# Techniques for Enhancing Test and Diagnosis of Digital Circuits

Sarvesh Pradeep Prabhu

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

Doctor of Philosophy
in
Computer Engineering

Michael S. Hsiao, Chair
Chao Wang
Sandeep K. Shukla
Lynn Abbott
Mark Shimozono

9 December, 2014
Blacksburg, Virginia

Keywords: LBIST, LFSR-reseeding, Diagnostic Test Generation, Automated Test Pattern Generation (ATPG), Property Checking

**Techniques for Enhancing Test and Diagnosis of Digital Circuits**

Sarvesh Prabhu

(ABSTRACT)

Test and Diagnosis are critical areas in semiconductor manufacturing. Every chip manufactured using a new or pre-mature technology or process needs to be tested for manufacturing defects to ensure defective chips are not sold to the customer. Conventionally, test is done by mounting the chip on an Automated Test Equipment (ATE) and applying test patterns to test for different faults. With shrinking feature sizes, the complexity of the circuits on chip is increasing, which in turn increases the number of test patterns needed to test the chip comprehensively. This increases the test application time which further increases the cost of test, ultimately leading to increase in the cost per device.

Furthermore, chips that fail during test need to be diagnosed to determine the cause of the failure so that the manufacturing process can be improved to increase the yield. With increase in the size and complexity of the circuits, diagnosis is becoming an even more challenging and time consuming process. Fast diagnosis of failing chips can help in reducing the ramp-up to the high volume manufacturing stage and thus reduce the time to market. To reduce the time needed for diagnosis, efficient diagnostic patterns have to be generated that can distinguish between several faults. However, in order to reduce the test application time, the total number of patterns should be minimized. We propose a technique for generating diagnostic patterns that are inherently compact. Experimental results show up to 73% reduction in the number of diagnostic patterns needed to distinguish all faults.

Logic Built-in Self-Test (LBIST) is an alternative methodology for testing, wherein all components needed to test the chip are on the chip itself. This eliminates the need of expensive ATEs and allows for at-speed testing of chips. However, there is hardware overhead incurred in storing deterministic test patterns on chip and failing chips are hard to diagnose in this LBIST architecture due to limited observability. We propose a technique to reduce the number of patterns needed to be stored on chip and thus reduce the hardware overhead. We also propose a new LBIST architecture which increases the diagnosability in LBIST with a minimal hardware overhead. These two techniques overcome the disadvantages of LBIST and can make LBIST more popular solution for testing of chips.

Modern designs may contain a large number of small embedded memories. Memory Built-in Self-Test (MBIST) is the conventional technique of testing memories, but it incurs hardware overhead. Using MBIST for small embedded memories is impractical as the hardware overhead would be significantly high. Test generation for such circuits is difficult because the fault effect needs to be propagated through the memory. We propose a new technique for testing of circuits with embedded memories. By using SMT solver, we model memory at a high level of abstraction using theory of array, while keeping the surrounding logic at gate level. This effectively converts the test generation problem into a combinational test generation problem and make test generation easier than the conventional techniques.

This work was supported in part by SRC grant 2011-TJ-2134 and NSF grant 1016675.

*~ To my Family ~*

# Acknowledgments

I would, first of all, like to thank my research advisor, Dr. Michael S. Hsiao, for his selfless guidance and support during my research. His exceptional teaching in the course, Testing of Digital Systems, inspired me to join his PROACTIVE research group. I was honored to get a chance to work with him and was deeply inspired by his extensive knowledge, dedication and amiable nature. I thank him for believing in me and motivating me for pursuing PhD.

I would like to thank Dr. Chao Wang, Dr. Sandeep K. Shukla, Dr. Lynn Abbott and Dr. Mark Shimozono for serving on my dissertation committee.

My sincere thanks also goes to Loganathan Lingappan and Vijay Gangaram from Intel Corporation for funding this research and for providing me an opportunity to intern in their team. Their suggestions during our meetings immensely helped me in this research work. I would also like to thank my team members at Intel Corporation Saparya Krishnamoorthy and Prashant Sanjay for the good time I had during the internship.

A special acknowledgement goes to my best friend Supratik Misra for being a wonderful roommate and labmate, for all the memorable fun-filled trips we have had, for helping me get through the difficult times, and for being a constant source of support and encouragement.

Special thanks to my roommates Apoorv Naik, Anup Mandlekar and Deepak Mane for making my stay at Virginia Tech truly memorable.

I would also like to thank my friends Amrapali Dengada, Amuru Sai Dhiraj, Vireshwar Kumar, Sriram Malladi, Krishna Chaitanya Pabbuleti, Sachin Hirve, Abhineet Parchure, Harmish Modi and Pallavi Deshmukh for the great time I had in Blacksburg.

I thank my PROACTIVE labmates Avinash Desai, Sharad Bagri, Indira Priyadarshani, Vineeth Acharya, Michael D'souza, Neha Goel, Maheshwar Chandrasekhar and Mainak Banga for building a fun environment in the lab and for all the fun we had outside the lab.

I would like to express my gratitude to my parents Pradeep Prabhu and Vidhya Prabhu, my sister Sheetal Prabhu and my extended family for their love, support and encouragement at every step of my life.

Sarvesh Prabhu

December 2014

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As the semiconductor industry extends its reach to the high-volume consumer electronics market space, a device that can offer a diverse set of functions and an enhanced user experience will be favored. This in turn drives feature integration by combining multiple complex blocks to form system-on-a-chip (SOC). On the other hand, advances in manufacturing technology continues to reduce the feature size. This enables the integration of more functionally on the chip. Thus, complexity of the design is increasing and as a result, the number of transistors on a die continue to grow. Modern System-On-Chip (SOC) designs embed more than 100 million transistors running at operating frequencies in the gigahertz range [1].

With reduction in feature size, precise control during the manufacturing process is becoming more challenging and the chips produced are more susceptible to defects. Hence, every chip manufactured based on a new or pre-mature technology needs to be rigorously tested for manufacturing defects to minimize customer returns and maximize customer satisfaction. For chips that fail, diagnosis needs to be performed to determine the location and the reason for the failure, so that the design, layout or the manufacturing process can be modified to improve the yield.

State-of-the-art tools and algorithms exist for efficient test and diagnosis, but with increasing design complexity, the cost of testing is also increasing. Hence, these tools and algorithms need to be regularly improved. Otherwise, the cost of testing a chip would eventually surpass the cost of manufacturing the chip as shown in Figure 1.1.

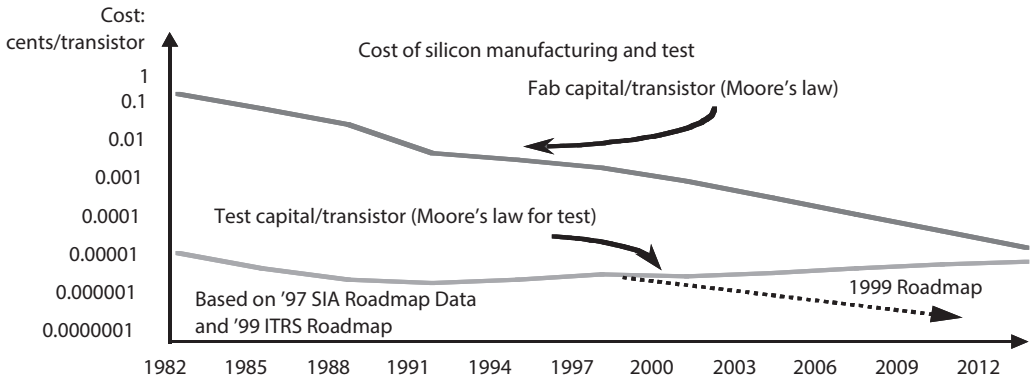



Figure 1.1: Manufacturing Cost vs Test Cost Trend (K.-T. Cheng, "Embedded Software-Based Self-Testing for SoC Design," in Embedded Systems Handbook, pp. 28-1-28-19, CRC Press, 2005. *Used with permission*)

## 1.1 Test Data Volume Challenge

Conventionally, manufacturing test is done using an Automated Test Equipment (ATE). ATEs are very expensive and ATE-based testing is slow because of the communication bottleneck between the ATE and the chip. Also, with the increase in complexity of the designs, the number of patterns needed to test the chip sufficiently in also increasing significantly. The International Technology Roadmap for Semiconductors (ITRS) is the fifteen-year assessment of the semiconductor industry's future technology requirements. Figure 1.2 shows the ITRS projections for the number of patterns needed to test all stuck-at faults in the SOC designs for the next fifteen years [2]. The ITRS predicts exponential increase in the number of test patterns, which directly affects the test application time and the test cost.

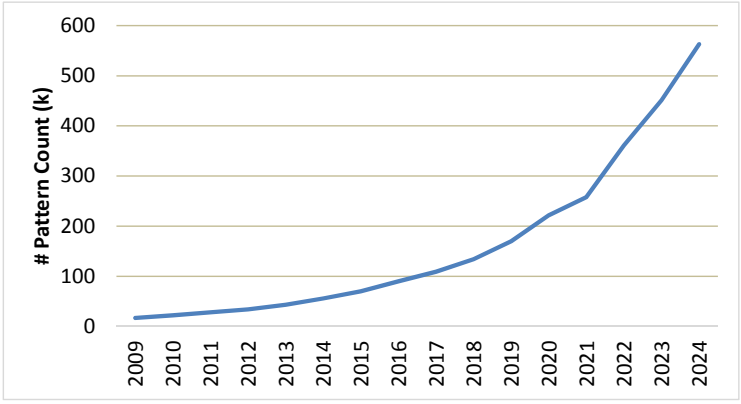



Figure 1.2: ITRS Projections for the Number of Patterns Needed to Detect All Stuck-at Faults for SOC (Graph plotted from data in International Technology Roadmap for Semiconductors, "ITRS 2010 Update", http://www.itrs.net/Links/2010ITRS/Home2010.htm. *Used under fair use*)

The patterns to be applied to a chip are stored in the memory of the ATE which is limited. With increase in the number of patterns, the memory requirement of the ATE is also increasing. Furthermore, the fault free responses of the chip also need to be stored in the ATE memory to be compared against the response of the chip. The test patterns, fault free responses, responses from the chip and several other test environmental parameters together constitute the total test data which is the amount of memory required on the ATE. The ITRS in its 2012 update [3], projects an exponential increase in the test data volume in the coming years as shown in Figure 1.3. This again contributes to increase in test application time and overall test cost. In this research, we propose a technique to reduce the number of patterns needed to distinguish between different faults which helps in reducing test data volume.

Logic Built-In Self-Test (LBIST) is an alternative test methodology which enables a chip to test itself. All components needed to test a chip are present on the chip itself. It thus eliminates the requirement of an expensive ATE. LBIST provides at-speed testing and multiple blocks on the chip can be tested in parallel. Thus, LBIST is the most promising solution for

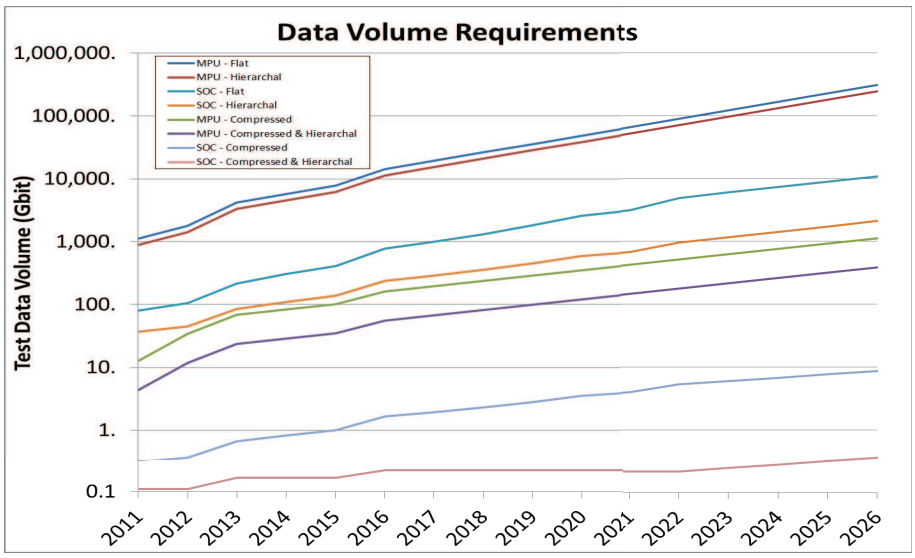



Figure 1.3: ITRS Projections for Test Data Volume Requirements (International Technology Roadmap for Semiconductors, "ITRS 2012 Update", http://www.itrs.net/Links/2012ITRS/Home2012.htm. *Used under fair use*)

reducing test time and test cost. However, significant hardware overhead and poor diagnosability have prevented wide adoption LBIST. In this dissertation, we propose techniques to reduce hardware overhead and to improve diagnosability in LBIST.

## 1.2 Embedded Memories Challenge

Currently, hardware development starts from a high-level description language (HDL); however, most of the methods for test and diagnosis still are at the gate-level. Ignoring the higher levels of abstraction misses tremendous opportunities for search space reduction and reasoning. Furthermore, the recent increase of small on-chip memories adds another layer of difficulty to test and diagnosis. Conventional test generation uses DFT methodology of scan to convert sequential test generation into combinational test generation. But using scan for entire memory is expensive in terms of time as a large number of patterns need to be applied

to test each cell of memory and scan-based testing is slow. Another technique for testing memories is using Memory Built-In Self-Test (MBIST), which similar to LBIST, enables the memory to test itself. However, employing MBIST for each embedded memory incurs huge hardware overhead.

A lot of work has been done on SAT-based test generation for combinational circuits. However, SAT solvers work at the Boolean level only. In recent years, Satisfiability Modulo Theory (SMT) solvers have been implemented on top of SAT solvers and can handle Boolean values, bit-vectors, linear arithmetic, uninterpreted functions etc. These SMT solvers have been used popularly in software test and validation but the use of SMT for hardware test and diagnosis has not been sufficiently explored. In this dissertation, we propose a technique for test generation of circuits with embedded memories using SMT solvers. We also make use of SMT solvers for reducing the test data volume.

Thus, this dissertation presents new techniques for test and diagnosis to tackle the challenges posed by increase in test data volume and presence of embedded memories.

## 1.3 Contributions of this Dissertation

In LBIST, an LFSR based pattern generator is used to apply pseudo random patterns to the Device Under Test (DUT). The random vectors can detect most of the faults but there are some faults which need specific vectors to be detected. For those hard to detect faults we find the vectors needed to detect them by using Automated Test Pattern Generation (ATPG). To reduce the overhead of storing these vectors directly on chip we try to find the initial state of the LFSR known as seed that can generate these vectors by solving complex linear equations. For the linear equations to have a solution the order of the vectors and number of vectors to be chained is critical and is very hard to determine. In Chapter 3,

we propose a SMT-based LFSR reseeding technique that chains faults instead of chaining vectors and finds a seed such that all the chained faults are detected. This reduces the total number of seeds needed to detect all the hard to detect faults [4].

To test a chip, vectors are generated using ATPG that can detect all the faults in the chip. To reduce the time needed for testing each chip, we ideally want the test set to be as small as possible. This compaction of the test set makes diagnosis difficult as there are many faults that cannot be distinguished by the compact test set. Conventionally we target each undistinguished fault-pair separately and use an Automated Diagnostic Test generation (ADTG) to generate a vector that can distinguish them. In Chapter 4, we propose a SMT-based technique for diagnostic test generation in combinational circuits which targets multiple fault-pairs in each iteration and finds a single vector that distinguishes them. This reduces the total number of vectors needed to distinguish all the fault-pairs and thus helps in reducing the test data volume. This makes diagnosis of failing chip easier without significant increase in the test application time [5].

Modern designs often consist of small embedded memories. Using memory BIST for such small memories is not practical because of hardware overhead incurred. If the circuit is flattened at the gate level, then conventional test generation techniques find it difficult to excite or propagate faults through the memory as the address bus and data bus are represented as unrelated Boolean variables. In Chapter 5, we propose a new technique for test generation of circuits with embedded memories by using SMT solvers. We represent the memories at higher level of abstraction using theory of arrays which allows for efficient excitation and propagation of fault effects through the memory and makes test generation of circuits with embedded memories easier than the current techniques [6].

In LBIST, test patterns are applied to the DUT and the responses from DUT are compressed by passing them through a Multiple Input Signature Register (MISR). The final state of the

MISR at the end of the test session is called a signature and is compared against the fault free signature stored on chip to determine whether the chip is good or faulty. Although compression helps in reducing the hardware overhead of storing all the fault free responses, it makes diagnosis difficult as cause of the failure need to be determined based on one faulty signature. In Chapter 6, we propose a new property checking based LBIST architecture to improve diagnosability in LBIST. We use missing patterns or invariants in fault free responses as hardware monitors to detect faults. If any property is violated then the failing vector and property number are stored in a dedicated fail memory and are used for diagnosis [7, 8].

## 1.4 Dissertation Organization

The rest of the dissertation is organized as follows.

- Chapter 2 discusses the basic concepts in test and diagnosis that are necessary to understand this dissertation.
- Chapter 3 presents an SMT-based technique for LFSR reseeding in logic BIST environment.
- Chapter 4 presents an SMT-based diagnostic test generation method for combinational circuits.
- Chapter 5 presents an SMT-based test generation technique for circuits with embedded memories.
- Chapter 6 presents a property checking based LBIST architecture that improves diagnosability in LBIST.
- Chapter 7 concludes the dissertation.

# Chapter 2

# Background

Every manufactured chip needs to be tested for manufacturing defects. A chip can be defective due to several reasons such as

- Silicon Defects
- Mask Contamination
- Process Variation
- Defective Oxide
- Interconnect Defects
- Photo Lithography Defects
- Transistor Open/Short

To test a chip exhaustively, all possible input patterns need to be applied. For a chip with $n$ primary inputs, there are $2^n$ possible input patterns. For a circuit with large $n$, it is not possible to apply all the exhaustive input patterns as the number of patterns

is significantly high and the test application time is impractical. A better, realistic approach is to select a subset of input patterns such that there is a high confidence that any physical defect present in the chip will be detected by these patterns. The easiest way to test a chip is to apply random patterns and verify that the responses of the chip are correct. Although random patterns can detect a lot of faults, there are some random resistant faults which can only be detected by some specific patterns. Since such faults cannot be detected by random patterns, the amount of confidence achieved by random testing is not high. Hence, deterministic techniques are used to generate efficient test patterns. In order to generate these patterns, fault models and automated test pattern generation targeted toward the fault model is used.

## 2.1 Fault Model

A fault is a logical effect that represents the behavior of a structural/physical defect present in the chip. In other words, physical defects in the chip are modeled as faults. Various fault models exist that represent different types of defects. The most popular fault model is the stuck-at fault model wherein a defect in the chip has an effect of holding a signal in the circuit to a constant 1 (VCC) or a constant 0 (GND) as shown in Figure 2.1.


(a) Stuck-at 0 fault (b) Stuck-at 1 fault

Figure 2.1: Stuck-at Fault Model

The stuck-at fault model has several advantages such as

– The fault list is easy to generate.

- A test set that detects all the stuck-at faults in the circuit has a high probability of detecting any type of physical defect present in the chip.
- The total number of faults is linear in number of gates in the circuit.
- It is independent of the process technology used to manufacture the chip.
- Many other fault models can be represented as a combination of different stuck-at faults.

The fault model usually assumes the presence of a single defect, but an actual failing chip can contain multiple defects. However, experiments have shown that a test set that detects all single faults has a high probability of detecting multiple faults [9]. The other popular fault models are bridging faults, transition faults and path delay faults. In this research we have only used the stuck-at fault model for test generation as well as diagnosis.

## 2.2 Automated Test Pattern Generation (ATPG)

To generate test patterns for a particular fault model, a fault list is first created. A fault list for the stuck-at fault model contains a stuck-at-0 and stuck-at-1 fault for each signal in the circuit. For each fault in the fault list, a test pattern is generated that can detect the fault. This process of generating a test pattern for a specific target fault is called Automated Test Pattern Generation (ATPG).

The test generation problem can be explained with a miter circuit as shown in Figure 2.2.

It contains two copies of the circuit, one is fault free and other has a target fault $f$ injected in it. The corresponding primary outputs are XORed and output of all

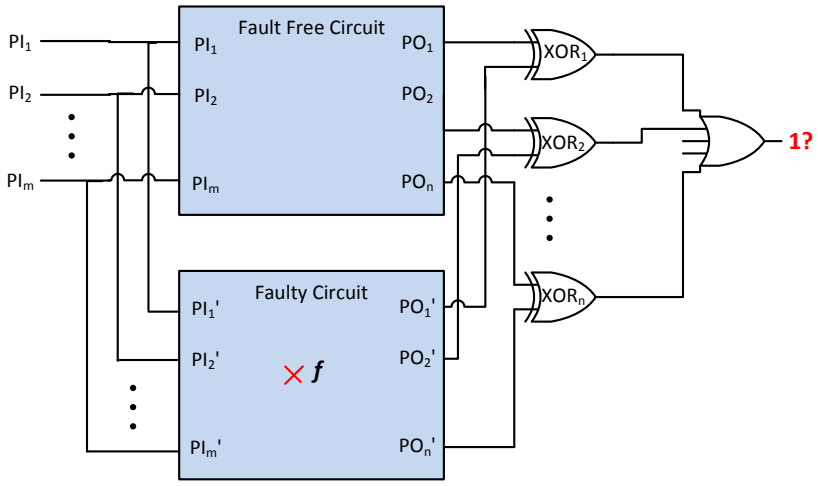



Figure 2.2: Miter Circuit for Test Generation

these XOR gates are connected to an OR gate. The corresponding primary inputs of both the circuits are tied together. The task is to assign values to the primary inputs $PI_1 - PI_m$ such that the output of the OR gate is set to 1. This means the fault free response and response of the circuit in presence of the fault are different which implies that the fault is detected by the generated pattern.

There are several popular structural algorithms for ATPG namely D-Algorithm [10], PODEM [11], FAN [12], SOCRATES [13], which assign values to the signals such that the target fault is excited and fault effect is propagated to a primary output. In these algorithms, a miter circuit is not used but instead a single copy of the circuit is used. However, instead of binary logic a 5-valued logic is used to represent the signal values namely $\{0, 1, X, D, \bar{D}\}$ where $D$ represents 1/0, which means that the fault free value of the signal is 1 and faulty value of the signal is 0. Similarly, $\bar{D}$ represents 0/1. Thus, in order to detect a fault, the primary inputs should be assigned values such that a fault effect ($D$ *or* $\bar{D}$) is seen at some primary output. These algorithms systematically sweep through all possible input patterns to find a pattern that detects the target

fault. Several techniques for speeding up PODEM based test pattern generation were proposed in [14].

Another popular technique of test generation is SAT-based ATPG [15, 16] in which the miter circuit is converted into a Boolean formula and the formula is given to a SAT solver. The SAT solver assigns values to the signals such that the target fault is detected or proves that no such assignment exists in which case the target fault is untestable. Details of SAT solver will be explained in Section 2.5.

In each iteration of ATPG, one fault is chosen from the fault list as the target fault and a pattern is generated that detects that fault. The remaining faults are simulated to determine which other faults are detected by the same pattern. These detected faults are removed or dropped from the fault list. This process is known as fault dropping. The aim of performing fault dropping is to reduce the size of test set which eventually reduces the time needed for testing every chip. Several static and dynamic test compaction techniques exist [17–24] which further reduce the size of the test set generated using ATPG. This compacted test set is applied to every manufactured chip so that minimal time is spent on testing each chip.

### 2.2.1 Fault Coverage

Fault coverage (FC) is a metric used to measure the quality of a test set. Fault coverage is defined as

$$Fault\ Coverage = \frac{Number\ of\ detected\ faults}{Total\ number\ of\ faults} * 100$$

To test a chip efficiently, a test set that can achieve 100% fault coverage for the fault model used is desired. However, if the circuit has some untestable faults, then it will

be impossible to achieve 100% fault coverage. For these circuits, a test set that can detect all the testable faults in the circuit is desired. Note that a test set that achieves 100% fault coverage is not guaranteed to detect all possible defects in the chip as the patterns are generated for a specific fault model and the actual defect present in the chip may be different than the fault model used.

## 2.3 Fault Diagnosis

For chips that fail test, the cause of failure needs to be determined so that the design, layout or manufacturing process can be improved to increase the yield. Since physical analysis of the chip is destructive and time consuming, it is necessary to narrow down the possible locations of the defect. This process is known as fault diagnosis. A fault diagnosis engine takes the netlist of the design, the failing vector(s) and the failing response(s) as input and returns a set of fault candidates that can best justify the behavior of the failing chip. These fault candidates then undergo physical failure analysis to determine the root cause of the failure.

Fault diagnosis can be broadly classified into following two paradigms [25]

**Cause-Effect Analysis:** In this technique, different faults or causes of failure are simulated and the faulty response is compared to the response of the chip. Faults having the best match to the observed response of the chip are returned as the fault candidates.

**Effect-Cause Analysis:** In this technique, we backtrace the fault effect from the primary outputs to determine the fault candidates that can justify the faulty behavior. Several heuristics such as paths sensitized by the failing vector, fan-in cone intersection

of primary outputs where fault effects propagated, controllability and observability values can be used to rank the different fault candidates.

To reduce the time needed for physical failure analysis, it is important that the fault diagnosis engine returns a small set of candidates.

### 2.3.1 Diagnostic Fault Simulation

Diagnostic fault simulation is used to determine the diagnostic quality of the test set as well as for cause-effect analysis of failing chips. In diagnostic fault simulation, we simulate all faults without fault dropping for the entire test set and determine which faults are distinguished by comparing the responses for all the faults for each vector. Two faults $f_1$ and $f_2$ are considered to be distinguished if there exists a vector $V$ in the test set that satisfies any of the following conditions:

- $V$ detects $f_1$ but does not detect $f_2$.
- $V$ detects $f_2$ but does not detect $f_1$.
- $V$ detects both $f_1$ and $f_2$ but the faulty responses for both faults are different.

Diagnostic fault simulation divides all the faults into equivalence classes. This can be explained with a simple example. Consider a circuit with 1 primary output. Table 2.1 is the fault response dictionary which records the fault free responses and the faulty responses for all 5 faults $\{f_1, f_2, f_3, f_4, f_5\}$ for all 4 vectors in the test set $\{v_1, v_2, v_3, v_4\}$.

Figure 2.3 shows the fault classes formed after application of each vector.

Initially all the faults $\{f_1, f_2, f_3, f_4, f_5\}$ are in the same equivalence class. Vector $v_1$ detects faults $\{f_4, f_5\}$ hence they are distinguished from $\{f_1, f_2, f_3\}$. For circuits with

Table 2.1: Fault Response Dictionary

| | Fault Free | $f_1$ | $f_2$ | $f_3$ | $f_4$ | $f_5$ |
|---|---|---|---|---|---|---|
| $v_1$ | 1 | 1 | 1 | 1 | 0 | 0 |
| $v_2$ | 0 | 1 | 0 | 0 | 0 | 0 |
| $v_3$ | 1 | 0 | 0 | 1 | 1 | 1 |
| $v_4$ | 0 | 0 | 1 | 0 | 1 | 1 |

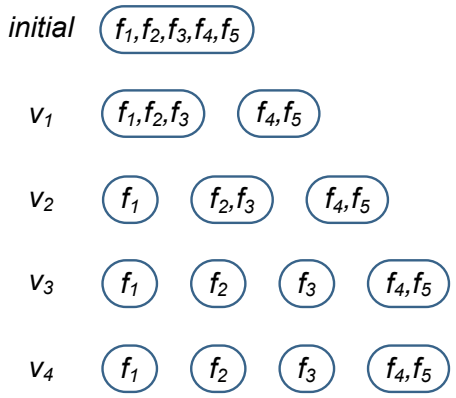



Figure 2.3: Fault Equivalence Classes formed during Diagnostic Fault Simulation

more than 1 primary output, the faulty responses for all the detected faults are compared and the faults with the same response are added into the same equivalence class. Vector $v_2$ detects fault $\{f_1\}$ only, hence it is distinguished from all other faults and need not be considered further for diagnostic fault simulation. Vector $v_3$ detects faults $\{f_1,f_2\}$. Hence $f_2$ is distinguished from $f_3$. Finally vector $v_4$ detects faults $\{f_2, f_4, f_5\}$ but this is not useful as we cannot divide any class any further. Thus, at the end of diagnostic fault simulation, fault classes are formed where all faults in the same class are not distinguished from each other. In the above example, faults $f_4$ and $f_5$ are undistinguished from each other, hence they are diagnostically equivalent for this test set. Diagnostic fault simulation is computationally intensive and if the actual defect in the chip is outside the fault model, then fault diagnosis based on diagnostic fault

simulation may lead to inaccurate fault candidates. Also, for large circuits the fault dictionary requires extremely high storage capacity.

Recall from Section 2.2, that conventional ATPG aims at detecting all the faults in the circuit and the test set generated by ATPG is compacted to reduce the test application time. To diagnose a failing chip, we need to determine which fault has a behavior that is similar to the response of the chip. But since the test set is compacted, multiple faults may exist that have the same response for each vector in the test set. Such faults are said to be diagnostically equivalent for the test set. Ideally a test set is desired which not only detects all the faults but also distinguishes all the faults from each other while keeping the test set size to a minimum. Satisfying all three requirements is extremely difficult, so there is always a trade-off between the size of the test set and its diagnostic quality.

### Diagnostic Resolution

Diagnostic Resolution is a metric used to determine the diagnostic quality of a test set. It is defined as

$$Diagnostic\ Resolution(DR) = \frac{Number\ of\ faults}{Number\ of\ equivalence\ classes}$$

In the above example the number of faults is 5 and the number of equivalence classes is 4. Hence the DR is 1.25. Ideally, a test set should be able to distinguish all the faults from each other. For such a test set, at the end of diagnostic simulation, each class will have a single fault. Hence, diagnostic resolution in the ideal case would be 1.

### 2.3.2 Automated Diagnostic Test Generation (ADTG)

In order to make diagnosis easier, the test set should be able to distinguish between every fault pair; hence, we generate more test patterns with the aim of distinguishing the faults that are not already distinguished by the ATPG test set. This process is called Automated Diagnostic Test Generation (ADTG).

After generating the ATPG test set, diagnostic fault simulation is performed to generate equivalence classes. For each fault pair in the same class, we try to generate a vector that distinguishes them. The ADTG problem can be explained with a miter circuit shown in Figure 2.4.

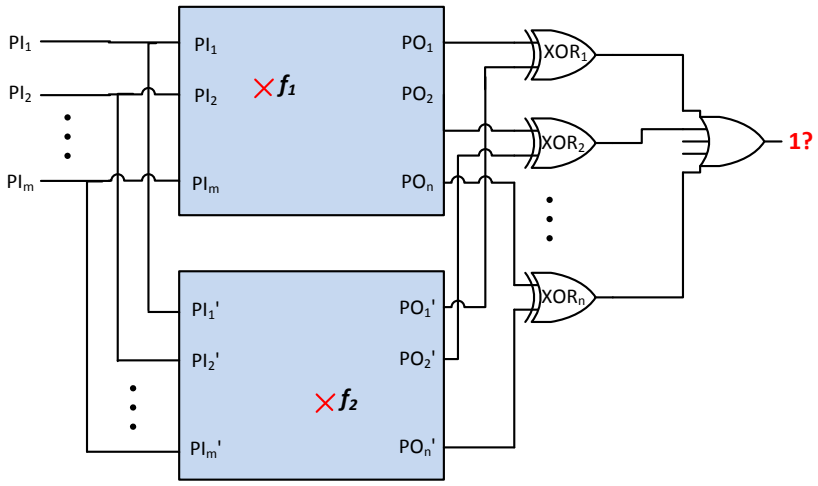



Figure 2.4: Miter Circuit for Diagnostic Test Generation

Here there are two copies of the circuit. One has fault $f_1$ injected in it and the other one has fault $f_2$ inject in it. The task is to assign values to the primary inputs such that the output of the OR gate is set to 1 which means the two faults $f_1$ and $f_2$ are distinguished from each other. A comparison between Figure 2.2 and Figure 2.4 shows a lot of similarity between ATPG and ADTG. Hence, ADTG can be performed by some modification to the ATPG algorithms. Several diagnostic test generation algorithms have been proposed in the literature [26–34]. The new patterns generated using ADTG

are then appended to the initial patterns generated by ATPG to form the final test set. Extending the ATPG test set by adding distinguishing patterns increases the test application time but it makes diagnosis of failing chips easier.

## 2.4 Logic Built-in Self Test (LBIST)

Conventional testing of a chip uses an external tester which is usually known as Automated Test Equipment (ATE). The chip is mounted on the ATE with a connection interface. The required environment configurations and test patterns are stored in the ATE memory. Test patterns are applied by the ATE to the chip and the responses of the chip are read back by the ATE. Finally, the ATE compares the fault free response and response of the chip. Thus, testing using ATE is slower because of the communication bottleneck between the chip and the ATE. Also a lot of time is spent on developing and validating the test program that runs on the ATE.

LBIST is an alternative way of testing the manufactured chip. In this technique, all the blocks necessary to test the chip are built into the chip itself. This leads to the following advantages.

- Eliminates the need of an expensive ATE.
- Allows at-speed testing as there is no communication bottleneck between the chip and the ATE.
- Allows for testing of chip after integration into the system.
- Multiple blocks on the chip can be tested in parallel

A basic block diagram of a conventional LBIST set-up is shown in Figure 4.1.

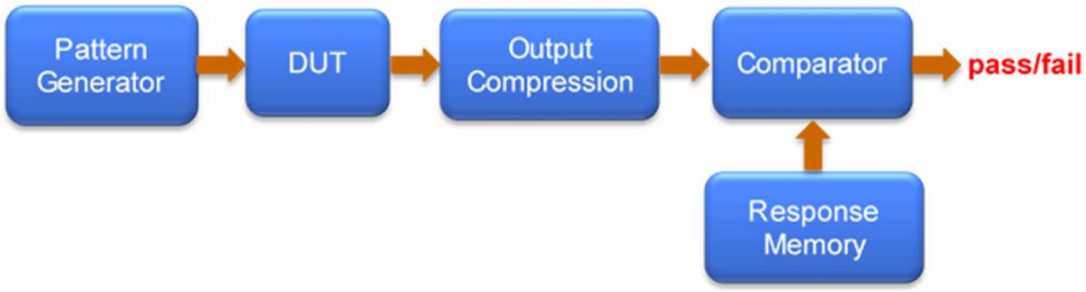



Figure 2.5: Conventional LBIST Setup

LBIST consists of a pattern generator which applies patterns to the device under test (DUT). The response of the DUT for these patterns is compressed by using some output compression. The corresponding compressed fault free responses are stored on a dedicated memory on chip known as response memory. An on-chip comparator compares the fault free and faulty responses to determine whether the chip is faulty or not.

### 2.4.1 Pattern Generator

The pattern generator in LBIST is usually a Linear Feedback Shift Register(LFSR) based Pseudo Random Pattern Generator(PRPG) [9, 35, 36]. An LFSR consists of D flip flops connected as shift registers and XOR gates. An example of a 4-bit LFSR is shown in Figure 2.6.

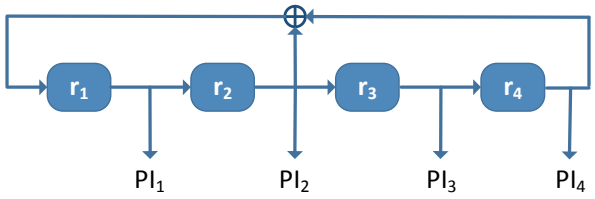



Figure 2.6: Example of 4-bit LFSR

Depending upon which flip flops are XORed in the feedback path, different configurations of LFSR exist for the same number of flip flops. This configuration is represented

by an equation known as characteristic equation. It is defined as

$$f(x) = 1 + h_1 x + h_2 x^2 + \cdots + h_{n-1} x^{n-1} + x^n$$

where the symbol $h_i$ is 1 if $r_i$ exists in the feedback path [37]. The characteristic equation for the LFSR in Figure 2.6 is $1 + x^2 + x^4$ since flip flops $r_2$ and $r_4$ are XORed in the feedback path.

For an LFSR with $n$ flip flops there are $2^n$ possible patterns. However, because of the structure of the LFSR, it cannot generate the all zero pattern. Hence the total number of valid patterns that can be generated by the LFSR is $2^n - 1$. However, not all patterns are generated for each configuration of LFSR. For any configuration of the LFSR, it generates a sequence of patterns of length $m \leq 2^n - 1$ in a random order and then the same sequence repeats indefinitely. The sequence generated by the LFSR in Figure 2.6 is shown in Figure 2.7.

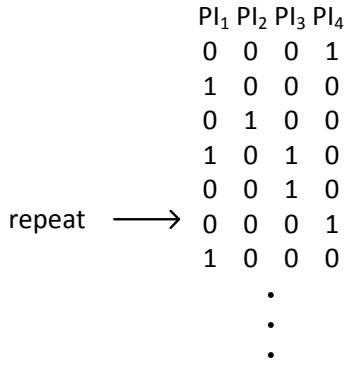



Figure 2.7: Pseudo Random Sequence Generated by LFSR

We can observe that this LFSR generates only 5 patterns and then the same sequence of patterns is repeated. A $n$ bit LFSR that can generate all $2^n - 1$ patterns is known as maximum-length LFSR. Maximum-length LFSR requires a specific characteristic equation known as a primitive polynomial. Finding a primitive polynomial for a given

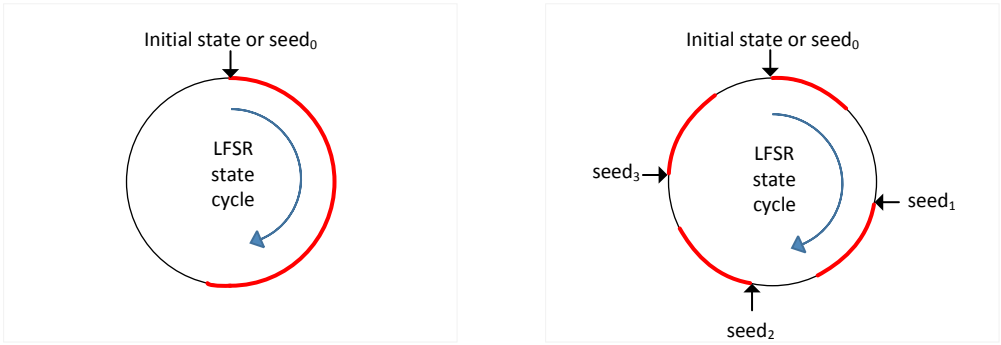
size of the LFSR is computationally intensive. However, the primitive polynomials for different sizes of LFSR have already been published [38–40] and can be readily used. Maximum-length LFSR is commonly used as PRPG, as it produces a sequence with 0.5 probability of generating 1's at every output [37]. But for large $n$ it is not possible to apply all $2^n - 1$ patterns due to time limitations. Hence only a subset of pseudo random patterns is applied. For some circuits, the fault coverage achieved by any subset of patterns of the maximal-length LFSR is low. Hence, a Weighted LFSR [41–43] is used to improve the fault coverages achieved by the PRPG. Use of spectral patterns was proposed in [44] which achieve better fault coverage than the weighted LFSR technique.

To test a chip using LBIST, starting from the initial state of the LFSR many patterns are applied to the chip. These pseudo random patterns detect all the easy to detect faults and usually achieve 70-90% fault coverage. For the remaining random resistant faults, deterministic patterns that are generated using some Automated Test Pattern Generation (ATPG) are applied. These deterministic patterns can be stored in a dedicated memory on-chip but this incurs huge hardware overhead. To reduce the hardware overhead of storing all the deterministic patterns, LFSR reseeding is used.

### LFSR Reseeding

The starting state of the LFSR is known as a seed. If patterns are applied only from the initial state or seed, then we explore a subset of the entire LFSR state space as shown in Figure 2.8a. Another technique is to apply patterns starting from different states of the LFSR as shown in Figure 2.8b. We apply a few patterns starting from the initial seed. Then we load the next state i.e. $seed_2$ into the LFSR and apply some more patterns starting from this seed. This process of loading a known state in the

LFSR is known as LFSR reseeding. Instead of loading random seeds into the LFSR, we intelligently compute seeds that will generate the deterministic patterns required to detect the faults that are missed by the initial seed.




(a) Single Seed Test Sequence (b) Multiple Seed Test Sequence

Figure 2.8: LFSR Reseeding

The patterns generated by LFSR are pseudo random but if the configuration of the LFSR and the starting state of the LFSR are known, then the patterns generated by the LFSR can be easily determined by solving the set of linear equations. Similarly if the LFSR configuration and the patterns that are to be applied are known then we can determine the starting state of LFSR required to generate those patterns by solving the linear equations. We first generate the deterministic pattern by using ATPG. Then we calculate the seed for the given LFSR that can generate the deterministic pattern. Storing the seed on chip instead of storing the complete pattern reduces the hardware overhead as the size of seed is significantly less than the size of complete pattern. We can convert each deterministic pattern into a seed or we can concatenate multiple vectors and generate a single seed that can generate the concatenated sequence of patterns.

While testing the chip, we first apply large number of patterns starting from the random initial state. Then a new seed is loaded from the memory into the LFSR and the LFSR is clocked for certain number of known cycles to generate the required

pattern(s). This is done for all the seeds stored in the memory. Several papers [45–51] have been published for efficient techniques to generate the LFSR seeds for the hard to detect faults. In Chapter 3 we have proposed a new technique for LFSR reseeding.

### 2.4.2 Output Compression

During LBIST test session, the output responses of the chip needs to be compared with the corresponding fault free responses to determine if the chip is faulty or not. Thus it is necessary to store the fault free values on chip. If the entire fault free response is stored for each vector then the hardware overhead of the memory needed to store these responses will be too high. To overcome this problem, the output responses of the chip are passed through a compressor which compresses the response into a smaller signatures. The corresponding fault free signatures are also compressed before storing on the chip. This reduces the hardware overhead significantly. Output compression techniques can be divided into space compression and time compression. The popular space compression techniques include Ones Count, Transition Count and Parity Bit. Since this compression is lossy, it leads to aliasing and fault masking. Aliasing happens when the compressed fault free response is same as the compressed faulty response and hence detection of the fault is missed. Fault masking occurs when multiple faults have same compressed response, hence it is not possible to distinguish between them.

Other type of compression is time compression. The most popular time compression technique and currently the state-of-the-art output compression technique used in LBIST is signature analysis. In this compression technique, all responses from the chip are passed through a Multiple Input Signature Register (MISR). At the end of the test session, the final state of the MISR is called the MISR signature. This final signature is compared to the fault free signature stored on chip. Thus, instead of stor-

ing a compressed response for every vector, we only need to store the final fault free MISR signature on chip. This results in extreme output compression. An example of a 4-bit MISR is shown in Figure 6.2.

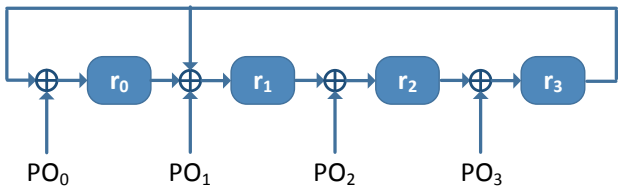



Figure 2.9: 4-bit Multiple Input Signature Register (MISR)

Although this extreme compression provides significant reduction in hardware overhead it makes diagnosis even more challenging. Now the failed chip needs to be diagnosed based on a single faulty signature, which is a tedious and time consuming task. We believe this poor diagnosability in LBIST has prevented wide adoption of LBIST. In Chapter 6 we propose a LBIST architecture which improves diagnosability in LBIST with minimum hardware overhead.

## 2.5 Satisfiability (SAT)

Satisfiability is one of the most fundamental problems in theoretical computer science namely, the problem of determining whether a formula expressing a constraint has a solution. The most well-known constraint satisfaction problem is propositional satisfiability SAT, where the goal is to decide whether a formula over Boolean variables, formed using logical connectives, can be made true by choosing true/false values for its variables [52].

The Boolean formula is usually represented in Conjunctive Normal Form (CNF). The CNF formula is a disjunction of clauses, and each clause is a conjunction of literals. A literal is a positive or negative phase of a Boolean variable. To satisfy a formula, each

clause of the formula needs to be satisfied i.e. atleast one literal in each clause must assigned true value. An example of a simple Boolean CNF formula is shown below.

$$f = (a + b + \bar{c})(\bar{b} + d)(\bar{a} + \bar{d})(c + b)$$

One possible solution to the above formula is $a = 0,\ b = 1,\ c = 0,\ d = 1$.

To convert the netlist of a circuit into a CNF formula, each gate is individually converted to its CNF formula and conjunction of all these formulas is the final formula. Table 2.2 lists the CNF formulas for basic gate types. For AND, NAND, OR, NOR, XOR and XNOR we assume 2 input gate with $a$ and $b$ as inputs and $c$ as output. BUFFER and NOT gate have single input $a$ and output $c$.

Table 2.2: CNF Formula for Basic Gate Types

| Gate Type | CNF Formula |
|---|---|
| AND | $(c + \bar{a} + \bar{b})(\bar{c} + a)(\bar{c} + b)$ |
| NAND | $(\bar{c} + \bar{a} + \bar{b})(c + a)(c + b)$ |
| OR | $(\bar{c} + a + b)(c + \bar{a})(c + \bar{b})$ |
| NOR | $(c + a + b)(\bar{c} + \bar{a})(\bar{c} + \bar{b})$ |
| XOR | $(\bar{c} + a + b)(\bar{c} + \bar{a} + \bar{b})(c + \bar{a} + b)(c + a + \bar{b})$ |
| XNOR | $(c + a + b)(c + \bar{a} + \bar{b})(\bar{c} + \bar{a} + b)(\bar{c} + a + \bar{b})$ |
| NOT | $(c + a)(\bar{c} + \bar{a})$ |
| BUFFER | $(\bar{c} + a)(c + \bar{a})$ |

For circuit in Figure 2.10 the CNF formula is

$$\begin{aligned} F \ = \ & (d + b)(\bar{d} + \bar{b}) \\ & (e + \bar{a} + \bar{d})(\bar{e} + a)(\bar{e} + d) \\ & (\bar{f} + b + c)(f + \bar{b})(f + \bar{c}) \\ & (\bar{g} + e + f)(\bar{g} + \bar{e} + \bar{f})(g + \bar{e} + f)(g + e + \bar{f}) \end{aligned}$$

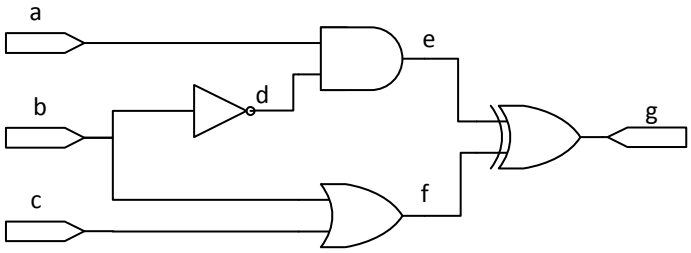



Figure 2.10: Example Circuit for CNF Formula

Each line in the formula corresponds to one individual gate in the circuit. Then the additional clauses for the objective are added to this formula. For example, if the objective is to set the ouput of the circuit shown in Figure 2.10 to 1, then a unit clause (i.e. clause with a single literal) of ($g$) is added to the CNF formula. This final formula is then given to a SAT solver which assigns values to the variables that justifies the circuit output to 1 or proves that no such assignment exists. For SAT-based test generation, the miter circuit shown in Figure 2.2 is converted to a CNF formula and given to a SAT solver.

The first algorithm to solve the Boolean satisfiability problem was proposed by Davis-Putnam in 1960 [53]. The DP Algorithm used binary resolution to find the solution to the Boolean satisfiability problem. This algorithm was improved to a branch and bound or decision tree based approach by Davis-Putnam-Love-Longland and the algorithm is known as DPLL algorithm [54]. All the popular algorithms for solving the SAT problem are extensions of the DPLL algorithm.

The problem of determining satisfiability of a Boolean formula is the first problem to be proven to belong to the class of NP-complete [55, 56]. However several advancements have been made over the years such as conflict-driven learning[57], non-chronological backtracking [57], two-literal watching [58] which considerably reduces the time taken by a SAT solver to solve any formula.

Some of the popular publicly available SAT solvers are GRASP [57], POSIT [59], BerkMin [60], SATO [61], Relsat [62], MiniSat [63], zChaff [58]. Currently zChaff is the most popular and most advanced publicly available SAT solver.

SAT solvers are extensively used for test generation [15, 16], diagnosis [64–66], bounded model checking [67, 68], equivalence checking [69–72], state justification [73–75], untestable fault identification [76] and other related areas.

## 2.6 Satisfiability Modulo Theory (SMT)

Conventional SAT solvers can only handle formulas with Boolean variables. However, Boolean logic is not expressive enough for representing many problems. Such problems are expressible in decidable fragments of first order logics (or simply theories henceforth), where the propositional variables are combined with constraints over individual variables. The problem of evaluating the satisfiability of first order formulas with respect to some background theories is called Satisfiability Modulo Theory (SMT) [77].

A theory is a set of axioms and rules of inference in which first order logic predicates are interpreted. Formally, a theory T is the set of axioms and all deducible formulas from the rules of inference (all true statements). Examples of theories typically used in SMT solvers are the theory of real numbers, the theory of integers, the theories of various data structures such as lists, arrays, bit vectors and so on. A formula F is T-satisfiable if F $\wedge$ T is satisfiable in the first order sense. If not, F is T-inconsistent, or T-unsatisfiable.[78]

SMT solvers can work at mixed level of abstraction and can reason about first order

formulas involving a mixture of Boolean values, bit-vectors, linear arithmetic, uninterpreted functions, difference logic, records, tuples, etc.

For deciding satisfiability or unsatisfiability of formulas in this kind of logics, during the last few years many successively more sophisticated techniques have been developed, most of which can be classified as being eager or lazy. In the eager approaches the input formula is translated, in a single satisfiability preserving step, into a propositional CNF, which is checked by a SAT solver for satisfiability. In the lazy approaches each atom in the SMT formula is internally assigned a Boolean variable. The Boolean formula is solved by an underlying SAT solver. The assignment returned by the SAT solver is then checked by the corresponding dedicated theory solvers. If any clause is violated then additional learning clauses are added in the context by the theory solvers and the search for a satisfying model continues. This process is repeated until a model compatible with the theory is found or all possible propositional models have been explored [79].

All the latest SMT solvers are based on DPLL(T) framework [79]. In this framework, the responsibility of Boolean reasoning is given to the Davis-Putnam-Logemann-Loveland (DPLL)-based SAT solver [53, 54] which, in turn, interacts with a solver for theory T through a well-defined interface. The theory solver need only worry about checking the feasibility of conjunctions of theory predicates passed on to it from the SAT solver as it explores the Boolean search space of the formula. For this integration to work well however, the theory solver must be able to participate in propagation and conflict analysis, i.e., it must be able to infer new facts from already established facts, as well as to supply succinct explanations of infeasibility when theory conflicts arise. In other words, the theory solver must be incremental and backtrackable [80]. If the

formula is satisfiable then the SMT solver returns a model i.e., values of the variables that satisfy the formula.

### 2.6.1 Application of SMT Solvers in Software Testing

SMT solvers are used in symbolic execution of software programs. In Symbolic execution [81], the program is executed using symbolic variables instead of concrete values. The program is first instrumented using an intermediate language such as CIL [82]. When the instrumented code is executed with symbolic inputs, it generates a trace of the path followed by the program. This trace consists of the symbolic expressions of the conditions encountered along the path. The conjunction of the symbolic expressions along the path forms a path constraint. This path constraint can be expressed as first order formulas in theories which the SMT solvers can handle. By negating one of the symbolic expressions along the current path, the path constraint of a target path is constructed. This path constraint is then converted into a formula that is solved by an underlying constraint solver, such as a Satisfiability Modulo Theory (SMT) solver [77, 78]. If the SMT solver returns a set of assignments that satisfies the formula, then the test input can be extracted from the assignment for which the formulated path can be exercised. If the SMT solver does not find a satisfying assignment, then the path is declared infeasible, i.e., the path cannot be exercised by any valid input. By negating one of the conditional expressions in the previous path, a new path can be formed, which is again given to the SMT solver. Thus the test generation continues by finding test inputs for every possible path in the code. Many tools like DART [83], CUTE [84], CREST [85], Java PathFinder [86], have been developed in recent years that use symbolic execution for automated generation of test inputs for software programs. To

tackle the path explosion problem in large programs, branch coverage is used instead of path coverage [85, 87, 88].

Some of the currently popular SMT solvers are Yices [89], Z3 [90], MathSAT 5 [91] and CVC3 [92]. SAT solvers are extensively used in hardware test and diagnosis and SMT solvers are used in software test and validation, but the use of SMT solvers in hardware test and diagnosis has not been explored in the past. In Chapter 3, Chapter 4 and Chapter 5, we have used SMT solvers for enhancing test and diagnosis of digital circuits. In this research, we have used SMT solvers Yices and Z3.

# Chapter 3

# SMT-based Technique for LFSR Reseeding

## 3.1 Chapter Overview

In order for LBIST to achieve coverages comparable with deterministic tests, multiple (and frequently many) seeds are often needed. Conventional techniques generated deterministic patterns and then chain the patterns and find a seed to generate these patterns. In this chapter, we propose a novel Satisfiability Modulo Theory (SMT) based technique that can reduce the number of seeds significantly while simultaneously achieving high coverage for LBIST. In this technique, we integrate the process of deterministic test generation and seed generation in one SMT process to eliminate the problems of chaining the separately generated deterministic patterns.

## 3.2 Introduction

In LBIST, starting from a random seed (initial state of the LFSR), a number of patterns can be derived and applied to the DUT, with which a number of the faults may be detected. For the remaining faults, one may apply additional sequences from other random seeds. However, this usually does not help to detect them, which most likely are either random-resistant or hard to test. Instead, an automatic test pattern generator (ATPG) is often used to generate an incompletely-specified test suite for these remaining faults. A test is incompletely specified if there exists don't-care bits in the test vector. Storing this entire test set directly on chip would be too expensive if the number of tests is large. So, one approach is to compress these ATPG vectors in such a way that these deterministic tests can somehow be derived from one (or few) seeds. This idea of LFSR reseeding was introduced in [93], whereby the derived ATPG vectors are chained together and a seed is computed that could reproduce the same ATPG vectors. Though promising, chaining all the ATPG vectors so that they can be produced from a single seed can be very challenging, especially if the number of ATPG vectors is large.

A number of other approaches for LFSR reseeding have been proposed in the past [45–51], most of which focus on finding and concatenating several incompletely specified test cubes. The linear equations for the specified bits in concatenated test cubes are then solved to find a suitable LFSR seed. In the following section, we will illustrate how existing methods work as well as point out the challenges involved.

### 3.2.1 Motivation

An example of conventional seed generation by chaining the test vectors is first given below. Let the characteristic equation of the LFSR be $1 + X^3 + X^4$, and let the deterministic test set consist of 6 incompletely-specified vectors: 11XXX, X0XX1, X1X0X, XX1X1, 1XXX0 and 0X1XX. For simplicity of discussion and illustration, we will concatenate 2 vectors for computing one seed. From the 6 ATPG vectors, the first concatenated 2-vector string would be 11XXXX0XX1. To find a seed such that the LFSR will generate the concatenated two vectors, the following set of linear equations needs to be solved:

$b0 = A0$

$b1 = A1$

$b6 = A0 + A1 + A2 + A3$

$b9 = A0 + A2$

Here, $b0 - b9$ are the bits of the concatenated vector and $A0 - A3$ are the bits of the LFSR seed. Essentially, these linear equations specify that the initial state of the LFSR contains the first vector 11XXX, and five clocks later, the second vector X0XX1 would be derived. Note that that linear equations only need to be solved for the bits that are specified in the test vector. The linear equations can be modified to suit other constraints, such as one vector is produced with every new clock, etc. After solving the linear equations, the seed found is $A0 = 1$, $A1 = 1$, $A2 = 0$, $A3 = 1$. The same process can be repeated for computing the other seeds for the remaining four vectors. At the end, one would end up with 3 seeds for these six test vectors.

From the above example, we see that in such a method, the deterministic test set generation and seed computation are two separate and independent processes. In other words, the vectors are first generated for a set of undetected faults by an ATPG,

followed by the linear equation solving to find the seed(s). The results obtained using this approach depends critically on the number of vectors concatenated for each seed and the LFSR polynomial. Longer concatenations will allow for fewer seeds, but it makes the set of linear equations harder to solve (or could be unsolvable). To tackle this problem a multiple polynomial LFSR reseeding method was presented in [94].

Table 3.1: Large Number of Seeds Needed for Conventional ATPG Vectors

| ckt | seeds | polynomials | ckt | seeds | polynomials |
|---|---|---|---|---|---|
| c432 | 16 | 2 | s953 | 46 | 4 |
| c499 | 23 | 5 | s1196 | 82 | 6 |
| c880 | 19 | 5 | s1238 | 84 | 7 |
| c1355 | 41 | 1 | s1423 | 26 | 8 |
| c1908 | 67 | 5 | s1488 | 56 | 3 |
| c2670 | 59 | 6 | s1494 | 55 | 3 |
| c3540 | 48 | 5 | s5378 | 129 | 3 |

Table 3.1 shows the number of seeds and the number of distinct polynomials needed if the number of concatenated vectors for each seed is set to 8 [45]. Note that as the number of concatenated vectors increases, the complexity of solving this set of linear equations also becomes harder. In almost all cases, multiple polynomials were needed to achieve full coverage. In total, we see that the number of seeds is always greater than 10, and more seeds were often needed for larger circuits. In addition to generating many seeds, the previous method of seed generation has the following disadvantages:

- It is unclear how many $m$ (out of $n$) ATPG vectors should be chained for one seed.
- The set of $m$ vectors to be chained has a direct effect on the result.
- Determining the order in which the $m$ vectors are chained can be challenging.

- If the linear equations for a concatenated vector has no solution, then
  - We need to change the order of concatenation, or
  - We need to change the set of $m$ vectors to be concatenated, or
  - Find a different ATPG test cubes for some of the faults in concatenated vector.

These aforementioned challenges render such an approach for seed generation inefficient. We propose and offer a completely different seed generation that would overcome the aforementioned challenges, and simultaneously reduce the number of seeds using only a single polynomial. To the best of our knowledge, this is the first SMT-based formulation for LFSR reseeding, for which a single process of test generation and seed computation is proposed.

## 3.3 The Proposed Technique

We propose a new SMT-based technique for LFSR reseeding in which we integrate the process of LFSR seed computation together with the test generation process, such that the test vectors generated would already be chained by the derived LFSR seed. Similar to the conventional LBIST technique, we first apply vectors starting from a random seed and drop all the detected faults. To detect the remaining faults, the method finds one or few LFSR seeds that can generate vectors to detect the remaining faults.

In the proposed SMT formulation, we represent every signal of the $k$-identical copies of the circuit as a bit vector of size $k$, as shown in Figure 4.3. Note that this circuit is not an unrolled circuit, since full-scan (could be with the STUMPS architecture [35, 95]) is assumed for the DUT. The $k$ copies of the signal $g1$ would be combined into a single

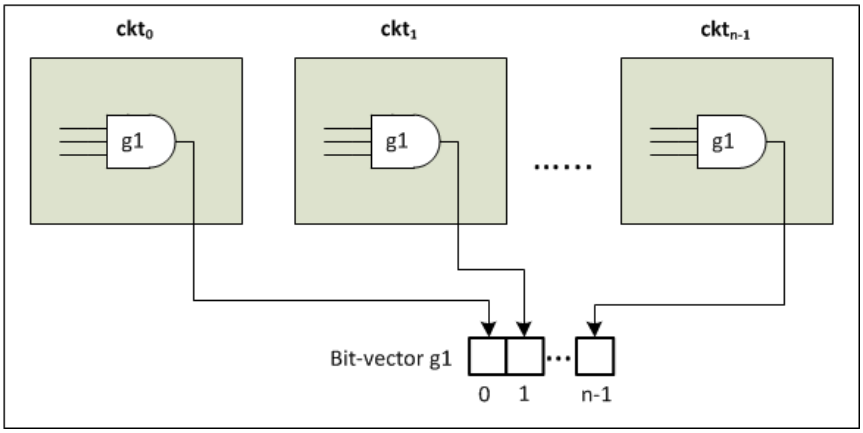



Figure 3.1: Bit-vector Representation of k-identical Copies of the Circuit

bit-vector. This is done for every signal in the circuit. Hence, in a sense, we only have one circuit, rather than $k$ copies, since we are using bit-vectors to represent each signal. We then add the LFSR constraints on the bit-vectors of the primary inputs of this $k$-copy circuit. The complete formula is then equivalent to a $k$-copy circuit with the inputs constrained by the LFSR. The complete illustration is shown in Figure 3.2.

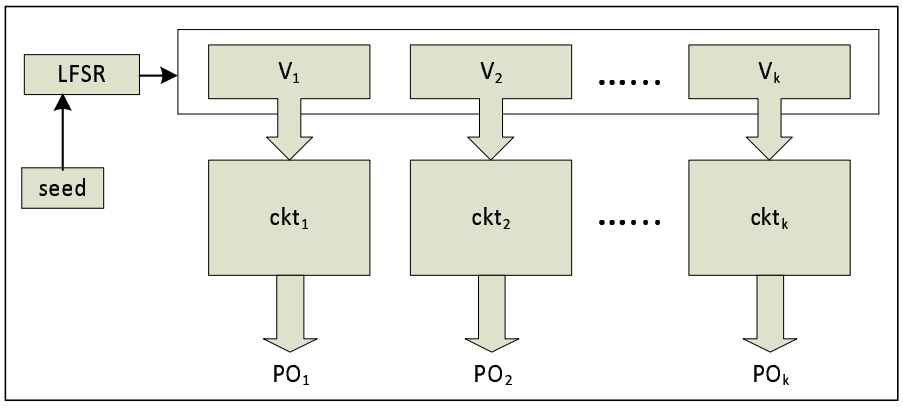



Figure 3.2: Addition of LFSR Constraints to the Setup

Next, the constraints for $m$ target faults are added to the SMT formula. Note that a fault can be detected in any of the $k$ copies, thereby implicitly solving the problem of vector-ordering faced by the previous equation-solving approach. The details for these constraints are described in the following subsections. The conventional way of

detection of these $m$ target faults would be to construct one fault-free circuit followed by $m$ faulty circuits with one distinct fault in each of the faulty circuits. However, this would result in a huge circuit structure and become too cumbersome and expensive to solve in both time and space. Hence, we excite all $m$ faults in one circuit and detect all $m$ faults in another circuit. Consequently, there are only two ($k$-copy) circuits. Note that in both circuits, each signal is a bit-vector of size $k$ as explained earlier. Also, there is no constraint on which fault should be detected by which of the $k$ vectors/cycles.

Note that the constraints for detecting the faults (described subsequently) can be automatically derived for each target fault. Further, SMT is ideal for solving such an instance since we have to target $m$ multiple faults in the $k$-copy circuit simultaneously. Conventional ATPGs target single faults and would not be able to target multiple faults.

### 3.3.1 Excitation Constraints

To excite a fault, we add constraints to ensure that each of the $m$ faults is excited by at least one of the $k$ vectors. In addition, if a fault is at the input of a gate, we also ensure that it is propagated across the corresponding gate. For example, consider an input stuck-at-1 fault at an AND gate $g4$ as shown in Figure 3.3a.

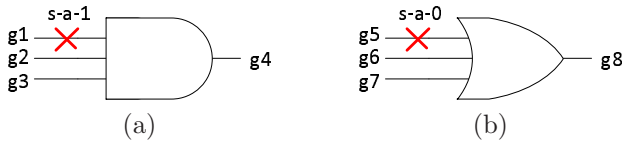



Figure 3.3: Example of Stuck-at Fault at the Input of a Gate

To excite this fault and propagate it across $g4$, we need the following constraint:

$((not\ g1)\ and\ g2\ and\ g3))\ != \ 00...0$

Note that $g1$, $g2$ and $g3$ are all bit-vectors. Thus, bitwise AND of (not $g1$), $g2$, and $g3$ is also a bit-vector. Let $exc\_vec1 = ((not\ g1)\ and\ g2\ and\ g3))$, then whenever $exc\_vec1$ contains a logical 1 bit, it indicates that the corresponding bits in $g1$, $g2$, and $g3$ have to be 0, 1, 1, respectively. In other words, the stuck-at 1 fault at $g1$ would have been excited and propagated across this AND gate. Therefore, we need to constrain this bit-vector to not equal to the all-0 vector to ensure that the stuck fault is excited at least once. In this example, if $exc\_vec1$ is 10101 (in a 5-copy circuit), then the fault is excited by vectors $v1$, $v3$ and $v5$. Similarly, for an input stuck-at-0 fault at an OR gate as shown in Figure 3.3b, the constraint is simply

$((not\ g5)\ or\ g6\ or\ g7))\ != \ 11...1$

One can deduce from the preceding discussion that if any bit in the above bit-vector is 0, the fault is excited by the corresponding vector. This is because a 0-bit indicates that the corresponding bits in $g5$, $g6$, and $g7$ would have been 1, 0, 0, respectively.

For stuck-at faults directly on the gate output, it is sufficient to make sure that the gate output is set to the correct value to excite the fault. For example, consider a stuck-at-0 fault at output of the AND gate $g4$ in Figure 3.3a; then the constraint added for excitation is simply

$(g4\ != \ 00...0)$

Likewise, for the stuck-at-1 fault at output of OR gate $g8$ in Figure 3.3b, the constraint for fault excitation is

$(g8\ != \ 11...1)$

### 3.3.2 Detection Constraints

As explained earlier, we excite each fault in the fault-free $k$-copy circuit and detect the corresponding fault in the other $k$-copy faulty circuit. Consider the fault at an input $g1$ of the AND gate shown earlier. The fault is excited whenever any bit of $exc\_vec1$ is set to 1. (Recall that $exc\_vec1 = ((not\ g1)\ and\ g2\ and\ g3))$) To inject this fault in the faulty circuit, we add a multiplexer at the output of the faulty gate, and set the select line of the multiplexer to be $select\_vec1$, also a bit vector of size $k$. If any bit of $select\_vec1$ is 0 we allow the fault-free value to pass to the output of the MUX, i.e., no fault is injected. But if any bit of $select\_vec1$ is 1, we reverse the value at the gate output, i.e., fault is injected.

From the previous section we know that the fault is excited when any bit of $exc\_vec1$ is set to 1. Hence, we only need to inject the fault whenever the bit of $exc\_vec1$ is 1 and do not inject when $exc\_vec1$ is 0. Thus, the constraint added to ensure this is simply $select\_vec1 == exc\_vec1$.

However, since all $m$ injected faults share the same circuit, it may be possible that some fault is hyperactive, i.e., excited frequently but may not be detected. In such a case, we may not want to inject the fault whenever the fault is excited since they can cause excessive masking. To handle such cases, we change the constraint to be such that the fault is not injected whenever $exc\_vec1$ is 0:

$(not\ exc\_vec1) \rightarrow (not\ select\_vec1)$, which can be re-written as

$(select\_vec1\ and\ (not\ exc\_vec1))\ =\ 00...0$

To ensure detection of the fault, we create a miter circuit. In a miter, the bit-vectors of the corresponding POs are XORed as shown in Figure 3.4.

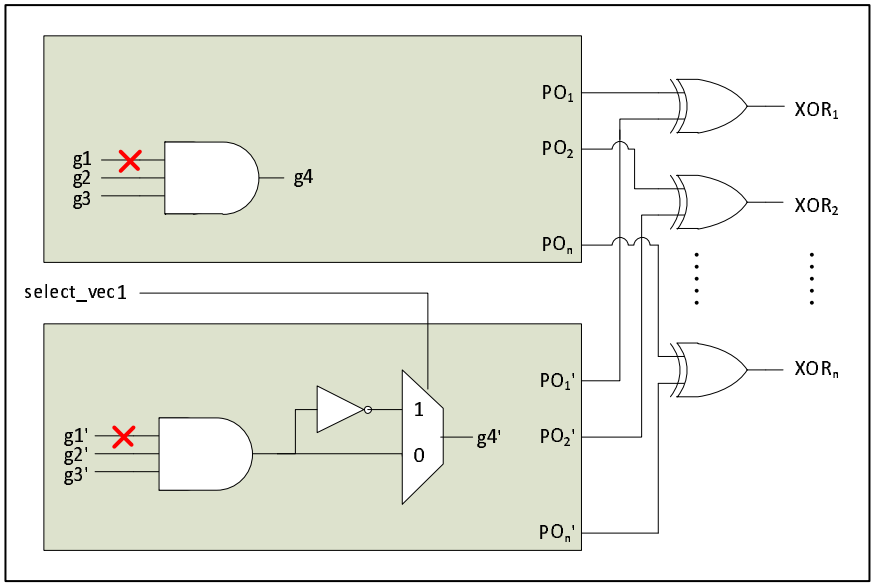



Figure 3.4: Addition of Fault Detection Constraints

If any bit within *select_vec*1 is 1 and any of the XOR gate in the fanout-cone of that fault is also 1, then the fault is considered to be detected. So, if the AND gate had $p$ POs in its fanout-cone, the detection constraint added for detection of this fault is

$((select\_vec1\ and\ XOR_1)\ or\ (select\_vec1\ and$
$XOR_2)\ ...\ (select\_vec1\ and\ XOR_p))\ != \ 00...0$

The excitation and detection constraints for every fault are ANDed. After adding all of the constraints, the final formula is given to the SMT solver, YICES [89]. If the formula is satisfiable, then the SMT solver returns a seed that can generate a $k$-vector sequence that detect all the $m$ target faults. However, if the constraints are unsatisfiable then the SMT solver returns a seed that detects maximum number of faults amongst the $m$ faults.

### 3.3.3 Use of Polarized z-sets and Dominators to Prevent Masking

In the SMT formulation described thus far, there may exist some fault masking, i.e., some faults may not be detected even if the constraints for that fault are satisfied. This is due to the following reasons.

– The approach assumes that whenever $((select\_vec1\ and\ XOR_1)\ or\ ...\ (select\_vec1\ and\ XOR_p)) != 00...0$ is satisfied, the target fault is detected. This is true if the output cones of all $m$ faults are non intersecting. If the same vector excites many faults (that have the same output cones) but detects only one of them, then according to the current formulation all faults having the PO at which the fault is detected in its output cone will be considered detected.

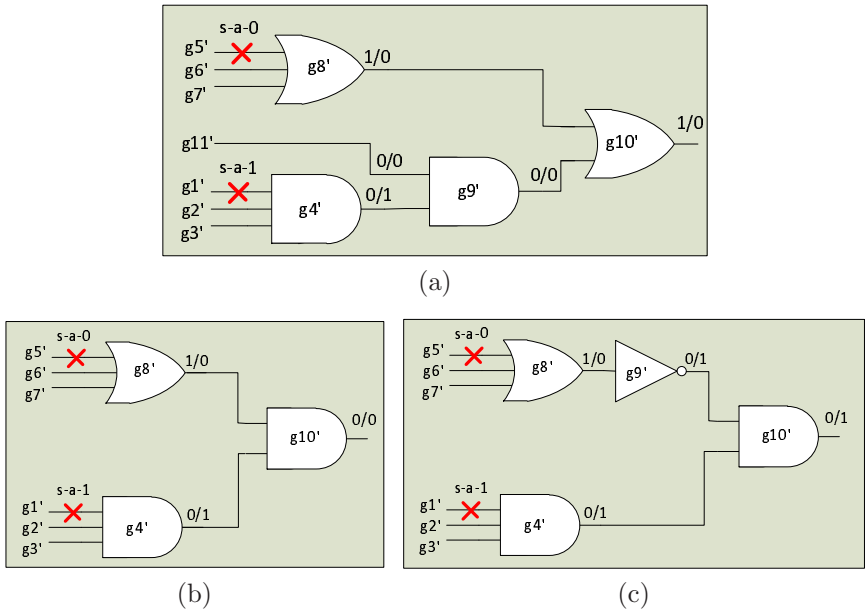



Figure 3.5: Different Scenarios of Fault Masking

For example, consider the circuit fragment shown in Figure 3.5a. Let the s-a-1 fault at input of gate $g4'$ be $f1$ and s-a-0 at input of gate $g8'$ be $f2$. Note that

for a given vector that sets $g11' = 0$, $f1$ should have been blocked. However, if both faults are injected in the same circuit copy and $f2$ has been detected, $f1$ is also considered detected since $g10'$ lies in the fanout of both the faults. Hence, if $f2$ is detected by some vector, then based on the current formulation, $f1$ would also be considered detected (as we only use one faulty circuit for all $m$ faults) even if the vector does not detect $f1$.

– If the presence of one fault is masking the effect of another fault in the faulty circuit, then the fault is considered undetected even if the vector actually detects the fault.

  Consider the circuit fragment shown in Figure 3.5b. It can be seen that because of simultaneous injection of both faults, the propagation of fault $f1$ (input of $g4'$) is blocked at gate $g10'$ due to the 1/0 value at $g8'$ (since we use one faulty circuit for all $m$ faults). Hence, $f1$ is considered undetected due to a fault effect from another fault even if this vector actually detects it.

– Similarly, the presence of one fault may falsely claim detection of another fault. In this case, the fault is considered detected even if the vector actually does not detect the fault.

  Consider a circuit fragment shown in Figure 3.5c. If both faults are injected, then the fault effects of both faults propagate through gate $g10'$. But if only one fault is injected then the propagation of either fault effect would have been blocked at gate $g10$. Hence both the faults may be considered detected even if this vector does not detect any of the two faults.

**$z$-set**: Consider a circuit with $n$ outputs denoted as $z_0$, $z_1$,...,$z_{n-1}$. For a gate $g$ in the circuit, the $z$-set($g$) is the set that contains every output $z_i$ such that there is a directed path in the circuit from $g$ to $z_i$ [96].

**Polarized $z$-set**: Polarized $z$-set of a fault gate $g$ is a set of tuples $\{(z_0, p_0), (z_1, p_1), \ldots (z_n, p_n)\}$ such that $\{z_0, z_1....z_n\}$ is the $z$-set of gate $g$ and $\{p_0, p_1...p_n\}$ are the polarities of faulty values that can be propagated at the respective POs in the $z$-set. The polarity $p_i$ of PO $z_i$ is 0(1) if $D(\bar{D})$ can be propagated to the PO $z_i$ through *all* paths from gate $g$ to $z_i$. The polarity is set to $X$ if both $D$ and $\bar{D}$ can be propagated to the PO $z_i$ through different paths.

For a target fault to be considered as detected, we ensure that the correct value is propagated to the PO where the fault is detected. For example, if the polarized $z$-set of a fault $f1$ is $\{(z_1, 1), (z_3, 0)\}$, then the fault is considered to be detected only if either $\bar{D}$ is propagated to the PO $z_1$ or $D$ is propagated to PO $z_3$. If a $D$ is propagated to PO $z_1$ (and no effect propagated to $z_3$), then it is considered to be the fault effect of some other fault and fault $f1$ is considered to be undetected. For the PO having polarity $X$, the fault is considered to be detected if either $D$ or $\bar{D}$ is propagated to that PO. Thus, an injected fault is considered detected only if a fault effect of the correct polarity is propagated to at least one of the gates in its polarized $z$-set.

If a PO $z1$ has a polarity 1, the constraint added for detection is

$(select\_vec1\ and\ ((not\ z1)\ and\ z\_1)\ !=\ 00...0$

Here $z1$ and $z\_1$ are the corresponding gates in the fault free and faulty copy of the circuit. This constraint ensures that in at least one copy when $select\_vec1$ was 1 (i.e. the fault was excited and injected), the fault free value of $z1$ is 0 and faulty value is 1.

Similarly the constraint added if polarity of $z1$ is 0 is

$(select\_vec1\ and\ (z1\ and\ (not\ z\_1))\ !=\ 00...0$

If the polarity of $z1$ is $x$, then we simply add the XOR constraint explained in the earlier section:

$(select\_vec1\ and\ (z1\ xor\ (not\ z\_1))\ !=\ 00...0$

This constraint ensures that either $D$ or $\bar{D}$ is propagated to $z1$.

To further mitigate the effect of the fault masking, in addition to the polarized z-sets, we add additional constraints to ensure that fault effect of correct polarity propagates to all the dominators of the target fault. The dominators for a signal $g$ is a set of gates that any path from $g$ to any PO must go through. We first find the dominators and their polarities for each fault. Then we add constraints similar to the constraints added for polarized $z$-set for the fault-free and faulty gate variables of each dominator of the faulty gate. This ensures that a fault is considered detected only if the correct fault effect is present at every dominator gate.

We note that the use of polarized $z$-sets and gate dominators does not completely eliminate the possibility of fault masking, but it helps to reduce masking considerably.

### 3.3.4 Equivalent Gates

Since we are using one fault-free and one faulty copy of the circuit, the number of variables needed to represent the gates in the SMT formula is equal to twice the number of gates in the circuit. By finding equivalent gates in the two copies of the circuit, we can enforce that the values of the corresponding gates are the same. This can reduce the time required by the SMT solver to find a satisfying model.

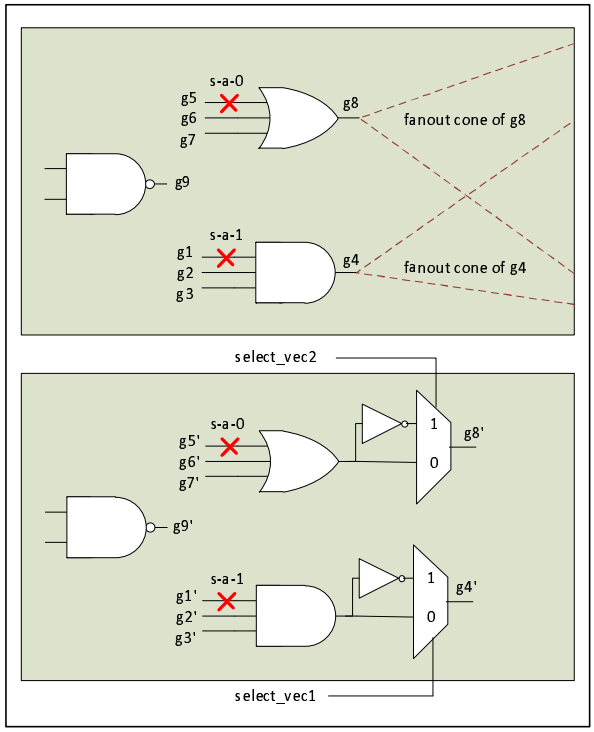



Figure 3.6: Example of Equivalent Gates

In order to find the equivalent gates between the fault-free and faulty circuits, we first find the fanout cones of all the $m$ faults that are injected. We then find the union of all these $m$ fanout cones. The gates which are not in union of these fanout cones are considered outside of the propagation cones. Therefore, they should always be at the same logic values. We replace the variables for these gates in the faulty copy of the circuit with the corresponding values in the fault-free circuit.

For example, Figure 3.6 shows two faults injected in the faulty circuit. These are faults at the inputs of gate $g4$ and $g8$. After finding the union of the fanout cones, we observe that gate $g9$ does not lie in the union. Hence gate $g9$ is unaffected by the injection of the two faults, and the value of $g9$ in the faulty copy of the circuit, $g9'$, can be safely replaced by the fault-free value $g9$. This can reduce the number of variables in the

formula significantly and help to reduce the time required by the SMT solver to find a solution.

### 3.3.5 Benefits of the Proposed Technique

The following summarizes the benefits of our technique:

- Since all faults are taken care of in a single instance of the SMT formulation, there is no question of selection of which faults to target.
- The formulation has no constraint on which fault is detected by which of the $k$ vectors.
- Our technique implicitly takes care of the vector-ordering problem in the previous linear-equation-based approach.
- The formulation is compact as it combines bit-level and word-level reasoning, eliminating the need to explicitly duplicate the bit-level circuit.

## 3.4 Experimental Results

Experiments were performed on Red Hat Linux workstations with Intel® Pentium® D 3.0GHz CPU, 8GB memory and 4 cores. A number of ISCAS85 and full scan versions of ISCAS89 circuits available from [97] were used to demonstrate the effectiveness of the proposed approach. SMT solver YICES [89] was used as the underlying SMT solver. A single LFSR equation was used for every circuit, and the LFSR had the same size as the number of inputs (including flops) of the circuit. The polynomials

were referred from the library by Xilinx [40]. We note that the LFSR size can easily be modified and the overall formulation would be similar.

We first fault simulate 1000 vectors starting from a random seed and drop all the detected faults. For the remaining set of faults we add the excitation and detection constraints. The number of vectors (i.e., '$k$' in our formulation) is set to be 20 greater than the number of faults injected. Table 3.2 shows the results for this technique.

Table 3.2: Experimental Results

| Circuit | Total faults | Faults det by init. random seed | Faults injected | Seeds (include random seed) | Time(s) |
|---|---|---|---|---|---|
| c880 | 942 | 921 | 21 | 3 | 6071 |
| c1355 | 1574 | 1536 | 38 | 4 | 30917 |
| c1908 | 1879 | 1823 | 76 | 12 | 73002 |
| s953f | 1079 | 969 | 110 | 15 | 183809 |
| s1423f | 1515 | 1467 | 48 | 9 | 92346 |
| s1488f | 1486 | 1444 | 42 | 13 | 159219 |

In Table 3.2, Column 1 reports the name of the circuit. Column 2 reports the total number of collapsed faults. Column 3 reports the number of faults detected by the initial random vectors. Column 4 reports the number of faults injected, which is $m$ in our formulation. Column 5 reports the number of seeds needed to detect all the $m$ faults, including the initial random seed. Column 6 reports the cumulative execution time needed to generate all seeds.

According to the results, it can be seen that very few seeds are needed to detect all the faults. For example, for circuit c1355 with a total of 1574 faults, 1536 faults were detected with an initial random seed, leaving 38 faults. To detect all remaining faults, 58 (38+20) copies of the circuit were used. The SMT solver could not detect all 38 faults with a single seed, but 3 seeds were needed and found by the SMT approach.

Together with the initial random seed, there is a total of 4 seeds. Note that in our SMT formulation, we need not detect a fault in each and every vector produced by the seed. Hence, it would be more flexible than the previous vector-chaining approaches. In a sense, it would be trying to chain a number of vectors such that we allow for any arbitrary number of don't-care vectors inserted between every pair of vectors. Finally, the time needed to generate a seed depends directly on the number of faults injected. Hence, the most amount of time is needed to generate the first seed and the time needed reduces after every seed since there would be fewer faults left after each seed. For c1355, 30917 seconds were taken by the SMT solver to compute the 3 seeds. We would also like to point out the state of the SMT solvers is still in its infancy, and we believe the performance of SMT solvers will steadily increase in the coming years.

For other circuits, the effect of fault masking and the fact that some of the final remaining faults are hard to detect would require us to have more seeds. In previous methods, they sometimes would have to allow for multiple polynomials to help chaining the vectors. However, we can avoid having multiple polynomials and still find all seeds necessary to detect all faults.

These results were further improved in [98] wherein the author used an iterative procedure to reduce the time and number of seeds significantly.

## 3.5 Summary

We have proposed a new SMT-based formulation for the LFSR reseeding problem for LBIST. Instead of separate engines to compute the vectors and chaining them, our method unifies the two steps into one, eliminating the need to chain ATPG-generated vectors. Excitation and detection constraints are encoded as SMT constraints, and

Polarized z-sets are proposed as well to enhance the distinguishability between detected faults. Results show potential and promise of the approach, whereby very few seeds are needed to achieve full coverages. The proposed approach also shows that a single LFSR polynomial with few seeds is sufficient to achieve complete coverage.

# Chapter 4

# SMT-based Diagnostic Test Generation

## 4.1 Chapter Overview

In this chapter, we propose diagnostic test pattern generator using a Satisfiability Modulo Theory (SMT) solver. Rather than targeting a single fault pair at a time, the proposed SMT approach can distinguish multiple fault pairs in a single instance. Several heuristics are proposed to constrain the SMT formula to further reduce the search space, including fault selection, excitation constraint, reduced primary output vector, and cone-of-influence reduction. Experimental results for the ISCAS85 and full-scan versions of ISCAS89 benchmark circuits show that fewer diagnostic vectors are generated compared with conventional diagnostic test generation methods. Up to 73% reduction in the number of vectors generated can be achieved in large circuits.

## 4.2 Introduction

The increase in the design complexity and reduced feature sizes have elevated the probability of manufacturing defects in the silicon. These defects could result from shorts between wires/vias, breakage in wires/vias, transistor opens/shorts, etc. Fault diagnosis is the process of finding the fault candidates from the erroneous response. To reduce the number of fault candidates, a test set that is able to distinguish between all faults is highly desirable. A vector that produces different responses for two faults is called a distinguishing vector for those faults. The process of generating such distinguishing patterns is termed as Automated Diagnostic Test Generation (ADTG). Recall from Section 2.3.2 that the goal of ADTG is to generate a set of test patterns that can distinguishable all (detectable) faults that are not equivalent to each other [26]. In addition, we often prefer such a set of vectors to contain a small number of vectors.

Diagnostic test generation was first introduced using a D-algorithm in [99, 100]. Later, instead of exclusively finding test patterns that can distinguish all fault pairs in the circuit, finding distinguishing patterns for only those faults not yet distinguished by the ATPG (automatic test pattern generation) vectors was proposed in [26]. In [29], the authors proposed a method of modifying ATPG tool to generate diagnostic patterns. In [32], the authors defined an *exclusive test* for a pair of faults as a test that detects exactly one fault from a given pair of faults. In [101], an incremental learning-based ADTG flow was proposed which incrementally utilized the information learned during ATPG for ADTG. An output dependent approach for diagnostic test generation was proposed in [102] in which outputs of the circuit are considered one at a time.

Conventional ADTG engines target single fault pairs at a time with Boolean values only. However, targeting single fault pairs may miss opportunities to find vectors that

can simultaneously distinguish multiple pairs. In addition, by representing the circuit only at the bit level, important high-level information is lost. For example, the signals in a data bus or an address bus are represented as unrelated Boolean variables. Such information can be more efficiently represented by packing the related signals together and representing it as a bit-vector. The relations between the variables can then be represented using first order formulas. Such formulas can be efficiently solved by an SMT solver.

We propose a new diagnostic test generation technique using a SMT formulation that can produce a compact diagnostic test set that can distinguish all the distinguishable fault pairs. By using bit-vectors, we can view multiple faulty circuits together as a single circuit, except that each gate is now represented by bit-vectors. The number of variables in the problem instance thus remains the same as a single circuit even though multiple faulty circuits are being considered. Such a formulation allows for efficient handling of multiple fault pairs, and each diagnostic test vector generated will be able to distinguish a large number of fault pairs at each iteration. Subsequently, smaller diagnostic test sets are generated.

## 4.3 Background

Two faults are said to be equivalent if there exists no vector that can distinguish them. Further, two faults are said to be diagnostically equivalent for a given test set if the two faults have the same output responses for all the vectors in the test set. Most of the diagnostic test generation approaches start with a test set. This test set could be randomly generated or from an ATPG. Diagnostic fault simulation is then performed as explained in Section 2.3.1. At the end of the diagnostic fault simulation, $k$ equivalence

classes of faults are produced. All the faults in the same equivalence class are currently diagnostically equivalent with respect to the initial test set. Those faults belonging to different equivalence classes are already distinguished by the initial test set and no ADTG is required for distinguishing them. For example, if faults $f_1$ and $f_2$ come from different equivalence classes, it indicates that the two faults produce different output responses for at least one vector in the initial test set. Next, a fault belonging to an equivalence class of cardinality 1 is distinguished from all other faults since it produces a distinct output from each of the other faults for at least one vector.

A diagnostic test generation is only applied to find distinguishing patterns for the faults that are currently in the same equivalence class. When two faults from the same equivalence class are given to a ADTG engine, either a vector is generated that distinguishes the two faults or the faults are proven to be functionally equivalent. Conventional ADTG techniques target one fault pair at a time. Theoretically, an ADTG creates a miter circuit with the two faults injected in the two copies of the circuit and use an ATPG to justify miter output to 1, as shown in Figure 4.1.

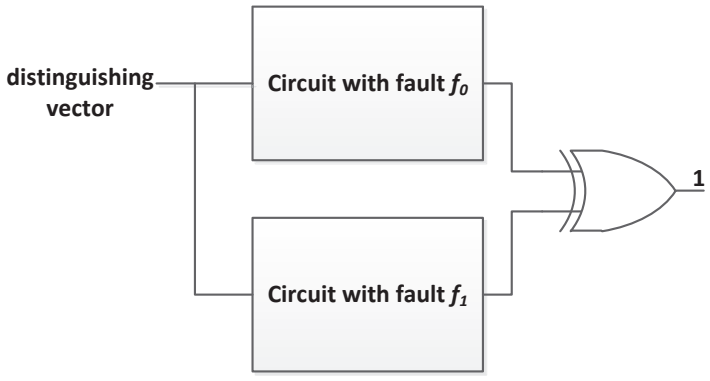



Figure 4.1: Conventional Model for ADTG

If no vector exists that can justify the miter output to 1 (i.e., detect the output fault stuck-at-0), then the two faults are declared to be equivalent. Otherwise, the distinguishing vector that is generated is also simulated to check if it can incidentally

distinguish any other fault pairs. This process is repeated for every pair of faults. At the end of the process, a set of $m$ distinguishing vectors is generated. Generally, $m$ is much greater than the minimal number of distinguishing vectors needed, since the ADTG only considers one pair of faults at a time.

In [64], the authors proposed a technique to convert the previous miter-based ADTG problem to a single-circuit ADTG formulation by a clever use of multiplexers. An example of this technique is shown in Figure 4.2.

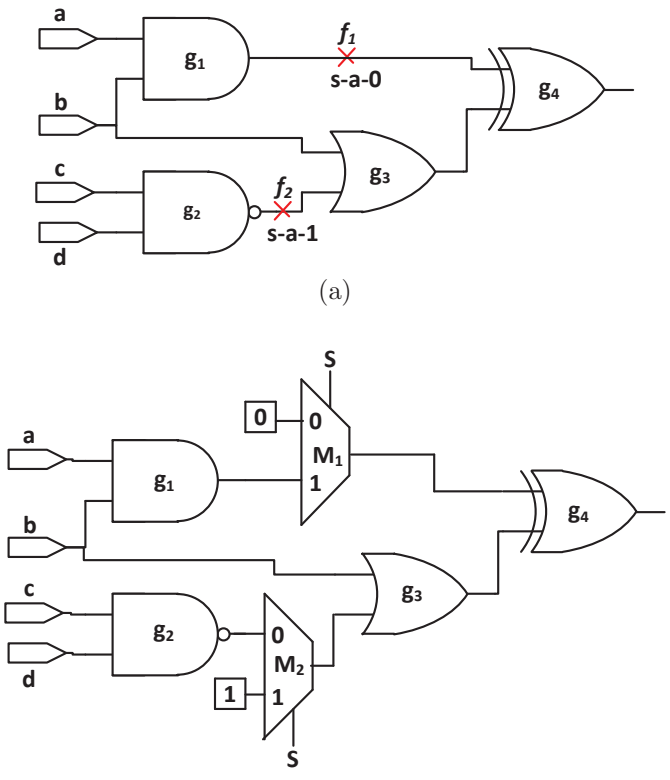



Figure 4.2: Model for Mux-based ADTG

The original circuit is shown in Figure 4.2a. Here, consider $f_1$ and $f_2$ to be the two faults to be distinguished. Two multiplexers ($M_1$ & $M_2$) are added to the original circuit which share the same select line S. The fault-free signal for fault $f_1$ is connected to the input 1 of multiplexer $M_1$ while the faulty value is connected to the other input.

For the other multiplexer $M_2$, the fault-free value and the faulty value are connected to the opposite polarities as that of $M_1$: the fault-free value of fault $f_2$ is connected to input 0 of $M_2$ and the faulty value is connected to input 1 of $M_2$. The modified circuit is shown in Figure 4.2b. If S=0 then fault $f_1$ is injected in the circuit; otherwise, fault $f_2$ is injected in the circuit. A test vector that detects either the *s-a-0* or a *s-a-1* fault at select line S will be a distinguishing pattern for faults $f_1$ and $f_2$. This is because it ensures that the output responses for the vector for the two faults are different. If both *s-a-0* and *s-a-1* fault at S are untestable, then the pair ($f_1$, $f_2$) is undistinguishable.

With the above formulation, this technique no longer needs to create a miter circuit, and instead a single copy of the circuit is sufficient to generate a distinguishing vector for a pair of faults. However, similar to the miter-based ADTG approach, it can only target one fault pair at a time and cannot be easily extended to target multiple fault pairs simultaneously.

All the previous ADTG approaches target one fault pair at time. While one may intelligently fill the incompletely specified distinguishing pattern to distinguish more fault pairs with the same vector, it still remains that only one fault pair is targeted by the ADTG. We propose an entirely new ADTG method which can target multiple fault pairs using a SMT formulation. By targeting multiple fault pairs, our approach can generate smaller diagnostic test sets than conventional methods.

## 4.4 The Ideal SMT-based Diagnostic Test Generation Formulation

After a circuit is fault simulated with an initial test set, the equivalence classes are formed as explained in the previous section. Faults in equivalence classes of cardinality

1 are removed as they are already distinguished from the other faults. Ideally, the goal is to generate diagnostic vectors, each of which can distinguish a maximum number of undistinguished fault pairs. We use an SMT-based formulation to realize this goal. Note that we only need to distinguish a fault from the other faults in the same equivalence class, since faults from different equivalence classes are already distinguished.

Suppose there is only one equivalence class with $n$ undistinguished faults, then we represent every signal of the circuit as a bit-vector of size $n$ as shown in Figure 4.3.

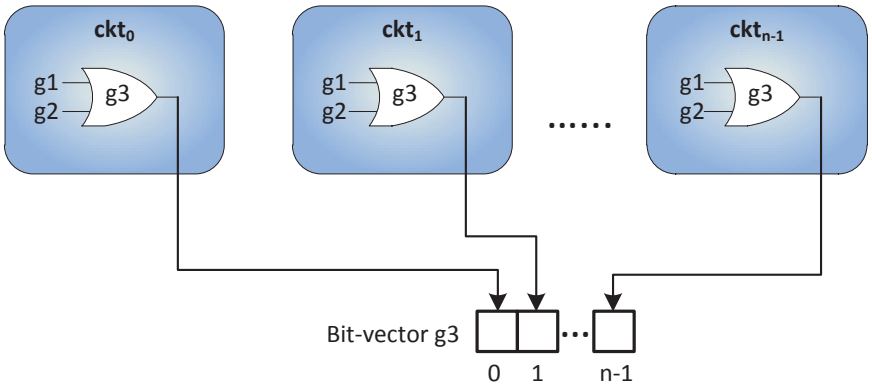



Figure 4.3: Bit-vector Representation of $n$ Copies of the Circuit

Here, $g3$ is an OR gate with inputs $g1$ and $g2$. In the SMT formulation, $g3$ is a bit-vector of size $n$ and every bit of the bit-vector represents the value of gate $g3$ in the corresponding copy of the circuit. Representing the circuit in terms of bit-vectors is efficient because the structural relations among the gates are the same across all the copies and still only $N$ variables are needed for a circuit with $N$ gates. On the other hand, if bit-level representations were used, $N * n$ variables would be needed to represent $n$ copies of the circuit.

To construct the SMT formula for the circuit, each gate is represented as a separate clause in the SMT solver context. Hence, there is still only one copy of the circuit, except that the value of every gate is represented with a $n$-bit bit-vector. For example,

for the OR gate in the previous example the clause added in the SMT context is

$$g3 \ = \ g1 \ OR \ g2$$

Here, both $g1$ and $g2$ are bit-vectors of size $n$, and OR is bitwise OR-operation of $g1$ and $g2$. We then inject one fault in each of the $n$ copies. Each copy will get a different fault from the same equivalence class. No fault-free circuit is needed in our formulation since we are interested only in differentiating among the undistinguished faults. To inject a stuck-at fault, an AND or OR operation is inserted. For example, to inject a stuck-at-1 fault at the first input of gate $g3$ in the $4^{th}$ copy of the circuit, the following constraint is added:

$$g3 \ = \ (g1 \ OR \ 0001000..0) \ OR \ g2$$

Here the constant vector 0001000..0 ensures that the fault is injected only in the $4^{th}$ copy of the circuit. Injecting the fault in this manner retains the relations among the gates across all the copies, i.e., although the additional OR operation is inserted, the fault is injected in only the $4^{th}$ copy.

We want to find a vector that can distinguish as many of undistinguished fault pairs (from the $n$ injected faults) as possible hence, the bit-vectors of the primary inputs of the circuit are constrained to be either all zeros or all ones to ensure that every copy of the circuit receives the same vector. For example, the constraint added for the primary input $PI_1$ is

$$PI_1 \;=\; 0000\ldots00 \;||\; PI_1 \;=\; 1111\ldots1$$

Next, for every copy of the circuit, a primary output (PO) vector is formed by concatenating the bits of primary outputs. For example, if the circuit has three primary outputs $g10$, $g13$ and $g16$, then the PO vector for the first copy of the circuit is formed by concatenating bit 0 of bit-vectors $g10$, $g13$ and $g16$. Similarly the PO vector for other copies is obtained by concatenating the corresponding bits of primary outputs:

$$PO_i \;=\; concatenate(g10[i],\; g13[i],\; g16[i])$$

Thus, the size of each PO vector is equal to the number of primary outputs in the circuit. In the running example, the constraint added to ensure that the fault injected in the $i^{th}$ copy of the circuit is distinguished from the fault injected in the $j^{th}$ copy, is

$$PO_i \;\neq\; PO_j$$

In other words, this constraint makes sure that the output signatures for the $i^{th}$ faulty circuit and $j^{th}$ faulty circuit are different, where $i \neq j$. Thus, the vector generated either detects only one of the two faults or detects both the faults but with different output signature. For $n$ faults, there would be $n(n-1)$ such PO constraints.

We can extend the above formulation to handle multiple equivalence classes as well. Suppose there are $c$ classes, each class having $n_i$ undistinguished faults, $0 \leq i \leq c-1$. If the sum of all $n_i$ faults among all $c$ classes is $n$, then we can likewise inject all $n$ faults into the formulation. However, for the output constraints, we no longer need to insert $n(n-1)$ constraints, since fault pairs from different classes are already distinguished

by the initial test set. Instead, we just need to insert $n_i(n_i - 1)$ constraints for each class, resulting in a total of $\sum_{i=0}^{c-1} n_i(n_i - 1)$ constraints. The complete illustration of the formulation is shown in Figure 4.4.

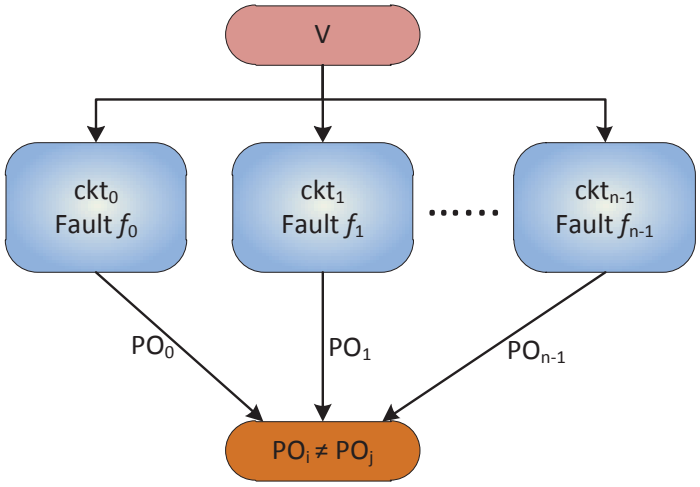



Figure 4.4: The Complete SMT-based ADTG Model

Faults $f_0$ to $f_{n-1}$ are injected in separate copies of the circuit $ckt_0$ to $ckt_{n-1}$, respectively. All the copies receive the same input vector $V$. Note that we show the explicit $n$ copies for ease of illustration, but in the actual SMT formulation only one copy of the circuit is needed, where the value for each gate is a bit-vector.

In the formulation described above, all $n$ faults of the circuit are injected and combined into a single SMT formula. This entire formula is then given to the SMT solver. The solver tries to satisfy the formula, i.e., it tries to find a single vector that can distinguish all the undistinguished fault pairs among the $n$ injected faults. But such a vector generally does not exist since the conditions to detect different faults may conflict. Hence, the SMT solver would generate a vector that satisfies the maximum number of clauses, i.e., it generates a vector that distinguishes the maximum fault pairs.

If the vector generated by the SMT solver distinguishes $m$ fault pairs, then these $m$ faults are declared pair-wise distinguished and the equivalence classes are updated

accordingly. The next iteration is performed with the remaining undistinguished fault pairs. This process continues until all the fault pairs are distinguished from one another or are proved equivalent. Such a formulation does not guarantee to achieve the smallest diagnostic test set size, since the solver greedily tries to distinguish as many fault pairs in each iteration. Nevertheless, it is expected to produce a smaller diagnostic test set compared to existing methods.

## 4.5 The Practical SMT-based Diagnostic Test Generation Formulation

The formulation described in the previous section considers all the undistinguished fault pairs left after initial fault simulation together in the first iteration. Even for medium-sized circuits, the number of undistinguished fault pairs initially can be large, and the SMT solver may take a long time to find the vector that distinguishes the maximum of these faults. Hence, though ideal, it makes the formulation unsuitable for practical use.

To make the formulation practical, we first set a limit $k$ on the number of target faults considered in each iteration. After each iteration we fault simulate the distinguishing vector generated and update the equivalence classes. Any fault having an equivalence class of cardinality 1 is not considered in subsequent iterations.

### 4.5.1 Fault Selection

Once $k$ is decided, the next task is to choose the $k$ faults from the set of undistinguished faults. This can be done using either of the following two approaches.

**One fault-pair from each equivalence class**

In this approach, we choose one fault pair to be distinguished from $k/2$ different equivalence classes. In other words, we pick 2 faults from each of the first $k/2$ equivalence classes. If there are less than $k/2$ classes, then we choose more faults from each class such that $k$ faults are added to the list. This selection scheme aims to generate a vector that can distinguish multiple fault pairs from different equivalence classes.

**All faults from same class**

In this approach, we choose all the $k$ faults from the first equivalence class. If the first equivalence class has a size less than $k$, then we add more faults from the next class(es) to reach $k$ faults. This selection scheme aims to generate a vector that can distinguish as many faults from the same class as possible.

### 4.5.2 Excitation Constraint

To distinguish any fault pair, a vector is needed that either (1) detects one fault but not the other or (2) detects both faults with different output signatures. Hence, in order to distinguish a fault pair, at least one of the two faults must be detected. For example, consider the circuit shown in Figure 4.2a. Here the two faults to be distinguished are *s-a-0* at output of gate $g1$ and *s-a-1* at output of gate $g2$. In order to distinguish these two faults, at least one of them has to be detected which implies that at least one of them has to be excited. To excite fault $f_1$, $g1$ should be 1 and to excite $f_2$, $g2$ should be 0. For each fault pair, we add a constraint to ensure that at least one of the faults is excited. In the running example, if fault $f_1$ is injected in the $4^{th}$ copy of the circuit and fault $f_2$ is injected in the $6^{th}$ copy of the circuit, then the excitation constraint

added will be

$$g1[4] = 1 \;||\; g2[6] = 0$$

Addition of the excitation constraint helps in pruning the search space by ensuring at least one fault of every pair is excited, and it thus reduces the time taken by the SMT solver to generate a distinguishing vector.

### 4.5.3 Reduced PO Vector

**$z$-set**: Consider a circuit with $o$ outputs denoted as $z_0$, $z_1$,...,$z_{o-1}$. For a gate $g$ in the circuit, the $z$-set($g$) is the set of primary outputs to which $g$ can reach [96].

Recall that in the ideal SMT-based ADTG formulation, the constraint added for distinguishing the faults injected in the $i^{th}$ copy of the circuit and $j^{th}$ copy of the circuit is

$$PO_i \neq PO_j$$

Here, each PO vector is of size equal to the number of primary outputs in the circuit. As long as at least one bit differs between $PO_i$ and $PO_j$, it suffices to conclude that the faults injected in the $i^{th}$ copy and $j^{th}$ copy are distinguished.

From the definition of $z$-sets it is clear that a fault at gate $g$ can only propagate to the primary outputs in $z$-set($g$). Hence, it suffices to include only the POs defined in the $z$-sets of the union of the faults being distinguished when making the output comparisons, thereby reducing the search space.

Consequently, the size of the PO vector of each copy of the circuit will be equal to the union of the $z$-set of the fault gates of all the faults in the current iteration that belong to the same equivalence class. The constraint added for distinguishing the faults injected in the $i^{th}$ copy & $j^{th}$ copy of the circuit is still the same. Only the size of PO vector will change. Reducing the size of the PO vector helps to reduce the search space and increases the speed of diagnostic test generation.

### 4.5.4 Cone-of-Influence Reduction

To distinguish a fault pair, at least one of the two faults has to be detected. For detecting a fault $f$, only the *region* that contains all the gates that can potentially influence the excitation and/or propagation of $f$ need to be considered.

**Cone of Influence (COI)**: For a fault $f$ at input/output of gate $g$, the Cone of Influence of $f$ is defined as the union of gates in the fan-in cones of the gates in $z$-set($g$) [103].

Any gate that is outside the COI of fault $f$ cannot affect the excitation and propagation of $f$ and hence can be ignored during test generation for fault $f$.

To reduce the number of gates that need to be considered, i.e., the number of variables in the SMT formula, we first find the COI of all the $k$ faults considered in the current iteration. We then take the union of the COI of all these faults. All the gates that are outside the union of COI have no influence on excitation or propagation of any of the faults in the current iteration. Hence, we exclude the clauses of these gates from the SMT context.

For example, in Figure 4.5, $f_1$ and $f_2$ are the two faults to be distinguished. Regions $FO_1$ and $FO_2$ (regions bounded by dotted blue and red lines respectively) are the

fanout cones of $f_1$ and $f_2$. Similarly, regions $COI_1$ and $COI_2$ (regions bounded by solid blue and red lines respectively) are the cones of influence for $f_1$ and $f_2$. All the gates outside $COI_1 \cup COI_2$ (i.e. the shaded region) can be neglected while generating a distinguishing vector for $f_1$ and $f_2$.

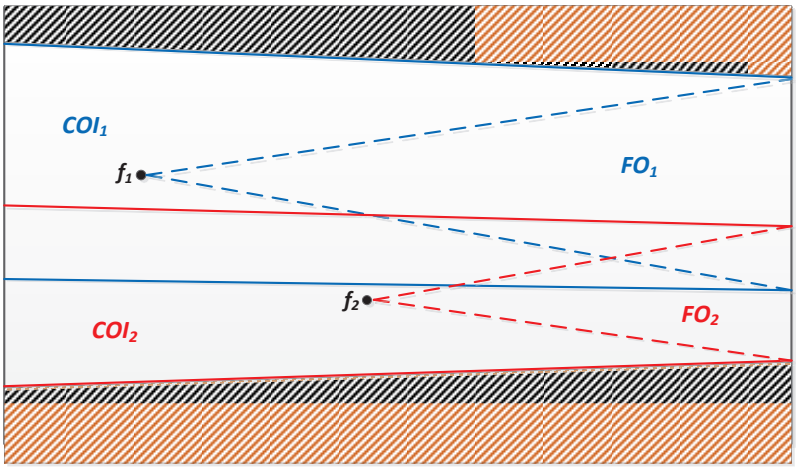



Figure 4.5: Example of Cone of Influence

Excluding the clauses of the gates outside the COI can potentially reduce the formula size, especially for large circuits. For many cases, the number of gates in the COI can consist of only a small percentage of the total gates in the circuit.

## 4.6 Experimental Results

Experiments were performed on Red Hat Linux workstations with Intel® Pentium® D 3.0GHz CPU, 8GB memory and 4 cores. The ISCAS85 and full-scan versions of ISCAS89 benchmark circuits were used to demonstrate the effectiveness of the proposed technique. SMT solver Yices [89] was used as the underlying SMT solver.

Here, we want to compare the number of diagnostic vectors generated by our approach against conventional method of considering only single fault pairs at a time. The initial

test set used was Mintest test sets [23]; however, other test sets can be used. We note that a smaller initial test set that can detect all detectable faults is likely to have larger equivalence classes. Mintest vectors are compact test sets that detect all detectable faults for each circuit. We simulated the circuits with Mintest test sets and constructed the equivalence classes for each circuit. All the faults that are already distinguished by the Mintest vectors are dropped, and ADTG is performed on the remaining faults using three different approaches by considering (1) Single fault pair, (2) One fault-pair from each equivalence class and (3) All faults from the same class. For approaches (2) and (3), the number of faults in each iteration ($k$) was set to 10.

Table 4.1 shows the results for all three approaches. Column 1 lists the circuits. Column 2 reports the number of Mintest vectors for the corresponding circuit. Columns 3 and 4 report the number of diagnostic vectors generated and the time taken when a single fault-pair is targeted in each iteration. Columns 5, 6 and 7 report the number of diagnostic vectors generated, time and the % reduction in diagnostic test set achieved over the single fault-pair approach by considering one fault-pair from each equivalence class. The last three columns report the corresponding numbers when all faults were chosen from the same class. The smallest test sets (and the corresponding greatest % reductions) are highlighted in **bold**.

The results clearly show that significant reduction in test set is achieved by our approach, especially for larger circuits by considering multiple fault pairs in each iteration. For example, for circuit s38417f with 68 initial Mintest vectors, an additional 505 vectors were generated by targeting single fault pairs at a time. But with our approaches (2) and (3), only 135 and 144 vectors are generated. This is a reduction of 73.26% and 71.4%, respectively. Overall the % reduction achieved using approach (2) and (3) are 56.63% and 55.79%, respectively.

Table 4.1: Results for SMT-based Diagnostic Test Generation

| Circuit | Mintest Vectors | Single fault-pair | | Multiple Fault Pairs **(k=10)** | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | One fault-pair from each class | | | All faults from same class | | |
| | | # vec | time(s) | # vec | time(s) | % reduction | # vec | time(s) | % reduction |
| c432 | 27 | 39 | 6.8 | 31 | 135 | 20.5 | **27** | 128 | **30.7** |
| c499 | 52 | 15 | 1.7 | **5** | 71 | **66.6** | **5** | 66 | **66.6** |
| c880 | 16 | 25 | 2.3 | **8** | 41 | **68.0** | 10 | 34 | 60.0 |
| c1355 | 84 | 14 | 4.4 | 7 | 96 | 50.0 | **6** | 85 | **57.1** |
| c1908 | 106 | 27 | 41.0 | 20 | 749 | 25.9 | **19** | 833 | **29.6** |
| c2670 | 44 | 72 | 105.0 | 40 | 1718 | 44.4 | **36** | 1490 | **50.0** |
| c3540 | 84 | 57 | 1173.0 | **41** | 4210 | **28.0** | 43 | 3744 | 24.5 |
| c5315 | 37 | 125 | 180.0 | **75** | 3689 | **40.0** | **75** | 3456 | **40.0** |
| c7552 | 73 | 110 | 92.0 | 41 | 2556 | 62.7 | **39** | 2223 | **64.5** |
| s1423f | 20 | 38 | 26.0 | **19** | 335 | **50.0** | 20 | 327 | 47.3 |
| s1488f | 101 | 37 | 1.5 | **35** | 23 | **5.4** | **35** | 22 | **5.4** |
| s1494f | 100 | 43 | 1.6 | 42 | 27 | 2.3 | **41** | 24 | **4.6** |
| s5378f | 97 | 61 | 15.0 | **34** | 311 | **44.2** | **34** | 287 | **44.2** |
| s9234f | 105 | 267 | 197.0 | **112** | 3588 | **58.0** | 119 | 3415 | 55.4 |
| s13207f | 233 | 57 | 46.0 | **23** | 675 | **59.6** | 25 | 496 | 56.1 |
| s15850f | 95 | 89 | 764.0 | **50** | 21424 | **43.8** | 52 | 28595 | 41.5 |
| s35932f | 12 | 92 | 98.0 | 43 | 122 | 53.26 | **42** | 115 | **54.3** |
| s38417f | 68 | 505 | 581.0 | **135** | 28879 | **73.26** | 144 | 19393 | 71.4 |
| s38584f | 110 | 250 | 113.0 | **73** | 494 | **70.8** | 78 | 448 | 68.8 |
| TOTAL | 1464 | 1923 | 3449.3 | **834** | 69143 | **56.63** | 850 | 65181 | 55.79 |

Smallest number of vectors generated (and greatest % reductions) are highlighted in **bold**.

By comparing approaches (2) and (3), it can be seen that they generated similar numbers of diagnostic vectors. However, approach (3) is generally a little faster in most of the cases. For example, for circuit c3540, approach (2) takes 4210 seconds to generate 41 vectors while approach (3) generated 43 vectors in 3744 seconds. This may be attributed to the fact that the faults from the same equivalence class have a greater number of similar necessary assignments, which can help to reduce the search space.

We note that the execution time by considering multiple fault pairs is longer than conventional methods of targeting one fault pair at a time. However, SMT solvers are still in their infancy and we believe the run times will improve as the solvers mature.

## 4.7 Summary

We proposed a new SMT-based diagnostic test pattern generation technique for combinational circuits that can target multiple fault pairs in single iteration. A number of heuristics have been proposed to reduce the search space. Experimental results show that significant reductions in diagnostic test sets are achieved using the proposed technique over the conventional method of targeting single fault pairs. Up to 73% reduction in the number of vectors generated can be achieved in large circuits.

# Chapter 5

# Test Generation of Circuits with Embedded Memories

## 5.1 Chapter Overview

In this chapter, we propose a method for automatic test pattern generation for circuits containing non-scan embedded memories using a Satisfiability Modulo Theory (SMT) formulation. Modern circuits can contain many small embedded memories for which it would be impractical to insert MBIST or scan every embedded memory. By modeling the entire circuit using SMT, the proposed method is able to detect faults in the circuits by exciting/propagating each target fault through both the logic and embedded memories. Constraints are automatically added to ensure the correct propagation of fault effects, whether it is through the address or data buses, or both. Experimental results show the effectiveness of the proposed method; the time needed for test generation was small and was scalable with increase in the size of memory. The formulation can also be extended for circuits with multiple embedded memories

## 5.2 Introduction

One of the important challenges in testing modern SOCs is the presence of small embedded memories. These memories are too small to employ memory BIST, and large enough that scanning them becomes unattractive. In particular, making these embedded memories scanable would increase both the area overhead and test application time. Likewise, employing MBIST for an embedded memory requires an additional MBIST controller, which would not be practical if the embedded memory is small.

In order to detect some faults in the memory or the logic around the memory, the target fault may need to be excited with a specific value in the embedded memory or the fault-effect may need to be propagated through the memory. Conventional gate-level automatic test pattern generators (ATPGs) and Satisfiability (SAT) solvers work on instances containing Boolean values only. When one flattens the embedded memory to the gate level, even for small memories, the number of memory elements can grow to be very large. In addition, if the circuit is represented only at the Boolean level, important high-level information is lost. For example, the signals in a data or address bus are now represented as unrelated Boolean variables. Similarly, the flip-flops (FFs) corresponding to a single memory location should only be accessed with a specific address. Without such knowledge, the ATPG can face tremendous challenges if it assigns values to FFs that cannot be read out in the same cycle. Thus, without the high-level knowledge regarding the address logic and the corresponding word-level data, huge numbers of backtracks can often result during the search. Hence, it is extremely difficult to generate test vectors for circuit with memories if they are not scannable. We believe that portions of the circuit, such as the embedded memory and surrounding logic, can be more efficiently represented by packing the related signals together and representing it as a bit-vector. The relations among the variables can

then be represented using first-order formulas which can be efficiently handled by an SMT solver. SMT can model circuits at a higher level of abstraction like RTL or in mixed modes where few blocks are represented at the RTL level while other blocks are represented at the bit level. In such a setting, SMT can efficiently model circuits containing FIFOs, embedded memories with the theory of arrays and uninterpreted functions.

We propose a new test generation method for circuits containing embedded memories using a SMT formulation. By using theory of bit-vectors and theory of arrays, we model the embedded memories at a higher level while keeping the logic around the memory at the gate/Boolean level. Such a formulation allows for efficient excitation of faults and propagation of fault-effects through the memory. Subsequently, to some degree, the test generation for the entire circuit can still be regarded as a combinational, rather than sequential, ATPG. In addition, the proposed technique is also able to generate test vectors for faults within the memory. Constraints are automatically added to ensure the correct propagation of fault effects, whether it is through the address or data buses, or both. The proposed technique thus eliminates the need for making the embedded memory fully controllable as in scan. Furthermore, by eliminating scan it reduces both the associated hardware overhead and test application time.

To the best of our knowledge, this is the first approach for test generation of circuits where SMT has been used to model embedded memories. Since there are no readily available benchmarks for circuit with embedded memories, we created our own circuits by surrounding the embedded memory with existing ISCAS85 blocks. The results show that the time needed for test generation was small and was scalable with increase in the size of memory.

## 5.3 Model for Circuit with Embedded Memory

As there are no readily available benchmarks for circuits with embedded memories, we first created our own circuits by inserting a memory between two ISCAS benchmark circuits as shown in Figure 5.1.

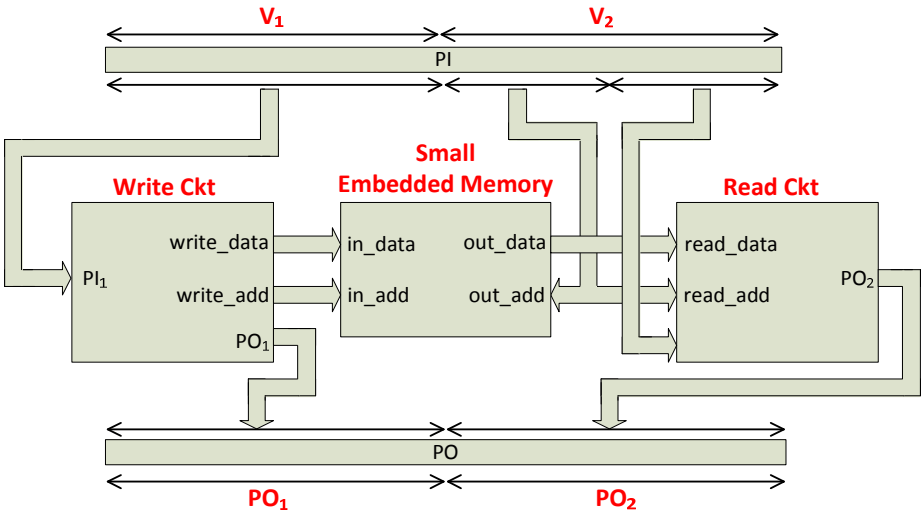



Figure 5.1: Circuit with Embedded Memory

Two ISCAS circuits were used. One combinational logic is used as the write circuit that writes into the memory and another combination logic is used as the read circuit that reads from the memory.

Let the size of the embedded memory be $m \times n$, where $m$ is the width of address bus and $n$ is the width of data bus of the memory. Thus, there are $2^m$ $n$-bit words in the memory. All the primary inputs of the write circuit are considered to be primary inputs in the combined circuit. The first $n$ primary outputs of the write circuit are connected to the write data bus of the memory. The next $m$ primary outputs of the write circuit are connected to the write address bus of the memory. The remaining primary outputs of the write circuit, if any, are left as additional primary outputs. On the other side of the embedded memory, the first $n$ primary inputs of the read circuit

are driven by the read data bus of the memory. The next $m$ primary inputs of the read circuit are connected to the read address bus by adding an additional fanout. The remaining primary inputs of the read circuit, if any, are left as primary inputs. All the primary outputs of the read circuit are considered to be primary outputs of the combined circuit. Note that the circuit can consist of multiple embedded memories as well, with a similar construction.

The values written to the embedded memories will also be remembered in the model, which may become useful during subsequent test generation sessions. This is needed in two scenarios: first, when the fault-effect propagates to the write-address bus, the fault-free and faulty circuits would thus read from different locations. Thus, we would need to know the value in both locations. Second, if the circuit contains two or more embedded memories, it may be necessary to use previously stored values in some of the memories to help propagate the fault-effect.

## 5.4 SMT Formulation for Test Generation

Figure 5.2 illustrates the setup for our SMT-based formulation for test generation. First, the entire circuit with embedded memories is duplicated to form a fault-free and a faulty copy of the circuit. While *PI* is tied to both copies, in our formulation, we distinguish the primary inputs of the two circuits by calling the inputs of the fault-free circuit as *PI* and the inputs of the faulty circuit as *PI'*. If the target fault is in the write circuit, then it may be necessary that the fault-effect be propagated across the embedded memory and the read circuit, before it can be detected at the primary outputs of the read circuit. In such a case, the fault-free and the faulty copies of the

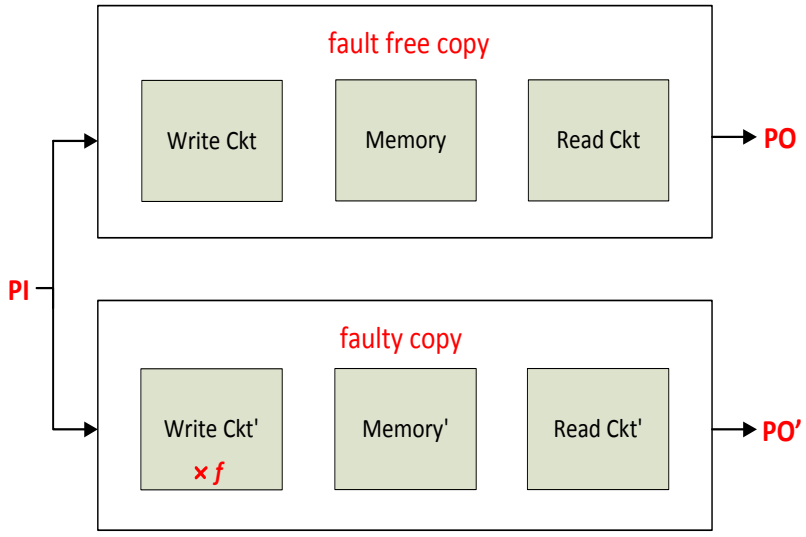



Figure 5.2: Miter Circuit with Embedded Memory

read circuit will read different values from the embedded memory since the faulty copy of memory contains the faulty value of the write circuit.

The constraints to the SMT formula are added as follows: To ensure that both the fault-free and faulty copies of the circuit receive the same vector, the constraint is

$PI \ == \ PI$'

Next, to ensure that the fault is propagated to the combined circuit output, the constraint added is

$PO \ \neq \ PO$'

Note that the fault may be detected at one of the POs of the write circuit that were not connected to the embedded memory ($PO_1$ in Figure 5.1). In this case, the fault-effect does not need to propagate through the memory. However, in the general case, it needs to be detected at one of the POs of the read circuit (i.e., the fault propagates through the memory).

To propagate the fault-effect through the read circuit we need to ensure that the read

circuit reads the data from the *same* memory location that was written by the write circuit, especially in the fault-free copy of the circuit. The constraint added to ensure this is

$$faultfree\_write\_address == faultfree\_read\_address$$

However, the same constraint is not valid in the faulty copy of the circuit because if the fault-effect propagates to the address bus, then the write and read circuits may address different memory locations. Details of how we model these cases will be given in Section 5.4.3.

The memory module is implemented using the theory of arrays. Different SMT solvers have different constructs to support theory of arrays. We chose Z3 [104] from Microsoft research as the underlying SMT solver. Z3 provides store and select constructs to write and read values in an array respectively. Details related to implementation of array decision procedures in Z3 are available in [105].

An example of modeling an 8×16 bit memory is shown in Figure 5.3. First, we declare all memory variables with domain of bit-vector of size equal to the address bus size and range of bit-vector of size equal to the data bus size. We then declare the variables for fault-free and faulty values of write_address, write_data, read_address and read_data. The memory is then initialized to all-0. The store construct is then used to write the write_data to write_address in the fault-free copy. The updated memory after the write is assigned to a new variable faultfree_mem1. Similarly, the faulty value is written to the faulty address and the updated memory is assigned to new variable faulty_mem1. The select construct is used to read the fault-free and faulty values from the corresponding memories. The rest of the logic is represented using Boolean variables. Test generation is performed for all the stuck-at faults similar to a SAT-based ATPG, but instead of a SAT solver we use an SMT solver.

```
(declare-const init_mem (Array (_ BitVec 8) (_ BitVec 16)))
(declare-const faultfree_mem1 (Array (_ BitVec 8) (_ BitVec 16)))
(declare-const faulty_mem1 (Array (_ BitVec 8) (_ BitVec 16)))
(declare-const faultfree_write_address (_ BitVec 8))
(declare-const faulty_write_address (_ BitVec 8))
(declare-const faultfree_write_data (_ BitVec 16))
(declare-const faulty_write_data (_ BitVec 16))
(declare-const faultfree_read_address (_ BitVec 8))
(declare-const faulty_read_address (_ BitVec 8))
(declare-const faultfree_read_data (_ BitVec 16))
(declare-const faulty_read_data (_ BitVec 16))
(assert (= init_mem ((as const (Array (_ BitVec 8) (_ BitVec 16))) 0)))
(assert (= faultfree_mem1 (store init_mem faultfree_write_address
                                                    faultfree_write_data)))
(assert (= faulty_mem1 (store init_mem faulty_write_address faulty_write_data)))
(assert (= faultfree_read_data (select faultfree_mem1 faultfree_read_address)))
(assert (= faulty_read_data (select faulty_mem1 faulty_read_address)))
```

Figure 5.3: Modeling Memory using SMT

After test generation for each target fault, the updated memory at the end of the current iteration is used as the starting memory content (instead of init_mem) for the next iteration. Thus, the values written to the memory are carried over for next test generation. This is necessary for detecting the faults whose fault-effects affect the address of the write address bus. This will be explained in detail in Section 5.4.3. To improve the speed of the SMT solver, we also add the clauses for the necessary assignments and active clauses for the target fault, explained next.

### 5.4.1 Necessary Assignments

For every fault, $f$, in the fault list, we compute the set of global necessary assignments needed to detect $f$. This is done in a manner similar to MUST [106], in which a list of necessary assignments is kept for every signal in the circuit. This could be conducted very efficiently using static logic implications, and the time for generating

the necessary assignments is negligible compared to the ATPG cost. The necessary assignments obtained using this technique are global and are larger than those obtained by simple forward and backward implications as in [102].

### 5.4.2 Active Clauses

In order to detect a fault, it is necessary that there exists at least one sensitizable path from the fault site to one of the primary outputs. Additional clauses can be added to ensure that at any instant during the search, there exists a potentially sensitizable path which can propagate the fault-effect. Adding such clauses was proposed in [15] and these clauses were termed as active clauses and the sensitizable path to be found was termed as active path. These clauses reduce the search space and make test generation faster. It was reported in [15] that active clauses are especially useful for proving faults untestable quickly.

In order to add these active clauses, new variables $Act_i$ are added in the SMT context for the faulty signal and every gate in its fanout cone. These new variables are termed as active variables and every active variable $Act_i$ corresponds to gate $G_i$. A gate is active means that there exists a fault-effect at the gate. For example, consider the circuit shown in Figure 5.4.

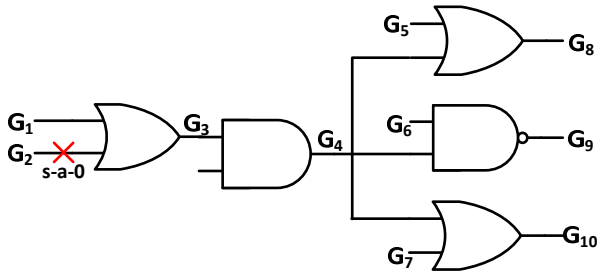



Figure 5.4: Circuit for Active Clauses

If gate $G_2$ is active, it means that the fault-free value ($G_2$) and the faulty value ($G'_2$) are different. This can be specified in the SMT context by adding the clause

$Act_2 \implies (G_2 \neq G'_2)$

Similarly, for each gate $G_i$ in the fanout cone of the fault gate, we add the clause

$Act_i \implies (G_i \neq G'_i)$

These clauses indicate that if a gate is active, the fault-free and faulty values are required to be different.

In this example, gate $G_2$ has a single fanout. Thus, the fault-effect from $G_2$ must propagate through $G_3$ in order to reach a primary output (i.e., $G_3$ must also be active). To ensure that the fault-effect propagates to $G_3$ the clause added is

$Act_2 \implies Act_3$

Similarly, for each gate $G_x$ that feeds a single gate $G_y$, we add the clause

$Act_x \implies Act_y$

Essentially, this means that if $G_x$ is active, $G_y$ is also active, i.e., if the fault-effect reaches the input of a gate on the active path it also propagates to its immediate output.

For gates with multiple fanout gates, such as $G_4$ in Figure 5.4, the fault-effect from $G_4$ can propagate to the primary output through at least one of the fanout gates namely $G_8$, $G_9$ and $G_{10}$ (i.e., at least one of the fanout gates must be active). This is specified by adding the clause

$Act_4 \implies (Act_8 \vee Act_9 \vee Act_{10})$

In general, for gates with multiple fanouts, $G_x$ with outputs $G_y$ and $G_z$, we add the clause

$Act_x \implies (Act_y \vee Act_z)$

In other words, if $G_x$ is active, then at least one of $G_y$ or $G_z$ must be active.

Addition of the above clauses enforces that there exists a sensitizable path (i.e., consecutive active gates) from the faulty gate to a primary output.

Consider the circuit in Figure 5.4. The fault to be detected is $G_2$ stuck-at 0. The active clauses for this fault would be

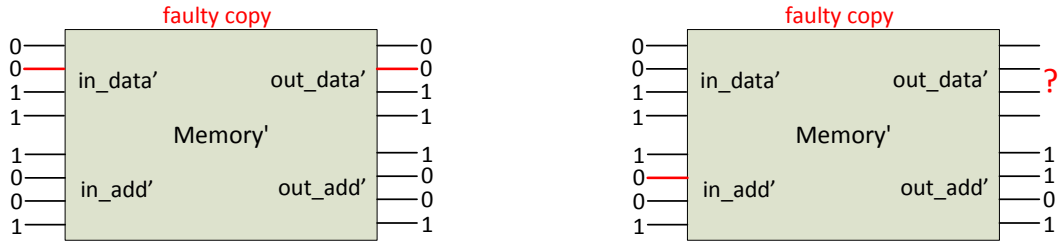
$(Act_2 \implies (G_2 \neq G_2')) \wedge (Act_2 \implies Act_3) \wedge$
$(Act_3 \implies (G_3 \neq G_3')) \wedge (Act_3 \implies Act_4) \wedge$
$(Act_4 \implies (G_4 \neq G_4')) \wedge$
$(Act_4 \implies (Act_8 \vee Act_9 \vee Act_{10})) \wedge$
$(Act_8 \implies (G_8 \neq G_8')) \wedge$
$(Act_9 \implies (G_9 \neq G_9')) \wedge$
$(Act_{10} \implies (G_{10} \neq G_{10}'))$

### 5.4.3 Propagating Fault Effect through the Memory

For the fault in the write circuit, the fault-effect has to pass through the memory. Depending on the fanout cone of the fault gate, the fault-effect may affect a write address line, a write data line, or both.

#### Fault Effect at Data Line ONLY

If the fault-effect reaches a write data line only then the faulty value will be written to the correct memory location as there is no fault-effect on the address line. In such a case, the read circuit will read the same memory location and will get the same faulty value that was written by the write circuit.




(a) Fault Effect at Data Line (b) Fault Effect at Address Line

Figure 5.5: Propagating Fault Effect through Memory

For example, consider the faulty memory in Figure 5.5a. Here the fault-effect 1/0 propagates to the $2^{nd}$ bit of the data bus. Hence the faulty value 0011 is written to the memory location 1001 instead of the fault-free value 0111. Since there is no fault on the address line both write and read circuit will write and read the same memory location. Subsequently, the faulty copy of read circuit will read 0011 from the memory location 1001. Thus, if a fault-effect is present on the $i^{th}$ bit of the write data line, then the fault-effect can only propagate through the corresponding $i^{th}$ bit of the read data line. To encode this constraint, the active clause added will be

$$Act_{write\_data[i]} \implies Act_{read\_data[i]}$$

In this constraint, we state that if the $i^{th}$ bit of write data bus is active, then the $i^{th}$ bit of read data bus is also active. We further add active clauses for all gates in the read circuit those are in the fanout cone of the $i^{th}$ read data line.

### Fault Effect at Address Line

If a write address line is in the fanout cone of the faulty gate, then the value (may or may not have any fault-effect) will be written to a different address in the faulty copy of the memory than the intended one. The read address, being fault-free, will read

the value from the correct address. Thus, the fault-effect is propagated by reading the value from a different memory location than the memory location that was written. In particular, the value being written to the faulty address should be different than the existing value at the correct address and should be able to produce a different output.

For example, consider the faulty memory in Figure 5.5b. Here the fault-effect 1/0 propagates to the $2^{nd}$ bit of the address bus. The data bus has the value 0011 and has no fault-effect on it. This value is written to memory location 1001 in the faulty circuit instead of memory location 1101. The read circuit will read from the correct memory location 1101. The value present at this memory location will be a value written by one of the previous test vectors or it will be 0000 (initialization value) if the memory location was never written by any of the previous vectors. To propagate the fault-effect the value at address 1101 should be different than 0011, i.e., there has to be a fault-effect on one of the read data lines. Thus, if a write address line is in the fanout of the fault site, then one of the read data lines has to be active. In other words, if a fault-effect is present on the $i^{th}$ bit of the write address line, the active clause added will be

$$Act_{write\_address[i]} \implies \bigvee_{k=0}^{n-1} Act_{read\_data[k]}$$

Note that, since the read circuit will read the value from the fault-free address, it will read the value that was written to that location by some previous vector, or it will read an all-0 value if that memory location was never written by any of the previous vectors. Therefore, it is necessary to preserve the state of the memory after each iteration, which can be easily done using select and store constructs of SMT as explained in Section 5.4.

### 5.4.4 Extending the Formulation to Multiple Memories

The proposed SMT formulation can be readily extended to circuits having multiple embedded memories, by adding more memory variables and adding the relevant constraints. For example, consider circuit in Figure 5.6.

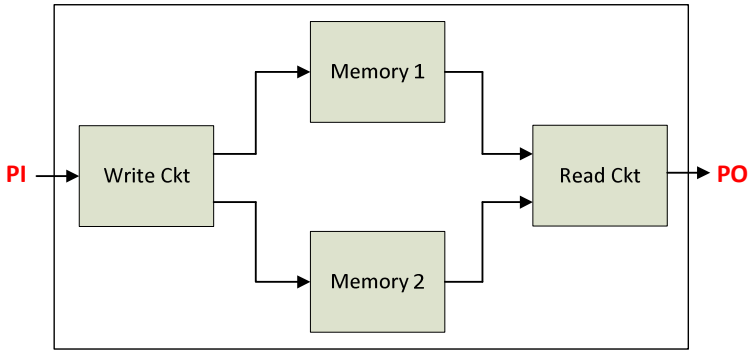



Figure 5.6: Multiple Memories in the Circuit

Here the write circuit is writing to two different memories and read circuit is also reading from two memories. A fault in the write circuit can be propagated through Memory 1, Memory 2, or both. For some faults, the fault-effect may propagate through Memory 1 but value read from Memory 2 may be important to sensitize the path to propagate the fault-effect through the read circuit. Storing the values written to the memory by previous vectors becomes even more important in such cases because the read circuit may read a value from Memory 2 that was written by some previous vector and is able to propagate the current target fault.

After adding all the clauses, the SMT solver checks if the formula is satisfiable. If it is not satisfiable, then the fault is proven to be untestable. If it is satisfiable, then we extract the needed vector from the satisfiable model. After every ATPG iteration in which a vector was derived, fault simulation is performed to drop all the additional faults detected.

## 5.5 Experimental Results

Experiments were performed on a Red Hat Linux workstation with Intel® Pentium® R D 3.0GHz CPU, 8GB memory and 4 cores. The circuits used for the experiments were generated using the model explained in Section 5.3. SMT solver Z3 [104] was used as the underlying SMT solver.

Table 5.1 shows the results of test generation of circuits for hard-to-detect stuck-at faults in the circuit, including the faults in the memory cell. Random vectors were first applied and all the easily detected faults were dropped. The remaining hard-to-detect faults were targeted using the proposed SMT-based formulation. The results were compared against random vectors as well as a gate-level sequential test generator STRATEGATE based on techniques described in [107–109].

In Table 5.1, Columns 1 and 2 list the ISCAS circuit used as the write and read circuits, respectively. Columns 3 and 4 report the size of address bus and data bus of the memory respectively. Column 5 lists the total number of collapsed faults. Column 6 list the number of random vectors used for initial simulation. Column 7 lists the total number of faults detected by these random vectors, and Column 8 reports the total number of target faults left after initial random simulation. Next, column 9 reports the total number of faults detected by random vector set of same size of the test set generated by our SMT based test generator. Column 10 reports the total number of faults detected by random vector set of 8× the size of the test set generated by our SMT based test generator. Columns 11, 12 and 13 report the faults detected, faults proven to be untestable and time taken in seconds by STRATEGATE. Columns 14 and 15 report the faults detected and faults proven to be untestable by our SMT based test generation. Columns 16 and 17 list the total number of vectors and time taken by our approach.

Table 5.1: Results for SMT-based Test Generation for Hard-to-Detect Faults

| Circuit Features | | | | | Initial Random Simulation | | | Random | | STRATEGATE | | | SMT-based | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| write ckt | read ckt | add bits | data bits | total faults | # vectors | faults det | faults left | A det | 8×A det | faults det | fault untest | time (sec) | faults det | faults untest | # vectors | time (sec) |
| c499 | c499 | 8 | 8 | 5612 | 6000 | 4977 | 635 | 127 | 515 | **619** | 0 | 2511* | **619** | 16 | 809 | 22 |
| c880 | c3540 | 8 | 8 | 8466 | 8000 | 5406 | 3060 | 82 | 491 | 960 | 125 | TO | **1563** | 1497 | 1391 | 225 |
| c1355 | c3540 | 8 | 8 | 9098 | 9000 | 8536 | 562 | 53 | 292 | **417** | 143 | 3960* | **417** | 145 | 611 | 110 |
| c1908 | c1908 | 8 | 8 | 7854 | 7000 | 7371 | 483 | 71 | 349 | **465** | 18 | 1722 | **465** | 18 | 623 | 48 |
| c1908 | c6288 | 8 | 8 | 13719 | 6000 | 12948 | 771 | 162 | 611 | **728** | 43 | 1995 | **728** | 43 | 880 | 223 |
| c499 | c499 | 8 | 16 | 9708 | 10000 | 9290 | 418 | 62 | 310 | 394 | 0 | TO | **402** | 16 | 565 | 17 |
| c880 | c3540 | 8 | 16 | 12562 | 20000 | 9639 | 2923 | 179 | 678 | 605 | 0 | TO | **2018** | 905 | 2364 | 632 |
| c1355 | c3540 | 8 | 16 | 13194 | 20000 | 12051 | 1143 | 5 | 32 | 110 | 0 | TO | **358** | 785 | 681 | 1584 |
| c1908 | c1908 | 8 | 16 | 11950 | 12000 | 11340 | 610 | 41 | 290 | 545 | 0 | TO | **592** | 18 | 675 | 60 |
| c1908 | c6288 | 8 | 16 | 17815 | 12000 | 17056 | 759 | 74 | 373 | 645 | 0 | TO | **716** | 43 | 691 | 161 |

A: Size of the derived SMT test set

TO: timeout of 2 hrs

*: test generation terminated with remaining faults unresolved

$8 \times 8$ memory contains $2^8 \times 8 = 2048$ memory bits

$8 \times 16$ memory contains $2^8 \times 16 = 4096$ memory bits

From the right-most four columns of Table 5.1, it can be seen that our formulation is able to detect all the hard to detect faults in a very short time. For example, for a circuit constructed using c1908 as the write circuit as well as the read circuit and 8×8 memory (containing $2^8 \times 8 = 2048$ data bits), the initial random simulation with 7000 vectors detect 7371 out of the 7854 faults. Next, all three approaches, including random, STRATEGATE, and our SMT-based test generator, were used to test the remaining 483 faults. The SMT based test generator detected 465 out of the 483 faults and proved the remaining 18 faults to be untestable. The total number of vectors generated for detecting the 465 faults were 623 and total time needed for test generation was just 48 seconds. When the same number of random vectors (623) were simulated, they were able to detect only 71 faults out of the 483 faults. Also, when the size of the random test set was increased by 8 times (i.e. 4984 vectors) the number of faults detected increased to 349. STRATEGATE also detected 465 faults and proved the remaining 18 faults to be untestable, but the time taken was 1722 seconds which is significantly longer than the time taken by our approach. For many of the test cases STRATEGATE timed out (2 hrs) before resolving all the faults. Note that, no compaction and compression technique has been used to reduce the size of the test set generated by the SMT-based test generation.

After increasing the size of the memory to $8 \times 16$ with the same c1908 as write and read circuits, our SMT based test generator needed only 60 seconds (12 seconds more than the $8 \times 8$ memory) for testing of 610 faults that were left after the initial random simulation. In STRATEGATE, because the number of FFs now increased from 2048 to 4096, the search space has increased significantly more. This in turn made STRATEGATE unable to complete in 2 hours. Results for other circuits show a sim-

ilar trend. This shows that our SMT formulation is scalable with increase in the size of the memory.

Table 5.2: Results for SMT-based Test Generation for All Faults

| Circuit Features | | | | | SMT-based | | | |
|---|---|---|---|---|---|---|---|---|
| write ckt | read ckt | add bits | data bits | total faults | det faults | untest faults | # vectors | time (sec) |
| c499 | c499 | 8 | 8 | 5612 | 5596 | 16 | 3199 | 100 |
| c880 | c3540 | 8 | 8 | 8466 | 6969 | 1497 | 1176 | 431 |
| c1355 | c3540 | 8 | 8 | 9098 | 8953 | 145 | 3220 | 607 |
| c1908 | c1908 | 8 | 8 | 7854 | 7836 | 18 | 3500 | 332 |
| c1908 | c6288 | 8 | 8 | 13719 | 13676 | 43 | 3061 | 1022 |
| c499 | c499 | 8 | 16 | 9708 | 9692 | 16 | 5525 | 162 |
| c880 | c3540 | 8 | 16 | 12562 | 11657 | 905 | 4848 | 1397 |
| c1355 | c3540 | 8 | 16 | 13194 | 12409 | 785 | 4450 | 2513 |
| c1908 | c1908 | 8 | 16 | 11950 | 11932 | 18 | 4247 | 494 |
| c1908 | c6288 | 8 | 16 | 17815 | 17772 | 43 | 3526 | 1327 |

Table 5.2 shows the results of test generation of circuits for *all* faults in each circuit including the faults in the memory when no initial random vectors are used. For example, for circuit with c1908 as write and read circuit and an $8 \times 8$ memory, there were a total of 7854 collapsed faults. Our SMT-based test generator detected 7836 of these 7854 faults, and 18 were proven to be untestable. The total number of vectors generated was 3500 and the time needed for test generation was 332 seconds. After increasing the size of memory to $8 \times 16$ the time needed for testing the new 11950 faults was only 494 seconds which again shows the scalability of our test generator.

## 5.6 Summary

We proposed an SMT-based formulation for test generation of circuits containing small embedded memories. Different classes of constraints are automatically added to ensure

that the target fault can be detected, whether it propagates through the address or data buses of the memory. Experimental results show that the proposed technique can efficiently detect the faults around the memory as well the faults within the memory. The proposed SMT formulation is scalable with increase in the size of memory and can also be extended for test generation of circuits with multiple embedded memories.

# Chapter 6

# LBIST Architecture for Improved Diagnosability

## 6.1 Chapter Overview

In LBIST, output compression techniques are used to reduce hardware overhead of storing test responses but such compression makes diagnosis extremely challenging as the failing vector and the output to which the fault effect propagated are unknown. In this chapter, we propose a new property checking based LBIST architecture which monitors certain properties in the output responses. If any property is violated, the failing vector and property number are stored for diagnosis. The proposed architecture improves diagnosability considerably with minimal hardware overhead. Experimental results show that the diagnostic resolution achieved by our architecture is comparable to the diagnostic resolution achieved in a non-BIST setup for many circuits.

## 6.2 Introduction

LBIST uses output compression to reduce the hardware overhead of storing the fault free responses (refer Section 2.4.2). Since the output compression is lossy, it makes diagnosis difficult. Previous methods that attempt to increase the diagnosability include enhancing the observability by capturing/storing intermediate states and comparing them with the pre-determined values off-line. The overheads of such methods are that additional on-chip storage and comparison with expected intermediate state values are needed. To reduce the additional on-chip storage and avoid off-line processing, several output response compression techniques have been proposed [110–117] that convert the intermediate output responses into smaller signatures.

Although output compression reduces the overhead of storing the responses on chip, storing all intermediate responses can still be extremely costly in terms of area overhead. To reduce the area overhead the intermediate signatures are often compressed using a Multiple Input Shift Register (MISR). Only the final fault-free signature is stored on chip. Although this reduces the hardware overhead significantly, it makes diagnosis extremely difficult because if the chip fails the diagnosis has to be performed based on a single faulty signature. It is hard to determine the cause for the failure based on a single signature. All the detectable faults by all the LBIST patterns can be potential fault candidates for every fault model. This list can be pruned by fault simulating each fault and checking if the final signature matches the faulty one. But many faults can lead to the same final signature which is known as aliasing effects in compression. In some cases, the faulty signature may match the fault-free signature which is known as fault masking. This fault aliasing and fault masking further complicates diagnosis.

To address the aforementioned challenges, we propose an entirely different and new

approach to enhance the diagnosability of LBIST. In our approach, we extract properties or invariants in the fault-free responses from the circuit. These invariants are monitored by a set of on-chip hardware monitors. If any property is violated, we can ensure that a defect is present and is captured by the violated property-monitor. Both the vector number and violating property number are stored for diagnosis. Thus, our technique improves diagnostic resolution with a small area overhead. Experimental results show that the diagnosability of our proposed architecture is comparable to the diagnosability in a non-BIST setup for many circuits.

## 6.3 Previous Work

Several output compression techniques have been proposed in the literature to reduce the storage requirements. Some are based on time compression and some are based on space compression. Space compression using parity tree [110–118] is one of the most popular technique for output response compression due to its simplicity to generate/implement and its independence of the circuit under test (CUT) and the test pattern set [117]. The parity of the output or scan cycle is computed and compared with the pre-computed fault-free parity bits stored on chip. To compute the parity of the output, parity trees are constructed which are XOR gates connected in a tree-like fashion. An example of a parity tree is shown in Figure 6.1.

While extreme output response compression can be achieved with a parity tree, such techniques have the following drawbacks: Test generation time increases because of XOR gates used to implement parity trees. When there is extreme compression, fault aliasing and masking will make both test generation and diagnosis difficult. In addition, the fault-free parity-tree and the resulting values stored in a response memory incur

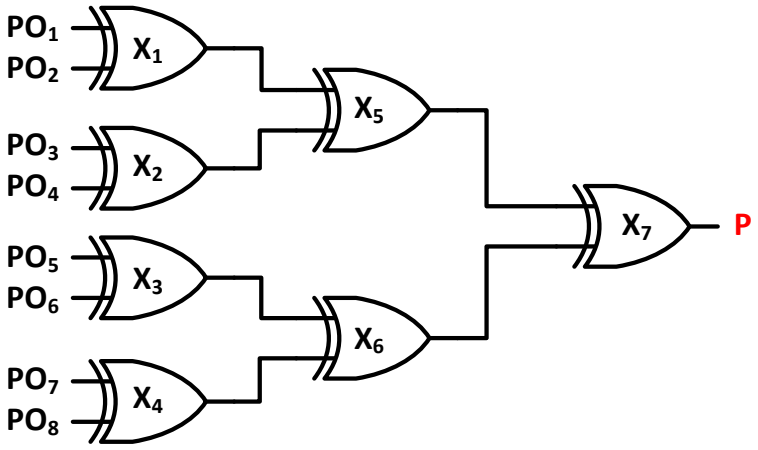
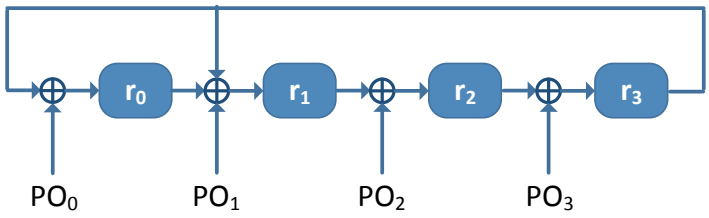



Figure 6.1: Parity Tree

hardware overhead. In [118], instead of parity, a two-level XOR tree has been proposed to improve diagnosability.

Another popular technique for output response compression is Multiple Input Signature Register (MISR). An example of a 4-bit MISR is shown in Figure 6.2a.




(a)

$r_0 = r_3 + PO_0$
$r_1 = r_3 + r_0 + PO_1$
$r_2 = r_1 + PO_2$
$r_3 = r_2 + PO_3$

(b)

| $PO_0$ | $PO_1$ | $PO_2$ | $PO_3$ | $r_0$ | $r_1$ | $r_2$ | $r_3$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

(c)

| $PO_0$ | $PO_1$ | $PO_2$ | $PO_3$ | $r_0$ | $r_1$ | $r_2$ | $r_3$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

(d)

Figure 6.2: Example of Compression using MISR

The four flip-flops in the MISR are labeled $r_0$ to $r_3$. Let $PO_0$ to $PO_3$ be the four outputs of the CUT. The equations of the transition functions for the four flops are shown in Figure 6.2b. An example of the signature computation is shown in Figure

6.2c for five vectors. Let the fault-free responses for the 5 vectors be 1011, 0110, 0000, 1101 and 0001 respectively. Then, the corresponding values of the shift registers are shown in Figure 6.2c. The final state of the MISR is 0011 which is the fault-free signature. Figure 6.2d shows the case where a fault is detected by vector 3. The faulty response for vector 3 is 0010. After passing the responses through the MISR the final signature is 1111. Since this signature is different than the expected fault-free signature, the fault will be detected. This kind of compression is known as time (or sequential) compression.

From the example above we see that the MISR provides extremely high compression ratio since only a single signature needs to be stored and compared irrespective of the number of vectors applied. This is good from an area overhead point of view but makes diagnosis extremely difficult as the exact failing vector and the failing response cannot be easily determined from the observed faulty signature.

To reduce fault aliasing and fault masking, storing intermediate MISR signatures after every $k$ vectors was proposed in [119]. This helps diagnosis as the search for failing vector is restricted to the $k$ vectors. But storing multiple signatures can incur additional overhead and still cannot point to the exact failing vector or faulty response.

The STUMPS architecture [35, 95] combines BIST and internal scan chains for the testing of sequential circuits, shown in Figure 6.3. In this architecture, the pseudo random patterns generated by LFSR are shifted into the scan chains. After capturing the response the scan cells are shifted out into the MISR. The outputs of the circuit are also fed to the MISR (not shown in the figure). This results in time as well as space compression.

Much work has been done in the past on diagnosis of circuits containing BIST and MISR structures. Simulation based techniques [120, 121] have been used that find the

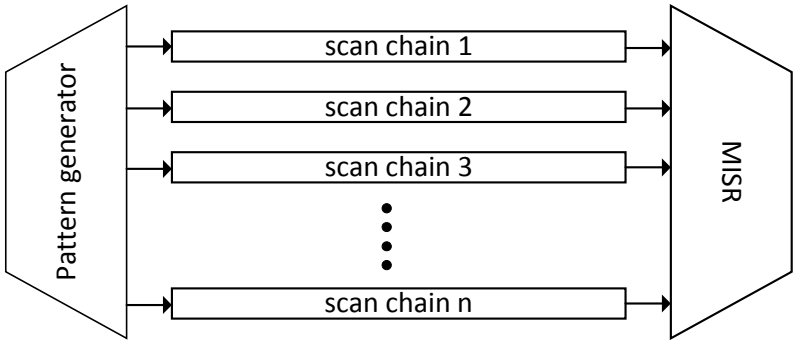



Figure 6.3: STUMPS Architecture

signatures for different faults using simulation and match the signature to the actual observed faulty signature. But a large number of faults may render such techniques impractical. Even after dropping some faults with pre-processing methods, the number of faults to be simulated could be very large.

In [122], the same test patterns are applied multiple times and each time a different set of outputs is passed to the MISR by using a selector network. Thus different signatures corresponding to different groups of outputs (with the same set of vectors) are collected. Then, an off-line analysis of the obtained error-free and possible erroneous signatures is performed to locate the failing scan cell.

In [123–127], a failing window is first determined by comparing intermediate signatures over fixed, variable, or overlapping windows. Once a suspect window is determined, then detailed analysis is performed over the suspect window to isolate the fault. In [123, 124], failing patterns are applied one at a time in the suspect window. Multiple signature analyzers are used in [125] to improve diagnostic resolution. In [126], the authors proposed observing the responses in the suspect window using a slow speed tester. In [127], the scan chains are offloaded for all the vectors in the suspect window for off-line analysis. In all of these techniques, multiple BIST sessions are used. In the first session, the good chips are separated from the failing chips. The defective chips are

then again subject to multiple BIST sessions for diagnosis. In these sessions, diagnosis is conducted by either extracting the responses before compression or by applying new vectors to distinguish between different faults, making diagnosis a tedious and time consuming task. Also high volume of data needs to be transferred between the tester and chip during the diagnosis sessions.

In our proposed approach, the failing vectors and failing property are automatically recorded during the test session by small property monitors. These recorded data can be used for subsequent diagnosis. Hence, it can reduce time as multiple BIST sessions are no longer required.

A lot of work has been done in the past on the use of additional hardware for on-line/concurrent fault detection. In Built-in Concurrent Self Test (BICST) [128], additional hardware is used to generate the fault-free responses for some pre-computed vectors. Whenever any of these pre-computed vectors appear during normal operation of the circuit, the response of the circuit is compared with expected fault-free response generated by the predictor hardware. However, this technique results in significant hardware overhead. Reduced Observation Width Replication (ROWR) [129] aims at reducing this hardware overhead by predicting the responses for only a subset of primary outputs for every vector. Even with this optimization, the hardware overhead is significant since a separate response predictor needs to be synthesized. Use of static logic implications for on-line fault detection has been proposed in [130]. Static logic implications are relationships between signals in a circuit that always hold. This idea was extended to multinode implications in [131] to improve the fault coverage achieved. In [132, 133] the authors proposed storing the individual error signal of each implication in a distinct flip-flop and using this diagnostic data for offline diagnosis. Even though static implications are very useful as they are vector independent, the fault coverages

achieved by using only implications is low. In our test-set dependent approach, we use 1-bit, 2-bit and 3-bit invariants on the primary outputs only to detect deviations in the responses.

### 6.3.1 Diagnostic Resolution and Diagnostic Fault Simulation

Diagnostic Resolution is the degree of accuracy with which a fault can be deduced/isolated based on the observed behavior (refer Section 2.3.1). In case of normal tester based diagnosis, the diagnostic resolution depends on the difference in the faulty responses of different faults. To measure diagnostic resolution, diagnostic fault simulation is performed (refer Section 2.3.1). In diagnostic fault simulation, the faulty responses for all the detected faults by each vector are compared and the faults with different responses are considered to be distinguished and thus placed in different classes. At the end of simulating all the vectors, equivalence classes of faults are formed and faults in each class are pairwise indistinguishable from each other for the given test set. We note that a high diagnostic resolution does not necessarily mean that the diagnosis is easy. However, it can reduce the complexity in isolating the faults.

For diagnosing a circuit without a LBIST setup, we can either simulate all the failing vectors to determine the faults that have the same matching signature as the observed faulty response or we can diagnose without explicit simulation using a SAT-based technique [64].

For diagnosis of circuits containing MISR structures, the fault simulation is even more expensive as each fault needs to be simulated for the entire test set separately to determine its final signature. However, if none of the fault signature matches the observed faulty signature, which may happen due to the presence of multiple faults or unmodeled defect, it is extremely difficult to diagnose using only the signature.

In a diagnosis-friendly architecture, we not only want the diagnostic resolution to be high, but we also want to have sufficient observability of the faulty responses so that diagnosis can be carried out without much simulation.

In the proposed architecture, whenever a property is violated, both the failing vector and property number are stored for diagnosis. From the violating property, the outputs to which the fault-effect was propagated can be determined. Hence, diagnosis can be performed without much simulation and without the need of multiple LBIST sessions.

## 6.4 The Naive Property Monitor based Approach

We propose a new diagnosis-friendly LBIST architecture in which the output response of every BIST vector satisfies some pre-computed, circuit-specific property. An on-chip property monitor is added to check if the output responses of the chip satisfy the property. If the property is violated, we know a fault must have been present. The BIST flow with our proposed naive property monitor approach is shown in Figure 6.4.

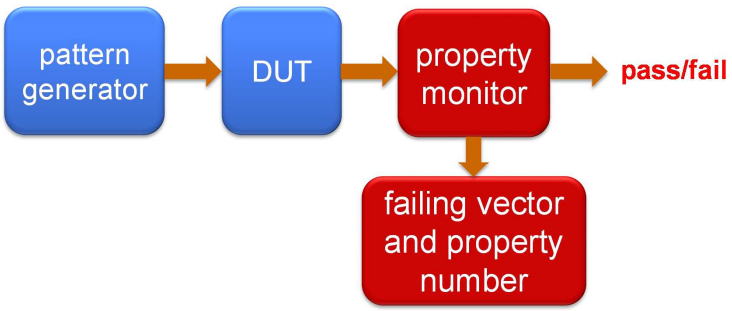



Figure 6.4: Naive LBIST Setup with Proposed Property Monitor

As the property monitor directly checks the properties in the output response of the chip, no other comparison is needed with the fault-free responses. Hence, there is no need to store the fault-free responses/signatures on chip. In addition, whenever the

property is violated, the cycle number and/or property number (in case of multiple properties) can be automatically recorded in this setup. Such information can offer an immense value to the diagnosis step, reducing both the time and cost associated with diagnosing the failed part. When the right set of properties is chosen, this approach can significantly reduce the area overhead while enhancing diagnostic capabilities of LBIST.

The on-chip test generator in LBIST is usually in the form of an LFSR. For a given LFSR and circuit, the output responses can be readily computed by logic simulation. We then extract properties, or invariants, that hold for all the fault-free responses obtained. An example of this invariant based property monitor is shown in Figure 6.5.

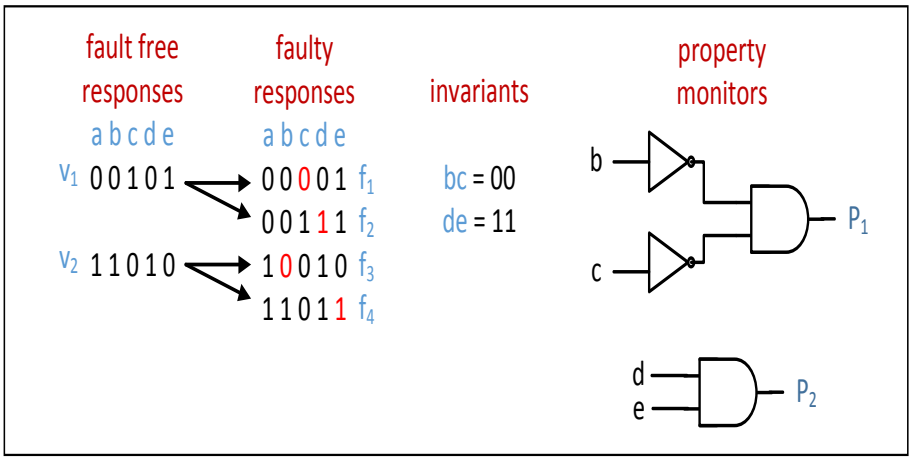



Figure 6.5: Invariant-based Property Monitor

Suppose the circuit has five primary outputs namely a, b, c, d and e. Two vectors $v_1$ and $v_2$ are applied and the corresponding fault-free responses are 00101 and 11010. Vector $v_1$ detects two faults $f_1$ and $f_2$ with corresponding faulty responses 00001 and 00111. Similarly vector $v_2$ detects two faults $f_3$ and $f_4$ with corresponding faulty responses 10010 and 11011. By comparing the fault-free and faulty responses we can observe that values $bc = 00$ and $de = 11$ never occur in the two fault-free responses

but they are present in the faulty responses. In fact, monitoring these two invariants is sufficient to detect all the 4 faults. The corresponding property monitors are also shown in Figure 6.5.

The 1-bit, 2-bit and 3-bit invariants in the fault-free responses can be easily computed using any heuristic data mining approach. We then attach these invariants to the primary outputs of the original circuit. Next, we fault-simulate the (collapsed) faults of the CUT to separate the useful invariants from the useless and/or redundant invariants. For a fault to be detected, at least one faulty response for the test set needs to violate at least one invariant.

In an ideal scenario, a property monitor should satisfy the following requirements:

- High fault coverage must be achieved for modeled and unmodeled faults/defects.
- The monitor should be small enough to not cause significant hardware overhead.
- The monitor should be relatively easy to synthesize/construct.
- Test generation of the circuit should be easy with the monitor attached to the outputs of the circuit.
- Fault masking and aliasing should be minimized to help diagnosis.

Algorithm 6.1 outlines the computation of the naive property monitor.

**Algorithm 6.1** Naive Property Monitor Algorithm

1: Logic simulate the test vectors and record the fault-free responses.
2: Find all possible invariants (missing patterns) in the fault-free responses.
3: Fault simulate the test vectors and create a fault dictionary.
4: Use an ILP solver to find minimal cover of properties that detect all faults.

As mentioned earlier, we first logic simulate all the LBIST vectors and record the corresponding fault-free responses. Then we find all the 1-bit, 2-bit and 3-bit invariants

in the fault-free responses. Fault simulating all the faults without fault dropping is performed to construct a matrix (dictionary) of which properties detected which faults. For each detected fault, the dictionary can tell us if the fault is violated by any of the properties. In order to find the minimal cover of properties that detect all faults we used Gurobi optimizer [134] which is a state-of-the-art solver for linear programs.

### 6.4.1 Experimental Results for Naive Approach

To test the feasibility and effectiveness of the naive approach, we performed several experiments on some of the ISCAS85, ISCAS89 full scan and open core circuits from IWLS 2005 benchmark suite. The experiments were conducted on a 4 core 3.20GHz Intel®Xeon™ CPU with 2 GB memory running Ubuntu 10.04 32-bit Operating system. The results for the naive approach are shown in Table 6.1.

For this experiment we simulated 10240 random vectors for each circuit. In this work, we assume full scan architecture for sequential circuits. No specific scan configuration is used. All the flip-flop values and primary outputs are considered in the fault-free response and are used for finding the invariants.

In Table 6.1, columns 1 and 2 list the circuit name and number of primary outputs in the circuit. Column 3 reports the total number of 1-bit, 2-bit and 3-bit invariants found. Columns 4 and 5 report the number of faults detected by the test set (without properties) and the number of faults caught/detected by the properties, respectively. column 6 reports the fault coverage achieved by the property monitors. Column 7 reports the size of compact properties after using the ILP solver and column 8 reports the area overhead for the property monitors.

From Table 6.1, we can observe that for the same number of vectors the number of invariants found is significantly different for different circuits. Circuits c880 and c3540

Table 6.1: Experimental Results for Naive Approach

| circuit | # PO | # invariants | # fault det orig | # fault det prop | fault cov prop (%) | # compact prop | over head (%) |
|---|---|---|---|---|---|---|---|
| c880 | 26 | 89 | 941 | 260 | 27.63 | 16 | 8.5 |
| c2670 | 140 | 1728 | 2315 | 1622 | 70.00 | 91 | 14.11 |
| c3540 | 22 | 86 | 3288 | 282 | 8.5 | 12 | 1.49 |
| c5315 | 123 | 17495 | 5291 | 4780 | 90.34 | 258 | 18.25 |
| c7552 | 108 | 605 | 7096 | 1036 | 14.59 | 27 | 1.59 |
| s1423f | 79 | 66 | 1495 | 322 | 21.53 | 58 | 14.14 |
| s5378f | 228 | 160109 | 4536 | 4213 | 92.87 | 295 | 18.93 |
| s9234f | 250 | 10476 | 5833 | 2965 | 50.83 | 270 | 12.94 |
| s13207f | 790 | 3113039 | 9217 | - | - | - | - |
| s15850f | 684 | 374812 | 10702 | 9509 | 88.85 | 860 | 19.54 |
| s35932f | 2048 | 20621 | 35110 | 12230 | 34.83 | 2111 | 23.58 |
| s38417f | 1742 | 5232834 | 28797 | - | - | - | - |
| s38584f | 1730 | 3392023 | 34324 | - | - | - | - |
| aes_coref | 659 | 6455 | 50242 | 272 | 5.41 | 44 | 0.2 |
| des3_areaf | 192 | 384 | 9388 | 384 | 4.09 | 192 | 5.56 |
| des_areaf | 128 | 128 | 8652 | 256 | 2.95 | 128 | 4.22 |
| i2cf | 142 | 1708 | 2405 | 1377 | 52.25 | 200 | 25.72 |
| wb_dmaf | 738 | 871350 | 9422 | - | - | - | - |

have fewer primary outputs, hence there were few invariants which resulted in very few faults that could be caught by the property monitors. Some circuits such as s13207f and s38584f have large numbers of primary outputs and we were able to find a lot more invariants. But the number of invariants found was so large that we could not fault simulate all of them to check which of the properties are useful.

From the preceding discussion, it may seem that the number of invariants depends on the number of primary outputs, but that is not necessarily true. For example, for c5315 with 123 outputs, we were able to find 17495 invariants but for des_areaf with 128 outputs we were able to find only 128 invariants. Furthermore, for circuits c5315 and s5378f, where the fault coverage is higher as compared to the other circuits, we see that the hardware overhead of the monitors after compaction is very high.

From these results we see that the Naive property-monitor based approach has several drawbacks, including large area overhead and low fault coverage. Also for some circuits, the number of invariants is too large which increases time needed for fault simulation and the memory required for the fault dictionary which makes the naive approach unscalable to large circuits. We also observe that for some circuits, even though the number of invariants is large, many faults remain undetected by the property monitors. In addition, as the number of vectors increase, the number of missing patterns in the fault-free responses, i.e., the number of invariants will decrease which will reduce both fault coverage and diagnosability even further. Moreover, we note that this naive approach is only able to detect the faults whose faulty responses are not identical to any of the fault-free responses for any vector. In other words, if the faulty response of a vector is the same as a fault-free response for some other vector, then that fault will not be detected by this approach.

## 6.5 The Practical Property Monitor based Approach

In order to overcome the drawbacks in the naive approach, we propose a new, practical and scalable approach. Instead of detecting all the faults using the property monitors, we now try to detect as many faults as possible under a certain hardware area overhead limit. This is to improve diagnosability without compromising fault coverage.

The LBIST set-up for the practical property monitor approach is shown in Figure 6.6. In this approach, we keep the MISR portion of the conventional LBIST untouched. This ensures that complete fault coverage is achieved. We observed in the naive approach that even when we have a large number of invariants, the number of faults detected

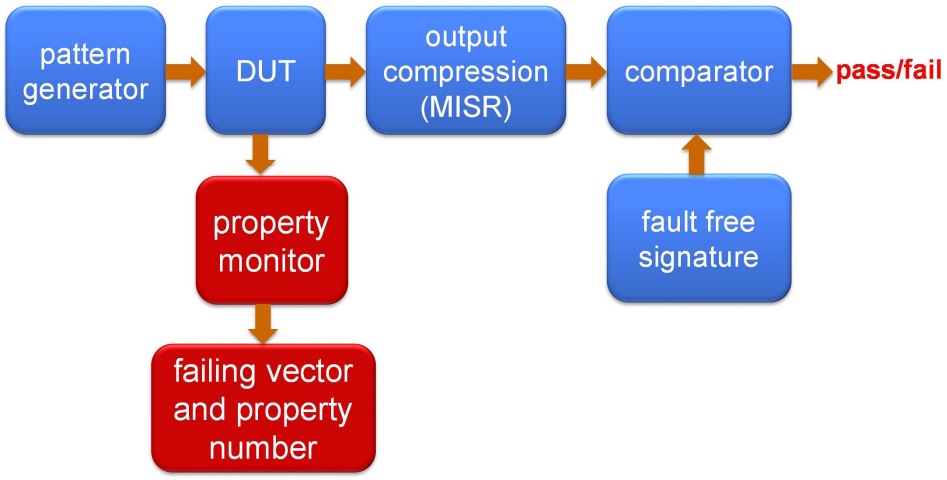



Figure 6.6: Practical LBIST Setup with Property Monitor

can still be low. This means that we are not finding the *right* invariants. The fault coverage can be improved by finding higher-order invariants instead of only 1-bit, 2-bit and 3-bit invariants. However, finding higher-order invariants can be computationally expensive and can lead to greater area overhead.

In order to increase the number of small-sized invariants, we divide the vectors into $k$ equivalent sets. This can be achieved by using a modulo-$k$ counter which increments with every vector. For each of the sets, the invariants/properties are computed separately. A log-$k$ bit decoder is used to select the relevant properties to be considered in every cycle. This is shown in Figure 6.7.

In the figure, we can see that the output of the circuit is passed through all property monitors but only the output of the relevant property set for the current cycle is selected by using the decoder. As a result, the number of invariants increases significantly which increases the number of faults that can be detected/distinguished by using the property monitors.

Dividing the vectors into $k$ equivalent sets also reduces the fault masking issue in the naive approach where a fault cannot be detected if faulty response is identical

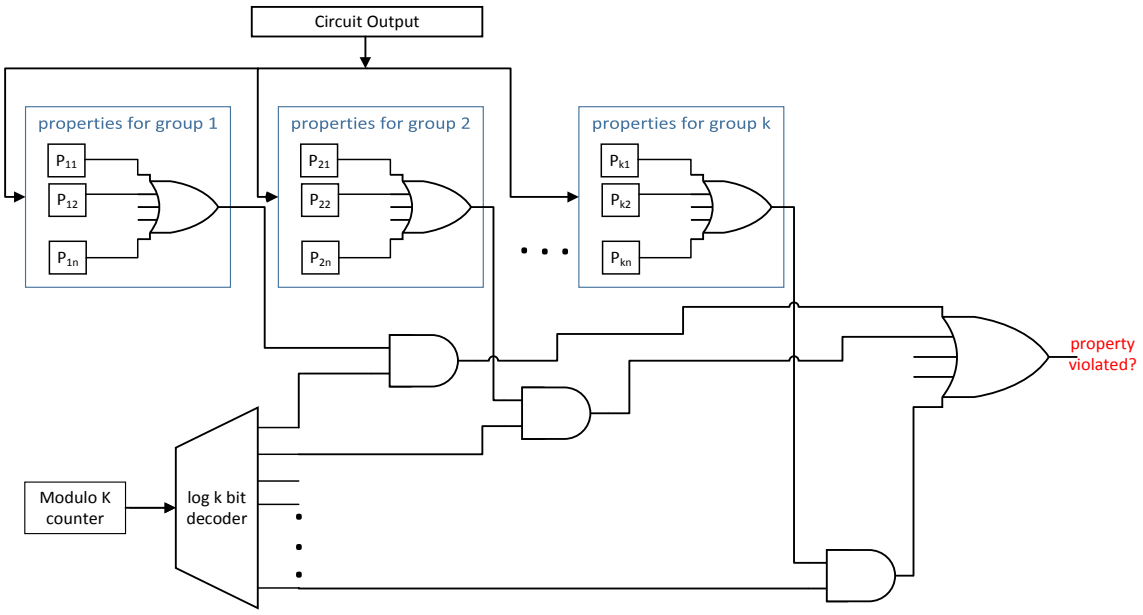



Figure 6.7: Use of Counter to Increase the Number of Invariants

to any of the fault-free responses. Since the vectors have been divided into $k$ sets, the corresponding fault-free responses are also divided into $k$ sets. Thus, if the faulty response for a fault is identical to the fault-free response from some other set, then the fault can still be detected. Hence the fault masking is significantly reduced in the practical approach. However, if all the faulty responses for a fault are identical to the fault-free responses in the same set then the fault will remain undetected. Also, since we are restricting our approach to 1-bit, 2-bit and 3-bit invariants, those faults that require higher-order invariants to be detected will remain undetected.

Even though we increase the total number of invariants by dividing the test set into multiple groups, we want to limit the number of invariants selected for fault simulation. This is because large numbers of invariants can lead to large fault dictionaries and a large area overhead. In order to reduce the number of invariants selected for fault simulation, we limit the number of properties for every output. We keep a count of the number of properties for both 0 and 1 values of each output. Thus we maintain

two counters $C0_i$ and $C1_i$ for each output $PO_i$ where $C1_i$ is the number of properties that have $PO_i = 1$ and $C0_i$ is the number of properties that have $PO_i = 0$. Every time a new invariant/property is found, we increment the corresponding counters of each output in the property.

We can keep a common threshold $T$ for the number of properties allowed at every output. However, not all outputs detect the same number of faults. The fault detection at every output depends on several factors such as test vectors used, number of sensitizable paths to the output, controllability and observability of the gates along those paths. In order to determine the threshold for the number of properties that can be allowed at each output, we first determine the number of faults detected at each output. This is done by fault simulation without fault dropping and comparing the faulty and fault-free signatures. For the outputs that detect more faults, we want the threshold for the number of properties to be higher than for the outputs that detect fewer faults. Thus, we keep the threshold of properties, $T_i$, at each output $PO_i$ in proportion to the number of faults detected $fd_i$ at each output:

$T_i = m \times fd_i$, where $m$ is a constant

In other words, the threshold for the number of properties at each output is equal to the number of faults detected at that output multiplied by a constant $m$. For some outputs, the number of faults detected $fd_i$ can be too high or too low which will cause a similar effect on the values of $T_i$. To avoid this, we keep minimum and maximum thresholds for values of $T_i$. Thus the value of $T_i$ is always in the range

$MIN\_THRESHOLD \leq T_i \leq MAX\_THRESHOLD.$

Since we limit the number of properties, we must order the outputs in a clever way so that outputs at which many faults are detected can be considered more often. To do so, we first order the outputs in the descending order of number of faults detected.

This ensures that the outputs that detect more faults have better chance of finding sufficient properties.

Once the outputs are arranged in the desired order, we compute the invariants with the property threshold constraints. Similar to the naive property based approach, we fault simulate all the faults without fault dropping and create a fault dictionary which stores the faults detected by each property. We then use a greedy algorithm to pick the best $n$ properties that detect maximum number of faults under the specified hardware overhead limit.

The algorithm used for the practical property monitor scenario is outlined in Algorithm 6.2.

**Algorithm 6.2** Practical Property Monitor Algorithm

1: Fault simulate the vectors and record the number of faults detected at every output.
2: Determine the property count threshold for each output.
3: Arrange the outputs in descending order of fault detection.
4: Logic simulate the test vectors and record the fault-free responses.
5: Find the 1-bit, 2-bit and 3-bit invariants within the threshold for each output.
6: Fault simulate the test vectors with monitors attached and create a dictionary for faults detected by each monitor.
7: Arrange the properties in the descending order of fault detection.
8: Select the first $n$ properties with the hardware overhead limit.

## 6.6 Test and Diagnosis Flow with the Proposed Property Monitor Approach

The algorithm for the test and diagnosis flow with our proposed property monitor approach is outlined in Algorithm 6.3. The test flow is similar to the conventional LBIST where the test patterns are applied using the LFSR-based pattern generator followed by deterministic top-off patterns which are usually applied using LFSR reseeding. The

**Algorithm 6.3** Test and Diagnosis Flow with the Proposed Property Monitor Approach

```
for int i = 0; i < #vectors; i++ do
  apply vector i
  if any property violated then
    record property number and vector number
    abort the test cycle
  end if
end for
if MISR signature matches fault-free then
  chip is fault-free!
else
  use signature-based diagnosis
end if
```

output responses of these vectors are given to the property monitors as well as MISR based compactor. If any of the property is violated, the chip is faulty and the failing property and vector number is recorded for diagnosis. From the failing property, the outputs where the fault effect was propagated can be easily determined. The test cycle can be aborted as soon as the first property fails or we can allow the entire test set to be executed and record all the failing vectors and property numbers as it will help in reducing the fault candidates. Diagnosis can subsequently be conducted using fan-in cone analysis or SAT based technique similar to [64].

However, since the property monitors cannot detect all the faults due to the hardware area overhead limit, some faults will be missed. These faults need to be detected at the MISR output. If at the end of test cycle, none of the properties are violated but the MISR signature is different from the fault-free signature then we know that the mismatch is caused by one of the faults that were not captured by the property monitors. We can remove all the faults that can be detected by the selected property monitors from the fault candidate list. From the results in Section 6.7, it can be seen that the property monitors can detect more than 90% faults for all the circuits. Thus

we will be left with less than 10% faults which can be even pruned further using any signature based diagnosis technique.

## 6.7 Experimental Results

We tested the effectiveness of the practical approach on the same set of circuits, with the same 10240 vectors that were used for evaluating the naive approach. The $MIN\_THRESHOLD$ and $MAX\_THRESHOLD$ were set to 100 and 2000 respectively. In order to measure the diagnosability of our architecture, we performed diagnostic fault simulation of all the detectable stuck-at faults at the property monitors. We did not consider the MISR portion of the LBIST for this analysis. Thus, a fault was considered to be detected only if it was caught by any of the property monitors.

Note that we assume the architecture similar to STUMPS for sequential circuits. No specific scan configuration is used. All the flip-flop values and primary outputs are considered in the fault-free response and are used for finding the invariants. Hence if a LBIST architecture such as shown in Figure 6.3 is used, the flip-flops or primary outputs in any property monitor could either belong to same scan chain or may be spread across different scan chains. In a scan configuration, the invariants may be violated during the shift cycles. Hence the invariants need to be monitored only during the capture cycle.

During diagnostic fault simulation, we compare the properties where the fault was detected instead of comparing against all faulty responses as we also want to compute the diagnostic resolution achieved by our property monitors.

We compare our achieved diagnostic resolution, to the diagnostic resolution achievable in a non-BIST setup. Recall that in a non-BIST environment, the entire responses

of the circuit are without any MISR/compression or monitors. The results of our experiment are summarized in Table 6.2.

Columns 1 and 2 list the circuit name and the number of primary outputs. Columns 3 and 4 report the number of faults detected and the diagnostic resolution in the non-BIST setup. Column 5 reports the maximum value of the counter used, i.e., the number of sets the vectors are divided. For each circuit, we performed experiments with 8, 16, 32, 64 and 128 sets and we report the best results achieved for each circuit. Column 6 reports the average number of invariants per group. The next 8 columns report the number of faults detected/caught by the property monitors and the diagnostic resolution when the area overhead for property monitors was restricted to 3%, 5%, 10% and 15%.

By comparing Columns 4 and 14 we see that the diagnostic resolution achieved by our approach is comparable to the diagnostic resolution in a non-BIST setup. For example, for circuit c5315 with 123 outputs, the random test set of 10240 vectors detect 5291 faults in a non-BIST set-up, i.e., without any MISR or property monitors and the diagnostic resolution achieved was 1.08. With the counter value of 64 (the 10240 vectors were divided into 64 groups of 160 vectors each), the average number of invariants found per group was 7838. After building the fault dictionary and selecting the property monitors using our greedy algorithm with 3% area overhead limit, the selected properties detected 4359 faults and the diagnostic resolution was 1.62. Similarly, the number of faults detected by the property monitors with area overhead limit of 5%, 10% and 15% were 4784, 5196 and 5291, respectively and the corresponding diagnostic resolution achieved was 1.37, 1.16 and 1.11 respectively. Thus the diagnostic resolution of 1.11 achieved by our approach is comparable to the diagnostic resolution of 1.08 achieved by non-BIST set-up.

Table 6.2: Experimental Results for Practical Approach

| circuit | #PO | without prop | | counter | inv | 3% overhead | | 5% overhead | | 10% overhead | | 15% overhead | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | fdet | DR | | | fdet | DR | fdet | DR | fdet | DR | fdet | DR |
| c880 | 26 | *941* | 1.06 | 128 | 868 | 643 | 2.13 | 766 | 1.66 | *880* | 1.30 | *917* | 1.18 |
| c2670 | 140 | *2315* | **1.31** | 128 | 5551 | 2032 | 1.72 | *2113* | 1.58 | *2206* | 1.47 | *2289* | **1.38** |
| c3540 | 22 | *3288* | **1.11** | 128 | 384 | *2996* | 1.47 | *3144* | 1.30 | *3270* | 1.17 | *3287* | **1.15** |
| c5315 | 123 | *5291* | **1.08** | 64 | 7838 | 4359 | 1.62 | *4784* | 1.37 | *5196* | 1.16 | *5291* | **1.11** |
| c7552 | 108 | *7096* | **1.21** | 128 | 6129 | 6278 | 1.53 | *6642* | 1.38 | *6954* | **1.27** | *7043* | **1.24** |
| s1423f | 79 | *1495* | 1.11 | 128 | 5425 | 684 | 4.46 | 864 | 2.92 | 1164 | 1.74 | *1393* | 1.33 |
| s5378f | 228 | *4536* | 1.10 | 32 | 32482 | 3288 | 2.04 | 3685 | 1.68 | *4098* | 1.38 | *4304* | 1.25 |
| s9234f | 250 | *5833* | 1.38 | 128 | 15248 | 3876 | 3.12 | 4551 | 2.41 | *5310* | 1.81 | *5645* | 1.59 |
| s13207f | 790 | *9217* | 1.29 | 64 | 31365 | 5391 | 3.17 | 6386 | 2.45 | 7704 | 1.87 | *8319* | 1.65 |
| s15850f | 684 | *10702* | **1.28** | 64 | 27861 | 7559 | 2.49 | 8875 | 1.91 | *10124* | 1.52 | *10579* | **1.39** |
| s35932f | 2048 | *35110* | 1.39 | 64 | 64375 | 23899 | 2.41 | 27804 | 2.01 | *32062* | 1.71 | *33320* | 1.57 |
| s38417f | 1742 | *28797* | 1.21 | 64 | 86674 | 15613 | 3.05 | 18997 | 2.31 | 24033 | 1.63 | *26317* | 1.41 |
| s38584f | 1730 | 34324 | 1.09 | 64 | 64222 | 17036 | 2.91 | 21559 | 2.15 | 27297 | 1.59 | 30183 | 1.39 |
| aes_coref | 659 | *50242* | **1.04** | 64 | 3201 | *45362* | 1.22 | *47674* | 1.13 | *49436* | **1.07** | *49530* | **1.07** |
| des3_areaf | 192 | *9388* | **1.02** | 128 | 567 | *9121* | **1.07** | *9322* | **1.04** | *9384* | **1.03** | *9384* | **1.03** |
| des_areaf | 128 | *8652* | **1.04** | 128 | 193 | *8483* | **1.08** | *8632* | **1.05** | *8650* | **1.04** | *8650* | **1.04** |
| i2cf | 142 | *2405* | 1.06 | 64 | 15268 | 1117 | 3.39 | 1469 | 2.31 | 2018 | 1.47 | *2220* | 1.25 |
| wb_dmaf | 738 | 9422 | 1.03 | 8 | 35924 | 3773 | 3.39 | 5067 | 2.32 | 6791 | 1.63 | 7946 | 1.36 |

Coverages greater than 90% are in *italics*. Diagnostic resolution within 10% of the non-BIST setup are in **bold**

In Table 6.2, the cases where more than 90% of the faults were detected/caught by the property monitors are highlighted in *italic*. We observe that for all circuits except s38584f and wb_dmaf more than 90% of the faults were detected by the property monitors. The cases where the diagnostic resolution is within 10% of the diagnostic resolution achieved by the non-BIST setup are highlighted in **bold**. We see that for almost half the circuits we achieve the diagnostic resolution within 10% achieved by the non-BIST environment. Note that the diagnostic resolution of our approach can improve further if the faults detected at the MISR and the faulty MISR signature is considered for diagnosis.

A comparison with the naive (Table 6.1) and practical approach shows that the practical approach is able to achieve significantly higher fault coverage over the naive approach. Also, by dividing the test vectors into sets, we were able to find enough invariants to achieve high fault coverage. Depending on the circuit structure and the number of vectors used, the number of sets can be changed to achieve high fault coverage.

For the results reported in this dissertation, we have used pseudo random patterns only. But the technique can be easily extended to the deterministic patterns or top-off patterns. In such a scenario, the grouping of vectors and the generation of the invariants will need to consider the entire test set, i.e., pseudo random patterns and deterministic patterns will have to be cascaded for our analysis.

## 6.8 Summary

We proposed a property-checking based LBIST architecture for improving the diagnosability in LBIST. The proposed architecture makes diagnosis easier as the failing

vector and failing property can be recorded. The computation of invariant properties is easy and the area overhead can be restricted to the level set by the designer. The diagnostic resolution achieved by our architecture is close to the diagnostic resolution in a non-BIST environment.

# Chapter 7

# Conclusion

In this dissertation, we proposed techniques to address two current challenges in test and diagnosis namely, increase in test data volume and presence of embedded memories. To reduce the test data volume, we proposed multiple techniques; Firstly, we proposed ways to improve test and diagnosis in LBIST architecture, which eliminates the need of an external tester and solves the problem of test data volume. To this end, we proposed a technique to reduce the number of seeds needed in LBIST to achieve complete coverage and property-checking based LBIST architecture to improve diagnosability. Secondly, we proposed a technique to generate efficient and compact diagnostic vectors which makes diagnosis easier as well as reduces test data volume. To address the challenge posed by presence of embedded memories with regards to test generation, we proposed a technique that can generate test vectors for faults in the memory as well as faults in the logic around the memory without using scan or MBIST.

In Chapter 3, we proposed an SMT-based technique for the LFSR reseeding problem for LBIST. Instead of separate engines to compute the vectors and chaining them, our method unifies the two steps into one, eliminating the need to chain ATPG-generated

vectors. Excitation and detection constraints are encoded as SMT constraints, and Polarized z-sets were proposed as well to enhance the distinguishability between detected faults. Experimental results show that very few seeds are needed to achieve complete fault coverage. The proposed approach also shows that a single LFSR polynomial with few seeds is sufficient to achieve complete coverage. Thus the technique reduces the hardware overhead of storing the seeds.

In Chapter 4, we proposed a new SMT-based diagnostic test pattern generation technique for combinational circuits to reduce the number of diagnostic patterns needed to distinguish all fault pairs. Rather than targeting a single fault pair at a time, the proposed SMT-based approach targets multiple fault pairs in a single instance. Several heuristics were proposed to constrain the SMT formula to further reduce the search space, including fault selection, excitation constraint, reduced primary output vector, and cone-of-influence reduction. Experimental results for the ISCAS85 and full-scan versions of ISCAS89 benchmark circuits show that fewer diagnostic vectors are generated compared with conventional diagnostic test generation methods. Up to 73% reduction in the number of vectors generated can be achieved in large circuits.

In Chapter 5, we proposed an SMT-based technique for test generation of circuits containing small embedded memories. By using theory of bit-vectors and theory of arrays, we modeled the embedded memories at a higher level while keeping the logic around the memory at the gate/Boolean level. Different classes of constraints are automatically added to ensure that the target fault can be detected, whether it propagates through the address or data buses of the memory. Experimental results show that the proposed technique can efficiently detect the faults around the memory as well the faults within the memory. The proposed SMT formulation is scalable with increase

in the size of memory and can also be extended for test generation of circuits with multiple embedded memories.

Finally in Chapter 6, we proposed a property-checking based LBIST architecture for improving the diagnosability in LBIST. The missing patterns or invariants in the fault-free responses are used as hardware property monitors to detect faults. The proposed architecture makes diagnosis easier as the failing vector and failing property can be recorded. The computation of invariant properties is easy and the area overhead can be restricted to the level set by the designer. The diagnostic resolution achieved by our architecture is close to the diagnostic resolution in a non-BIST environment.

To sum up, several techniques for enhancing test and diagnosis were proposed in this dissertation. These techniques enable reduction in test data volume and test application time, thus reducing the overall time and cost of manufacturing test and diagnosis.

# Bibliography


[1] L.-T. Wang, C. E. Stroud, and N. A. Touba, *System-on-Chip Test Architectures: Nanometer Design for Testability*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[2] "International Technology Roadmap for Semiconductors,"ITRS 2010 Update"." http://www.itrs.net/Links/2010ITRS/Home2010.htm.

[3] "International Technology Roadmap for Semiconductors,"ITRS 2012 Update"." http://www.itrs.net/Links/2012ITRS/Home2012.htm.

[4] S. Prabhu, M. S. Hsiao, L. Lingappan, and V. Gangaram, "A Novel SMT-based Technique for LFSR Reseeding," in *25th International Conference on VLSI Design (VLSID)*, pp. 394–399, 2012.

[5] S. Prabhu, M. S. Hsiao, L. Lingappan, and V. Gangaram, "A SMT-based Diagnostic Test Generation Method for Combinational Circuits," in *30th IEEE VLSI Test Symposium (VTS)*, pp. 215–220, 2012.

[6] S. Prabhu, M. S. Hsiao, L. Lingappan, and V. Gangaram, "Test Generation for Circuits with Embedded Memories using SMT," in *18th IEEE European Test Symposium (ETS)*, p. 1, 2013.

[7] S. Prabhu, V. V. Acharya, S. Bagri, and M. S. Hsiao, "Property-checking based LBIST for Improved Diagnosability," in *19th IEEE European Test Symposium (ETS)*, pp. 1–2, 2014.

[8] S. Prabhu, V. V. Acharya, S. Bagri, and M. S. Hsiao, "A Diagnosis-friendly LBIST Architecture with Property Checking," in *IEEE International Test Conference (ITC)*, 2014.

[9] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

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

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

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

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

[14] I. Hamzaoglu and J. Patel, "New techniques for deterministic test pattern generation," in *Proceedings of 16th IEEE VLSI Test Symposium*, pp. 446–452, Apr 1998.

[15] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, pp. 4 –15, Jan 1992.

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

[17] J.-S. Chang and C.-S. Lin, "Test set compaction for combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 1370–1378, Nov 1995.

[18] P. Goel and B. C. Rosales, "Test generation and dynamic compaction of tests," in *International Test Conference*, 1979.

[19] S. Kajihara, I. Pomeranz, K. Kinoshita, and S. M. Reddy, "Cost-effective Generation of Minimal Test Sets for Stuck-at Faults in Combinational Logic Circuits," in *Proceedings of the 30th International Design Automation Conference*, DAC '93, (New York, NY, USA), pp. 102–106, ACM, 1993.

[20] S. Kajihara, I. Pomeranz, K. Kinoshita, and S. Reddy, "Cost-effective generation of minimal test sets for stuck-at faults in combinational logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 1496–1504, Dec 1995.

[21] I. Pomeranz, L. Reddy, and S. Reddy, "COMPACTEST: a method to generate compact test sets for combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1040–1049, Jul 1993.

[22] G. Tromp, "Minimal Test Sets for Combinational Circuits," in *Proceedings of International Test Conference*, pp. 204–, Oct 1991.

[23] I. Hamzaoglu and J. Patel, "Test Set Compaction Algorithms for Combinational Circuits," in *Digest of Technical Papers, IEEE/ACM International Conference on Computer-Aided Design (ICCAD).*, pp. 283 – 289, Nov. 1998.

[24] A. El-Maleh, "Efficient test compaction for combinational circuits based on fault detection count-directed clustering," *IET Computers and Digital Techniques*, vol. 1, pp. 364–368(4), July 2007.

[25] B. M. A. Abramovici, M. and A. D. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, NJ, USA: IEEE Press, 1994.

[26] P. Camurati, A. Lioy, P. Prinetto, and M. Sonza Reorda, "Diagnosis Oriented Test Pattern Generation," in *Proceedings of the European Design Automation Conference (EDAC)*, pp. 470 –474, March 1990.

[27] P. Camurati, D. Medina, P. Prinetto, and M. Sonza Reorda, "A diagnostic test pattern generation algorithm," in *Proceedings of International Test Conference*, pp. 52–58, Sep 1990.

[28] I. Hartanto, V. Boppana, J. Patel, and W. Fuchs, "Diagnostic test pattern generation for sequential circuits," in *15th IEEE VLSI Test Symposium*, pp. 196–202, Apr 1997.

[29] T. Gruning, U. Mahlstedt, and H. Koopmeiners, "DIATEST: A Fast Diagnostic Test Pattern Generator for Combinational Circuits," in *Digest of Technical Papers, IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 194 –197, Nov. 1991.

[30] I. Pomeranz and W. Fuchs, "A diagnostic test generation procedure for combinational circuits based on test elimination," in *Proceedings of 7th Asian Test Symposium (ATS)*, pp. 486–491, Dec 1998.

[31] I. Pomeranz, S. Reddy, and S. Venkataraman, "z-Diagnosis: A Framework for Diagnostic Fault Simulation and Test Generation Utilizing Subsets of Outputs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 1700–1712, Sept 2007.

[32] V. Agrawal, D. H. Baik, Y. C. Kim, and K. Saluja, "Exclusive Test and its Applications to Fault Diagnosis," in *Proceedings of 16th International Conference on VLSI Design*, pp. 143 – 148, Jan. 2003.

[33] Y. Zhang and V. Agrawal, "A Diagnostic Test Generation System," in *IEEE International Test Conference (ITC),*, pp. 1–9, Nov 2010.

[34] A. Veneris, R. Chang, M. Abadir, and M. Amiri, "Fault Equivalence and Diagnostic Test Generation using ATPG," in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, vol. 5, pp. V–221 – V–224, May 2004.

[35] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-in test for VLSI: Pseudorandom Techniques*. New York, NY, USA: Wiley-Interscience, 1987.

[36] J. Rajski and J. Tyszer, *Arithmetic Built-in Self-test for Embedded Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[37] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI Test Principles and Architectures: Design for Testability (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[38] M. Zivkovi, "Table of primitive binary polynomials," *Math. Comp*, vol. 63, pp. 38–5, 1994.

[39] K. Metzer Jr and R. Bouwens, "An Ordered Table of Primitive Polynomials Over GF (2) of Degrees 2 Through 19 for Use with Linear Maximal Sequence Generators," 1972.

[40] "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators." http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf, 1996.

[41] H. Schnurmann, E. Lindbloom, and R. Carpenter, "The Weighted Random Test-Pattern Generator," *IEEE Transactions on Computers*, vol. C-24, pp. 695–700, July 1975.

[42] F. Brglez, C. Gloster, and G. Kedem, "Built-in self-test with weighted random pattern hardware," in *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pp. 161–166, Sep 1990.

[43] A. Al-Khalili and D. Al-Khalili, "A controlled probability random pulse generator suitable for vlsi implementation," in *6th IEEE Instrumentation and Measurement Technology Conference (IMTC)*, pp. 247–255, Apr 1989.

[44] A. Giani, S. Sheng, M. Hsiao, and V. Agrawal, "Novel spectral methods for built-in self-test in a system-on-a-chip environment," in *19th IEEE Proceedings on VLSI Test Symposium (VTS)*, pp. 163–168, 2001.

[45] S. Hellebrand, B. Reeb, S. Tarnick, and H. Wunderlich, "Pattern generation for a deterministic BIST scheme," in *Digest of Technical Papers, IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 88 –94, Nov. 1995.

[46] P. Trouborst, "LFSR reseeding as a component of board level BIST," in *Proceedings of International Test Conference*, pp. 58 –67, Oct. 1996.

[47] S. Neophytou, M. Michael, and S. Tragoudas, "Efficient Deterministic Test Generation for BIST schemes with LFSR reseeding," in *12th IEEE International On-Line Testing Symposium (IOLTS) 2006.*, p. 6 pp.

[48] Z. Wang, K. Chakrabarty, and M. Bienek, "A Seed-Selection Method to Increase Defect Coverage for LFSR-Reseeding-Based Test Compression," in *12th IEEE European Test Symposium (ETS).*, pp. 125 –130, May 2007.

[49] Y.-H. Fu and S.-J. Wang, "Test Data Compression with Partial LFSR-Reseeding," in *Proceedings of 14th Asian Test Symposium*, pp. 343 – 347, Dec. 2005.

[50] Y.-Z. Yan, H. Wang, Z.-J. Yang, and S. Yang, "A New LFSR Reseeding Method for BIST," in *8th International Conference on Solid-State and Integrated Circuit Technology (ICSICT).*, pp. 2145 –2147, Oct. 2006.

[51] C. Krishna and N. Touba, "Reducing Test Data Volume using LFSR Reseeding with Seed Compression," in *Proceedings of International Test Conference, 2002.*, pp. 321 – 330.

[52] L. M. de Moura and N. Bjørner, "Satisfiability Modulo Theories: An Appetizer," in *SBMF*, pp. 23–36, 2009.

[53] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.

[54] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[55] S. A. Cook, "The Complexity of Theorem-proving Procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, (New York, NY, USA), pp. 151–158, ACM, 1971.

[56] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[57] J. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506–521, May 1999.

[58] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pp. 530–535, 2001.

[59] J. W. Freeman, "Improvements To Propositional Satisfiability Search Algorithms," 1995.

[60] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust Sat-solver," *Discrete Appl. Math.*, vol. 155, pp. 1549–1561, June 2007.

[61] H. Zhang, "SATO: An Efficient Propositional Prover," in *Proceedings of the 14th International Conference on Automated Deduction*, CADE-14, pp. 272–275, 1997.

[62] R. J. Bayardo, "Using CSP look-back techniques to solve real-world SAT instances," pp. 203–208, AAAI Press, 1997.

[63] N. Srensson and N. Een, "MiniSat v1.13 - A SAT solver with conflict-clause minimization," tech. rep., International Conference on Theory and Applications of Satisfiability Testing, Poster, 2005.

[64] A. Smith, A. Veneris, and A. Viglas, "Design Diagnosis Using Boolean Satisfiability," in *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 218–223, 2004.

[65] A. Smith, A. Veneris, M. Fahim Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, pp. 1606–1621, Oct 2005.

[66] M. F. Ali, A. Veneris, S. Safarpour, M. Abadir, R. Drechsler, and A. Smith, "Debugging Sequential Circuits Using Boolean Satisfiability," *Fifth International Workshop on Microprocessor Test and Verification (MTV'04)*, vol. 0, pp. 44–49, 2004.

[67] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pp. 108–125, 2000.

[68] P. Bjesse and K. Claessen, "SAT-Based Verification Without State Space Traversal," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pp. 372–389, 2000.

[69] Y. Zheng, M. S. Hsiao, and C. Huang, "SAT-based Equivalence Checking of Threshold Logic Designs for Nanotechnologies," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pp. 225–230, ACM, 2008.

[70] W. Hu, H. Nguyen, and M. Hsiao, "Sufficiency-based Filtering of Invariants for Sequential Equivalence Checking," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp. 1–8, Nov 2011.

[71] H. Nguyen and M. Hsiao, "Sequential equivalence checking of hard instances with targeted inductive invariants and efficient filtering strategies," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp. 1–8, Nov 2012.

[72] N. Goel, M. Hsiao, N. Ramakrishnan, and M. Zaki, "Mining Complex Boolean Expressions for Sequential Equivalence Checking," in *19th IEEE Asian Test Symposium (ATS)*, pp. 442–447, Dec 2010.

[73] W. Wu and M. Hsiao, "SAT-based State Justification with Adaptive Mining of Invariants," in *IEEE International Test Conference (ITC)*, pp. 1–10, Oct 2008.

[74] P. Tafertshofer and A. Ganz, "SAT based ATPG using fast justification and propagation in the implication graph," in *Digest of Technical Papers, IEEE/ACM International Conference on Computer-Aided Design*, pp. 139–146, Nov 1999.

[75] A. Parikh, W. Wu, and M. Hsiao, "Mining-guided state justification with partitioned navigation tracks," in *IEEE International Test Conference (ITC)*, pp. 1–10, Oct 2007.

[76] D. Tille and R. Drechsler, "A fast untestability proof for SAT-based ATPG," in *12th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pp. 38–43, April 2009.

[77] A. Cimatti, "Beyond Boolean SAT: Satisfiability Modulo Theories," in *9th International Workshop on Discrete Event Systems (WODES)*, pp. 68 –73, May 2008.

[78] I. Johnson, "Formal Verification with SMT Solvers: Why and How."

[79] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL (T): Fast Decision Procedures," in *Computer aided verification*, pp. 175–188, Springer, 2004.

[80] B. Dutertre and L. M. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *CAV*, pp. 81–94, 2006.

[81] P. D. Coward, "Symbolic Execution Systems - a review," *Softw. Eng. J.*, vol. 3, pp. 229–239, November 1988.

[82] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction* (R. Horspool, ed.), vol. 2304 of *Lecture Notes in Computer Science*, pp. 209–265, Springer Berlin / Heidelberg, 2002.

[83] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pp. 213–223, 2005.

[84] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pp. 263–272, 2005.

[85] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2008.*, pp. 443 –446.

[86] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test Input Generation with Java PathFinder," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 97–107, July 2004.

[87] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs," *Asian Test Symposium*, pp. 59–64, 2010.

[88] S. Prabhu, M. S. Hsiao, S. Krishnamoorthy, L. Lingappan, V. Gangaram, and J. Grundy, "An Efficient 2-Phase Strategy to Achieve High Branch Coverage," in *20th Asian Test Symposium (ATS)*, pp. 167 –174, Nov. 2011.

[89] B. Dutertre and L. D. Moura, "The Yices SMT solver," tech. rep., 2006.

[90] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, pp. 337–340, 2008.

[91] A. Griggio, "A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic," *JSAT*, vol. 8, pp. 1–27, January 2012.

[92] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, vol. 4590 of *Lecture Notes in Computer Science*, pp. 298–302, July 2007.

[93] B. Koenemann, "LFSR-coded test patterns for scan designs," in *European Test Symposium (ETS)*, 1991.

[94] S. Hellebrand, S. Tarnick, J. Rajski, and B. Courtois, "Generation Of Vector Patterns Through Reseeding Of Multiple-Polynomial Linear Feedback Shift Registers," in *Proceedings of International Test Conference*, pp. 120–129, 1992.

[95] P. H. Bardell and W. H. McAnney, "Self-testing of multichip logic modules," in *Proceedings of International Test Conference.*, pp. 200–204, 1982.

[96] I. Pomeranz, S. Venkataraman, S. Reddy, and B. Seshadri, "Z-Sets and Z-Detections: Circuit Characteristics that Simplify Fault Diagnosis," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition.*, vol. 1, pp. 68 – 73, Feb. 2004.

[97] "iwls 2005 benchmarks." http://www.iwls.org/iwls2005/benchmarks.html.

[98] D. Bakshi and M. S. Hsiao, "LFSR seed computation and reduction using SMT-based fault-chaining," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 1071–1076, 2013.

[99] J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Transactions on Electronic Computers*, vol. EC-16, pp. 567 –580, Oct. 1967.

[100] J. Savir and J. P. Roth, "Testing for, and distinguishing between failures," *In 12th Fault Tolerant Computing Symposium*, pp. 165–172, 1982.

[101] M. Chandrasekar, N. P. Rahagude, and M. S. Hsiao, "Search State Compatibility Based Incremental Learning Framework and Output Deviation Based X-filling for Diagnostic Test Generation," *J. Electron. Test.*, vol. 26, pp. 165–176, April 2010.

[102] I. Pomeranz and S. Reddy, "Output-Dependent Diagnostic Test Generation," in *23rd International Conference on VLSI Design*, pp. 3 –8, Jan. 2010.

[103] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Form. Methods Syst. Des.*, vol. 19, pp. 7–34, July 2001.

[104] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, pp. 337–340, 2008.

[105] L. M. de Moura and N. Bjørner, "Generalized, Efficient Array Decision Procedures," in *FMCAD*, pp. 45–52, 2009.

[106] Q. Peng, M. Abramovici, and J. Savir, "MUST: Multiple-Stem Analysis for Identifying Sequentially Untestable Faults," in *International Test Conference*, pp. 839 –846, 2000.

[107] M. Hsiao, E. Rudnick, and J. Patel, "Sequential circuit test generation using dynamic state traversal," in *Proceedings of European Design and Test Conference*, pp. 22 –28, mar 1997.

[108] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Dynamic state traversal for sequential circuit test generation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, pp. 548–565, July 2000.

[109] D. Krishnaswamy, M. Hsiao, V. Saxena, E. Rudnick, J. Patel, and P. Banerjee, "Parallel genetic algorithms for simulation-based sequential circuit test generation," in *Proceedings of 10th International Conference on VLSI Design*, pp. 475–481, Jan 1997.

[110] M. Elm and H. Wunderlich, "BISD: Scan-based Built-In Self-diagnosis," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1243–1248, 2010.

[111] W.-T. Cheng, M. Sharma, T. Rinderknecht, L. Lai, and C. Hill, "Signature Based Diagnosis for Logic BIST," in *IEEE International Test Conference* , pp. 1–9, 2006.

[112] S. Holst and H. Wunderlich, "A Diagnosis Algorithm for Extreme Space Compaction," in *Design, Automation Test in Europe Conference Exhibition*, pp. 1355–1360, 2009.

[113] M. Kochte, S. Holst, M. Elm, and H. Wunderlich, "Test Encoding for Extreme Response Compaction," in *14th IEEE European Test Symposium*, pp. 155–160, 2009.

[114] A. Cook, S. Hellebrand, and H. Wunderlich, "Built-in Self-diagnosis Exploiting Strong Diagnostic Windows in Mixed-mode Test," in *17th IEEE European Test Symposium*, pp. 1–6, 2012.

[115] A. Kahng and S. Reda, "New and Improved BIST Diagnosis Methods from Combinatorial Group Testing Theory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 533–543, 2006.

[116] O. Sinanoglu and A. Orailoglu, "Parity-based Output Compaction for Core-based SOCs [logic testing]," in *The 8th IEEE European Test Workshop*, pp. 15–20, 2003.

[117] H. Vranken, S. Goel, A. Glowatz, J. Schloeffel, and F. Hapke, "Fault Detection and Diagnosis with Parity Trees for Space Compaction of Test Responses," in *43rd ACM/IEEE Design Automation Conference*, pp. 1095–1098, 2006.

[118] S. Mitra and K. S. Kim, "X-compact: An Efficient Response Compaction Technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 3, pp. 421–432, 2004.

[119] T. W. Williams, W. Daehn, M. Gruetzner, and C. W. Starke, "Comparison of Aliasing Errors for Primitive and Non-Primitive Polynomials," in *ITC*, pp. 282–289, 1986.

[120] R. Aitken and V. Agarwal, "A Diagnosis Method using Pseudo-random Vectors without Intermediate Signatures," in *Digest of Technical Papers., IEEE International Conference on Computer-Aided Design. ICCAD*, pp. 574–577, Nov 1989.

[121] J. Waicukauski and E. Lindbloom, "Failure Diagnosis of Structured VLSI," *IEEE Design Test of Computers*, vol. 6, pp. 49–60, Aug 1989.

[122] J. Rajski and J. Tyszer, "Diagnosis of Scan Cells in BIST environment," *IEEE Transactions on Computers*, vol. 48, pp. 724–731, Jul 1999.

[123] J. Savir, "Salvaging Test Windows in BIST Diagnostics," in *15th IEEE VLSI Test Symposium*, pp. 416–425, Apr 1997.

[124] T. Clouqueur, O. Ercevik, and K. Saluja, "Efficient Signature-based Fault Diagnosis using Variable Size Windows," in *14th International Conference on VLSI Design*, pp. 391–396, 2001.

[125] C. Liu and K. Chakrabarty, "Failing Vector Identification Based on Overlapping Intervals of Test Vectors in a scan-BIST Environment," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 22, pp. 593–604, Nov. 2006.

[126] Y. Nakamura, T. Clouqueur, K. Saluja, and H. Fujiwara, "Diagnosing At-Speed Scan BIST Circuits Using a Low Speed and Low Memory Tester," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, pp. 790–800, July 2007.

[127] P. Wohl, J. Waicukauski, S. Patel, and G. Maston, "Effective diagnostics through interval unloads in a BIST environment," in *Proceedings of 39th Design Automation Conference.*, pp. 249–254, 2002.

[128] R. Sharma and K. Saluja, "An implementation and analysis of a concurrent built-in self-test technique," in *Digest of Papers, 18th International Symposium on Fault-Tolerant Computing (FTCS)*, pp. 164–169, June 1988.

[129] P. Drineas and Y. Makris, "Concurrent fault detection in random combinational logic," in *Proceedings of Fourth International Symposium on Quality Electronic Design*, pp. 425–430, March 2003.

[130] N. Alves, A. Buben, K. Nepal, J. Dworak, and R. Bahar, "A Cost Effective Approach for Online Error Detection Using Invariant Relationships," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 788–801, May 2010.

[131] N. Alves, Y. Shi, J. Dworak, R. Bahar, and K. Nepal, "Enhancing online error detection through area-efficient multi-site implications," in *29th IEEE VLSI Test Symposium (VTS)*, pp. 241–246, May 2011.

[132] J. Dworak, K. Nepal, N. Alves, Y. Shi, N. Imbriglia, and R. Iris Bahar, "Using Implications to Choose Tests Through Suspect Fault Identification," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, pp. 14:1–14:19, Jan. 2013.

[133] N. Alves, Y. Shi, N. Imbriglia, J. Dworak, K. Nepal, and R. Bahar, "Dynamic Test Set Selection Using Implication-Based On-Chip Diagnosis," in *16th IEEE European Test Symposium (ETS)*, pp. 211–211, May 2011.

[134] Gurobi Optimization, Inc., "Gurobi Optimizer Reference Manual," 2013.