IMPLICIT CUBE-DISTANCE FAULT MODELING FOR VERIFICATION AND FUNCTIONAL TESTING APPLICATIONS

Lucas W. B. Lee

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science (Electrical Engineering) at
Concordia University
Montreal, Quebec, Canada

September 2005

© Lucas W. B. Lee, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-10241-1
Our file Notre référence
ISBN: 0-494-10241-1

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

Implicit Cube-Distance Fault Modeling for Verification and Functional Testing Applications

Lucas W. B. Lee

With device manufacturing entering the sub-micron technology while complexity of the circuits continues to grow, the different types of design errors and manufacturing errors also spread in variety. It remains important to have a fault model that properly emulates the large classes of problems which are most likely to occur throughout the design cycle. The implicit cube error replacement modeling is proposed to satisfy modeling of design errors, while maintaining a comparable or shorter fault list to current methods, such as explicit gate replacement and gate replacements with MIGSE modules. The proposed error modeling is also applied to functional testing of FPGAs, where the look-up table (LUT) based logic may contain an even larger variety of functional errors which the implicit cube error replacement modeling is used to check. Simulation techniques with random or upper-lattice layer vectors will be looked at, as well as incorporating satisfiability (SAT)-based automatic test pattern generation (ATPG) algorithms from the *s-a-v* model into the new model. A prototype implementation of the design-verification-testing flow of FPGA is also examined.

To my mother

ACKNOWLEDGEMENTS

I thank my supervisor Professor Katarzyna Radecka. This work would not have been possible without her support and countless amounts of valuable guidance. I also thank her for her generosity of sending me down to the IEEE MWSCAS 2005 conference in Cincinnati to deliver lecture presentations. I thank her for allowing me the flexibility and freedom in all aspects of the research, the chance to supervise her undergraduate project group and be the tutor of her digital system design course. It has been a blessing to be able to work with her throughout the duration of my graduate studies at Concordia. I also thank Professor Zeljko Zilic from McGill for providing informative suggestions for my work and for allowing me to attend the ICPP 2004 conference in Montreal. I wish their family the best in years to come.

I thank Ngoc Lam Bui for all his morale support and his positive outlook of life. He is truly empathic, logical and supportive, and has been there not only during the highs, but also the lows. I feel very blessed that our paths have crossed in life and it has been an honor to know someone like him.

I thank Steven Shi for giving me the opportunity to work at helpdesk and for providing me some of the computer equipment I needed for my studies.

I thank Tadeusz (Ted) Obuchowicz for his help in the VLSI lab and his support on the Xilinx boards.

Most importantly, I would like to show my biggest wholehearted gratitude to my mother for her support and unlimited patience throughout every breathing moment of my life. I thank her for her continuous encouragement throughout my studies.

CONTENTS

1 Introduction 1

1.1 Design-Manufacturing Flow 2

1.1.1 FPGA Emulation 2

1.1.2 Verification Techniques 3

1.1.3 Simulations 4

1.2 Functional Verification and Testing Relationship 5

1.3 Thesis Overview 5

2 Introduction to Error Modeling 7

2.1 Error Modeling in Testing: Stuck-At-Value Fault Model 7

2.1.1 Fault Detection 8

2.1.2 Limitations of S-A-V Modeling 10

2.2 Design Functional Error Modeling 10

2.2.1 Gate Replacement Error Modeling with Gates of Compatible I/O Count 10

2.2.2 Gate Replacement Errors by MIGSE Modeling Modules with S-A-V Faults 11

2.2.3 Other Fault Modeling Methods 14

2.3 Explicit and Implicit Error Modeling 16

2.3.1 Explicit Error Modeling 16

2.3.2 Fault List Sizes Based on Explicit Error Modeling 16

2.3.3 Motivation of Implicit Error Modeling of Design Faults 18

3 Implicit Error Modeling Method by Cube Error Distances 19

3.1 Implicit Error Modeling of Gate-Level Errors 19

3.1.1 Explicit Errors as Functional Cube-Distances 21

3.1.2 Cube-Distance Fault List Reduction Using Dominance 22

3.1.3 Verification with Minimal Cube-Distance Errors 25

3.2 Fault List Sizes with Explicit Error Modeling 26

3.2.1 Explicit Gate Replacements 26

3.2.2 Gate Replacements with MIGSE Modules 28

4 Fault Detection by Simulation and SAT-based ATPG 29

4.1 Fault Detection by Simulation 29

4.1.1 Simulation Using Random Vectors 30

4.1.2 Simulation Using Lattice Structure 30
4.2 Automatic Test Pattern Generation by Satisfiability 31
4.2.1 Obtaining SAT Clauses of Boolean Logic 32
4.2.2 Complexity 33
4.2.3 Good, Faulty and Active Clauses with Presence of a *S-A-V* Fault 33
4.3 Redundant Fault Identification Method for Cube Replacement Errors 36
4.3.1 Redundant Faults Caused by Redundancy in Circuits 36
4.3.2 Impact of Don't-Care (DC) Conditions on Redundant Cube Distance Replacement Faults 37
4.3.3 Redundancy Identification of Cube Distance Replacement Errors 39

5 Fault Detection of Combinational Designs with Berkeley SIS **41**
5.1 Overview of Experiments with SIS 41
5.1.1 SIS Programming Libraries 42
5.1.2 Design Preprocessing and Synthesis 44
5.2 Fault List Generation and Modifications to ATPG Routines for Gate-Level Replacement Error Models 46
5.2.1 Gate Replacement with Gates of Compatible I/O Count 47
5.2.2 MIGSE Module Replacement 49
5.2.3 Cube Distance Error Replacement 50
5.3 ATPG Experiments 52
5.3.1 ATPG Test Case Execution 52
5.3.2 Benchmark Results 52
5.4 Simulation Experiments 55
5.4.1 Simulation Test Case Execution 55
5.4.2 Simulation Results with Random Vectors 55
5.4.3 Simulation Results with Upper Lattice Layer Vectors 57

6 Implicit fault modeling of FPGA errors **60**
6.1 Previous Work 61
6.1.1 Error Modeling of the Gate 61
6.1.2 Error Modeling of the FPGA Faults 61
6.1.3 Online Testing via Partial Reconfiguration 62
6.2 Cube Distance Replacement Error Modeling 62
6.3 Fault Insertion 63

6.4 Functional Simulation Experiments 64
6.4.1 Random Combinational Circuit Generation 65
6.4.2 Extraction of the LUT Function 65
6.4.3 Generation of LUT Cube-Distance Errors 66
6.4.4 Mapping of Cube-distance Faults to S-A-V Faults 67
6.4.5 Fault Redundancy Identification 67
6.4.6 Simulation Execution Output 68
6.4.7 Functional Simulation Results 70
6.5 FPGA Functional Testing 71
6.5.1 Tool Flow 71
6.5.2 Changing of LUT Equations 71
6.5.3 Test Circuitry and DUT on FPGA 73
6.5.4 Test Vector Transmission from Workstation 80

7 Conclusion **82**
7.1 ATPG and Simulation 83
7.2 FPGA Functional Testing 83
7.3 Summary of Results 84
7.3.1 Cube Distance Model on Designs Mapped to Boolean Gates 84
7.3.2 Cube Distance Model on LUT Mapped Circuits 84
7.3.3 Simulations with Lattice Vectors 84
7.4 Future Work 84
7.4.1 FPGA Testing 84
7.4.2 Algorithm Optimization and Diagnostics 85
7.4.3 Application of the Proposed Method to Other Device Technologies 85

List of Figures

1.1 FPGA design flow . . . 3

2.1 Stuck short faults represented by s-a-v faults . . . 8
2.2 Stuck open faults represented as s-a-0 and s-a-1 faults . . . 9
2.3 Stuck at fault example . . . 9
2.4 MIGSE module of the AND3 gate . . . 12
2.5 MIGSE module of the AND2 gate . . . 13
2.6 MIGSE module of the OR2 gate . . . 13
2.7 MIGSE module of the XOR2 gate . . . 13
2.8 MIGSE module of the OR3 gate . . . 13
2.9 MIGSE module of the XOR3 gate . . . 14
2.10 RTL abstraction of a circuit . . . 15

3.1 Illustration examples of cubes and cube distances . . . 20
3.2 Distance-1 and Distance-2 Error Cubes for 2-Input and 3-Input Gates . . . 21
3.3 AND2 $\rightarrow$ XNOR2 replacement as a c_1 cube error . . . 22
3.4 Gate replacement errors in terms of erroneous cubes . . . 23
3.5 C_1 cubes of AND2 dominating cube of distance-2. Each distance-2 cube is sorted by which the parent c_1 cube is spawned . . . 23
3.6 Circuit with an undetectable cube replacement $f \rightarrow f_c_1$ . . . 26

4.1 4-bit lattice vectors . . . 31
4.2 AND2 Satisfiability . . . 32
4.3 Circuit with s-a-v fault to be solved by SAT . . . 34
4.4 Redundant s-a-v fault . . . 37
4.5 Controllability and observability DC . . . 38
4.6 Redundant cube distance replacement $f \rightarrow f_c1$ . . . 38
4.7 Circuit with observability DC . . . 39
4.8 Cube redundancy check . . . 40

5.1 Functional simulation flow . . . 42
5.2 SAT-based ATPG/ simulation flow for replacement model fault experiments . . . 43
5.3 Decomposing inverted inputs and fan-outs of nodes using *split_neg* . . . 45
5.4 Original ATPG algorithm in SIS . . . 46

5.5 ATPG algorithm for gate replacement errors . . . 49
5.6 ATPG algorithm for MIGSE module replacements . . . 50
5.7 ATPG algorithm for minimal cube distance error replacements . . . 51
5.8 Circuit test case to be executed in customized SIS routines . . . 52
5.9 ATPG execution output by gate replacements . . . 53
5.10 ATPG execution output by MIGSE module replacements . . . 54
5.11 ATPG execution output by cube distance error replacements . . . 55
5.12 Simulation execution output by gate replacements . . . 56
5.13 Simulation execution output by MIGSE module replacements . . . 57
5.14 Simulation execution output by cube distance error replacements . . . 58

6.1 Basic mapped circuit with error in one of its LUT . . . 63
6.2 Simulation flow of LUT verification with cube distance errors . . . 64
6.3 5-LUT decomposition of random combinational network . . . 65
6.4 Generation of 3-LUT Cube Distance-1 Error Functions . . . 66
6.5 Circuit Modified for Redundancy Identification . . . 68
6.6 Random network generation and LUT-5 mapping . . . 68
6.7 Output of LUT simulation by lattice vectors . . . 69
6.8 FPGA design flow . . . 72
6.9 Payload of the 8N1 UART convention . . . 73
6.10 Workstation-FPGA communication via RS-232 for testing . . . 73
6.11 UART module . . . 74
6.12 UART clock divider waveform . . . 75
6.13 UART clock divider state diagram . . . 75
6.14 UART receiver state diagram . . . 76
6.15 UART receiver simulation waveform . . . 77
6.16 UART transmitter . . . 77
6.17 UART transmitter simulation waveform . . . 77
6.18 Simulation controller module . . . 78
6.19 Simulation controller state diagram . . . 78
6.20 Top-level simulation with all component instantiations . . . 79
6.21 Area usage of UART and simulation controller . . . 80

LIST OF TABLES

1.1 Percent of designs failing first silicon 5

2.1 Gate replacement errors with *mux* MCNC benchmark 16
2.2 Fault list size using replacement module errors with *mux* MCNC benchmark 17

3.1 Implicit cube-distance errors for *mux* MCNC benchmark 25
3.2 Erroneous cubes and detection vectors of circuit in Fig. 3.6 27

4.1 CNF SAT expressions of basic 1 and 2-input gates 32

5.1 Output functions of *AND2* and *OR2* 47
5.2 Output functions of *AND2* and its C_1 cubes 50
5.3 ATPG results on MCNC benchmarks 53
5.4 Results of simulation by random vectors 59
5.5 Results of simulation by top lattice layer vectors 59

6.1 FPGA functional simulation results by random vectors 70
6.2 FPGA functional simulation results by lattice vectors 70
6.3 Serial port device mapping in UNIX and Linux 80

CHAPTER 1

INTRODUCTION

Complexity of integrated circuits continue to grow with more assortment of cores to be integrated onto the common substrate. With design engineers continuing to compact more functionality in electronic devices that are to operate at higher speeds, it poses challenges in verification and manufacturing testing, in that more efforts are needed to guarantee proper operation and reliability. With circuits entering submicron fabrication technologies, errors and defects are even more difficult to minimize to meet all functionality requirements. The purpose of verification and testing in the product cycle is to provide confidence that manufactured devices perform under promised specifications, and be able to identify defects and failures should they occur. Unfortunately, a device failing a test may be due to a large range of errors. The problem may originate from the fabrication process, design errors, poor specifications, temperature of the environment, or the test itself. Challenges in testing are even more overwhelming during diagnosis when engineers look for the problem source and find ways of improving the design-manufacturing process.

Failing products cause loss in the customer's confidence and the company's reputation, which may impact the company's selling volume and revenue in the longer run. Yet, in the time-to-market environment, where devices are produced and sold in large volume, engineers require high quality of tests. Factors of test quality are decided by the ability of detecting malfunctioning, the ability of locating the error, the duration of test, and the manufacturing process technology affecting the likelihood of certain types of physical defects to occur. As it is critical to maintain a reasonable cost in design verification and testing efforts, each of the factors need to be regarded with care to deliver reliable products in the shortest amount of time. Typical measurement of test quality is fault coverage, defined as "the ratio of the number of faults that can be detected to the total number of faults in the assumed fault domain" [RT98].

As cited in [Haj03], verification takes up to 70% of the ASIC design cycle's effort and time. Process technology advancements allow large-scale manufacturing of smaller transistors, but may create more chances and classes of failures. It is reported in [Mil04] that over 80% of devices fabricated under the

$0.18\mu m$ technology, (the current mature process technology at the time of writing,) operate as expected. Just over 60% function correctly with the devices fabricated under the $0.13\mu m$ technology, while only 40% work under the $90nm$ technology. Therefore, having verification and test strategies that effectively identifies device failures are critical.

1.1 Design-Manufacturing Flow

Previously, design engineers and chip designers worked separately from test engineers in pre- and post-production parts of the project respectively. As it remained much less affordable in cost and time to resolve errors further down the design-manufacturing cycle, design flows needed to be adjusted by incorporating more interaction in testing and verification of the design and test groups. Not only did this reduce costs and time for the competitive time to market environment, but also accommodated the overly design complexities while making designs easier to test. There is now the popular *design for testability* (DfT) notion which focuses on the incorporation of special features that enables high quality of tests. Typical DfT elements include built-in-self-test (BIST) and scan-based circuits. For BIST, additional circuitry is added into a circuit such that it is capable of checking for unfavorable behavior by itself, independent of expensive tester equipment which may cost thousands of dollars more than the device itself. In scan designs, all (full scan) or selected (partial scan) design registers are modified to mimic pseudo primary I/Os for the purpose of easy testing of internal elements. To goal of BIST is to ease testing efforts without adding significant area overhead and performance degradation to the design.

While *verification* is checking the correctness of a design and is more associated with simulation, hardware emulation for formal methods, *testing* checks the correctness of hardware. The testing process consists of *test generation* during the design cycle by software, and *test application* on the device hardware.

1.1.1 FPGA Emulation

A popular design methodology involves prototyping with a programmable device such as a field-programmable gate array (FPGA). The main advantage is the flexibility, in that incremental mistakes and errors can be rectified quickly by reconfiguration. Without the overhead manufacturing costs during each reconfiguration, the use of FPGA implementations for prototyping therefore greatly reduces design costs. The process, known as FPGA emulation, attempts to emulate the desired behavior of the final implemented ASIC on a FPGA. As shown in Fig. 1.1, the FPGA [Xil] and ASIC design flows remain vastly similar.

The designer begins by writing description of the design in a hardware description language (HDL), which is to be synthesized into actual gates, represented as a netlist. To ensure proper logic mapping, functional simulation is performed on the netlist. The original design and synthesis processing scripts are adjusted

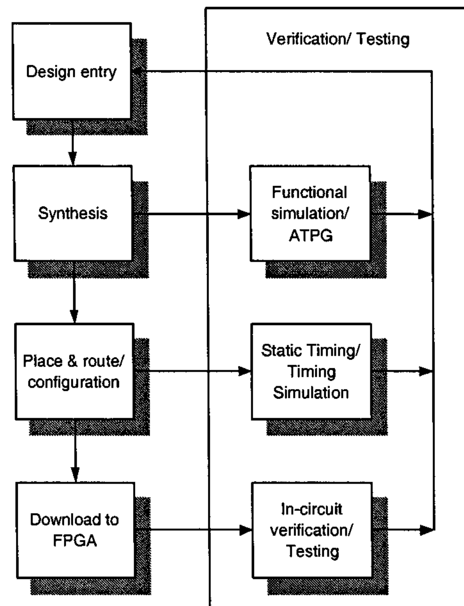


Figure 1.1: FPGA design flow

if necessary. The designer then executes place and route, also known as design layout, where the actual physical placement of logic blocks and their wire interconnections are performed. Knowing the positions of the logic gates, modules and precise wire locations, timing and delay information could be automatically extracted. Verification is then performed with timing information back-annotated into the netlist. When this is completed, the design is ready to be downloaded onto the FPGA. For ASICs the design is sent for tape-out for fabrication. The implemented hardware is then tested for errors in the device. Throughout each stage of the design flow, errors may be introduced as functional errors where the circuit is not operating as its intended Boolean function. They may also be introduced as timing or delays errors. Because of the diversification within each of these categories, the task of testing is highly complex. The scope of the thesis will focus on verification and functional error modeling methodologies but will exclude timing and delay-typed faults.

1.1.2 Verification Techniques

Two main methods of verification are predominant: simulation and formal verification. Simulation is the common classic method of exciting a circuit with series of vectors and monitoring the circuit's output for erroneous behavior. Formal verification uses a different approach by using formal proofs to show that the circuit possesses certain desired properties. Although there have been large advances in formal verification recently, simulation-based methods remains the common technique at the present time and are to be the focal highlight in this thesis.

1.1.3 Simulations

Simulation requires defining an error model and generation of vectors targeting the model. The error model is needed to emulate possible errors that may occur. A set of vectors are then injected into the design under test (DUT) via simulation. The outputs of the DUT are examined for any erroneous response. In many cases, simulation involves storage of large amounts stimulus and response data. This poses a problem especially in BIST where all the test circuitry is along side with the device, and becomes infeasible when the storage elements become larger than the device itself. One solution is response compaction, which takes the set of response data for post-processing and compression [RT98].

1.1.3.1 Fault Modeling

A design with n inputs has a total of 2^n possible input combinations. However, exhaustive simulation would take up far too much time to be practically feasible in any time-to-market environment. As a result, only a subset of test vectors is used, and a fault model is necessary to be defined to represent possible error scenarios the DUT may suffer. Because of the seemingly large classes of errors possible in design and manufacturing, there needs to be a method of classifying them into small, finite manageable classes to make verification and testing easier. First, a fault list is created, which consists of every possible fault defined by the model. After some prepossessing on the fault list to minimize its size, the verification routine traverses each fault of the compact fault list and simulates the circuit. Discussions of fault modeling will be looked at in Chapters 2 and 3.

Redundant faults are ones not detectable by test sets, as a circuit's functional behavior is not affected by their presence regardless of any vector input. Exhaustive simulations can determine whether a fault is redundant, but with only a non-exhaustive subset, other methods must be applied to identify redundant errors. These redundant faults may waste verification times, as applying a subset of test vectors may never truly determine whether they are impossible to detect or simply difficult to identify. Therefore it is important to remove redundancy from the fault list such that time is not wasted during testing. Methods of identifying redundant faults are discussed in Sect. 4.3.

1.1.3.2 Test Vector Generation

In simulation, test vectors may be generated either deterministically or non-deterministically. Deterministic vectors are created by targeting a particular fault of a fault list. Based on the circuit network topology, the software verification algorithm traverses it and attempts to determine at least one vector capable of detecting the fault of interest. Automatic test pattern generation (ATPG) is one of the most popular deterministic methods. An alternative to deterministic methods is by vectors generated randomly or other algorithms such

as lattice layers. Unlike deterministic methods, these vectors are not targeted towards any particular fault of a list. Both deterministic and non-deterministic methods will be discussed in Chapter 4.

1.2 Functional Verification and Testing Relationship

Of all faults, such as logical/ functional, clocking and cross-talk induced errors, logical and functional errors leads the pack where 70% of designs fail first-silicon [Mil04](Table 1.1). Functional components on silicon fail, and the problems are attributed to both design errors that are introduced before design tape-out and manufacturing errors introduced during silicon processes. As it has also been reported in [SF03] that the success rate of first-silicon success rate has dropped from 50% to 39% between 2001 and 2002. The article in [SF03] also suggests that 82% of design re-spins as attributed to design errors, meaning that these hard to detect faults propagated unnoticed all the way into tape-out.

With such large efforts in time spent on finding vectors for verification as well as test vectors for manufacturing faults, it would be wise to combine these two efforts and come up with a test set which can cover both classes of errors in design and manufacturing faults. For that, an abstract functional fault model is needed that could represent the faults in the most compact way.

Table 1.1: Percent of designs failing first silicon

Error type	Percentage (%)
Logic/functional	70
Tuning analog circuit	35
Slow path	26
Fast path	25
Mixed-signal interface	22
Clocking	20
Yield/reliability	16
Crosstalk induced	16
Firmware	14
IR drops	14
Power consumption	12
Other flaw	7
RET	4

1.3 Thesis Overview

The purpose of this thesis is to propose an implicit method of modeling functional errors in terms of cube-distance errors. Then, the proposed fault model is applied to FPGA functional testing. Chapter 2 introduces the stuck-at-value (*s-a-v*) fault modeling techniques in manufacturing testing, as well as the other

methods such as gate replacements and a method proposed by Hayes et al. [AAH95] [BH97], which cover larger classes of design errors than what the *s-a-v* is capable of modeling. The chapter also further discusses shortcomings of modeling these types of errors explicitly, and explains the need of an implicit model that is not only able to cover larger classes of design errors, but is also capable of maintaining a compact fault list. Details of modeling errors implicitly, specifically as cube-distance errors, are proposed in Chapter 3. The implicit model is compared with other previously proposed modeling techniques. Chapter 4 presents simulation with random and lattice vectors and looks at fault detection methods of various fault models by *satisfiability techniques* (SAT) and *automatic test pattern generation* (ATPG). Chapter 4 also describes fault redundancy identification of the cube-distance error model. Chapter 5 details experiments used with the Berkeley SIS [SSL+92] CAD tool, where customized routines are implemented to model implicit gate-level errors in combinational designs. Details of implementation, including a way of mapping the error models to original software routines targeting *s-a-v* faults are also discussed. Finally, Chapter 6 demonstrates modeling of functional errors in *look-up tables* (LUT) and describes experiments used to perform functional FPGA emulation testing based on the proposed model.

CHAPTER 2

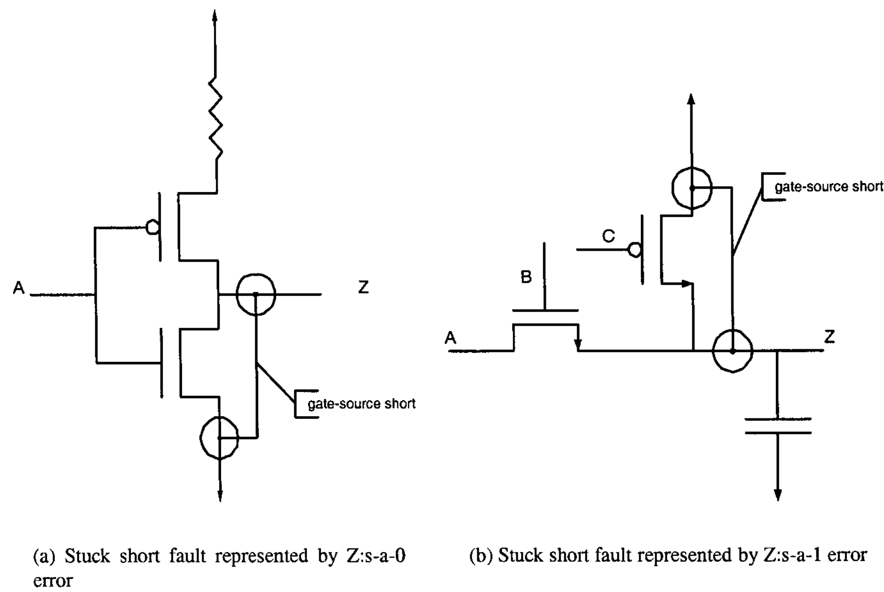
INTRODUCTION TO ERROR MODELING

For any digital system, the circuit may be viewed simply as a mapping of input ports to output ports after some propagation delay. The circuit can be looked at *functionally* through its Boolean mapping, or *behaviorally*, by its functional properties with timing propagating information [ABF94]. The circuit, at any level of abstraction, may be regarded by these mappings. At the gate-level, the functional model of the circuit describes the logic outputs of gates based on the type of the Boolean gates the circuit has. At the register transfer level (RTL), the functional model of the circuit describes the word level logic between registers and larger circuit modules.

The purpose of error modeling is to emulate behaviors posed by different types of physical defects and manufacturing errors as functional errors. Where the categories of errors in physical reality are broad and copious, error modeling attempts to map the majority to a mathematical abstraction. Such a mapping is to achieve a much smaller number of manageable types of functional errors. Like any mathematical model, as is Boolean in this case, error modeling has limitations as to describing all the physical problems affecting device operation. Creating a suitable model is therefore very important. Another challenge is to have a small enough class of errors in the model such that the test time required to cover all of them is minimized. In testing, this entire fault list of the design also has to be kept at a minimum for the test generation software and ATE routines to operate at a smallest memory footprint.

2.1 Error Modeling in Testing: Stuck-At-Value Fault Model

The most popular model currently used in manufacturing testing is the stuck-at-value (*s-a-v*) model, where manufacturing failures are categorized as one of two possible functional errors – wire connections permanently tied to either logical '1' (*s-a-1*) or logical '0' (*s-a-0*). Its obvious attraction is its simplicity in classifying all physical errors into two functional error polarities. Moreover, this type of modeling is very



(a) Stuck short fault represented by Z:s-a-0 error

(b) Stuck short fault represented by Z:s-a-1 error

Figure 2.1: Stuck short faults represented by s-a-v faults

mature with various fast optimized test generation algorithms developed over the years (Chapter 4). Fig. 2.1 shows examples of circuits where stuck-shorts to the power supply and ground could be represented as *s-a-v* faults. In Fig. 2.1(a), node Z is constantly tied to ground, without giving a chance for it to charge even when the input A is '0'. In Fig. 2.1(b), the node Z is stuck-shorted to the power supply, making the node equivalent to *s-a-1*.

Both *s-a-0* and *s-a-1* faults may also cover numerous cases of stuck-open faults. It can be observed in Fig. 2.2 an inverter whose pMOS transistor has its gate-source stuck open. Despite the voltage level at node A, the output Z never has a chance to charge up to logic level '1'. Because of this, this pMOS stuck-open case can be represented by *Z:s-a-0*. Another case is when stuck-open fault is located between the gate and source of the nMOS circuit of the inverter. The output Z never has a chance of discharging, with only a chance to charge when A is 0. The faulty circuit can therefore be represented as *Z:s-a-1*.

2.1.1 Fault Detection

Any method of fault detection requires a difference between the fault-free and faulty outputs of the circuit. Let f be the fault in the circuit. If z and z_f are the fault-free and faulty responses of a circuit respectively, the *Boolean difference* between z and z_f is defined as the functional difference between the two. Formally, the Boolean difference of two functions is written as the exclusive-OR of the two: $z \oplus z_f$. Whenever a fault is

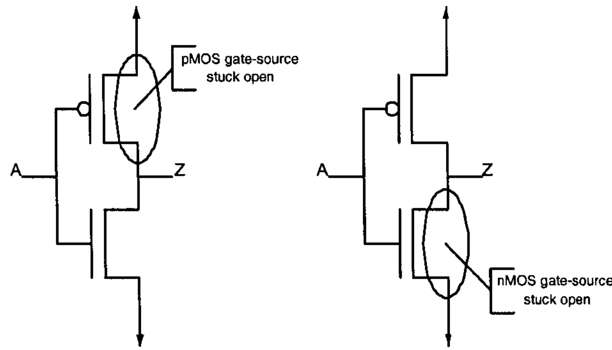


Figure 2.2: Stuck open faults represented as s-a-0 and s-a-1 faults

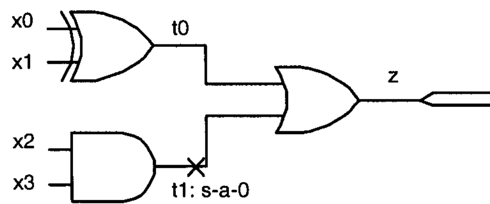


Figure 2.3: Stuck at fault example

detectable, the Boolean difference between the z and z_f needs to be evident:

$$z \oplus z_f = 1 \tag{2.1}$$

Hence, any test vectors capable of satisfying Eqn. 2.1 are ones that may detect the presence of fault z_f.

Example 2.1.1. *Fig. 2.3 shows a combinational circuit with* $z = (x_0 \oplus x_1) + (x_2 x_3)$. *Consider the error* s-a-0 *introduced at node* t_1. *This nodal error is modeled as permanently tied to ground, independent of logical inputs at* x_2 *and* x_3. *Let* z_f *be the erroneous function resulted by introduction of this stuck fault. With* t_1*: s-a-0, the output function has no dependence on the lower branch entering the OR gate. The erroroneous function can thus be described as* $z_f = x_0 \oplus x_1$. *To find the vectors capable of detecting the fault, there needs to be a discrepancy between the fault-free and faulty outputs, by Eqn. 2.1. In this example:*

$$z \oplus z_f = \{(x_0 \oplus x_1) + (x_2 x_3)\} \oplus (x_0 \oplus x_1) = (\overline{x_0 \oplus x_1}) x_2 x_3 = 1 \tag{2.2}$$

Since $(\overline{x_0 \oplus x_1})$, x_2 *and* x_3 *need to be* 1, *the set of vectors capable of detecting* t_1*: s-a-0 is given by:*

$$(x_0, x_1, x_2, x_3) = \{(0, 1, 1, 1), (1, 0, 1, 1)\} \tag{2.3}$$

Various systematic techniques of fault detection exist, such as deterministic and simulation. These methods shall be discussed in Chapter 4.

2.1.2 Limitations of S-A-V Modeling

While the *s-a-v* model attempts to simplify many classes of possible physical faults into two possible functional errors, the obvious limitation is its over generalization of all possible failures into two possible stuck polarities. As discussed recently in [Sal03] [SKG+03] [THML04], while the *s-a-v* fault model may be arguably sufficient in testing of current CMOS devices, the model is not good enough for circuits under sub-micron technology. For nanotechnology-based integrated circuits, generally classified as even smaller than sub-micron technologies, the sources of errors come in an even wider variety. For these reasons, strategies of fault modeling of circuits of these technologies need to be updated.

Although arguably sufficient, many believe that the *s-a-v* model itself fails to cover the entire spectrum errors fully, even in CMOS. Moreover, it also fails to model various types of design errors. For example, it fails to model conditional stuck logical values, nor is it capable of representing gate substitutional errors. An alternative of modeling such errors is by representing each gate replacement errors explicitly.

2.2 Design Functional Error Modeling

Design errors are errors introduced during the design process before layout. With the high dependence on design automation for synthesis, these errors are not only attributed to human errors but also to the Computer-Aided Design (CAD) tools. Although *s-a-v* faults remain appropriate in manufacturing testing this fault model becomes insufficient to represent the variety of design error types.

2.2.1 Gate Replacement Error Modeling with Gates of Compatible I/O Count

Gate replacements are common design errors. Each gate in the netlist may have been erroneously replaced by another gate with a different function but with the same number of inputs and outputs from its design libraries. For example, let the circuit of Fig. 2.3 be synthesized with design libraries consisting of the six standard 2-input gates (AND2, OR2, XOR2, NAND2, NOR2, XNOR2) and six 3-input gates (AND3, OR3, XOR3, NAND3, NOR3, XNOR3). Errors on the XOR2 gate in the original circuit may be modeled as a replacement with either one of the five 2-input AND2, OR2, NAND2, NOR2 or XNOR2 gates. Likewise, errors on the AND2 gate may be represented by replacements with OR2, XOR2, NAND2, NOR2, XNOR2 gates. Finally, the OR2 gate can be modeled as replacements with AND2, XOR2, NAND2, NOR2, XNOR2 gates. These errors can commonly occur by human intervention with post-synthesis netlists,

for improvements such as critical path delays (by replacing existing gates with faster logic) or area/power minimization.

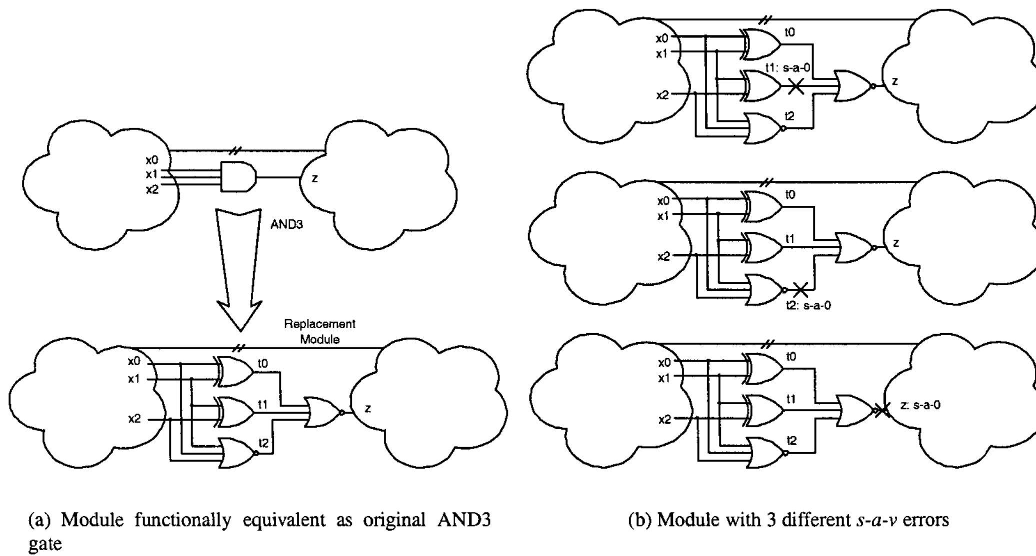
2.2.2 Gate Replacement Errors by MIGSE Modeling Modules with S-A-V Faults

Although the basic concept of gate replacement errors is straightforward, modeling of such errors for the purpose of their detection can pose a significant problem. The vast variety of possible replacement cases to be considered makes the simulations prohibitively expensive. Several alternatives have been proposed to overcome this difficulty. For example, in [AAH95][BH97] Hayes et al. applied *s-a-v* faults to represent explicitly erroneous gates, specifically multiple input gate substitution errors (MIGSE). Instead of having a set of gates from the synthesis library as candidate replacements, the model uses only one replacement module for each type of gate in the library with *s-a-v* faults placed internally within each module to represent errors. One of the main advantages of this approach is being able to use existing *s-a-v* testing routines (such as ATPG) for error detection of gate replacement faults.

2.2.2.1 Construction of the MIGSE Fault Replacement Module

A MIGSE module is first constructed such that its Boolean logic is functionally equivalent to the one of the original gate. Next, *s-a-v* faults are strategically placed within the module to represent the several possible error types of the gate. These stuck faults are placed such that the complete set of vectors, classified as V_{null}, V_{all}, V_{odd} and V_{even} are capable of detecting all these stuck faults. The four classes of "V" vectors are defined as input combinations at the input of the gate. Specifically, each category describes whether the input combinations are all zeros, ones, odd parity, or even parity:

- V_{null}: All '0's in input vector
- V_{all}: All '1's in input vector
- V_{odd}: Odd parity input vector that is neither V_{null} nor V_{all}
- V_{even}: Even parity input vector that is neither V_{null} nor V_{all}



(a) Module functionally equivalent as original AND3 gate

(b) Module with 3 different *s-a-v* errors

	Inputs			Internal			Fault-Free	Faulty Responses		
AND3	x0	x1	x2	t0	t1	t2	z	z(t1:s-a-0)	z(t2:s-a-0)	z(z:s-a-0)
X Vnull	0	0	0	0	0	1	0	0	1	0
X Vodd	0	0	1	0	1	0	0	1	0	0
Vodd	0	1	0	1	1	0	0	0	0	0
Veven	0	1	1	1	0	0	0	0	0	0
Vodd	1	0	0	1	0	0	0	0	0	0
Veven	1	0	1	1	1	0	0	0	0	0
X Veven	1	1	0	0	1	0	0	1	0	0
X Vall	1	1	1	0	0	0	1	1	1	0

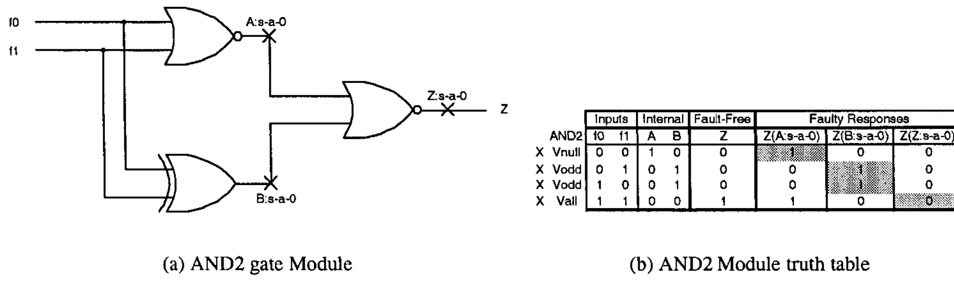
Erroneous output (shaded)
X Vector is capable of detecting fault

(c) Logic table of AND3 gate replacement module

Figure 2.4: MIGSE module of the AND3 gate

Example 2.2.1. *Fig. 2.4(a) shows an example of a 3-input AND gate ($AND3$) within an arbitrary logic network. A construction of the replacement module is made by interconnections of two XOR2 gates and three NOR3 gates such that the overall module's logic represents the same logic original AND3 function, i.e.,* $x_0x_1x_2 = \overline{(x_0 \oplus x_1) + (x_1 \oplus x_2) + \overline{(x_0 + x_1 + x_2)}}$. *Next, three* s-a-0 *faults are placed at* t_1, t_2 *and* z *in Fig. 2.4(b). Fig. 2.4(c) lists all Boolean values at each internal node for each* s-a-v *cases, as well as the vector type capable of detecting the fault.* $z_{t_1:s-a-0}$ *can be detected by inputs* $(x_0, x_1, x_2) = (0, 0, 1)$ *and* $(1, 1, 0)$, *which are* V_{odd} *and* V_{even} *vectors respectively.* $z_{t_2:s-a-0}$ *can be detected by* $(0, 0, 0)$, *the* V_{null} *vector.* $z_{z:s-a-0}$ *can be detected by* $(1, 1, 1)$, *the* V_{all} *vector. Each of the four categories of vectors is required to detect the three injected* s-a-v *faults.*

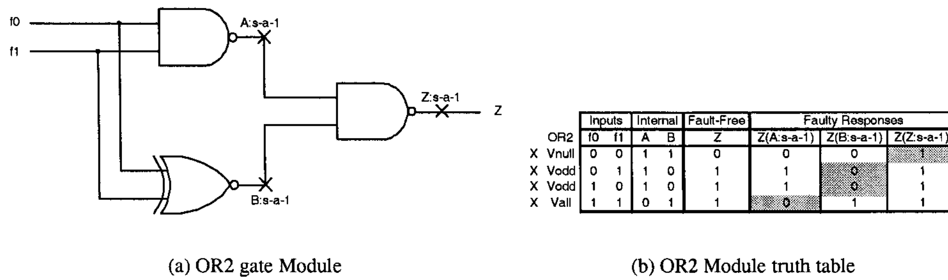
The modules for AND2 (Fig. 2.5), OR2 (Fig. 2.6), XOR2 (Fig. 2.7), OR3 (Fig. 2.8), XOR3 (Fig. 2.9) are all created in the similar fashion as described in Ex. 2.2.1. All marked stuck fault locations are candidate fault locations within the module.



	Inputs		Internal		Fault-Free	Faulty Responses		
AND2	f0	f1	A	B	Z	Z(A:s-a-0)	Z(B:s-a-0)	Z(Z:s-a-0)
X Vnull	0	0	1	0	0	1	0	0
X Vodd	0	1	0	1	0	0	1	0
X Vodd	1	0	0	1	0	0	1	0
X Vall	1	1	0	0	1	1	0	0

(a) AND2 gate Module (b) AND2 Module truth table

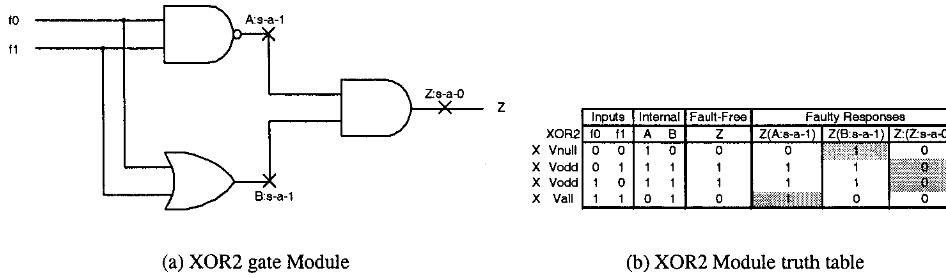
Figure 2.5: MIGSE module of the AND2 gate



	Inputs		Internal		Fault-Free	Faulty Responses		
OR2	f0	f1	A	B	Z	Z(A:s-a-1)	Z(B:s-a-1)	Z(Z:s-a-1)
X Vnull	0	0	1	1	0	0	0	1
X Vodd	0	1	1	0	1	1	0	1
X Vodd	1	0	1	0	1	1	0	1
X Vall	1	1	0	1	1	0	1	1

(a) OR2 gate Module (b) OR2 Module truth table

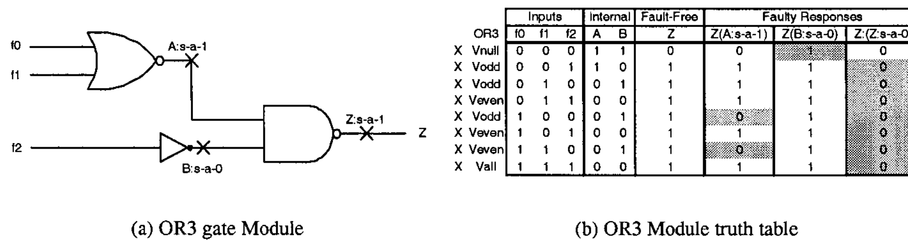
Figure 2.6: MIGSE module of the OR2 gate



	Inputs		Internal		Fault-Free	Faulty Responses		
XOR2	f0	f1	A	B	Z	Z(A:s-a-1)	Z(B:s-a-1)	Z:(Z:s-a-0)
X Vnull	0	0	1	0	0	0	1	0
X Vodd	0	1	1	1	1	1	1	0
X Vodd	1	0	1	1	1	1	1	0
X Vall	1	1	0	1	0	1	0	0

(a) XOR2 gate Module (b) XOR2 Module truth table

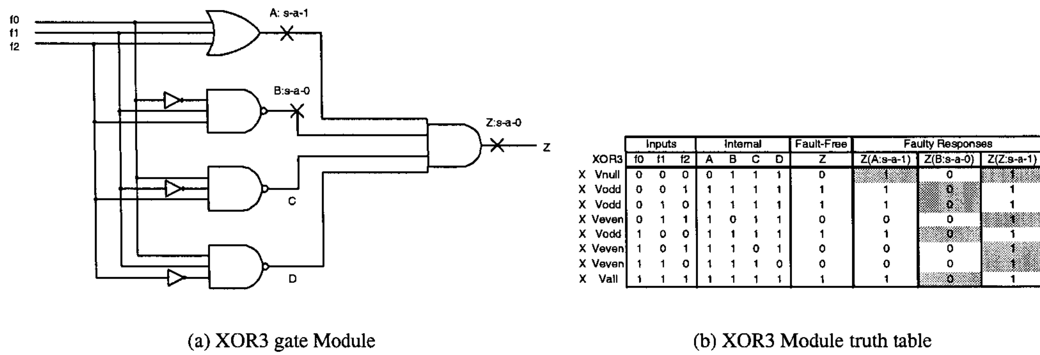
Figure 2.7: MIGSE module of the XOR2 gate



	Inputs			Internal		Fault-Free	Faulty Responses		
OR3	f0	f1	f2	A	B	Z	Z(A:s-a-1)	Z(B:s-a-0)	Z:(Z:s-a-0)
X Vnull	0	0	0	1	1	0	0	1	0
X Vodd	0	0	1	1	0	1	1	1	0
X Vodd	0	1	0	0	1	1	1	1	0
X Veven	0	1	1	0	0	1	1	1	0
X Vodd	1	0	0	0	1	1	0	1	0
X Veven	1	0	1	0	0	1	1	1	0
X Veven	1	1	0	0	1	1	0	1	0
X Vall	1	1	1	0	0	1	1	1	0

(a) OR3 gate Module (b) OR3 Module truth table

Figure 2.8: MIGSE module of the OR3 gate



		Inputs			Internal				Fault-Free	Faulty Responses		
	XOR3	f0	f1	f2	A	B	C	D	Z	Z(A:s-a-1)	Z(B:s-a-0)	Z(Z:s-a-1)
X	Vnull	0	0	0	0	1	1	1	0	1	0	1
X	Vodd	0	0	1	1	1	1	1	1	1	0	1
X	Vodd	0	1	0	1	1	1	1	1	1	0	1
X	Veven	0	1	1	1	0	1	1	0	0	0	1
X	Vodd	1	0	0	1	1	1	1	1	1	0	1
X	Veven	1	0	1	1	1	0	1	0	0	0	1
X	Veven	1	1	0	1	1	1	0	0	0	0	1
X	Vall	1	1	1	1	1	1	1	1	1	0	1

(a) XOR3 gate Module

(b) XOR3 Module truth table

Figure 2.9: MIGSE module of the XOR3 gate

2.2.2.2 Detection of Faults Represented by Module Replacements

With module replacement modeling of gate errors, each faulty gate is represented by a small cluster of interconnected gates with *s-a-v* faults within the module. Each of the *s-a-v* faults is executed with test generation routines to determine whether the errors within the module could be detected as being part of the entire circuit under test. More detailed description of errors by module replacement can be seen in Chapter 5.

2.2.2.3 Limitations of Module Replacement Modeling

For each gate used in the library, a module needs to be created to model its replacement. There may be difficulties in finding descriptions of complex combinational gates – gates that are not part of the basic $\{AND, OR, XOR, NAND, NOR, XNOR\}$ categories. The problem would be most evident in FPGAs when the fundamental logic block is no longer the gate but the look-up table (LUT). Since LUT implements a variety of Boolean logic, there would be an unrealistic number of modules to create. Therefore, the module replacement model would not be suitable for functional verification and testing of FPGAs (Chapter 6).

2.2.3 Other Fault Modeling Methods

2.2.3.1 Bridging Faults

With devices approaching sub-micron technologies, closely placed wires are more susceptible to crosstalk effects. The bridging fault model, discussed in textbooks such as [BA00] [ABF94] has been used to check errors on the Pentium^{TM}4 processor [KMV01].

2.2.3.2 Error Modeling at RTL Abstractions

Obviously, fault models are not limited to the gate-level. Other methods have been proposed in [TAZ00] [TAZ03] which abstracts faults at the register transfer level (RTL). Faults are injected at this level, specifically,

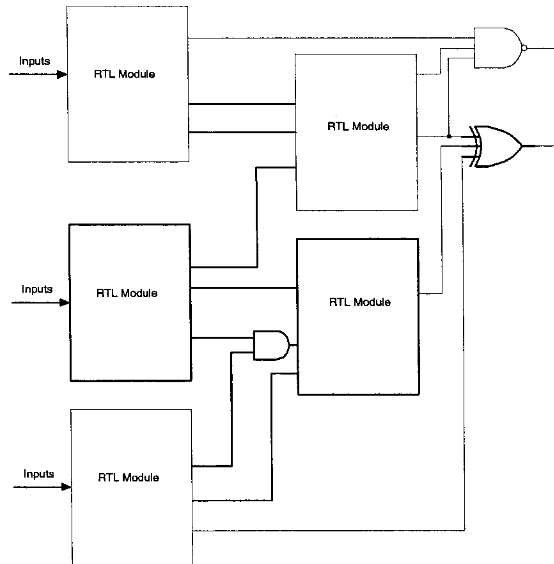


Figure 2.10: RTL abstraction of a circuit

before synthesizing constructs at the RTL down into gates. Therefore, at the time of fault insertion, there is no knowledge of the structural information of the circuit except the interconnections at the RT-level.

RTL modules created with detection probabilities similar to a collapsed gate fault list. Detection probability is defined as the probability of detecting a fault by a random vector. For a complete test set that has n test patterns that is capable of detecting the fault k times, the detection probability is defined as k/n.

Error modules at the RTL level have a fault list constructed in a similar way gate fault lists are constructed. In *s-a-v* faults, wires are assumed to have constant error logical values at either '0' or '1', while gates are assumed error-free. The RTL method proposed by the authors of [TAZ00] [TAZ03] uses the same model assumptions with the added property that the RTL modules are also assumed to be fault-free. At this level, *s-a-v* faults are injected at the inputs and fan-outs, but not at the internal nodes as their information is not available yet. For RTL constructs that are defined as Boolean operators, *s-a-v* faults are also injected within the internal nodes as well, as the internal signals of these components are also available at the RT-level. Fig. 2.10 shows an example of the design at the RTL abstraction. In most cases, the blocks represent functions that are not mapped to logic gates yet. Stuck faults are injected at the wires surrounding the RTL block. In other instances, logic information is available and faults could be injected into the inputs and outputs of these.

A technique known as stratified fault sampling is issued to create RTL modules for simulation. These RTL modules are representative samples of the gate-level fault list, and the coverage results by simulations of these modules are very close to what would be produced at the gate-level. The overall design fault coverage is determined by a weighted sum of each individual RTL fault modules. The authors in [GSTT01] have also

Table 2.1: Gate replacement errors with *mux* MCNC benchmark

MCNC benchmark:	*mux*
2-Input Gates in synthesis library:	AND2, OR2, XOR2, NAND2, NOR2, XNOR2
3-Input Gates in synthesis library:	AND3, OR3, XOR3, NAND3, NOR3, XNOR3
Number of gates in synthesis library:	6 *(2-input)* +6 *(3-input)* = 12
Number of gates in circuit:	18 *(2-input)* +58 *(3-input)* = 76
Possible gate replacement errors *(per gate)*:	$6-1=5$
Total gate replacement errors:	$5(76)=380$

attempted to improve the statistic measurement of these RTL modeling techniques.

2.3 Explicit and Implicit Error Modeling

2.3.1 Explicit Error Modeling

Definition 2.3.1. *Gate error replacement is replacement of a correct gate with a gate of different Boolean function, but with compatible number of I/O ports.*

Explicit design error modeling is widely used, due to its intuitiveness. One of the most popular examples of explicit errors in manufacturing testing is a single stuck-at value (s-a-v) model representing defects in transistors implementing digital gates. The fault list size is made up of all possible faults given by the model at each wire. The list may be minimized only to some extent by fault collapsing and dominance properties [BA00][ABF94]. The strong point of s-a-v model is its ability to represent various manufacturing defects in terms of an internal circuit connection stuck either to Vss (s-a-1) or ground (s-a-0). The analogue explicit simple and uniform model is impossible to derive in the case of functional faults, where the diversity of the design errors is hard to capture.

Due to the variety of types of functional faults, the straightforward explicit modeling would therefore not be appropriate for representing errors in even moderate netlists. The appealing solution is to classify several explicit faults by merely one implicit fault, such that its detection would guarantee detectability of all explicit faults it represents.

2.3.2 Fault List Sizes Based on Explicit Error Modeling

2.3.2.1 Gate Replacement Error Modeling with Gates of Compatible I/O Count

Consider using the *mux* MCNC benchmark circuit as an example (Table 2.1), where the design is synthesized and decomposed into gates of 2 and 3-inputs. 6 gates of each 2-input and 3-input gates from

Table 2.2: Fault list size using replacement module errors with *mux* MCNC benchmark

MCNC benchmark:	*mux*
2-Input gates in Synthesis Library:	AND2, OR2, XOR2, NAND2, NOR2, XNOR2
3-Input gates in Synthesis Library:	AND3, OR3, XOR3, NAND3, NOR3, XNOR3
Number of *s-a-v* faults in each replacement module:	3
Total number of gates:	76
Total *s-a-v* in replacement module errors:	$76 \times 3 = 228$

the synthesis libraries are as shown in the table, including functions of {AND, OR, XOR, NAND, NOR, XNOR}, making the total number of gates in the synthesis library twelve. After decomposition, the netlist consists of 18 and 58 2-input and 3-input gates respectively, for a total number of 76 gates. With 6 gates for each 2-input and 3-input gate classes, the number of possible replacements for a gate in any class is 5, because it is redundant to check for a gate-replacement by a gate with an identical Boolean function. Since replacements are only performed on gates with compatible number of inputs and outputs, the total possible number of gate replacements is 5 times the number of gates, i.e., 380. In general, a netlist with M gates synthesized to the N-gate library, (assuming each the same number of $j-$input gates in the library,) has the total number of explicit gate replacement errors equate to $M \times (N - 1)$. If the same gate libraries are used as the one in the example, then the fault list size is $5M$.

2.3.2.2 Gate Replacement Errors by MIGSE Replacement Module

Depending on the way the replacement modules are constructed for each corresponding gate used in the design libraries, the number of *s-a-v* faults may vary. However, all standard gates of 2- and 3- inputs previously derived in Sect. 2.2.2.1 have been derived to contain three *s-a-v* faults consistently.

Each module of negative polarity gate types, NAND, NOR and XNOR can be derived from the positive polarity counterpart – AND, OR, and XOR, respectively. The modules are merely an added inverter at the output, so the number of stuck faults in a module are made to be the same as one created for its opposite polarity. Similar to the gate replacement example, the library of gates are assumed to be two and three-input gates.

With 3 *s-a-v* faults in each replacement module, the *mux* MCNC benchmark, which has a total of 76 gates, the fault list size (Table 2.2) is 228. In general, a netlist with M gates, with N- *s-a-v* faults in each gates' replacement module, the total number of explicit gate replacement errors equate to $M \times (N - 1)$. If the same replacement modules are used as the ones in the example, then the fault list size is $3M$.

2.3.3 Motivation of Implicit Error Modeling of Design Faults

As shown in Table 2.1 for gate replacements and Table 2.2 for module replacements with s-a-v faults, the fault list sizes are large. These types of explicit fault modeling would therefore not be appropriate, because with the variety of possible errors, it becomes even more difficult for the fault list to remain compact in size. The most appealing solution to this problem would be to classify several explicit faults by merely one implicit fault, such that detectability of such implicit fault would guarantee detectability of all explicit faults it is representing. The fault list consisting on implicit faults would be much smaller in size. With this in mind, each implicit fault can be used to describe and cover a broader range of errors, with fewer tolls on the fault list size that what would have been with explicit modeling.

CHAPTER 3

IMPLICIT ERROR MODELING METHOD BY CUBE ERROR DISTANCES

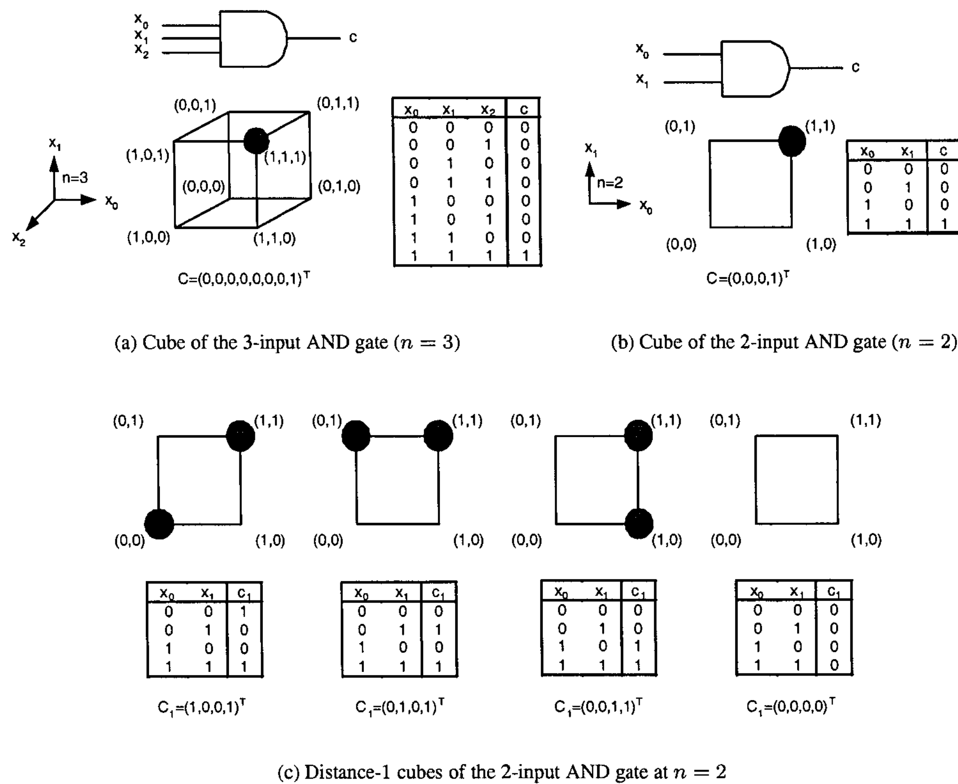
With gate-level functional errors such as gate replacement errors, fault list sizes by these error modeling methods result in fault lists that are large and more difficult to manage than the traditional *s-a-v* faults. The issue is even more serious in error modeling of lookup tables (LUT) in FPGA since LUT represents larger classes of Boolean logic as opposed to the standard Boolean gates in ASICs. This chapter proposes a method of modeling errors implicitly as opposed to the explicit methods previously shown in Chapter 2, with the purpose of not only shrinking the fault list sizes smaller to be more manageable, but also be able to cover more functional errors than other gate-level modeling techniques. The proposed model is to be capable of modeling all types of functional errors at the gate-level, as well as the basic LUT level in FPGAs.

3.1 Implicit Error Modeling of Gate-Level Errors

Definition 3.1.1. *A cube (Fig. 3.1(a)) is a single product term in a sum-of-products representation, for which a Boolean function is true.*

Definition 3.1.2. *The cube-distance (Hamming distance) of two functions is the number of cubes that differ between them.*

Definition 3.1.3. *An error function representing a cube-distance error i is one that differs in i cubes from the original fault-free function. If c denotes the original function, then c_i represents a function that has a cube-distance error of i.*



(a) Cube of the 3-input AND gate ($n = 3$)

(b) Cube of the 2-input AND gate ($n = 2$)

(c) Distance-1 cubes of the 2-input AND gate at $n = 2$

Figure 3.1: Illustration examples of cubes and cube distances

In a truth table, function c would be the vector making up the "output column", whose cube is represented by sole product term when all inputs are 1. For example, function c of a 2-input AND gate from Fig. 3.1(b) is represented by $(0,0,0,1)^T$ whose entries correspond to input combinations $(0,0),(0,1),(1,0),(1,1)$ respectively. The final entry in c, corresponding to input combination $(1,1)$ implies the cube x_0x_1.

From Fig. 3.1(c), a possible cube distance-1 function c_1 would be $(1,0,0,1)^T$, where the difference in the output corresponds to input combination $(0,0)$. Another possible c_1 function is $(0,1,0,1)^T$, where the output difference corresponds to input combination $(0,1)$.

Fig. 3.2 shows a comprehensive list of cube errors of distances one and two for 2-input and 3-input AND, OR and XOR gates. The shaded areas mark location(s) of error cubes in the function. Each erroneous function (column) of cube-distance i would have i shaded location(s). The class of distance-1 cubes (C_1) implicitly represents all possible c_1 replacements. C_1 correspond to all functional errors, which manifest themselves as cube-distance-1. Similarly, C_2 implicitly represent all possible c_2 replacements, i.e., all functional errors classified as cube-distance-2 errors.

AND2			C1				C2					
0	0	0	1	0	0	0	0	0	1	0	1	1
0	1	0	0	1	0	0	0	1	0	1	0	1
1	0	0	0	0	1	0	1	0	0	1	1	0
1	1	1	1	1	1	0	0	0	0	1	1	1
OR2												
0	0	0	1	0	0	0	0	0	1	0	1	1
0	1	1	1	0	1	1	1	0	1	0	1	0
1	0	1	1	1	0	1	0	1	1	0	0	1
1	1	1	1	1	1	0	0	0	0	1	1	1
XOR2												
0	0	0	1	0	0	0	0	0	1	0	1	1
0	1	1	1	0	1	1	1	0	1	0	1	0
1	0	1	1	1	0	1	0	1	1	0	0	1
1	1	0	0	0	0	1	1	1	1	0	0	0

AND3				C1								C2																											
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1
0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	1	0	1
0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	0	1	1	0
0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	1	1	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
OR3																																							
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1
0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	0	1	0
0	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0	0	1
0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	1	0	0	0	1	1	1
1	0	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
XOR3																																							
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1
0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	0	1	0
0	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1	0	1	1	0	0	1
0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	1	1	0	0	0
1	0	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	0	0	0	1	1	1	1	1	1
1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.2: Distance-1 and Distance-2 Error Cubes for 2-Input and 3-Input Gates

3.1.1 Explicit Errors as Functional Cube-Distances

Faults modeled as cube-distances are suitable to describe explicit erroneous gate replacements (Ex. 3.1.1).

Example 3.1.1. *Consider Fig. 3.3 where the original 2-input AND gate (AND2) is replaced by a 2-input XNOR (XNOR2) gate. Both gate functions are identical except for the entry corresponding to inputs* $(0, 0)$*. Due to this single difference, the AND2* $\rightarrow$ *XNOR2 gate replacement is functionally equivalent to a* c_1 *(cube-distance-1) replacement.*

Fig. 3.4 shows the truth table of the original AND2 gate and all its functional replacements sorted by the error cube-distances. The first table in the figure illustrates the complete set of sixteen two-input Boolean functions, with the AND, OR, XOR, NAND, NOR and XNOR entries highlighted. The second table shows each of the four possible cube-distance-1 c_1 errors with respect to AND2 (a correct gate). Note that the XNOR2 replacement discussed in Ex. 3.1.1 belongs to this category. Both AND2→OR2 and AND2→NOR2 replacements are cube-distance-2 c_2 errors, while AND2→XOR2 is a c_3 error. Similarly, AND2→NAND2

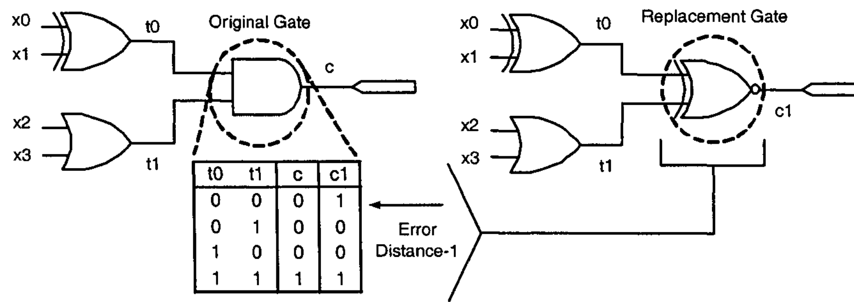


Figure 3.3: AND2 $\rightarrow$ XNOR2 replacement as a c_1 cube error

is a sole c_4 replacement. Additionally to gate replacements, cube-distance errors can model *s-a-v* faults. In Fig. 3.4, a c_1 represents a *s-a-0* at the output of AND2 ($(0,0,0,0)^T$ entry of the second table), while a c_3 describes *s-a-1* at the output of AND2 ($(1,1,1,1)^T$ entry of the fourth table). As the figure shows, the maximum possible cube-distance for the 2-input gate is 4, when all 2^2 possible outputs of the cube are faulty. Likewise, the maximum possible cube-distance for a 3-input gate would be $2^3 = 8$.

Lemma 1. *Every gate replacement fault can be represented as a cube-distance fault.*

Proof. Every gate replacement fault result in a change in the gate's original function. The ON-set of their difference constitutes a cube-distance error, by Def. 3.1.2 and 3.1.3. □

It is shown how implicit error modeling by cube distances can be used to represent gate replacement errors. This model is even more powerful when considering design and mapping errors in FPGAs. Not only are the typical gate replacement errors in ASIC reduced to Boolean functional errors, but the same simplification can be applied to the Boolean implementations in each LUT. The notion of functional faults is even stronger in this case than in dedicated ASIC designs.

3.1.2 Cube-Distance Fault List Reduction Using Dominance

Every n-input module can be wrongly represented in $2^{2^n} - 1$ ways. In consequence, the fault list comprised of all possible errors attributed to all modules in a given design can be prohibitively large. However the characteristics of cube-distance error modeling allow a simple but significant fault list reduction based on the fault dominance principle.

Lemma 2. *Detectability of all error cube replacement functions at a distance i implies detectability of all cube-distance errors greater than i.*

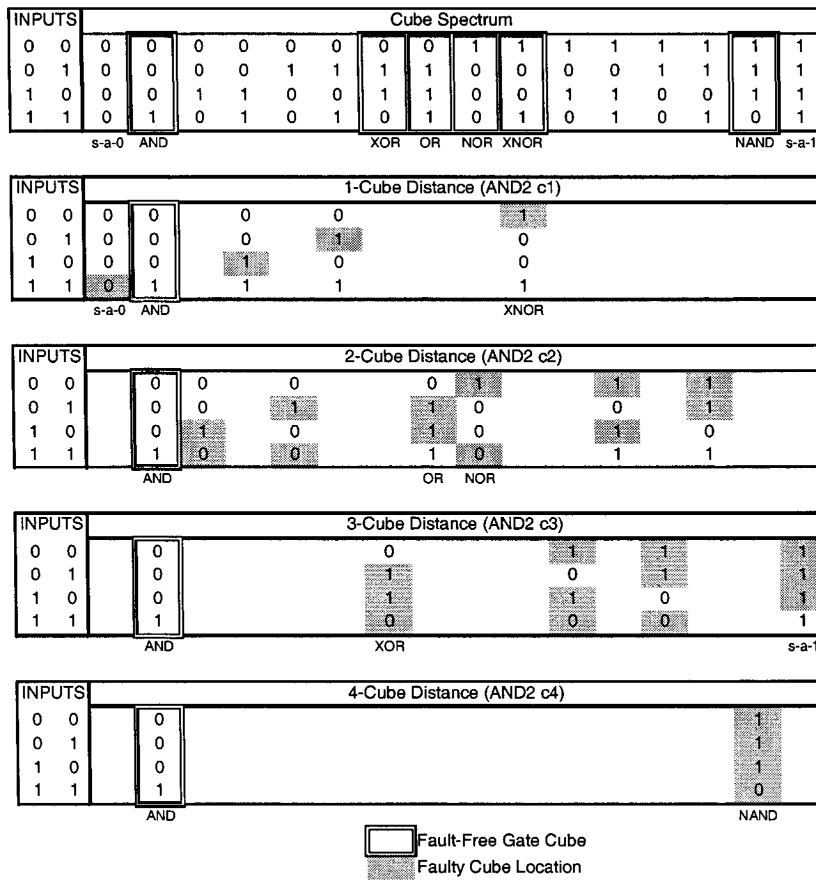


Figure 3.4: Gate replacement errors in terms of erroneous cubes

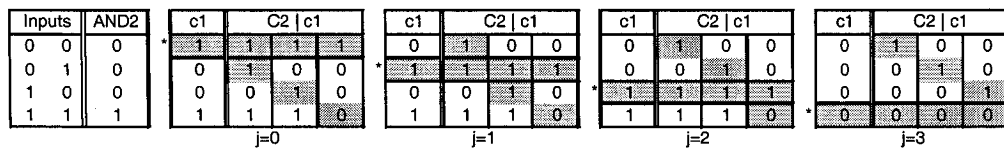


Figure 3.5: C_1 cubes of AND2 dominating cube of distance-2. Each distance-2 cube is sorted by which the parent c_1 cube is spawned

Proof. Let $[c_i]_j$ be a cube that differs from the original gate by distance-i. Let C_i represent the set of all cubes that is distance-i from original function. i.e.,

$$C_i = \bigcup_{\forall j} \{[c_i]_j\}; \quad 0 \leq j \leq \binom{2^n}{i} - 1. \tag{3.1}$$

Using the AND2 gate as an example (Fig. 3.5), C_1 comprises of $(1,0,0,1)^T$, $(0,1,0,1)^T$, $(0,0,1,1)^T$ and $(0,0,0,0)^T$. By definition, C_{i+1} represents the complete set of cubes that possess distance-$(i+1)$ from the original function. To further characterize the cubes, allow $[C_{i+1}|c_i]_j$ to be a subset of C_{i+1} whose cubes also possess a distance of one from a $[c_i]_j$ cube. Then, every cube in $[C_{i+1}|c_i]_j$ is spawned from cube $[c_i]_j$ by one cover difference. Fig. 3.5 shows each C_2 cubes categorized by the four possible $[c_i]_j$ cubes as $[C_{i+1}|c_i]_j$ sets $(0 \leq j \leq 3)$. Using $[c_1]_0 = (1,0,0,1)^T$ as an example, $[C_2 \mid c_1]_0 = \{(1,1,0,1)^T, (1,0,1,1)^T, (1,0,0,0)^T\}$.

In general, each $[C_{i+1}|c_i]_j$ sets may not necessarily be disjoint, but their union equates to the entire C_{i+1}. Specifically:

$$C_{i+1} = \bigcup_{\forall j} [C_{i+1}|c_i]_j; \quad \forall j : 0 \leq j \leq \binom{2^n}{i} - 1 \tag{3.2}$$

Let $T(.)$ represent a test set of an error cube (or a set or error cubes). Using the same example for $j=0$, $T([c_1]_0) = \{(0,0)\}$. The same test set also detects every cube in $[C_2 \mid c_1]_0$. This is true since all of the higher distanced cubes may have its errors triggered by the same inputs. In general, all cubes in $[C_{i+1}|c_i]_j$ cube spawned by $[c_i]_j$ may be detected by $T([c_i]_j)$.

For any set of cubes spawned by cube $[c_i]_j$:

$$T(\{[c_i]_j\}) \subseteq T([C_{i+1}|c_i]_j) \tag{3.3}$$

Taking the union of all $[c_i]_j$ cubes and their spawned cubes:

$$T(\bigcup_{\forall j} \{[c_i]_j\}) \subseteq T(\bigcup_{\forall j} [C_{i+1}|c_i]_j) \tag{3.4}$$

By Eqn. 3.1 and 3.2:

$$T(C_i) \subseteq T(C_{i+1}) \tag{3.5}$$

By induction, up to a maximum distance of 2^n:

$$T(C_i) \subseteq T(C_{i+1}) \subseteq T(C_{i+2}) \subseteq \cdots \subseteq T(C_{2^n}) \tag{3.6}$$

Table 3.1: Implicit cube-distance errors for *mux* MCNC benchmark

MCNC benchmark:	*mux*
Number of gates in circuit:	18 *(2-input)* +58 *(3-input)* = 76
Total implicit cube distance errors with C_1:	76
Total explicit gate replacement errors:	$5(76) = 380$

□

Qualitatively, the higher-distanced cube errors always dominate the lower-distanced ones (Eqn. 3.6). Therefore, detectability of any lower-distanced directly implies detectability of their higher-distanced counterpart.

Lemma 3. *All explicit gate replacement errors are detectable if their implicit cube-distance-1 faults are detectable.*

Proof. According to Lemma 1, every explicit gate replacement error can be represented as an implicit cube-distance fault, C_i. Then, by dominance principle of cube-distance errors (Lemma 2), if a test set for C_1 error functions is sufficient to detect all error functions of cube distance greater than 1, there is no need to verify errors with larger distances.

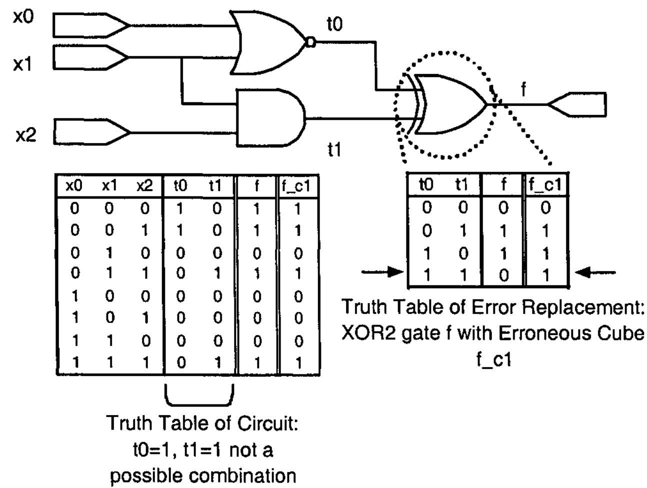
□

Note, that the if some C_1 implicit faults, $[c_1]_i$ are not detectable (redundant), then it does not imply that higher order cube-distance faults spawning from $[c_1]_i$ can also be declared as redundant.

The fault list size for implicit error modeling is calculated in Table 3.1. By lemma 3, the class of implicit C_1 cubes is sufficient in representing all explicit gate replacement errors. Comparing the fault list size of explicit gate replacement modeling of 380 ($5M$), the size based on C_1 is 76 (M), equaling to the number of gates in the netlist.

3.1.3 Verification with Minimal Cube-Distance Errors

Ideally, the best case scenario is the detectability of all C_1 errors. Recalling from Lemma 3, when all of C_1 are detectable, there is no need to proceed with any errors with larger distances, since their detectability is implied. Hence, the minimum number of required errors to be checked equates to 2^n. However, it is highly possible and frequent to have replacement errors at distance-1 undetectable.



x0	x1	x2	t0	t1	f	f_c1
0	0	0	1	0	1	1
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	1	0	1	1	1
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	1	1	1

t0	t1	f	f_c1
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

Figure 3.6: Circuit with an undetectable cube replacement $f \to f_c_1$

Example 3.1.2. *Consider a circuit in Fig. 3.6. The XOR2 gate driving output f is replaced with its cube-distance-1 error, c_1: $f_c_1 = (0,1,1,1)^T$, which is excited if its inputs $(t0, t1)$ can be set to $(1,1)$. However, as the truth table of the same figure indicates, primary inputs x_0, x_1 and x_2 are incapable of setting both of these intermediate nodes to 1 at the same time.*

In this case, only based on the fact that a c_i of C_i is undetected, it is impossible to conclude whether the higher distance cube errors would be also undetected. Consequently, all error functions at cube distance $i+1$, C_{i+1} must therefore be checked. In this example, since not all of the C_1 are detectable, C_2 errors (next higher in distance) are investigated. The set of considered errors includes in particular $[C_2|f_c_1] : \{(1,1,1,1)^T, (0,0,1,1)^T, (0,1,0,1)^T\}$. Each of these errors can be detected by indirectly setting $(t0, t1)$ to $(0,0)$, $(0,1)$ and $(1,0)$ respectively via x_0, x_1 and x_2.

3.2 Fault List Sizes with Explicit Error Modeling

3.2.1 Explicit Gate Replacements

The circuit in Fig. 3.6, can be verified by the following example (Ex. 3.2.1):

Table 3.2: Erroneous cubes and detection vectors of circuit in Fig. 3.6

Gate	C_1	Vectors	C_2	Vectors
XOR2	$(1,1,1,0)^T$	$(0,1,0)$	$(1,0,1,0)^T$	$(1,1,1)$
	$(0,0,1,0)^T$	$(0,0,0)$	$(1,1,0,0)^T$	$(0,0,0)$
	$(0,1,0,1)^T$	$(1,1,1)$	$(1,1,1,1)^T$	$(0,1,0)$
	$(0,1,1,1)^T$	—	$(0,0,0,0)^T$	$(0,0,0)$
			$(0,0,1,1)^T$	$(0,1,0)$
			$(0,1,0,1)^T$	$(0,0,0)$
AND2	$(1,0,0,1)^T$	$(0,0,0)$	—	—
	$(0,1,0,1)^T$	$(0,1,0)$	—	—
	$(0,0,1,1)^T$	$(1,0,1)$	—	—
	$(0,0,0,0)^T$	$(1,1,1)$	—	—
NOR2	$(0,0,0,0)^T$	$(0,0,0)$	—	—
	$(1,1,0,0)^T$	$(0,0,1)$	—	—
	$(1,0,1,0)^T$	$(0,1,0)$	—	—
	$(1,0,0,1)^T$	$(1,1,1)$	—	—

Example 3.2.1. *Consider the 3-gate circuit in Fig. 3.6 as an example. Starting at the gate closest to the output, XOR2,* $(0,1,1,0)^T$ *has four distance-1 cubes:* $XOR2_{C_1} = \{(1,1,1,0)^T, (0,0,1,0)^T, (0,1,0,1)^T, (0,1,1,1)^T\}$ *(Table 3.2). The first three cubes can be detected by vectors* $(x_0,x_1,x_2) = (0,1,0), (0,0,0), (1,1,1)$ *respectively. However with the cube* $(0,1,1,1)^T$*, there are no input combinations that are capable of resulting in an output* f *that reflects discrepancies by the presence of this faulty cube. Therefore this particular distance-1 cube is undetectable. To have a better portrayal of errors at this gate, cubes with the next higher distance* (C_2) *is checked. At distance-2, the XOR2 gate has the six cubes:* $XOR2_{C_2} = \{(1,0,1,0)^T, (1,1,0,0)^T, (1,1,1,1)^T, (0,0,0,0)^T, (0,0,1,1)^T, (0,1,0,1)^T\}$*. Each* C_2 *cubes can be detected by* $(1,1,1), (0,0,0), (0,1,0), (0,0,0), (0,1,0)$ *and* $(0,0,0)$ *respectively.*

The next gate to verify is the AND2 gate. Once again, starting at distance-1, all its C_1 *cubes can be detected by* $(0,0,0), (0,1,0), (1,0,1), (1,1,1)$*. With all* C_1 *detectable, there is no need to verifying cubes of higher distances (Lemma 2). The final OR2 gate has all* C_1 *cubes (Fig. 3.2) detectable by* $(0,0,0), (0,0,1), (0,1,0), (1,1,1)$

From Ex. 3.2.1, the total number of implicit faults that underwent verification includes two for the XOR2 gate and one each for the AND2 and NOR2 gates for a total of 4. The algorithm only accounts for highest verified distances for each cube distances applied on each gate. For instance, only $XOR2_{C_2}$ cubes are accounted for in the AND2 gate since checking of the C_2 cubes are essentially a more through check of C_1 cubes. Likewise, since C_1 cubes of both AND2 and NOR2 gates' C_1 cubes were detectable, these are the ones in calculating the fault list size and coverage.

Despite of only partial detection of all C_1 XOR2 cubes, full detectability of C_2 cubes makes any results in C_1 insignificant. With $XOR2_{C_2}$, $AND2_{C_1}$ and $NOR2_{C_1}$ cubes fully detectable, the circuit is 100%

detectable. Likewise, in terms of explicit faults, only higher-distanced cubes that are verified are accounted for. The explicit fault list size includes 6 for $XOR2_{C_2}$, and 4 each in $AND2_{C_1}$ and $NOR2_{C_1}$ for a total of 14. The $14/14$ coverage equates to 100%.

3.2.2 Gate Replacements with MIGSE Modules

Using the same example as Ex. 3.2.1, but except with MIGSE modules, the fault list size is 9, since each of the three gate's replacement module can be represented by 3 *s-a-v* faults. Although the number of explicit cube replacement faults is over four times more than the number of *s-a-v* faults in the module replacements with stuck faults, it seems the cube replacement implicit error models are able to represent a larger number of errors. Additionally, simulating the circuit with the module replacement method shows a result of 88.89%, compared to the higher coverage result by cube distance method at 100%. Other MCNC benchmarks are also ran through various experiments, including ATPG and simulation. Their results, including fault list sizes shall be further discussed in Chapter 5.

CHAPTER 4

FAULT DETECTION BY SIMULATION AND SAT-BASED ATPG

Circuits require a way of systematically generating a fault list and a method of finding a set of tests corresponding to these faults. Test vectors may be generated several ways, such as randomly, and applied at the circuit's inputs for simulation. The drawback of this technique is that there is no guarantee of covering all possible errors defined by the model with a limited set of vectors over a limited period of time. Automatic test pattern generation (ATPG) is a method of generating a list of tests/vectors targeting the fault list. Satisfiability (SAT) -based ATPG is a method of determining an expression based on the overall Boolean network structure and its fault location. By solving the SAT expression, suitable tests are found should they exist or claimed redundant otherwise. Deterministic methods such as ATPG (or SAT-based ATPG) are different when compared to simulation, as they generate vectors based on a targeting fault as opposed to injections of vectors to the design's inputs in hopes that at least one is able to detect the fault. This chapter will first discuss simulation techniques by (pseudo-) random vectors and top-lattice layers and the final half of the chapter will look at deterministic methods such as ATPG, specifically SAT-based ATPG.

4.1 Fault Detection by Simulation

Simulation is a method through injection by a set of vectors that are not generated deterministically from any fault information. The naive solution would be to simulate the circuit with every possible input vector combination. Unfortunately, any circuit with n inputs has a total of 2^n vector combinations. The number of vectors, being exponentially related to the number of inputs is purely unacceptable. Simulating circuits exhaustively is therefore largely infeasible as runtimes would be far too large and costly. To keep verification and test times at more practical durations, only a subset of vectors are chosen in simulation. These must

be chosen carefully for the highest test quality – large fault coverages in the shortest amount of time. Two non-deterministic techniques will be discussed: random and upper lattice layer vectors.

4.1.1 Simulation Using Random Vectors

Random vector simulation is a common way of simulation. In software, these could be generated by the random function, which depends on the CPU clock and/or some sort of seed value. In reality, the numbers/ vectors generated are more pseudo-random, as they are generated based on some form of a complex mathematical algorithm. These results generated are based on a probabilistic distribution. For example, in terms of relative frequency (occurrence) of the vectors, they can be uniformly distributed, meaning every vector has an equal chance of appearing, or they can also be Gaussian or Bernoulli distributed. Depending on the network, the vectors can be generated with a probabilistic distribution that would have a higher chance of detecting certain faults within the network.

In hardware, the same theory applies. A common form of pseudo-random vector generation is via derivations of shift registers. The linear feedback shift register (LFSR) is one which is capable of generating weighted or uniform distributed vectors. Essentially it consists of several flip-flops connected in series. A subset of the flip-flop outputs form a feedback mechanism and are XOR'ed together. The outputs used for feedback determine the characteristic function of the LFSR.

4.1.2 Simulation Using Lattice Structure

Lattice vectors are derived by a representation of vectors as a Boolean lattice. Each vector in the structure possesses a partial order amongst each other. Fig. 4.1 shows the complete lattice structure for 4-bit vectors. Vector 0000 is related to vector 0001, which is related to 0011, with only one bit different between each relation. On the other hand, vector 0001 is not related to 1100 and is therefore not connected in the figure. The lattice is organized by levels, where each level represents the weight (number of ones) of each vector. Because of this, each n-bit lattice at level l has $\binom{n}{l}$ vectors.

It has been proposed in [RZ04] that with an n-variable function, at most $\lceil log_2(n+1)\rceil - 1$ upper layers of the lattice are necessary to describe the function of the network. In other words, if the network has any slight difference in functional properties, the response given by this set of upper lattice layers is sufficient to reflect any difference. This is therefore the minimum number of vectors that are required to detect errors of the circuit – which are essentially modeled as functional errors. For example, a 4-input circuit would require simulation with the top 2 levels, which include vectors: $\{1111, 0111, 1011, 1101, 1110\}$

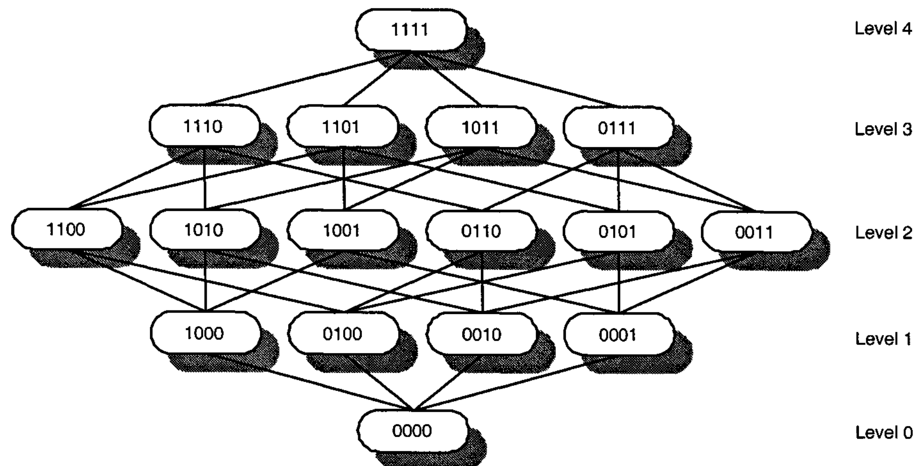


Figure 4.1: 4-bit lattice vectors

4.2 Automatic Test Pattern Generation by Satisfiability

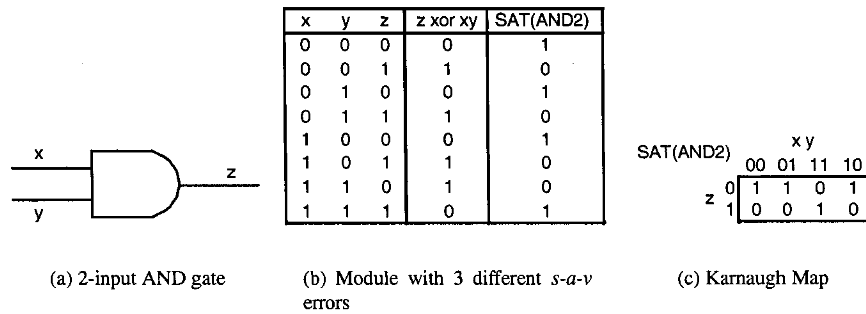
The ATPG method of test generation is based on a topological search given by a specific fault. The search begins at the fault site and traverses forward towards the primary outputs and backwards towards the primary inputs of the circuit. Algorithms for ATPG have been developed, optimized and matured substantially over the years for manufacturing testing *s-a-v* faults. From the first D-Algorithm [RBS67] in 1966 to PODEM [GR81] a decade and a half later, the recent fastest ATPG algorithms [TGH97] [HC00] have been shown to have estimated speedups of 25000 compared to the original implementation [BA00].

An ATPG variant is via satisfiability (SAT) where a Boolean expression is first derived based on the fault of interest and information of the circuit network. This derived expression is then passed onto a SAT solver to determine whether it can be set true, i.e., satisfied. If the expression is capable of ever being true, then the SAT expression is satisfiable and the fault is detectable. There has been plenty of research on SAT solver optimizations and the current fastest ones are known to be some of the fastest test pattern generation algorithms available.

Definition 4.2.1. *Boolean satisfiability (SAT) is a decision problem concerning whether a given expression can be ever be true (or satisfied), when variables in the expression are assigned true or false values. If the entire expression can be true, then the expression is satisfiable.*

A SAT expression can be written in conjunctive normal form (CNF), otherwise also known as the product-of-sums form. It may also be expressed in disjunctive normal form (DNF), otherwise also known as the sum-of-products form.

In obtaining the SAT expression of the network, the first thing to do is to obtain the expressions for each gate or any other type of Boolean logic in the network. The overall SAT is a combination of these expressions



(a) 2-input AND gate

(b) Module with 3 different *s-a-v* errors

(c) Karnaugh Map

Figure 4.2: AND2 Satisfiability

Table 4.1: CNF SAT expressions of basic 1 and 2-input gates

Gate	CNF clauses
$z = BUF(x)$	$(\overline{x}+z)(x+\overline{z})$
$z = INV(x)$	$(x+z)(\overline{x}+\overline{z})$
$z = AND2(x,y)$	$(x+\overline{z})(y+\overline{z})(\overline{x}+\overline{y}+z)$
$z = OR2(x,y)$	$(\overline{x}+z)(\overline{y}+z)(x+y+\overline{z})$
$z = XOR2(x,y)$	$(\overline{x}+y+z)(x+\overline{y}+z)(x+y+\overline{z})(\overline{x}+\overline{y}+\overline{z})$
$z = NAND2(x,y)$	$(x+z)(y+z)(\overline{x}+\overline{y}+\overline{z})$
$z = NOR2(x,y)$	$(\overline{x}+\overline{z})(\overline{y}+\overline{z})(x+y+z)$
$z = XNOR2(x,y)$	$(\overline{x}+\overline{y}+z)(\overline{x}+y+\overline{z})(x+\overline{y}+\overline{z})(x+y+z)$

based on the way each Boolean logic is interconnected amongst each other.

4.2.1 Obtaining SAT Clauses of Boolean Logic

To first obtain clauses of each Boolean gate, one must realize that both inputs and outputs of the gates need to be described in the clauses. To mimic the gate connections amongst the circuit, each of the clauses containing such an information are combined together.

Let $z = f(X)$ be the gate function on the set of inputs X. SAT requires gate output to agree with gate inputs, i.e. output cannot be different from gate function. The following needs to be true:

$$z \oplus f(X) = 0 \tag{4.1}$$

Example 4.2.1 (AND2, Fig. 4.2). *Consider the two-input AND gate, $z = xy$ shown in Fig. 4.2(a). SAT requires z to agree with the AND function xy. In other words, z is not permitted to have a logical value '1' whenever the AND product xy gives a logical '0' and vice versa: $z \oplus xy = 0$. Figure 4.2(b) also shows a truth table relating this expression with output z and inputs x and y. SAT of the AND2 gate is therefore merely the inversion of the exclusive-OR: $SAT_{AND} = \overline{z \oplus xy}$. Using Karnaugh map for minimization (Fig. 4.2(c)), the DNF (Eqn. 4.2) and CNF (Eqn. 4.3) forms can be achieved. Other CNF SAT expressions of 1 and 2-input basic gates can be found in the same way and are listed as shown in Table 4.1. Each product term in the CNF is known as a* clause.

$$SAT(AND2)_{DNF} = xyz + \overline{x} \cdot \overline{z} + \overline{y} \cdot \overline{z} \tag{4.2}$$

$$SAT(AND2)_{CNF} = (\overline{x} + \overline{y} + z)(x + \overline{z})(y + \overline{z}) \tag{4.3}$$

The advantage of this method is the ability to obtain any SAT expression of any type of complex logic gates, i.e., it is not limited to the standard AND, OR, XOR, NAND, NOR, XNOR functions. Because of this, verification and testing of complex logical components such as lookup table (LUT) in FPGAs becomes most convenient (Sect. 4.3, Chapter 6).

4.2.2 Complexity

If the number of terms (literals) in SAT clauses is two (2-SAT), then the clause can be solved in polynomial time. However, if the number of literals is three (3-SAT) or greater, then the SAT solution needs to be solved in exponential time (NP-complete) [Lar92] [BA00]. It is therefore always better to limit the number of literals in SAT clauses to two when possible, although heuristic methods [Lar92] may reduce solving time.

4.2.3 Good, Faulty and Active Clauses with Presence of a *S-A-V* Fault

It is shown in [Lar92] techniques in obtaining the SAT expression of a circuit that include *s-a-v* faults in CNF form. Satisfiability of an expression in this form requires all individual clauses to be satisfied. The complete satisfiability expression to be solved in presence of a *s-a-v* fault involves four categories of clauses:

- Good Clauses
- Faulty Clauses
- Active Clauses
- Fault Site and Goal Clauses

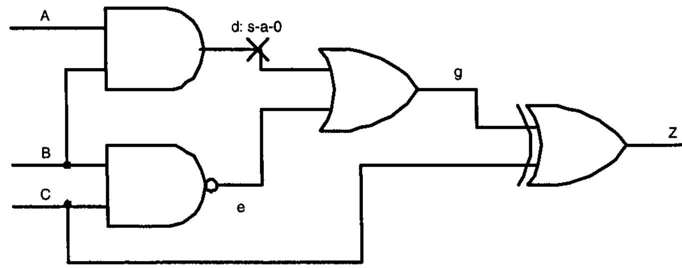


Figure 4.3: Circuit with s-a-v fault to be solved by SAT

Good clauses are determined by the fault-free circuit. Faulty clauses are determined with presence of the *s-a-v* fault. Active clauses are conditions required to active the fault and propagate it to an output. The clauses are also conditions to make sure that each faulty logical values on the sensitization path to the output oppose the unfaulted value and that the fault. The fault size/ goal clauses are mappings of the unfaulted, activated fault, and faulty values at the fault size to actual logical '0' and '1' values. The following are examples of finding and solving finding a test vector of a full circuit by SAT.

Example 4.2.2 (Satisfiability expression of circuit with a s-a-v fault). *Consider the circuit shown in Fig. 4.3 with* s-a-0 *fault at node d. The clauses from all the gates can first be determined from Table 4.1. The good, faulty, active, fault site and goal clauses can be determined from the circuit structure.*

$AND2 : (\overline{A}+\overline{B}+d)(A+\overline{d})(B+\overline{d})$

$NAND2 : (B+e)(C+e)(\overline{B}+\overline{C}+\overline{e})$

$OR2 : (\overline{d}+g)(\overline{e}+g)(d+e+\overline{g})$

$XOR2 : (\overline{g}+C+Z)(g+\overline{C}+Z)(g+C+\overline{Z})(\overline{g}+\overline{C}+\overline{Z})$

Example 4.2.3 (Good clauses). *The good circuit clauses can then be derived from each gates' expression by $ANDing$ them together:*

$$(\overline{A}+\overline{B}+d)\cdot(A+\overline{d})\cdot(B+\overline{d})\cdot(b+e)\cdot(C+e)\cdot(\overline{b}+\overline{C}+\overline{e})$$
$$\cdot(\overline{d}+g)\cdot(\overline{e}+g)\cdot(d+e+\overline{g})\cdot(\overline{g}+C+Z)(g+\overline{C}+Z)(g+C+\overline{Z})(\overline{g}+\overline{C}+\overline{Z})$$

Example 4.2.4 (Faulty clauses). *Faulty clauses are determined by splitting the faulty node into two components. All clauses involving logic entering the faulty node retain its name d to represent the fault free value, as the fault does not take into effect until the signal propagates past the fault site. The erroneous value is denoted by d_f and all logic from this point on up to the output is denoted by subscript f. The faulty clauses are as follows:*

$$(\overline{A}+\overline{B}+d)\cdot(A+\overline{d})\cdot(B+\overline{d})\cdot(B+e_f)\cdot(C+e_f)\cdot(\overline{B}+\overline{C}+\overline{e_f})$$
$$\cdot(\overline{d_f}+g_f)\cdot(\overline{e}+g_f)\cdot(d_f+e+\overline{g_f})\cdot(\overline{g_f}+C+Z_f)\cdot(g_f+\overline{C}+Z_f)\cdot(g_f+C+\overline{Z_f})\cdot(\overline{g_f}+\overline{C}+\overline{Z_f})$$

Example 4.2.5 (Active clauses). *Active clauses involve clauses describing conditions required to sensitize an "active" path to circuit outputs. It also includes clauses which ensure discrepancy between the unfaulted value with the faulty value whenever the sensitized path is active. The active clauses are as follows:*

$$(\overline{d_a}+d+d_f)\cdot(\overline{d_a}+\overline{d}+\overline{d_f})\cdot(\overline{g_a}+g+g_f)\cdot(\overline{g_a}+\overline{g}+\overline{g_f})\cdot(\overline{Z_a}+Z+Z_f)\cdot(\overline{Z_a}+\overline{Z}+\overline{Z_f})\cdot(\overline{d_a}+g_a)\cdot(\overline{g_a}+Z_a)$$

Each of the first three pairs of clauses describes conditions on each of nodes on the sensitizing path going from d through g to Z. The first pair states, "When d is active, the fault-free and faulty value of d cannot be equal to show discrepancy." The next two pairs of clauses are similar but represent nodes g and Z. The final two clauses represent activation between each node from the fault to the output. Specifically, when d is activated, g has to be activated. Likewise, when g is activated, output Z is also activated.

Example 4.2.6 (Fault site and goal clauses). *The fault size and goal clauses are as follows:*

$$(d_a)(d)(\overline{d_f})(Z_a) \tag{4.4}$$

The d_a clause is to explicitly activate node d. The d and $\overline{d_f}$ clauses are there since the fault d_f is 0 while the original d is 1. The single Z_a clause is to explicitly ensure this output port is activated such that any discrepancies are visible.

Example 4.2.7 (Solve the SAT expressions determined in Ex. 4.2.3, 4.2.4, 4.2.5, 4.2.6). *Since all fault size and goal clauses in Eqn. 4.4 are unary (1-SAT), there can be used as the starting point in solving the entire overall expression to find suitable vector(s) capable of detecting the fault: $d_a = 1$, $d = 1$, $d_f = 0$, $Z_a = 1$*

$$\begin{array}{ll}
\textit{From } d = 1: & (A+\overline{d}) \Longrightarrow \underline{A=1} \\
 & (B+\overline{d}) \Longrightarrow \underline{B=1} \\
 & (g+\overline{d}) \Longrightarrow g=1 \\
\textit{From } d_a = 1: & (\overline{d_a}+g_a) \Longrightarrow g_a=1 \\
 & (\overline{g_a}+Z_a) \Longrightarrow Z_a=1 \\
\textit{With } g=1 \textit{ and } g_a=1: & (\overline{g_a}+\overline{g}+\overline{g_f}) \Longrightarrow g_f=0 \\
\textit{With } g_f=0: & (\overline{e}+g_f) \Longrightarrow e=0 \\
\textit{With } e=0: & (C+e) \Longrightarrow \underline{C=1}
\end{array}$$

The set of vectors capable of detecting d :s-a-0 is therefore: $(A, B, C) = \{(1, 1, 1)\}$.

4.3 Redundant Fault Identification Method for Cube Replacement Errors

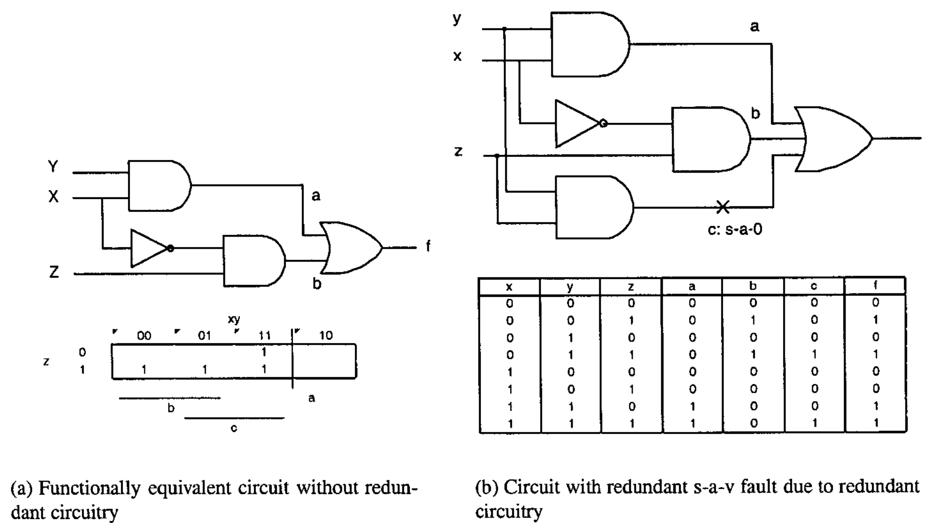
Often enough, faults in a circuit are either very difficult to detect or simply impossible to detect at all. Unfortunately during simulation, a fault's presence is not revealed until the very first vector that is capable of showing differences is simulated. In the worst case, the fault remains undetected even after simulation with the last vector when only a subset of vectors is simulated. In this scenario, it remains inconclusive whether the fault would ever be detectable even if more vectors are used continue simulating it. Redundant faults are known as faults that are impossible to detect. The term "redundant" suggests that the presence of the fault does not have any affect on the circuit's operation. Having the fault in the circuit is merely extraneous.

As can be seen in simulation, finding exactly whether a fault is redundant is not possible without at least simulating the circuit exhaustively. The main concern in finding redundant faults is to be able to identify them with pure confidence without declaring a fault that is not redundant, or even one that is difficult to detect, as redundant. The Berkeley SIS tool [SSL+92] uses SAT-based methods in determining these with exact accuracy. Methods of identifying redundant gate replacement faults have also been discussed previously in [RZ01].

The proposed algorithm in this thesis begins with simulation by a set of vectors, either generated randomly or predefined. If the set of vectors are unable to check for the fault's detectability, a redundancy removal routine is performed. The redundancy removal processes is merely discarding the redundant fault from the overall fault list, such that the fault is not to be checked later during testing. This is to ensure to not waste any valuable time checking for an error after already knowing that it is purely impossible to detect.

4.3.1 Redundant Faults Caused by Redundancy in Circuits

In combinational circuits redundant faults exists in circuits containing reconvergent fan-outs [ABF94]. This is when a node in the network branches out and reconverges back together at a node as the signal propagates further. Looking at this logically, the fault is more tolerant when other parts (branches) of the circuit are able to compensate and override errors and their propagations originating from the fault site. When this happens, conditions on the faulty branch are essentially Don't Care (DC) cases, meaning that despite any problems in a faulty branch, another one would override the values to give the correct circuit output, masking the presence of the fault.



x	y	z	a	b	c	f
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	0	0	0
0	1	1	0	1	1	1
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	1
1	1	1	1	0	1	1

(a) Functionally equivalent circuit without redundant circuitry

(b) Circuit with redundant s-a-v fault due to redundant circuitry

Figure 4.4: Redundant s-a-v fault

Example 4.3.1 (Redundant s-a-v fault). *A circuit with its Karnaugh map is shown in Fig. 4.4(a). Its Sum-Of-Products (SOP) form is realized by the product terms denoted by* $a = xy$ *and* $b = \overline{x}z$ *in the figure. Consider the case when product* $c = yz$ *is included in the circuit realization. Fig. 4.4(b) is a functionally equivalent circuit of Fig. 4.4(a) with includes term c is in its realization. From the Karnaugh map, it is clear that the additional product term highlighted by c is a not necessary in the canonical form of SOP. The AND gate on the bottom branch of Fig. 4.4(a) is the result of this added term.*

Suppose a s-a-0 *fault is injected in to the circuit at c. The output* f *is reduced to the logic produced only by the upper two branches. Taking the logical outputs of nodes a and b has no effect on* f*, despite any input combinations of* x*,* y *and* z*. Fault* c: s-a-0 *is redundant because its functional output is unaffected even with its presence in the circuit.*

As far as the circuit's functional operation is concerned, realizing the extraneous product term produced by node c is advantage of making it more fault tolerant to faults such as c: s-a-0*, but the use of redundant hardware makes the circuit more difficult to verify and test.*

4.3.2 Impact of Don't-Care (DC) Conditions on Redundant Cube Distance Replacement Faults

Similar to *s-a-v* faults, DC conditions have a large impact on the detectability of error cube distance errors. The origin of redundant faults is due to DC conditions in the circuit. In cube distance error or gate replacements, when all input combinations corresponding to the cube differences are DC, the Boolean difference between the fault-free and faulty networks may never be detected. The error replacements are

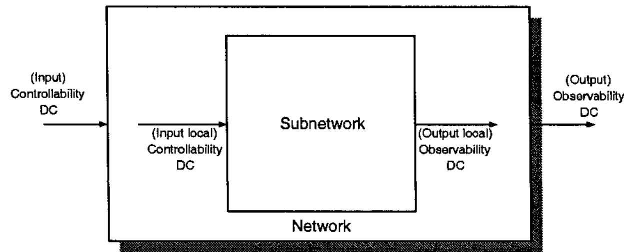
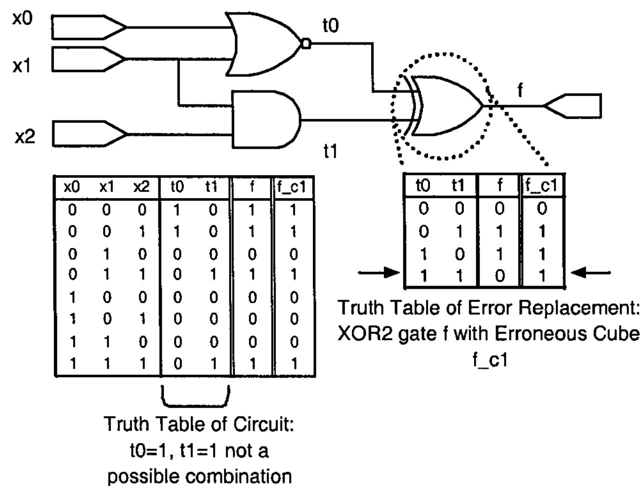


Figure 4.5: Controllability and observability DC



x0	x1	x2	t0	t1	f	f_c1
0	0	0	1	0	1	1
0	0	1	1	0	1	1
0	1	0	0	0	0	0
0	1	1	0	1	1	1
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	1	1	1

t0	t1	f	f_c1
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

Figure 4.6: Redundant cube distance replacement $f \rightarrow f_c1$

therefore redundant. If DC conditions impair detectability of cube distance-i errors, cube distance-$i+1$ errors are used for checking (Sect. 3.1.3).

To look at DC a little closer, a circuit environment's over DC can be categorized into controllability DC (CDC) and observability DC (ODC) (Fig. 4.5). CDC (or input CDC) is defined as input conditions that can never be produced at the network's inputs. ODC (or output DC) conditions "are input patterns that represent situations when an output is not observed by the environment" [Mic94]. The CDC and ODC can be analyzed as information of a subcircuit, or as a circuit part of a network. Ex. 4.3.2 shows an example of a redundant cube distance replacement due to CDC conditions of a circuit. Ex. 4.3.3 shows an example where faults are redundant due to ODC conditions.

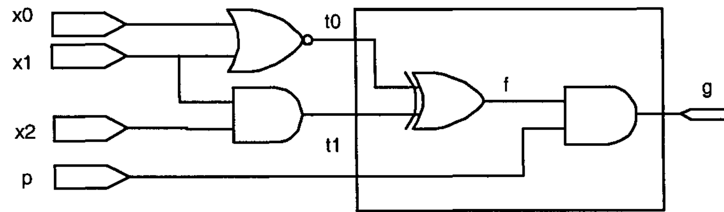


Figure 4.7: Circuit with observability DC

Example 4.3.2 (Redundant cube distance replacement and CDC). *Recalling from Ex. 3.2.1, the circuit (shown again in Fig. 4.6) is undetectable when the XOR gate near the output is replaced with the cube distance-1 error function* $f_{c1} = (0, 1, 1, 1)^T$.

Let the subcircuit be defined as the lone XOR gate. The input CDC of the subcircuit is $(t_0, t_1) = \{(1, 1)\}$*, because this is the (only) combination that is impossible to be produced at the local subcircuit's inputs. The constraint is due to the logic that drives the subcircuit, "filtering" out this input combination possibility. CDC conditions impair the ability to "control", or set the appropriate values to activate the fault. Because of this local CDC, this replacement error fault can never be activated and would never be detected when simulated with overall any input vector at* (x_0, x_1, x_2)*. Hence, this replacement fault is redundant.*

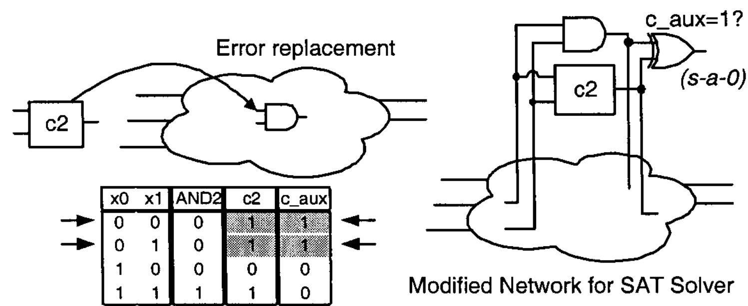
Example 4.3.3 (Redundant cube distance replacement and ODC). *Consider a similar circuit shown in Fig. 4.7, where the subcircuit is defined by the bounding rectangular box. If the input p is set to* 0*, output g would always be* 0*. In this case, even if the XOR gate experiences any fault, the output g would remain low. All vectors that has* $p = 0$ *are considered to be ODC conditions, since these are cases that forbid the circuit from reflecting any sort of error even if the fault is activated.*

4.3.3 Redundancy Identification of Cube Distance Replacement Errors

Exact gate replacement error identification had previously been proposed in [RZ01]. Techniques in identifying redundant cube-distance errors are the basis of the approach proposed here. Let c represent the original function, and let c_i be one with cube-error of distance-i, as previously defined. Also, let function c_{aux} be defined as the auxiliary (Boolean difference) function between c and c_i:

$$c_{aux} = c \oplus c_i \tag{4.5}$$

The detectability of c_i is translated to the ability of monitoring any difference between the two cubes in the output. Should there be the ability of setting c_{aux} to 1, fault c_i is detectable.



x0	x1	AND2	c2	c_aux
0	0	0	1	1
0	1	0	1	1
1	0	0	0	0
1	1	1	1	0

Figure 4.8: Cube redundancy check

Consider a c_2 replacement shown in Fig. 4.8, where the first two input combinations mark the cube errors. The difference between the original gate AND2 and c_2 is used to form caux. In the example, $c_{aux} = (1,1,0,0)^T$. Notice that c_{aux} also marks the locations of all the error locations in c_i. Any instances of '1' in the c_{aux} correspond to the input combinations that may trigger the fault.

The technique used for redundancy identification requires a generation of satisfiability (SAT) clauses via the conjunctive normal form (CNF)[Lar92] of c_{aux}. The purpose of these clauses is to describe a fault free circuit, fault location as well as any restrictions imposed on the circuit by the given fault (observability and controllability conditions). From the example in Fig. 4.8, the requirement is $c_{aux} = \overline{x_0}$. Hence the SAT clause: $(x_0 + c_{aux})(\overline{x_0} + \overline{c_{aux}})$ is combined with the clauses that described the entire original network before error injection. In practice the constraining SAT clauses could be constructed by taking the xor of both the original and erroneous nodes. Since the requirement is to set the output of the XOR, i.e., c_{aux} to 1, a *s-a-0* fault can be inserted at the XOR output to force the ATPG-SAT solver into generating all redundancy identification clauses automatically.

CHAPTER 5

FAULT DETECTION OF COMBINATIONAL DESIGNS WITH BERKELEY SIS

Berkeley SIS is a logic synthesis and optimization tool developed at University of California at Berkeley [SSL+92]. Its source code is available online, which makes it convenient for people to modify and add customized routines to tailor their research. Included are numerous built-in routines such as logic minimization, netlist decomposition, SAT-based ATPG and equivalence checking. MCNC benchmarks in SIS are also available and have been used countless number of times in publications. For these reasons, SIS is the tool of choice for the experiment implementations for the thesis. All experiments are implemented on top of the UCLA version (*sis-1.3*) [UCL] for Solaris, since it has all the unnecessary harmless compile warnings removed from the original Berkeley version. Also, an unofficial Linux port [Cho] is used from time to time to check for compatibility and memory leak issues. For runtime memory leak tracking, several tools such as Electric Fence [Per] and Alleyoop [Ste] for Linux have been used during the development process.

Recalling the design flow discussion depicted in Fig. 1.1, this chapter concerns the first part of the flow as shown in more detail in Fig. 5.1. This involves performing experiments involving initial synthesis followed by SAT-based ATPG on benchmarking circuits. These parts are all added to the original SIS package (in C) with numerous customizations.

5.1 Overview of Experiments with SIS

The SAT-based ATPG experiment flow in SIS is shown in Fig. 5.2. After reading the design, the netlist undergoes decomposition to a maximum gate input limit and other pre-processing to emulate the synthesis process before the experiments. Next, the netlist is traversed where each gate is analyzed and identified by its Boolean function. The fault list consisting of erroneous cube-distance functions, gates or MIGSE faults

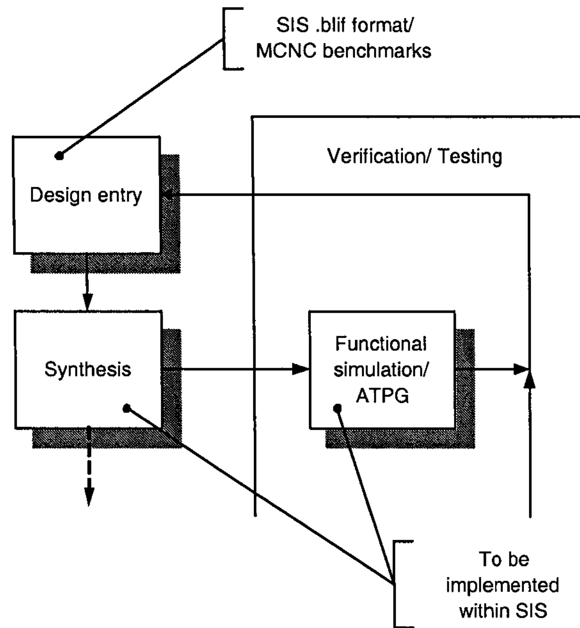


Figure 5.1: Functional simulation flow

from [AAH95][BH97] (Chapter 2) are created to be substituted in place of the original node, depending on the type of fault modeling of interest. Either simulation or (SAT-based) ATPG is used in attempt to detect the error, and is repeated throughout the entire fault list. At the end of the experiments, the fault coverage, execution time and fault list size is recorded for comparison. This section will first look at most relevant SIS libraries that facilitated the programming part of the experiments. Afterwards, details of fault list generation and detection of each error model will be looked at.

5.1.1 SIS Programming Libraries

SIS contains several categories of internal libraries that facilitate large parts of the programming process used to set up the experiments.

5.1.1.1 Node

Each connection of a loaded netlist in SIS consists of nodes. Predefined node data structure and libraries, are available. Hence, it is not necessary to tamper with graphs and manipulate binary decision diagrams (BDD) of circuits. The node may carry any sort of Boolean function such as AND and OR, as well as any type of complex functions that are not part of the standard Boolean logic library. In other words, the node function is essentially a superclass of the standard gate function.

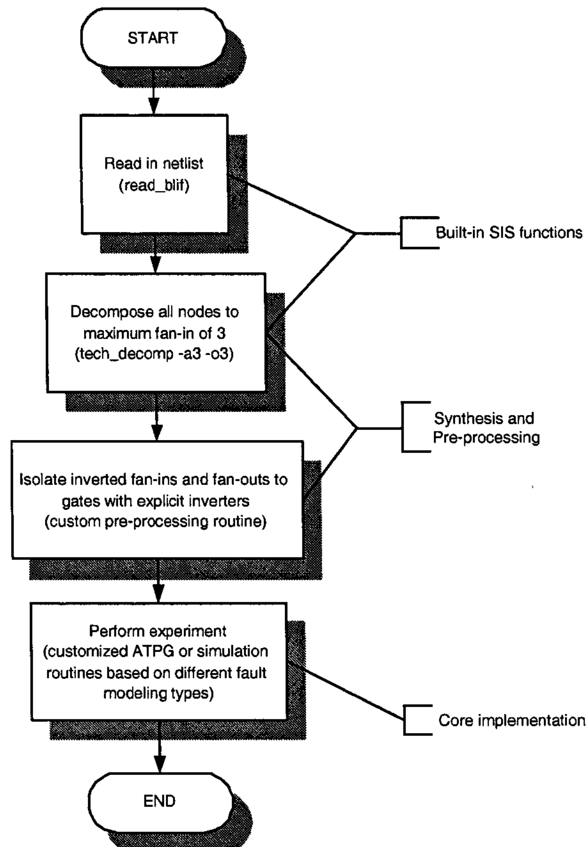


Figure 5.2: SAT-based ATPG/ simulation flow for replacement model fault experiments

5.1.1.2 Network

The network library contains functions that may be used to manipulate the circuit in memory. Some of these include functions for inserting and removing nodes from a network which becomes convenient in gate/node/module replacement experiments.

5.1.1.3 Espresso

The espresso library contains data structures storing information of the node's logical information. This information is stored as ON-sets can be used to read and manipulate the "onset" of the node. This is extremely useful when the ON-set of a node need to be changed to reflect the error the tool needs to verify for cube-distance errors discussed in the thesis. Each ON-set of a node is stored as type *pset*, and the overall node's set of ON-sets are stored as type *pset_family*. The *pset_family* (complete set of cubes) is stored in the *node_t* structure defined in the *node* library.

5.1.1.4 Sim

The simulation library takes any vector and generates a vector of outputs. Routines in this library are used to apply vectors for each of the gate, MIGSE *s-a-v* module and cube distance replacement models.

5.1.1.5 ATPG

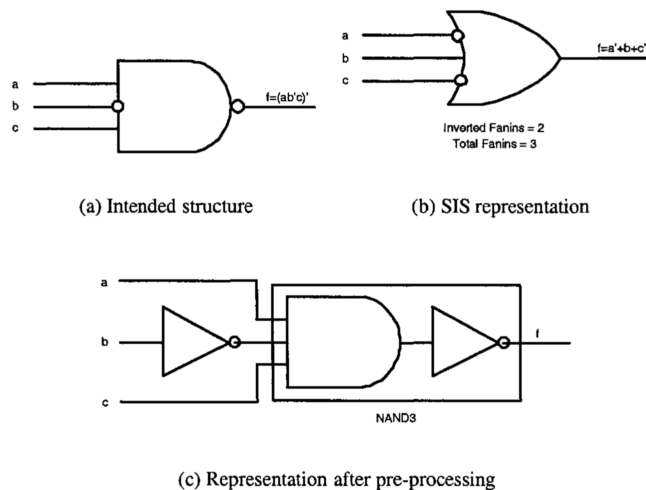
The ATPG library includes the routines which first generates a list of *s-a-v* faults. The list then undergoes fault collapsing, followed by execution of SAT-based ATPG. The ATPG library is modified to perform experiments on each of the gate, MIGSE *s-a-v* module and cube distance replacement models.

5.1.2 Design Preprocessing and Synthesis

Preprocessing is only necessary in experiments to be performed in SIS, as original ATPG for *s-a-v* faults and simulation routines were written to perform on the network of nodes described in the *.blif* format, as opposed to a netlist mapped to a specific technology and library of gates. Hence the design in memory is decomposed into gates of maximum inputs of three to mimic the synthesis gate mapping process. This constraint place on the gates is also to help maintain a relatively faster SAT execution than the time required with gates of larger inputs.

5.1.2.1 Dealing with Negative Logic Gates

Internally, all the gates within SIS are represented in terms of nodes, whose cubes are merely a mapping of the inputs to outputs via a look-up table. Depending on the ON-set table of the node, SIS has several types of logical node function defined to which they are mapped, including NODE_AND (ex. $A \cdot B$), NODE_OR (ex. $A + B$) and NODE_COMPLEX. NODE_COMPLEX represents all *XOR*-typed nodes as well as any other complex logic types not part of the standard Boolean gate class. It is noteworthy that negative logic gates are not defined as part of the enumeration. In fact, negative-logic nodes are mapped as existing positive *node_function* types based on De Morgan properties. For example, a node with the function $NAND(a, b)$ is seen as an $OR(\overline{a}, \overline{b})$ node. Because of this De Morgan equality, the node is recorded as NODE_OR as opposed to NODE_AND (or its negative). As a result, this poses a challenge when doing gate replacement experiments when it is imperative to properly detect the original gate type. Another small obstacle comes with the need to distinguish gates such as $OR(\overline{a}, b)$ and $OR(a, \overline{b})$ apart, where only a subset of all the inputs is inverted. In this particular example, SIS records both node types as an NODE_OR. To mitigate these subtleties for the sake of the flow of the experiment, a command is written to pre-process the circuit in memory so that ATPG/ simulation routines for error replacement fault model tests will properly identify these gates.



(a) Intended structure

(b) SIS representation

(c) Representation after pre-processing

Figure 5.3: Decomposing inverted inputs and fan-outs of nodes using *split_neg*

5.1.2.2 Isolation and Minimizing the Number of Inverted Inputs and Outputs

The idea used in the pre-processing script is isolate all negative inputs and outputs from the node. All the negative logic gates are decomposed to a single inverter attached to the output of the positive node. For example, a NAND gate would be decomposed into an AND gate with an attached inverter at its output. Also, all the inverted inputs are also isolated from the gate. Another rule is to keep the number of inverted inputs to a minimum. If the number of inverted inputs exceeds half of the total number of inputs, De Morgan is applied to the node. This allows the netlist to have a mixture of both positive and negative typed logic. Fig. 5.3 shows an example of a gate of function $\overline{\overline{a} \cdot b \cdot \overline{c}}$ before and after the execution of this SIS script. In Fig. 5.3(a), the intended structure is shown as the NAND gate with an inverted input at b. The gate function that SIS should detect in the netlist should be a NAND gate. In Fig. 5.3(b), because of gates need to be represented as either AND, OR or (XOR) COMPLEX, SIS shows the node as an OR gate with inputs a and c inverted, by De Morgan properties. After performing the customized routine within SIS to formally isolate inverted inputs, whenever SIS detects a node with an inverter tied to its immediate output, it realizes this as an negative-logic gate of type NAND, NOR or (XNOR) Complex, as shown in Fig. 5.3(c).

5.1.2.3 New SIS command: *split_neg*

The new SIS command *split_neg* applies the De Morgan equality to limit the number of inverted inputs to a minimum and isolates all negative inputs and fan-outs from a node. For now, the script works for all *AND*, *OR* and *XOR* nodes, but only works for a maximum number inputs of 3, since all the experiments are checked with this maximum constraint. It is recommended that this command be executed after applying

```
1  void original_atpg() {
2
3     /* Fault list generation: */
4     foreach node in network {
5        faultlist = append(s-a-0, node, faultlist);
6        faultlist = append(s-a-1, node, faultlist);
7     }
8
9     fault_collapsing(faultlist);
10    foreach fault in faultlist{
11       clauses = setup_sat_clauses(fault);
12       [vector, isDetectable] = solve_sat(clauses);
13
14       /* Other optional routines such as: */
15       reverse_simulation(vector);
16
17       tally_coverage_statistics (isDetectable);
18    }
19
20    report_results();
21 }
```

Figure 5.4: Original ATPG algorithm in SIS

decomposition of the netlist using *decomp* or *tech_decomp* to satisfy the maximum input limit.

Since all three types of replacement models represent logic in both polarities, only the node, excluding the inverters attached at the inputs and outputs attached, needs to be replacement. As long as performing error replacement models do not touch the attached inverters, the experiment shall be able to check for replacements in both polarities. For example, in an *NAND* isolation (*AND* in series with *NOT*), checking for *AND* $\rightarrow$ {*OR*, *XOR*, *NAND*, *NOR*, *XNOR*} replacements is the same as checking for the intended *NAND* $\rightarrow$ {*NOR*, *XNOR*, *AND*, *OR*, *XOR*} replacements.

5.2 Fault List Generation and Modifications to ATPG Routines for Gate-Level Replacement Error Models

It is necessary to modify the ATPG routines targeting single *s-a-v* faults for manufacturing testing to accommodate the various models that need to be checked. The unmodified abridged SAT-ATPG algorithm (pseudo-code) in SIS is shown in Fig. 5.4.

For the most part, the original fault list routine traverses the netlist and first generates a fault list. During the traversal, it stores information including the fault site location (node pointer reference), as well as the *s-a-v* fault value associated with the node for each fault. This is followed by *s-a-v* fault collapsing on the fault list, The first task was to modify the fault list generation routines to contain more information from the type of fault modeling technique being used. With the appended information for each fault, the latter part of ATPG routines apply added constraints expressed as additional SAT clauses for the solver.

Table 5.1: Output functions of *AND2* and *OR2*

x_0	x_1	$AND2(x_0, x_1)$	$OR2(x_0, x_1)$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

5.2.1 Gate Replacement with Gates of Compatible I/O Count

Before generating the fault list for gate replacement modeling, the first task was to create hard coded table lookup for all standard 2- and 3-input Boolean gates (AND, OR, XOR, NAND, NOR, XNOR). Contents of this table lookup include cube mappings for all input combinations. The software routine then traverses the netlist gate by gate, and determines the function of the gate. Despite limited enumeration of the NODE_FUNCTION type, the gate function can be determined easily with the netlist pre-processed and formalized before execution of the ATPG routine as discussed in Section 5.1.2.

With the gate function extracted, the routine goes though all five candidate gate replacements and checks the table lookup for input combinations that trigger a difference in the gate output. Whenever such a difference is found, the routine stores information including the original gate, the replacement gate, the input combination that triggers the error, as well as the fault-free/ faulty gate output. For each difference in the table lookup, the information is stored as one *s-a-v* fault. Therefore, each explicit candidate gate replacement may map to more than one *s-a-v* fault in the *s-a-v* fault list.

Example 5.2.1 (Fault list from AND2 → OR2 replacement). *Consider the AND2 → OR2 replacement. From the table lookup in the routine, the data in Table 5.1 can be found. Two rows mark the input patterns that may excite the output difference are stored as individual stuck faults as follows:*

Subfault 1: input: $(0, 1)$ *correct o/p:* 0 *wrong o/p:* 1 *original: AND2* *replacement: OR2*

Subfault 2: input: $(1, 0)$ *correct o/p:* 0 *wrong o/p:* 1 *original: AND2* *replacement: OR2*

Upon completing the fault list, each fault in the fault list can be executed by the ATPG-SAT solver to determine whether the fault can be detected. This is achieved by adding 1-SAT clauses corresponding to each input that corresponding to the stuck fault. If the input variable x corresponding to input is 1 for the stuck fault, then the unary clause (x) is appended to the list of clauses. Otherwise, if the input variable x is 0, the unary clause $(\overline{x})$ is added instead. More details of setting up SAT clauses for pure *s-a-v* have been previously discussed in Section 4.2. The default SAT routines can be reused within SIS for this purpose. As a reference, explicit gate replacement fault identification using ATPG has been previously described in [RZ01].

The technique is essentially identical here for cube distance replacement errors. Ex. 5.2.2 traces how unary clauses are created from Ex. 5.2.1.

Example 5.2.2 (ATPG on gate replacement fault list). *Using Ex. 5.2.1, each row of error can be each analyzed as follows:*

Subfault 1: Can gate be satisfied such that $x_0 = 0$, $x_1 = 1$ *with* s-a-1@*output?*

- *Step 1: Add stuck at fault* s-a-1 @ *output to be checked by ATPG*
- *Step 2: Reference fault size to the gate's output*
- *Step 3: Set up good, faulty, active, fault site/goal by original SIS ATPG routine*
- *Step 4: Append SAT clauses:* $(\overline{x_0}) \cdot (x_1)$ *to list of SAT clauses created by Step 3.*

Subfault 2: Can gate be satisfied such that $x_0 = 1$, $x_1 = 0$ *with* s-a-1@*output?*

- *Step 1: Add stuck at fault* s-a-1 @ *output to be checked by ATPG*
- *Step 2: Reference fault size to the gate's output*
- *Step 3: Set up good, faulty, active, fault site/goal by original SIS ATPG routine*
- *Step 4: Append SAT clauses:* $(x_0) \cdot (\overline{x_1})$ *to list of SAT clauses created by Step 3.*

Despite two separate errors represented in the fault list in Ex. 5.2.1 and 5.2.2, the combination of the two is necessary to represent the AND2-OR2 gate replacement. It is up to the modified routines to take this into account and count each gate replacement as one single error when calculating the coverage, as opposed to the number of stuck faults needed to represent the error. The rule is that as long as at least one error (row) of the gate replacement is detectable, then the gate replacement is irredundant as there is at least one satisfied vector capable of detecting the replacement fault. The updated overview of the customized ATPG routine for gate replacement error is shown in Fig. 5.5.

Compared to the original algorithm for *s-a-v* faults, the fault collapsing routine is purged as this is no longer applicable in gate replacement error modeling when the stuck fault errors are only limited to a subset of input combinations, which are depending fully on the functions involved in the gate replacement.

Another subtle detail is that the modified ATPG routines do not involve actual gate replacements. The gates in the netlist are left untouched, as the unary SAT clauses that constrain the inputs together with the *s-a-v* for output differences are sufficient to represent the error. Explicitly performing a node replacement at the fault site may have negative effects, possibly causing the *s-a-v* at the gate output not being able to be activated. Using Ex. 5.2.1 and 5.2.2, if the gate is explicitly replaced by the OR gate, the fault size and activation clauses would need to be updated to accommodate the OR gate. It is therefore much simpler to leave the gates unchanged.

```
1  void gate_replacement_atpg() {
2
3     /* Fault list generation: */
4     foreach node in network {
5        get_node_function(node);
6        get_node_num_fan-in(node);
7
8        /* foreach gate in library  */
9        for (replace=AND; replace<=XNOR; replace++) {
10
11          /* make sure the replacement is not the same as the original */
12          if (node_function(node) != replace) {
13             /* Look up table and find differences between each gate */
14             gate_diff_list = gate_diff(node, replace);
15             foreach diff in gate_diff_list {
16                faultlist = append(gate_diff_list(s-a-v), gate,
17                                   replace, node, inputs, faultlist);
18             }
19          }
20
21       }
22    } // fault list generated
23
24    foreach fault in faultlist {
25       clauses = setup_sat_clauses(fault, gate_diff);
26       [vector, isDetectable] = solve_sat(clauses);
27
28       /* Other optional routines such as: */
29       reverse_simulation(vector);
30
31       /* update number of tested/redundant faults for next iteration */
32       tally_coverage_statistics (isDetectable, gate, gate_diff);
33    } // foreach fault
34
35    report_results();
36 }
```

Figure 5.5: ATPG algorithm for gate replacement errors

5.2.2 MIGSE Module Replacement

ATPG experiments using the MIGSE module replacement with *s-a-v* faults are different compared to the explicit gate replacement model where substitution of gates of compatible I/O count could be represented by introduction of a stuck fault with input constraints by unary SAT clauses. When modeling replacement errors with MIGSE modules, each module needs to be explicitly replaced into the netlist. Several candidate *s-a-v* faults in the MIGSE module constitutes as part of the fault list, but each stuck fault in the module represents one type of error in the gate, with a one-to-one mapping. Since each *s-a-v* fault is embedded within the MIGSE module, there is no way of referencing the actual location of the *s-a-v* fault prior to insertion and after removal of the module. Thus, an individual fault list is created with each MIGSE module replacement, followed a local ATPG execution to check only for the specific *s-a-v* faults internal to the module. Fault coverage statistics are accumulated with each replacement at each node throughout the netlist.

Similar to the gate replacement mode, each fault in the fault list representing the module replacement also requires knowledge of the fault site by a node reference. Fig. 5.6 shows the algorithm of performing ATPG on a synthesized netlist based on this model.

```
1  void MIGSE_with_sav_atpg()
2  foreach node in network {
3     find the node function (AND, OR, etc)
4     get the number of node fan-ins
5     replace node with replacement module
6     from the replacement module, get list of s-a-v within module
7     set the list of stuck faults in module to be the fault list
8     run ATPG with module inserted
9     record/update number of tested and redundant faults
10    restore network by replacing module with original gate
11 }
12 report overall results
```

Figure 5.6: ATPG algorithm for MIGSE module replacements

Table 5.2: Output functions of *AND2* and its C_1 cubes

x_0	x_1	$AND2(x_0, x_1)$	c_1	c_1	c_1	c_1
0	0	0	1	0	0	0
0	1	0	0	1	0	0
1	0	0	0	0	1	0
1	1	1	1	1	1	0

5.2.3 Cube Distance Error Replacement

Modifications of *s-a-v* ATPG routines for cube distance replacement error modeling are similar to the gate replacement method discussed in Section 5.2.1 for the most part. Instead of having several candidate gate replacements of compatible I/O count for each gate throughout the netlist, the routine generates a list of candidate error functions based on the gate's function. Starting at distance-1, if all error functions at this distance (C_1) is detectable, then the checks at this node is complete. Otherwise, it checks for error functions at one distance higher (C_2). ATPG on the node is complete as soon as all errors of any distance are detectable, or until a maximum distance is reached.

For example, an AND2 gate in gate replacements involve considering all five candidate gates for replacements. In cube distance replacement, the set of (4) distance-1 cubes are generated (Table 5.2). If all are detectable, then verification on this AND2 gate is complete, by Lemma 2 and 3. Otherwise, it generates the set of (6) distance-2 cubes to be verified. The process is repeated as necessary for this gate until it reaches a maximum distance threshold. In the experiments, the maximum distance is arbitrarily set to 3.

Since all candidate error functions of the standard Boolean gates are always fixed and the netlist consists of multiple instances these gates, a hard coded table is created associating each gate in the library with its list of cube distance replacement candidates. This is to speed up verification time. Unary ATPG clauses are added the same way as gate replacement errors to constrain inputs targeting the differences represented by *s-a-v* faults. Fig. 5.7 shows the ATPG algorithm on designs with minimal implicit cube distance replacement model.

```
1  void cube_atpg()
2
3  generate lookup table for error functions from standard gates
4
5  foreach node in network {
6     find the node function (AND, OR, etc)
7     get the number of node fan-ins
8
9     for (distance=1; (distance<=max_distance); distance++) {
10
11       get error replacement list (cube distance error lookup table)
12       for each replacement error {
13          /* perform ATPG */
14          generate SAT clauses by default ATPG routine
15          append SAT clauses for input constraints
16          set faulty clauses to stuck fault given by erroneous output
17          run SAT solver
18          record statistics
19       }
20       update coverage statistics
21
22     record/update number of tested and redundant faults
23
24     }
25  }
26  report overall results
```

Figure 5.7: ATPG algorithm for minimal cube distance error replacements

5.2.3.1 Fault Coverage Calculation

Example 5.2.3. *Consider the following examples (Ex. 5.2.3 and Ex. 5.2.4) where the fault coverage of a 3-input gate is calculated. The total number of possible of functional faults is* $2^{2^3} = 256$ *and represented by* $2^3 = 8$ *implicit cube distance faults:* $C_1, C_2, \cdots, C_7, C_8$. *With the arbitrary maximum distance set at 3, only* C_1, C_2 *and* C_3 *will be checked.*

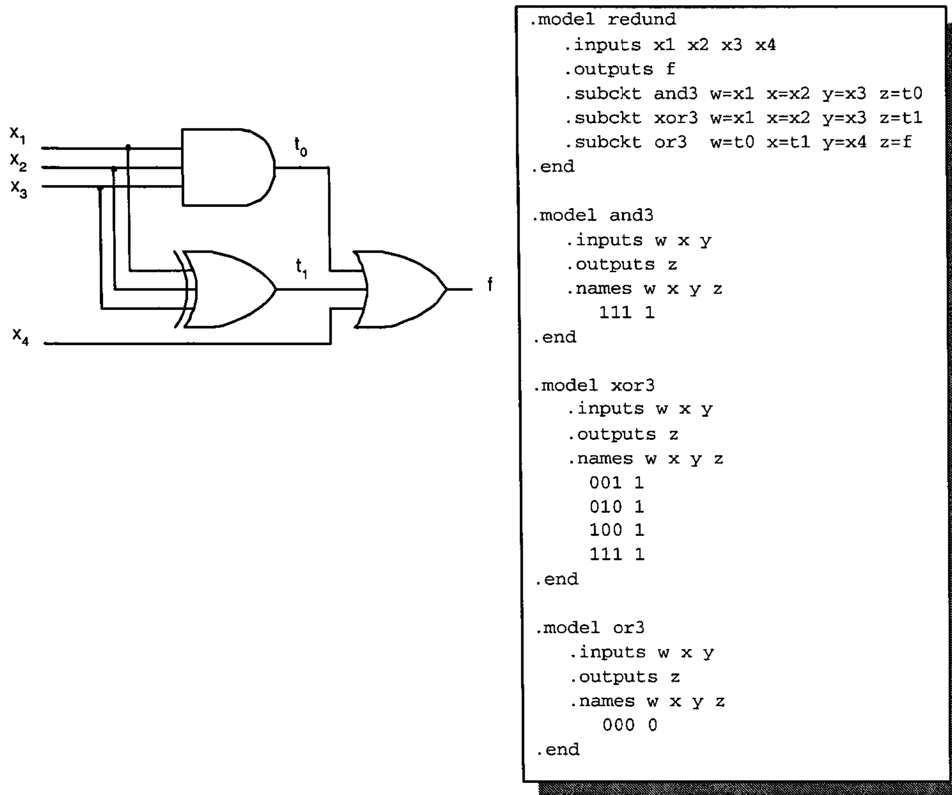
If none of the implicit faults are fully detected, i.e. C_1*: 7 of 8 detected,* C_2*: 20 of 28 detected,* C_3*: 55 of 56 detected,*

Fault coverage of the 3-input gate: $\frac{7+20+55}{8+28+56} = 89.13\%$

Example 5.2.4. *If one implicit fault fully detected, i.e.* C_1*: 7 of 8 detected,* C_2*: 28 of 28 detected,* C_3*: Untested,*

Fault coverage of the 3-input gate: $\frac{7+28+56}{8+28+56} = 98.91\%$

In the Ex. 5.2.3, as none of the implicit faults can be fully detected, all C_1, C_2 and C_3 need to be tested. Calculation of fault coverage merely takes the ratio of all the detectable faults to the total number of replacements. In Ex. 5.2.4, as C_2 faults are fully detectable, it is unnecessary to check C_3 errors as detection is implied. Fault coverage needs to take into account of all of these 56 unchecked C_3 replacement errors. This maintains an equal weight of all the gates' testability when calculating the overall coverage of the network. Since only a subset of all implicit faults are verified, the coverage results should be a lower bound estimate of what the coverage would be at higher distances. This is because higher distanced cube replacements are




```
.model redund
   .inputs x1 x2 x3 x4
   .outputs f
   .subckt and3 w=x1 x=x2 y=x3 z=t0
   .subckt xor3 w=x1 x=x2 y=x3 z=t1
   .subckt or3  w=t0 x=t1 y=x4 z=f
.end

.model and3
   .inputs w x y
   .outputs z
   .names w x y z
      111 1
.end

.model xor3
   .inputs w x y
   .outputs z
   .names w x y z
     001 1
     010 1
     100 1
     111 1
.end

.model or3
   .inputs w x y
   .outputs z
   .names w x y z
      000 0
.end
```

Figure 5.8: Circuit test case to be executed in customized SIS routines

easier to detect than lower distanced ones (Sect. 3.1.3).

5.3 ATPG Experiments

5.3.1 ATPG Test Case Execution

Consider the circuit shown in Fig. 5.8 with its *blif* circuit description. Fig. 5.9, Fig. 5.10 and Fig. 5.11 are ATPG execution results based on gate, MIGSE module and cube distance error replacement fault models. With only 3-input AND, OR and XOR specified in the *blif* description, design preprocessing routines discussed in Section 5.1.2 are not executed.

5.3.2 Benchmark Results

Table 5.3 compares fault coverage of the proposed method (listed as cube) with two other modeling methods previously proposed in explicit gate replacement errors (listed as gate) and gate-replacement error modeling by MIGSE modules (listed as module). In ATPG, the proposed model shows equal or better

```
sis> read_blif thesis_testcase.blif
sis> repl_atpg -gg
GATE_ALL
node_name: {f}
--->VECTOR:     0010
--->VECTOR:     1110
--->VECTOR:     0000
--->VECTOR:     0000
--->VECTOR:     0000

node_name: t1
--->VECTOR:     1000
--->VECTOR:     1100
--->VECTOR:     0000
--->VECTOR:     0000
--->VECTOR:     0000

node_name: t0
--->VECTOR:     1100
--->VECTOR:     0000
--->VECTOR:     0000
--->VECTOR:     0000


Total simulation time:  0.00s
Fault coverage
(14/15):        93.33%
```

Figure 5.9: ATPG execution output by gate replacements

Table 5.3: ATPG results on MCNC benchmarks

Test case	ATPG								
	Fault List Size			Coverage (%)			Time (s)		
	Cube	Gate	Module	Cube	Gate	Module	Cube	Gate	Module
9symml	15885	895	537	98.16	90.84	82.87	64	24	7
cm138a	615	125	75	100	100	100	0	1	0
decod	750	250	150	100	100	100	0	1	0
f51m	16140	1300	780	98.49	95.77	92.18	44	35	10
i1	2835	145	87	99.79	98.62	97.70	0	1	0
mux	15540	380	228	93.92	95.26	85.09	32	4	2
my_adder	21840	880	528	97.22	85.45	79.17	137	26	6
parity	675	225	135	93.33	80.00	66.67	2	3	1
z4ml	12945	1035	621	98.73	95.85	92.91	25	23	5
alu2	45390	2570	1542	93.96	90.00	81.13	622	64	67
C432	9780	980	588	97.40	90.82	84.69	26	15	10
C499	8010	2030	1218	95.73	83.94	73.23	100	48	43
C880	10695	1565	939	99.01	93.23	88.71	29	21	20
C1355	9570	2550	1530	95.34	83.14	71.90	192	88	85
C1908	17730	1990	1194	97.73	87.04	78.39	166	50	46
C2670	36645	3015	1809	98.19	87.83	80.65	354	97	131
C6288	35280	11760	7056	95.32	85.99	76.64	4556	2327	6020
ripple8	1005	335	201	94.83	84.48	74.13	2	2	0
ripple12	23490	515	309	97.08	84.47	74.11	6	4	1
ripple16	2085	695	417	94.82	84.46	74.10	10	8	2

```
sis> read_blif thesis_testcase.blif
sis> hayes
SAV faults in replacement model: 3
3 total faults
0 faults covered by RTG
S_A_1   NODE: [4]       OUTPUT
Tested
fault simulation covered 0
2 faults remaining
S_A_0   NODE: [5]       OUTPUT
Tested
fault simulation covered 0
1 faults remaining
S_A_1   NODE: [6]       OUTPUT
Tested
0 faults remaining
SAV faults in replacement model: 3
3 total faults
0 faults covered by RTG
S_A_1   NODE: [7]       OUTPUT
Tested
fault simulation covered 0
2 faults remaining
S_A_0   NODE: [8]       OUTPUT
Tested
fault simulation covered 0
1 faults remaining
S_A_1   NODE: [11]      OUTPUT
Tested
0 faults remaining
SAV faults in replacement model: 3
3 total faults
0 faults covered by RTG
S_A_0   NODE: [12]      OUTPUT
Tested
fault simulation covered 0
2 faults remaining
S_A_0   NODE: [14]      OUTPUT
Tested
fault simulation covered 0
1 faults remaining
S_A_1   NODE: [15]      OUTPUT
Redundant
0 faults remaining
----------------------------------------------------
module replacement atpg time:0.00
fault coverage:88.89    (8/9)
```

Figure 5.10: ATPG execution output by MIGSE module replacements

```
sis> read_blif thesis_testcase.blif
sis> repl_atpg -dd
DISTANCE_ALL
node_name: {f}
[distance=1]
[distance=2]
[distance=3]

Detected: 252, Redundant: 3
node_name: t1
[distance=1]
[distance=2]

Detected: 506, Redundant: 4
node_name: t0
[distance=1]
[distance=2]
[distance=3]
[distance=4]
[distance=5]
Detected: 746, Redundant: 19

Total simulation time:  1.00s
Fault coverage
(746/765):      97.52%
```

Figure 5.11: ATPG execution output by cube distance error replacements

coverage over the other two methods with for all benchmark circuits. The gate replacement technique itself shows better coverage for all benchmarks when comparing to the MIGSE module replacement method. Looking at fault list size of the proposed model, the numbers represent the number of actual functional replacements, as opposed to the implicit classes of cube-distance faults checked. Due to the nature of the proposed model, the number of actual replacements is larger resulting in larger fault list sizes.

5.4 Simulation Experiments

5.4.1 Simulation Test Case Execution

Fig. 5.12, Fig. 5.13 and Fig. 5.14 are simulation execution results based on gate, MIGSE module and cube distance error replacement fault models. The test case is based on the circuit shown previously in Fig. 5.8. With only 3-input AND, OR and XOR specified in the *blif* description, design preprocessing routines discussed in Section 5.1.2 are not executed.

5.4.2 Simulation Results with Random Vectors

In the simulation experiments, each replacement is checked with an arbitrarily number of random vectors set to as 25 times the number of primary inputs. The first two columns of Table 5.4 show the proposed minimal cube-distance replacement method. The first of these shows the compact implicit cube-distance fault list size, i.e., the number of distance increments, and the second column shows the actual number

```
sis> read_blif thesis_testcase.blif
sis> repl_sim -gr
GATE
RANDOM
Simulating fault-free circuit
Random Simulation
Simulating faulty circuit
* Total number of nodes: 3
* Each fault is simulated with AT MOST 100 vectors
--------------------------------------------
--->VECTOR:     1001
--->VECTOR:     1001
--->VECTOR:     0000
--->VECTOR:     0000
--->VECTOR:     0000
Detectable: 5 of 5
        Detected: 5, Redundant: 0
        Nodes remaining: 2
--->VECTOR:     0100
--->VECTOR:     1100
--->VECTOR:     0000
--->VECTOR:     0000
--->VECTOR:     0000
Detectable: 5 of 5
        Detected: 10, Redundant: 0
        Nodes remaining: 1
--->VECTOR:     1100
        Running redundancy identification...
--->VECTOR:     0000
--->VECTOR:     0000
--->VECTOR:     0000
Detectable: 5 of 5
        Detected: 15, Redundant: 0
        Nodes remaining: 0

Total simulation time:  0.00s
Fault Coverage
(15/15):        100.00%
Implicit Faults: 0
```

Figure 5.12: Simulation execution output by gate replacements

of functional replacements performed in the experiments, implied by the faults in the first column. In all cases, the implicit cube-distance fault list uses a smaller fault list size than the other two models. Similarly, when looking at fault coverage, cube-distance error replacements show slightly lower coverage compared to gate replacements. This can be explained that most gate replacements are generally higher-distanced cube replacements, which are easier to detect. Also, the redundancy removal routine for gate replacements are run whenever the set of random vectors fail to detect the fault. For the proposed model, the fault redundancy identification is only executed when the set of maximum distance cube replacements fail detection. Despite this, with the exception of the smaller *9symml* circuit, the coverage of the proposed model is comparable to gate replacements. Slight discrepancies exist between simulation and ATPG results for the MIGSE module replacement model, as fault redundancy removal is not implemented for this model.

The duration of each simulation is shown in Table 5.4. On average, cube distance replacement modeling requires longer times, with the exception smaller circuits such as *parity*. The longer average simulation times are because of the number of explicit faults all the cube distance faults imply. In implicit cube distance replacement, there are far more erroneous functional faults to be simulated, even if the number of implicit

```
sis> read_blif thesis_testcase.blif
sis> repl_sim -hr
HAYES
RANDOM
Simulating fault-free circuit
Random Simulation
Simulating faulty circuit
* Total number of nodes: 3
* Each fault is simulated with AT MOST 100 vectors
----------------------------------------------
Simulating Hayes replacement error
Creating S-A-1 faulty node
--->VECTOR:     0100
Creating S-A-0 faulty node
--->VECTOR:     0000
Creating S-A-1 faulty node
--->VECTOR:     1001

        Detected: 3, Redundant: 0
        Nodes remaining: 2
Simulating Hayes replacement error
Creating S-A-1 faulty node
--->VECTOR:     0000
Creating S-A-0 faulty node
--->VECTOR:     0100
Creating S-A-1 faulty node
--->VECTOR:     0000

        Detected: 6, Redundant: 0
        Nodes remaining: 1
Simulating Hayes replacement error
Creating S-A-0 faulty node
--->VECTOR:     1100
Creating S-A-0 faulty node
--->VECTOR:     0000
Creating S-A-1 faulty node

        Detected: 8, Redundant: 1
        Nodes remaining: 0

Total simulation time:  0.00s
Fault Coverage
(8/9):  88.89%
Implicit Faults: 0
```

Figure 5.13: Simulation execution output by MIGSE module replacements

faults is far smaller. For some larger circuits, such as *C499*, *C880*, *C1355*, *C2670* and *C6288*, simulation time is comparable or lower than the other two models in some cases. For the three self-created ripple adder test cases (*ripple8*, *ripple12*, *ripple16*), the cube-distance modeling shows higher coverage and shorter execution time while keeping a low number implicit faults.

5.4.3 Simulation Results with Upper Lattice Layer Vectors

In each simulation routine programmed in SIS for this thesis, the lattice is set to simulate with at most $\lceil log_2(n+1) \rceil - 1$ upper layers. As soon as a vector from the lattice layers detects a fault, the simulator moves onto the next fault in the fault list.

Similar to fault coverage by random vectors in Table 5.4, lattice simulation in Table 5.5 also shows higher coverage with minimal cube distance replacement error modeling in all benchmark circuits. The minimum

```
sis> read_blif thesis_testcase.blif
sis> cub_repl -r
{f} = t0 + t1 + x4
     t0 = x1 x2 x3
     t1 = x1 x2 x3 + x1 x2' x3' + x1' x2 x3' + x1' x2' x3
Simulating faulty circuit
* Total number of nodes: 3
* Each fault is simulated with AT MOST 100 vectors
----------------------------------------------
node_name: {f}
        Distance 1: Detected: 6, Redundant: 2
        Distance 2: Detected: 27, Redundant: 1
        Distance 3: Detected: 56, Redundant: 0
        Total [Detected: 89, Redundant: 3]
        Nodes remaining: 2
node_name: t1
        Distance 1: Detected: 7, Redundant: 1
        Distance 2: Detected: 28, Redundant: 0
        Total [Detected: 180, Redundant: 4]
        Nodes remaining: 1
node_name: t0
        Distance 1: Detected: 4, Redundant: 4
        Distance 2: Detected: 22, Redundant: 6
        Running redundancy identification...
        Running redundancy identification...
        Running redundancy identification...
        Running redundancy identification...
        Distance 3: Detected: 56, Redundant: 0
        Total [Detected: 262, Redundant: 14]
        Nodes remaining: 0
----------------------------------------------

Primary Inputs: 4
Arbitrary max distance: 3
Fault Coverage: (262/276)=      94.93%
Total simulation time:  1.00s
Implicit Faults:        8
```

Figure 5.14: Simulation execution output by cube distance error replacements

coverage is 99.60% for *z4ml*. With gate replacement error, *9symml*, *parity* and the ripple-adders are below 90%. With the module replacement model, only *cm138a*, *decod*, *f51m* and *z4ml* have fault coverages above 90%. Comparing fault coverage numbers with simulation by random vectors, the cube distance replacement error model shows better correlation as compared to the other models. It shows that lattice simulation suits well for cube distance replacement detection, and possibly easier to find.

As Table 5.5 shows, implicit cube distance replacement errors take more time to simulate than the other two models, particularly in *i1* and *mux* benchmarks. For the same reason as random simulation, the number of replacements each implicit fault implies is larger than the other two models. Between the gate and module replacement modeling, the latter generally requires a longer time to simulate, most notably in *f51m*, *i1*, *mux*, *ripple8* and *ripple16*.

Table 5.4: Results of simulation by random vectors

	Simulation (Random)									
	Fault list size				Coverage (%)			Time (s)		
Test case	Cube (Impl)	Cube	Gate	Module	Cube	Gate	Module	Cube	Gate	Module
9symml	442	6796	895	537	85.77	97.54	73.56	5052	907	874
cm138a	25	428	125	75	100	100	100	4	3	3
decod	50	700	250	150	100	100	97.33	7	6	9
f51m	388	7618	1300	780	96.09	98.62	90.90	1589	438	699
i1	44	1186	145	87	97.13	98.62	94.25	106	12	27
mux	155	5744	380	228	93.33	98.95	78.95	1877	169	333
my_adder	384	8704	880	590	93.75	92.73	90.91	4867	1558	836
parity	90	630	225	135	92.86	93.33	66.67	39	54	73
z4ml	326	6096	1035	621	96.06	98.36	88.89	945	270	410
alu2	1162	19442	2570	1542	89.08	97.43	77.11	27244	4255	5257
C432	334	3992	980	588	95.39	98.06	84.35	1987	1301	1514
C499	754	6308	2030	1218	94.10	94.58	72.74	9302	8903	10492
C880	470	6332	1565	939	97.14	97.19	85.41	4966	4328	5156
C1355	977	7764	2550	1530	93.44	87.53	71.37	15603	17490	19452
C1908	773	9394	1990	1194	93.09	95.08	75.88	17209	7999	8622
C2670	1209	17412	2668	1809	90.00	93.18	66.33	416814	603714	201970
C6288	4002	32928	11760	7056	94.98	86.11	76.64	172625	337051	267579
ripple8	116	938	335	201	94.46	84.48	74.13	90	112	124
ripple12	183	1442	515	309	94.45	84.47	74.11	309	390	371
ripple16	247	1946	695	417	94.45	84.46	74.10	740	946	990

Table 5.5: Results of simulation by top lattice layer vectors

	Simulation (Lattice)									
	Fault list size				Coverage (%)			Time (s)		
Test case	Cube (Impl)	Cube	Gate	Module	Cube	Gate	Module	Cube	Gate	Module
9symml	510	6796	895	537	75.16	97.54	43.39	2441	345	217
cm138a	75	428	125	75	60.28	100	32	2	2	1
decod	131	700	250	150	68.71	100	50.67	5	4	2
f51m	676	7618	1300	780	80.78	97.85	55.67	676	407	295
i1	64	1186	145	87	85.75	98.62	70.11	1375	264	220
mux	195	5744	380	228	89.75	98.95	69.30	13265	1111	995
my_adder	384	8704	880	528	93.75	92.73	90.91	373802	137847	52176
parity	90	630	225	135	92.86	93.33	66.67	75	167	129
z4ml	617	6096	1035	621	62.84	98.36	22.71	378	148	71
alu2	1431	19442	2570	1542	77.87	97.43	49.68	1431	3465	2440
C432	330	3992	980	588	95.34	98.06	83.67	116096	98680	104503

CHAPTER 6

IMPLICIT FAULT MODELING OF FPGA ERRORS

With FPGA devices now fabricating under deep submicron manufacturing (DSM) technologies, transistor density continues to increase while wafer sizes are also getting larger. The inherent unreliability of DSM FPGA devices demands more testing methodologies. Errors of FPGA prototyping designs not only originate from manufacturing, but also from environmental factors, such as temperature and radiation [WJR+03][GCW+03]. In mission critical applications such as space and avionics, it is important to realize such problems and be able to reconfigure the circuitry on the fly. Another probable error source comes as design errors introduced by the designers and the CAD tools they use. Testing and verification of the FPGA designs becomes a very challenging problem.

Numerous issues regarding design error modeling are involved in FPGA. With large circuits often needed to be mapped into the very limited space of the FPGA, the process usually incorporates algorithms attempting to optimize the design hardware implementation. Specifically, one of the largest problems comes during the mapping of the function into lookup table (LUT) blocks under the hard-pressed area constraints. As a result, the originally intended function could be inadvertently modified, and therefore needs to be verified afterwards. If simulations are used to check the design correctness, then it is required to specify a fault model that represents LUT mapping errors.

This chapter concentrates on functional error modeling of logic blocks. As discussions will only be on combinational and not sequential logic designs, errors associated with flip flops in the CLB will therefore not be irrelevant. Also interconnect errors and detection techniques such as [MZ02] will not be looked at, although the proposed method can be extended to handle these cases in future work.

An error modeling of LUT by the implicit cube distances is proposed in this chapter. The discussion of a similar design error representation in ASIC (Chapter 3), where netlists are represented by combinational logical gates is extended to FPGAs with LUTs as atomic blocks. With each LUT representing a large spectrum of logic functions, the number of potential errors could be larger than in the case of gate replacement

faults in ASIC. Therefore, the explicit consideration of design errors in FPGAs can become even more difficult than what is already challenging in ASICs. The appealing alternative is to model implicitly the variety of complex Boolean LUT errors, as such an approach results in a more compact fault list. In consequence, a unified test procedure addressing versatile explicit faults can be derived. First, the cube distance error model will be defined, followed by a discussion of the overall testing algorithm including LUT function extraction, fault injection and mapping of cube distance faults to original *s-a-v* verification routines. Finally, fault redundancy identification and fault coverage calculations will be looked at. Experiments are based on several modifications of the Berkeley SIS tool [SSL+92].

6.1 Previous Work

6.1.1 Error Modeling of the Gate

Gate replacement fault models have been looked at in Chapters 2 and 3. These models are compatible with *s-a-v* manufacturing faults, in that methods for their detection can take advantage of existing automatic test pattern generation (ATPG) and simulation algorithms for *s-a-v* faults [BA00]. For example, in [AAH95] and [BH97], *s-a-v* faults are injected into modules that represent erroneous gate replacements. Each gate in the design library has a corresponding fault module. Then satisfiability- (SAT-) based ATPG algorithms are used for the fault detection [BH97][RZ01]. The model in Chapter 3 categorizes all potential functional errors based on their cube distances from the error free function. Due to the fault dominance, only faults with the smallest error distances need to be checked. The deterministic method used in this case is also based on the SAT-based ATPG.

6.1.2 Error Modeling of the FPGA Faults

In [RPFZ00] authors use the traditional single *s-a-v* model to represent faults in the MUX and LUT. Authors in [MSM98] mimic *s-a-v* faults at output of the LUT by modifying the implemented functions. Either all LUT input combinations are mapped to 0 (for *s-a-0*) or are mapped to 1 (for *s-a-1*). In [PTS03], a variation of the *s-a-v* model named combinational stuck-at (CSA) is proposed. Here, errors are injected into the LUT, based on the input combination. Unlike [MSM98], the stuck values in this case are conditional. This solution, similar to the one presented in this thesis, modify the functionality of a LUT as opposed to modeling a mere stuck value at a LUT output wire.

6.1.3 Online Testing via Partial Reconfiguration

Authors in [SMSP97] and [PTS03] present ways of testing a device in real time, i.e., while it remains under operation. The algorithm allocates testing cells (column of logic blocks). Columns of logic cells are then moved or copied temporarily to a column adjacent to the test cells for checks and restored afterwards. Although basic unit is a column, the same concept may be extended to real-time testing of LUTs. The functional fault model proposed in this thesis represents an erroneous Boolean function stored in a LUT. Simulations under similar fault models often require FPGA to be partially reconfigurable [Xil04][SMSP97]. In this thesis a testing scenario is presented, where a design under test is simulated in software instead of mapping it first into hardware. However, partial reconfiguration, which allows altering the contents of specific cells in a device without touching remaining cells, is applicable. For example, the Xilinx Virtex-family permits on maintaining operation while the LUT or column is reconfigured. In [MUPRS05], authors describe specifics of using partial reconfiguration by direct manipulation of a configuration bitstream given a precise location of the target component or cell to be modified. The JBits API for Xilinx devices in [GLS99] also allow ways of manipulating the bitstream directly.

6.2 Cube Distance Replacement Error Modeling

Each LUT can be viewed as an equivalence of a complex multi-input (single/multi)-output hardware ASIC gate. This can be further simplified, with the LUT regarded as an element implementing a Boolean equation in this thesis. It is then natural to implicitly represent potential LUT mapping errors in terms of differences between the intended and resulting erroneous functional implementation. One way of describing such differences is in terms of the number of cubes to distinguish the two Boolean functions.

Example 6.2.1. *Consider a circuit illustrated in Fig. 6.1. The logic is mapped into two 3-input LUT (LUT3) whose outputs are controlled by input signal* w. *Assume that the top LUT3 whose fault-free function is:* $c = LUT(x,y,z) = (0,1,1,0,0,0,1,1)^T = x'y'z + yz' + xy$. *A possible error cube replacement of distance-1 is* $c_1 = (0,1,1,0,0,0,0,1)^T = x'y'z + x'yz' + xyz$. *The error is excited when* $(x,y,z) = (1,1,0)$ *as shown in its truth table in Fig. 6.1.*

In the above example (6.1), neither the original function nor its replacement constitutes a regular Boolean gate function, such as AND, OR and XOR. With larger LUT sizes, at typically four or five, the range of functions is even wider. Thus considering every possible case of explicit LUT replacement error would be highly impractical.

The implicit model categorizes all possible functional faults of the cell by their appropriate Hamming distance from the original function. It is also proven in Chapter 3 that if all replacement faults at distance-i

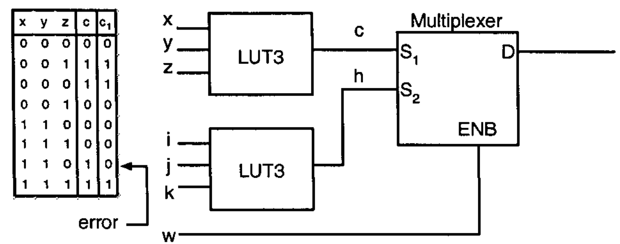


Figure 6.1: Basic mapped circuit with error in one of its LUT

(C_i) are detectable, then it is implied that all faults at distances larger than i will also be detectable. Therefore the detection of minimal cube distances errors automatically guarantees the detection of all LUT functional replacements with larger distances from the original fault-free function.

6.3 Fault Insertion

Simulation based methods are unique in the case of FPGA testing, not only due to the unique nature of the hardware structure but also because of its programmability, which directly affects fault insertion specifics during testing. As the proposed fault model represents the error as functional errors in the LUT, testing of the device would require repeated configuration of the LUT. Unfortunately programming the device is generally a large bottleneck in time. Numerous Xilinx devices support partial reconfiguration [Xil04], which can be used to solely modify the LUT function without changing the other LUT components in the circuit. Furthermore, the rest of the device would remain in operation, which is particularly useful for simulation purposes. Another feature of the dynamic reconfiguration of the LUT is that the bitstream used to reprogram the device depends only on the LUT difference, and thus speeds up the implementation. However with the number of candidate cube-distance replacements (Chapter 3), the overhead required to generate a difference bitstream and to reprogram still remains costly.

Without a safe methodology of manipulating a bitstream, without the risk of tampering the design, performing simulation experiments on FPGA would be meaningless. However, for the purpose of showing the test quality of the proposed error modeling of the LUT, software simulations can be used. The proposed algorithm is implemented in the Berkeley SIS tool platform [SSL+92]. If however, the ease of the FPGA reprogramming can be guaranteed to be efficient and easily executable, then the emulations of the error functions on FPGA would be the final goal.

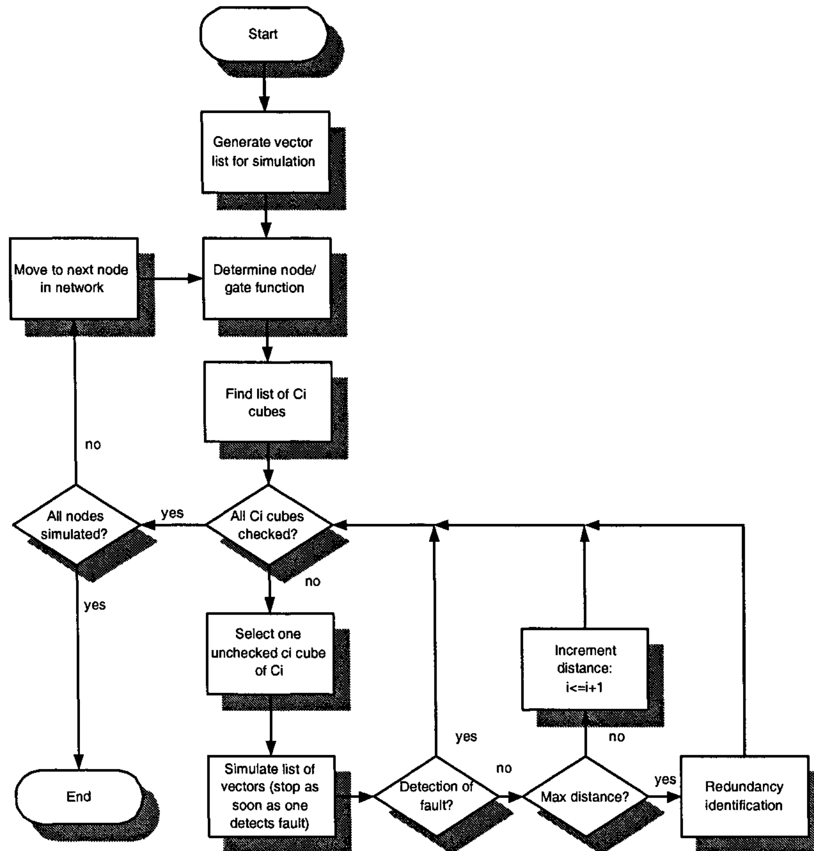


Figure 6.2: Simulation flow of LUT verification with cube distance errors

6.4 Functional Simulation Experiments

The cube-distance fault simulator is implemented in SIS. Instead of using concrete designs for test purposes random functions with primary inputs of 8, 12, 16 and 24 are generated. The function then undergoes LUT mapping to Xilinx-like LUT architectures [SSL+92]. After the initial mapping, the netlist is ready for simulation. The circuit is traversed, where each node's function is extracted from its LUT data-structure. From this, cube distance errors could be generated, injected and simulated.

Fig. 6.2 presents a basic overview of the simulation algorithm within the modified SIS platform. For each extracted LUT function, the routine starts with the set of cube distance errors of the smallest distance, i.e., one (C_1). If all candidate replacements could not be detected, the algorithm proceeds with the higher cube distance errors. All errors are subjected to random vector simulations, and as soon as one vector is capable of detecting the error, the verification routine moves onto the next error replacement. However, if at least one LUT replacement c_i at distance i is undetected, then all errors at the level $i+1$ (C_{i+1}) spawned from this c_1 are considered. In this thesis an arbitrary maximum distance of 3 is set for all the experiments. Therefore, a

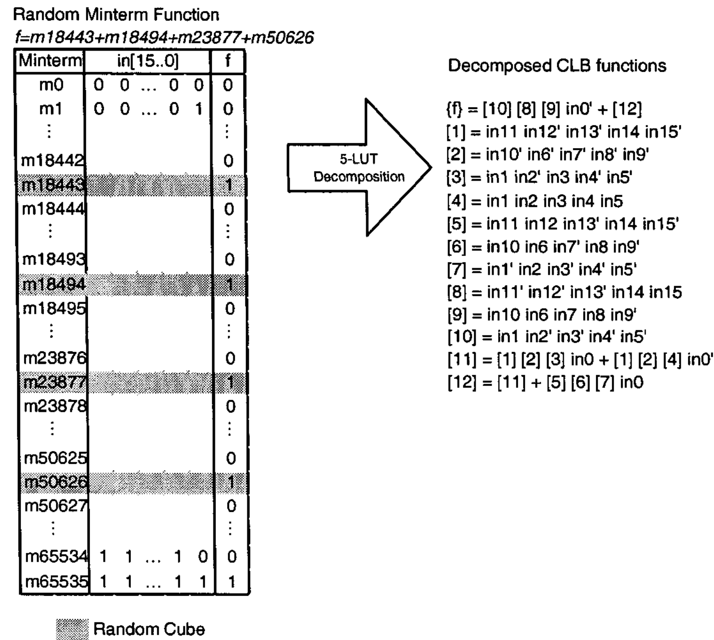


Figure 6.3: 5-LUT decomposition of random combinational network

set of undetectable LUT replacements at this distance, i.e., C_3 will undergo a deterministic fault redundancy identification to confirm their detectability.

6.4.1 Random Combinational Circuit Generation

Experiments involve checking of numerous random combinational circuits. Fig. 6.3 shows a random combinational network generated by random minterms. A 16:1 function is created based a random selection of four minterms (out of a possible maximum of 2^{16}). The example shows one generated given by the relation $f(inputs) = \sum m(18443, 18494, 23844, 50626)$. After the network is created, it undergoes synthesis and decomposition into 5-LUT components within SIS. This is done by built-in Xilinx *"xl"* (*"xl_ao"*, *"xl_partition -tm -n5"*) functions. A sweep is then done to remove extraneous inverters in series with the *sweep* command. The resulting network from the example shown in Fig. 6.3 is comprised of interconnections between thirteen 5-LUT components – nodes 1 to 12 and node f.

6.4.2 Extraction of the LUT Function

Determining the cube-distance errors requires finding first the complete ON-sets of the Boolean function of a LUT node, i.e. combinations of inputs that would give an output of 1. This can be done through data structures defined in the *espresso* library. In most cases, these ON-sets also incorporate don't care (DC)

Inputs			c	c_aux	c1	c_aux	c1	c_aux	c1	...	c_aux	c1
0	0	0	0	1	1	0	0	0	0	...	0	0
0	0	1	1	0	1	1	0	0	1	...	0	1
0	1	0	1	0	1	0	1	1	0	...	0	1
0	1	1	0	0	0	0	0	0	0	...	0	0
1	0	0	0	0	0	0	0	0	0	...	0	0
1	0	1	0	0	0	0	0	0	0	...	0	0
1	1	0	1	0	1	0	1	0	1	...	0	1
1	1	1	1	0	1	0	1	0	1	...	1	0

Figure 6.4: Generation of 3-LUT Cube Distance-1 Error Functions

conditions. For each ON-set, inputs with DCs are expanded into a 0-case and a 1-case. The verbose ON-set information is then sorted by the minterm order to mimic a truth table. With the complete table, one may generate the cube distance errors.

6.4.3 Generation of LUT Cube-Distance Errors

Generation of LUT cube-distance errors involve determining the cube difference auxiliary function c_{aux}, which is defined as the Boolean difference between the LUT function c and its distance-i replacement c_i (Eqn. 6.1), as previously defined by Eqn. 4.5:

$$c_{aux} = c \oplus c_i \tag{6.1}$$

An input vector which sets c_{aux} to 1 is one that detects fault c_i [BA00][ABF94]. The set of all auxiliary functions, C_{aux} represents all functions that have i cubes in its ON-set. In other words, all c_{aux} of C_{aux} have a weight (number of 1s) of i. The c_{aux} functions also have no relationship to the LUT Boolean function.

The inverse in Eqn. 6.1 can be used to create the list of candidate replacement errors from the original LUT function:

$$c_i = c \oplus c_{aux} \tag{6.2}$$

With the LUT function c extracted, all needs to be done is generating the list of C_{aux} functions and XOR-ing them with the fault-free function c.

Example 6.4.1. *Assume that an original function c implemented in a given LUT is defined as:* $(0,1,1,0,0,0,0,1)^T$. *Let* C_{aux} *be the complete set of* c_{aux} *at distance 1. The entire set of* c_{aux} *is therefore:*

$C_{aux} = \{(1,0,0,0,0,0,0,0)^T, (0,1,0,0,0,0,0,0)^T, (0,0,1,0,0,0,0,0)^T,$
$(0,0,0,1,0,0,0,0)^T, (0,0,0,0,1,0,0,0)^T, (0,0,0,0,0,1,0,0)^T,$
$(0,0,0,0,0,0,1,0)^T, (0,0,0,0,0,0,0,1)^T\}.$

Fig. 6.4 shows c_1 *error LUT functions derived from the set of* C_{aux}. *For the figure's simplicity, only the first 3 and the final* c_{aux} *functions are shown. By Eqn. 6.2, each* c_i *can be generated by toggling c at the locations where* c_{aux} *is marked true. The entire* C_1 *set of* c_1 *is thus:*

$C_1 = \{(1,1,1,0,0,0,1,1)^T, (0,0,1,0,0,0,1,1)^T, (0,1,0,0,0,0,1,1)^T,$
$(0,1,1,1,0,0,1,1)^T, (0,1,1,0,1,0,1,1)^T, (0,1,1,0,0,1,1,1)^T,$
$(0,1,1,0,0,0,0,1)^T, (0,1,1,0,0,0,1,0)^T\}.$

6.4.4 Mapping of Cube-distance Faults to S-A-V Faults

To allow the SAT-based algorithm to constraint the input conditions for fault activation, unary clauses are appended to the good, faulty, active and fault site expressions that describe the circuit (Sect. 4.2). Each cube difference is mapped to one *s-a-v* fault with input conditions determined by SAT clauses. For faults with larger cube distances i, the fault is mapped to i *s-a-v* faults with the corresponding unary SAT clauses.

Example 6.4.2. *Assume that a fault-free LUT function c from Ex. 6.4.1 (Fig. 6.4) is replaced by one of its* C_1 *element:* $c_1 = (1,1,1,0,0,0,1,1)^T$. *If input variables are defined as* x_0, x_1 *and* x_2 *the additional constraining SAT clauses will be:* $(\overline{x_0})$, $(\overline{x_1})$ *and* $(\overline{x_2})$.

6.4.5 Fault Redundancy Identification

A fault c_i is undetectable if there is no Boolean difference between itself and fault-free c, by Eqn. 6.1. Hence a cube-distance fault c is redundant, when there is no such an input setting which would set c_{aux}. In practice, redundant fault identification is executed when simulations are unable to detect the fault at the maximum distance (of 3, in this thesis).

In the SIS experiments, the outputs of both nodes: the original and an inserted erroneous one are XOR-ed representing a c_{aux} node, by Eqn. 6.1. In order to force the ATPG-SAT solver to look for the conditions satisfying the above Boolean difference, and to generate a test vector detecting a cube-distance fault c, a *s-a-0* fault is injected at the XOR output (Fig. 6.5). After running the modified ATPG-SAT solver, the program reverts any changes to the network by removing the auxiliary XOR gate, and the duplicated original node is backed up in the memory.

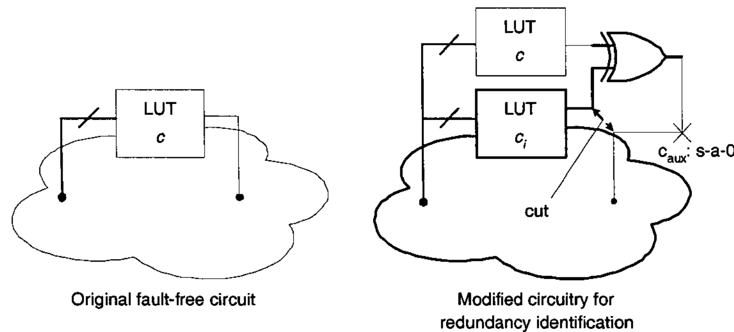


Figure 6.5: Circuit Modified for Redundancy Identification

```
1  sis> gen_rand_function -i16
2  sis> xl_ao
3  sis> xl_partition -tm -n5
4  sis> cube_repl -1
5  Inputs: 16
6  Minterms: 5
7  Minterms: m_1981 m_29307 m_39790 m_46197 m_48406
8  Generated Function
9       {f} = in0 in1 in10' in11' in2' in3 in4 in5 in6 in7' in8' in9 + in0 in1'
            in10 in11' in2 in3 in4 in5 in6' in7 in8 in9 + in0 in1' in10 in11' in2
            in3' in4 in5 in6 in7' in8' in9' + in0' in1 in10 in11 in2 in3' in4 in5'
            in6' in7' in8 in9' + in0' in1 in10' in11 in2 in3 in4' in5 in6 in7' in8
            in9
10
11 {f} = [10] [9] in0' in1 + [14]
12      [1] = in10' in11' in7' in8' in9
13      [2] = in2' in3 in4 in5 in6
14      [3] = in10 in11' in7 in8 in9
15      [4] = in2 in3 in4 in5 in6'
16      [5] = in10 in11' in7' in8' in9'
17      [6] = in2 in3' in4 in5 in6
18      [7] = in10' in11 in7' in8 in9
19      [8] = in2 in3 in4' in5 in6
20      [9] = in10 in11 in7' in8 in9'
21      [10] = in2 in3' in4 in5' in6'
22      [11] = [1] [2] in0 in1
23      [12] = [11] + [3] [4] in0 in1'
24      [13] = [12] + [5] [6] in0 in1'
25      [14] = [13] + [7] [8] in0' in1
```

Figure 6.6: Random network generation and LUT-5 mapping

A primary difference in redundancy identification is with the larger number of inputs in the LUT over the average of the 2- and 3-input gate. As a result, performing exact redundancy check yield to longer run times, because clauses derived by the larger input logic modules have larger number of terms.

6.4.6 Simulation Execution Output

A randomly generated combinational circuit of 16 primary inputs is synthesized to LUT of size 5 (Fig. 6.6). Simulation execution is shown in the Fig. 6.7.

```
1    num_vectors to generate is 79
2    LEVEL: 0
3    LEVEL: 1
4    LEVEL: 2
5    Number of inputs = 12
6    Number of lattice layers = 3
7    Simulating faulty circuit
8    * Total number of nodes: 15
9    * Each fault is simulated with AT MOST 79 vectors
10   ---------------------------------------------
11   node_name: {f}
12           Distance 1: Detected: 5, Redundant: 27
13           Distance 2: Detected: 145, Redundant: 351
14           Distance 3: Detected: 4674, Redundant: 286
15           Total [Detected: 4824, Redundant: 664]
16           Nodes remaining: 14
17   node_name: [14]
18           Distance 1: Detected: 8, Redundant: 24
19           Distance 2: Detected: 220, Redundant: 276
20           Distance 3: Detected: 4596, Redundant: 364
21           Total [Detected: 9648, Redundant: 1328]
22           Nodes remaining: 13
23   node_name: [13]
24           Distance 1: Detected: 7, Redundant: 25
25           Distance 2: Detected: 196, Redundant: 300
26           Distance 3: Detected: 4596, Redundant: 364
27           Total [Detected: 14447, Redundant: 2017]
28           Nodes remaining: 12
29   node_name: [12]
30           Distance 1: Detected: 11, Redundant: 21
31           Distance 2: Detected: 286, Redundant: 210
32           Distance 3: Detected: 4505, Redundant: 455
33           Total [Detected: 19249, Redundant: 2703]
34           Nodes remaining: 11
35   node_name: [11]
36           Distance 1: Detected: 7, Redundant: 9
37           Distance 2: Detected: 84, Redundant: 36
38           Distance 3: Detected: 560, Redundant: 0
39           Total [Detected: 19900, Redundant: 2748]
40           Nodes remaining: 10
41   node_name: [2]
42           Distance 1: Detected: 0, Redundant: 32
43           Distance 2: Detected: 0, Redundant: 496
44           Distance 3: Detected: 4960, Redundant: 0
45           Total [Detected: 24860, Redundant: 3276]
46           Nodes remaining: 9
47   node_name: [1]
48           Distance 1: Detected: 6, Redundant: 26
49           Distance 2: Detected: 171, Redundant: 325
50           Distance 3: Detected: 4960, Redundant: 0
51           Total [Detected: 29997, Redundant: 3627]
52           Nodes remaining: 8
53   node_name: [4]
54           Distance 1: Detected: 1, Redundant: 31
55           Distance 2: Detected: 31, Redundant: 465
56           Distance 3: Detected: 4960, Redundant: 0
57           Total [Detected: 34989, Redundant: 4123]
58           Nodes remaining: 7
59   node_name: [3]
60           Distance 1: Detected: 1, Redundant: 31
61           Distance 2: Detected: 31, Redundant: 465
62           Distance 3: Detected: 4960, Redundant: 0
63           Total [Detected: 39981, Redundant: 4619]
64           Nodes remaining: 6
65   node_name: [6]
66           Distance 1: Detected: 0, Redundant: 32
67           Distance 2: Detected: 0, Redundant: 496
68           Distance 3: Detected: 4960, Redundant: 0
69           Total [Detected: 44941, Redundant: 5147]
70           Nodes remaining: 5
71   node_name: [5]
72           Distance 1: Detected: 1, Redundant: 31
73           Distance 2: Detected: 31, Redundant: 465
74           Distance 3: Detected: 4960, Redundant: 0
75           Total [Detected: 49933, Redundant: 5643]
76           Nodes remaining: 4
77   node_name: [8]
78           Distance 1: Detected: 0, Redundant: 32
79           Distance 2: Detected: 0, Redundant: 496
80           Distance 3: Detected: 4960, Redundant: 0
81           Total [Detected: 54893, Redundant: 6171]
82           Nodes remaining: 3
83   node_name: [7]
84           Distance 1: Detected: 1, Redundant: 31
85           Distance 2: Detected: 31, Redundant: 465
86           Distance 3: Detected: 4960, Redundant: 0
87           Total [Detected: 59885, Redundant: 6667]
88           Nodes remaining: 2
89   node_name: [10]
90           Distance 1: Detected: 0, Redundant: 32
91           Distance 2: Detected: 0, Redundant: 496
92           Distance 3: Detected: 4960, Redundant: 0
93           Total [Detected: 64845, Redundant: 7195]
94           Nodes remaining: 1
95   node_name: [9]
96           Distance 1: Detected: 0, Redundant: 32
97           Distance 2: Detected: 0, Redundant: 496
98           Distance 3: Detected: 4960, Redundant: 0
99           Total [Detected: 69805, Redundant: 7723]
100          Nodes remaining: 0
101  ---------------------------------------------
102
103  Primary Inputs: 12
104  Arbitrary max distance: 3
105  Fault Coverage: (69805/77528)=  90.04%
106  Total simulation time:  26659.00s
107  Implicit Faults:        45
```

Figure 6.7: Output of LUT simulation by lattice vectors

Table 6.1: FPGA functional simulation results by random vectors

Primary inputs in circuit	LUT size	# of LUTs	Fault list size (implicit)	# of Configurations	Fault coverage (%)
8	3	30	79	2682	94.22
	4	20	60	13316	94.51
	5	13	39	66552	97.51
	6	11	32	437454	92.48
12	3	30	136	4416	84.31
	4	20	108	25056	87.46
	5	13	45	77528	90.62
	6	11	36	486672	91.95
16	3	84	252	7728	76.85
	4	18	54	93992	89.91
	5	20	60	104968	89.74
	6	16	48	661648	96.46
24	3	156	468	14274	63.89
	4	119	357	82220	81.99
	5	118	140	247656	88.11
	6	104	80	267766	90.55
32	3	256	768	23474	63.27

Table 6.2: FPGA functional simulation results by lattice vectors

Primary inputs in circuit	LUT size	# of LUTs	Fault list size (implicit)	# of Configurations	Simulation time (s)	Fault coverage (%)
8	3	25	74	2300	186	69.91
	4	18	54	11846	726	85.17
	5	10	30	49484	1734	94.43
	6	9	27	393696	9774	88.79
12	3	47	141	4246	1125	68.75
	4	36	108	25056	6021	82.16
16	3	77	231	7084	35883	65.46
	4	57	171	39672	131459	82.09
	5	52	156	280584	968211	89.74
24	3	154	462	14090	507210	62.32
	4	119	357	82220	1995829	81.30

6.4.7 Functional Simulation Results

Simulation results based on functional simulation within SIS are shown in Table 6.1. The actual number of LUT functional errors is the number of configurations that would be needed. The size of the LUT also directly impacts the number of functional faults. Larger randomly generated circuits mapped to smaller LUT sizes also tend to have smaller coverage. Looking at the number of implicit cube distance faults, which equal to total number of distance increments in the algorithm, the numbers are generally small, and within a small multiple of the number of LUTs in the circuit. Majority of the fault coverage percentages are above 80%, with the exception of 16-, 24-, and 32-input random circuits mapped to LUT sizes of 3.

Simulation results using top layers of the lattice are shown in Table 6.2. In all cases, the lattice simulation the fault coverages are lower with lattice vectors, with higher execution time.

6.5 FPGA Functional Testing

With the repetitiveness of changing the cube of the LUT function, the time used to program the FPGA device needs to be short. A feature known as partial reconfigurability [Xil04] is the ability to re-program a module or LUT while the rest of the circuit maintains its operation. It enables reconfiguring hardware without downloading the entire bitstream as long as the one that is used to implement the original design. The advantage is that it saves time in the device loading overhead. This section proposes a methodology and issues of functional testing of cube-distance errors based on the software emulation discussed in Sect. 6.4.

From [Xil04], there are two methods of pursuing partial reconfiguration of the FPGA device – module based or difference-based. The module based technique requires knowledge of all the candidate reconfigurations and proves to more complicated than the difference-based counterpart. The difference-based method looks at the changes necessary to the bitstream used for a configuration and generates another marking only the differences. Because of the smaller in size in this new difference bitstream, reconfiguration time targeting a specific LUT specified by this model is considerably faster.

6.5.1 Tool Flow

A test flow shown in Fig. 6.8 is repeated throughout, with the LUT function modified during each iteration. To accommodate partial reconfiguration, each time a LUT is changed during functional simulation in SIS, as depicted by the top loop in the figure, a new netlist with the new error is exported in the equation (*eqn*) format. After completion of functional simulations, each exported *eqn* circuit, along with a VHDL wrapper on the circuit, are processed by Synopsys Design Compiler. The Synopsys tool inserts appropriate input and output buffer to the netlist, and assigns the appropriate user defined pin mappings on the development board. A synthesized *edif* netlist is then exported and used by Xilinx tools for implementation via generating a bitstream (*bit*) file. The place and route is purely automated by the Xilinx design tools.

6.5.2 Changing of LUT Equations

The Xilinx *bitgen* utility is used to compare the new netlist with the one used to program the FPGA at the present state. With the utility, the difference bitstream is then generated, which could then be loaded onto the FPGA to modify only the LUT equation.

The proposed tool flow illustrated in Fig. 6.8 poses several issues. Recalling the functional simulation results from Sect. 6.4.7, that the number of necessary configurations is large. The first problem is the large bottleneck involved in re-synthesizing and re-implementing the all the error cases. An option would be to implement an *edif* netlist exporter such that it would bypass *dc_shell*. The problem with this is that the cell libraries available in *dc_shell* are unavailable under SIS. Thus implementing an *edif* exporter is not feasible, as

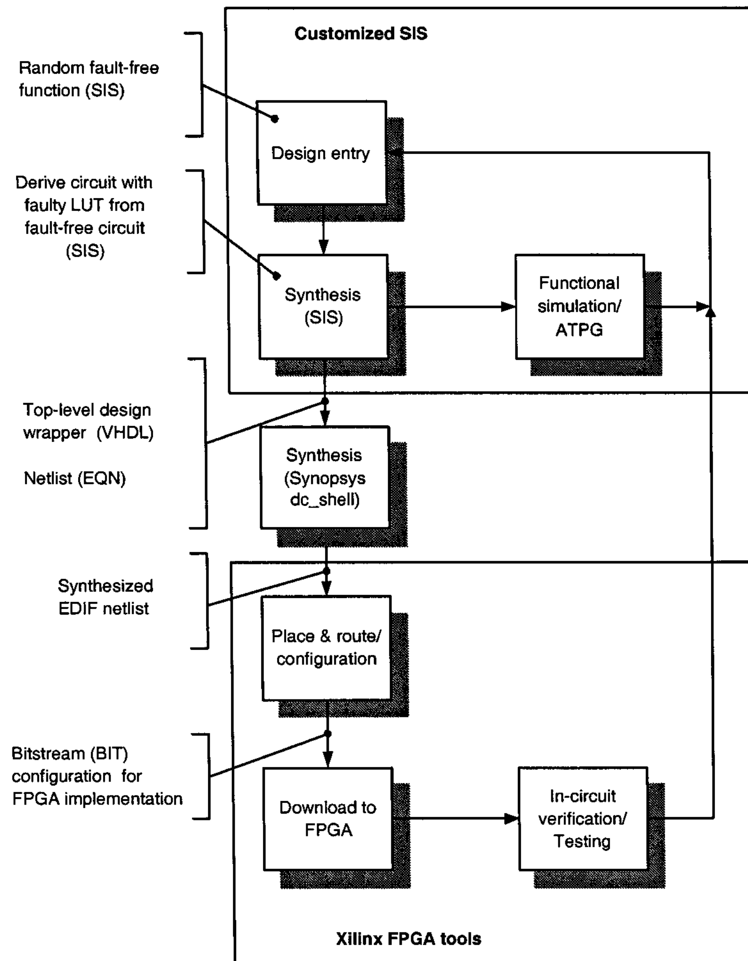


Figure 6.8: FPGA design flow

the exported circuit will contain no information on the LUT mapping. Bypassing the commercial tool would not resolve the issue. Furthermore, SIS primarily exports equations as opposed to netlist formats bounded to cells. The equation format is considered to behavioural with the commercial tools, and any slight change at this hierarchy is likely to reflect more changes than the sole single LUT. The final problem is related to the methodology described in [Xil04], where manipulating LUT requires interaction with the FPGA Editor GUI tool before generating a difference bitstream. This procedure can become tedious considering the high number of cube distance replacements and the repetitiveness required. Without a method of automating the LUT functional tweaking, the quality of the test is potentially bounded by more undesirable human errors. The solution to these problems is to have a program or script is capable of systematically modify the bitstream information. In other words, it needs to be able to replace the two-part "FPGA editor to *bitgen*" process. The test algorithm needs to generate an initial fault-free bitstream via the design flow, and to use JBits API to

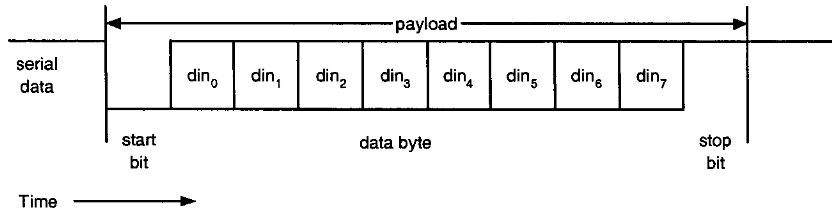


Figure 6.9: Payload of the 8N1 UART convention

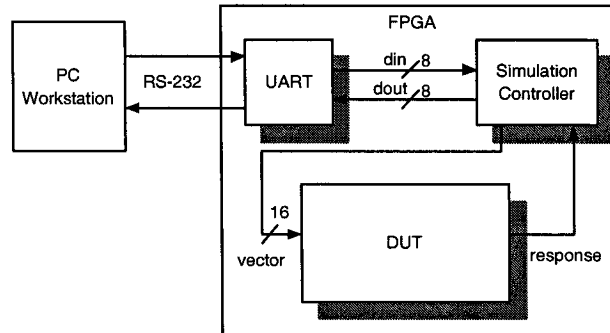


Figure 6.10: Workstation-FPGA communication via RS-232 for testing

directly manipulate a single bistream. With this method, the SIS circuit is only exported and synthesized with commercial tools once, reducing a large part of the original bottleneck. Although details of JBits is beyond the scope of this thesis, the interested reader may refer to [GLS99] and [MUPRS05] for detailed techniques involving direct bitstream manipulation.

6.5.3 Test Circuitry and DUT on FPGA

Vectors such as top-lattice layers may be generated by on-chip logic for simulation in testing, but the module would use up a large number of logic cells, which are generally scarce in FPGA with designs occupying majority of the cells. A 16-input DUT requires $log_2(16) = 4$ layers of the lattice. The test vector generator has been designed and synthesized to the Xilinx Virtex Pro^{TM}II technology, using 2849 nets and 2808 cells.

Another common technique is by generating the vectors off-chip by the workstation. The data is transmitted through the serial port, using the RS-232 protocol to the FPGA and back to the PC using the same protocol. Transmission and reception of data at the FPGA is managed by an Universal Asynchronous Receiver Transmitter (UART) module. The transmission convention in the thesis uses the standard 8N1, which comprises of 1 start bit, 8 data bits, and 1 stop bit (Fig. 6.9). The N in 8N1 denotes that no parity bit is used. The less significant data bits are always transmitted before the more significant ones.

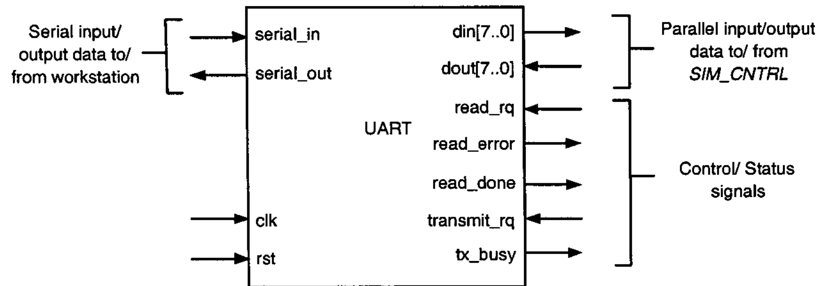


Figure 6.11: UART module

The FPGA receives serial data at the UART receiver and converts it into parallel (Fig. 6.10). Additional logic is then designed to process this parallel data for simulating the DUT. The circuit response is transmitted back through the serial port to the workstation.

6.5.3.1 UART Interface on FPGA

UART cores are available under the Xilinx's application note [Cha] and are also available as part of its embedded microprocessor cores, such as PicoBlaze™and MicroBlaze™. As the cores are heavily depended on Xilinx libraries, a basic UART module is implemented from scratch. To keep the module at a minimal area, the modules do not have any FIFO buffering capabilities. Also, unlike the implementations in serial modems, there is no interpretation of any read data as a control characters, as input data is only interpreted as vector data in simulation.

Fig. 6.11 shows the UART module with its ports. Ports *serial_in* and *serial_out* are data input and outputs on the RS-232 serial line. Other lines are used to interface between the UART module with the simulation controller used to test the DUT. Ports *read_rq* and *transmit_rq* requests the UART module to read and transmit data. Lines *din* and *dout* are parallel data presentation of data of *serial_in* and *serial_out*, respectively. Status registers in *read_error*, *read_done*, *tx_busy* marks whether the read data is invalid, has been completed, or that serial transmission is taking place. The UART interface consists of a clock frequency division unit, a receiving unit and a transmission unit.

6.5.3.2 Clock division - Data synchronization

Serial data reception and transmission rate is at the standard 9600bps, and it is the responsibility of the FPGA to scale down a clock rate to synchronize to this speed. For the 50MHz clock used for the FPGA, if each received bit is sampled once, then the clock would need to be divided (slowed down) 50Mhz/9.6kHz = 520.83 times. In reality, data could get out of synchronization after a while, so each bit of data should be sampled more times, and close to the middle of the received bit for better accuracy. Each data would be

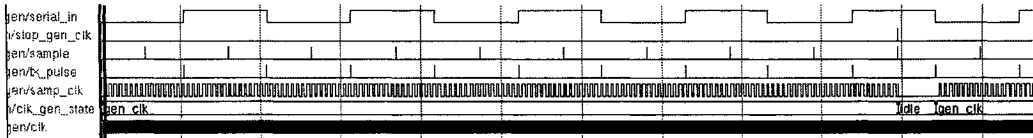


Figure 6.12: UART clock divider waveform

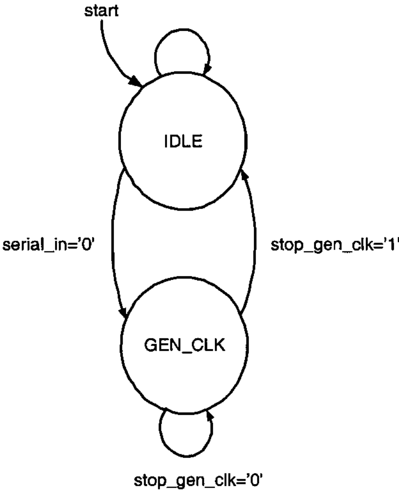


Figure 6.13: UART clock divider state diagram

sampled 16 times, where the 8th sample would be interpreted as the received data. The sampling clock rate is therefore 9.6kHz x 16 = 153.6 kHz. The clock frequency scaling factor from the system clock is then: 50Mhz/153.6kHz = 325.521 $\approx$ 326 times. Frequency divisions are implemented by counters based on the system clock. The waveform in Fig. 6.12 demonstrates the multiple clock domains and samples generated by the component.

With streams of received data entering the serial line at relatively random times, the clock division unit also needs to resynchronize the sampling clock with each new set of input stream. A finite-state machine is used to manage this with the two states: *IDLE* and *GENERATE* (Fig. 6.13). During IDLE, the block waits for a read request (*read_rq*) pulse and then the serial input line is monitored for '0' , signifying the start bit of the *8N1* convention. As soon as the bit is detected, sampling clock signals are generated in the *GENERATE* state. As soon as the receiver unit finishes receiving data, it triggers a *stop_gen_clk* signal to stop generating clock sampling. The process repeats itself each time a new stream of inputs enters the serial data, giving the sampling clock a chance to synchronize with the new input data. For the transmission clock, there is no need to re-adjust the clock as long as transmission is synchronization to the slower clock at the baud rate.

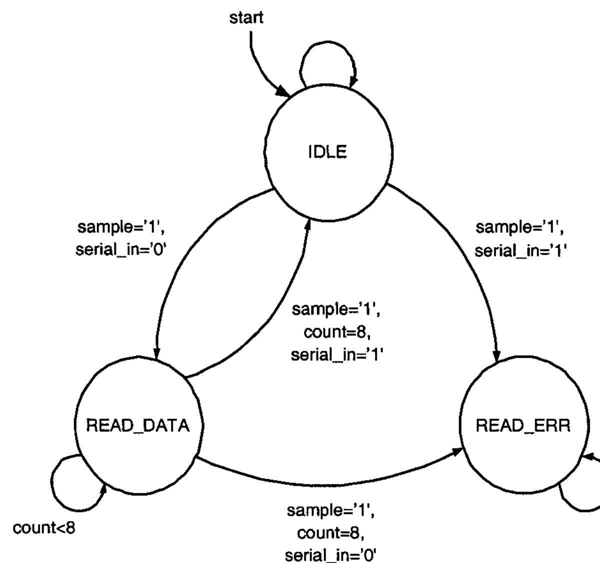


Figure 6.14: UART receiver state diagram

6.5.3.3 UART Receiver

The core function of the UART receiving unit is a serial to parallel conversion by sampling the input serial data stream. From the clock generating block pulse samples are generated for the receiving block to sample data in the middle of each incoming serial bit. Knowing that the samples are always synchronized to every input stream, the receiver is handled by 3 states: *IDLE*, *RD_DATA* and *RD_ERR* (Fig. 6.14).

During *IDLE*, the receiver waits for the sample pulse to be high and checks to see if the serial line is set low. If it is, the state moves onto *RD_DATA* when upcoming data bits are read. Otherwise, it moves into the error state *RD_ERR*, knowing that the start bit can never be high by convention. In *RD_DATA*, each of the next 8 bits are sampled and shifted in a register. The bit after this is expected to be the stop bit and *RD_DATA* is expected to be high. If it is true, the block latches the shift register into the parallel output din, sends a pulse on *read_done* and returns to *IDLE* to wait for the next input stream. Otherwise, it enters the *RD_ERR* state, as the stop bit can never be low. All data processing is synchronized to the system clock, with the exception of sampling of input data. The waveform in Fig. 6.15 shows two error-free input serial sequences of $0, 1, 0, 1, 0, 1, 0, 1, 0, 1$ and $0, 1, 1, 1, 1, 0, 0, 0, 0, 1$. The receiver performs two successful reads and converts the data into parallel at *din[7..0]* as 01010101 and 00001111.

6.5.3.4 UART Transmitter

The UART transmitter is a parallel serial conversion of data from *dout*. This block is managed by three states: *IDLE*, *LOAD* and *SHIFT* (Fig. 6.16). During *IDLE*, the transmitter waits for a request pulse on

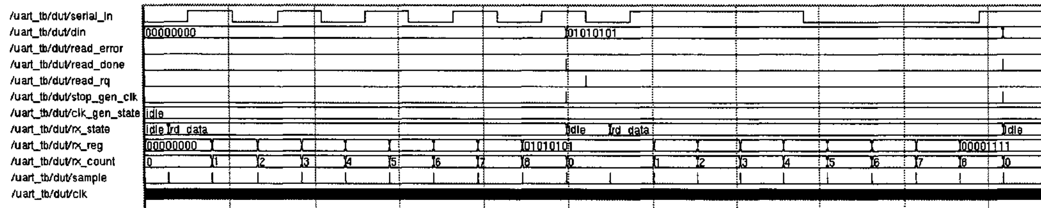


Figure 6.15: UART receiver simulation waveform

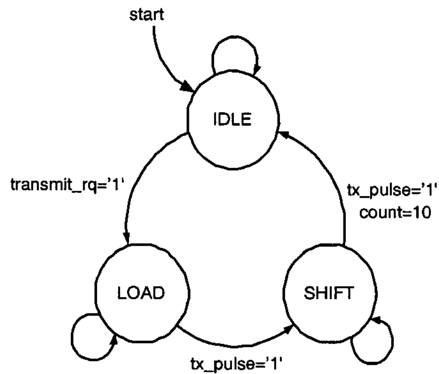


Figure 6.16: UART transmitter

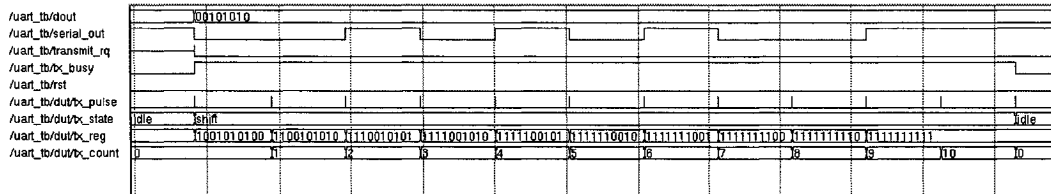


Figure 6.17: UART transmitter simulation waveform

transmit_rq and enters the *LOAD* state. It is assumed that the data is already present at *dout* at this time. During *LOAD*, a start bit of '0' is loaded into a shift register, along with the data byte and the stop bit of '1'. It then moves onto the SHIFT state when the data is ready to be shifted out on each transmission pulse at the baud rate. Throughout *LOAD* and *SHIFT*, *tx_busy* is set true to prevent other blocks of requesting new data to be sent out, as it is not ready. After transmission is complete, it returns to the *IDLE* state when *tx_busy* is set low until a new transmission request. In Fig. 6.17, *dout* is set to 00101010 and shifted out during the *SHIFT* state. The *LOAD* state occurs before the *SHIFT* state, but occurs within 1 system clock cycle and thus could not be seen in the waveform at the current zoom view.

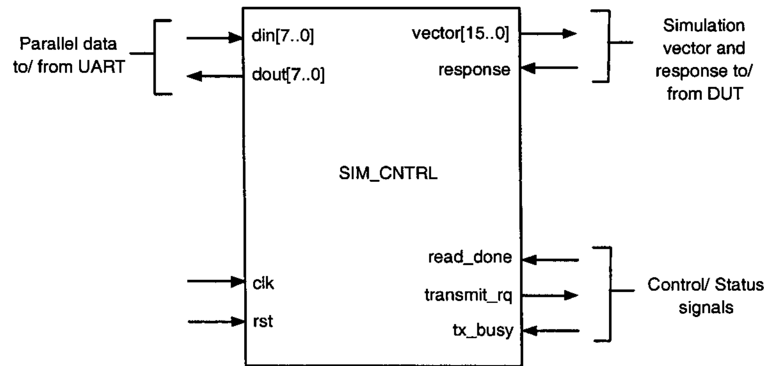


Figure 6.18: Simulation controller module

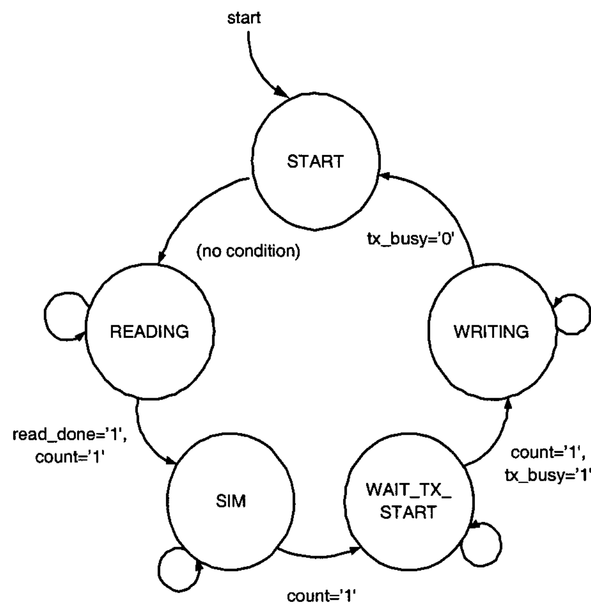


Figure 6.19: Simulation controller state diagram

6.5.3.5 Simulation Controller

The simulation controller (Fig. 6.18) has the responsibility of processing any incoming data from the processor via the UART (*din* bus) for DUT vector simulation (*vector* bus). The controller reads the 8-bit vector from the UART when the *read_done* signal is asserted by the UART. The vector is concatenated with the necessary number of reads to match the input width of the design. For instance, a circuit with 16-inputs would require two 8-bit reads from the UART. The response of the test circuit is then processed in this controller and sent back via the UART to the processor first by signaling *transmit_rq* to request transmission. If the UART is not busy performing another transmission, as dictated by *tx_busy*, the response is transmitted.

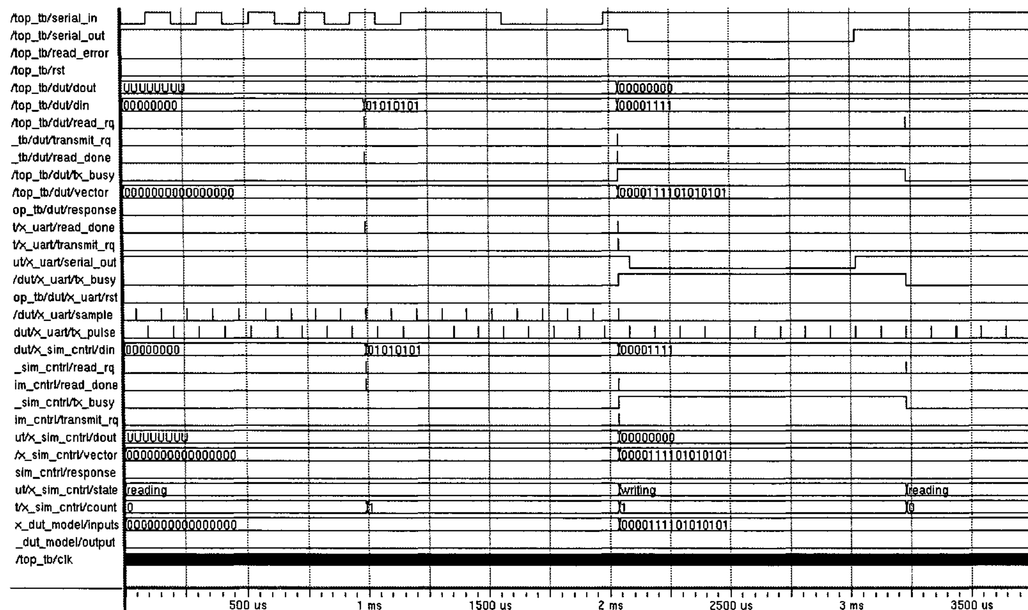


Figure 6.20: Top-level simulation with all component instantiations

The simulation controller is designed to have 5 states: *START*, *READING*, *SIM*, *WAIT_TX_START* and *WRITING* (Fig. 6.19). Beginning in the START state, the controller requests a read on the *read_rq* line for the UART receiver to send it data. With only a byte of data per payload in the $8N1$ convention, a 16-bit input DUT would require two successive reads to make up a matching simulating vector size. Therefore, in the *READING* state, it waits for two successful reads from the UART (by monitoring *read_done*) to construct the 16-bit simulation vector. Next, the vector is applied to the DUT's inputs for simulation in the SIM state. The controller requests transmission on *transmit_rq* line to the UART and loads up a shift register with the appropriate start, data and stop bits in the *WAIT_TX_START* state. As soon as the UART responses that it is busy sending the data back to the PC with *tx_busy*, the controller waits in the *WRITING* state until the transmission is complete when *tx_busy* is set low before starting over again for the next vector in the *START* state.

The waveform in Fig. 6.20 is a simulation with all the UART components instantiated with the controller. A 16-bit simulation vector 0000111101010101 is constructed by two consecutive reads issued by the controller in the *READING* state. The data is captured during which the *read_done* signal issued to the controller by the receiver.. The *SIM* state then latches the vector into the DUT model. The model is represented by a 16-bit XOR parity function, and is given an even parity output by the vector. The expected simulation output of '0' is returned as part of *dout* and is returned serially during states *WAIT_TX_START* and *WRITING*. After the transmission, the state controller returns to the *START* state.

```
1  Logic Utilization:
2    Number of Slice Flip Flops:         99 out of  47,232    1%
3    Number of 4 input LUTs:            137 out of  47,232    1%
4  Logic Distribution:
5      Number of occupied Slices:                       94 out of  23,616    1%
6      Number of Slices containing only related logic: 94 out of      94  100%
7      Number of Slices containing unrelated logic:     0 out of      94    0%
8        *See NOTES below for an explanation of the effects of unrelated logic
9  Total Number 4 input LUTs:         139 out of  47,232    1%
10       Number used as logic:                      137
11       Number used as a route-thru:                 2
12
13    Number of bonded IOBs:            5 out of     692    1%
14       IOB Flip Flops:                              3
15
16 Total equivalent gate count for design:  1,650
17 Additional JTAG gate count for IOBs:  240
18 Peak Memory Usage:  149 MB
```

Figure 6.21: Area usage of UART and simulation controller

Table 6.3: Serial port device mapping in UNIX and Linux

System	Port 1	Port 2	Virtual Term N
Solaris/SunOS	/dev/ttya	/dev/ttyb	–
Linux	/dev/ttyS0	/dev/ttyS1	/dev/tty$N$

6.5.3.6 Test Circuitry Area

With limited space on FPGA devices, it is imperative to limit the amount of test circuitry to a minimum. The circuitry is implemented on Virtex II Pro device (part number XC2VP50) on the FF1152-5 board. In Fig. 6.21, the Xilinx ISE/Impact implementation tool reports less than 1% of area usage required for both the UART and simulation controller modules.

6.5.4 Test Vector Transmission from Workstation

Test vectors are generated by the workstation and transmitted via the RS-232 serial protocol to the FPGA. Table 6.3 lists the available serial ports available under SunOS and Linux. Virtual terminals, (which could be triggered by CTRL+ALT+FN on the workstation,) emulate serial terminal behavior, may be accessed to by /dev/tty$N$, where N is an integer. With the 8N1 serial convention, the 8-bit data is merely transmission of a character. Unfortunately, not all vectors constitute 8-bits and only a small subset constitutes alphanumeric characters. A program in C based on the standard *termios.h* C POSIX libraries is written to handle these situations. Specifics of the protocol settings, such as baud rate and protocol (8N1) were set based on functions in this library. Routines for port initialization are based on the guides in [Swe05][Fre01]. Then functions are written to assign each 8-bit partition to its ASCII equivalent character and each character is concatenated to form a string. With all data being as part of a test vector, there is no need of identifying any standard escape/ control characters, such as ones implemented in modem terminal applications, such as *minicom*. Using the

"write" system call the program sends the characters to the port via the proper protocol specified in the program. For the 16-bit input DUT, each string transmission consists of two characters.

CHAPTER 7

CONCLUSION

With all the recent technology advancements and complexity in hardware, testing and verification remains as the essential ingredient in the design-manufacturing flow of ensuring that the final product works and behaves as desired. This thesis has placed emphasis on these topics, particularly in fault modeling, test generation, simulation methods and fault redundancy identification.

One of the major challenges is having a fault model that can represent the abundant classes of design errors while maintaining a relatively small manageable fault list. The size of the fault list directly affects the performance of the simulator, and having a large fault list would only adds more burden in resources. Simulation time would take longer and slower, which is obviously undesirable in any time-to-market environment.

Gate-level faults may be represented by models such as gate replacements with another gate of matching I/O count and with MIGSE module [AAH95] [BH97]. The thesis has shown the shortcomings of these techniques in that fault lists are large and difficult to manage. In particular, the latter model requires constructing a replacement module for every gates in the synthesis library used in the design. When it comes to FPGA applications, the classes of errors are even larger, as the LUT is capable of representing more functionality than the general standard Boolean gate class found in ASIC. In this case, the demand of using a model capable of representing more functional errors compactly is even higher.

Erroneous gate replacement by cube distances is then proposed, which is capable of representing larger classes of functional errors. Each gate error function is represented by the Hamming distance from the original fault-free gate. Also, all of the possible 2^{2^n} functional errors of the n-input gate is then naturally classified by their Hamming distances. Each of these implicit category represents a group of faults such that detectability of each implicit fault implies detectability of all faults it represents. By fault dominance, it is shown that smaller distances are able to represent higher distanced ones, and therefore only minimal distanced errors are needed to be checked. From this, a minimal cube distance algorithm is derived where ATPG and

simulation routines begin checking errors at the lowest distance of 1 and progress to higher distances as required. The fault dominance property has allowed the fault list size to be dramatically reduced.

7.1 ATPG and Simulation

The background of ATPG and simulation has been covered in that ATPG are deterministic algorithms that target a particular fault and finding a test capable of detecting it, should one exist. Specifically, an ATPG variant, known as SAT-based ATPG has been looked at, which first sets up clauses based on the network structure, followed by solving the clauses with a SAT solver. Simulation on the other hand involves injection of vectors and monitoring the output for any erroneous responses, while each fault of the fault list is in turn introduced into the network.

In this thesis, implementation techniques of accommodating the error modeling techniques have been discussed. Specifically, ways of incorporating these into existing ATPG and simulation routines originally designed for the widely popular s-a-v has been looked at. This is to take full advantage of the speed and maturity in the ATPG-SAT algorithms available for s-a-v faults.

Redundant faults exist due to the nature of fault modeling such that its presence is undetectable. At the test generation cycle, redundant faults need to be recognized and removed from the fault list. The cause of these faults has been looked at and techniques in identifying them have been discussed closely for the cube-distance error replacement model.

7.2 FPGA Functional Testing

A testing algorithm based on the implicit cube-distance model has been proposed for modeling LUT errors in the FPGA. The overall algorithm includes simulation and fault detection by deterministic methods to identify redundant faults. The deterministic methods are based on a mapping of the proposed model to the classical s-a-v model. The purpose of such a mapping is so that existing routines for the s-a-v may be reused. This is executed when errors at the maximum cube distances cannot be detected by random simulation vectors. Berkeley SIS is used to emulate the mapping and testing process in hardware.

The thesis looked at checking for the netlist's correctness before bitstream generation and downloading the design onto the FPGA device. The FPGA chapter then looks at methods of integrating functional testing into the design flow. It is concluded that the most practical solution for the cube distance model would be to use the JBits API for Xilinx devices. The API allows direct manipulation of the bitstream. With partial reconfiguration, the API generates a bitstream, which only reflects the change and errors within the LUT. Test circuitry is introduced on the FPGA which uses less than 1% of the Virtex II Pro device area. Partial

reconfiguration would also allow functional testing while the FPGA is in operation.

7.3 Summary of Results

7.3.1 Cube Distance Model on Designs Mapped to Boolean Gates

The cube-distance error replacement model is adequate to represent explicit gate replacements, as well as s-a-v errors. The fault list sizes of the proposed implicit model have shown to be smaller than the explicit gate replacement model, and the MIGSE model proposed in [AAH95][BH97]. Finally, a technique of adjusting standard ATPG-SAT s-a-v routines for the detection of the cube-distance fault model has been proposed. The proposed cube-distance implicit fault model significantly reduces a fault list compared to the explicit representation of the same class of functional faults. Additionally, the proposed model is capable of representing any complex logic to complex logic replacement.

7.3.2 Cube Distance Model on LUT Mapped Circuits

The cube-distance error replacement model is adequate to represent explicit gate replacements, as well as s-a-v errors. The fault list sizes of the proposed implicit model have shown to be smaller than the explicit gate replacement model, and the MIGSE module replacement model. Results have shown that the number implicit faults are low and are able to represent a large number of actual functional errors. The number of these functional errors is proportional to the size of the LUT, but is high considering that it is the number of configurations that would be needed to test the hardware. Also the larger circuits mapped to the smallest LUT size of 3 has relatively low coverage.

7.3.3 Simulations with Lattice Vectors

Simulations with lattice vectors have shown lower fault coverages than with random vectors in many cases. Also, even with $log_2(n)$ layers for an n-input network, fault coverages are often lower than with random vectors, while simulation takes a longer time.

7.4 Future Work

7.4.1 FPGA Testing

As all experiments on FPGA testing are implemented in Berkeley SIS to emulate the functional operation on hardware, a migration of the software emulation proposed for this fault model into real time online FPGA testing will be desirable for future work.

7.4.2 Algorithm Optimization and Diagnostics

In any software implementation, particularly simulators and SAT-ATPG, algorithms could be optimized further to shorten execution times. As such, an interesting element would be to have better diagnostics should errors occur. At the present time in the proposed functional FPGA testing method, the erroneous LUT information is retained. Having more statistical data on the type of functional errors on the LUT would be a useful hint for the designer to use alternative synthesis methods to achieve better error-free functional designs as well as having better testability results in the implementation.

7.4.3 Application of the Proposed Method to Other Device Technologies

With devices entering the submicron era of manufacturing, various nanotechnology methods have undergone research as alternatives to the current CMOS technology. These include alternatives to silicon, DNA manipulation techniques and the use of quantum dots (QD) in quantum cellular automata (QCA). At the present time formal fault modeling techniques is in a relatively early stage. Because of the way implicit cube-distance errors' ability of categorizing all functional errors into a compact fault list, this can be applied to newer technologies as well, where the categories of errors are even larger and varied.

The model may be used in reversible logic and computing, where the fundamental logic is more complex than the Boolean gate. Logical elements in reversible logic is to consume no (or minimal) power as charges maintain in within the circuit, as opposed to charging-discharging cycles in standard CMOS. Although there is a large assortment of realization technologies for reversible logic, such as CMOS (adiabatic circuits), nano-mechanical and quantum, the model will be sufficient in covering functional faults at the reversible logic level.

REFERENCES

[AAH95] H. Al-Assad and J. P. Hayes. Design verification via simulation and automatic test pattern generation. In *Proceedings of International Conference on Computer Aided Design*, pages 174–180, 1995.

[ABF94] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* Wiley-IEEE Press, New York, NY, 1994.

[BA00] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.* Kluwer Academic Publishers, Boston, 2000.

[BH97] R.D. Blanton and J.P. Hayes. The input pattern fault model and its application. In *Proceedings of the 1997 European conference on Design and Test Conference*, page 628, 1997.

[Cha] Ken Chapman. *XAPP223: 200 MHz UART with Internal 16-Byte Buffer.* Xilinx Inc.

[Cho] Phil Chong. SIS 1.3 unofficial distribution – linux port. http://www-cad.eecs.berkeley.edu/~pchong/sis.html.

[Fre01] Gary Frerking. *Serial Programming HOWTO.* 2001. http://tldp.org/HOWTO/Serial-Programming-HOWTO/index.html.

[GCW+03] P. Graham, M. Caffrey, M. Wirthlin, E. Johnson, and N. Rollins. Reconfigurable computing in space: From current technology to reconfigurable systems-on-a-chip. In *24th Annual IEEE Aerospace Conference*, Big Sky, MT, March 2003.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. Jbits: A java-based interface for reconfigurable computing. In *2nd Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, page 27, 1999.

[GR81] P. Goel and B. C. Rosales. Podem-x: An automatic test generation system for VLSI logic structures. In *Proc. 18th Design Automation Conference*, pages 260–268, June 1981.

[GSTT01] Fernando M. Gonçalves, Marcelino B. Santos, Isabel Teixeira, and João Paulo Teixeira. Implicit functionality and multiple branch coverage (ifmb): a testability metric for rt-level. In *Proceedings of IEEE International Test Conference, ITC*, pages 377–385, 2001.

[Haj03] Hossain Hajimowlana. Design verification and debugging FPGA implementations, November 2003. http://www.edn.com/filtered/pdfs/contents/images/333612.pdf.

[HC00] Chung-Yang Huang and Kwang-Ting Cheng. Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 118–123. ACM Press, 2000.

[KMV01] Venkatram Krishnaswamy, A. B. Ma, and Praveen Vishakantaiah. A study of bridging defect probabilities on a pentium (tm) 4 cpu. In *Proceedings of IEEE International Test Conference, ITC*, pages 688–695, 2001.

[Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 11(1):4–15, January 1992.

[Mic94] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[Mil04] Mark Miller. Manufacturing-aware design helps boost IC yield, September 2004. http://www.eedesign.com/features/exclusive/showArticle.jhtml?articleId=47102054.

[MSM98] S. Mitra, P. P. Shirvani, and E.J. McCluskey. Fault location in fpga-based reconfigurable systems. In *IEEE Intl. High Level Design Validation and Test Workshop*, 1998.

[MUPRS05] Grégory Mermoud, Andres Upegui, Carlos-Andres Peña-Reyes, and Eduardo Sanchez. A dynamically-reconfigurable fpga platform for evolving fuzzy systems. In *8th International Work-Conference on Artificial Neural Networks, IWANN*, 2005.

[MZ02] S. McCracken and Z. Zilic. Fpga test time reduction through a novel interconnect testing scheme. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2002.

[Per] Bruce Perens. Electric fence. http://www.perens.com/FreeSoftware/ElectricFence/.

[PTS03] A. Parreira, J. P. Teixeira, and M.B. Santos. A novel approach to fpga-based hardware fault modeling and simulation. In *Design and Diagnostics of Electronic Circuits and Syst. Workshop*, pages 17–24, April 2003.

[RBS67] J. P. Roth, W.G. Bouricius, and P. R. Schneider. Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits. *IEEE Transactions on Electronic Computers*, EC-16(10):567–579, October 1967.

[RPFZ00] M. Renovell, J.M Portal, J. Figueras, and Y. Zorian. An approach to minimize the test configuration for the logic cells of the xilinx xc4000 fpgas family. *Electronic Testing: Theory and Applications*, pages 289–299, 2000.

[RT98] J. Rajski and J. Tyszer. *Arithmetic Built-In Self-Test*. Prentice Hall PTR, 1998.

[RZ01] K. Radecka and Z. Zilic. Identifying redundant gate replacements in verification by error modeling. In *Proceedings of IEEE International Test Conference, ITC*, pages 803–812, 2001.

[RZ04] Katarzyna Radecka and Zeljko Zilic. Design verification by test vectors and arithmetic transform universal test set. *IEEE Trans. Comput.*, 53(5):628–640, 2004.

[Sal03] Kewal K. Saluja. Outstanding challenges in testing nanotechnology based integrated circuits. In *Asian Test Symposium*, page 2, 2003.

[SF03] Rindert Schutten and Tom Fitzpatrick. Design for verification methodology allows silicon success, April 2003. http://www.eetimes.com/story/OEG20030418S0043.

[SKG+03] S. K. Shukla, R. Karri, S. C. Goldstein, F. Brewer, K. Banerjee, and S. Basu. Nano, quantum, and molecular computing: Challenges in verification and test. In *IEEE International High Level Design Validation and Test Workshop*, pages 3–7, 2003.

[SMSP97] N. R. Shnidman, W. Mangione-Smith, and M. Potkonjak. On-line fault detection for programmable logic. In *17th Conference on Advanced Research in VLSI (ARVLSI)*, Sept 1997.

[SSL+92] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis*. Department of Electrical Engineering and Computer Science University of California, Berkeley, CA 94720, May 1992. ftp://ic.eecs.berkeley.edu/pub/sis/.

[Ste] Jeffrey Stedfast. Alleyoop. http://alleyoop.sourceforge.net.

[Swe05] Mike Sweet. *Serial Programming Guide for POSIX Operating Systems*. 2005. http://www.easysw.com/~mike/serial/.

[TAZ00] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul. Register-transfer level fault modeling and test evaluation techniques for vlsi circuits. In *Proc. International Test Conf.*, pages 940–949, October 2000.

[TAZ03] P. A. Thaker, V. D. Agrawal, and M. E. Zaghloul. A test evaluation technique for VLSI circuits using register-transfer level fault modeling. *IEEE Trans. CAD*, 22(8):1104–1113, August 2003.

[TGH97] Paul Tafertshofer, Andreas Ganz, and Manfred Henftling. A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists. In *ICCAD '97: Proceedings*

of the 1997 IEEE/ACM international conference on Computer-aided design, pages 648–655. IEEE Computer Society, 1997.

[THML04] M. B. Tahoori, J. Huang, M. Momenzadeh, and F. Lombardi. Testing of quantum cellular automata. *IEEE Transaction on Nanotechnology*, 2004.

[UCL] SIS – ULCA variant. http://www.cs.ucla.edu/classes/CS258G/sis-1.3/sis.tar.gz.

[WJR+03] M. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, and P. Graham. The reliability of fpga circuit designs in the presence of radiation induced configuration upsets. In *IEEE Symp. FPGAs for Custom Computing Machines (FCCM)*, Napa, CA, April 2003.

[Xil] Xilinx Inc. *Xilinx 5 Software Manuals and Help*.

[Xil04] Xilinx Inc. *XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based*, 2004.