October 1998

UILU-ENG-98-2225
DAC-69

University of Illinois at Urbana-Champaign

# Multiple Design Error Diagnosis and Correction in Digital VLSI Circuits

Andreas G. Veneris

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

| | | |
|---|---|---|
| 1a. REPORT SECURITY CLASSIFICATION Unclassified | | 1b. RESTRICTIVE MARKINGS None |
| 2a. SECURITY CLASSIFICATION AUTHORITY | | 3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-98-2225 (DAC 69) | | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois | 6b. OFFICE SYMBOL (If applicable) N/A | 7a. NAME OF MONITORING ORGANIZATION Office of Naval Research |
| 6c. ADDRESS (City, State, and ZIP Code) 1308 W Main St Urbana, IL 61801 | | 7b. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program | 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-96-1-0129 |
| 8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217 | | 10. SOURCE OF FUNDING NUMBERS |

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO. |
|---|---|---|---|
| | | | |

11. TITLE (Include Security Classification)
Multiple Design Error Diagnosis and Correction in Digital VLSI Circuits

12. PERSONAL AUTHOR(S) Veneris, Andreas G.

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Technical | FROM ____ TO ____ | 98 Sep 22 | 131 |

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

| FIELD | GROUP | SUB-GROUP |
|---|---|---|
| | | |
| | | |

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Design, error, diagnosis, VLSI, digital, circuit

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

With the increase of circuit size and complexity, logic design errors can occur. Logic design errors are functional mismatches between the specification and gate-level implementation. Once a verification tool has found that the design is erroneous, logic debugging must be performed. The research presented in this thesis provides a methodology for *multiple design error diagnosis and correction.*

To diagnose an erroneous design, two algorithms based on test-vector simulation are presented. The first algorithm is exhaustive on the error space as it exhaustively enumerates the set of all possible error lines and returns the lines of this set that a correction can be applied and rectify the design. The proposed approach exhibits good run-time performance when the number of design errors is less than or equal to two.

The second diagnosis approach, uses the results of a test-vector simulation procedure to build a graph. Different operations on the graph allow us to explore the error space without performing an explicit enumeration of all error candidates. This makes the method run-time and space efficient for designs corrupted with a larger number of errors.

To correct the design, we propose two techniques, one based on test-vector simulation, and one based on Boolean function manipulation techniques. Both correction approaches are based on a design error dictionary which is an extension of the one proposed by Abadir et al. [2].

Our experimental results show that our algorithms have good error resolution and run-time performance as they are able to rectify designs with one, two and three errors within minutes of CPU time. In addition, our experiments suggest that diagnosis and correction of multiple design errors with input test-vector simulation is an attractive alternative to symbolic techniques. This makes our test-vector simulation based methods applicable to designs where a symbolic representation might not be available.

| | | |
|---|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION Unclassified | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) | 22c. OFFICE SYMBOL |

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

# MULTIPLE DESIGN ERROR DIAGNOSIS AND CORRECTION IN DIGITAL VLSI CIRCUITS

BY

ANDREAS G. VENERIS

Diploma, University of Patras, 1991
M.S., University of Southern California, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

MULTIPLE DESIGN ERROR DIAGNOSIS
AND CORRECTION IN DIGITAL VLSI CIRCUITS

Andreas G. Veneris, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1998
Ibrahim N. Hajj, Advisor

With the increase of circuit size and complexity, logic design errors can occur. Logic design errors are functional mismatches between the specification and gate–level implementation. Once a verification tool has found that the design is erroneous, logic debugging must be performed. The research presented in this thesis provides a methodology for *multiple design error diagnosis and correction.*

To diagnose an erroneous design, two algorithms based on test–vector simulation are presented. The first algorithm is exhaustive on the error space as it exhaustively enumerates the set of all possible error lines and returns the lines of this set that a correction can be applied and rectify the design. The proposed approach exhibits good run–time performance when the number of design errors is less than or equal to two.

The second diagnosis approach, uses the results of a test–vector simulation procedure to build a graph. Different operations on the graph allow us to explore the error space without performing an explicit enumeration of all error candidates. This makes the method run–time and space efficient for designs corrupted with a larger number of errors.

To correct the design, we propose two techniques, one based on test–vector simulation, and one based on Boolean function manipulation techniques. Both correction approaches are based on a design error dictionary which is an extension of the one proposed by Abadir et al. [2].

Our experimental results show that our algorithms have good error resolution and run–time performance as they are able to rectify designs with one, two and three errors within minutes of CPU time. In addition, our experiments suggest that diagnosis and correction of multiple design errors with input test–vector simulation is an attractive alternative to symbolic techniques. This makes our test–vector simulation based methods applicable to designs where a symbolic representation might not be available.

To my parents for their unconditional
love, faith, patience, and support

# ACKNOWLEDGMENTS

I would like to begin by expressing my gratitude to my advisor, Professor Ibrahim N. Hajj, for giving me the freedom, allowance, guidance and support throughout the course of my graduate years.

I would also like to thank the members of my committee, Professors Farid Najm, Steve Kang, William Kubitz, C.L.Liu, and Saburo Muroga for their valuable comments and input in my research. A special thanks to Professors William Kubitz and C.L.Liu who served as the chairpersons of my final and preliminary doctoral exams, respectively.

I would also like to take the opportunity to thank Professors Dennis Mickunas, Michael Faiman, Lefteris Kirousis, Constantine Polychronopoulos, Edward Reingold, Naresh Shanbhag and Douglas West for treating me with kindness and sharing their experience with me. A special acknowledgment goes to Ms. Carol Genzel from the Coordinated Science Laboratory and the people at the graduate office of the Digital Computer Laboratory Ms. Barbara Cicone, Ms. Julie Legg, and Ms. Felice Long, for making my life easy while I was making their lives hard.

I owe much to my friends in Coordinated Science Laboratory and Digital Computer Laboratory for understanding and sharing the blues and greys of my daily life since 1993: Nikos Bellas, Sudhakar Bobba, Howard Chen, Georgios Dimitriou, Shashank Goel, Tong Li, Raj Panda, Sumant Ramprasad, Mallikarjun Shankar, Prashant Saxena, and Georgios Stamoulis. I would also like to acknowledge Shi-Yu Huang from National Semiconductor Corp. and Chang-Chao Hsieh from Prof. Muroga's group for their interest comments throughout my research. Many thanks also go to Professor Martha Escobar, Wes Groves, Michael Johnson, Professor Rodanthi Kitridou, and Luke Wroblweski for being close to me whenever I needed them.

Lastly, but most importantly, I am forever indebted to my parents, Georgios and Irini, whose unselfish love forged the foundation of my life.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# Introduction

## 1.1 Problems Addressed

The process of designing VLSI circuits involves many stages of **verification** and **debugging**. Circuit verification is the process of checking if a design meets its specification. Once a design has been proven to be erroneous by a verification tool, design debugging occurs so that a correct version that satisfies the desired specifications is obtained. With the increased emphasis on product quality, reduction of manufacturing cost and improved yield for semiconductor circuits, automated design verification and debugging remains a significant research area.

Although existing research has focused extensively on the design verification issue (logic verification [2] [5] [8] [29] [30] [31] [34], timing verification [24], power estimation [46], layout verification [18] [49]) little research has been carried on logic debugging which is still, in most cases, carried manually by the designer. It has been reported that more than half of the total design effort of a high–performance microprocessor is devoted to the verification and debugging process [30]. Therefore, with the increase of both circuit size and circuit complexity today, there is an urgent need to develop automated methods that perform debugging. In this thesis, we describe a set of logic debugging tools that rectify a design at the logic level that has been already proven to be erroneous by a verification tool.

During the design cycle of a VLSI digital circuit, functional mismatches between the specification and the gate–level implementation often occur. These mismatches usually happen when the designer has to manually interfere with the synthesis process and modify the netlists in order to achieve different optimization goals. Errors in high level synthesis can also be sources of errors in a gate–level implementation. Finally, software bugs in automated synthesis/optimization tools can also be sources of functional mismatches. These functional mismatches between the specification and the gate–level implementation are called **design errors** hereafter.

Experimental data has shown that the nature of the design errors usually involves the functional misbehavior of some gate element(s) or some wire interconnection error(s) [1] [2]. The average number of design errors that usually occur depends on the amount of manual resynthesis performed and it has been experimentally observed to be less than or equal to 2 [1]. In this thesis, we adopt a design error model which is an extension of the one presented in [2]. This model includes eight commonly encountered design errors. An experimental study carried in [1], has shown that the model of [2] covers 97.8% of all design errors that usually occur during a manual resynthesis procedure.

Once a verification tool has found a functional mismatch between the design and the specification, **Design Error Diagnosis and Correction (DEDC)** needs to be performed. The *objective* of diagnosis is to identify parts of the circuit that might contain an error. The *quality* of diagnosis is determined by its ability to narrow down the potential error space, that is, its resolution to identify parts of the circuit that contain potential erroneous signals.

Once the error space has been narrowed down, rectification is performed. The *goal* of correction is to suggest the appropriate modifications on the netlist. Modifications are usually selected from a list of possible modifications also known as **design error model**. These modifications will be able to rectify the design and make it functionally equivalent to the specification. Since verification tools cannot give any information on the location and type of error(s), it is necessary to develop automatic methods that solve the problem of Design Error Diagnosis and Correction efficiently.

A problem closely related to Design Error Diagnosis and Correction is the **Engineering Change (EC)** one. In a typical VLSI synthesis process, specifications often change. Since tools used for synthesis and optimization [44] tend to find a minimal representation of the requested function, engineering changes on the original specification may require large changes in the existing gate–level implementation if a conventional re-synthesis procedure is used. This is undesirable since the engineer might have already invested a lot of effort in synthesizing the existing design. Therefore, in order to preserve most of the previous engineering effort and re-use as much of the existing design as possible, efficient algorithms to attack the EC problem need to be developed so that rectification is performed with as little modifications as possible.

Depending upon the information available, there are two different versions of the Engineering Change problem. In the first version, a naming equivalence between some signals of the new and old specification and the existing gate–level implementation exists. For example, the old and new specification are both in a netlist format before technology decomposition and the gate–level implementation under rectification is the netlist after decomposition. Research in this area usually uses this fact of the aforementioned signal naming equivalence and reuses large parts of the existing design.

In the second version of the EC problem, the specification acts like a "black box", that is, it can only provide the correct output responses given some input stimulus. For example, the specification might be available in a higher level of abstraction model, such as in a register–transfer level (RTL) format coded in some hardware–description language (Verilog or VHDL). In such a case, the aforementioned naming correspondence between the specification and implementation does not exist.

The problem of Design Error Diagnosis and Correction can be also viewed as an *instance* of the second EC version [39]. This is achieved if we view the existing netlist that implements the old specification as the erroneous circuit and the new specification as the desired circuit. Nevertheless, EC is an inherently more difficult problem than DEDC as we cannot necessarily expect that a few modifications can always provide a solution as it is usually happens for DEDC.

The importance of this difference comes from the fact that the error space during diagnosis usually grows exponentially with the amount of erroneous signals [56]. More specifically, for a brute force diagnosis method, the error space is upper bounded by

$$(\#\ of\ circuit\ lines)^{(\#\ of\ erroneous\ lines)} \tag{1.1}$$

while the error correction space is *at least* as big as the equation above depending on the design error model that is used. It is straightforward to see that for even small circuits corrupted with multiple errors, a naive DEDC approach might fail.

In this thesis we present a methodology for combinational circuit [1] design error diagnosis and rectification when a number of modifications on some lines of the erroneous circuit are sufficient to correct an erroneous design. We also compare the quality of test–vector simulation and BDDs [12] for the DEDC problem.

## 1.2 Problem Formulation

### 1.2.1 Preliminaries

As explained earlier, the **input** to the DEDC problem is a functional specification $F_C$ and a netlist of an erroneous gate–level implementation $G_C$. The specification $F_C$ acts as a "black–box" as it can only provide the correct primary output responses given some values at the primary inputs. $F_C$ can be given in terms of a truth–table, cubes etc. Without loss of generality, we assume that both $F_C$ and $G_C$ are combinational circuits, or they are both synchronous sequential circuits with the same state variables and the same state assignments. In this work, we examine incorrect combinational gate–level descriptions $G_C$ with `NOT`, `BUFFER`, `AND`, `NAND`, `OR` and `NOR` gates [2]. During the execution of our algorithm, we introduce one buffer for every fan-out line of a branch.

The **functional description** $F_C$ of a circuit $C$ with $n$ primary inputs, $PI_{F_C} = \{PI_1, PI_2, \ldots, PI_n\}$, and $m$ primary outputs, $PO_{F_C}(PI) = \{PO_1(PI), PO_2(PI), \ldots, PO_m(PI)\}$, is a function from a set of input $n$–tuple values drawn from universe $B = \{0,1\}$ to a set of output $m$–tuple values drawn from $B$. In other words, the functional

[1] As explained in [2], a synchronous sequential circuit can be viewed as a combinational circuit if the inputs and outputs of the flip–flops are considered as pseudo–primary inputs and outputs, respectively. A detailed discussion on this issue can be found in Chapter 6.

[2] If `XOR` gates are present in the circuit, these gates are substituted.

description $F_C$ of a circuit $C$ is a function that returns the *correct* values at each primary output of the circuit for all values at the primary inputs of $C$. A **gate–level description** $G_C$ of a circuit $C$ designed to realize $F_C$, is defined along the same lines. $G_C$ is *incorrect* when for the same set of input $n$–tuples, the return set of $G_C$ is a set of $m$–tuples *different* than the one returned by $F_C$. For the DEDC problem, the functional description acts as a "black box" as it can only provide the correct primary output responses given some primary input stimulus. The only netlist available is the one of the erroneous $G_C$.

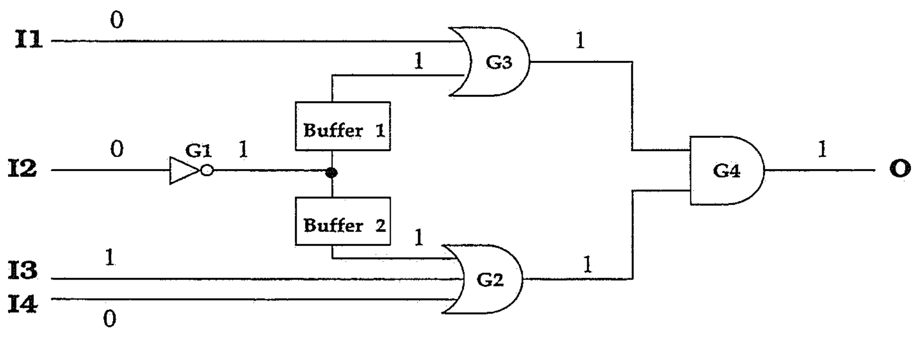
For the purpose of logic verification in our technique, formal techniques [12] [34] or parallel [66] test–vector simulation [1] [2] can be used. In any case, a set $V_{act}$ of input test vectors is obtained each of which **activates the inconsistencies** of the incorrect design, that is, each such vector produces a different response at the primary outputs of $F_C$ and $G_C$. In our experiments, we compile $V_{act}$ by simulating vectors for stuck-at faults and random test–vectors. The size of $V_{act}$ in our experiments is usually less than 100 input vectors. In Section 4.3.3 we explain how we can use symbolic techniques to derive vectors for $V_{act}$.

We say that a line $l$, fan–in to an `AND` or `NAND` (`OR` or `NOR`) gate, has **controlling value** for input vector $v$ if the value of $l$ is 0 (1). If $l$ drives a `NOT` or a `BUFFER` it always has controlling value. A line whose value changes during simulation under the presense of some fault(s) is called a **sensitized line** and a path of sensitized lines is called a **sensitized path** [10].

**Example 1** *Fig. 1.1 shows an example of a design specification and its respective correct gate–level implementation. The circuit has four primary inputs, namely $I_1, I_2, I_3, I_4$ and* *one primary output $O$. The implementation is simulated for input vector* $(0,0,1,0)$ *and lines* $I_2, G_1, Buffer_1, Buffer_2, I_3$ *have controlling values.*

$$O = I_1 I_3 + I_1 I_4 + \overline{I_2}$$

(a) Functional Specification



(b) Gate Level Implementation

**Figure 1.1** Example of Input Specification and Implementation

The following terminology is used during error correction. Underlined values denote vector values.

Let Boolean functions $f$ and $g$, defined over the $n$–input vector space $\underline{X} = \{x_1, x_2, \ldots, x_n\}$. We say that $f(\underline{X}) \leq g(\underline{X})$ if and only if $f(\underline{X})\overline{g(\underline{X})} = \underline{0}$. Following this terminology, we say that function $h(X)$ is in the interval of $[g(\underline{X}), f(\underline{X})]$, denoted as $h(\underline{X}) \in [g(\underline{X}), f(\underline{X})]$, if and only if $g(\underline{X}) \leq h(\underline{X}) \leq f(\underline{X})$. It can be proved [11] that $h(\underline{X}) \in [g(\underline{X}), f(\underline{X})]$ if and only if $g(\underline{X})\overline{h(\underline{X})} + h(\underline{X})\overline{f(\underline{X})} = \underline{0}$.

### 1.2.2 Design Error Model

In this thesis, we use an error model which is an *extension* of the one proposed in [2]. An experimental study described in [1] has shown that design errors as defined in [2] cover 97.8% of all design errors that usually occur during a manual resynthesis procedure.

Fig. 1.2 contains the simple logic design error model of [2]. Boxes with indexed letters stand for simple `AND`, `OR`, `NAND`, `NOR` gates. This model includes eight commonly encountered design errors [1]. On a theoretical basis, Abadir et al. [2] proved that a complete set of test vectors for stuck-at faults for the erroneous circuit guarantees to detect the majority of the design errors of their model (types a, b, c, d, e, and f) and has a very good chance of detecting the remaining ones (types g, h, i, and j). This result is experimentally confirmed in [5]. In the same work [5] the authors develop a test vector generator for design errors.

We now proceed to the definition of our design modification model. Throughout this thesis, the words **correction** and **modification** are used interchangeably.

**Definition 1** *With respect to Fig. 1.2, we define a* **modification** *to be one of the following three types:*

- **Wrong Gate** *, namely types a, b, c, e, f, or i.*
- **Wrong Wire** *, namely types d, g, or h.*
- **Wrong Gate/Wrong Wire***, an occurrence of both previous errors on the gate driving a single line.*

**Definition 2** *We define an $N-$***error line tuple** *$L = \{l_1, l_2, \ldots, l_N\}$, $N \geq 1$, to be a set of $N$ distinct circuit lines. We also define an $N$-correction tuple $Corr = \{c_1, c_2, \ldots, c_N\}$, $N \geq$*

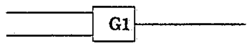
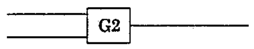
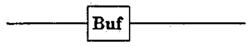
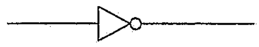
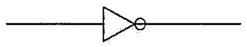
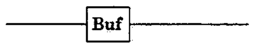
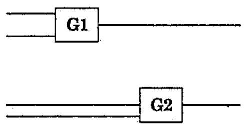
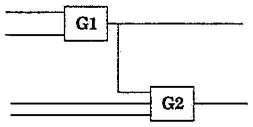
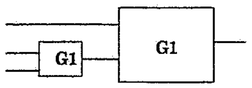
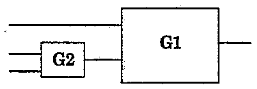
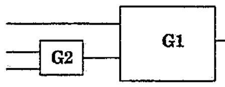
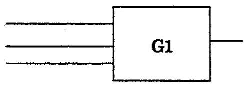
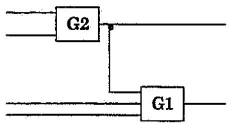
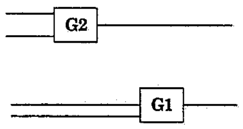
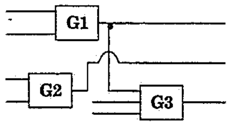
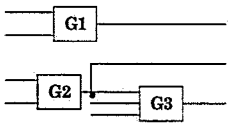
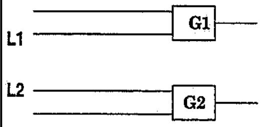
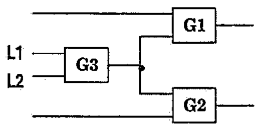
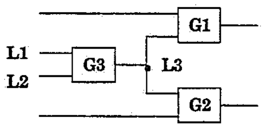
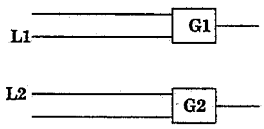
| MODIFICATION | CORRECT | INCORRECT |
|---|---|---|
| Type a<br>Single Gate Replacement |  |  |
| Type b<br>Extra Inverter |  |  |
| Type c<br>Missing Inverter |  |  |
| Type d<br>Extra Wire |  |  |
| Type e<br>Extra Gate |  |  |
| Type f<br>Missing Gate |  |  |
| Type g<br>Missing Wire |  |  |
| Type h<br>Incorrectly Placed Input Wire |  |  |
| Type i<br>Extra Gate<br>(Complex Case) |  |  |
| Type j<br>Missing Gate<br>(Complex Case) |  |  |

**Figure 1.2** Abadir et al. design error model

*1, to be a set of $N$ modifications from Def. 1. We say that $Corr$* **is applied on** *$L$ when we replace the existing function of $G_C$ on the gate driving $l_i$ with the function performed by $c_i$, $\forall i, 1 \leq i \leq N$.*

**Definition 3** *An incorrect gate–level description $G_C$ of a circuit $C$ is* **$N$-source correctable** *if there exist an $N$–error line tuple $L$ and an $N$–correction tuple $C$ that when it is applied on $L$, it makes the functions implemented at respective primary outputs of $F_C$ and $G_C$ agree.*

**Example 2** *Applying $C = \{$ gate replacement* `AND` *$\}$ on $L = \{G_3\}$ for the circuit of Fig. 1.1 we get a new circuit where $G_3$ has been replaced by an* `AND` *gate. Applying $C = \{$ missing wire $I_4$, extra wire $I_4\}$ on $L = \{G_3,\ G_2\}$, we get a new circuit where primary input $I_4$ is disconnected from $G_2$ and connected to $G_3$.*

Comparing the modification model of Def. 3 to the one proposed in [2] we observe that it covers all types of design errors except type $j$. This is because a single design error of type $j$ can propagate to the primary outputs either through $G_1$ or $G_2$ (Fig. 1.2) and the design is modeled as 2-source correctable. In addition, the Wrong Gate/Wrong Wire error is an extension of the error model of [2].

It should be noted that the set of modification types of Def. 1 is not a mandatory set for the execution of the diagnosis algorithms presented in this thesis. In contrast to most of the previous work for DEDC, the algorithms presented here can work under any modification set as long as this set is known beforehand, and each candidate modification applies to only one line so that the modification respects the $N$-source correctability of Def. 3.

For the correction phase of our approach, we choose to use the design error model of Def. 3 because of its *simplicity.* A desirable solution to the DEDC problem must keep the structural differences between the initial and final implementation minimal so that it preserves most of the previous engineering effort which may involve many synthesis and optimization steps. If we use a conventional re-synthesis approach [44] for correction, there might be a large number of structural differences.

We should emphasize the fact that the rectification problem discussed here is based on the type and number of modifications needed to correct the erroneous implementation and it is not based on the number and type of **actual** design errors. As explained in Chapter 4, since there might be more than one way to synthesize a particular function there can be more than one way to correct an erroneous design. This explains the existence of **equivalent** corrections.

Therefore, the **output** of our DEDC methodology is a *set* of $N$–correction tuples that correct the design. This set might contain the actual and some equivalent corrections. Throughout this thesis, we will refer to an actual or an equivalent modification tuple as a **valid** modification tuple.

The above discussion gives rise to the following definition.

**Definition 4** *Given a design error model, a diagnosis method is* **exact on the error space** *if every error line tuple it returns is a set of lines of some valid modification tuple. A DEDC method is* **exhaustive on the error space** *if it guarantees to return all set of lines of all valid modification tuples.*

**Definition 5** *Given a design error model, a DEDC method is* **exact on the correction space** *if every modification tuple it returns is a valid modification tuple. A DEDC method* *is* **exhaustive on the correction space** *if it guarantees to return all valid modification tuples from the design error model used.*

In our presentation throughout this thesis we make the following assumption:

**Error Assumption:** For every line $l$ that is in some valid set of modification locations there exists a vector $v \in V_{act}$ that produces a sensitized path from $l$ to some primary output.

In Section 3.5, we relax this assumption and discuss its implications.

## 1.3 Previous Work

In this section we will briefly review previous work for the DEDC and EC problems.

Most of the previous methods for the DEDC problem have adopted the design error model of Abadir et al. [2]. Methods for DEDC can be divided in two categories with respect to the underlying technique used for error location (diagnosis) and error correction: (1) the ones based on **Boolean function manipulation** techniques [20] [22] [23] [25] [38] [39] [40] [54] [62] and (2) the ones based on **Test Vector Simulation** [27] [28] [32] [37] [50] [51] [55] [56] [64] [65].

### 1.3.1 DEDC Symbolic Methods

The work of [38] and [40] applies to single design errors. In [40], formal verification is based on Typed Decision Graphs (TDG) and Boolean equations guide the error location process. Once an erroneous gate with $k$ inputs is found, $2^k$ new variables need to be introduced to the TDG for correction, which make the proposed approach impractical. Liaw et al. [38] improved upon the method of [40] by introducing the concept of dominators during error location and by providing a better gate correction process. The experimental results of both papers are based on small circuits and the error model contains only gate changes.

[54] introduces an algorithm where the functional description is partitioned into control and data–path circuits and rectification of each of them is carried out separately, but correction has to be done manually. In [20] and [62], extra circuitry derived by Boolean comparison methods is added at the primary inputs and outputs of the existing implementation. This extra circuitry rectifies the design but it can increase the circuit area and circuit delay.

The work in [22] is exact and exhaustive on the error space for single design errors and the work in [23] considers certain classes of multiple design errors. Verification is performed with the use of BDDs and error location is carried at the intersection of the backtrace cones of the erroneous primary outputs. An error equation with a single unknown [22] is formed at candidate lines of the circuit and lines that give no solution to this equation are deleted for the purpose of error location. The results of the error equation also drive the correction procedure for single errors in [22] but the method does not guarantee to return a solution for multiple errors [23].

The work in [39] is also based on Boolean comparisons. It applies to multiple errors and no error model is assumed. For locating potential modifications an error equation is formed in a way similar to [22] and heuristics on individual erroneous primary outputs of $G_C$ are employed if this error equation does not provide enough information for error location. For correction, a combination of existing correction and logic minimization synthesis tools are used [15] [16] [45], which means that the differences between the original and final design may not be minimal. In [25] a novel technique for dynamic support for constructing BDDs on equivalent signals is presented so that the BDD memory explosion problem is eliminated. The technique of [25] borrows from existing synthesis algorithms for engineering change [8] and a mapping of equivalent signals between the specification and the incorrect design is established. This method is independent of any particular error model.

### 1.3.2 DEDC Test–Vector Simulation Methods

The work of Tomita et al. in [55] and [56] is based on simulation of IPLDEs (Input Patterns for Locating Design Errors) drawn from the set of values $\{0, 1, X\}$. These vectors produce erroneous responses at the primary outputs of the circuit and they are generated with the use of BDDs [57]. In [55], single output circuits with single design errors are examined and the gates where the $X/\overline{X}$ values stop propagating give information on potential error locations. The method of [37] is along the same lines, but the test vectors are generated with a method similar to the one in [21]. The work in [56] applies to circuits with multiple primary outputs and multiple design errors but uses a subset of the error model of [2]. Based on simulation of the IPLDEs, an error possibility index on every line in the circuit is used to reduce the number of potential candidates. A unique six–valued simulation reduces this set further.

[27] [28] use the results of an iterative stuck-at fault simulation procedure to reduce the space of potential candidates for multiple design error diagnosis. Observations on

dominating signals of the circuit, similar to those presented in this thesis and in [22] [23] [38], speed up the proposed approach that can perform diagnosis with good resolution, but the method does not perform rectification. In [28] the authors extend the results so that they diagnose sequential circuits that one–to–one flip–flop correspondence between the implementation and the specification does not exist. In such a case, combinational approaches cannot be used.

Wahba and Borrione in [64] [65] propose an exhaustive on the error space diagnostic algorithm for single design errors based on a backward–propagation procedure, but the error model is restricted to three types of errors only. Moreover, the method does not work on circuits corrupted by multiple design errors. In [65] the concept of possible next states is defined so that the method can diagnose sequential circuits. Kuehlmann et al. [32] propose a modified critical path tracing algorithm [3] that starts from failing primary outputs and identifies suspicious areas in the circuit. The algorithm has good performance for single errors but the resolution diminishes as the number of errors increases.

Finally, [50] and [51] present two different test simulation–based approaches. In [50], the minterm differences at the output of the circuit under consideration are used to devise a correction hardware at the primary outputs and rectify the design. Multiple gate type only changes are then applied and retained if they reduce the size of the hardware. The method is not exhaustive on the error space as it is based on an experimental observation that minterm differences at the primary outputs usually increase monotonically with the number of injected errors. The work of [51] applies to macro–based circuits and guarantees to return a solution for single macro errors as long as there are vectors that propagate the error to some primary output(s). Two different types of errors are considered: (1) input combination errors, where a macro produces erroneous responses for some input combinations, and (2) line interconnection errors. Test–vectors are applied and, for every macro, four different counters are updated. The values of these counters are used to screen out macros that are not suitable for correction. The experimental results of [50] and [51] are available only for small circuits and the run times are not available.

## 1.4 Thesis Outline and Contribution

In this work we develop a diagnosis and correction method that applies to a wide variety of design errors [2]. The method detects and corrects multiple errors and is time efficient for one, two and three design errors. We actually approach the problem from two different points of view and develop a symbolic (BDD based) method and a method based on test–vector simulation.

We also examine the quality of test–vector simulation for design error diagnosis and correction. We compare the results with those obtained by BDDs [12] and conclude that a test–vector simulation method is an attractive alternative to methods based on global BDDs.

The importance of test–vector simulation for multiple design error rectification comes from the fact that for some circuits we cannot obtain their global BDD representation. Some circuits, like the ISCAS'85 benchmark multiplier C6288, require exponential size BDDs [13]. This makes methods based on global BDDs not applicable to some circuits due to their exponential memory requirements. Test–vector simulation has been also experimentally proven to be a run–time efficient approach for the problem of multiple DEDC [27] [28] [55].

Our work for error location falls along the lines of the work of [27] [28] [32] [51] and [56]; it uses test–vector simulation of stuck-at fault vectors [47] together with random vectors. This approach has been experimentally proven to provide good and fast design error isolation [2] [5] [27] [28] [32] [51] [56].

For correction we propose two methods. The first method uses test–vector simulation. In the second method we extend the results of the symbolic method in [22] [23] so that we correct multiple errors.

Unlike most of the previous methods for the problem of multiple DEDC, our diagnosis algorithms are independent of the design error model used during correction. Moreover, no information on the correct gate level description of the design is required and, as explained earlier, the design error model used for correction is an extension of the one in [2]. Finally, the run–time performance and error resolution returned by our approach, is better than previous methods proposed for this problem.

An overview of the proposed methodology appears in Fig. 1.3. The input to the method is $F_C$, $G_C$, $V_{act}$, and an initial estimate (or guess) for the desired number of modifications $N$. The output of the algorithm, as shown in Fig. 1.3, is a set of $N$–correction tuples. If the test–vector simulation based DEDC method is used, this set of $N$–correction tuples is subsequently filtered by a verification tool [12] [9] [25] [34] [36] [31] [41]. The output of this verification process is the final set of valid corrections. If the method fails to return a valid correction then it is repeated for a higher value of $N$.

In detail, Chapter 2 proposes a two step error location method. The first step of the method does an explicit enumeration of error tuples. This makes the method exhaustive on the error space which means that it guarantees to capture all erroneous lines. On the

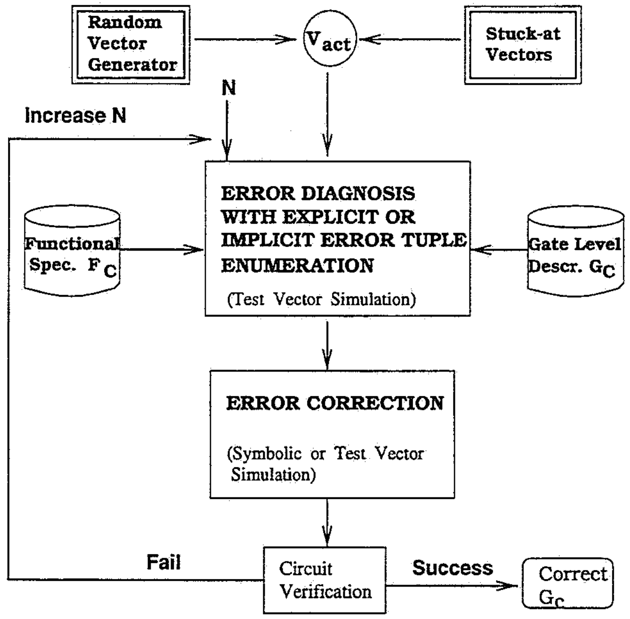


**Figure 1.3** Method Overview

other hand, as is the case with other methods that use explicit error tuple enumeration [27] [28] [55] [56] [51], the method is not efficient on circuits with a large number of design errors because the error space grows exponentially to the number of design errors according to Eq. 1.1. Nevertheless, the method exhibits good run–time performance for single and double errors as long as the set $V_{act}$ is not empty. A proof of the correctness of the method, which we believe it is interesting on its own, is also included in Chapter 2.

In Chapter 3, we propose a test–vector simulation procedure that does an implicit enumeration of error tuples. The procedure is not exhaustive on the error space but we develop techniques to make it behave exhaustively throughout our experiments. This method returns a diagnostic solution in a short computational time for one, two, and three errors. In the same chapter, the implication of error masking [10] [51] on the problem of DEDC is discussed.

In Chapter 4 we present our correction procedures. First, an algorithm based on test–vector simulation is developed. This algorithm is robust and run–time efficient. We then proceed by describing a symbolic correction approach that is exact on the correction space but uses global BDDs. Both approaches are exhaustive on the correction space.

Chapter 5 contains the experimental results that show the efficiency and robustness of the proposed methodology. The experimental results also suggest that test simulation is an attractive alternative to Boolean function manipulation for multiple DEDC.

In Chapter 6 we describe applications of this research in different VLSI CAD areas. The same chapter contains the conclusion of this thesis.

# CHAPTER 2

# Multiple Design Error Diagnosis With Explicit Enumeration of Error Tuples

## 2.1 Introduction

In Chapter 1 we defined the problem of *Design Error Diagnosis and Correction (DEDC)*. In this chapter, we will describe an effective multiple design error diagnosis method that is based on test vector simulation. In Section 2.2 we prove a theorem that shows that the proposed method is *exhaustive on the error space*, that is, the method guarantees to return all lines of the circuit where a rectification can be applied and rectify the design. In addition, the proposed diagnosis methodology is independent of the design error model used, and does not require a netlist representation of the specification. Finally, since it is based on simulation of test vectors, it is applicable to large circuits where methods based on BDDs might fail.

The proposed approach includes two error location steps (Figure 2.1). The first step of error diagnosis method is **Total Observability Measure**. Recall that $N$ is the number of required modifications that we suspect can rectify the erroneous implementation and $V_{act}$ is a set of vectors that produce erroneous primary output responses. During this step, for each vector $v \in V_{act}$, all $N$–error line tuples are *explicitly* enumerated and disregarded from subsequent iterations of the algorithm if they do not meet certain requirements. The number of $N$–error tuples at the beginning of the algorithm is equal to $(\# \ of \ circuit \ lines)^N$ and at every step of the algorithm this number is decreased. The main objective of Total Observability Measure is to reduce the initial error space significantly in an efficient and fast manner.

Next, another explicit test vector simulation procedure is introduced, **Inverted Simulation**. In Inverted Simulation, each remaining error tuple is examined separately and disregarded if it cannot correct the circuit for the test vectors that produced erroneous primary output responses. Inverted Simulation is based on a novel simulation procedure of every design error excitation scenario at the fan-out cones of the error lines under consideration. Inverted Simulation is used as part of the diagnosis algorithm presented in Chapter 3 as well.

Observations on the dominance relation of the lines of the circuit allows us to speed up both aforementioned error location steps considerably. The experimental results in Chapter 5 show the effectiveness of our diagnosis methodology by providing sufficiently good error resolution.

## 2.2 A Necessary Condition for Circuit $N$-Source Correctability

In this section we present an exhaustive on the error space test–vector simulation procedure that performs design error diagnosis with explicit enumeration of error tuples.

The method borrows from [56], but unlike the work of [56] it only takes into account lines under simulation with well specified values 0 and 1, ignoring the ones with unknown

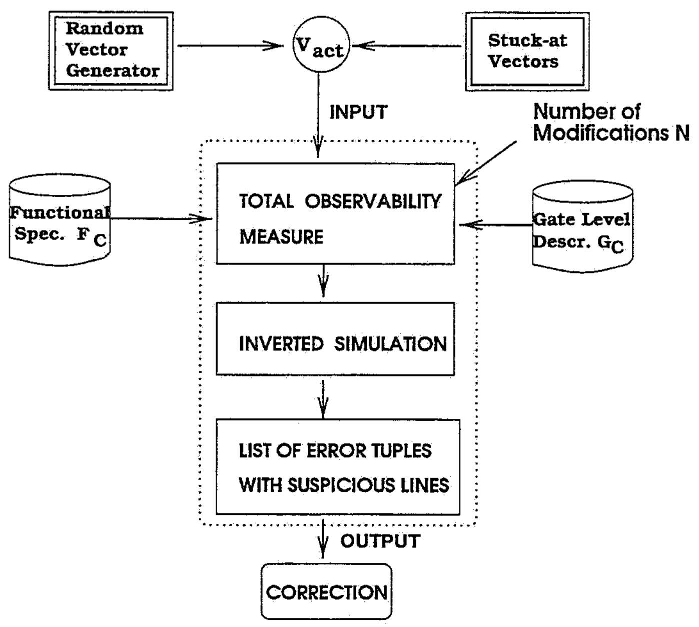


**Figure 2.1** Overview of Diagnosis With Explicit Enumeration of Error Tuples

value $X$. In addition, a formal proof of its correctness is presented. This proof is of combinatorial nature and it is interesting on its own.

Finally, the proposed method does not require the need of BDDs to compile the input test vector set $V_{act}$. In our experiments, $V_{act}$ is compiled from input test–vectors for stuck-at faults [47] and random test–vector simulation. Abadir et al. [2] proved that a complete (that is, 100% error coverage) test set for stuck-at faults for the incorrect circuit is guaranteed to detect errors from $a$ to $f$ (Fig. 1.2) and has a high probability of detecting the remaining ones. In addition, [2] contains a proof that the substitution of a gate with a unate function (and vice versa) will always be detected with a complete stuck-at fault test set. The experiments in [5] confirm the above claims.

On the other hand, random test–vector simulation has been used for design error verification and diagnosis in the past [27] [51]. For these reasons, we choose to verify the design with vectors for stuck-at faults along with random test–vector simulation and compile the set $V_{act}$ from all those vectors that produce erroneous primary output responses. The size of $V_{act}$ throughout our experiments is less than 100 vectors, on the average.

We are now ready to establish the theoretical foundation of our approach.

**Definition 6** *Let vector $v \in V_{act}$ and let $PO_i \in PO_{G_C}$ with* incorrect *output value under $v$. We define the* **observability measure,** *$OM_l^i$, of a line $l$ for vector $v$ and over $PO_i$ recursively as follows:*

*(a) If $l$ is a primary output, then:*

- $OM_l^i = 1$ *if* $l = PO_i$.
- $OM_l^i = 0$ *if* $l \neq PO_i$.

*(b) if $l$ is fan-in of a gate $G$ and $l'$ is the fan-out of $G$ then:*

- $OM_l^i = 0$, *if $l$ has non-controlling value and some other fan-in of $G$ has controlling value.*
- $OM_l^i = OM_{l'}^i$, *if $l$ has non-controlling value and all fan-ins of $G$ have non-controlling value.*
- $OM_l^i = \frac{OM_{l'}^i}{cv}$, *if $l$ has controlling value, where $cv$ is the number of fan-ins of $G$ with controlling value.*

*(c) if $l$ is a fan-out stem, then $OM_l^i = min(1, \sum_{l'} OM_{l'}^i)$, $l'$ drives a buffer at the fan-out branch of $l$.*

**Definition 7** *We define the* **accumulated observability measure** *$AOM_l^v$ of a line $l$ over a vector $v \in V_{act}$ to be the sum of the observability measures of $l$ for all erroneous primary outputs $PO_i$ for $v$, that is:*

$$AOM_l^v = \sum_{err.\ PO_i\ for\ v} OM_l^i$$

Intuitively, the observability measure, $OM_l^i$, of a line $l$ of $G_C$ over an erroneous primary output $PO_i$ for $v \in V_{act}$ denotes the *potential* of $l$ to change $PO_i$ into its correct value (for $v$). Similarly, the accumulated observability measure, $AOM_l^v$, indicates the potential of $l$ to change *all erroneous* primary outputs (for $v$). The recursive definition of both quantities result in efficient ways for their computation with minimal memory requirements.

Theorem 1 that follows is essential for the correctness of the error location procedure presented in Section 2.3. Before we state and prove this theorem we need to prove the following lemma.

**Lemma 1** *Let $v \in V_{act}$, $PO_e \in PO_{G_C}$ erroneous for $v$, and let $L_v = (l_1, \ldots, l_k)$ be a minimal set of lines such that when their values are complemented, $PO_e$ turns into its correct value. If $l$ is a line such that there exists at least one sensitized path from $l$ to each member of $L_v$, then:*

$$OM_l^e \geq \sum_{i=1\ldots k} OM_{l_i}^e$$

**Proof.** The proof of the lemma is by induction. Let a *k-cut* on $G_C$ be the set of all lines of $G_C$ at level $k$. In this proof, we define the level of the members of $L_v$ to be 0, their immediate fan-ins 1 and so on. The level of a fan-out stem is equal to the maximum level of its branches plus one. The level of all lines not in the backtrace cone of some member of $L_v$ need not be defined. Observe that under this definition, the level of $l$ is bounded by the primary input with the minimum such level. Let $j$ be the level of $l$, and let $MS_i$ be any minimal set of lines $l'$ of the $i$-cut, $i \leq j$, such that for every member of $L_v$ there exists at least one sensistized path from some member of $MS_i$. We claim that $\sum_{l' \in MS_i} OM^e_{l'} \geq \sum_{l_v \in L_v} OM^e_{l_v}$. Proving the claim proves the lemma.

To prove the claim, we use induction on the level of cuts starting from the members of $L_v$ towards $l$. For the base case, the claim obviously holds since the members of $L_v$ are a minimal set. Assume it is true for $k$, we prove that it holds for all minimal sets of the $(k+1)$-cut. Let $MS_{k+1}$ be any such set. Let $S_k$ be the set of lines of the $k$-cut compiled as follows: for every set of lines of $MS_{k+1}$ that drive the same gate $G$, $S_k$ contains the output of $G$ and for every line $l' \in MS_{k+1}$ that drives a branch, $S_k$ contains *all* fan-out branches of $l'$. It is straightforward to see that $S_k$ contains at least one minimal set of lines $MS_k$ (otherwise $MS_{k+1}$ is not minimal). Moreover, if $l'$ is a line of $MS_k$ driven by gate $G$, then by Definition 3(b) the minimal set of lines fan-in to $G$ that belong in $MS_{k+1}$ should have a sum of observability measures equal to $OM^e_{l'}$. If they don't, they will not be able to complement the value of $l'$. If $l''$ is a fan-out stem of $MS_{k+1}$, then by Definition 3(c) $l''$ will have an observability measure value equal to the sum of the observability measure values of *all* of its branches (or 1). By the pigeonhole principle, the sum of the observability measures of all lines of $MS_{k+1}$ should be greater or equal to that of the $MS_k$ ones. This proves the claim and completes the proof.

**Theorem 1** *Let $L = \{l_1, l_2, \ldots, l_n\}$, $n \geq 1$ be a set of valid modification locations for an incorrect $G_C$. If $v \in V_{act}$ then:*

$$\sum_{l_i \in L} AOM_{l_i}^{v} \geq \#\ erroneous\ POs\ for\ v$$

**Proof.** To prove the theorem it suffices to show that if $PO_e$ is an erroneous primary output for $v$ then $\sum_{l_i \in L} OM_{l_i}^{e} \geq 1$. To prove this, we use induction on the number of modification locations $n$. For the proof of this theorem, the level of the primary outputs is 0, their immediate fan-ins 1 and so on up to the primary inputs. The level of a fan-out stem is equal to the maximum level of its branches plus one. The base case, $n = 1$, holds if we let $L_v$ in Lemma 1 be the singleton $PO_e$ and $l$ be the only member of $L$. If it is true for $n$, we show that it holds for a set $L$ with $n + 1$ elements. Let $L' = L - \{l \in L\ with\ maximum\ level\}$. Let also $l'$ be the line of $L'$ with maximum level $k$ and let $MS_k$ be a minimal set of lines, as defined in Lemma 1, that contains $l'$ and there exists at least one sensitized path from $l$ to every member of $MS_k - l'$. It is straightforward to see that such a minimal set exists because $L$ is a set of valid modification locations. Let pseudo primary input $PI_p$ drive all lines of $MS_k$. By the induction hypothesis

$$\sum_{l'' \in \{PI_p, L' - l'\}} OM_{l''}^{e} \geq 1 \tag{2.1}$$

Using Lemma 1 and Equation 2.1 the induction step follows and completes the proof.

The significance of Theorem 1 lies in the fact that it provides a tool to screen out candidate modification locations that have an accumulated observability measure less than the total number of erroneous primary outputs for some vector $v \in V_{act}$ and, therefore, cannot be a set of valid modification locations. Further observations on the structure of the circuit, presented in Section 2.3.4, allow us the speed the above computation further more. Observe that the theorem is one way only; $L$ might satisfy the condition of Theorem 1 but not be a set of valid modification locations.

The initial set of potential modification locations, $C_{error}$, for a $N$-source correctable $G_C$ is upper bounded by the number of all lines of $G_C$ to the power of modifications $N$. By applying the results of Theorem 1 repeatedly on the reduced set $C_{error}$ for each vector of $V_{act}$, we are able to reduce the size of $C_{error}$ and guarantee that we do not delete any valid candidate lines. The following example illustrates the above results.

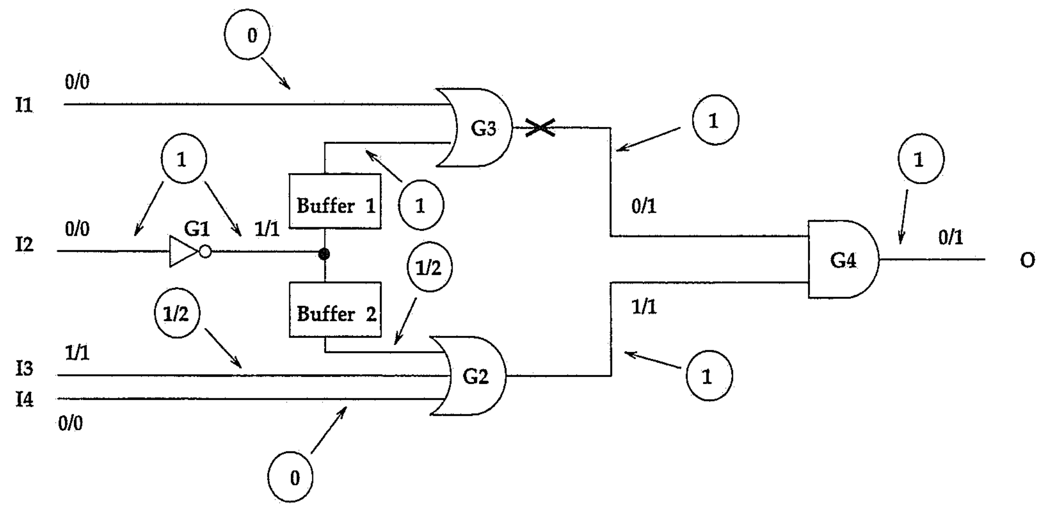


**Figure 2.2** Erroneous circuit for Example 3

**Example 3** *Fig. 2.2 shows the incorrect implementation of a circuit and the values of the circuit lines when input vector* $V = (PI_1, PI_2, PI_3, PI_4) = (0, 0, 1, 0)$ *is applied.*

*The correct implementation of the circuit is obtained if $G_3$ is replaced by an* AND *gate, therefore, $G_C$ is* 1*-source correctable. Vector $V$ activates this difference and the tuples on the lines are pairs of correct/incorrect simulation values. The AOM values of the lines are in circles. Since there is only one primary output, and this output is erroneous, the OM and AOM values coincide.*

*Because of OM's recursive definition, the values are computed from primary outputs towards the primary inputs. Although the circuit of Fig. 2.2 is* 1*-source correctable observe that $AOM_{G_1}^{V} = 1$, but no single modification, from the modification model of Def. 3, can apply on $G_1$ and correct it. Location $G_1$ is screened out from $C_{error}$ if we simulate input vector $V' = (1,1,1,0)$ and apply Theorem 1. In this case $AOM_{G_1}^{V'}$ is computed to be* 0.

## 2.3 Error Location

The error location procedure with explicit enumeration of error tuples consists of two steps, **Total Observability Measure** and **Inverted Simulation** (Fig. 2.1). These two steps screen out elements of $C_{error}$ where no modification can be applied and rectify the design.

During Total Observability Measure, the accumulated observability measure value of each line of $G_C$ is evaluated for all erroneous outputs and all vectors of $V_{act}$ according to Def. 7. In Section 2.3.1 we introduce a technique, based on properties of the observability measure concept, and prune the search–space dynamically during the computation.

The next step of the error location algorithm is Inverted Simulation where every error tuple of potential modification locations is examined separately and disregarded if it does not meet certain criteria. Finally, in Section 2.3.4, observations on the structure of the circuit allows us to speed up both error location steps.

In the following discussion, $machine-word$ is an integer value equal to the computer machine word length used. Every line $l$ of $G_C$ is considered to be a data object with four variable fields: $l \rightarrow level$, $l \rightarrow Vgroup$, $l \rightarrow Fgroup$ and $l \rightarrow Tgroup$. $level$ is an integer value that denotes the level of $l$ in the combinational circuit $G_C$, where the level of the primary inputs is equal to 0 and the level of a gate in the circuit is equal to the maximum level of its fan-ins plus one. $l \rightarrow Vgroup$ is an array of float numbers with cardinality $\mid V_{act} \mid$ that keeps the AOM values of $l$ for the vectors of $V_{act}$. The last two fields are lists of bit–vectors used during the Inverted Simulation and Correction procedures. We will explain the use of these lists later. Initially, the value of all entries of $Vgroup$ are zero and the $Fgroup$, $Tgroup$ linked lists point to $NIL$ for all lines of the circuit.

### 2.3.1 Total Observability Measure

The pseudocode that implements the theory presented in Section 2.2 is the **Total Observability Measure(TOM)** procedure and it is shown in Fig. 2.3.

The inputs to the algorithm are $F_C$, $G_C$, a set of lines $C$, the set $V_{act}$ and the number of modifications $N$. At the end of this section, we explain how the algorithm can be modified to return *all* modification tuples of cardinality $N$ or less. Evaluation of the $OM$ values takes place in a backward fashion, i.e. from primary outputs towards primary inputs. First, all vectors of $V_{act}$ are simulated in parallel [66], $machine-word$ vectors at a time, and the $Vgroup[k]$ fields of the respective erroneous primary outputs are updated (lines 1–7).

```
TOTAL_OBSERVABILITY_MEASURE( $F_C$, $G_C$, $C$, $V_{act}$, $N$)
        (* assign POstamps for all vectors of $V_{act}$ *)
1.     repeat
2.         Simulate $V_{machine-word}$ vectors in parallel from $V_{act}$
3.         for every vector $v_k \in V_{machine-word}, k = 1, \ldots, \mid V_{act} \mid$, do
4.             for every erroneous $PO_i$ do
5.                 $PO_i \rightarrow Vgroup[k]++$
6.                 $total\_aom[k] = total\_aom[k] + 1$
7.     until all vectors of $V_{act}$ are simulated
               (* now evaluate $OM$ values for all vectors of $V_{act}$ and lines $C$ *)
8.     for $j$ = (number of levels of $G_C$) down to 1 do
               (* evaluate $OM$ for all $j$-level lines driven by gates *)
9.         for every line $l$, driven by gate $G$, with $l \rightarrow level = j$ do
10.            for every fan-in line $l'$ of $G$ do
11.                for every $Vgroup[k] \neq 0$ of $l$ do
12.                    evaluate $OM_{l'}^i$ from $OM_l^i$ as in Definition 3(b)
13.                    if $OM_{l'}^i \neq 0$ then $l' \rightarrow Vgroup[k]++$
               (* evaluate $OM$ for all $j-1$-level fan-out stem lines *)
14.        for every fan-out stem $l$ with $l \rightarrow level = (j-1)$ do
15.            for every $Vgroup[k] \neq 0$ of the union of the $Vgroups$ of the branches of $l$ do
16.                for every erroneous $PO_i$ for vector $v_k$ do
17.                    $OM_l^i = \sum_{l'} OM_{l'}^i$, $l'$ branch of $l$
18.                    if $OM_l^i \neq 0$ then $l \rightarrow Vgroup[k] += min\{1, OM_l^i\}$
19.    for all $N$-tuples $L = \{l_1, l_2, \ldots, l_N\} \in C_{error}$, $l_1 \ldots l_N \in C$ do
20.        if $\sum_{l_i \in L} l_i \rightarrow Vgroup[k] \geq total\_aom[k]$, for all $k$, then
21.            add $L$ in $C_{error}$
22.    return( $C_{error}$ )
```

**Figure 2.3** Total Observability Measure Procedure

The loop of lines 8–18 propagates this information towards primary inputs. Let *circuit_level* denote the maximum level of a gate in $G_C$. During the $i$-th iteration of the loop, the $OM$ values of all gate–driven lines (lines 9–13) and fan–out stems (lines 14–18) of level $(circuit\_level - i - 1)$ are computed according to Definition 3 (b) and (c), respectively. Lines with zero AOM values are deleted from the computation since they have no values to propagate backwards (lines 11 and 15). Theorem 1 is used to screen out sets of potential modification locations that have an AOM value strictly less than the *total_aom* (lines 19–21).

It can be seen that the *time* and *space complexity* of the Total Observability Measure step is linear to the number of lines in the circuit to the power of required modifications $N$. This is because the algorithm has to go through all error tuples and apply the TOM procedure (Fig. 2.3, lines 19–21). This makes the method not efficient for high values of $N$ on large circuits, a problem that all exhaustive on the error space diagnosis methods exhibit [27] [28] [51] [56].

Observe that the algorithm, with a slight modification, can return *all* potential sets of modification locations of cardinality $N$ or less, if such sets exist. This can be achieved if we modify the code at lines 19–21 (Fig. 2.3) and apply Theorem 1 on *all* tuples with cardinality *equal or less* than $N$.

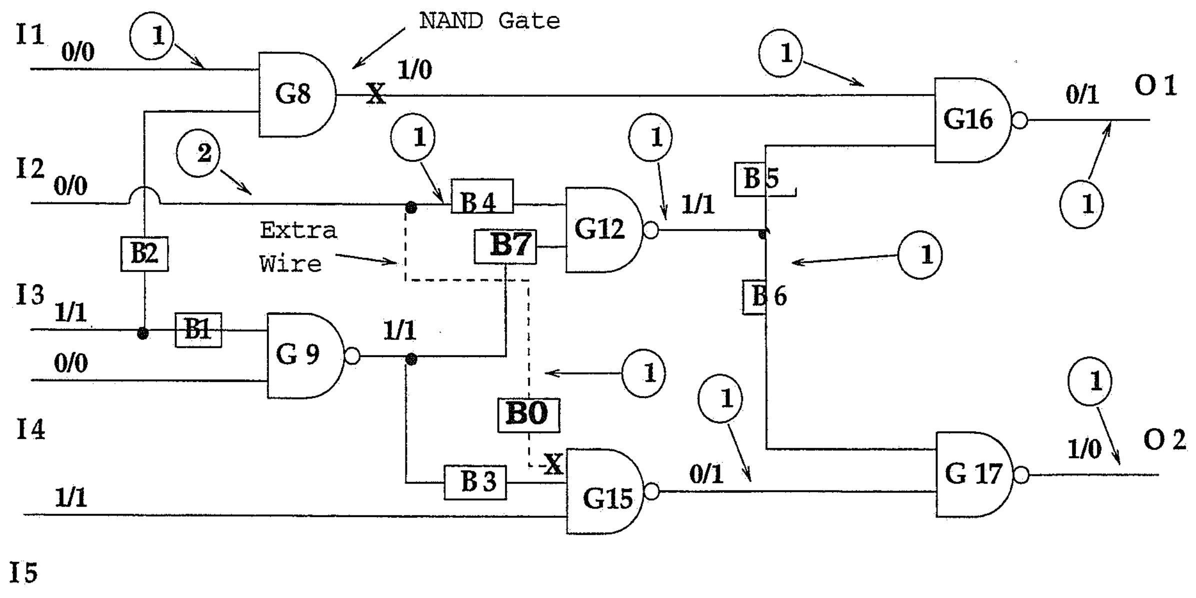


**Figure 2.4** Erroneous Circuit for Example 4

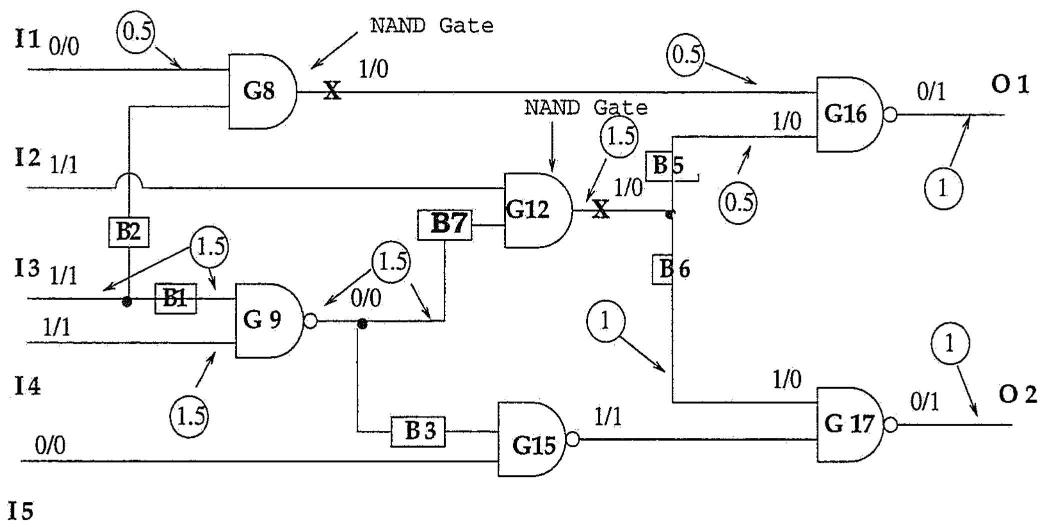


**Figure 2.5** Erroneous Circuit for Example 5

**Example 4** *Fig. 2.4 shows the* incorrect *gate-level implementation of a circuit where* $G_8$ *is supposed to be an* NAND *gate and the connection from* $I_2$ *to* $G_{15}$ *(dotted line) is incorrect. This circuit is a modified version of the ISCAS'85 benchmark C17 circuit.*

*The vector applied is* $V_1 = (0,0,1,0,1)$ *at primary inputs* $I_1, I_2, I_3, I_4, I_5$*, respectively. The pairs of incorrect/correct values for input vector* $V_1$ *are shown above the respective lines and the total observability measure of each line* $l$*,* $AOM_l^{V_1}$*, is in a circle when nonzero.*

*The value of* $total\_aom[1]$ *is equal to* 2 *since* $V_1$ *propagates the inconsistencies to both primary outputs. Initially, the size of* $C_{error}$ *is* 361*. After completion of the procedure for vector* $V_1$ *and* $N = 2$*, we have that* $\mid C_{error} \mid = 109$*, that is, all pairs of lines with an AOM sum bigger or equal than* 2*.*

**Example 5** *As another example, consider the erroneous circuit of Fig. 2.5 where two gate replacement errors on $G_8$ and $G_{12}$ produce erroneous responses at both primary outputs for input vector $(0,1,1,1,0)$. The simulation and observability measure values follow the notation of the previous example.*

*After completion of the procedure for this erroneous vector, the error set reduces to* $| C_{error} | = 58$ *pairs.*

### 2.3.2 Inverted Simulation

The main contribution of the previous step is to reduce the size of $C_{error}$ significantly enough so that a more accurate, but computationally "expensive", screening process can be applied. Such a process is **Inverted Simulation** that uses the information of the $Fgroup$ fields at each line.

During the simulation of vectors for the initial verification step of $G_C$, we create two bit-vectors at every line $l$ of $G_C$, $Fgroup$ and $Tgroup$. The $i$-th bit of the $Fgroup$ ($Tgroup$) list holds the value of the line when the $i$–th vector that activates (does not activate) the inconsistencies is simulated. Inverted Simulation, and the Correction procedure described later, make use of these lists. These bit–lists are shown in Fig. 2.6 for two erroneous input vectors of $Fgroup$ and an `AND` gate.

During Inverted Simulation, shown in Fig. 2.8, every candidate tuple of $C_{error}$ is examined separately and deleted if it cannot correct the circuit for all vectors of $V_{act}$. Since the size of $C_{error}$ has been significantly reduced after TOM, Inverted Simulation is efficient. The idea behind this process is as follows.

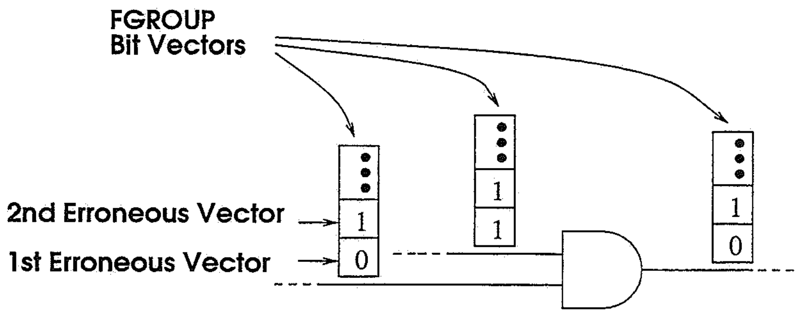


**Figure 2.6** *Fgroup* Bit–Vector Entries

Let $L = \{l_0, l_1, \ldots, l_{N-1}\} \in C_{error}$ and $v \in V_{act}$. If $L$ is a set of valid modification locations, then there exists *at least* one line $l_i \in L$ such that its value is incorrect and this difference is propagated to some primary output(s) during simulation for $v$. Let $L' \subseteq L$ be the set of all such lines that the design errors on these lines are excited for $v$. If the values of the lines of $L'$ for $v$ are complemented and this difference is propagated to the respective fan-out cones of $L'$ then the primary outputs of $F_C$ and $G_C$ should agree for $v$.

If this is not the case, then it is safe to delete $L = \{l_0, l_1, \ldots, l_{N-1}\}$ from the candidate error list $C_{error}$ as it is *guaranteed* not to be an valid set of modification locations.

**Example 6** *The numbers in the bit–lists in Fig. 2.7 are the new values of the lines at the fan–out cones of $G_{12}$ and $G_{15}$ when Inverted Simulation is performed for pair $\{G_{12}, G_{15}\} \in C_{error}$ and input test vector $V_1 = (0,0,1,0,1)$ for the erroneous circuit of Example 4.*

*Fig. 2.7(a) contains the erroneous circuit under simulation. Fig. 2.7(b) shows the Inverted Simulation scenario when the potential error on line $G_{12}$ is excited and the one on $G_{15}$ is not. For this reason, the value of the faulty bit–list of $G_{12}$ is complemented while* *the one of $G_{15}$ is not. Observe that simulation of this potential design error excitation scenario influences the lines in the union of the fan–out cones of $G_{12}$ and $G_{15}$, namely $\{G_{12}, B_5, B_6, G_{16}, G_{17}\}$ and only these lines.*

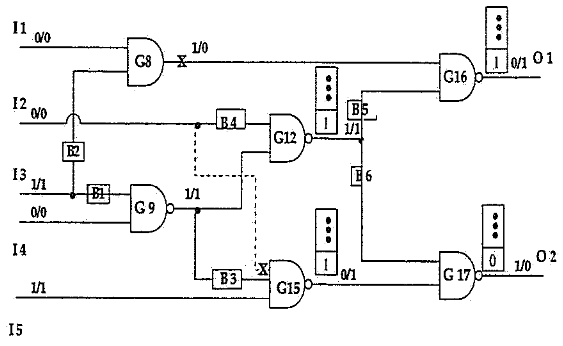
*It can be seen from Fig. 2.7(b) that the values at the primary outputs after simulation of this excitation scenario still differ from the correct ones. This implies that during simulation of vector $V_1 = (0, 0, 1, 0, 1)$ locations $G_{12}$ and $G_{15}$ cannot contain design errors that are and are not excited, respectively.*

*The remaining two simulations of the potential design error excitation scenarios for Inverted Simulation and vector $v$ are shown in Fig. 2.7(c) and (d). Since none of the simulations yields a correct response at the primary outputs of $G_C$, the error pair $\{G_{12}, G_{15}\}$ is guaranteed not to contain a pair of design errors and it should be deleted from $C_{error}$.*

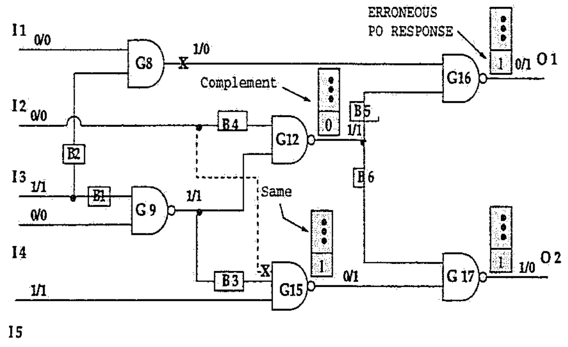
Fig. 2.8 outlines the overall procedure. For every set of lines $L \in C_{error}$ and every vector $v \in V_{act}$, $2^N - 1$ simulations are performed at the fan–out cones of the lines of $L$ (lines 1–7).

Each such simulation represents a different *design error excitation configuration* scenario for the lines of $L$ where a set $L' \subseteq L$ contains lines with excited errors and set $L - L'$ contains lines with errors that are not excited. The minus one term in $2^N - 1$ accounts for the case where no line of $L$ has an erroneous value (for $v$) and this case should be obviously excluded.

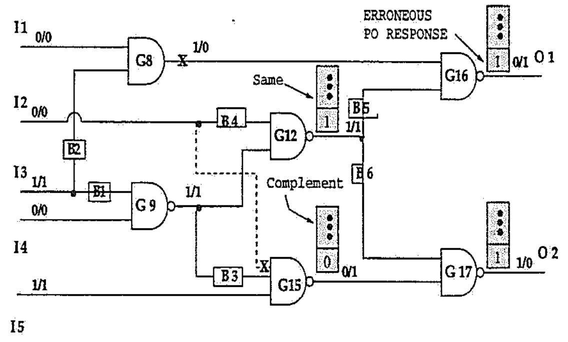
Inverted Simulation step is carried out efficiently with the use of the $Fgroup$ values so we need to resimulate *only* at the fan-out cones of the lines of $L'$. If $L$ is a set of valid modification locations then there would be *at least* one out of the $2^N - 1$ simulations that yields correct primary output responses for $v$. If this is not the case for some vector of



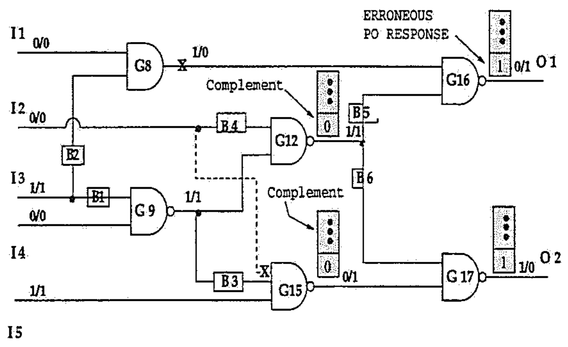
(a) Original Circuit and Faulty Bit Lists



(b) Inverted Simulation for (G12-excited, G15-not excited)



(c) Inverted Simulation for (G12-not excited, G15-excited)



(d) Inverted Simulation for (G12-excited, G15-excited)

**Figure 2.7** Inverted Simulation Procedure for Example 6

$V_{act}$, $L$ gets deleted from $C_{error}$ (line 8), otherwise, as shown in lines 9–11, the entry of $L$ in $C_{error}$ is replaced with tuple(s) of the form:

$$(l_0,\ l_1,\ \ldots,\ l_{N-1},\ not(k),\ v)$$

for each **error excitation configuration** $k$, $0 \leq k \leq 2^N - 2$, that rectifies the circuit during Inverted Simulation at line 7 for input test vector $v$. The excitation number $not(k)$ is a O(N) bit number where the value 1 at the $i$–th position indicates that the

```
INVERTED_SIMULATION( $F_C$, $G_C$, $l_0$, $l_1$, $\ldots$, $l_{N-1}$)
1.  for every vector $v \in V_{act}$ do
2.      for $i = 0$ to $2^N - 2$ do
3.          $i = not(i)$
4.          for $j = 0$ to $N - 1$ do
5.              if ($i$ AND $2^j$) $\geq 1$ then
6.                  $l_j \rightarrow Fgroup(v) = not(l_j \rightarrow Fgroup(v))$
7.              simulate at fan-out cones of $l_0$, $l_1$, $\ldots$, $l_{N-1}$ for $v$
8.      delete error tuple ($l_0$, $l_1$, $\ldots$, $l_{N-1}$) from $C_{error}$
9.      for every $k$, $0 \leq k \leq 2^N - 2$ do
10.         if simulation of excitation configuration $not(k)$ rectifies $C$ for $v$ at step 7 then
11.             add error tuple ($l_0$, $l_1$, $\ldots$, $l_{N-1}$, $not(k)$, $v$) in $C_{error}$
12.     restore $Fgroup$ values and values at $l_0$, $l_1$, $\ldots$, $l_{N-1}$ fan-outs
```

**Figure 2.8** Inverted Simulation Procedure

value of line $l_i$ should be complemented during Inverted Simulation for $v$ and the value of 0 indicates that it should remain unchanged.

The first reason we add multiple copies of error location tuples with their respective excitation numbers for the same vector comes from the fact that we need to capture valid error line tuples of cardinality *smaller* than $N$ as well. To see this, consider the box in Fig. 2.9 being a circuit under simulation of vector $v$ with only one failing primary output. The dotted lines are sensitized paths from locations $A$ and $B$ that merge to some gate $G$ and then propagate to the failing primary output. Without loss of generality, assume that $G$ has only two fan–ins and $v$ is the *only* failing input vector. The Inverted Simulation algorithm for $N = 2$ will include four entries for the error tuple $(A, B)$ and vector $v$, one for each design error excitation scenario. This is a desired result because all $(G)$,

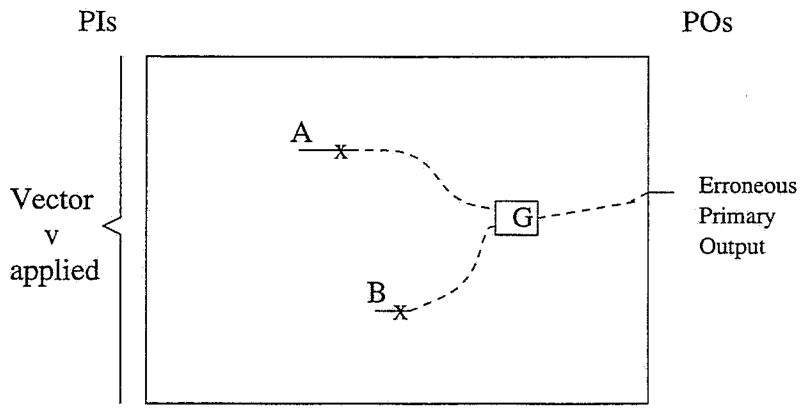


**Figure 2.9** Multiple Path Sensitization

$(A)$, $(B)$, and $(A, B)$ are *valid* modification locations for $N \leq 2$. Observe that the actual error(s) might or might not be some of the above error tuples.

The second reason of multiple error tuple addition will become evident during error correction, presented in Chapter 4. However, the example that follows gives the intuition behind this multiple error tuple addition.

**Example 7** *Fig. 2.10(a) shows an incorrect design with an extra inverter $G_1$ on line $I_3$ and a gate replacement error on line $G_3$. Observe that under simulation of vector $V = (1, 0, 1, 0)$, the extra inverter is not observable* [1] *at the output $O$ of the circuit. Since $O$ is erroneous for vector $V$, Inverted Simulation will return error tuples $(G_1, G_3, 2, v)$ and $(G_1, G_3, 3, v)$ (Fig. 2.10(c) and (d), respectively). Error excitation configurations 2 and 3 together imply that the value of $G_1$ for $V$ is a don't care $(X)$. This is expected, since $G_1$ is not observable for $V$ and it is a fact that will prove useful during correction.*

[1] An error is *observable* at a primary output $O$ if there is a vector that causes a sensitized path from the erroneous line to that primary output.

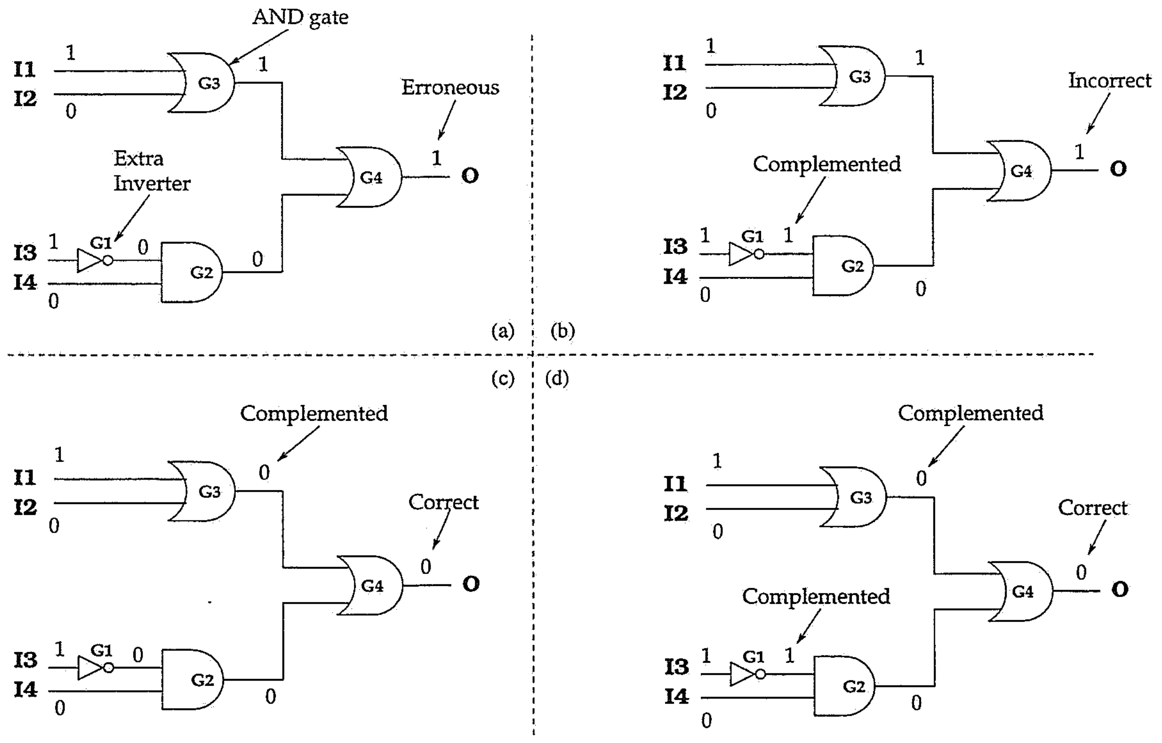


**Figure 2.10** Recording *don't care* values

One might argue that Inverted Simulation is an expensive diagnosis procedure since the *time complexity* is exponential to the number of desired modifications $N$. However, this is not a drawback since the values of $N$ for multiple DEDC are relatively small [2]. Our experimental results suggest that Inverted Simulation is an efficient procedure for small values of $N$ ($\leq 3$). Moreover, since the majority of the machines today use at least a 32 word bit–length, all $2^N - 1$ steps of Inverted Simulation for a $N$–error line tuple $L$ can be carried efficiently with one simulation at the fan-outs of the lines of $L$. Finally, Inverted Simulation works at the back end of the TOM procedure (Fig. 2.1) when the majority of error tuples have been already eliminated.

[2]Recall that the experimental results of [1] show that $N$ is usual less than or equal to 2.

### 2.3.3 Handling Unknown Values During Diagnosis

The simulation of good and faulty circuits is usually performed using three–valued logic: 0, 1, $X$. Therefore, our diagnosis strategy should cover the unknown value $X$.

In our presentation in Section 2.2, we assumed that the TOM procedure starts from a failing primary output where the good circuit has a fully specified value (0 or 1) and the faulty one has the opposite response. Considering how the procedure works, this failing primary output cannot have the unknown value $X$. Now observe that a gate whose output corresponds to the primary output under consideration either has one or more controlling inputs, or it has all non–controlling values at its inputs [3]. Therefore, we can set the $OM$ value of all lines with unknown value $X$ equal to 0 and Theorem 1 will still hold.

Unknown values present no problem to the Inverted Simulation procedure as the lines of the candidate error tuple we simulate always have well-defined (0 and 1) values.

### 2.3.4 Overall Diagnosis Approach

In this Section we describe the overall approach for error location that uses the concept of checkpoints.

**Definition 8** *We define a* **checkpoint** $B \in G_C$ *to be either a primary output or a fan-out stem. We also let the* **clan** $B_{clan}$ *of a checkpoint* $B$ *be the set of all lines* $l$*, including* $B$*, such that every path from* $l$ *to some primary output passes through* $B$ *and* $B$ *is the checkpoint with minimum level.*

[3]We assume that the output of an inverter with input $X$ is still $X$ and not $\overline{X}$.

**Example 8** *The black circles in Fig. 2.11 are the checkpoints for the erroneous circuit of Example 4. We also have that* $O_{2_{clan}} = \{I_5, B_0, B_3, G_{15}, B_6, G_{17}\}$ *and* $G_{12_{clan}} = \{B_7, B_4, G_{12}\}$.

It should be noted that our definition and use of checkpoints is different from the one presented in [68]. Moreover, for a line $l \in B_{clan}$, $B$ is a *dominator* for $l$ but it is not necessarily an immediate dominator [22].

Computing the set of checkpoints for a circuit $G_C$ takes time linear to the number of lines of $G_C$ and can be done easily with an one pass over the circuit lines. The following theorem, taken from [22] [38] is crucial for the correctness of the overall diagnosis strategy. Note that the theorem is independent of any design error model.

**Theorem 2** *Let lines* $l, l' \in G_C$ *where* $l'$ *dominates* $l$ [4]. *If there exists no modification on* $l'$ *that can rectify* $G_C$ *then there exists no modification on* $l$ *that rectifies* $G_C$ *as well.*

Intuitively the theorem holds because if a line $l'$ dominates an erroneous line $l$ then every sensitized path from $l$ to the erroneous primary output(s) must necessarily pass through $l'$.

The overall strategy for error location, shown in Fig. 2.12, is as follows: at first, the procedures of Total Observability Measure and Inverted Simulation are applied to the set of checkpoints $B$ of the circuit $G_C$ (lines 1–4). In this manner, the error location procedure deletes clans of the circuit that do not contain a design error. In this step of the algorithm we need not save the excitation configuration number during Inverted Simulation (line 4).

[4]We say that a line $l'$ dominates line $l$ if and only if all paths from $l$ to all primary outputs pass through $l'$.

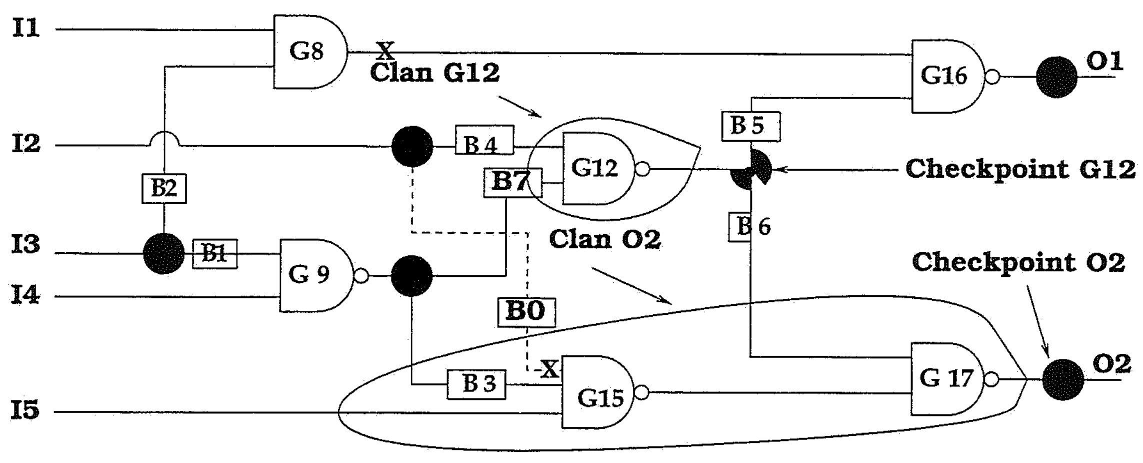


**Figure 2.11** Checkpoints and Clans for the Circuit of Example 8

Next, a new set of candidate lines is formed from the lines of the clans of checkpoints of $C'_{error}$ (line 5). Finally, the two steps of the error location procedure are applied to this new set of lines in order to compile the final $C_{error}$ set (lines 6–9). At the end of the invocation of the Inverted Simulation procedure at line 8 the format of the candidate list $C_{error}$ may contain multiple entries for error line tuples but each of them will have a different excitation configuration number (or vector) signature.

## 2.4 Summary

In this chapter we presented an exhaustive on the error space method for multiple design error diagnosis that is based on test vector simulation. The method is independent of the design error model used as it identifies lines that can correct a design for a certain amount of vectors $V_{act}$ and regardless of the error model that is used. In addition, the proposed approach does not require any naming equivalence between the design and the

```
ERROR_LOCATION( $F_C$, $G_C$, $V_{act}$, $N$)
1.   $B_{checkpoints} = \{B_1, \ldots, B_c\} =$ checkpoints of $G_C$
2.   $C'_{error}$ =Total_Observability_Measure( $F_C$, $G_C$, $B_{checkpoints}$, $V_{act}$, $N$)
3.   for every error-tuple $L_0, \ldots, L_{N-1} \in C'_{error}$ do
4.        Inverted_Simulation( $F_C$, $G_C$, $L_0, \ldots, L_{N-1}$)
5.   C = union of clans of checkpoints $B_i \in$ some pair of $C'_{error}$
          (* find the final $C_{error}$ from the checkpoints *)
6.   $C_{error}$ =Total_Observability_Measure( $F_C$, $G_C$, $C$, $V_{act}$, $N$)
7.   for every error-tuple $L_0, \ldots, L_{N-1} \in C_{error}$ do
8.        Inverted_Simulation( $F_C$, $G_C$, $L_0, \ldots, L_{N-1}$)
9.   return($C_{error}$)
```

**Figure 2.12** Overall Diagnosis Approach

specification and it is applicable to large circuits where methods based on BDDs might fail.

The first step of the diagnosis method is Total Observability measure. During this step, for each erroneous vector $v$, erroneous tuples of cardinality less than or equal to $N$, where $N$ is usually less than or equally to 2, are explicitly enumerated and disregarded from subsequent iterations of the algorithm if they do not meet the requirements of Theorem 1. The main objective of Total Observability Measure is to reduce the error space in an efficient and fast manner.

Next, an efficient test vector simulation procedure is introduced, Inverted Simulation. In Inverted Simulation, each remaining error tuple is examined separately and disregarded if it cannot correct the circuit for the test vectors with erroneous primary output responses. Inverted Simulation is based on a novel simulation procedure of every design error excitation scenario at the fan-out cones of the error line tuple under consideration.

Finally, observations on the dominance relation between lines of the circuit allow to speed up both aforementioned error location steps considerably.

# CHAPTER 3

# Multiple Design Error Diagnosis With Implicit Enumeration of Error Tuples

## 3.1 Introduction

In this chapter we describe a diagnosis method for multiple DEDC that does an implicit enumeration of error tuples. Our experimental results in chapter 5 show that the method is efficient for diagnosis of designs with a large number of design errors ($N > 2$). The method is not exhaustive on the error space, however, we develop techniques and present heuristics that make it behave very close to exhaustive throughout our experiments.

An overview of the proposed methodology is shown in Fig. 3.1. The method described here is an extension of the work presented in [3] [4] [52] [60] [61]. Instead of explicitly enumerating the set of all error tuples and reducing the error space from there [27] [28] [55] [56] [51] [58] [59], we start with an initial estimate of the set of error tuples. This set is further reduced by Inverted Simulation. If at the end of the procedure the error set has qualifying candidates, we proceed with correction (Fig. 3.1). If the error list is empty it means that either the circuit is $N$–source correctable but the error estimate was not accurate enough, or the circuit is $M$–source correctable for some $M > N$. Assuming that the design is $N$–source correctable, we repeat the error diagnosis procedure

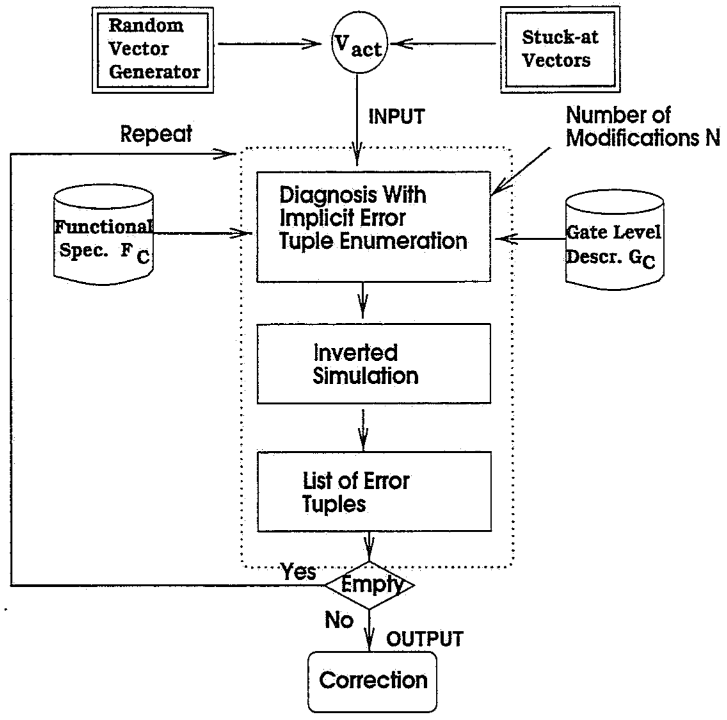


**Figure 3.1** Overview of Diagnosis With Explicit Enumeration of Error Tuples

with a different estimate. If the error list is still empty after a number of iterations of the algorithm, usually 3 or 4, then we can conclude that the circuit is not $N$–source correctable and we run the algorithm for a higher value of $N$.

The experimental results in Chapter 5 show that the proposed approach has good error resolution and it is run–time efficient as it can return results for designs corrupted with multiple errors within seconds.

Recall from Chapter 1 that a line $l$, fan–in to an AND or NAND (OR or NOR) gate, has **controlling value** for input vector $v$ if the value of $l$ is 0 (1). If $l$ drives a NOT or a BUFFER it always has controlling value. Moreover, a line whose value changes during simulation under the presense of some fault(s) is called a **sensitized line** and a path of sensitized lines is called a **sensitized path** [10].

## 3.2 Tracing Backwards from Erroneous Outputs

In this section we describe the *path–traceback* procedure developed in [60] to diagnose bridging faults. We also give a proof of correctness of the procedure in the context of multiple design errors.

Path–traceback borrows from *critical path tracing* [3], star algorithm [4], and support sets [52]. However, there are some subtle differences.

The above procedures were developed for diagnosis of single stuck–at faults. In other words, the procedures assume that only one line can be the source of error. For multiple design errors [1] the source of failing primary outputs can be more than one line. Additionally, critical path tracing can result in approximations that will affect design error diagnosis due to *multiple path sensitization* and *partial self–masking* [60].

### 3.2.1 Critical Path–Tracing

Before we describe the path–traceback procedure, we will give a brief overview of the critical path–tracing algorithm [10]. In the following discussion, we consider lines with well specified values (0 or 1).

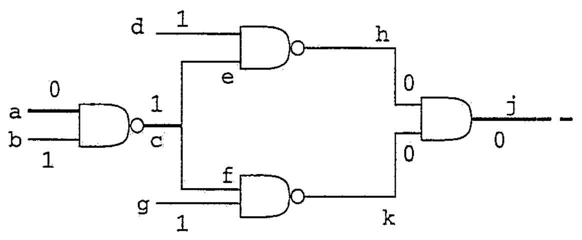
**Definition 9** *A line $l$ has* critical *value $x$ for vector $v$ if and only if vector $v$ detects the error stuck–at $\overline{x}$. If $l$ has critical value for some vector $v$ then $l$ is a critical line for $v$.*

[1] A circuit corrupted by a bridging fault can also be viewed as a $N$–source correctable design for $N \leq 2$. $N$ can be 1 for a bridging fault because of the *Byzantine Generals* problem [42].

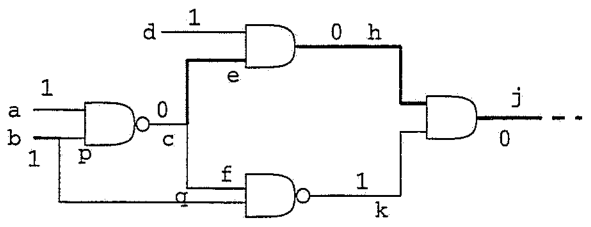
**Definition 10** *A gate input is* sensitive *for vector $v$ if and only if changing the value of the gate input changes the value of the gate output.*

The main idea behind critical path–tracing is to identify paths composed by critical lines starting from primary outputs.

By definition, every primary output is critical under test $v$. Critical path–tracing begins at a primary output and marks as critical all the sensitive inputs of a gate with a critical output [3]. This process is repeated until a primary input is reached. In this manner, critical path–tracing can identify single stuck-at faults detected by test vector $v$.



**(a) Approximation Due to Multiple Path Sensitization**



**(b) Approximation Due to Partial Self-Masking**

**Figure 3.2** Critical Path–Tracing Approximations

However, critical path tracing can result in approximations that will affect diagnosis and, therefore, miss potential error lines. The following example, taken from [60], illustrates this problem.

**Example 9** *In Fig. 3.2(a) and (b), bold lines are critical lines. In Fig. 3.2(a), line c stuck–at 0 is critical but path–trace entering from line j stops at j as none of the fan-ins h and k of the gate that drive j are sensitive. Thus, path–trace procedure will not include line c which is critical. This problem is referred as* multiple path sensitization.

*A similar problem occurs in Fig. 3.2(b). Line b stuck–at 0 is critical but path–trace stops at e since line c stuck–at 1 is not critical. This problem is known as* partial self–masking.

Techniques to alleviate the above problems by separate examination of reconvergent fan–outs have been proposed [43]. However these procedures add to the computational cost of the algorithm. Further, critical path–tracing was developed for single stuck-at faults and not for sets of faults.

Critical path–traceback [60], presented in the next section, alleviates these problems by doing a "conservative" line selection.

### 3.2.2 Critical Path–Traceback

Critical Path–Traceback [60] starts from an *erroneous* primary output that marks as critical. Then it traces backwards towards the primary inputs selecting critical lines as follows (Fig. 3.3): If the output of the gate $G$ has been marked as critical and $G$ has one or more fan–in(s) with controlling values then it randomly selects (marks) as critical any one of them. If $G$ has all fan–ins with non–controlling inputs, then it marks as critical all fan-ins. If a branch is critical, then the algorithm automatically marks critical the stem of the branch.

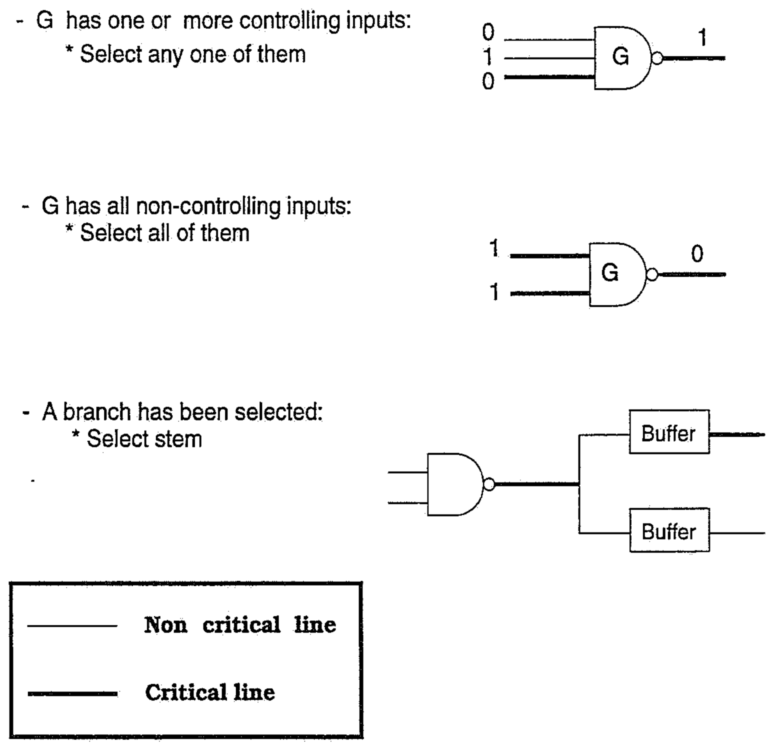


**Figure 3.3** Critical Path–Traceback Line Selection Algorithm

**Example 10** *For the circuit of Fig. 3.2(a), assuming that line $j$ is selected by critical path–traceback, the algorithm can proceed by selecting lines $\{j, h, d, e, c, a\}$. For the circuit of Fig. 3.2(b) and assuming that line $j$ is selected by critical path–traceback, the algorithm proceeds by selecting lines $\{j, h, e, c, a, p, b\}$.*

*Observe that in both cases, the procedure includes critical lines $a$ and $b$.*

Define $V_j^i$ to be the set of lines selected by path–traceback when tracing from erroneous primary output $PO_i$ and vector $v_j$. The following theorem, which we prove here, is crucial for the correctness of our approach.

**Theorem 3** *Let $N$–source correctable design $G_C$ and $L = \{l_1, l_2, \ldots, l_N\}$ be* any *valid correction tuple. If $v_j$ is a vector that activates the inconsistencies, and $PO_i$ is an erroneous primary output for $v_j$, then $V_j^i$ contains at least one element from $L$.*

**Proof.** Let $L'$ be the minimal subset of lines of $L$ so that when their values get complemented for $v_j$ and this difference is propagated towards the primary outputs, $PO_i$ returns to its correct value.

Clearly, by definition of $L'$, there is one or more sensitized paths from each member of $L'$ to $PO_i$. We claim that $V_j^i$ contains some element $l \in L'$.

Let $V_{j,n}^i$ be the set of all lines selected that far by path-traceback at the $n$–th step of the algorithm. For example, for the circuit of Fig. 3.2(b) we have that $V_{v,2}^j = \{j, h\}$. With the use of induction, we will show that for *every* $n$, $V_{j,n}^i$ contains a line of some sensitized path from some $l \in L'$ to $PO_i$. Proving this proves the claim as the algorithm will eventually select $l$ in $V_{j,n}^i$ for some $n \leq$ *max circuit level.*

For the base case $n = 1$, path–traceback does select the erroneous primary output $PO_i$. Let us assume that it holds for $n$ steps, that is, $V_{j,n}^i$ contains line $l'$ of some sensitized path from $l$ to $PO_i$.

To prove that it holds for the next iteration of path–traceback, observe that if $l' \in V_{j,n}^i$ is a branch then the stem will automatically be included in $V_{j,n+1}^i$ by the algorithm. If $l'$ is a fan–out of a gate with multiple controlling fan–ins, then observe that all fan–ins with controlling values need to be changed so that $l'$ changes its value. Thus, every such fan–in will belong to some sensitized path(s) from elements of $L'$ and induction holds for $V_{j,n+1}^i$.

Same reasoning as above shows that induction holds for the case where $l'$ is a fan–out of a gate with all non–controlling fan–ins. This completes the induction and proves the claim.

## 3.3 Single Error Location

In this section we describe our diagnosis method for the single design error case. This case is the simplest one and the method presented here is a direct extension of the results of Section 3.2.2.

How does Theorem 3 translate under the Error Assumption of Section 1.2.2 for the single design error case?

Let $l_e$ be a valid error location. For the single error case, the Error Assumption reduces to the statement that $V_{act}$ is a non–empty set of vectors that activate the inconsistencies. Further, according to Theorem 3, critical path–traceback is guaranteed to include $l_e$ in *every* set $V_j^i$ and for *all* erroneous primary outputs $PO_i$ and vectors $v_j \in V_{act}$. In other words, $l_e$ will exist in the *intersection* of lines of all $V_j^i$'s, that is,

$$l_e \in \bigcap_{PO_i \ erroneous \ for \ v_j} V_j^i, \quad j = 1, \ldots, \mid V_{act} \mid \tag{3.1}$$

This observation is also shown in Fig. 3.4. The box represents the initial error space, that is, all the lines of the circuit. $V_2^1$, $V_4^3$ and $V_2^3$ are three sets of lines selected by path–traceback for 3 different runs of the algorithm. The error set, $C_{error}$, is the shaded area which also is the intersection of the lines of $V_2^1$, $V_4^3$ and $V_2^3$ according to Equation 3.1. Due to Theorem 3, $C_{error}$ in the figure above must contain all single valid modification locations.

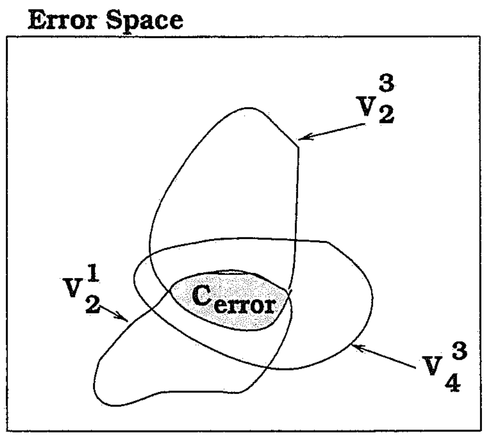


**Figure 3.4** Space Pruning For Single Error Diagnosis

It can be seen from Fig. 3.4 that every run of path–traceback for a different set of erroneous primary output and input vector has the potential to reduce the error space but never increases it. Diagnosis for single design errors with implicit enumeration of error pairs is shown in Fig. 3.5. Initially, we quickly reduce the error space by applying path–traceback for the vectors of $V_{act}$. Once the algorithm has deleted the majority of error candidates, Inverted Simulation is performed (Section 2.3.2) to reduce the error space further.

## 3.4 Multiple Design Error Diagnosis

For a design $G_C$ corrupted by multiple errors, Theorem 3 says that every run of the path–trace back procedure is guaranteed to include *at least* one line of each valid error

```
IMPLICIT_SINGLE_ERROR_DIAGNOSIS( $F_C$, $G_C$, $V_{act}$)
1.  $C_{error} = \{G_C \textit{ circuit lines}\}$
2.  for every vector $v_j \in V_{act}$ do
3.      for every erroneous $PO_i$ do
4.          $C_{error} = C_{error} \cap$ PATH_TRACE_BACK($v_j, PO_i$)
5.  for every tuple ($l_0$, $l_1$, ..., $l_{N-1}$) in $C_{error}$ do
6.      INVERTED_SIMULATION( $F_C$, $G_C$, $l_0$, $l_1$, ..., $l_{N-1}$)
7.  return($C_{error}$)
```

**Figure 3.5** Implicit Diagnosis for Single Design Errors

tuple. However, since the different $V_i^j$ sets might contain different element(s) of different error tuple(s), Equation 3.1 does not hold any longer.

In this section, we define the concept of an **Intersection Graph** originally presented in [60] for bridging fault diagnosis. This graph is a data structure useful in keeping track of the different kind of information that path–trace back has to offer. We also define a novel concept of a $N$–**graph reduction** that will allow us to direct our search for valid $N$–error tuples in the potential error space of an $N$–source correctable design.

**Definition 11** *The* **Intersection Graph(IG)** $G = (V, E)$ *of a* $G_C$ *is an undirected graph where each vertex* $V_i \in V$ *contains a set of lines from* $G_C$ *denoted as* $lines(V_i)$*. Edge* $(V_i, V_j)$ *is in* $E$ *if and only if* $lines(V_i) \cap lines(V_j) \neq \emptyset$*.*

**Example 11** *The intersection graph of Fig. 3.6 contains 6 vertices* $V_1, V_2, \ldots, V_6$*. Each vertex contains some lines from the circuit of Fig. 3.2(b) and two vertices are adjacent if and only if the intersection of their line sets is non–empty according to Def. 11.*

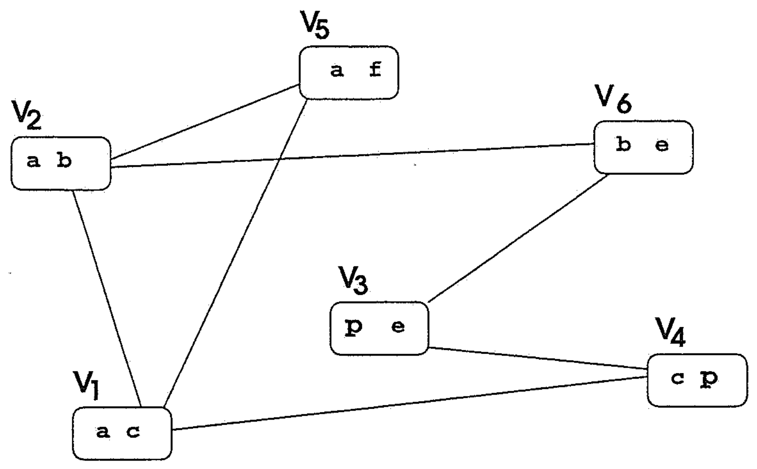


**Figure 3.6** An Example of an Intersection Graph

**Definition 12** *Let IG $G = (V, E)$ and let $V_R = \{V_1, V_2, \ldots, V_N\}$, $N > 1$, be $N$ distinct vertices of $G$ such that $\forall i, j, 1 \leq i, j \leq N$ and $i \neq j$, we have that $lines(V_i) \cap lines(V_j) = \emptyset$. For every $V_i \in V_R$ let $V_i^{adj}$ be the set defined as follows:*

$$V_i^{adj} = \{V_{i_j} \; : \; V_{i_j} \in V \textit{ is adjacent to } V_i \textit{ and not adjacent to any vertex of } V_R \; - \; V_i\}$$

*We define an $N$–***graph reduction** *that respects $V_R$ on $G$ the new IG $G' = (V', E')$ that we get if $\forall i, 1 \leq i \leq N$, we replace every set $V_i^{adj} = \{V_{i_1}, V_{i_2}, \ldots, V_{i_k}\}$ with a new node $V_i'$ where $lines(V_i') = lines(V_i) \cap lines(V_{i_1}) \cap lines(V_{i_2}) \cap \ldots \cap lines(V_{i_k})$ and compute the new edge adjacencies of $G'$.*

**Definition 13** *An IG $G = (V, E)$ is called $N$–***reducible** *if and only if there exists an $N$-graph reduction on $G$.*

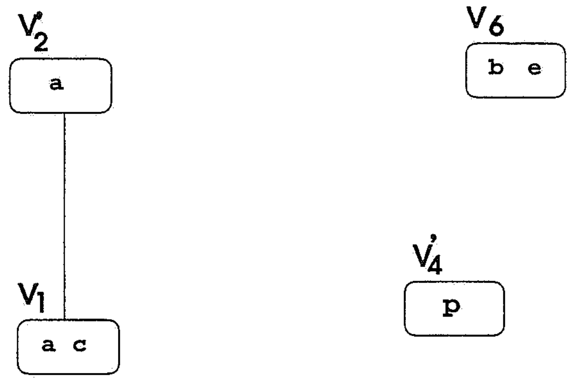


**Figure 3.7** A Reduced Intersection Graph

**Example 12** *Fig. 3.7 shows the resulting graph when a 2–graph reduction that respects $V_R = \{V_2, V_4\}$ is performed on the IG of Fig. 3.6. In detail, $V_2^{adj} = \{V_5\}$ and $V_4^{adj} = \{V_3\}$. Observe that $V_1 \notin V_2^{adj}$ because it is adjacent to $V_4$.*

*According to Def. 12 we have that $lines(V_2') = lines(V_2) \cap lines(V_5) = \{a\}$ and $lines(V_4') = lines(V_4) \cap lines(V_3) = \{p\}$. Notice that the reduced graph of Fig. 3.7 is both 2 and 3 reducible.*

### 3.4.1 Pruning the Error Space Through Graph Reductions

In this section we describe how we can use the concept of a reducible IG so that we prune the error space. We first give a procedure to construct and process an IG for an erroneous design $G_C$ and then we present some interesting properties of this data–structure.

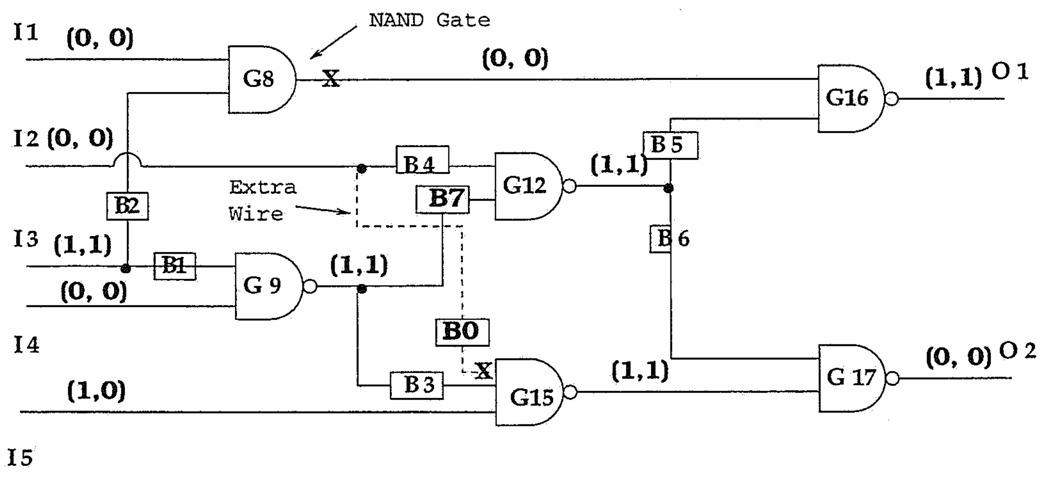


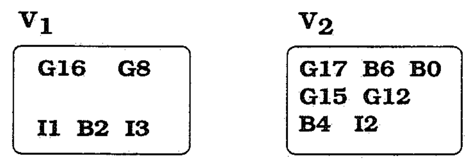
**Figure 3.8** Erroneous Implementation for Example 13

**Definition 14** *Let $G_C$ be a N–source correctable design. Define IG $G^0 = (V^0, E^0)$ to be the initial IG. Every vertex in $V^0$ is the set of lines from a run of the path–traceback procedure for a different erroneous primary output $PO_i$ and a different vector $v_j \in V_{act}$. Define $G^i = (V^i, E^i)$, $i > 0$ to be the resulting IG after $i$ consecutive N–graph reductions on $G^0$ where an arbitrary number of vertex additions from path–trace back can interleave the reductions.*

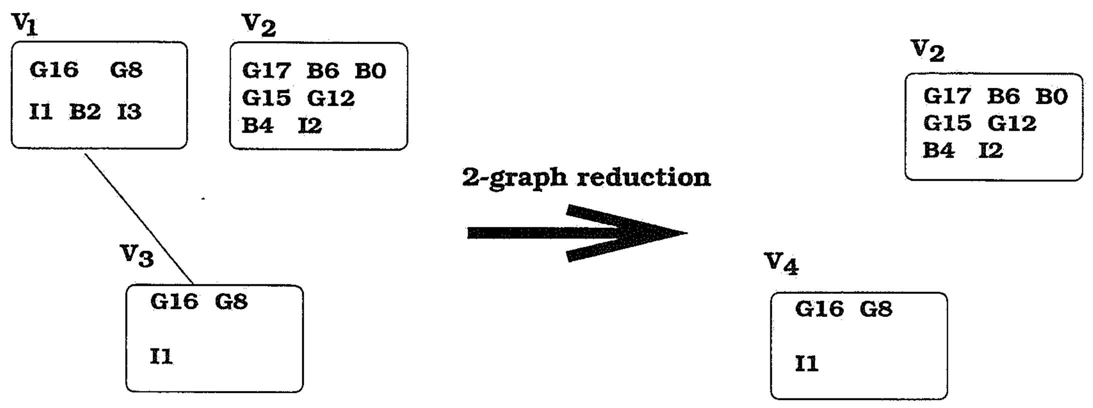
The following examples illustrate the construction of the IG for two erroneous circuits.

**Example 13** *Fig. 3.8 contains the erroneous circuit of Example 4 (Section 2.3.1) simulated for two input vectors* $(0, 0, 1, 0, 1)$ *and* $(0, 0, 1, 0, 0)$. *The values of the vectors at respective lines are shown within parentheses.*

*The first vector produces erroneous results for both primary outputs, while the second vector activates the inconsistencies at primary output $G_{16}$. Fig. 3.9(a) contains the initial*



**(a) $G^0$ path-trace back for first erroneous vector**

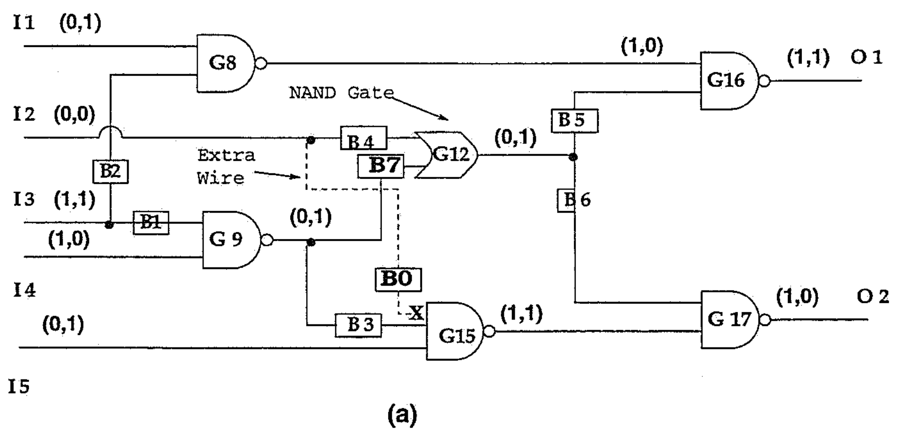


**(b) $G^0$ after path-trace back for second vector (left) and $G^1$ after a 2-graph reduction (right).**

Figure 3.9 Example of Circuit Graph Construction and Graph Reduction

*IG $G^0$ when path–traceback is performed from* each *of the erroneous primary outputs for the first input vector. $V_1$ is the vertex created when path–traceback originates from output $G_{16}$ and $V_2$ when path–traceback starts from $G_{17}$.*

*The left side of Fig. 3.9(b) contains the situation when path–trace is performed for the second input vector and erroneous primary output $G_{17}$. Vertex $V_3$ is added to $G^0$ and it is adjacent to $V_1$. The result of this vertex addition is the existence of a 2–graph reduction that respects $V_R = \{V_1, V_2\}$ (or, equivalently, $V_R' = \{V_3, V_2\}$). The result of this* *reduction is shown to the right of Fig. 3.9(b) where vertices $V_1$ and $V_3$ have been replaced by $V_4$ where $lines(V_4) = lines(V_1) \cap lines(V_3)$.*



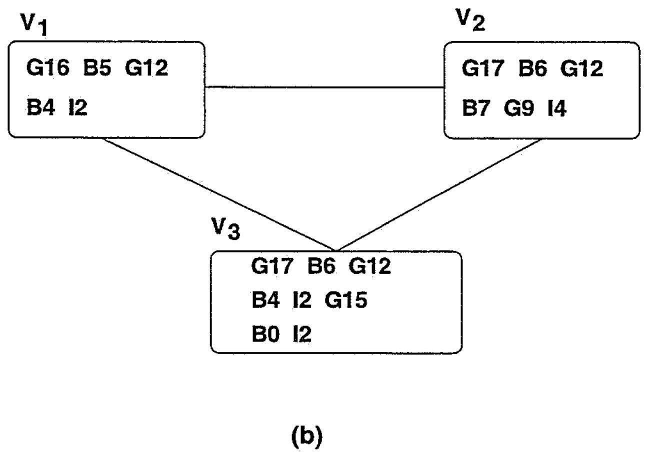


**Figure 3.10** Erroneous Implementation and IG for Example 14

**Example 14** *Consider the erroneous circuit of Fig. 3.10(a) where $G_{12}$ is a* NAND *gate replacement error and the connection $I_2$–$G_{15}$ is an extra wire error. The circuit is simulated for two input vectors, $(0,0,1,1,0)$ and $(1,0,1,0,1)$ and the values are shown in parenthesis above each line. The first vector produces erroneous results in both primary outputs, while the second vector produces erroneous results at $O_2$.*

*The resulting IG is a clique and it is shown in Fig. 3.10(b). $V_1$ and $V_2$ are the resulting nodes when path–trace back begins from $O_1$ and $O_2$ for the first vector, respectively. $V_3$ is the IG node that results from path–trace back for the second vector.*

The following theorem is important for the correctness of our diagnosis approach. We should emphasize the fact that the theorem holds for $N$–graph reductions on an IG for an $N$–source correctable design only (Def. 14). We will have the chance to discuss the implications of this requirement shortly after the proof of the theorem.

**Theorem 4** *Let $G_C$ be an $N$–source correctable design. Every vertex of $G^i = (V^i, E^i)$, where $i \geq 0$ contains at least one line from each valid modification tuple. Moreover, $G^i$ can have* at most *$N$ vertices non–adjacent to each other.*

**Proof.** Initially, we show that if the theorem holds for $G^i$ then it also holds after a vertex has been added to $G^i$.

If every vertex of $G^i$ contains at least one line from each valid modification tuple, adding a new vertex from path–trace back will not violate this property (Theorem 3) and the first part of Theorem refig1thm holds. The second part of the theorem also holds for suppose, towards a contradiction, that there was a set $V_{big}$ of $N+k$, $k > 0$, non–adjacent vertices after a vertex addition occurred on $G^i$. Then, since every vertex contains at least one element of every valid modification tuple and $G_C$ is $N$–source correctable, there should be at least two vertices of $V_{big}$ that are adjacent, a contradiction.

Now we show that the theorem holds for any number of $N$–graph reductions. We will prove this with the use of induction on the number of reductions.

For $i = 0$ the theorem holds as every vertex in $V^0$ contains at least one element of every valid modification tuple due to Theorem 3. Furthermore, $G^0$ cannot have more than $N$ non–adjacent vertices because assume, towards a contradiction, that there were $N+k$, $k > 0$, vertices $V_1, V_2, \ldots, V_{N+k}$ non–adjacent to each other. Let $L = \{l_1, l_2, \ldots, l_N\}$ be a set of modification locations. Theorem 3 implies that each vertex contains at least one distinct line of $L$ there should be at least two vertices in $V^i$ that contain the same line and, thus, are adjacent, a contradiction.

Assume that the theorem holds for $G^n$, we'll prove that it holds for $G^{n+1}$ for a reduction respecting set $V_R$.

To prove that the first part of the theorem holds for $G^{n+1}$, observe that each set $V_i^{adj}$, $V_i \in V_R$, should contain the same line from each valid modification tuple, otherwise there would be two vertices $V_i, V_j, i \neq j$ of $V_R$ that are adjacent. Since every vertex in $V_i^{adj}$ contains the same line of every valid modification tuple, these lines should also appear in their intersection.

To prove the second part of the theorem, assume that $G^{n+1}$ has $N + k, k \geq 1$ non–adjacent vertices $V = \{V_1, V_2, \ldots, V_{N+k}\}$. Since each vertex contains at least one line from each valid modification tuple (first part of this theorem proved in previous paragraph) and the design is $N$–source correctable, this implies that there would be *at least* two vertices in $V$ that are adjacent, a contradiction.

Observe that the above theorem also gives a *lower bound* on the number of modifications needed to rectify an erroneous $G_C$; if $G^i$, for some $i$, has $N$ non–adjacent vertices, then the design is guaranteed not to be $K$–source correctable for $K < N$ due to Theorem 4. However, the design might be $M$–source correctable for some $M > N$ since,

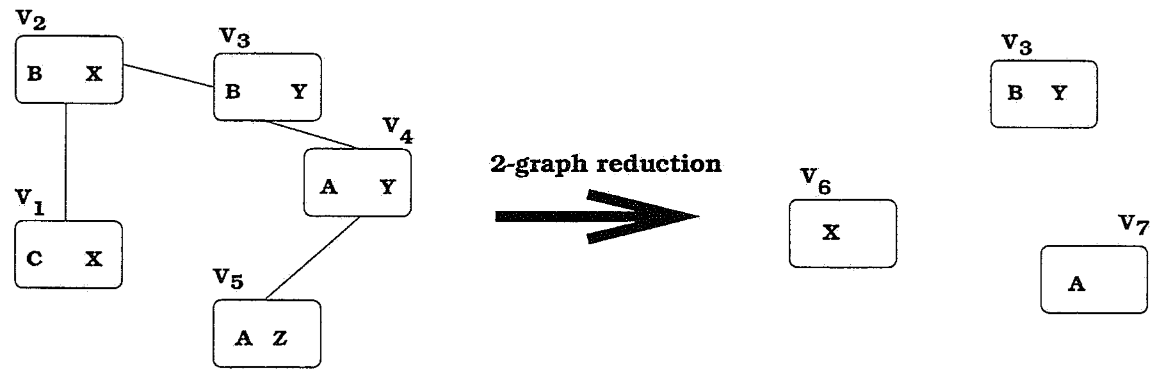


**Figure 3.11** A 2–Graph Reduction on a 3–Source Correctable Design

according to the theorem above, an $M$–source correctable design can have *at most* $M$ non–adjacent vertices.

It was emphasized earlier that Theorem 4 holds only if we perform $N$–graph reductions on an $N$–source correctable design. What happens if we waive this requirement and allow a $K$–graph reduction on an $N$–source correctable design for $K < N$? Unfortunately, as the following example illustrates, Theorem 4 does not longer hold and we may jeopardize the error resolution.

**Example 15** *The IG on the left of Fig. 3.11 is an IG for some* 3*–source correctable design. Assume that* $(A, B, C)$ *and* $(X, Y, Z)$ *are two valid modification location triples.*

*Suppose we perform a reduction that respects* $V_R = \{V_2, V_4\}$*. The resulting graph is shown in the right side of the figure. The reduction on* $V_2$ *deletes lines* $B$ *and* $C$*, while the reduction on* $V_4$ *deletes lines* $Y$ *and* $Z$ *and Theorem 4 does not longer hold.*

**Theorem 5** *Let $G^i = (V_i, E_i)$ for $N$–source correctable $G_C$. If we perform a $K$–graph reduction, $K < N$ then Theorem 4 does not necessarily hold.*

**Proof.** Immediate from Example 15.

### 3.4.2 Implicit Enumeration

In this section we describe our implicit design error tuple enumeration diagnosis algorithm. The question that we address is how to enumerate a set of valid error tuples $C_{error}$ given an IG $G^i$.

Given an IG $G^i$, the first action is to find the maximum number $K$ of non–adjacent vertices. As explained in the previous section, $K$ also serves as a lower bound on the *necessary* number of modifications to correct the design. However, given $G^i$, the design can be $N$–source correctable for any $N > K$. Therefore, in our discussion and final algorithm, an initial guess $N$ of the potentially desired number of modifications is required. This guess $N$ should always be bigger than or equal to $K$.

**Definition 15** *Given an IG $G^i = (V^i, E^i)$ we define an $n$–***sample of** $V^i$, *$n \leq | V^i |$, to be the union of lines of $n$ randomly picked vertices of $V^i$.*

Depending on the structure of $G^i$, a different error enumeration technique for $C_{error}$ is used. These techniques are as follows:

- $\underline{K = 0}$: If $K = 0$ it implies that $G^i$ is a *clique*, that is, there exists an edge between every two vertices of $G^i$. To compile $C_{error}$, we pick the vertex $V \in V^i$ with the *smallest* number of lines. Observe that $line(V)$ is guaranteed to contain one line from each valid modification location due to Theorem 4.

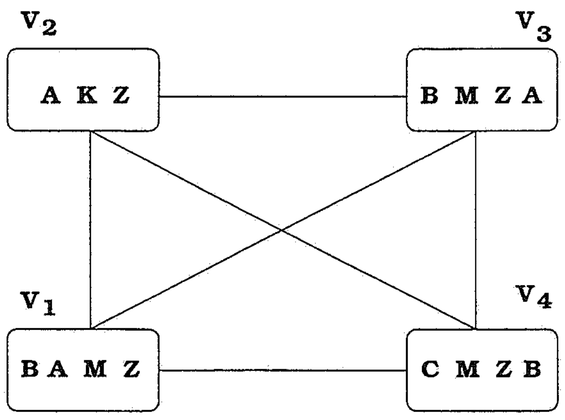


**Figure 3.12** Subgraph of a Clique IG

We then pick an $n$–sample of $V^i$ for some small $n$. In our experiments, $n$ is usually less than 10. Finally, we *exhaustively* compile $N$–tuples where the first element of each tuple is a line from $line(V)$ and the remaining $N-1$ elements are from the union of the lines in $line(V)$ and the $n$-sample.

- $\underline{K > 0}$: If $K > 0$ it means that there is at least one set of $K$ non–adjacent vertices $\{V_1, V_2, \ldots V_K\}$ in $V^i$. To compile $C_{error}$ for this case, we pick an $n$–sample of $V^i$ for some small $n$. Subsequently, we *exhaustively* compile $N$–tuples where the $j$–th entry of the first $K$ elements of each tuple is taken from $line(V_j)$ and the remaining $N-K$ elements are selected from the $n$–sample [2].

[2]Observe that the $n$–sample is an empty set when $N = K$.

Then for *every* set of $K$ non–adjacent vertices of $G^i$, $\{V_1', V_2', \ldots, V_K'\}$, we delete the error tuples from $C_{error}$ that don't have a subset of $k$ lines, each of them in some distinct $line(V_l')$.

**Example 16** *Implicit enumeration for the IG of Fig. 3.9(b) yields 21 error pairs as* $K = N = 2$. *These pairs are:* $(G_{16}, G_{17}), (G_{16}, B_6)$, $(G_{16}, B_0)$,$(G_{16}, G_{15})$,$(G_{16}, G_{12})$, $(G_{16}, B_4)$,$(G_{16}, I_2)$,$(G_8, G_{17})$, $(G_8, B_6)$,$(G_8, B_0)$,$(G_8, G_{15})$, $(G_8, G_{12})$,$(G_8, B_4)$,$(G_8, I2)$, $(I_1, G_{17})$,$(I_1, B_6)$,$(I_1, B_0)$ $(I_1, G_{15})$,$(I_1, G_{12})$,$(I_1, B_4)$ *and* $(I_1, I2)$.

*Equivalently, for the clique of Fig. 3.10(b) we have a total of 40 pairs if we pick as a* 1*–sample vertex* $V_3$ *and 30 pairs if the* 1*–sample is* $V_2$. *In both cases, the vertex with the smallest line set is* $V_1$.

Assume that $G_C$ is indeed $N$–source correctable. Unless $K = N$, the resolution of the implicit error enumeration method described above clearly depends on the $n$–sample. The importance of a good $n$–sample is demonstrated in the example that follows.

**Example 17** *Fig. 3.12 contains a subgraph of a clique IG for some* 3*–source correctable design. Clearly,* $K = 0$. *Assume that* $(A, B, C), (M, N, K)$ *and* $(X, Y, Z)$ *are three modification tuples. If* $n = 2$ *and our* 2*–sample is* $V_1$ *and* $V_3$ *(*$V_2$ *is the smallest element) then* $C_{error}$ *will not contain none of the above modification tuples. A good* 2*–sample is* $V_1$ *and* $V_4$.

How do we know if we picked a good $n$–sample? Fortunately, Inverted Simulation, an exhaustive on the error space procedure for the set of vectors $V_{act}$, provides an answer to this question.

Suppose that we compile $C_{error}$ in the forms of $N$–error tuples for some erroneous $G_C$ that we suspect it is $N$–source correctable as described above. If $C_{error}$ contains one or more valid corrections, then these corrections should qualify during Inverted Simulation.

If $C_{error}$ is empty after Inverted Simulation it means that it did not contain any valid correction and we repeat the implicit $C_{error}$ enumeration process for a different, and possibly larger, $n$–sample. If after many iterations of this algorithm $C_{error}$ remains empty, then we can conclude with high confidence that $G_C$ is $M$–source correctable for some $M > N$.

```
IMPLICIT_ERROR_DIAGNOSIS( $F_C$, $G_C$, $N$, $n$, $iters$)
1.   for every vector $v_j \in V_{act}$ do
2.       for every erroneous $PO_i$ do
3.           Place $V_j^i$ = PATH_TRACE_BACK($v_j$, $PO_i$) in $V_{all}$
4.   $V'_{all} = V_{all}$
5.   while $V'_{all} \neq \emptyset$ do
6.       Randomly choose $V_j^i \in V'_{all}$ and add it in IG G
7.       Delete $V_j^i$ from $V'_{all}$
8.       while there are $N$-graph reductions in $G$ do
9.           REDUCE_IG($G$, $N$)
10.  $C_{error}$=IMPLICIT_ERROR_ENUMERATION($N$, $n$)
11.  for every error tuple ($l_0$, $l_1$, ..., $l_{N-1}$) in $C_{error}$ do
12.      INVERTED_SIMULATION($F_c$, $G_c$, $l_0$, ..., $l_{N-1}$)
13.  if $C_{error} = \emptyset$ goto line 4 unless max # of iterations $iters$ is reached
14.  return($C_{error}$)
```

**Figure 3.13** Implicit Diagnosis for Multiple Design Errors

### 3.4.3 Multiple Error Location

We are now ready to describe the overall error diagnosis approach with implicit enumeration of error tuples for $N > 1$. The case where $N = 1$ is covered in Section 3.3.

The multiple diagnosis algorithm is shown in Fig. 3.13. The input to the algorithm are the specification, implementation, a guess $N$ of the design correctability, a number $n$ for the random sample and a maximum number of iterations $iters$ the procedure should be repeated if $C_{error}$ is empty.

In lines 1–3 we compile a set of distinct vertices $V_{all}$ from consecutive path–trace back runs for all vectors of $V_{act}$ and all erroneous primary outputs. The IG graph is built in lines 5–9; vertex insertions (line 6) are followed by $N$-graph reductions (lines 8–9) until $V'_{all}$ is empty and no more reductions are possible.

The error set $C_{error}$ is created in line 10 in the way described in Section 3.4.2. Every invocation of IMPLICIT_ERROR_ENUMERATION() uses a different sample.

If $C_{error}$ becomes empty during Inverted Simulation (lines 11–12) then we repeat the process (line 13) unless we reached the maximum number of iterations $iters$ and we exit with an empty error set.

In our implementation, the procedure of Fig. 3.13 is applied on the checkpoints of the circuit $G_C$. Due to Theorem 2 from Section 2.3.4, the theory developed in this chapter also holds for the checkpoints of the circuit alone. In our experiments, once the algorithm of Fig. 3.13 terminates for the checkpoints of $G_C$ with output the set $C_{error}$, we create a new $C'_{error}$ from the clans of the checkpoint of $C_{error}$. Then we run Inverted Simulation on the elements of $C'_{error}$ so we get the final set of modification tuples.

### 3.4.4 Implementation Issues

In this section, we discuss an efficient implementation of the multiple design error diagnosis algorithm and present heuristics that improve error resolution.

- We represent the IG in an adjacency list format where the set $lines(V_j^i)$ of every vertex $V_j^i$ is being kept as a bit–vector; the $k$–th entry of the vector is 1 if and only if the $k$–th line of $G_C$ is contained in $V_j^i$. This makes it efficient to perform reductions and do implicit error enumeration through the use of the bit–wise operations AND and OR [60].

- The complement $\overline{G}$ of $G$ is also maintained. This makes it efficient to find the non–adjacencies used for reduction and implicit enumeration.

- Recall from Section 3.2.2 that path–trace back has sometimes to make a choice on the line to select for the next iteration. A different selection of lines gives different, and possibly better, results.

  In our implementation, we keep a **reference counter** on each line of the circuit. This counter is initially equal to zero. Every time a line is selected by path–trace back, the respective reference counter is incremented; every time the line is deleted due to a graph reduction the reference counter gets decremented for *all* instances the line is present. When path–trace back has to select from many lines, it selects the one with the lowest reference–counter. This line is less likely to to introduce adjacencies into the IG.

If all lines have the same reference counters, then the one that is closest to the primary inputs is selected. This line has the potential to create a shorter path from the primary outputs to the primary inputs during path–trace back.

Finally, if one of the choices is a branch whose stem has already been selected by path–trace, we select the branch since it adds nothing to the selected set of lines.

## 3.5 Error Masking

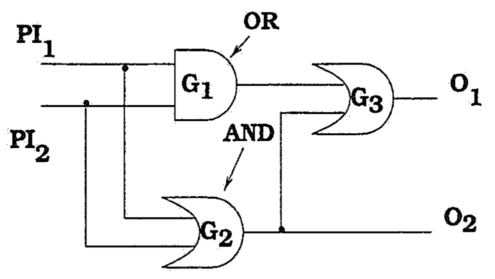
Consider the simple circuit of Fig. 3.14(a). This circuit is corrupted by two design errors; $G_1$ should be an OR gate and $G_2$ should be an AND gate. Nevertheless, the error on $G_1$ is not observable [3] at no primary output. However, observe that the error on $G_1$ is observable when the error on $G_2$ is corrected (Fig. 3.14(b)). This situation is referred in the literature as **error masking** [10][51].

Error masking has not been a problem in our presentation so far because of the Error Assumption of Section 1.2.2 that required that at least one sensitized path exist from the error location to some primary output [4]. However, the implication of error masking on multiple DEDC is important.

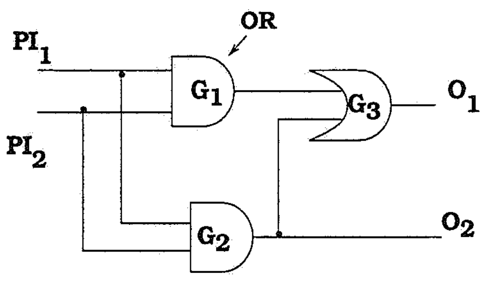
Observe that the circuit of Fig. 3.14(a) is 2–source correctable, but every algorithm that will perform a single rectification will fail because the circuit of Fig. 3.14(b) is still

[3] See section 2.3.2

[4] The other implication if we drop the Error Assumption is that $V_{act}$ might not contain vectors that activate and propagate *all* design error to some primary output(s). The experiments in this thesis and [27] show that a relatively small size of input test vectors used for verification is sufficient for Error Assumption to hold. However, there is no theoretical foundation to support the argument. For a discussion of how to have a $V_{act}$ via symbolic methods so that the Error Assumption holds, see Section 4.3.3.



(a) Error at $G_1$ masked by error at $G_2$



(b) Error $G_1$ observed at output $O_1$

**Figure 3.14** Example of Error Masking

erroneous. On the other hand, no algorithm, to our knowledge, will try to attack the design as a 2–source correctable since the error on $G_1$ is not observable.

If a design fails to be $N$–source correctable for small values of $N$ it might be because of error masking. In our experiments, error masking did not occur in 2–correctable designs and it was rare in 3–correctable design. In less than 10% of the cases for the ISCAS'85 benchmark circuits C3540, C5315, and C7552 it happened for an error to be masked for all input vectors by some other design errors. Our intuition and experiments suggest that error masking is more likely to occur as the value of $N$ increases.

A solution to this problem with our diagnosis methodology is to track the erroneous primary outputs for *each* input vector of $V_{act}$. Diagnosis is performed as explained in Chapters 2 or 3 and if some simulation of a vector $v \in V_{act}$ for Inverted Simulation yields new erroneous primary outputs, diagnosis is performed for *all* such new outputs.

**Example 18** *Line $G_2$ qualifies Inverted Simulation for input vector $(0,1)$ as it corrects $O_2$. However, $O_1$ will give a new erroneous primary output response for Inverted Simulation on $G_2$. Since there's no sensitized path when we complement the value of $G_12$ to $O_1$ for vector $(0,1)$, the erroneous response on $O_1$ should come from an error previously masked.*

Multiple diagnosis steps obviously add to the computational cost of the algorithm but they are necessary so that we can handle error masking.

## 3.6 Summary

In this chapter we presented a multiple design error diagnosis approach that performs an implicit enumeration of error pairs. This eliminates the exponential explosion of error space (Eq. 1.1) but it does not guarantee, on a theoretical level, to return *all* valid modification tuples, that is, it is not exhaustive on the error space. This happens because the set of error tuples returned sometimes depends on a random sample.

The algorithm is based on a unique path–traceback algorithm, similar to path–tracing used for diagnosis of stuck-at faults. The information produced by path–trace back is used to construct a graph. Different graph operations allow us to process this graph and reduce the error space dynamically. Finally, we use the processed graph and the concept of an $n$–sample to implicitly enumerate $N$–error tuples.

In the last part of this chapter, we discuss the implication of error masking in the area of multiple DEDC and we propose an approach to a solution for this problem.

# CHAPTER 4

# Design Error Correction

## 4.1 Introduction

The purpose of diagnosis is to reduce the number of potential error tuples significantly. Once this is achieved, circuit rectification needs to be performed. In this Chapter, we describe two Design Error Correction techniques; the first uses *test–vector simulation* and the second is *symbolic.*

Both methods are exhaustive on the correction space in the sense that they will not miss a correction if such correction exists in the modification model of Def. 3. This is because the symbolic method runs over the one based on test–vector simulation which is exhaustive on the error space by construction.

The symbolic method is also exact on the correction space. This means that there is a guarantee that each member of the output list of corrections will successfully rectify the design. The experimental results in Chapter 5 suggest that test–vector simulation gives exact on the correction space results too although there is no theoretical foundation to prove it. Since the simulation results are based on a small fraction of the input vector space, there is no guarantee that they will correct the circuit for *all* vectors. Nevertheless, such a method is still useful and interesting to explore because once the list of potential corrections has been narrowed down to a very small number, an appropriate verification tool can be used to check the correctness of the new implementation for each such correction.

## 4.2 Correction with Test Vector Simulation

In this section we describe a run–time efficient and exhaustive on the correction space test vector simulation technique for multiple design error correction. The technique compiles *exhaustively* the correction list and verifies it with random vectors along with the vectors for stuck-at faults that did not activate the inconsistencies during diagnosis.

More in detail, a list of all possible corrections from Def. 3 and for every remaining candidate of $C_{error}$ is compiled *exhaustively.*

Recall that during the Inverted Simulation step we keep track of potential $N$–error line tuples in the form of the following equation:

$$L = \{l_0,\ l_1,\ \ldots,\ l_{N-1},\ not(k),\ v\}$$

The above tuple implies that error location tuple $\{l_0,\ l_1,\ \ldots,\ l_{N-1}\}$ can correct vector $v \in V_{act}$ if the $Fgroup$ bit–list value for $v$ gets complemented (retains its value) for line $l_i$ when the $2^i$–th bit in the error excitation scenario $not(k)$ is equal to 1 (0).

The main idea behind our error correction scheme is to find *all* possible corrections from Def. 3 for *each* line $l_i$ so that when applied, they produce $Fgroup$ values that respect at least one of the error excitation configurations for *every* input vector $v \in Vact.$

The theorem that follows formalizes this idea.

**Theorem 6** *Let input vector $v \in V_{act}$ and let $\{l_0,\ l_1,\ \ldots,\ l_{N-1}\}$ be a suspicious set of error lines for error configurations $not(k_1), not(k_2), \ldots, not(k_m)$ of an $N$–source cor-*

*rectable circuit. In other words, the following $N-$error line tuples are part of the output of Inverted Simulation:*

$$\{l_0,\ l_1,\ \ldots,\ l_{N-1},\ not(k_1),\ v\}$$

$$\{l_0,\ l_1,\ \ldots,\ l_{N-1},\ not(k_2),\ v\}$$

$$\ldots$$

$$\{l_0,\ l_1,\ \ldots,\ l_{N-1},\ not(k_m),\ v\}$$

*Let $C_0, C_1, \ldots C_{N-1}$ be a set of corrections that rectify the design. Then the value of $l_i$, $\forall i,\ 0 \leq i \leq N-1$, under simulation of vector $v$ when $C_i$ is applied should be in the set:*

$$\{2^i\ AND\ not(k_1), 2^i\ AND\ not(k_2), \ldots, 2^i\ AND\ not(k_m)\}$$

*Equivalently, the new value of $l_i$ for $v$ can be either a 0, or a 1, or a don't care ($X$).*

**Proof.** Immediate from the way Inverted Simulation works (Fig. 2.8).

### 4.2.1 Wrong Gate Correction

Suppose that we are considering a Wrong Gate type correction for a line $l_i$ of some error tuple and let erroneous vector $v \in V_{act}$ ($v \in Fgroup$, equivalently). The new gate configuration that will drive $l_i$, when applied, should produce a new value on $l_i$ for $v$ that exists in the union of the excitation configurations that rectify the design according to Theorem 6. Moreover, this condition should hold for *all* vectors $v \in V_{act}$.

The following example illustrates the above observations.

**Example 19** *Correction is applied for the pair of error locations* $\{G_1, G_2\}$ *shown in Fig. 4.1(a) and error tuples*

$$(G_1,\ G_2,\ 1,\ v_1)$$

$$(G_1,\ G_2,\ 1,\ v_2)$$

$$(G_1,\ G_2,\ 3,\ v_2)$$

*Excitation configuration* 1 *for vector* $v$ *implies that* $v$ *produces correct results at the primary outputs if the potential error on* $G_1$ *is activated and the error on* $G_2$ *is not. Equivalently, excitation configuration* 3 *for vector* $v$ *means that both errors on* $G_1$ *for this error tuple during Inverted Simulation for* $v$ *are activated.*

*Assume that the correction pair under consideration is a missing gate for* $G_1$ *and a gate replacement error for* $G_2$*. This is shown in Fig. 4.1(b), where* $G' \neq G_1$ *and* $G_2 \neq G_3$*. Moreover, assume that when we apply this correction pair and simulate according to the fan–in values shown in Fig. 4.1(a) we get the new fan–out values of the shaded boxes in Fig. 4.1(b).*

*We observe that the new values for* $v_1$ *are* $\{G_1, G_2\} = \{0, 1\}$ *and they respect excitation scenario* 1 *and Theorem 6 . The same holds for the second vector* $v_2$ *and this correction pair qualifies. Observe that if the error tuple*

$$(G_1,\ G_2,\ 3,\ v_2)$$

*did not exist in the original list, then the above correction pair would not qualify and it would have been removed from the list of potential corrections.*

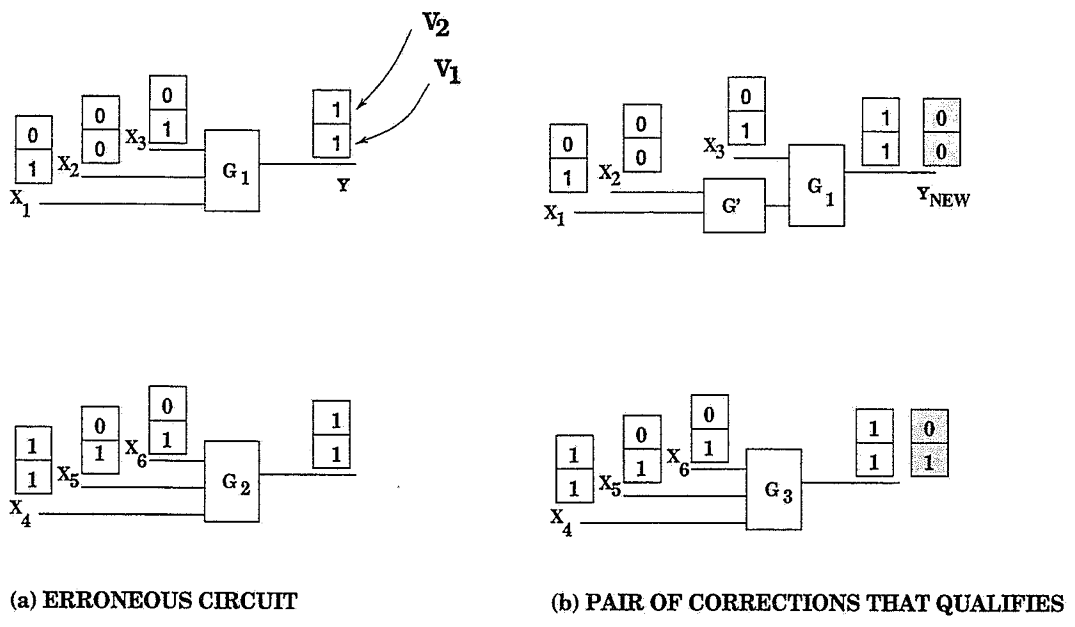


**Figure 4.1** Wrong Gate Correction on an Error Pair

### 4.2.2 Wrong Wire Correction

Wrong Wire Corrections are performed in a similar manner to the one described above as they also obey Theorem 6. The only additional work for wire related corrections is that we need to consider wires that do not create combinational loops.

Once we satisfy this requirement for a particular candidate, we proceed exhaustively to compile the list of Wrong Wire Corrections.

### 4.2.3 Wrong Gate/Wrong Wire Correction

For corrections of the **Wrong Gate/Wrong Wire** case, we compile an exhaustive list of all Wrong Gate corrections from Def. 3 and apply the techniques of the previous paragraphs on each candidate correction of this list.

```
TVS_CORRECTION($C_{error}$, $V_{act}$)
1.  for every distinct set of error lines $L = \{l_0, l_1, \ldots, l_{N-1}\} \in C_{error}$
2.      Let $v_i$ be a vector of $V_{act}$ and
        $not(k_1), not(k_2), \ldots, not(k_m)$ all excitation scenarios of $L$ for $v_i$
3.      $Corr_L$ = TVS_CORRECT_VECTOR($L, not(k_1), not(k_2), \ldots, not(k_m), v_i$)
4.      for every vector $v_j \in V_{act}, j \neq i$, do
5.          Let $not(k_1), not(k_2), \ldots, not(k_n)$ all excitation scenarios of $L$ for $v_j$
5.          $Corr_L = Corr_L \cap$ TVS_CORRECT_VECTOR($L, not(k_1), not(k_2), \ldots, not(k_n), v_j$)
6.  Let $Corr = \cup Corr_{L_l}$ for every distinct set of error lines $L_l$.
7.  Verify every member of $Corr$ with stuck-at/random vector simulation
```

```
TVS_CORRECT_VECTOR( $l_0$, $l_1$, ..., $l_{N-1}$, $not(k_1)$, $not(k_2)$, ..., $not(k_m)$, $v$)
1.  for $i = 0$ to $N-1$ do
2.      Apply $Wrong\ Gate$ correction on $l_i$
3.      Compute new_fan-out_value($l_i$) for $v$
4.      if new_fan-out_value($l_i$) $\in \{2^i ANDnot(k_1), 2^i ANDnot(k_2), \ldots,$
        $2^i ANDnot(k_m)\}$ then
5.          add correction in $Corr_{l_i}$
6.      Apply $Wrong\_Wire$ correction on $l_i$ (when necessary, consider
        every line $l' \notin$fan-out_cone($l_i$))
7.      Compute new_fan-out_value($l_i$) for $v$
8.      if new_fan-out_value($l_i$) $\in \{2^i\ AND\ not(k_1), 2^i\ AND\ not(k_2), \ldots,$
        $2^i\ AND\ not(k_m)\}$ then
9.          add correction in $Corr_{l_i}$
10. Let $Corr_v$ be the cartesian product $Corr_{l_1}\ X\ Corr_{l_2}\ X \ldots X\ Corr_{l_{N-1}}$
11. return($Corr_v$)
```

**Figure 4.2** Design Error Correction With Simulation

### 4.2.4 Overall Correction Strategy

Once the list of potential corrections has been compiled exhaustively as described in the previous three sections, an additional verification step is performed.

This step uses the vectors that did not activate the inconsistencies in the first place (*Tgroup* values, see section 2.3.2). For every set of potential corrections, the corrections are applied and simulation is performed at the fan-out cones of the respective lines. This step can be carried efficiently with parallel bitwise test–vector simulation [66].

The candidate corrections that give an erroneous response at the primary outputs of the circuit are deleted. The set of $N$–correction tuples that remain is also the *output* of the correction algorithm.

It is straightforward to see that for each error location tuple, the correction procedure described above, although exhaustive on the correction space, it is an efficient one. Gate related corrections are performed *locally* to the line under consideration, while wire related corrections require a *single* pass over the circuit lines.

The overall procedure for error correction with test–vector simulation is illustrated in Fig. 4.2. TVS_CORRECTION is the procedure that performs correction by calling TVS_CORRECT_VECTOR iteratively for every vector. TVS_CORRECT_VECTOR is the procedure that returns a set of corrections for an error tuple and a single vector $v$ according to Theorem 6. TVS_CORRECTION returns the final set of corrections by taking the intersection of the sets returned by TVS_CORRECT_VECTOR (Fig. 4.2, line 6) and verifying them with stuck–at and random vectors. We omit the Wrong Gate/Wrong Wire Case from Fig. 4.2 since it is a straightforward extension of the pseudocode described.

## 4.3 Correction with BDDs

### 4.3.1 Boolean Equations

#### 4.3.1.1 Forming an Error Equation

In this section, we generalize the results of [22] and [23] and describe a BDD based method for correction of multiple errors. The method is computationally more expensive than the one described in the previous section because it involves the invocation, maintenance and manipulation of BDDs. However, in comparison to the work of [23] and [39] that also use similar symbolic techniques, the algorithm described here *guarantees* to return all $N$–correction tuples for $N > 1$ that rectify the design, if such modifications exist in the modification model that is used in the particular run of the algorithm.

First, we need to give some useful definitions and define the concept of an **error equation with multiple unknowns.** In the discussion that follows we use the terminology of Section 1.2.1. When a symbol is underlined it denotes a *vector element.*

**Definition 16** *Let $\underline{x} = \{x_1, x_2, \ldots, x_N\}$ be a set of distinct circuit lines of an incorrect $G_C$ and let $\underline{X} = \{X_1, X_2, \ldots, X_N\}$ be a set of new input variables. The* **modified network** *$G_C^{\underline{X}}$ for $G_C$ is obtained if we replace the set of lines of $\underline{x}$ with the respective elements of $\underline{X}$.*

**Definition 17** *Let $EQ^x(PI, \underline{X})$ be the sum of the exclusive-ORs of respective $F_C$ and $G_C^{\underline{X}}$ primary outputs, that is:*

$$EQ^x(PI, \underline{X}) = \sum_{\forall erroneous\ PO_i} PO_i(F_C) \oplus PO_i(G_C^{\underline{X}}) \tag{4.1}$$

*We call $EQ^x(PI, \underline{X}) = 0$ an* **error equation** *with* **unknown vector value** *$\underline{X}$.*

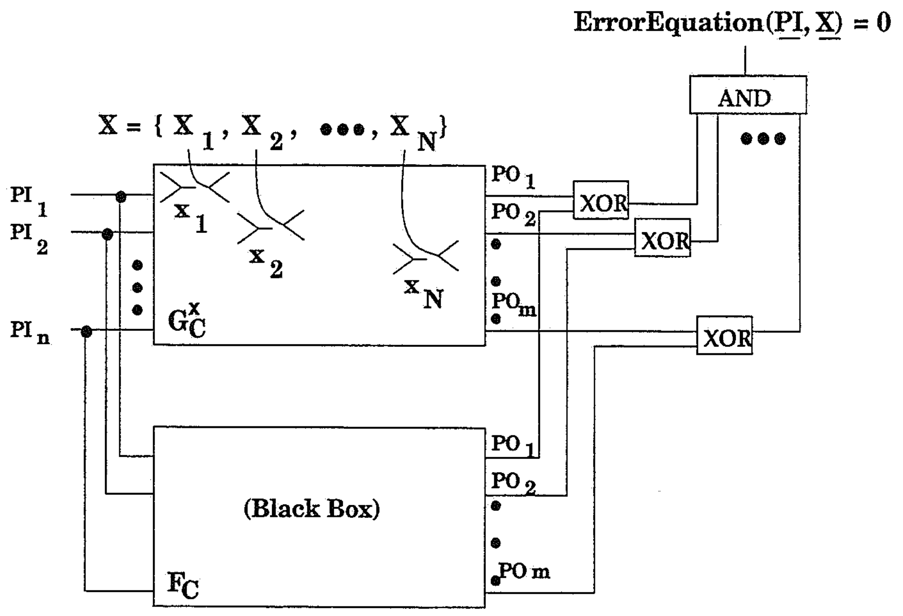


**Figure 4.3** Circuitry of an Error Equation

The concept of an error equation can be clearly explained if we describe the *circuitry* that forms an error equation for a network $G_C$. This is shown in Fig. 4.3 where the upper box represents $G_C$ and the bottom box $F_C$.

In the same figure, we observe that the signals of $\underline{x} = \{x_1, x_2, \ldots, x_N\}$ in $G_C$ have been disconnected from their original functions and they have been connected to the set of new input variables $\underline{X} = \{X_1, X_2, \ldots, X_N\}$. This also creates the modified network $G_C^{\underline{X}}$ of Def. 16. Originally, the primary outputs of both $G_C$ and $F_C$ are functions of the (common) primary input variables $PI = \{PI_1, PI_2, \ldots, PI_N\}$. Once the modified network is formed, the primary outputs of $G_C^{\underline{X}}$ become a function of $PI$ and the set of variables $\underline{X}$.

The XOR operations at the primary outputs of $F_C$ and $G_C^{\underline{X}}$ is also known in the literature as the *mitter* [9]. The operation of the XOR operation between $PO_i^{F_C}$ and $PO_i^{G_C^{\underline{X}}}$ is equal to zero if and only if both outputs produce the same boolean function. The following Example illustrates the above concepts.

**Example 20** *Fig. 4.4 shows the circuitry that forms an error equation for the modified network $G_C^{\underline{X}}$, shown in the dotted box.*

*The two design errors injected in $G_C$ are on gate $G_1$, an extra inverter error, and gate $G_3$, an AND gate replacement error. The function implemented at the primary output of the specification, $PO_{F_C}$, is $PI_1PI_2$ and the function implemented at the primary output of $G_C$ is $PI_1PI_3 + \overline{PI_2}$. The error equation of Fig. 3.3 is equal to $EQ^x(PI, \underline{X}) = (X_1 + PI_3)X_2 \oplus PI_1PI2$.*

#### 4.3.1.2 Solving an Error Equation

What can we say about the **solution** of an error equation? Intuitively, a solution for an error equation $EQ^x(PI, \underline{X}) = 0$ is such a set of $N$ functions for the elements of $\underline{X}$ that depend on $PI$ and when implemented on the respective lines of $\underline{x}$, they make respective primary outputs of $F_C$ and $G_C$ implement the exact same functions. Simply put, every solution to an error equation is a valid set of circuit corrections.

An error equation does not always have a unique solution. As explained shortly after, there can be cases that an error equation has a *set of solutions* [11]. The fact that there can be more than one set of lines where the respective error equation has a solution in

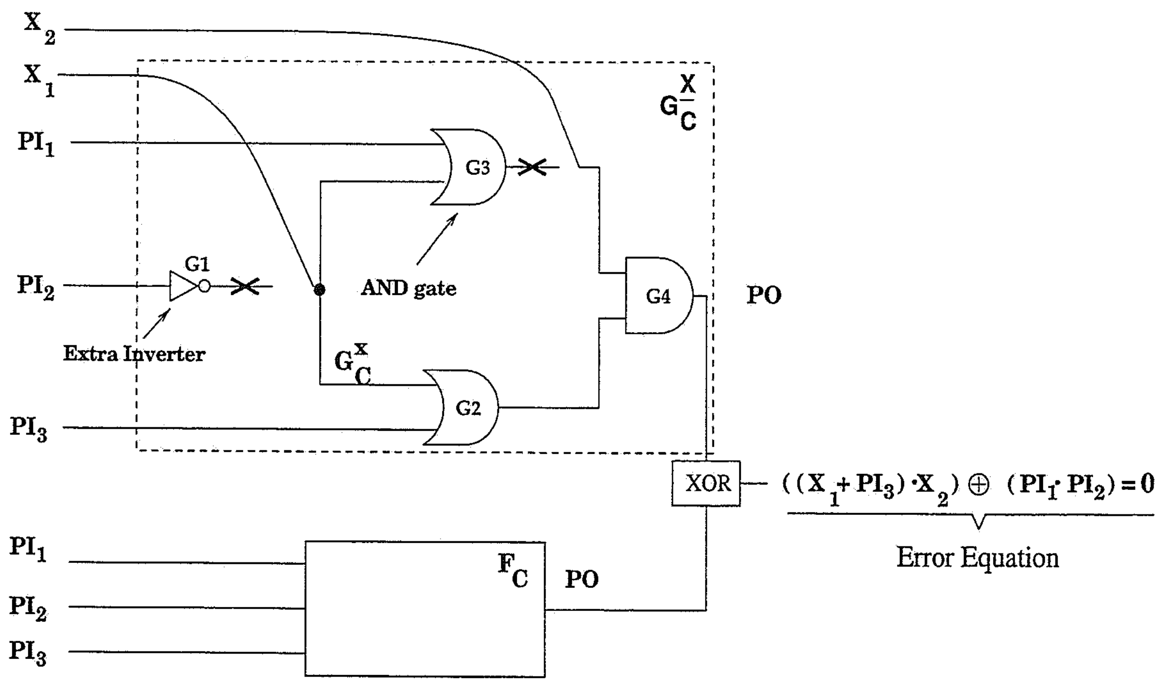


**Figure 4.4** Error Equation Circuitry for Examples 20 and 22

addition to the fact that an error equation can have multiple solutions even for the same set of lines, justifies the existence of **equivalent corrections** for an erroneous $G_C$.

Finally, if an error equation $EQ^x(PI, \underline{X}) = 0$ does not have a solution for the elements of the vector set $\underline{X}$, it simply means that the circuit *cannot* be rectified with *any* modifications on the lines of $\underline{X}$ regardless the design error model that is used.

The following theorem [11] provides a necessary and sufficient condition for a solution to an error equation to exist and also describes a recursive way to find such a solution. It is based on **successive elimination of variables** where the problem of solving a single $N$-variable equation is transformed to that of solving $N$-single variable equations. Under this approach, the solution of the $i$-th equation depends on the solutions assigned to the previous $i-1$ equations.

**Theorem 7** *Let $EQ^x(PI, \underline{X}) = 0$ be an error equation and let the function $f_N$ be defined as follows:*

- $f_N(\underline{X}) = EQ^x(PI, \underline{X})$
- $f_{i-1}(X_1, X_2, \ldots, X_{i-1}) = f_i(X_1, X_2, \ldots, X_{i-1}, 0) f_i(X_1, X_2, \ldots, X_{i-1}, 1), \; 1 \le i \le N$

*The error equation has a* **solution** *if and only if $f_0 = 0$ and the* **solution interval** [1] *for the i-th variable, $1 \le i \le N$, is:*

$$f_i(X_1, X_2, \ldots, X_{i-1}, 0) \le X_i \le \overline{f_i(X_1, X_2, \ldots, X_{i-1}, 1)} \tag{4.2}$$

The example that follows, taken from [11], illustrates the application of Theorem 7 on an error equation.

**Example 21**

<u>*Equation:*</u> $f(a,\ b,\ x_1,\ x_2,\ x_3) \;=\; bx_1 + bx_2'x_3 + b'x_1'x_2 + a'x_2'x_3' + a'x_1x_2' +$

$+a'x_1'x_2 + ab'x_2x_3 = 0$

<u>*Variables:*</u> $\underline{PI} \;=\; (a,\ b).$

<u>*Unknowns:*</u> $\underline{X} \;=\; (x_1,\ x_2,\ x_3),\ N \;=\; 3.$

<u>*Successive Variable Elimination:*</u>

[1]Recall that $f(x) \le g(x)$ for boolean functions $f, g$ if and only if $f(x)\overline{g(x)} = 0$.

$$\begin{aligned}
f_3 &= \textit{formula for } f \\
f_2 &= bx_1 + b'x_1'x_2 + a'x_1x_2' + a'x_1'x_2 + a'bx_2' && \textit{(eliminate } x_3\textit{)} \\
f_1 &= bx_1 + a'bx_1' && \textit{(eliminate } x_2\textit{)} \\
f_0 &= a'b && \textit{(eliminate } x_1\textit{)}
\end{aligned}$$

*Condition for a EQ solution to exist:*

$$f_0 = a'b = 0$$

*If $a'b = 0$ then solution intervals are:*

$$a'b \leq x_1 \leq b'$$

$$a'b + a'x_1 + bx_1 \leq x_2 \leq b'x_1 + abx_1'$$

$$a'x_1' + bx_1 + b'x_1'x_2 + a'x_2' \leq x_3 \leq b'x_1'x_2' + abx_1'x_2 + ab'x_2' + a'b'x_1x_2$$

It can be seen from Theorem 7 that an error equation does not necessarily have one unique solution, if such a solution exists. This is because a solution interval can sometimes allow a variable to have more than one valid values within the interval. Moreover, successive elimination requires that the solution interval of $X_i$, $1 \leq i \leq N$, be dependent on the value of all variables $X_j, \forall j < i$. Nevertheless, the order of variable elimination does not change the solution interval for the variables [11].

**Example 22** *With respect to the erroneous circuit of Example 20 we have that* $EQ^x(PI, \underline{X}) = f_2(\underline{X}) = \overline{PI_1}X_2X_1 + \overline{PI_1}X_2PI_3 + \overline{PI_2}X_1X_2 + \overline{PI_2}X_2PI_3 + PI_1\overline{X_1}PI_2\overline{PI_3} + PI_1PI_2\overline{X_2}$, $f_1(X_1) = PI_1\overline{X_1}PI_2\overline{PI_3}$, *and* $f_0 = 0$. *The latter implies, due to Theorem 7, that the error equation has a solution.*

*The solution interval for variable $X_1$ is $PI_1PI_2\overline{PI_3} \leq X_1 \leq 1$. If we let $X_1$ take the constant value of 1, we have that $f_2(X_1 = 1, X_2) = \overline{PI_1}X_2 + \overline{PI_2}X_2 + PI_1PI_2\overline{X_2}$ and the solution interval for $X_2$ becomes $PI_1PI_2 \leq X_2 \leq PI_1PI_2$. It can be seen that this interval contains only one solution for $X_2$, that is, $PI_1PI_2$.*

## 4.3.2 Symbolic Diagnosis and Correction

In this section we describe the overall approach for DEDC with the use of BDDs. The modified algorithm for *error diagnosis* and *correction* with the use of BDDs is shown in Fig. 4.5.

Diagnosis is performed in a way similar to the one presented in Chapters 2 and 3 (Fig. 4.5, lines 2 and 9) with the exception of lines 5–7 where the error equation is built for the qualifying checkpoints from Inverted Simulation. Since, by definition, the checkpoints dominate over the lines of their clans, if the union of clans for a set of checkpoints contains a set of valid modification locations for $G_C$, then the error equation will have a solution for the checkpoints as well [17]. This observation along with the code in lines 5–7 allow us to speed up the diagnosis procedure and disregard at an early point of the algorithm clans that *guarantee* not to contain any valid modification locations.

Once diagnosis has been performed for the circuit (lines 1–11), an exhaustive list of corrections is compiled from the results of test vector simulation according to the procedure described in Section 4.2 (line 12).

Subsequently, each correction tuple $CT = \{CT_1, CT_2, \ldots, CT_N\} \in C_{corrections}$ of this list is examined separately and disregarded if some correction $CT_i$ violates the correction

```
BDD_DIAGNOSIS_CORRECTION( $F_C$, $G_C$, $V_{act}$, $N$)
1.  $B_{checkpoints} = \{B_1, \ldots, B_c\}$ = checkpoints of $G_C$
2.  $C_{error}$ =Implicit_or_Explicit_Error_Enumeration( $F_C$, $G_C$, $B_{checkpoints}$, $V_{act}$, $N$)
3.  for every error-tuple $L_0, \ldots, L_{N-1} \in C'_{error}$ do
4.      Inverted_Simulation( $F_C$, $G_C$, $L_0, \ldots, L_{N-1}$)
5.  for every error-tuple $L_N = \{L_0, \ldots, L_{N-1}\} \in C'_{error}$ do
6.      build error equation $EQ(PI, \underline{X})$
7.      delete $L_N$ from $C'_{error}$ if $EQ(PI, \underline{X})$ has no solution
8.  C = union of clans of checkpoints $B_i \in$ some pair of $C'_{error}$
        (* find the final $C_{error}$ of checkpoints *)
9.  $C_{error}$ =Implicit_or_Explicit_Error_Enumeration( $F_C$, $G_C$, $C$, $V_{act}$, $N$)
10. for every error-tuple $L_0, \ldots, L_{N-1} \in C_{error}$ do
11.     Inverted_Simulation( $F_C$, $G_C$, $L_0, \ldots, L_{N-1}$)
12. $CT_{corrections} = TVS\_Correction(C_{error}, V_{act})$
13. for every correction tuple $CT \in CT_{corrections}$ do
14.     form error equation by composing resp. checkpoint error equation
15.     Check solution intervals for each member of $CT$
16.     If $CT$ violates some solution interval delete from $CT_{corrections}$
17. return($C_{corrections}$)
```

**Figure 4.5** Symbolic Design Error Correction

interval given by Theorem 7 (lines 13–16). As mentioned earlier, the correction interval for each variable remains the same regardless of the order the variables are eliminated.

Finally, we should mention the fact that the error equation for $CT$ can be easily constructed from the error equation of the checkpoints (lines 5–7) of the diagnosis algorithm (line 14). This can be achieved with the *bdd_compose* command which is available in most BDD packages. This observation allows us to save time and reuse parts of the previous computation.

The list of corrections that qualify the solution interval checking of lines 13–16 is also the output of the algorithm (line 17).

### 4.3.3 Producing Vectors With Erroneous Responses

Lets return to equation 4.1 in Def. 17. Observe that if the set $\underline{X}$ is the *empty set*, that is, $G_C$ and $G_C^{\underline{X}}$ coincide, then equation 4.1 can be used to provide input test vectors with incorrect primary output responses for an erroneous $G_C$.

All these vectors are exactly the ones that give such a primary input assignment so that 4.1 gets a true value. Intuitively, such an assignment of input values is the one that produces *at least* one different primary output response in $G_C$ and $F_C$. Returning to Example 20, we observe that an input vector with erroneous primary output response is any vector that makes $(PI_1PI_3 + \overline{PI_2}) \oplus (PI_1PI_2)$ equal to 0.

## 4.4 Summary

In this chapter, we described two procedures that perform design error correction once diagnosis is completed. Both of them are *exhaustive on the correction space*, that is, they will return all applicable corrections that rectify the design if such corrections exist in the modification model that it is used. The modification model we used is the one of Def. 3 (Chapter 1 which is an *extension* of the one described in [2].

The first correction approach is based on *test vector simulation* and it exhaustively produces a list of potential corrections from the results of the stuck-at vectors that *activate the inconsistencies.* Verification with random test vector simulation is performed as a last step so that potential corrections with erroneous primary output responses are deleted from the correction list.

Next, we described a *symbolic* approach for correction that uses the concept of an error equation with multiple unknowns. The symbolic approach works on top of the test vector simulation correction procedure. Although it is not a run–time efficient procedure since it involves the use of BDDs, it is *exact on the correction space* as it guarantees to return *all* and *only* the corrections that indeed rectify the design, if such corrections exist in the modification model that it is used.

Since exhaustive input vector simulation is prohibited for most circuits, it is of interest to know the quality of a simulation procedure driven by a small subset of the input test vector space for multiple DEDC. This is an issue that we investigate throughout our experiments in Chapter 5.

# CHAPTER 5

# Experimental Results

## 5.1 Introduction

We implemented the diagnosis and rectification algorithms presented in the previous chapters in C language, and ran it on a Sparc 10 workstation with 220MB of memory. We tested the algorithms on ISCAS'85 benchmark circuits with the characteristics of Table 5.1 for 1–source, 2–source and 3–source correctable designs with all three types of design errors of Def. 5.1. The types and locations where the errors were injected were selected randomly.

The initial number of error candidates for each of these designs is shown in Table 5.2. These numbers are computed according to equation 1.1 from Chapter 1. Observe that the correction space, assuming the error model of [2] or Def. 3, is an order of magnitude larger since we have to consider additional input wires for error locations.

The average values of the results of our experiments on the circuits of Table 5.1 are reported in the following sections. All run–times of our results are in seconds.

| ckt name | Description | # of primary inputs | # of primary outputs | # of gates | # of lines | # of checkpoints | average clan size |
|---|---|---|---|---|---|---|---|
| C432 | Priority Decoder | 36 | 7 | 234 | 545 | 89 | **6.1** |
| C499 | Error Correcting | 41 | 32 | 620 | 1224 | 155 | **7.8** |
| C880 | ALU and Control | 60 | 26 | 385 | 880 | 126 | **7.0** |
| C1355 | Error Correcting | 41 | 32 | 548 | 1355 | 259 | **5.1** |
| C1908 | Error Correcting | 33 | 25 | 882 | 1908 | 384 | **4.9** |
| C2670 | ALU and Control | 157 | 63 | 1193 | 2670 | 454 | **5.8** |
| C3540 | ALU and Control | 50 | 22 | 1169 | 3540 | 601 | **5.8** |
| C5315 | ALU and Selector | 178 | 123 | 2309 | 5315 | 806 | **6.5** |
| C7522 | ALU and Control | 207 | 108 | 3514 | 7552 | 1300 | **5.8** |

**Table 5.1** ISCAS'85 Circuit Characteristics

## 5.2 Results on Diagnosis With Explicit Enumeration of Error Tuples

In this section we discuss the experimental results of the diagnosis algorithm described in Chapter 2.

We tested our diagnosis algorithm with explicit enumeration of error tuples for 1–source and 2–source correctable designs. We repeated the experiments 60 times for each circuit, 30 times the design was 1–source correctable and 30 times it was 2–source correctable. The results of our experiments are reported in Tables 5.3 and 5.4.

The second column contains the average time needed during the initial circuit verification step. As explained in Section 1.2.1, this step simulates vectors for stuck-at faults, and random vectors, if necessary, in order to compile $V_{act}$. This is also the step where the $Fgroup$ and $Tgroup$ linked lists at every line of the circuit are created. In our experi-

| ckt | 1–source Corr. | 2–source Corr. | 3–source Corr. |
|---|---|---|---|
| C432 | 545 | 297025 | 161878625 |
| C499 | 1224 | 1498176 | 1833767424 |
| C880 | 880 | 774400 | 681472000 |
| C1355 | 1355 | 1836025 | 2487813875 |
| C1908 | 1908 | 3640464 | 6946005312 |
| C2670 | 2670 | 7128900 | 19034163000 |
| C3540 | 3540 | 12531600 | 44361864000 |
| C5315 | 5315 | 28249225 | 150144630875 |
| C7522 | 7552 | 57032704 | 430710980608 |

**Table 5.2** Initial Error Space for Error Diagnosis

ments, $V_{act}$ was never empty unless the error injected was **redundant**, that is, it doesn't change the functionality of the circuit at the primary outputs [17]. The average size of $V_{act}$ required for an efficient solution to DEDC is given in the third column.

Columns 4 and 5 contain the average times for the Total Observability Measure and Inverted Simulation diagnosis procedures, respectively. These values contain the run–times for *both* iterations of the procedures, first on the checkpoints and then on the lines of clans that qualified. The computational savings during error location due to the structural observations of a circuit presented in Section 2.3.4 is significant, but expected, if we observe the values of column 8 in Table 5.1. This column contains the average clan size for each circuit. This number also denotes the average speed up for 1–source correctable designs versus a naive approach that considers all lines of the circuit. For the 2–source correctable designs, the speed up is larger and it is lower bounded by the square of the numbers of column 8.

| ckt name | Verif. Time | $\mid V_{act} \mid$ | TOM Time | Inv.Sim. Time | **Total Error Location Time** | **# of Error Tuples** |
|---|---|---|---|---|---|---|
| C432 | 8.8 | 40 | 1.9 | 0.9 | **1.8** | **5.9** |
| C499 | 20.1 | 50 | 11.1 | 4.9 | **16.0** | **8.1** |
| C880 | 8.9 | 50 | 4.9 | 2.3 | **7.2** | **3.2** |
| C1355 | 16.4 | 60 | 6.8 | 8.5 | **15.3** | **9.0** |
| C1908 | 21.2 | 80 | 7.0 | 9.4 | **16.4** | **6.7** |
| C2670 | 24.5 | 60 | 8.1 | 7.8 | **15.9** | **8.9** |
| C3540 | 18.2 | 80 | 9.9 | 7.1 | **17.0** | **6.3** |
| C5315 | 29.9 | 80 | 9.6 | 11.1 | **20.7** | **7.1** |
| C7522 | 36.0 | 100 | 10.8 | 11.5 | **22.3** | **6.0** |

**Table 5.3** Explicit Diagnosis for 1–Source Correctable Designs

Another way to view the above result is the diagram of Fig. 5.1 that shows the average number of error tuples over the time needed for diagnosis for the 2–source correctable C432. The dotted line indicates the end of the first iteration of the two procedures on the checkpoints of C432 and the beginning of the application of the diagnosis procedures on the clans that qualified. The reduction in the number of error tuples is dramatic; within 3.9 seconds, that is, 10.0% of the overall CPU time for diagnosis, the algorithm has deleted 67.1% of the potential error pairs, while within 27.2 seconds the algorithm deletes 99.75% of $C_{error}$ (on the average).

Finally, the last column of each of the Tables 5.3 and 5.4 contains the average number of error tuples after the completion of the diagnosis procedure. These numbers indicate

| ckt name | Verif. Time | $\mid V_{act} \mid$ | TOM Time | Inv.Sim. Time | Total Error Location Time | # of Error Tuples |
|---|---|---|---|---|---|---|
| C432 | 13.1 | 150 | 18.2 | 20.7 | **38.9** | **31.8** |
| C499 | 26.2 | 220 | 37.4 | 25.4 | **68.2** | **44.2** |
| C880 | 15.0 | 150 | 30.1 | 17.2 | **47.3** | **36.1** |
| C1355 | 20.2 | 220 | 55.8 | 69.1 | **124.9** | **40.5** |
| C1908 | 29.7 | 220 | 89.9 | 138.4 | **228.3** | **60.8** |
| C2670 | 40.3 | 300 | 101.4 | 183.6 | **285.0** | **49.2** |
| C3540 | 38.1 | 300 | 78.1 | 111.2 | **189.3** | **24.1** |
| C5315 | 44.3 | 320 | 30.1 | 62.1 | **92.1** | **47.2** |
| C7522 | 58.0 | 320 | 128.3 | 199.8 | **328.1** | **39.3** |

**Table 5.4** Explicit Diagnosis for 2–Source Correctable Designs

that the proposed diagnosis algorithm has good resolution. We should mention that the *worst case* run–time behavior of the proposed approach was less than three times the average case.

## 5.3 Results on Diagnosis With Implicit Enumeration of Error Tuples

In this section we present our experimental results of the diagnosis algorithm with implicit enumeration of error tuples presented in Chapter 3.

We tested our diagnosis algorithm for 1, 2, and 3–source correctable designs. We repeated our experiments 20 times for each circuit and each different correctability sce-

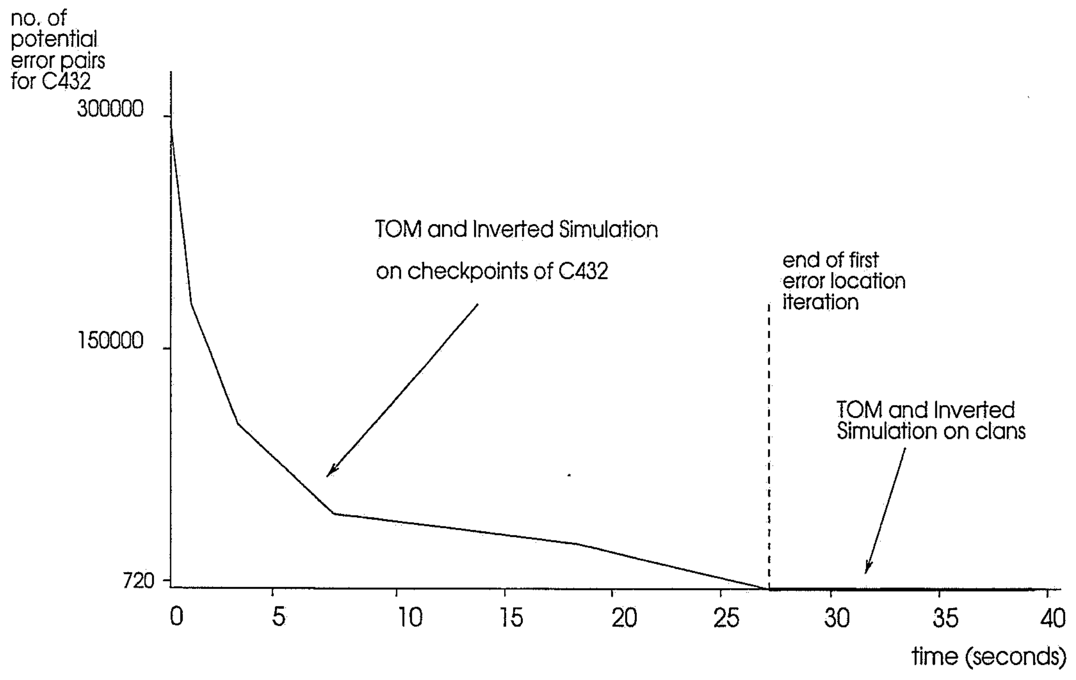


**Figure 5.1** Diagnosis Speed Up for the 2–Source Correctable C432

nario. The average results of our experiments are shown in Tables 5.5, 5.6, 5.7, 5.8, 5.9, and 5.10.

Tables 5.5, 5.6, and 5.7 contain the IG characteristics for the different correctability scenarios. The second column contains the number of graph vertices (or, equivalently, the number of runs for path–trace back ) created from different vectors of $V_{act}$ and different erroneous primary outputs (Fig. 3.13, lines 1–3). The average number of failing primary outputs per vector was less than 5. The next column(s) contains the average number of vertices that contain some error(s). For Table 5.5 this number is obviously equal to the number of vertices since the design is 1–source correctable (Theorem 4).

Finally, the last 3 columns of each table contain the maximum, minimum and average size of $line(V)$ for the vertices of the IG including the ones that are created after a

| ckt | Total # of IG | # of IG | # Lines Per Node | | |
|---|---|---|---|---|---|
| name | Nodes | Nodes With Error | Min | Max | Ave |
| C432 | 30 | 30 | 3.8 | 50.3 | 18.3 |
| C499 | 32 | 32 | 2.0 | 113.1 | 31.2 |
| C880 | 28 | 28 | 3.1 | 51.2 | 18.9 |
| C1355 | 40 | 40 | 8.3 | 175.2 | 54.3 |
| C1908 | 40 | 40 | 4.5 | 171.8 | 23.7 |
| C2670 | 40 | 40 | 3.6 | 283.5 | 51.3 |
| C3540 | 40 | 40 | 5.2 | 241.4 | 39.1 |
| C5315 | 40 | 40 | 5.9 | 122.0 | 27.8 |
| C7522 | 45 | 45 | 8.3 | 191.9 | 51.3 |

**Table 5.5** IG Characteristics for 1–Source Correctable Designs

reduction. It follows from our discussion in Section 3.4.2 that the smaller the average size of a vertex is, the better performance our implicit enumeration algorithm will have. For this reason, reductions on the IG is a desired operation as they shrink the size of $line(V)$ for all the vertices $V$ that participate in the reduction.

Tables 5.8, 5.9, and 5.10 contain the results of our implicit enumeration diagnosis algorithm presented in Section 3.4.3 [1].

Column 2, for each table, contains the size $n$ of the random vertex sample and the next column contains the average number of algorithm iterations $iters$. The algorithm iterates when the $n$–sample does not provide a good estimate for diagnosis and $C_{error}$ is empty after Inverted Simulation (Fig. 3.13, lines 4–13). Our results indicate that the

[1]Some of the columns are not applicable to the 1–source correctable experiments.

| ckt | Total # of IG | # of IG With Error | | # Lines Per Node | | |
|---|---|---|---|---|---|---|
| name | Nodes | Err1 | Err2 | Min | Max | Ave |
| C432 | 40 | 29.8 | 24.8 | 9.4 | 42.3 | 28.2 |
| C499 | 65 | 34.2 | 39.7 | 14.3 | 93.2 | 55.9 |
| C880 | 50 | 39.4 | 34.6 | 60.8 | 19.2 | 10.6 |
| C1355 | 58 | 37.7 | 33.5 | 34.9 | 145.2 | 82.4 |
| C1908 | 80 | 42.8 | 47.0 | 20.8 | 172.3 | 55.2 |
| C2670 | 80 | 47.9 | 51.0 | 8.6 | 122.3 | 43.8 |
| C3540 | 100 | 73.9 | 64.4 | 22.5 | 231.8 | 89.0 |
| C5315 | 100 | 62.3 | 57.4 | 7.3 | 89.3 | 22.9 |
| C7522 | 150 | 89.5 | 101.7 | 9.0 | 102.3 | 31.8 |

**Table 5.6** IG Characteristics for 2–Source Correctable Designs

algorithm is able to give a solution for multiple error diagnosis without any repetition in most of the times.

Column 4 indicates the CPU time of our implicit enumeration excluding the Inverted Simulation step (Fig. 3.13, lines 4–13) for *one* iteration of the algorithm. To obtain the average time for *all* iterations, one should simply multiply the value of this column with the one of column 2.

The fifth column contains the type of graph that we most often obtained during IG processing. A number in this column indicates the maximum number of non-adjacencies. The clique case is the most computationally expensive case to handle, while a $N$–disconnected component IG for a $N$–source correctable design is the faster to handle as each component will contain a distinct element of some error tuple(s). In addition, the latter case provides with better error resolution.

| ckt name | Total # of IG Nodes | # of IG With Error | | | # Lines Per Node | | |
|---|---|---|---|---|---|---|---|
| | | Err1 | Err2 | Err3 | Min | Max | Ave |
| C432 | 80 | 78.3 | 49.4 | 44.4 | 6.1 | 55.6 | 23.0 |
| C499 | 110 | 34.5 | 40.6 | 36.9 | 7.1 | 88.8 | 43.3 |
| C880 | 80 | 84.3 | 69.9 | 71.3 | 3.3 | 40.6 | 14.9 |
| C1355 | 100 | 37.8 | 42.2 | 31.0 | 28.8 | 138.9 | 85.6 |
| C1908 | 120 | 56.7 | 47.0 | 42.9 | 12.3 | 141.2 | 41.5 |
| C2670 | 120 | 56.7 | 48.2 | 51.4 | 10.8 | 104.3 | 44.0 |
| C3540 | 150 | 72.1 | 77.6 | 69.8 | 14.7 | 177.1 | 85.6 |
| C5315 | 200 | 56.4 | 53.3 | 44.5 | 11.0 | 104.5 | 48.7 |
| C7522 | 200 | 111.2 | 102.8 | 85.6 | 19.8 | 188.9 | 77.3 |

**Table 5.7** IG Characteristics for 3–Source Correctable Designs

The next three columns for each table contain the average number of error tuples generated by the implicit enumeration procedure (column 5), the average number of error tuples after Inverted Simulation (Fig. 3.13, line 12) and the Inverted Simulation time for one iteration of the algorithm. The total average time is equal to the time of this column multiplied by the average number of *iters* (column 3).

The total time of our implicit diagnosis algorithm is shown in the last column. Considering the initial error space of Table 5.1 and Tables 5.8, 5.9, and 5.10, we conclude that the algorithm is able to produce diagnostic results with good error resolution within a short computational time.

Recall Theorem 5 from Section 3.4.1. This theorem says that if we perform a $K$–graph reduction on an IG for a $N$–source correctable design, we might sacrifice on error resolution. In our experiments for 3–source correctable designs, we performed 2–graph

| ckt name | $n$ | *iters* | Graph Proc. Time | IG Type | Error Tuples Before IS | **Error Tuples After IS** | IS Time | **Total Time** |
|---|---|---|---|---|---|---|---|---|
| C432 | - | - | 0.8 | - | 7.3 | **1.5** | 0.4 | **1.2** |
| C499 | - | - | 1.1 | - | 13.1 | **2.1** | 0.9 | **2.0** |
| C880 | - | - | 0.7 | - | 3.8 | **1.4** | 0.3 | **1.0** |
| C1355 | - | - | 1.0 | - | 7.6 | **2.2** | 0.2 | **1.2** |
| C1908 | - | - | 1.5 | - | 12.3 | **2.9** | 0.4 | **1.9** |
| C2670 | - | - | 1.9 | - | 14.4 | **3.4** | 0.4 | **2.3** |
| C3540 | - | - | 2.0 | - | 11.8 | **3.8** | 0.9 | **2.9** |
| C5315 | - | - | 2.1 | - | 18.0 | **3.1** | 1.1 | **3.2** |
| C7522 | - | - | 2.1 | - | 21.8 | **2.9** | 1.3 | **3.4** |

**Table 5.8** Implicit Diagnosis for 1–Source Correctable Designs

reductions when applicable. In more than 60% of the cases the error resolution diminished in such a level that we were not able to get any valid location triples, that is, Inverted Simulation returned with an empty set. In the remaining cases, we were able to return with a solution although the majority of valid triples had been eliminated as Theorem 5 implies. However, the run–time performance of the algorithm improved significantly compared to the one when 2–graph reductions were not allowed.

Our experiments suggest that we should allow such $K$–graph reductions since they improve the run–time performance. If Inverted Simulation returns with no results for a small number of *iters*, we should gradually increase the value of $K$ until a valid set of locations has been returned.

| ckt name | $n$ | *iters* | Graph Proc. Time | IG Type | Error Tuples Before IS | **Error Tuples After IS** | IS Time | **Total Time** |
|---|---|---|---|---|---|---|---|---|
| C432 | 2 | 1.0 | 1.3 | clique | 72 | **8.3** | 1.6 | **2.9** |
| C499 | 2 | 1.0 | 1.8 | 2 | 1722.3 | **16.1** | 4.9 | **6.7** |
| C880 | 2 | 1.5 | 1.6 | 2 | 194.2 | **4.1** | 2.9 | **4.5** |
| C1355 | 3 | 1.9 | 2.7 | 2 | 4018.6 | **7.7** | 8.3 | **11.0** |
| C1908 | 4 | 1.5 | 6.9 | clique | 3709.5 | **7.3** | 7.1 | **14.2** |
| C2670 | 4 | 1.6 | 5.8 | 2 | 3252.0 | **16.3** | 11.1 | **16.9** |
| C3540 | 3 | 1.3 | 4.9 | clique | 7195.8 | **14.1** | 12.3 | **19.0** |
| C5315 | 4 | 1.0 | 3.9 | 2 | 631.7 | **7.3** | 5.8 | **9.7** |
| C7522 | 4 | 1.5 | 4.4 | 2 | 1311.7 | **13.4** | 7.1 | **11.5** |

**Table 5.9** Implicit Diagnosis for 2–Source Correctable Designs

## 5.4 Results on Error Correction

In this section we present the experiments for our error correction methodology presented in Chapter 4. For the purpose of the symbolic approach, all boolean functions are expressed by Binary Decision Diagrams (BDDs) [12]. We used the shared BDD library developed by Brace et al [7] with dynamic variable ordering (sift algorithm) for circuits C2670 and C7552 [53].

Without loss of generality, we implemented and obtained the experimental results of our error correction techniques as a back end of the diagnosis algorithm with explicit enumeration of error tuples. These results are reported in Table 5.11, for 1–source correctable designs, and Table 5.12, for 2–source correctable designs. As mentioned in section 5.2, we repeated the experiment 30 times for each circuit and each correctability scenario.

| ckt name | $n$ | *iters* | IG Proc. Time | IG Type | Error Tuples Before IS | **Error Tuples After IS** | IS Time | **Total Time** |
|---|---|---|---|---|---|---|---|---|
| C432 | 5 | 1.5 | 8.3 | 2 | 8136.5 | **11.7** | 23.1 | **31.4** |
| C499 | 6 | 1.9 | 10.3 | 2 | 24344.5 | **18.0** | 47.1 | **97.4** |
| C880 | 5 | 1.8 | 8.1 | 3 | 1088.0 | **2.9** | 21.3 | **29.4** |
| C1355 | 7 | 1.5 | 10.9 | clique | 175300 | **21.3** | 139.7 | **180.6** |
| C1908 | 8 | 1.5 | 14.4 | 2 | 68544.9 | **28.1** | 89.2 | **183.6** |
| C2670 | 6 | 1.5 | 19.1 | 2 | 74355.6 | **29.5** | 108.4 | **227.5** |
| C3540 | 5 | 1.9 | 21.0 | clique | 212776 | **22.3** | 254.1 | **345.1** |
| C5315 | 8 | 1.5 | 19.2 | 2 | 77001.2 | **18.1** | 142.9 | **362.1** |
| C7522 | 14 | 1.8 | 31.3 | 2 | 871104.3 | **31.9** | 495.1 | **626.4** |

**Table 5.10** Implicit Diagnosis for 3–Source Correctable Designs

Columns 2 and 3 from each table contain the run–time results for the test vector simulation based correction procedure of Section 4.2. Column 2 has the time needed to compile exhaustively the list of *all* possible corrections and column 3 contains the average time for their verification. The average number of random test vectors used during this verification step is given in column 4. Column 5 contains their hit–ratio in activating the inconsistencies. Experimental results on the performance of vectors generated for stuck-at faults for design error verification can be found in [5].

The run–times reported in column 6 of Tables 5.11 and 5.12 exhibit the performance of the BDD based correction procedure described in Section 4.3 when it returns *all* applicable corrections for $G_C$. The values of this column also contain the time needed to build the error equation for each error location checkpoint tuple.

| ckt name | TVS Corr. Time | TVS Verif. Corr. Time | # Random Vectors | Rand.Vect. hit-ratio | BDDs Corr. Time | # Corr. Tuples |
|---|---|---|---|---|---|---|
| C432 | 1.7 | 0.9 | 10000 | 10.8 % | 1.8 | **1.9** |
| C499 | 4.9 | 1.2 | 10000 | 21.7 % | 8.7 | **2.5** |
| C880 | 4.4 | 0.9 | 8000 | 18.1 % | 2.4 | **1.0** |
| C1355 | 4.3 | 1.4 | 12000 | 20.0 % | 19.8 | **2.8** |
| C1908 | 6.9 | 1.8 | 12000 | 18.5 % | 9.3 | **2.1** |
| C2670 | 9.0 | 2.1 | 12000 | 17.3 % | 98.2 | **2.5** |
| C3540 | 8.2 | 3.6 | 14000 | 24.6 % | 77.8 | **2.3** |
| C5315 | 9.9 | 1.9 | 14000 | 24.8 % | 17.1 | **2.2** |
| C7522 | 12.0 | 2.3 | 14000 | 29.3 % | 244.3 | **1.4** |

**Table 5.11** Error Correction for 1–Source Correctable Designs

The algorithm can be easily modified to exit after the first correction is found. In such a case, the run–times are a fraction of the ones shown in the tables (columns 1 and 5). Nevertheless, in our experiments, we were also interested in the number of *equivalent* corrections as well. This number was usually less than 20 valid correction tuples. Since the algorithm is exhaustive on the error space, it guarantees to return all equivalent modifications from the modification model that is used at the particular run. The average number of correction tuples returned by our symbolic correction approach is shown in column 7 of the tables.

Table 5.13 contains the **correction hit–ratios** of the error correction technique of Section 4.2 when a smaller number of random test–vectors is used for correction verification (Section 4.2.4, Fig. 4.2). With hit–ratio we mean the percentage of the corrections returned by the symbolic correction technique of Section 4.3 versus the number of cor-

| ckt name | TVS Corr. Time | TVS Verif. Corr. Time | # Random Vectors | Rand.Vect. hit-ratio | BDDs Corr. Time | # Corr. Tuples |
|---|---|---|---|---|---|---|
| C432 | 14.3 | 8.2 | 10000 | 24.2 % | 38.1 | **8.2** |
| C499 | 39.8 | 4.1 | 10000 | 52.8 % | 188.8 | **19.1** |
| C880 | 9.3 | 1.3 | 8000 | 61.8 % | 68.5 | **2.3** |
| C1355 | 94.4 | 10.9 | 12000 | 59.9 % | 179.5 | **6.7** |
| C1908 | 92.4 | 14.5 | 12000 | 38.0 % | 101.2 | **4.2** |
| C2670 | 160.9 | 9.0 | 12000 | 61.7 % | 132.6 | **13.4** |
| C3540 | 128.3 | 10.1 | 14000 | 58.3 % | 149.2 | **8.9** |
| C5315 | 132.1 | 6.5 | 14000 | 53.2 % | 142.3 | **4.7** |
| C7522 | 155.7 | 14.2 | 14000 | 46.2 % | 240.0 | **18.2** |

**Table 5.12** Error Correction for 2–Source Correctable Designs

rections returned by the test vector simulation correction algorithm alone. Since, for most circuits, exhaustive test vector simulation is prohibited, the numbers of this Table indicate the *quality* of test vector simulation for the number of vectors used.

For the vectors indicated in column 4 of Tables 5.11 and 5.12 the correction hit–ratio is 100% for *all* circuits. It should be emphasized the fact that this is an experimental result only. The work of [2] contains a formal proof that some interconnection errors are indeed hard to detect. This is also confirmed in the work of [5] where a test–vector generation algorithm for design errors is developed; the error coverage for some design errors is less than 100% since a small number of less than 200 input vectors is used. The numbers of Table 5.13 agree with the above results as they suggest that there are design errors that are hard to detect and correct.

| ckt | 1-source Corr. | | 2-source Corr. | |
|---|---|---|---|---|
| name | # of Random Vectors | hit-ratio | # of Random Vectors | hit-ratio |
| C432 | 2000 | 97.8 % | 2500 | 100 % |
| C499 | 3500 | 96.6 % | 3500 | 90.1 % |
| C880 | 1200 | 100 % | 2500 | 100 % |
| C1355 | 3000 | 99.1 % | 3500 | 91.8 % |
| C1908 | 4000 | 99.0 % | 4000 | 96.5% |
| C2670 | 4200 | 97.2 % | 4500 | 89.8 % |
| C3540 | 4200 | 93.2 % | 5000 | 94.1 % |
| C5315 | 4500 | 93.5 % | 6000 | 98.9 % |
| C7552 | 5000 | 94.9 % | 7000 | 91.3 % |

**Table 5.13** Correction hit-ratio for a Reduced Number of Random Vectors

### 5.4.1 On the Performance of Test-Vector Simulation to DEDC

In perspective, how useful is a DEDC method solely based on test-vector simulation? The 100% correction hit-ratios with the random vectors of Tables 5.11 and 5.12 and the run-time results of Section 5.3 indicate that diagnosis and correction provided by a relatively small fraction of the input test-vector space is indeed a good and attractive alternative to the one based on BDDs.

Even if there is no formal proof to establish the above result, a method that performs diagnosis and correction with test-vector simulation is very useful since it is run-time efficient. Furthermore, our experiments show that the proposed test-vector simulation based methodology is able to narrow down the number of potential corrections significantly. Next, a verification tool [12] [9] [25] [34] [36] [31] [41] can perform individual correction verification Fig. 1.3 and guarantee that the circuit is rectified.

# CHAPTER 6

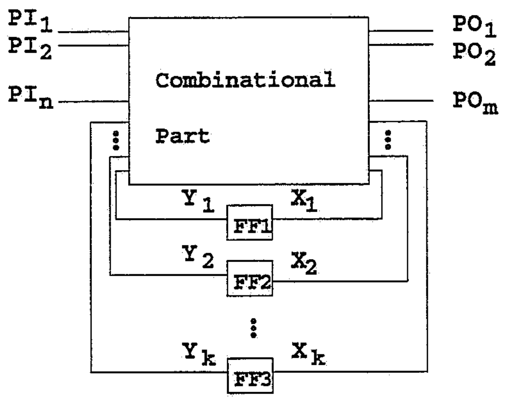
# Related Research Topics

## 6.1 Design Error Diagnosis of Sequential Circuits

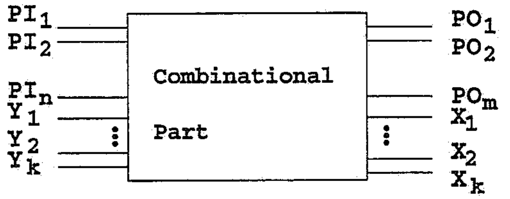
As explained in Section 1.2.1, the combinational design error methodology described in this thesis can apply to sequential circuit diagnosis if there is a one–to–one flip–flop correspondence between the specification and implementation.

Consider the sequential *gate–level implementation* of Fig. 6.1(a) where $\{PI_1, PI_2, \ldots, PI_n\}$ are the primary inputs, $\{PO_1, PO_2, \ldots, PO_m\}$ are the primary outputs, $\{X_1, X_2, \ldots, X_k\}$ is the present state, and $\{Y_1, Y_2, \ldots, Y_k\}$ is the next state. If a one–to–one correspondence of the flip–flops $\{FF_1, FF_2, \ldots, FF_k\}$ of the gate–level implementation with those of the specification exists, then we can apply the transformation shown in Fig. 6.1(b) and apply a combinational DEDC method. In the circuit of Fig. 6.1(b), each current (next) state is represented as a pseudo primary output (input).

If such a transformation is not feasible, for example, there is a different number of state elements between $F_C$ and $G_C$, combinational DEDC techniques are no longer applicable. The problem of sequential circuit DEDC has been examined in [28] [64]. These methods use an iterative array representation of the implementation, shown in Fig. 6.2, where the design is expanded in time. The set of inputs for time frame $i$ is equal to the new primary input vector $v_i$ for frame $i$ and the value of the state–elements from frame $i-1$. Next,



(a) Sequential Circuit



(b) Extraction of Combinational Logic

**Figure 6.1** Sequential Circuit Diagnosis

combinational techniques are adapted where the erroneous design is the set of circuitry in time frames $t_1, t_2, \ldots, t_l$, where $t_l$ is the first time frame with an erroneous $PO$ response.

This iterative array representation seems to be a mandatory requirement when a flip–flop equivalence between $F_C$ and $G_C$ is not available. Nevertheless, such a representation is expensive when vectors that activate the inconsistencies need to be expanded for many time frames. In addition, conventional combinational DEDC methods will fail for situations that the circuit needs to be expanded for many time frames because the error space increases according to Eq. 1.1 for the circuitry under diagnosis [28].

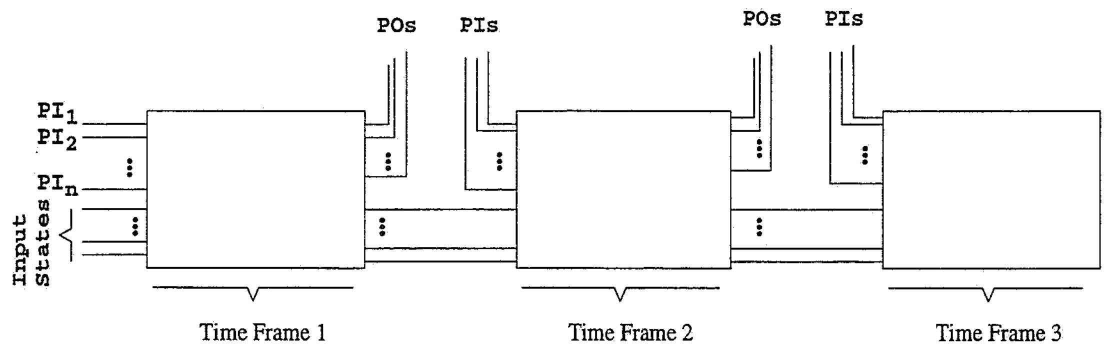


**Figure 6.2** Iterative Array Expansion of a Sequential Circuit

## 6.2 Engineering Change

Given an old and a new specification along with a design that implements the old specification, the problem of Engineering Change is to resynthesize the design so that it implements the new specification. It is also desired that this resynthesis procedure will reuse as much from the old design as possible. A detailed description of the problem and its relation to DEDC can be found in Chapter 1.

Previous work for the EC problem includes [8], [26], [39], and [62]. Most of the previous work assumes that both old and new specifications are given in terms of a netlist. For example, the old and new specifications are netlists before an automated tool performed some optimization steps and the design is the optimized circuit.

If this is the case, a naming correspondence between signals of the specification and the implementation might exist. This allows a mapping of equivalent signals between the specification and the existing design [8] [39]. This mapping is utilized so that the new design will re-use as much as possible from the existing one. A detailed description of the work of [39] and [62] can be found in Section 1.3.

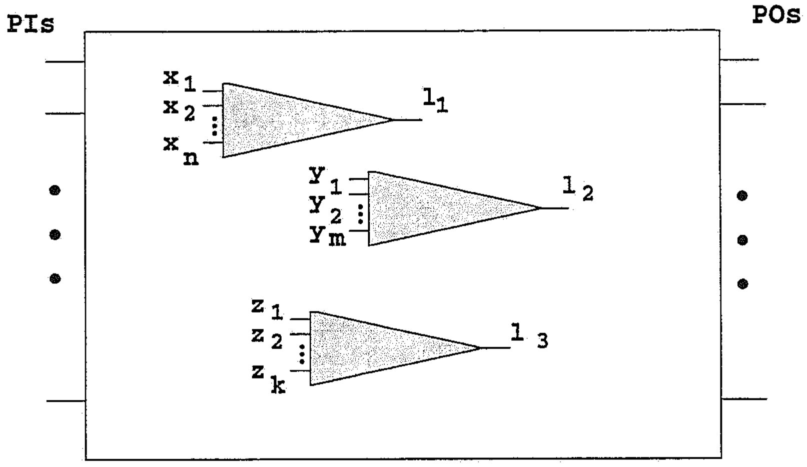


**Figure 6.3** An Approach for Engineering Change

In many cases, though, such a naming correspondence is not available and the specification is viewed as a "black box" that can only give the primary output responses in terms of the primary inputs. This version of the Engineering Change problem is very similar to that of DEDC, however, one must not necessarily expect that a few modifications (1, 2 or 3) are always enough to appropriately resynthesize the design.

Prompted by the results of our diagnosis algorithm with implicit enumeration of error pairs (Chapter 3) we propose a two–stage resynthesis approach for the Engineering Change problem when no line naming equivalence is available. During the first stage, implicit enumeration will be used in order to identify a small number of candidate *check-*

*point* signals (Section 2.3.4) that need to be modified to correct the design. We denote these signals with $l_1, l_2$, and $l_3$ in the design of Fig. 6.3.

During the second stage of diagnosis, implicit or explicit enumeration can be used in the clans of the checkpoints (shaded areas in Fig. 6.3) to identify more signals and, subsequently, perform rectification (Chapter 4) on them. In this manner, we manage to address the increase of problem complexity due to high error multiplicity.

In addition, observe that although the specification is not able to provide any signal values for lines $x_1, \ldots, x_n, y_1, \ldots, y_m$, $z_1, \ldots, z_k$, and $l_1, l_2, l_3$ during this second stage of diagnosis, these values are available from Inverted Simulation during the first diagnosis step (Theorem 6).

Finally, if the above method is not able to return results for some circuits, rectification of individual erroneous outputs can be utilized in a way similar to that of [25]. This approach requires that each erroneous primary output be rectified *individually*. Such an approach might not offer the highest design reuse rate, but it has a good chance to achieve rectification.

## 6.3 Design Optimization

Lately, a number of multi–level logic optimization techniques have been developed that rely on gate substitutions and rewiring [14] [16] [19] [45] [35] [48] [67]. The main idea behind all these methods is to perform some transformations at the netlist level of the design in order to achieve certain optimization goals such as timing, area, and power. These transformations usually involve gate substitutions, wire additions and wire deletions.

The algorithms that drive the transformation in the above literature differs; some work uses ATPG techniques [35], some other uses redundancy addition/removal [14] [16] [19] [67] and some work incorporates symbolic techniques [45] [48]. However, the important observation is that most of the proposed transformations are very similar to the correction scheme proposed in [2] and used in this thesis.

The following example, adapted from [19], outlines an optimization procedure based on redundancy addition/removal.

**Example 23** *Consider the* irredundant *circuit in Fig. 6.4(a) where the shaded wire is the target wire to be removed. The additional connection from $O_1$ to $G_9$ (Fig. 6.4(b)) is redundant, that is, it does not alter the functionality of the circuit at $O_2$. However, adding this extra wire, wires $G_1$–$G_4$ and $G_6$–$G_7$ become redundant. These wires and some of the gates associated with them can be removed, resulting in the optimized circuit of Fig. 6.4(c).*

The procedure described in the example above has a direct relation to the diagnosis/rectification procedures described in this thesis. To see that, observe that since the circuit of Fig. 6.4(a) is irredundant, removing wire $G_1$–$G_4$ will make it erroneous. Consequently, the diagnosis algorithm with *explicit* enumeration of error tuples followed with rectification will indeed return the extra wire in Fig. 6.4(b) in the list of potential corrections.

Moreover, consider the situation where we allow more than one simultaneous wire additions/removals and gate substitutions. Considering the results in Chapter 5, we observe that multiple circuit perturbations [1] will result in a greater number of modifications

[1]A circuit perturbation is a single wire addition, wire removal, or gate substitution [14]

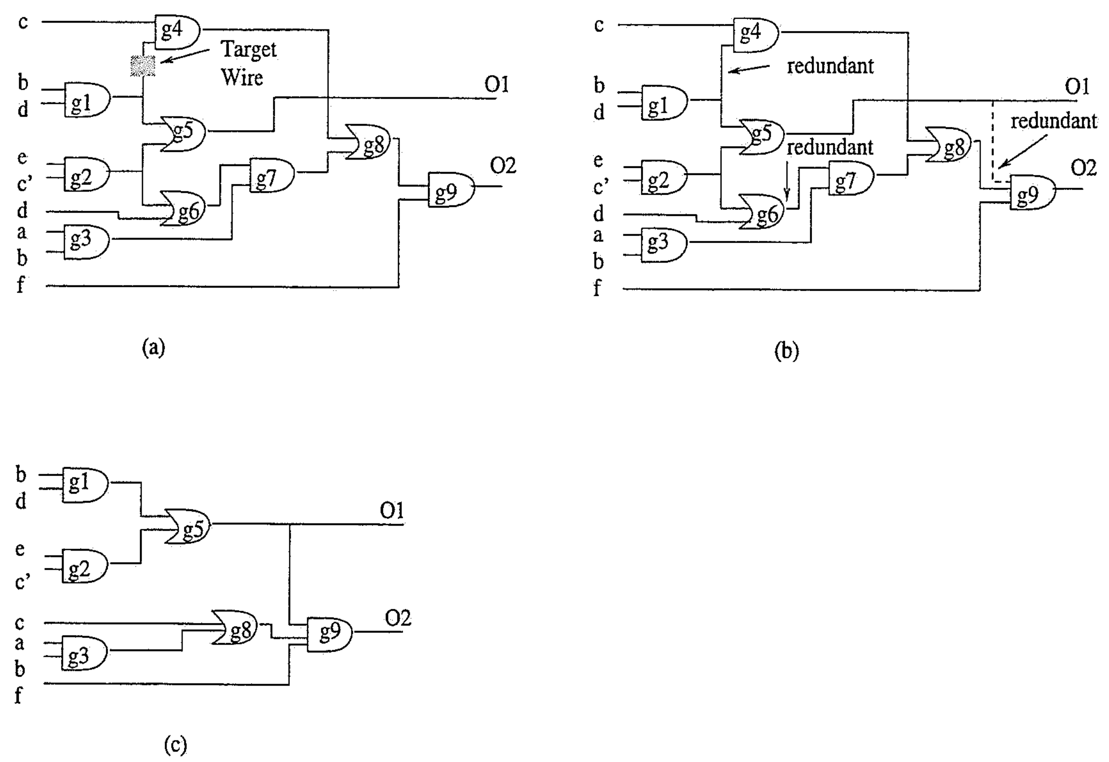


**Figure 6.4** Optimization via Redundancy Addition/Removal

that rectify the design. This is because a bigger number of perturbations gives a better opportunity to exploit the don't care space of the design. However, selecting a set of perturbations and corrections that yields the best optimization results is still an open problem.

## 6.4 Conclusion

With the increase of logic size and complexity, logic design errors can occur. Logic design errors are functional mismatches between the logic implementation and the specification. In this thesis, we examined the problem of multiple design error diagnosis and correction. Experimental results were used to exhibit the robustness of the proposed approach and confirm theoretical results.

For error diagnosis, we proposed two different techniques. The first guarantees to return the actual and all equivalent error locations that rectification can be performed. However, the complexity of this method increases exponentially with the number of error locations which makes it not applicable to circuits corrupted with a high cardinality of errors. For this reason, a non exhaustive on the error space diagnosis method was developed. The method exhibits good run–time performance although it does not guarantee to return all possible modification locations. Both methods use test–vector simulation as the underlying technique, thus, they are applicable to all circuits.

For error correction, two methods were proposed, one based on test–vector simulation, and one based on Boolean function manipulation techniques. In our experimental results, we compare the quality of test–vector simulation for multiple design error diagnosis and correction with the one offered by symbolic techniques, and we conclude that test–vector simulation is indeed an attractive alternative.

Since many resynthesis methods rely on circuit modifications similar to the design errors discussed in this work, we believe that the tools and techniques developed in this thesis will be helpful to provide solutions to problems in other CAD areas.

# References

[1] E.J.Aas, K.Klingsheim, and T.Steen, "Quantifying design quality: A model and design experiments," in *Proc. of EURO–ASIC, pp.172–177,* 1992.

[2] M.S.Abadir, J.Ferguson, and T.E.Ferguson, "Logic Verification via Test Generation," in *IEEE Trans. on Computer–Aided Design, vol.7, pp.138–148,* January 1988.

[3] M.Abramovici, P.R.Menon, and D.T.Miller, "Critical Path Tracing: An Alternative to Fault Simulation," in *IEEE Design and Test of Computers, vol.1, pp.89–93*, February 1984.

[4] S.B.Akers, B.Krishnamurthy, S.Park, and A.Swaminathan, "Why is Less Information from Logic Simulation More useful in Fault Simulation?, " in *Proc. of the IEEE Int'l Test Conf., pp.786–800,* 1990.

[5] H.A.Asaad, and J.Hayes, "Design Verification via Simulation and Automatic Test Pattern Generation," in *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design, pp.174–180,* 1995.

[6] C.L.Berman, and L.H.Trevillyan, "Functional Comparison of Logic Designs for VLSI Circuits," in *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design, pp. 468–471,* 1991.

[7] K.S.Brace, R.L.Rudell, and R.E.Bryant, "Efficient Implementation of a BDD package," in *Proc. ACM/IEEE Design Automation Conference, pp.40–45,*, 1990.

[8] D.Brand, A.Drumm, S.Kundu, and P.Narain, "Incremental Synthesis," in *Proc. of the IEEE/ACM Int'l Conference on Computer–Aided Design, pp.14-18,* 1994.

[9] D.Brand, "Verification of Large Synthesized Designs," in *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design, pp. 534–537,* 1993.

[10] M.Abramovici, M.Breuer, and A.Friedman, "Digital Systems Testing and Testable Design," *Computer Science Press,* 1990.

[11] F.M.Brown, "Boolean Reasoning: The Logic of Boolean Equations," *Kluwer Academic Publishing,* 1990.

[12] R.E.Bryant, "Graph–Based Algorithms for Boolean Function Manipulation," in *IEEE Trans. on Computers, vol.C–35, no.8, pp.677–691,* 1986.

[13] R.E.Bryant, "On the Complexity of VLSI implementation and graph representations of boolean functions with application to integer multiplication," in *IEEE Trans. on Com[puters, Vol.40, No.2, pp.205-213,* Feb.1991.

[14] S.-C.Chang, K.-T.Cheng, N.-S.Woo, and M.M.-Sadowska, "Layout Driven Logic Synthesis for FPGAs," in *Proc. of ACM/IEEE Design Automation Conference, pp.308–313,* 1994.

[15] K.C.Chen, Y.Matsunaga, M.Fujita and S.Muroga, "A resynthesis approach for network optimization," in *Proc. of ACM/IEEE Design Automation Conference, pp.667–691,* 1986.

[16] K.-T.Cheng, and L.A.Entrena, "Multi-level logic optimization by redundancy addition and removal," in *Proc. of European Conference on Design Automation, pp.373–377,* 1993.

[17] P.-Y.Chung, "Diagnosis and Correction of Logic Design Errors," Ph.D. Thesis, Dept. of Electrical and Computer Engineering, Univ. of Illinois at Urbana–Champaign, 1993.

[18] C.Ebeling, "GeminiII: A Second Generation Layout Validation Tool," in *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design, pp.322–325,* 1988.

[19] L.A.Entrena, and K.-T.Cheng, "Combinational and Sequential Logic Optimization by Redundancy Addition and Removal," in *IEEE Trans. on Computer–Aided Design of Integrated Circuits and Systems, vol.14, no.7, pp.909-916,* July 1995.

[20] M.Fujita, Y.Tamiya, Y.Kukimoto, and K.-C.Chen, "Application of boolean unification to combinational synthesis," in *Proc. of the IEEE/ACM Int'l Conference on Computer–Aided Design, pp.510–513,* 1991.

[21] P.Goel, "An implicit enumeration algorithm to generate test for combinational circuits," in *IEEE Trans. on Computers, vol.C-30, pp.215–222,* March 1981.

[22] P.-Y.Chung, Y.-M.Wang, and I.N.Hajj, "Logic Design error diagnosis and correction," in *IEEE Trans. on VLSI Systems, vol.2, pp.320-332,* September 1994.

[23] P.-Y. Chung, and I.N.Hajj, "Diagnosis and Correction of Multiple Design Errors in Digital Circuits," in *IEEE Trans. on VLSI Systems, vol. 5, no. 2, pp. 233-237,* June 1997.

[24] R.B.Hitchock, "Timing Verification and Timing Analysis Program," in *Proc. ACM/IEEE Design Automation Conference, pp.594–604,* 1992.

[25] S.-Y.Huang, K.-C.Chen, and K.-T.Cheng, "Error Correction Based on Verification Techniques," in *Proc. of ACM/IEEE Design Automation Conference, pp.258–261,* 1996.

[26] S.-Y.Huang,K.-C.Chen, and K.-T.Cheng, "Incremental Logic Rectification," in *Proc. of IEEE VLSI Test Symposium, pp.143–149,* 1997.

[27] S.-Y.Huang, K.-T.Cheng, and K.-C.Chen, "ErrorTracer: A Fault Simulation–Based Approach to Design Error Diagnosis, " in *Proc. of IEEE Int'l Test Conference, pp.974–981,* 1997.

[28] S.-Y.Huang, K.-T.Cheng, K.-C.Chen, and J.-Y.J.Lu, "Fault–Simulation Based Design Error Diagnosis for Sequential Circuits," in *Proc. ACM/IEEE Design Automation Conference, pp.667–691,* 1998.

[29] S.-Y. Huang and K.-T. Cheng, "Formal Equivalence Checking and Design Debugging," *Kluwer Academic Publishers,* May 1998.

[30] W.Hunt, "Microprocessor Design Verification,", in *Journal of Automated Reasoning, vol5(4), pp.429–460,* Dec. 1989.

[31] J.Jain, J.Bitner, D.S.Fussell, and J.A.Abraham, "Probabilistic Design Verification," in *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design, pp.468–471,* 1991.

[32] A.Kuehlmann, D.I.Cheng, A.Srinivasan, and D.P.LaPotin, "Error Diagnosis for Transistor Level Verification," in *Proc. of Design Automation Conf., pp.218–224,* 1994.

[33] Y.Kukimoto, M.Fujita, "Rectification Method for Lookup–Table Type FPGA's," in *Proc. of Int'l Conf. on Computer Aided Design, pp.54–61* 1992.

[34] W.Kunz, D.Pradhan, and S.Reddy, "A Novel Framework for Logic Verification in a Synthesis Environment," in *IEEE Trans. on Computer–Aided Design. vol.15, pp.20–32,*, 1996.

[35] W.Kunz, and P.R.Menon, "Multi–Level Logic Optimization by Implication Analysis," in *Proc. of Int'l Conf. on Computer Aided Design, pp.6–13,* 1994.

[36] W.Kunz, and D.Stoffel, "Reasoning in Boolean Networks: Logic Synthesis and Verification Using Testing Techniques,", *Kluwer Academic Publishers,* 1997.

[37] S.-Y.Kuo, "Locating logic design errors via test generation and don't–care propagation," in *Proceedings of European Design Automation Conference, pp.466–471,* 1992.

[38] H.-T.Liaw, J.-H.Tsaih, and C.-S.Lin, "Efficient Automatic Diagnosis of Digital Circuits," in *Proc. of IEEE/ACM Int'l Conference on Computer–Aided Design, pp.464–467,* 1990.

[39] C.-C. Lin, K.-C. Chen, S.-C. Chang, and M.M-.Sadowska, "Logic Synthesis for Engineering Change," in *Proc. of ACM/IEEE Design Automation Conference, pp.647–652,* 1995.

[40] J.C.Madre, O.Coudert, and J.P.Billon, "Automating the diagnosis and the rectification of digital errors with PRIAM," in *Proc. of the IEEE/ACM Int'l Conference on Computer-Aided Design, pp.30-33,* June 1989.

[41] Y.Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," in *Proc. of Design Automation Conf., pp.629–634,* 1996.

[42] P.C.Maxwell, and R.C.Aitken, "Biased Voting: A Method for Simulating Cmos Bridging Faults in the Presence of Variable Gate Logic Thresholds," in *Proc. IEEE Int'l Test Conference, pp.63–72,* 1993.

[43] P.R.Menon, Y.Levendel, and M.Abramovici, "SCRIPT: A Critical Path Tracing Algorithm for Synchronous Sequential Circuits," in *IEEE Trans. on Computer Aided Design, vol.10, Pp.738–747,*, June 1991.

[44] G.DeMicheli, "Synthesis and Optimization of Digital Circuits," McGraw-Hill, Inc., 1994.

[45] S.Muroga, Y.Kambayashi, H.C.Lai, and J.N.Culliney, "The transduction method – design of logic networks based on permissible functions, " in *IEEE Transactions on Computers, pp.1404–1424,* 1989.

[46] F.Najm, "A Survey of Power Estimation Techniques in VLSI Circuits," in *IEEE Trans. on Very Large Scale Integration Systems, 2(4), pp.446–455,* Dec. 1994.

[47] T.M.Niermann, and J.H.Patel, "HITEC: A test generation package for sequential circuits," in *Proc. of European Automation Conf., pp.214–218,* 1991.

[48] R.V.Panda, "Synthesis Techniques for VLSI Low–Power Circuits," Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, September 1996.

[49] G.Pelz, and U.Roettcher, "Circuit Comparison by Hierarchical Pattern Matching," in *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design, pp.322-325,* 1991.

[50] I.Pomeranz and S.M.Reddy, "On diagnosis and correction of design errors," in *IEEE Trans. on Computer–Aided Design, vol.14, pp.255–264,* February 1995.

[51] I.Pomeranz and S.M.Reddy, "On error correction in macro-based circuits," in *IEEE Trans. on Computer–Aided Design, pp.1088–1100,* 1997.

[52] A.Raghunathan, and S.T.Chakradhar, "Acceleration Techniques for Dynamic Vector Compaction," in *Proc. of the IEEE/ACM Intl. Conf. on Computer–Aided Design, pp.310–317,* 1995.

[53] R.Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," in *IWLS Workshop Notes, session 3a, pp.1–12,* 1993.

[54] K.A.Tamura, "Locating Functional Errors in Logic Circuits," in *Proc. of the Design Automation Conference, pp. 185–191,* June 1989.

[55] M.Tomita,H.-H.Jiang,T.Yamamoto, and Y.Hayashi, "An algorithm for locating design errors," in *Proc. of IEEE/ACM Int'l Conference on Computer–Aided Design, pp.468–471,* 1990.

[56] M.Tomita, T. Yamamoto, F.Sumikawa, and K. Hirano, "Rectification of Multiple Logic Design Errors in Multiple Output Circuits," in *Proc. of the Design Automation Conference, pp.212–217,* 1994.

[57] M.Tomita, N.Suganuma, and K.Hirano, "Pattern Generation for Locating Logic Design Errors," in *IEICE Trans. Fundamentals, vol.E77-A,* 1994.

[58] A.G.Veneris, and I.N.Hajj, "A Fast Algorithm for Locating and Correcting Simple Design Errors in VLSI Digital Circuits," in *Proc. of 7th IEEE Great Lakes Symposium on VLSI, pp.45–50,*, 1997.

[59] A.G.Veneris, and I.N.Hajj, "Error Diagnosis and Correction in VLSI Digital Circuits," in *Proc. of IEEE Midwest Symposium on Circuits and Systems, pp.1005–1008,* 1997.

[60] S.Venkataraman, "Simulation and Deduction Based Techniques for Fault Diagnosis," Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, September 1997.

[61] S.Venkataraman, and W.K.Fuchs, "A Deductive Technique for Diagnosis of Bridging Faults," in *Proc. IEEE/ACM Int'l Conf. on Computer Aided Design, pp.562–567,* 1997.

[62] Y.Watanabe and R.K.Brayton, "Incremental Synthesis for Engineering Changes," in *Proc. of IEEE/ACM Int'l Conf. on Computer Design, pp.40-43,* 1991.

[63] A.M.Wahba, and E.J.Aas, "Verification and diagnosis of digital systems by ternary reasoning," in *Proc. IFIP WG10.2 Advanced Research Working Conf. on Correct Hardware Design Methodologies,* 1993.

[64] A.M.Wahba, and D.Borrione, "Design Error Diagnosis in Sequential Circuits," in *Proc. of Correct Hardware Designs and Verification Methods, Lecture Notes in Computer Science No.987, pp.171–188, Spriger Verlag,* 1995.

[65] A.M.Wahba, and D.Borrione, "A Method for Automatic Design Error Location and Correction in Combinational Logic Circuits," in *Journal of Electronic Testing, Theory, and Applications, vol.8, no.2, pp.113-127,* April 1996.

[66] J.A.Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy, "Fault Simulation for Structured VLSI," in *VLSI Systems Design, pp. 20–32,* Dec. 1985.

[67] Q.Wang, and S.B.K.Vrudhula, "Multi–Level Optimization for Low Power Using Local Logic Transformations," in *Proc. ACM/IEEE Design Automation Conference, pp.270–277,* 1996.

[68] N.Yanagida, H.Takahashi, Y.Takamatsu, "Multiple Fault Diagnosis in Sequential Circuits Using Sensitizing Sequence Pairs," in *Proc. of Fault Tolerant Computing Systems, pp.86–95,* 1996.

# Vita

Andreas G. Veneris was born in Athens, Greece, in 1969. He received his Diploma degree in Computer Engineering and Informatics from University of Patras, Greece, in 1991, and M.Sc. degree in Computer Science from University of Southern California, Los Angeles, in 1992. During his undergraduate years he was the co-author of a book on the programming language FORTRAN. From 1992 to 1998 he was employed as a research assistant at the University of Southern California and at the Coordinated Science Laboratory at the University of Illinois. In January 1998 he was a Visiting Lecturer for the University of Illinois, Department of Computer Science, which he will be joining again as Visiting Assistant Professor after completing his Ph.D. degree.

Since 1984, Andreas G. Veneris has been working as a music journalist for international publications. He also worked for an early version of Mosaic and collaborated for the first Internet cybercast (Grammy Awards, Los Angeles, 1995). His research interests include VLSI synthesis and testing, CAD for VLSI, and combinatorics.

He is currently a member of AAAS, IEEE, Mufon, and The Planetary Society.