order to obtain a compact solution-set. Next, we analyze the ’symmetry in search-states’ that was exploited in ‘Success-Driven Learning’ [44] and extend it to prune conflict subspaces as well. Finally, we propose a new metric that determines the use of learnt information a priori. This information is stored and used efficiently during ‘success driven learning’. Experimental results show that we can compute the complete solution-set with our new heuristics for large ISCAS ’89 and ITC ’99 circuits, where conventional guidance heuristics fail.

3.1 Introduction

In recent years, Automatic Test Pattern Generation (ATPG) / Boolean Satisfiability (SAT) based methods have offered a *potential substitute* for Reduced Ordered Binary Decision Diagrams (ROBDD) based methods, to the verification community [38, 44, 45, 46, 47]. Unlike ROBDD based methods that can suffer from memory explosion, ATPG/SAT based methods can perform image or preimage computation with reduced memory requirements. Image/Preimage computation is performed by modifying the underlying ATPG algorithm, to generate all the available solutions, and it is a key step in sequential equivalence checking and unbounded model checking. In addition to design verification, ATPG engines that are able to generate multiple solutions can also be used to generate different and distinct multiple-detect test vectors for a given fault, thus improving the overall defect coverage of a test suite.

Traditionally, ATPG engines are guided by the testability measures of the circuit. Several heuristics have been developed to determine these measures [33] that help to find *any* available solution quickly. The distance based testability measures account for the difficulty of testing a gate, based on its distance from the primary inputs and primary outputs. In [48], the authors compute the 0/1 probability at the output of each gate in the circuit and use them as testability measures. In [49], the authors derive certain numbers called SCOAP measures, for each gate in the circuit, that represent the difficulty of justifying and propagating a value. All these testability measures guide the ATPG engine while *backtracing* from the objective to select a decision variable. Improvements to obtain better testability measures have been incorporated in [50] using the concept of super-gates. However, the worst-case complexity of obtaining these measures can be exponential. Recently, in [51], Chang et al. obtained a better approximation for the testability measures using implications generated in the circuit. Based on implication reasoning, they estimate a correlation factor that accounts for signal correlations in the circuit. However, all these testability measures aim at finding a *single* solution quickly. Likewise, in SAT, many variable/decision selection strategies have been

proposed, such as MOMS (Maximum Occurences in clause of Minimum Size), DLIS (Dynamic Largest Individual Sum) [22] and VSIDS (Variable State Independent Decaying Sum) [23]. However, these methods lack the structural information available to ATPG engines. In [52], Iyer et al. integrated SAT and ATPG to develop an ATPG based SAT solver. They use the variable selection strategy of ATPG due to their superiority in choosing variables related to the objective.

An *all-solutions ATPG* attempts to build a complete decision tree (which can be reduced to a graph by sharing common sub-trees) that is essentially a Free Binary Decision Diagram (FBDD). Conventionally, the testability measures of the circuit guide an ATPG engine to select decisions as they search for the solution. As a result, the variable order in the Free BDD conform to the testability measures of the circuit. It is necessary to bias the variable order in such a way that we obtain a compact Free BDD as a whole. A compact Free BDD helps to reduce the size of the decision tree and in turn speeds up the ATPG engine. If the number of solutions is very large, in the order of billions, then it is not possible to store each solution one-by-one due to memory and time limitation. This phenomenon is referred to as *solution explosion* in [44] and it was shown that the final solution-set can be efficiently represented by the decision tree as a Free BDD. Therefore, it is necessary to obtain a compact Free BDD in order to address the solution explosion problem as well. Furthermore, as each ATPG decision leads to a different search-state in the decision tree, the number of search-states can be *exponential* in the number of inputs. Since each search-state is stored in a hash-table, for use in success-driven learning [44], the memory required to store all the search-states becomes a critical issue. Storing all the search-states in a knowledge-base may potentially lead to memory explosion for large circuits. In order to reduce the size of the *knowledge-base* and still benefit from useful search-states, it may be sufficient to store only the *frequently* occurring search-states.

Given the above discussions, the contribution of this work is three-fold:

1. We propose a new *decision selection heuristic* that guides an ‘all-solutions ATPG engine’ to

obtain an efficient variable order for the Free BDD.

2. We introduce the concept of *symmetry in search states* for ATPG and analyze a theoretical formulation for *success driven learning* proposed in [44].

3. In order to reduce the number of search-states stored during success driven learning, we propose a new *metric* that determines the *use* of a search-state.

Experimental results show that the proposed heuristic reduces the size of the final solution-set and speeds up the 'all-solutions ATPG engine' as compared to existing methods.

The rest of the chapter is organized as follows. We introduce the background on variable ordering and learning in section 3.2. The new *decision selection heuristic* for an 'all-solutions ATPG engine' is presented in section 3.3. In section 3.4, we introduce 'symmetry in search states' for ATPG and show that success driven learning is a restricted realization of search-state based symmetry. A new metric that is developed to determine the use of a search-state is introduced in section 3.5. Section 3.6 presents the experimental results and section 3.7 summarizes the paper.

3.2 Related work on BDD Variable Ordering & Learning

OBDD based methods are very sensitive to their variable order and thus are limited to small and medium sized circuits. Significant amount of work has been done on finding an efficient variable ordering technique in [53, 54, 55, 56, 57, 58]. Most of the work aim at placing *related* variables together in order to obtain a compact BDD. In [56], a PODEM based variable ordering technique is considered for building ROBDDs. The testability measures of PODEM are used to backtrace to the primary inputs by a depth-first search and the inputs connected by shorter paths are placed together. Recently, Aloul et al. proposed static variable ordering techniques in [57, 58], and their experimental results showed that their techniques can be better than dynamic variable ordering

techniques. They conjectured that placing *connected* variables together and partitioning the variables lead to compact BDDs and faster SAT. On the other hand, Free BDDs (FBDDs) are relaxed versions of Ordered BDDs, in which variables can appear in different orders along different paths but each variable occurs only once along any given path from the root to terminal. They are more compact than Ordered BDDs and sometimes lead to exponential savings in memory as shown in [59]. Although significant amount of work has been done to develop good variable ordering heuristics for ROBDDs, not much work has been done for Free BDDs.

In addition to variable selection heuristics, *learning* plays an important role in SAT/ATPG based methods. It helps to overcome the inherent time limitation of these methods and compete with BDD based methods. In [32, 60, 61, 62, 63, 64], powerful learning techniques were introduced for ATPG. In [22, 23], efficient conflict-driven learning techniques were introduced for SAT based methods. In [65, 45, 46], learning techniques have been proposed for an *all-solutions SAT solver*. These learning techniques improve the efficiency of the SAT solver that is an integral part of unbounded model checking. In [52, 66], Iyer et al. combine the strengths of both SAT & ATPG and present efficient learning techniques for the *sequential justification* problem. In [47], Lu et al. proposed signal correlation guided learning for an ATPG based SAT solver and obtained a speedup for hard industrial circuits. In all these aforementioned ATPG/SAT engines, the knowledge is in the form of implications [32], assertions [61] or conflict clauses [22, 23, 64]. Efficient manipulation of knowledge is required to reduce the overhead in storing and using the knowledge base.

Recently, in [44], Sheng and Hsiao introduced a new type of ‘success-driven learning’ that efficiently prunes the search-space for ‘ATPG based preimage computation’, by identifying identical solution-subspaces. The Transition Relation is represented by a levelized circuit and the set of states is stored in a Free BDD. A PODEM based ATPG engine is invoked to find all the solutions, resulting in a preimage where all the current state variables are quantified. *Equivalent search-states* that lead to the same solution subspace are identified to prune the search-space. The decision tree, obtained during solution-search, is stored as a Free BDD that represents the complete preimage

set. As solution subspaces heavily overlap during preimage computation, considerable savings is obtained in terms of time and memory. In [43], 'augmented success driven learning' and 'search-state based conflict driven learning' were introduced to further prune the search space for ATPG based preimage computation. However, the work in [44] and [43] used conventional testability measures to guide the ATPG engine, resulting in suboptimal all-solutions FBDDs.

3.2.1 Success-driven learning

Because an ATPG engine implicitly explores the entire search-space to generate a solution, 'an all-solutions ATPG engine' must continue and search for the next solution after each solution is found, until all solutions have been found. Each decision is considered a node in the decision tree, and the entire search-space is explored to find all solutions. A few terms are introduced before explaining the concept of *success-driven learning* [44].

- **Decision Tree:** The tree obtained by the branch-and-bound procedure of ATPG, with input assignments as internal decision nodes, is called the decision tree.
- **Search-State:** After choosing each decision and performing logic simulation, logic values of all the internal gates form a state in the circuit. This internal state of the circuit after each decision is considered a *search-state* for the decision tree.
- **Cut-set:** Consider the circuit as a directed acyclic graph, C, with edges directed from primary inputs to primary outputs. If we remove the fanout-stems of a set of gates from C to partition the graph into two sub-graphs X and Y, such that all the edges (that exist in C) across X and Y are directed from primary inputs to primary outputs, then the set of gates is called a cut-set.
- **Cut-set for search-state:** Each search-state can be uniquely represented by a cut-set in the circuit. After each decision, the cut-set can be obtained by a multiple backtrace from the

ATPG objective. The first frontier of *specified nodes*, encountered during backtrace, is the *cut-set for search-state*. In the sequel, we use the term cut-set to refer to "cut-set for search-state". Cut-sets that lead to solution subspaces and their decision tree nodes are stored in a hash-table.

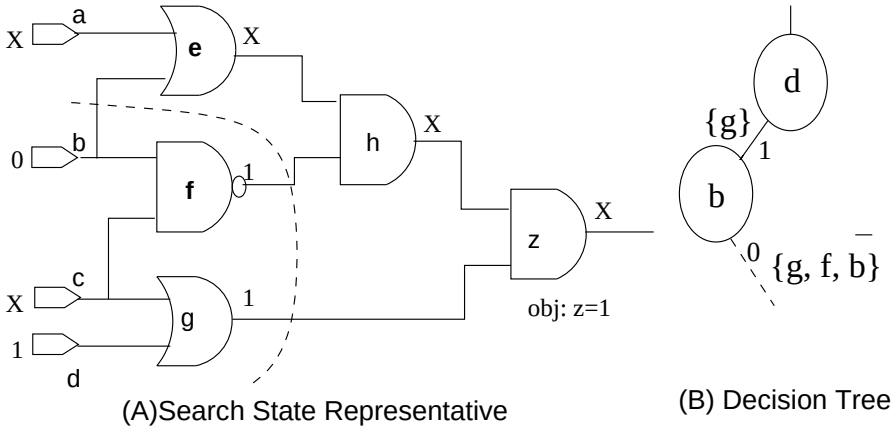


Figure 3.1. Cut-sets in the Search Space

For the circuit in Figure 3.1(A), a partial decision tree is shown in Figure 3.1(B). The sets indicated on the decision edges (eg. $\{g\}$ and $\{g, f, \bar{b}\}$) are cut-sets for the corresponding search-states. For the last decision $(b = 0)$, cut-set $\{g, f, \bar{b}\}$ is denoted by the dashed line, in Figure 3.1(A).

- **solution/conflict branch:** A branch in the decision tree that has at least-one/no solution below it.
- **solution/conflict cut-set:** A cut-set for the search-state in the *solution/conflict* branch.
- **solution/conflict subspace:** A search subspace below a *solution/conflict* branch.

In [44], the authors stored the cut-sets that lead to solution sub-spaces in a hash-table. A particular solution cut-set in the circuit will lead to a specific solution subspace in the decision tree. If the

same cut-set is encountered again, then we can simply link to that portion of the decision tree, instead of re-searching the same search space. After each decision, we search for the current cut-set in the hash-table. If an equivalent cut-set exists, we simply link the current branch to the stored node of the decision tree. Otherwise, we proceed with the usual search process. In this way, previously encountered search subspaces are not repeatedly explored using this technique.

3.3 New Decision Selection Heuristic

With the help of existing testability measures, conventional ATPG engines backtrace only through the easy (highly testable) portions of the circuit and stop as soon as *one* solution is found. Usually, this solution can be considered as the easiest solution with respect to the ATPG search process. However, an 'all-solutions ATPG engine' needs to explore the entire circuit to find *all* the available solutions. So the guidance heuristic for an all-solutions ATPG engine need not necessarily depend on the measures that only focus on the easier portions of the circuit.

We view the problem of finding all solutions that satisfy an objective as the process of constructing the decision tree as quickly as possible. This in turn is the problem of finding an efficient variable ordering technique for a Free BDD. The variable ordering heuristic is integrated into the backtrace routine of the ATPG engine. Unlike the dynamic variable ordering techniques in BDDs, we do not try to move the variable *up or down* to find a suitable position for a variable. Instead, after choosing each variable, we dynamically choose the next variable that is suitable for that position.

In [57], a hypergraph is built from the CNF formula and a *static variable order* is obtained by partitioning the variables and placing connected variables together. In order to obtain a good variable order, it is conjectured that *well connected* variables should be placed together for SAT and BDD. In our technique, we exploit the inherent graph structure of the circuit to dynamically find the gates that are well connected (by combinational paths) to the previously made decisions (represented by search-state in the circuit). After each decision, we estimate the *connectivity* of

each gate to the previous decisions. While backtracing, we always choose the gate that has the highest connectivity measure. In some cases, we only consider the connectivity of a gate to the objective. In this way, we attempt to put all connected variables together in the decision tree. Gates that are well connected are likely to be highly correlated and hence should be chosen together. In a PODEM based ATPG, we decide on the primary inputs that are obtained by the backtrace routine. The following subsections further explain the concept of connected variables.

```
// # paths through a gate = # paths in its fanin_cone *
                                  // # paths in its fanout_cone
// # paths in fanin/fanout cone of each gate is counted once
function guidance_measures(levelized_ckt) {
    Initialize the dyn_msr of all gates to 0
    // step1. Initialize the dyn_msr of gates in the cutset
    for (each gate in the cut_set)
         dyn_msr[gate] = stat_msr[gate];

    // step 2. Update the dyn_msr for all gates in
    // the fanout_cone of the cutset
    for (each gate in fanout_cone of cut_set)
        dyn_msr[gate] = Sum(dyn_msr[fanins]) *
                                  #paths[fanout_cone]
}
```

Figure 3.2. Update Dynamic Connectivity measures

Connected Variables: At each decision during ATPG, the connectivity of a gate is estimated as the *number of combinational paths* that connect the gate to the cut-set (formed by previous decisions) and the objective. From Graph Theory, a well-known linear time algorithm is sufficient to compute the number of paths passing through each gate in the circuit. A similar algorithm was used in [67] to estimate the fault coverage of path delay faults. Due to its linear time complexity, the computation overhead in estimating the connectivity measures is usually very small and it can

easily be integrated into the backtrace routine of the ATPG engine.

Initially, the number of paths that connect a gate to the objective and the primary inputs is estimated. This measure is estimated for all gates in the circuit and stored as a *static connectivity measure*. After each decision, a *dynamic connectivity measure* is assigned for all gates in the fanout cone of the cut-set. A basic algorithm is shown in Figure 3.2. The gates in the cut-set are initialized to their static measures. These gates represent the previously made decisions in the decision tree. The gates outside the cut-set fanin/fanout cone (R3 in Figure 3.3) are not *directly* connected to the decisions made so far. So they are initialized to 0. Next, for each gate in the fanout-cone of the cut-set (R2 in Figure 3.3), the number of paths (dynamic connectivity measure), it occurs in, is recursively estimated from cut-set to the objective. Note that the paths due to the gates in R3 are ignored. In this way, gates that connect the cut-set to the objective are assigned dynamic connectivity measures depending on their connectivity to the objective and cut-set.

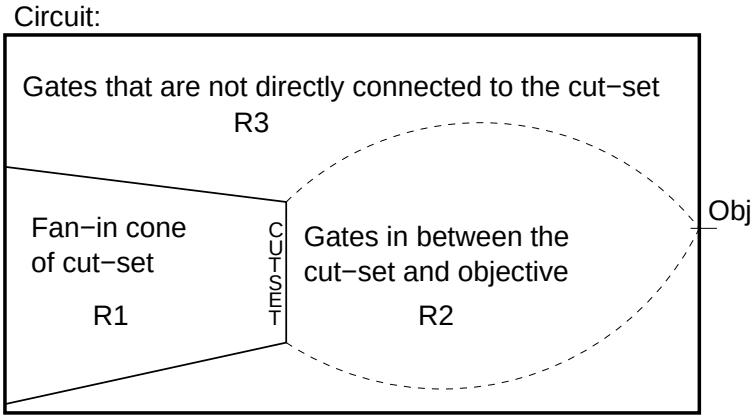


Figure 3.3. Dynamic & static connectivity

While backtracing from the objective, we will encounter two types of gates, as illustrated in Figure 3.3:

1. **Gates in between the cut-set and the objective - R2:** For these gates, we make use of the dynamic connectivity measures, since it is a representative of the connectivity of the gate to

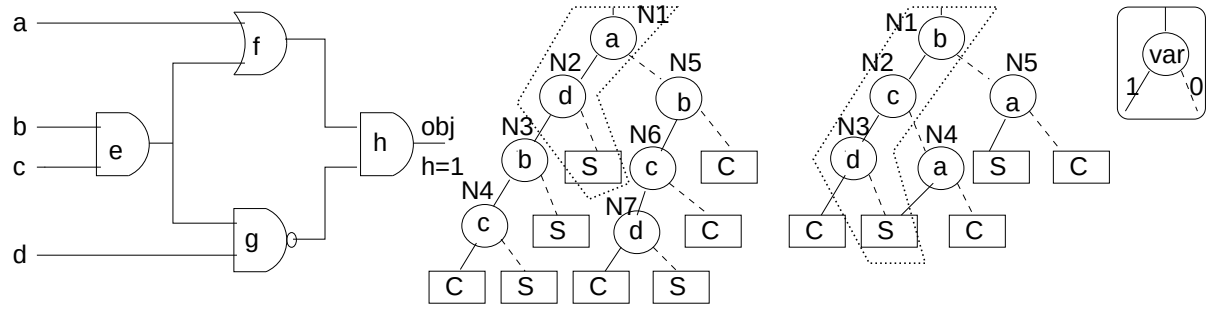
previous decisions and objective.

2. **Gates that are not directly connected to cut-set - R3:** For these gates, we use the static connectivity measures to select the gate that is well connected to the objective. Note that the dynamic connectivity measures for these gates are '0', because they are not directly connected to the cut-set.

It may be noted that if a gate, g, does not lie in between the cut-set and objective, i.e. $g \in R3$, then all the gates in the fanin-cone of g cannot lie in between the cut-set and objective, as shown in Figure 3.3. While backtracing, once we reach a gate with 0-dynamic connectivity measure we can start using the static connectivity measure for all the gates in its fanin cone. This helps us to implement an easy single switch from dynamic connectivity measures to static connectivity measures while backtracing in the circuit.

As discussed previously in section 3.2.1, a cut-set is obtained by a multiple backtrace from the objective to the primary inputs of the circuit. Since we are choosing connected variables together, all elements in the cut-set tend to be closer to each other. As a result, we are more likely to obtain cut-sets with smaller widths.

An example: A slight variation of our technique is to select the variables based on static connectivity measures alone. In that case, we need not update the dynamic connectivity measures for every backtrace. Due to its ease of explanation, we use the static connectivity measures in this example. Figure 3.4 demonstrates our technique for a reconvergent structure that is present in many circuits. For the circuit shown in Figure 3.4(A), the objective of the ATPG engine is to find all solutions that satisfy the objective $h = 1$. The SCOAP measures, (C1, C0) and our static connectivity based measures, (m) are tabulated and listed in Figure 3.4(D). The traditional guidance heuristic in PODEM traverses the path $h - f - a$ and picks a as the first decision. Note that there was a choice at gate f on the path in which either a or e could be selected. a is chosen by SCOAP since



Gate	a	b	c	d	e	f	g	h
SCOAP (C1, C0)	1,1	1,1	1,1	1,1	3,2	2,4	2,5	5,5
Ours (m)	1	2	2	1	4	3	3	6

(D) ATPG Guidance Measures

Figure 3.4. An Example to Illustrate the Effect of Different Testability Measures

it is easier-to-control when compared to *e*. This decision process continues and the final complete decision tree obtained for all solutions is shown in Figure 3.4(B). In this tree, a solid edge indicates the 1-branch and a dashed edge represents the 0-branch for each node.

For the same circuit in Figure 3.4(A), our new heuristic backtraces through *e* (after $h - f$) first, since it occurs in many paths as compared to *a*. If we follow our guidance heuristics for the rest of the backtraces, the corresponding decision tree obtained is shown in Figure 3.4(C). The first solution obtained by each technique is highlighted by a dotted enclosure in the corresponding decision tree. It is seen that SCOAP-guided ATPG finds the first solution in only 2 decisions - $\{a, \bar{d}\}$, and our technique finds the first solution in 3 decisions - $\{b, c, \bar{d}\}$. However, the number of nodes in the decision tree 3.4(C) obtained by our technique (5) is less than the number of nodes in the decision tree obtained by using SCOAP measures (7). It may be noted that the nodes, N4 and N5 in Figure 3.4 (C) are identical. These nodes will be shared in the Free BDD and the effective number of nodes required to store the solution-set is only 4. Through this example, we see that although SCOAP finds the first solution in fewer decisions, our technique computes the complete solution-set in a fewer number of decisions.

3.4 Search-state based symmetry

In this section, we introduce a few definitions and theorems that help to analyze the search-states occurring in the decision tree of an ATPG engine.

Equivalence: Two search-states are said to be equivalent if they lead to the same sub-decision tree for a given ATPG.

Symmetry: Two partial input assignments are said to be symmetric if they form equivalent search-states.

For example, in Figure 3.4 (C) the search-states at nodes N4 and N5 are equivalent and the corresponding partial input assignments, $\{b, \overline{c}\}$ and $\{\overline{b}\}$ are symmetric. Note that this notion of symmetry is different from the ones that are generally used. Unlike previous methods, where permutations of fully specified input assignments and symmetry on two variables were used, we define symmetry on partial input assignments that form different decompositions of the circuit during ATPG.

Theorem: *If two cut-sets are equal (same), then they represent equivalent search-states.*

Proof: Cut-sets that are equal decompose a circuit to identical sub-circuits. If the same decision process is used, then the decision tree for identical sub-circuits will definitely be isomorphic. Since the sub-decision trees are the same, the corresponding search-states represented by equal cut-sets are equivalent. $\diamond$

Remark: *All equivalent search-states are NOT necessarily represented by the same cut-set.*

Proof: We prove the Remark by showing a counter-example, where two different (unequal) cut-sets represent equivalent search-states. Figure 3.5 (B) shows the ISCAS '89 circuit - s27, modified to find the one-cycle preimage for 001 at the next state flip-flops. The objective is to justify a 1 at the output of gate 22 in the circuit. The guidance measures are shown in brackets near each gate in

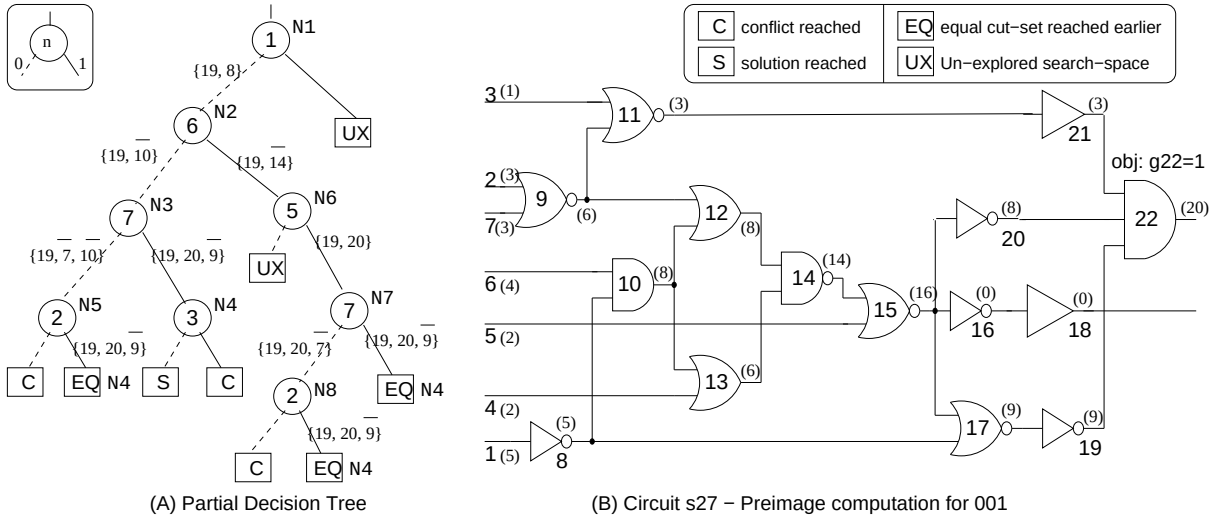


Figure 3.5. Counter-example for Remark

the circuit. Note that the guidance measures are only heuristics, and they change the structure of the decision tree. Different guidance measures lead to different cut-sets. We show that for a fixed guidance heuristic, different cut-sets can sometimes lead to the same sub-decision tree.

A partial decision tree obtained during the ATPG is shown in Figure 3.5 (A). The solid edge is the 1-branch and the dashed edge is the 0-branch for each node. The nodes are labeled in chronological order of decisions. The terminal nodes refer to a solution or a conflict or an equivalent search-state (due to equal cut-sets) that was *stored* earlier. The link for the equivalent search-state terminals are shown by the node numbers next to the terminal nodes. For example, the 1-branch of N5 is connected to N4, since the corresponding cut-sets are equal. It may be observed in the decision tree that the 0-branch of N2 and 1-branch of N6 have isomorphic sub-trees below them. However, the corresponding cut-sets are unequal. By definition of equivalence, the two search-states represented by cut-sets - $\{19, \overline{10}\}$ and $\{19, 20\}$ are equivalent. However, the corresponding cut-sets are different.$\diamond$

Corollary: *Success driven learning is a restricted realization of equivalence in search-states.*

Proof: In success driven learning, we use the cut-sets as a representative for the search-states. A cut-set is stored only if it leads to at least one solution. If an *exact* match for the solution cut-set occurs again, then we use the cut-set. From the Remark, we saw that unequal cut-sets can also represent equivalent search-states. On the other hand, even if a search-state leads to a conflict sub-space then its equivalent search-state will also lead to a conflict sub-space. This phenomenon is not exploited in success driven learning. $\diamond$

From the above discussion, it is seen that certain cut-sets that lead to the same search sub-space are not identified by success driven learning. Since equivalent cut-sets lead to the same search sub-space, it is immaterial if they are solution cut-sets or conflict cut-sets. As a result, success driven learning can be extended to prune conflict subspaces as well.

3.5 Cut-set Occurring Probability

Cut-sets that occur frequently during ATPG help to prune search subspaces, while other cut-sets may be less useful to the 'all-solutions ATPG engine'. It is desirable to store only the cut-sets that occur frequently during the search process. Cut-sets that do not occur again or with low probability of occurrence can be ignored to save memory without loss of performance. Therefore, it prompted us to develop a metric that determines the usefulness of a cut-set. Based on this metric, we may decide to store a cut-set or ignore it. This will help to reduce the memory requirement for the knowledge base, without significant loss of information.

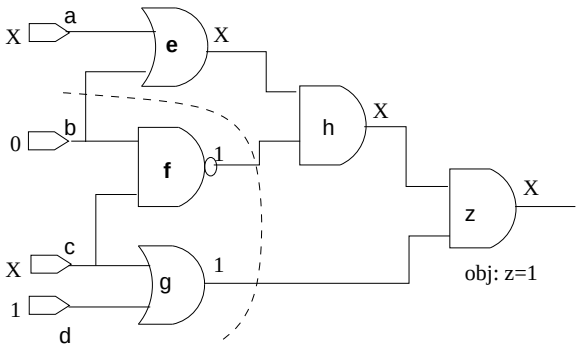
A cut-set, C, comprises of a set of specified gates, say

$$C = \{g_1, g_2, g_3, ..., g_n\}$$

where, g_i is a gate at the frontier of the search-state/cut-set

and, n is the number of gates in the cut-set.

A cut-set can occur multiple times only if it has many symmetric input assignments. An appropriate way to analyze the use of a cut-set is to count the number of symmetries that generate the same cut-set. If there are many symmetric assignments, then the cut-set is likely to occur again. For example, in Figure 3.6, the symmetric input assignments that generate the same cut-set $\{g, f, \bar{b}\}$ for the circuit are listed. It can be observed that it is very time-consuming to count the number of symmetric assignments that generate the same cut-set after each decision. Even if we count all the symmetric input assignments, a given ATPG engine may not consider all of them. As an alternative, we *estimate the use* of a cut-set using probabilistic measures.



Note: a = X

b	c	d
0	0	1
0	1	0
0	1	1
0	1	X
0	X	1

(B) Symmetric input assignments

Figure 3.6. Symmetric Input Assignments

The probability of occurrence of a 0 or 1 at the primary inputs is initialized to be 0.5. Then, we recursively estimate the probability of occurrence of a value at the output of each gate assuming all the inputs are independent. For example, the probability of occurrence of 1 at the output of a 2 input AND gate with inputs a, b and output c is: $P(c = 1) = P(a = 1) * P(b = 1)$. These probabilities are estimated from the primary inputs to the output and updated for all the gates in the circuit. Note that the probabilities are measured only once and are computed similar to COP [48] in determining the controllability measures of gates in the circuit.

Let the probability of occurrence of a value at a gate g_i in the cut-set be $P(g_i)$. Assuming that the values at the individual gates occur independent of each other, the probability of occurrence of a particular cut-set (set of specified gates together) is

$$P(Cutset) = P(g_1) \times P(g_2) \times P(g_3) \times ... \times P(g_n)$$

After each ATPG decision, we compute the cut-set and estimate the probability of its occurrence. With this new probability as a metric, we can decide if we should store a cut-set or not. We store a cut-set only if it has a high probability of occurrence. Cut-sets with low probability of occurrence are ignored since they are deemed unlikely to be used again. Likewise, while searching for a cut-set in the hash-table, only cut-sets with higher probability of occurrence are considered. A cut-set with low probability of occurrence need not be searched in the hash-table since it would not have been stored in the first place. It may also be noted that the probability of occurrence of a cut-set decreases with an increase in its size, from the above equation. This explains the phenomenon that it is easier to obtain short cut-sets that are equivalent, whereas equivalent long cut-sets cannot be easily formed. Furthermore, the long cut-sets usually occur near the terminal nodes of the decision tree. Even if we ignore these cut-sets, the ATPG engine is likely to make the remaining few decisions to reach the terminal nodes without significant time overhead. On the contrary, cut-sets in the upper portions (near the root) of the decision tree are very important and would cost more if they are missed.

In order to reduce the number of cut-sets, we need to decide a CUTSET_THRESHOLD. All cut-sets with probability of occurrence less than the specified threshold will be ignored. However, deciding the CUTSET_THRESHOLD is non-trivial. In our techniques, we chose the probability of occurrence of the objective as the CUTSET_THRESHOLD. If a cut-set has a higher probability of occurrence than the objective, it is likely to occur often during ATPG. Experimental results show that the number of cut-sets stored is reduced, without much loss in speed of the ATPG engine.

3.6 Experimental Results

The above techniques were implemented in C++ and integrated into a PODEM based ATPG engine to generate all the solutions that satisfy a target objective. Success driven learning was integrated into the ATPG engine. In addition to the solution cut-sets suggested in [44], the conflict cut-sets were also stored to learn from the conflict subspaces encountered during search. Experiments were conducted to compute the one-cycle preimage for some large ISCAS '89 and ITC '99 benchmark circuits. Random conjectures of a set of flip-flops were chosen and set to random 0/1 values. The ATPG engine was used to find all the previous states that can lead to the target state in one transition. A backtrack limit of 1,000,000 was set for the ATPG engine. The experiments were conducted on a 1.8 GHz Pentium 4 machine with 512MB RAM, running the Linux Operating System.

The proposed techniques aim at enhancing an ATPG based framework for design verification problems and hence we compare our techniques with existing heuristics for an ATPG based framework only. We compare the results of our guidance heuristic with those of conventional heuristics - distance based (dst), COP (cop), and SCOAP (scp) measures [33] - in Tables 3.1 and 3.2. For each circuit, these three guidance heuristics and two of our heuristics were used individually for computing all solutions that satisfy a target objective. In our first heuristic - *scn*, only the static connectivity measures were used. In our second heuristic - *dcn*, the dynamic connectivity measures were updated after each decision and the backtrace routine was guided according to these measures. For each guidance heuristic, the number of solution cubes obtained, number of backtracks, number of Free BDD nodes used to store the solution-set, and the time taken for an 'all-solutions ATPG engine' are reported. Only large circuits are considered, since the number of BDD nodes in small circuits is usually small and the corresponding BDDs can be easily stored.

According to the results, it was observed that there is a reduction in the number of final BDD nodes for the proposed technique. In s3384-1, ATPG with Distance based and COP based measures fail to

Table 3.1. All Solutions ATPG for ISCAS '89 circuits

Circuit	Hrst	#soln	#bktr	#nd	T (s)	Circuit	Hrst	#soln	#bktr	#nd	T (s)
s3384 - 1	dst	464M	Abt			s5378 - 1	dst	0	361	0	0.28
	cop	19M	Abt				cop	0	59	0	0.26
	scp	1.8G	2.6K	2K	0.69		scp	0	93	0	0.26
	scn	1.9G	1K	396	0.43		scn	0	8	0	0.25
	dcn	1.7G	518	398	0.62		dcn	0	3	0	0.26
s3384 - 2	dst	0	Abt			s5378 - 2	dst	2M	3K	1.3K	0.71
	cop	0	Abt				cop	82K	1.1K	525	0.5
	scp	0	Abt				scp	218K	1.9K	779	0.58
	scn	0	Abt				scn	6.3M	887	392	0.49
	dcn	108M	158K	2.5K	78.58		dcn	299K	389	178	0.66
s3384 - 3	dst	0	Abt			s5378 - 3	dst	43K	302	303	0.42
	cop	0	Abt				cop	26K	521	505	0.44
	scp	1G	69K	57K	10.64		scp	32K	467	440	0.43
	scn	1.1G	16K	7K	2.19		scn	13K	286	278	0.41
	dcn	1.8G	9K	1.4K	4.7		dcn	19K	178	167	0.52
s13207 - 1	dst	798K	349	342	0.44	s15850 - 1	dst	308M	Abt		
	cop	241K	255	250	0.42		cop	2.4G	Abt		
	scp	467K	338	339	0.44		scp	605M	38K	27K	7.25
	scn	974K	782	784	0.51		scn	3.8G	Abt		
	dcn	952K	195	193	0.64		dcn	1.2G	313K	26K	59.92
s13207 - 2	dst	1.5G	Abt			s15850 - 2	dst	3.5G	Abt		
	cop	2.5G	Abt				cop	3.3G	Abt		
	scp	1.4G	40K	7K	9.24		scp	1.6G	Abt		
	scn	2.5G	Abt				scn	3.6G	Abt		
	dcn	889M	3.8K	2.7K	5.22		dcn	1.7G	177K	101K	269.99
s13207 - 3	dst	0	1526	0	1.32	s15850 - 3	dst	0	Abt	0	69.83
	cop	0	2220	0	1.45		cop	0	2	0	0.68
	scp	0	1836	0	1.41		scp	0	2	0	0.68
	scn	0	100	0	0.41		scn	0	1	0	0.71
	dcn	0	215	0	1.06		dcn	0	1	0	0.71
s9234 - 1	dst	537M	Abt			s38584 - 1	dst	104M	20K	20K	13
	cop	41M	43K	31K	5.19		cop	1G	Abt		
	scp	367M	34K	28K	4.32		scp	1.1G	Abt		
	scn	3.9G	Abt				scn	1.3G	51K	28K	29.71
	dcn	1.6G	8K	6.5K	7.81		dcn	1.4G	21K	8.5K	27.57
s9234 - 2	dst	0	1.9K	0	0.56	s38584 - 2	dst	1.4G	4.5K	4K	1.8
	cop	0	409	0	0.36		cop	1G	Abt		
	scp	0	1.6K	0	0.53		scp	1.5G	3.8K	3.2K	1.81
	scn	0	15	0	0.32		scn	1.2G	3.2K	3.1K	1.67
	dcn	0	19	0	0.34		dcn	1.5G	1.7K	1.7K	7.79
s9234 - 3	dst	0	42K	0	5.8	s38584 - 3	dst	3.5G	Abt		
	cop	0	Abt				cop	1.3G	Abt		
	scp	0	Abt				scp	3.7G	Abt		
	scn	0	24K	0	3.02		scn	1.2G	Abt		
	dcn	0	7K	0	6.28		dcn	1.9G	46K	45K	203.05

Note a) #solns: number of solution cubes
Note b) Different guidance heuristics lead to different solution cubes

Table 3.2. All Solutions ATPG for ITC '99 circuits

Circuit	Hrst	#soln	#bktrk	#nd	T (s)	Circuit	Hrst	#soln	#bktrk	#nd	T (s)
b04	dst	20K	3K	1.2K	0.49	b10	dst	118	107	85	0.24
	cop	32K	3.5K	2.2K	0.5		cop	33	60	45	0.23
	scp	5M	Abt				scp	36	59	50	0.24
	scn	26K	1.5K	617	0.34		scn	22	39	33	0.23
	dcn	27K	2K	906	0.71		dcn	31	46	37	0.23
b12 - 1	dst	936K	23K	10K	1.99	b13 - 1	dst	594	46	44	0.24
	cop	8.5K	1.8K	1.5K	0.36		cop	432	175	149	0.24
	scp	445K	38K	7.5K	3.3		scp	2K	225	207	0.24
	scn	128K	978	793	0.31		scn	1.2K	113	93	0.24
	dcn	478K	504	295	0.39		dcn	612	70	45	0.25
b12 - 2	dst	261K	Abt			b13 - 2	dst	36K	2K	1K	0.32
	cop	1.7K	6K	5K	0.65		cop	540	241	187	0.25
	scp	1M	66K	9.6K	6		scp	9.5K	2.6K	1.6K	0.35
	scn	51K	1.7K	740	0.38		scn	3.5K	128	103	0.23
	dcn	114K	764	338	0.44		dcn	2K	124	121	0.25
b12 - 3	dst	38K	3.3K	2.3K	0.52	b13 - 3	dst	112	177	106	0.34
	cop	8.8K	1.3K	1K	0.36		cop	21	91	78	0.25
	scp	16K	915	734	0.32		scp	72	133	96	0.24
	scn	30K	1.3K	795	0.35		scn	60	118	55	0.25
	dcn	131K	717	486	0.43		dcn	21	63	42	0.25

Note a) #solns: number of solution cubes
Note b) Different guidance heuristics lead to different solution cubes

find all the solutions. Although SCOAP based ATOG manages to find all the solutions, the number of BDD nodes is reduced by an order of magnitude, while using our connectivity based measures. In s38584-1, where SCOAP *fails* to find all the solutions, the connectivity based measures help to find all the solutions. In s38584-2, the number of backtracks is less for dcn than scn. However, longer cut-sets occur during ATPG and the estimation of dynamic-connectivity measures after each decision probably slows down the ATPG engine. But the pay-off is in the number of BDD nodes that is reduced to almost half when compared to scn. In some cases, such as s15850-2 and s33584-3, when all the other three methods fail, dcn is able to compute all the solutions. In Table 3.2, the results for ITC '99 circuits also confirm that we can obtain a compact solution-set using our guidance measures as compared to Dist, COP and SCOAP. Although the connectivity based measures decide on related variables together, the first set of variables that are targeted are not

Table 3.3. Efficiency of Our Cut-set metric

	Without Cut-set Metric					With Cut-set Metric				
ckt	#soln	#nd	#cutsets	#used	T (s)	#soln	#nd	#cutsets	#used	T (s)
s349	200	193	366	207	0.26	200	193	**101**	125	0.27
s444	24	65	81	19	0.25	24	65	**54**	18	0.24
s526	54	84	94	41	0.25	54	84	**43**	35	0.24
s4863	0	0	132K	8K	70.71	0	0	**125K**	8K	71.46
s1488	57	95	172	59	0.26	57	95	**3**	38	0.28
s1494	9	29	95	36	0.26	9	29	**7**	45	0.28
s1269	183K	113K	236K	120K	39.89	183K	113K	**148K**	204K	57.61
s1423	All	cut-sets	exhausted			330K	2.5K	**150K**	50K	64.73
s1423.1	912	810	123K	954	23.66	912	810	**110K**	825	23.10
s5378	805M	272	462	211	0.56	805M	272	**444**	211	0.56
s9234	All	cut-sets	exhausted			0	0	**200K**	190K	491.71
s9234.1	0	0	84K	56K	133.19	0	0	**77K**	59K	151.24
s15850.1	981M	3K	4.8K	2.7K	14.1	981M	3K	**2.2K**	8K	38.97
s15850	0	0	12K	10K	41.44	0	0	**10K**	11K	48.66
b04	6.5K	6.8K	102K	2.6K	254.2	6.5K	6.8K	**20K**	2.3K	282.49
b11	559	1.5K	15K	1.3K	35.72	559	1.5K	**189**	705	36.32
b12	65K	1.2K	1.2K	576	6.18	65K	1.2K	**1K**	576	6.18

Note a) #solns: number of solution cubes
Note b) Dyn-Con is used. Hence, the same #soln cubes are obtained for both techniques

tuned by these methods. These set of variables may result in a conflict or influence all the other solutions that occur during the ATPG. They are sometimes targeted by other heuristics, where they perform better than connectivity based measures as seen in s15850-1. In s15850-1, although the number of backtracks is less for SCOAP, we obtain a compact solution-set as good as the one obtained using SCOAP measures.

We conducted a second set of experiments to verify the effectiveness of the *cut-set metric* for reducing the cut-set storage. The ATPG environment was set up similar to the previous experiments to generate all solutions. In addition, the number of cut-sets we can store was limited to 300,000. After each decision, the cut-set was determined and its probability of occurrence was estimated. Cut-sets with probabilities less than a fixed CUTSET_THRESHOLD were ignored. *The probability of occurence of the objective was fixed as the CUTSET_THRESHOLD*. The corresponding results are tabulated in Table 3.3. For each circuit shown in column 1, columns 2-6 report the number of

solutions, number of BDD nodes, number of stored cut-sets, number of times the stored cut-sets are used and the time taken without using the cut-set metric. The number of times the stored-cutsets, as a whole, are used is counted every time a hash-hit for a cut-set is successful. Columns 7-11 report the results for the same ATPG by storing only the cut-sets with a high probability of occurrence.

For the objectives in smaller circuits like s349, s444 and s526, the cut-sets that are stored is reduced. However, since they are very small in number, all the cut-sets can be easily stored. On the other hand, in cases where large number of cut-sets are encountered, as in s1423.1, s4863, b04 and b11, the number of cut-sets is significantly reduced without significant loss in time. The main advantage in using the cut-set metric is exhibited in the cases of s1423 and s9234. Without using the proposed cut-set metric, all the cut-sets are exhausted and the ATPG engine fails to complete. The number of times the stored-cutsets are used varies according to the target objective in the two methods. Sometimes, longer cut-sets in the lower part of the decision tree are NOT stored in the second method. However, the ATPG quickly makes the fewer decisions to reach the terminal nodes. This results in fewer hash-hits in the second method without significant loss in time as seen in b11. On the other hand, some cut-sets that are used fewer times in the first method are not available to the second method. So the available cut-sets are used many times in the second method, leading to an increase in the number of times the stored-cutsets are used, as seen in s1269 and s15850.1.

Although the metric is helpful to obtain a probabilistic estimate for the use of a cut-set, a low value does not completely guarantee that the cut-set will not be re-used again. This uncertainty may lead to losing useful cut-sets occasionally. In such cases, the ATPG engine may have to re-search a previously visited search space and take more time as seen in s15850.1. This phenomenon may increase as the size of the circuit increases due to correlation of gates in the circuit.

3.7 Summary

In this chapter, we explored the factors that govern an all-solutions ATPG engine. While conventional guidance heuristics for ATPG are suitable for finding a single solution quickly, they do not account for the complete-solution set. We presented a new *connectivity based guidance heuristic* for an 'all-solutions ATPG engine' that reduces the size of the Free BDD to store all solutions. We also developed a probabilistic *cut-set metric* to determine the future use of stored information, learnt during success driven learning. Based on the cut-set metric, we showed that the knowledge base can be reduced depending on the CUTSET_THRESHOLD determined for the cut-set. Experimental results show that we can achieve significant reductions in memory and our techniques can guide the ATPG engine to find all the solutions where conventional guidance heuristics fail.

Chapter 4

State Set Management

In this chapter, we propose a simple, yet efficient, decision diagram to circuit/CNF conversion technique to manage the state set obtained during iterative preimage computation [68]. In the case of ATPG/SAT, the state set is stored as a decision diagram like Free BDD (FBDD) / Zero-suppressed Binary Decision Diagram (ZBDD). We use the shared structures in the decision diagram to aggressively avoid repeated manipulation of common subsets in the state-set. In the case of SAT-based Unbounded Model Checking, the resulting number of clauses, generated for the state set, now depends on the number of nodes in the ZBDD, rather than the number of solutions found. To evaluate the efficacy of the proposed technique, we integrated the proposed techniques in an Unbounded Model Checking framework that uses a pure SAT solver. The experimental results show that we can attain orders of magnitude improvement in both performance and capacity as compared to the existing techniques.

4.1 Introduction

As discussed in the earlier chapters, it is generally agreed that symbolic methods are widely used for Model Checking sequential hardware systems. The representation and manipulation of Boolean expressions are critical for the scalability and robustness of Symbolic Model Checking. One of the

questions that is of interest in Symbolic Model Checking is: *"Can we reach the target state (could be a buggy state) from the initial state in a sequential design?"* Traditionally, Reduced Ordered Binary Decision Diagrams (ROBDDs) have been used for this purpose, since they are canonical and they can be efficiently manipulated. However, they suffer from potential memory explosion for large designs. Boolean Satisfiability (SAT) has received significant attention in recent years, and it offers an alternative to BDD-based methods.

It should be noted that the ATPG based framework discussed in previous chapters is also similar to the SAT-based framework. Since we are dealing with the iterations of pre-image computation, the core pre-image computation engine can be a SAT solver that stores the state-set as a Zero-suppressed Binary Decision Diagram (ZBDD) or an ATPG engine that stores the state-set as a Free Binary Decision Diagram (FBDD). In this chapter, we discuss our contribution with respect to a SAT based framework due to its higher impact on the preimage computation problem. We can easily interpret the FBDD to circuit conversion technique in terms of our ZBDD to clause conversion technique. Nevertheless, at appropriate points in the discussion, we note the analogy to our ATPG based framework.

We use McMillan's [45] Unbounded Model Checking framework to demonstrate our experimental results. In SAT-based UMC [45, 46, 69], we compute the preimage iteratively until either the desired state or fixed point is reached. The major factors that contribute to an efficient SAT-based UMC framework are: (i) efficient learning techniques in the base-line solver, (ii) enlarged state space captured by each solution and, (iii) compact representation and efficient manipulation of image/preimage across time-frame iterations.

In this work, we target the SAT-based UMC problem. Similar to [45], we store both the preimage and reached states, in a Zero-suppressed Binary Decision Diagram (ZBDD). Two novel features in our approach are:

- We consider different configurations of a ZBDD node and derive the corresponding clause

representations.

- We propose a simple, yet efficient, node-based clause conversion scheme to *compactly* represent the state-set.

Subsequently, the resulting number of clauses generated for the state set depends on the number of nodes in the ZBDD, rather than the number of solutions found. Experiments show that we can obtain orders of magnitude reduction in both the number of state-set clauses and time taken for UMC.

4.1.1 Previous Work

Initially, certain variants to Binary Decision Diagrams were proposed for image/preimage computation. In [38, 37], non-canonical structures such as Reduced Boolean Circuit (RBC) and Boolean Expression Diagrams (BED) are used. The transition relation of the circuit is represented as an RBC/BED and the authors propose efficient quantification rules for specific sub-structures. However, in the general case, the length of formulas may grow exponentially due to the quantification. In [36], SAT solvers are integrated with BDDs to perform image computation. The SAT solvers are used to provide a disjunctive decomposition of clauses and BDDs are used subsequently to solve the end problems.

In [45], McMillan proposed a basic preimage computation technique for unbounded model checking using a pure SAT solver. The transition relation and negation of the property are represented as a set of clauses, and variable quantification is performed during solution enumeration. After each solution is found, an enlarged cube is identified by re-building the implication graph. The negation of this cube is added as a *blocking clause* to the clause database. The set of all blocking clauses represent the preimage and they are stored in a Zero-suppressed Binary Decision Diagram (ZBDD). A major bottleneck in SAT-based UMC is the *solution explosion* problem that occurs due to the huge

number of solutions/blocking clauses. In order to alleviate this problem, the ZBDD inherently performs a light-weight optimization of the clauses on the fly. In [46], the authors use a simple circuit-based justification procedure to identify enlarged solution cubes (and avoid re-building the implication graph). The preimage is stored as a list of cubes and periodically fed to Espresso for logic minimization to reduce the total size of the set of solution cubes/blocking clauses.

In [44], an ATPG engine is used for preimage computation. The transition relation is represented as a Boolean circuit and an ATPG engine is invoked to enumerate all the solutions that represents the complete preimage. The basic ATPG engine is accelerated using Success-Driven Learning that avoids re-searching overlapping solution sub-spaces. The preimage is stored as a free BDD. Further, success-driven learning was later integrated into a hybrid SAT-solver to take advantage of both circuit based success-driven learning and clause based conflict-driven learning in [65]. In [70, 71], the authors use an implicit time-frame expansion of the sequential circuit and integrate efficient heuristics in the SAT solver for state space traversal.

In a recent approach for SAT-based UMC in [69], the transition relation is represented by an or-inverter graph and efficient solution enumeration is performed using a specialized hybrid SAT solver. After obtaining each solution, the authors use the circuit cofactors, with respect to the input assignment, as the enlarged state space. The circuit cofactors can capture a larger space than the enlarged solution cubes. The preimage is stored incrementally as a non-canonical or-inverter graph that uses function hashing and two level minimization on the fly. The size of or-inverter graph will increase with each preimage computation step / iteration in the UMC framework. In their experiments, they unroll the time-frames for preimage computation, instead of using their state set representation. Overall, their experiments show a significant run-time improvement over the technique in [45], since they capture a larger state space at each enumeration.

We introduce the preliminaries and our basic UMC framework in section 4.2. In section 4.3, we explain the new clause conversion technique to represent the state-set. Experimental results for

ISCAS '89 and ITC '99 circuits are presented in section 4.4. Finally, we summarize the paper in section 4.5.

4.2 Preliminaries

In this section, we give a brief outline of SAT-based UMC and ZBDD to understand the rest of the paper. For a thorough exposition of UMC, Computational Tree Logic (CTL), and SAT-based UMC, we refer the reader to [11, 45]. Before we go into the details of UMC, we provide certain definitions for Boolean formula representations.

Definition 1 *Literal: It is a variable in either its positive or negative form.*

Definition 2 *Conjunctive Normal Form (CNF): It is a Boolean formula $\phi(V)$ that is expressed as a conjunction of clauses, where each clause is a disjunction of literals.*

Definition 3 *Disjunctive Normal Form (DNF): It is a Boolean formula $\phi(V)$ that is expressed as a disjunction of cubes, where each cube is a conjunction of literals.*

Definition 4 *Equivalence: Two Boolean formulas $\phi_1(V)$ and $\phi_2(V)$ are said to be equivalent, if and only if for every assignment $\alpha \mapsto V$, $\phi_1(\alpha) \Leftrightarrow \phi_2(\alpha)$.*

Definition 5 *Equisatisfiability: Two Boolean formulas $\phi_1(V_1)$ and $\phi_2(V_2)$ are said to be equisatisfiable, if and only if for every assignment $\alpha_1 \mapsto V_1$ that satisfies ϕ_1, there exists an assignment $\alpha_2 \mapsto V_2$, that satisfies ϕ_2, and vice-versa.*

It should be noted that any Boolean formula can be converted into CNF. Consider a Boolean formula: $(a \wedge b) \vee c$ An equivalent CNF representation is $(a \vee c) \wedge (b \vee c)$, which is obtained by using the distributive law of Boolean algebra. An equisatisfiable CNF representation is $(v \vee c) \wedge (\neg v \vee a) \wedge (\neg v \vee b) \wedge (v \vee \neg a \vee \neg b)$. The equisatisfiable CNF is obtained by introducing an auxiliary variable $v = (a \wedge b)$. When we need to negate a Boolean formula, it is generally easier to obtain an

equisatisfiable CNF. For example, suppose we wish to negate the Boolean formula $(a \wedge b) \vee c$. First, we assign auxiliary variables, $v = (a \wedge b)$ and $F = (v \vee c)$. Then, we obtain the clauses for these assignments. Finally, we add the unit clause $(\neg F)$. In special cases, where the Boolean formula is in DNF, we can obtain its negation by simply negating each cube in the DNF to form the clauses of an equivalent CNF. *Note that the support variables in equivalent Boolean formulas are the same, whereas the support variables for equisatisfiable ones can be different.* Also, equivalence implies equisatisfiablity, but the reverse is not true.

4.2.1 SAT-based UMC

In order to perform SAT-based UMC, it is sufficient to have an algorithm that performs preimage computation iteratively, until a fixed point is reached. The algorithms to evaluate the CTL properties such as AX, AF, AG, EX, EF, EG can be derived using different characterizations of this algorithm.

A Finite State Machine (FSM) can be represented by the 6-tuple: $< I, O, S, \delta, \lambda, S_0 >$, where $I, O, S, \delta, \lambda, S_O$ represent the primary inputs, primary outputs, state set, next state function, output function and initial states, respectively. Symbolically, the preimage for a given set of next states, $S(X')$, is given by:

$$Preimage(X): \ \exists_{I,X'} \ T(X, I, X') \ \wedge \ S(X')$$

X represents the present state variables

X' represents the next state variables

T represents the transition relation

Suppose we have a Boolean formula, $\phi(V)$, that depends on variables in $V = \{v_1, v_2, ..., v_n\}$. The Boolean Satisfiability problem (SAT) consists of determining whether $\phi(V)$ is satisfiable or not.

Most of the present day SAT solvers employ DPLL [21] procedures on the Boolean formula that is in Conjunctive Normal Form (CNF). They can return a *satisfying assignment/solution*, if $\phi(V)$ is satisfiable. Basically, they perform existential quantification on all the variables in V. A single solution SAT solver can be augmented to find all the solutions that satisfy $\phi(V)$ [45], as follows:

1. After each solution cube is enumerated, add its negation as a *blocking clause* and solve for SAT incrementally.
2. Repeat Step 1, until no more solutions are obtained.

Suppose we get a solution (α) for a Boolean formula ($\phi(V)$) that represents an FSM. Then, a subset of inputs and present state variables ($V_I \subset V$) can imply the values of all the outputs and next state variables ($V_O \subset V$) in α. In other words, it is sufficient to block the values of those variables in V_I. Further, in preimage computation, because we are interested in the state variables only, we can use the conjunction of present state variables in V_I as an *enlarged solution cube* in Step 1. This enlarged solution cube can be identified by redrawing the implication graph [45] or circuit justification [46].

If $\phi(V) = T(X,I,X') \wedge S(X')$, then the set of enlarged *solution cubes* represents the preimage as a DNF. This preimage corresponds to the set of states that can reach $S(X')$ in one cycle/iteration. In UMC, we are also interested in searching all the states that can reach $S(X')$ in multiple iterations. So, we iterate until no new preimage states are obtained, i.e., fixed point is reached.

The Preimage in iteration j, $Iter_j$ is given by

$$\exists_{I,X'}\, T(X,I,X') \wedge S_j(X'/X) \wedge \neg R_j(X) \tag{4.1}$$

$S_0(X')$: initial set of next states

$S_j(X') = Preimage_{j-1}(X'/X)$: new states obtained in $Iter_{j-1}$

X'/X : next state variables substituted for present state variables

$R_j(X) = \bigvee_{k=0}^{j-1} Preimage_k(X)$: set of states reached upto $Iter_{j-1}$

$Note: Here, Preimage_j(X)$ is set of new states obtained in $Iter_j$

On the other hand, if $\phi(V) = T(X,I,X') \wedge \neg S(X')$, then the set of *blocking clauses* (negation of solution cubes) represents the preimage as a CNF. Then, we iterate using the following formula in Iteration j:

$$\forall_{I,X'}\, T(X,I,X') \wedge \neg S_j(X'/X) \wedge R_j(X) \tag{4.2}$$

$S_j(X') = Preimage_{j-1}(X'/X)$: new states obtained in $Iter_{j-1}$

$R_j(X) = \bigwedge_{k=0}^{j-1} Preimage_k(X)$: set of states reached upto $Iter_{j-1}$

The latter approach is used in [45]. We use the former approach in our experiments. We do not claim this as a major contribution of our work, but point out that both the techniques will lead to the same preimage, since they are equivalent. It is easier to see the equivalence of Formulas (4.1) and (4.2), if we consider $T(X,I,X')$ as a sequential circuit, $S_j(X')$ as the objective and $R_j(X)$ as the restriction on the solutions.

Most of the existing SAT solvers operate on a Boolean formula in CNF. In Formula (4.1), $S_j(X')$ is represented by an equisatisfiable CNF by adding an auxiliary variable for each cube. $\neg R_j(X)$ is represented by an equivalent CNF, since the negation of the DNF is an equivalent CNF. Similarly, in Formula (4.2), $\neg S_j(X')$ is represented by an equisatisfiable CNF and $R_j(X)$ is represented by an equivalent CNF. So, it is admissible to use any one of the above two approaches without a theoretical change in the problem's complexity.

4.2.2 State-set Decision diagram

In this section, we discuss the basics of a Zero-suppressed Binary Decision Diagram (ZBDD) and a Free Binary Decision Diagram (FBDD), which are used in the SAT-based UMC framework and the ATPG-based UMC framework respectively.

Zero-suppressed Binary Decision Diagram (ZBDD)

In [72], Minato introduced ZBDDs to represent *sets of combinations* compactly and perform set operations efficiently. Sets of combinations $S = \{\{a,b\},\{\neg a,c\},\{b,c\}\}$ are represented in a ZBDD as shown in Figure 4.1. Each path from the root to TERMINAL_ONE represents a set in the ZBDD. A 1-edge from a node denotes the presence of the element in the set and a 0-edge denotes its absence.

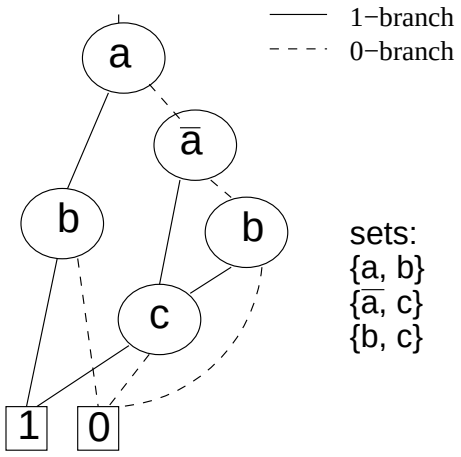


Figure 4.1. Zero-suppressed Binary Decision Diagram (ZBDD)

In [45], the state-set that is stored in a ZBDD is converted into a CNF as follows. Suppose the ZBDD shown in Figure 4.1 represents the $Preimage_{j-1}(X) : (a \vee b) \wedge (\neg a \vee c) \wedge (b \vee c)$ for $Iter_{j-1}$. In $Iter_j$, we need to add $\neg S_j(X')$ as shown in Formula (4.2). First, we substitute the present state variables with next state variables to get $S_j(X')$. Then, we add an auxiliary variable for each clause to obtain an equisatisfiable CNF for: $(V_1 = a' \vee b') \wedge (V_2 = \neg a' \vee c') \wedge (V_3 = b' \vee c') \wedge (V = V_1 \wedge V_2 \wedge V_3) \wedge (\neg V)$. If the ZBDD represents $R_j(X)$, we can simply add the equivalent CNF:$(a \vee b) \wedge (\neg a \vee c) \wedge (b \vee c)$. Note that the number of clauses generated for the state set depends on the number of paths in the ZBDD. We call this technique as the *path-based* conversion technique.

Free Binary Decision Diagram (FBDD)

The free-BDD is a relaxed version of a ROBDD. It is a generalized BDD, where a variable can occur only once along each path from the root to a terminal. The decision tree obtained during all-solutions ATPG is essentially a free BDD. The Free BDD, shown in Figure 4.2, represents the states-set as cubes: $\{(abc), (a\neg b), (\neg acb)(\neg a\neg cd)\}$.

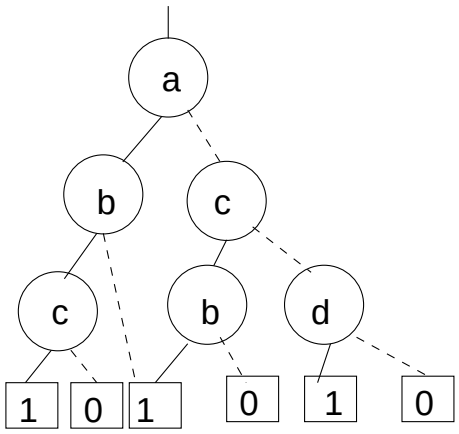


Figure 4.2. Free Binary Decision Diagram (ZBDD)

4.3 State-set Management

In this section, we propose a new clause conversion technique that generates clauses for each node in the ZBDD. This can lead to a drastic reduction in the total number of clauses generated, as compared to the technique used in [45].

4.3.1 Basic Idea

Basically, a state-set ZBDD represents a particular function in CNF/DNF that depends on the state variables only. The clauses/cubes that are represented by the ZBDD generally have a lot of sub-

clauses/sub-cubes in common. They are *shared* as sub-ZBDDs in the state-set ZBDD. However, when we represent the state-set as clauses using the path-based conversion technique, we do not make use of this sharing in the CNF representation.

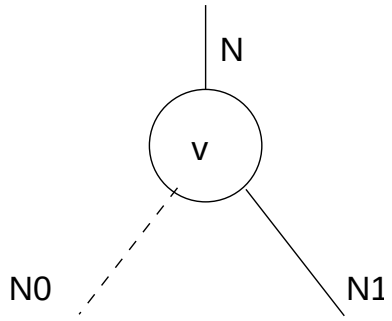


Figure 4.3. ZBDD Node

On the other hand, each node in the ZBDD represents a particular function that depends on the node index (v), 0-branch sub-ZBDD (N_0) and 1-branch sub-ZBDD (N_1) - see Figure 4.3. Let F_0 and F_1 be the functions represented by N_0 and N_1 respectively. If the ZBDD represents a state set in DNF as computed by Formula (4.1), then the function represented by this node can be recursively defined as:

$$F = (v \wedge F_1) \vee F_0 \tag{4.3}$$

This can be represented using the following clauses in CNF:
$(\neg F \vee v \vee F_0) \wedge (\neg F \vee F_1 \vee F_0) \wedge (\neg v \vee \neg F_1 \vee F) \wedge (\neg F_0 \vee F)$. With the base conditions: TERMINAL_ONE=true and TERMINAL_ZERO=false, we can derive the functions for all the nodes by a bottom up traversal in the ZBDD. If the state set is in CNF as computed by Formula (4.2), then

$$F = (v \vee F_1) \wedge F_0 \tag{4.4}$$

Now, this can be represented using the following clauses:
$(\neg F \vee v \vee F_1) \wedge (\neg F \vee F_0) \wedge (\neg v \vee \neg F_0 \vee F) \wedge (\neg F_1 \vee \neg F_0 \vee F)$. Again, with the base conditions: TERMINAL_ONE=false and TERMINAL_ZERO=true, we can likewise derive the functions for all the nodes in the ZBDD.

Table 4.1. Cases for a ZBDD node

Case	N_0	N_1	nodeDNF, F	nodeCNF, F
1.	T_0	T_0	$false$	$true$
2.	T_0	T_1	v	v
3.	T_1	T_0	$true$	$false$
4.	T_1	T_1	$true$	$false$
5.	T_1	Int	$true$	$false$
6.	Int	T_0	F_0	F_0
7.	T_0	Int	$v \wedge F_1$	$v \vee F_1$
8.	Int	T_1	$v \vee F_0$	$v \wedge F_0$
9.	Int	Int	$(v \wedge F_1) \vee F_0$	$(v \vee F_1) \wedge F_0$

T_O - TERMINAL_ZERO; T_1 - TERMINAL_ONE; Int - Internal Node

The children (nodes) of an *internal* node may be TERMINAL_ONE, TERMINAL_ZERO or another internal node. Since there are two children for any given internal node and there are three possibilities for each child, we have to consider $3^2 = 9$ configurations. We tabulate them in Table 4.1 to implement in an algorithm that is described next. However, it is possible that some of these cases are avoided in certain implementations of the ZBDD package.

4.3.2 Algorithm

In Algorithm 4, we provide the procedure to convert a ZBDD, representing a set of cubes, to an equisatisfiable CNF for Formula (4.1). The procedure takes a ZBDD node as its input argument, creates the clauses corresponding to all the nodes in the ZBDD and returns the function variable that represents the disjunction of cubes in the ZBDD.

The conjunction of clauses for each node in the ZBDD will lead to an equisatisfiable CNF for the state-set. The variable corresponding to the root node represents the function for the state-set, say F_s. We traverse the ZBDD in a depth-first manner and generate clauses and a CNF variable for each node, recursively, in a bottom up fashion. If the node is a terminal node, then we return dummy CNF variables, CNF_ONE/CNF_ZERO to represent true/false. Otherwise, we have an

internal node and we can use Table 4.1 to identify the configuration of the internal node.

Algorithm 4: DNF-ZBDD to equisatisfiable CNF

```
1  zbdd2cnf(node){
2      if (node == TERMINAL_ONE) then
3          return CNF_ONE ;
       end
4      if (node == TERMINAL_ZERO) then
5          return CNF_ZERO ;
       end
6      cnf_var := check_cache(node) ;
7      if (cnf_var != NULL) then
8          return cnf_var ;
       end
9      cnf_var0 := zbdd2cnf (0_branch(node)) ;
10     cnf_var1 := zbdd2cnf (1_branch(node)) ;
       /*Trivial cases 1-6 in Table 4.1 */
11     if (cnf_var0 == CNF_ONE) then
12         return CNF_ONE ;
       end
13     if (cnf_var1 == CNF_ZERO) then
14         return cnf_var0 ;
       end
15     if (cnf_var0 == CNF_ZERO && cnf_var1 == CNF_ONE) then
16         return index(node) ;
       end
       /*Generate clauses for the node */
17     cnf_var := nodeCls (index(node), cnf_var0, cnf_var1) ;
18     storeInCache(node, cnf_var) ;
19     return cnf_var ;
   }
```

First, we check the cache to see if the node was traversed earlier. If yes, we use the previously computed CNF variable for the node. Otherwise, we get the CNF variables for the 0-branch and 1-branch of the node. In lines 11-16, we eliminate some trivial cases: Cases 1-6 in Table 4.1. It is not necessary to create new clauses/variables for these cases. In line 17, we create the clauses

corresponding to the node: Cases 7-9 in Table 4.1. We store the CNF variable corresponding to the function of the node in a hash table for reuse in the future. At the end of recursion, the algorithm will return the CNF variable, F, corresponding to the function of the state set.

Algorithm 5: Clauses for a DNF-ZBDD node

```
1  nodeCls(node_index, cnf_var0, cnf_var1){
2      v := node_index ;   F :=createNewVar() ;
3      F0 := cnf_var0 ;   F1 := cnf_var1 ;
       /*Case 7:  0-branch is TERMINAL_ZERO */
4      if (cnf_var0 == CNF_ZERO) then
5          addClause(¬F ∨ v) ;
6          addClause(¬F ∨ F1) ;
7          addClause(¬v ∨ ¬F1 ∨ F) ;
8          return F ;
       end
       /*Case 8:  1-branch is TERMINAL_ONE */
9      if (cnf_var1 == CNF_ONE) then
10         addClause(¬F ∨ v ∨ F0) ;
11         addClause(¬v ∨ F) ;
12         addClause(¬F0 ∨ F) ;
13         return F ;
       end
       /*Case 9:  In General */
14     addClause(¬F ∨ v ∨ F0) ;
15     addClause(¬F ∨ F1 ∨ F0) ;
16     addClause(¬v ∨ ¬F1 ∨ F) ;
17     addClause(¬F0 ∨ F) ;
18     return F ;
   }
```

If the ZBDD represents the next state set, then we add this CNF variable as a unit clause, (F_s). If the ZBDD represents the reached state set, then we add the negation of this CNF variable ($\neg F_s$) - see Formula (4.1). It is not hard to interpret a similar algorithm that converts a ZBDD, representing a set of clauses, to an equisatisfiable CNF for Formula (4.2) as well.

In Algorithm 5, we elaborate the procedure called in line 17 of Algorithm 4. The procedure takes

the details of an internal node that belongs to Cases 7-9 in Table 4.1, creates the clauses and returns the function variable for the node. We check the three cases and generate corresponding clauses for the node configurations in Table 4.1.

4.3.3 An Example

We now illustrate our idea with the help of an example, when using Formula (4.1) to iterate in the UMC framework. Let the preimage (set of cubes) obtained in $Iter_{j-1}$ be stored in the ZBDD shown in Figure 4.4. For $Iter_j$, we substitute the present state variables with next state variables and convert the resulting set of cubes into an equisatisfiable CNF.

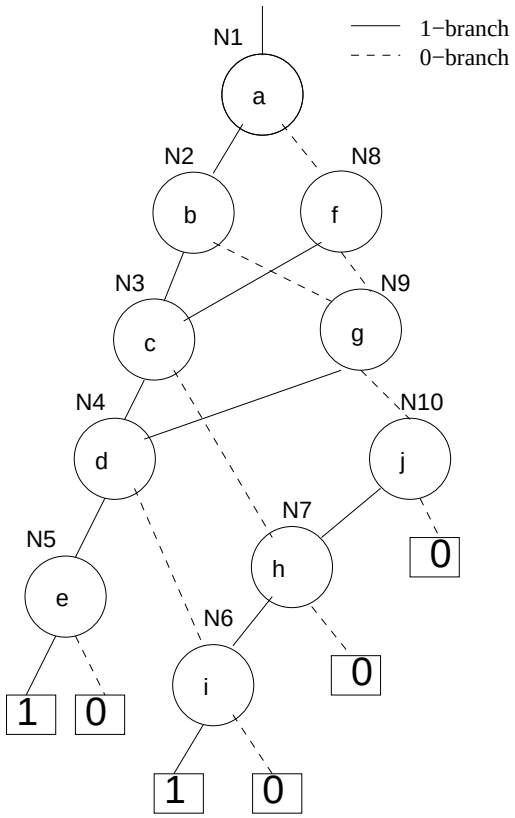


Figure 4.4. Preimage ZBDD

We compare the number of auxiliary variables, clauses and literals to represent the state set using the previous path-based clause conversion with those of our technique in Tables 4.2 and 4.3. In the case of the conventional path-based clause conversion, we need to add an auxiliary variable for each cube. So the number of auxiliary variables is 12. Correspondingly, the number of clauses and the literals generated for each cube is tabulated in Table 4.2. For example, in $V_{10} = g'.i'$, we add the 3 clauses $(\neg V_{10} \vee g'), (\neg V_{10} \vee i'), (V_{10} \vee \neg g' \vee \neg i')$ that contains 7 literals. As reported in the last row in the table, we need 12 auxiliary variables, 61 clauses and 159 literals to represent the state set. On the other hand, in our node-based clause conversion, we assign the variable V_k to node N_k in the ZBDD.

No new clauses, variables or literals are necessary for nodes N_5 and N_6 since they can be simply represented by e and i, respectively. For the other nodes, the results are estimated based on their configurations in Table 4.1. The last row shows that we need 8 auxiliary variables, 30 clauses and 80 literals to represent the state set.

In this example, we are able to reduce the number of clauses and number of literals to approximately half. For larger ZBDDs, we will be able to obtain even larger reductions as the sharing of nodes in the ZBDD increases.

4.3.4 Complexity Analysis

In order to compare Algorithm 4 with the path-based clause conversion used in [45], we perform a simple analysis on the run time complexity, number of clauses generated and the number of auxiliary variables necessary to represent the state set for the two approaches.

In Algorithm 4, we traverse each node only once. So the run time complexity of the algorithm is linear to the number of nodes in the ZBDD, i.e., $O(n)$, where n is the number of nodes in the ZBDD. To estimate the number of clauses generated for the ZBDD, we use the cases in Table 4.1.

Table 4.2. Path based ZBDD to CNF

Auxiliary var	#C	#L
$V_1 = a'.b'.c'.d'.e'$	6	16
$V_2 = a'.b'.c'.i'$	5	13
$V_3 = a'.b'.h'.i'$	5	13
$V_4 = a'.g'.d'.e'$	5	13
$V_5 = a'.g'.i'$	4	10
$V_6 = f'.c'.d'.e'$	5	13
$V_7 = f'.c'.i'$	4	10
$V_8 = f'.h'.i'$	4	10
$V_9 = g'.d'.e'$	4	10
$V_{10} = g'.i'$	3	7
$V_{11} = j'.h'.i'$	4	10
$V = \bigvee_{k=1}^{11} V_k$	12	34
12	**61**	**159**

Table 4.3. Node based ZBDD to CNF

Auxiliary var	#C	#L
$V_1 = a'.V_2 + V_8$	4	11
$V_2 = b'.V_3 + V_9$	4	11
$V_3 = c'.V_4 + V_7$	4	11
$V_4 = d'.e' + i'$	4	11
$V_7 = h'.i$	3	7
$V_8 = f'.N_3 + N_9$	4	11
$V_9 = g'.N_4 + N_{10}$	4	11
$V_{10} = j'.N_7$	3	7
8	**30**	**80**

The number of clauses added is equal to: $3 \times (C_7 + C_8) + 4 \times C_9$, where C_i represents the number of nodes that belong to Case i in Table 4.1. An upper bound on the number of clauses is thus $4n$. Finally, the number of auxiliary variables created for the ZBDD is equal to: $C_7 + C_8 + C_9$. An upper bound on the number of auxiliary variables added to the CNF is n.

On the other hand, the path-based clause conversion procedure used in [45] depends on whether the state set represents reached states or next-state set (see Example in section 4.2). Basically, clause(s) are added for each path in the ZBDD. Therefore, the complexity is linear to the number of paths in the ZBDD, i.e., $O(m)$, where m is the number of paths in the ZBDD that lead to TERMINAL_ONE. For reached state CNF in Formula (4.1), we add one clause for each 1-path (a path ending at TERMINAL_ONE). So the number of clauses is m. For the next-state CNF, we assign a new variable for each 1-path and add the corresponding clauses. The number of clauses added is $\sum_{j=1}^{m}(|p_j|+1)+m+1$, where $|p_j|$ represents the length of the 1-path j. An upper bound for the number of clauses added is $(l+2) \times m + 1$, where l is the length of the longest 1-path. Finally, no auxiliary variables are created for the reached state CNF. For next-state CNF, the number of auxiliary variables created is m.

It is shown that the number of clauses generated by our approach depends on the number of nodes in the ZBDD, whereas the number of clauses generated by [45] depends on the number of paths in the ZBDD. Therefore, our approach can potentially lead to an exponentially smaller number of clauses, since the number of paths can be exponential to the number of nodes in a ZBDD. For the cases where the number of paths is linear to the number of nodes in the ZBDD, the savings would not be as significant. For example, a single cube in a ZBDD, where the number of clauses generated by the path-based clause conversion [45] can be fewer than the our approach. Nevertheless, this is the uncommon case for our application, since ZBDDs were chosen for the very purpose of exploiting the significant amount of sharing among solutions computed.

In general, reducing the number of clauses may not necessarily lead to an improved performance for SAT solving. However, in our case, we are targeting at reducing the number of clauses by considering the sharing of cubes/clauses in the ZBDD. In other words, for any sub-ZBDD structure that is shared s times in the state set, we will create the CNF for this structure only once and represent it by a single variable. In the path-based clause conversion, this sub-ZBDD structure will be traversed s times. In a sense, the time spent on these clauses, by the SAT solver, can be reduced by a factor s. Another attractive feature of our node-based clause conversion is that the clauses are always of length 3 or 4. On the other hand, the clauses generated by the path-based conversion depends on the size of cube, which can be very large. Generally, smaller clauses help to boost the performance of a SAT solver than larger clauses.

4.3.5 FBDD State-set Management

In the case of an ATPG-based framework [44], we need to convert a free BDD into a circuit. It should be noted that we can similarly derive the function of each node in a free BDD as well. It is well known that each node in a BDD can be converted into a multiplexer. Basically, we need to replace each node with a multiplexer. The multiplexer output of the root node will be the state-set

circuit output function. We omit the details of this conversion, since it will be a repetition of the basic concepts that lead to the ZBDD to CNF conversion.

4.4 Experimental Results

We implemented a SAT-based Unbounded Model Checking framework in C++. The preimage is obtained using the formulation in Formula (4.1) until *fixed point* is reached.

Table 4.4. Preimage Computation: Solution cube enlargement using Blocking Clauses

Ckt	Path-based Conv. in [45]			Our approach		
	dep	Pcls	Time(s)	dep	Pcls	Time(s)
s349	3	62K	13.19	3	3K	1.13
s420.1	573	51K	1548.54	573	2K	6.74
s499	2	886K	811.5	2	108K	767.5
s991	2*	M_O	*	3	9K	1747.4
s5378	5	740K	1503.04	5	45K	293.48
s38417	1*	M_O	*	2	248	517.61
b08	2*	M_O	*	21	6K	3982.68
b10	10	138K	3354.61	10	9K	50.8
b12	9	171K	1544.71	9	9K	91.65
b14	2*	M_O	*	3	634	1765.72

*M_O - Memory Out (1M clauses); * - incomplete*

We used the recently developed MINISAT [27] as the baseline SAT solver; however, other solvers may be used as well. After a solution is obtained, we enlarge the solution to obtain the preimage cube and then block the preimage space. For solution enlargement, we implemented two well-known methods: blocking clause [45] and circuit cofactoring [69]. We use circuit justification [46] to obtain the blocking clause. For circuit cofactoring, we store the resulting cofactor as a set of equivalent clauses in a ZBDD. We used CUDD package [18, 73] to manipulate the state set ZBDD and ran experiments on some abstracted models of ISCAS '89 and ITC '99 benchmark circuits. We chose some random cubes to define a set of target states. Then, we computed the preimage of these states until fixed point is reached. The experiments were performed on a Pentium IV, 3GHz

machine running Redhat Linux Operating System. We limit the number of clauses that can be handled by the SAT solver to 1 million.

Table 4.5. Preimage Computation: Solution cube enlargement using Circuit Cofactoring

Ckt	Path-based Conv. in [45]			Our approach		
	dep	Pcls	Time(s)	dep	Pcls	Time(s)
s349	3	37K	1.68	3	5K	0.53
s420.1	573	51K	1550.78	573	2K	11.63
s499	2	8K	0.02	2	1K	0.01
s991	2*	M_O	*	3	2K	98.07
s5378	5	534K	393.58	5	3K	40.22
s38417	1*	M_O	*	2	248	5.67
b08	2*	M_O	*	21	9K	4866.93
b10	10	175K	6627.77	10	7K	18.32
b12	9	71K	345.41	9	3K	6.71
b14	2*	M_O	*	3	634	1880.67

*M_O - Memory Out (1M clauses); * - incomplete*

In Tables 4.4 and 4.5, we report the results, when we use blocking clauses and circuit cofactoring for solution cube enlargement respectively. The first column shows the circuit name. Columns 2, 3 and 4 report the depth of state space traversal / #iterations, peak number of clauses generated for the state set and time taken when we use the previous path-based clause conversion to convert the state set into CNF. Columns 5, 6 and 7 show these results for the proposed approach. We achieved two orders of magnitude reduction in the number of clauses for circuits b10 and b12 in Table 4.4 and, s5378 and b10 in Table 4.5. In most of the cases, we obtained an order of reduction in the number of clauses. For s991, s38417, b08 and b14 in both the Tables, the SAT solver aborted when we use the path-based conversion due to capacity limits. On the other hand, we are able to complete all the iterations until fixed point is reached. A similar phenomenon is seen in the time taken to compute all the reachable states. We see orders of magnitude improvement in time for circuits: s420.1, s5378, b10 and b12 in both the Tables 4.4 and 4.5. In most of the cases, the cofactoring based approach performs better than the blocking clause approach. Further, it may be possible to get a better performance by using a hybrid SAT solver. Albeit, the size of the or-inverter

graph for the state set representation may still be large as compared to a ZBDD. The state-set or even the unrolled transition relation may require large or-inverter graphs when we encounter very deep circuits like s420.1. Since the benchmarks and the hybrid solver experimented in [69] are not publicly available, we limit our experimentation to a pure SAT solver only.

To see the consistency of clause reduction across different iterations, we plotted the number of state set clauses generated for the circuit b10, on a *log scale*, in Figure 4.5. The top two lines show the plot for the previous path-based conversion using blocking clauses and circuit cofactoring. The bottom two lines show these results for our node-based clause conversion. It is seen that the size of the clauses, necessary to represent the state set, in our approach is an order of magnitude smaller than that of the conventional path-based conversion in *all* the iterations.

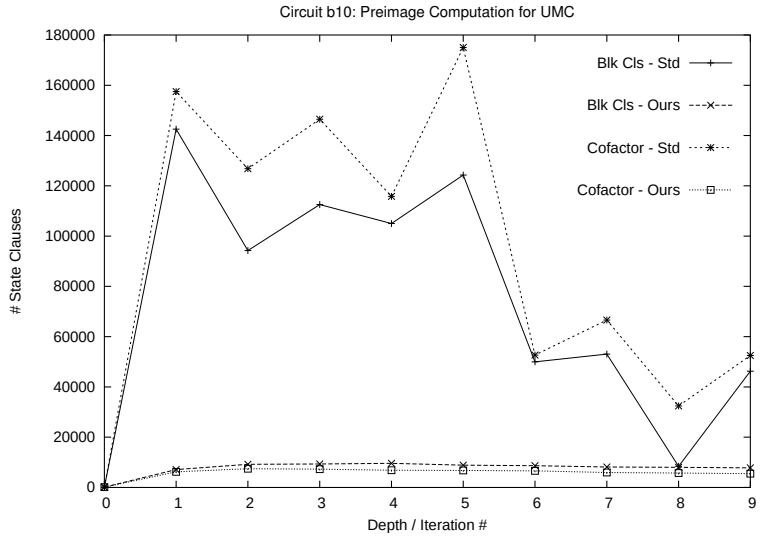


Figure 4.5. Preimage computation for b10

4.5 Summary

We discussed the decision diagram representation for the state-set that occurs during pre-image computation in an iterative framework for both SAT-based and ATPG-based techniques. In particular, we have presented a new clause conversion technique to account for the state-set, added in each iteration of an Unbounded Model Checking framework that operates on a pure SAT solver. We derived the clauses that represent different configurations of a node in the state set ZBDD. We exploit the node sharing feature of the ZBDD to obtain a significant reduction in the number of clauses that is required to represent the state set ZBDD (set of cubes/clauses). Experimental results show that our approach can lead to orders of magnitude reduction in both the number of state-set clauses and run-time as compared to previous approaches.

Chapter 5

ATPG based Preimage Computation

In this chapter, we introduce a novel preimage computation technique that directly computes the circuit cofactors without an explicit search for any satisfiable solution [74]. We use an implicit search on the primary inputs of the sequential circuit to compute all the necessary circuit cofactors for the preimage. In order to alleviate the computational cost, aggressive learning techniques are introduced that reason on the search states by analyzing the relations among circuit cofactors. Such analysis generates search-state induced clauses that directly help to prune the cofactor space during preimage computation and to perform non-chronological backtracking. Experimental results show that a significant improvement can be achieved in both performance and capacity as compared to the existing techniques.

5.1 Introduction

The quantification of the next state and input variables in the transition relation of finite state machines is the fundamental step in the pre-image computation for Symbolic methods. The all-solutions ATPG, discussed in Chapter 3, or the all-solutions SAT [45], attempts to find each solution and then quantifies the inputs. In contrast, it is possible to quantify the combinational inputs of a circuit and find the circuit cofactors directly, by restricting the solution search. This will be a

search for circuit cofactors, rather than the search for solutions, as seen in conventional techniques.

In this Chapter, we propose a novel circuit based preimage computation technique. The novel features of this work are as follows:

1. We propose a decision tree search technique that directly computes the circuit cofactors for the preimage by quantifying the input variables.
2. We derive relations among the circuit cofactors based on the search states reached during the decision tree search.
3. We introduce search-state induced learning, based on the relations derived, to prune the cofactor space and to perform non chronological backtracking.

We implemented the proposed method and compared the results against three other preimage computation techniques: (1) McMillan's blocking clause approach [45], (2) Sheng and Hsiao's ATPG (Automatic Test Pattern Generation) based approach [44], and (3) Ganai et al.'s cofactor blocking approach [69]. We implemented all the four techniques on top of a publicly available verification tool called ABC [75]. We chose publicly available circuits from ISCAS '85 and VIS-ITC '99 benchmark suite for our experiments. The experimental results show that direct circuit cofactoring with search-state induced learning can lead to several orders of magnitude improvement in both run-time and memory as compared to some of the existing techniques.

5.1.1 Previous Work

Initially, certain variants to Binary Decision Diagrams were proposed for image/preimage computation. In [38] [37], non-canonical structures such as Reduced Boolean Circuit (RBC) and Boolean Expression Diagrams (BED) are used to represent the transition relation, and quantification rules are proposed for specific sub-structures. However, in the general case, the length of formulas may

grow exponentially due to the quantification. In [36], SAT solvers are integrated with BDDs to perform image computation. The SAT solvers are used to provide a disjunctive decomposition of clauses and BDDs are used subsequently to solve the disjunctions. In [45], McMillan proposed the seminal preimage computation technique for unbounded model checking using a pure SAT solver. The transition relation and negation of the property are represented as a set of clauses, and variable quantification is performed during solution enumeration. After each solution is found, an enlarged cube is identified by re-building the implication graph. The negation of this cube is added as a *blocking clause* to the clause database. The set of all blocking clauses represent the preimage and they are stored in a Zero-suppressed Binary Decision Diagram (ZBDD) to perform light weight optimization. In [46], the authors use a simple circuit-based justification procedure to identify enlarged solution cubes (and thus avoid re-building the implication graph). In [68], the authors improved the ZBDD-to-clause conversion step for preimage iteration, by generating clauses for each node in the ZBDD rather than direct clause representation for the state-set. Recently, in [76], Jin et al. proposed to perform conflict analysis on the blocking clause and prune both the solution and conflict space efficiently.

In [44], an ATPG engine is used for circuit based preimage computation. The transition relation is represented as a Boolean circuit and an ATPG engine is invoked to enumerate all the solutions that represents the complete preimage. Success-driven learning is proposed to avoid re-searching overlapping solution sub-spaces, thereby accelerating the search. The preimage is stored as a free BDD which shares all the common subgraphs and is generally more compact than ROBDD. Success-driven learning was further augmented in [43] to prune larger search-spaces. Success-driven learning was also integrated into a hybrid SAT-solver to take advantage of both circuit-based success-driven learning and clause-based conflict-driven learning in [65].

In a recent approach for SAT-based UMC in [69], the transition relation is represented by an OR-inverter graph and efficient solution enumeration is performed using a specialized hybrid SAT solver. After obtaining each solution, the authors use the circuit cofactors, with respect to the input

assignment, as the enlarged state space. The circuit cofactors can capture a larger space than the enlarged solution cubes. The preimage is stored incrementally as a non-canonical or-inverter graph that uses function hashing and two level minimization on the fly. Overall, their experiments show a run-time improvement over the technique in [45], since they are able to capture a larger state space at each enumeration.

We re-visit the preliminaries in the next section, with primary focus on pre-image computation. In section 5.3, we explain the basic search technique to compute all the circuit cofactors that represent the complete preimage. We also discuss interesting relations among the search states and the circuit cofactors generated during the search. In section 5.4, we describe the search state induced learning to prune the cofactor space and perform non-chronological backtracking. In section 5.5, an experimental evaluation of the proposed approach is presented. Finally, we summarize the paper in section 5.6.

5.2 Preliminaries

A Finite State Machine (FSM) can be represented by the 6-tuple: $< I, O, S, \delta, \lambda, S_0 >$, where $I, O, S, \delta, \lambda, S_O$ represent the primary inputs, primary outputs, state-set, next state function, output function and initial states, respectively. Symbolically, the preimage for a given set of next states, $S(X')$, is given by:

$$Preimage(X): \ \exists_{I,X'} \, T(X, I, X') \ \wedge \ S(X')$$

X represents the present state variables

X' represents the next state variables

T represents the transition relation

Please note that $T(X,I,X') \wedge S(X')$ can be represented as a single output Boolean circuit as shown in Figure 5.1, which is the model for preimage computation. We refer the model as circuit or formula interchangeably when the meaning is clear from the context. We also refer to the circuit gates and Boolean variables interchangeably when the meaning is clear from the context.

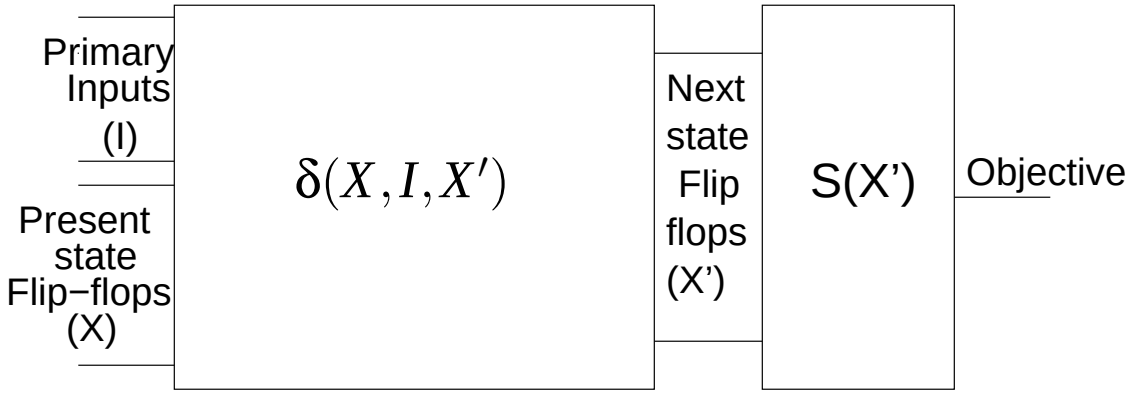


Figure 5.1. Preimage computation model

5.2.1 Review of SAT-based Preimage Computation

For SAT-based approaches, the model is usually represented as a Boolean formula in Conjunctive Normal Form (CNF): formula $\phi(V)$ which depends on variables in $V = \{v_1, v_2, ..., v_n\}$. The Boolean Satisfiability problem (SAT) consists of determining whether $\phi(V)$ is satisfiable by an assignment to the variables V. Most of the present day SAT solvers employ DPLL [21] procedures on the CNF formulas. A single-solution SAT solver can be augmented to find all the solutions that satisfy $\phi(V)$ [45], as follows:

1. After each solution cube is enumerated, add its negation as a *blocking clause* and continue to search for more solutions
2. Repeat Step 1, until no more solutions can be obtained.

Suppose we obtain a solution (α) for a Boolean formula ($\phi(V)$), where $V \mapsto \alpha$ and $\alpha \in \{0,1\}$. For the model shown in Figure 5.1, the inputs and present state variables ($V_{IX} \subseteq V$) will imply the values of the output. It is sufficient to block the values of these variables in V_{IX} rather than blocking the complete solution $V \mapsto \alpha$. Further, in preimage computation, because we have to quantify away the primary input variables, we can ignore the input variables and simply use the conjunction of present state variables in V_{IX} as an *enlarged solution cube* for the blocking clause. The solution cube can be further enlarged by redrawing the implication graph [45], circuit justification [46] or circuit cofactoring [69].

If $\phi(V) = T(X,I,X') \wedge S(X')$, then the set of enlarged *solution cubes* represents the preimage for $S(X')$ as an equivalent Boolean formula in Disjunctive Normal Form (DNF). On the other hand, if $\phi(V) = T(X,I,X') \wedge \neg S(X')$, then the set of *blocking clauses* (negation of solution cubes) represents the preimage as an equivalent CNF. This preimage is used as the target state-set for the next iteration, and we iterate the preimage computation until a fixed point is reached. In order to check for fixed point, we restrict the reached states in the model and stop in an iteration, where no more new states can be found.

5.2.2 Review of Circuit-based Preimage Computation

In the case of circuit-based approach, preimage computation is performed directly on the Boolean circuit shown in Figure 5.1. Some hybrid SAT-based approaches such as [69] [71] make use of the circuit structure to accelerate the performance of their baseline SAT solver. On the other hand, in the pure circuit-based approach [44], the preimage is computed by finding all the input patterns that satisfy the objective using a *decision tree* search as follows:

1. Pick an input / present state variable that has an X-path (i.e., a path of unspecified values) to the objective.

2. Logic simulate the input or flip-flop value in the circuit.

3. If the objective is satisfied/conflicted, add a one/zero terminal.

4. Backtrack/Repeat Step 1 to implicitly enumerate all the possible patterns.

The decision tree becomes a free BDD that finally represents the set of all solutions that satisfy the objective. The free BDD can be directly transformed into a Boolean circuit and then used as the target state-set for preimage computation in the next iteration. Similar to SAT-based approaches, we can then iterate until no more new preimage states can be found, i.e., fixed point is reached.

5.2.3 Motivation

The main reason behind the enormous success of SAT-based approaches for preimage computation is the intelligent learning techniques that prune the search space and aid in performing non-chronological backtracking. SAT-based techniques branch on all the variables in the CNF to compute the solution space and the learning is generally performed at the solution and conflict terminals. In the case of preimage computation, we are specifically interested in computing the cofactor space of the circuit and it is sufficient to branch on the primary inputs alone. If we explore *only* the primary inputs in a SAT-like fashion, then the terminals of the decision tree are simply circuit cofactors instead of solutions or conflicts. Therefore, the practical limitations of this approach are the (1) lack of cofactor-driven learning techniques and (2) potentially huge cofactor sizes that might have to be blocked at each terminal.

In order to overcome these limitations, we propose search-state driven learning techniques that can deduce clauses from logic decompositions of the circuit. Instead of blocking a circuit cofactor using the entire logic decomposition of the circuit, we deduce a **single clause** that can account for that logic decomposition. Based on these clauses we show that it is possible to perform non-chronological backtracking in the decision tree. Further, we use the well-known AND-inverter

graphs for the circuit and incorporate elegant DAG-aware rewriting rules [77] while constructing the preimage.

5.3 Basic Circuit Cofactor Expansion

Consider a circuit with m inputs, say $I = \{i_1, i_2, ...i_m\}$, n flip-flops, say $X = \{x_1, x_2, ..., x_n\}$ and p outputs, say $O = \{o_1, o_2, ..., o_p\}$. Suppose we wish to compute the preimage for a target state-set $S(X')$. We first construct the single output circuit $C(I,X)$, as in Figure 5.1. Now the preimage on C is given by:

$$
\begin{aligned}
Preim(X) &= \exists_I C(I,X) && (5.1)\\
&= \exists_{I\setminus i_1} \left[C_{(i_1=0)} \vee C_{(i_1=1)}\right] && (5.2)\\
&= \exists_{I\setminus\{i_1,i_2\}} \left[C_{(i_1=0,i_2=0)} \vee C_{(i_1=0,i_2=1)} \vee C_{(i_1=1,i_2=0)} \vee C_{(i_1=1,i_2=1)}\right] && (5.3)\\
&\quad ... && (5.4)\\
&= \exists_{I\setminus\{i_1,i_2,...,i_{m-1}\}} \left[C_{(i_1=0,i_2=0,...,i_{m-1}=0)} \vee ... \vee C_{(i_1=1,i_2=1,...,i_{m-1}=1)}\right] && (5.5)\\
&= C_{(i_1=0,i_2=0,...,i_{m-1}=0,i_m=0)} \vee ... \vee C_{(i_1=1,i_2=1,...,i_{m-1}=1,i_m=1)} && (5.6)
\end{aligned}
$$

It may be noted that each term in the last equation of $Preim(X)$, i.e. Equation 5.6, is a generalized cofactor of $C(I,X)$ w.r.t. an input assignment. We refer to it as the circuit cofactor or simply cofactor in the sequel. A brute-force approach to compute $Preim(X)$ is to simulate the circuit with all the 2^m input patterns one by one. Each simulation leads to a circuit cofactor and the disjunction of all these circuit cofactors is the preimage.

A systematic way of exploring all the 2^m input enumerations is to use a decision tree search, as shown in Figure 5.2(A). We branch only on the primary input variables in I and the corresponding cofactors are computed at the terminal nodes of the decision tree. Then, each terminal of the decision tree will be a circuit cofactor that represents exactly one term in Equation 5.6 for $Preim(X)$. In the decision tree search, the disjunction of all the terminal cofactors is the preimage.

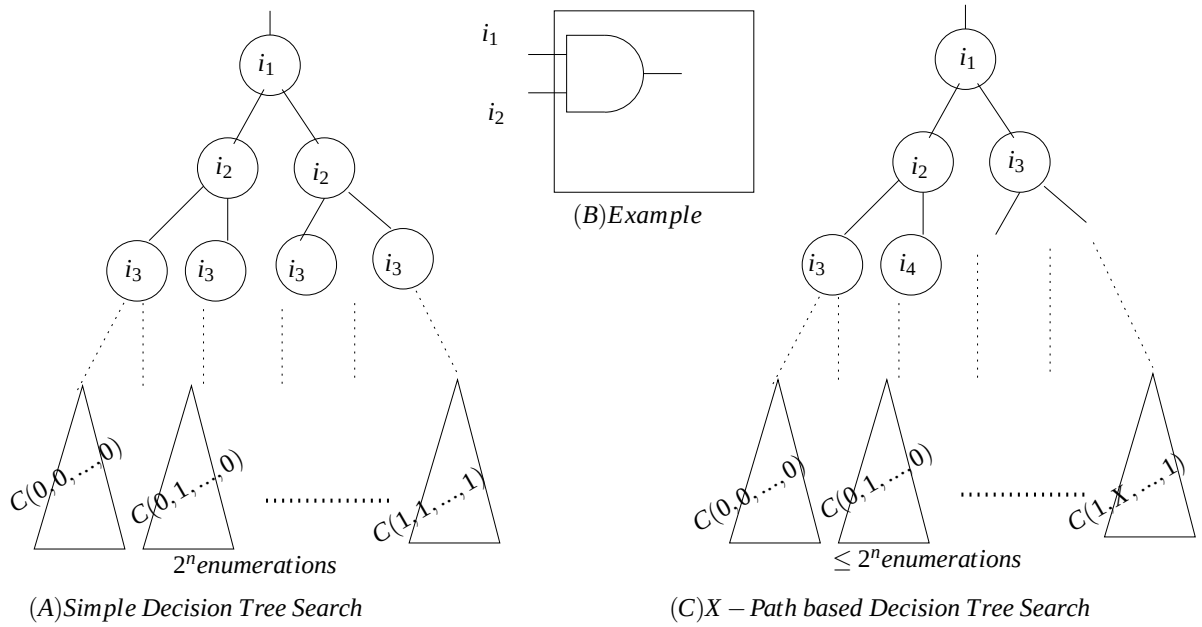


Figure 5.2. Decision Tree Search for Cofactor Enumeration

In order to reduce the number of cofactor enumerations, we choose to branch on a primary input variable (as a decision) only if it has an X-path (a chain of unspecified/unassigned gates) to the output of the circuit. This simply avoids, in a sense, the structurally symmetric inputs and prunes some of the redundant cofactors. For example, suppose we have two primary inputs, i_1 and i_2, that feed a 2-input AND gate as shown in Figure 5.2 (B). It is sufficient to compute either $C_{(i_1=0,i_2=X,\ldots)}$ or $C_{(i_1=X,i_2=0,\ldots)}$, since the two cofactors are equivalent. In our decision tree search, after $i_1 = 0$ is chosen, there will be no X-path from i_2 and we will not choose i_2 thereafter. Therefore, we no longer have to compute all the 2^m circuit cofactors, as we may skip many of the redundant cofactor spaces. Each terminal cofactor in Figure 5.2 (C) represents *atleast* one term (or a disjunction of terms) in Equation 5.6 for $Preim(X)$. It may be noted that the disjunction of all terminal cofactors is, again, the complete preimage that we intend to compute.

5.3.1 Circuit Search-state Relations

In order to prune the cofactor space during preimage computation, it is possible to learn from the search-state relations that exist at different scenarios during preimage computation. We first review a few terms from [44] and [43] that are demonstrated in Figure 5.3, to explain the following discussion.

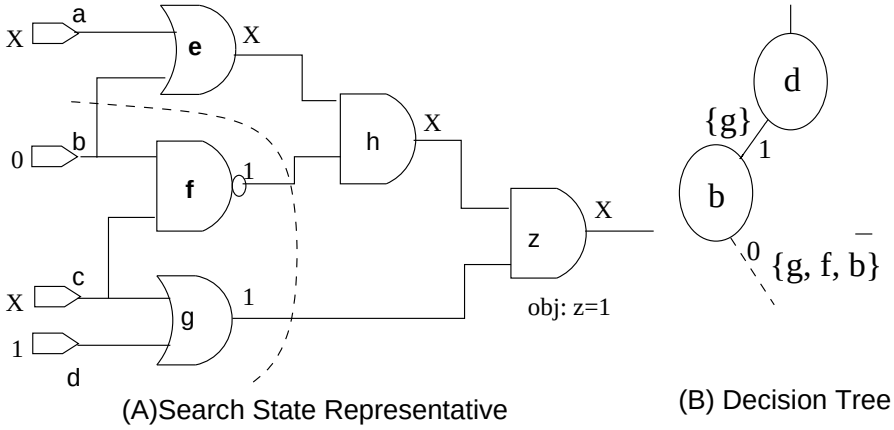


Figure 5.3. Cut-sets in the Search Space.

For the circuit, $C(a,b,c,d)$, shown in Figure 5.3(A), a partial decision tree is shown in Figure 5.3(B).

- **Search-State** *SS***:** It is the internal state of the circuit at each branch in the decision tree. It simply captures the set of all implied gate assignments at each branch in the decision tree. Example: Figure 5.3(A) for branch $b = 0$ in the decision tree Figure 5.3(B).
- **Cut-set for search-state** $CS(SS)$**:** It is the set of specified gates in a search-state that has at least one X-path (a chain of unspecified/unassigned gates) to the circuit output. Example: dashed line in Figure 5.3(A).

In order to facilitate the discussion on search states, we use the following abbreviations that correspond to a particular search state, SS (examples from Figure 5.3(A)):

- **Gate Value Assignment** $GVA(SS)$**:** It is the set of gate value pairs for all gates in C. Example: $\{(a=X),(b=0),(c=X),(d=1),(e=X),...,(z=X)\}$.
- **Cutset Assignment** $CSA(SS)$**:** It is the set of gate value pairs for all gates in the cutset for a search state. Example: $\{(b=0),(f=1),(g=1)\}$.
- **Cutset Conjunction** $CSC(SS)$**:** It is the Boolean formula that represents the conjunction of literals that are obtained from all the gates in $CSA(SS)$. Example: $\neg b \wedge f \wedge g$.
- **Input Value Assignment** $IVA(SS)$**:** It is the set of gate value pairs for all the input gates in $GVA(SS)$. Example: $\{(a=X),(b=0),(c=X),(d=1)\}$. We also say that SS is the search state *induced* by the input value assignment $\{(a=X),(b=0),(c=X),(d=1)\}$.
- **Specified Input Value Assignment** $SIVA(SS)$**:** It is the set of all specfied gate value pairs in $IVA(SS)$. Example:$\{(b=0),(d=1)\}$.
- **Input Value Conjunction** $IVC(SS)$**:** It is the Boolean formula that represents the conjunction of literals that are obtained from all the gates in $SIVA(SS)$. Example: $\neg b \wedge d$.
- **Logic decomposition / Circuit Cofactor for search-state** $Cof(SS)$**:** It is the reduced circuit obtained by projecting the input value conjunction $IVC(SS)$ on the circuit, C. Example: $C(a,0,c,1)$. This can also be referred as $C \downarrow_{IVC(SS)}$ or $C \downarrow_{CVC(SS)}$.
- **Preimage Circuit Cofactors** $PreimCof(SS)$**:** It is the disjunction of all terminal circuit cofactors that can be obtained for the circuit $Cof(SS)$.
- **Preimage Cofactor Space** $PreimCofSpace(SS)$**:** It is the Boolean space that refers to $PreimCof(SS)$.

In order to perform learning on the search states, we define the following relations among search states:

- **Consistent Search states:** A search state A is said to be *consistent* with search state B, if $CSA(A) \subseteq GSA(B)$.
- **Extendable Search states:** A search state A is said to be *extendable* to search state B, if we can induce a search state D, such that $SIVA(D) \supseteq SIVA(A)$ and $CSA(D) = CSA(B)$.

Lemma 1: If search states A and B are such that $CSA(A) = CSA(B)$, then $PreimCofSpace(A)$ is equivalent to $PreimCofSpace(B)$.

Proof: We know that $IVC(A) \rightarrow CVC(A)$ and $IVC(B) \rightarrow CVC(B)$ due to logic simulation. Therefore, $C \downarrow_{IVC(A)} = C \downarrow_{CVC(A)} = C \downarrow_{CVC(B)} = C \downarrow_{IVC(B)}$. Since the generalized circuit cofactors under $IVC(A)$ and $IVC(B)$ are the same, $PreimCofSpace(A)$ is equivalent to $PreimCofSpace(B)$. $\diamond$

Lemma 2: If search state A is consistent with search state B, then A is extendable to B.

Proof: We will construct a search state D such that $SIVA(D) \supseteq SIVA(A)$ and $CVC(D) = CVC(B)$.

First, let us construct an implication graph $IG(A,B)$ as follows: Start from the literals in $CVC(B)$ and find its antecedents in the search state of B. Recursively find the antecedents until, we reach the gates in $CVA(A)$ or $IVA(B)$, whichever is earlier.

Let $SIVA(IG)$ be the set of specified input value assignments in the root nodes of the implication graph, which are not in $CVA(A)$ or its fanin cone (see Figure 5.4). Let $IVC(IG)$ be the conjunction of all the literals in $SIVA(IG)$. From the implication graph, $IVC(IG) \wedge CVC(A) \rightarrow CVC(B)$.

Since none of the gates in $SIVA(IG)$ are in $CVA(A)$ or its fanin cone and all the gates in $SIVA(A)$ are in $CVA(A)$ or its fanin cone (see Figure 5.4), $SIVA(IG) \bigcap SIVA(A) = \Phi$. Let us induce a search state D, such that $SIVA(D) = SIVA(IG) \bigcup SIVA(A)$. Then $IVC(D) = IVC(IG) \wedge IVC(A) \rightarrow$

$IVC(IG) \wedge CVC(A) \rightarrow CVC(B)$. Therefore, $CVC(D) = CVC(B)$. $\diamond$

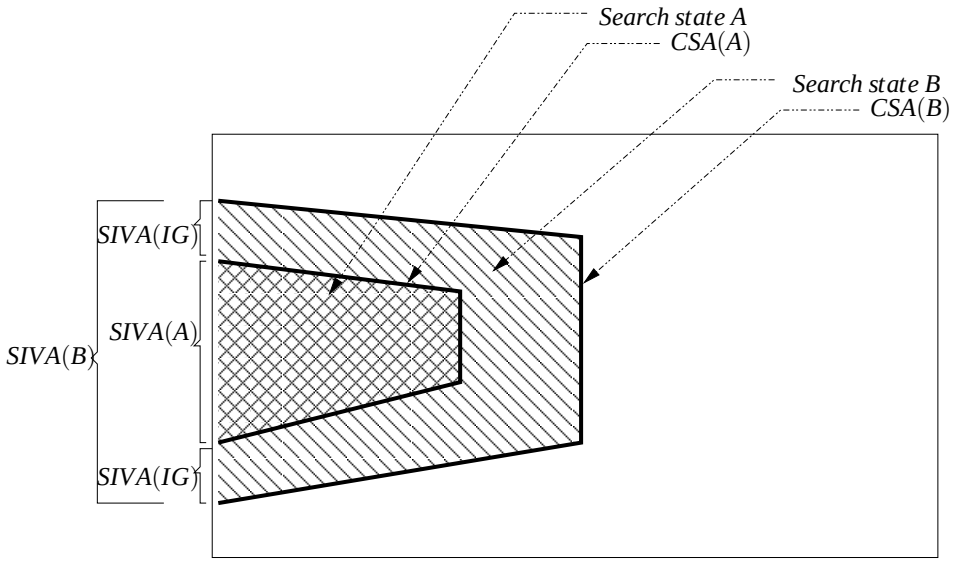


Figure 5.4. Proof for Lemma2

Theorem: If search state A is consistent with search state B, then $PreimCofSpace(B)$ implies $PreimCofSpace(A)$.

Proof: From Lemma 2, we can induce a search state D such that $SIVA(D) = SIVA(A) \bigcup SIVA(IG)$ and $CSA(D) = CSA(B)$. Since $CSA(D) = CSA(B)$, $PreimCofSpace(D) = PreimCofSpace(B)$, from Lemma 1.

From search state A, if we assign $SIVA(IG)$ to the primary inputs, we will reach search state D and induce a cofactor $Cof(D)$. Other preimage cofactors are also present that can be obtained by different input value assignments, starting from A. A decision tree representation for the scenario is shown in Figure 5.5. Since the preimage cofactor space for A is a disjunction of terminal preimage cofactors, $PreimCofSpace(D) \subseteq PreimCofSpace(A)$. $\diamond$

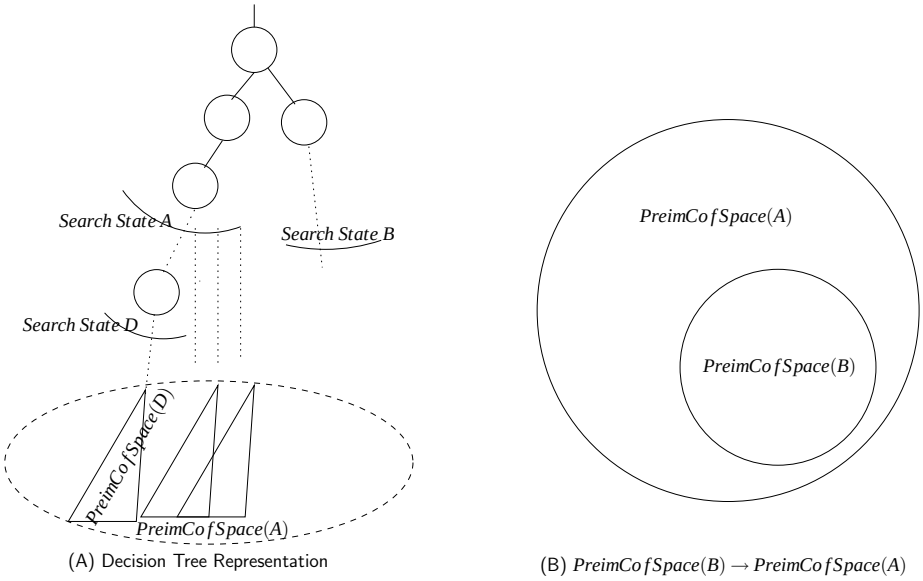


Figure 5.5. Proof for Theorem

5.4 Search-state aware Circuit Cofactor Expansion

It can be seen that the relations among search-states mentioned in Theorem 2 covers all other relations in the earlier theorems. Therefore, it is sufficient to learn from the consistency relations of cut-sets only. In the following discussion, we analyze the situations, where we can perform search-state induced learning.

5.4.1 Search-state induced clauses

In the scenario shown in Figure 5.6, suppose we have obtained a new cut-set, *B*, at a branch E_1 in the decision tree. Then, we explore the branches under *B* to obtain the circuit cofactors for the logic decomposition C_{LB}. After backtracking from E_1, suppose we reach a branch E_2 (see Figure 5.6 (B)) such that the cut-set, *A*, obtained here is consistent with *B*. Then, according to Theorem 2, the cofactors that will be obtained from the logic decomposition, C_{LA}, is already contained in

the circuit cofactors under B. Therefore, we need not compute the cofactors under C_{LA} and can immediately backtrack in the decision tree.

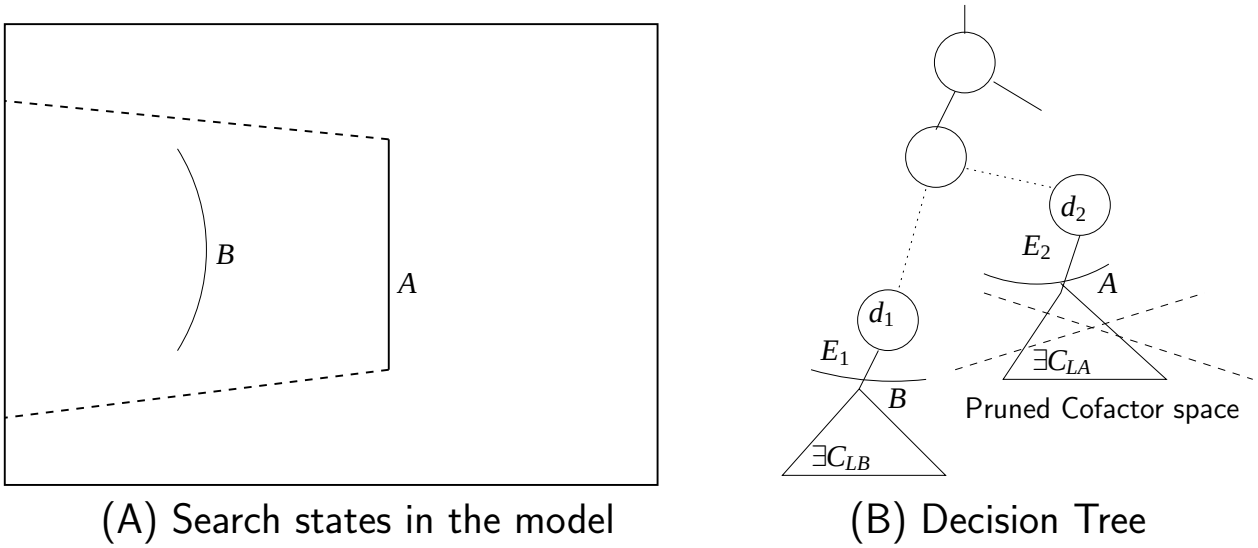


Figure 5.6. Search-state induced learning

In order to take advantage of this kind of scenario, we derive a search-state induced clause from the cut-set B. Let $B = \{b_1, b_2, ..., b_k\}$ be the cut-set that induces the logic decomposition C_{LB}, i.e., $(b_1 \wedge b_2 \wedge ... \wedge b_k) \Rightarrow C_{LB}$. By contrapositivity, $\neg C_{LB} \Rightarrow (\neg b_1 \vee \neg b_2 \vee ... \neg b_k)$. Consequently, we can add the search-state induced clause $(\neg b_1 \vee \neg b_2 \vee ... \neg b_k)$ to block the cofactor space in C_{LB}. Furthermore, when we reach any cut-set A that is consistent with B (recall that if A is consistent with B, then the variable assignments for gates in B are the same in the search-state of A), this added search-state induced clause will be violated and will help to backtrack immediately. However, it should be noted that this clause is not equivalent to C_{LB}. Therefore, it is only claimed that we will not reach the same logic decomposition structurally in the circuit. This is analogous to the fact that we can reach the conflict terminal multiple times during SAT search and each conflict terminal induces different conflict clauses.

We store the complement of cut-sets at each branch in the decision tree as search-state induced

clauses in a clause database. We check if any of these clauses are unsatisfied or not, during the logic simulation of input decisions in our approach. We do not perform Boolean Constraint Propagation (BCP) on these clauses, since the unit implications on the internal variables in the circuit may violate the search-state analysis discussed in the previous section.

It is worth mentioning that this type of learning cannot be performed in conventional SAT solvers, since they search for the solution space, whereas we directly search for the cofactor space. The arbitrary decisions made by a conventional SAT solver also disallows the construction of the circuit cut-set for a search state, which is necessary in the proposed approach. On the other hand, it might be possible to integrate the proposed learning into the hybrid SAT solver approach with circuit cofactor blocking, proposed in [69]. We can possibly find the terminal cut-sets after blocking the circuit cofactors and use their complements as conflict clauses. However, it is not possible to represent the internal search states in the hybrid solver using the circuit cut-sets used in our approach.

5.4.2 Non-chronological backtracking

The fundamental idea for our non-chronological backtracking is as follows: Let $L_I \subseteq (I \mapsto \alpha)$ be a set of literals that is obtained by assigning Boolean values to a subset of the primary input variables. Consider partial input assignments, $\omega_1 = \bigwedge_{k_1=1}^{n_1} i_{k_1}$ and $\omega_2 = \bigwedge_{k_2=1}^{n_2} i_{k_2}$ such that $i_{k_1}, i_{k_2} \in L_I$. Let $l \notin L_I$ be a literal which is a Boolean assignment to a primary input variable. If $(\omega_1 \wedge l) \Rightarrow C_1$ and $(\omega_2 \wedge \neg l) \Rightarrow C_2$, then by resolution, $(\omega_1 \wedge \omega_2) \Rightarrow (C_1 \vee C_2)$, where C_1, C_2 are cut-sets (or circuit decompositions) for the corresponding search-states.

This scenario is shown in the Decision Tree in Figure 5.7. Let $a \wedge c \wedge e \Rightarrow C_1$ and $a \wedge b \wedge \neg e \Rightarrow C_2$. Then by resolution, $a \wedge b \wedge c \Rightarrow C_1 \vee C_2$. Therefore, after computing the circuit cofactors for C_1 and C_2, we can bypass the other branch of d and directly return to c. This is because the cofactor space in the second branch of d is already contained in $C_1 \vee C_2$. We can thus avoid the redundant

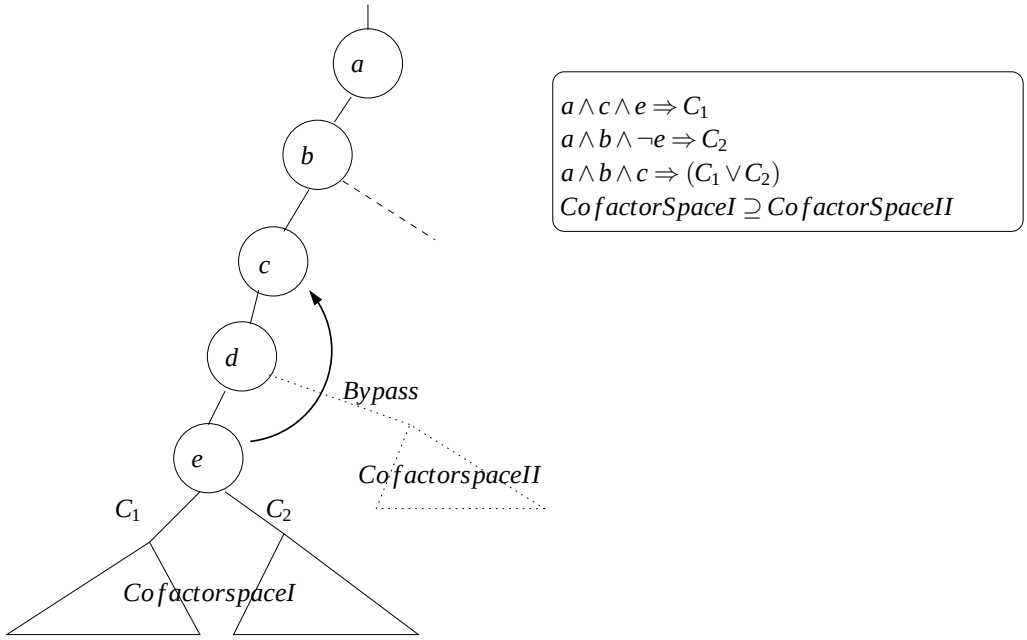


Figure 5.7. Non-chronological backtracking

cofactor computation by backtracking non-chronologically.

In our implementation, we use bit vectors to record the decision levels that are responsible for each gate assignment, similar to [78]. The length of the bit vector for a gate is generally equal to the decision level when it is assigned. The bit positions correspond to the antecedent decisions responsible for the gate assignment. For each decision, we assign a 1 to the bit in the position of the decision level. Then, during logic simulation we propagate the bit vectors to reflect the antecedents responsible for each gate assignment. The bits in the antecedent decision levels are assigned a 1 and the other bits are assigned a 0. The OR of all bit vectors of the gate assignments in a cut-set corresponds to the decisions that implied the cut-set. After computing each circuit cofactor, we directly backtrack to the highest decision level in the cut-set bit vector.

5.5 Experimental Evaluation

We implemented the proposed cofactor expansion technique in C on top of a publicly available verification tool called ABC [75] to compute the preimage until fixed point is reached. In addition, we implemented three other preimage computation techniques: all-solutions SAT solver on top of MINISAT SAT solver [45], all-solutions ATPG with success driven learning [44] and a hybrid SAT solver [69]. In all the techniques, we represent the circuit as an AND-inverter graph, and use DAG-aware rewriting and other low-cost structural optimization techniques across iterations. For the SAT-solver, we use circuit justification [46] to deduce the blocking clause and an efficient ZBDD-to-clause conversion technique proposed in [68]. For this approach, we limit the number of clauses to 1 million and the number of ZBDD nodes to 200K. For the all-solutions ATPG, we limit the number of the gates in the preimage state-set to 125K and the size of the hash table to 1 million. For hybrid SAT solver, we limit the number of conflict clauses to 1 million and the number of gates in the preimage state-set to 125K. For our approach, we limit the number of search-state induced clauses to 1 million and the number of gates in the preimage state-set to 125K. We conducted a set of experiments on ITC'99 and ISCAS'85 benchmark circuits for some of the hard-to-reach states and VIS invariant properties, on a Pentium IV, 3GHz machine running the Redhat Linux Operating System. The results are tabulated in Table 5.1.

In Table 5.1, the first two columns list the circuit and the number of latches in the circuit. Columns 3 and 4 report the preimage computation depth (the number of iterations) and the time taken respectively. Similar results are reported for other techniques in the subsequent columns. It is seen that SAT *times out* for most of large circuits, with dense reachable states (many preimage states per iteration), since they have to compute a large number of solutions and block them cube by cube. Since they also consider the flip-flops during solution enumeration, they have to explore a larger space. Similarly, the ATPG technique needs to store a huge number of search states to perform learning, since they also branch on the circuit flip-flop variables. This leads to memory explosion

Table 5.1. Preimage Computation until convergence

Ckt	#	SAT [45]		ATPG [44]		Hybrid [69]		Our approach	
	ltch	dpth	Time(s)	dpth	Time(s)	dpth	Time(s)	dpth	Time(s)
s3384	183	1*	TO	1*	9(MO)	4	3.0	4	1.8
s1423	74	2	307.8	1*	11(MO)	2	0.20	2	0.09
s5378	164	1*	TO	1*	154(MO)	4*	2K(MO)	8	69.7
s4863	104	1*	TO	1*	94(MO)	1*	14K(MO)	2	1.7K
s1269	37	1*	19K(MO)	1*	302(MO)	8	229.7	8	272.3
s3271	116	1*	TO	1*	473(MO)	6	15	6	723
s420.1	16	51	11	51	27.1	19*	2.7K	51	107.1
b07	45	2*	TO	29	1160.4	10*	7K(MO)	29	8.7K
b08	21	19	4.2	19	6.3	19	990.3	19	173.3
b09	28	15*	TO	21	4.7	11*	TO	21	371.3

*MO - Memory Out; TO - Time Out; * - incomplete;*

in many of the larger circuits. On the other hand, hybrid solver and our approach, block the cofactors and prune a larger search space. However, the hybrid approach also searches the solution space, even though it blocks the cofactor space. This solution enumeration again lead to memory out in s5378 and s4863 due to the huge number of conflict clauses that were generated. We were able to compute the cofactors quickly and backtrack from that cofactor space due to search-state induced learning. Although we computed a larger number of cofactors, that were overlapping in the solution space, we save the time required to find a particular solution as compared to the hybrid SAT solver.

On the other hand, for smaller circuits, with sparse reachable states (few preimage states per iteration), the solution-based techniques perform significantly better than the cofactor-based approaches. In circuits such as s420.1 (counter) and b08 (finds inclusions in sequence of numbers), SAT performs significantly better than our approach. This is mainly because, they can find the smaller subset of satisfiable solutions quickly. In general, the cofactoring expansion technique is more suitable for circuits with dense reachable states, since each cofactor may cover a huge number of preimage states that will require a large number of solutions to be computed.

5.6 Summary

In this work, we presented a novel preimage computation technique that directly quantifies the primary inputs in the circuit and computes the cofactors required for preimage computation. We used the relations between the search-states and the circuit cofactors to prune the cofactor space and perform non-chronological backtracking. Experimental results show that the cofactor expansion technique performs significantly better than existing approaches for large circuits, where a large number of preimage states needs to be computed at each iteration.

Chapter 6

ATPG based Image Computation

Image computation is a fundamental step in formal verification of sequential systems, including sequential equivalence checking and symbolic model checking. Since conventional Reduced Ordered Binary Decision Diagram (ROBDD) based methods can potentially suffer from memory explosion, there has been a growing interest in using Automatic Test Pattern Generation (ATPG) / Boolean Satisfiability (SAT) based techniques in recent years. While ATPG has been successful for computing pre-image, image computation presents a very different set of problems. In this chapter, we present a novel backtracing-based ATPG technique for forward image computation [79]. We carefully alter the ATPG engine to compute the image cubes and store them incrementally in a Zero-Suppressed Binary Decision Diagram (ZBDD). In order to improve the efficiency of image computation, we propose three heuristics: (i) gate-observability based decision selection heuristics to accelerate ATPG, (ii) search-state based learning techniques supported with a proof for correctness, and (iii) on-the-fly state-set minimization techniques to reduce the size of computed image set. Experimental results on ISCAS '89 and ITC '99 benchmark circuits show that we can achieve orders of magnitude improvement over OBDD-based and SAT-based techniques.

6.1 Introduction

Image computation is required to perform forward state space traversal during reachability analysis in symbolic model checking. In essence, computing image requires the computation of the set of all next states that can be reached from a given set of states in one cycle. Similar to preimage computation, image computation requires the quantification of variables from the transition relation of the finite state machine. They present the same problem to BDD-based quantification method, since BDDs directly manipulate the decision diagrams. We have to perform the quantification operation on the present state variables in the BDDs, instead of next state variables (in preimage), for image computation. BDDs do not require any specific learning techniques that require the circuit information.

Some of the non-canonical methods such as [38, 37], where Reduced Boolean Circuit (RBC) and Boolean Expression Diagrams (BED) are integrated with SAT solvers to perform efficient variable quantification, target preimage computation only. Basically, the transition relation of the circuit is represented as an RBC/BED and the authors propose efficient quantification rules for specific substructures. These quantification rules such as inlining and scope reduction are specific for preimage computation and cannot be directly interpreted for image computation. In [36], SAT solvers are integrated with BDDs to perform image computation. The transition relation of the circuit is represented in Conjunctive Normal Form (CNF). The quantification of the variables is performed during solution enumeration. The SAT solvers are used to provide a disjunctive decomposition of clauses and later BDDs are used to solve the end problems. This technique was improvised in [80, 81], where the inactive clauses are avoided dynamically and efficient variable selection heuristics are proposed.

In pure SAT solvers, all solutions can be found for a particular SAT instance. We can compute all-solutions for the transition relation and initial state-set. Then, we can eliminate the primary input variables and present state variables from each solution and compute the image. The pre-

image computation problem [45, 69] is viewed as formula construction with the initial state-set as the objective. With this assumption, advanced solution driven learning techniques have been proposed to block larger search spaces in [45]. It is not possible to redraw the implication graph, which is necessary for generating enlarged blocking clauses, by using the next-state variables as decisions. In a recent approach for SAT-based model checking in [69], the transition relation is represented by an OR-INVERTER graph and efficient solution enumeration is performed by a specialized SAT solver. After obtaining each solution, the state-cube is enlarged by specifying the unspecified inputs and it is demonstrated to prune a larger search-space. The circuit cofactoring approach also assumes the existence of an objective during logic decomposition of the circuit for cofactor computation and does not seem to be feasible for image computation.

In [82], an approximate image is computed using the interpolants derived from refutation proofs of unsatisfiable CNF instances. This approximate image can be used to restrict the search space during SAT based unbounded model checking.

From an ATPG point of view, in [44], an ATPG engine is used for pre-image computation. The transition relation is represented as a Boolean circuit and an ATPG engine is invoked to enumerate all the solutions that represents the complete pre-image. In order to prune the search-space, success-driven learning that avoids re-searching overlapping solution sub-spaces was introduced. Success-driven learning was further augmented in [43, 42] to prune larger search-spaces. However, ATPG based pre-image computation techniques are devoid of inbuilt conflict-driven learning that is inherent in SAT solvers. In [65], success-driven learning was integrated into a SAT-solver to take advantage of both success-driven learning and conflict-driven learning. These ATPG-based methods are also specific for preimage computation, since the cutsets obtained during success driven learning require a logic simulation from the present state variables and the primary inputs in the circuit.

It should be noted that some of the above techniques [65, 43, 46, 69, 45, 44], the ATPG-based

techniques in particular, are specific to pre-image computation and cannot be directly extended to compute the image. While pre-image computation can be very useful, the pre-image space may potentially contain many unreachable states. On the other hand, all the states that are traversed during image computation from a legal initial state are guaranteed to be reachable. In this regard, we focus on image computation for sequential circuits using an ATPG engine. The novel features of this work are as follows:

1. We propose a novel backtracing-based ATPG for forward image computation. The image cubes are incrementally stored in a Zero-Suppressed Binary Decision Diagram, instead of adding them one by one.

2. We use a new decision selection heuristic and search-state based learning technique for the purpose of image computation.

3. We use cube minimization techniques 'on-the-fly' to reduce the size of the final state-set ZBDD.

Experimental results on ISCAS '89 and ITC '99 benchmark circuits show that we can achieve orders of magnitude improvement over OBDD-based and SAT-based techniques.

The rest of the chapter is organized as follows. We introduce the preliminaries of image computation and ZBDDs in section 6.2. In section 6.3, we illustrate the novel image computation technique that stores the image cubes incrementally in a ZBDD. In section 6.4, we propose new heuristics to improve the efficiency of the image computation technique. Experimental results for large ISCAS '89 and ITC '99 circuits are presented in section 6.5. We summarize the chapter in section 6.6.

6.2 Preliminaries

6.2.1 Image computation

Given a Transition Relation $T(X,I,X')$ and a set of initial states $S(X)$, the image of $S(X)$ can be computed by,

$$Image(X') = \exists_{x,i}\, T(X,I,X')\ .\ S(X)$$

where,

- X represents initial state elements
- I represents primary inputs
- X' represents next state elements

Essentially, the above equation can be used for image computation using formula manipulation, where the state sets and the transition relation are efficiently represented. We explain an image computation technique using the circuit structure in the following example. Consider the ISCAS89 circuit s27 shown in Figure 6.1. Gates $1,2,3,4$ are the primary inputs, $5,6,7$ are the present-state flip-flops and $5',6',7'$ are the next state flip-flops. The primary inputs (outputs) and present (next) state flip-flops are together called as inputs (outputs) to the circuit, when the meaning is clear from the context.

Let us compute the image for the initial state 000 using a **naive** technique. First, we assign the initial state values to the present state flip-flops $(5,6,7)$. Then, we have to assign all possible values at the primary inputs $(1,2,3,4)$. For each possible assignment at the primary inputs, we logic simulate the circuit and store the values obtained at the next-state flip flops $(5',6',7')$ as an

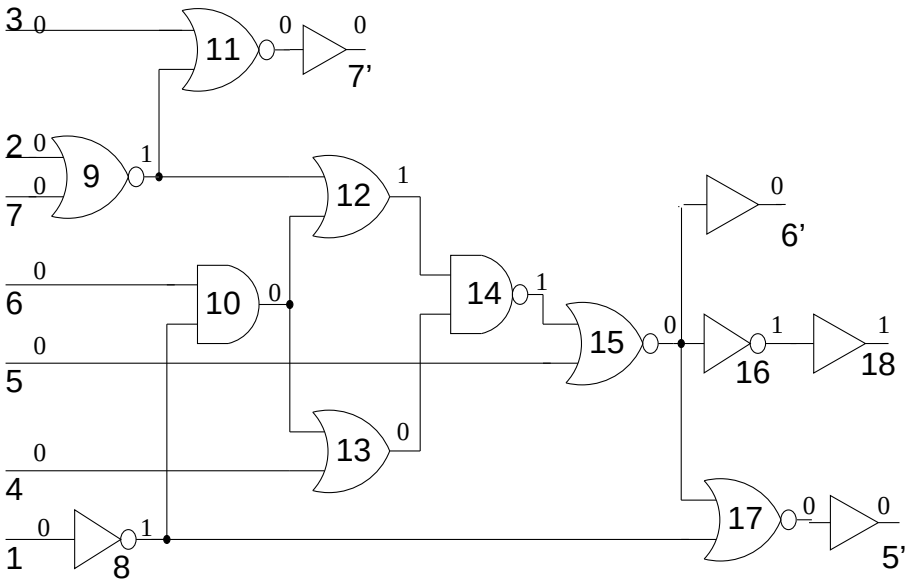


Figure 6.1. Image computation for s27 - 000 state

image cube. For example, when we assign 0000 at the primary inputs, the next state values is 000. After enumerating the 16 possible values at $1,2,3,4$ in this naive technique, the following image cubes are obtained in the above example: $000,001,100,101,010$. It should be noted that we may obtain the same cube multiple times in this technique, albeit, the final solution set gives the complete image. We refer the reader to [83, 36, 84, 85] for existing BDD/SAT based image computation techniques.

6.2.2 Zero-suppressed BDD (ZBDD)

In [72], Minato introduced ZBDDs to represent *sets of combinations* compactly and perform set operations efficiently. Sets of combinations $S = \{\{a,b\},\{\bar{a},c\},\{b,c\}\}$ are represented in a ZBDD as shown in Figure 6.2 (A). Each path from the root to TERMINAL-1 represents a set in the ZBDD. A 1-edge from a node denotes the presence of the element in the set and a 0-edge denotes its absence. If the variables in the ZBDD are linearly ordered, then it is called an ordered ZBDD.

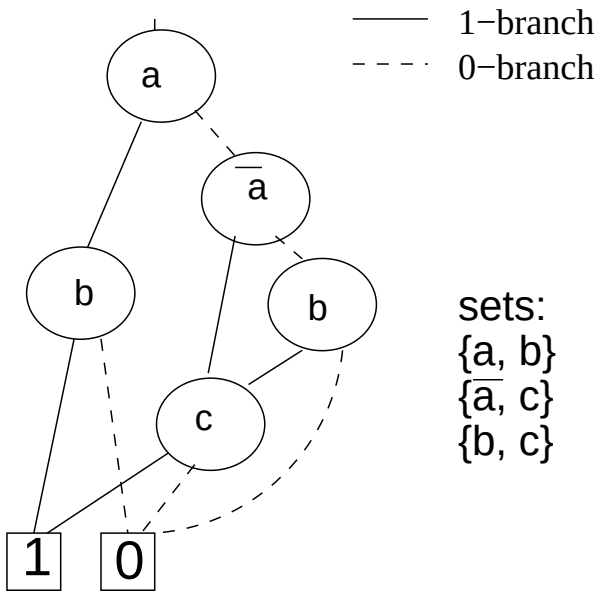


Figure 6.2. ZBDD

Otherwise, it is a general ZBDD. An excellent tutorial on ZBDDs is available at the website [86], and ZBDD algorithms have been implemented in publicly available CUDD [18] and Extra [73] packages.

ZBDDs are used in [45] to store the pre-image clauses. Each clause is added one by one to a ZBDD to compress its size. In this chapter, we construct a general ZBDD incrementally during image computation to represent the image state cubes.

6.3 Basic Image Computation

In general, an ATPG engine attempts to generate an assignment that satisfies a given objective. This leads to a straight-forward application in pre-image computation as seen in [43, 42, 44], where the target next state-set is naturally set as the objective. In their setup, the primary inputs are chosen as decisions and circuit search-states are used for learning. The input assignments generated in

the decision tree represent the complete pre-image set. This cannot be directly extended for image computation due to the following reasons:

1. Image computation is not symmetric to pre-image computation. In pre-image computation, exploring all possible assignments at the inputs is equivalent to exploring the complete search-space for all variables in the circuit. This is not true for image computation, if we explore all the possible assignments at the outputs.

2. A circuit represents a many-to-one mapping from input assignments to output assignments. Hence, it is not possible to set the target initial state as the objective and make outputs as the only decisions.

3. In order to benefit from search-state based learning similar to [43, 42, 44], it is necessary to choose the inputs as decisions and logic simulate the circuit i.e., we need a PODEM-like ATPG.

In this section, we explain a new procedure to perform PODEM like ATPG-based image computation. The basic procedure is outlined in Algorithm 6, where variables are quantified and a ZBDD is incrementally built that represents the complete image set.

Initially, a multi-level Boolean Circuit that represents the initial state set is constructed. This circuit is appended to the circuit-under-verification that represents the transition relation. The unspecified present-state flip-flops and primary inputs are chosen as decisions for the ATPG engine. After choosing each decision, the initial state circuit is also logic simulated to verify if we are still in the initial state space (care space). This is analogous to the use of don't-care space to constrain the search as in [87], but we use the care space instead of the don't care space. To choose a decision, we backtrace from an *unspecified* next state flip-flop through an X-path and pick a circuit input. Then, similar to PODEM, we logic simulate starting from the decision using an event driven mechanism. The next-state flip-flops that are specified in that decision level are noted. This is done recursively

until all the next state flip-flops are specified, i.e., a fully-specified image cube is obtained. Then, the decision is flipped (enforce a backtrack) and the other branch of the decision tree is explored similarly. Finally, a partial ZBDD is stored for the next state flip-flops specified at each decision level.

Algorithm 6: Image Computation

```
1  computeImage(){
2      if (!belongsToInitState()) then
           /*present state assignment ∉ initial state */
3          return TERMINAL0 ;
       end
4      if (allNxtStatesSpecified()) then
           /*found an image cube */
5          return TERMINAL1 ;
       end
       /*choose an unspecified input */
6      < decision, value >:= backtraceToInput ( ) ;
7      logicSimulate (decision, val) ;
       /*new next states specified */
8      nxt_states0 := getSpecNxtStates ( ) ;
9      z_node0 := computeImage ( ) ;
10     logicSimulate (decision, !val) ;
       /*new next states specified */
11     nxt_states1 := getSpecNxtStates ( ) ;
12     z_node1 := computeImage ( ) ;
       /*reset the input */
13     logicSimulate (decision, X) ;
14     z_node := buildPartialZbdd (nxt_states0, z_node0,
                                   nxt_states1, z_node1) ;
15     return z_node ;
   }
```

A simple mechanism to construct the partial ZBDD at one recursive call of Algorithm 1 is as follows: Let next-state values $\{x1, x2, ...\}$ be specified for $decision := val$ and $\{y1, y2, ...\}$ be specified for $decision :=!val$. The partial ZBDDs z_node0 and z_node1 are constructed earlier in

computeImage() during previous recursive calls to the algorithm. Since we construct the ZBDD in a bottom-up fashion, the resulting partial image ZBDD can be constructed as shown in Figure 6.3. Basically, the image cubes are added as sets to the ZBDD. After exploring all inputs assignments in such a fashion, this will incrementally lead to a ZBDD that represents the complete image at the end of ATPG.

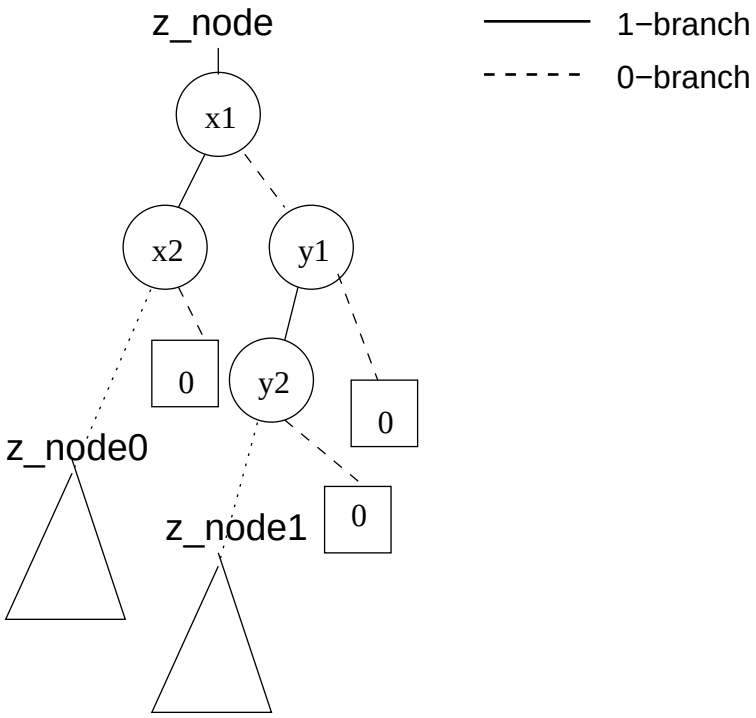


Figure 6.3. Partial Image ZBDD

6.4 Efficient Image Computation

In this section, we propose certain optimization techniques, such as decision selection heuristics, search-state based learning and on the fly state-set minimization, to improve the efficiency of basic image computation algorithm.

6.4.1 Decision selection heuristics

Several heuristics have been proposed in ATPG literature in choosing better decisions. These techniques aimed at detecting a fault and hence were mostly dependent on the testability measures of the circuit. Some of the popular techniques include [48, 49, 50, 51], where the difficulty of justifying/propagating a gate value is measured as controllability/observability measure. A recent decision selection heuristic for an all-solutions ATPG was proposed in [42] that considers the connectivity of variables to choose a decision. However, their technique specifically targets pre-image computation, where both solutions and conflicts can occur depending on the objective chosen.

In the case of image computation, our main idea is to efficiently explore all possible assignments at the inputs, and we backtrack only when the next state flip-flops are completely specified. Note that there is no specific objective in the case of image computation. We consider the following two factors to derive a decision selection heuristic:

1. If we always choose a decision that restricts the input assignment to the initial state space, then there will never be conflict-terminals, and we need not backtrack unnecessarily during ATPG.

2. If a small set of inputs can simply *specify* all the next-state flip-flops, then we can reach an image cube quicker in the decision tree. This can potentially reduce the depth of the decision tree.

Based on these two observations, we propose the following decision selection heuristic:

- If the output of the initial-state circuit is unspecified, we backtrace through the initial state circuit and choose a decision. This will help, albeit does not guarantee, to choose the input assignments that are within the initial state space.

- If the output of the initial-state circuit is specified, we backtrace in the original circuit using the *observability measures* of each gate. Since we want to specify the next-state flip-flops using minimum number of input assignments, it is desirable to choose an input that can easily propagate its value to the next state flip-flop.

In our method, we use the observability measures proposed in SCOAP [49] with a slight modification. For the sake of completeness, we explain how the observability measures are derived. For image computation, we are only interested at observing the input values at next-state flip-flops. For the next-state flip-flops, we assign the observability values as 0. For the primary outputs, we assign very high values (theoretically infinity) as their observability values. Then the observability values at the input of each gate is derived level by level based on equations proposed in [49]. For example, the observability at the input of an AND gate depends on the sum of the observability of its output and controllability of its other inputs to 1. Based on these measures, we backtrace through a gate with highest observability and finally select an input as a decision.

6.4.2 Search-state based learning

ATPG based techniques largely depend on learning techniques that help to accelerate their performance. We show that the search-state based learning techniques, initially proposed in [88, 44] and later analyzed and enhanced in [43, 42], can be construed for image computation as well.

Let the logic decomposition of the circuit after each decision be called a search state for the ATPG. It should be noted that we are exhausting all 2^n input assignments in a decision tree fashion. Each branch in the decision tree corresponds to a search state in the circuit. This search state can be uniquely represented by a cutset of gate values in the circuit as shown in [42, 44]. These cutsets were used to identify equivalent search-states and the corresponding sub-tree was shared in the final pre-image.

In the case of the image computation discussed in the previous section, we store a ZBDD as the resulting image. We do not have a decision tree as such. However, it is not hard to see that we can share sub-ZBDDs and take advantage of search-state based learning. Similar to [42, 44], we store the cutsets, but instead of decision tree nodes we store the ZBDD nodes that correspond to the cutset. When the same cutset/search-state occurs again during ATPG, we simply link that ZBDD node instead of re-searching the search-space corresponding to that logic decomposition of the circuit. In order to formally define this technique we propose the following theorem:

Theorem 1: If two equivalent cutsets are obtained using Algorithm 6, then the corresponding partial ZBDDs generated thereafter will be isomorphic, for a given ATPG.

Proof: If two equivalent cutsets are obtained, then the logical decomposition of the circuit will be the same for both the cutsets. This will lead to a sub-circuit. Thereafter, the ATPG engine targets to compute the complete image of that sub-circuit. This image will be stored as a partial ZBDD. Since the sub-circuits are identical, the partial ZBDDs will be isomorphic.

6.4.3 On the fly state-set minimization

All image/pre-image computation techniques generally suffer from solution explosion problem, which in turn leads to temporal explosion to obtain all the solutions and memory explosion to store them. The search-state based learning in ATPG helps to avoid the solution explosion problem by reusing previously explored search spaces and sharing partial ZBDDs. In order to reduce the size of the ZBDD further, we take advantage of two simple minimization techniques:

1. Subsumption:

   $v_1 \ldots v_{i-1}.v_i.v_{i+1} \ldots v_n + v_1 \ldots v_{i-1}.v_{i+1} \ldots v_n = v_1 \ldots v_{i-1}.v_{i+1} \ldots v_n$

2. Cube minimization:

   $v_1 \ldots v_{i-1}.v_i.v_{i+1} \ldots v_n + v_1 \ldots v_{i-1}.\overline{v_i}.v_{i+1} \ldots v_n = v_1 \ldots v_{i-1}.v_{i+1} \ldots v_n$

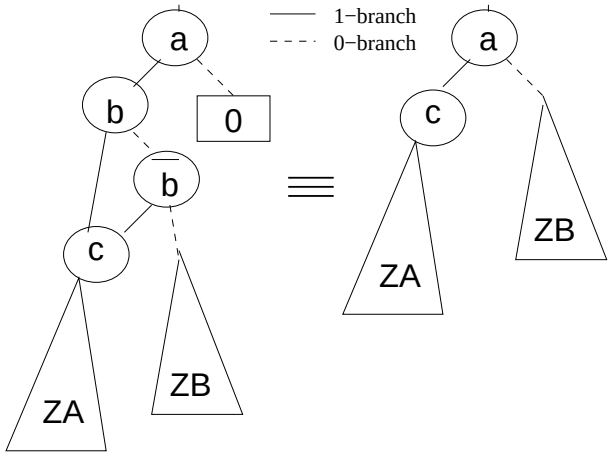


Figure 6.4. Minimization of ZBDD nodes

Although these are simple rules, they are very effective in reducing the state set since a lot of such structures occur during ATPG-based image computation. For illustration, we show the cube minimization technique in the ZBDD in Figure 6.4. In our implementation, we check for these structures while constructing the ZBDD in a bottom up fashion. For the Figure 6.4, ZA and ZB are considered to be minimized. It should be noted that we check only two levels to perform this minimization. Hence, the method is not complete, but it is sound. In other words, our minimization is correct but all the cubes that can be minimized are not essentially minimized in the state-set ZBDD. It is easier to detect all such cubes if we have an ordered ZBDD. It should be noted that we can convert the general ZBDD into an ordered ZBDD and detect all the cubes that can be minimized. But, this will lead to an extra overhead that will generate a ZBDD with lesser cubes and not necessarily smaller size.

6.5 Experimental Evaluation

The suite of techniques discussed in sections 6.3 and 6.4 were implemented in C++ and integrated into a PODEM-like ATPG engine. We aimed at computing the one-cycle image for the all-zero

initial state of large ISCAS89 and ITC99 circuits. Note that different initial states could easily be used instead of the all-zero state. We conducted the experiments on a Pentium 4, 3GB RAM machine, running the Linux Operating System. We compare the basic ATPG and the efficient ATPG with SAT and BDD techniques. The time limit was set to 1800 seconds and memory limit was set to 1 million nodes. For comparison with a SAT-based technique, we modified MINISAT [27] to compute the image for sequential circuits. After each solution is obtained, an enlarged blocking clause (obtained by eliminating the primary input and present state variables only) is added to constrain the solution state space until the complete search space is explored. Then we use the inbuilt *compute_reach* command of VIS [83] with dynamic variable ordering option to compute the one-cycle image for sequential circuits using BDDs.

Table 6.1. Image computation for ISCAS '89 & ITC '99 circuits

Circuit	#FF	#gates	SAT		BDD		Basic ATPG			Efficient ATPG		
			#enum	Time(s)	#nodes	Time(s)	#enum	#nodes	Time(s)	#matches	#nodes	Time(s)
s1196	18	575	823	0.13	404	0.02	3187	7776	0.13	805	2326	0.08
s1269	37	634	4339	0.3	8423	0.1	9501	55K	0.36	1309	8071	0.11
s1512	57	887	6144	0.57	74	0.13	27724	56154	0.23	34	87	0.03
s9234	228	5866	24	0.14	6179	1.8	63	934	0.03	14	362	0.03
s3271	116	1728	-	T_O	1380	0.8	-	M_O	-	196	866	0.06
s3384	183	1937	-	T_O	322	0.5	-	M_O	-	60	144	0.03
s5378	179	3042	-	T_O	1M	54.7	-	M_O	-	1528	6495	0.34
s38417	1636	24K	-	T_O	3893	17.6	-	M_O	-	24	1692	0.1
s38584	1452	20K	66	0.51	11K	10.1	66	2131	0.06	8	1637	0.06
b11	31	770	64	0.01	55	0.02	64	153	0.02	7	27	0.02
b14	245	11K	-	T_O	-	T_O	-	M_O	-	32	215	0.04
b15	448	9K	1	0.23	734	11.8	1	451	0.03	1	451	0.03
b17	1415	32K	1	0.86	2735	263.6	1	1417	0.06	1	1417	0.06
b20	490	20K	-	T_O	-	T_O	-	M_O	-	34	460	0.06
b22	735	30K	-	T_O	-	T_O	-	M_O	-	34	805	0.07

Note: a) T_O - Time Out (1800s) *b) M_O - Memory Out (1M nodes)*

The results are reported in Table 6.1. The circuit name, number of flip-flops and number of gates are reported in Columns 1, 2 and 3, respectively. Columns 4 and 5 report the number of solutions and time taken by the SAT-based technique. Columns 6 and 7 report the size of the solution set and time taken by BDD based technique. The number of solutions, size of solution set and time taken by the basic ATPG are reported in Columns 8, 9 and 10. The same results are reported for our proposed ATPG in Columns 11, 12 and 13. We count each search-state match as an enumeration,

since we backtrack at that point. It can be seen that the proposed ATPG with enhanced heuristics consistently outperforms all other techniques. The basic ATPG aborts in many cases, such as s5378, s38417, b20 and b22, since the number of ZBDD nodes exceeds the pre-set limit. It is seen in Column 12 that the size of solution-set is reduced by several orders of magnitude because of the optimization techniques proposed in section 6.4. On the other hand, the SAT based technique aborts for these large circuits because they lack the circuit knowledge and try to specify all the variables in the CNF. Due to the lack of structure in the CNF, it is difficult to learn from equivalent search-states in SAT. For small circuits, such as s1196, s1269 and s1512, the time taken by SAT and ATPG are almost the same. But for larger circuits, ATPG with optimization techniques clearly outperforms SAT. The BDD based techniques perform very well for medium sized circuits and are able to complete for a few large ISCAS circuits such as s5378, s38417, s38584, due to partitioning and dynamic variable ordering. However, in these cases, the solution-set is represented in a more compact manner by the general ZBDD. This is mainly because the BDD in VIS follows a linear order, whereas our general ZBDD trades-off space for canonicity. The BDD based technique spends a lot of time in finding a suitable variable order for these large circuits and this leads to Time-Out for circuits such as b14, b20 and b22. In fact, VIS could not finish constructing the transition relation for b14 which has 11,000 gates. The ATPG technique, on the other hand, directly computes the image on the Boolean circuit structure and does not need a separate representation for the transition relation.

6.6 Summary

We have proposed a novel image computation technique using an ATPG engine which constructs the image set on-the-fly as a general ZBDD. We used observability based decision selection heuristics to choose an input decision that can propagate quickly to the next state flip-flop. In addition, search-state-based learning is incorporated into the ATPG engine to avoided searching repeated spaces. A proof for correctness has also been provided. Finally, we use simple minimization

techniques to reduce the size of the image ZBDD. Experimental results show that we can achieve several orders of magnitude improvement in both space and time over SAT and BDD based techniques for a number of large ISCAS 89 and ITC 99 circuits. Possible directions for future work include converting the image ZBDD into a Reduced Boolean Circuit (RBC) and developing an iterative framework for fixed point computation. Such a framework would help in promoting the proposed ATPG-based technique to check properties in Symbolic Model Checking.

Chapter 7

Path Delay Fault Testing

In this chapter, we target the problem of generating a complete test-suite for the path delay fault (PDF) model. We develop an ATPG engine that is based on an Incremental Satisfiability framework and learns from (1) static logic implications, (2) segment-specific clauses, and (3) unsatisfiability cores of each untestable partial PDF [89]. These learning techniques improvise the test generation for path delay faults that have common testable and/or untestable segments. The experimental results show that a significant portion of PDFs can be excluded dynamically in the proposed incremental SAT formulation for large benchmark circuits, thus potentially achieving speed-ups for PDF test generation.

7.1 Introduction

The increasing clock frequencies and reduced feature sizes have made delay testing a necessity. While various fault models have been proposed to capture the effect of delay defects, the path delay fault (PDF) model is the most accurate in characterizing the cumulative effect of distributed delays along each path in a circuit. However, the main bottle-neck in the PDF model is the exponential number of paths in a circuit. As a result, Automatic Test Pattern Generation (ATPG) suffers from temporal explosion if each path delay fault needs to be targeted in the circuit. In order to reduce the

number of PDFs that need to be considered for test generation, several methods have been proposed to identify the untestable PDFs *a priori*. In [90], static logic implications are used to identify many untestable path delay faults in a circuit. Other methods have been proposed in [91, 92] to quickly identify the untestable PDFs. In general, these techniques identify a set of PDFs that are untestable and predict a lower bound on the number of untestable PDFs. However, integrating these techniques into the test generation algorithm leads to an additional overhead for the ATPG engines that directly work on the circuit structure and use different logic systems [40, 93, 94]. More recently, in [95], the authors propose a Zero-suppressed Binary Decision Diagram (ZBDD) based technique to identify all the testable PDFs in a circuit. The PDFs are stored as a ZBDD and set operations are used to remove all the untestable PDFs from the ZBDD. Subsequently, an ATPG engine can be invoked to generate the test vectors for all the testable PDFs identified by their technique.

Boolean Satisfiability (SAT) was first used for delay testing in [96], in which all the paths in a circuit are enumerated, and hence it may not be applicable for large circuits. In [97], the authors suggest that incremental SAT is suitable for PDF testing. They provide a basic framework to generate non-robust test vectors for *all* the PDFs in the circuit. However, their framework is devoid of problem-specific learning techniques that can improve the performance of the SAT solver. To improve the performance of SAT, it has been suggested in [98, 99, 100] that circuit-related information can enrich the clause database of the SAT solver for equivalence checking and model checking problems. This opens an opportunity for problem-specific learning to discover the knowledge that could enhance the SAT solver.

In our work, we present efficient learning techniques to incrementally reduce the number of paths that need to be considered, thereby speeding up the Incremental SAT solver, for PDF test generation. We target the problem of generating a complete test suite for all the path delay faults in a circuit. The contributions of our work are as follows:

1. We convert the static logic implications in a circuit into clauses and add them to the clause database of SAT solver. Since implications were shown to be efficient in identifying the untestable PDFs in a circuit, they will help to quickly identify the untestable path delay faults during test generation.

2. For each fanout-free segment in the circuit, we try to sensitize the segment-fault constraints individually and store the conflict clauses generated during the search as *learned segment clauses*. Then, during the ATPG, we add these clauses along with the sensitization constraints for each fanout-free segment.

3. When an untestable partial PDF is encountered during test generation, we extract the unsatisfiable core of clauses. These clauses are used to identify other untestable PDFs, on the fly, during test generation.

The experimental results show that a significant portion of PDFs can be excluded dynamically in the proposed incremental SAT formulation for large benchmark circuits, thus potentially achieving speed-ups for PDF test generation.

The rest of the chapter is organized as follows. In section 7.2, we provide a brief introduction to incremental SAT and path delay faults. In section 7.3, we describe the mechanism to add static logic implications and *learned segment clauses* to the SAT solver. Section 7.4 presents the idea of extracting *unsatisfiable cores* for untestable PDFs and using them to avoid other untestable path delay faults. The experimental results on ISCAS 85 and ISCAS 89 circuits are reported in section 7.5 and section 7.6 summarizes the chapter.

7.2 Preliminaries

In this section, we introduce the preliminaries for the Incremental Satisfiability problem and how it is used for non-robust path delay fault ATPG.

7.2.1 Incremental Satisfiability (ISAT)

The Boolean Satisfiability problem is to determine whether a satisfying variable assignment, V, exists for a given Boolean formula, $\Phi(V)$, commonly expressed in conjunctive normal form (CNF). A CNF is a conjunction of clauses, where each clause is a disjunction of literals. A literal is a variable occurring in its positive or negative polarity. The general form for the formula is as follows:

$$\Phi(v_1, v_2, ...v_n) = C_1 . C_2 . ... C_m \tag{7.1}$$

where,

- Φ is a propositional formula in CNF
- v_i is the i^{th} Boolean variable in the CNF
- C_j is the j^{th} clause in the CNF

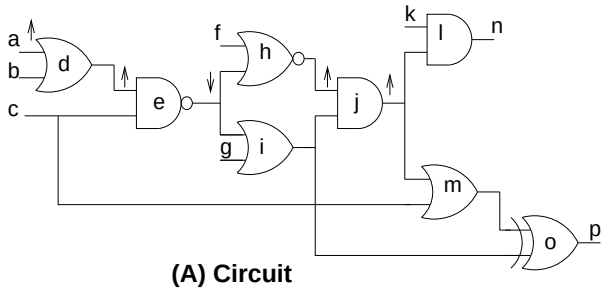
In conventional SAT, all the clauses in the CNF are manipulated to find a satisfying variable assignment for Φ. In the case of incremental SAT, we partition the clauses into different groups. Initially, only a single partition of clauses is in the clause database. If a satisfying assignment is found for the existing set of clauses, we add the next partition of clauses iteratively. If no satisfying assignment is found in an iteration, then the SAT solver can stop and report that no satisfying assignment exists for Φ, since a subset of clauses in Φ cannot be satisfied.

Incremental SAT is advantageous if the SAT solver can conclude that Φ is unsatisfiable using a smaller set of clauses. In the worst case, we have to add all the partitions into the clause database of the SAT solver. It should be noted that the solver can proceed from its previous variable assignments when we add-in the next partition of clauses to the SAT solver in the subsequent iteration. Since a huge number of untestable PDFs exist in a circuit [101], and many untestable path delay

faults have common untestable segments, incremental SAT can identify the untestable PDFs in groups and speed up the test generation.

7.2.2 Path Delay Fault Testing

For each structural path in the circuit from the primary input to the primary output, two path delay faults are associated - rising PDF and falling PDF. In order to detect a path delay fault, we require two test vectors $< t_1, t_2 >$, where a rising (falling) transition is asserted at the beginning of the path for the corresponding rising (falling) PDF. In addition to the initial transition, we sensitize the off-path inputs of the path to propagate the transition to the output. A test $< t_1, t_2 >$, for a given path delay fault, is said to be *robust* if the test can detect the fault independent of other path delay faults in the circuit. On the other hand, if the path delay fault can be masked by the presence of other faults in the circuit, then the test $< t_1, t_2 >$ is called a *non-robust* test. In our work, we target non-robust tests, since non-robust test generation is a direct application of incremental SAT and the non-robust conditions are present in robust tests as well. Nevertheless, the proposed techniques can be extended to robust test generation or any other circuit problem that is an application of incremental satisfiability by modifying the incremental clauses for each iteration in the CNF.



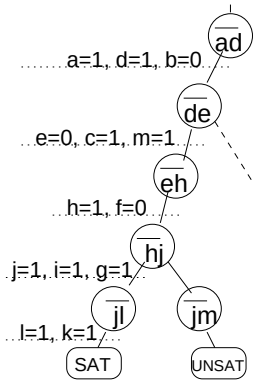
(A) Circuit

Path Recursion		Clauses Added
Depth	Segment	(On & Off path)
1	ad ↑	(a) (d) (−b)
2	de ↑	(−e) (c)
3	eh ↓	(h) (−f)
4	hj ↑	(j) (i)
5	jl ↑	(l) (k)
5	jm ↑	(m) (−c)

(C) Sensitization clauses

(−a + d) (−b + d) (a + b + −d) (c + e) (d + e) (−c + −d + −e) (−e + −h) (−f + −h) (e + f + h)
(−e + i) (−g + i) (e + g + −i) (h + −j) (i + −j) (−h + −i + j) (j + −l) (k + −l) (−j + −k + l)
(−c + m) (−j + m) (c + j + −m) (−i + −m + −o) (i + m + −o) (−i + m + o) (i + −m +o)

(B) Circuit CNF



(D) Incremental SAT

Figure 7.1. Incremental SAT for Non-Robust PDF Testing

For non-robust PDF test generation, it is sufficient to sensitize the off-path inputs in the second test vector t_2, such that the on-path transition propagates to the primary output. In Figure 7.1, we demonstrate the non-robust test generation for PDFs using incremental satisfiability. The circuit is illustrated in Figure 7.1 (A). The consistency clauses for the gates in the circuit is given in part (B) of the Figure. The sensitization clauses for each segment is shown in part (C). We start from segment $ad \uparrow$ and recursively add the sensitization clauses for the fanout segments. The recursive addition of segments is illustrated by the tree structure in Figure 7.1 (D). When a fanout stem is reached (eg. segment $hj \uparrow$), we add the segments in its branches recursively in a depth-first fashion. In this example, we first add $jl \uparrow$. If we reach a primary output during recursion, then all the sensitization clauses for the current PDF have been satisfied, and the variable assignments at the primary inputs is the test vector for that PDF. In this example, the rising path delay fault at $a-d-e-h-j-l$ is testable. Then, we backtrack in the segment recursion tree, delete the sensitization clauses of $jl \uparrow$ and try to sensitize the next fanout segment - $jm \uparrow$. If the SAT solver reports that a partial PDF is untestable, all PDFs with this partial PDF as a prefix are untestable. In this example, the partial rising PDF $a-d-e-h-j-m$ is untestable. So, the ATPG can backtrack immediately in the segment recursion tree without sensitizing the segment mp. In this way, the ATPG proceeds for all the paths in the circuit and generates test vectors for all the testable path delay faults in the circuit.

7.3 Static Learning

In general, for solving circuit problems using Boolean Satisfiability, the clauses for all the gates in the circuit are built to form the complete clause database. It has been shown in [98, 99, 100] that additional constraints can be added to the clause database to speed up the search process. In order to take advantage of static learning for path delay faults, we add the *static logic implications*, generated from the circuit, as well as certain *segment-specific clauses* to the clause database. It should be noted that these clauses are quickly generated in a preprocessing technique and are

computed only once, before the actual test generation.

7.3.1 Static Logic Implications

The implications in a circuit provide useful information that helps to deduce relations between different lines in the circuit. They have been widely used to detect a huge number of untestable path delay faults in [90, 91, 92]. In the Boolean Satisfiability domain, implications can be seen as the circuit information that aid in accelerating the Boolean Constraint Propagation (BCP) (the most time consuming routine in the SAT solver [23]). In order to speed up the SAT solver and detect untestable path delay faults faster, during ATPG, we convert the implications into clauses and add them to the circuit CNF. As direct implications are already encoded in the circuit clauses, we add only the indirect implications and extended backward implications to the circuit CNF. These implications are generated using the techniques proposed in [102].

Each implication is converted into a clause as follows. Suppose we have a static implication: $a \rightarrow b$. The clause equivalent of this implication is $(\bar{a} + b)$. Similarly, for all the implications obtained from the circuit, we add the corresponding clauses to the circuit CNF. Because of the low computational overhead involved in generating static logic implications, this step is simple. Though simple, it can enrich the clause database with additional constraints that will increase the deductive power of the SAT solver.

7.3.2 Segment-specific learned clauses

In incremental satisfiability for path delay fault testing, we solve the clauses for partial PDFs first and then augment the clause database incrementally. The clauses added for each increment corresponds to the sensitization of a segment-fault (also referred as segments in the sequel). The initial segments (partial PDFs) that form the common prefix for various path delay faults (see Figure 7.2

(A)) are targeted once and are re-used while backtracking in the path recursion. Similarly, the segments in the latter part of the circuit (near the primary outputs) may be the common post-fixes for different path delay faults (see Figure 7.2 (B)).

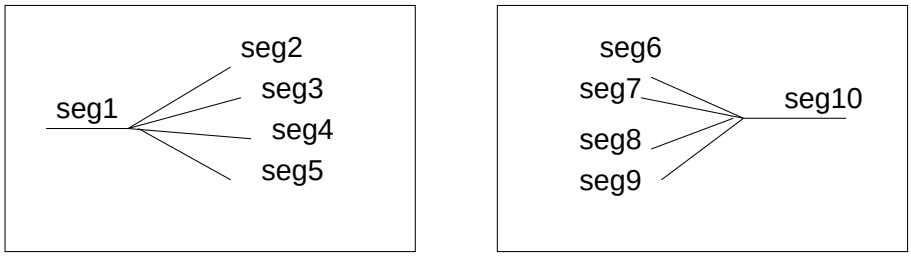


Figure 7.2. Prefix & Postfix partial PDFs

The existing test generation methods do not account for the latter scenario in the circuit. Moreover, in order to sensitize each segment, we add only the on-path and off-path clauses. It will be beneficial to the SAT solver if we add the clauses that prune the conflict search subspace for that segment. When a segment is encountered multiple times as a prefix/postfix, these clauses will guide the SAT solver to avoid the associated conflict-space and reach at a conclusion more quickly.

Motivated by these factors, we learn the clauses that can be added along with the sensitization constraints, for each segment, to speed up the SAT solver. As a pre-processing step, we assert the sensitization constraints for each segment and perform an all-solutions SAT search [65]. We store the conflict-induced clauses, generated during the SAT search, as *learned segment clauses*. All these conflict-induced clauses constrain the solution space for the target segment. These clauses should be satisfied (a necessary condition) for detecting any PDF involving this segment. It should be noted that a proper subset of the conflict clauses define a solution super-space for that segment. Hence, these clauses represent constraints for the given segment and will preserve the satisfiability of the original problem.

In general, solving for the sensitization constraints of a single segment is very easy. Moreover, it

is not necessary to complete the all-solutions SAT search for every segment, since each conflict clause can be individually stored as a *learned segment clause*. Considering the time constraints, it may be sufficient to learn the clauses only for certain segments, instead of all the segments. Due to the nature of segment recursion from input to output in incremental SAT, the segments with large number of fanout paths is more likely to be traversed fewer times (Figure 7.2 (A)) as compared to the segments with the same number of fanin paths (Figure 7.2 (B)). We use a scoring mechanism to determine the segments that will be traversed many times during PDF test generation. We assign the number of fanin paths in each segment (found using a linear time algorithm) as its score. Then, we compute the weighted average of all the scores and learn for segments whose scores are higher than the weighted average. Since the segments with higher scores will be encountered more times during ATPG, it is necessary to quickly traverse these segments to accelerate the ATPG.

7.4 Dynamic Learning from Unsatisfiable Cores of untestable partial PDFs

Definition 6 *Given an unsatisfiable CNF formula Φ, a subset of clauses, $\phi \subseteq \Phi$ that is unsatisfiable by itself is called an unsatisfiable core of clauses.*

The unsatisfiable cores were initially used to validate a SAT solver in [103, 104]. In their work, all the conflict clauses that are generated during SAT search and their resolvent clauses are stored externally. They propose an algorithm to generate an unsatisfiable core from these clauses. Starting from the final conflict clause, the algorithm backtraces through the resolvent clauses. Finally, the set of all original clauses reached during the backtrace is identified as the unsatisfiable core. We use the same procedure to generate the unsatisfiable core. However, we do not need to store the conflict clauses externally, since our objective is to extract the unsatisfiable core rather than validate the SAT solver.

In the segment recursion of incremental SAT, we obtain a partial PDF that is untestable. The corresponding clauses in the database of the SAT solver form an unsatisfiable CNF. The sensitization

clauses in the unsatisfiable core for this CNF can be mapped to a group of segments that cannot be sensitized together. All the path delay faults that contain these segments are guaranteed to be untestable. They are called the *untestable core of segments* (or simply untestable core) in the sequel. A motivating scenario is given in Figure 7.3. Suppose we obtain an unsatisfiable CNF corresponding to an untestable partial PDF $(a-b-c-d-e-f-g-h-i)$. The untestable core of segments is found to be $\{ab, ef, hi\}$. It can be concluded that all the path delay faults that contain all these three segments are untestable. This will help to avoid testing for other untestable path delay faults and to speed up the test generator.

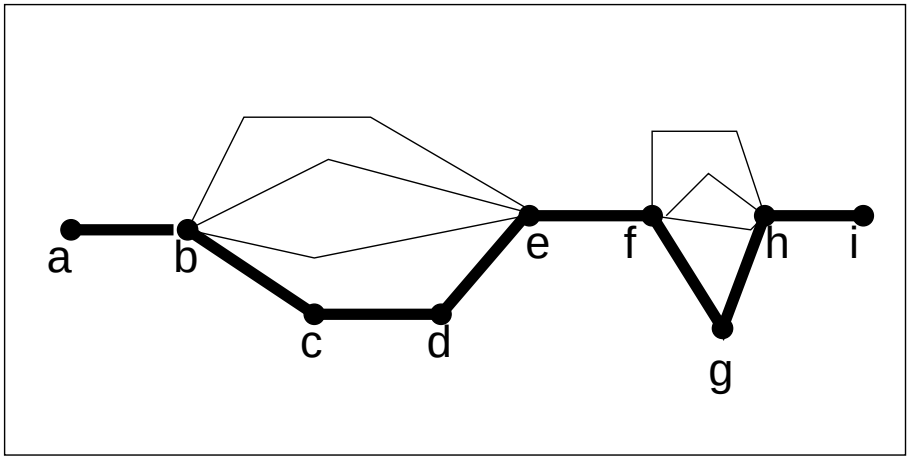


Figure 7.3. Unsatisfiable core of segments

After encountering each untestable partial PDF, we identify the untestable core of segments and store them in a table, T. If any partial PDF contains all segments of any untestable core, then that PDF is guaranteed to be untestable. Before choosing a segment for path recursion, we need to look up the table, T, to check if the current partial PDF will cover any of the untestable core of segments. If it covers any untestable core of segments, we ignore that segment and look for the next segment in the path recursion. This step helps to avoid one iteration of adding clauses, solving for SAT and deleting clauses in the incremental SAT framework.

Consider the untestable core of segments (UC) and the current partial PDF (PP) as *sets* of segments. If an untestable core of segments is a subset of the partial PDF, $UC \subseteq PP$, then we have to backtrack in the path recursion. However, checking for a subset in the table of unsatisfiability cores, for every iteration of ISAT, is time consuming. As an alternative, we introduce a *single watched segment* technique (motivated by the two-literal watching rule in zChaff [23]) to look for subsets on the fly.

While adding an untestable core to the table, T, we assign *one* segment, which is not yet chosen in the path recursion, as the watched segment for that row. For each segment, we store a watched list that contains the row numbers of the table, T, where it is a watched segment. Please note that each row corresponds to an untestable core. During test generation, before a segment is chosen, we traverse the rows of T in its watched list. For each row/untestable core, we search for the next available segment to be assigned as the watched segment for that core. If no free segment is available, then that core has been covered and we can conclude that the corresponding partial PDF is untestable.

The *watched segment* technique is explained in the following example. Let $P = \{s_1, s_2, s_3, s_4\}$ be the set of segments in the current partial PDF, and we have to choose s_5 as the next segment in the path recursion. Let us consider the following UTC, a table of untestable core of segments, where $*$ denotes the watched segment in each row.

$$\begin{aligned} UTC \;=\; & \{\{s_1, s_3, s_5*, s_7, s_9\}, && \text{(Row 1)} \\ & \{s_5, s_{10}, s_{15}*, s_{20}\}, && \text{(Row 2)} \\ & \{s_1, s_3, s_5*\}\} && \text{(Row 3)} \end{aligned}$$

The watched list for s_5 contains row 1 and row 3 in the table. For row 1, $\{s_1, s_3\}$ have already been chosen in P, and $\{s_7, s_9\}$ are free segments. So, we can change the watched segment and assign s_7 or s_9 as the next watched segment. However, for row 3, all the other segments in that row have

already been chosen in P. This means that the untestable core in row 3 will become a subset of the partial PDF if we choose s_5. Therefore, we avoid s_5 and look for the next segment in the segment recursion to proceed with the test generation. Note that we did not consider row 2 during this entire manipulation and it is not necessary to update the watched segments for the backtracks in segment recursion.

7.5 Experimental Evaluation

The above techniques were implemented in C++ and integrated with zChaff [23], downloaded from the web-site [26]. However, the proposed techniques can be integrated into any DPLL based SAT solver. In fact, it may be beneficial if we use a SAT solver with sophisticated incremental SAT solving capabilities such as MINISAT [27]. zChaff was chosen since it has an inbuilt feature to extract unsatisfiable cores. The experiments were conducted on a 3 GHz Pentium 4 machine with 1 GB RAM running the Linux OS. A non-robust PDF ATPG was implemented to generate test vectors for *all* the path delay faults in ISCAS 85 and ISCAS 89 benchmark circuits with a time limit of 5000 seconds. We compare our technique with [97] only, since it is the most recent Boolean Satisfiability technique proposed for PDF test generation.

The experimental results are shown in Table 7.1. For each circuit, the number of fanout-free segments and the number of path delay faults are first reported. Next, the results of PDF ATPG using incremental SAT are listed for (1) no learning (similar to the ISAT in [97]), (2) static learning only, and (3) combined static and dynamic learning. The final column of the table reports the total number of detected PDFs by all three methods. Note that each method is able to complete detection of all testable PDFs in the circuit. Under the Static-learning ISAT, both the number of clauses added and the time taken for test generation are reported. For the combined static and dynamic learning, the number of PDFs excluded due to unsatisfiability cores during dynamic learning, the number of PDFs excluded as a percentage (%) of total number of untestable PDFs, and the time taken for test

generation are given.

Table 7.1. PDF ATPG for ISCAS '85 & '89 circuits

Circuit	#FF-seg	#PDFs	Incremental Satisfiability						
			Basic	Static Learning		Static+Dynamic Learning			#Testable
			Time(s)	#cls	Time(s)	#PDFs excl.	% excl.	Time(s)	PDFs
c432	236	167.8K	0.59	480	0.46	7,044	**4.3**	0.49	2.5K
c499	264	18.9K	12.13	2.6K	13	1.5K	**14.7**	13.08	8.8K
c880	451	17.3K	4.51	137	4.59	135	**21.4**	4.58	16.7K
c1908	995	1.5M	309.25	2.1K	297.2	911K	**82.6**	257.21	355.2K
c1355	776	8.3M	1.4K	1,8K	1.5K	1.6M	**22.9**	1.4K	1.1M
c2670	1.4K	1.4M	407.87	1.3K	297.12	277.1K	**22.5**	297.2	130.6K
c3540	1.8K	57.3M	3.K	1.5K	3.6K	15.3M	**27.4**	3.4K	1.2M
c5315	2.9K	2.7M	911	1.6K	971.93	482.3K	**20.6**	919.36	342K
s635	315	2.5K	0.36	91	0.38	0	**0**	0.38	2.5K
s641	260	3.5K	1.08	54	1.09	248	**20.4**	1.09	2.3K
s713	320	43.6K	4.08	336	3.68	10K	**26.1**	3.39	4.9K
s991	436	14.9K	5.12	89	5.04	2.3K	**52.7**	4.92	10.6K
s1512	648	7K	3.95	434	3.81	1.2K	**48.2**	3.66	4.4K
s1269	646	79.1K	17.19	237	17.64	14.2K	**31.1**	16.9	33.4K
s1423	704	89.4K	25.49	285	26.32	11.4K	**25.7**	26.04	45.2K
s3271	1.7K	38.4K	51.81	5.8K	48.29	4.4K	**23.2**	47.59	19.3K
s3384	1.7K	39.6K	61.3	1.8K	57.15	367	**4.8**	57.03	32K
s5378	2.4K	27K	130.35	3.9K	123.76	1.3K	**24.7**	126.01	22K
s9234	3.5K	489.7K	678.04	3.6K	641.5	159K	**37.0**	635.06	59.8K

Note a) #FF-segments - #fanout-free segments
Note b) #Addtnl cls - #additional clauses learned due to static learning
Note c) excl. - untestable PDFs identified by untestable cores

From Table 7.1, it is observed that there is an improvement in time for many circuits, such as c1908, c2670, c3540, s5378 and s9234. The efficacy of dynamic learning is demonstrated by the number of untestable path delay faults identified by the untestable cores. The untestable cores are able to dynamically identify that more than 20% of the PDFs are eligible to be excluded during ATPG for most of the circuits. While there is some overhead in pruning the number of PDFs, the corresponding speedup is also obtained for larger circuits. For several small circuits, there is no appreciable improvement in time taken to complete the test generation. It is likely that the clauses added, during static learning for fanout-free segments, and the unsatisfiability cores, take more time than simply running the original SAT solver without learning for these cases. Finally, it should be

noted that the implementation of the proposed techniques and the variable ordering are deciding factors for the performance of a SAT solver. In zCHAFF, the solver has to re-adjust its clause database and variables' status, every time a new clause is added incrementally. So the addition of many statically learned clauses may lead to a significant time-overhead as seen in c1355.

7.6 Summary

In this chapter, we introduce static and dynamic learning techniques for an incremental satisfiability framework. We use the framework to generate a complete non-robust test suite for the path delay fault model in a circuit. We learn from static logic implications and segment-specific clauses, at each iteration of the increment, in order to enrich the clause database of the SAT solver. We use the unsatisfiability cores of unsatisfiable CNFs, generated for untestable PDFs, to identify other untestable path delay faults. Experimental results show that these unsatisfiable cores help to identify a large number of untestable PDFs and they are subsequently avoided during test generation.

Chapter 8

Conclusion

In this dissertation, we addressed two difficult problems in Verification and Testing: Unbounded Model Checking and Path Delay Fault Testing. We introduced search-space aware learning into the basic ATPG algorithm to accelerate solution generation. We analyzed the solution and conflict space and tuned the algorithm to make use of the previously explored solution spaces and avoid the conflict spaces. This helped to accelerate the underlying ATPG algorithm.

First, we developed an all-solutions ATPG that can be used as the core engine in our Unbounded Model Checking framework. We proposed an efficient decision selection heuristic that considers the "connectivity of variables" while making decisions and constructing the decision tree as a free BDD. This heuristic lead to smaller BDD and faster solution generation. We analyzed the search space explored during solution generation and introduced a cut-set metric to evaluate the use of a search state. This metric was instrumental in reducing the number of search states stored, thus leading to a significant memory savings. Experimental results, when comparing all-solutions ATPG with and without our techniques, showed that we are able to complete solution generation in cases where conventional heuristics fail.

Second, we considered the state-set that is generally stored as a decision diagram and needs to be manipulated in an iterative framework for unbounded model checking. We provided a decision

diagram to clause/circuit conversion technique, and demonstrated its impact on a SAT-based unbounded model checking framework. We proposed to convert each node in the decision diagram into a set of clauses, in contrast to the existing technique, where each path is translated into a set of clauses. Thus, we were able to achieve significant reduction in the size of state set clasues. The experimental results also showed that we can represent the state-set more compactly, using our node based conversion technique, as compared to the existing path-based conversion technique.

Third, we presented a novel cofactor-search technique for preimage computation. Instead of finding all solutions, using an ATPG engine, we geared the ATPG to find all the cofactors in the circuit. Basically, we restricted the decision search to the primary inputs and decomposed the remaining circuits as cofactors. Then, we analyzed the cofactor spaces, thus obtained, and proposed theorems to learn from previously explored search spaces. We compared the novel preimage computation technique with existing SAT based approaches. In general, it is conjectured that this technique is more efficient in cases where the reachable state space is sparse, since we attempt to capture a huge number of solutions in one cofactor. Experimental results showed that direct circuit cofactoring, with search-state induced learning, can lead to several orders of magnitude improvement in both run-time and memory as compared to some of the existing techniques.

Fourth, we developed a novel image computation procedure using a backtracing ATPG engine. We assert that the image computation problem is not symmetrical to preimage computation from a SAT/ATPG point of view. We explore the input space and project the values on the next state flip-flops as image. We store the image in a generalized decision diagram that is suitable to perform search-state learning. We compared our technique with existing SAT and BDD based techniques. Experimental results showed that we are able to obtain significant improvement over the existing techniques in both performance and capacity.

Finally, we introduced search space aware learning for an ATPG that targets the path delay fault model. We used an incremental satisfiability framework that works on a CNF representation of the

problem. Initially, we enrich the clause database with static logic implications in the circuit and segment specific sensitization clauses. During ATPG, we identify “untestable core of segments” from the “unsatisfiable core of clauses” obtain for untestable path delay faults. We use these segments to prune a huge number of untestable path delay faults, on the fly, during test generation. Experimental results on ISCAS 85 and ISCAS 89 circuits showed that a huge number of untestable path delay faults are avoided and this accelerates ATPG.

Bibliography

[1] S. Huang and K. T. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998.

[2] D. Brand, "Verification of Large Synthesized Designs," in *Proceedings of International Conference on Computer Aided Design*, pp. 534–537, 1993.

[3] C. Pixley, "A Computational Theory and Implementation of sequential hardware equivalence," in *Proceedings of Computer Aided Verification*, pp. 293–320, 1990.

[4] J. C. M. O. Coudert and C. Berthet, "Verifying Temporal Properties of Sequential Machines without building their state diagrams," in *Proceedings of Computer Aided Verification*, pp. 75–84, 1990.

[5] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[6] J. Waicukauski, E. Lindbloom, B. Rosen, and V. Iyengar, "Transition Fault Simulation," in *IEEE Design and Test of Computers*, pp. 32–38, 1987.

[7] J. Carter, V. Iyengar, and B. Rosen, "Efficient Test Coverage Determination for Delay Faults," in *Proceedings of International Test Conference*, pp. 418–427, 1987.

[8] A. Pramanick and S. Reddy, "On the Fault Coverage of Gate Delay Fault Detecting Tests," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 78–94, January 1997.

[9] K. Heragu, J. Patel, and V. Agrawal, "Segment Delay Faults: A new fault model," in *Proceedings of VLSI Test Symposium*, pp. 32–39, 1996.

[10] G. Smith, "Model for Delay Faults based upon Paths," in *Proceedings of International Test Conference*, pp. 342–349, 1985.

[11] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Pres, 2000.

[12] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proceedings of Design Automation Conference*, pp. 317–320, 1999.

[13] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a sat solver," in *Proceedings of FMCAD, Lecture Notes in Computer Science*, pp. 108–125, 2000.

[14] N. Eén and N. Sörensson, "Temporal Induction by Incremental SAT solving," in *Proceedings of BMC, ENTCS*, vol. 89, 2003.

[15] S. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. 27, pp. 509–516, June 1978.

[16] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 671–691, August 1986.

[17] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD package," in *Proceedings of Design Automation Conference*, pp. 40–45, 1990.

[18] F. Somenzi, "CUDD: CU Decision Diagram Package," in *http://vlsi.colorado.edu/ fabio/CUDD/*.

[19] S. Cook, "The complexity of theorem proving procedures," in *Third Annual ACM Symposium on the Theory of Computing*, pp. 151–158, 1971.

[20] M. Davis and H.Putnam, "A computing procedure for quantification theory," in *Journal of the ACM*, pp. 201–215, 1960.

[21] M. Davis, G. Logeman, and D. Loveland, "A machine program for theorem proving," in *Proceedings of the Communications of the ACM*, pp. 394–397, 1962.

[22] J. P. Marques-Silva and K. A. Sakallah, "A Search Algorithm for Propositional Satisfiability," in *IEEE Transactions on Computers, Vol. 48, No. 5*, pp. 506–521, 1999.

[23] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability," in *Proceedings of International Conference on Computer Aided Design*, pp. 279–285, 2001.

[24] L. Zhang and S. Malik, "Chaff: Engineering and Efficient SAT solver," in *Proceedings of Design Automation Conference*, pp. 530–535, 2001.

[25] L. Zhang and S. Malik, "The quest for efficient boolean satisfiability solvers," in *Proceeding of Computer Aided Verification*, pp. 17–36, 2002.

[26] S. Malik, "Boolean Satisfiability Research Group at Princeton," in *http://www.princeton.edu/ chaff/zchaff.html*.

[27] N. Eén and N. Sörensson, "An extensible sat-solver.," in *SAT '03: Proceedings of SAT*, 2003.

[28] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology and Transfer*, vol. 7, pp. 156–173, April 2005.

[29] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, July 1966.

[30] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C-30, pp. 215–222, March 1981.

[31] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, vol. C-32, pp. 1137–1144, December 1983.

[32] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation," in *IEEE Transactions on Computer Aided Design, Vol.7, No.1*, pp. 126–137, 1988.

[33] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[34] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing*. Kluwer Academic Publishers, 2000.

[35] N. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2003.

[36] A. Gupta, Z. Yang, P. Ashar, and A. Gupta, "Sat-based image computation with application in reachability analysis," in *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, LNCS*, vol. 1954, pp. 354–371, 2000.

[37] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining Decision Diagrams and SAT Procedures for Efficient symbolic model checking," in *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pp. 124–138, 2000.

[38] P. A. Abdullah, P. Bjesse, and N. Een, "Symbolic Reachability Analysis based on SAT solvers," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 1785 of Lecture Notes in Computer Science*, pp. 124–138, 2000.

[39] A. Krstić and K. T. Cheng, *Delay Fault Testing for VLSI Circuits*. Kluwer Academic Publishers, 1998.

[40] K. Fuchs, M. Pabst, and T. Rossel, "RESIST: A Recursive Test Pattern Generation Algorithm for Path Delay Faults considering Various Test Classes," *IEE Transactions on CAD of Integrated Circuits and Systems*, vol. 13, no. 12, pp. 1550–1561, 1994.

[41] K. Fuchs, F. Fink, and M. H. Schulz, "DYNAMITE: An Efficient Automatic Test Pattern Generation System for Path Delay Faults," *IEE Transactions on CAD*, vol. 10, pp. 1323–1335, October 1991.

[42] K. Chandrasekar and M. S. Hsiao, "Decision selection and learning for an 'all-solutions atpg engine'," in *Proceedings of the IEEE International Test Conference on Test*, pp. 607–616, 2004.

[43] K. Chandrasekar and M. S. Hsiao, "Atpg-based preimage computation: Efficient search space pruning with zbdd," in *HLDVT '03: Proceedings of the 8th IEEE International High-Level Design Validation and Test Workshop*, pp. 117–122, 2003.

[44] S. Sheng and M. Hsiao, "Efficient preimage computation using a novel success-driven atpg," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 822–827, 2003.

[45] K. L. McMillan, "Applying SAT methods in unbounded Symbolic Model Checking," in *Proceedings of International Conference on Computer Aided Verification, Vol. 2404 of Lecture Notes in Computer Science*, pp. 250–264, 2002.

[46] H.-J. Kang and I.-C. Park, "Sat-based unbounded symbolic model checking," in *DAC '03: Proceedings of the 40th conference on Design automation*, pp. 840–845, 2003.

[47] F. Lu, L. Wang, K. Cheng, and R. Huang, "A Signal Correlation Guided ATPG Solver and Its Applications For Solving Difficult Industrial Cases," in *Proceedings of Design Automation Conference*, pp. 436–441, 2003.

[48] F. Brglez, "On testability of combinational networks," in *Proceedings of the International Symposium on Circuits and Systems*, pp. 221–225, 1984.

[49] L. H. Goldstein, "Controllability/observability analysis of digital circuits," *IEEE Transactions on Circuits and Systems*, vol. 26, pp. 685–693, September 1979.

[50] S. Seth, L. Pan, and V. D. Agrawal, "Predict: Probabilistic estimation of digital circuit testability," in *Proceedings of Fault Tolerant Computing Symposium*, pp. 220–225, 1985.

[51] S. Chang, W. Jone, and S. Chang, "Tair: Testability analysis by implication reasoning," *IEEE Transactions on Computer Aided Design*, vol. 19, pp. 152–160, January 2000.

[52] M. I. and. Parthasarathy and K.-T. Cheng, "SATORI - A Fast Sequential SAT Engine for Circuits," in *Proceedings of International Conference on Computer Aided Design*, p. 320, 2003.

[53] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," in *Proceedings of International Conference on Computer Aided Design*, pp. 42–47, 1993.

[54] M. Thornton, J. Williams, R. Drechsler, and N. Drechsler, "Variable re-ordering for shared Binary Decision Diagrams using output probabilities," in *Proceedings of Design Automation and Test in Europe*, pp. 758–759, 1999.

[55] S. Panda and F. Somenzi, "Who are the variables in your neighbourhood," in *Proceedings of International Conference on Computer Aided Design*, pp. 74–77, 1995.

[56] P. Chung, I. Hajj, and J. Patel, "Efficient Variable Ordering Heuristics for Shared ROBDD," in *Proceedings of ISCAS*, pp. 1690–1693, 1993.

[57] F. Aloul, I. Markov, and K. Sakallah, "MINCE: A Static Global Variable-ordering for SAT and BDD," in *Proceedings of IWLS*, 2001.

[58] F. Aloul, I. Markov, and K. Sakallah, "FORCE: A Fast and Easy-To-Implement Variable Ordering Heuristic," in *Proceedings of GLSVLSI*, pp. 116–119, 2003.

[59] J. Gregov and C. Meinel, "Efficient Boolean Manipulation with OBDD's can be extended to FBDDs," *IEEE Transactions on Computers*, vol. 43, no. 10, pp. 1197–1209, 1994.

[60] J. Giraldi and M. L. Bushnell, "EST: The New Frontier in Automatic Test Pattern Generation," in *Proceedings of Design Automation Conference*, pp. 667–672, 1990.

[61] J. P. Marques-Silva and K. A. Sakallah, "Dynamic Search-Space Pruning Techniques in Path Sensitization," in *Proceedings of Design Automation Conference*, pp. 705–711, 1994.

[62] W. Kunz and D. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems," *IEEE Transactions on CAD*, vol. 13, no. 9, pp. 1149–1158, 1994.

[63] I. Hamzaoglu and J. H. Patel, "New Techniques for Deterministic Test Pattern Generation," in *Proceedings of VLSI Test Symposium*, pp. 446–452, 1998.

[64] C. Wang, S. M. Reddy, I. Pomeranz, L. Xiang, and J. Rajski, "Conflict driven techniques for improving deterministic test pattern generation," in *Proceedings of International Conference on Computer Aided Design*, pp. 87–93, 2002.

[65] B. Li, M. Hsiao, and S. Sheng, "A novel SAT all-solutions solver for efficient preimage computation," in *Proceedings of Design Automation and Test in Europe*, pp. 272–277, 2004.

[66] M. K. Iyer, G. Parthasarathy, L. C. Wang, and K. T. Cheng, "On the development of ATPG based Satisfiability Checker," in *Proceedings of Microprocessor Test and Verification Workshop, IEEE*, 2002.

[67] I. Pomeranz and S. Reddy, "An Efficient Non-enumerative Method to Estimate Path Delay Fault Coverage," in *Proceedings of International Conference on Computer Aided Design*, pp. 560–567, 1992.

[68] K. Chandrasekar and M. S. Hsiao, "Efficient State-set Management for SAT based Preimage Computation," in *Proceedings of International Conference on Computer Design*, pp. 585–590, 2005.

[69] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient sat-based unbounded symbolic model checking using circuit cofactoring," in *ICCAD '04: Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 510–517, 2004.

[70] M. K. Iyer, G. Parthasarathy, and K. T. Cheng, "SATORI - A fast sequential SAT engine for circuits," in *Proceedings of International Conference on Computer Aided Design*, p. 320, 2003.

[71] F. Lu, M. K. Iyer, G. Parthasarathy, and K. T. Cheng, "An efficient sequential SAT solver with improved search strategies," in *Proceedings of Design Automation and Test in Europe*, pp. 1102–1107, 2005.

[72] S. Minato, "Zero-suppressed BDDs and their applications," in *Proceedings of Software Tools for Technology Transfer*, pp. 156–170, 2001.

[73] A. Mishchenko, "Extra v 2.0: Software library extedning cudd," in *http://www.ee.pdx.edu/ alanmi/research/extra.htm*.

[74] K. Chandrasekar and M. S. Hsiao, "Search-space Aware Cofactor Expansion for Circuit Based Preimage Computation," in *to be submitted*.

[75] A. Mischenko, "ABC: A System for Sequential Synthesis and Verification, Release 50905," in *http://www.eecs.berkeley.edu/ alanmi/abc/*.

[76] H. H. H. Jin and F. Somenzi, "Efficient conflict analysis for finding all satisfying assignments of a boolean circuit," in *Proceedings of TACAS*, pp. 750–753, 2005.

[77] P. Bjesse and A. Borälv, "DAG-aware Circuit Compression for Formal Verification," in *Proceedings of International Conference on Computer Aided Design*, pp. 42–49, 2004.

[78] A. Kuehlmann, M. Ganai, F. Krohm, and V. Paruthi, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer Aided Design*, vol. 21, no. 12, pp. 1377–1394, 2002.

[79] K. Chandrasekar and M. S. Hsiao, "Forward Image Computation with Backtracing ATPG and Incremental State-set construction," in *Proceedings of Great Lakes Symposium on VLSI*, pp. 254–259, 2005.

[80] A. Gupta, A. Gupta, Z. Yang, and P. Ashar, "Dynamic detection and removal of inactive clauses in sat with application in image computation," in *DAC '01: Proceedings of the 38th conference on Design automation*, pp. 536–541, 2001.

[81] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-based decision heuristics for image computation using sat and bdds," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pp. 286–292, 2001.

[82] K. L. McMillan, "Interpolation and sat-based model checking," in *CAV '03: Proceedings of the 15th International Conference on Computer Aided Verification, LNCS*, vol. 2725.

[83] T. V. Group, "Vis: A system for verification and synthesis," in *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pp. 428–432, 1996.

[84] I.-H. Moon, G. D. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pp. 73–90, 2000.

[85] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi, "To split or to conjoin: the question in image computation," in *DAC '00: Proceedings of the 37th conference on Design automation*, pp. 23–28, 2000.

[86] A. Mishchenko, "An introduction to zero-suppressed binary decision diagrams," in *http://www.ee.pdx.edu/ alanmi/research*.

[87] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, C.-Y. Huang, and F. Brewer, "Aquila: an equivalence checking system for large sequential designs," *IEEE Transactions on Computers*, vol. 49, pp. 443–464, May 2000.

[88] J. Giraldi and M. L. Bushnell, "Search state equivalence for redundancy identification and test generation," in *Proceedings of the IEEE International Test Conference on Test*, pp. 184–193, 1991.

[89] K. Chandrasekar and M. S. Hsiao, "Integration of Learning Techniques into Incremental Satisfiability for Efficient Path-Delay Fault Test Generation," in *Proceedings of Design Automation and Test in Europe*, pp. 1002–1007, 2005.

[90] K. Heragu, J. Patel, and V. Agrawal, "Fast Identification of Untestable delay faults using implications," in *Proceedings of International Conference on Computer Aided Design*, pp. 642–647, 1997.

[91] Z. Li, R. Brayton, and Y. Min, "Efficient Identification of Non-Robustly Untestable Path Delay Faults," in *Proceedings of International Test Conference*, pp. 992–997, 1997.

[92] Y. Shao, S. Reddy, S. Kajihara, and I. Pomeranz, "An Efficient Method to Identify Untestable Path Delay Faults," in *Proceedings of Asian Test Symposium*, pp. 233–238, 2001.

[93] Y. Shao, I.Pomeranz, and S. Reddy, "Path delay fault test generation for standard scan designs using state tuples," in *Proceedings of Design Automation Conference*, pp. 767–772, 2002.

[94] D. Battacharya, P. Agrawal, and V. Agrawal, "Test Generation for Path Delay Faults using Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 434–447, 1995.

[95] S. Padmanabhan and S. Tragoudas, "Using ZBDDs and BDDs for efficient identification of testable path delay faults," in *Proceedings of Design Automation and Test in Europe*, pp. 50–55, 2004.

[96] C. Chen and S. Gupta, "A Satisfiability-based Test Generator for Path Delay Faults in Combinational Circuits," in *Proceedings of Design Automation Conference*, pp. 209–214, 1996.

[97] J. Kim, J. Whittemore, J. Silva, and K. Sakallah, "On Applying Incremental Satisfiability to Delay Fault Testing," in *Proceedings of Design Automation and Test in Europe*, pp. 380–384, 2002.

[98] M. Ganai, L. Zhang, P. Ashar, A.Gupta, and S. Malik, "Combining the strengths of circuit-based and CNF-based algorithms for a high performance SAT solver," in *Proceedings of Design Automation Conference*, pp. 747–750, 2002.

[99] F. Lu, L. Wang, K.-T. Cheng, and R. Huang, "A Circuit SAT solver with Signal Correlation Guided Learning," in *Proceedings of Design Automation in Europe*, pp. 892–897, 2003.

[100] R. Arora and M. Hsiao, "Enhancing SAT based Equivalence Checking using static logic implications," in *Proceedings of High Level Design Validation and Test*, pp. 63–68, 2003.

[101] K.-T. Cheng and H. Chen, "Classification and Identification of Nonrobust Untestable Path Delay Faults," *IEEE Transaction on CAD Integrated Circuits and Systems*, vol. 15, no. 8, pp. 843–853, 1996.

[102] J. Zhao, E. Rudnick, and J. Patel, "Static Logic Implications with application to redundancy identification," in *Proceeding of VLSI Test Symposium*, pp. 288–293, 1997.

[103] L. Zhang and S. Malik, "Validating SAT solver Using an Independent Resolution-based Checker: Practical Implementations and other Applications," in *Proceedings of Design Automation and Test in Europe*, pp. 880–885, 2003.

[104] E. Goldberg and Y. Novikov, "Verification of Proofs of Unsatisfiability for CNF formulas," in *Proceeding of Design Automation and Test in Europe*, pp. 886–891, 2003.

[105] T. Fujino and H. Fujiwara, "An Efficient Test Generation Algorithm Based on Search State Dominance," in *Proceedings of Fault Tolerant Computing*, pp. 33–40, 1992.

Vita

Kameshwar Chandrasekar was born in Chennai, a metropolitan city in India. He did his early education in Chennai. He joined Government College of Technology, Coimbatore to obtain his technical education in Electronics and Instrumentation Engineering. He obtained his Bachelor of Engineering (BE) degree in 2000. He then worked as a Software Engineer at iNautix Technologies, India for a year. He joined Virginia Tech in August 2001 and then completed his Masters, in Electrical Engineering, in July 2003, under the guidance of Dr. Hsiao. He joined Dr. Hsiao and his research group in January 2002 and since then has been involved in research related to ATPG and Model Checking. His technical interests include VLSI Testing and Formal Verification Methods. He will be completing his PhD in May 2006 and then joining the ATPG group at Intel, Santa Clara.