REDUNDANCY IDENTIFICATION IN LOGIC CIRCUITS USING EXTENDED IMPLICATION GRAPH AND STEM UNOBSERVABILITY THEOREMS

BY VISHAL J. MEHTA

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Prof. Vishwani D. Agrawal and Prof. Michael L. Bushnell

and approved by

____

____

____

New Brunswick, New Jersey

May, 2003

ABSTRACT OF THE THESIS

Redundancy Identification in Logic Circuits using Extended Implication Graph and Stem Unobservability Theorems

by Vishal J. Mehta

Thesis Director: Prof. Vishwani D. Agrawal and Prof. Michael L. Bushnell

We make significant improvements in the implication graph and transitive closure technique to identify redundant faults in combinational circuits. We use an extended implication graph with higher order implications represented by anding nodes. We provide several enhancements to the computation of the transitive closure leading to an improved redundancy identification technique. Key contributions of this thesis are:

1. Completion of the existing method:
   (a) Implementation of all higher order relationships for observability and controllability nodes.
   (b) Implementation of the contrapositive rule in the transitive closure computation.
2. New concepts introduced:

(a) The fixed-value theorem to identify redundancies due to fixed signals in the circuit.

(b) Two new theorems to identify redundancies caused by the unobservability of certain fanout stems.

Implementing all higher order anding nodes, explores more comprehensively controllability and observability relationships, and in turn identifies more redundancies. The fixed-value theorem adds unconditional edges from all other nodes in the graph to the fixed nodes and then calculating transitive closure recursively until no more fixed nodes are found. We use the dominators defined in graph theory to identify the stem unobservability. According to the first stem unobservability theorem, a fanout stem is unobservable if its dominator set has fixed values and:

1. The stem is not fixed, or
2. The stem is fixed, but a local change in the value of the stem does not change the dominator set values.

The second stem unobservability theorem states that a fanout stem is unobservable if its dominator set is unobservable and:

1. The stem is not fixed, or
2. The stem is fixed and dominator set remains unobservable when the stem signal is locally changed.

Because of all these improvements, we obtain better performance than all previous fault-independent methods. Our execution speed remains much faster than any exhaustive ATPG. For example, in the ISCAS benchmark circuit c3540, we identify 111 out of 131 redundant faults in 16.2 sec CPU time on Sparc-5 machine. In comparison, TRAN, an ATPG program from the Rutgers University, can find all 131 redundancies in 24.9 sec CPU time. We demonstrate a limitation of this technique in identifying unobservable fanout stem redundancies that may be caused by neither fixed nor unobservable dominators.

Acknowledgements

I am thankful to Prof. Vishwani D. Agrawal, my advisor, for his direction of this work. I gratefully acknowledge Prof. Michael L. Bushnell, my co-advisor, for reading the thesis and suggesting many improvements, and Prof. Manish Parashar, a member of my thesis committee, for lending his expertise and time. I am also thankful to my friends, Kunal, Bhavin, Kedar, Krishna, Karthik and Tezaswi, and other colleagues in the VLSI laboratory.

Dedication

To my parents and sister

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Tables x

List of Figures xi

1. Introduction 1

1.1. Motivation 1

1.1.1. Why do we need to implement partial implications? 2

1.1.2. What is the contrapositive rule and why do we need to check it? 2

1.1.3. What is node fixation and why do we need to implement it? 2

1.1.4. Why is fanout stem observability a difficult problem? . . . 3

1.2. Problem statement 3

1.3. Original contribution of this thesis 3

1.4. Organization of the thesis 4

2. Prior Work 5

2.1. Introduction 5

2.2. Redundancy identification techniques 6

2.2.1. ATPG based techniques 6

2.2.1.1. Boolean satisfiability and implication graph methods 9

2.2.1.2. Learning algorithms 10

2.2.1.3. Implication graph ATPG algorithms 12

2.2.2. Testability analysis 13

2.2.3. Fault-independent techniques 16

2.3. Graph theory 18

2.3.1. Dominators 18

2.4. Summary 19

3. Boolean Satisfiability Equations for Controllability and Observability and other Definitions 20

3.1. Introduction 20

3.2. Controllability relationships for an AND gate 20

3.3. Observability relationships 24

3.4. Conditions for identifying redundant faults 26

3.5. Other definitions 27

3.6. Summary 29

4. Direct and Partial Implications and Transitive Closure 31

4.1. Controllability edges 31

4.2. Observability edges 32

4.3. Controllability edges for fanout stem and its branches 33

4.4. Example circuits 34

4.5. Summary 37

5. Fixed-Value Theorem and Contrapositive Rule Implementation 38

5.1. Fixed-value theorem 38

5.2. Example circuits justifying the fixed-value theorem implementation 39

5.3. Contrapositive rule implementation 41

5.4. Limitation 42

5.5. Summary 43

6. Fanout Stem Unobservability Theorems 44
6.1. Fanout stem unobservability and its effect on identifying other redundancies 44
6.2. Theorems to identify unobservable fanout stems 46
6.3. Example circuits justifying implementation of stem unobservability theorems 49
6.4. Summary 51

7. Results 52
7.1. Redundancies after each level of implementation 52
7.1.1. Previously unimplemented direct and partial implications . 52
7.1.2. Fixed-value theorem 54
7.1.3. Stem unobservability 55
7.1.4. Comparison 55
7.2. Analysis of the benchmark circuit results 57
7.3. Summary 58

8. Algorithm and Complexity 59
8.1. Algorithm 59
8.1.1. A detailed explanation of each step of algorithm 60
8.2. Calculations for number of nodes and edges 60
8.2.1. Controllability and observability nodes 61
8.2.2. Anding nodes 63
8.2.3. Direct and partial implications 63
8.3. Time complexity 64
8.4. Summary 65

9. Conclusions and Future Work 66
9.1. Limitations 66
9.2. Future work 66

References 69

List of Tables

3.1. Boolean false function for two-input Boolean gates. 21

3.2. Boolean false function for observability relationships for two-input Boolean gates. 26

7.1. Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after implementations of Section 7.1.1 53

7.2. Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after implementations of Section 7.1.2 54

7.3. Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after implementations of Section 7.1.3 55

7.4. Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after all the implementation and comparison with other fault-independent methods and ATPG technique. 56

List of Figures

2.1. Binary decision diagram [9]. 7

2.2. Example circuit showing the recursive learning procedure. 12

2.3. Example circuit showing that reconvergent fanouts cause correlation among signals. 14

2.4. Example circuit [9] showing the working of TOPS [42]. 18

3.1. A Boolean AND gate. 21

3.2. Direct implications for the two-input AND gate shown in the Figure 3.1. 23

3.3. Partial implications along with direct implications for the two-input AND gate shown in the Figure 3.1. 23

3.4. Observability AND gate for observing signal a of AND gate shown in Figure 3.1. 24

3.5. Implication graph for observing signal a of Boolean AND gate of Figure 3.1. 25

4.1. A Boolean AND gate with multiple fanouts. 32

4.2. Implication graph for signal a of Boolean AND gate of Figure 4.1. 32

4.3. A fanout stem. 34

4.4. An implication graph for controllability of the fanout stem and its branches. 34

4.5. Example circuit showing the limitation of the previous work [26]. . 35

4.6. Implication graph for identifying redundant faults in the circuit of Figure 4.5. 36

4.7. An example circuit showing a limitation of the previous work [26]. 36

4.8. Another example circuit showing a limitation of the previous work [26]. 36

5.1. Example circuit showing the implementation of node fixation. . . 40
5.2. Implication graph showing how the primary output redundant fault was identified redundant using Theorem 5.1.1 for example circuit of Figure 5.1. 41
5.3. Example circuit showing the consistency of limitations (of previous technique [26]) identified in Section 5.1. 41
5.4. Example circuit showing the decision taking inability of our algorithm. 43
6.1. Example circuit illustrating the cases of stem observability and unobservability (all faults shown are redundant). 45
6.2. Example circuit illustrating the cases of stem observability and unobservability. 45
6.3. Example circuit illustrating the cases of stem observability and unobservability. 46
6.4. Example circuit justifying the implementation of Theorem 6.2.1. . 49
6.5. Example circuit justifying the implementation of Theorem 6.2.2. . 50
6.6. Example circuit showing the limitation of our method. 50
9.1. Example circuit showing the working of fault equivalence. 67

Chapter 1
Introduction

The contributions of this work are: Improving the existing method for redundancy identification using transitive closure [17, 26, 49] that identifies redundant stuck-at faults without targeting specific faults. The details of the method that we improve upon are given in Chapter 3. Its complexity is polynomial in the number of gates in the circuit. Our improvements consist of incorporating all partial implications in the implication graph, including the effects of fixed-value nodes and contrapositives in the transitive closure, and new theorems on the unobservability of fanout stems. In this chapter, we address several basic questions whose answers provide the motivation for this work: Why do we need to implement partial implications? Why do we need to check the contrapositive rule? What is node fixation and why do we need to implement it? Why is fanout stem observability a difficult problem?

1.1 Motivation

In this section, we try to answer the above mentioned questions that motivated this work. In this research the implication graph representation of a digital function is used. A node in the implication graph represents the signal state of a line and an edge represents a relationship between signals. The graph also contains observability nodes that represent the observability status (true or false) of signal lines with respect to primary outputs of the circuit.

1.1.1 Why do we need to implement partial implications?

The implication graph is obtained from the Boolean false function [26, 35, 49] of a logic gate, explained in detail in Chapter 3. Traditionally, an implication graph includes the pair-wise relations between Boolean variables. A relation involving more than two variables leads to a situation where a variable may partially imply another variable. Partial implications, or the higher order implications as they are sometimes called, are represented through anding nodes [26, 34, 35, 36, 47]. An anding node has n input edges and one output edge, where n corresponds to the number of inputs of the digital gate being represented. If all of the edges are not implemented then the implication graph will be an incomplete representation of the Boolean function of the circuit and hence many redundancies will not be identified.

1.1.2 What is the contrapositive rule and why do we need to check it?

According to the contrapositive rule if a Boolean variable x implies another Boolean variable y, then $\overline{y}$ must imply $\overline{x}$ [69, 80]. This rule is followed implicitly when we build the implication graph with the input-output relationships of a gate, but it is important to check whether this rule is obeyed after the transitive closure is calculated, because when it is not obeyed we would be missing some implications and hence some redundancies may not be identified.

1.1.3 What is node fixation and why do we need to implement it?

In an implication graph, if an edge exists between a node p and its complement $\overline{p}$, then that means that $\overline{p}$ is always true, or the signal p always holds to 0. Previous algorithms for transitive closure do not correctly deal with the fixed nodes and hence some redundancies caused by fixed signals are not found. We give a novel

method of adding unconditional implications from all nodes to the fixed nodes.

1.1.4 Why is fanout stem observability a difficult problem?

The controllabilities of the fanout branches depend on the controllability status of the fanout stem. But the observabilities of the fanout branches and the stem do not have any well-defined relationship. A stem may not be observable even if some or all of its fanout branches are observable. Thus, it is not an easy task to implement observability edges from fanout branches to their stem. In the previous work, therefore, all redundancies due to the unobservability of fanout stems could not be identified.

1.2 Problem statement

The problems solved in this thesis are: *To complete the implication graph and transitive closure procedures with all direct and partial implications and to find techniques that identify redundancies in combinational circuits caused by fanout stem unobservability.*

1.3 Original contribution of this thesis

The following are the original contributions of this thesis:

- Fixed-value theorem: We check for the fixed nodes in transitive closure. If a fixed node is encountered in the graph traversal, we implement unconditional edges from all the other nodes in the graph to the fixed node. This new concept improved our results on benchmark circuits by identifying additional redundancies. This is discussed in Chapter 5.

- Contrapositive rule: We check the contrapositive rule to implement valid edges, if they are not already implicitly present in the graph (Chapter 5).

- Stem unobservability theorems: We derive two new theorems to solve the stem unobservability problem. The first theorem identifies an unobservable stem from the controllability status of signals in the dominator set of the stem. The second theorem determines the unobservability of the stem from the observability status of signals in the dominator set of the stem. See Chapter 6.

1.4 Organization of the thesis

Chapter 2 describes the related techniques already in the literature, in the area of redundancy identification. Chapter 3 describes how the implication graph is built for controllability and observability of the signals in the given circuit. Chapter 4 describes the completion of the previous technique with respect to direct and partial implications, with example circuits showing the improvements made. Chapter 5 describes the new enhancements to the transitive closure by the node fixation concept and contrapositive rule, with example circuits showing the improvements in the redundancy identification. Chapter 6 describes the new theorems to solve the fanout stem unobservability problem, with example circuits showing the improvements in redundancy identification. Chapter 7 describes results for ISCAS'85 and ISCAS'89 benchmark circuits. Chapter 8 gives the complete algorithm with a complexity analysis. Chapter 9 presents the conclusion and future work.

Chapter 2
Prior Work

2.1 Introduction

Redundancy identification is a necessary part of modern design and test strategies. Redundancy in digital logic increases the number of gates in the circuit, and in turn increases the chip area and reduces speed. It also complicates the *automatic test pattern generation* (ATPG) process. In combinational circuits, an undetectable fault is always caused by redundancy. The presence of redundancy does not cause any change in the function of the circuit. In general, test generation is not possible for *redundant faults* in combinational circuits. There are three basic methods for identifying redundant faults:

- ATPG–based method: This type of technique uses an exhaustive test pattern generation to determine whether or not each fault has a test. A complete test generation thus produces a list of all redundant faults [9, 17, 58].

- Testability analysis: This type of technique involves analysis of circuit topology and does not produce test vectors as in the case of the ATPG technique. It has linear complexity and it is an approximate analysis [4, 7, 9, 30, 31, 63]. Because of its approximate nature testability analysis does not identify redundancies.

- Implication-based method: This type of technique also analyzes circuit topology and function without test vectors [3, 26, 27, 33, 39, 49, 56]. The analysis may involve controllability and observability of signals. It has a polynomial complexity and can identify a subset of redundancies (shown to

be empirically linear by Gaur *et al.* [27]).

2.2 Redundancy identification techniques

2.2.1 ATPG based techniques

Eldred [23] began the era of structural logic circuit testing. Roth's *D-Algorithm* [64], at IBM marked the true beginning of systematic generation of tests for hardware faults in digital computers and laid the mathematical basis for test pattern generation. ATPG is the process of generating patterns to test a circuit, which is described as a logic-level netlist [9]. ATPG algorithms are multi-purpose, *i.e.*, they can generate circuit test-patterns, can find *redundant faults* and can prove whether one circuit implementation matches another circuit implementation [9, 51, 52, 79]. The basic principle of an ATPG algorithm is to inject a fault in the circuit and then to use various mechanisms to activate the fault and cause its effect to propagate through the hardware and manifest itself at a circuit output. If the output value changes from the expected value, then the fault is detected [9]. Eldred [23] switched the field from functional to structural testing [9]. The first publication about s-a-0 and s-a-1 faults for test generation was by Galey *et al.* [25]. Seshu and Freeman later mentioned the stuck-at fault model for parallel fault simulation [73]. Poage presented a theoretical analysis of stuck-at faults [61]. Ibarra and Sahni [37] analyzed the computational complexity of ATPG and found it to be an NP-complete problem.

Goel [28, 29] first used *binary search trees* for combinational ATPG, where in all possible input combinations are represented. The ATPG algorithm implicitly searches this tree to find a test-pattern and in the worst case must examine the entire tree to prove that fault is untestable. The number of tree leaves is

$$2^{\text{number of PIs}} \tag{2.1}$$

and that rises exponentially. Here, PI is the primary input. Lee [48] invented the *binary decision diagram* (BDD) that completely describes any switching function.

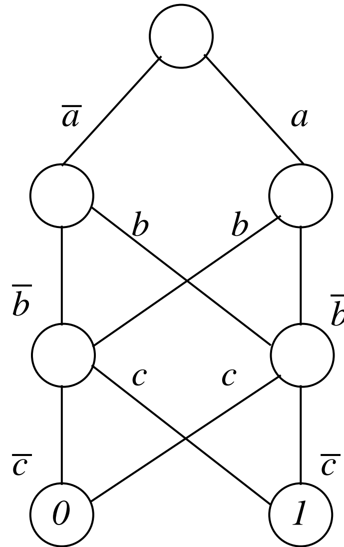

Figure 2.1: Binary decision diagram [9].

Akers [5, 6] applied BDDs to solve the problem of testing. In Figure 2.1, we start form the root node, and follow a path to one of the two bottom most nodes, 0 or 1, which gives the circuit output value. BDDs have been used for ATPG [24, 76], but suffer from the problems of computational intractability, particularly for multiplier circuits. One observes vast changes in computation time, depending on the order in which circuit PIs are expanded in the BDD [8]. ATPG is a complete procedure, because it searches through all possible input combinations, before saying that a fault is untestable. Thus, its computational complexity is exponential.

Roth [65] introduced a concept of the ATPG algebra, with the purpose of representing both the *good* and the *failing* circuit (or machine) values simultaneously. This has the advantage of requiring only copy of the circuit to determine signal values for both machines. Roth for this purpose introduced a *five-valued algebra*. Later, Muth [57] showed that in order to test finite state machines, the X (unknown) state must be used to represent the signals of the good or failing machines in addition to the 0 and 1 states. A *9-valued algebra* improved the combinational ATPG [11]. For computing the response of a logic gate to input

symbols of the algebra, we expand the symbols into the good machine and bad machine values. We then independently compute the logic gate response for both machines and combine the output values back into the algebra.

Agrawal and Agrawal [2] combined *random pattern generation* (RPG) with algorithmic ATPG. RPG uses a fault simulator to select useful patterns from those obtained with assigned input probabilities. Sellers *et al.* [71, 72] use *Shannon's expansion theorem* to characterize Boolean circuits. $F(x_1,..,x_k,..,x_n)$ can be expanded about variable x_k as:

$$F(X_1,..,x_k,..,x_n) = x_k F(x_1,..,1,..,x_n) + \overline{x_k} F(x_1,..,0,..,x_n) \tag{2.2}$$

Sellers *et al.* define the *Boolean difference* for the function F as:

$$\frac{\partial F}{\partial x_k} = F(x_1,..,1,..,x_n) \oplus F(x_1,..,0,..x_n) \tag{2.3}$$

Consider the fault on a line g in the circuit that is not a PI and has the function $g(X)$. The primary output is expressed as $G(X,g)$, where X is the vector at PIs, and G is the function of some PO. Then the *Boolean difference* of function G with respect to g will be:

$$\frac{\partial G}{\partial g} = G(X,1) \oplus G(X,0) \tag{2.4}$$

For g s-a-0 to be detected:

$$g(x) \;=\; 1 \; and \tag{2.5}$$

$$\frac{\partial G}{\partial g} = G(X,1) \oplus G(X,0) = 1 \tag{2.6}$$

Equation 2.5 indicates that in order to activate g s-a-0, we should sensitize the fault site to 1 and have the *Boolean difference* of the output with respect to fault site g shown in Equation 2.6 as 1. Unfortunately, due to high complexity the *Boolean difference* is not an efficient way to compute test patterns for large circuits. Most ATPG algorithms use path sensitization [64], which at the logic level of representation, consists of three steps:

1. Fault sensitization: Fault is activated by forcing the signal driving the targeted line to the opposite value from the fault value.

2. Fault propagation: Fault effect is propagated through one or more paths to a PO of the circuit to be observed.

3. Line justification: The internal signal assignments previously made to sensitize the fault or propagate its effect are justified by setting the PIs of the circuit.

In the second and third steps, we may find a conflict, where a necessary signal assignment contradicts some previously made assignments. This forces the ATPG algorithm to backtrack and make an alternative assignment.

2.2.1.1 Boolean satisfiability and implication graph methods

Boolean satisfiability means satisfying a Boolean expression [9]. The 2-SAT problem is solved by only satisfying the clauses having just two literals and it is solvable in polynomial time [22]. The 3-SAT problem is characterized by clauses having three literals and this has an exponential time complexity [9]. Chakradhar *et al.* [12, 14, 18] and Larrabee [46, 47] have derived Boolean satisfiability formulations for the ATPG problem. Given a target fault, one derives an energy function of a neural network or a Boolean product of sums expression in terms of signal variables of the circuit, such that any test for the target fault will minimize the energy function or satisfy the Boolean expression. The energy minimization is shown to be equivalent to a Boolean sum of products expression. Henftling [34, 35, 36] and others [75, 78] have extended this method. Henftling introduced an anding node concept, to represent the clause with three or more literals. An efficient way to minimize the energy function or to find satisfying variable assignments for the false or truth functions is an implication graph. This graph represents a signal line x of the circuit by two nodes, x and $\overline{x}$, and a relationship between such nodes is expressed with directed edges. For signal value 1 for x, node x assumes true state and for signal value 0, node $\overline{x}$ assumes true state. The implication graph can then be converted into a *transitive closure graph* [9, 13, 26, 49], so that when a node is

set to true, all nodes reachable from it are also set to true. This allows very efficient analysis of signal implications, because the transitive closure can determine more global signal relationships in the graph then any other branch-and-bound search method [9, 17]. Larrabee [47] presents a Boolean satisfiability method for generating test patterns for single stuck-at faults in combinational circuits. She generates test patterns in two steps: A formula expressing the *Boolean difference* between the good and faulty circuits is constructed and then a Boolean satisfiability algorithm is applied to the resulting formula. Equation 2.3 represents the Boolean difference of any function F with respect to its variable x_k. The set of tests for x_k stuck-at-0 is X_k and the set of tests for x_k stuck-at-1 is $\overline{X_k}$. The Boolean difference is $\partial F/\partial x_k$, where X_k is the function representing the output x_k of the sub-circuit. After generating the Boolean difference a directed acyclic graph representing the topological description of the circuit is developed, where circuit inputs, outputs, gates and fanout points are nodes, circuit lines form the edges, circuit outputs are source nodes and circuit inputs are sink nodes. The graph traversal is then used for test generation.

Niermann and Patel [58] present HITEC, a sequential circuit test generation package to generate test patterns for combinational and sequential circuits. A targeted D element technique greatly increases the number of possible mandatory assignments and reduces the over-specification of state variables that can sometimes result when using the standard PODEM algorithm [29]. The state knowledge of previously generated vectors for state justification, without the memory overhead of a state transition diagram, is used. Knowledge gained about fault propagation, by the fault simulator, is used to further investigate aborted faults.

2.2.1.2 Learning algorithms

Schulz *et al.* introduced Socrates [68, 69, 70], an ATPG program that does static and dynamic learning. The learning procedure systematically sets all determined signals to 0 and 1 to discover what other signal values are implied. These implications are saved in the circuit netlist in the form of implication arcs and are used

during search when they cause additional signals to be assigned. Socrates also uses dynamic learning, which invokes the learning procedure between search steps. This is costly but effective, because additional signal relationships can be learned when signals are already partially set in the circuit. Normal logic gate implications are not found to be worth learning but Boolean contrapositive rule implications are useful. The learning procedure is applied statically to the circuit before the search begins and frequently discovers redundant faults. Socrates [69] also uses the *constructive dilemma*, which is $[a = 0 \Rightarrow b = 0] \wedge [a = 1 \Rightarrow b = 0] \Rightarrow b = 0$, irrespective of value of a. Here, a and b are two signal lines in the circuit and $\Rightarrow$ means implication. The work done by Zhao *et al.* [80] also used the contrapositive rule, which states that if a variable x implies another variable y, then $\overline{y}$ must imply $\overline{x}$. They make use of the static implications, iterative method and set algebra. Their algorithm discovers many indirect implications, which are not discovered by dynamic learning without tremendous time cost.

Kunz and Pradhan [43] introduced the concept of recursive learning to accelerate the FAN ATPG algorithm. They applied Socrates-style learning recursively to determine even more circuit signals and learn them as circuit implications. Figure 2.2 illustrates recursive learning. Lines $i1$ and j are unjustified. They find the necessary assignments without using an implication stack and decision making, but by recursive learning which learns information about the current value assignments in the circuit. The main advance over Socrates-style learning is that the learning procedure is recursive, and the maximum recursion depth determines how much is learned about the circuit. The time complexity is exponential in the maximum recursion depth, but memory grows linearly with such depth. They showed many applications [44], which are elaborated in a book [45] by Kunz and Stoffel.

Bushnell and Giraldi [10] presented EST, a combinational circuit ATPG technique. It detects equivalent search states, which are saved for later use. The search space is learned and characterized using E-frontiers, which are circuit cut-sets induced by the implication stack contents. The search space is reduced by matching

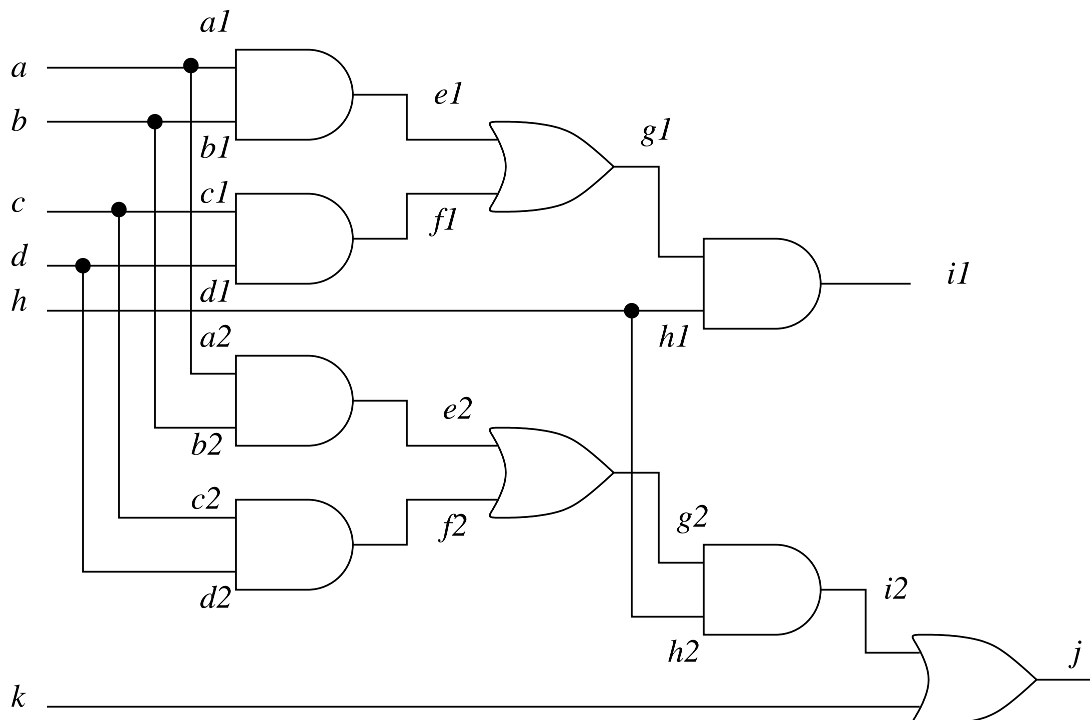

Figure 2.2: Example circuit showing the recursive learning procedure.

the current search state against previously-encountered search states and this reduces the length of the search. A calculus of redundant faults is also proposed which enables EST to make many more mandatory assignments before search, by effectively using the knowledge of prior faults proven to be redundant. This accelerates ATPG especially for hard-to test faults. Chen and Bushnell further used many of these features in a sequential circuit ATPG called SEST [19, 20].

2.2.1.3 Implication graph ATPG algorithms

Chakradhar *et al.* [13, 17] presented a test generation algorithm TRAN, where a test is obtained by determining signal values that satisfy a Boolean equation derived from the neural network model [12, 16] of the circuit incorporating necessary conditions for fault activation and path sensitization. Their program, TRAN, performs ATPG for huge circuits very rapidly [9]. A transitive closure is calculated from an implication graph consisting of binary and higher order implications. Transitive closure computation and decision-making are recursively incorporated

to identify redundancies in the circuit [3, 26, 27, 49]. The advantage of this approach was that TRAN made better signal assignment decisions using transitive closure calculations and had to back up far less often then the other ATPG methods [9]. Henftling *et al.* [34, 35, 36] presented an enhanced ATPG technique based on the implication graph. They represented the 3-SAT relationships directly in the graph. The graph contains direct and indirect implications. The indirect implications are implemented using anding nodes. Whenever ATPG assigns a signal, which converts a 3-SAT relation to a 2-SAT relation, some anding nodes automatically collapse into implication arcs. The forward edges represent input to output implications and backward edges represent the opposite implications. Several other satisfiability and implication based ATPG programs have been reported in the literature: GRASP [75], NEMESIS [47], TEGUS [77] and a program by Tafertshofer *et al.* [78].

2.2.2 Testability analysis

Raitu *et al.* [63] presents VICTOR, a linear testability analysis algorithm with linear storage requirements for combinational VLSI circuits. This algorithm identifies all potential single-fault redundancies and computes for every single-stuck fault location in the circuit both a controllability triplet and an observability triplet of label, weight and size. A sufficient test for controlling or observing a fault location constitutes a label. The weighted sum of the constrained primary inputs and fanout branches associated with a label defines the weight. Size is the count for all possible tests for controlling or observing a fault location.

Goldstein [30, 31] invented an algorithm to determine the difficulty of controlling and observing signals in digital circuits. His testability measures, also known as SCOAP measures, consist of 0-controllability and 1-controllability for each signal as measures of the effort (or difficulty) of setting the signal to a logic 0 or 1 value. There are additional measures for the observability of the signal. However, SCOAP has significant inaccuracies due to the assumption that signals

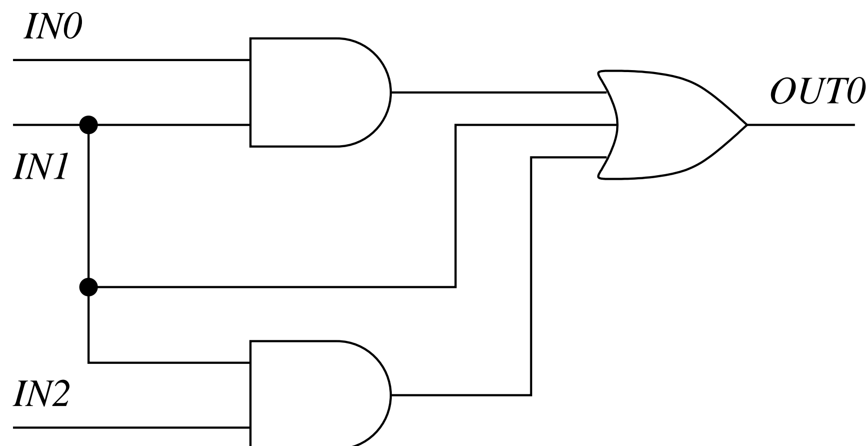

Figure 2.3: Example circuit showing that reconvergent fanouts cause correlation among signals.

at reconvergent fanout stems are independent, when they frequently are not. Figure 2.3 shows a circuit in which the reconvergent fanout stem signal *IN1* fans out to three different gates, and then all of these diverging paths reconverge at the OR gate producing the signal *OUT0*. The assumption of signal independence, which provides the speed of computation in SCOAP, severely reduces the accuracy of the measure. SCOAP has poor accuracy in predicting which individual faults will remain undetected and which will be detected. However, it is highly effective in predicting relative coverage levels for the entire circuit fault set [4]. The numerical values of the SCOAP measures range between zero and infinity. Higher values point to testability problems.

Another type of testability measures is probability-based. These overcome some limitations of SCOAP. Being related to signal probabilities, they are easier to interpret and their values always range between 0 and 1. The 1-controllability ($C1$) is the probability of a signal value on line l being set to 1 by a random vector. The 0-controllability ($C0$) is the probability of a signal value on line l being set to 0 by a random vector. Parker and McCluskey [59] give an algebraic procedure for computing line controllabilities. The capability of the exact signal probability calculations as well as that of their algorithm, is exponential. Seth and

Agrawal developed PREDICT, a numerical technique for this purpose [74]. They break the circuit into partitions, known as supergates, which completely include reconvergent fanouts. However, their algorithm has exponential computation cost in the number of reconvergent signals in a supergate. In the worst case, the entire circuit can be one supergate. While these techniques can compute exact probabilities, one must recognize that the computational complexity of the problem is exponential in the size of the circuit. Seth and Agrawal suggest several heuristics for approximation. Savir *et al.* [67] developed another procedure known as the *cutting algorithm.* They cut selected fanout lines to make the circuit a tree structure and initialize the cut lines to a controllability range [0,1]. The modified network has no reconvergent fanout and controllability ranges are readily computed for all lines.

The fault detection probability can be viewed as the 1-controllability of a signal that is the XOR of the good and faulty circuit outputs. Thus, the methods for computing probabilistic controllability can be used for computing fault detection probabilities. The disadvantage of this approach is that it almost doubles the size of the circuit for which controllabilities must be computed. An alternative approach is to define the observability $OB(l)$ of a line l as the probability of sensitizing a path from l to a PO. An obvious, though not always correct, definition of fault detection probability leads to a simple product formula. For example, the probability of detecting the s-a-0 fault on line l by a random input can be written as $C1(l) \times OB(l)$. Brglez [7] developed COP, a probabilistic testability algorithm, which derives computational efficiency by neglecting signal correlations. The error in the product formula for detection probability was observed by Savir [66]. Since the control and observation of a line are not independent events, their probabilities cannot be multiplied. To correct this error, Jain and Agrawal defined 0-observability and 1-observability of a line l as probabilities of l being observed, given that it assumed the appropriate value of 0 and 1 [41]. Since their observabilities are conditional probabilities, they can be multiplied to

appropriate controllabilities, without error, to obtain the probabilities of fault detection. The PREDICT algorithm of Seth and Agrawal [74] uses these definitions of observabilities to obtain exact detection probabilities.

2.2.3 Fault-independent techniques

Agrawal *et al.* [3] presented a Redundancy Identification algorithm, using transitive closure. Controllability variables representing logic values of signals and observability variables representing the observability status of the signals with respect to primary outputs are used. Binary terms are directly represented with the direct implications in the implication graph. However, for the higher-order terms, any signal assignments or relationships found from the transitive closure are substituted into the higher-order terms of the Boolean equation, some of which reduce to pair-wise terms. Such cases are iteratively included in the transitive closure until no more reductions are possible. In the final transitive closure the signals are examined for various conditions for redundancy, shown in Chapter 3. The advantage of this method is that transitive closure is calculated only once and not recomputed or updated separately for each fault as required in ATPG.

Gaur *et al.* [27] presented an improved redundancy identification algorithm that uses partial implications. These partial implications are the same as the higher-order implications represented as anding nodes of Henftling *et al.* [34, 35, 36]. In the presence of partial implications, a vertex can assume the true state when all vertices that partially imply it become true. Such graphs provide a more complete representation of a logic circuit than is possible with the conventional pair-wise implications. The use of partial implications or anding nodes gives the complete input-output relationship for any Boolean gate and provides improved results, with a linear empirical complexity. The work done by Gaur *et al.* is explained in detail, in Chapter 3.

Iyer and Abramovici [38, 39] presented FIRE, a fault-independent algorithm for combinational redundancy identification based on a simple concept that a fault

that requires conflicting assignments as necessary conditions for its detection is undetectable and hence redundant. Backtracking-based exhaustive search is not performed, unlike other fault-oriented ATPG algorithms, and FIRE identifies redundant faults without any search. They used the uncontrollability indicator $\overline{v_i}$ for every line i in the circuit that is uncontrollable to v_i. S_i indicates the set of redundant faults identified by each $\overline{v_i}$ and the final set of redundant faults is given by $S = \bigcap S_i$. They also give the Lemma – A stem s with all of its fanout branches marked as unobservable may also be marked as unobservable if for each fanout branch f of s, there exists at least one set of lines l_f, such that:

1. f is unobservable because of uncontrollability indicators on every line in l_f.

2. Every line in l_f is unreachable from s.

They also introduce a procedure to remove the redundant region in the circuit after identifying it [1]. A procedure, FILL [50], efficiently identifies a large subset of all illegal states in synchronous sequential circuits without assuming a global reset mechanism. Another procedure, FUNI [50], identifies sequentially untestable faults. Single stem analysis [40] and multiple stem analysis [60] were also introduced to identify sequential redundancy.

Menon *et al.* [33, 56] present methods to identify undetectable faults in a combinational circuit by analyzing the region between the fanout stems and reconvergent gates. They basically identify unpropagatable and undrivable redundant faults by single and multiple stem analysis. They identify controlling value paths to any gate G in the circuit from a stem s, make signal assignments of 0 and 1 to the stem s and check whether any assignment drives or propagates the stem faults. If none succeeds, the fault is identified as undetectable. They also present an algorithm to remove redundancy and simplify combinational circuits.

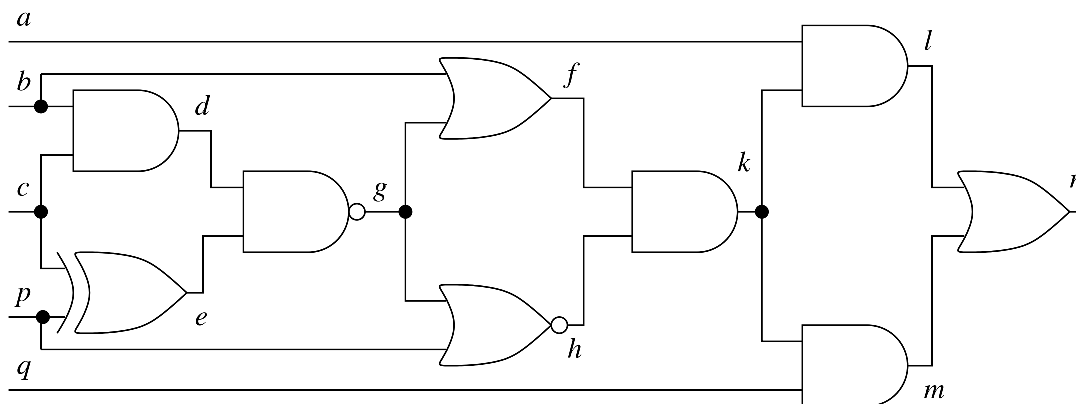

Figure 2.4: Example circuit [9] showing the working of TOPS [42].

2.3 Graph theory

2.3.1 Dominators

Kirkland and Mercer [42] developed TOPS, which found immediate signal assignments quicker than FAN using *dominators*. A *dominator* is a signal through which the fault effect must pass in order to be detected at a particular primary output. In Figure 2.4, l and n are absolute dominators of a, while k and n are absolute dominators of b, and g, k, and n are absolute dominators of c. Signals k and n are absolute dominators of p and m and n are absolute dominators of q. For absolute dominators of a fault, we must set the inputs to those absolute dominators that are not in the fault effect cone to non-controlling values to propagate the fault effect through the absolute dominator. These are *mandatory assignments*, meaning that they can be determined by topological circuit analysis rather than by search. This would sometimes cause faults to be proven redundant without any search. During search, whenever any dominators of a fault become constant values 0 or 1, fault propagation has been cut off, and the algorithm can back-up immediately. Socrates [68, 69, 70] also used the concept of *dominators*, as explained in Section 2.2.1.2.

2.4 Summary

In this chapter we reviewed the prior work done in the field of redundancy identification in three categories: ATPG techniques, testability analysis and fault-independent techniques. ATPG techniques identify all redundant faults with high cost in terms of CPU time. Conversely, fault-independent techniques identify a subset of all redundant faults with far less cost. In general, redundancies do not cause change in the circuit function, but increase the chip area and the delay. The following chapters will discuss the transitive closure method in detail with various implementation done in this thesis to improve it.

Chapter 3

Boolean Satisfiability Equations for Controllability and Observability and other Definitions

This chapter shows how the implication graph is derived for a given circuit using the knowledge of Boolean Satisfiability equations [3, 26, 27, 35, 47, 49].

3.1 Introduction

The Boolean false function is used for developing Boolean satisfiability relationships [15, 26, 49]. Lin [49] implemented only the pair-wise implications as direct implications and other higher order implications were included if some nodes in a higher-order term were true and a direct implication could be implemented. The higher order terms were implemented as partial implications using an anding node [26, 35]. The limitation was that not all of the anding nodes required to express a Boolean gate into the implication graph were implemented [55], as is explained in detail in Chapter 4. The implications drawn from these Boolean equations are then used to compute the transitive closure. The following subsections explain how the relationships are developed using an example of a two-input Boolean AND gate [26, 27, 55].

3.2 Controllability relationships for an AND gate

A Boolean controllability variable, defined later in this chapter, is assigned to every line in the circuit. The Boolean false function for the controllability relationships of a Boolean gate in general is given by the XOR operation of the gate

Table 3.1: Boolean false function for two-input Boolean gates.

Boolean gate	Boolean false function
OR	$(a+b) \oplus c = 0$
NOR	$\overline{(a+b)} \oplus c = 0$
NAND	$\overline{ab} \oplus c = 0$
NOT	$\overline{a} \oplus b = 0$

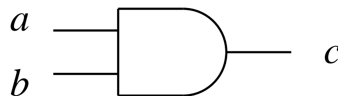

Figure 3.1: A Boolean AND gate.

function (with respect to its inputs) and the gate output signal [15]:

$$gate\ function \oplus output = 0 \tag{3.1}$$

Here, $\oplus$ is the symbol for the XOR operation. Thus, for a two-input AND gate with inputs a and b and output c, shown in Figure 3.1, the Boolean false function [26, 27] from Equation 3.1 will be,

$$ab \oplus c = 0 \tag{3.2}$$

The Boolean false functions for other Boolean gates are shown in Table 3.1, where a and b are inputs and c is the output of the gate, except for the NOT gate where a and b are input and output signals, respectively. The Boolean false function for gates with more than two inputs can be derived in the similar fashion [55]. Equation 3.2 can be expanded as follows:

$$\overline{a}c + \overline{b}c + ab\overline{c} = 0 \tag{3.3}$$

For Equation 3.3 to hold, all three terms on the left hand side 0. The term $\overline{a}c$ becomes 0 only when at least one of the following conditions is satisfied:

1. $a = 1$

2. $\overline{a}\ and\ \overline{c} = 1$

3. $\overline{c} = 1$

4. $c = 1$ *and* $a = 1$

The condition 2 is equivalent to the binary relation $\overline{a} \Rightarrow \overline{c}$, where $\Rightarrow$ denotes the logical implication. Condition 4 represent the binary relation $c \Rightarrow a$. Similarly, the term $\overline{b}c$ is equivalent to the relations $\overline{b} \Rightarrow \overline{c}$ and $c \Rightarrow b$ [26]. These binary relationships can be represented in the form of a directed graph, known as the *implication graph.* Both true and false values of a signal are represented in the implication graph as *nodes*. The logical implications are shown as *edges* [26]. The binary relationships obtained from the pair-wise terms of Equation 3.3 can be implemented in the implication graph as *direct implications.* Four direct implications are obtained for the AND gate:

- The binary relationship: $\overline{a} \Rightarrow \overline{c}$, gives a direct implication: node $\overline{a}$ implies node $\overline{c}$.

- The binary relationship: $\overline{b} \Rightarrow \overline{c}$, gives a direct implication: node $\overline{b}$ implies node $\overline{c}$.

- The binary relationship: $c \Rightarrow a$, gives a direct implication: node c implies node a.

- The binary relationship: $c \Rightarrow b$, gives a direct implication: node c implies node b.

These pair-wise implications are shown in Figure 3.2. The third term in Equation 3.3, $ab\overline{c}$, is a higher-order term, which cannot be implemented as a direct implication in the implication graph. Anding nodes [26, 27, 35] can be used to implement these higher-order terms in the implication graph. We require m anding nodes to represent a m literal higher-order term. A higher-order term implies a condition that $m-1$ literals must be true simultaneously to imply the mth literal. Each anding node has $m-1$ incoming edges and one outgoing edge. The incoming edges for the anding node are known as *partial implications.* For example,

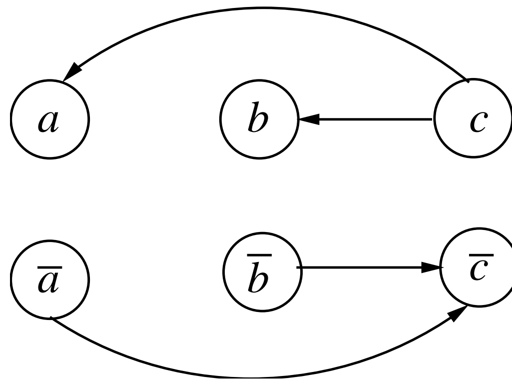

Figure 3.2: Direct implications for the two-input AND gate shown in the Figure 3.1.

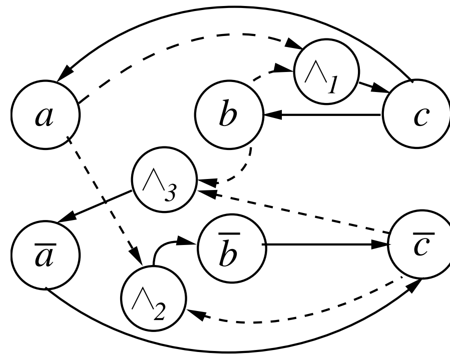

Figure 3.3: Partial implications along with direct implications for the two-input AND gate shown in the Figure 3.1.

consider the implication graph of the two-input Boolean AND gate of Figure 3.1 with anding nodes in Figure 3.3. $\bigwedge$ is the symbol used for the anding node in Figure 3.3. As the higher-order term has three literals, three anding nodes are required in the implication graph and each anding node has two incoming edges of partial implications. The three anding nodes are:

- anding node 1:
  - Inputs: a and b
  - Output: c

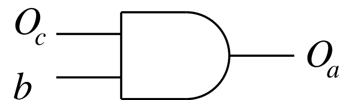

Figure 3.4: Observability AND gate for observing signal a of AND gate shown in Figure 3.1.

Meaning, if nodes a and b are true at the same time then node c is also true.

- anding node 2:
  - Inputs: a and $\overline{c}$
  - Output: $\overline{b}$

  Meaning, if nodes a and $\overline{c}$ are true at the same time then node $\overline{b}$ is also true.

- anding node 3:
  - Inputs: b and $\overline{c}$
  - Output: $\overline{a}$

  Meaning, if nodes b and $\overline{c}$ are true at the same time then node $\overline{a}$ is also true.

Here, inputs correspond to those nodes which are inputs to the anding node and the output corresponds to the node that is implied by the anding node. The partial implications are shown in the implication graph with broken lines.

3.3 Observability relationships

A Boolean observability variable, explained later in this chapter, is assigned to every line in the circuit [3, 26, 27, 55]. The following are the conditions to observe a signal (considered to be the input of some gate) at any of the primary outputs of the circuit:

- All of the other inputs of that gate should hold non-controlling values.
- The output of that gate must be observable at some primary output.

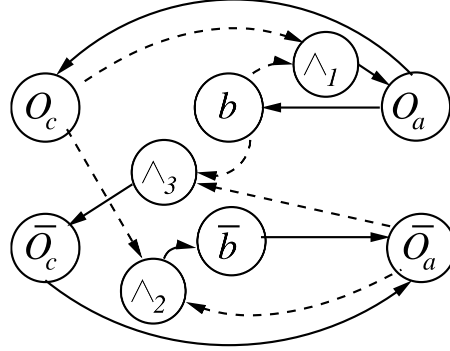

Figure 3.5: Implication graph for observing signal a of Boolean AND gate of Figure 3.1.

Thus, for observing signal a of the AND gate shown in Figure 3.1, signal b should hold logic value 1 and signal c must be observable. These two conditions can be formulated as an added Boolean AND gate, whose two inputs are binary signals O_c and b and the output is O_a, as shown in Figure 3.4. The implication graph for this AND gate can be developed in a similar way as discussed in Section 3.2. By just replacing a with O_c and c with O_a in Equation 3.2, we get following Equation:

$$bO_c \oplus O_a = 0 \tag{3.4}$$

The implication graph, for Equation 3.4, is shown in Figure 3.5. A similar relationship can be established via another observability AND gate for the second input signal b of the AND gate of Figure 3.1. The Boolean false functions for the other two-input Boolean gates can be derived for observability and are given in Table 3.2, where a and b are input signals and c is the output signal for all of the gates except for the NOT gate, where a and b are input and output signals, respectively. Similarly, the Boolean false function can be constructed for gates with more than two inputs.

Table 3.2: Boolean false function for observability relationships for two-input Boolean gates.

Boolean gate	Observability Boolean false function	
	For input a	For input b
OR	$\bar{b}O_c \oplus O_a = 0$	$\bar{a}O_c \oplus O_b = 0$
NOR	$\bar{b}O_c \oplus O_a = 0$	$\bar{a}O_c \oplus O_b = 0$
NAND	$bO_c \oplus O_a = 0$	$a\,O_c \oplus O_b = 0$
NOT	$O_a \oplus O_b = 0$	-

3.4 Conditions for identifying redundant faults

The following are the conditions for identifying redundant faults after transitive closure is calculated [3, 26, 27]:

1. If there exists an edge from node a to $\overline{a}$ then s-a-0 fault is redundant on line a due to uncontrollability, because a is permanently fixed to 0.

2. If there exists an edge from node $\overline{a}$ to a then s-a-1 fault is redundant on line a due to uncontrollability, because a is permanently fixed to 1.

3. If there exists an edge from node a to $\overline{O_a}$ then s-a-0 fault is redundant on line a, due to undrivability since when a is set to 1 to activate the fault, it becomes unobservable.

4. If there exists an edge from node $\overline{a}$ to $\overline{O_a}$ then s-a-1 fault is redundant on line a due to undrivability, since when a is set to 0 to activate the fault, it becomes unobservable.

5. If there exists an edge from node O_a to $\overline{a}$ then s-a-0 fault is redundant on line a due to undrivability (contrapositive edge of condition 3).

6. If there exists an edge from node O_a to a then s-a-1 fault is redundant on line a due to undrivability (contrapositive edge of condition 4).

7. If there exists an edge from node O_a to $\overline{O_a}$ then both s-a-0 and s-a-1 faults are redundant on line a, since the line is unobservable.

Here, a and $\overline{a}$ are true and false controllability nodes and O_a and $\overline{O_a}$ are observability nodes for signal a. The redundant faults identified from conditions 1 and 2 are classified as *unexcitable redundant faults*. The redundant faults identified from conditions 3, 4, 5 and 6 are classified as *undrivable redundant faults*. The redundant faults identified from condition 7 are classified as *unpropagatable redundant faults* [9, 26, 49]. Conditions 1, 2 and 7 are contrapositive edges of themselves. Also, conditions 5 and 6 are contrapositive edges of conditions 3 and 4, respectively.

3.5 Other definitions

- *Stuck-at fault*: A line l stuck to a logic value v, irrespective of logic values of other signals, is said to be *stuck-at-v* [9, 25].
- *Test vector*: A set of primary input logic values, which activate and propagate a stuck-at fault to primary output, is a *test vector* [9].
- *Testable fault*: If a test vector exists for a stuck-at fault, then that fault is known as a *testable fault* [9].
- *Redundant fault*: If no test vector exists for a stuck-at fault, then that fault is known as an *untestable fault*. An untestable fault in a combinational circuit is known as a *redundant fault* because it does not modify the input-output function of the circuit [9].
- *Fanout stem and branch signals*: A *fanout stem* is a signal that branches to feed into many gates, each of which is called a *fanout branch* [9].
- *Controllability*: *Controllability* for a signal is defined as the difficulty of setting that logic signal to a 0 or 1 [9].
- *Observability*: *Observability* for a signal (stem or a fanout branch) is defined as the difficulty of observing that signal at a primary output [9].

- Dominator set: A dominator set of a signal is the set of signals at any level in the circuit (all at same level) through which any path from the signal must pass in order to reach the primary output. The primary outputs of a circuit form a dominator set for all signals of the circuit [9, 54].

- Absolute dominator: A dominator set with only one signal is known as an absolute dominator [54].

- Implication graph: An Implication graph G is a pair (V, E) where V is a set of elements called vertices and $E \subseteq V \times V$ is a set of ordered pairs called directed edges. The cardinality of V is denoted by n and cardinality of E by e. The size of graph G is $n + e$. Given an edge (v, w), v is the tail and w is the head of the edge. A subgraph of a graph $G = (V, E)$ is a graph $S = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$, built from a set of relationships between V [26, 32].

- Transitive closure: The transitive closure of a graph $G = (V, E)$, where V represents vertices and E represents the edges connecting the vertices, is a graph $G_+ = (V, E_+)$ such that E_+ contains an edge (v, w) iff G contains a path $v \rightarrow w$ [21, 26, 32].

- Unexcitable redundant fault: A logic value $\overline{v}$ is required to excite the s-a-v fault on any line l in the circuit. But if the $\overline{v}$ logic value cannot be obtained on line l then the s-a-v fault on the line l is said to be unexcitable redundant. Here, v can be logic 0 or 1 [9, 27].

- Unpropagatable redundant fault: If line l is unobservable irrespective of its value then both faults on line l are said to be unpropagatable redundant [9, 27].

- Undrivable redundant fault: If a logic value v on any line l cannot be driven to the output, then fault s-a-$\overline{v}$ is said to be undrivable redundant on line l. If observability of line l forces a logic value v on line l then the s-a-v fault is said to be undrivable redundant on line l [9, 27].

- Shannon's expansion theorem: Any function, $f(x_1, .., x_k, .., x_n)$ can be expanded around any of its variables, using Shannon's expansion theorem [9]. For example, expansion of the function f around x_k can be done, as shown below:

$$f(x_1, .., x_k, .., x_n) = x_k f(x_1, .., 1, .., x_n) + \overline{x_k} f(x_1, .., 0, ..x_n) \tag{3.5}$$

- Boolean difference: The observability status of any line x_k in the circuit can be determined by taking the Boolean difference of the primary output function, $f(x_1, .., x_k, .., x_n)$, with respect to x_k, as shown below:

$$\text{Boolean difference} = f(x_1, .., 1, .., x_n) \oplus f(x_1, .., 0, .., x_n) \tag{3.6}$$

  Here, $\oplus$ is the symbol for XOR function. If the Boolean difference is 0, then the line x_k is unobservable and if it is non-zero then the line x_k is observable [9, 47].

- Fault equivalence: Two faults of a logic circuit are called equivalent iff they transform the circuit such that the two faulty circuits have the same output function. Equivalent faults are also called indistinguishable and have exactly the same set of tests [9].

- Equivalence checking: It is a technique to check whether the two given circuits, which may be two different levels of implementation, are functionally equivalent or different [26].

3.6 Summary

In this chapter, we discussed how the implication graph is developed for all the gates in a given digital circuit, using Boolean false function. We also showed how, from the Boolean false function, the pair-wise relationships can be converted into direct implications and higher-order relationships can be converted into partial implications using anding nodes for both controllability and observability. Then

we discussed the conditions under which we identify redundancies from the transitive closure graph. In the last part of the chapter we discussed several basic definitions used in this thesis. We will discuss the original contributions of this thesis starting from the next chapter.

Chapter 4
Direct and Partial Implications and Transitive Closure

In this chapter, we describe the previously unimplemented anding nodes and direct edges [26, 27], discuss how they are implemented and what are the improvements in the redundancy identification process. The results for this implementation on the ISCAS benchmark circuits are given in Chapter 7.

4.1 Controllability edges

As shown in Figure 3.3, a two-input Boolean AND gate requires three anding nodes. In general, an n-input Boolean gate requires $n+1$ anding nodes for a complete description of the controllability relationships, as explained in detail in Chapter 8. A limitation of the previous method by Gaur *et al.* [26] was that they did not implement all of the controllability anding nodes for a n-input Boolean gate. The anding node marked $\bigwedge_1$, in Figure 3.3, was the only one implemented. In general, only one anding node was implemented for an n-input Boolean gate, instead of $n+1$ required for the complete description of controllability relationships of the inputs and output of a Boolean gate. This method of redundancy identification is based on the implication graph and transitive closure calculated from the implication graph [26, 27]. As a result of the incomplete description of the given digital circuit into an implication graph, many redundancies were not identified. If the output of the Boolean AND gate shown in Figure 4.1 fans out into multiple branches, the controllabilities of fanout stem and branches are inter-related, as explained in Section 4.3.

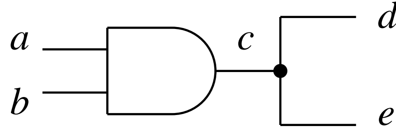

Figure 4.1: A Boolean AND gate with multiple fanouts.

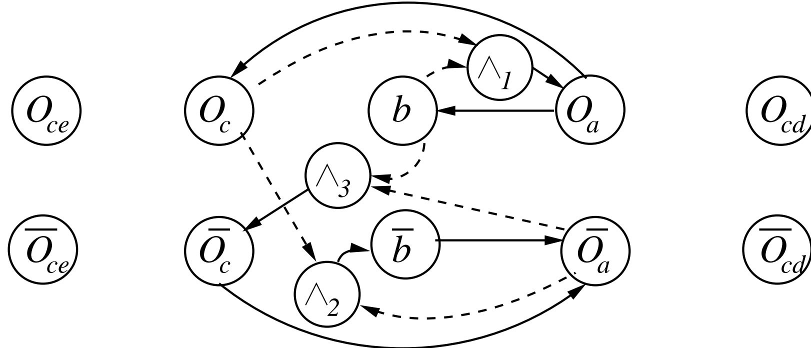

Figure 4.2: Implication graph for signal a of Boolean AND gate of Figure 4.1.

After identifying this limitation of the previous work, we implemented all of the remaining n unimplemented controllability anding nodes for the n-input Boolean gates. For instance, we implemented the anding nodes marked $\bigwedge_2$ and $\bigwedge_3$, shown in Figure 3.3. Thus, in general, now we have all of the $n + 1$ anding nodes required for the complete description of a n-input Boolean gate in terms of the controllability relationship of its inputs and output.

4.2 Observability edges

As shown in Figure 3.5, the observation of the input signal a of a two-input Boolean AND gate requires three anding nodes in the implication graph. In general, observing an input signal of an n-input Boolean gate requires $n + 1$ anding nodes in the implication graph. Thus, all n inputs of a Boolean gate requires $n(n+1)$ anding nodes. The limitation of the previous technique by Gaur *et al.* [26] was that they did not implement any of these $n^2 + n$ anding nodes.

If a gate, in the given digital circuit, fans out into multiple branches, then the stem and branch observabilities should be represented with different nodes. Consider the two-input Boolean AND gate shown in Figure 3.1. Let us suppose that the output signal c fans out into two branches to feed some other Boolean gates d and e as shown in the Figure 4.1. Then corresponding to Figure 4.1, the implication graph will change from Figure 3.5 to that shown in Figure 4.2. The only change in the two figures is an increase in the number of observability nodes. Here the nodes O_{cd} and $\overline{O_{cd}}$ will have implications to and from controllability and observability nodes of gate d and similarly nodes O_{ce} and $\overline{O_{ce}}$ will have implications to and from controllability and observability nodes of gate e. If this type of case was encountered in the previous work by Gaur *et al.* [26], then the direct implications from node O_a to O_c and $\overline{O_c}$ to $\overline{O_a}$ were not implemented, which according to Figure 3.5 should be present. These implications should be present even though the gate fans out into multiple branches. Thus, the implication graph showing the observability relationships of the n-input Boolean gate was incomplete. As explained earlier, if the implication graph is incompletely described then many redundancies may remain unidentified.

After identifying all of these limitations of the previous method we implemented all of the unimplemented but required anding nodes and direct implications [55]. We implemented the anding nodes marked $\bigwedge_1$, $\bigwedge_2$ and $\bigwedge_3$ shown in Figures 3.5 and 4.2. We also implemented the direct implications from nodes O_a and $\overline{O_c}$ to nodes O_c and $\overline{O_a}$, respectively, as shown in the Figure 4.2 where the output of a gate fans out into multiple branches. Thus, in general we implemented all $n^2 + n$ anding nodes for observability relationships and other direct implications in the implication graph.

4.3 Controllability edges for fanout stem and its branches

For a fanout in any given circuit, the controllabilities of the fanout branches are directly related to the controllability of the stem. Consider the fanout shown in the

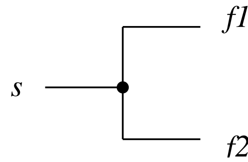

Figure 4.3: A fanout stem.

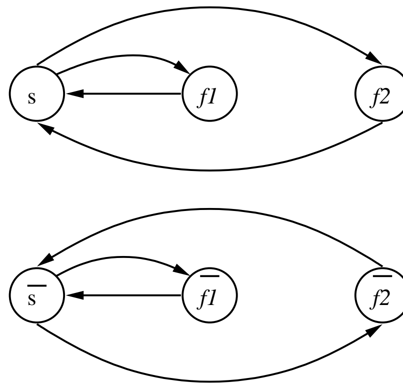

Figure 4.4: An implication graph for controllability of the fanout stem and its branches.

Figure 4.3. Figure 4.4 shows the implication graph for the controllability relationship of the fanout branches. Thus, in the implication graph, the controllabilities of a fanout stem and its branches are represented by the same node [26, 55]. The implication graph of the circuit as shown in Figure 4.1 will have nodes c and $\overline{c}$ to represent the controllability of the output of the Boolean AND gate and input to gates d and e, which are fed by c.

4.4 Example circuits

Consider the example circuit shown in Figure 4.5. Here the redundant faults d s-a-1 and $b1$ s-a-1 were identified by the previous method [26], but c s-a-1 was

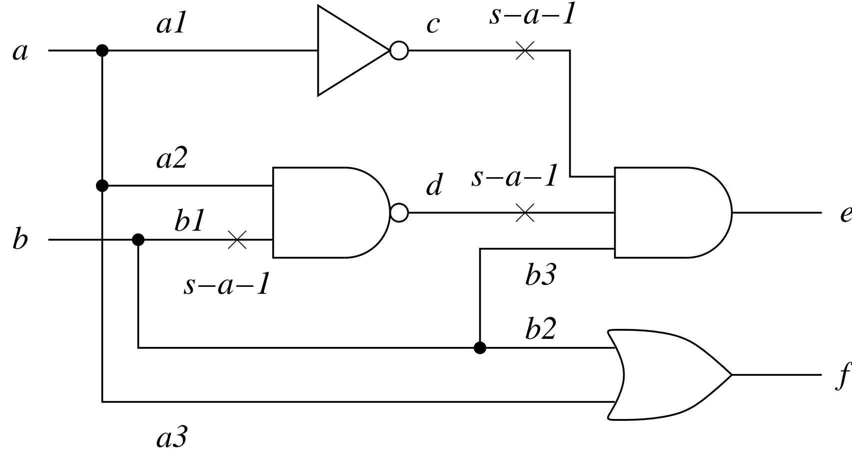

Figure 4.5: Example circuit showing the limitation of the previous work [26].

not identified. The reason is the unimplemented controllability anding node for Boolean NAND gate d ($d \bigwedge b$ implies $\overline{a}$, marked as $\bigwedge_1$ in Figure 4.6) as explained in Section 4.1. We could identify this redundant fault from the traversal: observing signal c implies that d and b are true, which in turn through anding node $\bigwedge_1$ implies $\overline{a}$. Node $\overline{a}$ further implies c, as shown in Figure 4.6. Thus, we end up with an implication from O_c to c after the transitive closure is calculated and that identifies c s-a-1 as undrivable redundant. Similarly, we have a path from node O_d to d which means that d s-a-1 is undrivable redundant. An edge from O_{b1} to $\overline{O_{b1}}$ implies that $b1$ is unpropagatable redundant. The circuit shown in Figure 4.7 has $b1$ s-a-1, d s-a-0 and $a1$ s-a-0 redundant faults. Out of these three, Gaur *et al.* [26] could identify only the first two [55]. Figure 4.8 has three redundant faults, a s-a-0, a s-a-1 and c s-a-0. Out of these Gaur *et al.* [26] and Mehta *et al.* [55] could identify only the c s-a-0 redundant fault. In a later chapter, we give new stem unobservability theorems [54] that will identify all the three redundant faults as explained in Chapter 6.

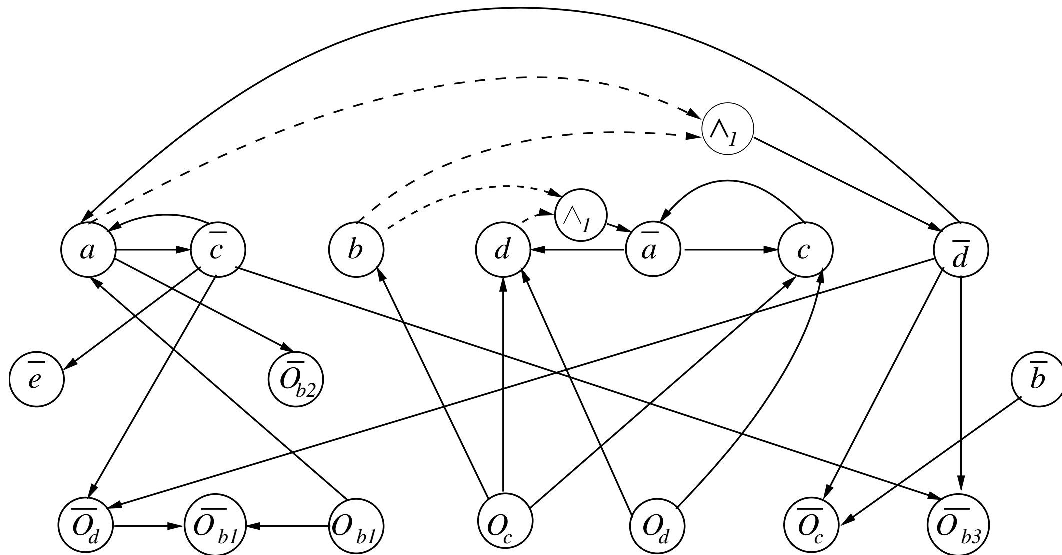

Figure 4.6: Implication graph for identifying redundant faults in the circuit of Figure 4.5.

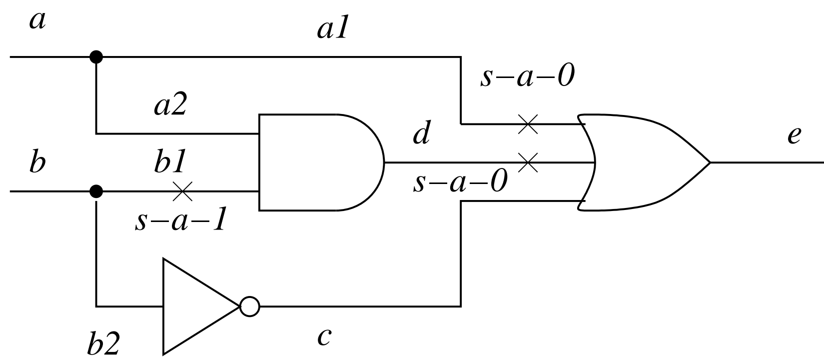

Figure 4.7: An example circuit showing a limitation of the previous work [26].

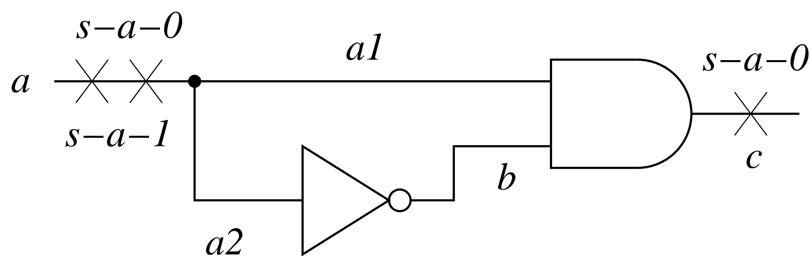

Figure 4.8: Another example circuit showing a limitation of the previous work [26].

4.5 Summary

In this chapter, we presented the limitations of the previous work [26] and showed that after implementing all of the anding nodes and direct implications for controllability and observability we could identify more redundant faults with example circuits. Our research, the fixed-value theorem and the stem unobservability theorems, will be discussed in Chapter 5 and Chapter 6, respectively. The experimental results with these improvements on ISCAS'85 and ISCAS'89 benchmark circuits are given in Chapter 7.

Chapter 5

Fixed-Value Theorem and Contrapositive Rule Implementation

In this chapter, we will discuss the effect of the fixed-value nodes on the transitive closure [55]. We will then discuss the contrapositive rule [69, 80] and its implementation. These two enhancements are new and described here for the first time.

5.1 Fixed-value theorem

We have found some limitations of the previously described transitive closure technique [26] in identifying certain redundancies. We consider:

- A primary output AND or NOR gate with an unexcitable s-a-1 redundant fault on its output line (see g s-a-1 in Figures 5.1 and 5.3).
- A primary output OR or NAND gate with an unexcitable s-a-0 redundant fault on its output line.

The above redundant faults are classified as unexcitable redundant, due to the following reasons:

- The primary output cannot have an unpropagatable redundant fault as they do not require propagation.
- The primary output cannot have an undrivable redundant fault as they are always observable, *i.e.*, they do not require propagation as stated in the above condition.

We derived a technique called *node fixation* to identify these types of redundant faults. Controllability and observability are represented as nodes in the graph and, as transitive closure shows, some nodes can assume fixed values. For example, in Figure 5.1 signals e and f are found to be fixed to 1. This leads to the fixation $g = 1$, which is not found by the conventional transitive closure. We developed this concept for identifying redundancies at the primary output side of the circuit, but then used it to enhance the transitive closure graph with respect to all fixed nodes. We prove the following theorem to implement node fixation concept.

Theorem 5.1.1 *If a node is fixed to a true or a complement value then there exist unconditional edges from all other nodes in the graph to the fixed node.*

Proof: Suppose there exists an edge from node a to $\overline{a}$ in the graph (where a and $\overline{a}$ represent true and complement values of signal line a in the circuit), then irrespective of the status of any other nodes in the graph, $\overline{a}$ will be unconditionally true. In other words, there exists an edge from all the nodes in the graph to the node $\overline{a}$. The status of signal line a will remain 0 even though node a is traversed in the graph traversal, because it will eventually lead to $\overline{a}$. ■

With Theorem 5.1.1, we can implement more valid edges in the transitive closure graph, which in turn identifies more redundant faults. The implementation of this theorem is explained with an example circuit and an implication graph in the following section.

5.2 Example circuits justifying the fixed-value theorem implementation

The example circuit shown in Figure 5.1 has 7 redundant faults. Previous work [26] could identify s-a-1 faults on lines e and f. They could not identify the following redundant faults:

- s-a-1 on line g, because of insufficient edges to identify the s-a-1 redundant fault on the primary output for the AND gate, as explained in Section 5.1.

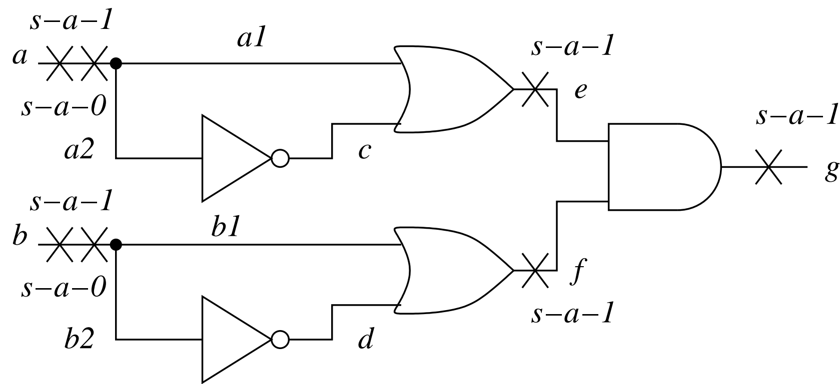

Figure 5.1: Example circuit showing the implementation of node fixation.

- s-a-0 and s-a-1 on lines a and b, due to a lack of stem unobservability relationships.

Without Theorem 5.1.1, we could not identify s-a-1 on line g as a redundant fault. The faults e and f s-a-1 are identified to be redundant with edges from nodes $\overline{e}$ and $\overline{f}$ in the transitive closure graph to nodes e and f, respectively. According to the classification given in Section 3.4, these faults are classified as unexcitable redundant faults. As these nodes are fixed to logic value 1, according to Theorem 5.1.1, we implemented edges from all other nodes in the graph to nodes e and f. Now consider the implication graph shown in Figure 5.2, where the fixed-value theorem allows us to implement edges from node $\overline{g}$ to nodes e and f. When the traversal is started from node $\overline{g}$ in the graph, nodes e and f both are true at the same time and they, in turn, through the anding node imply node g. Thus, we have an edge from node $\overline{g}$ to g, which identifies the s-a-1 fault on line g as redundant. The other four faults, a and b both s-a-0 and s-a-1, are not yet identified due to the absence of any false stem observability edges. These faults are identified as redundant [54] with the technique introduced in Chapter 6.

As another example, in Figure 5.1, replace:

- The AND gate with a NOR gate.

- Both OR gates with AND gates.

This is shown in Figure 5.3. Using similar arguments and Theorem 5.1.1, we

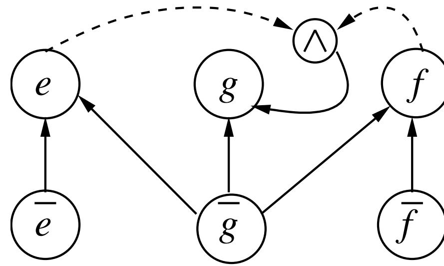

Figure 5.2: Implication graph showing how the primary output redundant fault was identified redundant using Theorem 5.1.1 for example circuit of Figure 5.1.

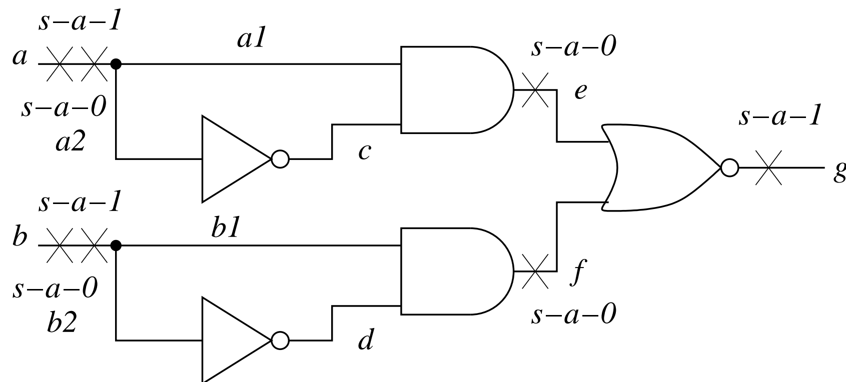

Figure 5.3: Example circuit showing the consistency of limitations (of previous technique [26]) identified in Section 5.1.

can identify g s-a-1 as a redundant fault. This example is consistent with the limitations identified (of the previous method [26]) in Section 5.1.

5.3 Contrapositive rule implementation

According to the contrapositive rule [69, 80], if there exists an edge from a node p to q in the graph, then there must exist another edge from node $\overline{q}$ to node $\overline{p}$. We will check this rule at three different levels of implementation in our method of redundancy identification:

1. When the implication graph for a gate is constructed. The Boolean difference equation, developed in Chapter 3, makes sure that the contrapositive rule is followed in building the implication graph for a gate. For example,

consider Figure 3.2 or 3.3. Here all direct implications obey the contrapositive rule.

2. When transitive closure is calculated. After a transitive closure is calculated there may be an increase in the number of edges in the graph. Here we need to check for the contrapositive rule to find if it is followed and if not then the corresponding edge should be added.

   We identify all of the redundant faults from the transitive closure calculations with the conditions listed in Section 3.4. These conditions also obey the contrapositive rule. We identify s-a-1 on line d of the example circuit shown in Figure 4.5 with condition number 6 of Section 3.4 (O_d implies d as shown in Figure 4.6). Also note that we have an edge from $\overline{d}$ to $\overline{O_d}$ as shown in Figure 4.6. Thus, the contrapositive rule is being obeyed after the transitive closure calculation. But this may not be true in all the cases. For example, consider the s-a-1 redundant fault on line c, of the same example circuit. We have an edge from node O_c to c, but we do not have an edge from $\overline{c}$ to $\overline{O_c}$. The reason is the unavailability of edges that could fix the status of node b. Although, we have implications from node b (with a through the anding node) and $\overline{b}$ to node $\overline{O_c}$, we do not have any edge to node b or $\overline{b}$. Thus, here we need to add the contrapositive edge from $\overline{c}$ to $\overline{O_c}$.

3. When the node fixation concept is implemented. After the edges are added due to the node fixation concept, we need to check for the contrapositive rule and if it is not followed we add those edges.

5.4 Limitation

The limitation of our method still lies in traversing the graph when a decision has to be taken as explained in Section 5.1, when the traversal has to start from the complement of the output of an AND or NOR gate, which is not fixed to some

Figure 5.4: Example circuit showing the decision taking inability of our algorithm.

constant value. Consider the example circuit shown in Figure 5.4. We identify a s-a-1, d s-a-1 and $c1$ s-a-1 as redundant. But we do not identify e s-a-1 and c s-a-0 and s-a-1 as redundant, even though they are. Observing line e requires f and g to be 1 (for propagation) and we can not traverse back from f and g because any of $b2$ or $c1$ and $b3$ or d can be 0. So, as our algorithm can not take a decision, we fail to ascertain b to be 0 and then go forward and imply that O_e implies e. The unobservability of stem c can not be ascertained by our algorithm and the reason for that will be given in the next chapter. In the combinational benchmark circuit c432, we identify 1 out of 4 redundant faults. The analysis of this circuit revealed that we could not identify three redundant faults because of the decision taking inability of our algorithm (similar to the problem with unidentified redundant fault e s-a-1 of Figure 5.4).

5.5 Summary

In this chapter, we introduced the fixed-value theorem. We discussed its motivation and explained how it is implemented using example circuits. We also discussed the contrapositive rule and showed where it is required to be checked with the example circuit of Chapter 4. In the following chapters we will discuss stem unobservability theorems, results on benchmark circuits, and the complexity analysis of our technique.

Chapter 6

Fanout Stem Unobservability Theorems

In this chapter, we will describe the solution we propose to solve the stem unobservability problem. We propose two theorems, one with respect to a fixed-valued dominator set and the other with respect to an unobservable dominator set. We will also discuss several example circuits for a better understanding of the theorems.

6.1 Fanout stem unobservability and its effect on identifying other redundancies

As explained in Section 4.2, there are no edges to relate the observability of a fanout stem to its branch observabilities. As the stem observability edges are absent, many redundancies from the fanout stem towards primary inputs will not be identified. We not only fail to identify any redundant faults on the fanout stem, but also fail to identify other redundant faults caused by the unobservability of the fanout stem. This is because a fanout stem may or may not be observable if no, some or all fanout branches are unobservable. Several cases are illustrated with example circuits:

- The fanout stem may or may not be observable, if all the fanout branches are observable.
  - The first case is illustrated by stem a and its branches $a1$ and $a2$ shown in Figure 4.7. There, the stem a and its branches $a1$ and $a2$ are observable.

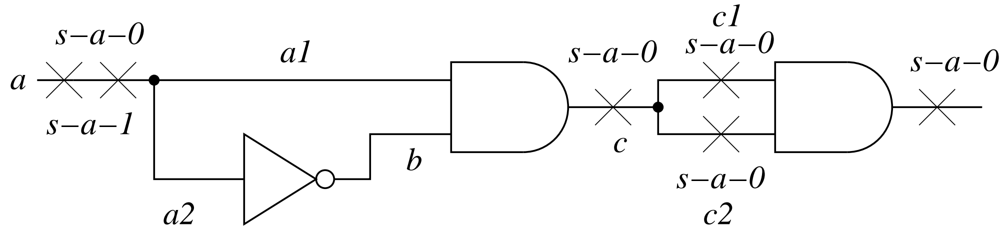

Figure 6.1: Example circuit illustrating the cases of stem observability and unobservability (all faults shown are redundant).

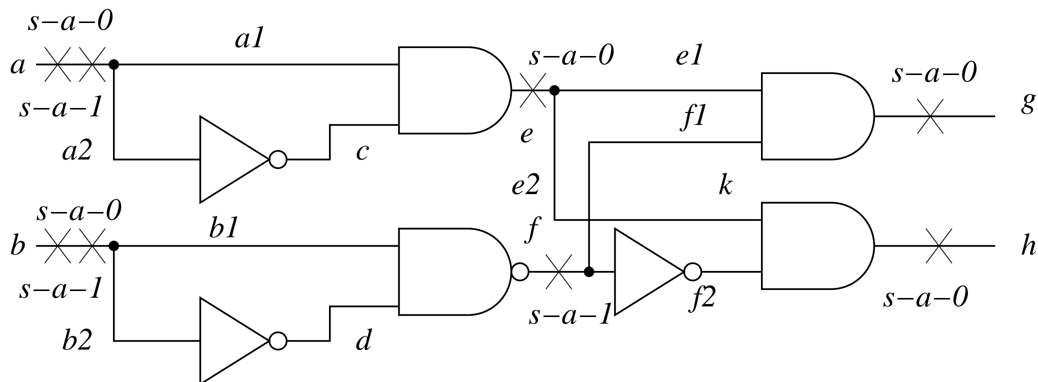

Figure 6.2: Example circuit illustrating the cases of stem observability and unobservability.

- The second case is illustrated by stem a, of Figure 6.1. Here, $a1$ and $a2$ are observable but a is unobservable.

- A fanout stem may or may not be observable if some or all of its fanout branches are unobservable.

  - The first case is illustrated by the fanout stem e and its branches $e1$ and $e2$ in Figure 6.2. Here, $e1$ is observable and $e2$ is unobservable, but e is observable.
  - The second case is illustrated by the fanout stem e and its branches $e1$ and $e2$ in Figure 6.3. Here, $e1$ is observable and $e2$ is unobservable, but e is unobservable.
  - The third case is illustrated by the fanout stem c and its branches $c1$ and $c2$ in Figure 6.1. Here, $c1$ and $c2$ are both unobservable, but c is

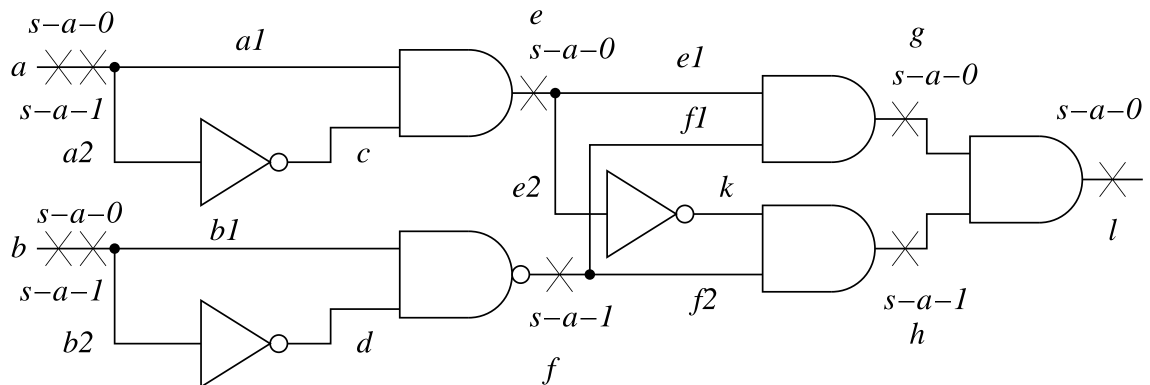

Figure 6.3: Example circuit illustrating the cases of stem observability and unobservability.

observable.

- The fourth case is illustrated by the fanout stem f and its branches $f1$ and $f2$ in Figure 6.2. Here, $f1$ and $f2$ are both unobservable and f is also unobservable.

6.2 Theorems to identify unobservable fanout stems

Theorem 6.2.1 *A fanout stem is unobservable, if each signal in its dominator set assumes a constant value and:*

- *the fanout stem does not hold a constant value, or*
- *the fanout stem holds a constant value and, in spite of any local change in the stem signal, the dominator set values do not change.*

A *local change* of a signal only affects the portion of the circuit between that signal and POs.

Proof: From the definition of the dominator set, if the stem under consideration is unobservable at the dominator set then it will be unobservable at primary outputs as well. So:

$$f_i(X, x_p) = c \tag{6.1}$$

where X is the set of primary inputs, x_p is the fanout stem under consideration, f_i is the dominator set function, that is a function of X and x_p and $i \in [1, k]$, where k is the cardinality of the dominator set and c is a constant, which can assume a logic 0 or 1 value. Expanding Equation 6.1 using Shannon's expansion theorem [9], we get:

$$x_p f_i(X, 1) + \overline{x_p} f_i(X, 0) = c \tag{6.2}$$

We consider two cases:

- $x_p \neq$ constant, then Equation 6.2 implies:

  $$f_i(X, 1) = f_i(X, 0) = c \tag{6.3}$$

  Taking the Boolean difference of Equation 6.1, with respect to x_p, we get:

  $$\frac{\partial f_i(X, x_p)}{\partial x_p} = f_i(X, 1) \oplus f_i(X, 0) \tag{6.4}$$

  From Equation 6.3:

  $$\frac{\partial f_i(X, x_p)}{\partial x_p} = c \oplus c = 0 \tag{6.5}$$

  Thus, x_p is unobservable.

- $x_p =$ constant. Substituting $x_p = 1$ in Equation 6.2, we get:

  $$f_i(X, 1) = c \textit{ and } f_i(X, 0) = \textit{unknown} \tag{6.6}$$

  and substituting $x_p = 0$ in Equation 6.2, we get:

  $$f_i(X, 1) = \textit{unknown and } f_i(X, 0) = c \tag{6.7}$$

  The Boolean difference of Equation 6.1, with respect to x_p gives:

  $$\frac{\partial f_i(X, x_p)}{\partial x_p} = f_i(X, 1) \oplus f_i(X, 0) \neq 0 \tag{6.8}$$

  In the special case, when $f_i(X, 1) = f_i(X, 0) = c$, *i.e.*, $f_i(X, x_p)$ remains unchanged when x_p assumes two different values, Equation 6.8 evaluates to 0 and the stem x_p becomes unobservable. ■

Theorem 6.2.2 *A fanout stem is unobservable, if each signal in its dominator set is unobservable and:*

- *the stem does not hold a constant value, or*
- *the stem holds a constant value and, in spite of any local change in the stem signal, the unobservable status of the dominator set remains unchanged.*

Proof: From the definition of the dominator set, it is sufficient to show that the stem under consideration is unobservable at the dominator set. Consider the primary output functions:

$$f_i(X, x_p, d_1, ..., d_n) \tag{6.9}$$

where, X is a set of primary inputs, x_p is the fanout stem under consideration, $d_1, ..., d_n$ are the dominator set signals for x_p, $i \in [1, k]$, where k is the cardinality of the PO set. Let the dominator function be $d_j(X, x_p)$, $j \in [1, n]$. Since dominator signal d_1 is unobservable:

$$\frac{\partial f_i}{\partial d_1} = f_i(X, x_p, 1, d_2, ..., d_n) \oplus f_i(X, x_p, 0, d_2, ..., d_n) = 0 \tag{6.10}$$

Similarly, $d_2, ..., d_n$ are unobservable. Thus, we get:

$$\frac{\partial f_i}{\partial d_n} = f_i(X, x_p) + f_i(X, \overline{x_p}) = 0 \tag{6.11}$$

Now the Boolean difference of the above equation with respect to x_p gives:

$$\frac{\partial f_i}{\partial x_p} = x_p f_i(X, 1) + \overline{x_p} f_i(X, 0) = 0 \tag{6.12}$$

We consider two cases:

- If $x_p \neq$ constant, then:

$$f_i(X, 1) = f_i(X, 0) = 0 \tag{6.13}$$

  Thus, x_p is unobservable.

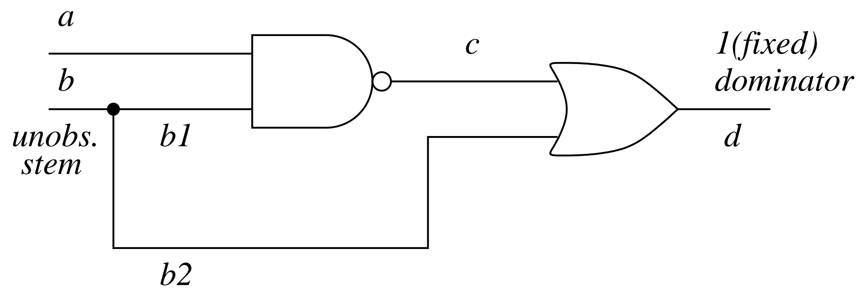

Figure 6.4: Example circuit justifying the implementation of Theorem 6.2.1.

- If x_p = constant. For $x_p = 1$, from Equation 6.12 we get, $f_i(X, 1) = 0$ and $f_i(X, 0)$ = unknown and for $x_p = 0$, we get, $f_i(X, 0) = 0$ and $f_i(X, 1)$ = unknown. Taking the Boolean difference of Equation 6.9, with respect to x_p, is thus not equal to 0. Thus, x_p is observable. But in a special case when unknown becomes equal to 0, x_p becomes unobservable. ∎

Theorem 6.2.2 of this chapter is consistent with a lemma proposed by Iyer and Abramovici [39] (see Chapter 2 for details). Their lemma is a special case of Theorem 6.2.2 because the theorem does not require the uncontrollability indicator condition required by the lemma.

6.3 Example circuits justifying implementation of stem unobservability theorems

The fanout stem a, shown in Figure 6.3, is not fixed and the dominator e has a constant value of 0. Thus, Theorem 6.2.1 identifies fanout stem a as unobservable. The fanout stem e, shown in Figure 6.3, is fixed to 0 and the dominator l holds a constant value of 0. Thus, Theorem 6.2.1, locally changes the value of e and checks the effect on the dominator, which is found to be unchanged. Thus, e is unobservable. The fanout stem f in the same figure is fixed to 1 and the dominator l holds a constant value of 0. But, changing the value of f does not change the dominator set value. Thus, Theorem 6.2.1 identifies f as an unobservable stem.

The fanout stem b, shown in Figure 6.4, is not fixed and the dominator d is fixed to 1. Thus, Theorem 6.2.1 identifies b as an unobservable fanout stem. The

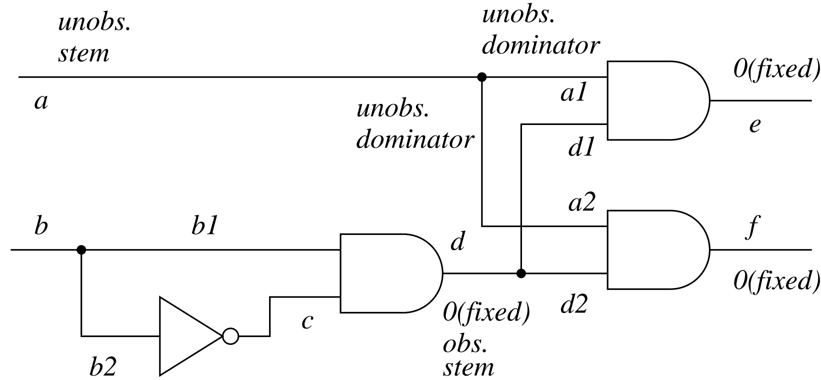

Figure 6.5: Example circuit justifying the implementation of Theorem 6.2.2.

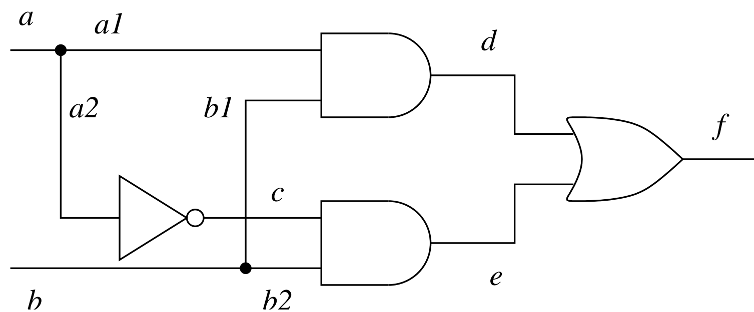

Figure 6.6: Example circuit showing the limitation of our method.

fanout stem c, shown in Figure 6.1, is fixed to 0 and the dominator set signals $\{c1, c2\}$ are unobservable. A change in the value of c changes the observability status of the dominator set. Thus, Theorem 6.2.2 does not identify c as an unobservable stem. The fanout stem a, shown in Figure 6.5, is not fixed and the dominator set signals $\{a1, a2\}$ are unobservable. Thus, Theorem 6.2.2 identifies a as an unobservable fanout stem. After the observability status of a fanout stem is decided, an unobservability edge is implemented in the implication graph.

Let us now consider a case, where our theorems cannot identify the unobservability of the fanout stems [53] as in Figure 6.6. The dominator f, for the fanout stem a, is neither fixed nor unobservable. Thus, none of the theorems can specify the unobservability status of the fanout stem a. This example illustrates a limitation of our method.

6.4 Summary

In this chapter, we studied the problem of fanout stem unobservability. We proved theorems that identify unobservable fanout stems due to:

- a fixed dominator set
- an unobservable dominator set

Using example circuits, we showed how Theorems 6.2.1 and 6.2.2 identify various types of unobservable fanout stems. We also showed a limitation of our method. We will discuss the results on ISCAS benchmark circuits and a complexity analysis of transitive closure for redundancy identification in next chapters.

Chapter 7
Results

In this chapter, we will show the results obtained by our method at different levels of implementation, with respect to the number of redundant faults identified and the CPU time[1] used to identify these redundancies. We will also give a table to show comparisons of results obtained by our method with those obtained by other fault-independent methods TC [3], TC_{AND} [27], FIRE [39] and also with the ATPG tool TRAN [17]. We will classify the redundant faults identified by our technique into unexcitable, unpropagatable and undrivable categories.

7.1 Redundancies after each level of implementation

The program implementation of this thesis was done in three steps. Sections 7.1.1, 7.1.2 and 7.1.3 explain each of them in detail.

7.1.1 Previously unimplemented direct and partial implications

The main aim here was to complete the previous work [26]. We implemented all of the unimplemented anding nodes and direct implications, which according to the Boolean difference formula, as explained in Chapter 4, must be present for controllability and observability relationships. The results obtained after implementing all anding nodes and direct implications are shown in Table 7.1. The first column of this result table shows the ISCAS'85 and ISCAS'89 benchmark

[1]Sparc 5 Sun workstation.

Table 7.1: Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after implementations of Section 7.1.1

Circuit	*TRAN* [17]		TC_{AND} [26]		*Section* 7.1.1		Classi. of Red. col. 6		
	Red. faults	CPU s	Red. faults	CPU s	Red. faults	CPU s	Unexc.	Unpro.	Undri.
c432	4	0.8	0	0.2	1	0.2	0	0	1
c499	8	1.8	0	0.2	0	0.2	0	0	0
c880	0	4.0	0	0.4	0	0.3	0	0	0
c1355	8	11.0	0	0.9	0	0.7	0	0	0
c1908	7	13.0	2	0.9	2	1.0	0	0	2
c2670	115	95.2	25	1.5	59	1.6	3	3	53
c3540	131	24.9	74	6.2	110	2.7	1	7	102
c5315	59	32.3	32	3.4	58	2.9	1	0	57
c6288	34	38.0	31	1.8	34	4.9	17	16	1
c7552	131	308.0	34	5.8	51	8.6	2	0	49
s349	2	0.3	2	0.2	2	0.2	1	0	1
s444	14	0.4	8	0.2	10	0.2	1	1	8
s713	38	3.1	35	0.3	38	0.4	16	19	3
s1238	69	17.4	6	0.6	20	0.7	0	2	18
s1423	14	8.5	8	0.7	12	0.8	0	1	11
s1494	12	3.7	1	0.8	2	1.0	0	0	2
s5378	40	73.0	22	3.0	27	4.1	10	8	9
s9234	452	803.7	135	11.2	233	13.5	29	59	145
s13207	151	806.5	60	13.6	77	39.0	6	6	65

circuits. The second and third columns are the number of redundant faults identified by the ATPG tool TRAN [17] and its CPU time in seconds. TRAN reports 0 aborted faults for all ISCAS'85 and ISCAS'89 benchmark circuits. The fourth and fifth column represents the total number of redundant faults identified by Gaur *et al.* [26] and the CPU time in seconds. The sixth and seventh columns represent the total number of redundant faults identified by our program and the CPU time in seconds. The rest of the columns show the classification of the redundancies identified by our method into (refer to Chapter 3 for definitions):

- *Unexcitable redundant faults.*
- *Unpropagatable redundant faults.*
- *Undrivable redundant faults.*

Table 7.2: Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after implementations of Section 7.1.2

Circuit	*TRAN* [17]		*Section* 7.1.2		Classification of Red. Faults in Column 4		
	Red. faults	CPU s	Red. faults	CPU s	Unexcit.	Unpropa.	Undriv.
c432	4	0.8	1	0.2	0	0	1
c499	8	1.8	0	0.2	0	0	0
c880	0	4.0	0	0.3	0	0	0
c1355	8	11.0	0	0.7	0	0	0
c1908	7	13.0	2	1.0	0	0	2
c2670	115	95.2	59	2.9	3	3	53
c3540	131	24.9	111	5.0	1	8	102
c5315	59	32.3	58	3.4	1	0	57
c6288	34	38.0	34	6.2	17	16	1
c7552	131	308.0	55	9.7	2	4	49
s349	2	0.3	2	0.2	1	0	1
s444	14	0.4	10	0.2	1	1	8
s713	38	3.1	38	0.4	16	19	3
s1238	69	17.4	20	1.7	0	2	18
s1423	14	8.5	12	0.8	0	1	11
s1494	12	3.7	2	4.2	0	0	2
s5378	40	73.0	27	6.4	10	8	9
s9234	452	803.7	233	48.3	29	59	145
s13207	151	806.5	77	69.8	6	6	65

We used ISCAS'85 combinational benchmark circuits directly. For ISCAS'89 sequential benchmark circuits, we made the circuits combinational by making the flip-flops fully controllable and observable. We used the inputs to the flip-flops as pseudo-primary outputs and the outputs of the flip-flops as pseudo-primary inputs.

7.1.2 Fixed-value theorem

The main aim here was to identify fixed nodes in the graph and implement the procedure of the fixed-value theorem, explained in Chapter 5. We implement more valid implications in the graph using Theorem 5.1.1, which in turn identifies more redundancies. Table 7.2 shows the results obtained after the implementation of the node fixation concept. For the explanation of the columns of this result table, refer to Section 7.1.1. Only for the combinational circuits c3540 and c7552, one

Table 7.3: Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after implementations of Section 7.1.3

Circuit	*TRAN* [17]		*Section* 7.1.3		Classification of Red. Faults in Column 4		
	Red. faults	CPU s	Red. faults	CPU s	Unexcit.	Unpropa.	Undriv.
c432	4	0.8	1	0.2	0	0	1
c499	8	1.8	0	1.3	0	0	0
c880	0	4.0	0	0.4	0	0	0
c1355	8	11.0	0	1.9	0	0	0
c1908	7	13.0	2	3.2	0	0	2
c2670	115	95.2	82	4.0	9	10	63
c3540	131	24.9	111	16.2	1	7	102
c5315	59	32.3	58	3.9	1	0	57
c6288	34	38.0	34	7.2	17	16	1
c7552	131	308.0	55	11.5	2	4	49
s349	2	0.3	2	0.2	1	0	1
s444	14	0.4	10	0.3	1	1	8
s713	38	3.1	38	0.6	16	19	3
s1238	69	17.4	20	2.6	0	2	18
s1423	14	8.5	12	1.0	0	1	11
s1494	12	3.7	2	8.8	0	0	2
s5378	40	73.0	27	8.3	10	8	9
s9234	452	803.7	233	106.0	29	59	145
s13207	151	806.5	77	158.8	6	6	65

and four more redundant faults are identified, respectively, all of which are of the unpropagatable type.

7.1.3 Stem unobservability

The main aim here is to implement the theorems proposed in Chapter 6 to identify stem unobservability. Table 7.3 shows the results obtained after implementing stem unobservability theorems. For an explanation of the columns in the result table, refer to Section 7.1.1.

7.1.4 Comparison

Table 7.4, compares the results obtained by our technique, with the ATPG tool TRAN [17] by Chakradhar *et al.* and other fault-independent techniques, FIRE

Table 7.4: Combinationally redundant faults identified in ISCAS '85 and ISCAS '89 benchmark circuits after all the implementation and comparison with other fault-independent methods and ATPG technique.

Circuit	Total faults	Redundant faults identified and run time									
		TRAN [17]		*FIRE* [39]		*TC* [3]		TC_{AND} [27]		Our Algo.	
		Red. fau.	CPU *s* [a]	Red. fau.	CPU *s* [b]	Red. fau.	CPU *s* [a]	Red. fau.	CPU *s* [a]	Red. fau.	CPU *s* [a]
c432	524	4	0.8	-	-	0	0.1	0	0.2	1	0.2
c499	758	8	1.8	-	-	0	0.2	0	0.2	0	1.3
c880	942	0	4.0	-	-	0	0.3	0	0.4	0	0.4
c1355	1574	8	11.0	-	-	0	0.7	0	0.9	0	1.9
c1908	1879	7	13.0	6	1.8	2	0.8	2	0.9	2	3.2
c2670	2747	115	95.2	29	1.5	23	1.3	25	1.5	82	4.0
c3540	3428	131	24.9	93	11.9	54	5.9	74	6.2	111	16.2
c5315	5350	59	32.3	20	2.8	20	3.4	32	3.4	58	3.9
c6288	7744	34	38.0	33	1.3	31	1.0	31	1.8	34	7.2
c7552	7550	131	308.0	30	4.7	32	5.2	34	5.8	55	11.5
s349	350	2	0.3	2	0.2	1	0.1	2	0.2	2	0.2
s444	474	14	0.4	11	0.2	2	0.1	8	0.2	10	0.3
s713	581	38	3.1	32	0.1	29	0.3	35	0.3	38	0.6
s1238	1355	69	17.4	6	1.9	4	0.5	6	0.6	20	2.6
s1423	1515	14	8.5	5	0.3	4	0.6	8	0.7	12	1.0
s1494	1506	12	3.7	1	1.1	1	0.7	1	0.8	2	8.8
s5378	4603	40	73.0	34	3.7	20	2.5	22	3.0	27	8.3
s9234	6927	452	803.7	165	20.6	16	9.8	135	11.2	233	106.0
s13207	9815	151	806.5	55	23.2	9	11.2	60	13.6	77	158.8

[a]Sun sparc5 CPU sec.

[b]Sun sparc2 CPU sec.

by Iyer and Abramovici [39], TC by Agrawal *et al.* [3] and TC_{AND} by Gaur *et al.* [26]. For a better understanding of the above-mentioned techniques, refer to Chapters 2 and 3. This comparison table shows that our technique identifies more redundancies then any other fault-independent techniques and in some circuits, *e.g.*, c6288, s349 and s713, we identify all redundancies as in the case of the ATPG tool.

7.2 Analysis of the benchmark circuit results

Results on ISCAS'85[2] and ISCAS'89[3] benchmark circuits are given in Tables 7.1, 7.2 and 7.3 at each level of implementation. We compare our final results with the fault-independent techniques such as FIRE [39], TC [49], $\mathrm{TC_{AND}}$ [26] and the ATPG technique TRAN [17]. Here, we try to present a comparison of our results with other methods:

- We identify 58 out of total 59 redundant faults in the c5315 combinational benchmark circuit in 3.9 CPU seconds, while ATPG tool TRAN [17] identifies all of the 59 redundant faults in 32.3 CPU seconds. FIRE [39] identifies 20 redundant faults in 2.8 CPU seconds, TC [49] identifies 20 redundant faults in 3.4 CPU seconds and TC_{AND} [26] identifies 32 redundant faults in 3.4 CPU seconds.

- We identify all 34 redundant faults in the c6288 combinational benchmark circuit in 7.2 CPU seconds, while ATPG tool TRAN [17] identifies all of the 34 redundant faults in 38.0 CPU seconds. FIRE [39] identifies 33 redundant faults in 1.3 CPU seconds, TC [49] identifies 31 redundant faults in 1.0 CPU second and $\mathrm{TC_{AND}}$ [26] identifies 31 redundant faults in 1.8 CPU seconds.

- We identify all 38 redundant faults in the s713 combinational benchmark circuit in 0.6 CPU seconds, while TRAN identifies all of these redundant faults in 3.1 CPU seconds. FIRE identifies 32 redundant faults in 0.1 CPU seconds, TC identifies 29 redundant faults in 0.3 CPU seconds and $\mathrm{TC_{AND}}$ identifies 35 redundant faults in 0.3 CPU seconds.

We identify more redundant faults then all other fault-independent techniques in all the benchmark circuits, with a comparable CPU time of execution. In some benchmark circuits such as c5315, c6288, s349, s713 and s1423 we identify almost

[2] Combinational benchmark circuits.

[3] Sequential benchmark circuits.

all redundant faults as TRAN does, but we do it much faster in terms of CPU time of execution.

7.3 Summary

In this chapter, we described the results obtained by our technique at three different levels of implementation for ISCAS'85 and ISCAS'89 benchmark circuits. We also classified the redundancies identified by our technique as unexcitable, unpropagatable and undrivable. We also showed the number of redundant faults identified and CPU time taken by our technique on these benchmark circuits. Finally, we compared our technique with an ATPG tool and other fault-independent techniques that identify redundant faults. From this comparison, we find that our technique gives the best results when compared to the fault independent group of methods.

Chapter 8
Algorithm and Complexity

In this chapter, we will discuss our algorithm and its complexity in terms of the number of nodes and edges in the graph. Our algorithm has many sub-parts, due to all of the implementations and new concepts introduced in this thesis. We will also derive the total number of nodes required for the given circuit and the total number of edges obtained after the implication graph is built.

8.1 Algorithm

Our technique is completely summarized by the following steps:

```
(1)     Build an implication graph (for each Boolean gate in the given circuit);
(*)     do{
(*)       if (first time in the do loop)
(2a)          Calculate transitive closure of the implication graph();
(*)       else
(2b)          Update the transitive closure already present();
(*)       for (each fixed node identified from the transitive closure graph) {
(3)         Implement fixed-value theorem();
(4)         For fanout stems, identify fixed or unobservable dominator set and
(*)         implement fanout stem unobservability theorems();
(*)       }}
(*)       while (more fixed nodes identified);
```

8.1.1 A detailed explanation of each step of algorithm

- Step (1) – Build an implication graph: We build an implication graph for all of the Boolean gates in the given circuit, with the help of the Boolean false function, explained in detail, in Chapter 3. Building an implication graph is a one-time process.

- Step ($2a$) – Calculate transitive closure of the implication graph: After building the implication graph, we traverse beginning from each node in the graph in the depth-first manner to all possible nodes in the graph. Based on the conditions mentioned in Chapter 3, we identify the redundancies from the graph traversal. Building a transitive closure is also a one-time process.

- Step ($2b$) – Update the transitive closure already present: We only update the transitive closure already built from the implication graph, with the fixed node or stem unobservability information. This update process goes on until no more fixed nodes are found.

- Step (3) – Fixed-value theorem: We identify fixed nodes from the Step ($2a$ and $2b$) and implement the node fixation concept for all fixed nodes, as explained in Chapter 5. This process goes on until no more fixed nodes are found.

- Step (4) – Stem unobservability theorems: We identify fixed or unobservable dominator sets from the Step ($2a$ and $2b$) of the algorithm and implement the two theorems to identify the unobservable fanout stems, as explained in Chapter 6. This process goes on until no more fixed or unobservable dominator sets are found for a fanout stem.

8.2 Calculations for number of nodes and edges

The timing complexity of our algorithm, to identify redundancy, depends on two things:

- The number of controllability, observability and anding nodes in the graph for a given circuit.
- The number of direct and partial implications, relating the above-mentioned nodes in the graph.

Throughout the implementation, the number of nodes and partial implications in the graph remains constant. But, the number of direct implications increases, from the first stage where the implication graph is built, at the following levels of the implementation:

- After the transitive closure is calculated.
- After the fixed-value theorem is implemented.
- After the stem unobservability theorems are implemented.

In order to get some idea of complexity, we will derive the general formulae for calculating the number of controllability, observability and anding nodes in a given circuit. Also, we will calculate the total number of direct and partial implications after the implication graph is built. The number of partial edges remains constant because the number of anding nodes remains constant. The reader can use the following formulae on any circuit illustrated in this thesis or anywhere else to verify the manual results, for number of controllability, observability, anding nodes, direct implications[1] and partial implications required.

8.2.1 Controllability and observability nodes

We will first derive the total number of controllability and observability nodes required for an n-input gate and then will generalize it for a circuit consisting of k gates. The number of nodes, in the graph for a Boolean gate, depends on the

[1] After the implication graph is built.

number of inputs to the gate, and not on the gate type. For an n-input Boolean gate, the total number of C_{nodes} (controllability nodes) [55] will be:

$$\#C_{nodes} = 2(n+1) \tag{8.1}$$

Here, 1 is added to n to account for the output variable of the gate. A factor of 2 appears, because every signal line, in the implication graph, is expressed with two logic values, viz., true and false. For example, for signal line a of the gate shown in Figure 3.1 the graph will have two nodes a and $\overline{a}$ showing the true and false status of signal line a. With similar reasoning, the total number of O_{nodes} (observability nodes) will be:

$$\#O_{nodes} = 2(n+1) \tag{8.2}$$

Hence, the total number of controllability and observability nodes, in the implication graph, for a gate will be the sum of Equations 8.1 and 8.2:

$$total\ number\ of\ nodes = 4n + 4 \tag{8.3}$$

The total number of nodes [9] in a k-gate circuit will be:

$$\#total\ nodes \ = \ 4 \times [\#PIs + k] + 2 \times [\#fanout\ branches] \tag{8.4}$$

Here, a factor of 4 appears for PIs and k gates because of controllability and observability, and this can be easily deduced from Equation 8.3. But, a factor of only 2 appears for fanout branches because we do not use different controllability nodes for fanout stem and its branches as explained in Section 4.3. Out of these, $2 \times [\#PIs + K]$ is the total number of controllability nodes and the rest are the observability nodes.

$$\#total\ nodes \ = \ \sum_{i=1}^{k}(4n_i + 4) \tag{8.5}$$

The reason why the total number of nodes in a k-gate circuit cannot be as shown in Equation 8.5 is that a signal line that is the output of some gate may be input to some other gate. In this case, we do not use different nodes to describe the signal's controllability and observability.

8.2.2 Anding nodes

The anding nodes are required for both controllability and observability. For controllability, the total number of anding nodes [55] is given by following equation:

$$\#C_{ANDing\ nodes} = n + 1 \tag{8.6}$$

Here, n is the number of inputs of the gate being considered. For observability, the total number of anding nodes required is given by the following equation:

$$\#O_{ANDing\ nodes} = n(n + 1) \tag{8.7}$$

Here, n is the number of inputs of the gate being considered. Because each input of the gate is individually considered for observability, we have n times $n + 1$. Hence, the total number of anding nodes, for a gate, in the implication graph, is the sum of Equations 8.6 and 8.7:

$$\#ANDing\ nodes = n^2 + 2n + 1 \tag{8.8}$$

Now, the total number of anding nodes in the given circuit will be:

$$\text{Total \# of ANDing nodes} = \sum_{i=1}^{k}(n_i^2 + 2n_i + 1) \tag{8.9}$$

Here, k is the total number of gates present in the circuit and n_i is the number of inputs of the ith gate of the circuit. Since the n_i's are bounded, the total number of nodes in the implication graph is proportional to k [55].

8.2.3 Direct and partial implications

The total number of direct implications for an n-input gate is given by:

$$\text{Number of direct implications} = 2n + 2n^2 \tag{8.10}$$

The first term in Equation 8.10 refers to the number of direct implications for the controllability relationship and the second term refers to the number of direct implications for the observability relationship. The total number of partial implications for an n-input gate is given by:

$$\text{\#Partial implications} = n \times (\text{number of anding nodes}) \tag{8.11}$$

Hence, from Equation 8.8:

$$\#\text{Partial implications} = n \times (n^2 + 2n + 1) \tag{8.12}$$

For the complete circuit the total number of direct implications will be added for all gates as shown below:

$$\text{Total \# of direct implications} \;=\; \sum_{i=1}^{k}(2n_i + 2n_i^2) \tag{8.13}$$

For the complete circuit the total number of partial implications will be added for all the gates as shown by the following formula:

$$\text{Total \# of partial implications} \;=\; \sum_{i=1}^{k}(n_i^3 + 2n_i^2 + n_i) \tag{8.14}$$

Here k is the number of gates in the circuit and n_i is the number of inputs for the ith gate. The number of direct implications shown here is after the implication graph is built. Because the number of direct implications is difficult to trace after the transitive closure is calculated, and node fixation and stem unobservability theorems are implemented, we cannot generalize its mathematical formula.

8.3 Time complexity

We take the benchmark circuit c2670, for explaining the time complexity of each part of the algorithm shown in Section 8.1. There are totally 8752 controllability and observability nodes and 7653 anding nodes required for representing c2670 in an implication graph. Code profiling was done on the result of benchmark circuit c2670 using the GNU utility, *gprof.* From Table 7.4 the total CPU time taken to identify 82 redundant faults out of total 115 redundant faults by our algorithm is 4.0 seconds on a Sun Sparc-5 machine. The following is the details of time taken by each step of the algorithm:

- Building an implication graph took 0.4 CPU second. This is a very fast process because only the edges are constructed in this phase relating the local input and output nodes of a gate.

- Calculating a transitive closure graph took 2.5 CPU seconds. This process takes most of the time in our algorithm, because all of the global implications are found in this phase.

- Updating a transitive closure graph took 0.3 CPU seconds. This is just an updating process and so it does not take much time.

- The fixed-value theorem took 0.4 CPU seconds. This process does not take much time because for all the fixed nodes found all the other nodes are just updated for their outgoing edges.

- The stem unobservability theorems took 0.4 CPU seconds. This process does not take much time, again, as just the dominator stem for fanout stems has to be verified for its fixation or unobservability.

8.4 Summary

In this chapter, we presented our algorithm for redundancy identification and showed the complexity analysis of our technique in terms of number of nodes and edges in the transitive closure graph. We also showed the analysis of benchmark circuit c2670 in terms of the CPU time taken for each step of the algorithm. The following chapter presents conclusions and future work.

Chapter 9
Conclusions and Future Work

Our method identifies more redundant faults than other fault-independent techniques such as TC by Lin [49], TC_{AND} by Gaur [26] and FIRE by Iyer and Abramovici [39]. The advantages of this method are: we classify the redundant faults identified, and we identify a subset of the stem redundant faults and primary output faults; all of this is achieved without exhaustive test pattern generation.

9.1 Limitations

The limitation of our method lies in identifying some stem unobservabilities that do not have fixed or unobservable dominator sets, as shown in Figure 6.6. The inability of taking decisions, while traversing nodes in the graph, is another limitation of our method, as shown in Figure 5.4. Another example of this limitation is traversing from the complement node of output $\bar{c}$ of the Boolean AND gate of the Figure 3.1. We partly solved this problem when node $\bar{c}$ is fixed as is explained in Chapter 5.

9.2 Future work

There are many applications of our algorithm. A few of them are listed below:

1. Fault equivalence: The structural as well as the functional fault equivalence can be found using our algorithm. Some work has already been done as a course project, where each line in the circuit is assigned a s-a-0 and s-a-1 node. If these fault nodes for two different lines imply each other than those

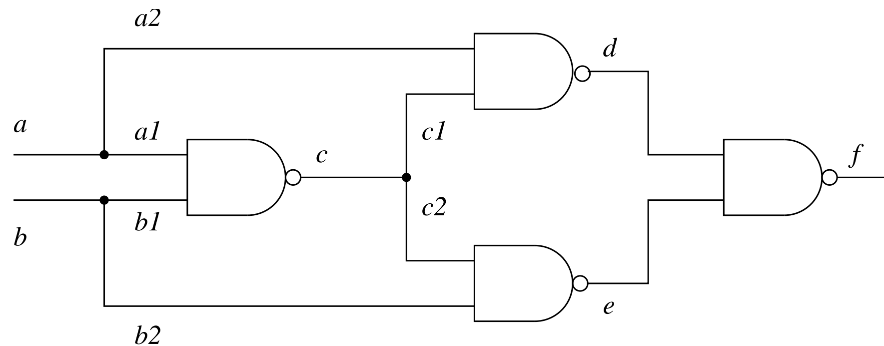

Figure 9.1: Example circuit showing the working of fault equivalence.

two faults are equivalent. If they relate the same gate they are classified as structurally equivalent or else they are functionally equivalent faults, defined in Chapter 3. A small experiment on an XOR gate implemented with NAND gates revealed that our technique can identify all of the structurally equivalent faults and a few functionally equivalent faults, such as $a2$ s-a-1 and $b2$ s-a-1 shown in Figure 9.1. The other functionally equivalent faults not identified were: (c s-a-0 and f s-a-0) and (c s-a-1, $c1$ s-a-1, $c2$ s-a-1). All functionally equivalent faults in this circuit are identified by Prasad *et al.* [62].

2. Equivalence checking: Finding the functional equivalence of two given circuits is a *NP-hard* problem [21]. Some work had been done by Gaur *et al.* [26] to find functional equivalence of two circuits, using the transitive closure technique. This application can be further explored.

3. Logic simulator:

   - 2-value logic simulator: By forcing the input signals to true or false nodes (with an edge from the true to false node or vice-versa), the test vector can be applied and the graph traversal (or transitive closure) will then be the simulation result for that test vector. So our technique may be an exact two value logic simulator. In Figure 4.8, some typical 2-value logic simulator will assign a test vector to the primary inputs and will propagate its effects to the primary output. Let us say the

test vector is 1. Then signal b will be 0 and c will be 0. Our algorithm will start the traversal from node a and will imply node $\overline{b}$ and $\overline{b}$ will then imply node $\overline{c}$. Thus, our algorithm does similar work as is done by a typical 2-value logic simulator.

- 3-value logic simulator: Without forcing the input signals to true or false nodes, it becomes a 3-value logic simulator. That is, for Figure 4.8 we start with X (unknown) state at the input of the circuit, as we are not forcing it to any particular state. Still we reach the primary output signal $\overline{c}$ as explained below. Although, we may start from any node a or $\overline{a}$, we end up at $\overline{c}$, so that means that without knowing the status of the primary input a the status of primary output c can be known. So, in a way we do much better than a typical 3-valued logic simulator, which would rather give the status of the primary output c as X (unknown), instead of $\overline{c}$ as our algorithm does.

The following improvements can be made to the existing program, to (maybe) identify more redundant faults:

- For all of the paths from a stem to its reconvergent gate, analysis can be done for a number of paths that have odd and even numbers of inversions to prove the stem unobservable.

- More general conditions to relate stem and branch observabilities can be developed depending on circuit topology.

- If traversal has to begin from the complement node of the output signal of an AND gate, then any of the input signal to the AND gate can be 0, which is not certain. So, some decision has to be taken to determine which input signal(s) is (are) false. Some method can be developed to tackle this decision making limitations.

References

[1] M. Abramovici and M. A. Iyer, "One-pass Redundancy Identification and Removal," in *in Proc. of the International Test Conf.*, September 1992, pp. 807–815.

[2] V. D. Agrawal and P. Agrawal, "An Automatic Test Generation System for Illiac IV Logic Boards," *IEEE Trans. on Computers*, vol. C-21, no. 9, pp. 1015–1017, September 1972.

[3] V. D. Agrawal, M. L. Bushnell, and Q. Lin, "Redundancy Identification Using Transitive Closure," in *Proc. of the 5th Asian Test Symp.*, November 1996, pp. 4–9.

[4] V. D. Agrawal and M. R. Mercer, "Testability Measures - What Do They Tell Us?" in *Proc. of the ITC*, November 1982, pp. 391–396.

[5] S. B. Akers, "Binary Decision Diagrams," *IEEE Trans. on Computers*, vol. 27, no. 6, pp. 66–73, June 1978.

[6] S. B. Akers, "Functional Testing with Binary Decision Diagrams," in *Proc. of the International Fault-Tolerant Computing Symp.*, June 1978, pp. 82–92.

[7] F. Brglez, "On Testability Analysis of Combinational Networks," in *Proc. of the International Symp. on Circuits and Systems*, May 1984, pp. 221–225.

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

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

[10] M. L. Bushnell and J. Giraldi, "A Functional Decomposition Method for Redundancy Identification and Test Generation," *Journal of Electronic Testing: Theory and Applications*, vol. 10, pp. 175–195, June 1997.

[11] C. W. Cha, W. E. Donath, and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Trans. on Computers*, vol. C-27, no. 3, pp. 193–200, March 1978.

[12] S. T. Chakradhar, *Neural Network Models and Optimization Methods for Digital Testing.* PhD thesis, Rutgers University, CS Dept., October 1990.

[13] S. T. Chakradhar and V. D. Agrawal, "A Transitive Colosure Based Algorithm for Test Generation," in *Proc. of the* 28th *Design Automation Conf.*, June 1991, pp. 353–358.

[14] S. T. Chakradhar, V. D. Agrawal, and M. L. Bushnell, "Automatic Test Pattern Generation Using Quadratic 0-1 Programming," in *Proc. of the* 27th *Design Automation Conf.*, June 1990, pp. 654–659.

[15] S. T. Chakradhar, V. D. Agrawal, and M. L. Bushnell, "Neural Net and Boolean Satisfiability Models of Logic Circuits," *IEEE Design and Test of Computers*, vol. 7, no. 5, pp. 54–57, October 1990.

[16] S. T. Chakradhar, V. D. Agrawal, and M. L. Bushnell, *Neural Models and Algorithms for Digital Testing.* Boston: Kluwer Academic Publishers, 1991.

[17] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Trans. on Computer-Aided Design*, vol. 12, no. 7, pp. 1015–1028, July 1993.

[18] S. T. Chakradhar, M. L. Bushnell, and V. D. Agrawal, "Automatic Test Generation Using Neural Networks," in *Proc. of the International Conf. on Computer-Aided Design*, November 1988, pp. 416–419.

[19] X. Chen, *State and Objective Learning for Sequential Circuit Automatic Test Pattern Generation.* PhD thesis, Rutgers University, ECE Dept., October 1993.

[20] X. Chen and M. L. Bushnell, *Efficient Branch and Bound Search with Application to Computer-Aided Design.* Boston: Kluwer Academic Publishers, 1996.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 1990.

[22] G. de Micheli, *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.

[23] R. D. Eldred, "Test Routines Based on Symbolic Logical Statements," *Journal of the ACM*, vol. 6, no. 1, pp. 33–36, January 1959.

[24] R. K. Gaede, M. R. Mercer, K. M. Butler, and D. E. Ross, "CATAPULT: Concurrent Automatic Testing Allowing Parallelization and Using Limited Topology," in *Proc. of the* 25th *Design Automation Conf.*, June 1988, pp. 597–600.

[25] J. M. Galey, R. E. Norby, and J. P. Roth, "Techniques for the Diagnosis of Swiching Circuit Failures," in *Proc. of the Second Annual Symp. on Switching Circuit Theory and Logical Design*, October 1961, pp. 152–160.

[26] V. Gaur, "A New Transitive Closure Algorithm to Identify Redundancies in Logic Circuits," Master's thesis, Rutgers University, ECE Dept., January 2002.

[27] V. Gaur, V. D. Agrawal, and M. L. Bushnell, "A New Transitive Closure Algorithm with Applications to Redundancy Identification," in *Proc. of the 1st International Workshop on Electronic, Design and Test Applications (DELTA'02)*, January 2002, pp. 496–500.

[28] P. Goel, "An Impicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," in *Proc. of the International Fult-Tolerant Computing Symp.*, August 1980, pp. 145–151.

[29] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, vol. C-30, no. 3, pp. 215–222, March 1981.

[30] L. H. Goldstein, "Controllbility/Observability Analysis of Digital Circuits," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, no. 9, pp. 685–693, September 1979.

[31] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia Controllability/Observability Analysis Program," in *Proc. of the* 17th *Design Automation Conf.*, June 1980, pp. 190–196.

[32] F. Harary, *Graph Theory.* Reading, MA: Addison-Wesley, 1972.

[33] M. Harihara and P. R. Menon, "Identification of Undetectable Faults in Combinational Circuits," in *Proc. of the International Conf. on Computer Design*, October 1989, pp. 290–293.

[34] M. Henftling and H. Wittmann, "A New Data Structure to Solve the Satisfiability Problem in Digital Circuits," *Archiv fiir Elecktronik Und Obertragungstechnik*, vol. 49, no. 1, pp. 29–43, January 1995.

[35] M. Henftling, H. Wittmann, and K. J. Antreich, "A Formal Non-Heuristic ATPG Approach," in *Proc. of the European Design Automation Conf.*, September 1995, pp. 248–253.

[36] M. Henftling and H. C. Wittmann, "Bit Parallel Test Pattern Generation for Path Delay Faults," in *Proc. of the European Design and Test Conference*, March 1995, pp. 521–525.

[37] O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Trans. on Computers*, vol. C-24, no. 3, pp. 242–249, March 1975.

[38] M. A. Iyer and M. Abramovici, "Low-cost Redundancy Identification for Combinational Circuits," in *in Proc. of* 7th *International Conf. on VLSI Design*, January 1994, pp. 315–318.

[39] M. A. Iyer and M. Abramovici, "FIRE: A Fault-Independent Combinational Redundancy Identification Algorithm," *IEEE Transactions on VLSI Systems*, vol. 4, no. 2, pp. 295–301, June 1996.

[40] M. A. Iyer, D. Long, and M. Abramovici, "Identifying Sequential Redundancies Without Search," in *in Proc. of* 33rd *ACM IEEE Design Automation Conf.*, June 1996, pp. 457–462.

[41] S. K. Jain and V. D. Agrawal, "Statistical Fault Analysis," *IEEE Design and Test of Computers*, vol. 2, no. 1, pp. 38–44, February 1985.

[42] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," in *Proc. of the* 24th *Design Automation Conf.*, June-July 1987, pp. 502–508.

[43] W. Kunz and D. K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," in *Proc. of the IEEE International Test Conf.*, September 1992, pp. 816–825.

[44] W. Kunz and D. K. Pradhan, "Recursive Learning – A New Implication Technique for Efficient Solution to CAD Problems," *IEEE Trans. on Computer-Aided Design*, vol. 13, no. 9, pp. 1143–1158, September 1994.

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

[46] T. Larrabee, "Efficient Generation of Test Patterns Using Boolean Difference," in *Proc. of the International Test Conf.*, August 1989, pp. 795–801.

[47] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 4–15, January 1992.

[48] C. Y. Lee, "Representation of Switching Circuits by Binary Decision Diagrams," *Bell System Technical Journal*, vol. 38, pp. 985–999, July 1959.

[49] Q. Lin, "Efficient Techniques for a Transitive closure-Based Test Generation Algorithm," Master's thesis, Rutgers University, ECE Dept., January 1996.

[50] D. E. Long, M. A. Iyer, and M. Abramovici, "Identifying Sequentially Untestable Faults Using Illegal States," in *Proc. of the* 13th *IEEE VLSI Test Symp.*, May 1995, pp. 4–11.

[51] H. K. T. Ma, S. Devadas, A. Sangiovanni-Vincentelli, and R. Wei, "Logic Verification Algorithms and their Parallel Implementation," in *Proc. of the* 24th *Design Automation Conf.*, June-July 1987, pp. 283–290.

[52] H. K. T. Ma, S. Devadas, A. Sangiovanni-Vincentelli, and R. Wei, "Logic Verification Algorithms and their Parallel Implementation," *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 2, pp. 181–189, February 1989.

[53] V. J. Mehta, V. D. Agrawal, and M. L. Bushnell, "Fixed-Value and Stem Unobservability Theorems for Logic Redundancy Identification," in *Proc. of the International Conference on Computer-Aided Design*, November 2003. Submitted.

[54] V. J. Mehta, V. D. Agrawal, and M. L. Bushnell, "Theorems on Redundancy Identification," in *Proc. of the* 12th *North Atlantic Test Workshop*, May 2003.

[55] V. J. Mehta, K. K. Dave, V. D. Agrawal, and M. L. Bushnell, "A Fault-Independent Transitive Closure Algorithm for Redundancy Identification," in *Proc. of the* 16th *International Conf. VLSI Design*, January 2003, pp. 149–154.

[56] P. R. Menon and H. Ahuja, "Redundancy Removal and Simplification of Combinational Circuits," in *Proc. of the* 10th *IEEE VLSI Test Symp.*, April 1992, pp. 268–273.

[57] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Trans. on Computers*, vol. C-25, no. 6, pp. 630–636, June 1976.

[58] T. M. Niermann and J. H. Patel, "HITEC: A Test Generation Package for Sequential Circuits," in *Proc. of the European Design Automation Conference*, February 1991, pp. 214–218.

[59] K. P. Parker and E. J. McCluskey, "Probabilistic Treatment of General Combinational Networks," *IEEE Trans. on Computers*, vol. C-24, no. 6, pp. 668–670, June 1975.

[60] Q. Peng, M. Abramovici, and J. Savir, "MUST: Multiple-stem Analysis for Identifying Sequentially Untestable Faults," in *in Proc. of the International Test Conf.*, September 2000, pp. 839–846.

[61] J. F. Poage, "Derivation of Optimum Tests to Detect Faults in Combinational Circuits," in *Proc. of the Symp. on Matehmatical Theory of Automata,(New York), Polytechnic Press*, April 1963, pp. 483–528.

[62] A. V. S. S. Prasad, V. D. Agrawal, and M. V. Atre, "A New Algorithm for Global Fault Collapsing into Equivalence and Dominance Sets," in *Proc. of the International Test Conf.*, October 2002, pp. 391–397.

[63] I. M. Ratiu, A. Sangiovanni-Vincentelli, and D. O. Pederson, "VICTOR: A Fast VLSI Testability Analysis Program," in *Proc. of the IEEE International Test Conference*, November 1982, pp. 397–401.

[64] J. P. Roth, "Diagonosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, no. 4, pp. 278–291, July 1966.

[65] J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. on Electronic Computers*, vol. EC-16, no. 5, pp. 567–580, October 1967.

[66] J. Savir, "Good Controllability adn Good Observability do not Guarantee Good Testability," *IEEE Trans. on Computers*, vol. C-32, pp. 1198–1200, December 1983.

[67] J. Savir, G. S. Ditlow, and P. H. Bardell, "Random Pattern Testability," *IEEE Trans. on Computers*, vol. C-33, no. 1, pp. 79–90, January 1984.

[68] M. H. Schulz and E. Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," in *Proc. of the International Fault-Tolerant Computing Symp.*, June 1988, pp. 30–35.

[69] M. H. Schulz and E. Auth, "Improved Determininstic Test Pattern Generation with Applications to Redundancy Identification," *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 7, pp. 811–816, July 1989.

[70] M. H. Schulz, E. Trischler, and T. M. Serfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. on Computer-Aided Design*, vol. 7, no. 1, pp. 126–137, January 1988.

[71] F. F. Sellers, M. Y. Hsiao, and L. W. Bearnson, "Analyzing Erros with the Boolean Difference," *IEEE Trans. on Computers*, vol. C-17, no. 7, pp. 676–683, July 1968.

[72] F. F. Sellers, M. Y. Hsiao, and L. W. Bearnson, *Error Detecting Logic for Digital Computers.* New York: McGraw-Hill, 1968.

[73] S. Seshu and D. N. Freeman, "The Diagnosis of Asynchronous Sequential Switching Systems," *IRE Trans. on Electronic Computers*, vol. EC-11, pp. 459–465, August 1962.

[74] S. C. Seth and V. D. Agrawal, "A New Model for Computation of Probabilistic Testability in Combinational Circuits," *Integration, the VLSI Journal*, vol. 7, no. 1, pp. 49–75, April 1989.

[75] J. P. M. Silva and K. A. Sakallah, "Grasp – A New Search Algorithm for Satisfiability," in *Proc. of the International Conf. on Computer-Aided Design*, November 1996, pp. 220–227.

[76] T. Stanion and D. Bhattacharya, "TSUNAMI: A Path Oriented Scheme for Algebraic Test Generation," in *Proc. of the International Fault-Tolerant Computing Symp.*, June 1991, pp. 36–43.

[77] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *IEEE Trans. on Computer-Aided Design*, vol. 15, no. 9, pp. 1167–1176, September 1996.

[78] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking and Optimization of Netlists," in *Proc. of the International Conf. on Computer-Aided Design*, November 1997, pp. 648–655.

[79] R. S. Wei and A. Sangiovanni-Vincentelli, "PROTEUS: A Logic Verification System for Combinational Circuits," in *Proc. of the International Test Conf.*, October 1986, pp. 350–359.

[80] J. K. Zhao, E. M. Rudnick, and J. H. Patel, "Static Logic Implication with Application to Redundancy Identification," in *Proc. of the* 15th *IEEE VLSI Test Symp.*, April 1997, pp. 288–293.