TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Engineering

IAF70LT

Emmanuel Ovie Osimiry

IASM144689

RANDOM DIAGNOSTIC TEST GENERATION FOR DIGITAL CIRCUITS

Master thesis

Prof. Raimund-Johannes Ubar

D.Sc. Institute of Computer Engineering, Tallinn University of Technology.

Professor, Chair of Computer Systems Test and Verification.

Tallinn 2016

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the materials used, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Emmanuel Ovie Osimiry

29.08.16

Abstract

In this thesis several experiments and research were carried out to generate random test vectors with better average diagnostic resolution for digital circuits. The Turbo Tester (TT) [**1**] tool suite was the target framework used for this experimental and research work, and a number of methods have been developed and proposed.

Typically when generating test vectors the simplest method used is a random pattern generator [**2**] [**3**], the main goal is to generate a minimum number of test vectors with very high fault coverage. However such a set of test vectors may not produce good diagnostic resolution whereby it is difficult to narrow down to the specific location that has a fault. Most times since the test vectors have a high fault coverage a high number of candidate location qualify as the source of a fault hence this makes diagnostic inspection of a circuit difficult and evasive.

A measure for evaluating the Average Diagnostic Resolution of a given test set is proposed and with this measure the average diagnostic resolution of the test patterns generated by the different methods proposed have been evaluated. The methods that have been developed in this thesis are based on the random pattern generation tool of Turbo Tester [**1**]. Also the effect of fault collapsing on the diagnostic resolution was also experimented with and shown.

To provide a very rich set of result three benchmark families, ISCAS’ 85 [**4**], ISCAS’ 89 [**5**] and ITC’ 99 [**6**] have been used for this work and the experimental results show that the methods proposed improve the average diagnostic resolution and have good potential. A comprehensive analysis and comparison of the new methods proposed has been carried out and suggestions are given on which particular new method is more advisable to use than the others, depending on the different constraints such as the test length, test generation time, and on the diagnostic resolution.

This thesis is written in English and is 90 pages long, including 6 chapters, 22 figures and 22 tables.

Annotatsioon

Testide genereerimine juhuslike arvude meetodil digitaalskeemide rikete diagnoosiks

Käesolevas töös on esitatud uurimus ja eksperimentide seeria, mille põhjal on välja töötatud uued meetodid testide genereerimiseks juhuslike arvude abil, mis võimaldaksid kõrget diagnostilist resolutsiooni. Uurimistöö läbiviimiseks on kasutatud Turbo-Tester [1] diagnostikakeskkonda. Läbi viidud uurimuse tulemusena töötati välja rida meetodeid, milliseid võrreldi nii omavahel kui ka senise tuntud meetodiga.

Traditsiooniliselt on kõige lihtsamaks testide genereerimise meetodiks stohhastiline juhuslike arvude kasutamisel põhinev testide genereerimise meetod [2], [3]. Kriteeriumiks on siin valida juhuslikult genereeritud testvektorite hulgast välja võimalikult väike alamhulk vektoreid võimalikult kõrge rikete kattega. Paraku selline traditsiooniline lähenemisviis ei garanteeri seejuures head rikete diagnoosi ehk siis kõrget diagnostilist resolutsiooni – võimalikult täpset rikke asukoha määramist.

Antud testi diagnoosivõime kvaliteedi hindamiseks on töös välja pakutud mõiste „testi keskmine diagnostiline resolutsioon“. Selle mõõdu abil on võimalik hinnata erinevate testide diagnoosivõimet ja ühtlasi ka erinevate testide genereerimise meetodite efektiivsust. Antud töös on aluseks võetud Turbo-Testris kasutatav juhuslike arvude kasutamisel põhinev testide generaator, mille juures on arvesse võetud ka nn. rikete kollapsi mõju diagnostilisele resolutsioonile.

Võimaldamaks väga laiaulatuslikku ja usaldusväärset erinevate meetodite võrdlust on eksperimentide läbiviimiseks kasutatud kolme katseskeemide perekonda ISCAS’ 85 [4], ISCAS’ 89 [5] and ITC’ 99 [6]. Läbi viidud eksperimendid demonstreerisid, et uued väljatöötatud testide genereerimise meetodid võimaldavad saada teste, mis märgatavalt suudavad parandada diagnostilist resolutsiooni rikete otsimisel, võrreldes seniste testide genereerimise meetoditega. On läbi viidud ka uute meetodite analüüs ning antud soovitused, milliste kriteeriumite puhul (nõuetena testi pikkusele, testi genereerimise ajale või diagnostilisele resolutsioonile) üks või teine meetod on paremini sobiv.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 90 leheküljel, 6 peatükki, 22 joonist, 22 tabelit.

Acknowledgements

I want to thank the God of my heart for giving me the opportunity, strong will and inner strength to complete this Master's degree program. Without your favour I would not have been able to complete the program.

I immensely thank my supervisor, Prof Raimund-Johanesse Ubar for his superb supervision and guidance during the course of this thesis. When I was lost you received me and guided me. I will always remember your kindness.

Many thanks to Steven Oyeniran Adeboye for your insightful advice and suggestions, now I see the benefits. Thank you once again.

Thank you to my friends and colleagues Nevin, Nish, Tsotne and Uzo for your friendship and for challenging me, studying with you guys was very interesting, I am glad I met you all. I want to acknowledge Siavoosh and Adeel for your advice and mentorship, I appreciate every bit of it.

A big thank you to the Department of Computer and Systems Engineering, Tallinn University of Technology and to the Estonian government for offering me an opportunity to study for a master's degree and for your collective support during the program, if not for the opportunity you gave me I would not have written this thesis. I am very grateful.

Much appreciation to my parents and siblings especially to my big sister, Peace, for encouraging and supporting me, your collective love reminded me of our everlasting unity and this was very instrumental to my success. I am thankful to my fiancée, Chioma, for showing and demonstrating understanding throughout the period, thank you dear.

This work was supported by Skype and the Estonia Information Technology Foundation for Education (HITSA).

Table of abbreviations and terms

ATPG	Automatic Test Pattern Generator
ADR	Average Diagnostic Resolution
BISD	Built In Self Diagnosis
BIST	Built In Self Test
DR	Diagnostic Resolution
DC	Diagnostic Coverage
FC	Fault Coverage
IC	Integrated Circuit
LC	Limiting Criterion
ORA	Output Response Analyser
RTG	Random Test Generation
RPG	Random Pattern Generator
SAF	Stuck-At Fault
SA0	Stuck-At-0
SA1	Stuck-At-1
SoC	System on Chip
SSBDD	Structurally Synthesized Binary Decision Diagrams
TPG	Test Pattern Generator
PCB	Printed Circuit Board

Table of contents

1. Introduction 13

1.1. Background and problem 14

1.2. Description of the task solved 16

1.3. Thesis Structure 17

2. Background 18

2.1. Diagnosis in Digital Circuits 18

2.2. Terminologies and Definition 18

2.3. Fault models 19

2.4. Logic Diagnosis Paradigms 19

2.4.1. Cause-Effect paradigm 19

2.4.2. Effect-Cause paradigm 19

2.4.3. Inject-and-Evaluate paradigm 20

2.5. Built In Self-Test and Built in Self Diagnosis 20

2.6. Some methods for generating diagnostic test 21

2.7. Challenges of deterministic diagnostic test generation and motivation for random diagnostic test generation 23

3. Methods for generating test patterns with high Average Diagnostic Resolution. 25

3.1. Calculation of the average diagnostic resolution of a given test set. 25

3.2. Generation of random test set with better ADR 26

3.3. Generating random test patterns with better ADR 28

3.3.1. RTG with better ADR - M1 (Δ = max) 28

3.3.2. Effect of fault collapsing on the ADR 29

3.3.3. RTG with better ADR - M1 (Δ = min) 30

3.3.4. Comprehensive result (M1). 30

3.3.5. Observations and summary for method M1 33

3.4. RTG with better ADR - M2 34

3.4.1. Comprehensive result (M2). 35

3.4.2. Observations and summary for M2 38

3.5. RTG with better ADR - M3 38

3.5.1. Comprehensive result (M1 M2 and M3) 39

3.6. Observations and summary for M1, M2 and M3 41

4. Improving the ADR after RTG 43

4.1. Improving the ADR after RTG - A1 43

4.2. Improving the ADR after RTG - A2 45

4.2.1. Weighing function for selecting additional test vectors 46

4.3. Comprehensive result of methods A1 and A2 50

4.4. Comparing methods A1 and A2 53

5. Experimental Results 54

5.1. Comparison of proposed methods 54

5.2. Strength and weakness of the proposed methods 58

6. Summary and conclusion. 59

References 61

Appendix 1 – Program Description and Manual 65

Using the tool 66

Example 1 – How to generate diagnostic test with M1 69

Example 2 - How to generate diagnostic test with M2 70

Example 3 – How to generate diagnostic test with M3 71

Example 4 – How to generate diagnostic test with A2 72

Appendix 2 - How to compute/extract the average diagnostic resolution from the test file. 73

Example – How to use the safdiag.jar tool to compute the average diagnostic resolution of a test file. 74

Appendix 3 – How to use the GUI tool DiagBoost.exe for A1..................................... 75

Appendix 4 – Source Code For method A2 80

List of figures

Figure 1 Two approaches for generation of random test patterns 27

Figure 2 Fault coverage and the average diagnostic resolution as the functions of random test length 27

Figure 3 Comparison of M1 (Δ=max) with the traditional approach............................ 28

Figure 4 Comparison of M1 (Δ=min) with the traditional approach............................. 30

Figure 5 Flow chart for method M2 35

Figure 6 Flow chart describing A1 45

Figure 7 normalizing the weights 49

Figure 8. Running command to generate diagnostic test with option M1...................... 69

Figure 9. Program output after running with option M1. 69

Figure 10. Running command to generate diagnostic test with option M2................... 70

Figure 11. Program output after running with option M2. 70

Figure 12. Running command to generate diagnostic test with option M3................... 71

Figure 13. Program output after running with option M3. 71

Figure 14. Running command to generate diagnostic test with option A2. 72

Figure 15. Program output after running with option A2... 72

Figure 16 How to compute the ADR of a test file... 74

Figure 17. ADR computation complete... 74

Figure 18. DiagBoost GUI tool. 76

Figure 19. DiagBoost Successfully loaded test file and SSBDD file............................ 77

Figure 20. Number of iterations DiagBoost should perform... 77

Figure 21. Running the DiagBoost tool... 78

Figure 22. Generated files after DiagBoost stops.. 78

List of tables

Table 1 Diagnostic matrix for fault diagnosis 26

Table 2 Two criterions for selecting patterns (circuit c432) 29

Table 3 Influence of the fault collapsing on M1 (c432) 29

Table 4 result for method M1 for ISCAS’85 31

Table 5 result for method M1 for ISCAS’89 32

Table 6 result for method M1 for ITC’99 33

Table 7 result for M2 for ISCAS’85 36

Table 8 result for M2 for ISCAS’89 37

Table 9 result for M2 for ITC’99 38

Table 10 ICAS’85 family, Comparing M3 with best results of M1 and M2 39

Table 11 ICAS’89 family, Comparing M3 with best results of M1 and M2 40

Table 12 ITC’99 family, Comparing M3 with best results of M1 and M2 41

Table 13. Example of weighted fault vectors 46

Table 14 Example of a large fault group 46

Table 15 truth table for logic function 48

Table 16 Example of logic function with candidate vector 48

Table 17. Comparing method A1 and A2 with ISCAS' 85 circuits 50

Table 18 Comparing method A1 and A2 with ISCAS' 89 circuits 51

Table 19. Comparing A1 and A2 with ITC' 85 circuits 52

Table 20. Comparing ADR of best proposed methods with ADR of original test set using ISCAS’85 55

Table 21. Comparing ADR of best proposed methods with ADR of original test set using ISCAS’89 56

Table 22. Comparing ADR of best proposed methods with ADR of original test set using ITC'99 57

1. Introduction

The focus of this thesis is on generating random test patterns with better average diagnostic resolution for digital circuits. The outcome of a better resolution aids better diagnostics of digital circuits, and this helps the test engineer to easily find the specific location that has fault.

The first section of this chapter begins with the problem statement, followed by a description of the scope of work carried out and the methodology. In concluding the chapter a summary of the work done in this thesis is presented.

1.1. Background and problem

Over the years integrated circuits have improved tremendously with the continuous miniaturization of the transistor and tight integration of more components on a single die to form complex systems. These technological improvements gave birth to technologies like Systems on Chips (SoC) and complex Integrated Circuits (IC) [**7**]. Most of these improvements were predicted by Gordon Moore [**8**] [**9**].

All of the aforementioned development has made it possible for modern day systems to keep up with the ever increasing demand of faster and efficient performance, but this has introduced very high complexities in testing and diagnosis of such systems.

During the design phase and after the design phase of any digital system, testing of the system is incorporated into the process to improve the yield and to ensure a certain level of acceptance [**2**] [**10**]. But testing of digital systems is costly [**10**] so a compact set of test vectors is desirable for reducing the cost of time when testing; hence the traditional goal when generating test set is for high fault coverage [**11**] and minimum test length.

The random test generator is popularly used for generating such high volume test vectors because of its quickness, simplicity and cheapness and this is one of the motivation for this thesis, taking advantage of the quickness, simplicity and cheapness of a random test generator and at the same time guaranteeing a good average diagnostic resolution. However a high fault coverage test set does not always guarantee a high diagnostic resolution [**2**] [**11**].

Diagnosis is important after a fault has been detected as it helps to locate the specific location of a fault and can help the designers to understand what caused a failure and to prevent them from reoccurring. A diagnostic test set is used for diagnosis and usually it must have a good diagnostic resolution in order for it to be very useful. This diagnostic test set is normally generated deterministically by a diagnostic generator, however the deterministic approach is computationally expensive because the diagnostic generator has to generate a distinguishing vector for every fault pair in the given test set.

In this thesis a random approach has been used to avoid the expensive deterministic approach that is mostly used in a traditional diagnostic test generator and the experimental

result show that this approach actually improves the ADR and also has a good test generation time.

There are two traditional diagnosis approaches effect-cause and cause-effect [**12**]. The cause-effect approach is the main focus in this thesis. A number of methods are proposed for randomly generating test vectors with good average diagnostic resolution.

1.2. Description of the task solved

A number of methods are proposed for random generation of test vectors with better average diagnostic resolution. To evaluate the diagnostic resolution, a measure for evaluating the Average Diagnostic Resolution (ADR) is proposed and with this measure the proposed methods have been evaluated and compared against each other and with traditional methods.

The first set of methods generate random test patterns with better diagnostic quality during the random test generation phase. The second set of methods are more like optimization methods but with very slight determinism, they try to improve the diagnostic resolution after the random test patterns have been generated.

The experimental results of the methods presented are compared against each other and with traditional methods that are used for generating test sets.

A practical and experimental approach was the main drive behind this thesis. All of the ideas and hypothesis were analysed first, then implemented to verify the outcomes.

In the end this thesis has contributed to the turbo tester tool by introducing additional functionality such as random diagnostic test generation.

1.3. Thesis Structure

Chapter two gives background information related to this thesis such as diagnosis in digital circuits, fault models, diagnosis paradigms, some methods proposed by other authors for generating diagnostic tests then finally the challenges with generating diagnostic test and the motivation behind this thesis.

In Chapter three, three out of the five methods proposed are presented and discussed. The effect of fault collapsing is also shown, and some experimental results are presented with a short discussion concluding the chapter.

The remaining two methods proposed are captured in chapter four, some experimental results are also presented and a comparison concludes the chapter.

Chapter five presents the general experimental result and compares the best methods that have been proposed to show the amount of improvement the random approach introduces to the normal test set that has been generated for testing using the Random ATPG and Deterministic ATPG. Finally chapter six summarises and concludes the findings of this thesis.

The experimental platform used for the experiments was an Intel i7 octal core at 2.13 GHz, 8 GB RAM Laptop.

2. Background

This chapter gives an overview and background information related to this thesis. First the description and importance of digital circuit diagnosis is established. Then fault models and the major paradigms and approaches used for diagnosis are discussed. Furthermore some methods for generating diagnostic test are presented and discussed; finally in concluding the chapter the motivation for this thesis is discussed.

2.1. Diagnosis in Digital Circuits

In digital circuits diagnosis is the process of locating the faults present within a given fabricated copy of a circuit [**13**]. Typically after the fabrication of the IC some of the chips may be defective; a manufacturing test is used to screen out the bad chips [**2**]. But knowing that some particular chips have failed a test is not enough so the next step will be to locate the point where the failure has occurred. This is where diagnosis comes in. for Printed Circuit Boards (PCB) when the site of the fault has been located it is possible to repair, however this is not the case for IC so the main purpose or benefit of diagnosing IC is to gain useful insight on what caused the fault. This is particularly important as it helps to clarify what could be the possible cause of the failure. It is therefore important for a diagnostic tool to be able to generate diagnostic test quickly and to provide high accuracy. Depending on the information obtained from the diagnosis the chip can be redesigned to handle such failures or the fabrication process can be improved. Ultimately this would improve the yield.

2.2. Terminologies and Definition

Diagnostic Resolution (DR) – In summary this is defined as the ratio or fraction of the total number of faults by the number of detected fault groups [**14**] [**15**] [**16**]. Another source defines it as the total number of defect candidates [**2**]. A proper name will be Average Diagnostic Resolution (ADR). Throughout the rest of this literature the term ADR is used.

Diagnostic Coverage (DC) – This is simply the inverse of the DR.

Section 3.1 of chapter 3 gives more details about calculation of ADR proposed.

2.3. Fault models

In order to generate logic test for digital circuits a fault model is used to represent the digital circuit. Fault modelling is the process of modelling defects at a higher level of abstraction in the design hierarchy [**13**] [**17**] [**18**] [**19**] [**20**]. The aim of the fault model is to provide an easy platform that could replicate possible faults which could occur in the circuit. Fault model is useful for both test generation and diagnostic test generation for the logic circuit; however no single fault model can reflect the behaviour of all possible defect that may occur in a digital circuit [**2**]. Several fault models have been proposed but in this thesis the Stuck-At Fault (SAF) model has been used. The SAF model is a logic fault model which could affect any of the primary Input/Output (I/O), internal I/O of gates etc. The idea is that any of the fault site of the digital circuit could either be Stuck-At-0 (SA0) or Stuck-At-1 (SA1). For instance for an SA0 fault the logic will remain at logic 0 even when it should be logic 1 and vice versa for SA1 fault.

2.4. Logic Diagnosis Paradigms

The traditional diagnosis algorithms follow two major paradigms: *cause-effect* and *effect-cause* analysis. Another paradigm is the inject-and-evaluate paradigm. The following sub sections describe each of these paradigms.

2.4.1. Cause-Effect paradigm

This technique maps the causes of failures to specific fault models e.g. SAF model. It also relies on fault dictionaries. With the help of fault simulation, the fault dictionaries are built [**21**] [**22**]. Once the fault dictionary is ready the syndrome of the failing chip is analysed using dictionary look-up.

2.4.2. Effect-Cause paradigm

This paradigm is somewhat like the reverse reasoning of the cause-effect paradigm. It begins by identifying the failing outputs then starts reasoning on the logic structure of the circuit to be diagnosed. The algorithms based on this paradigm are simple when the single

fault assumption is adopted. In this case intersections of the input cones of failing outputs are calculated [**23**], or back-trace critical paths from failing outputs are processed [**24**]. Because of the sequential character of fault reasoning, this approach is called sequential or adaptive fault diagnosis.

2.4.3. Inject-and-Evaluate paradigm

As an alternative to back-trace approaches, which is utilized in the effect-cause paradigm, this inject and evaluate paradigm is introduced in [**25**] [**26**]. In [**27**] and [**28**] this approach is further improved with an efficient metrics that relies on curable vectors. This method uses injection and evaluation to predict locations of fault sites. This is different from the effect-cause approach which uses back-trace starting from the failing output and into the circuit to locate the fault site [**29**]. One major benefit of this approach is its high accuracy.

2.5. Built In Self-Test and Built in Self Diagnosis

Due to the rising complexity in digital circuits as a result of high integration of more components, diagnosing and testing has become difficult. Design for Testability methods such as the integration of a Built In Self Test (BIST) into the circuit greatly improves the testability and cost of testing. Basically a BIST comprises of a Test Pattern Generator (TPG) and an Output Response Analyser (ORA). Unlike traditional test techniques which may not achieve optimal fault coverage with chips designed with the nanometre scale technology, integration of the BIST at the design stage of the chip offers a solution to such a problem [**30**] [**31**], and this is gradually gaining acceptance in the industry [**32**]. Although the BIST has been successful in testing but it does not perform well in diagnosing hence cannot be relied on for Built In Self Diagnosis (BISD) because of the limited information it gives which is insufficient for diagnosis [**33**]. Some challenges that need to be overcome in order for the BIST to be useful for diagnosing are highlighted in [**34**].

2.6. Some methods for generating diagnostic test

In this section a discussion of some methods for generating diagnostic test patterns is presented.

Basically the job of a diagnostic test generator is to generate a test vector that can distinguish between a pair of faults that is supplied to it. Diagnostic test generation problem is a complex problem which requires repeated run for every pair of faults available in the test set in order to generate a distinguishing vector for every case. Sometimes some faults may be equivalent and as such it is not possible to distinguish them except in cases where one fault dominates the other. Some ways to cope with the complexity of generating diagnostic test are by removing redundant faults or fault collapsing and also by using the traditional test set meant for fault detection as a starting point; the advantage is that such a test set is usually compact so this reduces the number of pairs of faults the diagnostic generator has to generate distinguishing test vectors for.

According to [**13**] the techniques for generating diagnostic test can be classified into two major categories. The first category uses the traditional test generation technique (which is used for generating fault detection test set) as a driver to obtain a vector that distinguishes between a given pair of faults, while the second category directly targets how to distinguish between a pair of faults.

Two methods described in [**13**] that use the first category are described in the following. The first method proposed by [**35**] is based on two principles

1. If there is at least one or more outputs that is in the transitive fan-out of one of the faults that needs to be distinguished but not in the other one, then generate a test vector that will detect a fault at the output(s).

2. If (1) is not successful, then select an output that is in the transitive fan-out of both, however generate a test that propagates the effect of one fault to the output but not the other.

If both (1) and (2) fail then another output is selected and the steps are continued.

The second method proposed by [**36**] uses the traditional fault detection test set. From the set a pair of faults that are not distinguishable by the test set, a vector *v* that detects both faults but at the same output is selected from the set. Such a vector is then used to generate a new vector *v'*, this new vector *v'* is then fault simulated to see if it can distinguish the pair of faults. If it does the process moves to the next pair otherwise the vector *v'* is discarded and procedure continues.

In [**14**] a pair of faults f_1 and f_2 is distinguished by utilizing three copies of a fault model, a fault free model (*M*), a model with fault f_1 (M_{f1}) and a model with fault f_2 (M_{f2}). Combining these three models a vector that can distinguish the pair of faults is generated. A second approach that uses two copies of a fault model was proposed by [**37**]. This method does not consider a fault free version of the model but only the models containing faults *f1* and f_2 (M_{f1} and M_{f2}). Using the two fault models it tries to generate a vector that produces different values at the output of each model. If the process is successful then it must be propagated to at least a primary output.

A number of methods were proposed in [**11**]. The main approach utilized here is targeting directly how to generate a distinguishing vector for a given pair of vectors. First they present a set of method which requires modification of circuit netlist in order to model it as a circuit with a single inserted fault then an ATPG is used to target that fault. They also present another algorithm which uses fault dropping. When a fault is distinguished it is dropped but it is done without fault equivalence checking. Their main target is for Diagnostic Coverage.

In [**38**] they propose a method that tries to avoid deterministic test generation. Their algorithm targets the equivalence classes of the test set as it is generated. The method does not take the approach of distinguishing a fault pair (one at a time) instead, all faults within the equivalent class are simultaneously targeted thus the number of test and the generation time is reduced. They also utilize a process based on test elimination for generating a test for every equivalence class. The algorithm begins with the test set for fault detection *V*, using fault simulation and fault dropping of *V* they find the set of collapsed single stuck-at faults *F* that are detected by test set *V*. Then a set of fault pairs *F'* that is not guaranteed to be distinguished by *V* is defined. A fault simulation of *F'* using

V is performed and then fault pairs in F' that are distinguished by V are dropped. The fault pairs remaining in F' are then used to define equivalence classes. When generating the diagnostic test in every iteration the largest equivalence class that has not been considered is selected. A test $v \in V$ which is the first test that is able to detect every fault in the selected equivalent class is recorded. Then the procedure proceeds by trying to detect at least a fault from the class in a single output while eliminating the detection of other faults. using a set of conditions to modify v for the test and a cost function to determine if all the faults within the selected class are distinguished, if the conditions are met then v is selected without any further modification. If not, modification of v continues until a certain constant number of consecutive passes of all the inputs do not improve the number of fault pairs distinguished by v. The test is selected if it is able to distinguish a fault pair from the supplied class.

2.7. Challenges of deterministic diagnostic test generation and motivation for random diagnostic test generation

In the previous section the methods presented for diagnostic test generation attempt to generate deterministic diagnostic test set. When generating diagnostic test set for digital circuits the two major challenges faced are the computational cost when trying to generate a test vector for distinguishing between a fault pair and the time. A diagnostic pattern generator that is able to achieve both of the goals would be considered highly useful for practical cases. Some of the methods presented in the previous section for diagnostic test generation have proposed some solutions to cope with some of these challenges such as fault collapsing (for eliminating equivalent faults), the use of the original test set meant for fault detection as a starting point for fault diagnosis.

This thesis has taken a different approach to the problem by using a random and semi random method to achieve the same goal. The motivation behind this approach is that by using such an approach it is possible to bypass the expensive deterministic operation of trying to generate a distinguishing vector between every pair of fault, improve the diagnostic test generation speed and finally improve the ADR.

To evaluate the potential and effectiveness of this approach five methods which include two semi random methods were developed and experimented with using a wide

collection of circuit model from different benchmark families, ISCAS'85 [**4**], ISCAS'89 [**5**] and ITC'99 [**6**]. The results from the experiment show that the random approach of diagnostic test pattern generation has some potential and is promising. The computational cost and time are reduced and it improves the ADR of the generated test set when compared to the original test set generated by the ATPG.

3. Methods for generating test patterns with high Average Diagnostic Resolution.

Section 3.1 introduces a method for evaluating the Average Diagnostic Resolution (ADR) of a given test set. Section 3.2 discusses about random test generation, fault coverage (FC) and diagnostic resolution (DR). Sections 3.3 to 3.5 presents the methods for generating Random test patterns with very good ADR.

3.1. Calculation of the average diagnostic resolution of a given test set.

Let's represent the fault table for a given circuit and a given test set as a diagnostic matrix DM = $|| d_{ij} ||$ where *i* denotes a test pattern and *j* denotes a fault. We say $d_{ij} = 1$, if the test pattern t_i detects the fault f_j, otherwise $d_{ij} = 0$.

Let's call the column vectors $CW_j = (d_{j1}, d_{j2}, \dots, d_{jn})$ of DM as diagnostic codewords. Here n is the number of test patterns. Each fault f_i has its own binary codeword, but several faults may have the same diagnostic codeword.

Let F be the set of all faults in the circuit. Partition all the faults in F into a set of groups G, so that the codewords of the faults in a particular group Gk $\in$ G are equivalent. Obviously, $|G| \leq |F|$, and $|G| = |F|$ only in the case when all the columns CWj in DM are different

We can calculate now the average diagnostic resolution of the given circuit as follows:

$$D = \frac{\sum_{k=1}^{|G|} |G_k|}{|G|} \qquad (1)$$

Consider, as an example, the diagnostic matrix DM in Table 1, which provides the following partition of faults

G = {{f1},{f2},{f3,f6,f9},{f4,f7},{f5,f8},{f10},{f11}}.

Table 1 Diagnostic matrix for fault diagnosis

D		**Faults f_j**										
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Tests	1	1				1			1			1
t_i	2		1			1			1			1
	3			1			1			1	1	1
	4				1			1			1	

In this partition, there are three groups of indistinguishable faults: G3 = {f3, f6, f9}, G4 = {f4, f7}, and G5 = {f5, f8}. The average diagnostic resolution of the given test set, according to (1), is D = 1.57.In the best case of diagnosis we may have Dmin = 1, but the worst case diagnostic resolution will be Dmax = 3. To improve the resolution, additional test patterns are needed to distinguish the faults in the groups G3, G4 and G5.

3.2. Generation of random test set with better ADR

Random Test Generation (RTG) is one of the simplest methods for generating test patterns. Patterns are randomly generated as packages and thereafter fault-simulated on the circuit under test (CUT) [**2**]. As an efficient and straightforward criterion for gradual test pattern selection has proven to select only those patterns which exceed a given lower level of the number of detected not yet covered faults [**1**] [**39**]. Denote the increment in the fault coverage as the contribution of a test pattern by Δ. The selection criterion when Δ=max supports fast convergence towards 100% FC with small test length as depicted in Figure 1a. The criterion is easy to calculate in the run of the test generation process. However, such an approach will not provide high diagnostic resolution δ' as depicted in Figure 1a.

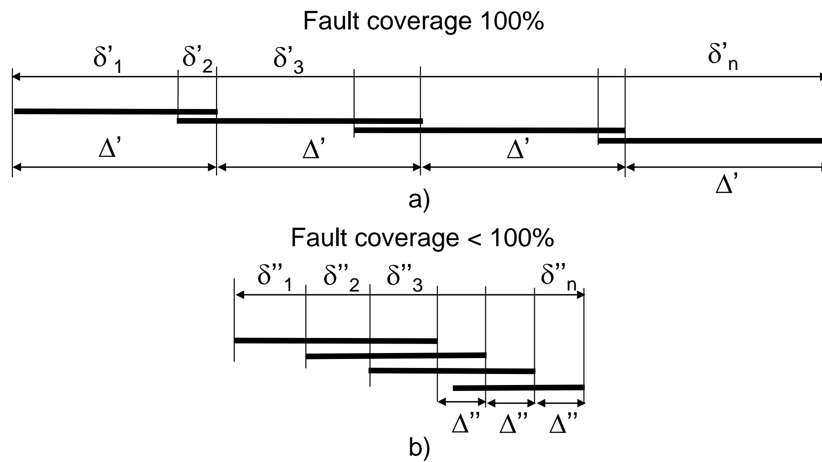

Figure 1 Two approaches for generation of random test patterns

Figure 1b illustrates the case where the pattern selection criterion is chosen so that Δ'' < Δ'. It is easy to see that in this case the final average diagnostic resolution δ'' may become far better than δ'. Figure 2 illustrates how the two parameters FC and ADR are evolving during the run of random pattern selection according to the criterion $\Delta = \max$.

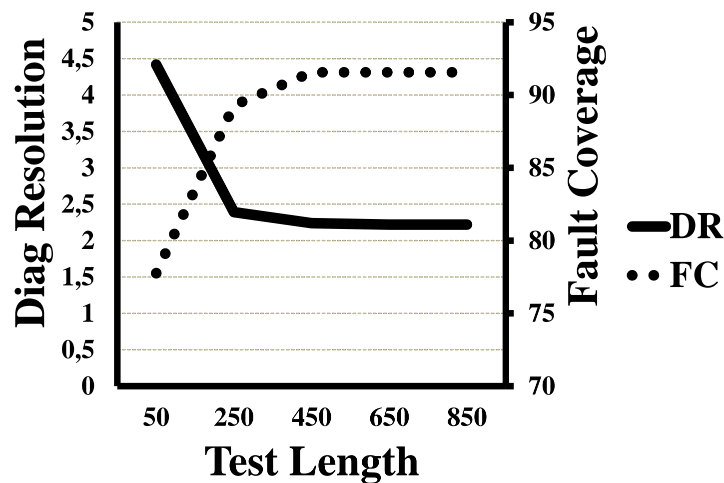

Figure 2 Fault coverage and the average diagnostic resolution as the functions of random test length

3.3. Generating random test patterns with better ADR

In this section a total of three (3) methods are presented. Let us refer to the methods with the following code names:

1. Method 1 will be referred to as M1
2. Method 2 will be referred to as M2, and
3. Method 3 will be referred to as M3.

M1 and M2 using two configurations each are first presented, then the effect of fault collapsing on the ADR is briefly shown. Finally M3 is presented. The methods presented in this chapter have small computational cost and they give better ADR when compared to a traditional random test generator (RTG).

3.3.1. RTG with better ADR - M1 (Δ = max)

Random test pattern generation with emphasis on high ADR and small TL, i.e. two targets are combined simultaneously. To increase the chances of getting a good ADR, a limit criterion LC is introduced so that only the patterns with Max Δ''< LC are selected (Figure 3). This is opposite to the traditional random test generation (RTG) where the patterns are selected according to Max Δ'. To slow down at the same time the growth of the test length, due to a number of other patterns satisfying the constraint Δ'' < LC, the patterns with Max Δ'' have to be selected

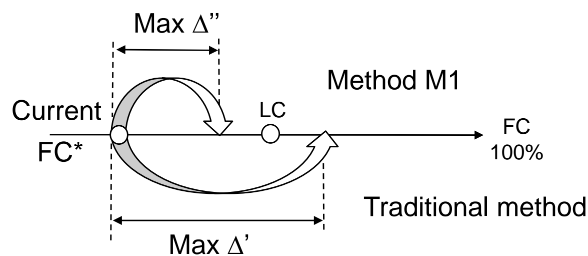

Figure 3 Comparison of M1 (Δ=max) with the traditional approach

The effects of the two criterions (the traditional Max Δ', and the proposed Max Δ''< LC) for a benchmark circuit c432 [**4**] are shown in Table 2 where TL is test length, FC is fault coverage, and ADR is average diagnostic resolution.

Table 2 Two criterions for selecting patterns (circuit c432)

Max Δ' (Trad. method)				Max Δ''< LC (M1)			
LC	**TL**	**FC%**	**ADR**	**LC**	**TL**	**FC%**	**ADR**
1	39	95.3	7.9	22	56	95.3	5.9
2	35	95	8.4	23	55	95.3	5.8
3	32	95.5	9	24	55	95.3	6.1
4	30	93.8	9.6	25	51	95.3	7
5	27	92.7	10.5	26	50	95.3	6.9
6	25	91.8	12.1	27	51	95.3	6.6
7	23	90.6	13.9	28	51	95.3	6.4

3.3.2. Effect of fault collapsing on the ADR

In Table 3 the result of the method described in section 3.3.1 are shown after taking into account fault collapsing. The number of gate-level stuck-at-faults in c432 is 974, and after fault collapsing – 616. The FC in Table 3 is calculated in relation to the number of faults after fault collapsing.

As we see, the impact of fault collapsing (in Table 3) on the diagnostic resolution is considerable.

Table 3 Influence of the fault collapsing on M1 (c432)

M1 (gate level faults)				M1 (collapsed faults)			
LC	**TL**	**FC%**	**ADR**	**LC**	**TL**	**FC%**	**ADR**
1	36	93	5	22	45	93	3.7
2	33	92.7	5.4	23	43	93	4
3	28	91.2	6.6	24	43	93	4.2
4	26	90.6	7.3	25	41	93	4.3
5	22	88.2	8.6	26	41	93	4.2
6	20	86.7	9.6	27	42	93	4.2
7	19	85.7	10.4	28	42	93	4

3.3.3. RTG with better ADR - M1 (Δ = min)

This is the same with what was described in section 3.3.1 except that in this case instead of taking patterns that meet the requirement of the LC where Δ=max, the patterns with Δ=min are selected, as illustrated in Figure 4.

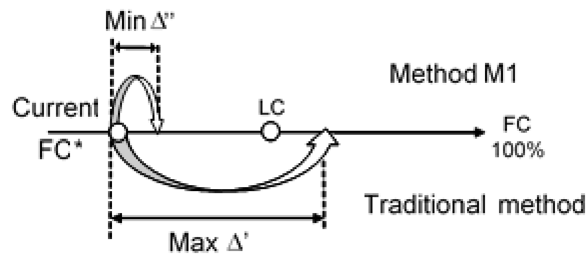

Figure 4 Comparison of M1 (Δ=min) with the traditional approach

3.3.4. Comprehensive result (M1).

The results of M1 with both configurations, Δ=max and Δ=min are presented and compared against the traditional random method. Three families have been used for the experiments ISCAS'85 [**4**], ISCAS'89 [**5**] and ITC'99 [**6**].

RTPG: Random Test Pattern Generator, **ADR**=Average Diagnostic Resolution, **FC**= Fault Coverage, **TL**=Test Length, **TIME**=Test generation time

Table 4 result for method M1 for ISCAS'85

		RTPG	M1 Δ=max	M1 Δ=min
c432	ADR	4.96	3.3	2.31
	FC	93.02	93.02	93.02
	TL	36	55	211
	TIME(S)	2.01	3.341	12.648
c499	ADR	2.33	2.19	2.03
	FC	99.33	99.33	99.33
	TL	84	90	288
	TIME(S)	4.62	5.697	28.897
c880	ADR	2.77	2.73	1.72
	FC	100	100	100
	TL	38	42	382
	TIME(S)	3.03	3.435	29.226
c1908	ADR	3.6	3.3	2.48
	FC	99.48	99.48	99.48
	TL	109	119	484
	TIME(S)	8.27	11.014	87.972
c2670	ADR	3.18	3.27	2.68
	FC	94.06	93.98	94.9
	TL	89	91	390
	TIME(S)	47.16	49.584	154.633
c3540	ADR	3.35	3.34	2.24
	FC	95.54	95.54	95.54
	TL	119	122	820
	TIME(S)	18.56	20.599	270.909
c5315	ADR	2.72	2.64	2.1
	FC	98.89	98.89	98.89
	TL	83	88	1043
	TIME(S)	25.78	31.337	552.499

In Table 4 where we have the result for ISCAS'85 [**4**] benchmark circuits, it is very obvious to see the trend when Δ=min (test patterns detecting the least faults are selected), the ADR is better when compared to the configuration where Δ=max. However the test generation time (TIME) is much longer because test patterns detecting the list faults that satisfy the limiting criteria LC are selected, so obviously this will take more time before the FC converges. Nonetheless both configurations of M1 have a better ADR for the ISCAS'85 [**4**] circuits when compared to the ADR of the normal random test set (RTPG) except for circuit c2670 when M1 has the configuration Δ=max.

Table 5 result for method M1 for ISCAS’89

		RTPG	M1 Δ=max	M1 Δ=min
s967mm	**ADR**	2.65	2.66	2.18
	FC	100	100	100
	TL	93	91	321
	TIME(S)	7.06	6.413	30.225
s1269mm	**ADR**	2.9	2.93	1.98
	FC	100	100	100
	TL	41	41	397
	TIME(S)	5.48	10.119	43.633
s1494mm	**ADR**	4.49	3.78	2.66
	FC	99.17	99.17	99.17
	TL	106	124	435
	TIME(S)	7.281	8.928	37.457
s3384mm	**ADR**	2.31	2.31	2.23
	FC	96.69	96.69	96.36
	TL	46	48	246
	TIME(S)	23.74	23.054	62.926
s13207mm	**ADR**	6.75	6.93	6.25
	FC	98.19	98.2	98.2
	TL	412	407	1137
	TIME(S)	167.50	1506.242	6651.089
s15850mm	**ADR**	3.2	3.22	2.78
	FC	95.05	95.48	94.85
	TL	375	402	1414
	TIME(S)	417.32	2302.948	13397.131

Just like in Table 4, we have almost the same situation in Table 5 which has the result of the ISCAS’89 [**5**] benchmark circuit. When Δ=min (test patterns detecting the least faults are selected) the ADR is better when compared to the configuration where Δ=max, however the test generation time (TIME) is much longer due to the fact that test patterns detecting the least faults that satisfy the limiting criterion (LC) are selected. This is also the reason for the shorter test generation time when M1 has the configuration Δ=max because patterns detecting the most faults that satisfy the LC are selected instead. With the configuration Δ=max for M1, the ADR does not improve very much for the ISCAS’89 [**5**] circuits when compared with the ADR of the normal random test.

.

Table 6 result for method M1 for ITC'99

		RTPG	M1 Δ=max	M1 Δ=min
b04	**ADR**	3.31	2.65	1.87
	FC	98.52	98.52	98.52
	TL	74	81	340
	TIME	8.07	10.424	48.892
b05	**ADR**	5.1	5.14	4.29
	FC	77.52	77.52	77.52
	TL	71	75	307
	TIME(S)	20.16	21.257	126.591
b07	**ADR**	2.83	2.89	2.09
	FC	97.09	97.09	97.09
	TL	44	43	221
	TIME	4.46	4.134	18.225
b11	**ADR**	5.07	4.44	2.69
	FC	95.37	95.37	95.37
	TL	77	81	301
	TIME	12.21	11.53	58.913
b12	**ADR**	2.65	2.71	1.95
	FC	99.15	99.12	99.06
	TL	129	134	549
	TIME	29.67	31.459	157.005
b14	**ADR**	2.81	2.82	2.16
	FC	91.34	91.41	91.01
	TL	542	554	2230
	TIME	1523.73	1562.862	15130.59
b15	**ADR**	3.57	3.48	2.75
	FC	90.99	91.08	90.02
	TL	462	475	2474
	TIME	947.09	1086.828	16117.49

3.3.5. Observations and summary for method M1

From all the results presented in Table 4, Table 5 and Table 6, the configuration where Δ=min has the best ADR and this can be attributed to growing the FC with smaller steps (Δ=min) this is also the reason why the test generation time (TIME) is very long. Additional test patterns improved the ADR of the traditional method. The ADR also improved for the case when Δ=max, although not in all cases and the level of improvement is not as good as when Δ=min.

One challenge encountered with M1 was with finding the appropriate value of the LC. This value was different for different circuit models hence it required several experiments with varying values to find the best value for the LC. For very small values of LC some experiments did not yield any test vectors while in other cases test vectors were found

however the FC was much below the maximum achievable. For example if you take a look at Table 2 you will notice that the LC values for M1 are much higher than the traditional method, e.g. the minimum value of LC was 22, and this was the least value that yielded the maximum FC obtainable..

3.4. RTG with better ADR - M2

To resolve the problem of small values of LC in M1, in M2 the first test pattern is selected instead without the constraint $\Delta < LC$, and then starting from the second pattern the constraint is taken into consideration when selecting new patterns (Figure 1b). To also reduce deadlocks when searching for suitable patterns the selection process in M2 is made more flexible by allowing the value of LC to change dynamically during test generation in cases where a test vector has not been found after a period of time. Figure 5 in the next page shows the flow chart for M2.

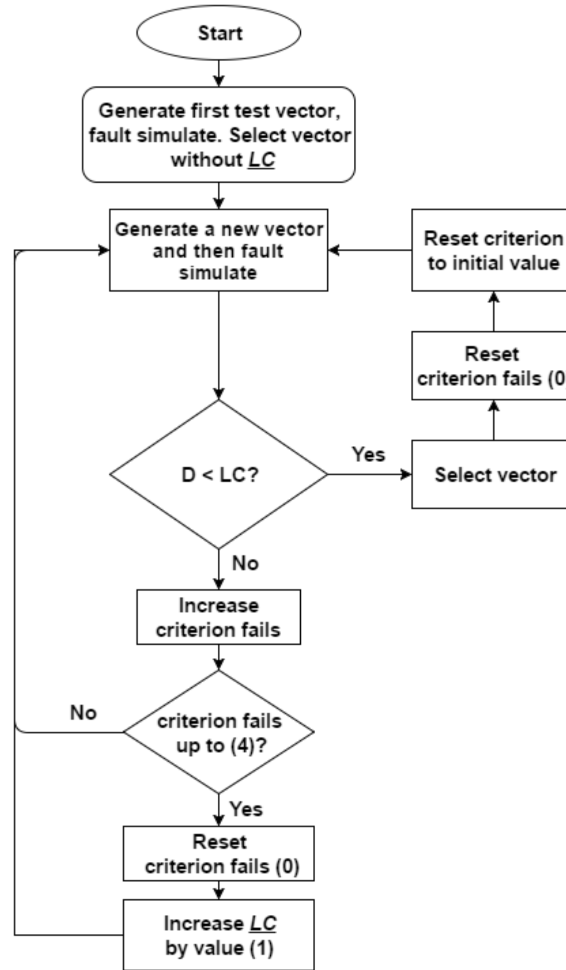

Figure 5 Flow chart for method M2

3.4.1. Comprehensive result (M2).

The same configurations that were used in M1 where the patterns that satisfied the LC with configuration Δ=max and Δ=min were selected, was also experimented with in M2. The following tables show the results for the following families ISCAS'85 [**4**], ISCAS'89 [**5**] and ITC'99 [**6**]. The traditional random pattern generation method is compared against the two configurations of M2.

Table 7 result for M2 for ISCAS'85

		RTPG	M2 Δ=max	M2 Δ=min
c432	**ADR**	4.96	2.5	2.28
	FC	93.02	93.02	93.02
	TL	36	102	209
	TIME(S)	2.01	14.962	11.39
c499	**ADR**	2.33	2.09	2.03
	FC	99.33	99.33	99.33
	TL	84	99	282
	TIME(S)	4.62	10.426	31.527
c880	**ADR**	2.77	2.13	1.72
	FC	100	100	100
	TL	38	56	404
	TIME(S)	3.03	6.601	33.661
c1908	**ADR**	3.6	2.77	2.48
	FC	99.48	99.48	99.48
	TL	109	176	483
	TIME(S)	8.27	30.005	57.19
c2670	**ADR**	3.18	3.25	2.68
	FC	94.06	93.91	94.9
	TL	89	90	389
	TIME(S)	47.16	50.378	157.986
c3540	**ADR**	3.35	3.37	2.24
	FC	95.54	95.54	95.54
	TL	119	122	819
	TIME(S)	18.56	22.72	266.176
c5315	**ADR**	2.72	2.77	2.09
	FC	98.89	98.89	98.89
	TL	83	86	1117
	TIME(S)	25.78	32.262	584.397

In Table 7Table 4 where we have the result for ISCAS'85 [**4**] benchmark circuits, it is very obvious to see the trend when Δ=min (test patterns detecting the least faults are selected) the ADR is better when compared to the configuration where Δ=max, however the test generation time (TIME) is much longer because patterns that satisfy the LC and that detect the least fault are selected so this takes more time. Both configurations of M2 have a better ADR when compared to the ADR of the normal random test set (RTPG) except for circuit's c2670, c3540, and c5315 the configuration Δ=max does not produce a better ADR when compared the normal random test set.

Table 8 result for M2 for ISCAS'89

		RTPG	M2 Δ=max	M2 Δ=min
s967mm	**ADR**	2.65	2.66	2.18
	FC	100	100	100
	TL	93	92	320
	TIME	7.06	6.319	33.614
s1269mm	**ADR**	2.9	3	2.01
	FC	100	100	100
	TL	41	44	418
	TIME(S)	5.48	5.367	41.9
s1494mm	**ADR**	4.49	3.26	2.65
	FC	99.17	99.17	99.17
	TL	106	161	437
	TIME	7.281	18.37	46.01
s3384mm	**ADR**	2.31	2.32	2.24
	FC	96.69	96.69	96.24
	TL	46	44	234
	TIME	23.74	20.389	65.98
s13207mm	**ADR**	6.75	6.82	6.23
	FC	98.19	98.2	98.2
	TL	412	413	1151
	TIME	167.50	297.972	5885.308
s15850mm	**ADR**	3.2	3.19	2.8
	FC	95.05	95.5	94.85
	TL	375	402	1406
	TIME	417.32	1671.041	13996.072

Table 8 shows the results for ISCAS'89 [**5**] family and we can see from the table that the ADR of M2 when the configuration Δ=min is used is better than when Δ=max. Also we can observe from the result that the ADR of the normal random test (RTPG) is worse than M2 with configuration Δ=min however it is almost similar in some cases to M2 with configuration Δ=max. For instance for all circuits, apart from circuit s1494mm the ADR for M2 with configuration Δ=max does not have a better ADR when compared to the ADR of the normal test set (RTPG).

Table 9 result for M2 for ITC'99

		RTPG	M2 Δ=max	M2 Δ=min
b04	ADR	3.31	2.68	1.88
	FC	98.52	98.46	98.52
	TL	74	80	339
	TIME	8.07	10.85	50.994
b05	ADR	5.1	4.96	4.29
	FC	77.52	77.52	77.52
	TL	71	73	306
	TIME(S)	20.16	21.899	147.168
b07	ADR	2.83	2.72	2.11
	FC	97.09	97	97.09
	TL	44	45	220
	TIME	4.46	4.326	21.36
b11	ADR	5.07	4.86	2.69
	FC	95.37	95.2	95.37
	TL	77	79	300
	TIME	12.21	11.843	60.356
b12	ADR	2.65	2.78	1.96
	FC	99.15	99.02	99.06
	TL	129	133	567
	TIME	29.67	27.763	159.898
b14	ADR	2.81	2.85	2.16
	FC	91.34	91.38	91.01
	TL	542	553	2229
	TIME	1523.73	1601.621	15356.816
b15	ADR	3.57	3.48	2.75
	FC	90.99	91.06	90.02
	TL	462	474	2473
	TIME	947.09	1085.863	16449.786

3.4.2. Observations and summary for M2

The introduction of a dynamic LC during test pattern generation made it easier to look for a suitable LC value. However the result of method M2 did not improve considerably when compared to method M1. Analysing the results of M2 and M1 closely showed that both methods performed well when the test vectors are selected with $\Delta_{min} <$ LC.

3.5. RTG with better ADR - M3

The very similar result when comparing M1 and M2, inspired the creation of M3. In M3 the LC is removed entirely and test vectors are ranked in ascending order with Δ=min where Δ is the amount of new faults detected (previously described in section 3.2). There is no criterion constraint LC.

3.5.1. Comprehensive result (M1 M2 and M3)

In this section the following tables compare the results of M3 with the best result of M1 and M2. The traditional method for random test pattern generation is also placed in the table to show the improvement each method introduced. The results in the following tables are presented for the following benchmark families' ISCAS'85 [**4**], ISCAS'89 [**5**] and ITC'99 [**6**].

Table 10 ICAS'85 family, Comparing M3 with best results of M1 and M2

		RTPG	M1	M2	M3
c432	**ADR**	4.96	2.31	2.28	2.31
	FC	93.02	93.02	93.02	93.02
	TL	36	211	209	211
	TIME(S)	2.01	12.65	11.39	11.13
c499	**ADR**	2.33	2.03	2.03	2.03
	FC	99.33	99.33	99.33	99.33
	TL	84	288	282	288
	TIME(S)	4.62	28.90	31.53	25.04
c880	**ADR**	2.77	1.72	1.72	1.72
	FC	100	100	100	100
	TL	38	382	404	382
	TIME(S)	3.03	29.23	33.66	25.56
c1908	**ADR**	3.60	2.48	2.48	2.48
	FC	99.48	99.48	99.48	99.48
	TL	109	484	483	469
	TIME(S)	8.27	87.97	57.19	77.19
c267	**ADR**	3.18	2.68	2.68	2.68
	FC	94.06	94.90	94.90	94.90
	TL	89	390	389	390
	TIME(S)	47.16	154.63	157.99	140.01
c3540	**ADR**	3.35	2.24	2.24	2.24
	FC	95.54	95.54	95.54	95.54
	TL	119	820	819	820
	TIME(S)	18.56	270.91	266.18	238.32
c5315	**ADR**	2.72	2.10	2.09	2.09
	FC	98.89	98.89	98.89	98.89
	TL	83	1043	1117	1118
	TIME(S)	25.78	552.50	584.40	652.60

Looking at Table 10 we can observe that all three methods, M1, M2 and M3 have similar ADR. Also comparing the ADR of all three methods to the ADR of the normal test set (RTPG) we can see that they all have a better ADR. In general all three methods have almost the same test TL except for circuit c5315 where M2 and M3 have the worse TL compared to M1. For circuit c880 M2 has the worse TL and M3 has the best TL for circuit c1908. As for the test generation time (TIME) M3 has the best or shortest time (apart from circuit c1908) when compared M1 and M2.

Table 11 ICAS'89 family, Comparing M3 with best results of M1 and M2

		RTPG	M1	M2	M3
s967mm	**ADR**	2.65	2.18	2.18	2.18
	FC	100	100	100	100
	TL	93	321	320	321
	TIME	7.06	30.23	33.61	41.84
s1269mm	**ADR**	2.90	1.98	2.01	2.01
	FC	100	100	100	100
	TL	41	397	418	419
	TIME(S)	5.48	43.63	41.90	52.93
s1494mm	**ADR**	4.49	2.66	2.65	2.66
	FC	99.17	99.17	99.17	99.17
	TL	106	435	437	435
	TIME	7.3	37.5	46.0	53.4
s3384mm	**ADR**	2.31	2.23	2.24	2.24
	FC	96.69	96.36	96.24	96.24
	TL	46	246	234	235
	TIME	23.74	62.93	65.98	80.59
s13207mm	**ADR**	6.75	6.25	6.23	6.26
	FC	98.19	98.20	98.20	98.20
	TL	412	1137	1151	1050
	TIME	167.50	6651.09	5885.31	1114.31
s15850mm	**ADR**	3.20	2.78	2.80	2.90
	FC	95.05	94.85	94.85	95.21
	TL	375	1414	1406	1305
	TIME	417.32	13397.13	13996.07	3317.01

Looking at Table 11 we can observe that all three methods, M1, M2 and M3 have similar ADR except for circuit s15850mm where M3 has a slightly worse ADR. If we compare the ADR of all three methods to the ADR of the normal test set (RTPG) we can see that they all have a better ADR. The TL for all three methods are similar for only circuit s967mm and s1494mm but different in other cases. For the bigger circuits s13207mm and s15850mm, M3 has the best TL. As for the test generation time (TIME) M3 has the best or shortest time for the bigger circuits (s13207mm and s15850mm) but the worse test generation time for all other circuits.

Table 12 ITC'99 family, Comparing M3 with best results of M1 and M2

		RTPG	M1	M2	M3
b04	ADR	3.31	1.9	1.88	1.87
	FC	98.52	98.52	98.52	98.52
	TL	74	340	339	340
	TIME	8.07	48.89	50.99	63.69
b05	ADR	5.10	4.29	4.29	4.29
	FC	77.5	77.5	77.5	77.5
	TL	71	307	306	307
	TIME	20.16	126.59	147.17	184.74
b07	ADR	2.83	2.09	2.11	2.09
	FC	97.1	97.1	97.1	97.1
	TL	44	221	220	221
	TIME	4.46	18.23	21.36	26.96
b11	ADR	5.07	2.69	2.69	2.69
	FC	95.37	95.37	95.37	95.37
	TL	77	301	300	301
	TIME	12.21	58.91	60.36	76.13
b12	ADR	2.65	1.95	1.96	1.96
	FC	99.15	99.06	99.06	99.06
	TL	129	549	567	568
	TIME	29.67	157.01	159.90	202.83
b14	ADR	2.81	2.16	2.16	2.16
	FC	91.34	91.01	91.01	91.01
	TL	542	2230	2229	2230
	TIME	1523.73	15130.59	15356.82	18031.69
b15	ADR	3.57	2.75	2.75	2.75
	FC	90.99	90.02	90.02	90.02
	TL	462	2474	2473	2474
	TIME	947.09	16117.49	16449.79	19060.77

3.6. Observations and summary for M1, M2 and M3

The methods presented in this chapter for random test pattern generation attempt to generate test pattern with very good ADR value using the FC as the feedback or cost function. While the fault coverage may not be the best cost function for use, the result of the experiments show that the overall ADR of the generated test set improves. Another important point to take note of that is common with methods M1, M2 and M3 is that the average diagnostic resolution is improved during the generation of the test set and not after the test generation, i.e. the RTG is modified to generate the vectors for testing but with very good ADR; as the RTG runs it selects only vectors that detect minimum number of faults within a fault group so by doing this larger fault groups are avoided. This leads to a better ADR in the end.

Method M3 has very similar result when compared to methods M1 and M2 for all the circuit models that were experimented with. Note that the results of M1 and M2 are based

on the experiment whereby Δ=min. Previously the symbol Δ was introduced as the increment in the FC due to the contribution of a test pattern. For methods M1 and M2 a limiting criterion LC was introduced such that Δ must be within this LC (Δ=max$<$LC or Δ=min$<$LC), however in the tables comparing M1, M2 and M3, the configuration where the test vectors were selected with Δ=min was selected since it has the best result. This was also the source of inspiration to implement method M3 but without a limiting criterion (LC); instead the vectors were selected with Δ=min.

4. Improving the ADR after RTG

The methods that were presented in Chapter 3 are for generating test patterns with better ADR. In this chapter two methods that aspire to achieve a better ADR after a test set has been generated are presented.

We refer to the methods with the following code names:

1. Improving ADR of a randomly generated test set by applying random test vectors and calculation of ADR in each step, referred to as A1

2. Improving ADR of a randomly generated test set by applying random test vectors and a cost function to estimate ADR in each step, referred to as A2

The methods presented here are applied to an already existing test set with a high FC and short TL, and then these methods try to optimize the ADR by adding test vectors that can improve the ADR.

After the description of the A1 and A2, section 4.3 and section 4.4 compares the results of both methods.

4.1. Improving the ADR after RTG - A1

From the result of the previous experiments it is clear that additional TL improves the ADR but not all of the additional test patterns would contribute to the improvement of ADR because the ADR was not the primary cost function used for selecting the patterns in methods M1, M2 and M3 - instead Δ (Number of new faults detected by a test vector) was used, and this is mostly related to the FC.

The FC and DR are related, for example a test vector that detects fewer faults within a group has a better DR than a test vector that detects more faults within a group, and of course selecting test patterns with this criteria indirectly improves the overall ADR, but using only this basis will not give the best result for both ADR and TL. The results in the previous section already serves as proof. In as much as we want a very good ADR we

still want to keep the cost low by making sure that the TL is small enough and also the speed at which the test set is generated is fast enough.

The method presented under this section takes the test set that has been generated with a normal RTG (a test set with maximum FC and small TL) and then attempts to add semi deterministic patterns in a random way to the test set. The aim is to improve the ADR; also since it is computationally expensive to generate deterministic patterns which will target the ADR, the random approach is desired because with it, we can bypass the costly computation which not only improves the speed but also improves the ADR as well.

In summary A1 basically generates a test vector, fault simulate to extract the fault vector then it introduces this fault vector into the already existing test set that was previously generated by the traditional RTG. It then calculates the overall ADR and if the newly introduced test vector and corresponding fault vector improves the ADR the test vector is added otherwise the vector is removed. This process continues until there are no improvements for a duration of time or for how long the test engineer wishes to run the process.

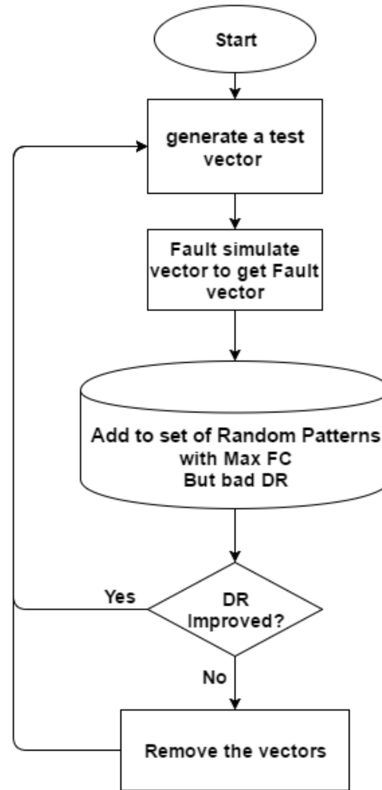

Figure 6 Flow chart describing A1

4.2. Improving the ADR after RTG - A2

In this method a cost function is used to assign weights to candidate test vectors then the best vectors are selected and then added to improve the ADR. The number of new test vectors that should be added to the original test set must be specified. The algorithm used in A2 is a greedy algorithm that always selects the best test vectors first; such vectors have the highest weights after each vector has been subjected to the cost function.

Unlike in method A1 in method A2 the direct evaluation of the ADR in each step is avoided, so this method is a lot faster compared to A1 which evaluates ADR for every step. Referring back to section 3.1 where the calculation of ADR was described, the faults detected in a fault vector (the set of faults that are detected by a test vector) belong to groups. The ADR improves when the number of groups increases. For example given a set of fault vector that has two groups, if by adding two new vectors the total group splits further into four groups then the ADR will improve. Since the aim is to improve the ADR

which is made worse by larger groups, A2 tries to search for such vectors that can effectively breakup larger groups into smaller groups because by so doing the ADR can be improved further.

All the vectors in the original test set are weighed by the number of faults detected and then the vectors are ranked from highest to lowest. Table 13 shows an example of how a fault vector is weighed. For each of the fault vectors in the original test set, a new pack of vectors is generated and then fault simulated to extract their corresponding fault vectors. In order to estimate which new fault vectors will split the group of a given fault vector further, the weighing function is used. The weighing function is described in section 4.2.1.

Table 13. Example of weighted fault vectors

Fault vector	1	2	3	4	5	6	7	8	9	10	11	Weight
1	1	1	1	1		1	1	1		1		8
2			1					1			1	3
3					1	1					1	3
4					1				1			2

4.2.1. Weighing function for selecting additional test vectors

Table 14 Example of a large fault group

Fault vector	1	2	3	4	5	6	7	8	9	10	11	Weight
i	1	1	1	1	x	1	1	1	x	1	x	8

Using Table 14 above the entry represents a fault vector which detects 8 out of 11 faults. For simplicity denote a stuck-out fault detection by ‘1’ and no fault detected by ‘x’.

If the number of faults detected are counted then we get eight (8) which will be the weight of this fault vector.

Also there are eight members in the group {f1, f2, f3, f4, f6, f7, f8 and f10} assuming there are no other vectors with overlapping members. To improve the ADR we need to split the group of eight members into several groups.

In other to determine the contribution of a test vector, the information on how much a group can be broken apart is determined by measuring its entropy using equation (2) below (Shannon's equation for information entropy).

$$I = - p \log_2 p - (1-p) \log_2 (1-p) \qquad (2)$$

This entropy information extraction is what has been referred to as the weight. Equation (2) is the general case for the amount of information.

It is important to note that from equation (2) assuming the expression $(1-p) = 0$, then $\log_2(1-p) = \log_2 0 \Rightarrow \infty$, however $(1-p) \log_2 (1-p) \Rightarrow 0 \log_2 0 \Rightarrow 0$.

Note that for every test vector there is an equivalent fault vector, i.e. every test vector v maps to its fault vector f after fault simulation. In equation (2) above p denotes the probability of overlap a candidate fault vector has with the base fault vector that we are interested in splitting. As an example assume that a given fault vector has a total of 8 detected faults in a single group and a candidate fault vector which will split the base vector has a total of 10 detected faults but only 4 out of the 10 overlaps with the base vector, we consider the probability p of overlap as shown in equation (3).

$$\frac{Candidate\ vector\ number\ of\ overlapping\ faults}{Total\ number\ of\ faults\ in\ base\ vector} \qquad (3)$$

Using equation (3) the probability will be 4/8 which gives ½.

To simplify the entropy extraction for each candidate fault vector, the logarithmic calculation is avoided by using a simplified process which happens in two phases, the first phase extracts the number of overlapping faults a candidate fault vector has with the base fault vector; this is achieved by using a logic operation defined in Table 15 and then the second phase uses the base vector to normalize the weight of the candidate vector.

The logic operation is used to find the initial number of overlapping fault a candidate vector has with the base vector. If the candidate vector, after undergoing the second phase is selected then the base vector is split in two and produces two fault vectors in the end. The first is a fault vector having the number of overlapping faults with the base

fault vector and the second is composed of the remaining faults in the base fault vector that do not overlap with the candidate fault vector.

For example let set *V* be the set of the original test vectors and let set *F* be the set of the equivalent fault vectors that correspond to *V*. For each test vector *v* there is a corresponding fault vector *f* which is the set of all the faults detected by test vector *v*. After *f* has been subjected to the cost function and it is found to be very useful in splitting a larger group, the corresponding test vector *v* that maps to *f* is put into the set *V* then *f* is split into two parts. Splitting *f* will yield f_1 and f_2 so after splitting, *f* is removed from set *F* and f_1 and f_2 are reintroduced into set *F* for the next iteration. The set *F* is then sorted in descending order using the weight of each fault vector *f*.

The truth table for the logic operation is defined in Table 15.

Table 15 truth table for logic function

x1	x2	result
0	0	0
1	1	1
0	x	x
1	x	x
1	0	x

In Table 15, value 0 refers to S-A-0 and value 1 refers to S-A-1. Refer to Table 16 for an example of the logic operation and three candidate vectors.

Table 16 Example of logic function with candidate vector

Fault vector	1	2	3	4	5	6	7	8	9	10	11	Original faults	Overlapping faults
Vector with large group	1	1	1	1	x	1	1	1	x	1	x	8	Not applicable
Candidate1	1	1	1	1	1	x	x	x	1	x	x	6	4
Candidate2	1	1	1	1	x	1	1	1	x	1	x	8	8
Candidate3	x	x	x	x	1	1	1	1	1	x	1	6	3

To show the effect of the logic operation I have intentionally added a candidate vector (Candidate2) which is the same as the fault vector that needs to be improved. Using the truth table in Table 15 since both vectors are one and the same the result will also be the

same so this gives an initial weight value of eight (8) overlapping faults which is the maximum achievable for this example.

Obviously selecting this fault vector just because it has the highest overlapping faults compared to the other candidate does not add any improvement to the ADR because it is simply a duplicate vector. The second phase of the weighing process, which applies equation (2) is used to extract the entropy and then each candidate vector is finally ranked.

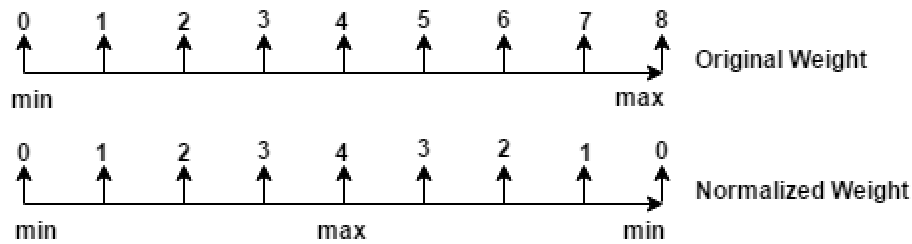

Figure 7 normalizing the weights

Using equation (2), a candidate fault vector that has complete overlap with the base vector e.g. candidate2 in Table 16 will have a probability of 1, substituting this value into equation (2) will yield

$$I = -1 \log_2 1 - (0) \log_2 0 = 0.$$

Hence a candidate vector that fully overlaps with the base vector does not help to split the base vector.

Figure 7 shows the original weight vs. normalized weight. The max weight value (8) in the figure is based on the example that was illustrated by Table 16 whereby the vector that needs to be improved has a weight of 8. After subjecting all the candidate vectors to the normalization function then the former candidate vector (Candidate2) which had a higher weight will become zero (0) and then Candidate1 and Candidate 3 will be 4 and 3 (1 and 0.9 if the logarithm function is used). With this arrangement Candidate1 will become the best since it is able to remove four members from the group of the original fault vector. This effectively splits the group and will result in a better ADR.

4.3. Comprehensive result of methods A1 and A2

ADR = Average Diagnostic Resolution, **FC**= Fault Coverage,
EP = Extra Patterns (Additional patterns added to the original test set)
TL = Test Length, **TIME** = Test generation time (h=hours, s=seconds)

Table 17. Comparing method A1 and A2 with ISCAS' 85 circuits

		RTPG	A1	A2
c432	ADR	5.0	2.3	2.3
	FC	93.0	93.0	93.0
	EP	-	63	175
	TL	36	99	211
	TIME	2.0s	01:17:07h	2.1s
c499	ADR	2.3	2.0	2.3
	FC	99.3	99.3	99.3
	EP	-	13	204
	TL	84	97	288
	TIME(S)	4.6s	02:38:54h	6.0s
c880	ADR	2.8	1.7	1.8
	FC	100.0	100.0	100.0
	EP	-	45	344
	TL	38	83	382
	TIME	3.0s	03:11:05h	7.3s
c1908	ADR	3.6	2.7	3.3
	FC	99.5	99.5	99.5
	EP	-	48	360
	TL	109	157	469
	TIME	8.3s	05:45:54h	18.4s
c2670	ADR	3.2	3.0	3.0
	FC	94.1	94.1	94.1
	EP	-	14	301
	TL	89	103	390
	TIME	47.2s	08:02:13h	23.7s
c3540	ADR	3.4	2.3	2.2
	FC	95.5	95.5	95.5
	EP	-	76	701
	TL	119	195	820
	TIME	18.6s	14:14:58h	71.2s
c5315	ADR	2.7	2.22	2.1
	FC	98.9	98.89	98.9
	EP	-	40	1035
	TL	83	123	1118
	TIME	25.8s	38:14:21h	187.4s

In Table 17 we can see that methods A1 and A2 have better ADR compared to the normal test set. On the other hand A2 is much faster than A1 due to the simplified cost function

it uses but even though A1 calculates the ADR each time a new candidate vector is introduced this gives it an advantage of very short TL.

Table 18 Comparing method A1 and A2 with ISCAS' 89 circuits

		RTPG	A1	A2
s967mm	**ADR**	2.7	2.2	2.2
	FC	100.0	100.0	100.0
	EP	-	31	228
	TL	93	124	321
	TIME	7.1s	02:04:26h	6.0s
s1269mm	**ADR**	2.9	2.0	2.0
	FC	100.0	100.0	100.0
	EP	-	43	378
	TL	41	84	419
	TIME(S)	5.5s	02:40:42h	13.8s
s1494mm	**ADR**	4.5	2.8	2.7
	FC	99.2	99.2	99.2
	EP	-	83	329
	TL	106	189	435
	TIME	7.3s	05:25:59h	9.2s
s3384mm	**ADR**	2.3	2.3	2.2
	FC	96.7	96.7	96.7
	EP	-	4	189
	TL	46	50	235
	TIME	23.7s	07:51:30h	20.6s
s13207mm	**ADR**	6.8	6.6	6.5
	FC	98.2	98.2	98.2
	EP	-	18	638
	TL	412	430	1050
	TIME	167.5s	134:41:0h	394.4s
s15850mm	**ADR**	3.2	3.1	3.0
	FC	95.1	95.1	95.1
	EP	-	7	930
	TL	375	382	1305
	TIME	417.3s	45:39:0h	795.6s

In Table 18 above the result of the traditional random test for testing (column RTPG) has been placed alongside with methods A1 and A2 and the reason for this, is to simply show the level of improvement the two methods proposed in this chapter can introduce. As you can see the two approaches improve the ADR. For example for circuit model s1494mm the ADR improved by almost 50% in both cases. A1 has a better test generation time

(TIME) because of the simplified cost function it uses but A1 on the other hand has a compact TL.

Table 19. Comparing A1 and A2 with ITC' 85 circuits

		RTPG	A1	A2
b04	**ADR**	3.3	1.9	1.9
	FC	98.5	98.5	98.5
	EP	-	30	266
	TL	74	104	340
	TIME	8.1s	02:45:32h	10.6s
b05	**ADR**	5.1	4.3	4.3
	FC	77.5	77.5	77.5
	EP	-	42	236
	TL	71	113	307
	TIME(S)	20.2s	04:58:33h	21.6s
b07	**ADR**	2.8	2.0	2.3
	FC	97.1	97.1	97.1
	EP	-	36	177
	TL	44	80	221
	TIME	4.5s	03:30:22h	4.5s
b11	**ADR**	5.1	2.7	2.8
	FC	95.4	95.4	95.4
	EP	-	67	224
	TL	77	144	301
	TIME	12.1s	05:57:55h	10.3s
b12	**ADR**	2.7	2.0	2.0
	FC	99.15	99.15	99.15
	EP	-	56	439
	TL	129	182	568
	TIME	29.7s	11:45:22h	36.6s
b14	**ADR**	2.81	2.64	2.3
	FC	91.34	91.34	91.34
	EP	-	16	1688
	TL	542	558	2230
	TIME	1523.7s	90:34:57h	3443.9s
b15	**ADR**	3.6	3.4	2.8
	FC	91.0	91.0	91.0
	EP	-	13	2012
	TL	462	475	2474
	TIME	947.1s	45:31:29h	2610.4s

From Table 19 we can also see that on the average methods A1 and A2 improve the ADR by a reasonable magnitude, for instance both methods introduce approximately 50% improvement to the ADR for circuits models b04 and b11.

4.4. Comparing methods A1 and A2

After running a reasonable number of experiments with a variety of circuits, the results of both methods A1 and A2 have been captured in Table 17, Table 18 and Table 19. The ADR of the original test set generated by a traditional random ATPG is also present on the table to show how much improvement the ADR can be benefit from by using either methods A1 or A2.

From the results A1 has the best ADR when compared to A2. Since both methods use an already existing test set the FC is the same the only difference would be the ADR. A1 has a compact TL when compared to A2 and this is because, in A1 each time a new test vector is introduced the ADR of the entire test set is calculated but in A2, the ADR is estimated using a cost function. The biggest drawback of A1 is that it takes too much time. The reason for this is because of the ADR calculation in each step and this is a very expensive operation. A2 avoids this expensive calculation by approximating the ADR with a faster and less expensive cost function, hence the speed.

In conclusion A1 trades-off speed for better ADR and shorter TL, while A2 trades-off better ADR and shorter TL for speed.

5. Experimental Results

In this chapter the best method (M3) from chapter 3 for generating random test set with good ADR and the two methods (A1 and A2) for improving the ADR of an already generated test set are presented. The three approaches are then compared with the ADR obtained from test set generated with a random ATPG and with deterministic ATPG. The results show a high potential in terms of improved ADR resulting from the proposed methods. A discussion accompanies the results presented which highlights the strengths and weakness of the proposed methods.

5.1. Comparison of proposed methods

The following tables tries to compare the best methods from all the proposed methods using the ISCAS'85 [**4**], ISCAS'89 [**5**] and ITC'99 [**6**] benchmark circuits. Also to show the contribution or improvement to the ADR each proposed method introduces, the ADR of the test set generated with random ATPG and deterministic ATPG are also captured in the table.

The following explain the meaning of the acronyms that are used in Table 20, Table 21, and Table 22.

RTPG: Random Test Pattern Generator

DTPG: Deterministic Test Pattern Generator.

ADR: Average Diagnostic Resolution.

FC: Fault Coverage.

EP: Extra Patterns.

TL: Test Length.

TIME(h,s): Test generation time (h=hours, s=seconds).

Table 20. Comparing ADR of best proposed methods with ADR of original test set using ISCAS'85

		RTPG	DTPG	M3	A1	A2
c432	**ADR**	5.0	3.3	2.3	2.3	2.3
	FC	93.0	93.0	93.0	93.0	93.0
	EP	-	-	-	63	175
	TL	36	84	211	99	211
	TIME	2.0s	45.7s	11.1s	01:17:07h	2.1s
c499	**ADR**	2.3	2.3	2.0	2.0	2.2
	FC	99.3	99.3	99.3	99.3	99.3
	EP	-	-	-	13	204
	TL	84	132	288	97	288
	TIME(S)	4.6s	82.8s	25.0s	02:38:54h	6.0s
c880	**ADR**	2.8	2.0	1.7	1.7	1.8
	FC	100.0	100.0	100.0	100.0	100.0
	EP	-	-	-	45	344
	TL	38	77	382	83	382
	TIME	3.0s	1.2s	25.6s	03:11:05h	7.3s
c1908	**ADR**	3.6	3.5	2.5	2.7	3.3
	FC	99.5	99.5	99.5	99.5	99.5
	EP	-	-	-	48	360
	TL	109	143	469	157	469
	TIME	8.3s	41.6s	77.2s	05:45:54h	18.4s
c2670	**ADR**	3.2	2.9	2.7	3.0	3.0
	FC	94.1	95.5	94.9	94.1	94.1
	EP	-	-	-	14	301
	TL	89	155	390	103	390
	TIME	47.2s	167.0s	140.0s	08:02:13h	23.7s
c3540	**ADR**	3.4	2.6	2.2	2.3	2.2
	FC	95.5	95.5	95.5	95.5	95.5
	EP	-	-	-	76	701
	TL	119	205	820	195	820
	TIME	18.6s	339.4s	238.3s	14:14:58h	71.2s
c5315	**ADR**	2.7	2.3	2.1	2.22	2.1
	FC	98.9	98.9	98.9	98.89	98.9
	EP	-	-	-	40	1035
	TL	83	171	1118	123	1118
	TIME	25.8s	13.41s	652.6s	38:14:21h	187.4s

Table 21 holds the result for the ISCAS'85 [**4**] family. Columns RTPG and DTPG hold result for normal test set for testing (not for diagnosis). DTPG has a better ADR compared to RTPG. We can see also that generally the three methods, M3, A1 and A2 all have better ADR. For example there is an improvement of over 50% for circuit c432macro. With the same TL M3 has a slightly better ADR than A2 but A2 has the better test generation time.

A1 on the other hand has the worse test generation time but compared to M3 and A2 has a compact TL.

Table 21. Comparing ADR of best proposed methods with ADR of original test set using ISCAS'89

		RTPG	DTPG	M3	A1	A2
s967mm	ADR	2.7	2.3	2.2	2.2	2.2
	FC	100	100	100	100	100
	EP	-	-	-	31	228
	TL	93	124	321	124	321
	TIME	7.1s	0.02s	41.8s	02:04:26h	6.03s
s1269mm	ADR	2.9	2.2	2.0	2.0	2.0
	FC	100	100	100	100	100
	EP	-	-	-	43	378
	TL	41	68	419	84	419
	TIME(S)	5.48s	0.1s	52.9s	02:40:42h	13.8s
s1494mm	ADR	4.5	3.6	2.7	2.8	2.7
	FC	99.2	99.2	99.2	99.2	99.2
	EP	-	-	-	83	329
	TL	106	175	435	189	435
	TIME	7.3s	0.04s	53.4s	05:25:59h	9.2s
s3384mm	ADR	2.3	2.2	2.2	2.3	2.2
	FC	96.7	100.0	96.2	96.7	96.7
	EP	-	-	-	4	189
	TL	46	113	235	50	235
	TIME	23.7s	0.03s	80.6s	07:51:30h	20.6s
s13207mm	ADR	6.8	6.5	6.3	6.6	6.5
	FC	98.2	98.2	98.2	98.2	98.2
	EP	-	-	-	18	638
	TL	412	600	1050	430	1050
	TIME	167.5s	293.5s	1114.3s	134:41:0h	394.3s
s15850mm	ADR	3.2	3.0	2.9	3.1	3.0
	FC	95.1	95.7	95.2	95.1	95.1
	EP	-	-	-	7	930
	TL	375	541	1305	382	1305
	TIME	417.3s	1516.9s	3317.0s	45:39:0h	795.6s

In Table 21 columns RTPG and DTPG hold result for normal test set for testing (not for diagnosis). DTPG has a better ADR compared to RTPG. We see that there is an improvement in the ADR for the traditional TPG under column (RTPG) when comparing with methods M1, A1 and A2. The test length (TL) is best with method A1 when compared to A2 and M3 but the test generation time (TIME) is best for method A2.

Table 22. Comparing ADR of best proposed methods with ADR of original test set using ITC'99

		RTPG	DTPG	M3	A1	A2
b04	**ADR**	3.3	2.1	1.9	1.9	1.9
	FC	98.5	98.5	98.5	98.5	98.5
	EP	-	-	-	30	266
	TL	74	121	340	104	340
	TIME	8.1s	18.3s	63.7s	02:45:32h	10.5s
b05	**ADR**	5.0	5.0	4.0	4.0	4.0
	FC	78	78	78	78	78
	EP	-	-	-	42	236
	TL	71	120	307	113	307
	TIME(S)	20.2s	0.3s	184.7s	04:58:33h	21.6s
b07	**ADR**	2.8	2.3	2.1	2.0	2.3
	FC	97.1	99.5	97.1	97.1	97.1
	EP	-	-	-	36	177
	TL	44	54	221	80	221
	TIME	4.5s	0.2s	27.0s	03:30:22h	4.4s
b11	**ADR**	5.1	3.1	2.7	2.7	2.8
	FC	95.4	95.4	95.4	95.4	95.4
	EP	-	-	-	67	224
	TL	77	118	301	144	301
	TIME	12.2s	0.2s	76.1s	05:57:55h	10.3s
b12	**ADR**	2.7	2.2	2.0	2.0	2.0
	FC	99.2	100.0	99.1	99.2	99.2
	EP	-	-	-	56	439
	TL	129	199	568	182	568
	TIME	29.7s	0.04s	202.8s	11:45:22h	36.6s
b14	**ADR**	2.8	2.5	2.2	2.6	2.3
	FC	91.3	97.0	91.0	91.3	91.3
	EP	-	-	-	16	1688
	TL	542	1128	2230	558	2230
	TIME	1523.7s	32991.2s	18031.7s	90:34:57h	3443.9s
b15	**ADR**	3.6	3.3	2.8	3.4	2.8
	FC	91.0	94.1	90.0	91.0	91.0
	EP	-	-	-	13	2012
	TL	462	740	2474	475	2474
	TIME	947.1s	35192.1s	19060.8s	45:31:29h	2610.3s

Table 21 holds the result for the ITC'99 [**6**] family. Columns RTPG and DTPG hold result for normal test set for testing (not for diagnosis). DTPG has a better ADR compared to RTPG. We can see also that generally the three methods, M3, A1 and A2 all have better ADR. For example there is an improvement of about 45% for circuit b11. With the same TL M3 has a slightly better ADR than A2 but A2 has the better test generation time. A1

on the other hand has the worse test generation time but compared to M3 and A2 has a compact TL

5.2. Strength and weakness of the proposed methods

The tables in section 5.1 show the results of the best methods that have been proposed. Comparing M3 with the original test set generated by the random or deterministic TPG in Table 20, Table 21 and Table 22 shows good improvement in the ADR however the TL is higher and also the time to generate the test is also high. In M3 the FC is used as a guide during the generation of the test and also due to the selection criteria of candidate test vectors whereby vectors that detect the minimum number of faults are selected; The consequences is that the FC converges to its maximum value very slowly hence the longer test generation time.

A1 and A2 on the other hand use a different approach to improve the ADR of an already existing test set generated by the random ATPG. In Table 20, Table 21, and Table 22 the results of A1 and A2 show good improvement in the ADR when compared to the random and deterministic test set generated for testing. A1 evaluates the ADR of the entire test set each time a new test vector is introduced, due to this approach it has an advantage of improved ADR with very short TL compared to the other proposed methods. However the calculation of ADR for the entire test set in each step impacts negatively on the speed because the calculation of ADR is computationally intensive.

A2 on the other hand avoids the expensive ADR calculation but instead uses a a simplified cost function to estimate the ADR in each step. The advantage is that the speed is improved but the disadvantage is that the TL is longer.

6. Summary and conclusion.

The traditional approach for generating a diagnostic test set is usually to generate deterministically such a set, however the method is very expensive in terms of time and computational cost because the deterministic generator has to generate a distinguishing test vector for every pair of faults in the test set.

In trying to solve the same problem two approaches were introduced and both approaches are based on some form of randomness which does not require high computational cost, but still at the end, is able to produce a test set with good ADR.

The first approach was introduced in chapter 3 and it produced three methods (M1, M2 and M3) which aim to achieve the goal by incorporating a measure for selecting test vectors with good DR during the generation of the test set. A side experiment showed the impact of selecting such test vectors with maximum number of detectable faults and minimum number of detectable faults and the latter produced a better result. Also the impact of fault collapsing on the ADR was shown, from the experimental result method M3 came out as the best method in the first approach.

The second approach produced two methods (A1 and A2) and each required two stages, first a random ATPG is used to generate a normal test set for testing with the main target of maximum FC and short TL. The first stage is common to both A1 and A2. The second stage involved finding additional test vectors that would ultimately improve the ADR of the original test set. A1 randomly generates a test vector, then introduces it into the current test set and calculates the ADR of the entire set in order to determine if the introduced test vector will improve the ADR. A1 produced good ADR with very good TL, however the calculation of ADR for the entire set is an expensive operation so this method suffered greatly in terms of longer diagnostic test generation time. Method A2 avoided the expensive operation of A1 by using a simplified cost function for estimating the relevance and contribution of a test pattern that would be introduced into the test set. This approach introduced a better performance due to short test generation time, but suffered in terms of longer TL and degraded ADR compared to method A1. All the methods proposed show very good improvement in the ADR and have very good potential for further development.

The goal of this thesis was to provide a tool for randomly generating diagnostic test set for digital circuits with better ADR, this is opposite to the deterministic approach. While it is desirous to achieve the optimal ADR this was not the main goal but instead to approach the problem in a random way and investigate the improvement the random approach introduces. As previously stated in section 2.7 the motivation behind the random approach is that by using such an approach it is possible to bypass the expensive deterministic operation of trying to generate a distinguishing vector between a pair of fault, improve the diagnostic test generation speed and finally improve the ADR.

This thesis yielded two papers the first paper is titled "A Tool for Random Test Generation Targeting High Diagnostic Resolution" and the paper was accepted as a conference paper in the 15th Biennial Baltic Electronics Conference - BEC, Tallinn on the 7th of July, 2016.

The second paper titled "A novel random approach to diagnostic test generation" was submitted on the 17th of August 2016 to the NORCAS 2016 conference and as at the time of writing this thesis no feedback of acceptance has been received yet.

In the future I would like to improve on the methods proposed to achieve a higher ADR. An interesting idea would be to introduce some more determinacy into the proposed algorithms

References

[1] A.Markus, P.Paomets, J.Raik, R.Ubar G.Jervan, "A CAD System for Teaching Digital Test," in *Proc. of the 2nd European Workshop on Microelectronics Education, Kluwer Academic Publishers*, Noordwijkerhout, the Netherlands, May 14-15, 1998, pp. 287-290.

[2] L.T. and Wu, C.W. and Wen, X. Wang, *VLSI Test Principles and Architectures: Design for Testability*.: MORGAN KAUFMANN PUBL Incorporated, 2006.

[3] D. and Paschalis, A. and Zorian, Y. Gizopoulos, *Embedded Processor-Based Self-Test*.: Springer US, 2013.

[4] H.Fujiwara F.Brglez, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *Int. Test Conference*, 1985, pp. 785-794.

[5] D.Bryan, K.Kominski F.Brglez, "Combinational Profiles of Sequential Benchmark Circuits," in *Int. Symp. on Circuits and Systems*, 1989, pp. 1929-1934.

[6] M.S.Reorda, G.Squillero F.Corno, "RT-level ITC'99 Benchmarks and First ATPG Results," *Proc. Of the IEEE Design & Test of Computers*, vol. 17, no. 3, pp. 44-53, 2000.

[7] E. J. Marinissen and Y. Zorian, "Challenges in testing core-based system ICs," *IEEE Communications Magazine*, vol. 37, no. 6, pp. 104-109, June 1999.

[8] Moore G., "Cramming More Components onto Integrated Circuits.," *– Reprint from IEEE proceedings on Electronics*, vol. 38, no. 8, 1965.

[9] A. Prabhu and V. Vorisek and H. Lang and T. Schumann, "Analysis of cell-aware test pattern effectiveness — A case study using a 32-bit automotive microcontroller," *2014 19th IEEE European Test Symposium (ETS)*, pp. 1-2, May 2014.

[10] Navabi, *Digital System Test and Testable Design: Using HDL Models and Architectures*.: Springer US, 2010.

[11] Zhang Y. and Agrawal V. D., "A diagnostic test generation system," in *2010 IEEE International Test Conference.*, Nov 2010, pp. 1-9.

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

[13] Jha N.K. and Gupta S.K., *Testing of Digital Systems*. London: Cambridge University Press, 2003.

[14] P. Camurati, D. Medina, P. Prinetto, and M. Sonza Reorda, "A diagnostic test pattern generation algorithm," in *Test Conference*, Washington, DC, 1990, pp. 52-58.

[15] Shung-Chih Chen and Jer Min Jou, "Diagnostic fault simulation for synchronous sequential circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 3, pp. 299-308, March 1997.

[16] I. Hartanto, W. K. Fuchs, E. M. S. Venkataraman, "Rapid Diagnostic Fault Simulation of Stuck-at Faults in Sequential Circuits using Compact Lists," in *Design Automation*, San Francisco, 1995, pp. 133-138.

[17] C.Timoc et al., "Logical Models of Physical Failures," in *Proceedings of the International Test*, 1983, pp. 546-553.

[18] J.A. Abraham and W.K. Fuchs, "Fault and error models for VLSI," *Proc. of IEEE,* vol. 74, no. 5, pp. 639–654, 1986.

[19] J.P Hayes, "Fault modeling," *IEEE Design and Test of Computers*, vol. 2, no. 2, pp. 88–95, 1985.

[20] J.P. Shen, W. Maly, and F.J. Ferguson, "Inductive fault analysis of MOS integrated circuits," *IEEE Design and Test of Computers*, vol. 2, no. 6, pp. 13–26, 1985.

[21] C. Liu, "Compact Dictionaries for Fault Diagnosis," *IEEE Trans. On Computers*, vol. 53, no. 6, June 2004.

[22] S.Holst and H.J. Wunderlich, "Adaptive debug and diagnosis without fault dictionaries.," in *12th European Test Symposium*, Freiburg, , 2007, pp. 7-12.

[23] S.Venkataraman and S.B.Drummonds, "Poirot: a logic fault diagnosis tool and its application.," in *Proc. IEEE International Test Conference*, 2000, pp. 253-262.

[24] A. Rousset and al et, "A tool for unified logic diagnosis," in *12th European Test Symposium*, Freiburg, 2007, pp. 13-20.

[25] I.Pomeranz and S.M.Reddy, "On correction of multiple design errors," *IEEE Trans. CAD*, vol. 14, no. 2, pp. 255-264, 1995.

[26] B.Boppana, R.Mukherjee, J.Jain, and M.Fujita., "Multiple error diagnosis based on Xlists," *DAC*, pp. 100-110, June 1999.

[27] Shi-Yu Huang, "On improving the accuracy of multiple defect diagnosis," in *VLSI Test Symposium*, Marina Del Rey, CA, 2001, pp. 34-39.

[28] T.Bartenstein and al et, "Diagnosing combinational logic design using the single location at-a-time (SLAT) paradigm," in *Proc IEEE ITC*, 2001, pp. 287-296.

[29] Horng-Bin Wang, Shi-Yu Huang, and Jing-Reng Huang, "Gate-delay fault diagnosis using the inject-and-evaluate paradigm," in *Defect and Fault Tolerance in VLSI Systems*,., 2002, pp. 117-125.

[30] Stroud C. E., *A Designer's Guide to Built-In Self-Test*. Norwell MA: Kluwer Academic, 2002.

[31] Mourad S. and Zorian Y., *Principles of Testing Electronic Systems*. Somerset, NJ: John Wiley & Sons, 2000.

[32] Kostin S., *Self-Diagnosis in Digital Systems (Ph.d Dissertation)*. Tallinn: TUT Press, 2012.

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

[34] Khang A. B. and S. Reda, "Combinationatorial group testing methods for BIST diagnosis problem," in *Proc. of the ASP-DAC.*, 2004, pp. 113-116.

[35] J. Savir and J.P. Roth, "Testing for, and distinguishing between failures," in *Proc. Int. Test Conference.*, 1982, pp. 165-172.

[36] Pomeranz I. and Fuchs W.K., "A diagnostic test generation procedure for combinational circuits based on test elimination," in *Proc. Asia Test Symposium.*, 1998, pp. 486-491.

[37] T. Gruning, U. Mahlstedt, and H. Koopmeiners, "DIATEST: a fast diagnostic test pattern generator for combinational circuits," in *Proc. Int. Conference on Computer-Aided Design.*, 1991, pp. 194-1197.

[38] Pomeranz I. and Reddy S. M., "Diagnostic Test Generation Targeting Equivalence Classes," in *16th Asian Test Symposium (ATS 2007).*, October 2007, pp. 301-306.

[39] (2016, May) http://www.pld.ttu.ee/tt/. [Online]. http://www.pld.ttu.ee/tt/

[40] Bushnell M. L. and Agrawal V. D., *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. New York: Springer Science, 2000.

Appendix 1 – Program Description and Manual

There are three separate tools. A java applet (safdiag.jar) developed by [**32**] (described in Appendix 2), a GUI based tool (DiagBoost.exe) for improving the average diagnostic resolution of a given test set using A1 (Described in Appendix 3) and finally the random command line tool (random.exe). The random command line tool is originally part of the Turbo Tester tool suite developed in [**1**]. This random tool has now been developed further to support random generation of test sets with better diagnostic resolution. It supports 4 new methods for providing such a test set and the methods are represented with the following names M1, M2, M3 and A2. M1, M2 and M3 are used during the generation of the test set and A2 is used after the generation of the test set.

It is assumed that the reader is familiar with the Turbo Tester tool suite, if not please refer to the reference manual [**39**] for more information on the “random” tool. The reference provided here focuses mostly on the contribution this Thesis has introduced into the tool and the original options of the tool that are relevant.

To setup the environment for using the tool on a windows machine follow the three steps below.

1. Copy the application (random.exe) into a folder.

2. Copy the SSBDD model file (*.agm) that you want to generate a test set for into the same folder

3. Open the command console (CMD) and navigate to the location of the folder created in step 1.

Using the tool

command: random

input: SSBDD model file (.agm)

output: test pattern file (.tst)

syntax: random [*options*] <*design*>

design:	Name of the design file without the .agm extension.

options relevant

-failure_limit <*limit*>	The maximum number of packages that can fail before program terminates Default 64.
-pack_size <*size*>	The number of vectors in a package is size multiplied by 32. Default for size is 1.
-criterion <*faults*>	Needed by options [-M1] and [-M2]. Specifies the maximum limit of detected faults below which a test vector can be selected.
-packages <*packages*>	Maximal number of packages to be simulated. Default is 1000.

-select_max <*vectors*>	Maximal number of vectors selected from a package. Default is 32.
-fault_table	Perform fault simulation for the final patterns.

Options for diagnostic patterns

-max_sort	Vectors are sorted with maximum weight (number of faults detected). By default the vectors are sorted with minimum weight.
-M1	Generate diagnostic pattern using M1. Select the patterns that meet the [-criterion] option. E.g. if criterion=7 then only vectors that detect below or equal to 7 faults will be selected.
-M2	Generate a diagnostic test set using M2. Less strict with option [criterion] uses option [-criterion_increment] to break deadlock.
-criterion_increment <*step*>	Only useful with option [-M2]. The number by which the criterion should increase after four consecutive fails. Default value is 1.

-M3	Generate a diagnostic test set using M3. Does not require any limiting criterion option [-criterion].
-A2	Optimize the normal random test set with additional test set to improve diagnostic resolution.
-extra_test_vectors <*value*>	Needed by option [-A2] to indicate how many extra vectors should be added. Has no default value so a valid input must be supplied by the user.

Example 1 – How to generate diagnostic test with M1

Assuming the environment has been setup as described in the beginning of Appendix 1. We can begin to run the command. In Figure 8 the name of the SSBDD model is c432macro.agm but only the name (without the .agm extension) has been used.

```
test>random -failure_limit 64 -pack_size 60 -M1 -criterion 22 -packages 1000 -select_max 1 -fault_table c432macro
```

Figure 8. Running command to generate diagnostic test with option M1.

```
Reading SSBDD-model file c432macro.agm... OK

Allocating test patterns... OK

Generating...

Starting random test pattern generation

Simulation size = 1000

Tested 573 of 616 faults
Fault coverage: 93.019481
211 vectors generated

Allocating test patterns... OK
OK
Time, used by process: 13.891000

Fault analysis... OK
Time, used by process: 0.000000

Writing test patterns file c432macro.tst... OK
```

Figure 9. Program output after running with option M1.

Figure 9 is the output after we run the command. The application then generates an output file c432macro.tst and this file contains the test set with better diagnostic resolution.

Example 2 - How to generate diagnostic test with M2

Assuming the environment has been setup as described in the beginning of Appendix 1. We can begin to run the command. Figure 10 shows how to run the command with the relevant options for generating diagnostic vectors with M2. The name of the SSBDD model is c432macro.agm but only the name (without the .agm extension) has been used.

```
D:\test>random -failure_limit 64 -pack_size 60 -M2 -criterion 7 -packages 1000
 -select_max 1 -fault_table c432macro
```

Figure 10. Running command to generate diagnostic test with option M2

```
Reading SSBDD-model file c432macro.agm... OK

Allocating test patterns... OK

Generating... Generating the first minimum weight pattern

First pattern was successfully generated

Starting random test pattern generation

Simulation size = 1000

Tested 573 of 616 faults
Fault coverage: 93.019481
209 vectors generated

Allocating test patterns... OK
OK
Time, used by process: 23.273000

Fault analysis... OK
Time, used by process: 0.003000
```

Figure 11. Program output after running with option M2.

When the command has finished executing it produces an output similar to Figure 11. The output file c432macro.tst is also generated.

Example 3 – How to generate diagnostic test with M3

Assuming the environment has been setup as described in the beginning of Appendix 1. We can begin to run the command. Figure 12 shows how to run the command with the relevant options for generating diagnostic vectors with M3; if you notice the option [-criterion] is omitted because it is not needed by M3. The name of the SSBDD model is c432macro.agm but only the name (without the .agm extension) has been used.

```
D:\test>random -failure_limit 64 -pack_size 60 -M3 -packages 1000 -select_max 1
-fault_table c432macro
```

Figure 12. Running command to generate diagnostic test with option M3.

Figure 13 below is the output produced when the program terminates. Upon completion the output file c432macro.tst is produced.

```
Reading SSBDD-model file c432macro.agm... OK

Allocating test patterns... OK

Generating...

Starting random test pattern generation

Simulation size = 1000

Tested 573 of 616 faults
Fault coverage: 93.019481
211 vectors generated

Allocating test patterns... OK
OK
Time, used by process: 13.126000

Fault analysis... OK
Time, used by process: 0.002000

Writing test patterns file c432macro.tst... OK
```

Figure 13. Program output after running with option M3.

Example 4 – How to generate diagnostic test with A2.

Assuming the environment has been setup as described in the beginning of Appendix 1. We can run the command to use option A2. In Figure 14 the name of the SSBDD model is c432macro.agm but only the name (without the .agm extension) has been used.

```
D:\test>random -failure_limit 64 -pack_size 60 -criterion 1 -A2 -extra_test_vectors 175
 -packages 1000 -select_max 1 -fault_table c432macro
```

Figure 14. Running command to generate diagnostic test with option A2.

```
Reading SSBDD-model file c432macro.agm... OK

Allocating test patterns... OK

Generating...

Starting random test pattern generation

Simulation size = 1000

Tested 573 of 616 faults
Fault coverage: 93.019481
36 vectors generated

Allocating test patterns... OK
OK
Time, used by process: 2.352000

Fault analysis... OK
Time, used by process: 0.002000

Writing test patterns file c432macro.tst... OK

Improving Average Diagnostic Resolution.
Will add 175 extra test vectors to improve the ADR

Allocating test patterns... OK
OK
Time, used by process: 2.288000

Writing test patterns file DR_c432macro.tst... OK
```

Figure 15. Program output after running with option A2.

The output of the program is shown in Figure 15, looking closely at the output in the figure you will notice that the random tool first generates a normal test set for testing (High fault coverage and short test length). After that A2 comes in to improve the generated test set by adding a number of extra diagnostic test vectors specified by the user, in this case 175 diagnostic vectors have been added. Two output files are generated the normal test file c432macro.tst and the diagnostic test file DR_c432macro.tst.

Appendix 2 - How to compute/extract the average diagnostic resolution from the test file.

To setup the environment for using the safdiag.jar tool on a windows machine follow the four steps below.

1. Must have Java JRE 8 installed on your PC.
2. Copy the application (safdiag.jar) into a folder.
3. Copy the test file (*.tst) that you want to compute average diagnostic resolution for.
4. Open the command console (CMD) and navigate to the location of the folder created in step 1.

---

tool: safdiag.jar

---

input: test file (*.tst)

output: diagnostic resolution report file (*.saf)

---

syntax: Java –jar safdiag.jar <design>

---

design:	Name of the test file but without the .tst extension.
options:	None.

Example – How to use the safdiag.jar tool to compute the average diagnostic resolution of a test file.

Assuming the test environment has been setup as described in the beginning part of Appendix 2 the figure below shows how to run the command. In the figure the test file is c432macro.tst but notice that it has been entered without the extension.

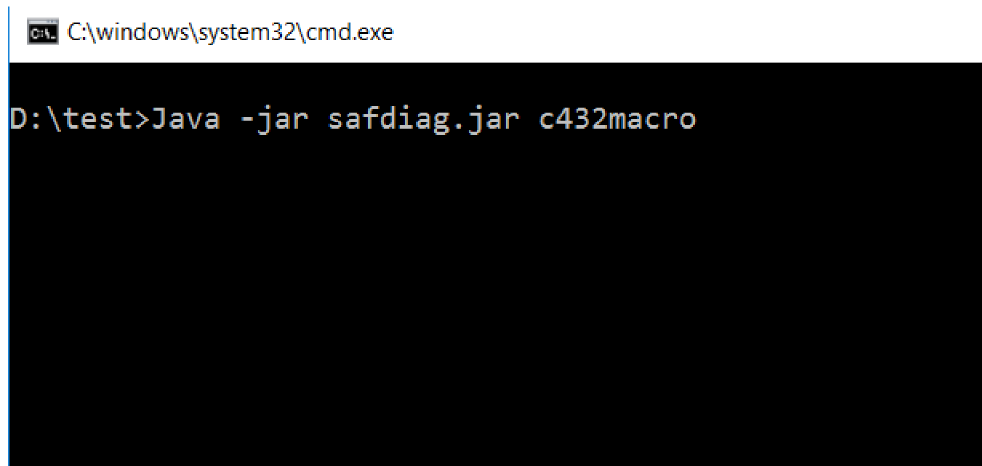

Figure 16 How to compute the ADR of a test file.

After the program executes it generates the output as shown below in Figure 17.

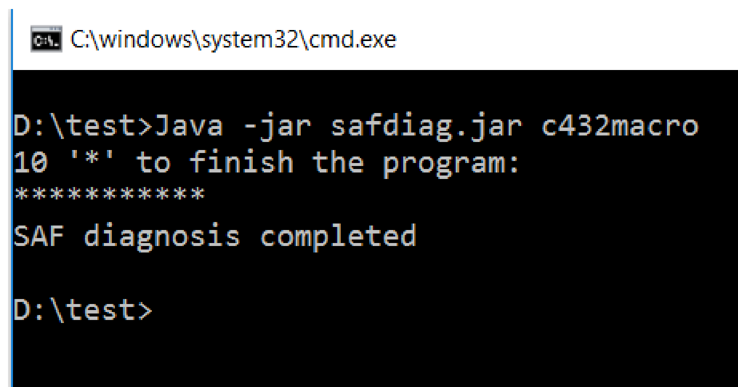

Figure 17. ADR computation complete.

An output file with .saf extension is also generated so in this case the file will be c432macro.saf. The file is a text file that contains the details of the diagnostic resolution.

Appendix 3 – How to use the GUI tool DiagBoost.exe for A1.

tool: DiagBoost	
input: test file (*.tst) and SSBDD file (*.agm)	
output: statistics file (.output), test file (*.tst) and diagnostic report file (*.saf).	
requirements:	Windows 7 and above, .NET framework 4.5 minimum, Java JRE 8.

DiagBoost.exe combines the analyse tool which is also a part of the Turbo Tester tool suite [**1**] and the safdiag.jar tool [**32**] into an easy to use GUI. It then uses both tools together with the algorithm described in section 434.1 to improve the average diagnostic resolution of an already generated test set.

The tool was developed with c# programming language and is only supported on the windows platform at the moment. To use the tool Windows 7 or above, the .NET framework 4.5 and Java JRE 8 must be available on your PC. No installation is required, a zipped folder DiagBoost contains all the necessary items required to use the application.

1. After unzipping the folder just copy it to suitable location on your PC.

2. Double click on DiagBoost.exe to bring up the GUI in Figure 18.

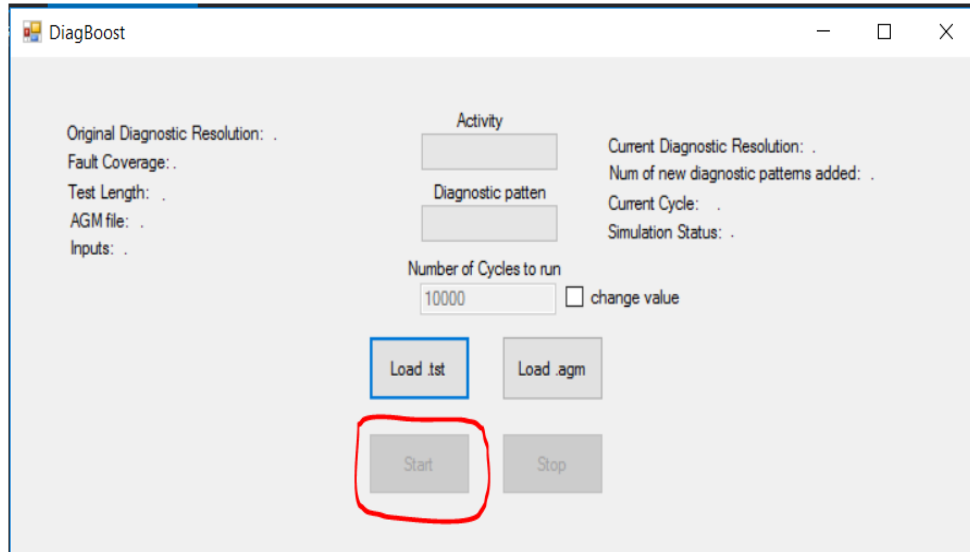

Figure 18. DiagBoost GUI tool.

3. Initially the only buttons that are active are the Load .tst and Load .agm buttons. The start (circled in red) and stop buttons are not active because no files have been provided.

4. Click on the Load .tst button then navigate to the location of the test file (*.tst), select the file.

5. Click on the Load .agm button navigate to the location where the SSBDD file (*.agm) corresponding to the test file is (Note the SSBDD file must match the selected test file in step 4).

6. Once the files have been successfully loaded by DiagBoost the start button becomes active also the initial status is displayed on the left corner of the GUI (circled in red) see Figure 19.

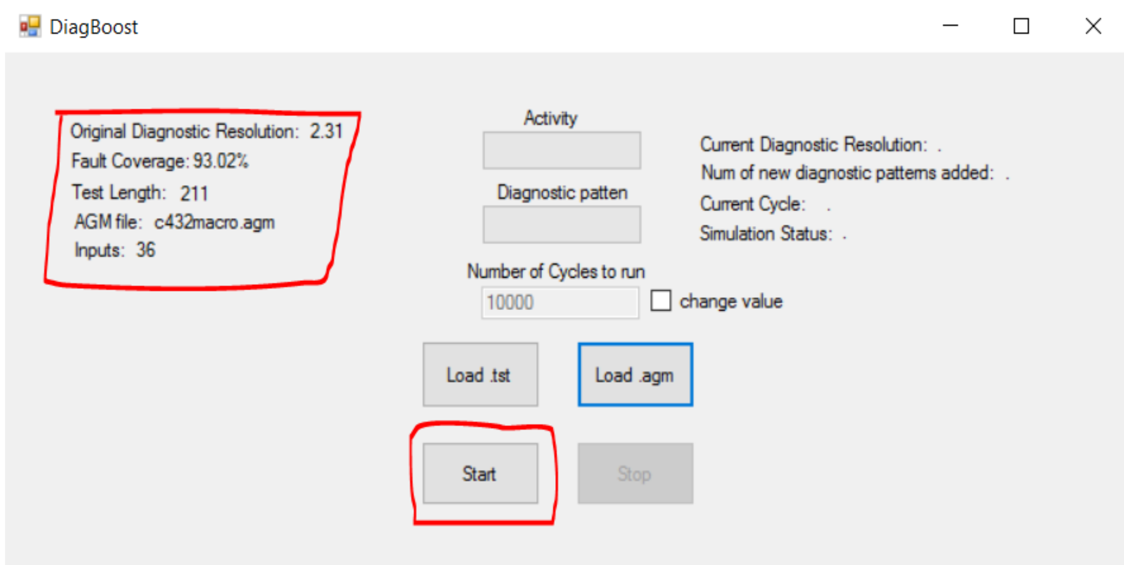

Figure 19. DiagBoost Successfully loaded test file and SSBDD file.

The status of current file loaded into DiagBoost in Figure 19 is displayed. The only values that will change are the ADR and FC.

7. By default the number of iterations the tool will perform is 10,000 but this value can be changed only after the test (*.agm) and SSBDD (*.tst) files have been loaded to the tool and before the tool begins to run.

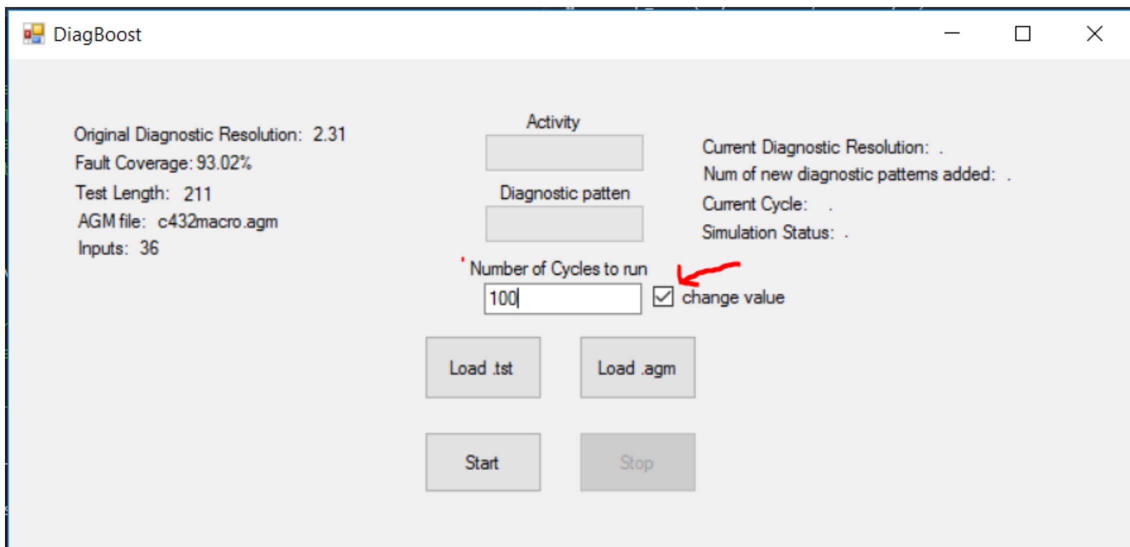

Figure 20. Number of iterations DiagBoost should perform.

8. Click on the Start button to run the DiagBoost tool.

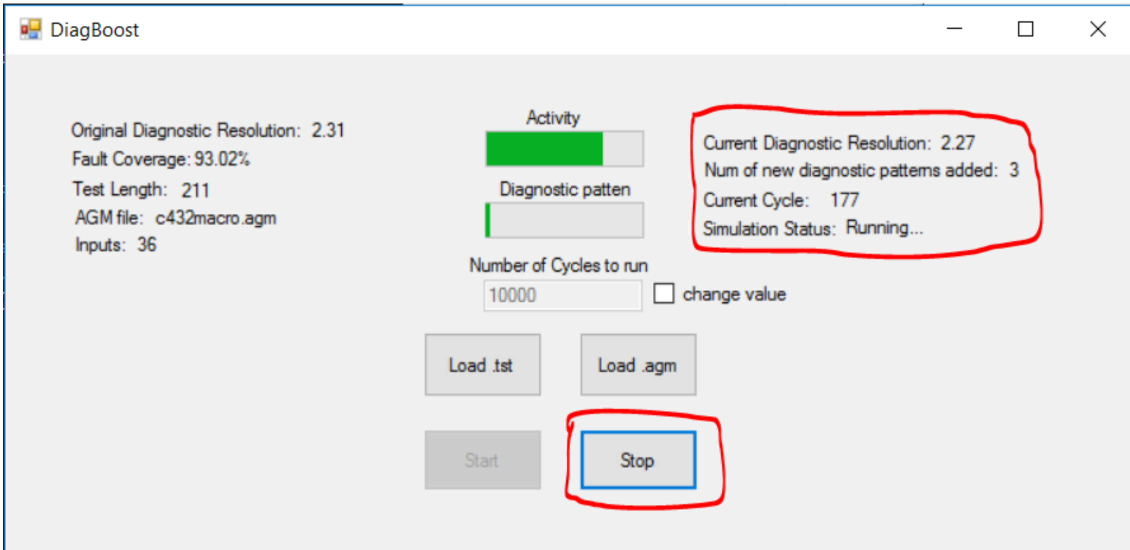

Figure 21. Running the DiagBoost tool.

Figure 21 shows what to expect when the tool begins to run. There are two animated bars (Activity) and (Diagnostic pattern) that gives the user a visual feedback. The top right corner of the tool displays the current status showing the ADR and the number of new test vectors that have been added. Also notice that the Stop button (circled in red) becomes active when the tool begins to run.

9. Click on the Stop button to stop DiagBoost.

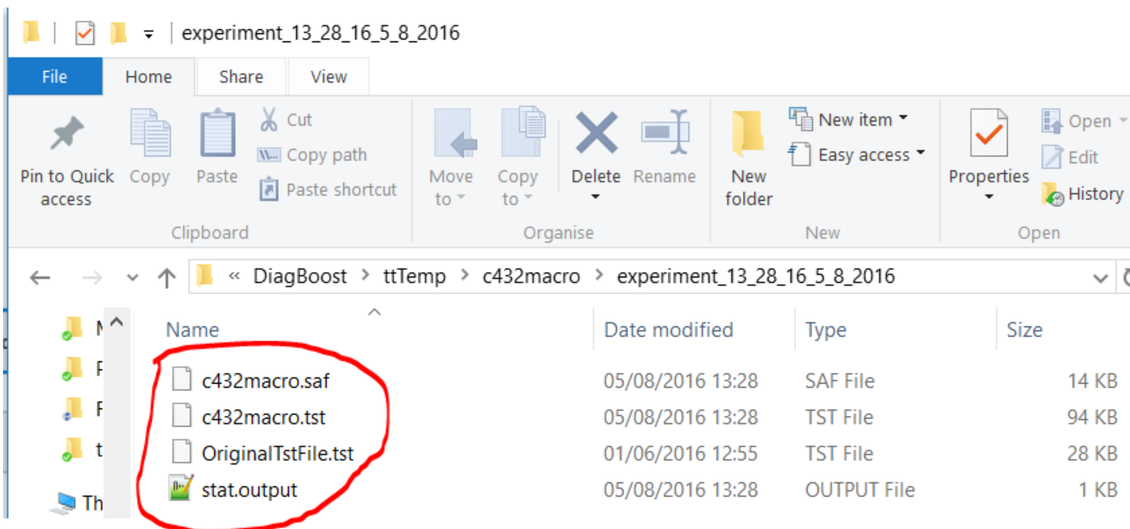

Figure 22. Generated files after DiagBoost stops.

DiagBoost can finish running in two ways, the first is when DiagBoost has run for the number of cycles specified or when the user presses the Stop button. When DiagBoost

finishes it will create a folder in its root directory with the following path ttemp/<Name_of_SSBDD_File>/experiment_HR_MM_SS_DD_MM_YYYY. The content of this folder will be similar to Figure 22.

Appendix 4 – Source Code For method A2

- genDiagPatterns.h -

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "messages.h"
#include "mudel.h"
#include "vector.h"
#include "psimul.h"
#include "randomgen.h"
#include "random.h"

#define LIST_SIZE_CHECKPOINT 128
#define MAX_PATTERN_THRESHOLD __INT_MAX__


typedef struct _list
{
	void (*init)(void **);
	void (*deinit) (void **);
	void (*push)(void **, char *);
	unsigned (*getCount)(void **);
	char *(*getItem)(void **, int i);
	char **items;
	unsigned * weights;
	unsigned count;
}c_list_t;

void CreateList(c_list_t ** list);
void init_list_type(void **);
void deinitListType(void **);
void push(void ** const, char *o);
unsigned getListcount(void **);
char *getitem(void **, int i);
void getWeightOfFaultVectors(void);
void assignFvectWeights(c_list_t ** list);
void sortAscendingByListWeight(c_list_t ** list);
void sortAscendingMainVectAndFtable(unsigned limit);
void sortAscendingResultVectAndFtable(void);
void normalizeResultFtable(int maxWeight);
void improveDiagResolution(unsigned maxNumOfVectsToAdd, char * origfileName);
void initMem(unsigned size);
void free_mem(void);
void multiplyFaultVectors(c_list_t ** list ,int index);
int vectAlreadyInList(c_list_t **list, char **vect);
void myRandVec(void);
void splitFvector(c_list_t **source,int src_index, int result_index);
```

```c
#include "genDiagPatterns.h"
unsigned * newFvWeights = NULL;
unsigned vcount_bkup;
char ** myVects = NULL;
char ** myResultFtable = NULL;
char * myFaults = NULL;
char fileName[] = "fvWeights.txt";

void initMem(unsigned size) {
	myVects = vects;
	vects = NULL;
	myResultFtable = (char**) malloc(size * sizeof(char*));
	if (!myResultFtable) {
		Error("Out of memory: genDiagPattern.c, line 20", -1);
	}

	for (int i = 0; i < size; i++) {
		myResultFtable[i] = (char*) malloc(NodCount);
		if (!myResultFtable)
			Error("Out of memory: genDiagPattern.c, line 25", -1);
		memset(myResultFtable[i], 'X', NodCount);
	}
}

void free_mem(void) {
	int i;
	if (myVects) {
		for (i = 0; i < vcount; i++) {
			_free(myVects[i]);
		}
		_free(myVects);
	}

#ifndef NO_FTABLE
	for (i = 0; i < vcount; i++) {
		_free(myResultFtable[i]);
	}
	_free(myResultFtable);

#endif
	free((unsigned*) newFvWeights);
}

void sortAscendingByListWeight(c_list_t ** list) {
	c_list_t * alist = (c_list_t *) *list;
	unsigned * fvWeight = alist->weights;

	if (fvWeight) {
		for (int i = 0; i < alist->count - 1; i++) {
			for (int j = i + 1; j < alist->count; j++) {
				char *p;
				if (fvWeight[i] < fvWeight[j]) {
					int ajut;

					ajut = fvWeight[i];
```

```
                                fvWeight[i] = fvWeight[j];
                                fvWeight[j] = ajut;
                                p = alist->items[i];
                                alist->items[i] = alist->items[j];
                                alist->items[j] = p;
                        }
                }
            }
        }
}

void sortAscendingMainVectAndFtable(c_list_t ** list) {
        c_list_t * alist = (c_list_t *) *list;
        unsigned * fvWeight = alist->weights;
        if (fvWeight) {
                for (int i = 0; i < alist->count - 1; i++) {
                        for (int j = i + 1; j < alist->count; j++) {
                                char *p;
                                if (fvWeight[i] < fvWeight[j]) {
                                        int ajut;

                                        p = myVects[i];
                                        myVects[i] = myVects[j];
                                        myVects[j] = p;
                                        ajut = fvWeight[i];
                                        fvWeight[i] = fvWeight[j];
                                        fvWeight[j] = ajut;
                                        p = alist->items[i];
                                        alist->items[i] = alist->items[j];
                                        alist->items[j] = p;
                                }
                        }
                }
        }
}

void sortAscendingResultVectAndFtable(void) {
        if (newFvWeights) {
                for (int i = 0; i < vcount - 1; i++) {
                        for (int j = i + 1; j < vcount; j++) {
                                char *p;
                                if (newFvWeights[i] < newFvWeights[j]) {
                                        int ajut;
                                        p = vects[i];
                                        vects[i] = vects[j];
                                        vects[j] = p;

                                        ajut = newFvWeights[i];
                                        newFvWeights[i] = newFvWeights[j];
                                        newFvWeights[j] = ajut;

                                        p = ftable[i];
                                        ftable[i] = ftable[j];
                                        ftable[j] = p;

                                        p = myResultFtable[i];
                                        myResultFtable[i] = myResultFtable[j];
```

```
                                myResultFtable[j] = p;
                            }
                        }
                    }
            }
}

void assignFvectWeights2(c_list_t ** list) {
        c_list_t * alist;
        char logic;
        if (list)
                alist = (c_list_t *) *list;
        else
                Error("Error the **list is NULL: line 261", -1);

        if (alist->weights != NULL) {
                free((unsigned *) alist->weights);
                alist->weights = NULL;
        }

        if (!(alist->weights = (unsigned*) calloc(alist->count,
        sizeof(unsigned)))) {
                Error("Out of memory: vector.c, line 298", -1);
        }

        for (int i = 0; i < alist->count; ++i) {
                for (int j = 0; j < NodCount; ++j) {
                        logic = alist->items[i][j];
                        switch (logic) {
                        case '1':
                        case '0':
                                ++(alist->weights[i]);
                                break;
                        default:
                                break;
                        }
                }
        }
}

void assignFvectWeights(char** fvec, unsigned **fvWeight, unsigned size) {
        char logic;
        if ((*fvWeight) != NULL) {
                free((unsigned *) *fvWeight);
                *fvWeight = NULL;
        }

        if (!(*fvWeight = (unsigned*) calloc(size, sizeof(unsigned)))) {
                Error("Out of memory: vector.c, line 64", -1);
        }
        unsigned * temp = *fvWeight; //get the main pointer
        for (int i = 0; i < size; ++i) {
                for (int j = 0; j < NodCount; ++j) {
                        logic = fvec[i][j];
                        switch (logic) {
                        case '1':
                        case '0':
```

```
                        ++temp[i];
                        break;
                        default:
                                break;
                        }
                }
        }
}

void multiplyFaultVectors(c_list_t ** list, int index) {

        c_list_t * alist;
        if (!list)
                Error("The list is empty: line 361", -1);

        alist = (c_list_t *) *list;

        for (int i = 0; i < vcount; ++i) {
                for (int j = 0; j < NodCount; ++j) {
                        /*
                         The table below is what I am trying to implement
                         X and X = X
                         X and 1 = X
                         X and 0 = X
                         1 and 0 = X
                         1 and 1 = 1
                         0 and 0 = 0
                         */
                        if (alist->items[index][j] == 'X' || ftable[i][j] == 'X')
                                myResultFtable[i][j] = 'X';
                        else {

                                switch (alist->items[index][j]) {

                                case '1':
                                        if (ftable[i][j] == '0')
                                                myResultFtable[i][j] = 'X';
                                        else
                                                myResultFtable[i][j] = '1';
                                        break;

                                case '0':
                                        if (ftable[i][j] == '1')
                                                myResultFtable[i][j] = 'X';
                                        else
                                                myResultFtable[i][j] = '0';
                                        break;

                                default:
                                        break;
                                }
                        }
                }
        }
}
```

```
void normalizeResultFtable(int maxWeight) {
	if (newFvWeights) {
		int maxW = maxWeight / 2;
		for (int i = 0; i < vcount; ++i) {
			/* Intent
			 *    maxWeight
			 *    |
			 *    |
			 *    middle| newFvWeights[i]=maxWeight - newFvWeights[i]
			 *    |
			 *    |             do nothing
			 *    0
			 * */
			//if the weight is equal or greater than half of
			maxWeight then enter.
			if (!(newFvWeights[i] < maxW)) {
				newFvWeights[i] = maxWeight - newFvWeights[i];
			}
		}
	} else {
		Error("[W] Null Pointer (*newFvWeights): genDiagPatterns.c, line
314",-1);
	}
}

int vectAlreadyInList(c_list_t **list, char **vect) {
	c_list_t * tempList = *list;

	if (!tempList)
		return 1;
	int limit = tempList->getCount((void **) &tempList);
	if (limit == 0)
		return 0;
	int retValue = 1;

	for (int i = 0; i < limit; ++i) {
		retValue = 1;
		for (int j = 0; j < InpCount; ++j) {

	if ((tempList->getItem((void **) &tempList, i))[j] != (*vect)[j]) {
				retValue = 0;
				break;
			}
		}

		if (retValue == 1) {
			return 1;
		}
	}
	return retValue;
}

void splitFvector(c_list_t **source, int src_index, int result_index) {
	c_list_t * srFvec;
	char * p;
	if (source == NULL)
		Error("splitFvector: source pointer is NULL, line 430", -1);
```

```
		srFvec = (c_list_t *) *source;

	if (!(p = (char*) malloc(NodCount))) {
		Error("Out of memory", -1);
	}
	for (int i = 0; i < NodCount; ++i) {
		//make a deep copy of the result, this will be the first part of
		the divided group;
		p[i] = myResultFtable[result_index][i];
	}

	srFvec->push((void**) source, p); //make a shallow copy;
	p = NULL; //A shallow copy was made so it cannot be freed. Instead it
	is assigned NULL so that it can be reassigned again.

	for (int j = 0; j < NodCount; ++j) {
		if (srFvec->items[src_index][j] ==
myResultFtable[result_index][j]) {
			srFvec->items[src_index][j] = 'X';
		}
		//These part takes care of the fault that will be detected in
		the 2nd part of the group.
		else if ((srFvec->items[src_index][j] != 'X')
				&& myResultFtable[result_index][j] == 'X') {

			;
		} else {
			srFvec->items[src_index][j] = 'X';
		}
	}
}

void improveDiagResolution(unsigned maxNumOfVectsToAdd, char * origfileName)
{

	free((char *) faults); //free the faults vector
	c_list_t * tempVecList;
	c_list_t * tempFtable1;
	unsigned nSize = 200; //buffer size is set to 200.
	unsigned numOfVectOptimize = vcount;
	unsigned numOfVectsAdded = 0;
	char *p = NULL;
	initMem(nSize);
	CreateList(&tempVecList);
	CreateList(&tempFtable1);

	//No need to initialize tempFtable1 since its been assigned to valid
	memory of ftable which has not been deallocated.
	tempFtable1->items = ftable; //copy the initial ftable.
	ftable = NULL; // reset pointer ftable.
	tempFtable1->count = vcount;

	tempFtable1->items =
	(char **) realloc(tempFtable1->items,
	sizeof(char *) * (tempFtable1->count + LIST_SIZE_CHECKPOINT));

	if (!tempFtable1->items) {
```

```
		Error("Out of memory",-1);
	}

	tempVecList->init((void **) &tempVecList);

	vcount_bkup = vcount;
	vcount = nSize;
	alloc_vec(); //reallocate the normal vectors.
	StartTimer();
	for (int i = 0; i < 1; ++i) {
		rand_vec(); //generate a random vector.
	}

	//initialize the weights pointer of the list and assign valid weights.
	assignFvectWeights2(&tempFtable1);
	sortAscendingMainVectAndFtable(&tempFtable1);

	for (int i = 0; i < tempFtable1->count; ++i) {
		rand_vec(); //generate a random vector.
		fsimul(); //simulate with the
		multiplyFaultVectors(&tempFtable1, 0);
		assignFvectWeights(myResultFtable, &newFvWeights, nSize);
		normalizeResultFtable(tempFtable1->weights[0]);
		sortAscendingResultVectAndFtable();
		int k = 0;
		while (k < nSize) {
			if (vectAlreadyInList(&tempVecList, &vects[k++]) == 0) {
				if (!(p = (char*) malloc(VarCount))) {
					Error("Out of memory", -1);
				}
				//take the first vect since it will be the max.
				memcpy(p, vects[k - 1], VarCount);
				tempVecList->push((void **) &tempVecList, p);
				p = NULL;
				++numOfVectsAdded;
				splitFvector(&tempFtable1, 0, k - 1);
				free(tempFtable1->weights);
				tempFtable1->weights = NULL;
				assignFvectWeights2(&tempFtable1);
				//initiliaze the weights pointer of the list and
				assign valid weights.
				sortAscendingByListWeight(&tempFtable1);
				break;
			}
		}

		if (numOfVectsAdded >=
			maxNumOfVectsToAdd|| numOfVectsAdded >=
			MAX_PATTERN_THRESHOLD)
			break;
	}
	tempFtable1->deinit((void **) &tempFtable1);

	int newSize = vcount_bkup +
	tempVecList->getCount((void **) &tempVecList);
	myVects = (char **) realloc(myVects, newSize * sizeof(char *));
	if (!myVects)
```

```
		Error("Out of memory: genDiagPattern.c, line 405",-1);

	for (int i = 0; i < tempVecList->count; ++i) {
		//make a shallow copy of the items in the list.
		myVects[vcount_bkup + i] =
		tempVecList->getItem((void **) &tempVecList, i);
	}
	//since. we have made a shallow copy we set to NULL to prevent the
	memory from being freed.
	tempVecList->items = NULL;

	free_vec(); // free the main vectors so that they can be reassigned

	vects = myVects;
	//set to null so that when we call free_mem it will not free myVects
	since we have made a shallow copy of it.
	myVects = NULL;

	free_mem();
	vcount = newSize; //change the size.

	if (!(ftable = (char**) malloc(vcount * sizeof(char*)))) {
		Error("Out of memory", -1);
	}
	for (int i = 0; i < vcount; i++) {
		if (!(ftable[i] = (char*) malloc(NodCount)))
			Error("Out of memory", -1);
		memset(ftable[i], 'X', NodCount);
	}
	if (!(faults = (char*) malloc(NodCount * sizeof(char)))) {
		Error("Out of memory", -1);
	}
	memset(faults, 'X', NodCount);

	alloc_psimul();
	init_psimul();
	psimul();
	EndProcessing();
	EndTimer();
	serial_vec();
	char newFileName[strlen(origfileName) + 1 + 7];
	newFileName[0] = 'D';
	newFileName[1] = 'R';
	newFileName[2] = '_';
	newFileName[3] = 0;
	strcat(newFileName, origfileName);
	strcat(newFileName, ".tst");
	write_vec(newFileName, DEFAULT_OUTPUT);
	free_vec();
	tempVecList->deinit((void **) &tempVecList);
}

void CreateList(c_list_t ** list) {
	*list = (c_list_t *) malloc(sizeof(c_list_t));
	if (!(*list)) {
		Error("Out of memory: genDiagpattens, line 451",-1);
	}
```

```
        c_list_t * _list = *list;
        _list->init = init_list_type;
        _list->weights = NULL;
        _list->items = NULL;
        _list->push = push;
        _list->getItem = getitem;
        _list->getCount = getListcount;
        _list->deinit = deinitListType;
}

void init_list_type(void ** list) {
//get a hold of the list pointer.
c_list_t * alist = (c_list_t *) *list;
        if (alist) {
                alist->count = 0;
                alist->items =
                (char **) malloc(sizeof(char *) * LIST_SIZE_CHECKPOINT);
        } else {
                Error("[W] Null Ptr (*myList): genDiagPatterns.c, line 489",-1);
        }
}

void deinitListType(void ** list) {
        //get a hold of the list pointer. Note this is just a copy by value
        (or a duplication of the original pointer) so this local pointer and
        the original
        //pointer that was passed as a pointer to pointer now points to the
        same memory. This does not mean that alist can change the content of
        tempFvList1 or tempFvList2.
        //These two pointer variables declared in "improveDiagResolution" are
        passed as pointer to pointer to this function. tempFtable2-
        >deinit((void **)&tempFtable2);
        c_list_t * alist = (c_list_t *) *list;
        if (alist) {
                //if the memory pointer is empty then we do not free because a
                shallow copy has been made.
                if (alist->items)
                {
                        for (int i = 0; i < alist->count; ++i) {
                                free((char *) alist->items[i]);
                        }
                        free((char *) alist->items);

                }

                //if the memory pointer is empty then we do not free because a
                shallow copy has been made.
                if (alist->weights)
                {
                        free((char *) alist->weights);
                }
                free((c_list_t *)alist);
                //This will only affect the local copy pointer and not the
                original pointer that was passed to this function
                alist = NULL;
                //set the original pointer that was passed to this function to
                point to NULL since the memory has been freed.
```

```
			*list = NULL;

	} else {
			Error("[W] Null Ptr (*myList): genDiagPatterns.c, line 464",-1);
	}

}

unsigned getListcount(void ** list) {
	//get a hold of the list pointer.
	c_list_t * alist = (c_list_t *) *list;
	if (alist)
			return alist->count;
	else
			Error("[W] Null Ptr (*myList): genDiagPatterns.c, line 502",-1);
	return 0; //should never get here
}

char *getitem(void **list, int i) {
	//get a hold of the list pointer.
	c_list_t * alist = (c_list_t *) *list;
	if (alist)
			return alist->items[i];
	else
			Error("[W] Null Ptr (*myList): genDiagPatterns.c, line 510",-1);
	return NULL; //should never get here
}

void push(void ** const list, char *o) {
	//get a hold of the list pointer.
	c_list_t * alist = (c_list_t *) *list;
	if (alist) {
			if (alist->count >= LIST_SIZE_CHECKPOINT) {
					if (!(alist->count % LIST_SIZE_CHECKPOINT))
							alist->items =
							(char **) realloc(alist->items,
							sizeof(char *) * (alist->count +
							LIST_SIZE_CHECKPOINT));
			}
			alist->items[alist->count++] = o;
			int c = alist->count;
	} else {
			Error("[W] Null Ptr (*myList): genDiagPatterns.c, line 519",-1);
	}
}
```