SAT-BASED ATPG FOR DIGITAL INTEGRATED CIRCUITS BASED ON MULTIPLE OBSERVATIONS

SAT-BASED ATPG FOR DIGITAL INTEGRATED CIRCUITS BASED ON MULTIPLE OBSERVATIONS

BY
DAVID WING YIN LEUNG, B. ENG. & MGT. (COMPUTER)
JANUARY 2008

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

MASTER OF APPLIED SCIENCE (2008) McMaster University
(Electrical and Computer Engineering) Hamilton, Ontario

TITLE:	SAT-based ATPG for Digital Integrated Circuits Based on Multiple Observations
AUTHOR:	David Wing Yin Leung, B. Eng. & Mgt. (Computer)
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xi,67

Abstract

This thesis presents a new approach to improve the efficiency of defect screening during manufacturing test of digital integrated circuits through the use of multiple observations during test generation. To address the limitations of test sets generated based on the single stuck-at fault model, we combine the advantages of multiple-detect and detection at all observable outputs in order to generate test sets that can improve surrogate detection.

Imposing additional constraints, such as multiple observations, on the test generation process motivates the development of a new constrained automatic test pattern generation (ATPG) work flow that leverages the recent advancements in the Boolean satisfiability (SAT) problem. Building this ATPG work flow brings its own technical challenges and solutions described in detail in this thesis. To assess the effectiveness of the test sets generated by the proposed ATPG work flow, we evaluate them using coverage metrics for fault models that are not targeted explicitly during test generation.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Nicola Nicolici, for providing me an opportunity to work on some interesting projects and for his patience and understanding for the prolonged delay of my work.

I am very grateful to my colleagues in the Computer-Aided Design and Test (CADT) Research Group, especially Johnny Xu, Henry Ko, Adam Kinsman and Ehab Anis, who have inspired me a lot and assisted me in many ways. I would also like to acknowledge Gennum Corporation for their financial and technical support, and Drs. Dumitrescu and Sirouspour for being my examiners and providing useful feedback to my thesis. Moreover, I would like to thank the graduate students, faculty, administrative and technical staff in Department of Electrical and Computer Engineering at McMaster University for their assistance during my study and research.

Last but not least, my parents Amy and Thomas, and my sister Phyllis deserve a great deal of thanks for their love, trust and continuous support which has made my work possible. I will not forget my friends who have cheered me up and shared happiness with me.

Terms & Abbreviations

ATPG Automatic Test Pattern Generation
CAD Computer-Aided Design
ATE Automatic Test Equipment
BCE Bridging Coverage Estimate
BCP Boolean Constraint Propagation
clause a disjunction of literals
CNF Conjunctive Normal Form
conflict a variable is implied to be both true and false
constraint conditions to restrict the search in a solution space
CUT Circuit Under Test
D-ALG D-Algorithm
decision a choice to make for assigning a value to a Boolean variable
defect a physical flaw in a circuit which produce incorrect behaviour
DPLL David-Putnam-Logemann-Loveland (Algorithm)
excitation the essential assignments to trigger a fault
FAN FAN-out-oriented test generation algorithm
fault a model of a defect that is to be targeted in ATPG
f-o pair fault-output pair, a pair of a reachable output and its corresponding fault
GE Gate Exhaustive (Fault Model)
literal positive or negative phase of a Boolean variable
PODEM Path Oriented DEcision Making
PASSAT PAttern Search using SAT
reachable output an output in the output cone of a fault

SAT Boolean SATisfiability (Problem)

scan a design-for-test methodology to access internal states of sequential circuits

sensitization the process to propagate fault effect to some outputs

TARO Transition fault to All Reachable Outputs

TEGUS TEst Generation Using Satisfiability

test compaction a process to reduce the number of patterns

ULA Undetect Look-Ahead

VSIDS Variable State Independent Decaying Sum (Decision-making Strategy)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Digital integrated circuits have been the enabling force to the recent advancements in many economic sectors, such as computer industry, consumer electronics and communication and information technologies, only to name a few. As the design complexity continues to grow and more transistors are integrated onto a single silicon die, the verification and test tasks are becoming increasingly difficult. Verification focuses on the comparison between the implemented design and its specification. Manufacturing test is concerned with screening out the defective chips after fabrication. Given the huge solution space of all the input patterns that can be applied to a circuit to screen out the fabrication defects, intelligent fault modeling and test generation are required to ensure that manufacturing test is done in a cost-effective manner.

The aim of the work described in this thesis is to give a new perspective on fault effect observation on the existing fault models. To better illustrate how the proposed method can be beneficial in the screening of fabricated chips, it is essential to first understand the basics of manufacturing test and test generation. The conceptual flow for the design of very large scale integrated (VLSI) circuits will be summarized in Section 1.1 and manufacturing test will be briefly introduced in Section 1.2. The concept of fault models and test generation will be discussed in Sections 1.3 and 1.4. Test application is discussed in Section 1.5. Finally, the motivation and the organization of this thesis is provided in Section 1.6.

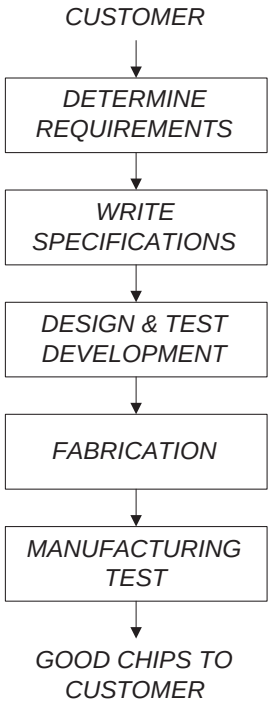

Figure 1.1: VLSI Realization Process [1]

1.1 VLSI Design

The realization of a VLSI design starts with determining requirements and writing specifications as shown in Figure 1.1.

Requirements are the criteria defined by the users that need to be satisfied by the designers. They are often related to the specific application of what the target product. Specifications can be set up once the requirements are determined. Usually specifications include various characteristics of the product to be built, namely functional, operating, physical, environmental and economical characteristics. The design process produces necessary information for fabrication, such as system-level structure of blocks and logic gates and physical layout.

Fabrication process involves photolithography, etching, ion implantation, chemical treatments and then dicing and packaging. Manufacturing tests are applied to chips to find out which ones are defective. The purpose of testing is only to determine if

the circuit is faulty, whereas the purpose of diagnosis is to determine what is the root cause of failure. Therefore diagnosis tests can also be done subsequently in order to determine how the design/process can be fixed.

The reader is to referred to [1, 19, 21] for a detailed introduction to the implementation cycle for digital integrated circuits.

1.2 Manufacturing Test

The fabrication process of semiconductor devices is prone to erroneous behavior due to anomalies in wafer processing, failure of manufacturing equipment, impurities in materials, or human errors. In order to detect and screen out the faulty circuits, manufacturing tests are employed. According to [1], manufacturing tests can be categorized as follows:

- *Parametric Tests*: DC parametric tests detect problems related to shorts and opens, maximum current, leakage, output drive current, and threshold level, while AC parametric tests detect defects related to propagation delay, setup and hold time, functional speed, access time, refresh and pause time, and rise and fall time. These tests are usually technology-dependent so only the physical characteristics of the devices are required but not the functionality of the design.

- *Functional Tests*: Functional Tests exhaustively test all input combinations of a circuit. This is equivalent to applying 2^n patterns to an n-input circuit. For a 64-bit ripple-carry adder, it will require 2^{129} patterns to test the circuit and it would take 2.158 x 10^{22} years for an automatic test equipment (ATE) that runs at 1GHz. Therefore a complete functional test is infeasible for a complex digital circuit due to the exhaustive nature of the test and can only be used for testing certain critical components in a large design.

- *Structural Tests*: On the contrary, structural tests verify a circuit according to its logical structure, or its *netlist*. Different fault models are introduced to model various types of possible logic and timing faulty behaviors caused by

electrical defects. A widely used fault model is the single stuck-at fault model, which assumes a fix logic 0 (sa0) or logic 1 (sa1) on a particular line in a logic network. For the same 64-bit ripple-carry adder, there are only 1728 stuck-at faults in total so at most 1728 patterns are required to test the adder. It only takes 1.728ms to apply these patterns on the same 1GHz ATE.

1.3 Fault Models

In this section we discuss three main fault models that are used to assess the effectiveness of a test set.

1.3.1 Stuck-at Fault Model

As briefly introduced in the previous section, in the stuck-at fault model a fault is assumed to affect only an interconnection between logic gates in a circuit. Stuck-at 0 (sa0) is a fault that assumes a logic state 0 at a particular fault site, disregarding how the line is driven; whereas stuck-at 1 (sa1) fault assume a logic state 1. Therefore, sa0, sa1 and fault-free are the three possible states in the model. Any combinations of faulty and fault-free lines are regarded as faults so there are $3^n - 1$ possible line combinations for an n-line circuit. Obviously, modeling every combination even for a moderate n is impossible. If only the combinations of one stuck-at fault are considered, the *single stuck-at fault model* can have only 2n faults, which makes it computationally feasible for VLSI circuits (where n lies in the tens to hundreds of thousands to even millions range). Furthermore, through fault equivalence [1], the number of faults can be further reduced.

The quality of a test set is often determined by *fault coverage*, which is the ratio of observable faults over the total number of faults. Therefore, most traditional stuck-at fault automatic test pattern generation (ATPG) algorithms (more details on ATPG are given in the next subsection) opt to achieve a high fault coverage of single stuck-at faults. This is despite the fact that there are physical defects that are not properly screened only through single stuck-at fault test sets. Therefore other fault

models have been proposed to deal with this problem. However, when using other models, this also increases the ATPG complexity. In order to increase the screening effectiveness, multiple-detect, or n-detect, stuck-at ATPG has been introduced [14]. Instead of sensitizing all faults once, which is sufficient to yield high fault coverage, all faults are sensitized several times. It is expected that multiple sensitizations of a fault excite a fault through different paths, such that untargeted faults, may also be detected. Surrogates, which are defects not modeled explicitly by the model used, could have been detected [6]. For example, a bridging fault (discussed later in this section), which usually has two excitation points, may become a surrogate detection. Through single stuck-at fault model, if one of the excitation points of the bridging fault is targeted, the other point could have been excited and sensitized. Multiple sensitizations increase the chance of having the other excitation point being sensitized.

1.3.2 Delay Fault Model

Another widely used fault model is the delay fault model. While the stuck-at fault model is suitable for timing-independent problems, the delay fault model is particularly useful for timing-dependent defects [31]. An illustration of timing-dependent defects is shown in Figure 1.2. When the inputs of a gate change, the outputs of the gate may change according to the logic function. The change is not instant, and would take certain amount of time to finish the transition of the logic state. A defect occurs when a transition takes more time than expected to be carried out and erroneous data is latched into the receiving state elements. There are two types of delay fault models, namely transition fault model and path delay fault model.

The *transition fault model* is also known as the *gate delay fault model*. In order to trigger a transition, a pair of patterns is required to sensitize a fault. The initialization pattern first initializes the fault site to a logic 0, whereas the capture pattern excites the fault site with logic 1 and sensitizes it to an observation output. The second step is very similar to stuck-at fault so a slow-to-rise (STR) fault behaves like a temporary sa0 fault. Likewise, a slow-to-fall (STF) fault is like a temporary sa1 fault [33].

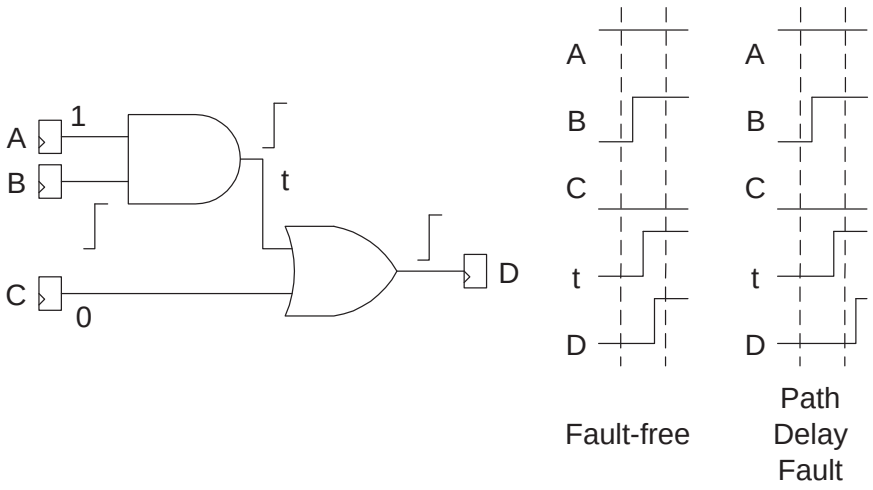

Figure 1.2: Illustration of Timing Failures

Besides multiple detection, the concept of Transition Fault to all Reachable Outputs (TARO) was introduced in [31]. TARO detects each transition fault as many times as the fault is observed at all the observable outputs at least once. A fault can be detected by only one pattern, while another may require a few patterns to be observed at all reachable outputs. It was demonstrated through experiments that TARO is capable of screening chips which escaped tests of 100% single stuck-at and 100% transition fault coverage.

Unlike the transition fault model, the *path delay fault model* considers the accumulated delay across a series of gates. A path refers to a series of logic gates in a circuit that begins from an input and it ends at an output. A transition at the beginning of the path will also trigger transitions through the path until its endpoint. A path delay fault occurs when a transition at the beginning of a path cannot propagate to the end of the path in one clock period. Again a two patterns test is required. The initialization pattern provides all the required off-path inputs along the path while the capture pattern triggers the transition [28]. Enumerating all the paths in a moderate-sized circuit is intractable. Therefore, the path delay fault ATPG tools often generate tests for the most time-sensitive paths (or critical paths) in the circuit. Besides screening for timing-dependent defects, speed binning is another major application of path delay test.

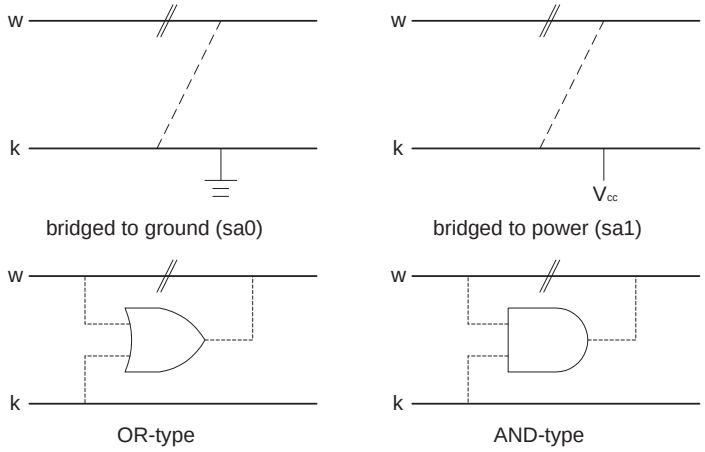

Figure 1.3: Four Types of Low Resistive Bridging Faults

1.3.3 Bridging Fault Model

Under the low resistive bridging model, there are four types of bridging faults between wires w and k, as shown in Figure 1.3. If k is power or ground line, it dominates w where $w = k$, and it behaves like a stuck-at fault. The defect can also be in OR-type model, where $w = OR(w, k)$. If k is set to logic 1, the defect can be modeled as sa1 fault on wire w. The chance to detect an OR-type defect is $(1 - (1 - p)^n)$ where p is the probability of $k = 1$, which is assumed to be one half, and n is the number of times the corresponding stuck-at fault is detected. Similarly for AND-type model, where $w = AND(w, k)$, it can be modeled as sa0 fault on wire w as $k = 0$.

Bridging Coverage Estimate (BCE) [2] is a metric to estimate low resistive bridging defects coverage with single stuck-at fault test set, based on the probabilistic model discussed above. Provided a test set T and its target fault list F, BCE is defined as

$$BCE = \sum_{i=1}^{n} \frac{f_i}{|F|}(1 - 2^{-i})$$

where f_i is the number of stuck-at faults detected i times by T, $|F|$ is the total number of stuck-at fault in F, and n is the maximum number of detections that a fault can be detected by T. In practice, n is usually limited to 10, which results in an upper bound of 99.9%. This is practically regarded as accurate enough to evaluate the quality of the test set.

1.3.4 Gate Exhaustive Model

In the gate exhaustive (GE) model [3], all reachable input combinations of all gates in the circuit are tested and the gate responses are observed. It can be modeled with the use of the single stuck-at fault [15]: sa1(sa0) can be used to represent a GE fault where a gate output is expected to be 0(1). Instead of assuming these faults on every single line of the circuit, only lines at gate outputs are considered. Though, all the combinations on the inputs of every single gate in the circuit are required to test the respective gate exhaustively.

A metric to compute GE coverage for evaluating stuck-at fault tests was introduced in [15]. It is defined as

$$GECoverage = \sum_{i=1}^{|F|} \frac{CC(f_i)}{|F|}$$

where $|F|$ is the total number of gate-output stuck-at faults; f_i is the i-th fault in the circuit; and CC is the coverage credit, which is the ratio of the number of distinct gate input combinations that detect a fault to the maximum number of gate input combinations that can detect the same fault. It is important to note that unlike [15], [3] does not consider the nonobservable input combinations, which include the unreachable input combinations and input combinations that cannot be sensitized to any observation output. Such input combinations can only be identified during test generation, but not through fault simulation. It is obvious that the maximum number of gate input combinations used in coverage credit will be larger when nonobservable input combinations are included in the above formula.

1.4 Test Pattern Generation

Given a fault model to perform a structural test, a set of patterns (or test set) needs to be generated. A set of patterns consists of a set of input vectors and output responses. An input vector is the set of logic values which are applied to the inputs of the circuit to sensitize a targeted fault. An output response is the set of the corresponding fault-free output values at the outputs of the circuit when an input vector is applied.

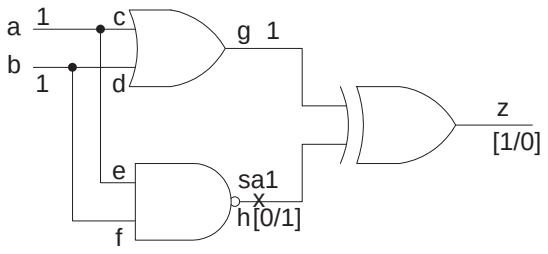

Figure 1.4: Simple Circuit Example for Illustrating Test Generation

- *Combinational ATPG*: Test generation for combinational circuits is called combinational ATPG. It is proven to be an NP-complete problem [1]. It is computationally intensive and involves a high volume of data for VLSI circuits. Heuristics are developed to make it practical in terms of runtime. For most of the heuristics, there are four major operations involving combinational ATPG, namely excitation, sensitization, justification and implication. These operations are better explained with a simple single stuck-at fault example as shown in Figure 1.4. To excite the sa1 fault at NAND gate's output h, the logic value of h is set to 0. This means at h it has an expected value of 0 and a faulty value of 1, notated as [0/1]. In order to observe the difference, the fault effect is sensitized to the output z as the observing point. Also to justify the value 0 at h, inputs a and b are justified as 1 and this implies the values of g to be 1 and the response of z is [1/0]. After the four operations a pattern a=1,b=1;z=1 is generated. By repeating these steps for all the faults in the circuit, a complete set of patterns can be generated.

- *Sequential ATPG*: Test generation for sequential circuits is called sequential ATPG. It is more complex than combinational ATPG for several reasons. The internal state elements are not being controlled directly so one or more input vectors are required to change them into certain states. Sensitization of a fault is also more difficult because it has to drive the circuit into a known state, which also requires multiple input vectors. When involving multiple patterns for a fault, the order of patterns becomes important too. Moreover, it could take multiple clock cycle for the fault effect to propagate to an observable output.

Furthermore, relations between multiple clock domains have to be accounted for [21].

- *Test Compaction*: Test compaction is the process used to reduce the size of the test set. Don't Cares are the unspecified values that can be either logic 0 or 1 during test. By filling the Don't Cares to make a pattern to detect more faults, fewer patterns are required to achieve the same coverage. There are two types of compaction, namely static and dynamic compaction. Static compaction is a post-processing step that finds the compatible patterns and combines them into a single pattern that detects multiple faults (more Don't Cares in the test mean high compatibility between patterns). On the other hand, dynamic compaction fills the Don't Cares of the partially specified patterns during test generation. *Fault simulation* is also used by ATPG and compaction algorithms to reduce the number of faults that need to be targeted during test generation, which leads to both lower pattern count and runtime. In general, dynamic compaction outperforms static compaction, though it makes the ATPG process more time consuming [1].

1.5 Scan and Test Application

Internal state signals are sequential elements which cannot be controlled by primary inputs directly. As discussed in the previous section, sequential ATPG is required to generate tests for sequential circuits. Scan is a widely used design-for-test (DFT) method employed to reduce ATPG for sequential circuits to combinational ATPG.

Scan provides controllability and observability to design state elements, i.e., flip-flops (FFs), by employing additional circuitry for the test mode. In the test mode all the FFs are configured into a shift register named scan chain. This scan chain will be accessed through the scan in (SI) and scan out (SO) ports. Desired values of the FFs can be shifted in through SI in the test mode. On the other hand, by shifting out the values from the scan chain through SO, states of FFs can be observed. Multiple scan chains can also be used to reduce the time of shifting, also called *scan time*. By the

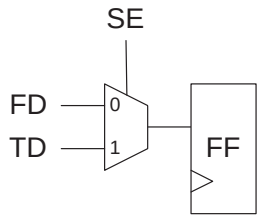

Figure 1.5: A Scan Flip-flop

employment of scan, all the FFs in the circuit become controllable and observable and they can be modeled as pseudo-inputs and pseudo-outputs. The more tractable combinational ATPG can thus be applied to the scan sequential circuits.

Scan-flip-flops (SFFs), or scan cells, are used to replace all the FFs in the circuit to construct the scan chain. A SFF, as shown in Figure 1.5, uses a 2-input multiplexer at the input of a FF to select between the data from the circuit in the functional mode and the data from the previous element in the scan chain in the test mode. Mode change is controlled by the scan enable (SE) signal, which is also used as the control signal for the multiplexer. An example of implementing a scan chain is illustrated in Figure 1.6. The logic cone has two input FFs and one output FF and they are replaced to SFFs to construct a scan chain. A common scan enable signal connects to the control ports of the multiplexers of the SFFs. A and B are the functional data (FD) ports of FF1 and FF2 respectively. The output of FF1 and FF2 connects to the test data (TD) ports of FF2 and FF3 respectively, forming a scan chain with a sequence of FF1, FF2 and FF3. To complete the scan chain, TD port of FF1 is connected to the scan in (SI) port and the output of FF3 connects to the scan out (SO) port.

Single-pattern tests, as required by stuck-at faults, can be applied to a scan circuit in a straightforward manner. The input pattern is shifted (scanned) in for n clock cycles, where n is the number of flip-flops. Then the circuit is switched from the test mode to the functional mode for one clock cycle in order to capture the output response. Then the circuit is switched back to the test mode and the output response is shifted out concurrently with shifting in the input pattern for the following test. For two-pattern tests, as required by delay faults, the use of scan poses test application

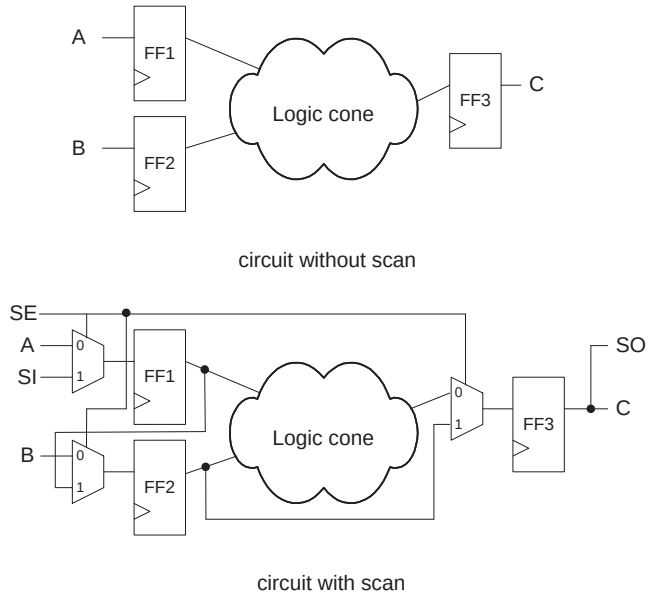

Figure 1.6: Example of Scan Design

problems. There are three strategies to apply two-pattern tests to a scan circuit.

- *Enhanced-scan*: Enhanced-scan uses two register scan cells for storing the pattern pair. Both patterns are shifted in before the test starts so they can be applied in two consecutive cycles. Its main limitations are doubling the area of state elements and the volume of test data.

- *Broadside*:

  Broadside uses the output response of the initialization pattern as stimuli for the excitation pattern. Figure 1.7 is an example of Broadside test application. A two-pattern test [X,Y] is the test to applied to the circuit under test (CUT) and a response Z is expected. The first pattern X is scanned in and applied to the CUT, and the response of X is loaded into the scan chain. The second pattern Y, which is the same as the response of X, is in place and can be reapplied to the CUT. Response Z is scanned out and observed. Y is restricted to be in the solution space of all the possible responses of the CUT so not all the input combinations can be applied as the second pattern.

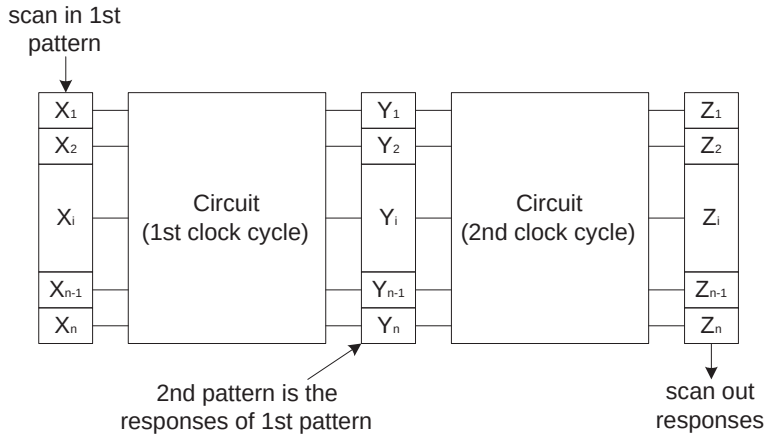

Figure 1.7: Broadside Test Application Strategy

- *Skewed-load*: With Skewed-load, the first pattern is applied and then one bit is shifted along the scan chain and the shifted pattern is used as the second pattern. Figure 1.8 illustrates an example of Skewed-load test application. Similar to the previous example, a two-pattern test [X,Y] is applied to the CUT and a response Z is expected. The first pattern X is scanned in and applied to the CUT. The first response R is discarded because the first pattern is only used for initialization. The first pattern X is still in the scan chain so by scanning in one extra bit, the values in the scan chain are shifted and become the second pattern Y consisting of $Y_1 = SI$ and $Y_i = X_{i-1}$. The second pattern Y is then applied to the CUT and response Z is obtained. Y faces a similar problem as in Broadside, that it is restricted to the shifted version of the first pattern X.

As discussed above, enhanced scan has area overhead and test data volume limitations. Broadside is used in practice despite the overhead in ATPG runtime (due to two time-frame ATPG). Skewed-load is widespread in practice despite the constraints on switching the scan enable at-speed (recent advancements in scan cell design and scan enable routing have been addressing this problem). Therefore for the remainder of this thesis we will focus on the Skewed-load test application strategy when dealing with two consecutive patterns applied through scan chains.

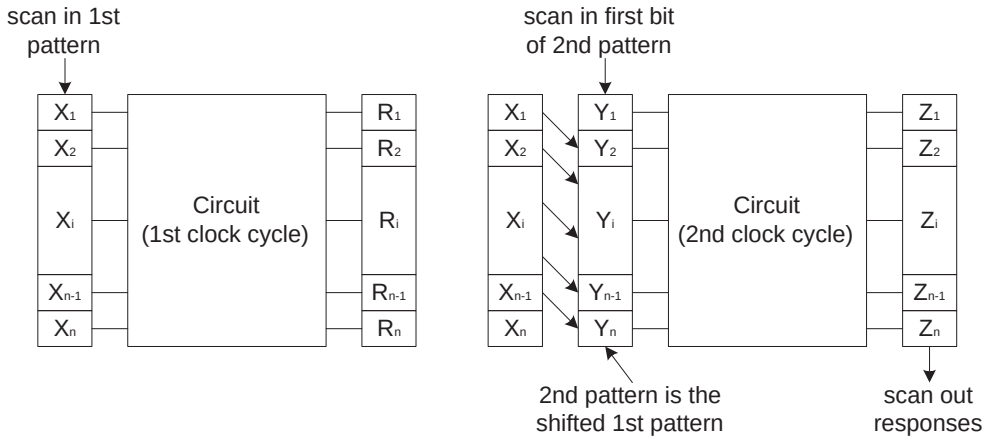

Figure 1.8: Skewed-load Test Application Strategy

1.6 Thesis Organization

This thesis presents a new way to improve the efficiency of defect screening during manufacturing test through the use of multiple observations during test generation. The basic intuition of our approach lies into leveraging the advancements in the satisfiability (SAT) problem to do constrained ATPG. More specifically, we combine the advantages of multiple-detect and detection at all observable outputs in order to generate stuck-at test patterns that can improve surrogate detection. The remainder of the thesis is organized as follows.

Chapter 2 starts with an introduction of the SAT problem and some state-of-the-art algorithms for solving it. Differences between the traditional and SAT-based ATPG techniques will be highlighted and the evolution of SAT-based ATPG will be presented. The motivation for our work will also be discussed.

In Chapter 3, a new method is proposed to increase observations of fault effects during test generation. Detailed algorithms on how to construct the test generation satisfiability problem will be presented in this chapter. Test generation for both single stuck-at fault and transition fault models will be discussed.

Experimental results will be presented in Chapter 4 to illustrate the effects on pattern counts and some metrics on test evaluation. Finally, conclusions and suggestions for further development are provided in Chapter 5.

Chapter 2

Satisfiability and ATPG

Boolean Satisfiability Problem, also known as SAT, is a decision problem that determines if a solution exists for a Boolean function provided in the conjunctive normal form (CNF) such that the function evaluates to *true*. It is a well-known NP-complete problem [10] and it is widely used in VLSI computer-aided design (CAD), artificial intelligence and formal verification. The problem is satisfiable if a satisfying assignment can be found, otherwise it is said to be unsatisfiable.

A propositional formula is represented as conjunction of clauses, where each clause is a disjunction of literals. Literals can be in positive or negative phase, denoted as x and $\overline{x}$ respectively. Consider a propositional formula $(x_1+x_2)(\overline{x_1}+\overline{x_3})(x_2+x_3+\overline{x_4})(x_4)$. A satisfying assignment $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$ would satisfy all clauses and thus the function is evaluated to *true*. Since a satisfying assignment can be found, the problem is claimed to be satisfiable.

In Section 2.1, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving SAT problems will be introduced. Some milestones of SAT advancements over the last decade will also be outlined in this section. Section 2.2 is a summary of various ATPG algorithms, including traditional structural based algorithms, early SAT-based ATPG attempts and recent SAT-based ATPG implementations utilizing state-of-the-art SAT solvers. Finally, the motivation and objectives of this research will be discussed in Section 2.3.

2.1 SAT Solving

There are two classes of SAT solvers. Probabilistic SAT solvers, based on stochastic local search algorithms, which perform well on random SAT instances. However, they are not suitable for structured instances such as formal verification and VLSI CAD problems, including ATPG [23]. Another class of solvers are based on DPLL algorithm [4, 5], which is a branching search algorithm with backtracking. This algorithm is explained next.

Algorithm 1: DPLL Algorithm [20]

```
if (!preprocessing()) then
    return(UNSAT);
end
while true do
    if (!decide()) then
        return(SAT);
    end
    while (!bcp()) do
        if (!resolveConflict()) then
            return(UNSAT);
        end
    end
end
```

At the beginning, there could be some preprocessing that discovers whether the instance is unsatisfiable. The main loop starts by the decide() operation, which chooses an unassigned variable, and assigns a value to it. If no unassigned variable is available, the instance is solved and it is thus satisfiable. If a variable is assigned, Boolean constraint propagation (BCP) is carried out in the bcp() operation. As a result of this assignment, all clauses with this variable will either be satisfied, or having one fewer unassigned literal. Some clauses will have only one unassigned literal left and thus become unit clauses. BCP identifies these unit clauses and implies values to those unassigned variables. The bcp() operation repeats itself when unassigned variables are implied within bcp(). However, if a variable is implied to be both 0 and 1, a conflict is discovered and bcp() returns false to enter resolveConflict().

resolveConflict() operation carries out backtracking, which undoes some decisions and the corresponding implications. If all decisions are restored, the instance is unsatisfiable as the entire solution space is searched.

In the last decade, there have been a few milestones that improve the efficiency of SAT solvers.

- *Conflict-driven Learning*: First introduced in GRASP [18], instead of performing chronological backtracking as in the DPLL algorithm, conflict analysis is performed to find out the reasons for a conflict. Conflict-driven learning uses the results of conflict analysis and avoids the same conflict by adding conflict clauses to the formula. With new conflict clauses added into the problem being solved, restarting the search may lead to a very different search space and approach to a conclusion faster. The restart strategy is proved experimentally and adapted in some latest SAT solvers such as Chaff and Berkmin [12].

- *Two-literal Watching*: The SATO SAT solver [34] introduced head/tail literal scheme to improve the performance of BCP. Every clause has a list of literals. A head pointer points to the first unassigned literal while a tail pointer points to the last unassigned literal in the clause. A clause becomes unit clause when the head and tail pointers point to the same unassigned literal. These pointers are only updated when assignment and backtracking occur at the literals that the pointers are pointing to. Therefore, it saves some of the overheads of updating the clause status during bcp().

  An alternative scheme is proposed in the Chaff SAT solver [20]. Instead of two pointers moving toward each other as in SATO, the two pointers Chaff uses can be any unassigned literals in the clause. A unit clause is detected when two pointers point to the same literal as SATO. Although this "lazy update" scheme has a higher overhead during bcp() while searching for another unassigned literal when updating a pointer, backtracking has virtually no overhead as no pointer update is required. Consequently, Chaff performs faster than SATO for a significant margin in most cases [20].

- *Decision Heuristic*: For difficult problems, many conflicts occur during solving and thus generating huge amount of conflict clauses. Consequently these conflict clauses dominate the problem and satisfying these conflict clauses becomes more important [20]. Chaff uses Variable State Independent Decaying Sum (VSIDS) to prioritize literals such that literals appearing in the most recent conflict clauses are more likely to be chosen; so recent conflict clauses can be satisfied faster. Berkmin modified the heuristic slightly by considering also the clauses involved in conflict analysis and their literals and thus getting a better estimate of the variable activities. Performance gain can be obtained through a "smarter" choice of variables that ultimately reduces backtracking in the search.

2.2 SAT-based ATPG

Before discussing the SAT-based ATPG algorithms we first take a look at the traditional ATPG algorithms.

2.2.1 Traditional ATPG Algorithms

D-Algorithm (D-ALG) is the first complete test generation algorithm proposed by Roth in 1966 [24]. The algorithm can be summarized as follows:

- *D and* $\overline{D}$: Besides logic value 0 and 1, two more symbols are introduced: D represents logic value 1 in the fault-free circuit and 0 in the faulty circuit (1/0) while $\overline{D}$ is the negation of D (0/1).

- *Fault Excitation*:

  When a fault is targeted, an appropriate D value is assigned to the fault site. If the fault is a sa0, a faulty value 0 and an exciting value 1 is required so D is assigned. Otherwise, $\overline{D}$ is assigned for sa1.

- *Fault Propagation*: Find a path from the fault site to the closest primary output. Assign appropriate D values along the path so that the fault effect is propagated to an observable output. The path is called a D-chain.

- *Line Justification*: With all the D values along the D-chain, backward justify all other lines and check for consistency. If there are inconsistencies, find the next shortest propagation path and justify the new D-chain again. If no consistency can be found among all paths, the fault is untestable.

D-ALG is the basis of most ATPG algorithms, including some milestones in ATPG such as PODEM (Path Oriented DEcision Making) [11], FAN (Fan-out-oriented test generation algorithm) [9] and SOCRATES (Structure-Oriented Cost Reducing Automatic TESt pattern generation system) [25].

- *PODEM*: During line justification in D-ALG, branching can occur at any line that has multiple possible justifications, which makes it exponentially complex to the number of lines. PODEM instead performs branching only at primary inputs. This reduces the worst-case complexity to only exponential to the number of primary inputs and hence it significantly improves ATPG runtime performance.

- *FAN*: FAN introduced four major contributions over PODEM. Immediate implications and unique sensitization result in signal assignments. The use of headlines pushes the branching points forward from primary inputs to the first level of fan-out lines. These three improvements reduce the depth of backtracing and avoid unnecessary searches. FAN also performs multiple backtrace in breadth-first fashion instead of inefficient depth-first single backtrace in PODEM.

- *SOCRATES*: Beside using fast random pattern generation with parallel fault simulation, and also some improvements over the FAN algorithm, SOCRATES also introduced dynamic learning. It performs a learning process between decision steps to find out more implications over multiple logic levels to further improve search speed.

Y = BUF(A)	$(Y + \overline{A})(\overline{Y} + A)$
Y = NOT(A)	$(Y + A)(\overline{Y} + \overline{A})$
Y = AND2(A,B)	$(\overline{Y} + A)(\overline{Y} + B)(Y + \overline{A} + \overline{B})$
Y = NAND2(A,B)	$(Y + A)(Y + B)(\overline{Y} + \overline{A} + \overline{B})$
Y = OR2(A,B)	$(Y + \overline{A})(Y + \overline{B})(\overline{Y} + A + B)$
Y = NOR2(A,B)	$(\overline{Y} + \overline{A})(\overline{Y} + \overline{B})(Y + A + B)$
Y = XOR(A,B)	$(Y + \overline{A} + B)(Y + A + \overline{B})(\overline{Y} + A + B)(\overline{Y} + \overline{A} + \overline{B})$
Y = XNOR(A,B)	$(\overline{Y} + \overline{A} + B)(\overline{Y} + A + \overline{B})(Y + A + B)(Y + \overline{A} + \overline{B})$

Table 2.1: The Formulae for the Basic Gates

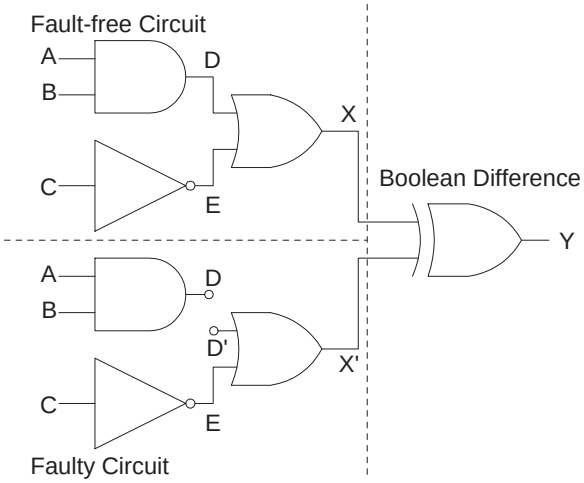

Figure 2.1: Fault-free and Faulty Circuit with Boolean Difference [16]

2.2.2 Early SAT-based ATPG

In 1989, Larrabee presented an algorithm that uses the Boolean difference for test generation [16]. The algorithm first translates the test generation problem into a SAT problem, and then tries to solve the SAT problem in order to generate a pattern.

To represent an AND gate in CNF, the translation starts with the Boolean function $D = A \cdot B$ where D is the output of the AND gate with inputs A and B. Due to the fact that the equality $P = Q$ is equivalent to two implications $(P \rightarrow Q) \cdot (Q \rightarrow P)$, the Boolean function can be translated into $(D \rightarrow (A \cdot B)) \cdot ((A \cdot B) \rightarrow D)$. Also, because of $P \rightarrow Q$ is identical to $\overline{P} + Q$, the formula becomes $(\overline{D} + A) \cdot (\overline{D} + B) \cdot (\overline{A} + \overline{B} + D)$. Similarly, the formulas for other basic gates can also be determined as shown in Table 2.1.

The test generation involves clauses generation for the fault-free circuit and the faulty circuit, as shown in Figure 2.1. The upper and lower part of the figure represents the fault-free and faulty circuit correspondingly, and the XOR at the end is the Boolean difference of the fault-free and faulty circuits. The Boolean difference mimics the D-value requirement at an observing output. The fault-free circuit can be represented in CNF by the conjunction of all formulae of individual gates in the circuit. As in the example, the formula for output X is

$(X + \overline{D}) \cdot (X + \overline{E}) \cdot (\overline{X} + D + E)$
$\cdot (\overline{D} + A) \cdot (\overline{D} + B) \cdot (\overline{A} + \overline{B} + D)$
$\cdot (C + E) \cdot (\overline{C} + \overline{E})$

On the other hand, the faulty circuit is a copy of the fault-free circuit, except that the lines affected by the fault are renamed. In the case of sa1 at line D, the lines D and X in the faulty circuit are renamed as D' and X'. Two single literal clauses $(\overline{D})$ and (D') are added to assign the exciting and faulty values to line D. Therefore, the formula for output X' is

(D')
$\cdot (\overline{D} + A) \cdot (\overline{D} + B) \cdot (\overline{A} + \overline{B} + D)$
$\cdot (C + E) \cdot (\overline{C} + \overline{E})$
$\cdot (X' + \overline{D}') \cdot (X' + \overline{E}) \cdot (\overline{X}' + D' + E)$

$(\overline{D})$ is implied because of the discontinuity at line D. By adding the formula for the final Boolean difference XOR gate

$(\overline{X} + X' + Y) \cdot (X + \overline{X}' + Y) \cdot (\overline{X} + \overline{X}' + \overline{Y}) \cdot (X + X' + \overline{Y})$,

the complete CNF representation for Figure 2.1 is

$(X + \overline{D}) \cdot (X + \overline{E}) \cdot (\overline{X} + D + E)$
$\cdot (\overline{D} + A) \cdot (\overline{D} + B) \cdot (\overline{A} + \overline{B} + D)$
$\cdot (C + E) \cdot (\overline{C} + \overline{E})$
$\cdot (D')$
$\cdot (X' + \overline{D}') \cdot (X' + \overline{E}) \cdot (\overline{X}' + D' + E)$
$\cdot (\overline{X} + X' + Y) \cdot (X + \overline{X}' + Y) \cdot (\overline{X} + \overline{X}' + \overline{Y}) \cdot (X + X' + \overline{Y})$

The SAT problem is then solved with its own SAT solver which inherits some properties of FAN and SOCRATES that speed up the search for an assignment.

X	Encoding		Interpretation
	c_x	c_x^*	
0	0	1	signal x is 0
1	1	0	signal x is 1
U	0	0	signal x is unknown
?	1	1	conflict at signal x

Table 2.2: 3-valued Encoding with Unknown [30]

The TEGUS (TEst Generation Using Satisfiability) algorithm [29] proposed by Stephan et al. has a few improvements over [16]. As a preprocessing step, TEGUS first converts the circuit into a network of AND gates with inverted inputs, as a unique gate function makes clause generation and fault simulation more efficient. Also, TEGUS uses a way similar to D-ALG to generate clauses, which yields fewer clauses and literals compared to [16]. This leads to a smaller SAT problem to be solved. While solving the SAT problem, a simple greedy variable selection strategy is used. A literal of the first found clause with more than two free literal is chosen for branching. Moreover, the clauses for the gates are added into the SAT problem in a depth-first fashion, starting from the primary inputs of the circuit. With the greedy variable selection, variables of input nodes are selected for branching, which mimics PODEM. Also clauses of nearby logics are grouped together, thus resulting in faster conflict detection.

2.2.3 SAT-Based ATPG with DPLL-based SAT Solvers

With the new generation of DPLL-based SAT solvers such as Chaff [20] and MiniSAT [7] which can be used as a function library, highly-efficient SAT solving functionalities can be easily integrated into any application. Since solving a SAT problem becomes a simple function call to the SAT solver library, subsequent SAT-based ATPG algorithms focus more on constructing the SAT problem than on SAT solving.

PASSAT (PAttern Search using SAT) [27] is the first algorithm which takes both unknown values and tri-states into consideration. Similar to the 3-valued encoding proposed by Tafertshofer et al. [30] for single stuck-at faults, as shown in Table 2.2, PASSAT uses two Boolean variables to represent its 4-valued encoding, as shown in

X	Encoding		Interpretation
	c_x	c_x^*	
0	0	1	signal x is 0
1	1	0	signal x is 1
U	1	1	signal x is unknown
Z	0	0	signal x is at high impedance

Table 2.3: 4-valued Encoding Optimized for AND Network

AND	0	1	Z	U
0	0	0	0	0
1	0	1	U	U
Z	0	U	U	U
U	0	U	U	U

Table 2.4: Truth Table of the 4-valued AND Gate [27]

Table 2.3 [27]. The authors of PASSAT also demonstrated that the use of different 4-valued encodings generates different number of clauses for various Boolean gates and bus components [8]. With different component composition in a design, a particular encoding can be selected to minimize the number of clauses that reduce the size of the SAT problem. The particular encoding in Table 2.3 is AND-network optimized as it gives the fewest clauses for AND gates among all 24 possible encodings. The truth table and clauses for the 4-valued AND gate are shown in Table 2.4 and 2.5 respectively.

Besides the 4-valued encoding, PASSAT is also capable of using different variable selection strategies. Advanced SAT solvers allow users to constrain which variables are branchable during decision-making [20]. Without any constraint, PASSAT uses Chaff directly with the VSIDS strategy [20] to allow branching on all variables. PASSAT is also able to constrain variable selection only at input variables or fanout nodes, which mimics PODEM and FAN. Experimental results show that combining input variable constraints with a second attempt to the abort faults without constraints is more efficient and robust than using only any single strategy.

TARO was also implemented within a SAT framework, as presented in [32]. By constructing two sets of variables representing two clock cycles of the circuit, a two-pattern test can be generated for transition faults. With different constraints connecting input and output variables of both time frames, this method can be applied

$$(\overline{c}_a + \overline{c}_b + c_c) \cdot (c_a^* + c_b^* + \overline{c}_c^*) \cdot (c_c + \overline{c}_c^*) \cdot (c_a + c_a^* + \overline{c}_c) \cdot$$
$$(c_b + c_b^* + \overline{c}_c) \cdot (\overline{c}_a^* + \overline{c}_b + c_c^*) \cdot (\overline{c}_a + \overline{c}_b^* + c_c^*) \cdot (\overline{c}_a^* + \overline{c}_b^* + c_c^*)$$

Table 2.5: Clauses for the 4-valued AND Gate

to both broadside and skewed-load application strategies. Since a pattern can hardly observe a fault at all reachable outputs, [32] proposed two approaches to find out a subset of outputs that can observe a fault with a pattern.

- *Conflicting Outputs Removal Approach*: UnSAT Core is a subset of clauses in the original SAT problem that cause the conflict during unsatisfiability. Using the information in the UnSAT Core, some output constraints can be removed so that these outputs are not required to observe the fault along the others with the same pattern. Beginning with all the reachable outputs in the SAT problem, output constraints are removed as the SAT solver returns unsatisfiable and an UnSAT Core. This process iterates until a satisfiable result is returned and a pattern is generated with the fault observed at some outputs.

- *Incremental Satisfiability-based Approach*: The exact opposite of conflicting outputs removal, output constraints are added incrementally to the SAT problem for solving. Beginning with only one output, if the problem is satisfiable, more outputs are added over iterations until it becomes unsatisfiable. All outputs except the one that triggers the conflict are the observation outputs of the targeting fault for a pattern. Outputs more distant to the fault site in term of logic levels are added first so the pattern generated will sensitize the fault through longer paths.

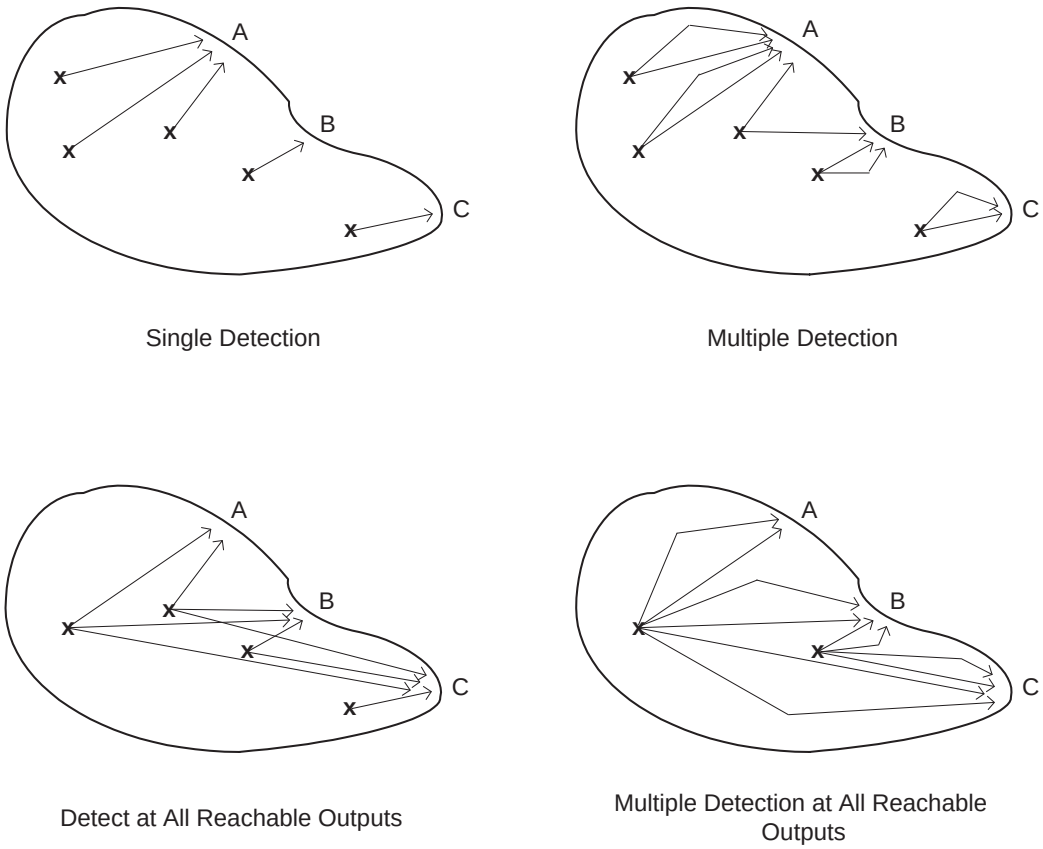

Figure 2.2: Observation Strategies

2.3 Motivation And Objectives

With reducing runtime as the main objective, traditional ATPG algorithms only require fault effects to be propagated to the shortest reachable output and only one observation is required for each stuck-at fault. In this respect stuck-at test sets have been shown to be insufficient to achieve a low defects-per-million (DPM) level [17, 31]. Building ATPG tools based on more complex fault models, such as the bridging fault model, is infeasible due to the huge fault space. Consequently the question is how can the existing stuck-at and delay-fault ATPG tools be extended in such way that surrogate detection is improved.

The importance of propagation diversity has not been considered until the introduction of multiple detection and TARO. As shown in Figure 2.2, traditional ATPG

suffers from an unbalanced observation, where most of the observations occur at only a few outputs. This is because propagating the fault effects to these easily reached outputs reduces the ATPG runtime. Multiple detection first addressed the significance of having a fault being sensitized along multiple paths. On top of multiple detection, TARO requires all faults to be observed at all reachable outputs to further increase the diversity of sensitization. In this thesis we want to take the benefits of multiple detection and TARO and combine them into multiple observations on all the reachable outputs of all faults.

Imposing additional constraints on the ATPG search leads us to build our ATPG work flow on a SAT-based approach. Building this ATPG work flow brings its own technical challenges and solutions described in Chapter 3. The contributions lie in putting this ATPG work flow together and assessing the effectiveness of multiple observations on different coverage metrics for models that are not targeted explicitly by ATPG. Our results are presented in Chapter 4.

Chapter 3

SAT-Based ATPG Based on Multiple Observations

In this chapter the detailed formulation of multiple observations ATPG using SAT will be presented. An overview of the ATPG work flow will be outlined in Section 3.1. A 3-value encoding system will be explained in Section 3.2. In Section 3.3, an in-depth illustration of the major steps in the ATPG work flow for generating single stuck-at fault tests will be presented. Since test generation is not a trivial decision problem, some modifications are made to a standard SAT solver to impose extra preferences during the search process. These modifications will be discussed in Section 3.4, followed by extending the SAT formulation from single stuck-at fault ATPG to skewed-load transition fault ATPG in Section 3.5. Finally in Section 3.6, a pattern post-processing step which reduces the number of fault simulations will be described.

3.1 Test Generation Work Flow

Traditional ATPG has a general work flow as shown in Figure 3.1. It starts with reading the netlist of the circuit under test (CUT). With the structural information obtained from the netlist, a fault list is generated corresponding to the fault model used for test generation. The objective of traditional ATPG is to detect all faults

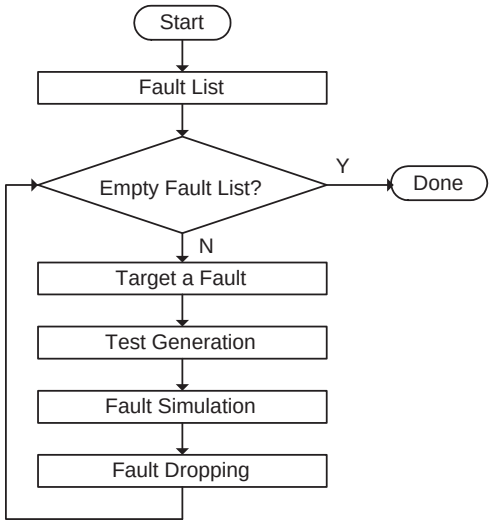

Figure 3.1: Traditional ATPG Work Flow

in the fault list with at least one pattern. Following the fault list generation, at the beginning of the main loop of ATPG, one fault is selected from the fault list at a time. During test generation, a pattern is generated to detect the targeted fault. Fault simulation is then carried out on the pattern for all other faults, to find out what other untargeted faults can be detected with this pattern. Finally, the targeted fault and untargeted faults found during fault simulation are dropped from the fault list. The main loop repeats itself with a smaller untargeted fault list over every iteration. When all faults are detected by at least one pattern, the fault list becomes empty and the whole ATPG process is completed. All the patterns generated become the test set for testing the CUT.

This approach is particularly efficient for single stuck-at fault model, where bit-level parallelism can be utilized. Since one bit is used per signal, multiple patterns or multiple stuck-at faults can be applied to different bits in a machine word so parallel fault simulation can be carried out at the same time [1]. However, because of the limitation of the representation of Boolean variables in SAT solvers and also due to the decision nature of SAT solving, bit-level parallelism can not be achieved in SAT solvers. Moreover, multiple fault simulations are separate decision problems in terms of SAT, so they cannot be formulated into a single SAT instance. As explained later

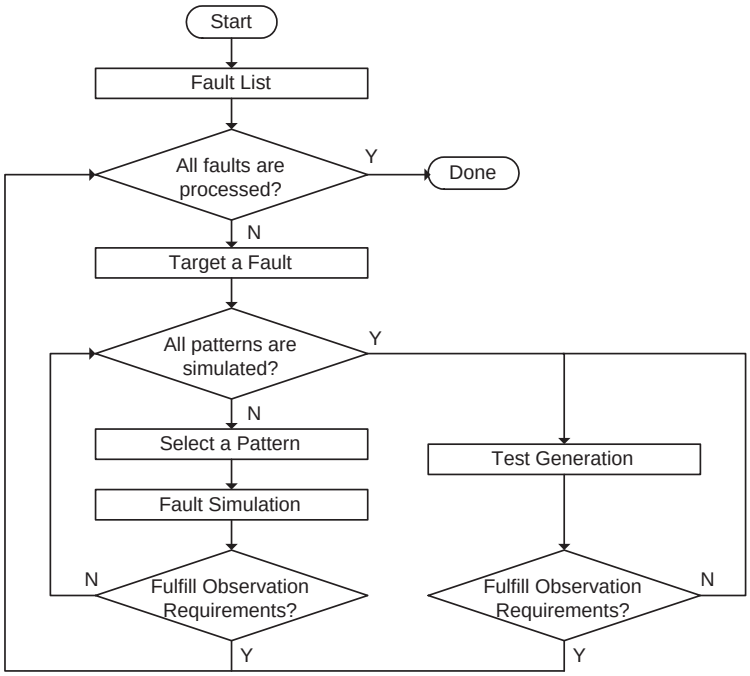

Figure 3.2: Proposed ATPG Work Flow

in this chapter, we employ SAT for fault simulation in order to perform also dynamic compaction capable of accounting for the multiple observations of each fault (which is unique to our work).

Figure 3.2 illustrates the work flow of the proposed ATPG system. The differences in the work flow reflect the adaptation of SAT-based ATPG, where both fault simulation and test generation are performed using SAT. Both single stuck-at fault and transition fault test generation follow the same work flow, although the ways to construct the SAT problems are slightly different for the two fault models.

- *Fault List*:

  Fault list for the transition fault model is identical to the stuck-at fault model. They both have a stuck-at value at fault sites located at all the inputs and at all the outputs of all gates. The stuck-at fault list however can be collapsed more efficiently than the transition fault list because of the difference in equivalent faults; hence more faults are in the collapsed transition fault list than in the

collapsed stuck-at fault list. The collapsed fault list is then sorted using SCOAP [13], a measure that estimates the controllabilities and observabilities of each logic element in the circuit. Faults at gates with lowest observabilites will go through test generation first because they are expected to be more difficult to observe. Patterns targeting difficult-to-observe faults have a better chance to detect also faults at locations with better observabilities; the contrary does not hold. As a result of using the SCOAP measures, fewer patterns are processed and a smaller test can be generated.

- *Target a Fault*:

  The outer loop iterates through all the faults in the sorted fault list. When a fault is targeted, the ATPG creates a foundation SAT problem that is specific to the targeted fault. This process is relatively long as it goes through all the structural information of the CUT. This foundation SAT problem is used in both fault simulation and test generation processes with additional constraints in the inner loops. It consists of characteristic equations, Boolean differences and fault excitation, and will be discussed in Section 3.3.1.

- *Fault Simulation*:

  A fault-output (f-o) pair is one reachable output of its corresponding fault [22]. Observation counts for all individual f-o pairs are needed to control the sensitization to favor f-o pairs which have lower counts. A fault simulator in a traditional ATPG flow does not track observation counts for individual f-o pairs. Therefore, a novel fault simulation algorithm in SAT is proposed with the capability of maintaining all the observation counts and also achieving dynamic compaction.

  Fault simulation in SAT is different to traditional ATPG. While simulating a pattern once in traditional ATPG can find out all the faults it can detect, every SAT problem can only target a single fault. Since creating the SAT problem for a targeted fault is relatively slow, it is better to iterate through all the generated patterns in the inner loop instead of iterating through all the faults.

Fault simulation starts with the foundation SAT problem constructed in the outer loop. To apply a pattern at the inputs, input assignments, discussed in Section 3.3.2 are provided to the SAT problem. Also at least one of the reachable outputs are constraints to observe the fault effect. The three constraints above are sufficient to perform basic fault simulation for a pattern for the targeted fault. After solving, the SAT solver returns satisfiable if the pattern can detect the targeted fault at some reachable outputs. On the contrary, it returns unsatisfiable if the pattern cannot detect the targeted fault.

In addition, output constraints and slightly modified input assignments, discussed also in Section 3.3.2 can be added to perform Don't Cares filling, which results in dynamic compaction. If the SAT problem is satisfiable, the values in the input variables of the SAT instance represent the pattern that detects the targeted fault. The returned pattern may be different from the applied pattern due to Don't Cares filling, which deals with assigning unspecified bits. On the other hand, if the SAT problem is unsatisfiable, the applied pattern cannot detect the targeted fault, even with any combination of Don't Care filling. If all the reachable outputs of the fault meet the observation requirements, the targeted fault can be dropped from the fault list and the next fault in the fault list will be processed in the outer loop. If the observation requirements are not met after simulating all the generated patterns, the test generation process is in place to generate new patterns to satisfy the requirements.

- *Test Generation*:

  Test generation is in fact very similar to fault simulation, as it uses the same foundation SAT problem from the outer loop and also output constraints for directing propagation paths. Though, input constraints are different from the input assignments used in fault simulation. Input constraints, discussed in Section 3.3.2, restrict the input variables to be the same as the combination of previous patterns and avoid generating repeated patterns. Since observation counts at each f-o pair change, in each iteration a new SAT problem is constructed with different output constraints and a pattern is generated if the

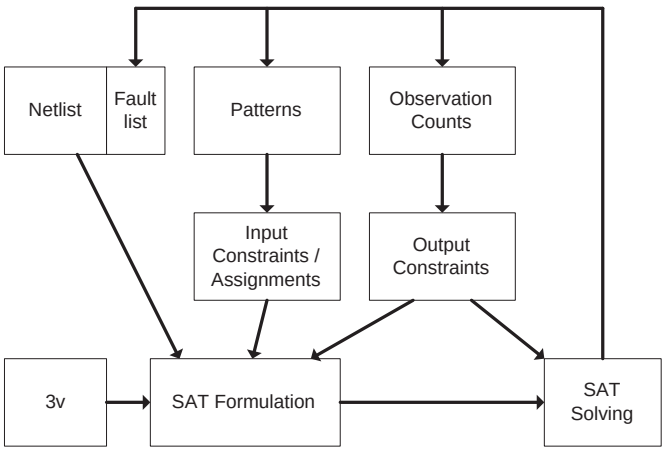

Figure 3.3: Inputs for SAT Formulation

problem is satisfiable. The test generation inner loop iterates until all the reachable outputs meet the observation requirements. The inner loop also ends when an unsatisfiable problem is encountered, when no more patterns can be generated under the current output constraints. The targeted fault is then dropped and the algorithm moves to the next fault in the fault list.

3.2 Encoding and Clause Generation

There are three major components of formulating the ATPG SAT problem as illustrated in Figure 3.3. Firstly, netlist provides the structural information of the CUT so that logic implications between elements can be included in the problem. Boolean differences and fault excitations are also included into the problem with the netlist. Secondly, input assignments and constraints are added to apply patterns at inputs during simulation and avoid duplicating patterns to be generated during test generation. Lastly, output constraints are added into the SAT problem to direct the SAT solver to search for solutions that observe the targeted fault at different outputs. Details of the SAT formulation will be discussed in Section 3.3. In this section, the 3-valued (3v) encoding system used for the formulation will be presented.

A	Encoding		Interpretation
	a_1	a_0	
0	0	1	signal A is 0
1	1	0	signal A is 1
U	0	0	signal A is unknown
	1	1	conflict at signal A

Table 3.1: 3-valued Encoding with Unknown (New Notation)[30]

3.2.1 3v Encoding

In order to support unspecified inputs and Don't Cares, a 3v encoding system is required for ATPG which is capable of representing logic 0, logic 1 and Unknown. Because the SAT problem can only be formulated in Boolean variables, two Boolean variables are needed to represent a 3v variable. Among all the possible 3v encoding schemes, the one proposed in [30] is used in this SAT formulation for two reasons.

- *Lazy Assignments*:

  For clarification purposes, Table 3.1 shows the same encoding table as in Table 2.2 with some new notations. Signal X is replaced with A to avoid confusion with Don't Cares, while c_x and c_x^* are renamed to a_1 and a_0 respectively because of the 3v value signal A represents when one of them is 1. To assign a logic 0 to a 3v variable A, only one Boolean assignment $a_0 = 1$ is needed and a_1 is implied to 0 because $a_0 = a_1 = 1$ is constrained. The same goes to logic 1 assignment to A as only $a_1 = 1$ is required.

- *Lazy Detection*:

  Similarly, through the check of only Boolean variable a_1, signal A can be determined as logic 1 if $a_1 = 1$, otherwise it is either logic 0 or unknown. This is particularly useful for checking the Boolean differences for the observing outputs.

Each type of standard cell has its own logic implications. In terms of SAT, logic implications can be formulated into characteristic equations, or group of clauses. Under 3v, the implications of logic gates are different from 2v because unspecified inputs

AND	0	1	U
0	0	0	0
1	0	1	U
U	0	U	U

Table 3.2: Truth Table of the 3v AND Gate

a	b	y	a_0	a_1	b_0	b_1	y_0	y_1
0	0	0	1	0	1	0	1	0
0	1	0	1	0	0	1	1	0
0	U	0	1	0	0	0	1	0
1	0	0	0	1	1	0	1	0
1	1	1	0	1	0	1	0	1
1	U	U	0	1	0	0	0	0
U	0	0	0	0	1	0	1	0
U	1	U	0	0	0	1	0	0
U	U	U	0	0	0	0	0	0

Table 3.3: Encoded Truth Table of the 3v AND Gate

and outputs are also considered. For example, a 2-input AND gate has nine entries in the truth table as there are three values at both inputs, as shown in Table 3.2. Table 3.3 shows the 3v-Boolean-encoded truth table and the corresponding characteristics equations are illustrated in Table 3.4. The three clauses $(\overline{y_0}+\overline{y_1})(\overline{a_0}+\overline{a_1})(\overline{b_0}+\overline{b_1})$ are implicit because they are constraining the 3v variable not to be the unused value.

Finding the characteristic equations for the 2-input AND gate in 3v is not as obvious as in 2v. It is even more difficult to find a set of optimized clauses for more complex standard cells. Inspired by PASSAT[27], logic minimizer *espresso*[26] is used to convert the truth table of any standard cell into characteristics equations.

3.2.2 Clause Generation with espresso

espresso is part of the UC Berkeley's *sis* suite developed primarily for logic synthesis [26]. Provided the minterms of a Boolean function, it can obtain a minimum representation of the function. In the case of clause generation, *espresso* takes the minterms from the truth table of a certain function and generates an optimized function in product of sums (POS) form. For example, to generate the clauses for a two-input AND gate in 2v, a function $f = (y = ab)$ is set up and its truth table is illustrated in Table 3.5. The minterms are taken by *espresso* and the minimum representation generated

Characteristics Equations	$(y_0 + \overline{a_0})(\overline{y_1} + a_1)(y_0 + \overline{b_0})(\overline{y_1} + b_1)(\overline{y_0} + a_0 + b_0)(y_1 + \overline{a_1} + \overline{b_1})$
Implicit Constraints	$(\overline{y_0} + \overline{y_1})(\overline{a_0} + \overline{a_1})(\overline{b_0} + \overline{b_1})$

Table 3.4: Characteristics Equations of the 3v AND Gate

a	b	y	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 3.5: Truth Table of f = (y = ab)

is shown in Table 3.6. By interpreting 0 as the positive phase (y), 1 as the negative phase ($\overline{y}$), and the dash as not in the sum, it yields $f = (a + \overline{y})(b + \overline{y})(\overline{a} + \overline{b} + y)$.

Similarly for 3v, a function $f(a_0, a_1, b_0, b_1, y_0, y_1)$ can be set up and the input to *espresso* is exactly the same as for the right hand side of Table 3.3 with f = 1 for all entries. All the other product terms with f = 0 are assumed by *espresso* if they are not specified. The interpreted results are shown in Table 3.4, except only two of the three implicit constraints are actually needed for a minimum representation. Through this truth table to clauses conversion with *espresso*, characteristics equations of any standard cell can be generated prior to test generation.

3.3 Test Generation SAT Formulation

As discussed in Section 3.1, there are three major components for constructing the test generation SAT problem. These constraints can be divided into two categories depending on how they are applied to construct SAT problems. In each inner loop of either fault simulation or test generation, clauses that describe the logic gates in the circuit, Boolean differences and fault excitation do not change once a fault is targeted. They comprise the foundation SAT problem for a fault and are defined as static constraints. On the other hand, information about generated patterns and observation counts is updated after every iteration of either test generation or fault simulation.

a	b	y	f
0	-	1	1
-	0	1	1
1	1	0	1

Table 3.6: POS Generated by *espresso*

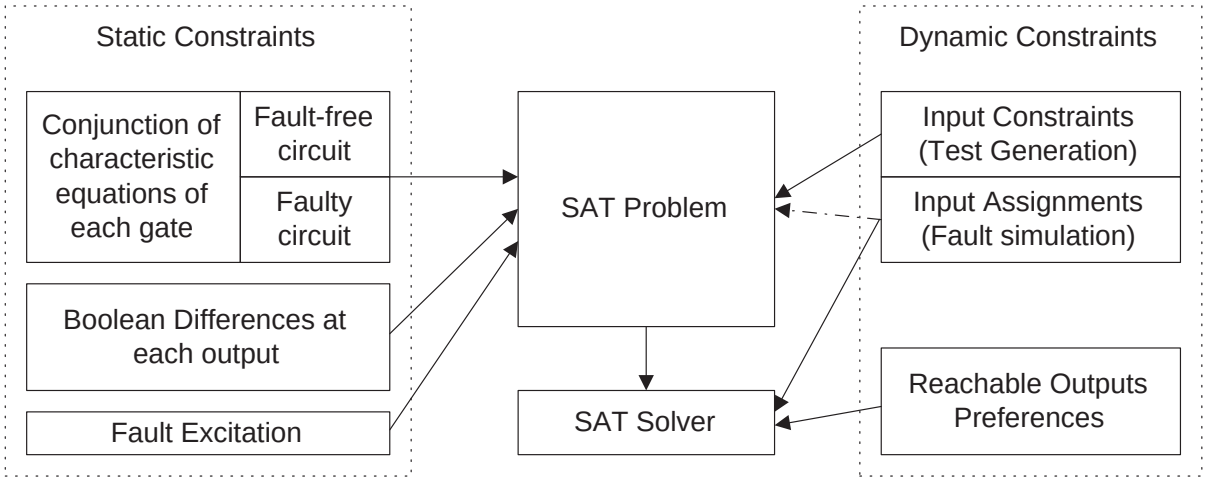

Figure 3.4: Generalization of Clauses for Test Generation

Clauses that are different in every iteration are defined as dynamic constraints. Figure 3.4 generalizes the two types of clauses that formulate the SAT problems. Details of the two types of clauses are explained for the single stuck-at fault test generation.

3.3.1 Static Constraints

- *Characteristic Equations*:

  Conjunction of characteristic equations provides all implications between logic gates in the circuit. As described in Section 2.2.2, the foundation SAT problem can be constructed through the conjunction of characteristic equations of the output cone of the fault site in the faulty circuit, and the input cones of all the reachable outputs in the fault-free circuit, as illustrated in Figure 3.5. The absence of unrelated logics out of the involved cones reduces the number of clauses to put into the SAT problem and minimizes the size of the problem to be solved. 3v encoding is used so unspecified inputs can be represented in

the foundation SAT problem and Don't Care inputs can also be generated in a pattern.

- *Boolean Differences*:

  Boolean differences are the XORs between an reachable output of a fault in the faulty circuit and the fault-free circuit. Since the list of reachable outputs for a fault does not change, these XORs remain the same throughout the fault simulation and test generation processes of the targeted fault.

  Since 3v encoding is used, there are three possible situations that occur at the Boolean differences. Logic 1 at the Boolean difference represents fault effect can be observed at the output the Boolean difference located, while logic 0 implies it is not the observable output. An unknown value at the Boolean difference means the observation status of the output cannot be determined due to unspecified inputs. It is assumed that the fault effect cannot be observed at this output with the current inputs, though it may be observable if some of the values assigned to the inputs are changed.

- *Fault Excitation*:

  The faulty value, or the stuck-at value, at the fault site of a fault has to be specified in the faulty circuit in order to excite a fault. At the same time, the complement of the faulty value could also be specified at the same site in the fault-free circuit. Though it is implied because at least one of the Boolean difference has to be logic 1 and results in different values between the fault site in both circuits, it is always faster for the SAT solver to solve with more known assignments.

3.3.2 Dynamic Constraints

- *Input Assignments*:

  Input assignments are used in the fault simulation process for applying patterns to input variables. SAT solver by nature is a propagation engine so it can be

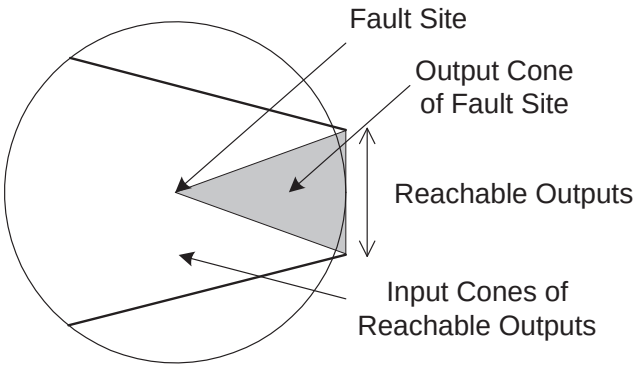

Figure 3.5: Cones Involved in SAT Problem Construction

used for fault simulation. To apply a pattern to the inputs of the circuit, single literal clauses can be used to assign values to the input variables. For example, to apply a pattern [1,0,X,X] for a logic segment with inputs A, B, C and D in 3v, two single literal clauses (A_1) and (B_0) can be added to SAT problem to assign A = 1 and B = 0 to the 3v variables for inputs A and B. For C and D, Don't Cares can be assigned as unknown values to 3v variables C and D so clauses $(\overline{C_0})(\overline{C_1})(\overline{D_0})(\overline{D_1})$ are added. In this case, a straightforward fault simulation is performed because the SAT solver cannot assign a logic value to C and D. However, if the Don't Cares are not assigned, these input variables are free and thus branchable and could be assigned by the SAT solver during solving. If the Care Bits of the inputs cannot detect the targeting fault, however by specifying some Don't Cares the fault can be detected, the SAT solver would assign values to those free input variables. Dynamic compaction is thus achieved through the filling of these Don't Cares for detecting multiple faults.

After applying a pattern to the SAT problem in an iteration of fault simulation, the added clauses have to be removed before applying another pattern. It involves a tedious clause removal process. An improvement had been made to the SAT solver to tackle this problem and will be described in Section 3.4.1.

- *Input Constraints*:

  In the test generation process, in order to avoid generating repeated patterns,

	One observation per output	A	B	C	D	E
	Start	0	0	0	0	0
1	(A + B + C + D + E)					
	A and C are observing	1	0	1	0	0
2	(B + D + E)					
	D is observing	1	0*	1	1	0
3	(E)					
	E and A are observing	2	0*	1	1	1
Observation requirements fulfilled						

Table 3.7: Example of Using Output Constraints

input constraints constructed with existing patterns are added to the SAT problem. Different from the input assignments that designate certain values to input variables, input constraints prevent certain combinations of values to occur at input variables. For example, to constrain the SAT solver not to generate two existing patterns [1,0,X,X] and [0,X,1,1] for inputs A, B, C and D in 3v in the subsequent test generation iterations, two clauses $(\overline{A_1} + \overline{B_0})(\overline{A_0} + \overline{C_1} + \overline{D_1})$ are added. $(\overline{A_1} + \overline{B_0})$ disallows A = 1 and B = 0 at the same time (which dissatisfies the clause) and similarly $(\overline{A_1} + \overline{B_0})(\overline{A_0} + \overline{C_1} + \overline{D_1})$ disallows the input combination of the second pattern. With these two clauses, the two patterns will not be generated again in the subsequent iterations of test generation. Also, as a new clause is added after each pattern is generated, no clause is removed in the test generation process.

- *Output Constraints*:

  Test generation of a fault is completed when observation requirements are met. The ATPG attempts to have all observation counts on all f-o pairs to reach as least a specified number of times, although there may be fewer possible distinctive ways for a fault to be observed at some outputs. Table 3.7 is an example of how output constraints are used to control sensitization of a targeted fault when the observation requirement is one per f-o pair.

  At the beginning of the test generation process, because of no observations at any output, there is no preference on which output to observe the fault effect. An output preference list (A,B,C,D,E) is added simply because of the sequence

of the outputs. In the first iteration, an output constraint $(A+B+C+D+E)$ is added to ensure at least one output would become observable output. If a pattern is generated and it observes the fault at outputs A and C, observation counts of A and C are incremented. Since A and C fulfilled the observation requirements, in the next iteration of test generation, they are dropped from the preference list and in the output constraint.

In the second iteration, preference list is updated to (B,D,E) and a new output constraint $(B + D + E)$ is added. The effect of the previous constraint (A + B + C + D + E) is dominated by the tighter new constraint so its removal is not required. A second pattern is generated and only output D is observing the fault; hence observation count of D is incremented. This also indicates that output B is not capable of observing the fault. A flag is set for output B to indicate it is unobservable output under current constraints so no further attempt will be made to sensitize the fault at output B.

In the third iteration, B and D are dropped from the preference list and the output constraint. E becomes the only output in the preference list and in the output constraint. A third pattern is generated and it observes at both outputs E and A. The observation count of E is now one and A is updated to two. Since all observation counts are at least one or the unobservable flag is set, all the outputs fulfill observation requirements and test generation is completed for this fault. Preference list is not a standard SAT solver feature and Section 3.4.2 describes the change required to make for the SAT solver to use the preference list.

A relaxed output constraint is also used when there is a time constraint violation during SAT solving. When it takes too much time to attempt to sensitize the fault at all outputs specified in the preference list and the SAT solver returns a time out error, a reduced preference list which consists of only the first output on the list is reattempted. The pattern generated may only be sensitized at the only output on the list, but it avoids abortion of a fault at a early stage for generating patterns for a fault.

3.4 SAT Decision Customizations

As described in the previous section, new features are required for the SAT solver to solve the ATPG problem. Unbacktrackable decisions enhance the application of multiple patterns in the SAT problem for fault simulation and test compaction. On the other hand, prioritized decision-making variables enable to select the observation outputs in a flexible way. Both of them are essential to make the proposed SAT-based ATPG to work correctly and efficiently.

3.4.1 Unbacktrackable Decisions

Unbacktrackable decision is an alternative of assigning a value to a Boolean variable, beside using a single literal clause (SLC). When a SLC is added, it creates new implications. These implications affect the conflict clauses generation during conflict analysis when solving. The conflict clauses are thus dependent to the added clause. When the added SLC is removed from the problem, the dependent conflict clauses have to be removed as well. Finding the dependent conflict clauses is time consuming, as all the conflict clauses are evaluated for each SLC removal.

The fault simulation process described in the previous section involves frequent addition and removal of SLCs for input variables for applying patterns. When a pattern is applied, multiple SLCs are added and thus results in conflict clauses to be generated during solving. After the pattern is evaluated, the SLCs are removed and the dependent conflict clauses are also removed, without being used even once. In addition to a very slow conflict clause removal process, conflict clauses specific to the current pattern are removed, thus the speedup gained through conflict analysis is diminished.

To avoid conflict clauses removal, the use of unbacktrackable decisions is proposed. Before actual decisions are made during solving, unbacktrackable decisions are added into the decision stack of the solver to imply the assigned values, as shown in Figure 3.6. Since these decisions are supposed to be assignments, they should not be backtracked or the SAT problem is invalidated. When no actual decision remains in the decision stack and a unbacktrackable decision is backtracked, the problem can be

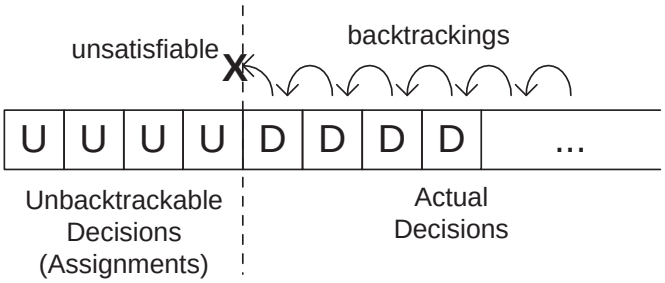

Figure 3.6: Decision Stack with Unbacktrackable Decisions

concluded as unsatisfiable.

Since these unbacktrackable decisions are not clauses, no dependent conflict clause is generated. Also, removal of these assignments is simply resetting the SAT instance to restore all the decisions that were made. No tedious conflict clauses removal process is carried out. Conflict clauses generated for the other part of the problem are also intact, and hence this speeds up reruns with new assignments and additional constraints.

3.4.2 Prioritized Decision-making Variables

SAT is a decision problem so it is only supposed to tell if there is a solution (although it is proven through finding a solution). SAT allows adding constraints to a SAT problem but no preferences to favor certain sets of solutions. Problems could arise for multiple observations over multiple patterns if only constraints can be added. This problem can be illustrated with a circuit consisting of 2-input AND, OR and XOR gates with the same inputs as shown in Figure 3.7.

For a sa1 fault at B, it requires both patterns [A,B] = [0,0] and [1,0] to detect the fault at all three outputs X, Y and Z at least once. Assuming that a fault is observed once at all outputs, and the first pattern [A,B] = [1,0] is generated, the fault is observed at X and Z as their Boolean differences are 1. Outputs X and Z fulfill the requirement so if BD_X and BD_Z are constrained to 0 in the next pattern generation to avoid detecting again at those outputs, the pattern $[A, B] = [0, 0]$ cannot be found because it detects the fault at outputs Y and Z. Output Z is constrained to

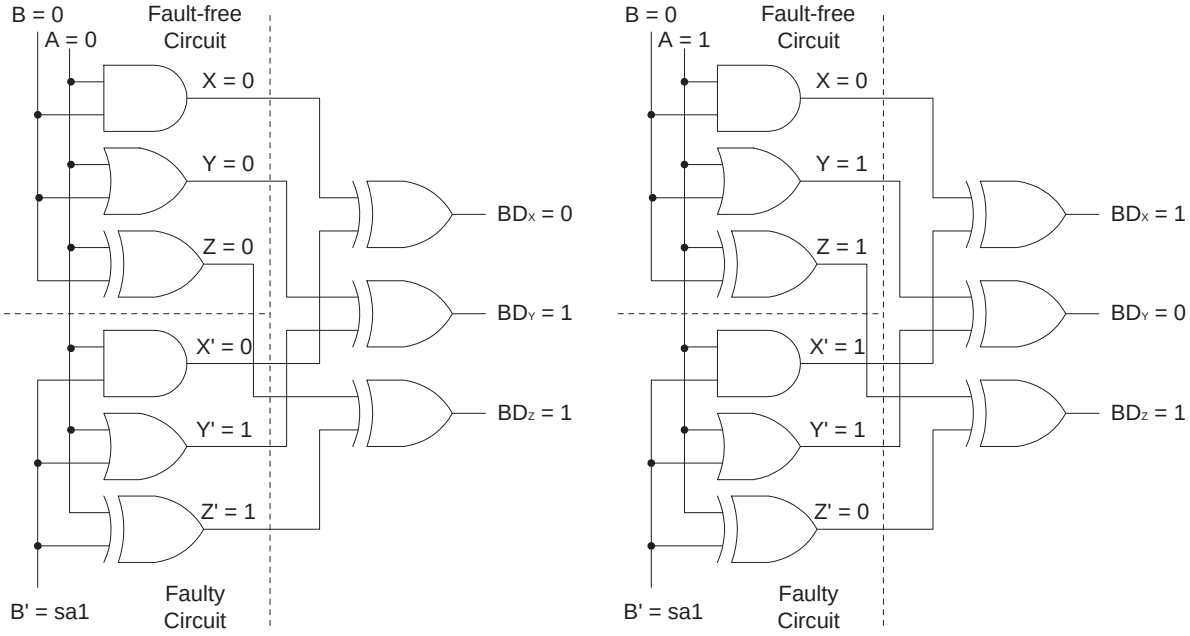

Figure 3.7: Example of 2-input AND, OR and XOR Circuit

0 so there would be a conflict to have $BD_Z = 1$ and the ATPG aborts observing at output Y because no pattern that uniquely observes the fault at Y can be found. [32] resolved the problem through removing Boolean differences of outputs which satisfied the observation requirements. However, as explained in the previous subsection, removing clauses diminishes the speedup brought by conflict clause analysis so it is not preferred. To avoid removing clauses, a list of variables are added to the SAT solver to be branched in order before using the default strategy in the SAT solver for choosing unassigned variables. In the above example, the list of prioritized variables are BD_X, BD_Y and BD_Z as no observation is made at any outputs. To have $BD_X = 1$ as the first decision, A has to be 1 and then $BD_Z = 1$. After the first pattern [A,B] = [1,0] is generated, the list is reduced to BD_Y only because outputs X and Z have fulfilled the observation requirements and removed from the list. This will allow [A,B] = [0,0] to be generated because $BD_Y = 1$ is the very first decision to be made in the SAT solver whereas BD_Z is not constrained to 0.

Another purpose of using prioritized variables is to find out the unobservable

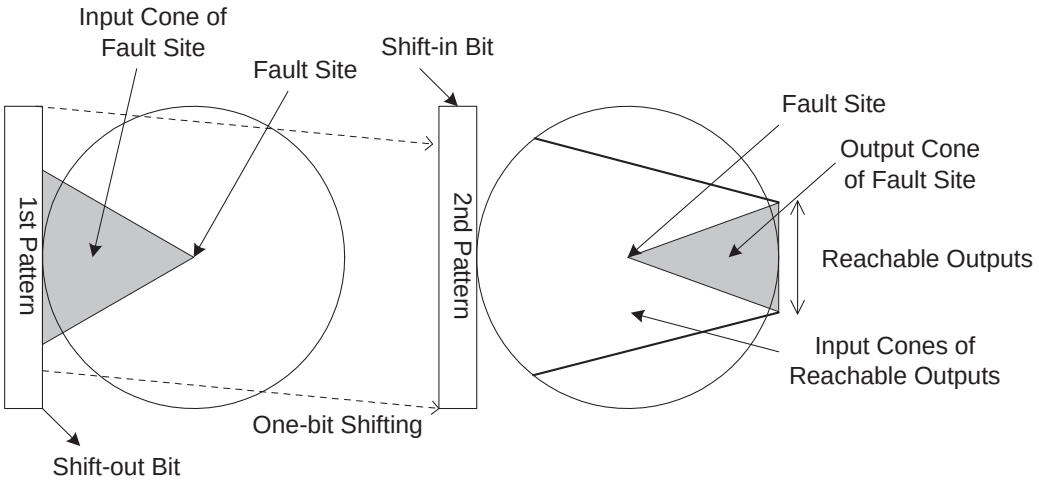

Figure 3.8: Cones Involved and One-bit Shifting in Skewed-load

outputs in the list. If the BD variables at the beginning of the list are assigned 0 after being solved, they can be claimed to be unobservable over the current constraints. For example, if the list of variables is BD_X, BD_Y and BD_Z and the solution has $BD_X = 0$, $BD_Y = BD_Z = 1$, output X cannot observe the fault. It is because BD_X is firstly branched to 1 but no solution is found so it has to be backtracked and reassigned to 0.

3.5 Skewed-load Transition Fault ATPG

In the previous sections SAT formulation for single stuck-at fault test generation was explained. Transition fault test generation, in terms of SAT, is an extension of single stuck-at fault test generation with an extra time frame for initialization. It relies on the same work flow described in Section 3.1, however it has a different way to construct the foundation SAT problem and the input and output constraints. Figure 3.8 shows the cones involved for skewed-load transition fault test generation.

In addition to the constraints for single stuck-at fault for the second pattern and output response generation, another set of variables of the fault-free circuit are created for the precedent clock cycle to generate the first pattern. The additional constraints are required for the input cone of the fault site and they can be constructed through

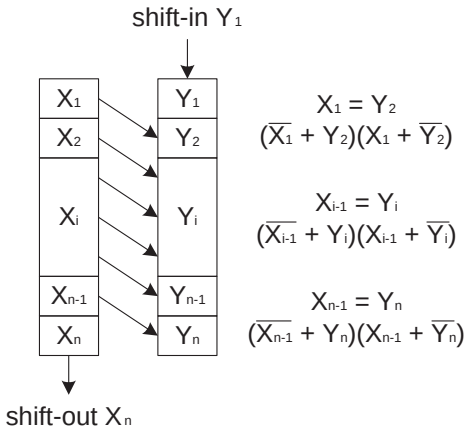

Figure 3.9: Constraints for Shifting

the technique described in Section 3.3.1. The stuck-at value is also assigned to the fault site in the circuit in the first clock cycle to complete the fault excitation for a transition fault. To impose the shifting property between the first pattern $[X_1...X_n]$ and the second pattern $[Y_1...Y_n]$, the relationship $X_i = Y_{i+1}$ can be formulated as $(\overline{X_i} + Y_{i+1})(X_i + \overline{Y_{i+1}})$, as shown in Figure 3.9. X_n and Y_1 are the shift-out and shift-in values so they do not have any relationship with the other patterns. By obtaining the values of the input variables of both time frames, a two-pattern test is generated for the targeted transition fault, although only the first bit of the second pattern is not in the shifted first pattern as required in skewed-load test application.

3.6 Undetect Look-ahead

Undetect Look-ahead (ULA) is a pattern post-processing step aimed at reducing the number of SAT problems to be solved. ULA takes place when a SAT problem is solved and a pattern is created during test generation, or updated during fault simulation with dynamic compaction. When a SAT problem is solved, the input pattern and the output response can be obtained from the solved SAT instance. Other specified values in the circuit are also available in the SAT instance. For example, wires along the sensitization paths must have specified values, though it is not limited to those

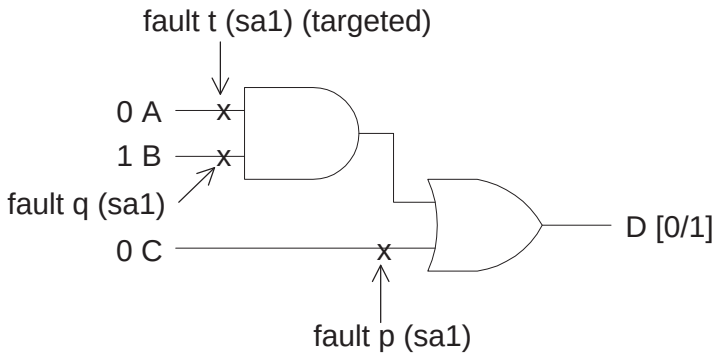

Figure 3.10: Example of Undetect Look-ahead

paths. After a pattern is generated or updated, a simple comparison is performed to the values in the SAT instance and all the subsequent faults.

For each subsequent fault, the stuck-at value of the fault and the current value at the fault site in the SAT instance are compared. No information can be obtained if the value at the fault site is unspecified. If the value is different from the stuck-at value of the fault, it is possible that the fault may be detected by this pattern because of the correct excitation value. However, if the value at the fault site is the same as the stuck-at value, it cannot excite the fault and thus there is no chance the fault can be detected by this pattern. The pattern number is then stored with the fault so that when the fault is being targeted, the pattern will not be simulated because it is known that the SAT problem will be unsatisfiable.

Figure 3.10 is an example of ULA. When fault t is targeted during generation, a pattern [A,B,C] = [0,1,0] is generated. Two other faults p and q are compared. Fault p has a sa1 value and the value at the fault site C is 0. The pattern provides the correct excitation value so this pattern will be simulated when fault q is processed. However, fault q has a sa1 value but the value at the fault site B is 1, which cannot excite the fault. The generated pattern cannot detect fault q so simulation on the pattern will be bypassed when fault q is processed.

Chapter 4

Experimental Results

The proposed SAT-based ATPG work flow is implemented in C/C++ with the integration of the Chaff [20] SAT solver. Single stuck-at fault and skewed-load transition fault test generation are performed on the three largest ISCAS89 benchmark circuits.

First the experimental flow is elaborated. This is followed by the results on pattern count and coverage (all coverage results are subjected to rounding error). It is worth mentioning that our ATPG work flow is not geared toward maximizing BCE or GE coverage results reported in this chapter. Due to the lack of experiments on field returns (caused by logistic and not technical reasons) that can empirically validate the test sets generated by our method, the BCE and GE coverage results are used only as a substitute to assess the quality in terms of surrogate detection. The chapter is concluded by discussing the runtime.

4.1 Experimental Flow

The experimental flow is illustrated in Figure 4.1. The three largest circuits in the ISCAS'89 benchmark set, namely s35932, s38417 and s38584, are used to generate the results presented in this chapter. The circuits are first modified to add a scan chain to join all the registers. Full scan makes all registers in the circuit to be accessible through an additional scan port and the test generation becomes a combinational one, which the proposed ATPG system is capable of.

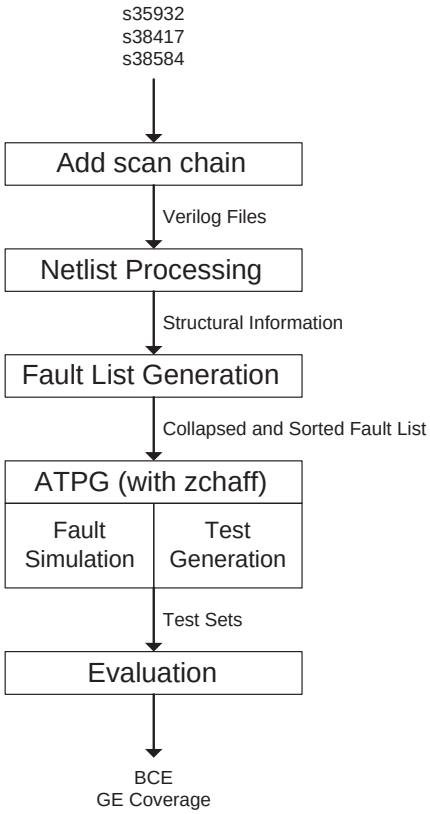

Figure 4.1: Experimental Flow

ATPG starts by reading the modified circuit in Verilog format. Structural information is gathered from the netlist of the circuit. Next, a fault list is generated according to the structural information and the fault model used. Equivalent faults are removed to reduce the size of the fault list. The fault list is then sorted by SCOAP, where faults with lowest observability are processed first to yield better compaction. Once the sorted fault list is available, the actual ATPG process begins.

Both fault simulation and test generation process involve solving SAT problems. *zchaff*, a C++ implementation of Chaff, is integrated into the ATPG system for the SAT solving capability with the modifications described in Section 3.4. By creating a SAT instance within the program, clauses can be added to construct the SAT problem. When all the clauses for the problem are added and all additional constraints are set, a solve function is invoked and *zchaff* solves the problem, returning either the problem

is satisfiable, unsatisfiable or a timeout error if a preset time constraint is violated. The time constraint for solving each SAT instance is set to 200ms. The values of all variables in the SAT instance can be accessed once *zchaff* concludes the problem is satisfiable.

Each fault is targeted once, and goes through both the fault simulation and test generation process. Fault simulation finds out if the existing patterns can detect the targeting fault at any of its reachable output. It also has the capability of dynamic compaction, where Don't Cares filling occurs. A SAT problem is solved for each pattern simulating a fault. If the problem is satisfiable, the pattern can detect the fault at some outputs which do not yet satisfy the observation requirements. On the other hand, the pattern cannot detect the fault if the problem is unsatisfiable. When all reachable outputs fulfill observation requirements during fault simulation, test generation process for this fault is not required. Otherwise, new patterns are needed to satisfy the requirements at all reachable outputs.

Test generation process is similar to fault simulation, except all patterns are applied as constraints to one SAT instance. Instead of applying a pattern as input assignment, constraining the inputs not to be any of the existing patterns to avoid repeated test generation. In each test generation iteration, one SAT instance is solved to generate one pattern. Because of the increments of observation counts at all reachable outputs, output constraints are different in every iteration, and thus distinguishing patterns which detect the fault at different outputs are generated.

After all faults are processed, a complete test set is generated. The test is then evaluated in a post-processing step. Bridging Coverage Estimate (BCE) is calculated by simulating the test over the sorted fault list, whereas Gate Exhaustive (GE) coverage is obtained by simulating the test over a newly created fault list that stuck-at faults are only located at the outputs of logic gates.

4.2 Pattern Count and Coverage Results

Table 4.1 is a summary of the test sets generated for the three circuits for both models using the proposed method over different observation requirements. Column 1 shows

Circuit	Type	any	1	2	3	4	5
s35932	#pat	23	30	56	82	107	131
(ssa)	BCE(%)	80.24	87.62	95.76	97.30	98.73	99.32
	GE(%)	85.00	85.90	86.58	87.16	87.61	87.71
s38417	#pat	199	582	971	1353	1748	2106
(ssa)	BCE(%)	89.47	94.86	97.94	99.04	99.50	99.71
	GE(%)	95.89	96.54	96.68	96.75	96.76	96.79
s38584	#pat	248	797	1411	2046	2254	2497
(ssa)	BCE(%)	84.73	93.62	97.56	98.96	99.45	99.69
	GE(%)	93.7	94.36	94.58	94.75	94.82	94.87
s35932	#pat	48	72	121	177	228	289
(sltr)	BCE(%)	91.18	95.12	98.73	99.5	99.74	99.83
	GE(%)	87.6	87.94	88.15	88.16	88.06	88.19
s38417	#pat	339	988	1670	2378	3078	3729
(sltr)	BCE(%)	94.47	97.07	98.91	99.48	99.71	99.81
	GE(%)	96.25	96.72	96.79	96.86	96.89	96.88
s38584	#pat	309	1319	2121	3004	4004	4834
(sltr)	BCE(%)	90.31	96.24	98.78	99.48	99.73	99.82
	GE(%)	94.05	94.64	94.96	95.14	95.25	95.26

Table 4.1: Evaluation Summary of the Six Tests

the circuit name and the fault model used. ssa and sltr represent single stuck-at and skewed-load transition fault models respectively. Column 2 shows the type of results the row is presenting. Number of patterns, Bridging Coverage Estimate (BCE) and Gate Exhaustive (GE) coverage are recorded in each test. Column 3 provides result for observation requirement of at least one observation at any reachable outputs for all faults. This mimics the results of single observation as in traditional ATPG, although the proposed method attempts to propagate the fault effect to as many reachable outputs as possible for the only generated pattern targeting a fault. Columns 4 to 8 provide results on the required minimum observations of one to five at all the reachable outputs for all faults.

- *Pattern Count*:

  Dynamic compaction plays an important role in keeping the pattern count low. The results for any-observation for the six tests are comparable to single observation test generated with state-of-the-art commercial products. According to the results, pattern count increase seems to be linear over the minimum number of observations. With one observation per fault-output pair as reference,

increase in pattern counts is about 0.6y to 0.9y times larger for y-observations in the six test sets. This is because some faults are detect more than y times at some fault-output pairs. No new pattern is required to target these "easy" fault-output combinations when the required minimum observations increases. This results in less-than-y-time increase in pattern counts.

- *Bridging Coverage Estimate*:

  Since BCE is only for evaluating single pattern test, only the second patterns, which are the excitation patterns, of the two-pattern transition fault tests is applied for calculating BCE. For one- and two-observations, the improvement in BCE over any-observation is significant. Among the six tests, the improvement is between 3.44% to 15.48%. All six cases have BCE of over 95% with two-observations, and most reach 99% with four-observations. However, improvement over three-observations is minimal and the effect of multiple observations seems to be saturated. The improvement is as little as 0.38% for the case of s38417(sltr) from two to five observations. It does not seems to be beneficial to use over two-observations for marginal BCE improvement, at the cost of considerable increase in pattern count.

- *Gate Exhaustive Coverage*:

  Similar to BCE evaluation, only the second pattern is applied for the two pattern tests. Also a different fault list with only gate outputs with both stuck-at values is used for the evaluation. As shown in the results, improvement in GE coverage over multiple observations is not significant. Only up to 2.71% of GE coverage is gained through at least five observations at all reachable outputs for all faults.

  The reason for the minor effect on GE coverage is that the objective of the use of multiple observations is to improve the diversification of fault excitations and fault effect propagations at the circuit level. How a fault is excited locally at gate level is not accounted for in our algorithm so it is not necessary to consider all input combinations of all gates during test generation. Also, nonobservable input combinations are not excluded in the calculation of the coverage credit,

Circuit	Obs	0	1-14	15+
s35932	any	27697	70978	206
(ssa)	o1	21184	75752	1945
	o2	20164	67574	11143
	o3	19357	58093	21431
	o4	18470	42276	38135
	o5	16844	27938	54099
s38417	any	78007	309436	77914
(ssa)	o1	10578	271607	183172
	o2	10148	221007	234202
	o3	10029	183727	271601
	o4	9999	156845	298513
	o5	9986	134453	320918
s38584	any	68633	134757	23162
(ssa)	o1	43512	116812	66228
	o2	42487	90383	93682
	o3	42083	71290	113179
	o4	42578	60467	123507
	o5	42486	52401	131665

Table 4.2: Number of f-o Pairs in Different Regions of Number of Observations

as explained in Section 3.1. Therefore the GE coverage has an upper bound that depends on the number of nonobservable input combinations in the circuit, which can only be identified in test generation under the gate exhaustive model.

- *Observability Distributions*

  Table 4.2 shows the number of f-o pairs in different regions of number of observations and Figure 4.2 shows the distribution of number of f-o pairs over the number of observations per f-o pair up to 14 observations. For any-observation, the count is expected to be more than traditional single observation approach because the proposed approach attempts to find as many sensitization paths for a pattern as possible. There are significantly more f-o pairs which have no observation for any-observation than n-observations. There are also very few f-o pairs that have 15 or more observations for any-observation; the number of f-o pairs increases considerably when the minimum observations required increases.

  Within the region of 1-14 observations per f-o pair, the number of f-o pairs which have fewer than the minimum observations required is very low. The number of f-o pairs surges at the minimum observations required per f-o pair

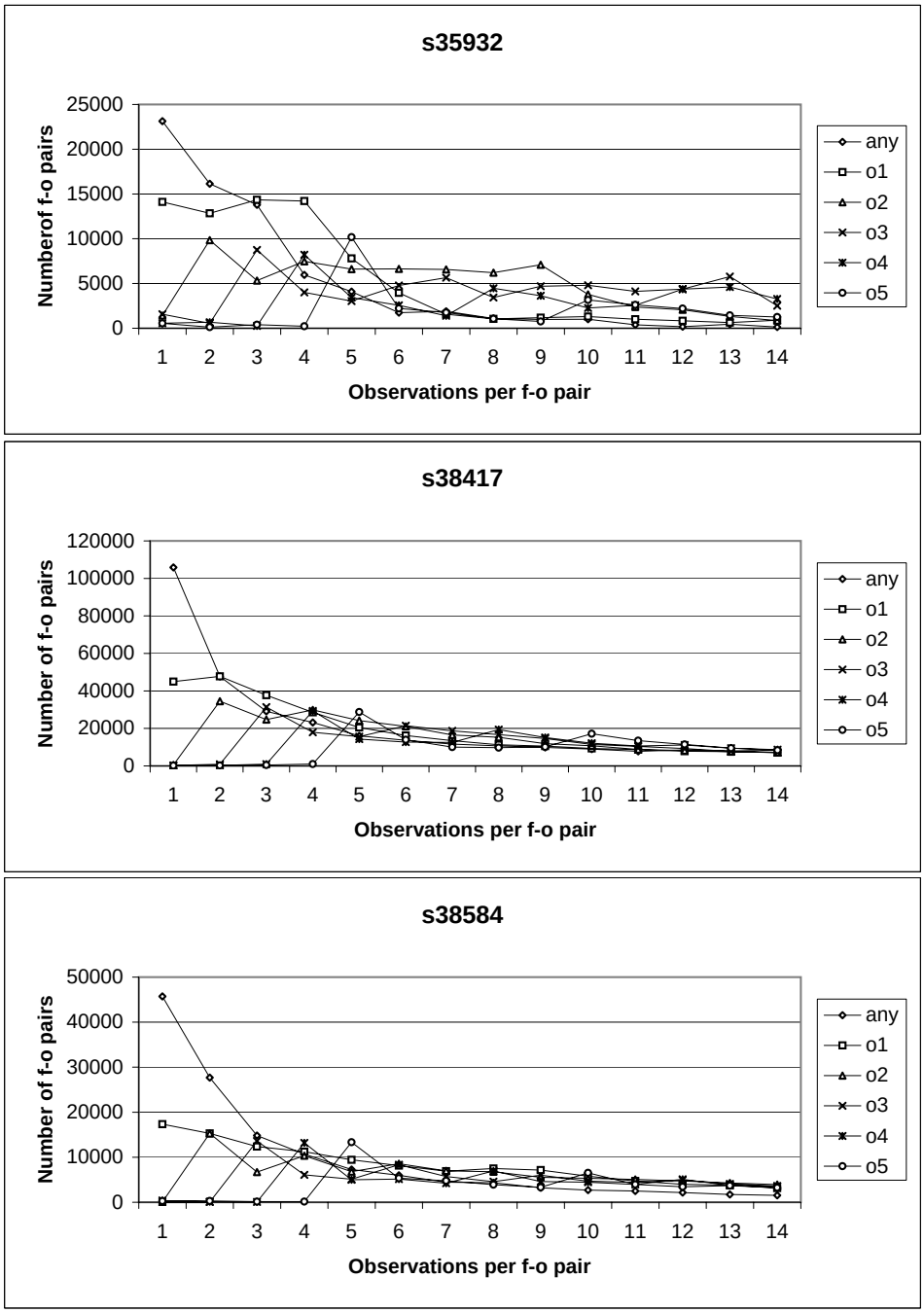

Figure 4.2: Observability Distribution for Stuck-at Fault Test Generation

Circuit	#Faults	Obs	FO	AO	PO	UO
s35932(ssa)	53888	o1	44692	0	3164	6032
		o2	44692	0	3164	6032
		o3	44692	0	3164	6032
		o4	44692	0	3164	6032
		o5	44692	0	3164	6032
s38417(ssa)	48513	o1	46415	0	1875	223
		o2	46415	0	1875	223
		o3	46415	0	1875	223
		o4	46413	2	1875	223
		o5	46411	3	1876	223
s38584(ssa)	55456	o1	49549	0	3183	2724
		o2	49549	0	3217	2690
		o3	49543	6	3227	2680
		o4	49547	2	3221	2686
		o5	49549	0	3220	2687

Table 4.3: Observablity over All Faults for Stuck-at Fault Test Generation

and it gradually decreases with an increasing number of observations per f-o pair. This shows the efforts of the proposed ATPG on keeping at least n observations for each f-o pair for n-observations.

- *Observability over All Faults*

  Table 4.3 shows the observability of all the faults of the three stuck-at fault test sets. Column 1-3 shows the circuit name, number of faults of the circuit, and the observation requirements respectively. Column 4 (FO) is the number of faults that all the reachable outputs of the faults observe the targeted fault more than required. Column 5 (AO) is the number of faults that all the reachable outputs observe the faults at least once, excluding those reported in Column 4. Column 6 (PO) is the number of faults that only some reachable outputs observe the faults. Column 7 (UO) is the number of unobservable faults.

  From the results from s38417, it is getting more difficult to fulfill full observation on all f-o pairs under the same time constraints. There are a few faults which can only be observed at all reachable outputs but not meeting the required observations. In s38584, the time constraint of 200ms per SAT instance seems to have more effect on the results. There are more time constraint violations for s38584 than for the other two circuits. These violations make the SAT solver

to abort some faults. The effect is significant for one-observation, where the abortions cause more faults not to be observed by any pattern.

4.3 Runtime

Runtime is determined by the complexity of the circuit, the fault model that is employed and also the number of observations required. The experiments are carried on computing grid nodes which run at various speeds and resources, and under unpredictable load at any instance. Hence the exact runtime of the experiments cannot be accurately determined. The rough range of runtime for s35932 test generation is from several hours to a few days, while s38417 and s38584 take days for test generation with both stuck-at and transition faults models and over different observation requirements.

The major reason for the long runtime is because SAT-based fault simulation (as required for dynamic compaction that accounts for multiple observations) cannot perform parallel fault simulation. Each SAT problem can only model one fault and a single pattern can be applied, so solving a SAT problem can only evaluate one fault per pattern. This results in large amount of SAT instances to be solved for the completion of the entire ATPG process.

Runtime profiling is performed on stuck-at fault test generations as shown in Table 4.4. Although the runtime cannot be measured precisely, a few findings need to be emphasized. About 87.6% of time is spent on average on the SAT_solve function, which is the total solving time zchaff spent on all the SAT problems. Another 10.5% of time was spent on the SAT_reset function, which resets all the assigned values and restores all the decisions made in the SAT solver. These two functions are scaled by the number of SAT instances required to solve in the entire ATPG process and thus dominate the total runtime of the program. Other parts of the program such as initializing a new SAT instance, setting up the basic SAT problem that can be shared between fault simulation and test generation, scale by the number of faults. Though significant, the number of faults is considerably lower than the number of SAT instances to be solved, thus resulting in a relatively smaller contribution to the

Circuit	Obs	SAT_reset	SAT_solve	Others
s35932	any	9.43%	84.41%	6.16%
(ssa)	o1	9.61%	85.71%	4.69%
w/o ULA	o2	10.04%	87.40%	2.55%
	o3	11.14%	87.14%	1.73%
	o4	10.55%	88.04%	1.40%
	o5	10.43%	88.43%	1.14%
s38417	any	10.01%	85.29%	4.70%
(ssa)	o1	10.86%	87.49%	1.65%
w/o ULA	o2	10.32%	88.73%	0.94%
	o3	10.92%	88.40%	0.69%
	o4	10.25%	89.20%	0.55%
	o5	11.02%	88.56%	0.42%
s38584	any	10.44%	86.03%	3.53%
(ssa)	o1	10.94%	87.87%	1.19%
w/o ULA	o2	10.51%	88.81%	0.68%
	o3	10.88%	88.45%	0.67%
	o4	10.66%	88.92%	0.42%
	o5	11.19%	88.42%	0.39%
Average		10.51%	87.63%	1.86%

Table 4.4: Runtime Profile without ULA

total runtime. Obviously, reducing the number of SAT instances to be solved is the most effective way to reduce the overall runtime. Therefore, ULA described in Section 3.6 has been proposed to achieve reduction of SAT instances to be solved during fault simulation.

Table 4.5 shows the number of SAT instances solved with and without the use of ULA. Due to the significance of SAT solving on the total runtime, as shown in Table 4.4, the reduction in number of SAT instances solved reflects the amount of runtime saved in general. In fact, the runtime reduction is even better than the reduction in number of SAT instances solved according to the experimental results. A possible explanation is that it takes more time on average for the SAT solver to disprove an unsatisfiable problem than finding a solution for a satisfiable problem. While it may only take several decisions to obtain a solution for a satisfiable problem, disproving an unsatisfiable problem requires the SAT solvers to search all the solution space. Since ULA avoids a large amount of unsatisfiable problems to be solved, runtime reduction can be achieved.

The number of SAT instances solved is also related to pattern counts. The increase

in number of instances solved over multiple observations has a similar trend as the increase in pattern counts. This is because there are more pattern-fault pairs to be simulated if more patterns are generated. Therefore, the increase in the number of SAT instances to be solved, which causes the increase in runtime, follows the increase in number of patterns generated.

With ULA, the runtime profile is also changed as shown in Table 4.6. The changes in the proportions of the other parts, which include the extra computation of ULA, decrease over multiple observations. The reason is that ULA scales only by the number of patterns, so tests with more SAT instances are less affected because the number of patterns becomes relatively insignificant when compared to the number of SAT instances.

The summary of results for tests generated with ULA is presented in Table 4.7. BCE and GE coverage over all the tests are similar to those without ULA. The largest discrepancies of BCE and GE coverage are 1.54% and 0.22% correspondingly. However, the use of ULA seems to have a tendency of generating more patterns. The increase is not significant for s35932 and s38417, however for s38584 the pattern count increases from 5% to 35%. A possible explaination is that it takes longer time to solve the SAT problems when there are fewer conflict clauses learned at a result of the reduction of the SAT instances solved. While there are more time constraint violations, relaxed output constraints discussed in Section 3.3 are used, and thus more patterns are required to detect the faults at all reachable outputs.

Circuit	Obs	w/o ULA	ULA	Reduce%
s35932(ssa)	any	308175	173183	43.80%
	o1	512325	284121	44.54%
	o2	999538	537169	46.26%
	o3	1461896	773869	47.06%
	o4	1908917	970831	49.14%
	o5	2346244	1181273	49.65%
s38417(ssa)	any	918502	454843	50.48%
	o1	2834478	1589392	43.93%
	o2	5175699	2851115	44.91%
	o3	7529171	4105087	45.48%
	o4	9895729	5348299	45.95%
	o5	12330191	6489566	47.37%
s38584(ssa)	any	1373982	875595	36.27%
	o1	4508660	3990726	11.49%
	o2	8700821	6615011	23.97%
	o3	12505344	8683667	30.56%
	o4	14487575	10767015	25.68%
	o5	16797986	12722458	24.26%
s35932(sltr)	any	688965	436819	36.60%
	o1	1225796	766003	37.51%
	o2	2330426	1422350	38.97%
	o3	3383506	2024793	40.16%
	o4	4438859	2603545	41.35%
	o5	5536833	3299873	40.40%
s38417(sltr)	any	2448023	1639444	33.03%
	o1	8533239	6071898	28.84%
	o2	15377942	11011341	28.40%
	o3	22327939	15990939	28.38%
	o4	29619372	20274186	31.55%
	o5	36023826	24717986	31.38%
s38584(sltr)	any	2356850	1720564	27.00%
	o1	8717596	7444701	14.60%
	o2	15368109	12530775	18.46%
	o3	22951452	17683184	22.95%
	o4	31267139	22728152	27.31%
	o5	37434169	26732028	28.59%

Table 4.5: Number of SAT Instances Solved

Circuit	Obs	SAT_reset	SAT_solve	Others
s35932	any	5.72%	71.80%	22.48%
(ssa)	o1	7.31%	77.55%	15.14%
ULA	o2	8.63%	82.50%	8.87%
	o3	9.20%	84.18%	6.62%
	o4	9.45%	85.11%	5.44%
	o5	9.71%	85.66%	4.64%
s38417	any	7.32%	78.01%	14.67%
(ssa)	o1	8.97%	86.01%	5.03%
ULA	o2	9.17%	87.95%	2.87%
	o3	9.26%	88.65%	2.10%
	o4	10.00%	88.61%	1.39%
	o5	10.86%	88.19%	0.95%
s38584	any	8.14%	82.68%	9.18%
(ssa)	o1	9.35%	88.27%	2.38%
ULA	o2	9.45%	89.02%	1.53%
	o3	9.46%	89.30%	1.23%
	o4	11.07%	88.22%	0.71%
	o5	9.77%	89.27%	0.96%
Average		9.05%	85.05%	5.90%

Table 4.6: Runtime Profile with ULA

Circuit	Type	any	1	2	3	4	5
s35932	#pat	23	30	55	80	101	132
(ssa)	BCE(%)	80.28	86.26	94.76	97.05	98.64	99.32
	GE(%)	85.02	85.94	86.33	86.82	87.49	87.72
s38417	#pat	198	628	1020	1372	1758	2153
(ssa)	BCE(%)	89.94	96.4	97.9	99.05	99.5	99.72
	GE(%)	95.97	96.55	96.68	96.77	96.77	96.839
s38584	#pat	238	1076	1699	2147	2676	3076
(ssa)	BCE(%)	85.52	92.88	97.45	98.87	99.45	99.7
	GE(%)	93.7	94.36	94.63	94.72	94.85	94.88
s35932	#pat	47	69	124	182	225	291
(sltr)	BCE(%)	90.99	94.51	98.2	99.38	99.74	99.83
	GE(%)	87.48	88.07	87.93	88.31	88.07	88.17
s38417	#pat	359	1018	1767	2456	3112	3794
(sltr)	BCE(%)	94.57	96.85	98.88	99.5	99.72	99.82
	GE(%)	96.3	96.71	96.82	96.87	96.9	96.9
s38584	#pat	344	1703	2552	3529	4454	5094
(sltr)	BCE(%)	90.29	95.94	98.63	99.49	99.74	99.83
	GE(%)	94.04	94.63	94.96	95.16	95.22	95.29

Table 4.7: Evaluation Summary of the Six Tests with Undetect Look-ahead

Chapter 5

Conclusion

Following the improvement in effectiveness of defect screening achieved by n-detect and TARO, this thesis has investigated the effects of multiple observations at all the reachable outputs for all faults. By formulating the ATPG problem into SAT, test generation constraints can be incorporated into a SAT problem through the addition of clauses, without the difficulty of modifying the entire test generation algorithm as in the traditional ATPG approaches.

To achieve multiple observations at all the fault-output pairs, multiple patterns are generated for each fault, with different constraints that drive the sensitization to all the reachable outputs according to the observation statistics. A new customized decision-making strategy has been implemented to achieve sensitization diversification. A special type of assignments is used in the SAT solver to reduce the inefficiency of frequently adding and removing the single literal clauses, as required during fault simulation. A pattern post-processing step is also implemented to avoid patterns not capable of detecting a particular fault to skip simulating the respective fault.

In terms of screening effectiveness, the tests generated with the proposed ATPG are shown to improve bridging coverage estimate when increasing the number of observations to two. The benefits of three-observations (and over) are found to be minimal. The effect of multiple observations on gate exhaustive coverage is also insignificant. It was also observed that pattern counts and runtime spent on test

generation grow considerably for multiple observations. It can be concluded that two-observations provide a good trade-off between coverage and pattern count increase.

There are plenty of directions to improve this work. Runtime has to be improved to make the proposed method practical on larger circuits. The major problem is obviously the number of SAT instances to be solved during fault simulation. In the current implementation we are limited by simulating one pattern per fault per SAT instance. A multiple-pattern SAT-based fault simulator will be beneficial to the overall performance, as the number of SAT instances to be solved decreases by a factor of how many patterns can be incorporated in one SAT instance (despite the larger size of each SAT instance). Another alternative is to modify an existing fault simulator from the traditional ATPG flow, which can take advantage of bit-level parallelism and to adopt it to count the observations at all fault-output pairs. In addition, output sorting during the construction of the preference list for observation output preference can be improved by also using the overall observation statistics for better observation balance.

The above discussed extensions will likely facilitate the scaling of the proposed techniques to circuits of practical relevance, thus enabling their evaluation on defective chips which have escaped the screening process dependent on patterns generated using a standard ATPG flow.

Bibliography

[1] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, Boston, 2000.

[2] S. Chakravarty, A. Jain, N. Radhakrishman, E. W. Savage, and S. T. Zachariah. Experimental evaluation of scan tests for bridges. In *Proc. Intl. Test Conf.*, 2002.

[3] K. Y. Cho, S. Mitra, and E. J. McCluskey. Gate exhaustive testing. In *Proc. Intl. Test Conf.*, 2005.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[6] J. Dworak, J. Wingfield, B. Cobb, S. Lee, L. Wang, and M. R. Mercer. Fortuitous detection and its impact on test set sizes using stuck-at and transition faults. In *Proc. 17th IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI Systems*, 2002.

[7] N. Een and N. Sorensson. An extensible sat-solver. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/, 2003.

[8] G. Fey, J. Shi, and R. Drechsler. Efficiency of multi-valued encoding in sat-based atpg. In *Proc. Intl. Symposium on Multiple-Valued Logic*, 2006.

[9] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. Comput.*, C-32(12), December 1983.

[10] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Fransico, 1979.

[11] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Tran. Comput.*, C-30, March 1981.

[12] E. Goldberg and Y. Novikov. Berkmin: a fast and robust sat-solver. In *Proc. Design Automation and Test in Europe*, 2002.

[13] L. H. Goldstein. Controllability/observability analysis of digital circuits. *IEEE Tran. on Circuits and Systems*, CAS-26(9), Sept 1979.

[14] M. R. Grimaila, S. Lee, J. Dworak, K. M. Butler, B. Stewart, H. Balachandran, B. Houchins, V. Mathur, J. Park, L. Wang, and M. R. Mercer. Redo - random excitation and deterministic observation - first commercial experiment. In *Proc. 17th VLSI Test Symposium*, pages 268–274, 1999.

[15] R. Guo, S. Mitra, E. Amyeen, J. Lee, S. Sivaraj, and S. Venkataraman. Evaluation of test metrics: Stuck-at, bridge coverage estimate and gate exhaustive. In *Proc. 24th IEEE VLSI Test Symposium*, 2006.

[16] T. Larrabee. Efficient generation of test patterns using boolean difference. In *Proc. Int. Test. Conf.*, August 1989.

[17] S. C. Ma, P. France, and E. J. McCluskey. An experimental chip to evaluate test techniques: Experiment results. In *Proc. Intl. Test Conf.*, 1995.

[18] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[19] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc., 1994.

[20] M. H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering and efficient sat solver. In *Proc. 38th Design Automation Conf.*, 2001.

[21] S. Mourad and Y. Zorian. *Principles of Testing Electronic Systems*. Wiley-Interscience Publication, 2000.

[22] I. Park, A. Al-Yamani, and E. J. McCluskey. Effective taro pattern generation. In *Proc. 23th IEEE VLSI Test Symposium*, 2005.

[23] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *Intl. Journal on Software Tools for Technology Transfer*, 7(2), 2005.

[24] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM J. Res. Develop.*, 10:278–291, July 1966.

[25] M. H. Schulz, E. Trischler, and T. M. Sarfert. Socrates: A highly efficient automatic test pattern generation system. *IEEE Tran. on CAD*, 7(1), January 1988.

[26] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. 1992.

[27] J. Shi, G. Fey, R. Drechsler, A. Glowatz, F. Hapke, and J. Schloffel. Passat: Efficient sat-based test pattern generation for industrial circuits. In *Proc. IEEE Annual Symposium on VLSI*, 2005.

[28] G. L. Smith. Model for delay faults based upon paths. In *Proc. Intl. Test Conf.*, 1985.

[29] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Tran. on CAD*, 15, 1996.

[30] P. Tafertshofer, A. Ganz, and M. Henftling. A sat-based implication engine for efficient atpg, equivalence checking and optimization of netlists. In *Intl Conf. on CAD*, 1997.

[31] C. Tseng and E. J. McCluskey. Multiple-output propagation transition fault test. In *Proc. Intl. Test Conf.*, 2001.

[32] B. Vaidya and M. B. Tahoori. Delay test generation with all reachable output propagation and multiple excitations. In *Proc. 20th IEEE Intl Symposium. on Defect and Fault Tolerance in VLSI Systems*, 2005.

[33] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar. Transition fault simulation. *IEEE Design and Test*, pages 32–38, April 1987.

[34] H. Zhang. Sato: An efficient propositional prover. In *Proc. Intl. Conf. on Automated Deduction*, 1997.

Index