博士論文

Automatic Test Pattern Generation Methods for Multiple Stuck-at Faults

（多重縮退故障のためのテストパタン生成手法）

令和元年１１月２９日提出

指導教員　藤田　昌宏教授

東京大学大学院工学系研究科

電気系工学専攻

37-177081

王　培坤

Abstract

As the number of transistors in the fabricated circuits becomes extremely larger, it is unavoidable for faults to happen in the fabricated chips. Not only single stuck-at faults, but also multiple stuck-at faults are likely to happen in the circuits, especially for the large scale circuit. In this thesis, we mainly focus on the combinational circuit. Multiple faults are difficult to be fully covered due to the exponentially enormous number of all possible faults. Although there are methods proposed to deal with the multiple faults, they fail to generate compact test patterns to detect all faults within an acceptable running time.

Nevertheless, there are exponentially more multiple faults than single faults in any given circuit design. However, it is shown in the previous research that only a few additional test patterns are needed to cover all of the multiple faults, if the test generation starts from the complete test set for single faults. In this thesis, we first show the case where test patterns for single faults are sufficient to cover all multiple faults, and then explain in which conditions some of the multiple faults may be overlooked. Based on this analysis, we propose a method which can efficiently generate the complete test set for double faults without traversing all the faults. Since most of the double faults can be detected by single faults' test set, the proposed method only selects the uncovered double faults by analyzing the propagation paths of single faults and then generating new test patterns only for those uncovered faults. The

experimental results show that based on the single faults' test set, the proposed method only needs to create a small number of additional test patterns to cover all double faults in most of the given circuits.

Based on the method for double faults, we proposes an incremental Automatic Test Pattern Generation method to deal with multiple stuck-at faults. Instead of traversing the entire *n* multiple fault list, the proposed method only selects the faults undetected by the existing test patterns for *n-1* faults, and then generates additional test patterns. Staring from a complete test set for single faults, the proposed method can be incrementally applied to handle all multiple faults. Moreover, since the number of undetected faults that are selected is extremely smaller comparing to the total number of the entire fault list, the proposed method can generate compact test patterns to cover all faults within an acceptable running time.

As the proposed incremental Automatic Test Pattern Generation method can be used to find redundant multiple faults, we have proposed a logic optimization method to remove the redundancy in the circuit. In order to remove as many as possible the redundancy, instead of removing the redundant single faults first, we clear up the redundant faults from higher cardinality to lower cardinality. The experiments prove that the proposed method can successfully eliminate more redundancy compared to the redundancy removal command in the synthesis tool SIS which is based only on redundant single faults.

Acknowledgements

It has been three years since I started the journey towards the Ph.D. degree. Undertaking this PhD has been truly life-changing experience for me. I have not only improved research ability, but also gained new mentors and friends. The thesis would not have been possible to do without the support and guidance that I received from many people.

My deepest gratitude goes first and foremost to my supervisor, Professor Masahiro Fujita, for his constant and illuminating guidance and encouragement. I remember that he used to said something like "You need to work harder" to encourage us to spend more time in doing research and coming up with more new ideas. He has given me freedom to pursue my research. He helped me to come up with the thesis topic and guided me to solve the problem. Under his supervision, I learned how to define a research problem, find a proper solution and finally publish it. Without his guidance, this Ph.D. degree would not have been achievable and I shall eternally be grateful to him for his assistance.

Also, I would like to express my sincere gratitude to Dr. Amir Masoud Gharehbaghi. He taught me a lot in these three years. I went to him for advice when I have questions about the implementation and optimization. He was always patient with us and spent a lot of time discussing the solution. Besides, before I submitted a paper, he always helped me check the mistakes very carefully. I want to thank him for his support and guidance.

In addition, I must thank Dr. Takashi Matsumoto, Ms. Shinobu Nagasawa and my labmates for their support. They gave me their help and time in listening to me and helping me work out my problems during the difficult time in the pass three years.

Finally, I would like to thank to my family for always believing in me and encouraging me to follow my dreams. They helped me in whatever way they could during this challenging period. Without their warm love and endless support, I would not have been where I am today and what I am today.

Table of contents

List of figures xi

List of tables xv

1 Introduction 1

1.1 Fault and Test 1

1.2 Automatic Test Pattern Generation 6

1.3 Thesis Overview 8

1.3.1 Problem and Motivation 8

1.3.2 Contribution of Thesis 10

1.3.3 Outline of This Dissertation 11

2 Related Works 13

2.1 Basic ATPG Technologies For Single Fault 13

2.2 SAT-based ATPG Technologies for Single Fault 16

2.3 Previous ATPG Technologies for Multiple Faults 17

2.3.1 Vector Pair Method 18

2.3.2 Genetic Algorithm (GA) based ATPG method 18

2.3.3 Representing the Fault List Implicitly 19
2.3.4 Generating test set based on ROBDD Structure 22
2.4 Previous Research of Logic Optimization 26
2.4.1 Basic Idea of Logic Optimization 26
2.4.2 Add Single Redundancy to Optimize Circuit 28
2.4.3 Majority-Inverter Graph Method 30
2.4.4 Logic Rewriting with Exact Databases 33
2.4.5 Logic Optimization by applying Deep Learning 35
2.5 Conclusion 38

3 ATPG Method for Double Faults 39

3.1 Concept of the Proposed ATPG Method 39
3.2 Applying Test Patterns for SSA Faults to Detect DSAFs 41
3.2.1 Definitions 41
3.2.2 When Test Patterns for SSA Faults is Sufficient to cover DSAFs . . 41
3.2.3 When Test Patterns for SSA Faults is Insufficient to cover DSAFs . 43
3.3 Proposed ATPG Method for DSAF 45
3.3.1 Focusing on undetected Faults 45
3.3.2 Proposed Fault Selection Method in Non-redundant Circuit 47
3.3.3 When Circuit has Redundancy 49
3.4 Four steps of Proposed Method 51
3.4.1 Algorithm Flow for Double Faults 51
3.4.2 Circuit Initialization 53
3.4.3 ATPG Initialization 55

3.4.4 First Filter: Checking the Path Constraint 56
3.4.5 Second Filter: Excluding the DSAFs detectable by the Generated SSA Test Patterns 61
3.4.6 Generation of Additional Test Patterns 63
3.4.7 Extensions for Multiple Faults of Larger Cardinalities 64
3.5 Experimental Results 64
3.5.1 Selected DSAFs 65
3.5.2 Performance Comparison 66
3.5.3 The Selected DSAFs in the Proposed Method 67
3.5.4 Runtime Analysis of the Proposed Method 70
3.6 Conclusion 70

4 Incremental ATPG Method for Multiple Faults **73**
4.1 Key Idea of the Proposed Method 73
4.2 Proposed ATPG Method for Multiple Faults 74
4.2.1 Definitions 74
4.2.2 ATPG Method for Triple Faults 74
4.2.3 General Method to Extend the Proposed Method form n-1 faults to n faults 79
4.2.4 Flow Chart of the Proposed Method 84
4.3 Experiment Results 85
4.3.1 Performance Comparison 86
4.3.2 Selected Faults Analysis 89
4.3.3 Runtime Analyzation 91

4.4 Conclusion 93

5 A Logic Optimization Method based on the Proposed ATPG Method for Multiple Faults 95

5.1 Introduction of Logic Optimization 95

5.2 Finding the Redundant Multiple Faults 96

5.3 Removal of the Redundant Multiple Faults from Higher to Lower Cardinality 97

5.4 Experimental Results 98

5.5 Conclusion 102

6 Conclusion and Future Work 103

6.1 Conclusion 103

6.2 Future Work 105

References 107

List of figures

1.1 Chip becomes much denser 2

1.2 VLSI Testing Process 2

1.3 Stuck-at Fault and Briding Fault 3

1.4 Truth Table When Stuck-at Fault Happens 3

1.5 Basic Operation to Detect Stuck-at Fault 4

1.6 Example of Single Stuck-at Fault 5

1.7 Example of the Test Pattern for Single Stuck-at Fault 5

1.8 Example of the Test Pattern for Multiple Stuck-at Fault 6

1.9 Combination part of s27 benchmark circuit 7

2.1 Use Test Pattern to Find the Faulty Circuit 17

2.2 Modeling stuck-at faults at a gate2[17] 20

2.3 The model can represent faults by two bits[17] 20

2.4 Modeling stuck-at faults at a gate[17] 21

2.5 ROBDD example [41] 23

2.6 Shannon expansion of the node in ROBDD [41] 24

2.7 The circuit corresponding to Fig.2.5 [41] 24

2.8 Part of the experimental results [41] 25
2.9 Redundant Circuit 28
2.10 Irredundant Circuit 28
2.11 Irredundant Circuit [16] 29
2.12 Add Redundancy [16] 29
2.13 Optimized Circuit [16] 29
2.14 Experimental Results [16] 30
2.15 Majority-Inverter Graph [4] 31
2.16 Experimental Results 1 for MIG [4] 32
2.17 Experimental Results 1 for MIG [4] 33
2.18 Experimental Results 2 for MIG [4] 33
2.19 Experimental Results 2 for MIG [4] 34
2.20 Examples of Circuit DataBase [53, 5] 35
2.21 Logic Optimization as MDP [23] 36
2.22 Experimental Results 1 of Deep Learning Method[23] 37
2.23 Experimental Results 2 of Deep Learning Method[23] 37

3.1 Fault f_2 lies in the propagation path of f_1. 43
3.2 Propagation paths of f_1 and f_2 intersect at a gate. 43
3.3 Example of redundant fault. 44
3.4 Example that DSAF is overlooked. 44
3.5 Propagation paths of two faults have no intersection. 47
3.6 One fault lies in the propagation path of another fault. 47
3.7 Path constraints of one fault is violated by another fault. 47

3.8 One fault lies in the propagation path of another fault (missing case 1) . . . 50

3.9 The proposed ATPG flow for DSAF. 52

3.10 Undetected DSAFs Selection Flow 54

3.11 Example of circuit partition 55

3.12 Path constraints of v_1. 57

3.13 f_2 violates the path constraints of v_1. 58

3.14 Path constraints of v_2. 58

3.15 f_1 violates the path constraints of v_2. 59

3.16 Double faults $\{f_1, f_2\}$ can be detected by v_2. 59

3.17 The fault indirectly violates the path constraints (missing case 2) 60

3.18 The remaining DSAFs in the list UDL after checking path constraints . . . 61

3.19 Example: when propagation path is changed 62

4.1 Three SSAFs are independent 75

4.2 Only two SSAFs mutually block the propagation path of each other 76

4.3 SSAF f_1 has two propagation paths which are blocked by f_2 and f_3, respectively 77

4.4 SSAF f_1 has one propagation path which is blocked by f_2 and f_3 77

4.5 DSAF $\{f_1, f_2\}$ is blocked by f_3 78

4.6 The proposed ATPG flow 83

List of tables

2.1 Test patterns required for SSAFs and MSAFs [17] 22

3.1 Test pattern 1001 detects both SSAFs, but it does not detect the DSAF. . . . 44

3.2 Number of DSAF overlooked by [34] 65

3.3 Number of additional test patterns generated to cover all the DSAFs 66

3.4 Total runtime (seconds) 67

3.5 Number of selected DSAFs in the proposed method 68

3.6 Number of non-redundant and redundant DSAF among all selected DSAFs 68

3.7 Number of Re-DSAF that are excluded since two Re-SSA faults are in same propagation path 69

3.8 Runtime of each step in the proposed method (seconds) 70

4.1 experiment circuits 86

4.2 Number of additional test patterns for DSAF and TSAF 87

4.3 Total runtime (minutes) 88

4.4 Test number by N-detect and proposed method 89

4.5 Number of selected TSAF in proposed method 90

4.6 Number of three types of selected TSAF 91

4.7 Runtime of each step in proposed method (minutes) 92

4.8 Runtime of fault selection, fault simulation and test generation (minutes) . . 93

4.9 The comparison of Test Generation with and without Equivalence Checking 94

5.1 Composition of the Redundant DSAF 98

5.2 Number of Gate and Redundant DSAF and SSAF 99

5.3 The Removed Gates by Optimizing Re-SSAF and DSAF 100

Chapter 1

Introduction

In this chapter, an introduction to the basic concept of the fault, test and Automatic Test Pattern Generation technologies for the combinational circuit are covered at first. Then, the thesis overview is stated, including the problem of the previous methods for multiple stuck-at faults and the motivation of this research, the contribution of this thesis, and the basic flow of the rest of the thesis.

1.1 Fault and Test

The fabricated chips are needed to be thoroughly tested and determined whether their functionalities are correct or not before shipping to consumers. This is due to the possibility that, some physical defects are accidentally introduced during fabrication, especially when more and more dense chips are produced in the past few decades, as shown in Fig. 1.1. As shown in Fig. 1.2, the fabricated chips are tested to check the correctness of their functionalities at first. The non-faulty ones are sent to consumers, while the faulty ones are

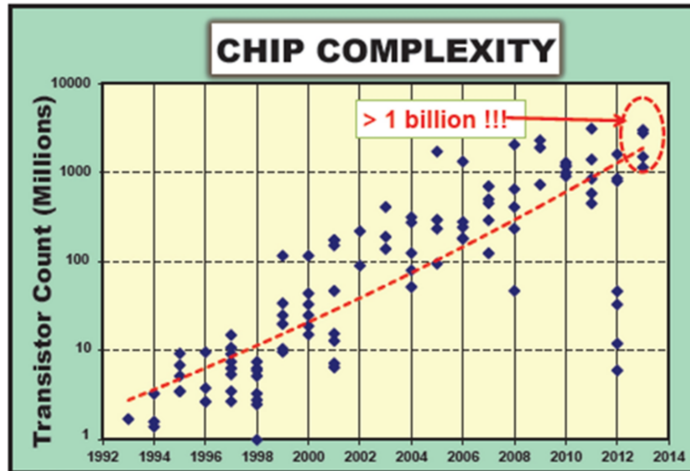


Fig. 1.1 Chip becomes much denser

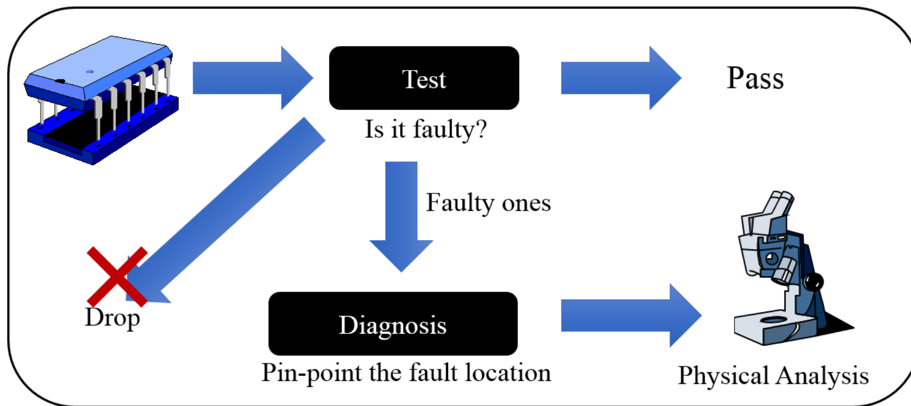


Fig. 1.2 VLSI Testing Process

dropped, or diagnosed to figure out the reason of the fault by analyzing the location and the number of faults. Notice that in the test process, instead of the locations and the number of faults, we only care about whether the circuit has at least one fault or not. In this thesis, we mainly discuss techniques about the test.

There are commonly used fault models to efficiently represent faults and generate test patterns, such as stuck-at fault, delay fault, bridge fault and others as illustrated in Fig. 1.3. The stuck-at fault model is the most commonly used model in industry because although it

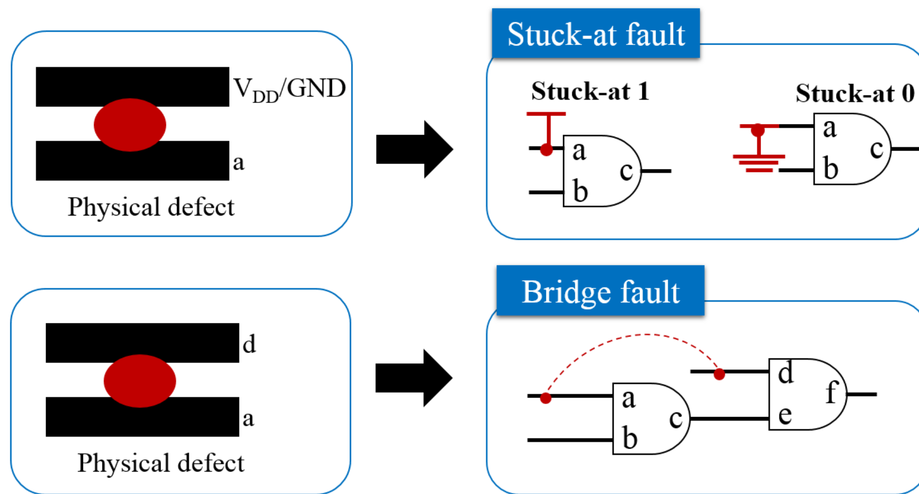
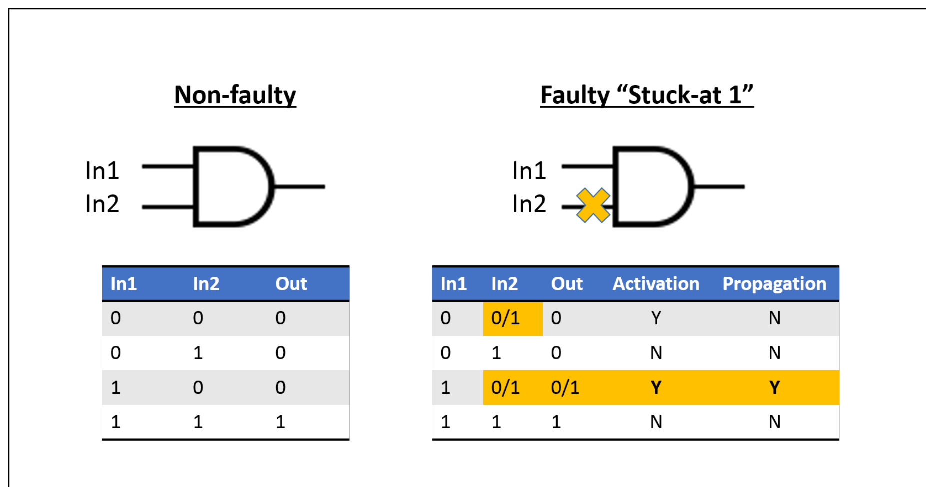


Fig. 1.3 Stuck-at Fault and Briding Fault

is simple, it covers, practically, a large variety of defects which actually may occur during fabrication. Therefore, we mainly discuss the research for stuck-at fault in this thesis.



In1	In2	Out
0	0	0
0	1	0
1	0	0
1	1	1

In1	In2	Out	Activation	Propagation
0	0/1	0	Y	N
0	1	0	N	N
1	0/1	0/1	Y	Y
1	1	1	N	N

Fig. 1.4 Truth Table When Stuck-at Fault Happens

A stuck-at fault indicates that faulty location always keeps either the constant 0 (stuck-at 0) or constant 1 (stuck-at 1) regardless the value of the input signal, as shown in Fig. 1.3. The truth table of the non-faulty and faulty circuit is shown in Fig. 1.4. The value in In1 is affected by the stuck-at 1 fault when its original value is 0. Moreover, only one of the four input configurations actually causes an incorrect value at the output of the gate.

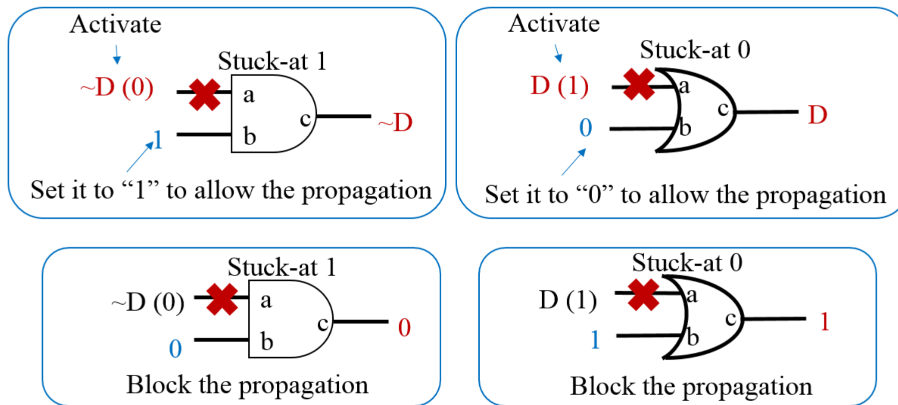


Fig. 1.5 Basic Operation to Detect Stuck-at Fault

The next question is how to determine a fabricated circuit is faulty or not. The post-fabrication circuit is essentially a black-box, which means that we can only physically manipulate and observe primary inputs (PI) and primary outputs (PO). As a result, it is impossible to examine the internal signals to find the fault. Generally, we can inspect the output values for some set of input values, which are also called test patterns. We can test the circuit by assigning the test patterns to PI and comparing the actual output values observed in fabricated circuits with the expected output values.

Fault activation (sometimes called fault sensitization) and fault propagation are two basic processes to generate test patterns. Activation indicates that setting an opposite value in the faulty location. For example, as shown in Fig. 1.5, we need to set the value in the faulty position to be 0 for stuck-at 1 fault, and 1 for stuck-at 0 fault. Given a single stuck-at fault f_i, **D (1/0)** is used to represent stuck-at 0 fault, which means that the signal value is 1 in non-faulty circuit and 0 in the faulty one. The complement of **D** is **D' (0/1)**. In order to successfully propagate the fault to a primary output, the values of the side input in the propagation path should be set to a non-controlling value, which is 1 for AND gate and 0 for OR gate, as illustrated in Fig. 1.5. We need to find the test pattern that can meet both

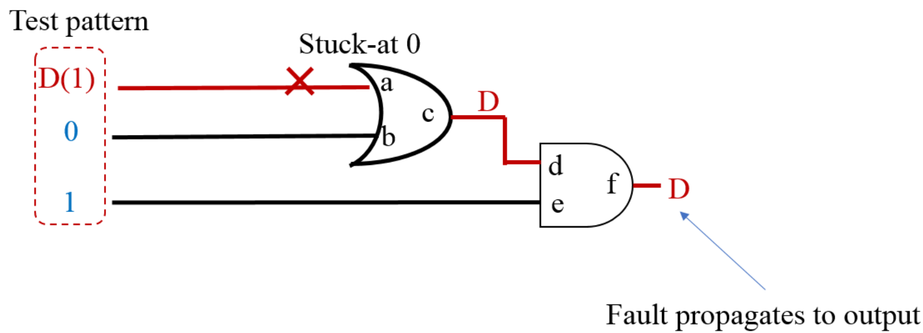


Fig. 1.6 Example of Single Stuck-at Fault

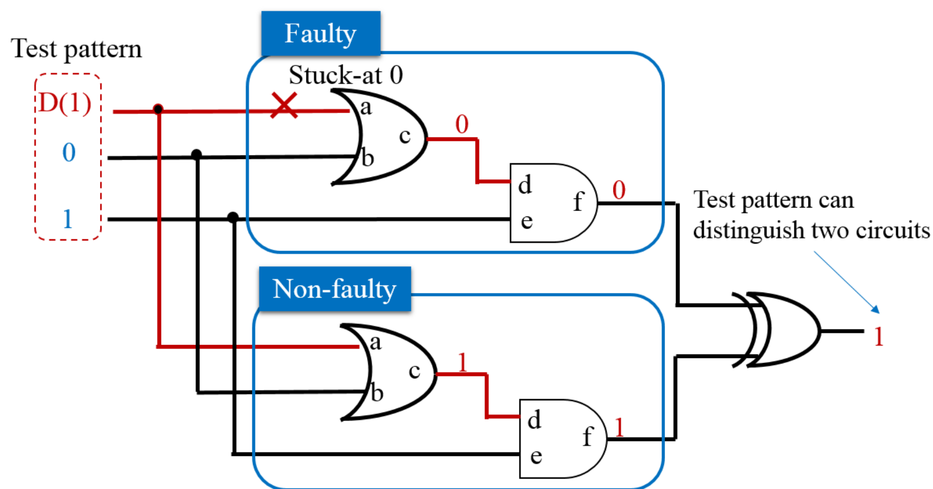


Fig. 1.7 Example of the Test Pattern for Single Stuck-at Fault

conditions at the same time; hence, the fault can be propagated to at least one primary output. When one or more faults exist on a circuit, this is simply referred to as a multiple fault, or more specifically in this case, a multiple stuck-at fault (MSAF). On the other hand, when only one fault exists on a circuit, this is called a single fault, or in the case of this thesis, a single stuck-at fault (SSAF).

Fig. 1.6 and Fig1.7 show examples of the test generation process for the SSAF. We assume that there is a stuck-at 0 fault in the wire a. The fault can be activated by assigning 1 in input a. In order to propagate the fault to a primary output, the value in b and e is assigned to 0 and 1, respectively. Finally "101" is a test pattern that can detect the stuck-at 0 fault,

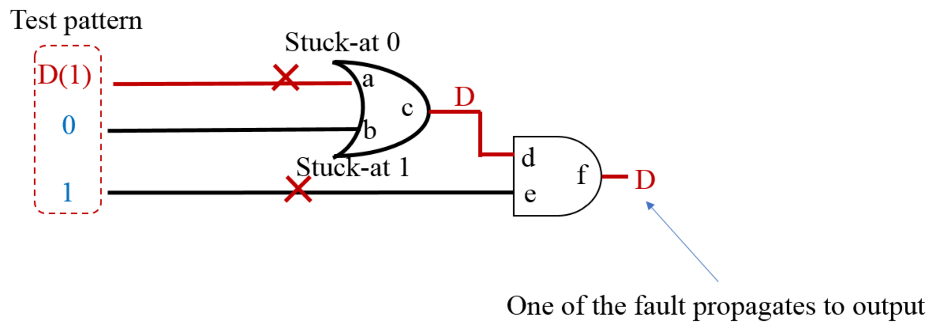


Fig. 1.8 Example of the Test Pattern for Multiple Stuck-at Fault

as it can distinguish the faulty circuit from the non-faulty one, as shown in Fig. 1.7. The Fig. 1.8 shows the example of test generation process for the MSAF. We should notice that instead of detecting all stuck-at faults, we only need to make sure that whether there is at least one fault in the circuit or not. The stuck-at 0 fault can be propagated to the output by using the test pattern "101"; hence, the multiple faults pair is detected. Another importation observation is that the test pattern for the single fault in Fig. 1.6 is the same as the multiple faults in Fig.1.8. Actually, most of the MSAFs can be detected by using the test patterns for SSAFs. The details is explained in the following chapters.

1.2 Automatic Test Pattern Generation

In order to detect all possible faults in the circuit, multiple test patterns need to be prepared. If the number of the possible faulty locations is K, the total number of the single stuck-at faults is $2 * K$, and 3^k-1 for all multiple faults, since each location may be stuck-at 0 fault, stuck-at 1 fault or non-faulty. The most naive way to cover all possible faults is preparing all input patterns. These test patterns can absolutely guarantee to detect all faults in the circuit. However, the number of test patterns increases exponentially with the number of PI.

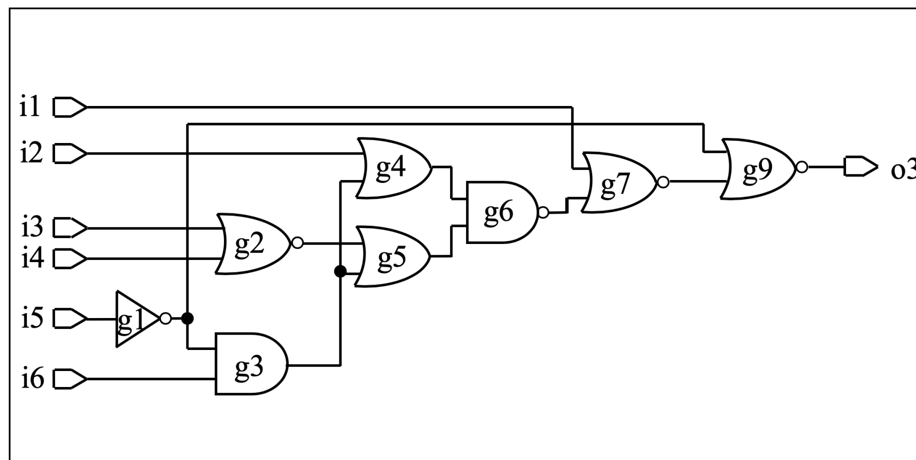


Fig. 1.9 Combination part of s27 benchmark circuit

When the number of PI is N, there are 2^N possible test patterns. It is time-consuming and impractical to check all test patterns.

Generally, less than 2^N test patterns are required to detect all possible faults. In fact, the subset of all test patterns to cover all faults is not unique. In other words, many subsets can be used to detect the faulty behavior, including the set (of all possible input patterns) itself, as well as other highly redundant subsets. Automatic Test Pattern Generation (ATPG) techniques are proposed to pick up as small as possible subset to detect all faults [18, 19, 22, 30, 38]. A smaller set of input test patterns means less inputs need to be checked, which leads to less time being spent on testing. Therefore, the main focus in ATPG research is finding the smallest possible subset of input patterns that still maintains full fault coverage. We should notice that, as mentioned in subsection 1.1, ATPG is for fault detection, not fault diagnosis. The purpose of the test patterns generated by ATPG is to determine if any fault exists somewhere in the circuit. The number, locations, and types of faults do not need to be calculated by ATPG.

Take the s27 circuit as an example, which is the smallest circuit among all ISCAS89 benchmark circuits, as shown in Fig. 1.9. The circuit has 6 inputs, 1 output, 2 AND gates, 5 OR gates. If we do not have any optimization on the test patterns and consider a potential fault at every input/output of every gate, there are (7 gates) x (3 locations) x (2 stuck-at faults) = 42 possible SSAFs and 2^6 = 64 test patterns. However, all of the 42 faults can be detected by checking just 5 specific test patterns [14]. The total number of the MSAFs is much larger. Recall that there are 21 different fault locations (number of gates multiply number of inputs and outputs). Each location has three possible states: stuck-at 1, stuck-at 0, and non-faulty. In other words, if multiple faults are considered, there would be 3^{21}-1 (10^{10}) possible multiple faults. The "-1" is to exclude the case when the entire circuit is non-faulty. However, only 4% of all possible input patterns need to be checked to determine whether or not the circuit is faulty by utilizing the ATPG technologies.

1.3 Thesis Overview

1.3.1 Problem and Motivation

With the increasing circuit density, not only the SSAF, but also more MSAF happen in the fabricated circuits. The MSAF indicates the case that several SSAFs happen on different nets of the circuit, simultaneously, due to multiple defects. Although the test patterns for single faults can detect most of the multiple faults, there are single faults that violate the propagation of each other. As the result, the single test patterns may not be sufficient to detect all multiple faults [17]. In addition, paper [6, 52] show that among 453 failing devices, less than 40% cases can be modeled using single stuck-at fault model, while the remaining 60%

cases require to be modeled with multiple stuck-at fault. In other words, without applying the multiple fault model, we cannot generate sufficient test patterns to cover all faults in those devices. Furthermore, paper [29] illustrates that multiple fault analysis is required for test generation and fault simulation in some situations. For instance, we can further improve the circuit optimization if we can find and remove not only the single redundancy, but also the redundant MSAFs. In addition, the diagnostic procedure can be improved if we can generate the test patterns for MSAFs constructed from the suspected fault set identified by the test patterns of SSAF.

Nevertheless, in most of the cases, the single test patterns cannot cover all MSAFs just like s27 circuit. In fact, instead of taking care of all MSAFs, most of the ATPG algorithms only consider the SSAF for two reasons. First, the problem space when considering all MSAFs is simply too large. In contrast with the SSAFs, MSAFs in the large circuit are very difficult to be completely checked due to its huge number of fault combinations. Current methods can generate tests for circuits of up to tens of thousands of gates, but this scale is far too small for practical use. It is even not at all easy to simply list up all of such fault combinations. Second, in most cases in recent history, the SSAF coverage has been enough. Even though MSAFs are not considered, a large number of them are covered anyway by the tests generated for SSAFs. Nevertheless, as the number of transistors in a chip becomes huge, the MSAF actually occur after the fabrication of chips. In the case that a MSAF which is not tested gets fabricated onto a circuit, it will result in a defective product that wrongly passes the testing phase.

Although there are methods proposed to deal with the multiple faults, they fail to generate compact test patterns to detect all faults within an acceptable running time. Furthermore,

some of their generated test patterns fail to cover all multiple faults. In order to solve these problems, this thesis proposes an incremental Automatic Test Pattern Generation method to deal with multiple stuck-at faults.

1.3.2 Contribution of Thesis

As we mention in previous sections, it is very hard to handle all multiple faults. In order to generate a sufficient test set to cover all faults, it may take a long time due to the exponential large number of faults. Moreover, the generated test patterns may be quite redundant without the compression and optimization. The experimental results of the previous methods for multiple faults indicates that they cannot generate a compact test set to cover all faults within acceptable runing time. Therefore, in this thesis, we propose an incremental ATPG methods to deal with multiple faults, and discuss the application of the proposed method. The contributions of this thesis are summerized as follows.

1) We propose a method that can efficiently generate the complete test set for double faults without traversing all the faults.

2) Based on the idea of the double faults method, we propose an incremental Automatic Test Pattern Generation method to deal with all multiple stuck-at faults.

3) We present a new logic optimization method by using the proposed incremental ATPG method.

1.3.3 Outline of This Dissertation

In chapter 2, some related works are covered, including detailed explanations of the most basic and well-known ATPG techniques.

In chapter 3, the ATPG method for double stuck-at faults is proposed.

In chapter 4, the incremental ATPG method for multiple faults and its implementation are discussed.

In chapter 5, the proposed logic optimization method is introduced.

Finally, in Chapter 6, concluding remarks and the future of this research will be considered.

Chapter 2

Related Works

In this chapter, we will discuss the previous works that significantly impact the field of research. Basic ATPG technologies for the single fault are present at first, including the D-algorithm, PODEM and FAN. Next, we explain the satisfiability (SAT) technologies and its background, which is the test generation method utilized in our proposed method. Then, some previous works which also attempt to extend ATPG for detecting MSAFs are presented. Finally, we introduce previous research about the logic optimization.

2.1 Basic ATPG Technologies For Single Fault

In 1966, J.P. Roth of IBM proposed D-algorithm, which is the first ATPG algorithm [30, 38]. As previously stated, it is the basis from which many modern ATPG methods are derived.

Two circuits are considered by the D-algorithm. One is the non-faulty circuit, and the other one is the faulty circuit, which is exactly the same as the non-faulty circuit, with

the exception that there exists a fault at some given location. In D-algorithm, there are 3 additional values over 2 initial binary values (0 / 1) as follows.

1) 0 : logic 0 in both the faulty and non-faulty circuit.

2) 1 : logic 1 in both the faulty and non-faulty circuit.

3) X: no value assigned yet.

4) D: logic 1 in the good circuit, logic 0 in the bad circuit.

5) D': logic 0 in the good circuit, logic 1 in the bad circuit.

The faulty signal is set to D (stuck-at 0) or D' (stuck-at 1). In order to observe the faulty signal, the D-algorithms find a test pattern to propagate one of the faulty signal to a primary output. This is the reason why we call it D-algorithm.

The process of the D-algorithm is as follows. First, the search space, which is the circuit itself, is constructed. The value of each node is assigned to X at the beginning. Then, we place D or D' in the fault location depending on the value of the fault. Next, in order to find a sufficient test pattern that can detect the fault, the algorithm explores backward to PI and forward to PO to activate the fault and propagate it to output, respectively.

As for the activation process, after all nodes are assigned X and the faulty location is set a proper faulty value, if any input or internal has implication on the faulty location, they will be set to a value that can activate the fault. Otherwise, we set a random value in these signals. For instance, if an AND gate has a stuck-at 0 fault in the output, we activate it by setting the value in both inputs to 1. On the contrary, if the AND gate has a stuck-at 1 in the output, it can be activated as long as one of the input values is 0, and we can assign a random

value in the other input. In the process of activation, if there is any conflict with the current assignment, the algorithm will backtrack, until all values in the circuit are set and no conflict is in circuit.

After the backward process for the fault activation is completed, we start the forward implication to propagate the fault to output. A list of nodes that have at least one input with D or D' and output value of X is stored, which need to be checked and assigned a value. The list is called D-Frontier. If we can propagate the fault to an output without any conflict, the input value is a sufficient test pattern to detect the fault. However, if D-Frontier is traversed and the fault cannot reach any PO, once again, we need to backtrack to activation process and find a new set of values to activate the fault. Then we can propagate the fault in new D-Frontier again.

D-algorithm can guarantee to find a test pattern if the fault is not redundant, since it tries to traverse the entire searching space. However, it has no heuristic to reduce the backtracking time. Obviously, the test generation process may take a long time in the case that the search space is very huge and there are many backtracks. Particularly, the redundant fault, which cannot be detected by any possible input patterns, needs a long processing time, since we have to exhaust the whole searching space. It is incredibly time-consuming since all combinations of the paths are exhausted to find a proper input pattern.

PODEM (Path-Oriented Decision Making) was proposed by Goel in 1981 [21]. The first main difference between this algorithm and D-algorithm is the selection of the search space. D-algorithm includes the entire circuit as part of the search space, but PODEM only includes PI. This is possible based on the observation that every node in the circuit is essentially just some function of some set of PI. Furthermore, if a conflict occurs for a specific assignment,

only one other assignment, the opposite one, needs to be checked. This greatly reduces the search space, as well as the number of required backtracks.

Finally, the last of the structural techniques to be examined is FAN, a method proposed by Fujiwara and Shimono in 1983 [18, 19]. FAN is very similar to PODEM, with the main difference being the special status which is given to fanout wires. In this thesis, the term fanout will refer to when the output signal of a gate splits and goes to multiple gate inputs. Consequently, if a gate is said to have fanout, this also refers to its output signal going to multiple gate inputs.

2.2 SAT-based ATPG Technologies for Single Fault

Next we will explain the SAT-based ATPG method. Before that, it is necessary to explain the SAT [9]. Generally, satisfiability problems has two solutions, either the solution existing (satisfiable, or SAT), or the solution does not exist (unsatisfiable, or UNSAT). More specifically, given a set of clauses made up of literals in Conjunctive Normal Form (CNF), the SAT technologies are trying to check whether there are some input values that can make the output of the CNF expression becoming true. The CNF expression of digital logical terms is the Product of Sums. Take the CNF expression (W | X) & (Y | !Z) as an example. The expression is SAT since one solution is X = true and Y = true. Although SAT is a NP-complete problem, it can be handled by the modern SAT-solver as they are powerful and efficient. Modern SAT-solver has a wide variety of applications. Since a logical circuit can be converted to a CNF expression, we can applied the SAT-solver to the test generation process.

One simple way to generate the test pattern for a fault is to have two copies of the circuit, one circuit is non-faulty, and the other one is faulty. The outputs of the circuits are connected by an XOR gate to compare their output values, as shown in Fig. 2.1. If the entire circuit is converted to a CNF expression, we can generate a test pattern to detect the fault by using the SAT-solver. However, this method is too naive to handle the large scale circuits. Actually, there are efficient SAT-based technologies that can deal with single stuck-at faults in the chip even with industrial scale [44, 42, 15, 13].

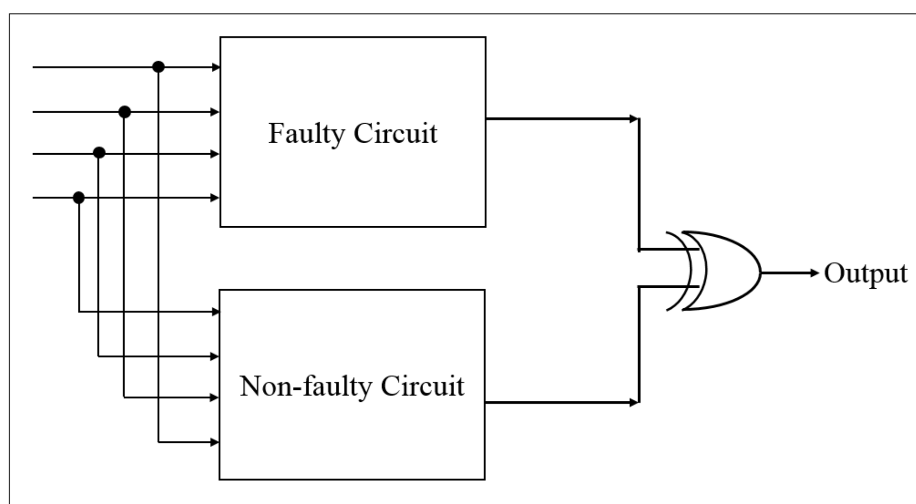


Fig. 2.1 Use Test Pattern to Find the Faulty Circuit

2.3 Previous ATPG Technologies for Multiple Faults

As previous stated, most of the previous ATPG techniques, including SAT-based ATPG methods, mainly focus on the single fault. Actually, there are technologies utilized to obtain the test patterns for multiple faults, although they cannot generate sufficient test patterns to cover all multiple faults within an acceptable time. Specifically, in most of the previous methods, the gate number of the circuits they can handle is smaller than 3,000. Furthermore,

in the worst case, the fault coverage of their generated test patterns may not be more than 80%.

2.3.1 Vector Pair Method

Inspired by the vector pair method for diagnosis [12], the authors in [27, 26, 25] propose a parallel vector pair method to generate the multiple test set. It repeatedly analyzes the vector pairs until all faults are covered or the number of the analyzed vector pairs reaches the upper limit. This repetition may take a long time if the circuit size is large or some MSAFs are hard to be detected by analyzing vector pairs. Authors in [1] also apply vector pairs to detect multiple faults. It first generates pairs of input vectors to cover a part of faults and then employs the branch and bound procedure to detect the remaining faults. This method may also result in a long processing time when many remaining faults need to be processed by the branch and bound procedure.

2.3.2 Genetic Algorithm (GA) based ATPG method

Paper [6] presents a Genetic Algorithm (GA) based ATPG method to find a compact set for multiple faults. The method tries to obtain the test patterns that can cover more faults by repeatedly performing selection, crossover and mutation operations. GA is a search technique that can be used to search problems and find optimal solution. The chromosomes in GA are substituted for the test patterns. The fitness function is used to determine how good the current test patterns is. The method tries to reach optimal solution by repeating the process of selection, evaluation, reproduction and replacement. The optimal solution in this problem

is the test patterns that can detect all given multiple faults. It needs to analyze many test patterns and faults to get the optimal solution.

1) Selection:

Initially, a set of test patterns (chromosomes) are picked up. The method randomly selects maximum possible set of test patterns. For a circuit of n inputs, the maximum possible number of test patterns is 2^n test patterns. Then, the fitness values of each initial test pattern are calculated.

2) Evaluation:

The fitness value of a generated test pattern is evaluated by the number of faults it can detect.

3) Reproduction:

After the evaluation process, new test patterns that derived from the initial test patterns by GA operations like crossover and mutation are reproduced.

4) Replacement:

The older test patterns with lower fitness value are replaced by the newer test patterns, while the older ones with higher fitness value are retained.

2.3.3 Representing the Fault List Implicitly

Traditionally ATPG processes include pattern generation as well as fault simulation in order to eliminate detectable faults with the current sets of test patterns from the sets of target faults. The problem here is the fact that fault simulators represent all faults explicitly. Therefore, fault simulators do not work if the numbers of fault combinations becomes exponentially large which is the case if we target all of multiple fault combinations.

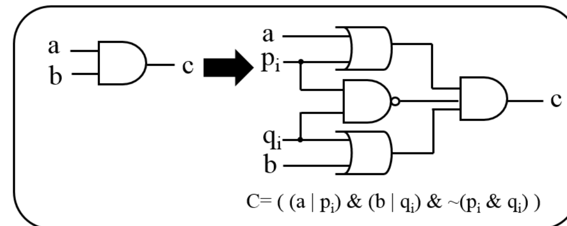


Fig. 2.2 Modeling stuck-at faults at a gate2[17]

4 Cases	p_i q_i	c
No fault	0 0	a and b
b sa-1	0 1	a
a sa-1	1 0	b
a/b/c sa-0	1 1	0

Fig. 2.3 The model can represent faults by two bits[17]

Authors in [17] propose an implicit way to efficiently represent all combinations of multiple faults. Fault lists are managed in implicit ways, and all of the ISCAS89 circuits can be processed. For example, in order to model stuck-at faults at a gate, the proposed method introduce the logic circuit (or logic function) with parameter variables, p, q as shown in Fig.2.2 and Fig.2.3. Here the target gate is an AND gate and its output is replaced with the circuit shown in the Fig. In this case, there are 3^3-1=26 faults combination, while it can be represented by only two bits in this model. It is obvious that the proposed modeling method can represent more multiple faults, which means it is much easier to cover all faults in larger design.

Stuck-at faulty behaviors for each location are realized with these additional circuits. That is, circuits with additional ones can simulate the stuck-at 1 and 0 effects by appropriately setting the values of p, q, r, ... For m possibly faulty locations, m of p, q, r, ... variables are used. In order to deal with multiple faults, these circuits for modeling various faults should be

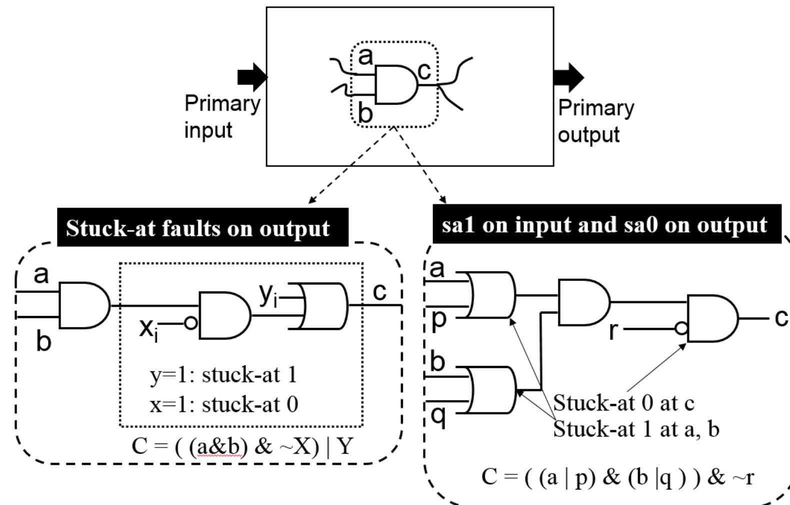


Fig. 2.4 Modeling stuck-at faults at a gate[17]

inserted into each gate in the circuit. By introducing appropriate circuit (or logic functions) with parameter variables, varieties of multiple faults can be formulated in a uniform way, such as the examples shown in Fig.2.4.

The experimental results illustrate that given a complete test set for SSAFs, the number of additional test patterns necessary to cover all MSAFs is not significantly larger, as shown in Table 2.1. The algorithm first attempts to use the single test patterns, as shown in the second column to detect all multiple faults, which succeeds in some circuits such as s298 and s386. If that fails, it will iteratively generate a new test pattern for one of the remaining faults undetected by single test vectors and then delete the faults covered by the new test pattern, until the implicitly represented list becomes empty. Obviously the algorithm needs to go through the entire fault list to create the complete test set, which is impractical in large circuits. The third and fourth columns show the numbers of the final MSAF test patterns and the additional test vectors, respectively.

Table 2.1 Test patterns required for SSAFs and MSAFs [17]

Circuit	Given Test Patterns for SSAF	Generated Test Patterns for MSAF	Additional Necessary Test Patterns
s298	28	28	0
s386	64	64	0
s400	28	30	2
s444	25	26	1
s820	99	99	0
s832	101	104	3
s1196	117	117	0
s1238	130	153	23
s1423	25	31	6
s1488	108	108	0
s1494	110	112	2
s5378	102	108	6
s9234	134	297	163
s13207	250	319	69
s15850	116	141	25
s35932	30	103	73
s38417	120	182	62
s38584	174	197	23

2.3.4 Generating test set based on ROBDD Structure

Exhaustive testing of a circuit-under-test would cover complex faults including MSAFs and bridging faults, however it is impractical. A practical approach called pseudo-exhaustive testing was proposed. This method was aimed at Built-In Self-Test architectures. Pseudo-exhaustive testing strategy has a potential application to a ROBDD (Reduced Ordered Binary Decision Diagram) based structure which is inherently modular in nature. There is a ROBDD based synthesis [41] which is possible to detect all MSAFs. It proves that the generated the number of MSAF test patterns is small than three times of SSAF test patterns, which means that the test set is not exponentially huge.

First is about BDD. One of the methods to describe Boolean functions is a data structure called Binary Decision Diagram or BDD. For the ROBDD, since the variable ordering is

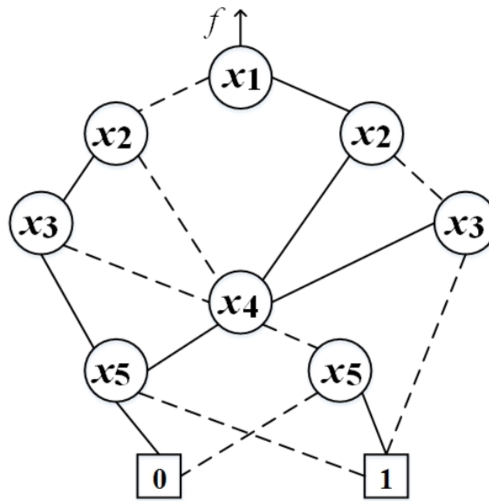


Fig. 2.5 ROBDD example [41]

fixed, the representation of the Boolean function is canonical. A circuit with m outputs will have a ROBDD with m root nodes and two terminal or leaf nodes: leaf node-0 and leaf node-1. This would be a shared ROBDD. For the current work we consider a single output function, hence a single output ROBDD as shown in Fig.2.5.

Every node in a BDD has two sucessors, one each for values 1 and 0, also known as high successor and low successor respectively. When the isomorphic sub-graphs of a BDD are combined, the resultant is a Reduced Ordered Binary Decision Diagram or ROBDD. Each ROBDD node v can be represented by the Shannon expansion as shown in Fig.2.6. The composite circuit for the ROBDD in Fig.2.5 is shown in Fig.2.7.

As it is mentioned in this proposed method, a larger circuit can be segmented into partitions having a few inputs so that they can be tested exhaustively for those inputs. This is the principle of pseudoexhaustive testing. Each node in a ROBDD is implemented using a sub-circuit. This makes ROBDD based circuits inherently segmented. The entire circuit is composed of 2:1 muxes making the mux a natural candidate for the partition.

Two theorems are proved in this thesis:

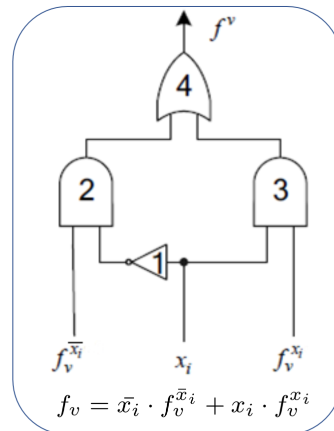


Fig. 2.6 Shannon expansion of the node in ROBDD [41]

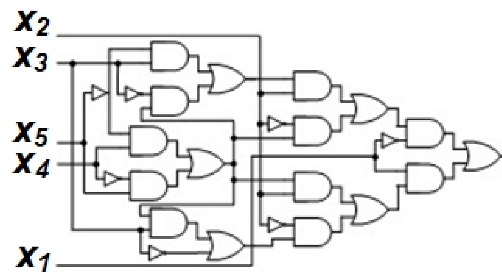


Fig. 2.7 The circuit corresponding to Fig.2.5 [41]

1. Though exhaustive values cannot be applied to the node, four values can definitely be applied to each node. These four values are enough to detect all irredundant multiple faults of the partition under test (2:1 mux).

2. A ROBDD with N nodes will have N partitions and hence it will require 4N test vectors to test the entire circuit. ROBDD or DSOP manipulations that are required to derive the test vectors can be done in polynomial time. Note that the actual number of test vectors required are much less than 4N. If N is the number of nodes comprising the ROBDD that represents circuit behavior, the upper bound on the total number of test vectors required to detect all MSAFs for the circuit synthesized using the 2:1 mux is 3N.

From these theorems we can understand that, the proposed method can generate test patterns for multiple faults which are not exponentially huge. It is also proved in the

Combinational benchmarks	PI/PO	Robdd nodes	DSOP terms	SSAF tests	MSAF tests	Base area (logic gates)	BDD area (logic gates)
b1	3/ 4	6	6	5	6	44	28
C17	5/ 2	9	7	7	5	16	21
decod	5/ 16	80	16	17	32	103	103
rd53	5/ 3	21	35	12	49	230	99
xor5	5/ 1	5	16	10	32	144	34
cm138a	6/ 8	48	48	10	3	59	52
5xp1	7/ 10	83	74	21	72	470	200
con1	7/ 2	16	9	11	9	49	56
rd73	7/ 3	35	147	28	223	968	233
z4ml	7/ 4	28	59	15	63	431	94
f51m	8/ 8	58	78	40	75	519	279
misex1	8/ 7	61	33	18	24	162	184
mm4a	8/ 4	439	508	184	1049	634	1820
rd84	8/ 4	50	294	28	405	1919	297
sqrt8	8/ 4	39	42	22	35	271	160
9sym	9/ 1	24	148	29	123	807	188
clip	9/ 5	141	150	66	183	1222	750
ldd	9/ 19	142	61	42	34	288	372
alu2	10/ 6	177	156	131	291	1408	977
x2	10/ 7	46	28	21	19	135	161
sao2	10/ 4	116	75	43	161	656	391
cm85a	11/ 3	50	48	19	121	106	153
cm151a	12/ 2	36	17	24	26	111	123
alu4	14/ 8	760	635	480	1452	8758	3589
cu	14/ 11	78	22	32	59	162	171
misex3	14/ 14	871	1306	272	2316	12391	3014
cm163a	16/ 5	35	27	21	20	93	98
cmb	16/ 4	48	26	18	25	126	58
pdc	16/ 40	4706	6062	230	1829	23242	3800
pm1	16/ 13	62	37	17	25	140	130
t481	16/ 1	30	481	51	887	3162	214
table5	17/ 15	1819	551	308	1284	2839	3704
tcon	17/ 16	24	24	17	7	136	112

Fig. 2.8 Part of the experimental results [41]

experimental results as shown in Fig. 2.8. The first colomn indicates the benchmark circuit under consideration. Number of SSAF tests are listed in column 5 and MSAF tests in column 6. The MSAF test vector set never exceeds the upper bound of 3 * N, N being the number of ROBDD nodes.

However, it should be noted that the test methodology presented in this paper is restricted to the circuit which can be represented by ROBDD. In circuits, where the number of paths are very large and generating ROBDD is increasingly difficult, the test generation approach of this thesis does not hold good.

There are also other authors proposing the BDD based method for multiple faults detection. Authors in [33] use the Reduced Ordered Binary Decision Diagrams (ROBDD) as a model to detect the MSAFs. The single test patterns are generated at first, and then they are transformed into MSA test patterns. In [46, 47], Structurally Synthesized BDD (SSBDD) is employed as the model with the double topology to deal with MSAFs. Both of the experimental results of the BDD based methods show that their generated multiple test sets are usually three times more than the single test sets.

2.4 Previous Research of Logic Optimization

2.4.1 Basic Idea of Logic Optimization

Many efforts have been made to develop the optimization algorithms. Logic optimization methods are divided into various categories, including the optimization based on circuit representation, circuit characteristics and type of execution. For example, the circuit may be optimized if we transfer a two-level logic to a multi-level logic. We assume that there are two functions F1 and F2:

F1 = AB + AC + AD,

F2 = A'B + A'c + A'E

where the above 2-level circuit takes six product terms and 24 transistors in CMOS representation. A functionally equivalent representation in multilevel can be:

P = B + C

F1 = AP + AD

F2 = A'P + A'E

where the number of levels here is 3. The total number of product terms and literals reduce because we share the term B + C.

In addition, we can do the circuit minimization in Boolean algebra, where we are trying to obtain the smallest logic circuit that can represents a given Boolean function or truth table. The methods such as Karnaugh-map, Binary Decision Diagram are used to optimize the algebra. Karnaugh-map (K-map) [28, 24] is a graphical technique for circuit minimization in Boolean algebra. We can hand-calculate the proposed graphical representation of a Boolean function and its reduction by using K-map. Using different colour schemes,multiple output functions involving the same inputs can be represented on a single K-map. Binary decision diagram(BDD) [31, 2] is also a graphical method that is easy enough to implement and visualise. We often use the BDD as a data structure to both represent the Boolean functions and to perform the operations efficiently. The usability of this approach is not constrained by the number of inputs. In addition, the Quine–McCluskey algorithm [36, 37] is functionally identical to Karnaugh mapping and is a deterministic way to check that the minimal form of a Boolean function has been reached. The Quine–McCluskey algorithm is easy and efficient to be implemented because of its tabular form, but, not being graphical, it is not as simple or as intuitive as Karnaugh-maps for use by the designer.

Here is an example for logic optimization in Boolean algebra. We assume that there is a circuit with the function AB' + A'B, which has two AND gates, one OR gate, and two inverters. The circuit can still be simplified. We can find the circuit simply represents the function A != B. The optimized circuit includes only one Exclusive OR gates, whose size are greatly reduced.

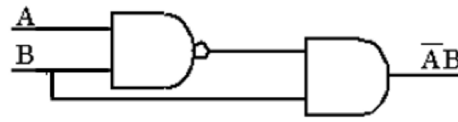


Fig. 2.9 Redundant Circuit

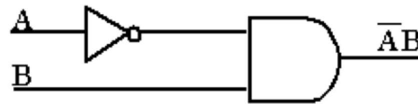


Fig. 2.10 Irredundant Circuit

2.4.2 Add Single Redundancy to Optimize Circuit

Single redundancy removal has achieved certain success for improving testability and optimizing logic for large multilevel combinational and sequential networks. For example, here is a circuit with the function (A+B)'B, as shown in Fig. 2.9. The circuit has some unnecessary parts, in other words, redundant sub-circuit. We can remove the redundancy and simplify the function to A'B, as shown in Fig. 2.10, whose logic is greatly simplified.

However, most of the circuits are compact and they have no more single redundancy to be removed. In order to further optimize the circuit, authors propose a new redundancy removal method in [16]. The new method is based on Automatic Test Pattern Generation (ATPG) techniques. This new method, named Redundancy Addition and Removal, optimizes the network through iterative addition and removal of redundant connections. Adding redundant wires to a network may cause one or multiple existing irredundant wires and/or gates to become redundant. If the amount of added redundancies is less than the amount of created redundancies, the transformation of adding followed by removing redundancies will result in a smaller network.

Take the circuit given in Fig. 2.11 as an example. Initially, this circuit is irredundant. If we add a cnnection from the output of gate g5 to the input of gate g9, the functionality of

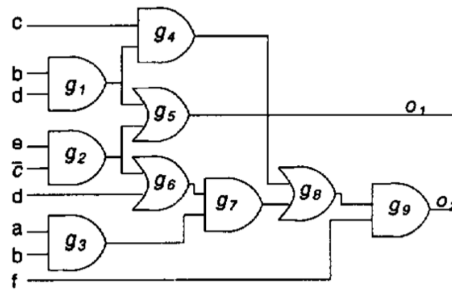


Fig. 2.11 Irredundant Circuit [16]

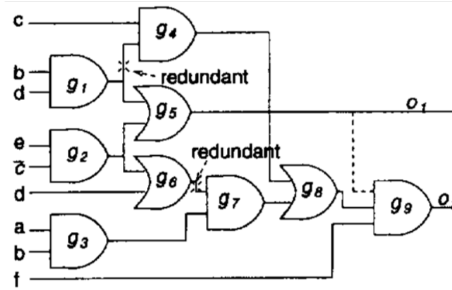


Fig. 2.12 Add Redundancy [16]

the circuit does not change. In other words, the added connection is redundant. However, the addition of the connection activate two originally irredundant wires as redundant ones as shown in Fig. 2.12. After removing these two wires and associated gates that either become floating (g6) or have a single fanin (g4 and g7), the circuit can be greatly optimized as shown in Fig. 2.13.

The experimental results are shown in Fig. 2.14. The table shows the results for some of the MCNC combinational benchmark circuits. The second column indicates the initial

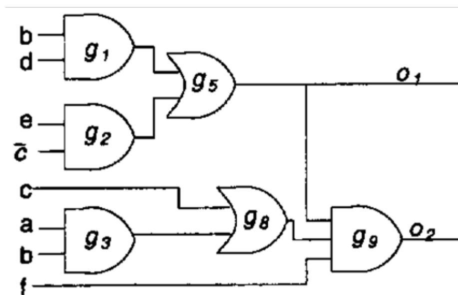


Fig. 2.13 Optimized Circuit [16]

Name	Initial		Proposed Method	
	#G	#C	#G	#C
C1355	378	859	374	837
C1908	442	1090	323	784
C2670	678	1827	531	1520
C3540	878	2172	727	1810
C432	125	380	78	271
C499	378	859	374	837
C5315	1751	3927	1408	3201
C6288	1888	3840	1885	3834
C7552	2177	4795	1500	3385
C880	259	683	240	643
alu2	284	685	208	509
alu4	567	1320	415	1006
apex6	548	1447	498	1327
apex7	179	497	141	412
dalu	1847	4159	808	2007
frg2	1111	3058	560	1734
k2	278	3066	262	2232
pair	1086	2866	977	2594

Fig. 2.14 Experimental Results [16]

size of the circuits. The remaining columns show the number of the gates using the proposed optimization method. The results show that the method can reduce redundancy in circuit.

2.4.3 Majority-Inverter Graph Method

How to efficiently represent the logic is one of a key factors to optimize the circuit. Authors in [4] propose an method, which applys the majority (MAJ) and inversion (INV) as basic operation, rather than using AND, OR and etc. The Majority-Inverter Graph (MIG) proposed by the authors consists of three-input nodes MAJ and inverter. MIG contains any AND/OR/Inverter Graphs (AOIGs). By introducing a novel Boolean algebra, the author provides a native manipulation of MIGs. A complete axiomatic system is constructed by a set of five primitive trnaformations. We can explore the entire MIG representation space by utilizing a sequence of such primitive axioms.

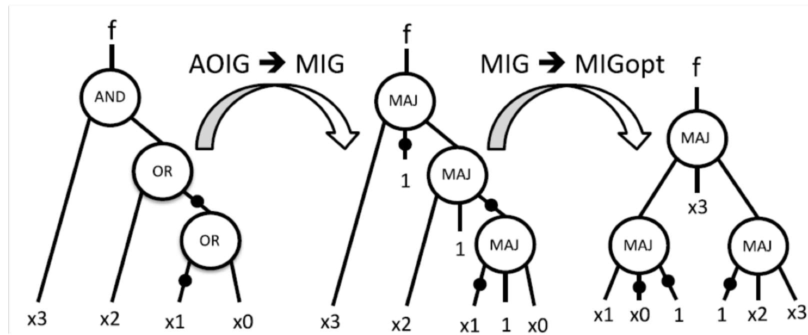


Fig. 2.15 Majority-Inverter Graph [4]

For the definition of MIG, it is an homogeneous logic network with indegree equal to 3 and with each node representing the majority function. In terms of representation expressiveness, the elementary bricks in MIGs are majority operators while in AOIGs there are conjunctions (AND) and disjunctions (OR). It is worth noticing that a majority operator M(x, y, z) behaves as the conjunction operator AND(x, y) when z = 0 and as the disjunction operator OR(x, y) when z = 1. Therefore, majority is actually a generalization of both conjunction and disjunction. Recall that M(x, y, z) = xy + xz + yz. This property leads to the following theorem.

Fig. 2.15 depicts an MIG representation example for f = x3 (x2 + (x1' + x0)'). The starting point is a traditional AOIG. Such AOIG has three nodes and three levels of depth, which is the best representation possible using just AND/ORs. The first MIG is obtained by a one-to-one replacement of AOIG nodes by MIG nodes. As shown by Fig. 2.15, a better MIG representation is possible by taking advantage of the majority function. In this way, one level of depth is saved with the same node count.

The results of the proposed method are shown in Fig. 2.16, 2.17, and 2.18. The Fig. 2.16 summarize the results for logic optimization. According to the experimental results, the average depth of MIGs is 18.6% smaller than AIGs and 23.7% smaller than decomposed

Logic Optimization		MIG				AIG			
Benchmarks	I/O	Size	Depth	Activity	Time	Size	Depth	Activity	Time
C1355	41/32	481	18	133.60	0.1	392	18	126.36	0.1
C1908	33/25	459	23	124.98	0.1	363	25	159.08	0.1
C6288	32/32	2237	86	784.62	0.2	2045	94	797.91	0.3
bigkey	487/421	4299	9	789.02	0.7	4834	9	846.57	0.5
my_adder	33/17	265	19	58.15	0.1	137	33	49.86	0.1
cla	129/65	1028	24	363.57	0.2	902	38	329.17	0.1
dalu	75/16	1443	21	283.12	0.1	1116	30	264.92	0.1
b9	41/21	97	6	16.95	0.1	84	7	16.65	0.1
count	35/16	176	7	32.77	0.1	127	19	18.87	0.1
alu4	14/8	1380	14	237.38	0.1	1421	14	249.52	0.1
clma	416/115	12449	42	3626.38	1.2	12928	46	3712.38	1.1
mm30a	124/120	1174	101	209.52	0.3	1004	125	164.49	0.2
s38417	1494/1571	8260	22	1932.78	0.8	8053	25	1854.26	1.0
misex3	14/14	1323	13	233.09	0.2	1274	14	209.27	0.1
Average	212/176	2505.1	**28.9**	630.42	0.30	2477.1	**35.5**	628.52	0.28

Fig. 2.16 Experimental Results 1 for MIG [4]

BDDs. The average size of MIGs is roughly the same than AIGs, just 0.9% of difference, but 2.1% smaller than decomposed BDDs. The average activity of MIGs is again the same as AIGs, just 0.3% of difference, but 3.1% smaller than decomposed BDDs. Fig. 2.17 represents the result in 3D (size, depth, activity) space. Using a size· depth· activity figure of merit, MIGs are 17.5% better than AIGs and 27.7% better than decomposed BDDs. As for the runtime, MIGs is slightly longer than ABC tool (+7.1%) but 68% shorter than BDS. Fig. 2.18 illustrates the experimental results for MIG-based logic synthesis. On average, the MIG flow generates delay, area, power estimated metrics that are 22%, 14%, 11% smaller than the best academic/commercial counterpart. Fig. 2.19 shows the results in 3D space (area,delay,power). While, in logic optimization, MIGs were mainly shorter than AIGs, in logic synthesis they enable also remarkable area and power savings. The reason for such improvement is twofold. On the one hand, the structure of MIGs is further simplifiable by technology mapping algorithms based on Boolean techniques, such as equivalence checking using BDDs,

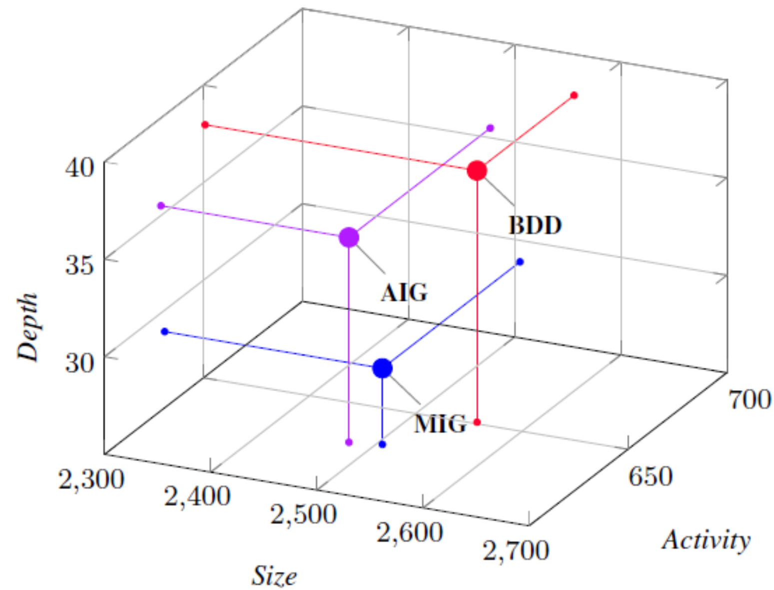


Fig. 2.17 Experimental Results 1 for MIG [4]

2.4.4 Logic Rewriting with Exact Databases

In order to further optimize the circuit logic, logic rewriting methods are proposed in [53, 5]. The proposed method not only consider physical models to compute more accurate timing and area costs during synthesis, but also create optimization engines and databases capable of capturing the specific logic opportunities distinctive of a technology. They optimize the

Logic Synthesis		MIG + Tech. Map.			AIG + Tech. Map.		
Benchmarks	I/O	A (μm^2)	D (ns)	P (μW)	A (μm^2)	D (ns)	P (μW)
C1355	41/32	56.34	0.74	226.68	56.27	0.76	203.55
C1908	33/25	44.72	0.78	132.98	53.47	1.06	155.07
C6288	32/32	361.47	3.18	1604.30	354.54	3.44	1822.21
bigkey	487/421	388.57	0.82	722.68	541.24	0.73	981.06
my_adder	33/17	22.68	1.19	36.17	23.23	1.68	41.10
cla	129/65	149.52	1.42	398.34	139.92	2.32	355.47
dalu	75/16	116.34	1.07	179.42	103.25	0.94	145.10
b9	41/21	12.88	0.22	19.75	13.72	0.22	20.67
count	35/16	20.16	0.91	28.04	18.76	1.07	24.87
alu4	14/8	150.15	0.65	225.16	254.80	0.67	386.71
clma	416/115	888.79	1.59	1806.65	1180.83	1.69	2191.77
mm30a	124/120	130.41	2.12	210.95	148.12	4.71	240.28
s38417	1494/1571	1287.44	1.20	2577.00	1268.05	1.34	2559.54
misex3	14/14	159.88	0.66	234.09	291.48	0.92	379.62
Average	212/176	270.67	1.18	600.16	317.71	1.53	679.07

Fig. 2.18 Experimental Results 2 for MIG [4]

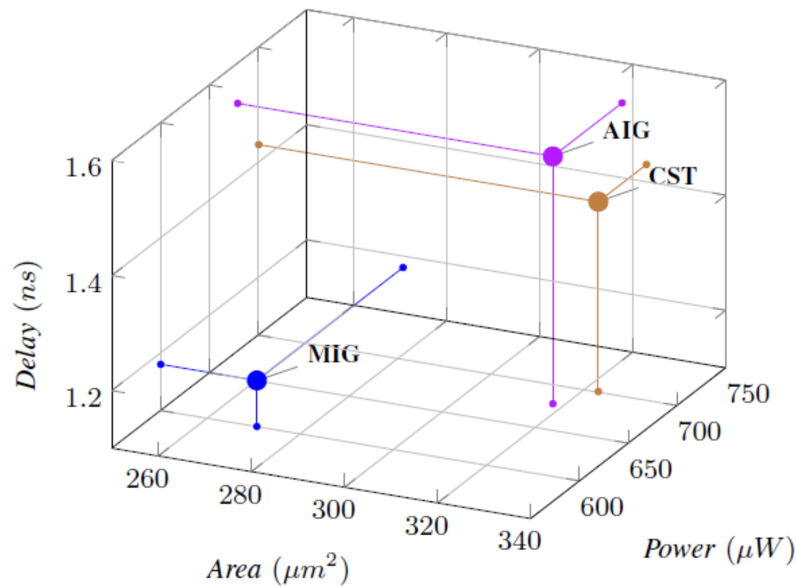


Fig. 2.19 Experimental Results 2 for MIG [4]

circuit logic by first create circuit database, and then rewriting the circuit with these optimal circuits in database.

The method develops circuit databases which contain optimum circuit realizations specific to a given technology. Such databases can be complete, i.e., containing circuits for all the functions of the circuit. The circuits stored in the databases are highly optimized and, in most cases, exact, i.e., achieving the global minimum for area, delay and power metrics. Fig. 2.20 shows a sample entry for an exact-delay database, corresponding to (i) a specific 4-input Boolean function, (ii) figurative library delay characteristics and (iii) input arrival times.

When a database storing the optimal circuits is built, the method can use fast greedy algorithm to optimize the circuit. It iteratively selects sub-circuits rooted at a node and replaces them with pre-computed sub-circuits with minimum local area, delay, power, or any combination of these metrics. The pre-computed sub-circuits are stored in a database. It is possible to build circuit databases for any design goal and with arbitrary optimization quality.

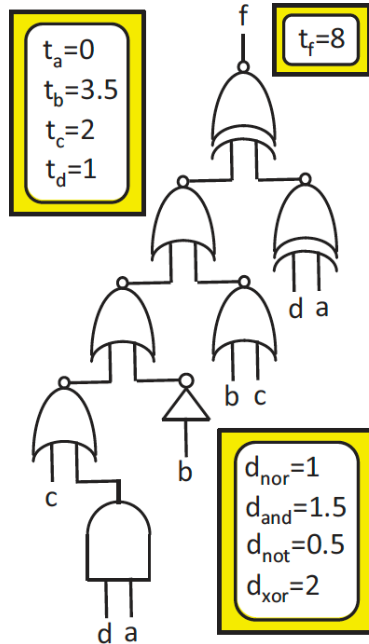


Fig. 2.20 Examples of Circuit DataBase [53, 5]

Of course, the more optimized a database is, the better the logic rewriting results will be. The method is scalable and it can be applied to a large scale circuit. However, the impact of each replacement must be evaluated at the global level, while the greedy heuristic results in the local optimization, which means that it may globally degrade some design metric. This is mostly due to global placement and routing adjustement, in particular to modified wire lengths and wire capacitances.

2.4.5 Logic Optimization by applying Deep Learning

Deep learning is a machine learning approach which is improved rapidly in these years. Recently, the advanced and powerful deep learning techniques have revolutionized machine learning. In order to build an automatic and generic system to optimize the circuit, the authors [23] apply the deep learning to learn the way to compress the logic.

Fig. 2.21 Logic Optimization as MDP [23]

The process of logic network optimization is casted as Markov decision process (MDP). In this process, the circuit moves from one state to another state. A movement is legal if the circuit before the movement and the circuit after the movement are equivalent. As shown in Fig. 2.21, the movement starts from the initial state S1, and the final goal is state Sn. Besides, we have a function score(s) that indicates how good a state s is with respect to the optimization criterion. For instance, if we want to have the size optimization, the score(s) can be the reciprocal of the circuit size. It means that we are trying to find the minimum size circuit in n steps.

Deep reinforcement learning are used to guide MDP. Assume there is a state St, and a move Mt. The reward of this transition will be r(St, At), which is defined with respect to the optimization objective. We can define this reward with respect to the score function as r(St, At) = score(St+1)-score(St), or rt = r(St; At). The sum of the score will be:

Sum = score(St+1)-score(St) + score(St)-score(St-1) + ... + score(S2)-score(S1) = score(St+1)-score(S1).

Because the initial state's score S0 is constant for all sequences starting from it, satisfying the optimality criterion corresponds to maximizing the expectation of the above sum for every initial state.

The experimental results are shown in Fig. 2.22 and 2.23. The potential improvement in the table means that for any pair (x, y) belongs to D, the maximum potential size improvement that can be made from a Shannon decomposition x to an MIG optimum is the score(x)-core(y).

Experiment	Optimization Objective	Potential Improvement %
3-input functions	SIZE	100.00%
3-input functions	DEPTH	92.60%
4-input functions	SIZE	83.00%

Fig. 2.22 Experimental Results 1 of Deep Learning Method[23]

Benchmark	I/O	Initial depth	ABC	DNN	Improvement
b9	41/21	10	8	7	12.5%
count	35/16	20	19	10	47.4%
my_adder	33/17	49	49	42	14.3%

Fig. 2.23 Experimental Results 2 of Deep Learning Method[23]

We say that it reaches 100% of potential improvement, since the model finds the optima for all 3-input functions. The authors also run the same experiment for all 4-input functions. The data set now consists of all 65536 4-input functions. They run the training procedure to convergence. The results show that model is able to find the global size optima for 24% of the functions. For other functions it does not reach the full optima within 20 inference steps. However, it reaches 83% of total potential improvement. This implies that for most functions the model is nearly optimal. The authors also select 3 benchmark circuits to have the experiment. They train a DNN model to perform depth optimization on them. After training, they use 50 inference steps to optimize depth. The results are summarized in Fig. 2.23. The results illustrate that the proposed method obtain significant improvements compared to resyn2 command of ABC, ranging from 12.5% to 47.5%, with an average of 24.7%.

Although there are many logic representation and optimization methods, redundancy addition and removal still works as one of the last method for optimization especially when redundant multiple faults are utilized, since it is time-consuming to find multiple redundant faults.

2.5 Conclusion

In this section, we firstly introduce basic ATPG technologies for single stuck-at fault, including the D-algorithm, PODEM and FAN. Next, as a powerful technology to generate the single test pattern, SAT-based ATPG technologies are discussed. Then, we present previous research to detect the multiple faults. Finally, we discuss previous technologies for logic optimization. Inspired by those research, we propose our method for multiple faults detection and logic optimization, which are introduced in following sections.

Chapter 3

ATPG Method for Double Faults

3.1 Concept of the Proposed ATPG Method

In order to efficiently find a compact test set to cover all MSAFs, this chapter proposes an ATPG method which generates test patterns by analyzing the propagation paths of the single faults[51, 49]. The ATPG method proposed in this chapter is used to deal with double stuck-at faults. We will introduce the way to extend the proposed method to handle all multiple faults in chapter 4.

The proposed method has two different features compared to all of the previous approaches. First, instead of considering all DSAFs, we try to obtain only the uncovered DSAFs and then generate additional test patterns for them one by one. As can be seen from the experimental results in [17], since there are very few uncovered faults, it takes a short time to generate the additional test patterns and the entire test set is kept small.

A general and simple way to get the uncovered faults is to check all DSAFs and exclude the covered ones. However, it is time-consuming to consider all DSAFs, especially for large

circuits. Given m possible SSA faults in the circuit, the time complexity to inspect all SSA faults is O(m), while it becomes O(m^2) for DSAFs, O(m^3) for triple faults and O(m^n) for n faults.

Second, the proposed method checks the propagation path of an SSA fault to find whether other SSA faults block its fault propagation or not, which is the first fault filtering method in our proposed approach. Therefore, instead of traversing the entire DSAF list, we only deal with a very small subset of DSAFs that are not detectable by SSA test patterns. The SSA fault pairs which mutually mask the propagation paths with each other are classified as the uncovered DSAFs. Note that only the propagation paths of SSA faults are inspected no matter which kind of faults (double, triple, quadruple, etc.) is targeted. Therefore, the complexity to find the undetected DSAFs is the same as the SSA fault analysis, whose complexity is O(m) if there are m possible SSA faults. By this way, we can drastically decrease the problem complexity to obtain the compact multiple test set compared to the previous approaches such as [17] that try to traverse the entire multiple fault list, which leads to exponentially large search space. The feasibility of our proposed method will be proved in Section 3.3.

In addition, among the undetected DSAFs selected by our algorithm, sometimes a few faults can still be covered by the SSA test because they may have other unblocked propagation paths to different primary outputs. Hence, redundant test patterns may be generated. In order to further compress the test set, the second fault filter is proposed to reinspect the undetected DSAFs, ensuring that all the remaining faults are really uncovered, which is the main difference from our previous method [34].

The proposed method can cover all the missing cases ignored by the previous research [34], with considerably reduced runtime owing to the adequate integration of the external

tools in the implementation, as well as the strategies such as circuit partitioning and bit parallel processing which can greatly decrease the runtime of fault simulation and SAT-based test generation. Moreover, the proposed method can be inductively employed to deal with all MSAFs by applying the same idea.

The rest of the chapter is organized as follows. Section 3.2 explains the conditions that when test patterns for SSAF are sufficient and insufficient to detect the DSAF. Section 3.3 proves the feasibility of the proposed method and explains the key idea of the proposed fault filtering. Section 3.4 introduces the four steps of the proposed method. Section 3.5 analyzes the experimental results. Section 3.6 summarizes this chapter.

3.2 Applying Test Patterns for SSA Faults to Detect DSAFs

3.2.1 Definitions

The term **masking gate** describes the case when the fault is blocked at this gate; hence, cannot be propagated to any output. In other words, the fault cannot be observed at any output when the current test pattern is applied.

3.2.2 When Test Patterns for SSA Faults is Sufficient to cover DSAFs

Proposition: Given a combinational circuit with single output, assuming all the gates are 2-input NAND gates with no fan-out, at least one of the test patterns, v_1 or v_2, which detect SSA faults f_1 and f_2, respectively, will also detect the DSAF $\{f_1, f_2\}$. This statement implies that any set of test patterns which can detect all SSA faults will also detect all DSAFs in the given circuit.

(Sketch of proof) Consider a combinational circuit with primitive gates. Without loss of generality, we assume all the gates are 2-input NAND gates, but it can be easily extended to any primitive gate with any number of inputs. Let's assume that there exists a DSAF, $\{f_1, f_2\}$, which consists of the two non-equivalent SSA faults, f_1 and f_2. Assuming there exists a pair of test patterns, v_1 and v_2, which detect f_1 and f_2, respectively. Because the circuit is a tree with one root node, it is guaranteed that the propagation path of one fault will intersect with the other.

We can ignore any case in which f_2 lies in the propagation path of f_1 (or vice versa), because this is a trivial case in which v_2 (or v_1) is guaranteed to detect $\{f_1, f_2\}$ as shown in Fig. 3.1. f_2 is completely unaffected by f_1 regardless of the value of f_1.

This means that the case which we must consider is when f_1 and f_2 propagate to a masking gate, as shown in Fig. 3.2. Three cases may happen, as follows.

1. f_2 does not fix the value at wire B, which means that the value at wire B can be changed by assigning the values of IN2, consequently v_1 can set wire B to 1 to propagate f_1 to the output.

2. f_2 is equal to the stuck-at 1 fault at wire B. In other words, regardless of the input value at IN2, wire B is fixed at 1 due to the fault f_2. Then f_1 is detected by v_1.

3. f_2 is equal to the stuck-at 0 fault at wire B, and hence the propagation of f_1 will be masked when using v_1 to detect it. In this case, the primary output value at wire C becomes 1 regardless of f_1, while the correct value at wire C is 0 under the test pattern v_2, since the precondition is that f_2 can be detected by v_2. Therefore, v_2 can detect f_2 because the faulty value and the correct value at the output wire C are different under the test pattern v_2.

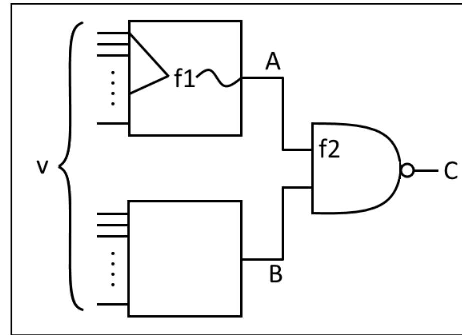


Fig. 3.1 Fault f_2 lies in the propagation path of f_1.

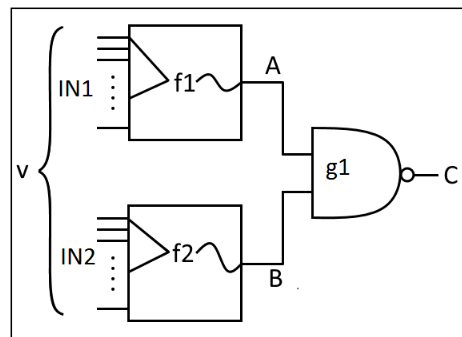


Fig. 3.2 Propagation paths of f_1 and f_2 intersect at a gate.

To conclude, the DSAF $\{f_1, f_2\}$ can be detected by v_1 in case1 and case2, while v_2 can cover it in case3. These are all the cases which can happen when two faults intersect at the same gate. Therefore, the DSAF $\{f_1, f_2\}$ is detectable by v_1 or v_2, when the circuit has no re-convergence. Similar analysis can be done in the case of other types of gates, and more number of inputs. *(End of sketch of proof)*

3.2.3 When Test Patterns for SSA Faults is Insufficient to cover DSAFs

The essential characteristic of tree like circuits, which is used in the above proof, is the fact that fan-out and re-convergence of signals do not exist. By considering them, we can have more complicated fault propagation paths, such as multiple faults intersecting on multiple gates and re-converging in another gates like XOR. In contrast with the tree structure circuits,

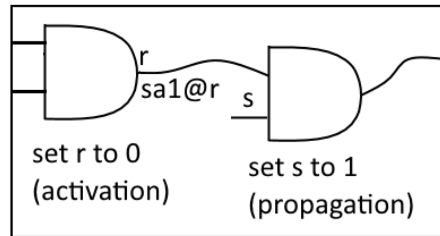


Fig. 3.3 Example of redundant fault.

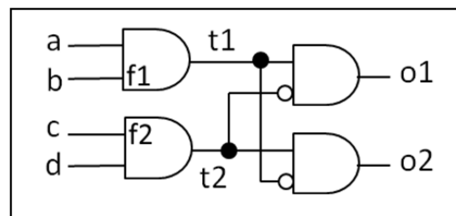


Fig. 3.4 Example that DSAF is overlooked.

the circuit with more fan-outs and re-convergence is prone to have the fault masking, since the fault propagation paths are more likely to intersect; hence, they are blocked by each other. In addition, redundancies may cause issues because they are "hidden" in the case of SSA faults but can be "unlocked" in the case of MSAFs, consequently, overlooked.

In the example shown in Fig. 3.3, it is assumed that an input pattern which can activate and propagate the fault does not exist. In other words, it is not possible to simultaneously set r to 0 and s to 1. Therefore, the stuck-at 1 fault at location r (sa1@r) becomes redundant. This fault will be overlooked by the ATPG for single faults. For example, assume that there is only a single fault sa1@r, the activation of sa1@r always results in the value of 0 at s as

Table 3.1 Test pattern 1001 detects both SSAFs, but it does not detect the DSAF.

abcd	Fault	t_1	t_2	o_1	o_2
10xx	SSA 1, f_1	D'	0/1	D'/0	0/D
xx01	SSA 1, f_2	0/1	D'	0/D	D'/0
1001	DSA 1, $\{f_1, f_2\}$	D'	D'	0	0
1000	DSA 1, $\{f_1, f_2\}$	D'	0	D'	0
0001	DSA 1, $\{f_1, f_2\}$	0	D'	0	D'

sa1@r is assumed to be redundant here. However, if we also have sa1@s, the previously untested redundant fault, sa1@r, is now non-redundant as a DSAF, {sa1@r, sa1@s}, and it may not be detected by the set of test patterns for SSA faults.

In the previous proof, we assumed a circuit with no fan-out. However, many circuits will violate this assumption. One example of a DSAF which may be undetected due to fan-out is shown in Fig. 3.4 and Table 3.1. Advanced ATPG tools for single faults may notice that the test pattern 1001 will detect both stuck-at 1 faults f_1 and f_2. However, this test pattern will not detect the case when both faults are simultaneously active. Furthermore, the DSAF is not redundant, as either of the test patterns 1000 or 0001 will detect it. In other words, this DSAF could potentially slip through the testing phase undetected by the test patterns for SSA faults.

3.3 Proposed ATPG Method for DSAF

The main problem of the previous works for MSAF is managing a huge fault list, including a large number of MSAFs. In order to decrease the list size, we can focus only on the MSAFs undetected by the SSA test set. Fault filters, which can efficiently obtain the undetected DSAFs, will be proposed in this section.

3.3.1 Focusing on undetected Faults

The bottleneck of the conventional ATPG methods for MSAFs is fault list generation and management of multiple faults, but not the fault detection. Take the DSAF as an example and ignore fault collapsing. If there are n SSA faults, the total number of DSAFs becomes

n(n-1)/2*, which grows quickly for large circuits. Therefore, all the possible DSAFs are hard to be represented and detected one by one.

In most of the cases, MSAFs are easier to be detected compared to SSA faults, if there are multiple propagation paths to different primary outputs, according to the results of the previous research as shown in Table 2.1. However, sometimes they may tend to be blocked by each other when faults propagate to the same gate or to the same primary output. Since the blocked faults are much fewer than the entire fault list, we can only take the uncovered faults into account, avoiding the detected ones. In order to obtain the undetected faults, the proposed method acts as a filter to pick up the DSAFs which cannot be detected by the test patterns for SSA faults. It can drastically reduce the size of the DSAF list, as also shown by our experiments.

As shown in Fig. 3.9, at first the proposed method only generates the SSA fault list instead of the DSAF list, and then filters out the DSAFs covered by the SSA test set. Note that the proposed method does not traverse all DSAFs to decide which DSAF is undetected by the SSA test set, as it is time-consuming. It only analyzes SSA faults and their propagation paths to obtain the DSAFs which are not detected by the test patterns. As the result, only a few DSAFs remain. New test patterns are generated by checking them one by one, as shown in (d), (e) and (f) of Fig. 3.9, which is basically the same as the conventional methods for SSAF ATPG. Details of how to select the undetected DSAFs will be explained in the following subsections.

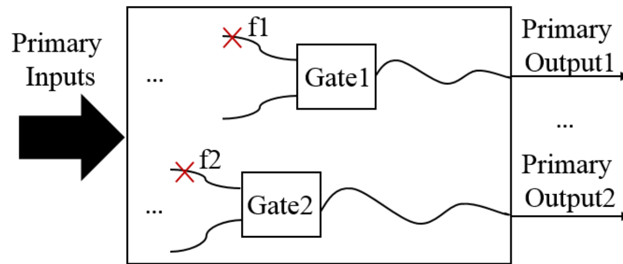


Fig. 3.5 Propagation paths of two faults have no intersection.

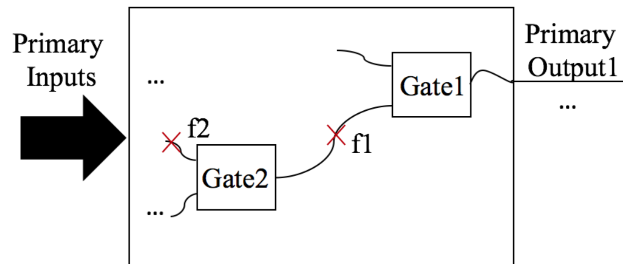


Fig. 3.6 One fault lies in the propagation path of another fault.

3.3.2 Proposed Fault Selection Method in Non-redundant Circuit

One of the main issues when using "single fault ATPG" to detect multiple faults is that the second fault may somehow interfere with the propagation of the first one. Hence, we can find the undetected faults by checking all of them and ignore the covered ones. However, it takes a long time to traverse all DSAFs due to the huge number of faults. This problem will become much severer when generating test patterns for more simultaneous faults. In order to keep the same problem complexity regardless of the number of simultaneous faults, we

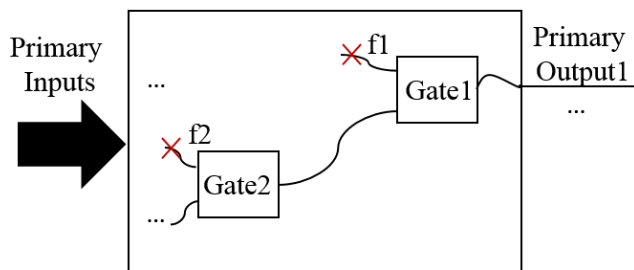


Fig. 3.7 Path constraints of one fault is violated by another fault.

propose a method based on the premise that, given DSAF $\{f_1, f_2\}$, as long as f_2 does not interfere with the propagation of f_1, f_1 will be detected. Therefore, the DSAF $\{f_1, f_2\}$ will be detected as well. Once a test pattern is found for some faults and the path constraints are in place, any other fault which may interfere with those path constraints can be easily identified and examined. Here the path constraints are the side values in the fault propagation path. After checking all the path constraints, the selected DSAFs will be reinspected to further compress the test patterns. The feasibility of the algorithm is proved as follows.

Proposition: Assuming there are two SSA faults f_1, f_2 with their test vectors v_1, v_2 in a non-redundant circuit. If f_1 and f_2 do not mutually violate the path constraints of each other, the DSAF $\{f_1, f_2\}$ can be detected by v_1 or v_2. This proposition means that in order to obtain all undetected DSAFs, we only need to check the path constraints of each SSA fault instead of traversing all DSAFs, which can drastically reduce the search space.

(Sketch of proof) There are three possible cases when DSAFs occur, as follows.

1. f_1 and f_2 are propagated in two different paths without any intersection, as shown in Fig. 3.5. The DSAF can be detected by v_1 or v_2, since the propagation paths of the two faults are not effected at all.

2. As we discussed in Section 3.2, in the case that one fault lies in the propagation path of another fault as shown in Fig. 3.6, f_1 is completely unaffected by f_2 regardless of the value of f_2. Therefore, the DSAF must be detected by v_1, and vice versa.

3. The final case is that one fault violates the path constraints of another fault as shown in Fig. 3.7. Two subcases are discussed below:

1) If only f_1 violates the path constraints of f_2 but f_2 does not block the propagation path of f_1, the DSAFs can be detected by v_1.

2) On the other hand, if two SSA faults mutually violate the path constraints of each other, $\{f_1, f_2\}$ cannot be detected by either v_1 or v_2, if both SSA faults have only one propagation path.

These are all the possible cases when DSAFs happen in the circuit without redundancy. However, the SSA faults may have multiple propagation paths. Hence, the last case and other cases may simultaneously happen in different paths. In other words, some of the undetected DSAFs selected by the proposed algorithm may still be detectable since the fault may be unblocked in other propagation paths, which results in the generation of redundant test patterns. But they can be compressed by reviewing the remaining DSAFs one by one and then picking up the actual undetected faults. This process is very fast because there are only a few remaining DSAFs.

To conclude, in order to find all the undetected DSAFs in a non-redundant circuit, we only need to consider the case that two SSA faults violate the path constraints of each other.

(End of sketch of proof)

3.3.3 When Circuit has Redundancy

Redundancy is unavoidable in the real circuit. If there are redundant SSAFs in the circuit, some cases may be overlooked by the proof in subsection 3.3.2.

1. One fault lies in the propagation path of another fault, and f_1 is a redundant fault. $\{f_1, f_2\}$ needs a new test pattern, since the propagation path of f_2 is blocked such that it cannot be detected at PO1 by applying the test pattern v_{2A}. Hence, we need to generate a

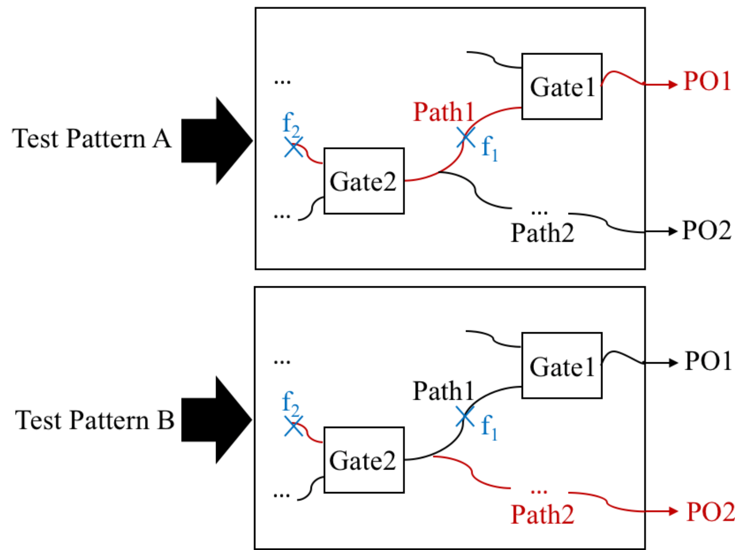


Fig. 3.8 One fault lies in the propagation path of another fault (missing case 1)

new test pattern v_{2B} to bypass the redundant fault f_1 and reach another output PO2. This is the one of the cases not covered by the previous research [34].

2. The DSAF consisting of two redundant SSA faults. If both SSAFs are redundant, we cannot analyze the propagation path to find the undetected DSAFs, as no test pattern is generated for them. However, we cannot ignore them since some of those faults are in fact activated as a non-redundant DSAF, although such cases are not frequent. Otherwise they can become the hidden problem when we incrementally extend the method to n multiple faults. The number of those DSAFs are enormous when the circuit has many redundant SSAFs. In order to quickly exclude as much as possible the real undetectable DSAF, we employ a fault filtering to eliminate the two redundant SSA faults pair that are in the same propagation path, which are definitely impossible to be activated again as the DSAF. The remaining redundant SSA faults pairs that are located in the same area are selected as the possible undetected DSAFs.

3.4 Four steps of Proposed Method

Four steps of the proposed ATPG method are introduced in this section, as shown in Fig. 3.9. The algorithm flow of the proposed method is introduced in subsection 3.4.1. The first step, circuit initialization process is shown in subsection 3.4.2. The second step, ATPG initialization process is shown in subsection 3.4.3. The third step, fault selection process, is presented in subsection 3.4.4 and 3.4.5. The fourth step, test generation process, is discussed in subsection 3.4.6. Finally, the way to extend the proposed ATPG method for DSAF to all MSAFs is discussed in subsection 3.4.7.

3.4.1 Algorithm Flow for Double Faults

1. Circuit initialization:

Initializing and preprocessing the circuit data to accelerate the later process, which is introduced in 3.4.2.

2. ATPG initialization:

Generating the SSA Fault list (SFL), and their corresponding Test Patterns and Path Constraints:

1) Generate the list of all SSA faults at the gate inputs that are connected to a fan-out wire or PI.

2) Pick a SSA fault f_i.

3) Generate a test pattern v_i, for f_i.

4) Based on f_i and v_i, find the necessary path constraints for the fault propagation.

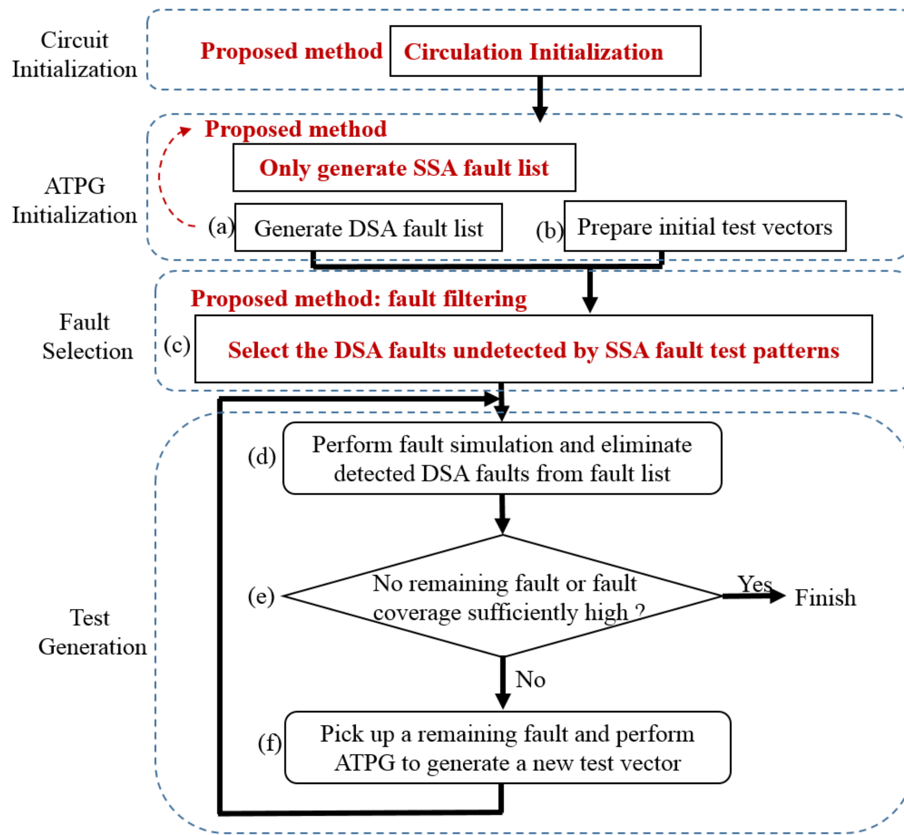


Fig. 3.9 The proposed ATPG flow for DSAF.

5) If there are SSA faults in SFL without a test pattern, repeat from step 2 (Pick a SSA fault). Else go to the next step.

3.1 The first filter of DSAFs:

checking the path constraints following the algorithm flow shown in Fig. 3.10.

1) Initialize a temporary list of potentially undetected DSAFs (PUDL).

2) Take an unchecked SSA fault from SFL as the “focused fault” and traverse the SFL to find other SSA faults which violate the path constraints and also mask the focused fault.

3) If there are unchecked SSA faults in SFL, repeat from the previous step (Take an unchecked SSA fault). Else go to the next step.

5) Go through the temporary list PUDL, and keep only the faults which are listed twice, i.e. $\{f_i, f_j\}$ and $\{f_j, f_i\}$.

6) Duplicate PUDL list as undetected DSAFs list (UDL).

Here the potentially undetected DSAF means the faults that may be masked since at least one of the SSA faults among it violates the path constraints of another.

3.2 The second filter of DSAFs:

To avoid the generation of redundant test patterns, we compress the test set by doing the fault simulation on the faults in list UDL to filter out the DSAFs that are covered by the generated SSA test patterns.

4. Test Generation:

Perform simple ATPG on the remaining DSAFs in the list UDL to generate new test patterns for them.

3.4.2 Circuit Initialization

At the beginning, the circuit data is read and the preprocessing is performed for the acceleration of the later steps. As it is well known, when the circuit size is small, the test generation is quite fast since both the circuit simulation and SAT process can be completed in a short time. However, as the size of the circuit significantly increases, it becomes much more time-consuming.

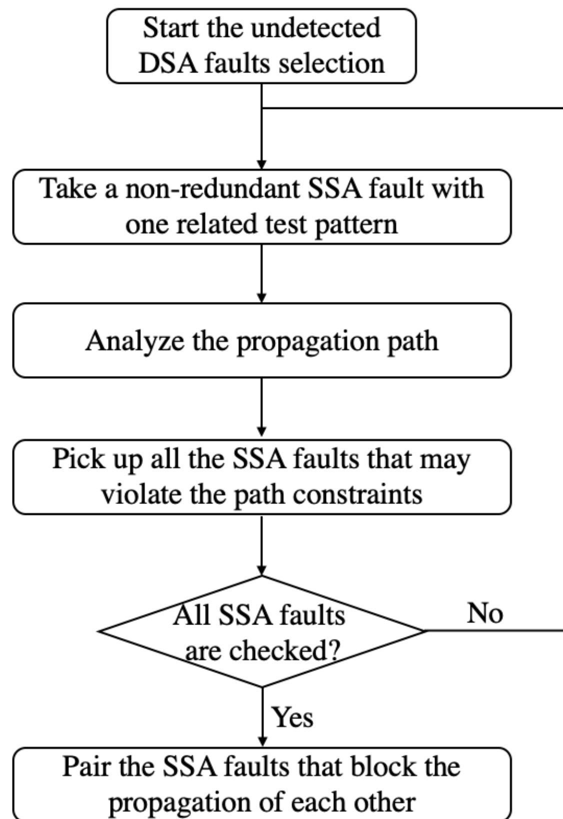


Fig. 3.10 Undetected DSAFs Selection Flow

We have proposed a circuit partitioning method to split the circuit to different parts and pair them with the related gates, which can avoid dealing with the entire circuit in the next steps of the ATPG process. Let us discuss the example circuit shown in Fig. 3.11, the gate 2 is paired with gate 1. If a stuck-at fault happens in gate 2, only the gate 1 and gate 2 are extracted to do the test generation, while gate 3 is not included since it does not affect the value in the primary output (PO) 1, which is connecting to gate 2. Maps of the partitioned sub-circuits {{gate 1: gate i, gate j. . . },{gate 2: gate m, gate n. . . }...} is generated and stored in the memory after circuit partitioning. Although the partitioning process takes some time as a trade-off, the total runtime of test generation is significantly reduced, since the circuit size

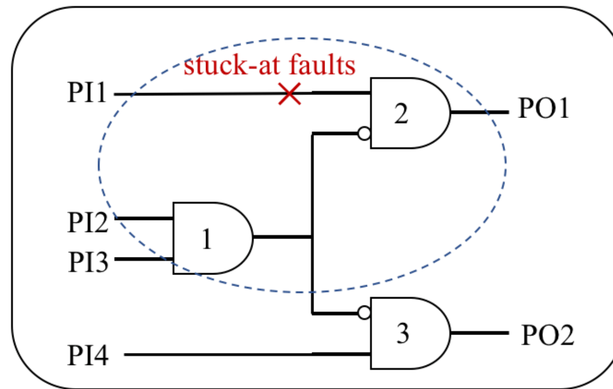


Fig. 3.11 Example of circuit partition

we need to handle in each circuit simulation and SAT process can be reduced, on average, to smaller than 1/10 of the original circuit size.

3.4.3 ATPG Initialization

In order to collapse the faults, at first the proposed method only generates the SSA faults at the gate inputs that are connected to a fan-out wire or PI (Primary Input) following the paper [45] and add them to the SSA fault list SFL. For more information about fault collapsing, please refer to [32, 39]. Since this fault selection model may overlook some redundant faults, we explicitly go through the entire circuit to pick up all the redundant SSA faults. Then, the naive SAT-based method is applied to generate the test patterns for the SSA faults. When the result is SAT, the solution will be stored as the test pattern. Otherwise, the target SSA fault is marked as a redundant fault. After all the SSA faults are checked, their path constraints are also analyzed. Then the proposed method activates the SSA fault and traverses the circuit by following a path where the fault is being propagated. When the propagation path reaches a primary output, it quits the search and generates the constraints based on the path it has taken. In addition, the redundant SSA fault has no path constraint since no test pattern can

detect them. They must be carefully taken care of, since the redundant fault may also violate path constraints.

Assume there is a SSA fault f_1 with its test pattern v_1. The path constraints of v_1 are the side-input values in the propagation path of f_1. There is no guarantee that every net on which there is a path constraint is in fact in our initial list of SSA faults. That is because the nets on the path constraints may not contain the gate inputs which are fan-out wires or PI, where the proposed method generates the initial SSA faults. Therefore, for every path constraint which is not directly translated to a SSA fault, the circuit is back traced from that location to determine every eligible SSA fault which may cause a violation of the path constraints.

3.4.4 First Filter: Checking the Path Constraint

As previously mentioned, only those faults which actually violate the path constraints of the initially focused fault are considered as the faults that may block the propagation. Since the focused SSA fault is propagated from the faulty wire, instead of PI, the faults at the paths from PI to the focused fault cannot mask it. In other words, as long as the path constraints of the focused fault which start from the focused fault to the primary outputs are not violated, the propagation cannot be masked. Therefore, the faults at the paths from PI to the focused fault are not inspected by the proposed method. Once the entire potentially undetected DSAF list PUDL which includes the focused faults and their respective troublesome faults are generated, that list must be examined to further reduce the number of DSAFs that are going to be checked explicitly.

Through the proposed method, only the DSAFs which are listed twice are kept. This is because even if some fault f_j may stop the propagation of another fault f_i given the test

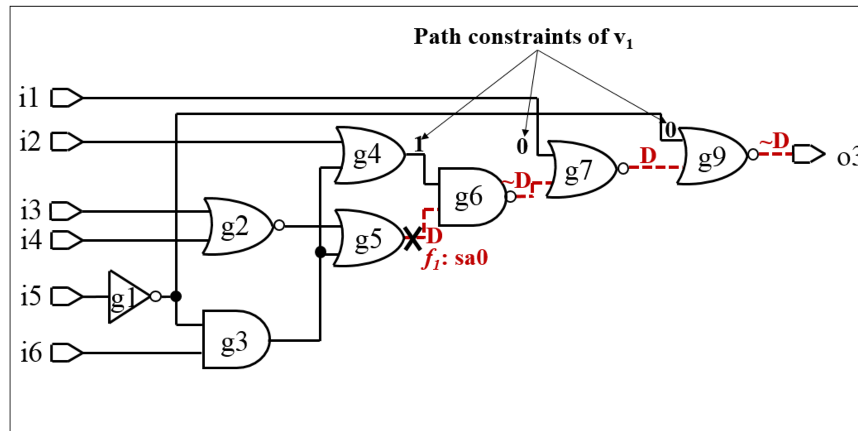


Fig. 3.12 Path constraints of v_1.

pattern v_i, if f_i does not stop the propagation of f_j given test pattern v_j, the DSAF can be detected by v_j. This is why we only keep those DSAFs which can be covered by none of the test patterns v_i and v_j.

There are three cases required to be considered as follows, the case that double faults mutually violate the path constraints, which is mentioned above, the case that one fault is redundant, and the case that both faults are redundant. The DSAFs which contain one redundant SSA fault are kept, because the redundant fault may also mask the propagation of another fault. In addition, we should not ignore the case that both f_i and f_j are redundant faults, since two redundant SSA faults may become non-redundant as a DSAF.

An example of DSAF detection is illustrated in Fig. 3.12-3.16 to explain how the first filter works. The circuit in the example is the combinational part of S27 which belongs to the ISCAS89 benchmarks. Here we assume that there are two SSA faults f_1 and f_2 which can be detected by test patterns v_1 and v_2, respectively.

To detect f_1 (stuck-at 0 on the output wire of the gate, g5), when the fault is activated and propagated to the output, the path constraints can be found as shown in Fig. 3.12. As we mentioned before, the path constraints of v_1 are the side-input values in the propagation path

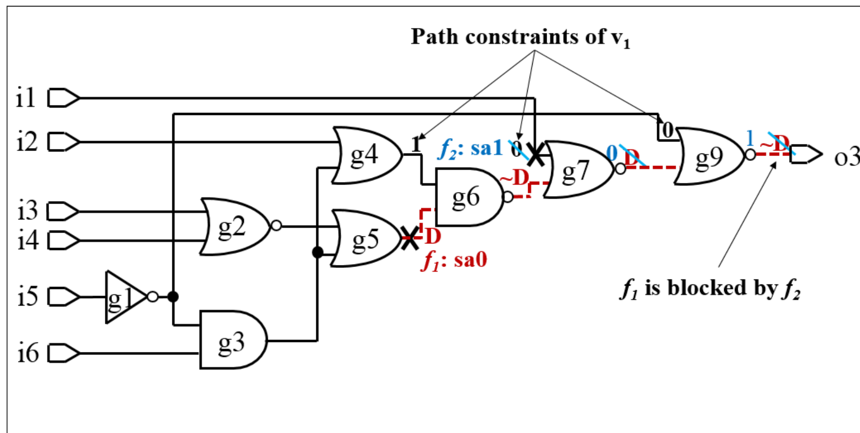


Fig. 3.13 f_2 violates the path constraints of v_1.

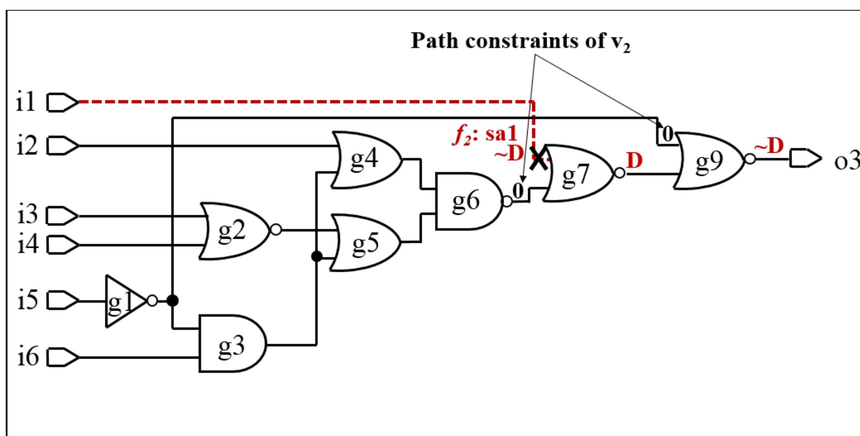


Fig. 3.14 Path constraints of v_2.

of f_1. Based on the path constraints of v_1, the circuit can be traversed to find the SSA faults which mask the propagation of f_1.

Notice that instead of traversing all SSA faults in the circuit, only the SSA faults that violate the path constraints of v_1 are checked, which can drastically reduce the search space. Assume there is another SSA fault f_2 (stuck-at 1 on the input wire of gate, g7), as shown in Fig. 3.13, f_1 is masked by f_2, which means that the DSAFs $\{f_1, f_2\}$ cannot be detected by v_1.

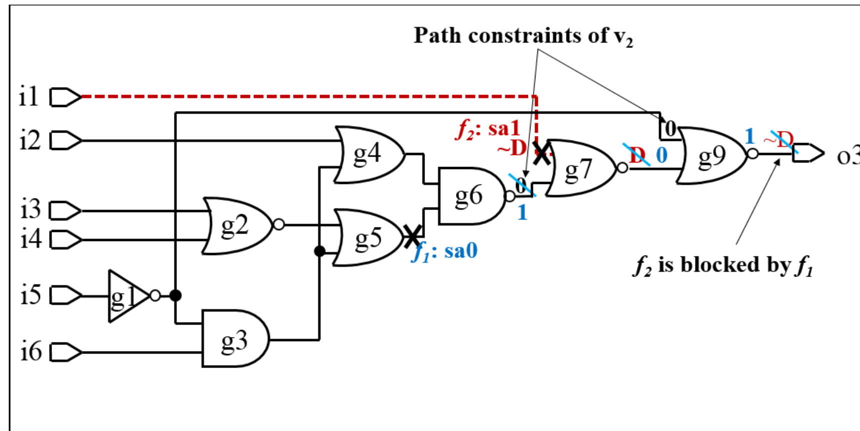


Fig. 3.15 f_1 violates the path constraints of v_2.

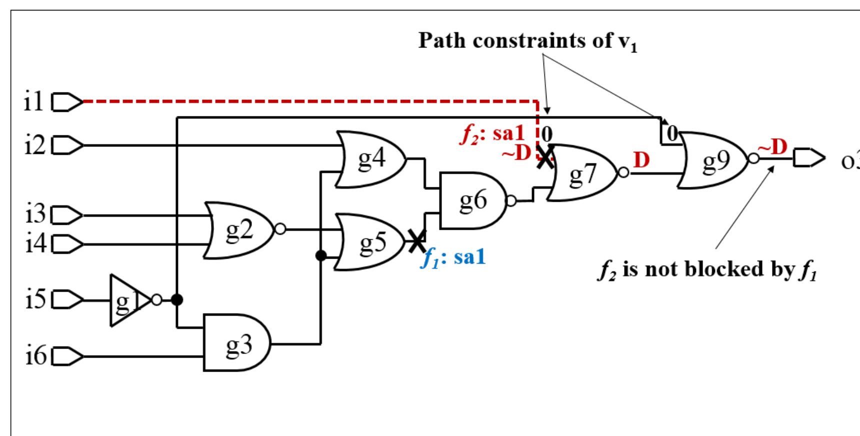


Fig. 3.16 Double faults $\{f_1, f_2\}$ can be detected by v_2.

The f_2 is activated and propagated to the output by v_2 as shown in Fig. 3.14. The path constraints of v_2 is also violated by f_1 as shown in Fig. 3.15. Therefore we take $\{f_1, f_2\}$ as a potentially undetected DSAF and stored it on the list PUDL.

However, the proposed algorithm does not pick up the DSAFs where only one fault violates the path constraints of the other. For example, consider the case where f_2 violates the path constraints of v_1 but f_1 does not block the propagation of v_2 as shown in Fig. 3.16, the DSAF $\{f_1, f_2\}$ is surely detectable by v_2, which means $\{f_1, f_2\}$ does not need new test pattern. In addition, if the signals at a gate are of the same polarity, the DSAFs intersecting at this gate will be overlooked, since they are definitely detected. With this first filtering

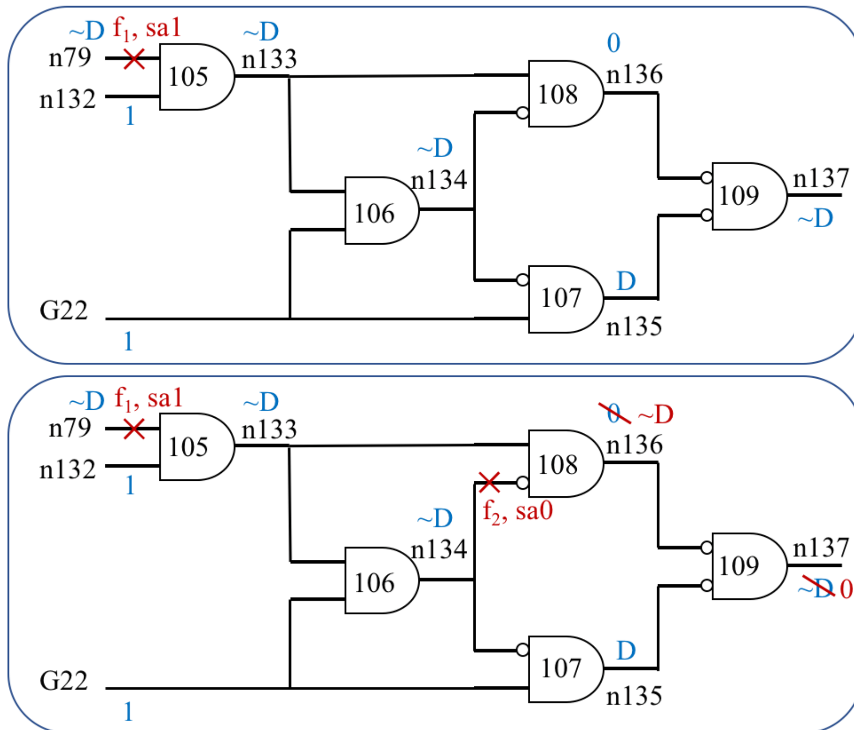


Fig. 3.17 The fault indirectly violates the path constraints (missing case 2)

method, the number of the DSAFs in the list PUDL becomes much fewer compared to the number of all DSAFs.

The fault that indirectly violates the path constraints should be checked as well, such as the example shown in Fig. 3.17. Here we assume that there is a non-redundant stuck-at 1 fault in the input wire n79 and the test pattern is generated. The f_1 is blocked in gate 108 due to the fan-outs at wire n133 and the re-convergence in gate 108, but it is propagated in gate 109 owing to the side value assigned by the output of gate 108. If there is another stuck-at 0 fault f_2 located at the input wire n134 of gate 108, f_1 is blocked in gate 109 since f_2 allows the propagation of f_1 in gate 108, which results in the counteraction of D and D of f_1 in gate 109. The previous method ignores this case since it only selects the fault that directly violets the path constraint and blocks the propagation, while f_2 indirectly blocks the propagation of f_1 by allowing the propagation of f_1 in gate 108. Therefore, we need to carefully inspect

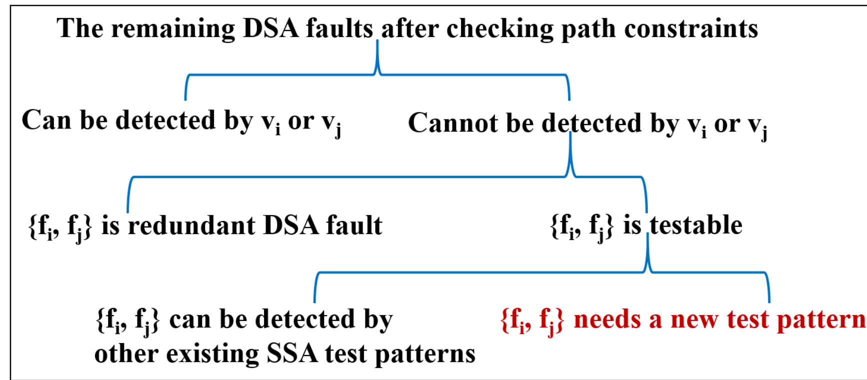


Fig. 3.18 The remaining DSAFs in the list UDL after checking path constraints

all propagation paths of the target fault to cover this case which is not considered by the previous method [34].

3.4.5 Second Filter: Excluding the DSAFs detectable by the Generated SSA Test Patterns

After checking the path constraints, only small number of DSAFs remain in the list UDL. If the ATPG process stops fault filtering here and starts to generate new test patterns, redundant test patterns may be produced since actually some of the DSAFs remaining in the list UDL can be detected by the generated SSA test set. When there is a DSAF $\{f_i, f_j\}$ with its SSA test patterns v_i and v_j, the DSAFs in the list UDL can be classified as shown in Fig. 3.18. From this figure we can conclude that, in order to minimize the final DSAF list and avoid generating redundant test patterns, we need to recheck the DSAFs to exclude the detectable DSAF which has passed in the first filtering.

In some cases, even if two faults f_i and f_j mutually violate the path constraints of each other, their test patterns v_i and v_j still can detect the DSAFs $\{f_i, f_j\}$ since there may be multiple fault propagation paths. In addition, the violation of the path constraints may just

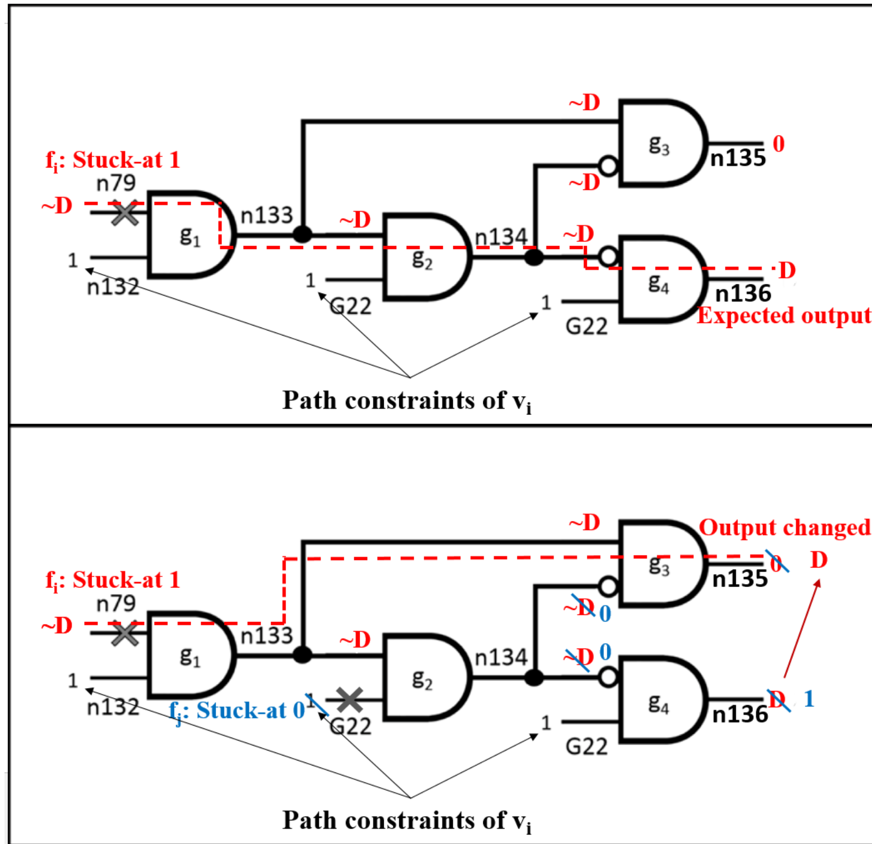


Fig. 3.19 Example: when propagation path is changed

make the fault reach a different primary output. For example, the SSA fault f_i shown in Fig. 3.19 is expected to be detected at the primary output wire n136 by the test pattern v_i. When there is another SSA fault f_j which violates the path constraints of v_i, f_i can still propagate to another primary output wire n135 though it is different from the previous output.

Besides, other test patterns may also cover the DSAF $\{f_i, f_j\}$ which cannot be detected by v_i or v_j, especially when the initial SSA test set is not compact enough. We have to exclude those DSAFs, otherwise some redundant test patterns are generated as shown in the experimental results of our previous method [34]. The proposed method rechecks the DSAFs in the undetected DSA list UDL to optimize the second fault filter and then makes the generated test set more compact compared with the previous method. Although checking

Algorithm 1: Excluding the DSAF that can be detected by the generated SSA Test Patterns

```
1: while There are unchecked faults in the list UDL do
2:    Take an unchecked {f_i, f_j} and v_i, v_j
3:    if both v_i and v_j do not exist then
4:       if other test patterns can detect {f_i, f_j} then
5:          Delete {f_i, f_j} from fault list UDL
6:       end if
7:    else if v_i or v_j exist then
8:       if v_i or v_j can detect {f_i, f_j} then
9:          Delete {f_i, f_j} from fault list UDL
10:      else if other test patterns can detect {f_i, f_j} then
11:         Delete {f_i, f_j} from fault list UDL
12:      end if
13:   end if
14: end while
```

all other SSA patterns takes extra processing time, such increasing time is acceptable since only a few DSAFs need to be rechecked here. The algorithm of the second filter is illustrated as Algorithm 1.

After the fault filtering is over, it becomes a simple matter of generating test patterns for the DSAFs in the list, since the number of the faults remaining in the undetected DSA list UDL is very small, as shown in the experimental results. The SAT-based ATPG method which is applied to the SSA faults, is also used on the final remaining DSAFs in the list UDL.

3.4.6 Generation of Additional Test Patterns

After obtaining all the possible undetected DSAFs, we repeat the process that employs the SAT solver to generate new test patterns, and performs the fault simulation to delete the fault covered by new patterns, until all the selected faults are covered. The speed of the fault simulation and SAT process is greatly accelerated since each time we only deal with a small part of the circuit related to the fault instead of the entire circuit, as discussed in

Section 3.1. Moreover, we utilize the bit parallel processing to simultaneously run 64 test patterns in the fault simulation to further improve the speed. As the number of selected DSAFs is significantly smaller than the total number of DSAFs and the speed of the fault simulation and SAT process is also considerably accelerated, the speed of the test generation is significantly improved, as shown in the experimental results.

3.4.7 Extensions for Multiple Faults of Larger Cardinalities

The above method can be extended to deal with multiple faults of larger cardinalities incrementally. After generating the test patterns for double faults, we can perform similar steps to deal with triple faults. In other words, given the test patterns for n faults (simultaneously happen), the proposed method can be employed to detect $n+1$ faults. Based on the undetected n fault list, the undetected $n+1$ faults can be quickly obtained. Please note that in such processes, as we are keeping the sets of fault propagation paths, only faults interfering the sets are examined, which is the same as the previous case. Such sets of MSAFs should always be small. This process is repeated until no undetected faults can be found in the circuit. By applying this approach, multiple faults of arbitrary cardinalities can be processed incrementally, which is one of our future work.

3.5 Experimental Results

The proposed ATPG method is implemented in C++, and operating on a machine running Linux kernel 4.17 with Xeon E5-2699 v4 2.20GHz CPU and 512GB memory. Glucose 4.1 [7] is used as the SAT solver to generate the test pattern, since it is efficient at solving

Table 3.2 Number of DSAF overlooked by [34]

Circuit	case1 Re-DSA	case1 non-Re	case2 Re-DSA	case2 non-Re
s444	0	174	9	100
s832	0	0	7	16
s1238	0	0	64	189
s1423	0	19	18	293
s1494	0	0	11	22
s5378	0	308	33	80
s9234	0	2,083	143	913
s35932	0	17,150	1,024	3,936
s38584	0	11,754	3,440	8,917

SAT problem. ISCAS89 benchmark circuits [11] are employed to perform the experiments. To analyze the performance of the proposed method in different aspects, we organize the experimental results and classify them into four subsections. Subsection 3.5.1 shows the DSAFs ignored by the previous research [34] but taken care of by the proposed method. Subsection 3.5.2 compares the performance of the proposed method to the previous research. Subsection 3.5.3 illustrates the details of the composition of the selected DSAFs in the proposed method. Finally, subsection 3.5.4 shows the runtime of each step in the proposed method.

3.5.1 Selected DSAFs

As we explained in previous sections, the previous method may not cover all DSAFs since two cases that we discussed in the previous Section are overlooked. Table 3.2 shows the details of the DSAF in case1 and case2. Re-DSA and non-Re indicate the redundant and the non-redundant DSAF ignored by the previous method, respectively. According to the result, in total, the number of non-redundant DSAFs are two to three times more than the redundant faults, which means that many detectable DSAFs are actually ignored by the

Table 3.3 Number of additional test patterns generated to cover all the DSAFs

Circuit	Result in [34]	ABC	Proposed Method
s444	2	1	1
s832	4	2	2
s1238	NR	20	18
s1423	2	2	22
s1494	Insufficient	2	2
s5378	Insufficient	3	4
s9234	Insufficient	10	9
s35932	NR	NR	21
s38584	NR	NR	20

previous research. Although most of those DSAFs can be actually covered by the SSA test patterns, we still need to pick up all of them to generate a complete test set. As we include those missing cases in the proposed method, the generated test patterns can cover all DSAFs, which is proved by exhaustively inspecting all the possible DSAFs.

3.5.2 Performance Comparison

Next, we perform the experiment in testing and verification tool ABC [10] using "&fftest" command and the proposed implementation based on the compact sets of the SSA fault test patterns [14]. Besides, we also take the results from the previous research for the comparison. The coverage of the test patterns is confirmed by exhaustively checking DSAFs in the circuit.

The number of the additional test patterns for the DSAFs when starting from the compact test patterns for the SSA faults is presented in Table 3.3. Number of additional test patterns generated to cover all the DSAFs. In the second column, "Insufficient" means that the generated test patterns cannot cover all DSAFs due to the missing case1 and case2. "No result" means that the test generation process of S35932 and S38584 circuit cannot be completed due to some problems. While the previous research and ABC fail to obtain

Table 3.4 Total runtime (seconds)

Circuit	Result in [34]	ABC	Proposed Method
s444	72	2	0.2
s832	642	248	0.5
s1238	NR	1,476	27
s1423	238	140	2.3
s1494	1,462	126	1.7
s5378	4,454	6,370	6.6
s9234	7,238	61,180	280
s35932	NR	NR	2,102
s38584	NR	NR	1,027

the test patterns in large circuits, the proposed method can successfully generate sufficient test patterns to cover all DSAFs. Moreover, for most of the circuits, the additional test patterns generated by the proposed method are as compact as ABC, though more test patterns are generated in S1423 circuit as shown in the fourth column, which may need further optimization in the future.

In addition, we do the comparison about the total runtime as shown in Table 3.4. The execution speed of the proposed method can achieve around 100 times faster than ABC, and 500 times faster than [34], on average. The execution speed is significantly improved by our method.

3.5.3 The Selected DSAFs in the Proposed Method

The third column in Table 3.5 is the total number of all possible DSAFs, which is calculated assuming no fault collapsing is performed for the DSAFs. The fourth and fifth column are the numbers of the selected DSAF that are undetected by the SSA test patterns, respectively. Owing to the proposed fault selection algorithm, less than 0.1% DSAFs are selected, which is significantly smaller than the number of total DSAFs.

Table 3.5 Number of selected DSAFs in the proposed method

Circuit	All SSA	All DSA	Selected DSA
s444	1,158	1,340,964	56
s832	2,328	5,419,584	51
s1238	3,452	11,916,304	1,714
s1423	3,464	11,999,296	480
s1494	4,194	17,589,636	41
s5378	10,426	108,701,476	314
s9234	14,052	197,458,704	18,633
s35932	88,084	7,758,791,056	441,931
s38584	88,824	7,889,702,976	116,559

Table 3.6 Number of non-redundant and redundant DSAF among all selected DSAFs

	All Selected DSA		Two Non-re SSA		Non-re SSA + Re-SSA		Two Re-SSA		Re-SSA
Circuit	Re-DSA	Non-re	Re-DSA	Non-re	Re-DSA	Non-re	Re-DSA	Non-re	Re-SSA
s444	51	5	0	0	21	5	30	0	18
s832	46	5	0	0	11	5	35	0	12
s1238	1,686	28	0	0	103	11	1,583	17	84
s1423	72	408	0	70	36	334	36	4	27
s1494	36	5	0	0	19	1	17	4	15
s5378	307	7	0	3	80	4	227	0	48
s9234	18,577	56	4	0	1,579	55	16,994	1	328
s35932	441,376	555	0	87	4,128	468	437,248	0	10,112
s38584	116,490	69	0	0	20,507	66	95,983	3	1,995

The details of the selected DSAF is shown in Table 3.6. The second and third column present the numbers of the redundant DSAFs and the non-redundant DSAFs among all the selected DSAFs, respectively. Only a small number of the selected DSAFs are detectable, hence, in the test generation process, we actually spend more than 90% of time to find and exclude the redundant DSAF. The results from fourth to ninth columns illustrate the number of three types of the selected DSAFs, which consists of two non-redundant SSA faults, one non-redundant SSA and one redundant SSA fault, and two redundant SSA faults. Notice that, very few DSAFs consisting of two non-Redundant SSA faults are classified as undetected

Table 3.7 Number of Re-DSAF that are excluded since two Re-SSA faults are in same propagation path

Circuit	Re-DSA Two Re-SSA	Deleted (Same Path)	Ratio
s444	30	27	90%
s832	35	8	22%
s1238	1,583	80	5%
s1423	36	32	89%
s1494	17	12	71%
s5378	227	55	24%
s9234	16,994	1,771	10%
s35932	437,248	107,072	24%
s38584	95,983	16,119	17%

DSA, since those faults have more propagation paths and smaller probability to be blocked, as shown in the fourth and fifth columns. On the other hand, most of the non-redundant DSAFs consist of one non-redundant and one redundant SSA faults, as shown in the seventh column, while most of the redundant DSAFs have two redundant SSA faults as shown in the eighth column, since generally they are unlikely to be activated and propagated. The number of the redundant SSA faults is illustrated in the tenth column.

Since most of the selected DSAFs consist of two redundant SSA faults but over 95% of them are actually undetectable, we employ a fault filter to quickly exclude some redundant DSAFs by eliminating the two redundant SSA faults pair that are in the same propagation path. In Table 3.7, the second column is the number of the DSAF that are redundant and consist of two redundant SSA faults. The third and fourth column are the DSAF excluded by our fault filter. The filter works for all the circuits, especially for S444, S1423 and S1494, as more than 50% of the faults are filtered out since they are on the same propagation path, while for other circuits we need to use SAT solver to eliminate more than 70% of undetectable faults. We still need to consider more efficient ways to find those undetectable DSAFs.

Table 3.8 Runtime of each step in the proposed method (seconds)

Circuit	Circuit Init	ATPG Init	Select DSA	Generate new Test	Total Runtime
s444	0.007	0.08	0.06	0.03	0.2
s832	0.009	0.2	0.1	0.1	0.5
s1238	0.02	1	0.4	26	27
s1423	0.2	0.6	0.6	0.9	2.3
s1494	0.3	0.3	0.2	0.09	1.7
s5378	0.2	2.1	1.1	3.07	6.6
s9234	0.3	5	11	264	280
s35932	6	79	706	1,311	2,102
s38584	7	101	400	519	1,027

3.5.4 Runtime Analysis of the Proposed Method

Table 3.8 presents the four steps in the proposed method. The second and third columns illustrate that the initialization process only takes a small portion of total runtime. In addition, the time to find the undetected DSAF and generate new test are almost the same, as shown in the fourth and fifth columns. However, S1238 and S9234 takes large amount of time to generate new test patterns, since they have more redundant SSA faults comparing to the circuits with similar size, as shown in the tenth column of Table 3.8, which results in more undetectable DSAFs consisting of two redundant SSA faults are selected. Hence, it takes more time to employ SAT-solver to exclude them. Therefore we can conclude that, the test generation time in the propose method is not only decided by the size of the circuit, but also affected by the circuit structure and the number of the SSA redundant fault.

3.6 Conclusion

In this chapter, we have proposed a new incremental ATPG method for DSAFs. Four steps of the proposed method are introduced, including the circuit initialization, ATPG initialization,

finding the undetected DSAF by using the fault filtering and generating the additional test patterns. In order to drastically decrease the size of the DSAF list, we propose two fault filters to excludes most of the DSAFs which do not need new test patterns, as shown in Table 3.5. Two missing cases in our previous research are considered to cover all double faults, as shown in Table 3.2. The proposed method starts with the ATPG for single faults and incrementally adds new test patterns for double faults. The complexity of each incremental process is the same as the ATPG for SSA faults, thus the proposed method can be very efficient for large circuits. According to the experimental results, the proposed method can generate complete test patterns for all the given benchmark circuits, with the processing speed that is more than two orders of magnitude faster than the previous work.

Chapter 4

Incremental ATPG Method for Multiple Faults

4.1 Key Idea of the Proposed Method

Based on the method introduced in chapter 3, an incremental ATPG method is proposed in this thesis to generate the test patterns for all multiple faults [50]. Staring from the test patterns for single faults, the proposed method inspects the propagation path of the SSAF to find the undetected MSAF and then generate the additional test patterns for those undetected faults. Differently from the chapter 3, we targets on all multiple faults. For any n multiple faults which is user-specified cardinality, the undetected faults by the test patterns for *n-1* faults are found, and the new test patterns for them are generated. By repeating the similar process, the proposed method can be inductively applied to handle all multiple faults. Moreover, since we only perform the test generation for the undetected faults, whose number are drastically smaller than all faults, we can generate a compact test set in an acceptable

running time. In addition, differently from the method introduced in chapter 3, we greatly improve the test generation speed by using the equivalence checking to quickly pick up and eliminate the redundant multiple faults, which is illustrated by the the experimental results up to the triple fault of ISCAS 89 and IWLS 2005 circuits.

The rest of the chapter is organized as follows. Section 4.2 explains the proposed incremental ATPG method for MSAF. Section 4.3 presents the experimental results for multiple faults. Section 4.4 summarizes the chapter and discusses the future work.

4.2 Proposed ATPG Method for Multiple Faults

In this section, the proposed method for the MSAF is explained. The way to extend the test patterns for the DSAF to the TSAF is presented and proved first. Then, we propose the general way to extend the method for all MSAFs.

4.2.1 Definitions

The term **blocking** describes the case when the fault cannot be propagated to any primary output through a specific path. For example, we assume there are two faults f_1 and f_2. f_1 is blocked by another fault f_2 under the test patterns v_1. However, it may be propagated to primary outputs through other propagation paths or by applying other test patterns.

4.2.2 ATPG Method for Triple Faults

As the number of the possible triple faults are very large, it is time-consuming to check the entire fault list to generate the test patterns. Therefore, similar to the double faults ATPG

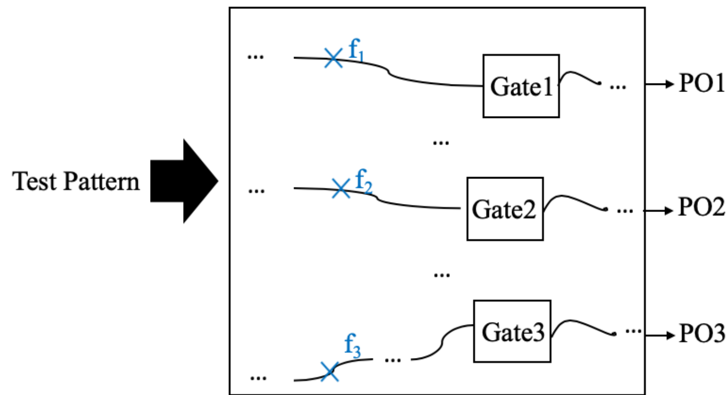


Fig. 4.1 Three SSAFs are independent

method, we inspect the propagation path to pick up the undetected triple faults. Comparing to the double faults method, there are more cases to be considered for the TSAF, which are introduced and proved as follows.

Proposition: We assume that the initial test patterns for single faults are given, and the test generation process of double faults is completed by picking up the undetected faults and generating additional test patterns. In other words, we have the test patterns to cover all single and double faults, and we also have the list of double faults that are undetected by the single test patterns.

There are three non-equivalent single faults f_1, f_2, and f_3. We check the propagation path of the single and double faults to obtain the list of undetected TSAF list. Only when three single faults violate the path constrains of each other, TSAF $\{f_1, f_2, f_3\}$ cannot be covered by the test patterns for single and double faults. In contrast, if three SSAFs are totally independent or only two SSAFs of them mutually block the propagation of each other, TSAF $\{f_1, f_2, f_3\}$ is definitely detectable by the existing test patterns.

(Sketch of proof) There are only three possible cases for the TSAF, as follows.

1. Three SSAFs respectively happen in different areas of the circuit, whose propagation paths

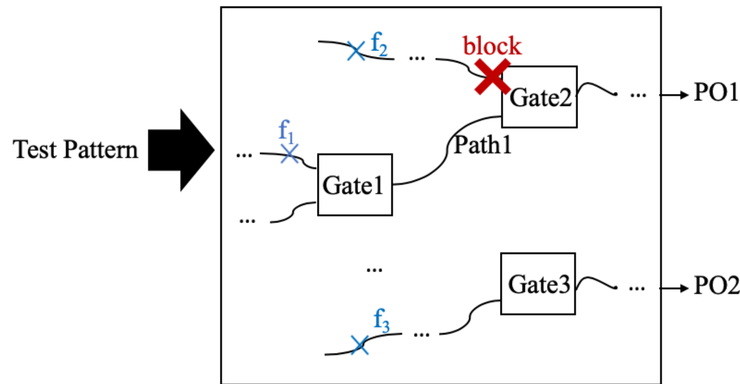


Fig. 4.2 Only two SSAFs mutually block the propagation path of each other

have no overlap, as shown in Fig. 4.1. It is obvious that three faults are totally independent, which means that their propagation paths have no intersection. Therefore, there is no way that three faults can violate the propagation constrains of each other. In this case, the TSAF $\{f_1, f_2, f_3\}$ is guaranteed to be detected by the single test patterns.

2. The second case is that only two faults f_1 and f_2 block the propagation path of each other, while the third fault f_3 is unaffected by other two faults, as shown in Fig. 4.2. Although the f_1 and f_2 may not be detected by the single test patterns, they must be selected as the undetected DSAF in the test generation of the double faults. Thus, the generated double test patterns can handle $\{f_1, f_2\}$. Moreover, the fault f_3 is detected by its related single test patterns. Consequently, we can ignore this kind of the TSAF.

3. Three SSAFs mutually block the propagation path of each other. In order to clearly explain in which case the TSAF is undetected, we want to classify the relationship of three SSAFs first, as follows. Here "$f_1 \rightarrow f_2$" is defined as that f_1 is blocked by f_2.

$f_1 \rightarrow f_2, f_3$

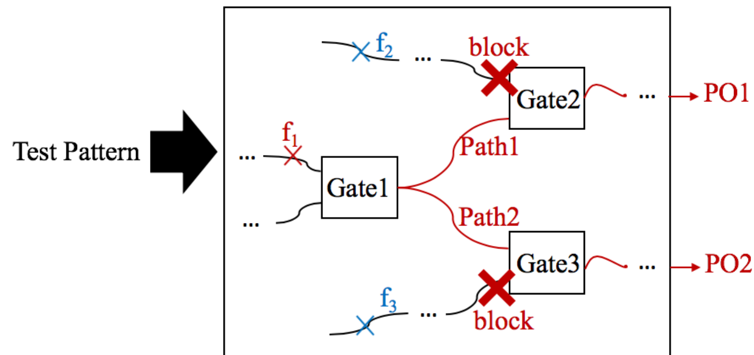


Fig. 4.3 SSAF f_1 has two propagation paths which are blocked by f_2 and f_3, respectively

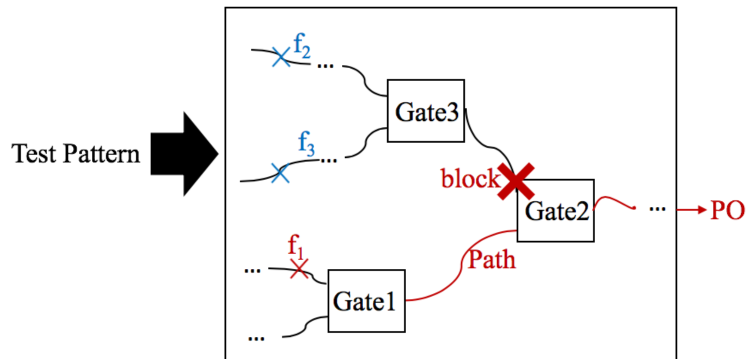


Fig. 4.4 SSAF f_1 has one propagation path which is blocked by f_2 and f_3

- f_1 can be propagated to output in two paths, which are blocked by other two faults f_2 and f_3, respectively, as shown in Fig. 4.3.
- There is only one propagation path for f_1. It blocked by the DSAF $\{f_2, f_3\}$ in Gate2, as shown in Fig. 4.4.

$\{f_1, f_2\} \rightarrow f_3$

- $\{f_1, f_2\}$ is an undetected DSAF selected by the fault filter of double faults. A new test pattern is generated to permit its propagation. However, the happening of f_3 violate its path constraints again, as shown in Fig. 4.5

The TSAF $\{f_1, f_2, f_3\}$ is undetected only in the following two cases.

1) $f_1 \rightarrow f_2, f_3,\quad f_2 \rightarrow f_1, f_3,\quad f_3 \rightarrow f_1, f_2.$

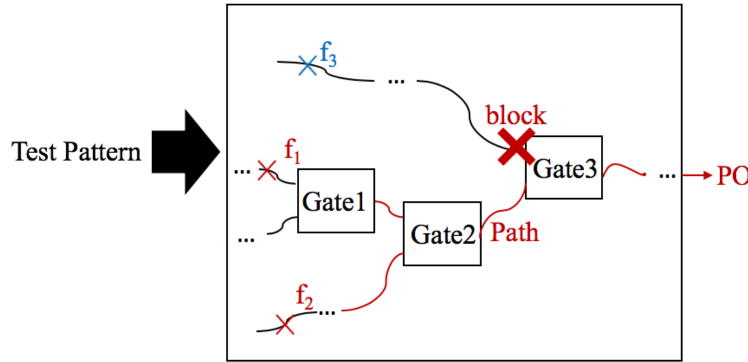


Fig. 4.5 DSAF $\{f_1, f_2\}$ is blocked by f_3

Each SSAF is blocked by another two SSAFs, simultaneously. As a result, three SSAFs are undetected by the related single test patterns. The TSAF $\{f_1, f_2, f_3\}$ is selected as a possible undetected fault. On the contrary, we ignore the case that only one or two SSAFs are blocked by other faults, since the fault unaffected by others is definitely detected by its single test pattern.

2) $f_1 \rightarrow f_2, f_3, \quad \{f_2, f_3\} \rightarrow f_1$

The propagation of the SSAF f_1 is blocked by the DSAF $\{f_2, f_3\}$. Meanwhile, the DSAF $\{f_2, f_3\}$ is selected in the test generation of double faults and has a new test pattern, but its path constraints are violated by the SSAF f_1 again. In this case, the TSAF $\{f_1, f_2, f_3\}$ can be detected by neither the single test pattern of f_1 nor the double test pattern for $\{f_2, f_3\}$. However, the TSAF is detectable if only the propagation of $\{f_2, f_3\}$ is prevented by f_1, while $\{f_2, f_3\}$ has no influence on f_1, and vice versa.

To conclude, these are all the cases when three faults simultaneously happen in the circuit. We do not target the TSAF belonging to case 1 and case 2, since they are already detected by the existing test patterns for the SSAF and DSAF. Only the TSAF in case 3 is selected by the proposed method. In fact, although the selected TSAF is undetected by the related DSAF and SSAF test patterns, they may be handled by other existing test patterns, if there

are multiple paths to be propagated to primary outputs. The fault simulation is performed to check the list of the possible undetected faults, and eliminate the actual detectable TSAFs to further compress the size of the undetected TSAF list. This process is very fast since only small number of faults are selected. Therefore, the total time to pick up the undetected TSAF which needs additional test patterns can be very small by using the proposed method.

(End of sketch of proof)

4.2.3 General Method to Extend the Proposed Method form n-1 faults to n faults

As we discussed in subsection 4.2.2, there are two cases needed to be checked to find the undetected faults.

1) $f_1 \rightarrow f_2, f_3$

- blocked in 2 paths.
- blocked in 1 path.

2) $f_{1(2)} \rightarrow f_3 \qquad (\{f_1, f_2\} \rightarrow f_3)$.

- blocked in 1 path.

where $f_{1(i)}$ indicates the multiple faults pair $\{f_1, f_2, f_3, ..., f_i\}$. f_1 in case 1) can be any of three faults, and in case 2) the DSAF $\{f_1, f_2\}$ is a combination of any two SSAFs out of three faults. The DSAF $\{f_1, f_2\}$ is transformed to a single fault following the method [29], which can simplify the fault selection process. Since case 2) is to find the single fault whose propagation path is blocked by another single fault, the fault selection method of the DSAF

can be reused to deal with this case. Hence, we only need to handle the case 1) to find the undetected TSAF. We should notice that, the faults in the left-hand side of the formula (f_1 in case 1) and $\{f_1, f_2\}$ in case 2)) are not redundant, otherwise we cannot analyze their propagation path. On the contrary, the faults in the right-hand side of the formula (f_2, f_3 in case 1) and f_3 in case 2)) may be redundant, since they can become non-redundant as TSAF.

Similarly, there are three cases we need to check to obtain the undetected quadruple faults.

1) $f_1 \rightarrow f_2, f_3, f_4$

- blocked in 3 paths.
- blocked in 2 paths.
- blocked in 1 path.

2) $f_{1(2)} \rightarrow f_3, f_4 \qquad (\{f_1, f_2\} \rightarrow f_3, f_4)$.

- blocked in 2 path.
- blocked in 1 path.

3) $f_{1(3)} \rightarrow f_4 \qquad (\{f_1, f_2, f_3\} \rightarrow f_4)$.

- blocked in 1 path.

where the DSAF $\{f_1, f_2\}$ and TSAF $\{f_1, f_2, f_3\}$ in case 2) and case 3) are converted to single faults $f_{1(2)}$ and $f_{1(3)}$, respectively, which means that they can be processed by the fault selection of the DSAF and the TSAF. Thus, we only need to inspect the case 1) to find the undetected quadruple faults.

Based on the example of the triple faults and quadruple faults, we want to discuss the general way to pick up the undetected n faults. Here we assume that we have already finished the fault selection and the test generation process for the fault whose cardinality is smaller or equal to *n-1*. Thus, we have the list of the undetected faults found in the previous fault selection process, and the test patterns to cover all single, double, ... *n-1* faults. If there are n SSAFs f_1, f_2, f_3, ... f_n in the circuit, in order to get the list of the undetected n faults pairs, we only need to take care of the case that n SSAFs mutually violate the path constraints with each other. The relationship of the faults can be classified as follows.

1) $f_1 \rightarrow f_2, f_3, f_4, ..., f_n$

- blocked in *n*-1 paths.
- blocked in *n*-2 paths.
- ...
- blocked in one path.

2) $f_{1(2)} \rightarrow f_3, f_4, ..., f_n \quad (\{f_1, f_2\} \rightarrow f_3, f_4, ..., f_n)$

- blocked in *n*-2 paths.
- blocked in *n*-3 paths.
- ...
- blocked in one path.

3) $f_{1(3)} \rightarrow f_4, ..., f_n \quad (\{f_1, f_2, f_3\} \rightarrow f_4, ..., f_n)$

- blocked in n-3 paths.
- blocked in n-4 paths.
- ...
- blocked in one path.

...

n-2) $f_{1(n-2)} \rightarrow f_{n-1}, f_n$ $(\{f_1, f_2, f_3, ..., f_{n-2}\} \rightarrow f_{n-1}, f_n)$

- blocked in two paths
- blocked in one path

n-1) $f_{1(n-1)} \rightarrow f_n$ $(\{f_1, f_2, f_3, ..., f_{n-1}\} \rightarrow f_n)$

- blocked in one path

$\{f_1, f_2, f_3, ..., f_i\}$ ($1 <= i < n$) is an undetected multiple faults pair found in the previous fault selection process. Similar to the triple and quadruple faults, following the method [29], the i faults pair $\{f_1, f_2, f_3, f_4, ..., f_i\}$ is transformed to a single fault $f_{1(i)}$ for the ease of the fault selection. The formula shown above indicates that although the undetected i faults pair is selected in the previous process and additional test patterns are generated for it, again, it is blocked by other n-i single faults. All we need to do is to find this kind of undetected i faults pair and generate new test for it. Moreover, as the case 2) to case n-1) can be processed by the fault selection of n-1 faults since we convert the multiple faults pair to a single fault, we only need to inspect the faults in case 1). When we inductively extend the method to all multiple faults, most of the cases we need to check can be handled by the previous fault

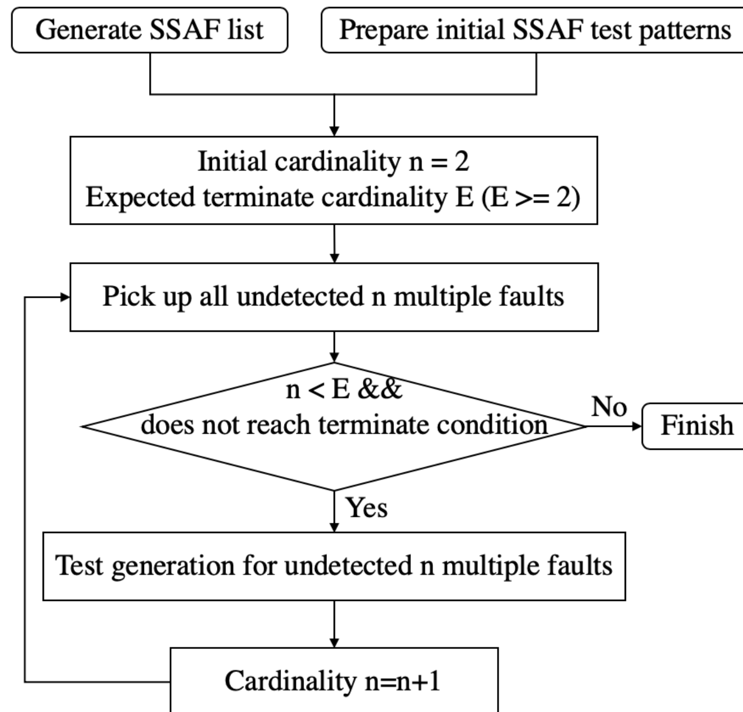


Fig. 4.6 The proposed ATPG flow

selection, while only small number of faults need to be taken care of. However, even if we only need to check the case 1), it seems to be complicated and time-consuming to find *n-i* SSAFs that violate the propagation constraints of $f_{1(i)}$, simultaneously, especially when n is large. Actually, because the number of the propagation paths and the possible faulty positions are limited in the circuit, as the increasing of the cardinality n, there are only few i faults pair blocked by exactly *n-i* SSAFs, and hence, they can be skipped to reduce the running time. In addition, many of the selected faults are actually detected by other existing test patterns since there may be multiple propagation paths to primary output. We can perform the fault simulation for them again to further compress the size of the undetected faults list.

4.2.4 Flow Chart of the Proposed Method

There is a termination point to stop the fault selection and test generation process. We assume that there are n SSAFs f_1, f_2, f_3, ..., f_n happen in the circuit. According to the previous discussion, the MSAF is selected as the undetected fault only when $\{f_1, f_2, f_3, ..., f_i\} \rightarrow f_{i+1}, ..., f_n$ ($1 <= i < n$). In other words, there are two conditions need to be satisfied for the formula. First condition is that there is undetected *i* faults pair $\{f_1, f_2, f_3, ..., f_i\}$ in the circuit. Meanwhile, we need to find exact *n-i* SSAFs blocking its propagation at the same time. We ignore the cases that there is no undetected *i* faults pair or there are undetected *i* faults pairs but we cannot find *n-i* SSAFs violate its path constraints. Actually, with the increasing of *i*, there are only few undetected *i* faults pair since they have more propagation paths and more probability to be propagated to primary outputs. On the contrary, when the *n-i* becomes larger, it is difficult to find exact *n-i* SSAFs block the propagation of *i* faults pair, because the propagation paths and the possible faulty positions are limited in the circuit. The test generation process can be terminated if we can find neither undetected *i* faults pair nor *n-i* SSAFs block its propagation. Actually, according to the Table 4.2 shown in Section 4.3, the DSAF test patterns for most of the circuits can already cover all TSAFs, which indicates that we can terminate the process after the test generation of triple faults. The reason is that we pick up the undetected *n* multiple faults based on the undetected *n-1* faults, and the number of undetected faults drastically decreases with the increasing of the cardinality. Therefore, few additional test patterns is required for triple or higher cardinality faults, as few undetected fault can be found.

After the fault selection process of the current fault cardinality is completed, the test generation is performed by using the SAT-solver. It should be finished very quickly because

the list of the undetected fault is extremely smaller comparing to all faults. Nevertheless, there are large amount of undetected faults that are actually redundant, which means that it is not necessary to generate test patterns for them. It takes a long time to exclude those redundant faults by using the SAT-solver, since SAT-solver have to try numerous solutions before it can decide that the problem is unsatisfied. Therefore, instead of utilizing the SAT-solver to skip the redundant multiple faults pairs, we apply the equivalence checking to make sure whether the fault is redundant or not. Specifically, after the processing of the SAT-solver exceeds a time limit, we remove the nodes which the target fault pairs located in, and if the removing does not affect the functionality of the circuit, we can skip the fault.

The flow chart of the proposed method is illustrated in Fig. 4.6. The initialization process includes the generation of the SSAF faults list and the preparation of the single test patterns. The initial SSAF list only includes the SSAFs at the gate inputs that are connected to a fan-out wire or PI (Primary Input), following the paper [45, 32, 39]. The test generation starts from double faults. After the process of fault selection and test generation of the current multiple faults pair is finished, the cardinality of the target fault is incremented for the next test generation. The test generation can be stopped if it reaches the user-specified cardinality E or the terminate condition we discuss before is satisfied. By repeating this process, the proposed method can be inductively applied to all multiple faults.

4.3 Experiment Results

We have implemented the proposed method in C++. Glucose 4.1 [7] is used as the SAT solver to generate the additional test patterns for multiple faults. ABC [10] is used to do the

Table 4.1 experiment circuits

Circuit	Gate #	PI #	PO #
s1494	712	14	25
s5378	1,912	214	218
s9234	2,534	247	250
s35932	16,047	1,763	2,048
s38417	16,047	1,664	1,742
s38584	16,009	1,464	1,730
i2c	1,575	147	142
spi	4,588	276	274
systemcdes	3,766	322	255
des_area	5,545	368	192
systemcaes	14,831	930	799
usb_funct	19,752	1,874	1,867
wb_conmax	52,658	1,900	2,168
des_perf	107,970	9,042	8,872

equivalence checking to exclude the redundant faults using "CEC" command. The proposed method is operated in a machine running Linux kernel 4.17 with Xeon E5-2699 v4 2.20GHz CPU and 512GB memory. ISCAS 89 [11] and IWLS 2005 [3] benchmark circuits are used to perform the experiment as shown in Table 4.1. The test generation of the ISCAS circuits starts from the compact single test set [14]. The single test patterns of the IWLS 2005 circuits are generated by a state-of-the-art commercial tool.

4.3.1 Performance Comparison

We compare the number of the additional test patterns and the running time of the proposed method to ABC using “&fftest" command. In order to verify the coverage of the generated test patterns, we exhaustively check all faults and test patterns in the circuit.

The number of the generated test patterns for the TSAF and DSAF are illustrated in Table 4.2. The second column is the number of the initial SSAF test patterns. The third and fourth

Table 4.2 Number of additional test patterns for DSAF and TSAF

Circuit	SSAF	ABC (DSAF)	ABC (TSAF)	Proposed Method (DSAF)	Proposed Method (TSAF)
s1494	110	2	0	2	0
s5378	102	3	0	4	0
s9234	134	10	4	9	0
s35932	30	NR	NR	21	5
s38417	120	NR	NR	18	0
s38584	174	NR	NR	20	0
i2c	77	NR	NR	5	0
spi	716	NR	NR	8	0
systemcdes	67	NR	NR	1	0
des_area	132	NR	NR	7	0
systemcaes	193	NR	NR	12	0
usb_funct	127	NR	NR	10	4
wb_conmax	152	NR	NR	309	1
des_perf	63	NR	NR	0	0

columns are the number of the additional test patterns for the DSAF and TSAF generated by ABC. Fifth and sixth columns are the number of the DSAF and TSAF test patterns generated by the proposed method. Notice that the TSAF test patterns are generated based on the test for the SSAF and DSAF. The “NR" (no result) in third and fourth columns indicates that ABC fails to finish the test generation within the time limitation (12 hours), while the proposed method can smoothly generate the test patterns for all circuits. We confirm the test coverage by exhaustively checking all the faults. The results illustrate that the test generated by our proposed method can detect all non-redundant DSAFs and TSAFs. Moreover, the additional test patterns for the DSAF and TSAF generated by the proposed method is as compact as ABC. In addition, in most of the circuits, there is no additional test patterns for the TSAF, since the test patterns for the SSAF and DSAF can already cover all TSAFs.

Table 4.3 Total runtime (minutes)

Circuit	ABC	Proposed Method
s1494	23.4	1.2
s5378	147	0.18
s9234	NR	7.9
s35932	NR	41.2
s38417	NR	8.4
s38584	NR	59.2
i2c	NR	5
spi	NR	183
systemcdes	NR	570
des_area	NR	89.7
systemcaes	NR	285
usb_funct	NR	11.4
wb_conmax	NR	93
des_perf	NR	160

The runtime comparison is shown in the Table 4.3. The processing speed of the proposed method is about 20 and 815 times faster than the ABC for s1494 and s5378 circuits, respectively, while ABC fails to generate test patterns for remaining circuits within the time limit. Besides, because the proposed method applies the SAT-solver to generate the additional test patterns, which is not good at handling the XOR gate, the test generation of the circuits such as wb_conmax or des_perf is time-consuming since they contain many XOR gates. It also causes the large number of additional double test patterns of wb_conmax in the Table 4.2.

N-detection has been proposed to detect each single fault by at least N different test patterns [8, 35, 48, 20]. We compare the number of test patterns generated by N-detection to the proposed method, as shown in Table 4.4. The test number of 2-detection and 5-detection are shown in the second, and third columns, respectively. The fourth column is the total number of the TSAF test generated by the proposed method. We verify their coverage of the

Table 4.4 Test number by N-detect and proposed method

Circuit	2-detect	5-detect	TSAF test (Proposed Method)
s1494	217	537	112
s5378	236	570	106
s9234	245	577	143
s35932	40	435	51
s38417	185	435	138
s38584	222	533	194
i2c	151	352	82
spi	1491	3551	724
systemcdes	129	309	68
des_area	227	465	139
systemcaes	338	749	205
usb_funct	242	585	137
wb_conmax	288	704	461
des_perf	130	282	63

TSAF by exhaustively checking all faults. Although the 2-detect and 5-detect test patterns can cover all TSAFs, their number are about 1.6 times and 4.2 times the test number of the proposed method, respectively.

4.3.2 Selected Faults Analysis

Table 4.5 illustrates the number of the selected TSAF on the basis of the existing DSAF and SSAF test patterns. The second and third columns are the total number of the SSAF and TSAF without fault collapsing. The fourth column shows the number of the selected TSAF, which is significantly smaller than total number of the TSAF owing to the proposed fault selection method. It drastically reduces the time of the test generation process. Notice that there is not selected TSAF in des_perf circuit, since its single and double test patterns can

Table 4.5 Number of selected TSAF in proposed method

Circuit	Total SSAF	Total TSAF	Selected TSAF
s1494	4,194	$7.3*10^{10}$	11
s5378	10,426	$1.1*10^{12}$	69
s9234	14,052	$2.7*10^{12}$	11,870
s35932	88,084	$6.8*10^{14}$	10,849
s38417	71,122	$3.6*10^{14}$	80
s38584	88,824	$7*10^{14}$	306,149
i2c	3,094	$2.9*10^{10}$	100
spi	9,442	$8.4*10^{11}$	35
systemcdes	7,904	$4.9*10^{11}$	62
des_area	11,316	$1.4*10^{12}$	12
systemcaes	30,466	$2.8*10^{13}$	633
usb_funct	39,678	$6.2*10^{13}$	1,400
wb_conmax	110,364	$1.3*10^{15}$	12,380
des_perf	210,944	$9.4*10^{15}$	0

already cover all TSAFs In addition, most of the selected faults are actually redundant. This is the reason why few additional test patterns are required to cover those undetected faults, as shown in Table 4.2.

The number of the three types of the TSAF introduced in 4.2.2 is shown in Table 4.6. The second, third and fourth columns illustrate the number of the SSAF blocked by other two SSAFs in one path, the SSAF blocked by other two SSAFs in two paths, and the DSAF blocked by a SSAF, respectively. The fifth column shows the number of the final selected TSAF by pairing the faults shown in second, third and fourth columns. Although many TSAF fault pairs are selected in second to fourth columns, only few of them remain in the fifth column after pairing the faults mutually violate the path constraints with each other.

Table 4.6 Number of three types of selected TSAF

Circuit	SSAF → Two SSAFs in one path	SSAF → Two SSAFs in two paths	DSAF → SSAF	Selected TSAF
s1494	33,631	4,179	25	11
s5378	236,286	10,370	122	69
s9234	976,799	13,720	35,958	11,870
s35932	897,463	77,396	417,278	10,849
s38417	11,996,276	70,968	2,315	80
s38584	3,516,390	86,787	3,871,139	306,149
i2c	76,917	8,793	965	100
spi	4,613,493	26,386	471	35
systemcdes	32,674,638	21,374	4,560	62
des_area	199,161,604	31,878	6,645	12
systemcaes	207,998,758	85,248	39,791	633
usb_funct	9,494,734	109,971	7,898	1,400
wb_conmax	22,248,329	298,572	38,382	12,380
des_perf	117,412,425	597,668	384	0

4.3.3 Runtime Analyzation

The runtime of each step in the proposed method is shown in the 4.7. The initialization process only takes few time as shown in the second column. In addition, the runtime of DSAF and TSAF in most of the circuits are almost the same order.

Table 4.8 shows the time of the fault selection, fault simulation and the test generation for the DSAF and TSAF, while the time of the initialization process is not included. The fault selection in second column is the process to find the undetected fault based on the method introduced in the Section 4.2. The fault simulation in third column is to exclude the selected faults that are actually detected by other existing test patterns. The test generation in the fourth column is to traverse the remaining faults in the list and generate the additional test patterns for them by applying the SAT-solver. In most of the circuit, fault selection method only takes small part of the execution time comparing to the fault simulation and test

Table 4.7 Runtime of each step in proposed method (minutes)

Circuit	Initialization	DSAF Processing	TSAF Processing	Total Runtime
s1494	0.4	0.4	0.4	1.2
s5378	0.07	0.08	0.03	0.18
s9234	0.1	4.4	3.4	7.9
s35932	1.5	33.7	6	41.2
s38417	1.7	1.4	5.3	8.4
s38584	1.8	11	46.4	59.2
i2c	0.1	2.9	2	5
spi	1	46	136	183
systemcdes	1	29	540	570
des_area	0.9	14	74.8	89.7
systemcaes	101	91	93	285
usb_funct	0.1	5.4	5.9	11.4
wb_conmax	1.3	20.9	70.8	93
des_perf	4.9	92.3	62.8	160

generation, which means that given efficient tools to perform the fault simulation and test generation, the runtime of the proposed method can be further reduced.

The equivalence checking is used to reduce the time to exclude the redundant fault. Table 4.9 compares the runtime of the test generation with the equivalence checking and without the equivalence checking. Notice that this table only includes the runtime of the test generation process, while the results in 4.3 shows the total runtime. The runtime of the smaller size circuits such as s1494, s5378, etc., does not change a lot since most of their redundant faults can be quickly excluded by using the SAT-solver, and hence the equivalence checking is rarely performed in those circuits. On the other hand, in the larger size circuits, the equivalence checking can reduce about 20% to 30% running time. Since it takes a long time to exclude the redundant faults in those circuits by utilizing SAT-solver, the proposed method uses the equivalence checking to quickly determine whether the fault is redundant

Table 4.8 Runtime of fault selection, fault simulation and test generation (minutes)

Circuit	Fault Selection	Fault Simulation	Test Generation
s1494	0.5	0.1	0.2
s5378	0.01	0.16	0.01
s9234	0.5	1.5	6.1
s35932	7.1	19	15.1
s38417	4.7	3.4	0.3
s38584	4.2	24.9	30.1
i2c	1	2	2
spi	60	67	51
systemcdes	210	220	140
des_area	26	41.7	22
systemcaes	87	47	151
usb_funct	5	1.9	4.5
wb_conmax	16.8	26.2	50
des_perf	68.7	9.3	82

fault or not. It proves that the equivalence checking is feasible to accelerate the speed of the test generation.

4.4 Conclusion

In this chapter, we have proposed an ATPG method for MSAF. Starting from a complete test set of the SSAF, the proposed method can be incrementally applied to all MSAFs. The way to extend the proposed method to the TSAF is introduced and proved at first as an example. The general way to inductively extend the method for all MSAFs is explained. In addition, we discuss the strategy to reduce the processing time for redundant faults. The experimental results up to the TSAF prove that the proposed method can successfully select the undetected faults and generate compact test patterns for the multiple faults within an acceptable running

Table 4.9 The comparison of Test Generation with and without Equivalence Checking

Circuit	Without Equivalence Checking	With Equivalence Checking
s1494	0.2	0.2
s5378	0.01	0.01
s9234	6.6	6.1
s35932	22.9	15.1
s38417	0.3	0.3
s38584	42.9	30.1
i2c	2	2
spi	55	51
systemcdes	205	140
des_area	28.3	22
systemcaes	208	151
usb_funct	4.5	4.5
wb_conmax	63	50
des_perf	114	82

time. Moreover, the order of the magnitude of the run-time for the TSAF is almost the same as the DSAF.

Our future work is to further optimize the implementation by integrating the proposed method with a more powerful commercial ATPG tool for SSAF, and figure out a more efficient way to check the coverage of the test patterns, instead of exhaustively inspecting the entire fault list.

Chapter 5

A Logic Optimization Method based on the Proposed ATPG Method for Multiple Faults

5.1 Introduction of Logic Optimization

Logic optimization methods have been studied since 1980's which resulted in SIS [40] logic synthesis tool and later in ABC [10] logic synthesis and verification tool. Redundancy removal command in logic synthesis tool SIS can optimize the circuit by eliminating the single redundant faults. Authors in [16] try to remove the redundancies in the circuit by repeatedly adding redundancy and then removing other redundancies. Authors in [43] utilize multiple faults to simplify circuit logics by generating a new circuit with the function approximate to the original circuit. In other words, the circuit function may be changed. However, none of these methods consider the optimization by removing the redundant

multiple faults in the circuit without the function change, which can potentially eliminate more redundancies and make circuit more compact.

In this chapter, we propose a new logic optimization method by identifying the redundant multiple stuck-at faults (MSAF) and removing their related gates. The fault selection method introduced in the previous chapters is applied to find the redundant multiple faults. According to the experimental results shown in chapter 6, by eliminating the MSAF from higher cardinality to lower cardinality, the proposed method can remove more redundant logic comparing to the methods that only eliminate single stuck-at faults (SSAF).

The rest of the chapter is organized as follows. Section 5.2 presents the fault selection method to find multiple faults. Section 5.3 explains the proposed fault removal method. Section 5.4 illustrates the experimental results. Section 5.5 concludes the chapter and discusses the future work.

5.2 Finding the Redundant Multiple Faults

The naive way to find redundant faults is traversing the entire fault list and checking one by one. However, it cannot handle multiple faults due to the large number of fault combinations. Inspired by the proposed method in the previous chapters, instead of checking all multiple faults, we want to check the fault propagation path to find the redundant multiple fault, which can greatly reduce the time, as shown in the experimental results of chapter 3 and 4. The DSAF consisting of two SSAFs that block propagation path with each other is selected as a potential redundant fault. In addition, the DSAF consists of two redundant single faults is also classified as the potential redundant fault. Then, we can pick up the redundant DSAF by

performing fault simulation and test generation. This process is very fast since the size of the potential redundant fault list is drastically smaller than all faults. Therefore, starting from a compact test set for the single fault, we can incrementally find the redundant multiple faults in an acceptable running time for fairly large circuits.

5.3 Removal of the Redundant Multiple Faults from Higher to Lower Cardinality

The composition of the redundant MSAF is shown in Table 5.1, whose second, third, and forth columns are the number of the redundant DSAFs including two non-redundant SSAFs, one non-redundant and one redundant SSAFs, and two redundant SSAFs, respectively. The redundant DSAF consisting of two redundant SSAFs is equal to two SSAFs as redundant faults, while the DSAF include at least one non-redundant SSAF is different from two redundant SSAFs. Obviously, most of the redundant DSAFs include at least one redundant SSAF.

There are two ways to remove the redundant multiple faults. The first way is to remove the faults from the lower cardinality to higher cardinality. However, according to the experimental results shown in Table 5.1, we cannot find any redundant multiple faults such as double faults if we remove all the single faults at first, because most of the redundant multiple faults include at least one redundant single fault. In other words, if we clear all redundant single faults, most of the original redundant multiple faults become non-redundant; hence, we cannot remove them to optimize the circuit structure. In contrast, if we remove the redundant faults from higher cardinality to lower cardinality, we can remove the redundant single fault

Table 5.1 Composition of the Redundant DSAF

Circuit	Two NoRe-SSAF	NoRe-SSAF+Re-SSAF	Two Re-SSAF	Total Re-DSAF
s444	0	21	30	51
s832	0	11	35	46
s1238	0	103	1,583	1,686
s1423	0	36	36	72
s1494	0	19	17	36
s5378	0	80	227	307
s9234	0	1,579	16,994	18,577
s35932	1	4,128	437,248	441,376
s38417	1	20,507	95,983	116,490

as well as the non-redundant single fault that are included in the redundant multiple faults, which means that we can make the circuit smaller in size and more compact.

Similarly, in order to optimize more redundancies, in the logic optimization process of the MSAFs with a same cardinality, the MSAF with more non-redundant SSAFs is removed first. Since many MSAFs include same redundant SSAFs, if we remove the MSAF with more redundant SSAFs first, many of the redundant MSAFs with non-redundant SSAFs is not redundant any more, which decreases the number of redundancies to be removed.

5.4 Experimental Results

We use the Glucose 4.1 [7] as the SAT solver to generate the test patterns and find redundant faults. We perform the experiment with ISCAS 89 [11] and IWLS 2005 [3] benchmark circuits. The fault selection of the ISCAS 89 circuits starts from the compact test set for the SSAF [14]. The single test patterns of the IWLS 2005 circuits are generated by a commercial tool.

Table 5.2 Number of Gate and Redundant DSAF and SSAF

Circuit	Gate	Re-SSAF	Re-DSAF (Has Re-SSAF)	Re-DSAF (No SSAF)	Used Re-DSAF
s444	212	18	51	0	9
s832	404	12	46	0	6
s1238	597	84	1,686	0	44
s1423	635	28	72	0	8
s1494	712	15	36	0	6
s5378	1,912	56	307	0	15
s9234	2,534	332	18,577	0	130
s35932	16,047	10,688	441,376	0	1,920
s38417	16,047	154	441	0	87
s38584	16,009	2,037	116,490	0	651
des_area	5,545	16	44	0	6
systemcaes	14,831	282	3,804	0	142
usb_funct	19,752	633	6,747	0	218
wb_conmax	52,658	8,918	221,264	0	3,127

In Table 5.2, the experiment results illustrated from the second column to the sixth column are the number of the gates and the redundant SSAF and DSAF. The second column is the number of gates in the circuit. The third column is the number of all redundant SSAFs. The fourth column is the number of the selected redundant DSAF if we do not remove the redundant single fault in the circuit. The fifth column is the number of selected redundant double faults after we remove all redundant single fault in the circuit. Obviously, no redundant double faults is found if we clear all single redundancy, because most of the redundant double faults include at least one redundant single fault in these circuits. The sixth column is the number of redundant DSAF that actually used to optimize the circuit. Many redundant DSAFs may include the same redundant or non-redundant SSAF, which means that only a portion of DSAFs can already cover all the gates we can optimize. Consequently,

Table 5.3 The Removed Gates by Optimizing Re-SSAF and DSAF

Circuit	Re-SSAF Removed Gates	Re-DSAF Removed Gates	Total runtime (minute)
s444	12	13	0.003
s832	9	9	0.008
s1238	60	60	0.45
s1423	16	16	0.03
s1494	11	11	0.8
s5378	23	23	0.16
s9234	185	201	4.5
s35932	2,304	2,560	35
s38417	134	139	3.1
s38584	841	865	13
des_area	12	13	14.9
systemcaes	177	204	200
usb_funct	336	344	5.5
wb_conmax	4,786	5,100	22.3

the actual number of DSAF used to eliminate the circuit redundancy is much smaller than the total number of DSAF.

The number of removed gates by deleting the faults is shown in second and third columns of Table 5.3. The seventh column is the number of gates that can be removed by optimizing the redundant single fault using the redundancy removal command in the logic synthesis tool SIS [40]. The eighth column illustrates the number of gates removed by eliminating the selected DSAF first and then remove the SSAF. Notice that the number of redundant SSAF and DSAF in the third and fourth columns are larger than the removed gates in seventh and eighth columns, respectively, because some of the redundant SSAFs locate in the same gate. In addition, although the number of the redundant DSAF is one or two order of magnitudes larger than the SSAF, the improvement ratio is not that much, because most of the DSAFs consist of two redundant single faults, and only a few of the DSAFs include a non-redundant

SSAF. Therefore, in most of the cases, the removal of the redundant DSAF is to remove the redundant SSAF. In the smaller size circuits, such as S444 and S832, the number of the removed gates by optimizing the single and double redundancy shown in seventh and eighth columns are almost the same, since most of the DSAFs in those circuits consist of only the redundant SSAF. Therefore, if we eliminate all the redundant DSAFs, the redundant SSAFs are eliminated as well. In contrast, in larger size circuits such as systemcaes, 15% more gates can be removed if we eliminate the DSAF first, since many of the redundant DSAFs in those circuits include the non-redundant SSAF. It illustrates that the proposed method is capable of optimizing the circuit and removing more redundancy. In addition, we have performed the experiment by removing the double faults in every possible combinations. However, the number of eliminated gates are smaller than the results in Table 5.3. It is because that the proposed method removes the redundant DSAF including a non-redundant SSAF first, which guarantees that more redundancies can be eliminated. The experimental results illustrate that the proposed method is capable of optimizing the circuit and removing more redundancies following the proposed heuristic of the fault removal.

The total running time is shown in the fourth column of Table 5.3. Actually, fault selection process of the redundant MSAFs takes most of the running time. Once we obtain the redundant fault list, the optimization process can be completed very fast. Most of the circuits can finish the entire optimization process in an acceptable time. Notice that, the circuit such as systemcaes takes more than 1 hour to finish the redundant fault selection process, since our method introduced in chapter 3 and 4 are based on SAT-solver, which is not good at handling the circuit with a large number of XOR gates.

5.5 Conclusion

In this chapter, we have proposed a logic optimization method by first identifying and then removing the redundant multiple faults. The incremental ATPG method is used to pick up the redundant multiple faults. By eliminating the redundant faults from higher cardinality to lower cardinality, more redundant logics can be optimized. The experimental results show that the proposed method can remove more redundancy comparing to the redundancy removal command of SIS, which proves the feasibility of the proposed method. Moreover, the experimental results prove that the proposed method can finish the process of the redundant fault selection and removal within an acceptable time. Our future work is to make our implementation more efficient and utilize higher cardinality of faults, such as redundant triple faults, in the optimization.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Since it is unavoidable for faults to happen in the fabricated chips, we need to use test patterns to thoroughly test and determine whether the functionalities of the fabricated chips are correct or not before shipping to consumers. Automatic Test Pattern Generation technology has been developed to prepare as small as possible set of test patterns which cover almost all the faults in a chip. Currently, not only single faults, but also multiple faults occur in the fabricated circuits. Previously research shows that we cannot generate sufficient test patterns to cover all faults without applying the multiple fault model. Besides, multiple fault analysis is required for test generation, fault simulation and logic optimization in some situations. Therefore, we need to analyze the possible multiple faults in the circuit and generate the multiple test patterns to thoroughly test and determine whether the functionalities of the fabricated chips are correct or not before shipping to consumers. Nevertheless, the previous

test generation methods cannot generate sufficient test patterns for multiple faults within an acceptable running time.

In this thesis, an incremental ATPG method for multiple faults is proposed. The ATPG method for double faults is introduced at first. It starts with the ATPG for single faults and incrementally adds new test patterns for double faults. The complexity of each incremental process is the same as the ATPG for the SSAF, thus the proposed method can be very efficient for large circuits. In order to drastically decrease the size of the DSAF list, the proposed DSAF filter excludes most of the DSAFs which do not need new test patterns. As shown in the experiments up to double faults, the numbers of the additional test patterns are small; hence, scalable.

Then, we propose the method to extend the ATPG method to all MSAF. Starting from a complete test set of the SSAF, the proposed method can be incrementally applied to all MSAFs. The way to extend the proposed method to the TSAF is introduced and proved at first as an example. The general way to inductively extend the method for all MSAFs is explained. In addition, we discuss the strategy to reduce the processing time for redundant faults. The experimental results show that we can generate compact test patterns within an acceptable running time. Moreover, the order of the magnitude of the run-time for the TSAF is almost the same as the DSAF.

Based on the proposed incremental ATPG method, we introduce a logic optimization method. By eliminating the redundant faults from higher cardinality to lower cardinality, more redundant logics can be optimized. The experimental results show that the proposed method can remove more redundancies comparing to the redundancy removal command of

SIS, which proves the feasibility of the proposed method. The results prove that the proposed method can remove more redundancy comparing to only delete the single redundancy.

6.2 Future Work

Our future work is to further optimize the implementation of the ATPG method for multiple faults by integrating the proposed method with a more powerful commercial ATPG tool for SSAF, and figure out a more efficient way to generate the test patterns for the selected faults. Current way of test generation is repeating the process that randomly picking up a undetected faults and generating test pattern. The method is naive and may results in the generation of redundant test patterns. The test pattern can be further optimized if we have a better heuristic to decide the sequence of choosing the undetected faults for test generation.

In addition, we will figure out a faster way to check the coverage of the test patterns. Presently we go through the entire fault list to check the generated test patterns, which is drastically time-consuming especially for large scale circuits. The runtime can be reduced if we can efficiently compress the fault list by combining the faults that are equivalent, and eliminate the faults such as the redundant faults, which are not necessary to be confirmed. Moreover, we can develop the emulator in hardware to examine the coverage, which can significantly accelerate the processing speed.

In order to improve the performance of the logic optimization method, we will utilize higher cardinality of faults, such as redundant triple faults, in the optimization. As we discussed in chapter 5, we can eliminate more redundant logic if we remove the redundant multiple faults which include at least one irredundant single fault. Currently, few research

utilize the redundant multiple fault to optimize the circuit since it is time-consuming to pick up the redundant faults. It is impractical to traverse the entire fault list to find the target faults especially for the faults with higher cardinality. However, following the fault selection method proposed in chapter 3 and chapter 4, we can incrementally select redundant multiple faults within an acceptable running time.

Although we only discuss the ATPG method for stuck-at fault in this thesis, we can apply the similar idea of the proposed method to detect the multiple faults of other fault models, such as the bridging fault. The test patterns for multiple faults can still be incrementally generated based on the single test patterns.

References

[1] Agrawal, A., Saldanha, A., Lavagno, L., and Sangiovanni-Vincentelli, A. L. (1997). Compact and complete test set generation for multiple stuck-faults. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 212–219. IEEE Computer Society.

[2] Akers, S. B. (1978). Binary decision diagrams. *IEEE Transactions on computers*, (6):509–516.

[3] Albrecht, C. (2005). Iwls 2005 benchmarks. In *International Workshop for Logic Synthesis (IWLS)*, page 9.

[4] Amaru, L., Gaillardon, P.-E., and De Micheli, G. (2015). Majority-inverter graph: A new paradigm for logic optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(5):806–819.

[5] Amarú, L., Vuillod, P., Luo, J., and Olson, J. (2017). Logic optimization and synthesis: Trends and directions in industry. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1303–1305. IEEE.

[6] Anita, J. and Vanathi, P. (2014). Genetic algorithm based test pattern generation for multiple stuck-at faults and test power reduction in vlsi circuits. In *2014 International Conference on Electronics and Communication Systems (ICECS)*, pages 1–6. IEEE.

[7] Audemard, G. and Simon, L. (2018). On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001.

[8] Benware, B., Schuermyer, C., Ranganathan, S., Madge, R., Krishnamurthy, P., Tamarapalli, N., Tsai, K.-H., and Rajski, J. (2003). Impact of multiple-detect test patterns on product quality. In *null*, page 1031. Citeseer.

[9] Biere, A., Heule, M., and van Maaren, H. (2009). *Handbook of satisfiability*, volume 185. IOS press.

[10] Brayton, R. and Mishchenko, A. (2010). Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer.

[11] Brglez, F., Bryan, D., and Kozminski, K. (1989). Combinational profiles of sequential benchmark circuits. In *IEEE international symposium on circuits and systems*, volume 3, pages 1929–1934. IEEE.

[12] Cox, H. and Rajski, J. (1988). A method of fault analysis for test generation and fault diagnosis. *IEEE transactions on computer-aided design of integrated circuits and systems*, 7(7):813–833.

[13] Czutro, A., Reddy, S. M., Polian, I., and Becker, B. (2014). Sat-based test pattern generation with improved dynamic compaction. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 56–61. IEEE.

[14] Eggersglüb, S., Schmitz, K., Krenz-Bååth, R., and Drechsler, R. (2014). Optimization-based multiple target test generation for highly compacted test sets. In *2014 19th IEEE European Test Symposium (ETS)*, pages 1–6. IEEE.

[15] Eggersglüß, S., Wille, R., and Drechsler, R. (2013). Improved sat-based atpg: More constraints, better compaction. In *Proceedings of the international conference on computer-aided design*, pages 85–90. IEEE Press.

[16] Entrena, L. A. and Cheng, K.-T. (1995). Combinational and sequential logic optimization by redundancy addition and removal. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(7):909–916.

[17] Fujita, M. and Mishchenko, A. (2014). Efficient sat-based atpg techniques for all multiple stuck-at faults. In *2014 International Test Conference*, pages 1–10. IEEE.

[18] Fujiwara, H. (1985). Fan: A fanout-oriented test pattern generation algorithm. In *IEEE International Symposium on Circuits and Systems*, pages 671–674.

[19] Fujiwara, H. and Shimono, T. (1983). On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, (12):1137–1144.

[20] Geuzebroek, J., Marinissen, E. J., Majhi, A., Glowatz, A., and Hapke, F. (2007). Embedded multi-detect atpg and its effect on the detection of unmodeled defects. In *2007 IEEE International Test Conference*, pages 1–10. IEEE.

[21] Goel, P. (1981). An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE transactions on Computers*, (3):215–222.

[22] Goel, P. (1995). An implicit enumeration algorithm to generate tests for combinational logic circuits. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995,'Highlights from Twenty-Five Years'.*, page 337. IEEE.

[23] Haaswijk, W., Collins, E., Seguin, B., Soeken, M., Kaplan, F., Süsstrunk, S., and De Micheli, G. (2018). Deep learning for logic optimization algorithms. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE.

[24] Holder, M. E. (2005). A modified karnaugh map technique. *IEEE Transactions on Education*, 48(1):206–207.

[25] Kajihara, S., Murakami, A., and Kaneko, T. (1999). On compact test sets for multiple stuck-at faults for large circuits. In *Proceedings Eighth Asian Test Symposium (ATS'99)*, pages 20–24. IEEE.

[26] Kajihara, S., Nishigaya, R., Sumioka, T., and Kinoshita, K. (1994). Efficient techniques for multiple fault test generation. In *Proceedings of IEEE 3rd Asian Test Symposium (ATS)*, pages 52–56. IEEE.

[27] Kajihara, S., Sumioka, T., and Kinoshita, K. (1993). Test generation for multiple faults based on parallel vector pair analysis. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 436–439. IEEE.

[28] Karnaugh, M. (1953). The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599.

[29] Kim, Y. C., Saluja, K. K., and Agrawal, V. D. (2002). Multiple faults: Modeling, simulation and test. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 592. IEEE Computer Society.

[30] Kirkland, T. and Mercer, M. R. (1988). Algorithms for automatic test-pattern generation. *IEEE Design & Test of Computers*, 5(3):43–55.

[31] Lee, C.-Y. (1959). Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999.

[32] Lioy, A. (1993). On the equivalence of fanout-point faults. *IEEE transactions on computers*, 42(3):268–271.

[33] Matrosova, A., Loukovnikova, E., Ostanin, S., Zinchuck, A., and Nikolaeva, E. (2007). Test generation for single and multiple stuck-at faults of a combinational circuit designed by covering shared robdd with clbs. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, pages 206–214. IEEE.

[34] Moore, C. J., Wang, P., Gharehbaghi, A. M., and Fujita, M. (2017). Test pattern generation for multiple stuck-at faults not covered by test patterns for single faults. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE.

[35] Pomeranz, I. and Reddy, S. M. (2007). Forming n-detection test sets without test generation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(2):18.

[36] Quine, W. V. (1952). The problem of simplifying truth functions. *The American mathematical monthly*, 59(8):521–531.

[37] Quine, W. V. (1955). A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631.

[38] Roth, J. P. (1966). Diagnosis of automata failures: A calculus and a method. *IBM journal of Research and Development*, 10(4):278–291.

[39] Sang-In, A. and Cheung, P. Y. (1994). A method of representative fault selection in digital circuits for atpg. In *Proceedings of IEEE International Symposium on Circuits and Systems-ISCAS'94*, volume 1, pages 73–76. IEEE.

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

[41] Shah, T., Matrosova, A., Kumar, B., Fujita, M., and Singh, V. (2017). Testing multiple stuck-at faults of robdd based combinational circuit design. In *2017 18th IEEE Latin American Test Symposium (LATS)*, pages 1–6. IEEE.

[42] Shi, J., Fey, G., Drechsler, R., Glowatz, A., Hapke, F., and Schloffel, J. (2005). Passat: Efficient sat-based test pattern generation for industrial circuits. In *IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05)*, pages 212–217. IEEE.

[43] Shin, D. and Gupta, S. K. (2011). A new circuit simplification method for error tolerant applications. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE.

[44] Stephan, P., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1996). Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176.

[45] To, K. (1973). Fault folding for irredundant and redundant combinational circuits. *IEEE Transactions on Computers*, 100(11):1008–1015.

[46] Ubar, R., Kostin, S., and Raik, J. (2012). Multiple stuck-at-fault detection theorem. In *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 236–241. IEEE.

[47] Ubar, R., Oyeniran, S. A., Scholzel, M., and Vierhaus, H. T. (2015). Multiple fault testing in systems-on-chip with high-level decision diagrams. In *2015 10th International Design & Test Symposium (IDT)*, pages 66–71. IEEE.

[48] Venkataraman, S., Sivaraj, S., Amyeen, E., Lee, S., Ojha, A., and Guo, R. (2004). An experimental study of n-detect scan atpg patterns on a processor. In *22nd IEEE VLSI Test Symposium, 2004. Proceedings.*, pages 23–28. IEEE.

[49] Wang, P., Gharehbaghi, A. M., and Fujita, M. (2019a). Automatic test pattern generation for double stuck-at faults based on test patterns of single faults. In *20th International Symposium on Quality Electronic Design (ISQED)*, pages 284–290. IEEE.

[50] Wang, P., Gharehbaghi, A. M., and Fujita, M. (2019b). An incremental automatic test pattern generation method for multiple stuck-at faults. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–6. IEEE.

[51] Wang, P., Moore, C. J., Gharehbaghi, A. M., and Fujita, M. (2017). An atpg method for double stuck-at faults by analyzing propagation paths of single faults. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(3):1063–1074.

[52] Wang, Z., Marek-Sadowska, M., Tsai, K.-H., and Rajski, J. (2006). Analysis and methodology for multiple-fault diagnosis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):558–575.

[53] Yu, C., Ciesielski, M., Choudhury, M., and Sullivan, A. (2016). Dag-aware logic synthesis of datapaths. In *Proceedings of the 53rd Annual Design Automation Conference*, page 135. ACM.

Publications and Awards

Publications

Journal

[1] Peikun Wang, Conrad Jinyong Moore, Amir Masoud Gharehbaghi and Masahiro Fujita. "An ATPG Method for Double Stuck-At Faults by Analyzing Propagation Paths of Single Faults." *IEEE Transactions on Circuits and Systems I*: Regular Papers, Volume 65 Issue 3, Page 1063-1074, 2017.

[2] Peikun Wang, Amir Masoud Gharehbaghi and Masahiro Fujita, "An Automatic Test Pattern Generation Method for Multiple Stuck-at Faults by Incrementally Extending the Test Patterns", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Accepted).

[3] Peikun Wang, Amir Masoud Gharehbaghi and Masahiro Fujita, "A Logic Optimization Method by Eliminating Redundant Multiple Faults from Higher to Lower Cardinality", *IPSJ Transactions on System LSI Design Methodology (T-SLDM)* (Accepted).

Conference

[4] Peikun Wang, Amir Masoud Gharehbaghi and Masahiro Fujita, "Automatic Test Pattern Generation for Double Stuck-at Faults Based on Test Patterns of Single Faults", *in 20th IEEE International Symposium on Quality Electronic Design (ISQED),* pp. 284-290, March 2019.

[5] Peikun Wang, Amir Masoud Gharehbaghi and Masahiro Fujita, "An Incremental Automatic Test Pattern Generation Method for Multiple Stuck-at Faults", in 2019 *IEEE 37th VLSI Test Symposium (VTS)*, pp. 1-6, April 2019.

Award:

- Third Award in International Programming Contest of 2017 International Conference on Computer Aided Design (ICCAD) (3/100).
- 2019 System LSI Design Methodology (SLDM) Best Presentation Award.
- 2019 IEEE CEDA All Japan Joint Chapter Academic Research Award.