國立中山大學電機工程研究所

碩士論文

指導教授：李錫智　博士

以第一原理診斷組合式數位電路

Diagnosis of Combinatorial Digital Circuits from First Principles

研究生　韓定中　撰

中華民國八十五年六月完成
中華民國八十六年十一月一校

Diagnosis of Combinatorial Digital Circuits from First Principles

Benjamin Han

Advisor: Shie-Jue Lee

June 25, 1996

November 12, 1997, First Revision

Department of Electrical Engineering

National Sun Yat-Sen University

Kaohsiung, Taiwan 80424

Tel: +886-7-5252000 ext. 4141 FAX: +886-7-5254199

Email: ben@water.ee.nsysu.edu.tw

致謝詞

首先我要感謝的是我的指導教授 — 李錫智博士。在這一段研究過程中，他不斷地給予我指引。他對學術上追求的熱情，以及他的生活及個性，都爲我樹立了一個印象深刻而難以磨滅的範例。

其次，我要感謝楊欣泰學長（雖然我從未把他當作學長看）。他在這一個研究工作一開始時與我一起合作。雖然其後他沒有再繼續這一個領域，但是他對我的研究，甚至爲人處事方面，提出了無數寶貴和深入的意見。我想，如果沒有這樣一位好友，這兩年研究生生活將會貧脊許多。

另外我要感謝實驗室的其他夥伴們。他們曾與我一同分享了這兩年來的酸甜苦辣。從他們每個人的身上，我都學到了一些寶貴的東西。

當然，我要感謝我的父母。謝謝他們長久以來對我的愛和支持，兩年來他們一直爲我操心，現在終於告一段落了。我希望藉著這一本微不足道的研究成果，作爲對他們無盡的愛的一份小小回饋。

最後，我要謝謝我的女友，玉珍。因著她的支持、愛和耐心，這份研究才得以順利完成。

Acknowledgments

First I would like to thank my advisor, Dr. Shie-Jue Lee, for his constant guidance in my humble research. His fervent pursuit in research, together with his life style and his personality, has established an impressive example for me.

My most gratitude goes to Hsin-Tai Yang. He has been involved in the early stages of this work. Although he discontinued his research in this work later, he has provided numeral useful and pointed suggestions – not only for my research, but also for my philosophy of life. Were I without such a good friend, it would not be possible for me to have such a fruitful graduate life.

Also I would like to acknowledge the other lab fellows, who have shared whatever happened in these two years with me. It is so fortunate for me that I have discovered something to learn from each of them. Guys, thank you.

I am indebted, of course, to my parents. I am grateful to their love and support, and sorry to make them worry about me in these two years. I wish what I have done here may serve as a nominal return to their endless love.

My final thanks go to my love, Vivian Hsu. Her support, tolerance and love have backed me up in these days, and in turn make this work possible.

學年度 八十四
校院代表 0009
系所代碼 542201
學位類別 M
論文名稱(中) 以第一原理診斷組合式數位電路
論文名稱(英) Diagnosis of Combinatorial Digital Circuits from First Principles
語文別 英文
學號 8331618
研究生姓(中) 韓
研究生名(中) 定中
研究生姓名(英) Han, Benjamin
指導教授(中) 李錫智
指導教授(英) Lee, Shie-Jue
關鍵字(中) 診斷；第一原理；命題邏輯；基因演算法
關鍵字(英) diagnosis; first principles; propositional calculus; genetic algorithm
提要(中)

診斷問題發生在當一個系統表現出與期望不符的行爲時。假設該系統設計無誤，一個診斷即是在所有已知對該系統的觀察下，能夠合理解釋系統行爲差異的一組假設。對一個錯誤系統診斷有其實用性和經濟性，因爲通常換掉整個系統比起換掉不正常的系統組件來得昂貴。

在本論文中，我們使用模型基礎的診斷理論，更明確地說，我們使用了第一原理的診斷理論。我們首先補充一些證明以加強並澄清 Hou 提出的量測理論，並且指出 Hou 提出的最小衝突集 (minimal conflict set) 的尋找方法並不完整 (incomplete)。接著我們提出一個使用標記集合之衝突集樹 (CS-tree with mark set) 的完整、有效且經過證明的最小衝突集導出方法。此外，我們發展了一個使用基因演算法 (genetic algorithm) 的量測選取策略 MSSGA，並且透過一系列的實驗證明它的有效性。最後，我們提出並實作了一個組合式數位電路診斷系統 (DSDC)。該系統接受一個 DSDCL 語言格式的電路描述檔，而後自動尋找出電路中的錯誤組件。此外，藉由使用 MSSGA 選取量測點，DSDC 具有線上自我改進效能的特色。

提要(英)

The diagnostic problem arises when the observed behavior of a system deviates from the expectation. Assuming that the design of the system is correct in the sense that it is consistent with its specification, a diagnosis is a hypothesis of faults within the system under the observations obtained so far which can explain the discrepancies between the expected and the actual behavior of the system. The reason to perform diagnosis is that by replacing only the faulty components of a malfunctioning system, one can repair the system instead of replacing it, and the former solution is usually more economical and practical.

In this thesis we focus on model-based diagnosis, in particular, the theory of diagnosis from first principles. We augment and clarify Hou's theory of measurement with supplementary proofs, and revise the incomplete method for minimal conflict set derivation with a complete, efficient and proved derivation approach using a CS-tree (conflict set tree) with mark set. A measurement selection strategy using the genetic algorithm (MSSGA) is proposed to select the best next measurement in a diagnostic process, and the results of a series of experiments show that this approach is promising.

Finally a complete diagnostic system for combinatorial digital circuits (DSDC) is proposed and implemented. DSDC receives a circuit specification in DSDC Language (DSDCL), and automatically locates the culprits within the circuit. In addition, it is capable of improving its performance online by incorporating MSSGA as its measurement selection strategy.

頁數	94

《 授 權 書 》

本人 韓定中 所撰（著） 八十四 學年度第 二 學期
國立中山大學　電機工程研究所　碩士 班

碩士　學位論文之提要　同意　開放供學術利用。

論文名稱： 以第一原理診斷組合式數位電路

Diagnosis of Combinatorial Digital Circuits from First Principles

立書人：姓名：

地址：臺北縣新店市長春路 18 巷 1 號

身分證統一編號：C120666801

聯絡電話：(02) 217-2113

中華民國 八十五 年 六 月 廿五 日

中文摘要

診斷問題發生在當一個系統表現出與期望不符的行爲時。假設該系統設計無誤，一個診斷即是在所有已知對該系統的觀察下，能夠合理解釋系統行爲差異的一組假設。對一個錯誤系統診斷有其實用性和經濟性，因爲通常換掉整個系統比起換掉不正常的系統組件來得昂貴。

在本論文中，我們使用模型基礎的診斷理論，更明確地說，我們使用了第一原理的診斷理論 [39]。我們首先補充一些證明以加強並澄清 Hou 提出的量測理論，並且指出 Hou 提出的最小衝突集 (minimal conflict set) 的尋找方法並不完整 (incomplete)。接著我們提出一個使用標記集合之衝突集樹 (CS-tree with mark set) 的完整、有效且經過證明的最小衝突集導出方法。此外，我們發展了一個使用基因演算法 (genetic algorithm) 的量測選取策略 MSSGA，並且透過一系列的實驗證明它的有效性。最後，我們提出並實作了一個組合式數位電路診斷系統 (DSDC)。該系統接受一個 DSDCL 語言格式的電路描述檔，而後自動尋找出電路中的錯誤組件。此外，藉由使用 MSSGA 選取量測點，DSDC 具有線上自我改進效能的特色。

關鍵字: 診斷，第一原理，命題邏輯，基因演算法

Abstract

The diagnostic problem arises when the observed behavior of a system deviates from the expectation. Assuming that the design of the system is correct in the sense that it is consistent with its specification, a diagnosis is a hypothesis of faults within the system under the observations obtained so far which can explain the discrepancies between the expected and the actual behavior of the system. The reason to perform diagnosis is that by replacing only the faulty components of a malfunctioning system, one can repair the system instead of replacing it, and the former solution is usually more economical and practical.

In this thesis we focus on model-based diagnosis, in particular, the theory of diagnosis from first principles formalized in [39]. We augment and clarify Hou's theory of measurement with supplementary proofs, and revise the incomplete method for minimal conflict set derivation with a complete, efficient and proved derivation approach using a CS-tree (conflict set tree) with mark set. A *m*easurement *s*election *s*trategy using the *g*enetic *a*lgorithm (MSSGA) is proposed to select the best next measurement in a diagnostic process, and the results of a series of experiments show that this approach is promising. Finally a complete *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC) is proposed and implemented. DSDC receives a circuit specification in DSDC Language (DSDCL), and automatically locates the culprits within the circuit. In addition, it is capable of improving its performance *online* by incorporating MSSGA as its measurement selection strategy.

Keywords: diagnosis, first principles, propositional calculus, genetic algorithm

Table of Contents

CHAPTER 1 INTRODUCTION 1

1.1 PROBLEM DESCRIPTION 1

1.2 THE THEORY OF DIAGNOSIS FROM FIRST PRINCIPLES 2

1.3 THE OUTLINE OF THE PROPOSED APPROACH 3

1.4 THE ORGANIZATION OF THIS THESIS 4

CHAPTER 2 THE THEORY OF MEASUREMENTS IN DIAGNOSIS FROM FIRST PRINCIPLES 7

2.1 INTRODUCTION 7

2.2 MEASUREMENTS, CONFLICT SETS AND DIAGNOSES 9

2.2.1 Review of Hou's Consequences 9

2.2.2 The Relationships among Measurements, Conflict Sets and Diagnoses 9

2.3 ENHANCED CONFLICT RECOGNITION 13

2.3.1 Review of Hou's Consequences 13

2.3.2 Comments on Enhanced Conflict Recognition 14

2.4 CONFLICT SET DERIVATION 19

2.4.1 Hou's Method for Deriving Minimal Conflict Sets 19

2.4.2 A Complete Method for Deriving Minimal Conflict Sets 21

2.4.3 Further Improvements to Our Method 25

2.4.3.1 Derivation with Extra Information 25

2.4.3.2 Derivation with an Incremental Prover 26

2.4.4 Top-down vs. Bottom-up 27

2.5 DISCUSSIONS 28

CHAPTER 3 MEASUREMENT SELECTION IN DIAGNOSIS FROM FIRST PRINCIPLES VIA THE GENETIC ALGORITHM 29

3.1 INTRODUCTION 29

3.2 GENETIC ALGORITHMS 30

3.3 (MSSGA) A MEASUREMENT SELECTION STRATEGY VIA THE GENETIC ALGORITHM 31

3.3.1 MRL Evaluation 33

3.3.2 The Genetic Operators of MSSGA 34

3.3.3 Adapting the Fitness Values of the Genetic Operators 35

3.4 EXPERIMENT RESULTS 35

3.4.1 Fixed Fault Tests 37

3.4.1.1 Testing Circuit c01_8 38

3.4.1.2 Testing Circuit c02_18 39

3.4.1.3 Testing Circuit c03_19 41
3.4.1.4 Testing Circuit c04_16 42
3.4.1.5 Testing Circuit c05_16 43
3.4.2 Noisy Tests 45
3.4.2.1 Testing Circuit c01_8 45
3.4.2.2 Testing Circuit c02_18 47
3.4.2.3 Testing Circuit c03_19 48
3.4.2.4 Testing Circuit c04_16 49
3.4.2.5 Testing Circuit c05_16 51
3.5 DISCUSSIONS 52

CHAPTER 4 A DIAGNOSTIC SYSTEM FOR DIGITAL CIRCUITS 55

4.1 INTRODUCTION 55
4.2 OVERVIEW OF DSDC 56
4.3 DSDC PREPROCESSOR AND DSDC LANGUAGE (DSDCL) 57
4.4 INFERENCE ENGINE 61
4.4.1 The Method of Davis and Putnam 61
4.4.2 A PC Prover Implemented by Trie Data Structure (PCPT) 62
4.4.3 Incremental Proving Using PCPT 65
4.5 DIAGNOSIS GENERATOR 65
4.6 MEASUREMENT REQUEST GENERATOR 67
4.7 EXAMPLE RUNS 68
4.7.1 The First Example: An Arbitrary Circuit with 8 Gates 68
4.7.2 The Second Example: A BCD to Decimal Decoder (7442) with 18 Gates 69
4.8 DISCUSSIONS 71

CHAPTER 5 CONCLUSIONS 73

APPENDIX A AN IMPLEMENTATION OF DSDC 75

A.1 DEVELOPMENT ENVIRONMENT 75
A.2 USING DSDC 75
A.3 SOURCE CODE 77

APPENDIX B THE COMPLETE LISTINGS OF FIVE TESTING CIRCUITS IN DSDCL 79

B.1 CIRCUIT C01_8: AN ARBITRARY CIRCUIT WITH 8 GATES 79
B.2 CIRCUIT C02_18: A BCD TO DECIMAL DECODER (7442) WITH 18 GATES 81
B.3 GATES CIRCUIT C03_19: A LOOKAHEAD CARRY GENERATOR (74182) WITH 19 83
B.4 CIRCUIT C04_16: A 3 TO 8 DEMULTIPLEXER (74138) WITH 16 GATES ... 85
B.5 GATES CIRCUIT C05_16: A DUAL 4 TO 1 DATA SELECTORS (74153) WITH 16 87

BIBLIOGRAPHY..... 91

List of Figures

FIG. 1.3.1. THE OUTLINE OF OUR APPROACH.....5

FIG. 2.2.1. THE RELATIONSHIPS BETWEEN THE MCSS BEFORE ADDING Π AND THE MCSS AFTER ADDING Π DEPICTED BY: (A). THEOREM 2.2.4; (B). THEOREM 2.2.5; (C). THEOREM 2.2.4 & THEOREM 2.2.5.....10

FIG. 2.2.2. THE RELATIONSHIPS BETWEEN THE DIAGNOSES BEFORE ADDING Π AND THE DIAGNOSES AFTER ADDING Π DEPICTED BY: (A). THEOREM 2.1.7 (1); (B). THEOREM 2.1.7 (2) & (3); (C). THEOREM 2.1.7 AS A WHOLE. 11

FIG. 2.4.1. DERIVING THE MCSS FROM C, USING HOU'S METHOD.....20

FIG. 2.4.2. DERIVING THE MCSS FROM C (ALTERNATIVE ORDER), USING HOU'S METHOD20

FIG. 2.4.3. AN EXAMPLE CIRCUIT ON WHICH HOU'S MCS DERIVATION METHOD FAILS...21

FIG. 2.4.4. DERIVING ALL MCSS FOR THE CIRCUIT IN FIG. 2.4.3 USING HOU'S METHOD .21

FIG. 2.4.5. THREE POSSIBLE SUBTREES IN A CS-TREE WITH MARK SET ROOTED IN $\{c_1, c_2\}$22

FIG. 2.4.6. DERIVING THE MCSS FROM C, USING OUR METHOD.....24

FIG. 2.4.7. A FAULTY CIRCUIT. THE ACTUAL VALUE OF THE INPUT/OUTPUT OF EACH GATE IS SHOWN IN THE FIGURE. ALSO, THE SHADED GATES ARE THE ONES CAUSING THE SYSTEM TO MISBEHAVE. 25

FIG. 2.4.8. DERIVING THE MCSS FROM $\{A_1, A_2, A_3, A_4, A_5, A_6, O_2\}$.....26

FIG. 2.4.9. DERIVING THE MCSS FROM C, USING AN INVERSE CS-TREE.....28

FIG. 3.4.1. CIRCUIT C01_8: PI VALUE VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS38

FIG. 3.4.2. CIRCUIT C01_8: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 38

FIG. 3.4.3. CIRCUIT C01_8: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 39

FIG. 3.4.4. CIRCUIT C01_8: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 39

FIG. 3.4.5. CIRCUIT C02_18: PI VALUE VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS39

FIG. 3.4.6. CIRCUIT C02_18: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 40

FIG. 3.4.7. CIRCUIT C02_18: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 40

FIG. 3.4.8. CIRCUIT C02_18: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 40

FIG. 3.4.9. CIRCUIT C03_19: PI VALUE VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS41

FIG. 3.4.10. CIRCUIT C03_19: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 41

FIG. 3.4.11. CIRCUIT C03_19: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS

IN FIXED FAULT TESTS 41
FIG. 3.4.12. CIRCUIT C03_19: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 42
FIG. 3.4.13. CIRCUIT C04_16: PI VALUE VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS42
FIG. 3.4.14. CIRCUIT C04_16: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 42
FIG. 3.4.15. CIRCUIT C04_16: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 43
FIG. 3.4.16. CIRCUIT C04_16: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 43
FIG. 3.4.17. CIRCUIT C05_16: PI VALUE VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS43
FIG. 3.4.18. CIRCUIT C05_16: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 44
FIG. 3.4.19. CIRCUIT C05_16: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 44
FIG. 3.4.20. CIRCUIT C05_16: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN FIXED FAULT TESTS 44
FIG. 3.4.21. CIRCUIT C01_8: PI VALUE VS. NUMBER OF EVALUATIONS IN NOISY TESTS 46
FIG. 3.4.22. CIRCUIT C01_8: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN NOISY TESTS 46
FIG. 3.4.23. CIRCUIT C01_8: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN NOISY TESTS 46
FIG. 3.4.24. CIRCUIT C01_8: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN NOISY TESTS47
FIG. 3.4.25. CIRCUIT C02_18: PI VALUE VS. NUMBER OF EVALUATIONS IN NOISY TESTS ..47
FIG. 3.4.26. CIRCUIT C02_18: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN NOISY TESTS 47
FIG. 3.4.27. CIRCUIT C02_18: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN NOISY TESTS 48
FIG. 3.4.28. CIRCUIT C02_18: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN NOISY TESTS48
FIG. 3.4.29. CIRCUIT C03_19: PI VALUE VS. NUMBER OF EVALUATIONS IN NOISY TESTS ..48
FIG. 3.4.30. CIRCUIT C03_19: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN NOISY TESTS 49
FIG. 3.4.31. CIRCUIT C03_19: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN NOISY TESTS 49
FIG. 3.4.32. CIRCUIT C03_19: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN NOISY TESTS49
FIG. 3.4.33. CIRCUIT C04_16: PI VALUE VS. NUMBER OF EVALUATIONS IN NOISY TESTS ..50
FIG. 3.4.34. CIRCUIT C04_16: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN NOISY TESTS 50
FIG. 3.4.35. CIRCUIT C04_16: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS

IN NOISY TESTS 50
FIG. 3.4.36. CIRCUIT C04_16: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN NOISY TESTS 51
FIG. 3.4.37. CIRCUIT C05_16: PI VALUE VS. NUMBER OF EVALUATIONS IN NOISY TESTS ..51
FIG. 3.4.38. CIRCUIT C05_16: NUMBER OF MEASUREMENTS VS. NUMBER OF EVALUATIONS IN NOISY TESTS 51
FIG. 3.4.39. CIRCUIT C05_16: SCALING COEFFICIENTS A & B VS. NUMBER OF EVALUATIONS IN NOISY TESTS 52
FIG. 3.4.40. CIRCUIT C05_16: OPERATOR FITNESS VS. NUMBER OF EVALUATIONS IN NOISY TESTS 52
FIG. 4.2.1. THE BLOCK DIAGRAM OF DSDC 56
FIG. 4.3.1. A CASCADED TWO-BIT FULL ADDER.....60
FIG. 4.4.1. THE TRIE STRUCTURE OF THE CLAUSE $(A \vee B) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg A \vee C)$ 63
FIG. 4.5.1. DERIVING ALL MHSS FOR THE CIRCUIT SHOWN IN FIG. 2.4.3 USING REITER'S PRUNED HS-TREE 66
FIG. B.1. CIRCUIT C01_8: AN ARBITRARY CIRCUIT WITH 8 GATES 79

List of Tables

TABLE 3.4.1. THE VALUES OF THE PARAMETERS FOR MSSGA USED ON EACH TESTING CIRCUIT IN FIXED FAULT TESTS.....36

TABLE 3.4.2. THE VALUES OF THE PARAMETERS FOR MSSGA USED ON EACH TESTING CIRCUIT IN NOISY TESTS.....36

TABLE 3.4.3. THE PERCENTAGE OF THE SEARCH SPACE EXPLORED ON EACH TESTING CIRCUIT.....37

TABLE 3.4.4. FIXED FAULT TESTS: IMPROVEMENT IN THE AVERAGE *PI* VALUES.....37

TABLE 3.4.5. FIXED FAULT TESTS: IMPROVEMENT IN THE NUMBER OF MEASUREMENTS N_M REQUIRED BY DSDC TO SETTLE DOWN TO ONE DIAGNOSIS.....37

TABLE 3.4.6. NOISY TESTS: IMPROVEMENT IN THE AVERAGE *PI* VALUES.....45

TABLE 3.4.7. NOISY TESTS: IMPROVEMENT IN THE NUMBER OF MEASUREMENTS N_M REQUIRED BY DSDC TO SETTLE DOWN TO ONE DIAGNOSIS.....45

TABLE 4.3.1. ALL TERMINALS OF DSDCL.....57

TABLE 4.3.2. THE MEANING AND THE PRECEDENCE OF EACH OPERATOR USED IN DSDCL..58

TABLE 4.3.3. THE BASIC STRUCTURE OF A CIRCUIT SPECIFICATION IN DSDCL.....58

TABLE 4.3.4. THE FORMULATION OF *DEFINE-BLOCK* IN DSDCL.....58

TABLE 4.3.5. THE FORMULATION OF *DECLARE-BLOCK* IN DSDCL.....59

TABLE 4.3.6. THE FORMULATIONS OF *CONNECT-BLOCK*, *OBSERVE-BLOCK* AND *ASSIGN-BLOCK* IN DSDCL.....59

TABLE 4.3.7. THE CIRCUIT SPECIFICATION FOR THE CASCADED TWO-BIT FULL ADDER AT LARGER GRANULARITY.....60

TABLE 4.3.8. THE CIRCUIT SPECIFICATION FOR THE CASCADED TWO-BIT FULL ADDER AT FINER GRANULARITY.....61

TABLE B.1. THE INITIAL FITNESS VALUES OF THE FOUR GENETIC OPERATORS USED IN NOISY TESTS.....79

Chapter 1

Introduction

In this chapter, we first briefly describe the diagnostic problem and make a concise survey of the approaches developed by other researchers. Among the different schemes we develop our diagnostic system for combinatorial digital circuits based on Reiter's theory of diagnosis from first principles [39], which is introduced in this chapter. The outline and the contributions of our approach are then presented. At the end of this chapter we show the organization of the rest of this thesis.

1.1 Problem Description

A diagnostic problem arises when the observed behavior of a system deviates from the expectation. Assuming that the design of the system is correct in the sense that it is consistent with its specification, a diagnosis is a hypothesis of faults within the system under the observations obtained so far which can explain the discrepancies between the expected and the actual behavior of the system. The reason to perform diagnosis is that by replacing only the faulty components of a malfunctioning system, one can repair the system instead of replacing it, and the former solution is usually more economical and practical.

Diagnosis is a complex cognitive process involving different knowledge [7, 30, 38, 41, 42, 43, 44, 45, 46]. Given a malfunctioning system, the task of diagnosis is to pinpoint the minimal sets of faulty components using general domain knowledge, design and structural information of the system, and past experience of diagnosing such system. Diagnosis has been one of the major topics in artificial intelligence (AI) research since 1960s. Many researchers have proposed various kinds of schemes to tackle the problem, and the approaches can be roughly divided into two categories based on the knowledge used. In the first kind of schemes, the diagnostic system tries to codify the statistical intuitions and the heuristic rules of diagnosis for future operation. When a conflict occurs between the supposed and the actual observation, the diagnostic system tries to locate the culprits by searching its knowledge base and trying to apply all pertinent heuristic rules. Such systems try to achieve the same competence level of human diagnosticians by using direct associations between symptoms and failures, hence they are called 'shallow-knowledge,' 'symptom-based' or 'association-based' diagnosis. Notable examples of such systems include MYCIN [4, 50], CRIB [26], LES [32] and REACTOR [37]. These systems contain a rule base of the specific domain knowledge in the form of *IF <symptom> THEN <failures or actions> with <confidence>*. The confidence is a quantitative indication of the intensity of the association between the antecedent and the consequence of the rule. A diagnostic process is then reduced to cycles of rule selections and rule executions.

Although shallow-knowledge methods provide a natural architecture for solving diagnostic problems, they have serious drawbacks. First of all, the process of knowledge acquisition via human experts is costly and impractical for the systems with short life cycles, such as digital circuits. Second, since they rely on the empirical data about the fault patterns of the system being diagnosed, it is difficult for them to recognize new disorders. Moreover, there is no guarantee that such systems are *sound* and *complete*.

To solve the difficulties of the shallow-knowledge approaches described above, there is the second kind of schemes for diagnosis which requires 'deep knowledge' of the system being diagnosed, i.e., not only they know the symptoms of the system, but they know *how* the system is supposed to work as well. To this end, a model for the system being diagnosed has to be built before performing diagnosis. The first approach is based on the fault model of the system (*fault-model-based* diagnosis). A fault model enumerates *all* possible faults within the system [36], such as stuck-at-zero in digital circuits. With the fault model, a diagnostic system can generate fault-detection tests for all possible faults using techniques such as path sensitization [1, 31, 33, 40, 20] and Boolean difference [49]. The one-to-one onto mapping between the faults and the tests is then found by minimizing the obtained fault table [31]. Therefore the task of diagnosis is reduced to a simple table lookup after applying pertinent tests.

The fundamental shortcoming of fault-model-based diagnosis is that without knowing *all* possible faults beforehand, we cannot use such approach. Essentially, this approach is more a scheme for verification of the system production than for diagnosis.

The other kind of approach, which also requires deep knowledge of the system being diagnosed, is *model-based* diagnosis. The difference is that model-based diagnosis requires a model which specifies the normal operations of the system, instead of a fault model which specifies all possible abnormal operations of the system required by fault-model-based diagnosis. A model-based diagnostic system *deduces* all possible fault hypotheses from the given model and the observations obtained thus far using techniques such as constraint suspension [11, 12], constraint propagation [15, 16, 24, 25] and minimal hitting set derivation [39]. The advantage of this approach is that theoretically it is guaranteed to be *sound* and *complete*. Some theories for model-based diagnosis have been developed, among them Reiter has formalized the theory of diagnosis from first principles [39]. Notable model-based diagnostic systems include HT [11, 12], DART [19], GDE [15] and XDE [24, 25].

The next section presents a brief introduction to Reiter's theory of diagnosis from first principles [39]. We also summarize the differences between Reiter's theory and de Kleer's approach described in [15].

1.2 The Theory of Diagnosis from First Principles

Although many researchers have proposed various kinds of schemes to solve the diagnostic problem, only a few theories have been formally developed. Among them, Reiter has developed a formal analysis of model-based diagnosis, which he called 'diagnosis from first principles [39],' based on the work of de Kleer [14] and Genesereth [19]. Reiter first defined a system as a pair (SD, COMP), where SD is a set of first-order sentences[1] describing the functional model of the system, and COMP is a finite set of constants denoting the system components. In SD, Reiter used McCarthy's [34] AB predicate such that $\neg$AB(A) denotes that component A is *not abnormal*. The description of an AND gate A, for example, would be written as follows

$$\neg \mathrm{AB}(A) \supset \mathrm{out}(A) = \mathrm{and}(\mathrm{in1}(A), \mathrm{in2}(A)).$$

A conflict set is then defined as a set of components such that at least one component within the set is abnormal. Together with a set of input and output vectors, i.e., the observation, of the system, the hitting sets, which are sets intersecting each conflict set, can be derived using a breadth-first search technique (HS-tree derivation). Moreover, the hitting set derived is shown to be a diagnosis of the system, in the sense that if all components outside the set are normal, the discrepancies between the expected and the actual observation can be explained.

[1] In fact, Reiter's theory is also applicable to other logics, such as *temporal logic* [35].

Since all occurrences of AB appear as ¬AB in antecedents in SD, Reiter's approach of diagnosis obeys the *ignorance of abnormal behavior* (IAB) condition, and it can be shown that IAB condition leads to the *minimal diagnosis hypothesis* [17], i.e., if Δ is a diagnosis, then for all Δ' such that $\Delta \subseteq \Delta' \subseteq$ COMP, Δ' is also a diagnosis. Therefore a complete characterization of all the diagnoses of a system consists of only the set of minimal diagnoses. The task of diagnosis is then reduced to the derivation of all *minimal* hitting sets for the collection of conflict sets of the system being diagnosed. A more detailed introduction on Reiter's approach is presented in Section 1 of the next chapter.

To fully discriminate among all possible diagnoses, however, we need to make measurements from the system being diagnosed. The primary effects of measurements in diagnosis are discussed in Reiter's paper [39], and a more detailed formal account has been given by Hou [29], which we will introduce and discuss in the next chapter. However, none of them addressed the problem of how to select the best next measurement.

Another important approach of diagnosis has been developed independently by de Kleer and Williams [15]. Essentially, their techniques for conflict recognition and candidate generation are equivalent to Reiter's theory under pertinent mappings. However, de Kleer's work lacks a formalization of their diagnostic theory. The second significant difference between these two approaches is that the workings of Reiter's algorithm require a sound, complete and decidable theorem prover, while de Kleer's approach uses an assumption-based truth maintenance system (ATMS) [17] for the underlying inferences. An ATMS precomputes the dependencies of all inferences and is able to retract the contradictory assumptions according to the recorded dependencies. However, such recordings introduce additional overhead, and usually an ATMS limits the expressive power of the logic used in exchange for better efficiency.

An important contribution of de Kleer's approach is the incorporation of probabilistic information for selecting the best next measurement. In short, he used a one-step lookahead method to determine which measurement will bring the most information based on the error rates of the system components from past experiences. However, this method will be rendered impractical as the number of possible measurements increases.

1.3 The Outline of the Proposed Approach

In the previous sections we briefly discuss the current solutions to the diagnostic problem, together with their limitations and difficulties. To address these difficulties, we propose a complete approach for diagnosis from first principles based on the work of Reiter [39] and Hou [29]. Furthermore, we choose the domain of digital circuits as the test-bed of our proposed diagnostic system. Our contributions include the following points.

1. We discuss and clarify Hou's measurement theory by deriving supplementary proofs and consequences.

2. We address the incompleteness problem of the minimal conflict sets (MCS) derivation method proposed by Hou, and give a complete, efficient and proved MCS derivation method using a CS-tree (conflict set tree) with mark sets.

3. An efficiency drawback of Reiter's approach is caused by the use of a prover for the underlying inferences. By contrast with Reiter's scheme, de Kleer used an ATMS as an underlying inference engine. However, an ATMS needs to precompute all inferences

beforehand for node labeling, and usually limits the expressive power of the logic used in exchange for better efficiency. In this thesis we use an incremental prover which exploits the idea of simple *chronological backtracking* [47] to improve the efficiency and at the same time preserve the completeness. To be more specific, we incorporate an incremental propositional-calculus prover (PC prover) using trie data structure based on the work of Zhang and Stickel [53] into our diagnostic architecture.

4. Although de Kleer used a one-step lookahead method for determining the best next measurement, his approach becomes costly and impractical as the number of possible measurements increases. In this thesis we propose a completely different scheme for automatic measurement selection: a *m*easurement *s*election *s*trategy using the *g*enetic *a*lgorithm (MSSGA).

5. We propose and implement a complete *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC). DSDC receives a circuit specification in DSDC Language (DSDCL), evaluates different possible measurement request lists (MRL) via the genetic algorithm and prompts for measurement result from the user input. The interactive diagnostic process is going on until only one possible diagnosis is left. Upon this point the user may replace the faulty components found within the system, and use a fault-detection method to verify the soundness of the system.

The proposed approach for diagnosing a combinatorial digital circuit is summarized in Fig. 1.3.1.

1.4 The Organization of this Thesis

The rest of this thesis is organized as follows. In Chapter 2 we first review the formal definitions and results from Reiter's work [39], then we introduce Hou's theory of measurements [29], discuss and clarify the theory with supplementary proofs and additional consequences. To be more specific, the complete picture of the relationships among measurements, conflict sets and diagnoses is presented, the incompleteness of Hou's MCS derivation method is addressed, and a complete, efficient and proved MCS derivation method using a CS-tree with mark sets is given.

In Chapter 3 we propose a *m*easurement *s*election *s*trategy using the *g*enetic *a*lgorithm (MSSGA). Four genetic operators are introduced, and five testing circuits are diagnosed under fixed fault patterns (fixed fault tests) and random fault patterns (noisy tests). The result is also presented in the chapter. In Chapter 4 we describe a complete *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC).

We conclude this thesis in Chapter 5 with discussions of our contributions and the possible extensions to this work. Appendix A gives a more detailed description of our implementation of DSDC in C++ language, and Appendix B contains complete listings of five testing circuits in DSDCL used in the experiments. Finally a bibliography is included at the end.

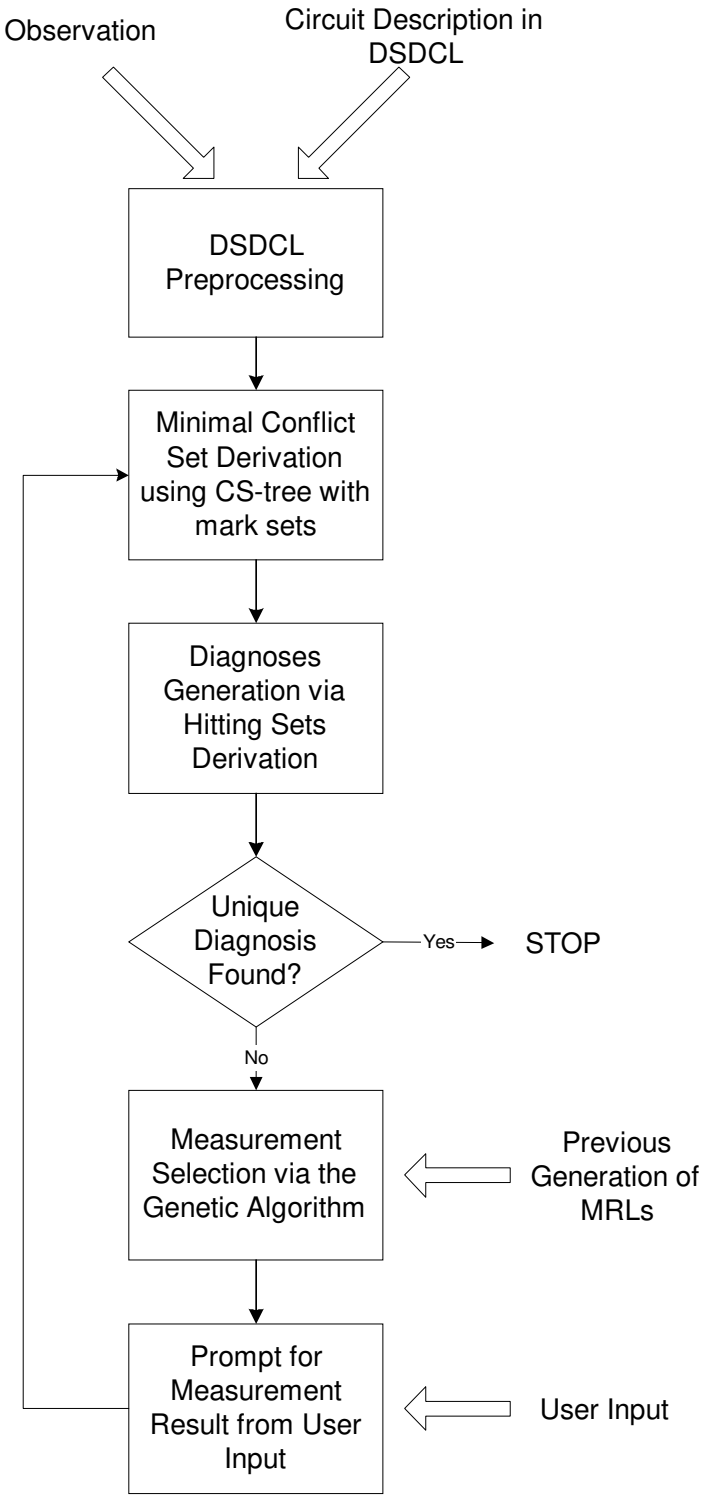

Fig. 1.3.1. The Outline of Our Approach

Chapter 2

The Theory of Measurements in Diagnosis from First Principles

When finding diagnoses from first principles, one needs to refine possible diagnoses by making measurements from the system being diagnosed. Based on the work of Reiter [39], Hou has developed and formalized an efficient incremental method for computing all diagnoses upon measurement [29]. However, we find that some points in Hou's paper need clarifications. In this chapter we describe the complete picture of the relationships among measurements, conflict sets and diagnoses, discuss the equivalence relation defined for enhanced conflict recognition, and revise the incomplete approach for conflict set derivation proposed by Hou.

2.1 Introduction

Under the assumption that the description of a system is consistent, if the inputs and the outputs, i.e., the observation, of the system conflict with the way the system is meant to behave, the diagnostic problem is to pinpoint the possible diagnoses, i.e., the possible sets of faulty components that cause the misbehavior. Many researchers have proposed various kinds of approaches to tackle the problem; among them, Reiter [39] has built and formalized the major theorems of diagnosis from first principles upon the work of de Kleer [14] and Genesereth [19]. In the paper, Reiter gave the following definitions:

Definition 2.1.1 [39, Def. 2.1] A *system* is a pair (SD, COMP) where
(1) SD, the *system description*, is a set of first-order sentences describing

(a) the functionality of a component within the system; e.g., let A be an AND gate with two inputs, the sentence describing it is:

$$\neg AB(A) \supset out(A) = and(\, in1(A), in2(A)\,)$$

where the predicate AB stands for *abnormal*, which is *true iff*[2] gate A is malfunctioning, and 'and' is a function returning *true iff* both inputs of gate A are *true*.

(b) the connections between the components of the system; e.g., $out(A) = in1(B)$ means that the output of component A is connected with the first input of component B.

(2) COMP is a finite set of constants denoting the collection of components of the system.

Definition 2.1.2 [39, Prop. 3.4] Δ is a *diagnosis* for (SD, COMP, OBS) *iff* Δ is a minimal set such that $SD \cup OBS \cup \{\neg AB(c) \mid c \in COMP\text{-}\Delta\}$ is consistent, where OBS stands for the observation of the system (SD, COMP).

---

[2] *If and only if.*

Definition 2.1.3 [39, Def. 4.1] $C \subseteq$ COMP is a *conflict set* (CS) for (SD, COMP, OBS) *iff* SD $\cup$ OBS $\cup$ $\{\neg AB(c) \mid c \in C\}$ is inconsistent. A *minimal* conflict set (MCS) is a CS such that none of its subsets is a CS.[3]

Definition 2.1.4 [39, Def. 4.3] H is a *hitting set* (HS) for a collection of sets S *iff* $H \subseteq \bigcup_{C \in S} C$ such that $H \cap C \neq \emptyset$ for each $C \in S$. A *minimal* hitting set (MHS) is an HS such that none of its subsets is an HS.

To locate all possible diagnoses, Reiter showed that all we have to do is find all MHSs for the collection of MCSs, as stated in the following theorem.

Theorem 2.1.5 [39, Corollary 4.5] $\Delta \subseteq$ COMP *is a diagnosis for* (SD, COMP, OBS) *iff* Δ *is an MHS for the collection of MCSs for* (SD, COMP, OBS).

However, to refine further the set of diagnoses, one needs to take measurements from the system. Let Π be a measurement. Each diagnosis then predicts either Π or $\neg\Pi$.

Definition 2.1.6 [39, Prop. 5.3] A diagnosis Δ for (SD, COMP, OBS) predicts Π *iff*

$$SD \cup OBS \cup \{\neg AB(c) \mid c \in COMP\text{-}\Delta\} \vdash \Pi.$$

It turns out that each diagnosis is either preserved or rejected according to the prediction, as stated in the following theorem.

Theorem 2.1.7 [39, Theorem 5.7] *Suppose that every diagnosis for* (SD, COMP, OBS) *either predicts* Π *or* $\neg\Pi$*, then*

(1) *Every diagnosis for* (SD, COMP, OBS) *which predicts* Π *is a diagnosis for* (SD, COMP, OBS $\cup$ $\{\Pi\}$).
(2) *No Diagnosis for* (SD, COMP, OBS) *which predicts* $\neg\Pi$ *is a diagnosis for* (SD, COMP, OBS $\cup$ $\{\Pi\}$).
(3) *Any diagnosis for* (SD, COMP, OBS $\cup$ $\{\Pi\}$) *which is not a diagnosis for* (SD, COMP, OBS) *is a strict superset of some diagnosis for* (SD, COMP, OBS) *which predicts* $\neg\Pi$*; i.e., any new diagnosis resulting from the new measurement* Π *will be a strict superset of some old diagnosis which predicts* $\neg\Pi$.

Although Theorem 2.1.7 sheds some light on the practical way for computing all diagnoses after adding Π, it is Hou who has developed and formalized an efficient incremental approach for computing them [29]. However, we find that some points in Hou's paper need clarifications. In the rest of this chapter we will supplement and revise some theorems and procedures described in [29]. In Section 2.2 we describe the complete picture of the relationships among measurements, CSs and diagnoses, and relate Reiter's theorems with Hou's consequences. In Section 2.3 we give comments on the equivalence relation defined by Hou for enhanced conflict recognition. In Section 2.4, we point out the incompleteness of Hou's approach for deriving all MCSs from a CS, and then propose an improved method for the derivation. Finally, we give concluding discussions in Section 2.5.

[3] For brevity, through out the rest of this thesis we will use the initials CS for *conflict set*, MCS for *minimal conflict set*, HS for *hitting set* and MHS for *minimal hitting set*.

2.2 Measurements, Conflict Sets and Diagnoses

In Section 2.2.1 we will make a concise review on the relationships between measurements and CSs described in [29]. Then in Section 2.2.2 we show the complete picture of the relationships among measurements, CSs and diagnoses by supplementing the theorems in [29].

2.2.1 Review of Hou's Consequences

To refine further the set of possible diagnoses by a measurement Π, Hou defined a CS for (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from the measurement* Π as follows.

Definition 2.2.1 [29, Def. 3.2] A *CS* for (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from a measurement* Π is a set $\{c_1, c_2, \ldots, c_k\}\subseteq$COMP such that SD $\cup$ OBS $\cup$ $\{\neg AB(c_i) \mid i=1, ..., k\}$ is *consistent*, and SD $\cup$ OBS $\cup$ $\{\Pi\}$ $\cup$ $\{\neg AB(c_i) \mid i=1, ..., k\}$ is *inconsistent.*

By combining Def. 2.2.1 with Def. 2.1.3 and Def. 2.1.6, Hou presented the following important relationship between the diagnoses for (SD, COMP, OBS) predicting $\neg\Pi$ and the CSs for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π:

Corollary 2.2.2 [29, Prop. 3.3] *If* $\Delta\subseteq$COMP *is a diagnosis for* (SD, COMP, OBS) *which predicts* $\neg\Pi$, *then* COMP-Δ *is a CS for* (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from* Π.

On the other hand, every MCS for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π is a subset of COMP-Δ, where Δ is a diagnosis for (SD, COMP, OBS) predicting $\neg\Pi$.

Theorem 2.2.3 [29, Theorem 3.6] *Every MCS for* (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from* Π *is one of the MCSs derived from* COMP-Δ_i, *which is a CS for* (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from* Π, *with* Δ_i *being a diagnosis for* (SD, COMP, OBS) *predicting* $\neg\Pi$.

Corollary 2.2.2 and Theorem 2.2.3 together show that by deriving all MCSs from COMP-Δ_i, where Δ_i is a diagnosis for (SD, COMP, OBS) predicting $\neg\Pi$, one can get all, nothing more than all, MCSs for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π. Moreover, the set of MCSs for (SD, COMP, OBS$\cup\{\Pi\}$) can be divided into two collections, as stated below.

Theorem 2.2.4 [29, Theorem 3.7] *Any MCS for* (SD, COMP, OBS$\cup\{\Pi\}$) *is either an MCS for* (SD, COMP, OBS) *or an MCS for* (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from* Π.

This theorem gives an interesting glimpse about the relationships between the MCSs for (SD, COMP, OBS) and the MCSs for (SD, COMP, OBS$\cup\{\Pi\}$). However, in the rest part of [29], no further explanation regarding the complete description of such relationships is given[4]. Therefore we will start from this point, exploring the complete picture of the relationships among measurements, CSs and diagnoses in Section 2.2.2.

2.2.2 The Relationships among Measurements, Conflict Sets and Diagnoses

First we show that the MCSs for (SD, COMP, OBS) can also be divided into two collections.

Theorem 2.2.5 *Any MCS for* (SD, COMP, OBS) *is either an MCS for* (SD, COMP, OBS$\cup\{\Pi\}$) *or a strict superset of some MCS for* (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from* Π.

[4]In particular, Theorem 3.10 in [29] seems to be a *tautology*.

Proof. Let $P = \{c_1, ..., c_k\}$ be an MCS for (SD, COMP, OBS) and $P_i = P - \{c_i\}$. From Def. 1.3 we know that

$$\text{SD} \cup \text{OBS} \cup \{\neg\text{AB}(c) \mid c \in P\} \text{ is inconsistent, and}$$
$$\text{SD} \cup \text{OBS} \cup \{\neg\text{AB}(c) \mid c \in P_i\} \text{ is consistent.}$$

(1) If $\text{SD} \cup \text{OBS} \cup \{\Pi\} \cup \{\neg\text{AB}(c) \mid c \in P_i\}$ is inconsistent, then by Def. 2.1, P_i is a CS for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π. Any MCS derived from P_i then is a strict subset of P.

(2) If $\text{SD} \cup \text{OBS} \cup \{\Pi\} \cup \{\neg\text{AB}(c) \mid c \in P_i\}$ is consistent, since $\text{SD} \cup \text{OBS} \cup \{\Pi\} \cup \{\neg\text{AB}(c) \mid c \in P\}$ is inconsistent, then by Def. 1.3 P is an MCS for (SD, COMP, OBS$\cup\{\Pi\}$).

□

Given both Theorem 2.2.4 and Theorem 2.2.5, one may wonder what relationships exist between the two collections of MCSs for (SD, COMP, OBS) and the two collections of MCSs for (SD, COMP, OBS$\cup\{\Pi\}$). The following figure shows the relationships depicted by Theorem 2.2.4 and Theorem 2.2.5:

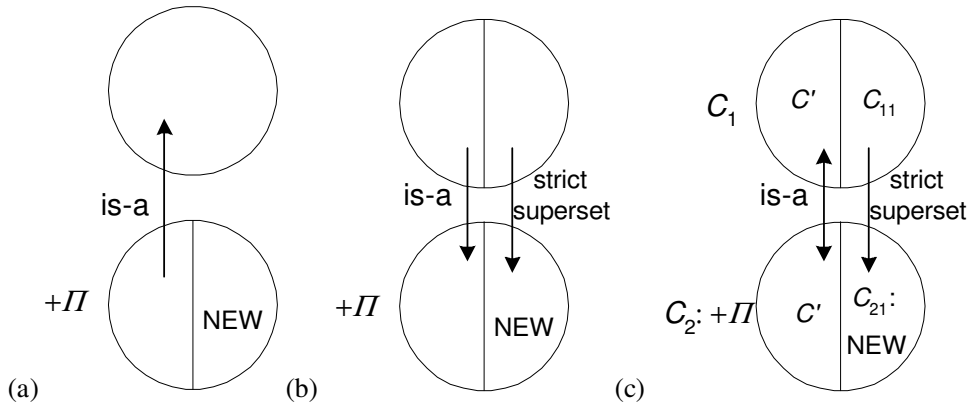

Fig. 2.2.1. The relationships between the MCSs before adding Π and the MCSs after adding Π depicted by: (a). Theorem 2.2.4; (b). Theorem 2.2.5; (c). Theorem 2.2.4 & Theorem 2.2.5.

In the figure, a circle stands for a set of MCSs. The lower circle is the set of MCSs after adding Π. The section denoted by 'NEW' is the collection of MCSs for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π. Fig. 2.2.1 (c) in effect states the following corollary.

Corollary 2.2.6 *Let C_1 be the collection of MCSs for* (SD, COMP, OBS) *and C_2 be the collection of MCSs for* (SD, COMP, OBS$\cup\{\Pi\}$). *Then C_1 can be partitioned into C' and C_{11}, and C_2 can be partitioned into C' and C_{21}, where C_{21} is the collection of MCSs for* (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from Π and $\forall c \in C_{11}$, c is a strict superset of some c' in C_{21}.*

It is easy to see from Fig. 2.2.1 (c) that a diagnosis for (SD, COMP, OBS), if it intersects all MCSs for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π, must be an HS for the collection of MCSs for (SD, COMP, OBS$\cup\{\Pi\}$). Further, Hou showed that such a diagnosis for (SD, COMP, OBS) is also a diagnosis for (SD, COMP, OBS$\cup\{\Pi\}$) [29, Corollary 3.12]. On the other hand, in Theorem 2.1.7 (1), Reiter has shown that if a diagnosis for (SD, COMP, OBS) predicts Π, then it is also a diagnosis for (SD, COMP, OBS$\cup\{\Pi\}$). To relate these two viewpoints, we have the following theorem:

Theorem 2.2.7 *Let Δ be a diagnosis for* (SD, COMP, OBS). *Then Δ predicts Π iff it*

intersects each MCS for (SD, COMP, OBS∪{Π}) *resulting from* Π.

Proof. (⇒) Since Δ predicts Π, from Def. 2.1.6 we have

$$\text{SD} \cup \text{OBS} \cup \{\Pi\} \cup \{\neg\text{AB}(c) \mid c \in \text{COMP-}\Delta\} \text{ is consistent,}$$

i.e., COMP-Δ is not a CS for (SD, COMP, OBS∪{Π}). Assume that P is an MCS for (SD, COMP, OBS∪{Π}) resulting from Π and $P \cap \Delta = \varnothing$, then $P \subseteq$ COMP-Δ, i.e., COMP-Δ is a CS for (SD, COMP, OBS∪{Π}), a contradiction.

(⇐) Assume that Δ predicts $\neg\Pi$, then we have

$$\text{SD} \cup \text{OBS} \cup \{\Pi\} \cup \{\neg\text{AB}(c) \mid c \in \text{COMP-}\Delta\} \text{ is inconsistent,}$$

i.e., COMP-Δ is a CS for (SD, COMP, OBS∪{Π}). Since all MCSs for (SD, COMP, OBS∪{Π}) derived from COMP-Δ do not intersect Δ, we have a contradiction.

□

Therefore the above result shows that Theorem 2.1.7 is in fact equivalent to Corollary 3.12 in [29].

The picture of the relationships between the diagnoses before adding Π and the diagnoses after adding Π can be drawn directly from Theorem 2.1.7:

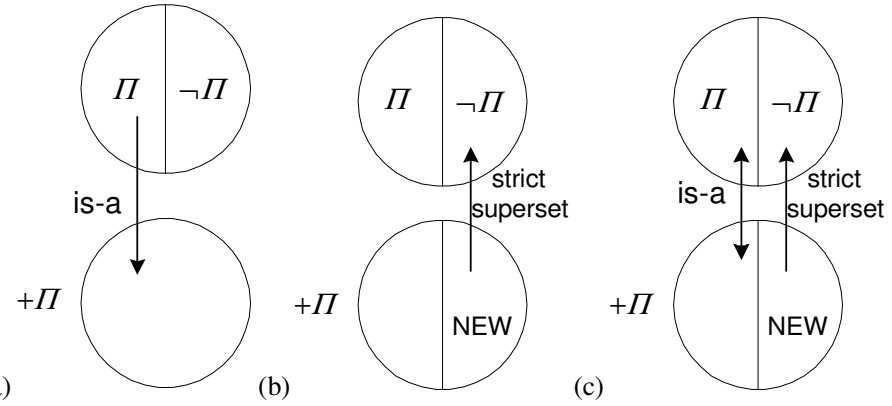

Fig. 2.2.2. The relationships between the diagnoses before adding Π and the diagnoses after adding Π depicted by: (a). Theorem 2.1.7 (1); (b). Theorem 2.1.7 (2) & (3); (c). Theorem 2.1.7 as a whole.

In the figure, a circle stands for a set of diagnoses. The upper diagnoses are divided into those predicting Π and those predicting $\neg\Pi$. The lower circle is the set of diagnoses after adding Π. The section denoted by 'NEW' is the collection of diagnoses for (SD, COMP, OBS∪{Π}) but not for (SD, COMP, OBS).

Fig. 2.2.2 essentially shows the way to compute all diagnoses for (SD, COMP, OBS∪{Π}). What we need to do is preserve all diagnoses for (SD, COMP, OBS) predicting Π, and then compute all 'new' diagnoses from the diagnoses for (SD, COMP, OBS) predicting $\neg\Pi$. To compute the new diagnoses, Hou introduced the idea of *candidates* as follows[5]:

Definition 2.2.8 A *candidate* k for (SD, COMP, OBS∪{Π}) is a set of components such that $k = \Delta \cup h$, where Δ is a diagnosis for (SD, COMP, OBS) predicting $\neg\Pi$ and h is an MHS for

[5] In [29], Hou did not give a formal definition for *candidates*, but he adopted the term 'candidate' for the same use. However, we find it helpful to single out the definition.

C, which is a collection of MCSs for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π such that $\forall P \in C$, P does not intersect Δ.

An immediate consequence follows from the definition above:

Corollary 2.2.9 *A candidate k for* (SD, COMP, OBS$\cup\{\Pi\}$) *is an HS for the collection of MCSs for* (SD, COMP, OBS$\cup\{\Pi\}$).

Next we show that every new diagnosis can be generated by some candidate.

Theorem 2.2.10 *Let Δ be a diagnosis for* (SD, COMP, OBS$\cup\{\Pi\}$) *but not for* (SD, COMP, OBS). *Then there exists a candidate k for* (SD, COMP, OBS$\cup\{\Pi\}$) *such that $k = \Delta$.*

Proof. By Theorem 2.1.7 (3), we know that $\Delta = \Delta_i \cup \Delta_j$, where $\Delta_i \cap \Delta_j = \varnothing$ and Δ_i is a diagnosis for (SD, COMP, OBS) predicting $\neg\Pi$. Since there must exist some candidate k such that $k = \Delta_i \cup h$, where h is an MHS for C, the collection of MCSs derived from COMP-Δ_i, all we have to do is prove that $\Delta_j = h$.

Assume that Δ_j is not an HS for C, then there must exist some $P \in C$ such that $P \cap \Delta_j = \varnothing$. Since Δ_i does not intersect any $P \in C$, Δ could not intersect every MCS for (SD, COMP, OBS$\cup\{\Pi\}$), a contradiction. Δ_j is also minimal, for if not, Δ could not be minimal.

□

As a matter of fact, the above result is equivalent to the theorem Hou suggested for computing all new diagnoses based on a measurement Π [29, Theorem 3.13]: (1) if a diagnosis for (SD, COMP, OBS) intersects all MCSs for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π, then it must be a diagnosis for (SD, COMP, OBS$\cup\{\Pi\}$) as well; (2) if a diagnosis for (SD, COMP, OBS) does not hit some MCSs, which form a collection C, for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π, then we simply compute the candidates, i.e., the union of the diagnosis and the MHSs for the collection C, and replace the diagnosis with all candidates which are not subsumed or duplicated by another candidate or diagnosis already generated. However, in the proof of the theorem, Hou did not show that every new diagnosis can be generated by at least one candidate.

It is interesting to know why a candidate could be non-minimal. Let Δ be a diagnosis for (SD, COMP, OBS) predicting $\neg\Pi$, and h be the MHS for C, the collection of MCSs which Δ does not intersect. Intuitively h patches the 'hole' that Δ misses. However, it is possible that h not only makes candidate k ($k = \Delta \cup h$) intersect each MCS in C, but also introduce for Δ more 'contact' with some MCS that Δ already touches. The following theorem states a sufficient condition under which the candidate must be *minimal*.

Theorem 2.2.11 *Let k be a candidate for* (SD, COMP, OBS$\cup\{\Pi\}$) *such that $k = \Delta_i \cup h$, where Δ_i is a diagnosis for* (SD, COMP, OBS) *predicting $\neg\Pi$ and h is an MHS for C, the collection of MCSs derived from* COMP-Δ_i. *If $P \cap h = \varnothing$* where *P is an MCS for* (SD, COMP, OBS), *then k is minimal*.

Proof. Since Δ_i is a diagnosis for (SD, COMP, OBS), $\Delta_i - \{c\}$ does not intersect some MCS P for (SD, COMP, OBS), where $c \in \Delta_i$. If P is also an MCS for (SD, COMP, OBS$\cup\{\Pi\}$) and $P \cap h = \varnothing$, $(\Delta_i - \{c\}) \cup h$ cannot intersect P, therefore $k - \{c\}$ cannot intersect P.

If P is not an MCS for (SD, COMP, OBS$\cup\{\Pi\}$), then by Theorem 2.2.5, P must be a

strict superset of some P', where P' is an MCS for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π. Since $\Delta_i - \{c\}$ does not intersect P' and $P' \cap h = \varnothing$, $(\Delta_i - \{c\}) \cup h$ cannot intersect P', hence $k - \{c\}$ cannot intersect any such P'.

□

2.3 Enhanced Conflict Recognition

First in Section 2.3.1 we make a concise review of Hou's consequences in [29]. Then in Section 2.3.2 we give comments on Hou's results.

2.3.1 Review of Hou's Consequences

From Theorem 3.13 in [29], we know that to compute all new diagnoses for (SD, COMP, OBS$\cup\{\Pi\}$), one has to derive all MCSs from COMP-Δ_i, where Δ_i is a diagnosis for (SD, COMP, OBS) predicting $\neg\Pi$. The process of the derivation requires a test for each subset of COMP-Δ_i by calling a sound and complete prover, therefore it needs substantial computation power[6]. To ease the situation, Hou showed that some Δ_i can be safely ignored from the derivation process by first introducing the notion of *equivalence* between two components and *homogeneity* between two diagnoses.

Definition 2.3.1 [29, Def. 4.1] For the system (SD, COMP, OBS), $\forall c_1, c_2 \in$ COMP, c_1 and c_2 are defined to be *equivalent* if $c_1 \in P \Leftrightarrow c_2 \in P$, where P is an MCS for (SD, COMP, OBS). This relation is indicated by $c_1 \sim c_2$.

Definition 2.3.2 [29, Def. 4.2] Two diagnoses Δ_1 and Δ_2 for (SD, COMP, OBS) are defined to be *homogeneous* if $\forall c_i \in \Delta_1$, there must exist some $c_j \in \Delta_2$ such that $c_i \sim c_j$, and $\forall c_j \in \Delta_2$, there must exist some $c_i \in \Delta_1$ such that $c_j \sim c_i$. This relation is indicated by $\Delta_1 \sim \Delta_2$.

Note that both the equivalence relation and the homogeneity relation are symmetric, transitive and reflexive. Next Hou showed that under certain condition, a homogeneous diagnosis predicting $\neg\Pi$ can be safely removed from the derivation process without losing any MCSs.

Theorem 2.3.3 [29, Theorem 4.5] *Suppose that Δ_1 and Δ_2 are two diagnoses for* (SD, COMP, OBS) *which predict $\neg\Pi$ and $\Delta_1 \sim \Delta_2$. If all MCSs for* (SD, COMP, OBS$\cup\{\Pi\}$) *derived from* COMP-Δ_1 *do not intersect Δ_2, then all MCSs derived from* COMP-Δ_1 and COMP-Δ_2, *respectively, are the same ones*.

The above theorem provides an efficiency enhancement: when two diagnoses Δ_1 and Δ_2 predicting $\neg\Pi$ are homogeneous, if all MCSs derived from COMP-Δ_1 do not intersect Δ_2, one can safely ignore Δ_2, i.e., one does not need to derive any MCS from COMP-Δ_2. However, to exploit Theorem 2.3.3, one has to maintain equivalence classes of components and homogeneity classes of diagnoses when adding more measurements. The following theorem points out the dynamics of the equivalence relation when adding measurement Π:

Theorem 2.3.4 [29, Theorem 4.7] *For every $\alpha \not\sim \beta$ for* (SD, COMP, OBS), *we still have $\alpha \not\sim \beta$ for* (SD, COMP, OBS$\cup\{\Pi\}$).

In other words, to maintain equivalence classes of components, one follows the procedure listed below:

[6] Later in Section 2.4 we will describe an improved procedure for deriving all MCSs from a CS.

Procedure 2.3.5 [29, p. 307] Update equivalence classes of components as follows:

(1) For every $\alpha\sim\beta$ for (SD, COMP, OBS), if one determines $\alpha\sim\beta$ based on all MCSs P_1, ..., P_m for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π, then one has $\alpha\sim\beta$ for (SD, COMP, OBS$\cup\{\Pi\}$). If one determines $\alpha\not\sim\beta$ based on P_1, ..., P_m, then one has $\alpha\not\sim\beta$ for (SD, COMP, OBS$\cup\{\Pi\}$).
(2) For some $\alpha\sim\beta$ for (SD, COMP, OBS), if all P_1, ..., P_m do not include both α and β, then one keeps $\alpha\sim\beta$ for (SD, COMP, OBS$\cup\{\Pi\}$).
(3) For every $\alpha\not\sim\beta$ for (SD, COMP, OBS), one must have $\alpha\not\sim\beta$ for (SD, COMP, OBS $\cup$ $\{\Pi\}$), and pay no particular attention to whether $\alpha\sim\beta$ or $\alpha\not\sim\beta$ based on P_1, ..., P_m.
(4) For every new $\alpha\sim\beta$ determined by all MCSs P_1, ..., P_m resulting from Π, if α and/or β do not occur in any previous equivalence class, namely α and/or β are not the members of any old MCS, then one keeps $\alpha\sim\beta$ for (SD, COMP, OBS$\cup\{\Pi\}$).

To determine homogeneity classes of diagnoses, one simply compares each diagnosis with another, and determines if $\Delta_i \sim \Delta_j$ [29, Section 5.1].

2.3.2 Comments on Enhanced Conflict Recognition

First we show an observation of Def. 2.3.1:

Corollary 2.3.6 *If α and β are two components such that no MCS for* (SD, COMP, OBS) *contains them, then $\alpha\sim\beta$ for* (SD, COMP, OBS).

Corollary 2.3.6 in effect states that each and every component is in one equivalence class from the beginning, regardless whether it hits an MCS or not. However, in [29] it is Hou's intention to ignore all the components which do not hit any MCS from being considered in any equivalence class. This implication does not make any difference in deciding if two diagnoses are homogeneous or not since all the components considered in such decisions actually show up in at least one MCS. Nevertheless, if we follow Def. 2.3.1 strictly Procedure 2.3.5 (2) will be rendered redundant, and at the same time we have the following corollary derived directly from Def. 2.3.1 and Theorem 2.3.4:

Corollary 2.3.7 *An equivalence class will only split into small classes; it will never grow.*

Therefore by following this line of reasoning there cannot exist any *new* $\alpha\sim\beta$ for (SD, COMP, OBS$\cup\{\Pi\}$) described in Procedure 2.3.5 (4). This result makes Procedure 2.3.5 (4) redundant as well. Consequently, Procedure 2.3.5 can be simplified to consist of only two rules as follows.

Procedure 2.3.8 Update equivalence classes of components as follows:

(1) For every $\alpha\sim\beta$ for (SD, COMP, OBS), if one determines $\alpha\sim\beta$ based on all MCSs P_1, ..., P_m for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π, then one has $\alpha\sim\beta$ for (SD, COMP, OBS$\cup\{\Pi\}$). If one determines $\alpha\not\sim\beta$ based on P_1, ..., P_m, then one has $\alpha\not\sim\beta$ for (SD, COMP, OBS$\cup\{\Pi\}$).
(2) For every $\alpha\not\sim\beta$ for (SD, COMP, OBS), one must have $\alpha\not\sim\beta$ for (SD, COMP, OBS $\cup$ $\{\Pi\}$), and pay no particular attention to whether $\alpha\sim\beta$ or $\alpha\not\sim\beta$ based on P_1, ..., P_m.

Next let us take a closer look at diagnoses and equivalence relation.

Corollary 2.3.9 *Let Δ be a diagnosis for* (SD, COMP, OBS). *Then for all distinct d_i, $d_j\in\Delta$,*

$d_i \not\sim d_j$.

Proof. Assume that there exist distinct d_i, $d_j \in \Delta$ such that $d_i \sim d_j$. Then both d_i and d_j must hit the same MCS for (SD, COMP, OBS), for if not, by Def. 2.3.1 neither of them is a member of any MCS for (SD, COMP, OBS), and Δ cannot contain them. Without the loss of generality, assume that Δ-$\{d_i\}$ does not intersect some MCS P. But then P does not contain d_j, and $d_i \not\sim d_j$, a contradiction.

□

The above corollary then leads us to an useful property for updating the homogeneity relation:

Corollary 2.3.10 *If* $\Delta_1 \sim \Delta_2$, *then* $|\Delta_1| = |\Delta_2|$.

Proof. Assume that $|\Delta_1| < |\Delta_2|$, then by Def. 2.3.2 there exist $d_i \in \Delta_1$ and distinct d_j, $d_k \in \Delta_2$ such that $d_i \sim d_j$ and $d_i \sim d_k$. But then we have $d_j \sim d_k$, a contradiction.

□

Corollary 2.3.10 suggests an improvement on finding homogeneous diagnoses: we do not have to compare two diagnoses with different sizes. A flash of reflection then leads us to another would-be-useful corollary.

Corollary 2.3.11 *Let n be the number of equivalence classes. Then all diagnoses of size n are homogeneous.*

Proof. From Corollary 2.3.9 we know that for a diagnosis Δ such that $|\Delta| = n$, every $c \in \Delta$ comes from one of n different equivalence classes.

□

It seems that corollary 2.3.11 could be used to improve the efficiency of conflict recognition, however, it is not the case. It is interesting to see why rarely the size of a diagnosis reaches the magnitude of n. To be more specific, we shall prove that only when we have the same number of equivalence classes and MCSs, and no two MCSs intersect each other, does the cardinality of a diagnosis equal the number of equivalence classes. First we define an equivalence matrix as follows.

Definition 2.3.12 Let MCS_1, MCS_2, …, MCS_m be m minimal conflict sets and EC_1, EC_2, …, EC_n be n equivalence classes for (SD, COMP, OBS). An equivalence matrix (EM) for (SD, COMP, OBS) at a particular moment is defined as an $n \times m$ matrix such that the element $e_{i,j}$ at the intersection of the i-th row and the j-th column is 1 if $EC_i \subseteq MCS_j$, otherwise $e_{i,j}$ is 0.

From this point we shall denote the i-th row in an EM by R_i, the j-th column by C_j and the element at their intersection by $e_{i,j}$.

Now we give an example showing a particular EM for a set of MCSs and ECs.

Example 2.3.13 Assume that we have three minimal conflict sets: $MCS_1=\{c_1, c_2, c_3\}$, $MCS_2=\{c_1, c_2, c_4, c_5\}$, $MCS_3=\{c_4, c_6\}$. The corresponding equivalence classes are (EC_1, …, EC_6, respectively) $\{c_1, c_2\}$, $\{c_3\}$, $\{c_4\}$, $\{c_5\}$, $\{c_6\}$, $\{c_7\}$. Then we have the equivalence matrix EM as shown below.

$$\begin{array}{c} \\ EC_1 \\ EC_2 \\ EC_3 \\ EC_4 \\ EC_5 \\ EC_6 \end{array} \begin{array}{c} \begin{array}{ccc} MCS_1 & MCS_2 & MCS_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

An intuition follows immediately after the definition. Since a diagnosis is an MHS for the collection of MCSs, we can first choose a minimal set Ω of ECs such that Ω 'covers' each MCS, i.e. for each MCS it intersects some EC in Ω, then we pick up one element from each EC in Ω to form a diagnosis. The procedure is described below.

Procedure 2.3.14 Derive diagnoses by simplifying an EM
Let EM be an $n \times m$ equivalence matrix for EC_1, …, EC_n and MCS_1, …, MCS_m.

1. Let R denote the first row and Ω be an empty set.
2. If R has no entry with value 1, let R be the next row and return to step 2.
3. Add the corresponding EC of row R to set Ω.
4. For each entry with value 1 in R, delete the corresponding column. Also delete row R itself.
5. If the simplified EM still has elements, advance R to the next row and go back to step 2.
6. For each EC in Ω pick an element and collect them to form a new diagnosis.

Note that in the procedure above we follow a strict top-down order to simplify the equivalence matrix and therefore the final set of diagnoses we have is only a subset of the real solution. Though one can obtain the complete set of diagnoses by furnishing the procedure with additional backtracking mechanism, it suffices for our purpose to prove our claim concerning Corollary 2.3.11.

Now we give an example showing how to derive (partial) diagnoses by simplifying the EM of Example 2.3.13.

Example 2.3.15 Let EM denote the same equivalence matrix shown in Example 2.3.13. First we observe that the first row has two entries with value 1: the first and the second entry. We then delete the first and the second column together with the first row itself. The simplified EM is shown below.

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Next we observe that the second row of the simplified matrix has an entry with value 1, and

delete the only remaining column. The corresponding ECs we have chosen are $\{c_1, c_2\}$ and $\{c_4\}$. By picking one element from each of the ECs we have chosen we form two diagnoses $\{c_1, c_4\}$ and $\{c_2, c_4\}$. Note that the diagnoses we have derived are only part of the complete set of diagnoses.

Still we have other useful observations about Def. 2.3.12.

Theory 2.3.16 *Let $EM=\{e_{i,j}\}_{n\times m}$ be an equivalence matrix for (SD, COMP, OBS). Then we have the following properties:*

(1) If column C_j contains only one entry $e_{i,j}$ such that $e_{i,j}=1$, then for all elements $e_{i,k}$ with $k\neq j$, $e_{i,k}$ must be 0.

(2) If row R_i has more then one entry with value 1, then for each column C_p and C_q which has an entry with value 1 in row R_i, there must be element $e_{x,p}$ and $e_{y,q}$ with value 1, where $x\neq y\neq i$.

Proof.

(1) The assumption that there is an element $e_{i,k}$ with value 1 implies that $EC_i \subseteq MCS_k$. Since we have $MCS_j=EC_i$ based on the fact that the column C_j contains only one entry $e_{i,j}$ such that $e_{i,j}=1$, we conclude $MCS_j \subseteq MCS_k$, a contradiction.

(2) Let $\Theta_p=\{x \mid e_{x,p} = 1 \text{ in } C_p \text{ where } x\neq i\}$. Assume that $\Theta_p \subseteq \Theta_q$, then we have $MCS_p \subseteq MCS_q$, a contradiction.

Now we shall show that only when we have equal number of MCSs and ECs and no two MCSs intersect each other do we have a diagnosis with its size being equal to the number of ECs.

Theory 2.3.17 *Given m MCSs and n ECs, we have a diagnosis of size n iff m=n and no two MCSs intersect each other.*

Proof.

($\Rightarrow$)

Assume $m\neq n$, we have two cases shown below.

Case I: $m<n$

From Procedure 2.3.14 we know that every selected row must have at least one column deleted. By selecting each row (EC) we then must have at least n columns deleted, which is a contradiction.

Case II: $m>n$

Since every EC has to be selected during the simplification process, without loss of generality we can rearrange the order of columns so that the equivalence matrix is partitioned into an $n\times n$ diagonal matrix and an $n\times(m-n)$ matrix, as shown below.

$$\overbrace{\quad}^{n}\ \overbrace{\quad}^{m-n}$$
$$n\left\{\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \end{bmatrix}\right.$$

For column $C_{n+1}, \ldots, C_m$ each column must have at least one entry with value 1. In fact, by

Property (2) in Theory 2.3.16 we know that such column, C_j (n+1 ≤ j ≤ m), must have an additional entry with value 1, say entry $e_{i,j}$. Then again by Property (2) in Theory 5 we know C_i (0 ≤ i ≤ n) must have one additional entry $e_{k,i}$ with value 1, where $k \neq i$. Therefore in the left diagonal matrix at least one row R_k has at least two entries with value 1.

Now consider Procedure 2.3.14. Since in the diagonal matrix there exists one row which has at least two entries with value 1, by selecting this row at least two columns of the diagonal matrix will be deleted. Therefore during the first (n-1) row selections all columns in the diagonal matrix will be deleted. Now we show that all columns in the right $n \times (m-n)$ matrix will also be deleted before the last row selection could happen. Assume that column C_j (n+1 ≤ j ≤ m) remains after the first (n-1) row selections and deletions. Since we know that in the unsimplified matrix element $e_{n,n}$ = 1, by Property (2) in Theory 2.3.16 we know that at least one entry in C_j, say $e_{k,j}$ must have value 1, where $k<n$. Then C_j must have been deleted when selecting R_k, which is a contradiction.

(⇐)
Since no two MCSs intersect each other, no two columns can have an entry with value 1 in the same row. Furthermore, since $m=n$ we have entries with value 1 distributed in every row. This in turn implies that each column can only have one entry with value 1, for if not the corresponding MCS intersects another one. By Property (1) in Theory 2.3.16 we then know that each row contains only one entry with value 1, which guarantees that a diagnosis of size n exists.

Theory 2.3.17 clearly shows that only when we have the same number of MCSs and ECs and no two MCSs intersect each other could Corollary 2.3.11 be useful. Note that the condition here is equivalent to that each MCS is an EC by itself. In practical situations this rarely occurs since usually only a small subset of all components is involved in the conflict sets. Therefore our claim concerning Corollary 2.3.11 is verified.

Having the results above we are now ready to give a complete procedure for updating homogeneity classes of diagnoses. Although in Theorem 2.3.4 Hou has shown the dynamics of the equivalence relation when adding more measurements, a corresponding result concerning homogeneity classes of diagnoses, as shown below, is not presented.

Corollary 2.3.18 *For every* $\Delta_i \not\sim \Delta_j$ *for* (SD, COMP, OBS)*, we still have* $\Delta_i \not\sim \Delta_j$ *for* (SD, COMP, OBS∪{Π})*, where* Δ_i *and* Δ_j *are two diagnoses predicting* Π*.*

Proof.

Case I. $|\Delta_i| \neq |\Delta_j|$: By Corollary 2.3.10 we know $\Delta_i \not\sim \Delta_j$ for (SD, COMP, OBS∪{Π}).

Case II. $|\Delta_i| = |\Delta_j|$: Since there must be $\alpha \in \Delta_i$ and $\beta \in \Delta_j$ such that $\alpha \not\sim \beta$ for (SD, COMP, OBS), from Theorem 2.3.4 we know that $\alpha \not\sim \beta$ for (SD, COMP, OBS∪{Π}), which implies $\Delta_i \not\sim \Delta_j$ for (SD, COMP, OBS∪{Π}).

□

In summary, to update homogeneity classes of diagnoses, one follows the procedure below:

Procedure 2.3.19 Let Δ be a new diagnosis after adding a measurement Π. Update homogeneity classes of diagnoses as follows:

(1) For each homogeneity class, delete the diagnosis which predicts $\neg\Pi$.
(2) For every $\Delta_i \sim \Delta_j$ for (SD, COMP, OBS), determine if $\Delta_i \sim \Delta_j$ or $\Delta_i \not\sim \Delta_j$ based on the updated equivalence classes.
(3) For every $\Delta_i \not\sim \Delta_j$ for (SD, COMP, OBS), one must have $\Delta_i \not\sim \Delta_j$ for (SD, COMP, OBS $\cup$ $\{\Pi\}$), and pay no particular attention to whether $\Delta_i \sim \Delta_j$ or $\Delta_i \not\sim \Delta_j$ based on the updated equivalence classes.
(4) If $|\Delta| = m \neq n$, compare Δ with Δ_m, where $|\Delta_m| = m$, as follows: if $\forall d_i \in \Delta$ there exists $d_j \in \Delta_m$ such that $d_i \sim d_j$, then $\Delta \sim \Delta_m$. If no such Δ_m exists, Δ forms a singleton homogeneity class.

2.4 Conflict Set Derivation

From the discussions of Section 2.2.2, we know that to refine old diagnoses, one has to keep those diagnoses predicting Π and recompute those predicting $\neg\Pi$. For a diagnosis predicting $\neg\Pi$, say, Δ_i, one determines new diagnoses from the collection of candidates $\Delta_i \cup h_j$, where h_j is the MHS for the collection of MCSs derived from COMP-Δ_i. To determine which candidate is accepted, one needs to check if it is minimal against other diagnoses and candidates.

To construct a practical diagnostic system, a complete and efficient method for deriving MCSs from a CS is vital. However, we find that in some cases Hou's method for the derivation fails to find all MCSs, hence it is incomplete. In Section 2.4.1, we describe Hou's method for the derivation and show two examples in which some MCS is missing from the derivation. In Section 2.4.2, we develop a complete method for deriving MCSs from a CS, illustrate the method with an example, and prove that the method is correct and at the same time efficient in that no *redundant* subsets are tested. In Section 2.4.3, we show two more possible improvements to our derivation method: exploiting extra information about the system being diagnosed and using an incremental prover. Finally, we describe a bottom-up derivation strategy, compare it with the top-down method described in Section 2.4.4, and conclude this chapter with discussions on the possible extensions to our method in Section 2.4.5.

2.4.1 Hou's Method for Deriving Minimal Conflict Sets

Definition 2.4.1 [29, Def. 5.1] A CS-tree T rooted in a CS C is defined as follows:

(1) Its root is labeled by C;
(2) Each node n of T is labeled by a set $S \subseteq C$. For each $c \in S$, node n has a descendant n_c such that node n_c is labeled by a non-empty set $S-\{c\}$[7].

To derive all MCSs from C, a pruned CS-tree T' rooted in C is generated by the following procedure.

Procedure 2.4.2 [29, p.312] Generate a pruned CS-tree T' rooted in a CS C by the following rules:

(1) Generate T' depth-first, i.e., generate the descendants of a node before generating its brothers.
(2) Pruning rules:
  (a) If node n is labeled by a non-CS, then we close node n.
  (b) Let node n be some node already generated in T' with the label set S. If node n' is a

[7] The definition here is slightly different from [29, Def. 5.1] in that we view the closing of a node labeled by a non-CS as a pruning rule (in pruning rule (a), Procedure 2.4.2) instead of part of the definition of a CS-tree. This viewpoint facilitates the comparison between Hou's method and our approach. Also, we preclude empty sets from being used for labeling any node in a CS-tree.

new node such that n' will be labeled by S, then we close node n'.

(c) If a non-root node n will be labeled by a set S and S is a proper superset of some MCSs already used for labeling some node in T', then we close node n.

If a node n is labeled by a CS, and all of its descendants are labeled by non-CSs, or it has no descendants at all, then the label set of node n is an MCS. It turns out that in some occasions pruning rule (c) tends to close some node too early, thus introduces errors to a diagnostic system. Consider the following example:

Example 2.4.3 Let $C = \{c_1, c_2, c_3, c_4\}$ and suppose that all the MCSs we can derive from C are $\{c_1\}$, $\{c_2, c_3\}$ and $\{c_2, c_4\}$. In Fig. 2.4.1 we use Hou's method to derive the MCSs from C:

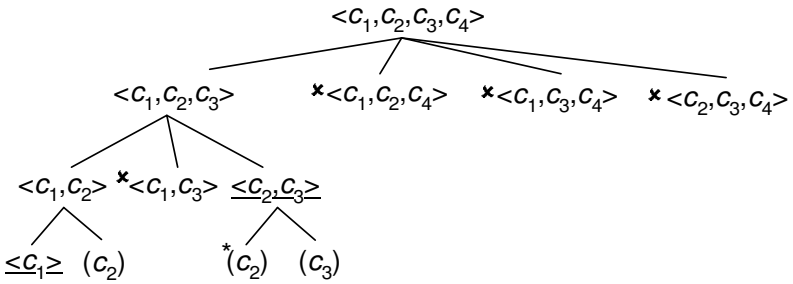

Fig. 2.4.1. Deriving the MCSs from C, using Hou's method

Note that we use <.> to denote a CS and (.) to denote a non-CS in the figure. Also, we underline a set to denote an MCS. The mark * on a node n indicates that n is closed due to the use of pruning rule (b), and the mark × on a node n shows that n is closed due to the use of pruning rule (c). Note that $\{c_2, c_4\}$ is lost during the derivation due to the use of pruning rule (c). Had we changed the order of generating the nodes in the CS-tree as shown in Fig. 2.4.2, the derivation would have yielded the correct result. Therefore Hou's method is order-dependent.

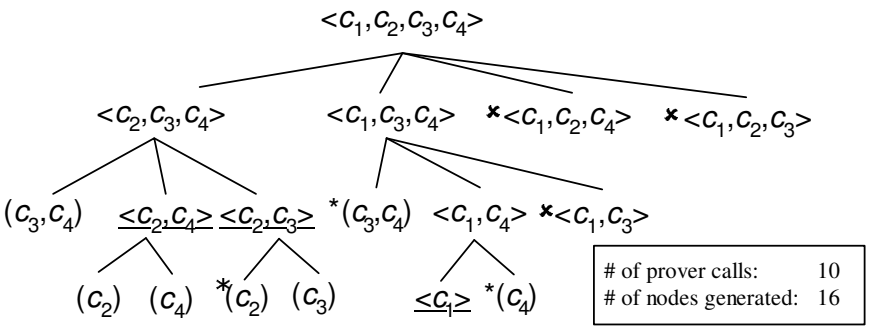

Fig. 2.4.2. Deriving the MCSs from C (alternative order), using Hou's method

During the derivation we use a prover to test the consistency of a label set if necessary. Since we assume that the set labeling the root node is a CS, the number of prover calls shown in the figure does not include the test for the set labeling the root node. Also note that we do not include the root node when counting the number of the nodes generated.

□

We now give an example circuit which shows that Hou's method for deriving MCSs is incomplete.

Example 2.4.4 Consider the circuit shown in the following figure.

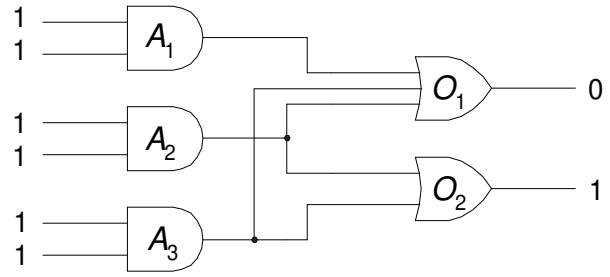

Fig. 2.4.3. An example circuit on which Hou's MCS derivation method fails

After adding all inputs as measurements, we add the measurement at the output of O_2. Since the initial diagnosis {} confirms the measurement, no derivation is needed. Next we add the measurement at the output of O_1. The diagnosis {} is rejected, and we have to derive all MCSs from $\{A_1, A_2, A_3, O_1, O_2\}$. If Hou's method were correct, we should have derived four MCSs: $\{A_1, O_1\}$, $\{A_2, O_1\}$, $\{A_3, O_1\}$ and $\{O_1, O_2\}$. However, as the following figure shows, $\{A_3, O_1\}$ and $\{O_1, O_2\}$ are lost during the derivation. The diagnoses found according to the wrong result are $\{O_1\}$ and $\{A_1, A_2\}$, where $\{A_1, A_2\}$ is obviously an erroneous diagnosis.

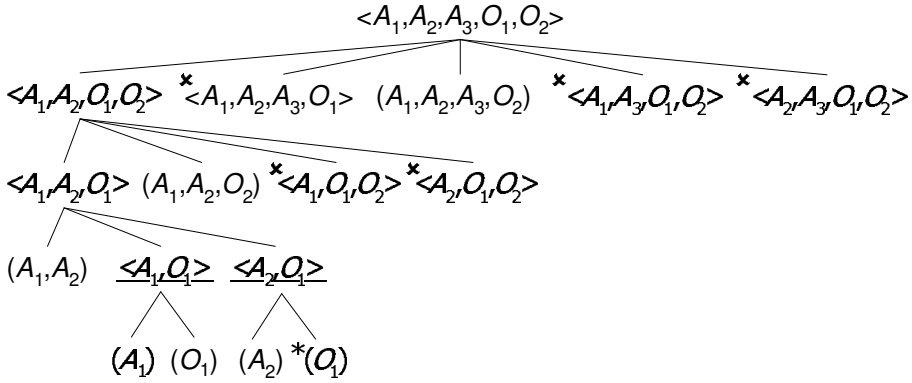

Fig. 2.4.4. Deriving all MCSs for the circuit in Fig. 2.4.3 using Hou's method

□

2.4.2 A Complete Method for Deriving Minimal Conflict Sets

First we define a *CS-tree with mark set* as follows.

Definition 2.4.5 A *CS-tree with mark set* T_M rooted in a CS C is defined as follows:

(1) Its root is labeled by $[C, \varnothing]$;

(2) Each node n of T_M is labeled by $[S_n, S_{M,n}]$, where $S_n \subseteq C$ is the *label set* of node n, and $S_{M,n} \subseteq S_n$ is the *mark set* of node n. If $S_{M,n} = S_n$ or $|S_n| = 1$[8], then node n has no descendants. Otherwise for each $c \in S_n - S_{M,n}$, node n_c is a descendant of node n such that node $n_{c'}$ is the *immediate* left brother of node n_c and node n_c is labeled by $[S_{M,n} \cup (S_n - S_{M,n} - \{c\}), S_{M,n_{c'}} \cup \{c'\}]$.

(3) For the leftmost node n in every subtree of T_M, $S_{M,n} = S_{M,n_p}$, where n_p is the parent node of node n.

[8] Hence we preclude empty sets from being the label sets of any node in a CS-tree with mark set.

To derive all MCSs from C, we generate a pruned CS-tree with mark set T_M' rooted in C by the following procedure.

Procedure 2.4.6 Generate a pruned CS-tree with mark set T_M' rooted in a CS C by the following rules:

(1) Generate T_M' depth-first, i.e., generate the descendants of a node before generating its brothers.
(2) Pruning rule:
   (a) If S_n of node n is not a CS, then we close node n.
   (b) If $S_{M,n}$ of node n is a superset of some MCS already used as a label set of some node in T_M', then we close node n and do not generate any right brothers of node n.

If a node n is labeled by a CS, and all of its descendants are labeled by non-CSs, or it has no descendants at all, then the label set of node n is an eligible MCS – one has to check if it is minimal against all other CSs found.

Note that not every label set requires a prover call for testing if it is a CS, because a superset of a known CS is a CS.

The intuition of the use of 'mark sets' is straightforward. In a T_M with the root node n_r labeled by $[C, \varnothing]$, for the leftmost descendant node n_1 labeled by $[C\text{-}\{c_1\}, \varnothing]$ we can derive all proper subsets of $C\text{-}\{c_1\}$ beneath the node. For the immediate right adjacent node n_2 of n_1, whose label set is $C\text{-}\{c_2\}$, to avoid generating duplicate sets when generating the proper subsets of $C\text{-}\{c_2\}$, i.e., to avoid generating some proper subset S' of $C\text{-}\{c_2\}$ such that $S' \subset C\text{-}\{c_1\}$, we mark c_1 and include c_1 first in every subset we generate. This way we are insured that no node beneath node n_2 will be labeled by a set already used for labeling some node beneath node n_1.

Now we give a formal proof of the correctness of our method. First we give a corollary directly from Def. 2.4.5.

Corollary 2.4.7 *The label set of each descendant and right brother of node n is a superset of the mark set of node n.*

Next we show that all nonempty proper subsets containing set $S_{M,n}$ of a set S_n can be generated in the CS-tree with mark set rooted in a node labeled by $[S_n, S_{M,n}]$.

Theorem 2.4.8 *For a node n labeled by $[S_n, S_{M,n}]$, each and every nonempty proper subset S_n' of S_n such that $S_{M,n} \subseteq S_n'$ is the label set of some node beneath node n.*

Proof. We prove it by induction on $|S_n|$. If $|S_n| = 2$, say $|S_n| = \{c_1, c_2\}$, then the following figure shows that the theorem holds under all possibilities.

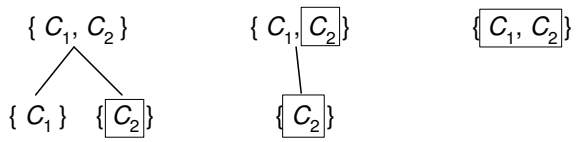

Fig. 2.4.5. Three possible subtrees in a CS-tree with mark set rooted in $\{c_1, c_2\}$

Note that the mark set of each node is denoted by putting a box around the elements it contains.

Now let $|S_n| = 3$. We first consider the case $|S_{M,n}| = 0$. In this case, the CS-tree with mark set consists of a root node labeled by $[\{c_1, c_2, c_3\}, \varnothing]$ and three subtrees joined the root node. The three subtrees are the ones shown in Fig. 2.4.5 with c_2 being replaced by c_3 in the middle tree and c_1 being replaced by c_3 in the right tree. Now consider the case $|S_{M,n}| = 1$. Without loss of generality we let $S_{M,n} = \{c_3\}$. From Corollary 2.4.7 we know that each node in the subtree rooted in node n is labeled by a set containing c_3. By stripping c_3 from each and every label set in the subtree we obtain a simplified CS-tree with mark set identical to the left tree shown in Fig. 2.4.5. Therefore we have two label sets in the subtree: $\{c_1, c_3\}$ and $\{c_2, c_3\}$, which are the all nonempty proper subsets of $\{c_1, c_2, c_3\}$ containing c_3. Similarly, the theorem holds when $|S_{M,n}| = 2$ and $|S_{M,n}| = 3$.

Assume that the theorem holds when $|S_n| = k$-1. Consider the case $|S_n| = k$. If $|S_{M,n}| = 0$, node n has k descendant nodes, and each node has a label set of size k - 1. According to the induction hypothesis, each and every nonempty proper subset S_n' of S_n is used to label a node in the CS-tree with mark set rooted in node n. If $|S_{M,n}| > 0$, we can strip one element of the mark set of the root node from each and every node in the tree. Since the root node has a label set of size k-1, each and every nonempty proper subset S_n' of S_n such that $S_{M,n} \subseteq S_n'$ must be used to label some node beneath node n.

□

From the above theorem we know by generating a CS-tree with mark set rooted in a node labeled by $[C, \varnothing]$, we generate each and every possible nonempty proper subset of C in the tree. Next we show that the pruning rule in Procedure 2.4.6 is correct. Since any subset of a non-CS is not a CS, we can safely prune away those nodes with non-CSs as their label sets, hence pruning rule (a) is correct. The correctness of pruning rule (b) is a direct result from Corollary 2.4.7, as stated in the following corollary.

Corollary 2.4.9 *Let T_M' be a pruned CS-tree with mark set. If the mark set of node n is a superset of some MCS P already used for labeling some node in T_M', then the label set of each descendant and right brother of node n is a CS.*

Corollary 2.4.7, Theorem 2.4.8 and Corollary 2.4.9 together prove the correctness of our method. Now we consider the efficiency of our approach. For comparison, note that Hou's method for the derivation will generate the nodes with duplicate labels; e.g., the node labeled by $\{c_2\}$ in Fig. 2.4.1 and the nodes labeled by $\{c_2\}$, $\{c_4\}$ and $\{c_3, c_4\}$ in Fig. 2.4.2 are duplicates, respectively, in the two derivations. The duplicates are then closed by using pruning rule (b) in Procedure 2.4.2. By contrast, our method prevents any node labeled by a duplicate set from being generated by exploiting the notion of mark sets, hence no test for duplicates is necessary in our approach. The following theorem states that no duplicate label set exists in a CS-tree with mark set.

Theorem 2.4.10 *Let T_M be a CS-tree with mark set rooted in a CS C. If $C' \subset C$, then only one node n exists in T_M such that C' is the label set of node n.*

Proof. We show it by induction on $|C|$. If $|C| = 2$, say $C = \{c_1, c_2\}$, then by Def. 2.4.5 each of $\{c_1\}$ and $\{c_2\}$ is used as a label set of one node in T_M, as already shown in Fig. 2.4.5 (the left tree).

Now assume that the theorem is valid when $|C| = k$-1, where C is a CS. When $|C| = k$, $C = \{c_1, \ldots, c_k\}$ is the label set of the root node n. By Def. 2.4.5 we know that n has k descendant nodes $n_1, \ldots, n_k$ whose label sets are $C_i = C - \{c_i\}$, $i = 1, \ldots, k$, respectively. If $C' \subset C_q$, by the induction hypothesis, there is only one node in the subtree rooted in C_q labeled by C'. From Def.

2.4.5 and Corollary 2.4.7, we know that the label set of each and every right brother of node n_q must contain c_q. Since $c_q \notin C'$, no node in the subtrees rooted in the right brothers of n_q is labeled by C'.

On the other hand, we need to prove that no node in the subtrees rooted in the left brothers of n_q has C' as its label set. Assume that the subtree rooted in node n_p has a node with C' as its label set, with n_p being a left brother of n_q. Since C_p does not contain c_p, neither does C'. However, since n_q is a right brother of n_p, from Def. 2.4.5 and Corollary 2.4.7 C_q and all the label sets of the descendants of n_q must contain c_p, so does C', a contradiction.

□

Another efficiency improvement of our method is: if C' is not a CS, then no node with $C'' \subseteq C'$ as its label set will be generated in a pruned CS-tree with mark set. By contrast, Hou's pruned CS-tree does have some node labeled by the subset of some known non-CS, e.g., the nodes labeled by $\{c_3\}$ and $\{c_4\}$ in Fig. 2.4.2 are redundant. In summary, our method has efficiency edges over Hou's approach in that we do not generate any node with a duplicate set or a subset of a known non-CS as its label set, while Hou generates them and prunes them away afterward.

Let us now illustrate our approach by deriving the MCSs from the same set in Example 2.4.3, using our method.

Example 2.4.11 Let $C = \{c_1, c_2, c_3, c_4\}$ and suppose that all the MCSs we can derive from C are $\{c_1\}$, $\{c_2, c_3\}$ and $\{c_2, c_4\}$ (the same as those in Example 2.4.3). The following figure shows the derivation using our method:

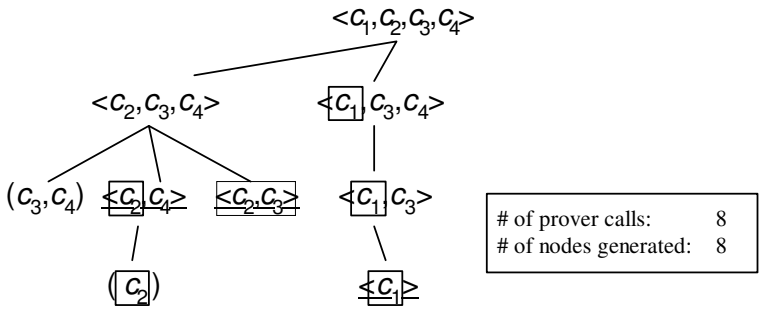

Fig. 2.4.6. Deriving the MCSs from C, using our method

Note how the numbers of prover calls and the nodes generated are reduced by using our method (compare them with the numbers shown in Fig. 2.4.2). Since all possibilities of generating the nodes labeled by duplicate sets are *implicitly* expelled by using the notion of mark sets, no node is labeled by * in Fig. 2.4.6.

□

One of the differences between Hou's method and our approach is that Hou closes a node whenever the label of it is a superset of some MCSs already discovered, while we close a node only when its mark set is a superset of some MCSs already found. This difference leads to the incompleteness of Hou's approach.

In another example, where $C = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$ and the only MCSs are $\{c_2\}$ and $\{c_4\}$, Hou's method required 15 prover calls and generated 41 nodes, while our method called the prover 14 times and generated 21 nodes. Note that in some orderings Hou's method failed

to generate all MCSs. Also, since it over-pruned the nodes, Hou's method could report a smaller number of prover calls in some orderings.

In summary, our method has three improvements over Hou's approach:

(1) Our method is capable of finding all MCSs from a given CS independent of the order of generating nodes in the CS-tree with mark set, therefore it is complete. By contrast, Hou's approach is incomplete because its pruning rule (c) tends to prune away some MCSs in some node generating orderings.
(2) By exploiting the concept of mark set, we *implicitly* discard any possibility of generating the nodes labeled by duplicate sets or the subsets of some known non-CS. The counterpart of the duplicate test is done in pruning rule (b) in Hou's method, which requires a search in the CS-tree, and no facility for testing the subset of some known non-CS is available in Hou's approach. Hence our method achieves better efficiency than Hou's approach.
(3) Our pruning rules are simpler than Hou's (2 rules vs. 3 rules), and yet are more powerful with respect to the completeness issue.

2.4.3 Further Improvements to Our Method

In this section we show two more possible improvements to our derivation method. In Section 2.4.3.1 we discuss the benefit brought by extra information about the system being diagnosed. In Section 2.4.3.2 an incremental prover is introduced to further speed up the derivation process.

2.4.3.1 Derivation with Extra Information

In this section we show that with the aid of extra information about the system being diagnosed, we can further improve the efficiency of our derivation method. First we define an input/output of a component of the system being diagnosed to be *fixed* if it is either in the observation of the system, or it is one of the previous measurements taken.

Example 2.4.12 Consider the faulty circuit shown in the following figure:

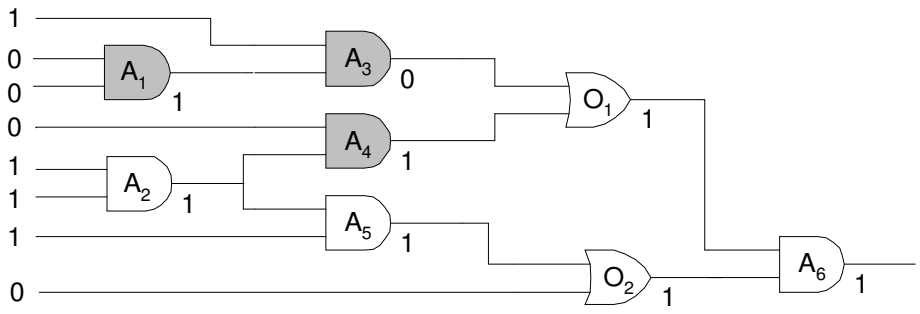

Fig. 2.4.7. A faulty circuit. The actual value of the input/output of each gate is shown in the figure. Also, the shaded gates are the ones causing the system to misbehave.

The two inputs of gate A_2 are *fixed* because they are part of the system observation. If we make a measurement at out(A_2), then both inputs of gate A_4 are fixed as well.

□

Now suppose Π is a measurement at the output of a component c such that all inputs of c are fixed, and Δ is a diagnosis for (SD, COMP, OBS) predicting $\neg\Pi$ such that $c \notin \Delta$, then by Def. 2.1.1 (1) (a) we know that SD $\cup$ OBS $\cup$ $\{\Pi\}$ $\cup$ $\{\neg$AB(c)$\}$ is *inconsistent*. Formally speaking,

we have the following corollary.

Corollary 2.4.13 *Let Π be a measurement at the output of a component c such that all inputs of c are fixed, and let Δ be a diagnosis for* (SD, COMP, OBS) *predicting $\neg\Pi$ such that $c \notin \Delta$. Then $\{c\}$ is an MCS for* (SD, COMP, OBS$\cup\{\Pi\}$) *resulting from Π.*

The above corollary offers an efficiency improvement as illustrated in the following example.

Example 2.4.14 Consider the faulty circuit in Fig. 2.4.7. After considering the observation only, we have five diagnoses: $\{A_1\}$, $\{A_3\}$, $\{A_4\}$, $\{A_6\}$ and $\{O_1\}$. Assume that we have taken a measurement at out(A_2). Now we take a measurement, Π, at out(A_4). It then turns out that $\{O_1\}$ predicts $\neg\Pi$, and we have to derive all MCSs from $\{A_1, A_2, A_3, A_4, A_5, A_6, O_2\}$. Since by Corollary 2.4.13 we know $\{A_4\}$ is an MCS for (SD, COMP, OBS$\cup\{\Pi\}$) resulting from Π, no prover call is necessary for determining if it is an MCS. Moreover, we can arrange the order of generating the nodes in the pruned CS-tree with mark set as shown in the following figure:

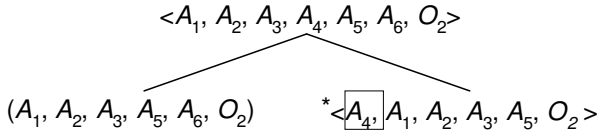

Fig. 2.4.8. Deriving the MCSs from $\{A_1, A_2, A_3, A_4, A_5, A_6, O_2\}$

In the derivation, we only call the prover once [for determining that $(A_1, A_2, A_3, A_5, A_6, O_2)$ is a non-CS], and only two nodes are generated in the tree. Without the use of Corollary 2.4.13, we would have called the prover 12 times and generated 17 nodes by using our method described in Section 2.4.2 alone.

□

Note that despite the great efficiency improvement brought by Corollary 2.4.13, one has to maintain extra information indicating if a particular connection is fixed or not.

2.4.3.2 Derivation with an Incremental Prover

During the derivation of MCSs using a CS-tree with mark set, what we do is insert lots of unit assumptions and test if these assumptions are consistent with the system description and the observation. Along a search path in a CS-tree with mark set, we drop unit assumptions away one by one and invoke a prover for testing the consistency. The simplest way to do the test is insert each unit assumptions at a CS-tree node and run the prover from scratch. However, it is not efficient because all of the efforts done in the parent CS-tree node are thrown away completely.

A straightforward way to improve the efficiency here is to use an incremental prover for testing. At the root node, we insert each unit assumption into the prover and place a mark at the point where the unit inferences *caused* by the unit assumption run out. On generating the descendant, we simply *retract* one unit assumption and all of its accompanying unit inferences to the previous mark, and run the prover from that mark again. When starting a new branch, we insert the mark set elements first, followed by the remaining label set elements. Then we retract the unit assumptions one by one as described above. The reason to insert mark set elements first is that the label sets of all the descendants and the right brothers of a node in a CS-tree with mark set must be supersets of the mark set of that node (as described in Corollary 2.4.7), hence we have to reserve them from retractions so that we can reuse the unit inferences caused by them.

As a matter of fact, the incremental prover described above exploits the idea of *chronological backtracking* [47]. Although simple enough, it is sometimes inefficient because by backtracking to the mark placed at the point where the previous unit assumption is inserted, we might throw some useful inferences away as well, i.e., the inferences which are not related to the unit assumption just retracted. To tackle this problem, one can use a truth maintenance system (TMS) [17] to implement the incremental prover, of course, at the cost of keeping track of dependencies of all inferences.

2.4.4 Top-down vs. Bottom-up

First we define an inverse *CS-tree* as follows.

Definition 2.4.15 An inverse *CS-tree* T_I of an ordered CS C is defined as follows:

(1) Its root is labeled by $\varnothing$;
(2) Let n be a node in T_I, n_p be the parent node of node n and n' be the left brother of node n. Then node n is labeled by an ordered set S_n such that $S_n = S_{n_p} \cup c$, where c is the right adjacent element of c' in C and c' is the last element in $S_{n'}$.
(3) For the leftmost node n in every subtree of T_I, $S_n = S_{n_p} \cup c$, where n_p is the parent node of node n, c is the right adjacent element of c_P in C and c_P is the last element in S_{n_p}. If $S_{n_p} = \varnothing$, c is the first element of C.

To derive all MCSs from an ordered CS C, we generate a pruned inverse CS-tree by the following procedure.

Procedure 2.4.16 Generate a pruned inverse CS-tree T_I' of an ordered CS C by the following rules:

(1) Generate T_I' depth-first, i.e., generate the descendants of a node before generating its brothers.
(2) Pruning rule: If S_n of node n is a CS, then we close node n.

Every CS found in the course of constructing the inverse CS-tree is an eligible MCS – one needs to check if it is minimal against all other CSs found. Note that not every label set requires a prover call for testing if it is a CS: if a label set is a subset of some known non-CS, it must not be a CS, either.

An interesting point to note here, is that generating nodes as depicted in Procedure 2.4.16 never yields any duplicate label set. Moreover, no newly generated label set is a superset of any label set already exists. This last fact precludes the possibility of generating a label set which is a superset of a CS already found.

Example 2.4.17 Let $C = \{c_1, c_2, c_3, c_4\}$ and suppose that all the MCSs we can derive from C are $\{c_1\}$, $\{c_2, c_3\}$ and $\{c_2, c_4\}$ (the same as those in Example 2.4.3). The following figure shows the derivation using an inverse CS-tree:

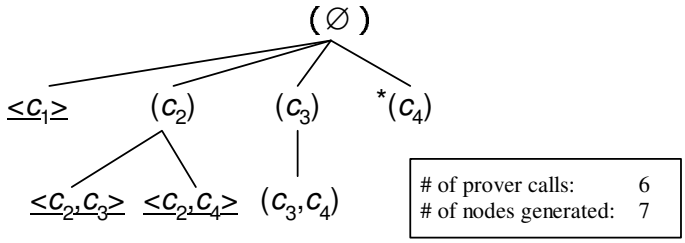

Fig. 2.4.9. Deriving the MCSs from *C*, using an inverse CS-tree

As in the previous examples we do not include the root node when counting the number of prover calls and the number of the nodes generated. A mark * on a node indicates that no prover call is necessary for testing if the label set of the node is a CS or not.

□

One might get the impression from the above example that the bottom-up approach is more efficient than the top-down method described in Section 2.4.2. However, such conclusion is not necessarily true for all cases. Consider the case in which we have to derive all MCSs from a CS C with n elements, and assume that all we can derive from C is an MCS with m elements ($m \le n$). Therefore among the 2^n subsets of C we have 2^{n-m} CSs, and the probability of having a CS as the label set of a node, P_{CS}, is $1/2^m$. Since we need a CS to close a node when using the bottom-up method, the lower P_{CS} the worse the efficiency of the bottom-up method. On the other hand, we need a non-CS to close a node when using the top-down method, lower P_{CS} could improve the efficiency of the top-down approach. In general, whether the top-down method has an efficiency edge over the bottom-up approach depends on the nature of the particular problem to be solved.

2.5 Discussions

In this chapter we have given comments on Hou's theory described in [29]. To be more specific, in Section 2.2 we have described the complete picture of the relationships among measurements, CSs and diagnoses, and related Reite's theorems with Hou's consequences. In Section 2.3 we have given comments on the equivalence relation defined by Hou for enhanced conflict recognition. In Section 2.4, we have indicated the incompleteness of Hou's approach for deriving all MCSs from a CS, and then proposed an improved method for the derivation.

To conclude this chapter, we point out that there are still some directions which can be pursued in this research. One possible extension to the work of this chapter would be adding a reordering mechanism to our MCS derivation approach. As shown in Example 2.4.14, such reordering could bring dramatic improvement to the efficiency of the derivation. This makes one wonder if one can find an optimal ordering of node generation in a CS-tree with mark set if one knows all MCSs before the derivation. If it is true, then we may use the error rate of each of the components within the system being diagnosed to 'approximate' such optimal ordering by first calculating all possible MCSs based on some assigned error threshold. Then we may use the result to guide us to generate nodes in a more efficient order. Moreover, we may discard some node in the construction of a CS-tree with mark set if we determine that all possible MCSs found beneath that node are not important enough (again based on some assigned error threshold). In this way, the diagnostic system will not be complete, but its efficiency could be greatly improved.

Chapter 3

Measurement Selection in Diagnosis from First Principles via the Genetic Algorithm

This chapter presents a *m*easurement *s*election *s*trategy using a steady-state with least-fit-deletion *g*enetic *a*lgorithm (MSSGA). This approach has been incorporated into our diagnostic system for combinatorial digital circuits (DSDC). Five testing circuits were diagnosed and the results are shown in this chapter.

3.1 Introduction

In a nontrivial diagnostic task using model-based approaches, such as Reiter's theory of diagnosis from first principles [39], one is always confronted with the problem of discriminating among competing diagnoses obtained. To narrow down the candidate diagnoses, a measurement has to be taken from the system being diagnosed. Under the assumption that the measurement will not cause any unwanted side effect, the result of the measurement is used to compute the new diagnoses which confirm the observation and the measurement result. Subsequent measurements are taken in order to settle down to only one possible diagnosis.

Although the computation of new diagnoses based on measurement results has been formalized in the work of Reiter [39] and Hou [29], the question as how to select the best next measurement is not addressed in their papers. A good measurement should bring as more information about the system being diagnosed as possible while keep the cost of updating the old diagnoses as lower as possible. However, as fault patterns are not fixed, there is no simple and definite rule which can tell us how to choose a better measurement in order to lower down the total cost required in such updating process. To tackle the problem, de Kleer has proposed a one-step lookahead method for computing the most promising measurement [15]. However, his method will be rendered impractical as the number of possible measurements increases.

In this chapter we propose a *m*easurement *s*election *s*trategy using a steady-state with least-fit-deletion *g*enetic *a*lgorithm (MSSGA). Genetic algorithms (GAs) have been developed and studied first by Holland [28], and have been applied to many fields with success [21, 3]. The reasons that we exploit a GA as the measurement selection strategy are [21]:

1. GAs operate on a set of strings, which is a natural representation for our problem here.

2. GAs do not require the fitness function to be well defined, or even explicitly stated.

3. Since GAs operate on a set of strings instead of only one, and use stochastic operators instead of deterministic ones, they can be very robust and efficient.

4. GAs implicitly find similarities among high-performance strings.

5. The genetic operators in GAs are very simple and straightforward.

In the following sections, we first briefly review the essence of GAs in Section 3.2. Then in Section 3.3 we describe MSSGA in detail. Section 3.4 presents the results of our experiments using the proposed strategy to diagnose five testing circuits. Specifically, a population of measurement request lists (MRLs) is evolved for each testing circuit using MSSGA, and their average performance over the evolution process is recorded. Finally Section 3.5 concludes this chapter with a summary of our contributions and comparisons to de Kleer's one-step lookahead measurement selection strategy.

3.2 Genetic Algorithms

Genetic Algorithms (GAs) were invented based on the inspirations from *natural selection* and *evolution*. The main theme behind GAs is that by mimicking the natural process of evolution, we may evolve better solutions to complex and difficult problems. GAs have been first proposed and studied by Holland [28]. He first worked on simple GAs that manipulate a population of binary strings, or *chromosomes*. Each chromosome represents an encoded solution to the problem to be solved. During the evolution process, first each chromosome is evaluated and the fitness value associated with the chromosome is computed. A fitness value is a positive number which reflects the quality of the corresponding chromosome, i.e., how good this particular solution is. The higher a fitness value, the better the corresponding chromosome (encoded solution). Two chromosomes are selected from the population according to the fitness distribution. The chromosome with higher fitness value is selected more frequently than the one with lower fitness value. Then two descendant chromosomes are generated by *crossing over* the selected chromosomes. The new chromosomes may also be *mutated* occasionally. This process is going on until a whole new population is generated. The old population is then replaced by the new one. The following procedure depicts the outline of a simple GA.

Procedure 3.2.1 A simple genetic algorithm has the following steps:

(1) Initialize a population P.
(2) Evaluate each chromosome within P.
(3) Test if one of the stopping criteria (time, fitness, etc.) holds. If yes, stop the procedure. For example, one would like to stop the procedure when the average fitness value goes up above some threshold.
(4) Select two chromosomes using the roulette wheel selection method.
(5) Generate two descendant chromosomes by crossing over the two chromosomes selected in (4).
(6) Mutate the descendant chromosomes occasionally.
(7) Evaluate the descendant chromosomes.
(8) Put the descendant chromosomes into a new population P'.
(9) Repeat (4) to (8) until P' has the same number of chromosomes as P does. Replace P with P'.
(10) Repeat (3) to (9)[9].

The roulette wheel selection used in the simple GA is essentially a weighted random selection method. Let the sum of the fitness values of all chromosomes within the population be

[9] A complete execution from (3) to (9) is called a 'generation.' In fact, Procedure 3.2.1 is a *generational* GA in that it does not update the original population until a whole new population is generated.

F. Then the probability that an MRL with fitness value f is selected is $\frac{f}{F}$.

For two chromosomes of length l, the one-point crossover proceeds as follows. First an integer k is randomly selected such that $1 \le k < l$. Two new descendant chromosomes are produced by swapping all bits between position $k+1$ and position l of two old chromosomes inclusively. For each of the descendant chromosomes, if mutation occurs, a position is randomly chosen and the bit at the position is altered. The crossover and the mutation are the two most frequently used *genetic operators*.

A significant difference between the simple GA and a traditional search method is that a population of candidate solutions is maintained and evolved in the simple GA. Different chromosomes exchange potentially good features, or *building blocks*, by generating new chromosomes from crossing over the old ones. Moreover, by mutating a chromosome, each solution in the search space is guaranteed not to be totally ignored.

The notion of a *schema* is introduced to further analyze the behavior of the simple GA. A schema is a pattern shared by a set of chromosomes. In the case of binary strings, a schema can be expressed as a pattern defined over the alphabet {0, 1, *}, where * represents 0 or 1. For example, two chromosomes 00100101 and 00101001 share a schema 0010**01. By manipulating a population of n chromosomes, it was shown by Holland that $O(n^3)$ schemata are processed implicitly by the simple GA. This characteristic, *implicit parallelism*, is the fundamental of the workings of the simple GA.

For a schema H, the order of H, $o(H)$, is defined as the number of fixed positions within H. The *defining length* of H, $\delta(H)$, is the distance between the first and the last specific string position. For example, let H_1 be **01**1*, and H_2 be 0*****1*, then $o(H_1) = 3$, $o(H_2) = 2$, $\delta(H_1) = 7 - 3 = 4$ and $\delta(H_2) = 7 - 1 = 6$. In one-point crossover, the schema with longer defining length will be more easily destroyed. This effect, together with the effect of selection and mutation, is described in the following equation:

$$m(H,t+1) \ge m(H,t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - p_m^{\,o(H)} \right],$$

where $m(H,t)$ denotes the number of chromosomes sharing schema H at time t, $f(H)$ is the fitness value of H, $\bar{f}$ is the average fitness value of the population, l is the length of a chromosome, and p_c and p_m are crossover and mutation rate, respectively. In fact, the simple GA can be viewed as an approximately optimal procedure for solving k-armed bandit problem [2, 28].

3.3 A Measurement Selection Strategy via the Genetic Algorithm (MSSGA)

In this section we present a genetic algorithm, MSSGA, for evolving good measurement request lists (MRLs). An MRL is an ordered list containing all possible measurement requests (MRs) for a specific circuit. In a diagnostic process, each MR in an MRL is taken sequentially in the order specified in the MRL, and the corresponding measurement is taken from the system being diagnosed. The result of the MR is then used to discriminate among competing diagnoses. This process is going on until only one possible diagnosis is left. A good MRL should shorten the diagnostic process while lower down the cost for obtaining the only one possible diagnosis. In particular, two important quantities are used as indications of the quality of an MRL: the first

one is the number of measurements needed for a diagnostic system to settle down to the only one possible diagnosis, and the second one is the total number of prover calls needed during the course of reasoning. These two numbers, n_M and n_P, respectively, are then combined into one quantity *PI* (performance index) in the following simple form:

$$PI = n_M \times n_P$$

MRL_i is said to be better than MRL_j if the *PI* value of MRL_i is smaller than that of MRL_j. Our goal here is to minimize the average *PI* values of the population of MRLs using MSSGA.

Although classic GAs use binary strings as the representation of candidate solutions, in MSSGA a chromosome is represented simply by an MRL. Essentially, the goal of MSSGA is to optimize the ordering of MRs. Bit-string representation is not natural for such order-based GAs since illegal strings may be produced after applying genetic operators [9]. Using MRLs as chromosomes facilitates the operations of the genetic operators used in MSSGA.

The outline of MSSGA is shown in the following procedure.

Procedure 3.3.1 The *m*easurement *s*election *s*trategy using a steady-state with least-fit-deletion *GA* (MSSGA) consists of the following steps:

(1) Initialization: Randomly initialize a population of MRLs.

(2) Initial evaluation:
Compute the *PI* value of each MRL within the population by performing diagnosis using the MRL. Then calculate the fitness value of the MRL based on its *PI* value.

(3) Test if one of the stopping criteria (time, fitness, etc.) holds. If yes, stop the procedure. For example, one would like to stop the procedure when the average *PI* value goes down under some threshold, or the unique diagnosis is found.

(4) Genetic operator selection:
Choose a genetic operator using the roulette wheel selection method based on the operator fitness value distribution. Assume that the operator selected requires *n* MRLs as operands.

(5) MRL selection:
Select *n* MRLs from the population using the roulette wheel selection method.

(6) Using the genetic operator:
Generate descendant chromosomes by using the genetic operator selected in (4) on the MRLs selected in (5).

(7) Evaluation:
Select one of the new MRLs generated in (6). Compute the *PI* value of the MRL by performing diagnosis using the MRL. Then calculate the fitness value of the MRL based on its *PI* value.

(8) Updating the population:
If the new MRL evaluated in (7) already exists in the population, replace the MRL in the population with the newly generated one. Otherwise replace the worst MRL, i.e. the MRL with largest *PI* value in the population, with the newly generated one.

(9) Updating the fitness values of the genetic operators:
Update the fitness values of all related genetic operators according to the performance of the new MRL.

(10) Repeat (7) to (9) until all of the descendant MRLs generated in (6) are evaluated and inserted into the population.

(11) Repeat (3) to (10).

The following sub-sections give details on MRL evaluation, the genetic operators of MSSGA, and dynamic updating of the fitness values of the genetic operators.

3.3.1 MRL Evaluation

An MRL is evaluated by performing diagnosis using it as the guideline on selecting the next measurement. The *PI* value is collected from the diagnostic process, and is used to compute the raw fitness value f_r of the MRL using the following formula

$$f_r = \frac{PI_{max}}{PI} \cdot f_{r,min}$$

where PI_{max} is the *PI* value of the worst MRL and $f_{r,\,min}$ is a constant. This conversion essentially inverts the direction of optimization from minimizing to maximizing, and at the same time preserves the reciprocal ratio between the *PI* values of different MRLs. For example, for MRL_i and MRL_j, if $PI_i = 4 \cdot PI_j$, where PI_i and PI_j are their respective *PI* values, then $f_{r,i} = 0.25 \cdot f_{r,j}$.

After the raw fitness value is determined for an MRL, the fitness value *f* of the MRL is computed by scaling the raw fitness value using a linear equation $f = a \cdot f_r + b$. The two coefficients *a* and *b* are obtained by solving the equations [21]:

$$\bar{f}_r = a \cdot \bar{f}_r + b$$
$$c \cdot \bar{f}_r = a \cdot f_{r,max} + b$$

where *c* is a constant typically in the range from 1.2 to 2, $f_{r,\,max}$ is the raw fitness value of the best MRL within the population, and $\bar{f}_r$ is the average raw fitness of the population. Intuitively, computing *a* and *b* this way insures that the average MRLs contribute one expected offspring, while the most successful MRL contributes *c* expected offspring in the selection process. In the beginning of the evolution process, there might be significant diversity in the population, and the fitness scaling insures that the superior MRLs will not take over the whole population too early. On the other hand, when the population converges in late stages of the process, the fitness scaling guarantees that the superior MRLs still have better chance to be selected, so that the evolution process will not become a random walk among those slightly inferior MRLs.

However, there are occasions when the whole population converges to some average raw fitness value and there still exists some bad MRLs whose raw fitness values are far below the average. In the cases the fitness scaling will yield some negative fitness value, which is unacceptable for the workings of MSSGA. To solve the difficulty, we recalculate *a* and *b* by solving the following equations:

$$\bar{f}_r = a \cdot \bar{f}_r + b$$
$$f_{min} = a \cdot f_{r,min} + b$$

where f_{min} is a given constant representing the reserved minimal fitness value, and $f_{r,min}$ is the raw fitness value of the worst MRL.

Another difficulty of linear scaling arises when the whole population converges to the same raw fitness value. In this case, MSSGA gives up attempting to linearly scale the raw fitness values by assigning $a = 1$ and $b = 0$.

3.3.2 The Genetic Operators of MSSGA

The four genetic operators used in MSSGA are listed below.

1. Crossover:

   Let MRL_i and MRL_j be the two MRLs. Compute the average number of prover calls of MRL_i and that of MRL_j, respectively. For each MR in MRL_i, if its *proverCall* is above the average, collect it into an ordered set Top_i, in which MR_x is in front of MR_y iff MR_x is in front of MR_y in MRL_i. Similarly, collect all MRs in MRL_j with above-average *proverCall* into an ordered set Top_j. Then the two descendant MRLs MRL_i' and MRL_j' are

   $MRL_i' = Top_i \cup (Top_j - Top_i) \cup ORDER_j(MRL_i - (Top_i \cup (Top_j - Top_i)))$, and
   $MRL_j' = Top_j \cup (Top_i - Top_j) \cup ORDER_i(MRL_j - (Top_j \cup (Top_i - Top_j)))$,

   where for two ordered sets A and B, $A \cup B$ returns an ordered set appending B to the end of A, $A - B$ returns an ordered set containing the elements only belonging to A and in the order specified in A, and $ORDER_x$ reorders the set it receives to the order specified in MRL_x.

   The rationale behind this operator is that if an MR has a higher *proverCall*, it might point out a promising measurement which will bring more information to help discriminate among the possible diagnoses, therefore it is worth being tried earlier. A measurement which confirms each of the n possible diagnoses simply requires n prover calls to validate the confirmation, and at most of the time, n is quite small compared to the number of prover calls required for MCS derivation when one of the possible diagnoses does not agree with the measurement.

   Although the crossover operator moves the MRs in an MRL with above-average *proverCall*s to the front of the MRL, the relative order of these promising MRs is preserved. The rest of the MRs are rearranged according to their relative order specified in the other MRL. In this way, good orderings in different MRLs can be exchanged for better performance.

2. Mutation:

   This operator acts on a single MRL. Let MRL_i be an MRL. Randomly choose two MRs within MRL_i and invert the ordering of all MRs in between (inclusive) the two selected MRs. The result is the descendant MRL.

   In fact, this operator is the inversion operator first introduced by Holland [28].

3. Fine-tune:

   This operator acts on a single MRL. Let MRL_i be an MRL. Compute the average number of prover calls of MRL_i and collect all above-average MRs of MRL_i into an ordered set Top_i as depicted in the description of the crossover operator. Randomly reorder the elements of Top_i, and denote the result as Top_i'. Then the descendant MRL is

   $MRL_i' = Top_i' \cup \{MRL_i - Top_i'\}$.

   The intuition of this operator is to move the promising MRs, i.e., the MRs with above-average *proverCall*s, to the front of the MRL, and rearrange the relative order of them randomly in the hope that the new ordering of these promising MRs will improve the *PI* value of the MRL.

4. Reevaluation:

   This operator simply returns the MRL it receives. The reason to reevaluate an MRL is to alleviate DSDC's dependence on the 'old' fitness value associated with an MRL. For real diagnostic tasks, even with the same circuit the fault patterns will not be the same all the time. If an MRL is associated with a high fitness value under some fault pattern, it is possible that the same MRL will receive a rather low fitness value under some other fault pattern. Without the reevaluation operator, this inaccurate association will never be updated, hence the association cannot reflect the real fault tendency adaptively anymore.

3.3.3 Adapting the Fitness Values of the Genetic Operators

It has been shown that the control parameters such as the size of the population, mutation and crossover rates, and the number of crossover points affect a GA's performance [48]. Although some of them can be preset according to the results of empirical studies [13, 22], the various operator rates need to be adapted constantly in the course of the evolution process in order to reflect the dynamics in the population and in the environment. In MSSGA, we use credit propagation for updating the fitness values of all four operators based on Davis' work [8]. In short, each of the operators receives a reward if it yields a superior descendant than the MRL replaced. This reward is then distributed exponentially to the operator producing the parent of the MRL, the operator producing the parent of the parent of the MRL, etc. The exact number of generations considered backward, *parent length*, and the constant for multiplying the original reward, *credit constant*, are the parameters of this method. Note that the updating method used here is slightly different from Davis' approach in that we give reward to a particular operator based on the difference between the fitness value of the new MRL and that of the MRL replaced, instead of assigning reward to an MRL based on the difference between the fitness value of the new MRL and that of the best MRL in the population.

3.4 Experiment Results

To test MSSGA, we incorporate it into our diagnostic architecture: a *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC). Two series of experiments were done, and in each experiment we first randomly generated an initial population of MRLs, and then used DSDC to diagnose five testing circuits under simulated fault patterns. Note that all fault patterns used here were randomly generated according to a given error rate distribution of the gates.

In the first experiment, which we call 'fixed fault tests,' a fault pattern was randomly generated before creating the initial population. Based on the fault pattern an initial population

of MRLs was randomly created and evaluated. This population was then evolved by using MSSGA. Note that during the course of the evolution process we fixed the simulated fault pattern of the circuit so that DSDC repeatedly diagnosed the same circuit under the same fault pattern using different MRLs.

In the second experiment, which we call 'noisy tests,' an initial population of MRLs was randomly generated, and each MRL within the population was evaluated by using it in a diagnostic process under a randomly generated fault pattern. The population was then evolved by using MSSGA. Note that in this experiment we randomly generated a fault pattern for each MRL evaluated during the course of the evolution process in order to understand the performance of MSSGA in a noisy environment.

The following tables give the values of the parameters of MSSGA used for each testing circuit in the two experiments. For the parameters which are not listed in the tables: the parent length is 5, the minimal reserved raw fitness value and the minimal reserved fitness value are both 10.0, the constant *c* used for linearly scaling raw fitness values is 1.5, and the number of faulty components in all simulated fault patterns used in the experiments is 2. Appendix B presents the details of each testing circuit.

Testing Circuit Name	**Population Size**	**Number of Evaluations**	**Initial Operator Fitness**				**Credit Constant**
			Crossover	**Mutation**	**Fine-tune**	**Reevaluation**	
c01_8	20	100	0.6	0.1	0.3	0[10]	0.001
c02_18	40	200	0.6	0.1	0.3	0	0.00001
c03_19	40	400	0.6	0.1	0.3	0	0.00001
c04_16	40	200	0.6	0.1	0.3	0	0.00001
c05_16	40	400	0.6	0.1	0.3	0	0.00001

Table 3.4.1. The values of the parameters for MSSGA used on each testing circuit in fixed fault tests

Testing Circuit Name	**Population Size**	**Number of Evaluations**	**Initial Operator Fitness**				**Credit Constant**
			Crossover	**Mutation**	**Fine-tune**	**Reevaluation**	
c01_8	20	100	0.55	0.05	0.25	0.15	0.001
c02_18	40	200	0.55	0.05	0.25	0.15	0.00001
c03_19	40	400	0.55	0.05	0.25	0.15	0.00001
c04_16	40	200	0.55	0.05	0.25	0.15	0.00001
c05_16	40	400	0.55	0.05	0.25	0.15	0.00001

Table 3.4.2. The values of the parameters for MSSGA used on each testing circuit in noisy tests

Note that the number of evaluations for each testing circuit is merely a small fraction of the number of the possible measurement orderings of the circuit. The following table shows the number of the possible MRLs for each testing circuit and the percentage of the search space explored.

[10] Since the simulated fault pattern is fixed during the course of the evolution process, it is not necessary to activate the reevaluation operator in fixed fault tests.

Testing Circuit Name	Number of Possible MRs	Number of Possible MRLs	% of the Search Space Explored
c01_8	7	5,040	1.984
c02_18	8	40,320	0.496
c03_19	14	87,178,291,200	4.588×10^{-7}
c04_16	8	40,320	0.496
c05_16	14	87,178,291,200	4.588×10^{-7}

Table 3.4.3. The percentage of the search space explored on each testing circuit

The following two sub-sections present the results of the experiments. For each testing circuit diagnosed, we graph four figures based on the data averaged from the 10 runs: *PI* values vs. number of evaluations, number of measurements vs. number of evaluations, scaling coefficients *a* and *b* vs. number of evaluations, and operator fitness values vs. number of evaluations.

3.4.1 Fixed Fault Tests

In this series of experiments, we used DSDC to evolve an initial population of MRLs 10 times for each testing circuit, with each time using a different initial population generated and evaluated under a different fault pattern. The data collected is averaged and presented in the following sub-sections.

In summary, the *PI* values, which are what MSSGA tries to minimize, are all reduced considerably for all testing cases. The numbers of measurements required by DSDC to settle down to only one diagnosis are also greatly reduced, although they are not necessarily so[11]. The following tables show the improvement achieved by MSSGA in diagnosing each testing circuit.

Testing Circuit Name	Initial Avg *PI*	Avg *PI* after Evolution	Improvement Rate (%)[12]
c01_8	668.25	256.72	61.585
c02_18	54219.903	32158.11	40.689
c03_19	410447	153526.9	62.595
c04_16	38748.25	19768.05	48.983
c05_16	71458.83	15050.2	78.939
Average			**58.558**

Table 3.4.4. Fixed fault tests: improvement in the average *PI* values

Testing Circuit Name	Initial Avg n_M	Avg n_M after Evolution	Improvement Rate (%)
c01_8	5.81	2.975	48.795
c02_18	5.885	3.585	39.082
c03_19	9.3125	3.025	67.517
c04_16	5.0625	3.12	38.370
c05_16	9.675	3.54	63.411
Average			**51.435**

Table 3.4.5. Fixed fault tests: improvement in the number of measurements n_M required by DSDC to settle down to one diagnosis

As to the linear scaling coefficients *a* and *b*: in these tests *a* always started from some small value under 1, and gradually moved toward 1. On the other hand, *b* started from some large value above 1, and slid toward 0 (with some undulations). These changes actually show that the population converged gradually to better solutions. However, since the results show

[11] Actually for some testing circuit in noisy tests the numbers of measurements required to discriminate to only one diagnosis were *increased*, as we shall see in Section 3.4.2.

[12] Improvement rate = (initial-after)/initial

that the coefficient *a* rarely became 1, the populations rarely fully converged in these tests. Moreover, the situation (discussed in Section 3.3.1) in which scaling the raw fitness values with the first set of equations yields negative fitness values did not happen in these tests. This fact shows that MSSGA uniformly evolved the whole population without leaving some unfortunate individual behind.

The fitness values of the genetic operators in MSSGA did not significantly change in these tests. This is because we took a rather conservative policy in updating the operator fitness values by setting a small credit constant.

3.4.1.1 Testing Circuit c01_8

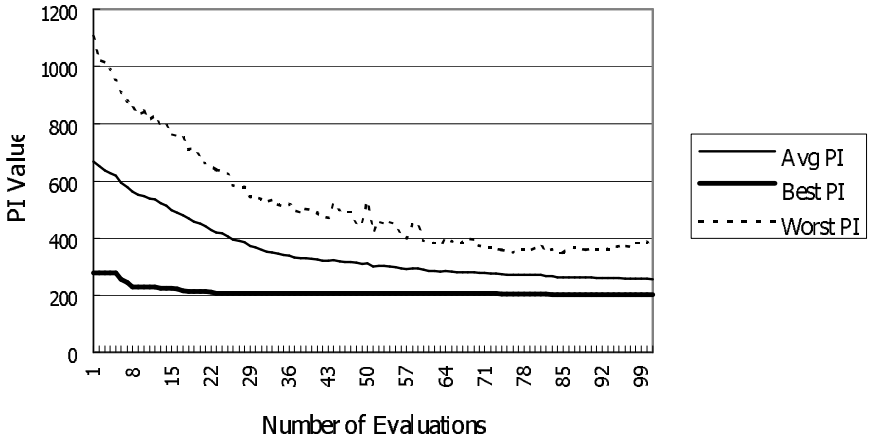

Fig. 3.4.1. Circuit c01_8: PI value vs. number of evaluations in fixed fault tests

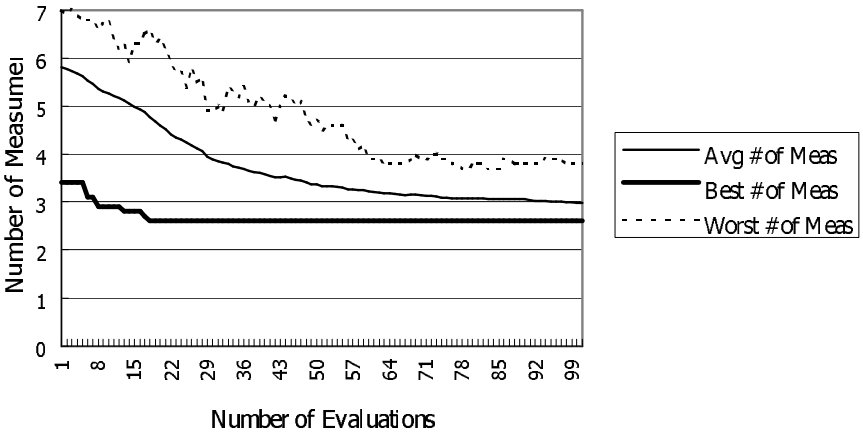

Fig. 3.4.2. Circuit c01_8: number of measurements vs. number of evaluations in fixed fault tests

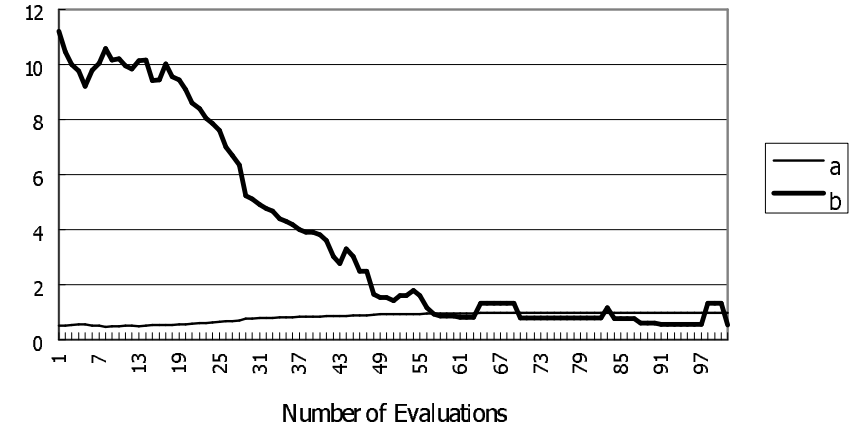

Fig. 3.4.3. Circuit c01_8: scaling coefficients a & b vs. number of evaluations in fixed fault tests

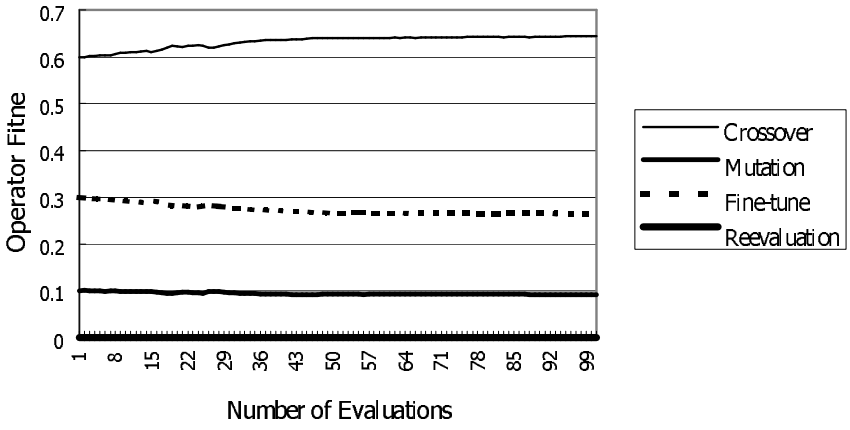

Fig. 3.4.4. Circuit c01_8: operator fitness vs. number of evaluations in fixed fault tests

3.4.1.2 Testing Circuit c02_18

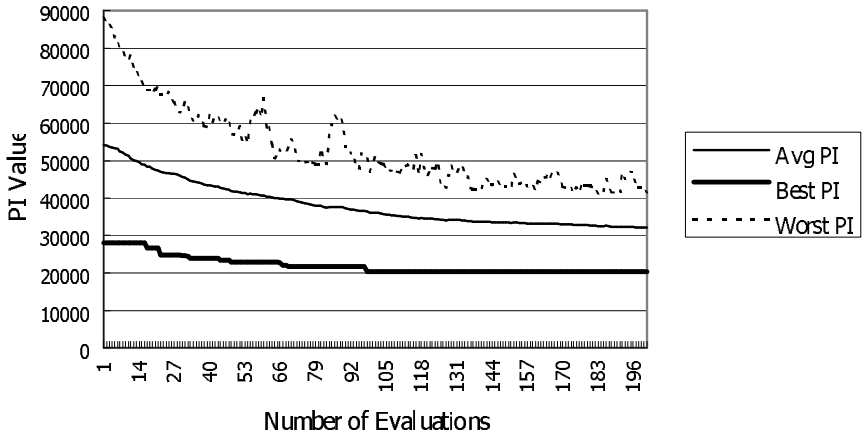

Fig. 3.4.5. Circuit c02_18: PI value vs. number of evaluations in fixed fault tests

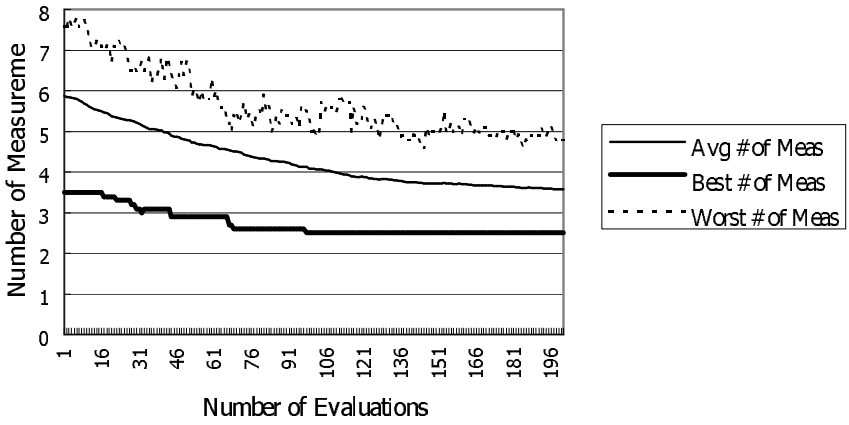

Fig. 3.4.6. Circuit c02_18: number of measurements vs. number of evaluations in fixed fault tests

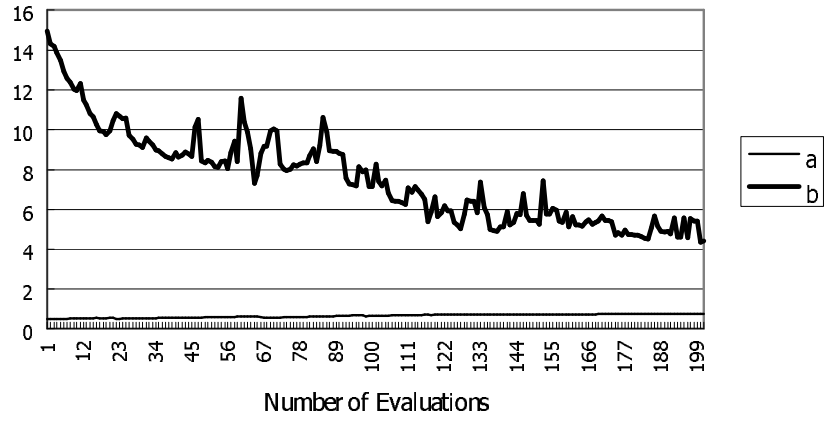

Fig. 3.4.7. Circuit c02_18: scaling coefficients a & b vs. number of evaluations in fixed fault tests

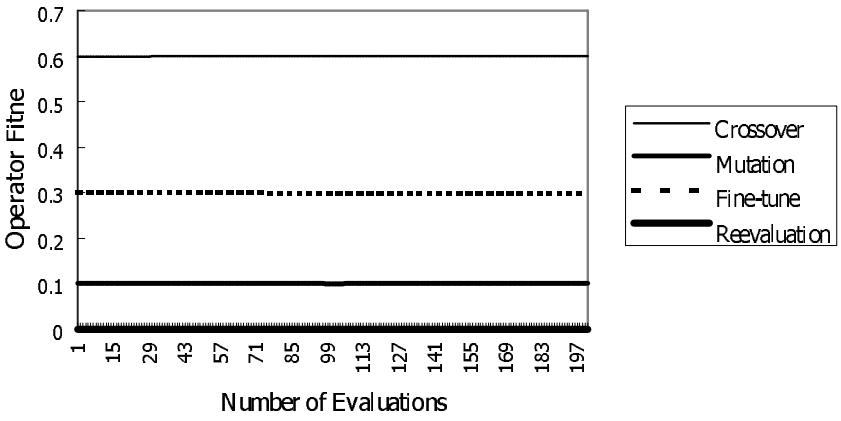

Fig. 3.4.8. Circuit c02_18: operator fitness vs. number of evaluations in fixed fault tests

3.4.1.3 Testing Circuit c03_19

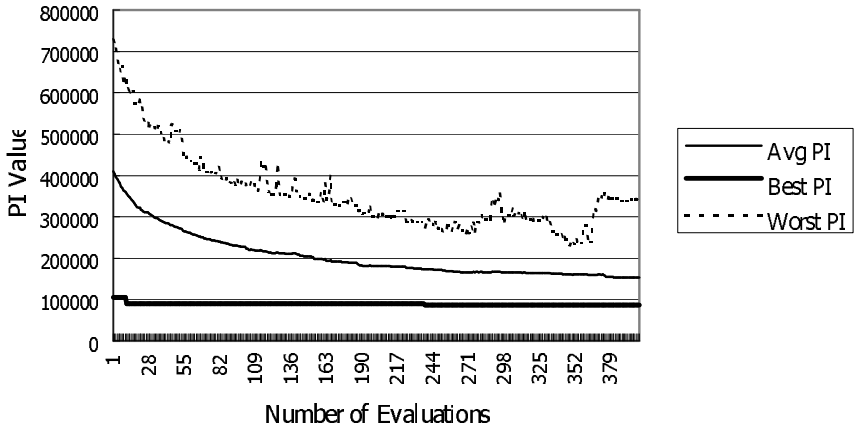

Fig. 3.4.9. Circuit c03_19: PI value vs. number of evaluations in fixed fault tests

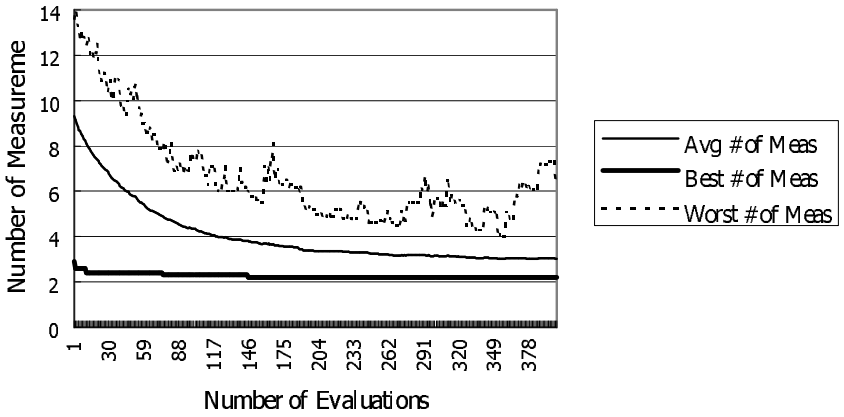

Fig. 3.4.10. Circuit c03_19: number of measurements vs. number of evaluations in fixed fault tests

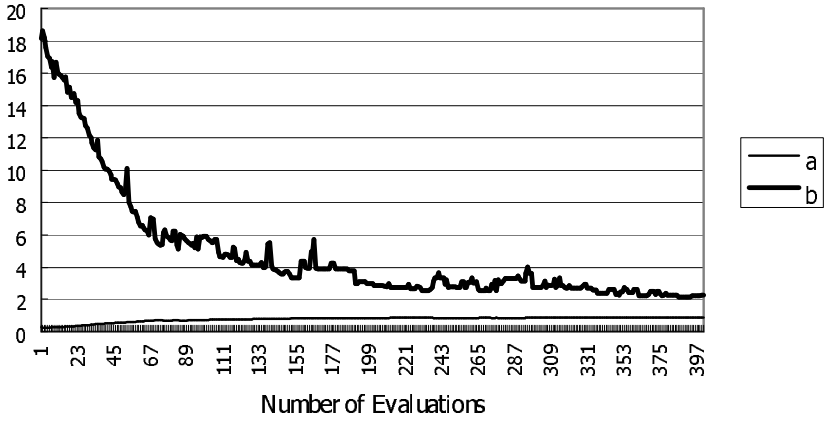

Fig. 3.4.11. Circuit c03_19: scaling coefficients a & b vs. number of evaluations in fixed fault tests

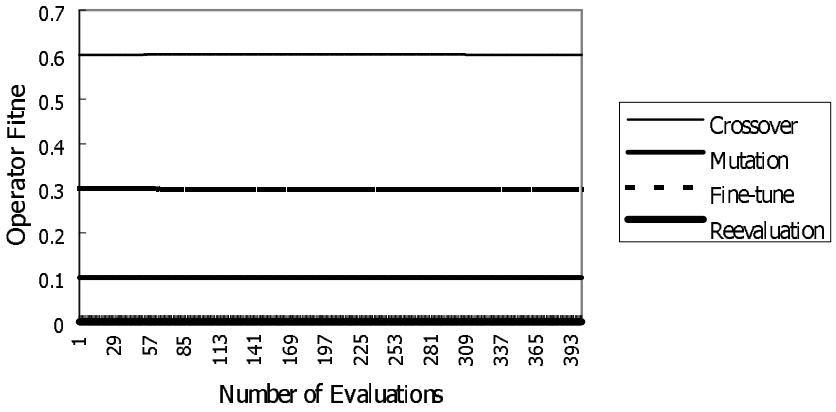

Fig. 3.4.12. Circuit c03_19: operator fitness vs. number of evaluations in fixed fault tests

3.4.1.4 Testing Circuit c04_16

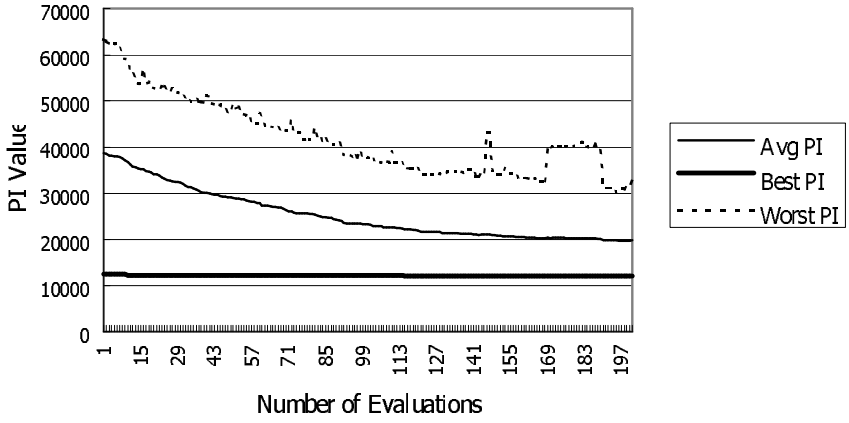

Fig. 3.4.13. Circuit c04_16: PI value vs. number of evaluations in fixed fault tests

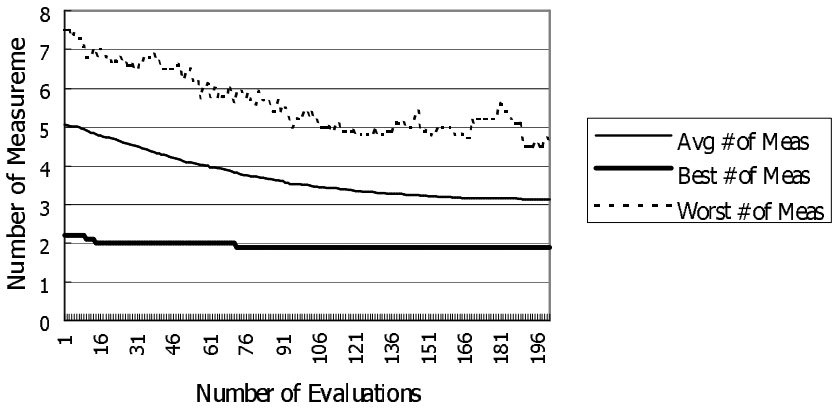

Fig. 3.4.14. Circuit c04_16: number of measurements vs. number of evaluations in fixed fault tests

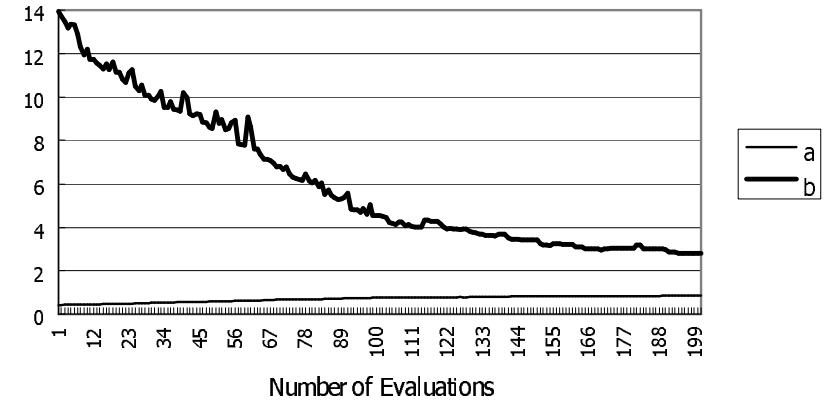

Fig. 3.4.15. Circuit c04_16: scaling coefficients a & b vs. number of evaluations in fixed fault tests

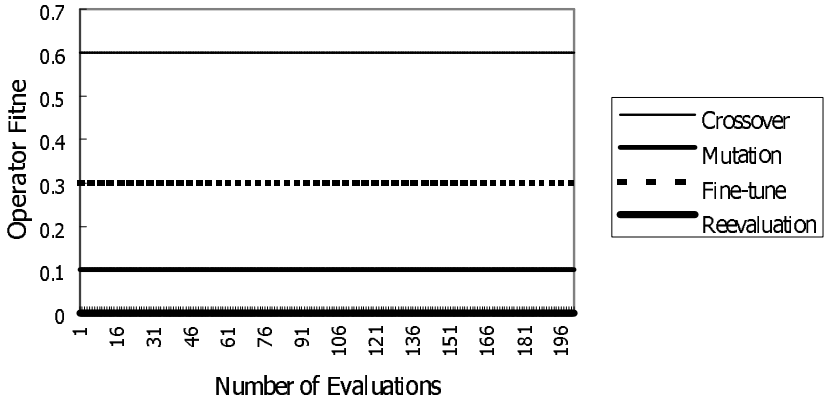

Fig. 3.4.16. Circuit c04_16: operator fitness vs. number of evaluations in fixed fault tests

3.4.1.5 Testing Circuit c05_16

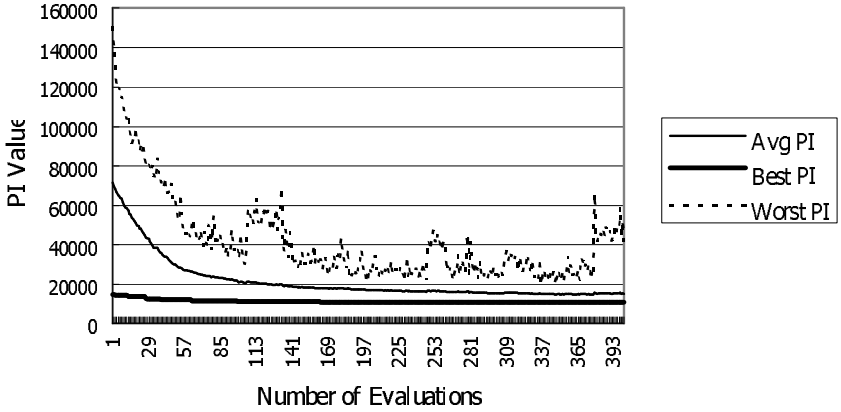

Fig. 3.4.17. Circuit c05_16: PI value vs. number of evaluations in fixed fault tests

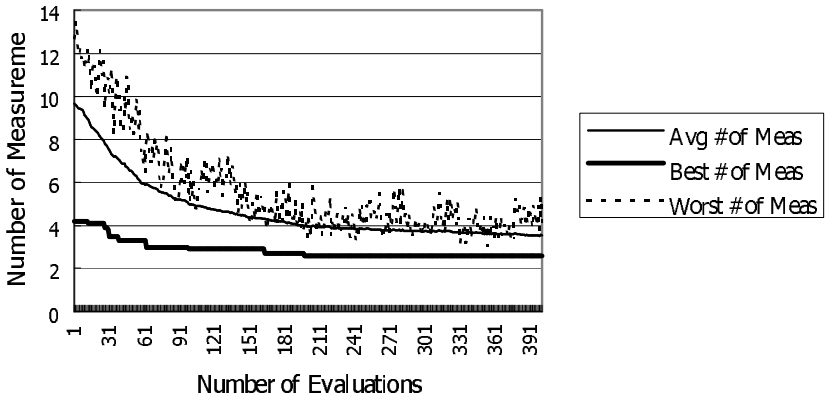

Fig. 3.4.18. Circuit c05_16: number of measurements vs. number of evaluations in fixed fault tests[13]

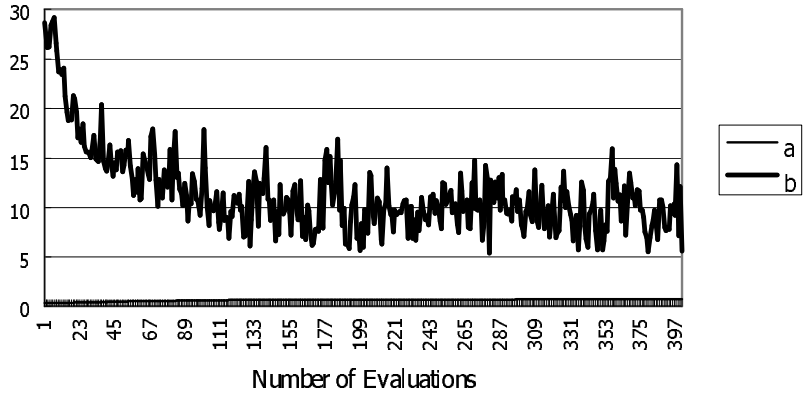

Fig. 3.4.19. Circuit c05_16: scaling coefficients a & b vs. number of evaluations in fixed fault tests

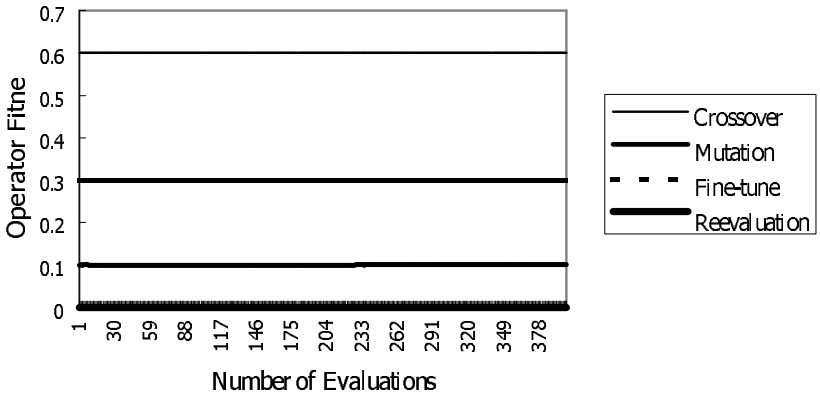

Fig. 3.4.20. Circuit c05_16: operator fitness vs. number of evaluations in fixed fault tests

[13] Note here that the worst number of measurements sometimes gets lower than the average number of measurements. It stems from the criterion by which we define what the 'worst' MRL is, that is, the MRL with the highest *PI* value, not with the highest number of measurements required.

3.4.2 Noisy Tests

In this series of experiments, we used DSDC to evolve an initial population of MRLs 10 times for each testing circuit. The difference between noisy tests and fixed fault tests is that in noisy tests, for each testing circuit the same initial population is used for the evolution process under random fault patterns. The data collected is averaged and presented in the following sub-sections.

In summary, the *PI* values are all reduced considerably for all testing cases. Surprisingly, in these experiments MSSGA performed much better in noisy tests than in fixed fault tests in optimizing *PI* values (compare Table 3.4.4 with Table 3.4.6). However, the average improvement rate of the average n_M is lower than that in fixed fault tests. In fact, as shown in Table 3.4.7, in one testing case the improvement rate is negative. This is not too surprising since MSSGA is designed to focus on minimizing *PI* value, and it is possible that an MRL with high n_M and low n_P defeats another MRL with low n_M but high n_P during the selection process.

Testing Circuit Name	**Initial Avg *PI***	**Avg *PI* after Evolution**	**Improvement Rate (%)**
c01_8	534.1	216.665	59.434
c02_18	145797.9	24529.58	83.176
c03_19	606380.9	51719.73	91.471
c04_16	46166.8	7602.59	83.532
c05_16	122290.9	12948.33	89.412
Average			**81.405**

Table 3.4.6. Noisy tests: improvement in the average *PI* values

Testing Circuit Name	**Initial Avg n_M**	**Avg n_M after Evolution**	**Improvement Rate (%)**
c01_8	5.8	4.37	24.655
c02_18	5.225	5.1075	2.249
c03_19	10	8.5875	14.125
c04_16	4.9	5.16	-5.306
c05_16	10.875	9.2275	15.149
Average			10.174

Table 3.4.7. Noisy tests: improvement in the number of measurements n_M required by DSDC to settle down to one diagnosis

Since in noisy tests MSSGA evolved a population in a noisy environment (i.e., the fault patterns were randomly generated), the values of all other variables depicted in the following figures exhibit more vibrations. In particular, the scaling coefficient b changed much more frequently than in fixed fault tests and did not always move toward 0. Another thing to notice is that it seems that the best MRL was frequently replaced by inferior MRLs. It is normal because an MRL which performed well under one particular fault pattern is not guaranteed to act the same under another different fault pattern. The reason why MSSGA can still decrease the *PI* values significantly is that it captures the fault tendency exhibited in each testing circuit.

3.4.2.1 Testing Circuit c01_8

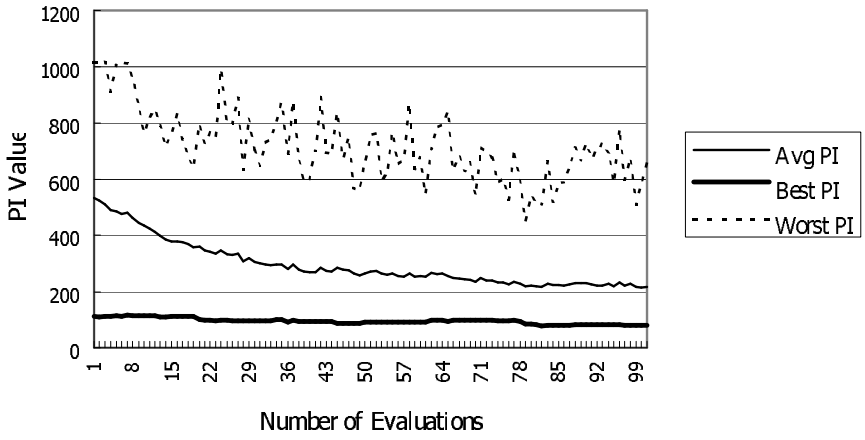

Fig. 3.4.21. Circuit c01_8: PI value vs. number of evaluations in noisy tests

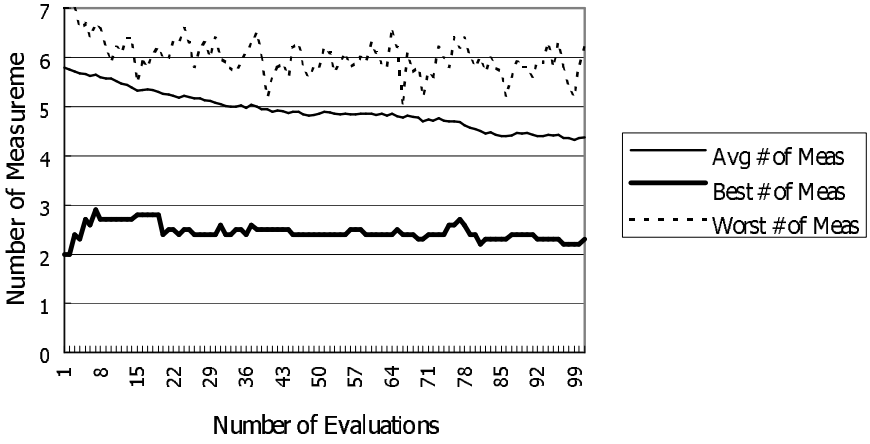

Fig. 3.4.22. Circuit c01_8: number of measurements vs. number of evaluations in noisy tests

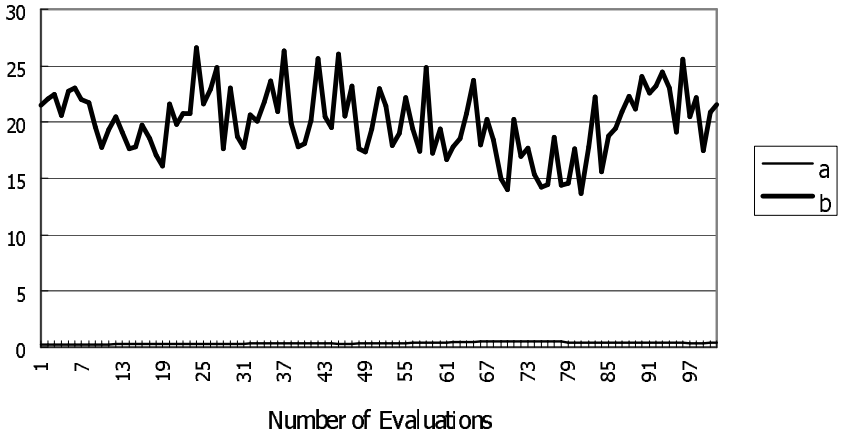

Fig. 3.4.23. Circuit c01_8: scaling coefficients a & b vs. number of evaluations in noisy tests

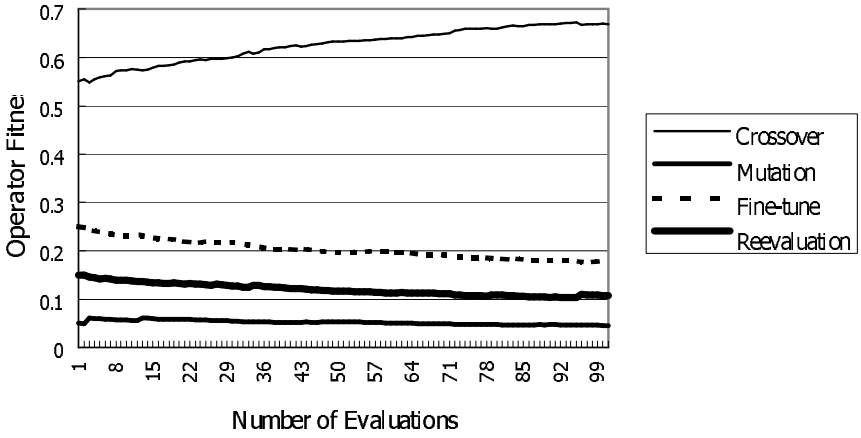

Fig. 3.4.24. Circuit c01_8: operator fitness vs. number of evaluations in noisy tests

3.4.2.2 Testing Circuit c02_18

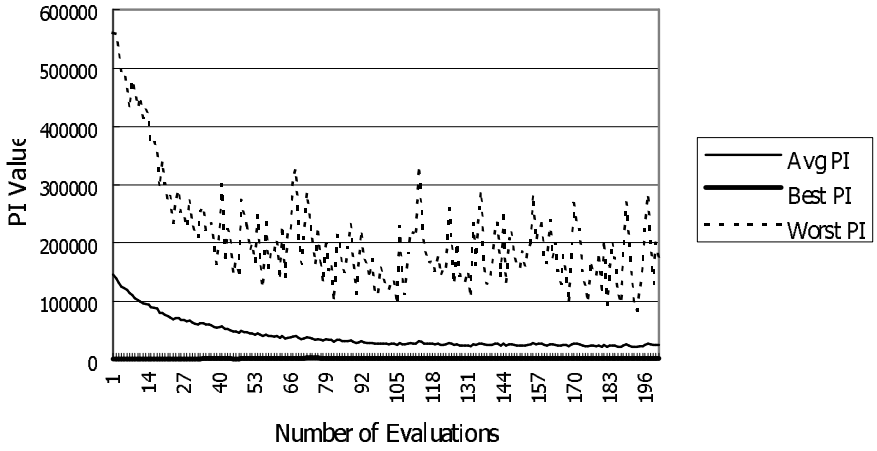

Fig. 3.4.25. Circuit c02_18: PI value vs. number of evaluations in noisy tests

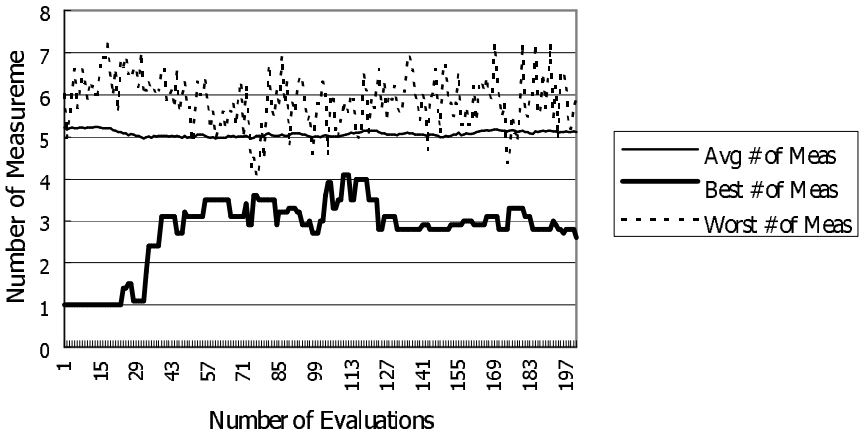

Fig. 3.4.26. Circuit c02_18: number of measurements vs. number of evaluations in noisy tests

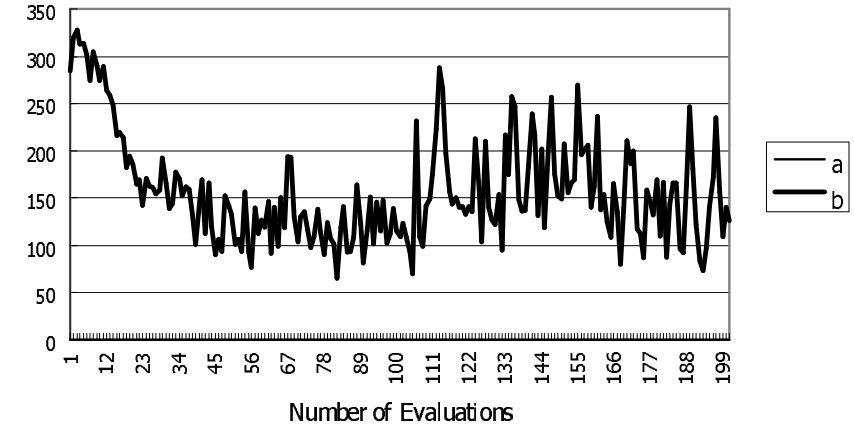

Fig. 3.4.27. Circuit c02_18: scaling coefficients a & b vs. number of evaluations in noisy tests

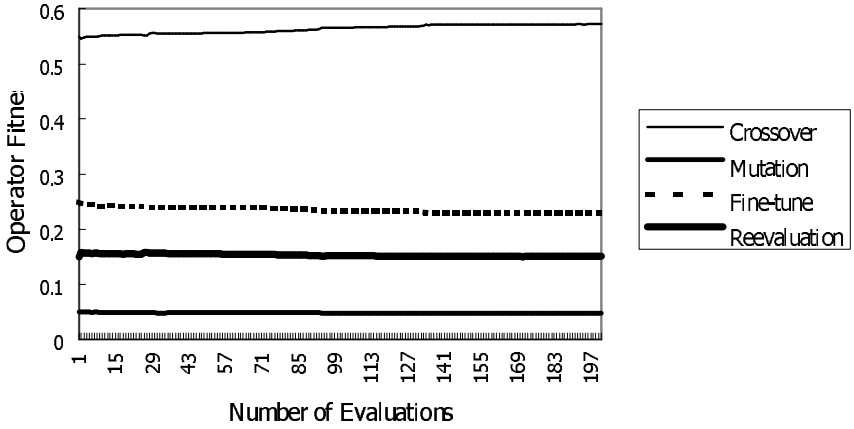

Fig. 3.4.28. Circuit c02_18: operator fitness vs. number of evaluations in noisy tests

3.4.2.3 Testing Circuit c03_19

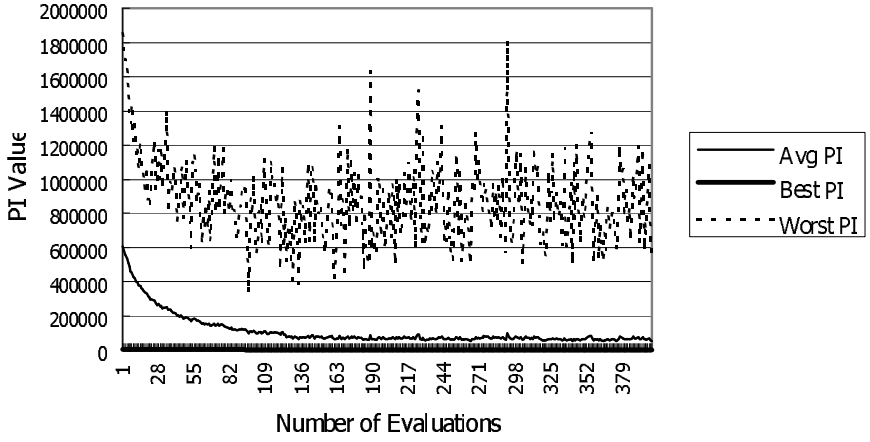

Fig. 3.4.29. Circuit c03_19: PI value vs. number of evaluations in noisy tests

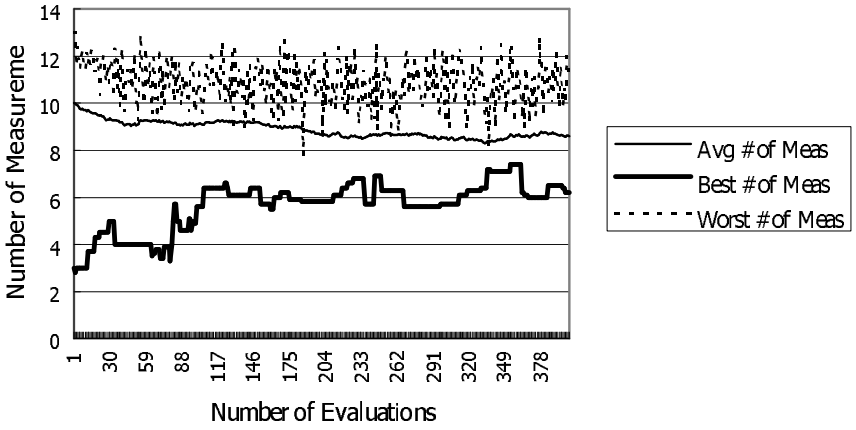

Fig. 3.4.30. Circuit c03_19: number of measurements vs. number of evaluations in noisy tests

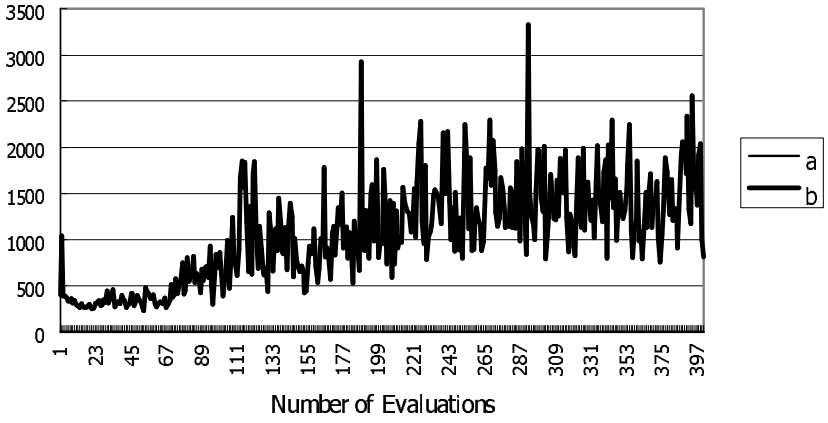

Fig. 3.4.31. Circuit c03_19: scaling coefficients a & b vs. number of evaluations in noisy tests

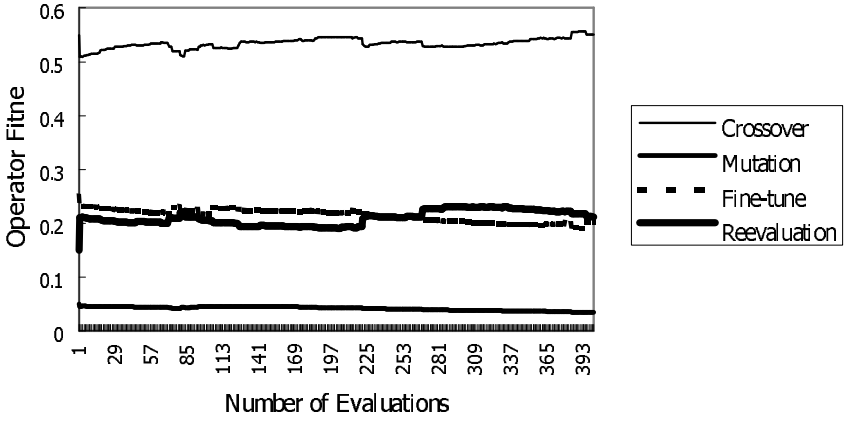

Fig. 3.4.32. Circuit c03_19: operator fitness vs. number of evaluations in noisy tests

3.4.2.4 Testing Circuit c04_16

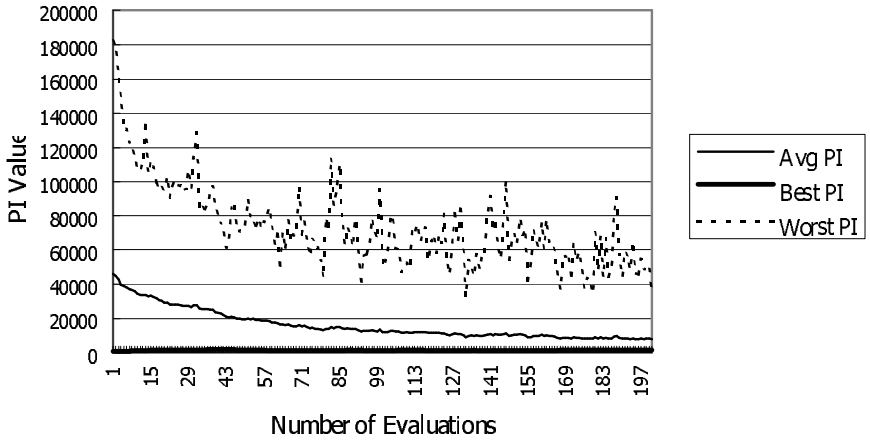

Fig. 3.4.33. Circuit c04_16: PI value vs. number of evaluations in noisy tests

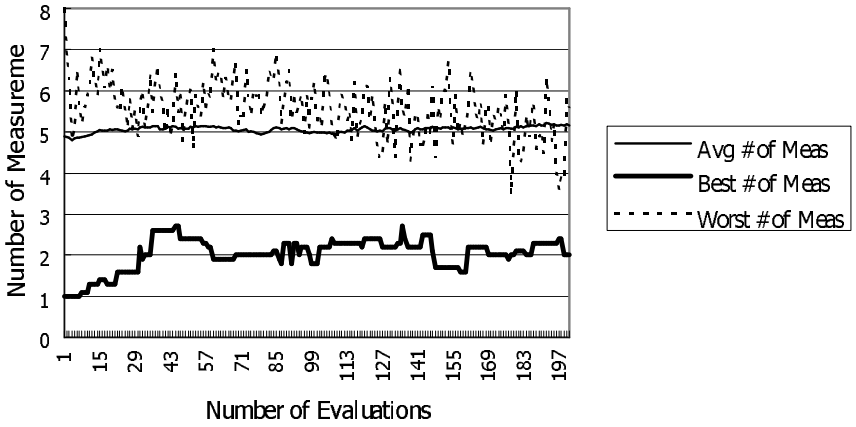

Fig. 3.4.34. Circuit c04_16: number of measurements vs. number of evaluations in noisy tests

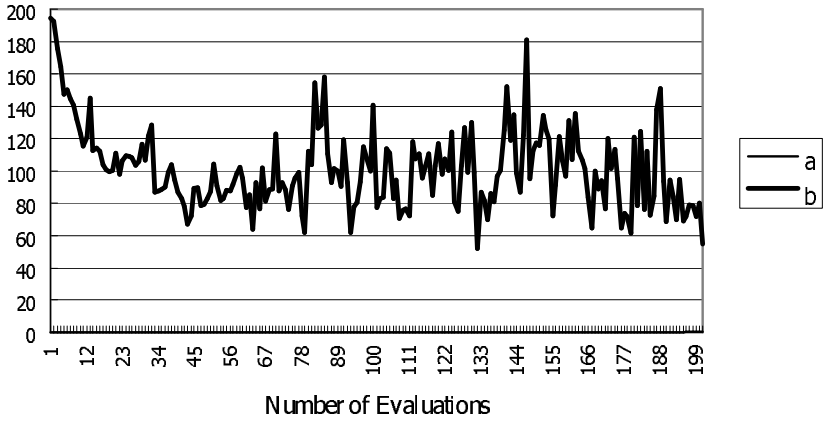

Fig. 3.4.35. Circuit c04_16: scaling coefficients a & b vs. number of evaluations in noisy tests

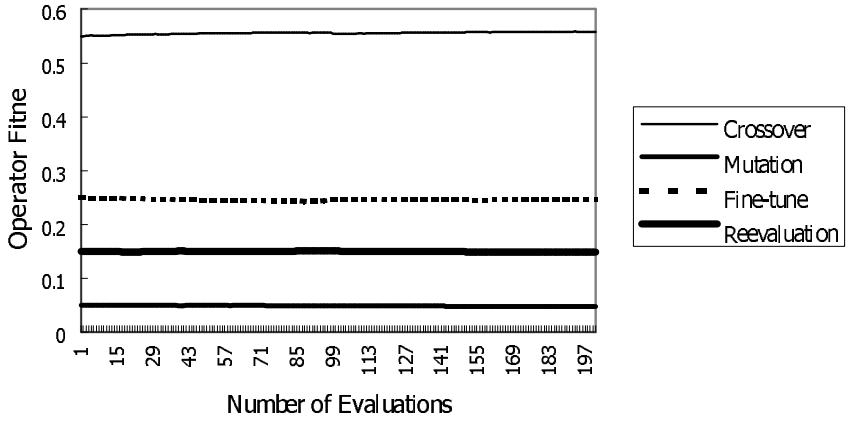

Fig. 3.4.36. Circuit c04_16: operator fitness vs. number of evaluations in noisy tests

3.4.2.5 Testing Circuit c05_16

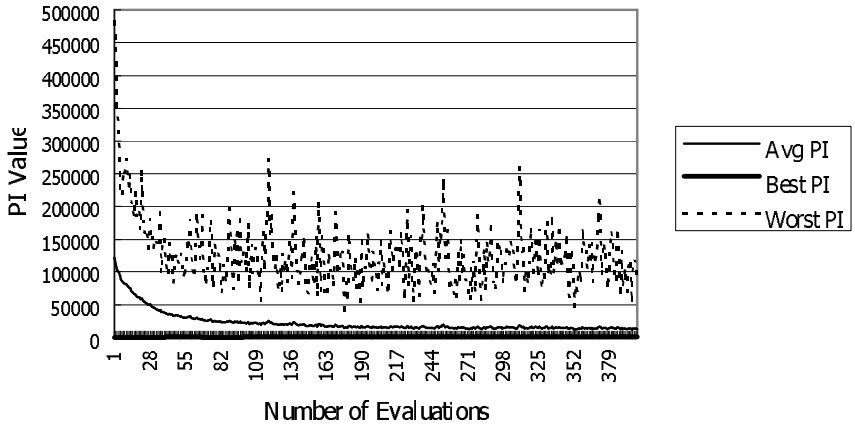

Fig. 3.4.37. Circuit c05_16: PI value vs. number of evaluations in noisy tests

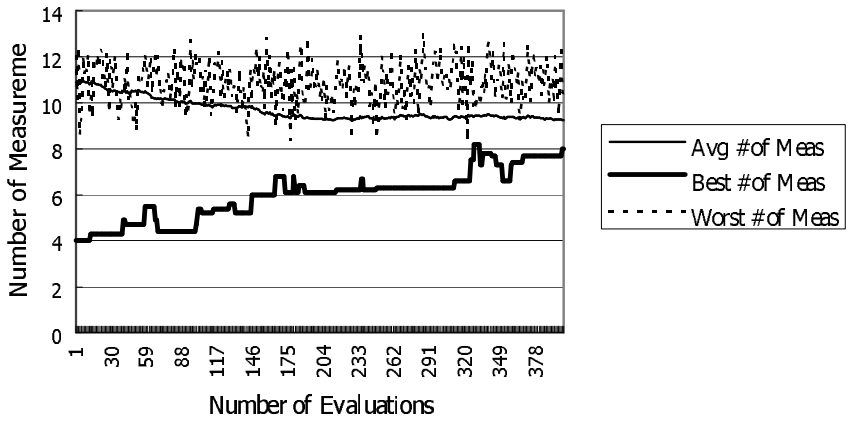

Fig. 3.4.38. Circuit c05_16: number of measurements vs. number of evaluations in noisy tests

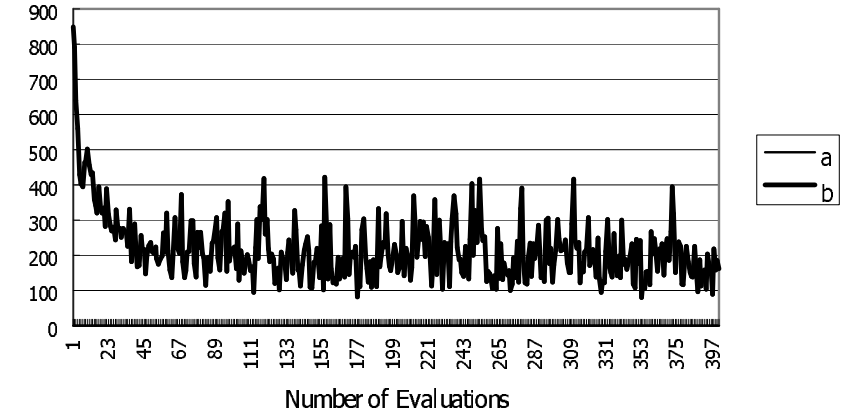

Fig. 3.4.39. Circuit c05_16: scaling coefficients a & b vs. number of evaluations in noisy tests

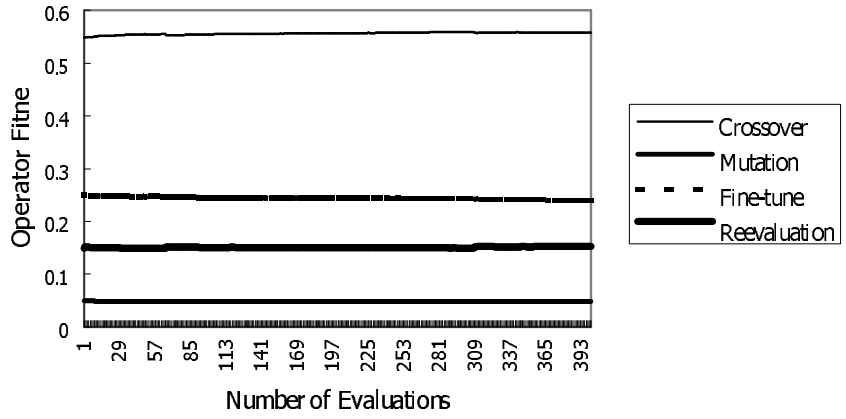

Fig. 3.4.40. Circuit c05_16: operator fitness vs. number of evaluations in noisy tests

3.5 Discussions

In this chapter we have proposed a new measurement selection strategy which can be incorporated into a diagnostic system based on the theory of diagnosis from first principles [39]. Specifically, we have proposed MSSGA, a *m*easurement *s*election *s*trategy using a steady-state with least-fit-deletion *g*enetic *a*lgorithm, and successfully integrated it into our *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC). Five testing circuits were diagnosed using DSDC, and the result shows that MSSGA is capable of improving the average performance of MRLs significantly both in noise-free and noisy environment.

To conclude this chapter, we compare de Kleer's measurement selection strategy in GDE (general diagnostic engine) [15] to MSSGA. In GDE, a one-step lookahead method is utilized to select the best next measurement. Essentially this method weighs all the possible measurements by computing the entropy of the system being diagnosed *after* each of the measurements is taken. The goal here is slightly different from that of MSSGA, in that it focuses on minimizing the number of measurements required for a diagnostic system to settle down to only one diagnosis (i.e., n_M in the definition of *PI* value in MSSGA). In other words, a good measurement ordering is supposed to shorten the diagnostic process, but is not required to lower down the total cost for computing the diagnoses. However, this approach requires a simulation of probing each of the possible measurement points with each of the possible

outcomes. Only after that the MCSs under each measurement with each possible outcome can be computed, and based on that data the entropy of the whole system being diagnosed under different measurements can be calculated. The measurement with which the entropy of the system is minimal is then selected as the next measurement. This process is going on for each measurement taken in the course of a diagnostic process. For a digital circuit with n possible measurements, the one-step lookahead method requires $2n$ MCS derivations based on the pseudo-measurements. If m measurements are required to settle down to only one possible diagnosis, the whole diagnostic process requires $m \cdot (2n-m+1)$ MCS derivations. For large n and m, the one-step lookahead method will be rendered impractical, for each MCS derivation is expensive. By contrast, MSSGA can efficiently work online, and the average *PI* value of the whole population of MRLs can be lowered down by *training* the diagnostic system incorporating MSSGA with random fault patterns (according to an error rate distribution collected from past experiences) *before* any online diagnostic process is engaged.

Chapter 4

A Diagnostic System for Digital Circuits

In this chapter we propose a complete *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC) based on the work of Reiter [39] and Hou [29]. Differences between DSDC and de Kleer's famous GDE [15] are also discussed.

4.1 Introduction

Although almost all theories of diagnosis intend to be domain-independent, digital circuits are the most frequent and suitable test-bed for these theories. One of the reasons is that the model of digital circuits can be easily built in a compact form, and their behaviors have been well studied and formalized. The other reason is that due to the advances of VLSI technology in recent years, the dramatically increasing complexity of digital circuits together with their short life cycles make any diagnostic approaches based only on intuitions impractical. It is then necessary and important to build automatic diagnostic systems for such diagnostic tasks.

In this chapter we propose a complete *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC) based on the work of Reiter [39] and Hou [29]. DSDC has the following characteristics:

1. DSDC is complete in the sense that all possible diagnoses of a malfunctioning circuit given the observations obtained so far can be found.

2. DSDC uses a simple language called DSDC Language (DSDCL) for describing digital circuits. Hierarchical diagnosis is possible by using circuit specifications at different granularity.

3. DSDC derives minimal conflict sets by constructing a pruned CS-tree with mark set as described in Procedure 2.4.6.

4. DSDC uses an incremental propositional calculus prover (PC prover) for logic inferences. To be more specific, a *PC p*rover implemented by *t*rie data structure (PCPT) with chronological-backtracking capability is incorporated in DSDC as the underlying inference engine.

5. DSDC selects the best next measurement by exploiting the measurement selection strategy via the genetic algorithm proposed in Chapter 3.

In Section 4.2 we present an overview of DSDC. A more detailed description of each part of DSDC is given in Section 4.3 to Section 4.6. Two example runs are presented in Section 4.7, and Section 4.8 gives discussions, comparisons between DSDC and GDE [15], and possible extensions to DSDC.

4.2 Overview of DSDC

DSDC is a complete diagnostic system for combinatorial digital circuits based on the work of Reiter [39] and Hou [29]. DSDC receives a circuit specification in DSDC Language (DSDCL), evaluates different possible measurement orderings via the genetic algorithm, selects the best next measurement and prompts for measurement results from user input. The interactive diagnostic process is going on until only one possible diagnosis is left. Upon this point the user may replace the faulty components found within the system, and use a fault-detection method to verify the soundness of the system [1, 20, 31, 33, 40, 49]. Fig 4.2.1 shows the block diagram of DSDC.

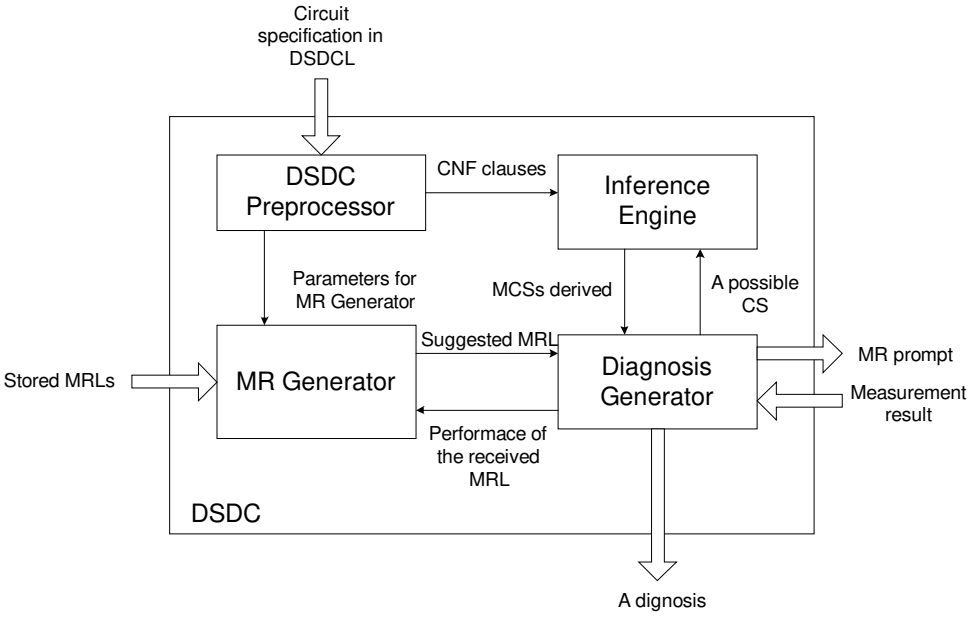

Fig. 4.2.1. The block diagram of DSDC

DSDC Preprocessor translates the circuit specification in DSDCL into a set of clauses in conjunctive normal form (CNF) and other parameters for the use of Inference Engine and MR Generator. A circuit specification contains component definitions, component declarations, component connections, circuit observation and other related parameters for DSDC.

Inference Engine receives the CNF clauses from DSDC Preprocessor and responds the MCS derivation requests from Diagnosis Generator with the derived MCSs. It uses an efficient incremental PC prover for deriving MCSs from a possible CS using the derivation method described in Procedure 2.4.6 (derivation using CS-tree with mark set). The PC prover exploits simple idea of chronological backtracking so that when deriving MCSs using CS-tree with mark set, the previous unit inferences made along the same path in CS-tree and for the mark set elements will be reserved and reused. To improve the efficiency of the PC prover, we choose trie data structure for representing CNF clauses internally, and various reasoning rules are performed based on the work of Zhang and Stickel [53].

Diagnosis Generator is responsible for generating all possible diagnoses, i.e., all MHSs from the collection of MCSs received from Inference Engine. It then updates equivalence classes of components and homogeneity classes of diagnoses using Procedure 2.3.8 and 2.3.19. To discriminate among competing diagnoses, Diagnosis Generator obtains a Measurement Request List (MRL) from Measurement Request Generator (MR Generator) and prompts for the measurement result from user input in the order specified in the MRL. Such interactive

diagnostic process is going on until only one possible diagnosis is left.

MR Generator generates a promising MRL according to past experiences of diagnosing the circuit. A promising MRL should minimize the effort for deriving the unique cause of the disorders of the circuit. Specifically speaking, the MR Generator of DSDC exploits MSSGA described in Chapter 3 for evolving and determining promising MRLs. The population of MRLs required by MR Generator is generated randomly or obtained from the previous runs on the same circuit. Various parameters for GA are given in the circuit specification in DSDCL.

In summary, DSDC receives a circuit specification in DSDCL, a population of MRLs, and user inputs of measurement results. It prompts for measurement results from user input, and deduces all possible diagnoses according to the information obtained thus far. DSDC then reports the only possible cause of the disorders of the circuit. Initially, the population of MRLs for DSDC can be generated randomly. The average cost of MRLs can be reduced by *training* DSDC to repeatedly diagnose the same circuit before it is used online. However, one can also skip such pre-training session and let DSDC improve itself online.

4.3 DSDC Preprocessor and DSDC Language (DSDCL)

In DSDC, the model of a combinatorial digital circuit is built by using DSDC Language (DSDCL). DSDC Preprocessor translates the circuit specification in DSDCL into a set of CNF clauses and other related parameters for the other modules of DSDC. This section presents the details of DSDCL, and a complete language formulation of DSDCL in Backus-Naur form is given.

First of all, DSDCL is a case-insensitive language. The terminals of DSDCL are listed in the following table.

keyword:	one of **define** **declare** **connect** **observe** **assign** **in/in***x* **out/out***x*
token:	Any string other than any one of the keywords
real:	A real number
integer:	An integer
output-predicate:	**out**[*token*] **out***x*[*token*]
input-predicate:	**in**[*token*] **in***x*[*token*]
truth-value:	either **0** or **1**
unary-operator:	~
binary-operator:	one of ( ) ~ * + @ = >

Table 4.3.1. All terminals of DSDCL

Note that **in***x* and **out***x* stand for **in1**, **in2**,... and **out1**, **out2**, ..., respectively. For **in1**

and **out1** one can also use **in** and **out** as a shorthand notation, provided that only one input/output exists. The meaning and the precedence of each operator used in DSDCL is listed below.

Operator	Meaning	Precedence
( )	Parentheses	0
~	Not	1
*	And	2
+	Or	3
@	Xor	3
=	Equality	4
>	Implication	5

Table 4.3.2. The meaning and the precedence of each operator used in DSDCL

DSDC Preprocessor translates an expression using *Huntington's postulates* [27] into its equivalent CNF clauses. In particular, the equality "$a = b$" is translated into $(\neg a \vee b) \wedge (\neg b \vee a)$. Since we assume that the design of the circuit is flawless, another approach to handle equalities is to replace all equivalent variables with one representative equivalent variable. However, this replacement can be only applied on connections between gates (with the representative variable being the only one output predicate in that equivalence class).

The following table shows the basic structure of a circuit description.

circuit-specification:
define-block declare-block connect-block observe-block assign-block

Table 4.3.3. The basic structure of a circuit specification in DSDCL

In short, we first define all kinds of the components used in a circuit in *define-block*, declare a name for each component in *declare-block*. Then in *connect-block* we connect the components, observe the inputs and the outputs of the circuit in *observe-block*, and finally assign values to the parameters used in DSDC in *assign-block*.

A *define-block* is defined as follows.

define-block:	**define** { *define-sub-blocks* }
define-sub-blocks:	*define-sub-block* *define-sub-block define-sub-blocks*
define-sub-block:	*token* { *define-expressions* }
define-expressions:	*define-expression* *define-expression define-expressions*
define-expression:	*output-predicate* = *define-primary-expression*;
define-primary-expression:	*input-predicate* *unary-operator define-primary-expression* *define-primary-expression binary-operator define-primary-expression* (*define-primary-expression*)

Table 4.3.4. The formulation of *define-block* in DSDCL

In practical diagnostic tasks it is helpful to decompose the whole circuit into several large parts first. After identifying which part causes the malfunctions we can then break the part down to

smaller pieces. This approach is called *hierarchical diagnosis*, and can be done in DSDC by carefully writing the definitions of *components*. Later we will show an example circuit to illustrate how to perform hierarchical diagnosis.

The formulation of *declare-block* is shown as follows.

declare-block:	**`declare`** { *declare-sub-blocks* }
declare-sub-blocks:	*declare-sub-block* *declare-sub-block declare-sub-blocks*
declare-sub-block:	*token* { *tokens* }
tokens:	*token*; *token*; *tokens*

Table 4.3.5. The formulation of *declare-block* in DSDCL

Note that *token* outside the braces of *declare-sub-block* must be one of the component names defined in *define-block*.

The following table shows the formulations of the other blocks of a circuit specification in DSDCL.

connect-block:	**`connect`** { *connect-expressions* }
observe-block:	**`observe`** { *observe-expressions* }
assign-block:	**`assign`** { *assign-sub-blocks* }
connect-expressions:	*connect-expression* *connect-expression connect-expressions*
observe-expressions:	*observe-expression* *observe-expression observe-expressions*
assign-sub-blocks:	*assign-expression* *assign-sub-block* *assign-expression assign-sub-blocks* *assign-sub-block assign-sub-blocks*
connect-expression:	*output-predicate* = *input-predicate*; *input-predicate* = *output-predicate*;
observe-expression:	*output-predicate* = *truth-value*; *input-predicate* = *truth-value*;
assign-sub-block:	*token* { *assign-sub-blocks* }
assign-expression:	*token* = *real*; *token* = *integer*;

Table 4.3.6. The formulations of *connect-block*, *observe-block* and *assign-block* in DSDCL

Note that DSDCL provides the facility for assigning a ‘group’ of related parameters in an *assign-sub-block*.

Now we illustrate the use of DSDCL with an example circuit: a cascaded two-bit full

adder. The circuit is shown in Fig. 4.3.1.

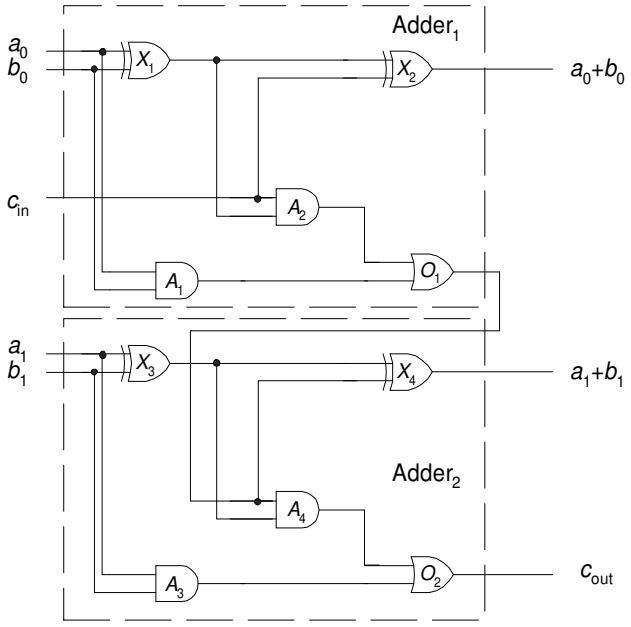

Fig. 4.3.1. A cascaded two-bit full adder

First we decompose the whole circuit into two one-bit full adders and write the following circuit specification.

```
// define a one-bit full adder
define
{
  ONE-BIT-ADDER
  {
    out1=(in1@in2)@in3;
    out2=((in1@in2)*in3)+(in1*in2);
  }
}

// declare all components
declare
{
  ONE-BIT-ADDER { fa1; fa2; }
}

// connect two adders
connect
{
  out2[fa1]=in3[fa2];
}
```

```
// describe the observation
observe
{
  in1[fa1]=1;
  in2[fa1]=1;
  in3[fa1]=1;
  in1[fa2]=0;
  in2[fa2]=1;

  out1[fa1]=0;
  out1[fa2]=0;
  out2[fa2]=1;
}

// assign values to the parameters
assign
{
  // omitted for brevity
}
```

Table 4.3.7. The circuit specification for the cascaded two-bit full adder at larger granularity

After the first run, DSDC concludes that full adder 1 is the faulty component. We can then decompose the circuit at a finer granularity for deriving a more detailed diagnosis, i.e., focus on the details of full adder 1. The specification of the circuit is shown below. Note that the value at the output of gate O_1 has to be measured before proceeding.

```
// define the gates used in a one-bit full
// adder
define
{
  AND { out=in1*in2; }
  OR { out=in1+in2; }
  XOR { out=in1@in2; }
}

// declare all components
declare
{
  AND { a1;a2; }
  XOR { x1;x2; }
  OR { o1; }
}

// connect the gates
connect
{
  out[x1]=in2[a2];
  out[x1]=in1[x2] ;
  out[a2]=in1[o1];
  in1[a2]=in2[x2];
  in1[x1]=in1[a1];
  in2[x1]=in2[a1];
  out[a1]=in2[o1];
}

// describe the observation
observe
{
  in1[x1]=1;
  in2[x1]=1;
  in1[a2]=1;

  out[x2]=0;
  out[o1]=1;
}

// assign values to the parameters
assign
{
  // omitted for brevity
}
```

Table 4.3.8. The circuit specification for the cascaded two-bit full adder at finer granularity

4.4 Inference Engine

The most important sub-tasks in a diagnostic process are MCS derivation and the underlying logic inferences. A sound and complete diagnostic system relies on the correctness of its reasoning mechanism. Moreover, the performance of such system is almost totally determined by the efficiency of the reasoning mechanism used, since most of the time of deriving a diagnosis in a diagnostic process is spent on these reasoning sub-tasks.

In DSDC, Inference Engine is responsible for deriving all MCSs from a possible CS. The derivation method used is Procedure 2.4.6, and the underlying inferences are made via a sound and complete propositional calculus prover (PC prover). A PC prover solves the satisfiability (SAT) problem, which is a well-known NP-complete problem [18]. To make DSDC practical, improvements must be devised to improve the efficiency of the PC prover used. We enhance the efficiency of Inference Engine in DSDC by using an incremental Davis-Putnam PC prover [10] implemented by trie data structure [53], as detailed in the following sub-sections.

4.4.1 The Method of Davis and Putnam

Although the workings of Reiter's algorithm for deriving diagnoses requires a sound, complete and decidable prover, he made no claim to the preferred theorem prover. In [5], an SAT competition has been held and the Davis-Putnam-based [10] program is considered the quickest one for solving the SAT problem. This encourages us to choose the Davis-Putnam method as the underlying algorithm for our PC prover. In short, Davis-Putnam procedure consists of the following rules [6]. Let S be a set of propositional clauses:

1. *Tautology Rule*: Delete all the clauses in S that are tautologies. If S becomes an empty set, then S is valid. Otherwise S is unsatisfiable *iff* the remaining set S' is.

2. *One-literal Rule*: Suppose L is a unit clause in S. If $\neg L$ is also a unit clause, then S is unsatisfiable. Otherwise, delete all the clauses that contain L. If the remaining set S' is empty, S is satisfiable. Otherwise we obtain a set S'' by deleting $\neg L$ from S', S is unsatisfiable *iff* S'' is.

3. *Pure-literal Rule*: A literal L is pure if $\neg L$ does not appear in S. If L is pure, then we can

obtain a set S' by deleting all the clauses containing L. If S' is empty, S is satisfiable. Otherwise, S is unsatisfiable *iff* S' is.

4. *Splitting Rule*: If the set S can be put into the form $(A_1 \vee L) \wedge ... \wedge (A_m \vee L) \wedge (B_1 \vee \neg L) \wedge ... \wedge (B_n \vee \neg L) \wedge R$, where A_i, B_i and R do not contain L and $\neg L$, then obtain the sets $S_1 = A_1 \wedge ... \wedge A_m \wedge R$ and $S_2 = B_1 \wedge ... \wedge B_n \wedge R$. S is unsatisfiable *iff* both S_1 and S_2 are unsatisfiable.

Among the rules listed above, *one-literal rule* is sometimes called 'unit propagation' and is divided into two parts: *unit resolution* [$A \wedge (\neg A \vee B)$ is equivalent to $A \wedge B$] and *unit subsumption* [$A \wedge (A \vee B)$ is equivalent to A]. Moreover, *pure-literal rule* is not necessary for the completeness.

4.4.2 A PC Prover Implemented by Trie Data Structure (PCPT)

This sub-section presents a Davis-Putnam PC prover implemented by trie data structure (PCPT) in DSDC based on the work of Zhang and Stickel [53]. PCPT represents ground CNF clauses in trie data structure, and efficiently performs unit propagation on the representation. The nondestructive approach used in PCPT makes it very suitable for repeated satisfiability computation.

We first associate each variable in a set of ground CNF clauses with a unique positive index. A negative literal is represented by the negation of the index of the corresponding variable. A trie structure of a set of ground CNF clauses is then defined as follows.

Definition 4.4.1 A trie structure of a set of ground CNF clauses is a 3-ary *tree* defined below.

(1) Each node of the structure is either labeled by empty (•), labeled by a clause end-mark (□), or labeled by a 4-tuple <*var*, *pos*, *neg*, *rest*>, where *var* is a variable index, *pos* is its positive child node, *neg* is its negative child node and *rest* is its brother node.

(2) An empty node or a node labeled by a clause end-mark has no child and brother nodes.

(3) The left edge of a node labeled by <*var*, *pos*, *neg*, *rest*> is interpreted as the positive literal of the variable with index *var*. The right edge of the node is interpreted as the negative literal of the variable with index *var*. Brother edges, namely the edges between brother nodes, represent nothing.

(4) A clause is obtained from the collection of left/right/brother edges of the path starting from the root node to some clause end-mark node.

Note that a trie structure cannot represent tautologies. However, in the context of digital circuit design, this is rarely a problem.

A trie structure of a set of ground CNF clauses is said to be *ordered* if for each node labeled by <*var*, *pos*, *neg*, *rest*> in the tree, *var* is smaller than any variable index appearing in *pos*, *neg* and *rest*. Fig. 4.4.1 shows the trie structure of the clause $(a \vee b) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee c)$. Note that how a node can be shared to save storage space. Also Note that if we index the variables as {a=1, b=2, c=3}, then the trie structure shown in Fig. 4.4.1 is ordered.

PCPT constructs an ordered trie structure from a set of ground CNF clauses. Clause subsumption automatically occurs at construction time. No pure-literal checking is done

because the situation rarely happens[14].

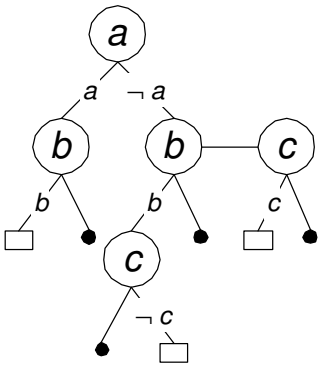

Fig. 4.4.1. The trie structure of the clause $(a \vee b) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee c)$

In PCPT, the *unit resolution* technique of Davis-Putnam procedure is implemented on an ordered trie structure by first building a *variable table*.

Definition 4.4.2 A *variable table* is an array of variable records. Each variable has its own variable record, which is a 4-tuple <*truth*, *head-list*, *positive-tail-list*, *negative-tail-list*>, where *truth* denotes the truth value (TRUE, FALSE or UNASSIGNED) of the corresponding variable, *head-list* is a list containing all occurrences, namely the corresponding nodes in the trie structure, of the corresponding variable as the first variable in the clauses, and *positive-tail-list* (resp. *negative-tail-list*) is a list consisting of all occurrences of the corresponding variable in positive (resp. negative) form as the last variable in the clauses.

The unit resolution can then be achieved by updating the variable table. The following procedure is used for updating the variable table in PCPT.

Procedure 4.4.3 Assume that we assign TRUE/FALSE to a variable v, whose variable record is <*truth*, *head-list*, *positive-tail-list*, *negative-tail-list*>. Update the variable table as follows.

(1) Assign TRUE/FALSE to *truth*.

(2) Update *head-list* if it is not empty:

Assume that the element in *head-list* of variable v points to a node n in the trie structure. Remove the element. Assume the case is *truth* = TRUE (resp. FALSE). If the right (resp. left) child of n in the trie structure is a clause end-mark, then a null clause is found. If the right (resp. left) child is empty, stop updating *head-list*. Otherwise add the elements pointing to the right (resp. left) child of n and the brothers of the right (resp. left) child of n into their corresponding *head-list* respectively if the truth values of the corresponding variables are UNASSIGNED. For each of those added elements, assume the variable of the node pointed by it is v'. If *positive-tail-list/negative-tail-list* of v' contains the element pointing to the same node, then either v' or $\neg v'$ is a unit clause. For the right (resp. left) child of n and each of the brothers of the right (resp. left) child of n such that the truth value of the corresponding variable v'' is not UNASSIGNED, set $v = v''$ and repeat this step.

[14] Although it is possible that only one polarity of a literal exists in a circuit's input vectors, the negation of the literal will be introduced when translating the DSDCL clauses involving the operators such as NOT, XOR and EQUALITY into CNF clauses. Another reason to ignore *pure-literal rule* in PCPT is that it is intrinsically difficult to implement for the trie structure.

(3) Update *positive-tail-list* and *negative-tail-list* if they are not empty:

For variable *v,* if it is assigned *truth* = TRUE (resp. FALSE), remove all elements of its *positive-tail-list* (resp. *negative-tail-list*). If no parent node exists for the nodes pointed by the elements of *negative-tail-list* (resp. *positive-tail-list*), then a null clause is found. Otherwise for each of the nodes pointed by the elements of *negative-tail-list* (resp. *positive-tail-list*), add an element pointing to the parent node of it in *positive-tail-list/negative-tail-list* of the variable of the parent node if the truth value of the variable of the parent node is UNASSIGNED. For each of those added elements, assume the variable of the node pointed by it is v'. If *head-list* of v' contains the element pointing to the same node, then either v' or $\neg v'$ is a unit clause. Moreover, if the tail list with the opposite polarity to the tail list which the new element is added contains the element pointing to the same node, then a null clause is found. For each of the nodes pointed by the elements of *negative-tail-list* (resp. *positive-tail-list*) such that the truth value of the corresponding variable v'' of its parent node is not UNASSIGNED, set $v = v''$ and repeat this step.

The unit clauses found in the unit resolution process are collected in a unit clause list in PCPT. If a null clause is found in the unit resolution process, PCPT will rewind the assignments to the most recent splitting point.

In [53], a complexity analysis has shown that Procedure 4.4.3 takes an amortized time of $O(N_v)$ when v is assigned to true and $O(P_v)$ when v is assigned to false, where P_v is the number of clauses in which variable v appears positively, and N_v is the number of clauses in which variable v appears negatively.

In addition to Procedure 4.4.3, PCPT uses another technique to efficiently subsume clauses in the course of reasoning. When a node n is labeled by a clause end-mark, the elements in all *positive-tail-lists* and *negative-tail-lists* pointing to node *n*, each of the brother nodes of node *n*, and all descendants of the nodes listed above are removed from the lists. This is done by traversing in the trie starting from node *n* and doing tail deletion. To avoid reaching node *n*, the brothers of *n* and all of their descendants again when an antecedent node of node *n* is labeled by a clause end-mark, we mark each node we reach when deleting the tails.

The splitting rule of Davis-Putnam method is handled in PCPT as follows. If no unit clause is available, an unassigned variable is chosen such that the total number of elements in its *head-list*, *positive-tail-list* and *negative-tail-list* is the largest one compared to all other unassigned variables. The intuition is that selecting such a variable might lead us to update more variable records than choosing other unassigned variables. Each assignment step is then recorded down in a stack so that future rewinding is possible.

In summary, PCPT has the following characteristics:

1. PCPT uses an ordered trie structure for representing a set of ground CNF clauses. Storage consumption is reduced by sharing common nodes among different clauses, and clause subsumption is taken care of automatically at the construction time.

2. Unit propagation in PCPT is nondestructive. The trie structure remains intact during the course of reasoning, and only the variable table is modified. In particular, a clause is considered satisfied when either its first literal or the last literal is TRUE. The literals between the first and the last literal are ignored from the assignment process if

the clause is already satisfied. The clause subsumption is also efficiently achieved in PCPT by updating the variable table.

3. Each assignment step in PCPT is recorded in a stack. The assignment process can then be backtracked *chronologically*. This characteristic enables incremental proving described in the next sub-section.

4.4.3 Incremental Proving Using PCPT

Essentially, deriving MCSs from a CS involves inserting and retracting unit assumptions ($\neg$AB literal of a particular component) and compute the satisfiability. By Procedure 2.4.6, a CS-tree with mark set is searched in depth-first style: a set of unit assumptions is inserted into the unit clause list of PCPT, and the satisfiability is computed. If the set is a CS, a descendant with only one element less is generated and tested. An efficient way for testing is try to reuse as more the inferences made at the parent set as possible.

In DSDC, we exploit simple idea of chronological backtracking [47] to improve the efficiency of deriving MCSs from a CS. At the root node in a CS-tree with mark set, for each unit assumption except for the last one, we insert it into PCPT's unit clause list and initiate PCPT to run the assignment process to the point where no unit clause is available and a splitting has to be made. An *s-point* mark is then placed at the top of the rewinding stack. For the last unit assumption, we do the same thing except that we run the assignment process to the end, and we do not place any s-point mark. When testing the descendant set, we simply rewind PCPT to the last s-point marked.

To make such incremental proving possible, the order of inserting unit assumptions is important. In Inference Engine of DSDC, we insert the unit assumptions in a mark set first, then we push the other unit assumptions into the unit clause list of PCPT.

4.5 Diagnosis Generator

Diagnosis Generator in DSDC is responsible for generating all possible diagnoses given the observations obtained so far. In [39], Reiter has shown that an MHS for the collection of all MCSs for (SD, COMP, OBS) is a diagnosis for (SD, COMP, OBS). Moreover, he has proposed an MHS derivation method using a pruned HS-tree to computes all MHSs for a collection of MCSs. His method is summarized as follows.

Definition 4.5.1 Let F be a collection of sets. An HS-tree T for F is an edge-labeled and node-labeled tree defined by the following rules.

(1) Its root is labeled by ✓ if F is empty. Otherwise, its root is labeled by a set of F.

(2) If n is a node of T, define $H(n)$ to be the set of edge labels on the path in T from the root node to n. If n is labeled by ✓, it has no descendant nodes in T. If n is labeled by a set S of F, then for each $c \in S$, n has a descendant node n_c joined to n by an edge labeled by c. The label for n_c is a set $S' \in F$ such that $S' \cap H(n_c) = \{\ \}$ if such set S' exists. Otherwise, n_c is labeled by ✓.

Reiter has shown that for a node n labeled by ✓, $H(n)$ is an HS for F. Moreover, the set of all $H(n)$ such that node n is labeled by ✓ includes all MHSs. To improve the efficiency, a pruned HS-tree is generated as follows.

Procedure 4.5.2[15] Generate a pruned HS-tree for F, a collection of MCSs, as described below.

(1) Generate the HS-tree breadth-first, generating nodes at any fixed level in the tree in left-to-right order.

(2) Reusing node labels: If node n is labeled by the set $S \in F$, and if n' is a node such that $H(n') \cap S = \{\ \}$, label n' by S. Such node n' requires no access to F.

(3) Tree Pruning:

(a) For node n' if node n is labeled by ✓ such that $H(n) \subseteq H(n')$, close n', i.e., do not compute a label for n' and do not generate any successors of n'.
(b) For node n' if node n has been generated such that $H(n') = H(n)$, then close n'.
(c) If nodes n and n' have been respectively labeled by sets S and S' of F, and if S' is a proper subset of S, then for each $\alpha \in S - S'$ mark as redundant the edge from node n labeled by α. A redundant edge, together with the subtree beneath it, may be removed from the HS-tree without losing any MHS.

After generating all the nodes in an HS-tree as described above, for each node n labeled by ✓ $H(n)$ is an MHS and all MHSs must be found this way. Note that Procedure 4.5.2 does not require that all MCSs are explicitly given. Instead, a sound and complete prover is called at node n to compute a CS which does not intersect $H(n)$. An important point is that the CS obtained is not necessarily minimal for the correctness of Procedure 4.5.2.

Although Reiter's method for deriving all MHSs is theoretically correct, it turns out that it is not suitable for a practical diagnostic system. Consider the example below.

Example 5.3 Consider the circuit in Example 2.4.4 (shown in Fig. 2.4.3). There are four MCSs: $\{A_1, O_1\}$, $\{A_2, O_1\}$, $\{A_3, O_1\}$ and $\{O_1, O_2\}$. The MHSs for the MCSs are $\{O_1\}$ and $\{A_1, A_2, A_3, O_2\}$. Figure 4.5.1 shows the derivation of all MHSs following Procedure 4.5.2.

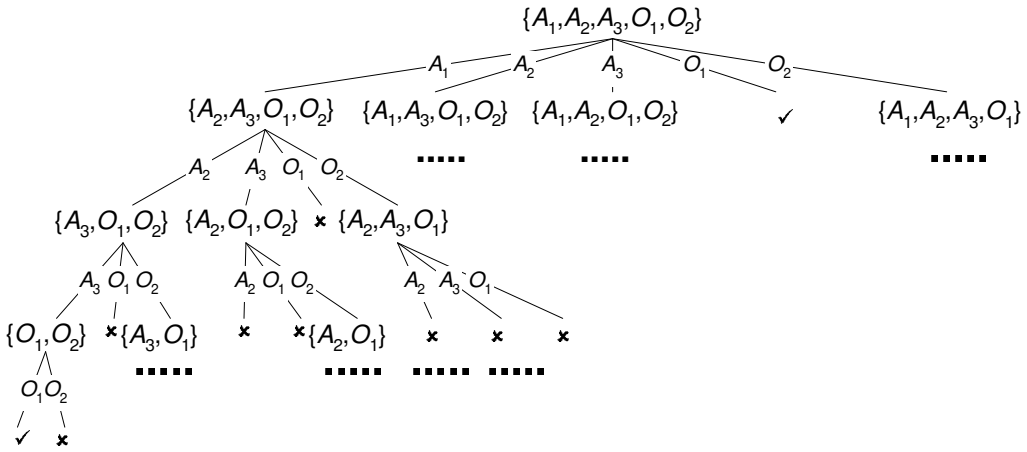

Fig. 4.5.1. Deriving all MHSs for the circuit shown in Fig. 2.4.3 using Reiter's pruned HS-tree

Note that no node reusing ever occurred in the derivation. This is because we generate the

[15] In [23], it has been shown that pruning rule (ii) may prevent some MHS from being computed, and a corrected algorithm using HS-dag (directed acyclic graph) has been given and proved.

nodes in the HS-tree in breadth-first style, and each node n is labeled by a maximal CS C such that $C \cap H(n) = \{\}$. Unless the prover returns a smaller CS, it is impossible to reuse any CS found before.

□

To ease the situation, one can either have the prover return an MCS, or generate the nodes in an HS-tree depth-first. In the first approach, the task of computing all MHSs is divided into the derivation of MCSs and the computation of all MHSs from all MCSs derived. The other approach breaks the minimality of $H(n)$ of node n labeled by ✓. In Diagnosis Generator of DSDC, the first approach is used, i.e., we first derive all MCSs by using Procedure 2.4.6, and then an HS-tree is constructed for computing all MHSs based on all MCSs derived. In this way the pruning rule (iii) in Procedure 4.5.2 will never be used, and the algorithm for constructing an HS-tree is greatly simplified.

After deriving all possible diagnoses, Diagnosis Generator prompts for the measurement result from user input according to the MRL it receives from Measurement Request (MR) Generator. It discriminates among competing diagnoses based on the measurement result using Hou's Theorem 3.13 in [29] (described in Section 2.2.2). Equivalence classes of components and homogeneity classes of diagnoses are then updated using Procedure 2.3.8 and 2.3.19. When only one possible diagnosis is left, Diagnosis Generator reports the number of prover calls of each measurement request to MR Generator as the indication of the MRL's performance. This data forms the basis on which MR Generator will gradually find effective MRLs for future diagnostic processes.

4.6 Measurement Request Generator[16]

For a nontrivial circuit with the initial observation only, we always have more than one possible diagnosis. To discriminate among these competing diagnoses, new information must be provided by probing some terminals of the components within the system. The effects brought by such measurements on the possible diagnoses have been formalized in the work of Reiter [39] and Hou [29]. However, they did not address the problem as how to select the best next measurement. In DSDC, Measurement Request (MR) Generator is responsible for pointing out the best next measurement by exploiting the *m*easurement *s*election *s*trategy using a steady-state with least-fit-deletion *g*enetic *a*lgorithm (MSSGA) proposed in Chapter 3. The reasons to choose a genetic algorithm (GA) as a measurement selection strategy are also discussed in Chapter 3.

In short, MSSGA evolves a population of measurement orderings (measurement request lists, MRLs) according to the performance index (*PI*) of each MRL within the population. MSSGA is a steady-state GA with least-fit-deletion, and no duplicate is allowed in the population. An MRL is treated as a chromosome directly in order to facilitate the workings of the order based GA. The raw fitness values are converted from the *PI* values, and a linear scaling mechanism is used to scale the raw fitness values. Individual MRL is selected using the roulette-wheel selection technique based on the scaled fitness values of the MRLs. Four genetic operators are used, with the reevaluation operator being used to update the inaccurate fitness value of an MRL in a noisy environment. The operator fitness values are adapted dynamically and automatically according to their online performance.

[16] Our measurement selection strategy using the genetic algorithm is detailed in Chapter 3 with the results of diagnosing five testing circuits. This sub-section simply provides an overview of the technique we use.

4.7 Example Runs

To illustrate the use of DSDC, the snapshots of two example runs are presented in this section. The first circuit diagnosed is an arbitrary circuit with 8 gates (circuit c01_8), and the second one is a BCD to decimal decoder (7442) with 18 gates. The complete listings of these two circuits in DSDCL format, together with the other three testing circuits used in this thesis, are given in Appendix B. To prepare the populations of MRLs to be used in these two tests, we let DSDC evolve them from initial random populations under random fault patterns[17]. In the first test the population size is 20, and the span of the evolution is 100 evaluations. In the second test the population size is 40, and the span of the evolution is 200 evaluations. The other GA parameters are listed in Appendix B (noisy test). The tests were run on an AMD5x86 PC with 32MB RAM in a console window under Microsoft Windows 95 operating system.

4.7.1 The First Example: An Arbitrary Circuit with 8 Gates

```
d:\bc5\works\dsdc>DSDC c01_8 diagAuto
-----------------------------------------------------------
# Number of components: 8
# Number of variables: 32
# Number of non-unit clauses: 40
# Number of unit clauses: 0
-----------------------------------------------------------

# Equivalence classes:
{ A1 }
{ O2 A5 }
{ A6 A2 }
{ O1 A4 A3 }

# Homogeneity classes (diagnoses):

## Class:
{ A2 }
{ A6 }

## Class:
{ A3 A5 }
{ O1 O2 }
{ A5 O1 }
{ A4 O2 }
{ A4 A5 }
{ A3 O2 }
-----------------------------------------------------------
# Time elapsed: 0.186 second(s).
# Number of PC-prover calls: 39
-----------------------------------------------------------

* Fine-tune:
4, 54, 216, 45, 49.1077: (7,5) (15,10) (3,6) (11,16) (30,0) (26,0) (19,0)

average proverCall = 13.5
  -----------
0, 0, 0, 45, 49.1077: (11,0) (7,0) (15,0) (3,0) (30,0) (26,0) (19,0)

* Input the measurement result at OUT1(A3) [type 'EXIT' to quit]: 1

# Equivalence classes:
{ A3 }
{ A1 }
{ O2 A5 }
{ O1 A4 }
{ A6 A2 }

# Homogeneity classes (diagnoses):

## Class:
{ A2 A3 }
{ A3 A6 }

## Class:
{ A3 A5 }
```

[17] Actually the random fault patterns are generated according to a preset error rate distribution.

```
{ A3 O2 }
-----------------------------------------------------------
# Time elapsed: 0.082 second(s).
# Number of PC-prover calls: 54
-----------------------------------------------------------

* Input the measurement result at OUT1(A2) [type 'EXIT' to quit]: 1

# Equivalence classes:
{ A6 }
{ A3 }
{ A2 }
{ A1 }
{ O2 A5 }
{ O1 A4 }

# Homogeneity classes (diagnoses):

## Class:
{ A2 A3 }
-----------------------------------------------------------
# Time elapsed: 0.025 second(s).
# Number of PC-prover calls: 63
-----------------------------------------------------------
* Only one diagnosis is left, hit any keys other than 'C' to stop...
```

4.7.2 The Second Example: A BCD to Decimal Decoder (7442) with 18 Gates

```
d:\bc5\works\dsdc>DSDC c02_18 diagAuto
-----------------------------------------------------------
# Number of components: 18
# Number of variables: 84
# Number of non-unit clauses: 154
# Number of unit clauses: 0
-----------------------------------------------------------

# Equivalence classes:
{ NA4_7 }
{ NA4_5 }
{ NA4_3 }
{ NA4_1 }
{ BN16 }
{ BN15 }
{ BN14 }
{ BN13 }
{ NA4_10 NA4_9 NA4_8 NA4_6 NA4_4 NA4_2 BN18 BN17 BN12 BN11 }

# Homogeneity classes (diagnoses):

## Class:
{ BN16 NA4_5 }

## Class:
{ NA4_5 NA4_7 }

## Class:
{ BN15 NA4_3 NA4_5 }

## Class:
{ BN13 BN14 BN15 NA4_3 }

## Class:
{ BN13 BN14 BN16 NA4_1 }

## Class:
{ BN13 BN15 NA4_3 NA4_7 }

## Class:
{ BN13 BN16 NA4_1 NA4_3 NA4_7 }
-----------------------------------------------------------
# Time elapsed: 46.528 second(s).
# Number of PC-prover calls: 5966
-----------------------------------------------------------

* Crossover:
3, 4321, 12963, 183.661, 171.191: (24,94) (9,1286) (15,162) (12,0) (18,0) (21,0)
(3,0) (6,0)
1, 22648, 22648, 105.122, 166.072: (18,2923) (3,0) (15,0) (9,0) (6,0) (24,0) (21
```

```
,0) (12,0)

average proverCall = 1440.33
average proverCall = 22648

  -----------
0, 0, 0, 183.661, 171.191: (18,0) (3,0) (15,0) (9,0) (6,0) (24,0) (21,0) (12,0)

0, 0, 0, 105.122, 166.072: (24,0) (9,0) (15,0) (12,0) (18,0) (21,0) (3,0) (6,0)


* Input the measurement result at OUT1(BN16) [type 'EXIT' to quit]: 1

# Equivalence classes:
{ NA4_7 }
{ NA4_5 }
{ NA4_3 }
{ NA4_1 }
{ BN16 }
{ BN15 }
{ BN14 }
{ BN13 }
{ NA4_10 NA4_9 NA4_8 NA4_6 NA4_4 NA4_2 BN18 BN17 BN12 BN11 }

# Homogeneity classes (diagnoses):

## Class:
{ BN16 NA4_5 }

## Class:
{ BN15 NA4_3 NA4_5 }

## Class:
{ BN13 BN14 BN15 NA4_3 }

## Class:
{ BN13 BN14 BN16 NA4_1 }

## Class:
{ BN13 BN15 NA4_3 NA4_7 }

## Class:
{ BN13 BN16 NA4_1 NA4_3 NA4_7 }
-----------------------------------------------------------
# Time elapsed: 3.194 second(s).
# Number of PC-prover calls: 6264
-----------------------------------------------------------

* Input the measurement result at OUT1(BN11) [type 'EXIT' to quit]: 1

# Equivalence classes:
{ NA4_7 }
{ NA4_5 }
{ NA4_3 }
{ NA4_1 }
{ BN16 }
{ BN15 }
{ BN14 }
{ BN13 }
{ NA4_10 NA4_9 NA4_8 NA4_6 NA4_4 NA4_2 BN18 BN17 BN12 BN11 }

# Homogeneity classes (diagnoses):

## Class:
{ BN16 NA4_5 }

## Class:
{ BN15 NA4_3 NA4_5 }

## Class:
{ BN13 BN14 BN15 NA4_3 }

## Class:
{ BN13 BN14 BN16 NA4_1 }

## Class:
{ BN13 BN15 NA4_3 NA4_7 }

## Class:
{ BN13 BN16 NA4_1 NA4_3 NA4_7 }
```

```
-----------------------------------------------------------
# Time elapsed: 0.092 second(s).
# Number of PC-prover calls: 6270
-----------------------------------------------------------

* Input the measurement result at OUT1(BN15) [type 'EXIT' to quit]: 1

# Equivalence classes:
{ NA4_7 }
{ NA4_5 }
{ NA4_3 }
{ NA4_1 }
{ BN16 }
{ BN15 }
{ BN14 }
{ BN13 }
{ NA4_10 NA4_9 NA4_8 NA4_6 NA4_4 NA4_2 BN18 BN17 BN12 BN11 }

# Homogeneity classes (diagnoses):

## Class:
{ BN16 NA4_5 }

## Class:
{ BN13 BN14 BN16 NA4_1 }

## Class:
{ BN13 BN16 NA4_1 NA4_3 NA4_7 }
-----------------------------------------------------------
# Time elapsed: 3.227 second(s).
# Number of PC-prover calls: 6537
-----------------------------------------------------------

* Input the measurement result at OUT1(BN13) [type 'EXIT' to quit]:

# Equivalence classes:
{ NA4_7 }
{ NA4_5 }
{ NA4_3 }
{ NA4_1 }
{ BN16 }
{ BN15 }
{ BN14 }
{ BN13 }
{ NA4_10 NA4_9 NA4_8 NA4_6 NA4_4 NA4_2 BN18 BN17 BN12 BN11 }

# Homogeneity classes (diagnoses):

## Class:
{ BN16 NA4_5 }
-----------------------------------------------------------
# Time elapsed: 3.256 second(s).
# Number of PC-prover calls: 7053
-----------------------------------------------------------
* Only one diagnosis is left, hit any keys other than 'C' to stop...
```

4.8 Discussions

To conclude this chapter, we first concentrate on comparing DSDC to de Kleer's famous GDE [15]. Then we point out the possible extensions to DSDC.

Although de Kleer's diagnostic procedure is equivalent to Reiter's theory, there are still some significant differences between DSDC and GDE. The first major difference lies in the strategies of measurement selection used, as discussed in Section 3.5. The second difference is the design of the underlying inference engine. While DSDC uses an incremental PC prover implemented by trie data structure for solving the SAT problem, GDE exploits *constraint propagation* [15, 16, 24, 25] technique to detect any disagreement of the values at the same terminal. A truth maintenance system (TMS) [17] is used for recording the dependencies of the inferences made, and contradictory assumptions can be retracted while retaining other irrelevant inferences. One of the disadvantages of using a TMS, however, is that extra overhead is incurred for recording the dependencies of the inferences. In particular, an assumption-based TMS (ATMS) has to precompute a complex label indicating all consistent

assumptions for each TMS node before any inference is made. Moreover, to improve the efficiency an ATMS usually limits the expressive power of the logic used. By contrast, PCPT only logs simple chronological dependencies, and does not require any precomputation nor limit the clauses accepted.

Another difference between DSDC and GDE is the conflict recognition strategies used. DSDC derives all MCSs from a possible CS using a pruned CS-tree with mark set, and adopts Hou's equivalence classes of components and homogeneity classes of diagnoses to safely ignore some redundant MCS derivations. GDE, on the other hand, uses a bottom-up approach with five efficiency refinements to derive all MCSs, and does not exploit any technique for detecting possible redundant derivations. Most part of the five refinements described in [15] are achieved by our MCS derivation method (with an incremental PC prover), except that DSDC cannot reuse all inferences made before (since the underlying PC prover can only retract the inferences made before *chronologically*). For the efficiency comparison between the top-down and the bottom-up approach for deriving MCSs, as discussed in Section 2.4.4, generally which one has an efficiency edge over the other depends on the characteristics of the particular problem to be solved.

One of the possible extensions to DSDC is using an intelligent MCS derivation method. As stated in Section 4.4, the performance bottleneck of the diagnostic system lies in the efficiency of MCS derivation. An intelligent MCS derivation method will automatically change the order of node generation in accordance with the empirical data of error rates known to the diagnostic system, and the derivation can be switched automatically to either top-down or bottom-up. Moreover, some sets can be ignored during the derivation if from the statistical data they and their subsets rarely are MCSs. Although such ignorance breaks the completeness of the diagnostic engine, it will improve the efficiency.

Another possible extension is incorporating other models to describe digital circuits. DSDC knows a digital circuit only from its functional (logical) model. It would be interesting to see how another model, say, a spatial model or a structural model of the circuit, help the diagnostic system locate the culprits more efficiently. Consider an internally disconnected circuit. Let the circuit be divided into two parts C_1 and C_2. Obviously, it is not necessary to consider the components from C_1 and C_2 simultaneously when deriving MCSs. However, it is not easy to obtain such insights of the circuit from its logical model only. A structural model can then be used to represent such information.

Automatically generating a testing observation (circuit inputs/outputs) is another possible extension to DSDC. Essentially, DSDC is a measurement-based diagnostic system - it uses the result of probing the intermediate variables of the circuit to discriminate among competing diagnoses. However, probing internal components is usually considered inconvenient and costly. By smartly using a different observation, the number of possible diagnoses may be reduced, and the number of measurements required to settle down to only one possible diagnosis may also be lowered down.

Chapter 5

Conclusions

To conclude this thesis, we first summarize our contributions in this work. In Chapter 2 we have provided supplementary proofs to augment and clarify some points in Hou's theory of measurement. In particular, we have described the complete picture of the relationships among measurements, conflict sets and diagnoses, discussed the equivalence relation for enhanced conflict recognition, addressed the incompleteness problem of the minimal conflict set (MCS) derivation approach described in [29], and proposed a complete, efficient and proved MCS derivation method using a CS-tree (conflict set tree) with mark set. In Chapter 3 we have proposed a new scheme for measurement selection in a diagnostic process: a *m*easurement *s*election *s*trategy using the *g*enetic *a*lgorithm (MSSGA). Two series of tests in which we used our diagnostic system incorporating MSSGA to repeatedly diagnose five testing circuits in noise-free and noisy environment were done, and the results are quite promising. We have also pointed out the difficulty of de Kleer's one-step lookahead measurement selection strategy used in GDE. In Chapter 4 we have proposed a complete *d*iagnostic *s*ystem for combinatorial *d*igital *c*ircuits (DSDC). DSDC receives a circuit specification in DSDC Language (DSDCL), evaluates different possible measurement request lists (MRL) via MSSGA proposed in Chapter 3, and prompts for measurement results from user input. It automatically locates the culprits using the measurement results fed by a user. Since DSDC incorporates MSSGA, it is capable of improving its efficiency of performing diagnosis *online*.

Some possible extensions to this work have been discussed in the related sections of Chapter 2, Chapter 3 and Chapter 4. Here we address a few more. In Chapter 2 we have suggested to add a reordering mechanism to our MCS derivation method based on the error rate distribution of the system components. In fact, any heuristic rules about the system (digital circuits) being diagnosed could be used to enhance the efficiency of the MCS derivation. More specifically, based on the architecture of DSDC, we may build a specialized rule base for diagnosing digital circuits. The goal of such rule base is to guide the diagnostic system to explore the fraction of the gigantic search space in which the solutions are most probable to exist, but not sacrifice the completeness if we do not want it to. In this way, the diagnostic system will be 'hybridized' in that it exploits both the shallow knowledge and the deep knowledge of the system being diagnosed.

Another possible extension is to incorporate other possible actions which can be used in a diagnostic process. DSDC only receives measurement results probed from the system being diagnosed. It would be interesting to investigate the effects brought by some other actions such as testing (generating another testing observation), component replacing, component repairing and component removing. Moreover, the *PI* values used in MSSGA can incorporate the cost of these actions for evolving a better population of action orderings.

Despite DSDC's capability of automatically diagnosing combinatorial digital circuits, there is a fundamental problem: the vast computation power needed to diagnose a large circuit. Two major sub-tasks in diagnosis from first principles, MCS derivation and MHS computation, are NP-complete problems. Therefore additional mechanisms are required to make such diagnostic system more practical. One possible solution is automatic hierarchical diagnosis. A

diagnostic system with such mechanism can automatically decompose the model of the whole system to be diagnosed into several parts according to some heuristic rules (such as error rate distribution), so that the total cost for obtaining a unique diagnosis can be minimized. Moreover, a parallel diagnostic algorithm can be used with such automatic decomposition method to further speed up the diagnostic process.

Appendix A

An Implementation of DSDC

This appendix introduces our implementation of DSDC. In the first section we briefly describe the development platform and the computer language used for the project. In Section A.2 the usage of DSDC and some example outputs are given. In Section A.3 we briefly discuss the organization of the source code and the portability issues.

A.1 Development Environment

All of the source code of DSDC was developed on an AMD5x86 PC with 32MB RAM using Microsoft Windows 95 as the operating system. We chose C++ as the development language because of its OO ability, its portability, and the availability of a powerful and friendly C++ development environment. In particular, all of the source code was created under Borland C++ version 5.0. The code was built as three Win32 console programs.

A.2 Using DSDC

The system consists of three independent programs: DSDCLP, DSDCINI and DSDC. DSDCLP is the DSDC Preprocessor, which translates a circuit specification file (*.dsf file) into a set of intermediate files for the other programs. DSDCINI is a population initializer, which randomly create a population of MRLs and evaluate each MRL within the population under a randomly generated fault pattern (according to the error rate distribution specified). DSDC is a diagnostic engine, which deduces all possible diagnoses and interacts with a user.

The eight types of files which could be produced by running the programs are explained below:

1. *.cdf files: contain the circuit description in CNF clauses.
2. *.obf files: contain the observation of the circuit in CNF clauses.
3. *.abf files: specify the faulty behavior of each component (inverse normal, stuck at 0, stuck at 1) for the internal fault pattern generator. The type of the faults is specified in the assignment block of the corresponding dsf file.
4. *.cef files: contain the auxiliary information about the malfunctioning circuit such as the lists of all inputs, outputs, possible measurements and the component error rate distribution.
5. *.cpf files: contain the initial values of all parameters used in the diagnostic process.
6. *.mrf files: contain a complete population of MRLs.
7. *.fpf files: specify a randomly generated fault pattern.
8. *.dat files: contain the complete history of a population of MRLs.

Note that the first five types of files are generated by DSDCLP, *.mrf files are created by DSDCINI and DSDC (under `fixed` or `noisy` operations, see below), *.fpf files are produced by DSDCINI, and *.dat files are generated by DSDC. All of these files are pure text files.

The user can launch DSDCLP, DSDCINI and DSDC as follows:

```
dsdclp    <circuit main file name>
```

```
dsdcini   <circuit main file name>
```

```
dsdc  <circuit main file name>
      <fixed | noisy | diagAuto | diagManual >
      [# of evaluations]    [# of faults]
```

where <*circuit main file name*> is the main file name of the circuit specification in DSDCL format (without the file extension name). In executing DSDC, `<fixed |...>` parameter specifies the type of the operation, and the last two parameters are optional. There are four types of operations for DSDC: `fixed` is fixed fault tests, `noisy` is noisy tests, `diagAuto` is diagnosis with MR Generator enabled (DSDC will choose the best next measurement and prompt for the measurement result from user input), and `diagManual` is diagnosis with MR Generator disabled (the user has to choose the next measurement himself). For `fixed` and `noisy`, the number of evaluations for evolving the population of MRLs has to be specified in the third command-line parameter, and for `noisy`, the user also has to specify the number of faulty components for the internal fault pattern simulator in the fourth command-line parameter.

Two example runs of DSDC under `diagAuto` operations have already been listed in Section 4.7. The following command is an example to test a circuit specified in c03_19.dsf in noisy tests with 100 evaluations and 2 faulty components in each random fault pattern simulated:

```
dsdc c03_19 noisy 100 2
```

The screen output will be something like this:

```
-----------------------------------------------------
# Number of components: 19
# Number of variables: 94
# Number of non-unit clauses: 169
# Number of unit clauses: 0
-----------------------------------------------------

* MRL 1:

 ***** Deriving an observable fault pattern: 0
 *** Deriving a fault pattern:  { 4 71 }
 *** Deriving a fault pattern: finished.
* Crossover:
9, 46719, 420471, 44.0265, 408.374: (58,16497) (33,12) (37,45) (45,141) (3,2) (73,2) (91,2)
(68,2) (49,11) (53,0) (63,0) (79,0) (85,0) (41,0)
14, 50010, 700140, 26.4402, 407.701: (33,15) (53,15) (91,8818) (37,8751) (68,9) (85,1067)
(73,6) (79,6) (49,272) (45,13) (3,2) (63,2) (41,2) (58,1027)

 average proverCall = 5191
 average proverCall = 3572.14

  -----------
```

```
0, 0, 0, 44.0265, 408.374: (58,0) (91,0) (37,0) (33,0) (53,0) (68,0) (85,0) (73,0) (79,0)
(49,0) (45,0) (3,0) (63,0) (41,0)
0, 0, 0, 26.4402, 407.701: (91,0) (37,0) (58,0) (33,0) (45,0) (3,0) (73,0) (68,0) (49,0)
(53,0) (63,0) (79,0) (85,0) (41,0)

# Avg fitness: 395.845
# Avg # of meas: 9.875
# Scaling factors: 0.038455, 380.623
# Op fitness: 0.550349 0.0499612 0.249806 0.149884

* Best: 3, 1037, 3111, 5542.7, 593.767: (73,2) (45,2) (53,515) (33,0) (58,0) (41,0) (63,0)
(37,0) (91,0) (79,0) (3,0) (85,0) (68,0) (49,0)
* Worst: 13, 132641, 1.72433e+06, 10, 381.007: (79,5) (33,5) (73,5) (41,5) (53,5) (58,32871)
(3,4) (63,4) (49,4) (45,261) (37,32805) (68,3) (85,32772) (91,0)
==========================================

* MRL 2:

 ***** Deriving an observable fault pattern: 0
 *** Deriving a fault pattern:  { 31 83 }
 *** Deriving a fault pattern: finished.
.........
```

The generation of each MRL from the use of four genetic operators is detailed in this output, and the various parameters concerning the performance of MSSGA are reported. Moreover, the complete log of the evolution process will be stored in file c03_19.dat, and the final population of MRLs is stored in file c03_19.noisy.mrf.

A.3 Source Code

The complete source code of the programs spans approximately 12,000 lines. It is functionally divided into several groups, with each group being placed under a directory. The directory structure is shown below.

1. Basic: A group of template source code is placed under this directory. Specifically, these modules implement various basic building blocks in DSDC project: stacks, queues, lists, exception classes, random number generator, etc.
2. IEngine: The code for Inference Engine of DSDC is placed here. In particular, the code of PCPT is under the sub-directory 'PCPT.' Inference Engine is responsible for MCS derivation and the underlying logic inferences required in DSDC.
3. Dg: This directory contains the code for Diagnosis Generator, which is responsible for MHS derivation and updating of equivalence classes of components and homogeneity classes of diagnoses.
4. Mrg: The code for MR Generator is placed here. MR Generator exploits MSSGA proposed in Chapter 3 for automatic measurement selection.
5. DSDCL: DSDC Preprocessor is implemented in the code under this directory. This version does not fully implement DSDCL as described in Section 4.3. In particular, syntax checking is not complete, and the user must assign the parameters in *assign-block* in a fixed order (see Appendix B for examples).
6. DSDCIni: The population initializer is implemented in the code under this directory. An internal fault pattern simulator is used to derive an observable fault pattern[18] according to

[18] A fault pattern of a circuit is observable if at least one output of the circuit deviates from the correct result. In some occasions the faults in a fault pattern cancel out the effects introduced by them with one another, thus the fault pattern becomes unobservable, i.e., one can never tell if the circuit is malfunctioning from its outside behavior.

the given error rate distribution of the gates in a circuit.

Appendix B

The Complete Listings of Five Testing Circuits in DSDCL

This appendix lists all of the five testing circuits used for the experiments in this thesis. Since the circuit specifications for noisy tests are identical to those for fixed fault tests except that the values of some parameters for MR Generator are different, we only list the specifications for fixed fault tests. The different values of the parameters used for noisy tests are listed below (all of the five circuit specifications use the same set of values for noisy tests).

The initial fitness values of the four genetic operators used in noisy tests:
crossover = 0.55
mutation = 0.05
fine_tune = 0.25
reevaluation = 0.15

Table B.1. The initial fitness values of the four genetic operators used in noisy tests

B.1 Circuit c01_8: An Arbitrary Circuit with 8 Gates

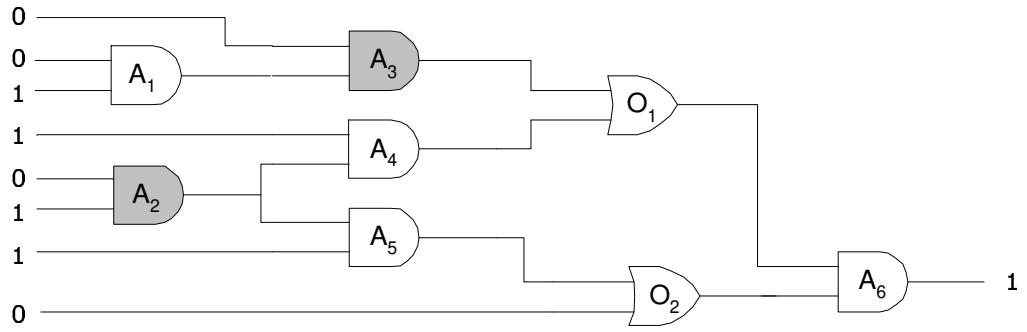

Fig. B.1. Circuit c01_8: An arbitrary circuit with 8 gates

```
// Circuit 01: An arbitrary circuit for testing DSDC
// Total: 8 components

// define the gates used in the circuit
define
{
  AND { out=in1*in2; }
  OR { out=in1+in2; }
}

// declare all gates
declare
{
  AND { a1;a2;a3;a4;a5;a6; }
  OR { o1;o2; }
```

```
}

// connect the gates
connect
{
  in2[a3]=out[a1];
  in2[a4]=out[a2];
  in1[a5]=out[a2];
  in1[o1]=out[a3];
  in2[o1]=out[a4];
  in1[o2]=out[a5];
  in1[a6]=out[o1];
  in2[a6]=out[o2];
}

// describe the observation
observe
{
  in1[a3]=0;
  in1[a1]=0;
  in2[a1]=1;
  in1[a4]=1;
  in1[a2]=0;
  in2[a2]=1;
  in2[a5]=1;
  in2[o2]=0;
  out[a6]=1;
}

// assign the necessary parameters for DSDC to run
assign
{
  // error probability distribution
  errorRate
  {
    a1=0.5;
    a2=0.1;
    a3=0.7;
    a4=0.6;
    a5=0.2;
    a6=0.1;
    o1=0.2;
    o2=0.1;
  }

  // specify the error type for fault pattern derivation:
  //   0: the abnormal output is equal to the inverse of the normal output
  //   1: stuck at 0
  //   2: stuck at 1
  errorType
  {
    a1=0;
    a2=0;
    a3=0;
    a4=0;
    a5=0;
    a6=0;
    o1=0;
    o2=0;
  }

  population=20;
  initialization=0;     // 0: fixed random, 1: random
  initNumFaults=2;
  credit=0.001;
  c=1.5;
  min=10.0;
  cMin=0.0;
  parentLen=5;

  // operators fitness distribution
  operators
  {
    crossover=0.6;
    mutation=0.1;
    fine_tune=0.3;
    reevalutation=0.0;
  }
}
```

B.2 Circuit c02_18: A BCD to Decimal Decoder (7442) with 18 Gates

```
// Circuit 02: BCD to Decimal Decoder (7442) p.48
// Total: 18 components

// define the gates used in the circuit
define
{
 buffer_not { out=~in; }
 nand4 { out=~(in1*in2*in3*in4); }
}

// declare all gates
declare
{
 buffer_not { bn11;bn12;bn13;bn14;bn15;bn16;bn17;bn18; }
 nand4 { na4_1;na4_2;na4_3;na4_4;na4_5;na4_6;na4_7;na4_8;na4_9;na4_10; }
}

// connect the gates
connect
{
 // 10 nand gates first
 in1[na4_1]=out[bn11];
 in2[na4_1]=out[bn13];
 in3[na4_1]=out[bn15];
 in4[na4_1]=out[bn17];

 in1[na4_2]=out[bn12];
 in2[na4_2]=out[bn13];
 in3[na4_2]=out[bn15];
 in4[na4_2]=out[bn17];

 in1[na4_3]=out[bn11];
 in2[na4_3]=out[bn14];
 in3[na4_3]=out[bn15];
 in4[na4_3]=out[bn17];

 in1[na4_4]=out[bn12];
 in2[na4_4]=out[bn14];
 in3[na4_4]=out[bn15];
 in4[na4_4]=out[bn17];

 in1[na4_5]=out[bn11];
 in2[na4_5]=out[bn13];
 in3[na4_5]=out[bn16];
 in4[na4_5]=out[bn17];

 in1[na4_6]=out[bn12];
 in2[na4_6]=out[bn13];
 in3[na4_6]=out[bn16];
 in4[na4_6]=out[bn17];

 in1[na4_7]=out[bn11];
 in2[na4_7]=out[bn14];
 in3[na4_7]=out[bn16];
 in4[na4_7]=out[bn17];

 in1[na4_8]=out[bn12];
 in2[na4_8]=out[bn14];
 in3[na4_8]=out[bn16];
 in4[na4_8]=out[bn17];

 in1[na4_9]=out[bn11];
 in2[na4_9]=out[bn13];
 in3[na4_9]=out[bn15];
 in4[na4_9]=out[bn18];

 in1[na4_10]=out[bn12];
 in2[na4_10]=out[bn13];
 in3[na4_10]=out[bn15];
 in4[na4_10]=out[bn18];

 // 8 buffer-not gates
 in[bn12]=out[bn11];
 in[bn14]=out[bn13];
 in[bn16]=out[bn15];
 in[bn18]=out[bn17];
}
```

```
// describe the observation
observe
{
  in[bn11]=0;
  in[bn13]=1;
  in[bn15]=0;
  in[bn17]=0;

  out[na4_1]=1;
  out[na4_2]=1;
  out[na4_3]=0;
  out[na4_4]=1;
  out[na4_5]=0;
  out[na4_6]=1;
  out[na4_7]=0;
  out[na4_8]=1;
  out[na4_9]=1;
  out[na4_10]=1;
}

// assign the necessary parameters for DSDC to run
assign
{
  // error probability distribution
  errorRate
  {
    na4_1=0.1;
    na4_2=0.2;
    na4_3=0.1;
    na4_4=0.1;
    na4_5=0.7;
    na4_6=0.1;
    na4_7=0.3;
    na4_8=0.4;
    na4_9=0.1;
    na4_10=0.1;
    bn11=0.1;
    bn12=0.1;
    bn13=0.1;
    bn14=0.1;
    bn15=0.9;
    bn16=0.8;
    bn17=0.1;
    bn18=0.1;
  }

  // specify the error type for fault pattern derivation:
  //   0: the abnormal output is equal to the inverse of the normal output
  //   1: stuck at 0
  //   2: stuck at 1
  errorType
  {
    na4_1=0;
    na4_2=0;
    na4_3=0;
    na4_4=0;
    na4_5=0;
    na4_6=0;
    na4_7=0;
    na4_8=0;
    na4_9=0;
    na4_10=0;
    bn11=0;
    bn12=0;
    bn13=0;
    bn14=0;
    bn15=0;
    bn16=0;
    bn17=0;
    bn18=0;
  }

  population=40;
  initialization=0;     // 0: fixed random, 1: random
  initNumFaults=2;
  credit=0.00001;
  c=1.5;
  min=10.0;
  cMin=0.0;
  parentLen=5;
```

```
  // operators fitness distribution
  operators
  {
    crossover=0.6;
    mutation=0.1;
    fine_tune=0.3;
    reevalutation=0.0;
  }
}
```

B.3 Circuit c03_19: A Lookahead Carry Generator (74182) with 19 Gates

```
// Circuit 03: Lookahead Carry Generator (74182) p.128
// Total: 19 components

// define the gates used in the circuit
define
{
  buffer_not { out=~in; }

  or4 { out=in1+in2+in3+in4; }

  nor2 { out=~(in1+in2); }
  nor3 { out=~(in1+in2+in3); }
  nor4 { out=~(in1+in2+in3+in4); }

  and2 { out=in1*in2; }
  and3 { out=in1*in2*in3; }
  and4 { out=in1*in2*in3*in4; }
}

// declare all gates
declare
{
  buffer_not { bn1; }

  or4 { or4_2;or4_3; }

  nor2 { nor2_4; }
  nor3 { nor3_5; }
  nor4 { nor4_6; }

  and2 { a2_7;a2_8;a2_9;a2_10;a2_11; }
  and3 { a3_12;a3_13;a3_14;a3_15; }
  and4 { a4_16;a4_17;a4_18;a4_19; }
}

// connect the gates
connect
{
  in1[or4_2]=out[a4_16];
  in2[or4_2]=out[a4_17];
  in3[or4_2]=out[a3_12];
  in4[or4_2]=out[a2_7];

  in1[nor4_6]=out[a4_18];
  in2[nor4_6]=out[a4_19];
  in3[nor4_6]=out[a3_13];
  in4[nor4_6]=out[a2_8];

  in1[nor3_5]=out[a3_14];
  in2[nor3_5]=out[a3_15];
  in3[nor3_5]=out[a2_9];

  in1[nor2_4]=out[a2_10];
  in2[nor2_4]=out[a2_11];

  in1[or4_3]=in1[a2_7];
  in2[or4_3]=in1[a2_8];
  in3[or4_3]=in1[a2_9];
  in4[or4_3]=in1[a2_11];

  in1[a4_16]=in2[a2_7];
  in2[a4_16]=in2[a2_8];
  in3[a4_16]=in2[a2_9];
  in4[a4_16]=in2[a2_11];
```

```
  in1[a4_17]=in1[a2_9];
  in2[a4_17]=in2[a2_7];
  in3[a4_17]=in2[a2_8];
  in4[a4_17]=in2[a2_9];

  in1[a3_12]=in1[a2_8];
  in2[a3_12]=in2[a2_7];
  in3[a3_12]=in2[a2_8];

  in1[a4_18]=in2[a2_8];
  in2[a4_18]=in2[a2_9];
  in3[a4_18]=in2[a2_11];
  in4[a4_18]=out[bn1];

  in1[a4_19]=in1[a2_11];
  in2[a4_19]=in2[a2_8];
  in3[a4_19]=in2[a2_9];
  in4[a4_19]=in2[a2_11];

  in1[a3_13]=in1[a2_9];
  in2[a3_13]=in2[a2_8];
  in3[a3_13]=in2[a2_9];

  in1[a3_14]=in2[a2_9];
  in2[a3_14]=in2[a2_11];
  in3[a3_14]=out[bn1];

  in1[a3_15]=in1[a2_11];
  in2[a3_15]=in2[a2_9];
  in3[a3_15]=in2[a2_11];

  in1[a2_10]=in2[a2_11];
  in2[a2_10]=out[bn1];
}

// describe the observation
observe
{
  // inputs
  in1[a2_7]=0;
  in2[a2_7]=1;

  in1[a2_8]=0;
  in2[a2_8]=0;

  in1[a2_9]=1;
  in2[a2_9]=1;

  in1[a2_11]=1;
  in2[a2_11]=0;

  in[bn1]=0;

  // outputs
  out[or4_3]=1;
  out[or4_2]=1;
  out[nor4_6]=0;
  out[nor3_5]=0;
  out[nor2_4]=1;
}

// assign the necessary parameters for DSDC to run
assign
{
  // error probability distribution
  errorRate
  {
    bn1=0.1;
    or4_2=0.2;
    or4_3=0.2;
    nor2_4=0.7;
    nor3_5=0.1;
    nor4_6=0.8;
    a2_7=0.1;
    a2_8=0.2;
    a2_9=0.1;
    a2_10=0.3;
    a2_11=0.1;
    a3_12=0.2;
```

```
   a3_13=0.3;
   a3_14=0.1;
   a3_15=0.4;
   a4_16=0.3;
   a4_17=0.1;
   a4_18=0.8;
   a4_19=0.1;
  }

  // specify the error type for fault pattern derivation:
  //   0: the abnormal output is equal to the inverse of the normal output
  //   1: stuck at 0
  //   2: stuck at 1
  errorType
  {
   bn1=0;
   or4_2=0;
   or4_3=0;
   nor2_4=0;
   nor3_5=0;
   nor4_6=0;
   a2_7=0;
   a2_8=0;
   a2_9=0;
   a2_10=0;
   a2_11=0;
   a3_12=0;
   a3_13=0;
   a3_14=0;
   a3_15=0;
   a4_16=0;
   a4_17=0;
   a4_18=0;
   a4_19=0;
  }

  population=40;
  initialization=0;     // 0: fixed random, 1: random
  initNumFaults=2;
  credit=0.00001;
  c=1.5;
  min=10.0;
  cMin=0.0;
  parentLen=5;

  // operators fitness distribution
  operators
  {
    crossover=0.6;
    mutation=0.1;
    fine_tune=0.3;
    reevalutation=0.0;
  }
}
```

B.4 Circuit c04_16: A 3 to 8 Demultiplexer (74138) with 16 Gates

```
// Circuit 04: 3 to 8 Demultiplexer (74138) p.93
// Total: 16 components

// define the gates used in the circuit
define
{
  buffer_not { out=~in; }
  nand3 { out=~in1*~in2*~in3; }
  nand4 { out=~(in1*in2*in3*in4); }
}

// declare all gates
declare
{
  buffer_not { bn9;bn10;bn11;bn12;bn13;bn14;bn15; }
  nand3 { na3_16; }
  nand4 { na4_1;na4_2;na4_3;na4_4;na4_5;na4_6;na4_7;na4_8; }
}

// connect the gates
connect
{
```

```
  // 8 nand4 gates first
  in1[na4_1]=out[bn12];
  in2[na4_1]=out[bn13];
  in3[na4_1]=out[bn14];
  in4[na4_1]=out[na3_16];

  in1[na4_2]=out[bn9];
  in2[na4_2]=out[bn13];
  in3[na4_2]=out[bn14];
  in4[na4_2]=out[na3_16];

  in1[na4_3]=out[bn12];
  in2[na4_3]=out[bn10];
  in3[na4_3]=out[bn14];
  in4[na4_3]=out[na3_16];

  in1[na4_4]=out[bn9];
  in2[na4_4]=out[bn10];
  in3[na4_4]=out[bn14];
  in4[na4_4]=out[na3_16];

  in1[na4_5]=out[bn12];
  in2[na4_5]=out[bn13];
  in3[na4_5]=out[bn11];
  in4[na4_5]=out[na3_16];

  in1[na4_6]=out[bn13];
  in2[na4_6]=out[bn11];
  in3[na4_6]=out[bn9];
  in4[na4_6]=out[na3_16];

  in1[na4_7]=out[bn12];
  in2[na4_7]=out[bn11];
  in3[na4_7]=out[bn10];
  in4[na4_7]=out[na3_16];

  in1[na4_8]=out[na3_16];
  in2[na4_8]=out[bn10];
  in3[na4_8]=out[bn9];
  in4[na4_8]=out[bn11];

  // 1 nand3 gate
  in1[na3_16]=out[bn15];

  // 3 buffer-not gates
  in[bn9]=out[bn12];
  in[bn10]=out[bn13];
  in[bn11]=out[bn14];
}

// describe the observation
observe
{
  // Inputs
  // ***** enable
  in[bn15]=1;
  in2[na3_16]=0;
  in3[na3_16]=1;

  // ***** select
  in[bn12]=0;
  in[bn13]=1;
  in[bn14]=0;

  // Outputs
  out[na4_1]=1;
  out[na4_2]=1;
  out[na4_3]=1;
  out[na4_4]=1;
  out[na4_5]=1;
  out[na4_6]=1;
  out[na4_7]=0;
  out[na4_8]=1;
}


// assign the necessary parameters for DSDC to run
assign
{
  // error probability distribution
```

```
  errorRate
  {
   na4_1=0.1;
   na4_2=0.1;
   na4_3=0.8;
   na4_4=0.1;
   na4_5=0.2;
   na4_6=0.3;
   na4_7=0.7;
   na4_8=0.1;
   bn9=0.1;
   bn10=0.8;
   bn11=0.1;
   bn12=0.2;
   bn13=0.1;
   bn14=0.3;
   bn15=0.1;
   na3_16=0.8;
  }

  // specify the error type for fault pattern derivation:
  //   0: the abnormal output is equal to the inverse of the normal output
  //   1: stuck at 0
  //   2: stuck at 1
  errorType
  {
   na4_1=0;
   na4_2=0;
   na4_3=0;
   na4_4=0;
   na4_5=0;
   na4_6=0;
   na4_7=0;
   na4_8=0;
   bn9=0;
   bn10=0;
   bn11=0;
   bn12=0;
   bn13=0;
   bn14=0;
   bn15=0;
   na3_16=0;
  }

  population=40;
  initialization=0;     // 0: fixed random, 1: random
  initNumFaults=2;
  credit=0.00001;
  c=1.5;
  min=10.0;
  cMin=0.0;
  parentLen=5;

  // operators fitness distribution
  operators
  {
    crossover=0.6;
    mutation=0.1;
    fine_tune=0.3;
    reevalutation=0.0;
  }
}
```

B.5 Circuit c05_16: A Dual 4 to 1 Data Selectors (74153) with 16 Gates

```
// Circuit 05: Dual 4 to 1 Data Selectors (74153) p.100
// Total: 16 components

// define the gates used in the circuit
define
{
  buffer_not { out=~in; }
  or4 { out=(in1+in2+in3+in4); }
  and4 { out=(in1*in2*in3*in4); }
}

// declare all gates
```

```
declare
{
  or4 { or4_1;or4_2; }
  and4 { and4_3;and4_4;and4_5;and4_6;and4_7;and4_8;and4_9;and4_10; }
  buffer_not { bn11;bn12;bn13;bn14;bn15;bn16; }
}

// connect the gates
connect
{
  // 2 or4 gates first
  in1[or4_1]=out[and4_3];
  in2[or4_1]=out[and4_4];
  in3[or4_1]=out[and4_5];
  in4[or4_1]=out[and4_6];

  in1[or4_2]=out[and4_7];
  in2[or4_2]=out[and4_8];
  in3[or4_2]=out[and4_9];
  in4[or4_2]=out[and4_10];

  // 8 and4 gates
  in1[and4_3]=out[bn13];
  in2[and4_3]=out[bn14];
  in3[and4_3]=out[bn15];

  in1[and4_4]=out[bn13];
  in2[and4_4]=out[bn14];
  in3[and4_4]=out[bn12];

  in1[and4_5]=out[bn13];
  in2[and4_5]=out[bn11];
  in3[and4_5]=out[bn15];

  in1[and4_6]=out[bn13];
  in2[and4_6]=out[bn11];
  in3[and4_6]=out[bn12];

  in2[and4_7]=out[bn14];
  in3[and4_7]=out[bn15];
  in4[and4_7]=out[bn16];

  in2[and4_8]=out[bn14];
  in3[and4_8]=out[bn12];
  in4[and4_8]=out[bn16];

  in2[and4_9]=out[bn11];
  in3[and4_9]=out[bn15];
  in4[and4_9]=out[bn16];

  in2[and4_10]=out[bn11];
  in3[and4_10]=out[bn12];
  in4[and4_10]=out[bn16];

  // 2 buffer-not gates
  in[bn11]=out[bn14];
  in[bn12]=out[bn15];
}

// describe the observation
observe
{
  // Inputs
  in[bn13]=1;

  // ***** data1
  in4[and4_3]=1;
  in4[and4_4]=0;
  in4[and4_5]=1;
  in4[and4_6]=1;

  // ***** select
  in[bn14]=1;
  in[bn15]=0;

  // ***** data2
  in1[and4_7]=1;
  in1[and4_8]=0;
  in1[and4_9]=0;
  in1[and4_10]=0;
```

```
  in[bn16]=1;

  // Outputs
  out[or4_1]=0;
  out[or4_2]=0;
}

// assign the necessary parameters for DSDC to run
assign
{
  // error probability distribution
  errorRate
  {
    or4_1=0.3;
    or4_2=0.1;
    and4_3=0.1;
    and4_4=0.2;
    and4_5=0.1;
    and4_6=0.7;
    and4_7=0.1;
    and4_8=0.1;
    and4_9=0.2;
    and4_10=0.1;
    bn11=0.8;
    bn12=0.1;
    bn13=0.3;
    bn14=0.1;
    bn15=0.2;
    bn16=0.3;
  }

  // specify the error type for fault pattern derivation:
  //   0: the abnormal output is equal to the inverse of the normal output
  //   1: stuck at 0
  //   2: stuck at 1
  errorType
  {
    or4_1=0;
    or4_2=0;
    and4_3=0;
    and4_4=0;
    and4_5=0;
    and4_6=0;
    and4_7=0;
    and4_8=0;
    and4_9=0;
    and4_10=0;
    bn11=0;
    bn12=0;
    bn13=0;
    bn14=0;
    bn15=0;
    bn16=0;
  }

  population=40;
  initialization=0;     // 0: fixed random, 1: random
  initNumFaults=2;
  credit=0.00001;
  c=1.5;
  min=10.0;
  cMin=0.0;
  parentLen=5;

  // operators fitness distribution
  operators
  {
    crossover=0.6;
    mutation=0.1;
    fine_tune=0.3;
    reevalutation=0.0;
  }
}
```

Bibliography

1. D. B. Armstrong, On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets, *IEEE Transactions on Electronic Computers* **15** (1966), 66 - 73.
2. D. A. Berry and B. Fristedt, *Bandit Problems: Sequential Allocation of Experiments*, Chapman and Hall, 1985.
3. L. B. Booker, D. E. Goldberg and J. H. Holland, Classifier Systems and Genetic Algorithms, *Artificial Intelligence* **40** (1989) 235 - 282.
4. B. G. Buchanan and E. H. Shortliffe (Eds.), *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, Reading, MA, 1984.
5. M. Buro and H. K. Büning, *Report on a SAT Competition*, Universität Paderborn, 1992.
6. C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.
7. H. C. A. Dale, Fault-finding in Electronic Equipment, *Ergonomics* **1** (1957) 356 - 385.
8. L. Davis, Adapting Operator Probabilities in Genetic Algorithms, *Proceedings of the Third International Conference on Genetic Algorithms* (1989), 61 - 69, Morgan Kaufmann.
9. L. Davis *et al.*, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.
10. M. Davis and H. Putnam, A Computing Procedure for Quantification Theory, *Journal of ACM* **7** (3) (1960), 201 - 215.
11. R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, M. Shirley and S. Polit, Diagnosis based on Description of Structure and Function, *Proceedings of the Second National Conference on Artificial Intelligence* (1982), 137 - 142, Morgan Kaufmann.
12. R. Davis, Diagnostic Reasoning Based on Structure and Behavior, *Artificial Intelligence* **24** (1984) 347 - 410.
13. K. A. De Jong, *Analysis of the Behavior of a Class of Genetic Adaptive Systems*, PhD Thesis, Department of Computer and Communication Sciences, University of Michigan at Ann Arbor, 1975.
14. J. de Kleer, Local Methods for Localizing Faults in Electronic Circuits, *MIT AI Memo* **394**, Cambridge, MA, 1976.
15. J. de Kleer and B. C. Williams, Diagnosing Multiple Faults, *Artificial Intelligence* **32** (1987) 97 - 130.
16. J. de Kleer and B. C. Williams, Diagnosis with Behavioral Modes, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (1989), 1324 - 1330, Morgan Kaufmann.
17. K. D. Forbus and J. de Kleer, *Building Problem Solvers*, the MIT Press, 1993.
18. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Bell Laboratories, N.J., 1979.
19. M. R. Genesereth, The Use of Design Descriptions in Automated Diagnosis, *Artificial Intelligence* **24** (1984) 411 - 436.

20. P. Goel, An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits, *IEEE Transactions on Electronic Computers* **30** (1981), 215 - 222.

21. D. E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.

22. J. J. Grefenstette, Optimization of Control Parameters for Genetic Algorithms, *IEEE Transactions on Systems, Man, and Cybernetics* **16** (1) (1986) 122 - 128.

23. R. Greiner, B. A. Smith and R. W. Wilkerson, A Correction to the Algorithm in Reiter's Theory of Diagnosis, *Artificial Intelligence* **41** (1989/90) 79-88.

24. W. Hamscher, *Model-based Troubleshooting of Digital Systems*, PhD Thesis, MIT Technical Report AI-TR-1074, 1988.

25. W. Hamscher, Temporally Coarse Representation of Behavior for Model-based Troubleshooting of Digital Circuits, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (1989), 887 - 893, Morgan Kaufmann.

26. R. T. Hartley, CRIB: Computer Fault-finding through Knowledge Engineering, *Computer* (1984), 76 - 83.

27. J. P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley, 1993.

28. J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

29. A. Hou, A Theory of Measurement in Diagnosis from First Principles, *Artificial Intelligence* **65** (1994) 281 - 328.

30. R. M. Hunt and W. B. Rouse, A Fuzzy Rule-based Model of Human Problem Solving, *IEEE Transactions on Systems, Man, and Cybernetics* **14** (1) (1984) 112 - 120.

31. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, second edition, 1978.

32. T. J. Laffey, W. A. Perkins, and T. A. Nguyen, Reasoning about Fault Diagnosis with LES, *IEEE Expert* (1986), 13 - 20.

33. P. K. Lala, *Fault Tolerant and Fault Testable Hardware Design*, Prentice-Hall, 1985.

34. J. McCarthy, Applications of Circumscriptions to Formalizing Common-sense Knowledge, *Artificial Intelligence* **28** (1986) 89 - 116.

35. B. Moszkowski, A Temporal Logic for Multilevel Reasoning about Hardware, *IEEE Computer* **18** (2) (1985) 10 - 19.

36. S. Mourad and E. J. McCluskey, Fault Models, in F. Lombardi and M. sami, editors, *Testing and Diagnosis of VLSI and ULSI*, 49 - 68, Kluwer Academic Publishers, 1987.

37. W. R. Nelson, REACTOR: An Expert System for Diagnosis and Treatment of Nuclear Reactor Accidents, *Proceedings of the Second National Conference on Artificial Intelligence* (1982), 296 - 301, Morgan Kaufmann.

38. J. Rasmussen and A. Jensen, Mental Procedures in Real-life Tasks: A Case Study of Electronic Trouble Shooting, *Ergonomics* **17** (3) (1974) 293 - 307.

39. R. Reiter, A Theory of Diagnosis from First Principles, *Artificial Intelligence* **32** (1987) 57 - 95.

40. J. P. Roth, Diagnosis of Automata Failures: A Calculus and a Method, *IBM Journal of Research and Development* **10** (1966), 278 - 291.

41. W. B. Rouse, Human Problem Solving Performance in a Fault Diagnosis Task, *IEEE Transactions on Systems, Man, and Cybernetics* **8** (4) (1978) 258 - 271.

42. W. B. Rouse, A Model of Human Decision Making in a Fault Diagnosis Task, *IEEE*

Transactions on Systems, Man, and Cybernetics **8** (5) (1978) 357 - 361.

43. W. B. Rouse, Problem Solving Performance of Maintenance Trainees in a Fault Diagnosis Task, *Human Factors* **21** (2) (1979) 195 - 203.
44. W. B. Rouse, Problem Solving Performance of First Semester Maintenance Trainees in Two Fault Diagnosis Task, *Human Factors* **21** (5) (1979) 611 - 618.
45. W. B. Rouse, S. H. Rouse, and S. J. Pellegrino, A Rule-based Model of Human Problem Solving Performance in Fault Diagnosis Tasks, *IEEE Transactions on Systems, Man, and Cybernetics* **10** (7) (1980) 366 - 376.
46. W. B. Rouse, Models of Natural Intelligence in Fault Diagnosis Tasks: Implications for Training and Aiding of Maintenance Personnel, *Proceedings of Joint Services Workshop on Artificial Intelligence in Maintenance* (1984), 193 - 212.
47. S. Russell and P. Norvig, "Logical Reasoning Systems," *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 1995.
48. J. D. Schaffer, R. A. Caruana, L. J. Eshelman and R. Das, A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization, *Proceedings of the Third International Conference on Genetic Algorithms* (1989), 51 - 60, Morgan Kaufmann.
49. F. F. Sellers, M. Y. Hsiao, and C. L. Bearnson, Analyzing Errors with the Boolean Difference, *IEEE Transactions on Computers* **17** (1968), 676 - 683.
50. E. H. Shortliffe, *Computer-based Medical Consultation: MYCIN*, North-Holland, New York, 1976.
51. G. Syswerda, Uniform Crossover in Genetic Algorithms, *Proceedings of the Third International Conference on Genetic Algorithms* (1989), 2 - 9, Morgan Kaufmann.
52. G. Syswerda, A Study of Reproduction in Generational and Steady-State Genetic Algorithms, *Foundations of Genetic Algorithms*, 94 - 101, Morgan Kaufmann, 1991.
53. H. Zhang and M. Stickel, *Implementing the Davis-Putnam Algorithm by Tries*, typescript, University of Iowa, 1994