On Necessary and Nonconflicting Assignments in Algorithmic Test Pattern Generation

Henry Cox

B. Eng., McGill University, 1986

Department of Electrical Engineering
McGill University

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

April 1991

Abstract

Necessary, *nonconflicting*, and *arbitrary* assignments can be distinguished during algorithmic test pattern generation. A necessary assignment is one which must be made in order to find a test—there is no test in the half-space defined by the opposite assignment. Certain other assignments are nonconflicting in the sense that they narrow the search space and never lead to backtracking—if the fault is testable, then there is at least one test vector in the half-space defined by the assignment. The remaining assignments are arbitrary—they may or may not lead in the direction of a test and may or may not cause backtracking. Since necessary and nonconflicting assignments do not lead to backtracking, an efficient test pattern generation algorithm should determine and apply them prior to any arbitrary branching.

This thesis presents algorithms based on the mathematical properties of images and inverse images of set functions to identify necessary and nonconflicting assignments in combinational logic circuits. Issues relating to the efficient implementation of these algorithms, are addressed from both a theoretical and practical perspective. Experimental results obtained on a variety of benchmark circuits show that algorithmic assignment identification can be used to reduce or eliminate backtracking in automatic test pattern generation.

Résumé

L'on peut discerner trois types d'allocations lors de la production d'échantillons pour fin de test: nécessaires, non-conflictuelles, et arbitraires. Une allocation nécessaire est une allocation qui doit être faite pour résulter en un test. C'est-à-dire qu'il n'y a pas de test possible dans l'espace défini par l'allocation inverse. D'autres allocations sont non-conflictuelles dans le sens qu'elles diminuent l'espace de recherche pour un vecteur de test et qu'elles ne requièrent pas de réajustements une fois l'allocation décidée. Si une faute est vérifiable par test, alors il existe au moins un vecteur de test dans le sous-espace défini par une allocation non-conflictuelle. Les allocations arbitraires sont incertaines. Il est possible qu'elles mènent à des réajustements (une revision de l'allocation dans un temps futur) Étant donné que les allocations nécessaires et non-conflictuelles ne nécessitent jamais de réajustements, un algorithme efficace pour la production de vecteurs de test devrait les considérer avant les allocations arbitraires.

Cette thèse présente des algorithmes basés sur les propriétés mathématiques des images et des images inverses des fonctions sur les ensembles. Ces algorithmes s'appliquent pour identifier les allocations nécessaires et non-conflictuelles dans les circuits de logique combinatoire. La réalisation efficace de ces algorithmes, du point de vue théorique et pratique, est étudiée. Des résultats expérimentaux sont donnés pour plusieurs circuits étalons. Ces résultats démontrent que l'on peut réduire ou éliminer les réajustements dans la production automatique d'échantillions pour fin de test.

Acknowledgements

I would like to thank my supervisor, Janusz Rajski, for his assistance and guidance throughout my studies. His vision and enthusiasm are greatly appreciated and cannot be overemphasized.

I am also grateful for the assistance and support of my friends in both the VLSI Design Lab and the Robotics Lab at McGill. In particular, I would like to thank Mike Parker (der Mouse), who seems to know an efficient way to code any algorithm and whose knowledge of the UNIX environment is legendary. Rob Aitken and Fadi Maamari have been sources of many useful ideas and discussions. I would also like to thank all the other members of the lab, past and present, for making it both a stimulating and enjoyable place to work over the last 5 years.

I would like to acknowledge the generous financial support provided me by the Natural Sciences and Engineering Research Council of Canada and the Centre de recherche informatique de Montréal.

Finally, I would like to thank Sally Duscha for putting up with a lot while I was coding the algorithms and preparing this manuscript as well as for proofreading above and beyond the call of duty. I have it on reliable authority that it is not much fun to be around me when I am debugging. I would also like to thank Dudley and Clara for giving me an excuse to stop work and go to the park for a few minutes.

Claim of Originality

The author claims originality for the following contributions of this dissertation:

The new concept of *reduction lists*, based on the mathematical properties of images and inverse images of set functions, is introduced. The properties of reduction lists are studied and algorithms to compute them are presented. Applied to the problem of automatic test pattern generation, the reduction lists are used to identify *necessary assignments* Necessary assignments are those assignments which must be made in order to generate a test pattern—no test exists in the space defined by the alternate assignment(s). Reduction analysis is unifying in the sense that all other methods of identifying necessary assignments, including conventional backward implication, dominator identification, and learning, are special cases of this general concept.

The new concept of *tendency lists*, based on mathematical properties of monotonicity (unateness) of Boolean functions, is defined and algorithms to compute them are presented. The tendency lists are used to identify a novel class of algorithmic assignments during automatic test pattern generation termed *nonconflicting assignments* Nonconflicting assignments lead in the direction of a test pattern by narrowing the remaining space which must be searched, but are guaranteed never to need to be backtracked.

These algorithmic assignment identification techniques imply that a global analysis of the effect of assignments to nodes in the circuit under test can be performed by a local computation at individual gates in the network. This information can be used to identify several classes of algorithmic assignments during test pattern generation. The global nature of the computation is achieved through the indexation of the node assignments analysed, which then propagate throughout the network on locally computed lists

Structural properties of the circuit under test determined by an analysis of reconvergent fanout are exploited to limit the number of assignments which are analysed and the area of the circuit which is processed in order to reduce the amount of computation required to identify necessary and nonconflicting assignments The use of preprocessed reduction information including ***generic reduction lists*** and ***propagate assignments*** to further reduce required computation is discussed.

A new test pattern generation algorithm is developed. The core of QUEST algorithm is the identification of necessary and nonconflicting assignments. Arbitrary branching is delayed as long as possible to avoid making assignments which may lead to backtracking. Necessary and nonconflicting assignments are extracted iteratively until a test pattern is generated or no more algorithmic assignments can be found, at which point an arbitrary branch is made. If a conflict is detected, then a backtrack is performed to reverse the most recent arbitrary assignment. After each arbitrary branch or backtrack, algorithmic assignments are again identified. The algorithm is complete; the process continues until a test pattern is generated or the target fault is proven to be untestable.

Table of Contents

Chapter 1 Introduction 1

1.1 The role of test pattern generation 3

1.2 Outline of dissertation 5

1.3 Notation conventions 7

1.4 Publication history 7

Chapter 2 An overview of algorithmic test pattern generation 9

2.1 Logic systems for test generation 10

2.2 Necessary assignment identification techniques 18

2.3 Developments in nonconflicting assignment identification 22

2.4 Deterministic test generation algorithms 23

Chapter 3 Images and inverse images of set functions 27

3.1 Images of set functions: forward propagation 27

3.2 Inverse images of set functions: backward implication 30

3.3 Limitations of set functions 34

Chapter 4 Reduction list calculation: a method to identify necessary assignments 37

4.1 Reduction lists and necessary assignment identification 40

4.2 Logical constraints and propagation of implications 48

4.3 General theorem of reduction 51

4.4 Stem correlation 53

4.5 Complexity of test pattern generation and the computation of reduction lists 56

Chapter 5 Exploiting circuit topology in efficient identification of necessary assignments 59

5.1 Structural analysis of reconvergence in combinational circuits 59

5.2 Candidate assignment identification 61

5.3 Region of propagation 69

Chapter 6 A method to identify nonconflicting assignments based on Boolean function monotonicity 71

6.1 Generalized function monotonicity and tendency lists 72

6.2 Tendency lists and nonconflicting assignments 74

6.3 Pessimism in the tendency analysis 78

Chapter 7 The QUEST test pattern generation algorithm 82

7.1 Organization of the test pattern generation system 83

7.2 Preprocessed information 84

7.3 Experimental results 89

Chapter 8 Conclusions 94

Chapter 9 References 95

List of Figures

1.1 Notation conventions . . . 7
2.1 Test pattern generation using a 5-valued alphabet . . . 11
2.2 Test pattern generation using a 9-valued alphabet . . . 12
2.3 Test pattern generation using the SPLIT circuit model . . . 13
2.4 Test pattern generation using a 16-valued alphabet . . . 15
2.5 Classical backward implication . . . 18
2.6 Unique sensitization in the FAN algorithm . . . 19
2.7 Learning in SOCRATES . . . 20
3.1 Images for a 2-input *AND* gate . . . 27
3.2 Forward propagation in a circuit . . . 30
3.3 Inverse images for a 2-input *AND* gate . . . 30
3.4 Backward implication in the circuit from Fig. 3.2 . . . 33
3.5 Pessimism in the forward implication step . . . 34
3.6 Three implementations of the exclusive-or function and corresponding characteristic equations . . . 35
4.1 Test pattern generation for f_{s_1} . . . 38
4.2 The search forest . . . 38
4.3 The *AND-OR* graph of justification points . . . 42
4.4 Reduction list calculation in a circuit . . . 44
4.5 Reduction list calculation with partially determined circuit values . . . 45
4.6 Necessary assignments in the region of the D symbol . . . 46
4.7 Test generation for B_{s_0} . . . 47
4.8 Circuit nodes with associated reduction lists . . . 48
4.9 Test generation for fault f_{s_0} . . . 50
4.10 Value justification of a full adder . . . 50
4.11 Correlation of assignments . . . 54
4.12 Stem correlation in the D-region—propagation of the fault effect 55

4.13 Stem correlation in the D-region—no propagation of the fault effect 55
4.14 A 2-input trivial function 57
5.1 Candidate assignment identification 62
5.2 Transfer of requirements from one justification point to another 65
5.3 Premise: No stems exist whose stem regions include $J1$ and at least one other justification point 66
5.4 Propagation of gate assignments 67
5.5 Test generation for z_{s_0} 68
5.6 Property 5.5: the region of reduction list propagation 69
6.1 Pruning assignments 71
6.2 Test generation for C_{s_0} 76
6.3 Nonconflicting assignment identification in redundancy identification 77
6.4 Test generation for E_{s_1} 79
6.5 Test generation for E_{s_1} 80
7.1 The QUEST test pattern generation algorithm 83
7.2 Propagate assignment identification 87
7.3 Propagate assignments example 87

List of Tables

2.1 Gate functions: $B_2^2 \times B_2^2 \rightarrow B_2^2$ 10

2.2 Gate functions in a 5-valued algebra 11

2.3 Gate functions in a 9-valued algebra 12

2.4 Gate functions in an 11-valued algebra 13

2.5 Three codings of a 16-element alphabet 15

2.6 Comparison between algebras 16

3.1 Gate functions in a 16-valued logic system: $B_{16} \times B_{16} \rightarrow B_{16}$ 29

3.2 Inverse images in B_{16} 33

7.1 Experimental results 90

7.2 The effect of nonconflicting assignments 92

Chapter 1 Introduction

With the increasing complexity of VLSI circuits and the growing demand for very high shipped product quality and reliability, test has become one of the most important and costly phases of integrated circuit production. Test pattern generation, whether manual or automatic, is a key part of any test methodology. Extremely high quality requirements, measured in single defective parts per million, have made it essential to find test patterns for all testable faults while identifying all untestable faults in order to guarantee complete coverage by the test set.

The processes used to fabricate integrated circuits are both very complex and imperfect. Only some of the circuits produced work correctly, as random defects introduced during the fabrication process cause a portion of them to fail. These failures may be *logical* in that they change the function of the faulty circuit, or *parametric* in that some operating parameter of the circuit, such as current drive capability, output voltage level, *etc.*, is affected [Ravi87, ShMaFe85, McCMou87]. The goal of testing is to screen out defective circuits so that only fault-free ones remain. This goal can be very difficult to achieve, as the circuits may contain hundreds of thousands or millions of potentially faulty individual devices yet have only a few hundred signals which can be directly stimulated and/or observed.

Testing is performed by applying a set of input patterns to the circuit, a *test set*, which differentiates between fault-free and faulty circuits—*i.e.* a faulty circuit will behave differently than a fault-free circuit when the test set is applied. For the purposes of this discussion, it is assumed that the circuit design is correct in the sense that if perfectly fabricated, the circuit would perform exactly according to specifications.

Techniques for obtaining a test set can be broadly categorized as *functional* or *structural* testing. Functional testing attempts to verify that the circuit under test performs properly, without reference to the way the circuit itself was designed and fabricated or the defects it is subject to [ThaAbr80, LaiSie83, AbBrFr90]. Although functional tests for certain types of regular structures such as memories thoroughly test the devices [NaThAb78], functional tests for less regular designs often cover an unacceptably low number of physical defects [Klug88, AbBrFr90]. Structural testing attempts to verify that the individual devices within the circuit work properly and that their connectivity is correct, and thus that the circuit as a whole is fault-free [AbBrFr90].

Structural testing usually makes use of a *fault model* which is intended as a high-level abstraction that represents the actual defects which the circuit under test may experience. Many different fault models have been proposed, of which the most common is the *single stuck-at* (SSA) model, originally proposed in [Eld59], in which it is assumed that all fabrication defects can be modeled by a single line in the circuit which permanently carries a logic 0 or logic 1. Many authors have questioned the applicability of the stuck-at model and have proposed alternate fault models—among them, MOS stuck-open and stuck-on faults [Wad78], bridging faults [Mei74], transition faults [WaLiRoly87] and crosspoint faults in programmable logic arrays [Smith79]—which may more accurately represent device failures. The most common justification for the continued use of the stuck-at fault model is that test sets developed under the stuck-at fault model tend to be excellent tests for other types of faults as well (termed "windfall coverage") [WilPar83]. Once a particular fault model is chosen, the quality (or goodness) of the test set is measured by the *fault coverage*—the proportion of faults from the fault model which are detected by the test—and has a direct impact on the *defect level*—the number of faulty circuits which are incorrectly declared good [WilBro81].

Digital logic circuits can be divided into two classes, *combinational* circuits, whose output depends only on their current input, and *sequential* circuits, whose output depends on both their current input and internal state, determined by previously applied input patterns. Sequential circuits perform a rich set of useful tasks which often cannot be performed by combinational circuits. Consequently, most circuits which are produced are sequential. Unfortunately, testing sequential circuits is significantly more difficult than testing combinational ones, as their memory must be considered [Kautz68, BoHsPu71]. This implies that the entire sequence of test patterns must be treated as a whole in sequential testing, rather than a single vector at a time in combinational testing. In addition, changes to the internal states and the transitions between them due to the presence of faults must be taken into account.

To cope with the complexities of analysing sequential circuits, structured design for testablility techniques have been used to convert sequential circuits into pseudo-combinational circuits during test mode by making the memory elements directly controllable and observable [Ando80, FuKaYa89, Stew77, WilAng73, EicWil77]. At the same time, scan design techniques have created a new set of extremely large "combinational" circuits, prompting a great deal of interest in efficient tools for combinational circuit testing.

1.1 The role of test pattern generation

Within the context of structural testing with a given fault model, there are two basic methods to obtain test: methods based on *fault simulation* of a set of patterns—for example, a set of random or pseudo-random patterns with [ScLiCa75, MuAgND90] or without input weighting [BaMcSa87, Golomb87, HorMcL89]—and methods based on *test pattern generation* [Roth66, Goel81, FujShi83, KirMer87, SchAut89]. Regardless of the chosen technique, however, deterministic test pattern generation plays a vital role.

It is sufficient to find a test pattern for each testable fault using any available technique—for example, either deterministic test generation or random pattern fault simulation can be used. However, in order to prove that a fault is untestable, it is necessary to prove that no test pattern exists for the fault. *Redundancy identification* cannot usually be done using fault simulation-based techniques in an acceptable amount of time, as the number of patterns which have to be fault simulated in order to exhaust the search space is huge—2^n patterns for an n-input combinational circuit, for example. In addition, testable faults which are extremely unlikely to be covered by random test patterns exist [Wun88]; generally, tests for these faults must be found deterministically.

The goal of *deterministic test pattern generation* is to find a test vector (or sequence of test vectors) for a given *target fault* from the fault model which will distinguish between a fault-free circuit and a circuit which contains the fault—*i.e.* the output of the circuit under test will differ depending whether or not it contains the fault. The target fault is *untestable* if no such test vector (test sequence) exists.

Test pattern generation can be viewed as a branch and bound problem [Goel81]; test generation algorithms usually search for a test pattern by systematically branching and bounding until either a vector is discovered or the search space is exhausted. Even for combinational circuits, the test generation problem is *NP*-complete [FujToi82, GarJoh78]—in the worst case, all known test pattern generation algorithms will require exponential time to find a test vector or prove that none exist.

Deterministic test pattern generation is a process of progressively translating a set of required values at some nodes in the circuit to a new set of requirements at other nodes which satisfy the original requirements, but are closer to primary inputs. A test pattern has been successfully generated when all requirements are satisfied by assignments to primary inputs. The fault has been proven untestable if no test pattern exists which will satisfy the requirements.

This thesis characterizes three types of assignments made during the course of deterministic test pattern generation. A *necessary assignment* to a node (also termed a "mandatory assignment" in [KirMer87] and "single pruning" in [RobRaj88]) is one which must be made in order to find a test—there is no test pattern in the space defined by alternate assignment(s) to the node. Viewed as a branch decision, assigning any other value to the node is equivalent to branching into an area of the search space which does not contain a test pattern, guaranteeing that a bound step must eventually be taken. A *nonconflicting assignment* (termed "monotone pruning" in [RobRaj88]) is one which leads in the direction of a test by restricting the space which remains to be searched, but never needs to be backtracked. If the fault is testable, then a test vector is guaranteed to be found in the space defined by the nonconflicting assignment. The remaining assignments are *arbitrary* or *branch assignments*—they may or may not lead to a test pattern, and must be backtracked if a test cannot be found after they have been assigned.

A general theory describing the identification of necessary and nonconflicting assignments, based on the mathematical concepts of images and inverse images of set functions, is developed in this thesis. New techniques to identify necessary and nonconflicting assignments in deterministic test pattern generation are presented. The identification of necessary and nonconflicting assignments is algorithmic in the sense that there is no element of choice or luck in the computation, no reliance on heuristics, and no possibility of these assignments causing a backtrack if the fault is testable.

Issues relating to the efficient implementation of the algorithms are presented from both a theoretical and practical point of view. In particular, the concept of stem regions [MaaRaj90] is applied to the problem of necessary and nonconflicting assignment identification to reduce both the memory and processing time required to find them in an implementation-independent fashion.

The goal of this research is to reduce or eliminate backtracking during test generation for any target fault in any circuit using algorithmic techniques rather than heuristics. To this aim, necessary and nonconflicting assignments are extracted iteratively until either the fault is tested, proven to be untestable, or no more algorithmic assignments can be found, at which point an arbitrary assignment (branch decision) is made. By making as few arbitrary assignments as possible through putting off branching, the potential for backtracking is reduced.

1.2 Outline of dissertation

The body of this thesis is divided into six chapters, as follows:

Chapter 2 reviews developments in test pattern generation over the last two decades in light of the work presented in this thesis. The review focuses on four topics: logic systems, identification of necessary assignments, identification of nonconflicting assignments, and deterministic test pattern generation algorithms.

Chapter 3 reviews the theory of images and inverse images of set functions, the basis of the algorithms to identify necessary and nonconflicting assignments presented in this thesis. Although the concepts of images and inverse images themselves are not new, the application to deterministic test pattern generation is novel. Throughout this thesis, the discussion and examples use a 16-valued algebra to describe the steps of a test pattern generation algorithm. However, the definitions and theorems developed are independent of the logic system in use, and are valid for any other algebra as well, including conventional 5, 9, and 11-valued systems. Using a different logic system enhances or restricts the ability of the algorithm to distinguish cases which arise during test pattern generation, but does not change the nature of the problem—the underlying theory is valid in all cases.

Chapter 4 generalizes and formalizes necessary assignment identification using the concept of *reduction lists*. The work is unifying in the sense that all other proposed necessary assignment identification techniques, including classical implication, dominator identification and "learning" are special cases of this general concept.

Chapter 5 addresses issues relating to the efficient implementation of the algorithms to identify necessary assignments. The use of structural properties of the circuit under test to reduce the computation and memory required to identify necessary assignments is discussed.

A generalized theory of Boolean function monotonicity is developed in chapter 6 and an algorithm to compute tendency lists, from which nonconflicting assignments are identified, is presented.

Finally, a new test pattern generation algorithm, QUEST, which exploits necessary and nonconflicting assignments is described and experimental results obtained by the algorithm when run on a variety of benchmark circuits are presented in chapter 7

Chapter 8 concludes the dissertation.

1.3 Notation conventions

Throughout this thesis, primary inputs and gates are labeled with capital letters; fanout branches are labeled serially with lower case letters corresponding to their stem. For non-stem circuit nodes, gates are referred to by their label (upper case letters) whereas their output lines are referred to by the corresponding lower case letters.

The stuck-at fault model, used throughout this thesis, assumes that faults in the network are represented by a line (or lines) in the circuit under test which permanently propagate a constant logic value—0 for a "stuck-at zero" (s_0) fault and 1 for a "stuck-at one" (s_1) fault—regardless of the signal applied to the line. The fault "line l stuck-at 1" is represented by "l_{s_1}"; fault "l stuck-at 0" is represented by "l_{s_0}". Multiple stuck-at faults are represented by the list of their component single stuck-at faults.

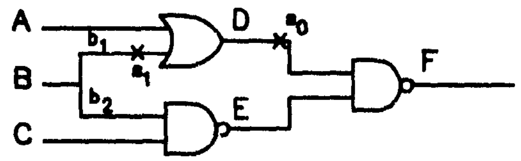


Figure 1.1 Notation conventions

Example 1.1: The 2-input *MUX* in Fig. 1.1 illustrates the notation conventions used in this thesis. Two faults are present in the network: $b1_{s_1}$ and d_{s_0}.

1.4 Publication history

The material contained in the chapters of this thesis discussing the 16-valued logic system for test generation (chapter 3), the identification of necessary assignments (chapter 4), and the experimental results obtained by the QUEST test pattern generation algorithm (section 7.3) have been published in the *Proceedings of the 1990 International Test Conference* [RajCox90]. A comprehensive paper discussing necessary and nonconflicting assignment identification, properties relating to an efficient implementation, and the QUEST algorithm has been submitted to *IEEE Transactions on Computer-Aided Design* [CoxRaj91] and is currently under review.

In addition, several papers discussing the application of a 16-valued logic system to the problems of multiple fault coverage analysis for test pattern generation and failure diagnosis have been published [RajCox86a, RajCox86b, CoxRaj87], of which the most complete discussion is found in [CoxRaj88]. The logic system used is isomorphic to the one presented in chapter 3. The algorithm to identify dominators described in section 7.2.2 was first presented in [CoxRaj87].

Chapter 2 An overview of algorithmic test pattern generation

Test pattern generation has been studied widely for the past three decades or more. Since Roth's classic paper [Roth66], many test pattern generation algorithms have been proposed. In this chapter, key developments in algorithmic test pattern generation for combinational circuits are reviewed from the perspective of their relation to the ideas presented in this thesis: the identification of necessary and nonconflicting assignments.

The choice of logic system (or alphabet) used by a test pattern generation algorithm has a major impact on its organization and efficiency. Therefore, it is necessary to review the logic systems for test generation which have been proposed before the algorithms themselves can be discussed. Section 2.1 reviews conventional 5 [Roth66], 9 [Muth76], and 11-valued [Cheng88] logic systems. In order to take advantage of formal concepts developed for Boolean algebras, a 16-valued logic system [Akers76, Raj88] is used by the algorithms for combinational test pattern generation presented in this thesis. The benefits of a 16-valued system are demonstrated through examples of faults which are not properly handled by other logic systems.

The algorithmic assignment identification techniques used by various test pattern generation algorithms are reviewed in sections 2.2 and 2.3, while the heuristics used by the algorithms are ignored. The categorization of assignments made during test pattern generation into necessary, nonconflicting, and arbitrary assignments and methods used for their identification is the basis for the differentiation between test generation algorithms found in section 2.4.

2.1 Logic systems for test generation

The two-element Boolean algebra $B_2^1 = \{0,1\}$ is widely used to analyse switching circuits, and is sufficiently precise to describe the behaviour of a fault-free combinational circuit. However, in order to describe the behavior of a possibly faulty circuit, a four-element Boolean algebra, $B_2^2 = \{0(0), 0(1), 1(0), 1(1)\}$, where $a(b)$ indicates that the response in the fault-free circuit is a and in the faulty circuit is b, is required. Using the D-symbols [Roth66], $B_2^2 = \{0, \overline{D}, D, 1\}$. The function of a 2-input gate is described as a mapping $B_2^2 \times B_2^2 \rightarrow B_2^2$. The functions *AND* and *OR* are shown in Table 2.1.

	0	$\overline{D}$	D	1
0	0	0	0	0
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$
D	0	0	D	D
1	0	$\overline{D}$	D	1

a) *AND* gate

	0	$\overline{D}$	D	1
0	0	$\overline{D}$	D	1
$\overline{D}$	$\overline{D}$	$\overline{D}$	1	1
D	D	1	D	1
1	1	1	1	1

b) *OR* gate

Table 2.1 Gate functions: $B_2^2 \times B_2^2 \rightarrow B_2^2$

The symbols 0, 1, D, and $\overline{D}$ describe the logic values at nodes in potentially faulty circuits in response to an applied input vector. During the course of test generation, however, these values can appear at circuit nodes in various combinations. For example, each primary input may be assigned to either 0 or 1 in the final test vector which is generated, but before the test pattern is determined, it is not known to which of the possible values it will be assigned. Logic systems differ in the way they represent the combinations of signal values which arise during test generation and in their ability to distinguish between them.

The use of an appropriate algebra can aid the test generation algorithm; similarly, an inappropriate algebra can hinder it. Comparisons between algebras typically focus on the number of elements each contains, the space required to store circuit values, and the time required to manipulate them. A better comparison is the ability of the logic system to resolve circuit values during test pattern generation. Increased resolution may reduce the amount of branching and backtracking performed by the test generation algorithm, reducing both CPU time and storage requirements despite using more values.

2.1.1 5-valued logic system

	0	$\overline{D}$	D	1	X
0	0	0	0	0	0
$\overline{D}$	0	$\overline{D}$	0	$\overline{D}$	X
D	0	0	D	D	X
1	0	$\overline{D}$	D	1	X
X	0	X	X	X	X

a) *AND* gate

	0	$\overline{D}$	D	1	X
0	0	$\overline{D}$	D	1	X
$\overline{D}$	$\overline{D}$	$\overline{D}$	1	1	X
D	D	1	D	1	X
1	1	1	1	1	1
X	X	X	X	1	X

b) *OR* gate

Table 2.2 Gate functions in a 5-valued algebra

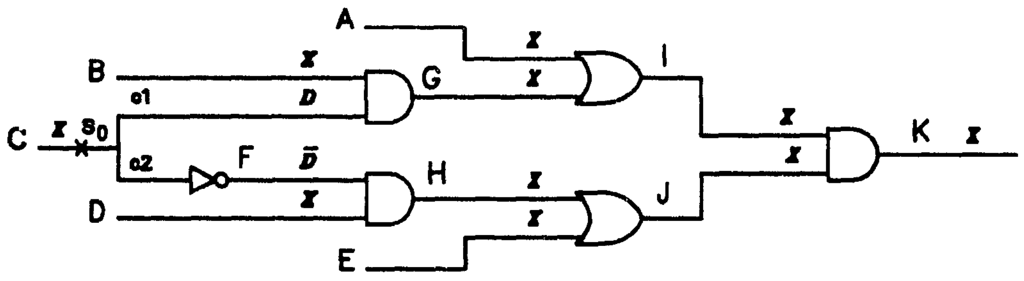


Figure 2.1 Test pattern generation using a 5-valued alphabet

In the 5-valued alphabet $A5$, the basic symbols 0, 1, D, and $\overline{D}$ are each represented individually, while the combinations of values are all represented by the same symbol, X (unknown). The 5-valued alphabet, $A5 = \{0, 1, D, \overline{D}, X\}$, has been used in many ATPG algorithms [Roth66, FujShi83, Goel81, SchAut89]. Table 2.2 illustrates the function of 2-input *AND* and *OR* gates, giving the output value for each possible combination of input values. When *targeting* (attempting to generate a test for) C_{s_0} in the circuit from Fig. 2.1, most circuit values are quickly determined to be X. Aside from $C = 1$, required to sensitize the fault, no necessary assignments can be identified. Several arbitrary branch decisions, each of which may or may not lead to a backtrack, must be made before a test vector can be found.

	00	10	01	11	$x0$	$0x$	$x1$	$1x$	xx
00	00	00	00	00	00	00	00	00	00
10	00	10	00	10	$x0$	00	$x0$	10	$x0$
01	00	00	01	01	00	$0x$	01	$0x$	$0x$
11	00	10	01	11	$x0$	$0x$	$x1$	$1x$	xx
$x0$	00	$x0$	00	$x0$	$x0$	00	$x0$	$x0$	$x0$
$0x$	00	00	$0x$	$0x$	00	$0x$	$0x$	$0x$	$0x$
$x1$	00	$x0$	01	$x1$	$x0$	$0x$	$x1$	xx	xx
$1x$	00	10	$0x$	$1x$	$x0$	$0x$	xx	$1x$	xx
xx	00	$x0$	$0x$	xx	$x0$	$0x$	xx	xx	xx

a) *AND* gate

	00	10	01	11	$x0$	$0x$	$x1$	$1x$	xx
00	00	10	01	11	$x0$	$0x$	$x1$	$1x$	xx
10	10	10	11	10	10	$1x$	$x1$	$1x$	xx
01	01	11	01	11	$x1$	01	$x1$	11	$x1$
11	11	11	11	11	11	11	11	11	11
$x0$	$x0$	10	$x1$	11	$x0$	xx	$x1$	$1x$	xx
$0x$	$0x$	$1x$	01	11	xx	$0x$	$x1$	$1x$	xx
$x1$	$x1$	11	$x1$	11	$x1$	$x1$	$x1$	11	$x1$
$1x$	$1x$	$1x$	11	11	10	$1x$	$1x$	$1x$	$1x$
xx	xx	$1x$	xx	11	xx	xx	$x1$	$1x$	xx

b) *OR* gate

Table 2.3 Gate functions in a 9-valued algebra

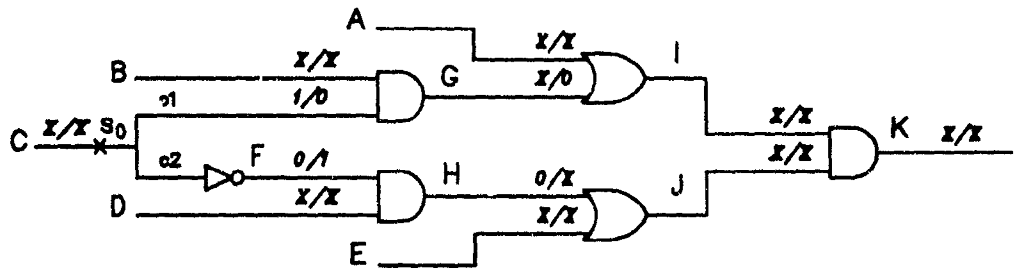


Figure 2.2 Test pattern generation using a 9-valued alphabet

2.1.2 9-valued logic system

The 9-valued alphabet $A9 = \{00, 11, 10, 01, x0, 0x, 1x, x1, xx\}$, where ab indicates the value in the fault-free circuit is a and in the faulty circuit is b and "x" indicates unknown—the value in the fault-free (faulty) circuit may be either 0 or 1[1]—has also been proposed for test pattern generation (Table 2 3) [Muth76]. Compared to $A5$, $A9$ has a unique symbol to represent four additional combinations of signal values. A test generator employing $A9$ [ChaDonÖzg78, JaMoChHa89, KirMer87] encounters a similar problem when generating a test for C_{s_0} (Fig. 2.2) as do those which use $A5$· no necessary assignments can be identified, and several arbitrary branches must be made before a test pattern can be found

[1] Note that the x symbol used by $A9$ is not the same as the X used by $A5$ In $A9$ x denotes that the value in the fault-free (faulty) machine is unknown, independent of the value in the faulty (fault free) machine, whereas in $A5$, "X" denotes that the combination of possible logic values carried by the node in the potentially faulty circuit is unknown—see Table 2 6

	00*E*	10*D*	01*D*	11*E*	*x*0*U*	0*xU*	*x*1*U*	1*xU*	*xxD*	*xxE*	*xxU*
00*E*	00*E*	00*E*	00*E*	00*E*	00*E*	00*E*	00*E*	00*E*	00*E*	00*E*	00*E*
10*D*	00*E*	10*D*	00*E*	10*D*	*x*0*U*	00*E*	*x*0*U*	10*D*	*x*0*U*	*x*0*U*	*x*0*U*
01*D*	00*E*	00*E*	01*D*	01*D*	00*E*	0*xU*	01*D*	0*xU*	0*xU*	0*xU*	0*xU*
11*E*	00*E*	10*D*	01*D*	11*E*	*x*0*U*	0*xU*	*x*1*U*	1*xU*	*xxD*	*xxE*	*xxU*
*x*0*U*	00*E*	*x*0*U*	00*E*	*x*0*U*	*x*0*U*	00*E*	*x*0*U*	*x*0*U*	*x*0*U*	*x*0*U*	*x*0*U*
0*xU*	00*E*	00*E*	0*xU*	0*xU*	00*E*	0*xU*	0*xU*	0*xU*	0*xU*	0*xU*	0*xU*
*x*1*U*	00*E*	*x*0*U*	01*D*	*x*1*U*	*x*0*U*	0*xU*	*x*1*U*	*xxU*	*xxU*	*xxU*	*xxU*
1*xU*	00*E*	10*D*	0*xU*	1*xU*	*x*0*U*	0*xU*	*xxU*	1*xU*	*xxU*	*xxU*	*xxU*
xxD	00*E*	*x*0*U*	0*xU*	*xxD*	*x*0*U*	0*xU*	*xxU*	*xxU*	*xxU*	*xxU*	*xxU*
xxE	00*E*	*x*0*U*	0*xU*	*xxE*	*x*0*U*	0*xU*	*xxU*	*xxU*	*xxU*	*xxE*	*xxU*
xxU	00*E*	*x*0*U*	0*xU*	*xxU*	*x*0*U*	0*xU*	*xxU*	*xxU*	*xxU*	*xxU*	*xxU*

a) *AND* gate

	00*E*	10*D*	01*D*	11*E*	*x*0*U*	0*xU*	*x*1*U*	1*xU*	*xxD*	*xxE*	*xxU*
00*E*	00*E*	10*D*	01*D*	11*E*	*x*0*U*	0*xU*	*x*1*U*	1*xU*	*xxD*	*xxE*	*xxE*
10*D*	10*D*	10*D*	11*E*	11*E*	10*D*	1*xU*	11*E*	1*xU*	1*xU*	1*xU*	1*xU*
01*D*	01*D*	11*E*	01*D*	11*E*	*x*1*U*	01*D*	*x*1*U*	11*E*	*x*1*U*	*x*1*U*	*x*1*U*
11*E*	11*E*	11*E*	11*E*	11*E*	11*E*	11*E*	11*E*	11*E*	11*E*	11*E*	11*E*
*x*0*U*	*x*0*U*	10*D*	*x*1*U*	11*E*	*x*0*U*	*xxU*	*x*1*U*	1*xU*	*xxU*	*xxU*	*xxU*
0*xU*	0*xU*	1*xU*	01*D*	11*E*	*xxU*	0*xU*	*x*1*U*	1*xU*	*xxU*	*xxU*	*xxU*
*x*1*U*	*x*1*U*	11*E*	*x*1*U*	11*E*	*x*1*U*	*x*1*U*	11*E*	11*E*	*x*1*U*	*x*1*U*	*x*1*U*
1*xU*	1*xU*	1*xU*	11*E*	11*E*	1*xU*	1*xU*	11*E*	1*xU*	1*xU*	1*xU*	1*xU*
xxD	*xxD*	1*xU*	*x*1*U*	11*E*	*xxU*	*xxU*	*x*1*U*	1*xU*	*xxU*	*xxU*	*xxU*
xxE	*xxE*	1*xU*	*x*1*U*	11*E*	*xxU*	*xxU*	*x*1*U*	1*xU*	*xxU*	*xxE*	*xxU*
xxU	*xxU*	1*xU*	*x*1*U*	11*E*	*xxU*	*xxU*	*x*1*U*	1*xU*	*xxU*	*xxU*	*xxU*

b) *OR* gate

Table 2.4 Gate functions in an 11-valued algebra

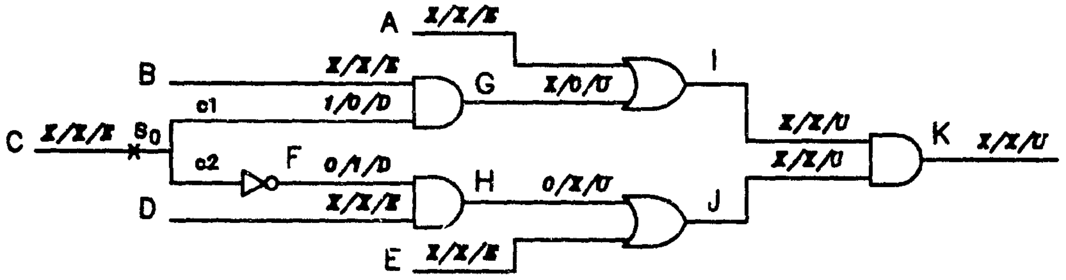


Figure 2.3 Test pattern generation using the SPLIT circuit model

2.1.3 11-valued logic system

As in *A*9, circuit values in the good and faulty machines are treated separately in the 11-valued logic system for test generation proposed in [Cheng88]—each value can be 0, 1, or x (either 0 or 1), independent of the value in the other machine. In addition, the relation between the values in the good and faulty machine is recorded—the values may be Equivalent, Different, or the relation between them may be Unknown. Signal values in $A11$ are identified by the triple GFR, where G is the value in the good machine, F is the value in the faulty machine, and R is the relation between the values in the good and faulty machines. For example, the value xxE means that the values in the good and faulty machine are unknown—the values could be either 0 or 1—but are the same—either both are zero or both are 1. Circuit values in the 11-valued logic system are taken from the set $A11 = \{00E, 11E, xxE, 10D, 01D, xxD, 0xU, x0U, 1xU, x1U, xxU\}$.

Compared to $A9$, $A11$ is able to distinguish the values xxE and xxD (represented by $\{0,1\}$ and $\{D,\overline{D}\}$ in the 16-valued alphabet discussed below), both of which are represented by xx in $A9$. Distinguishing these two values is particularly useful in circuits containing *XOR* gates. However, $A11$ does not distinguish the value combinations $\{0,1,D\}$, $\{0,1,\overline{D}\}$, $\{0,D,\overline{D}\}$, $\{D,\overline{D},1\}$, and $\{0,1,D,\overline{D}\}$ of B_{16}, all of which are represented by xxU. The inability of the model to resolve these values may lead to unnecessary branching and backtracking during test pattern generation. For example, in Fig 2.3, $A11$ is unable to determine whether D, $\overline{D}$, or both can be observed on the primary output of the circuit, and so must make several arbitrary assignments before a test pattern can be found.

2.1.4 16-valued logic system

As a response to an applied input vector, each node in the potentially faulty circuit will carry one of the four possible values from B_2^2. However, during test generation for a particular fault, the final test pattern is not known. Any of the 16 subsets of the set $\{0,1,D,\overline{D}\}$ is a possible node value during test generation;[2] therefore, a complete alphabet for test pattern generation contains 16 values.[3]

Since the subsets of B_2^2 are used to represent the sets of possible values which arise at network nodes during test generation, it is natural to introduce the power set $P(B_2^2)$

[2] The empty set, $\{\}$, indicating inconsistency, is one of the possible assignments—no test pattern exists with the current set of assignments

[3] The general mathematical theory which leads to the 16-valued alphabet is presented in chapter 3

of B_2^2 to represent them. The power set of the basic D symbols has been used by Akers for test generation [Akers76]; as it is a Boolean algebra, it is isomorphic to the 16-valued system used in [CoxRaj88, Raj88] for fault diagnosis. Three possible codings of $P(B_2^2)$ are shown in Table 2.5: the elements of $P(B_2^2)$ themselves, natural numbers from 0 to 15 (B_{16} in Table 2.5), and bitwise encoded quadruples describing the presence or absence of elements of B_2^2 (B_2^4 in Table 2.5).

B_{16}	$P(B_2^2)$	B_2^4
		$x_1 x_D x_{\overline{D}} x_0$
0	$\{\}$	0000
1	$\{0\}$	0001
2	$\{\overline{D}\}$	0010
3	$\{0, \overline{D}\}$	0011
4	$\{D\}$	0100
5	$\{0, D\}$	0101
6	$\{\overline{D}, D\}$	0110
7	$\{0, \overline{D}, D\}$	0111
8	$\{1\}$	1000
9	$\{0, 1\}$	1001
10	$\{\overline{D}, 1\}$	1010
11	$\{0, \overline{D}, 1\}$	1011
12	$\{D, 1\}$	1100
13	$\{0, D, 1\}$	1101
14	$\{\overline{D}, D, 1\}$	1110
15	$\{0, \overline{D}, D, 1\}$	1111

Table 2.5 Three codings of a 16-element alphabet

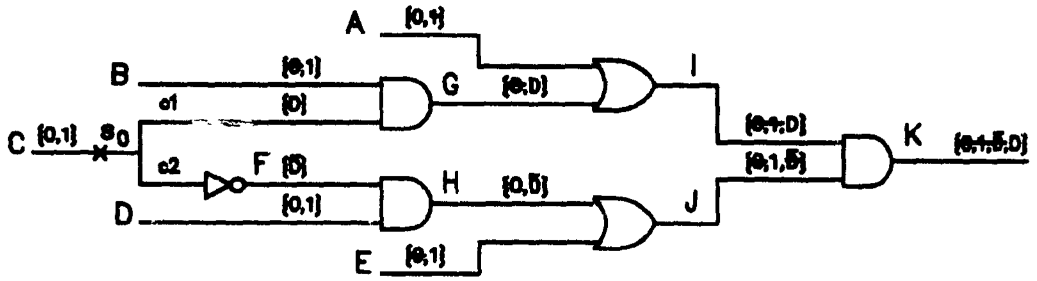


Figure 2.4 Test pattern generation using a 16-valued alphabet

During test generation for C_{s_0} in Fig. 2.4, the 16-valued logic system is able to identify that assignments $A = \{0\}$, $B = C = \{1\}$ are necessary in order to observe $\{D\}$ at the circuit output. Similarly, in order to observe $\{\overline{D}\}$ at the output, assignments $A = D = \{1\}$, $E = \{0\}$ are necessary. In both cases, a test is found with no arbitrary branching. As shown in Figs. 2.1–2.3, the 5, 9, and 11-valued logic systems are unable to determine if D, $\overline{D}$, or both can be propagated to line L. Therefore, test generation systems using these algebras cannot reason about necessary conditions for fault effect observation, and are forced to make several arbitrary branches, each of which may lead to a backtrack.

2.1.5 Comparison between logic systems

The fundamental difference between the logic systems presented in this section is the number of signal value combinations which arise at circuit nodes during test generation which are distinguishable.

B_{16}	$P(B_2^2)$	$A11$	$A9$	$A5$
0	$\{\}$	—	—	—
1	$\{0\}$	$00E$	00	0
2	$\{\overline{D}\}$	$01D$	01	$\overline{D}$
3	$\{0,\overline{D}\}$	$0xU$	$0x$	X^*
4	$\{D\}$	$10D$	10	D
5	$\{0,D\}$	$x0U$	$x0$	X^*
6	$\{\overline{D},D\}$	xxD	xx^*	X^*
7	$\{0,\overline{D},D\}$	xxU^*	xx^*	X^*
8	$\{1\}$	$11E$	11	1
9	$\{0,1\}$	xxE	xx^*	X^*
10	$\{\overline{D},1\}$	$x1U$	$x1$	X^*
11	$\{0,\overline{D},1\}$	xxU^*	xx^*	X^*
12	$\{D,1\}$	$1xU$	$1x$	X^*
13	$\{0,D,1\}$	xxU^*	xx^*	X^*
14	$\{\overline{D},D,1\}$	xxU^*	xx^*	X^*
15	$\{0,\overline{D},D,1\}$	xxU^*	xx^*	X^*

* indicates values which cannot be distinguished

Table 2.6 Comparison between algebras

Table 2.6 compares the values represented by B_{16} [Raj88, Akers76] with those of $A11$ [Cheng88], $A9$ [Muth76], and $A5$ [Roth66]. There are five elements of B_{16} which cannot be distinguished by $A11$—all are represented by xxU. For example, if a primary output carries value xxU, it is not possible to determine whether D, $\overline{D}$, or both D and $\overline{D}$ can be observed there. Resolution is progressively worse if $A9$ (seven indistinguishable values) or $A5$ (eleven indistinguishable values) is used. Element '0" ($\{\}$) of B_{16} indicates inconsistency, and has no representation in any of the other algebras.

The major advantage of a 16-valued logic system is better resolution of signal values. The value combinations which arise at circuit nodes during test generation are not compacted, but are represented as distinct sets of possible node values. Thus there is less loss of information with a 16-valued logic system than with the logic systems described above.[4] The increased resolution offered by a 16-valued logic system may lead to a more efficient search for a test pattern. In particular, it is possible to identify necessary and nonconflicting assignments in the region of the circuit reached by the fault effect, which is not possible using $A5$ and is restricted using $A9$ and $A11$. At the same time, the use of a 16-valued logic system does not increase the time required for logic system computations such as forward propagation and backward implication, as they can be performed in linear time using table lookup, regardless of the logic system in use.

A 16-valued logic system has a number of advantages in addition to increased value resolution. Value justification is the only operation required by a test generation algorithm which uses a 16-valued logic system—it need not perform "D-drive" or "X-path check" operations and need not maintain a "D-frontier". Forward propagation determines the set of possible values which could be carried by each line in the circuit, including those reached by the fault effect. The set of primary outputs to which the fault effect may propagate is known after forward propagation (those outputs whose set of possible values includes D and/or $\overline{D}$). There is no need to distinguish between sensitization and propagation of the fault, as both can be represented in terms of justification. The test generation process

[4] See section 3.3 for discussion of information loss in a 16-valued logic system, and section 4 4 for a technique to recover the lost information.

begins with the initial set of *justification points* (node/value combinations which must be justified) that the point of the fault must be driven to a value opposite that caused by the fault (sensitization) and the fault effect must propagate to at least one primary output—either D or $\overline{D}$ must be observed on at least one of the outputs whose set of possible values includes D and/or $\overline{D}$ (propagation).

2.2 Necessary assignment identification techniques

In 1966, Roth proposed the D-algorithm [Roth66]. The contribution of this paper was the 5-valued calculus described in the previous section—the *singular cover* describing the forward propagation of the D symbols and encoding the conditions required to justify each symbol at the output of a gate. The assignments which are implied by the values which must be justified are necessary—no test pattern will be found if these implied values are assigned any other way.

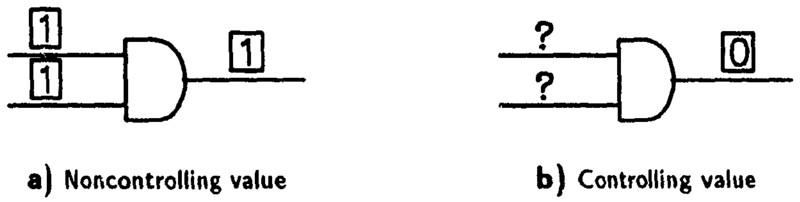


Figure 2.5 Classical backward implication

Example 2.1: Fig. 2.5 illustrates two important cases of classical local implication which arise during test generation. In **a**, the requirement at the output of the *AND* gate is uniquely translated to its inputs as there is only one combination of input values which can produce the required output. On the other hand, in **b** the requirement is not uniquely translated to the inputs, as there are several input combinations which could be used to produce the required output. In arbitrarily choosing to use one of the possible input combinations to produce the required output in **b**, conflicts may occur due to reconvergent fanout Excessive time may be required to generate a test or prove redundancy as, in the worst case, all of the input combinations must be explored.

Classical backward implication is powerful, yet simple: the computation moves from the output of a gate to its inputs. However, since the computation is local, relating to a single logic gate, global information which could aid the test generation process is not recognized. The inputs to the gate whose value must be justified may be correlated in that they are driven by overlapping input cones. Since backward implication does not identify the correlation, the input combination arbitrarily chosen to justify the required output value of the gate may lead to a conflict.

This weakness was partially overcome in the FAN algorithm [FujShi83], which uses structural information about the circuit under test to identify an additional class of necessary assignments which cannot be found by backward implication. When the D-frontier consists of a single gate, the "unique sensitization" step is performed. A topological search is done to determine those lines through which the fault effect must pass in order to appear on any primary output. As FAN uses a 5-valued algebra, it is unable to represent that these lines must be assigned to D and/or $\overline{D}$. Instead, unique sensitization identifies those assignments to noncontrolling values which are necessary to propagate the fault effect from the inputs to the output of the gate at which the fault effect must be observed. In the TOPS algorithm [KirMer87], the unique sensitization assignments of FAN were formalized using the concept of dominators and found using the algorithm in [Tarjan74], originally proposed to find dominators in flow graphs.

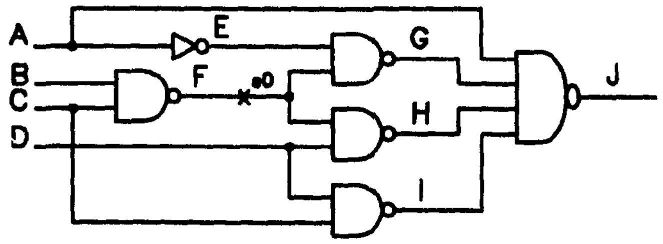


Figure 2.6 Unique sensitization in the FAN algorithm

Example 2.2: No necessary assignments are identified by local implications in generating at test for F_{s_0} in Fig. 2.6. However, in order for the fault effect to be observed at the primary output, it must propagate through J, implying that $A = I = 1$ are necessary assignments. After A is assigned to 1, the fault cannot propagate through G, so $D = 1$ is also a necessary assignment. To justify $I = 1$ after $D = 1$ is assigned, necessary assignment $C = 0$ is identified by local implication, and a test pattern is generated. Note that if a 16-valued logic system for test generation is used in this example then all necessary assignments can be identified by backward implication from justification points $F = 1$ (sensitization) and $J = D$ (propagation), without explicit dominator identification.

Dominator identification is a powerful technique and can be performed efficiently using the linear-time algorithm proposed in [Harel86], rather than the $O(n log(n))$ algorithm from [Tarjan74]. However, dominator analysis can be used to find necessary assignments only in the region of the circuit reached by the fault effect, and not in the rest of the circuit. Further, there may be no necessary assignments to noncontrolling input values even if dominators exist.

The contribution of SOCRATES [SchTriSar88, SchAut89] was to take advantage of information about the function of the circuit under test to identify additional necessary assignments which could not be found using classical implication or dominator analysis. SOCRATES identifies necessary assignments through "learning"—finding the effect of every assignment to every node in the circuit by injecting and determining the implications of each assignment individually (one at a time). If an assignment would make it impossible to justify a required value, then that assignment must be disallowed.

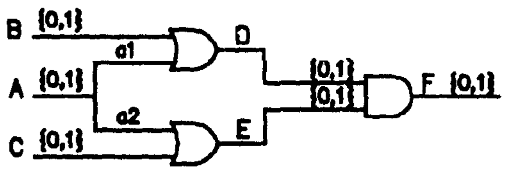


Figure 2.7 Learning in SOCRATES

Example 2.3: The effect of assigning input A to 1 in the circuit from Fig. 2 7 is to produce 1 on D, E, and F. When justifying $F = 0$ (for example, when generating a test for f_{a_1}), no necessary assignments can be identified by local implication Having determined through learning that assigning A to 1 causes $F = 1$, however, $A = 0$ is identified as a necessary assignment.

SOCRATES uses a 5-valued algebra, and thus cannot resolve values or identify necessary assignments through learning in the region of the circuit reached by the fault effect. The effect of assignments to 0 or 1 only can be learned, and only in the region not reachable by the fault. To partially overcome this problem, SOCRATES performs additional processing steps, similar to dominator identification, to identify necessary assignments in the D-region.[5] In [JaMoChHa89] a version of SOCRATES using a 9 rather than 5-valued logic system was presented; due to increased resolution of signal values, that algorithm is able to identify some necessary assignments in the D-region.

A different approach to test pattern generation achieving similar results in terms of necessary assignment identification was taken in the NEMESIS test generation system [Lar89]. The function of the good and faulty circuits are expanded into product-of-sums equations in terms of primary inputs and internal network nodes, the Boolean difference is taken [SeHsBe68], and the resultant equation is solved using techniques developed for Boolean satisfiability problems. The number of terms in a clause of the satisfiability equation is determined by the number of inputs of the gate to which it is related, with n-binate factors and 1 $(n+1)$-ate factor for each n-input gate. The 2SAT problem (all clauses have two or fewer terms) is equivalent to test pattern generation in fanout-free circuits, and can be solved in linear time; the 3SAT (and higher order) problem is $\mathcal{NP}$-complete.

In NEMESIS, necessary assignment identification (termed "nonlocal implication") is performed in a manner similar to learning in SOCRATES by determining the effect of each assignment on the satisfiability equation. Since the NEMESIS algorithm is based on the Boolean difference rather than D-calculus, it does not suffer the disadvantages of 5, 9, or 11-valued alphabets and is able to identify necessary assignments in the region reached by the fault effect.

[5] These techniques and their limitations are discussed more fully in chapter 4.

2.3 Developments in nonconflicting assignment identification

Nonconflicting assignments restrict the space remaining to be searched for a test pattern and cannot cause a backtrack. Thus, they need never be retracted. That is, if there was a test pattern in the search space before the nonconflicting assignment was made, then there is at least one test pattern in the subspace remaining after the assignment has been made. Conversely, if no test pattern exists in the search space after the nonconflicting assignment is made, then there were none in the unrestricted search space either. These assignments are extremely useful because they vastly and irrevocably reduce the space which must be searched in order either to find a test vector or prove the fault untestable.

The guarantee that they will never have to be backtracked is the distinguishing feature of nonconflicting assignment identification. Although analysis of the polarity of reconvergent paths has been used in LAMP2 [AbrKul85] and other test generation systems, this information was used to guide the test generator heuristically rather than algorithmically. Similarly, the multiple backtrace heuristic of FAN [FujShi83] implicitly uses a form of polarity analysis to find desirable arbitrary assignments, although again, these assignments may lead to backtracks.

Properties of nonconflicting assignments were first used in the PLANET test generation system [RobRaj88] to find algorithmic assignments during test pattern generation for crosspoint and delay faults in programmable logic arrays (PLA's). In two-level structures, *monotone pruning* is used to make irrevocable assignments to primary inputs. A *vote* is collected for all inputs connected to those product lines whose value is required in order to generate a test, and those inputs for which the vote is unanimous are assigned. For example, in order to test for a missing device fault at the crosspoint of a product line and an output line, the product line must carry a "1" and all other product lines connected to that output line must carry "0"; in order to produce a "0" on a product line, the desired value of inputs connected to it in true (complemented) form is "0" ("1")

Recently, function montonicity in general multi-level structures has been used to find desirable assignments in a test generation system based on Boolean difference and techniques developed to solve satisfiability problems [Lar89]. Clauses which contain variables that appear in only true or only complemented form in all clauses of the expression are removed, resulting in a drastic pruning of the search space. "Clause removal" is a similar to nonconflicting assignment analysis, except that there is no guarantee that the assignments made in this step can be justified since there is no restriction on the variables assigned. If the assignment used to prune a clause is not justifiable, then a backtrack will occur.

2.4 Deterministic test generation algorithms

Test pattern generation algorithms can be distinguished based on the methods they use to identify necessary, nonconflicting, and arbitrary assignments. Much work in the area of test pattern generation has focussed on finding better heuristics for test generation—methods of identifying branch decisions which are most likely to lead to a test vector or redundancy proof in the least amount of time. This thesis is concerned with algorithmic test pattern generation rather than with heuristics. Key algorithms differentiable on the basis of the algorithmic techniques they employ are discussed in this section and compared to the QUEST test pattern generation algorithm, proposed in this thesis and presented in chapter 7.

Four major test pattern generation algorithms are discussed in this section: the *D*-algorithm, PODEM, FAN, and SOCRATES. The algorithms are presented in the way they were originally proposed by their authors, presuming that they use a 5-valued logic system. As discussed earlier in this chapter, the use of an appropriate logic system has a major impact on the efficiency of the test pattern generation algorithm—for example, the number of necessary assignments it can identify. In addition, the choice of logic system influences the way the algorithm works—the order and type of operations it must perform.

The two basic requirements of any test pattern are that it must sensitize the fault and that the fault effect must be observed on at least one primary output. These requirements are referred to as *sensitization* and *propagation*, and are traditionally treated separately. The sensitization condition requires that the point of the fault be driven to a value opposite that caused by the fault (a line which is s_1 is driven to 0, and vice versa); the propagation condition requires that either D or $\overline{D}$ be observed on at least one primary output.

In addition to backward implication, the D-algorithm [Roth66] introduced the concepts of the "D-frontier" and "D-drive" to monitor and promote the propagation of the fault effect, respectively. The D-frontier consists of those gates whose output value is either D or $\overline{D}$ which drive gates whose output value is X. The D-frontier represents the extent to which the fault propagation requirement has been fulfilled—the limits of the region of the circuit in which it is known that the logic values in the fault-free and faulty circuits are different. D-drive is performed to advance the D-frontier toward primary outputs—to propagate the D symbol one step closer to primary outputs. At each stage, backward implication is performed to identify necessary assignments which appear after an arbitrary assignment is made.

The PODEM algorithm [Goel81] is similar to the D-algorithm, except that arbitrary assignments are made to primary inputs only. Since assignments to primary inputs can always be justified, this restriction has the effect of ensuring that "unjustified nodes" (other than the sensitization requirement at the point of the fault) do not occur in the PODEM algorithm, making it considerably easier to implement than the D-algorithm.

PODEM also introduced the "X-path check" step, which ensures that the fault effect can propagate to some primary output. The X-path check step is performed to identify those gates on the D-frontier from which it is not possible to propagate the fault effect to any primary output. That is, if there are no X-paths from a node on the D-frontier to at least one primary output (*i.e.* all paths are blocked by 0's and/or 1's), that node can be dropped from the D-frontier. If no gates remain on the D-frontier, the fault effect cannot propagate to any primary output, and a backtrack must be performed. If the X-paths are not checked, the test generation system will not recognize that the fault effect cannot propagate until much later, after unnecessary branching and bounding has occurred

The FAN algorithm [FujShi83] is an extension of the PODEM algorithm with two major modifications, intended to address weaknesses of PODEM. The restriction of branch assignments to primary inputs is removed, making FAN more difficult to implement (unjustified values caused by branch assignments to internal circuit nodes must be taken care of). However, by allowing branching at internal nodes, FAN is able to identify conflicts by exploring all node assignments in cases where exhausting input combinations affecting those nodes is not practical. In addition, FAN identifies *dominators*, a class of necessary assignments which cannot be found by conventional backward implication and not identified by either the PODEM or D-algorithms. By identifying additional necessary assignments, FAN is able generate a test or prove that none exist for certain difficult faults with fewer arbitrary branches and backtracks than PODEM or the D-algorithm.

Using the learning techniques described above, SOCRATES [SchTriSar88, SchAut89] is able to identify additional necessary assignments which are not found by either the D-algorithm, PODEM, or FAN, including certain necessary assignments in the region of the circuit reached by the fault effect (see chapter 4). SOCRATES itself is an extension of the FAN algorithm, performing the same steps as FAN, but identifying additional necessary assignments, and thus finding a test pattern or proving redundancy more efficiently.

Compared to other test pattern generation algorithms, the QUEST algorithm has a number of novel features. Certain of these features are due to the use of a 16-valued logic system for test pattern generation and others come about due to the analysis of necessary and nonconflicting assignments.

A 16-valued logic system significantly simplifies the test generation algorithm, as there is no need to perform any of the computation related to D-drive, X-path check, maintenance of the D-frontier, *etc.* In addition, there is no need to distinguish between sensitization and propagation of the fault—the only operation the algorithm must perform is justification of unjustified signal values. It has been noted that, for some untestable faults, the fault effect cannot be propagated to primary outputs, while other faults cannot be sensitized [MinRog89]. In order to prove that these faults are untestable with a minimum of backtracking using conventional algorithms, it is desirable first to attempt propagation or sensitization, respectively. The effectiveness of the strategy depends on the fault which is targeted—either will perform better for some faults and worse for others. Viewed from the perspective of a 16-valued logic system, the question becomes not which strategy to use, but if the traditional approach to test pattern generation is appropriate. Since both sensitization and propagation are required in order to test the fault and both are special cases of line justification, distinguishing between those assignments made for sensitization and those made for propagation is both arbitrary and counterproductive. Both sets of assignments are necessary, and should not be distinguished.

The formulation of the test generation problem as a set of justification points makes it easy to identify necessary and nonconflicting assignments systematically using the techniques presented in chapters 4 and 6, respectively. The systematic identification of necessary assignments replaces a number of operations performed by other algorithms such as conventional backward implication, dominator identification, and learning, while identifying additional necessary assignments which cannot be found using any of those techniques. The identification of nonconflicting assignments is novel and is not performed by any other general test pattern generation algorithm. At the same time, if no necessary or nonconflicting assignments can be identified, then any of the heuristics proposed by other test generation algorithms can be used by QUEST to choose arbitrary assignments.

Chapter 3 Images and inverse images of set functions

The mathematical concepts of images and inverse images of set functions form the foundation on which the methods to identify necessary and nonconflicting assignments developed in this thesis are built. In this chapter, these concepts are defined and applied to algorithmic test pattern generation.

3.1 Images of set functions: forward propagation

During *forward propagation*, the set of possible values at the output of each gate is determined given the sets of possible values at its inputs. The values at the inputs to the gate are assumed to be independent—thus the possible output values are simply those which can be produced by each of the possible combinations of input values.

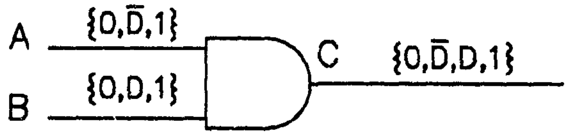


Figure 3.1 Images for a 2-input *AND* gate

Example 3.1: Consider the 2-input *AND* gate shown in Fig. 3.1. The sets of possible values on the inputs, are $\{0, \overline{D}, 1\}$ and $\{0, D, 1\}$ Thus the set of possible values at the output is:

$$\begin{aligned} AND(\{0,\overline{D},D\},\{0,D,1\}) &= \{AND(0,0),AND(0,D),AND(0,1),AND(\overline{D},0),\\ &\quad AND(\overline{D},D),AND(\overline{D},1),AND(1,0),AND(1,D),\\ &\quad AND(1,1)\}\\ &= \{0,\overline{D},D,1\} \end{aligned}$$

This calculation can be formalized using the concept of images of set functions [Raj88]:

Definition 3.1: Let $f : X \times Y \rightarrow Z$ be a function of two variables, and A, B, and C be nonempty subsets of X, Y, and Z, respectively, $A \subseteq X$, $B \subseteq Y$, $C \subseteq Z$. The image $f(A,B)$ of $A \times B$ under f is the set of all images $f(x,y)$ such that $x \in A$ and $y \in B$. Using set builder notation:

$$f(A,B) = \{\, f(x,y) \mid x \in A \text{ and } y \in B \,\}.$$

Using the bitwise encoding B_2^4 from Table 2.5, the function of a gate can be described by four characteristic equations. The equations determine the presence or absence of each possible value at the output of a gate given the sets of possible values of its inputs.

Example 3.2: The characteristic equations for a 2-input *AND* gate with inputs A and B and output C are:

$$\begin{aligned} c_0 &= a_0 + b_0 + a_D b_{\overline{D}} + a_{\overline{D}} b_D\\ c_{\overline{D}} &= a_1 b_{\overline{D}} + a_{\overline{D}} b_1 + a_{\overline{D}} b_{\overline{D}}\\ c_D &= a_1 b_D + a_D b_1 + a_D b_D\\ c_1 &= a_1 b_1. \end{aligned}$$

For example, the equation for c_1 says that 1 is a possible value at the output of the *AND* gate only if 1 is a possible value of both inputs. Characteristic equations with a similar form can be defined for *OR*, *XOR*, *etc.* gates (as well as for larger functional blocks, if desired). The image at the output of a 2-input *AND* gate and a 2-input *OR* gate for all 256 possible combinations of input values is shown in Table 3.1 using the B_{16} coding from

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	1	2	3	1	1	3	3	2	3	2	3	3	3	3	3
3	0	1	3	3	1	1	3	3	3	3	3	3	3	3	3	3
4	0	1	1	1	4	5	5	5	4	5	5	5	4	5	5	5
5	0	1	1	1	5	5	5	5	5	5	5	5	5	5	5	5
6	0	1	3	3	5	5	7	7	6	7	7	7	7	7	7	7
7	0	1	3	3	5	5	7	7	7	7	7	7	7	7	7	7
8	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	0	1	3	3	5	5	7	7	9	9	11	11	13	13	15	15
10	0	1	2	3	5	5	7	7	10	11	10	11	15	15	15	15
11	0	1	3	3	5	5	7	7	11	11	11	11	15	15	15	15
12	0	1	3	3	4	5	7	7	12	13	15	15	12	13	15	15
13	0	1	3	3	5	5	7	7	13	13	15	15	13	13	15	15
14	0	1	3	3	5	5	7	7	14	15	15	15	15	15	15	15
15	0	1	3	3	5	5	7	7	15	15	15	15	15	15	15	15

a) *AND* gate

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0	2	2	2	8	10	10	10	8	10	10	10	8	10	10	10
3	0	3	2	3	12	15	14	15	8	11	10	11	12	15	14	15
4	0	4	8	12	4	4	12	12	8	12	8	12	12	12	12	12
5	0	5	10	15	4	5	14	15	8	13	10	15	12	13	14	15
6	0	6	10	14	12	14	14	14	8	14	10	14	12	14	14	14
7	0	7	10	15	12	15	14	15	8	15	10	15	12	15	14	15
8	0	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
9	0	9	10	11	12	13	14	15	8	9	10	11	12	13	14	15
10	0	10	10	10	8	10	10	10	8	10	10	10	8	10	10	10
11	0	11	10	11	12	15	14	15	8	11	10	11	12	15	14	15
12	0	12	8	12	12	12	12	12	8	12	8	12	12	12	12	12
13	0	13	10	15	12	13	14	15	8	13	10	15	12	13	14	15
14	0	14	10	14	12	14	14	14	8	14	10	14	12	14	14	14
15	0	15	10	15	12	15	14	15	8	15	10	15	12	15	14	15

b) *OR* gate

Table 3.1 Gate functions in a 16-valued logic system: $B_{16} \times B_{16} \rightarrow B_{16}$

Table 2.5. Similar tables can be computed for all other 2-input gate types, at which point forward propagation through individual gates can be performed by table lookup.

The sets of possible output values of all gates in the network can be determined in linear time in a forward levelized selective trace from primary inputs to primary outputs using the sets of possible input values of each gate to find its set of possible output values. Since gate inputs are assumed to be independent, forward propagation through multi-input gates can be performed by forward propagation through a cascade of 2-input gates performing the same function.

The presence of the fault in the circuit under test is accounted for by changing the circuit structure slightly. The point of the fault becomes a new circuit node whose propagated value is assigned to $\{D\}$ or $\{\overline{D}\}$ if the fault is stuck-at zero or stuck-at one, respectively.[6]

[6] Multiple faults can be dealt with similarly there are several fault sites, each of which may propagate a fault effect The propagated value of each individual s_0 (s_1) fault site is $\{D, 0\}$ ($\{\overline{D}, 1\}$), as the only requirement to generate a test for the multiple fault is that the fault effect from at least one of the component single faults be observed on at least one primary output.

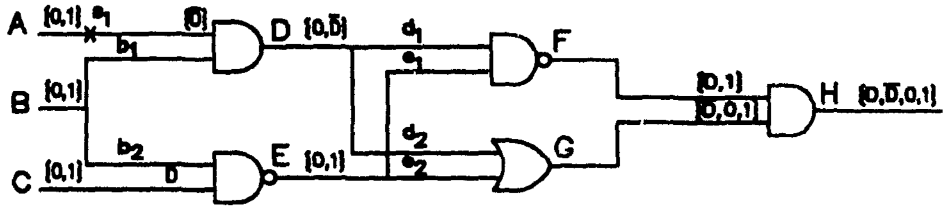


Figure 3.2 Forward propagation in a circuit

Example 3.3: Due to the presence of a s_1 fault on line a in Fig. 3.2, line a propagates $\{\overline{D}\}$ to input 1 of *AND* gate D. The sets of possible signal values of all nodes in the circuit are determined when all primary inputs are assigned to $\{0, 1\}$. For example, the only possible output values of gate D are $\overline{D}$ and 0: $\overline{D}$ if input B is assigned to 1 and 0 if B is assigned to 0. Note that the values propagated by the subcircuit driving the point of the fault (in this case, input A) do not affect values in the subcircuit driven by the point of the fault, as the presence of the fault alters the behavior of the circuit.

3.2 Inverse images of set functions: backward implication

Another operation which is required during test pattern generation is *backward implication*, where the smallest set of values at the inputs of a gate which could be combined to produce a restricted set of values at the output of the gate is determined—the inverse of the image function just described. Backward implication is the generalized analogue of conventional local implication discussed in section 2.2, and is used to derive the necessary input conditions to justify a restricted value at the output of a gate.

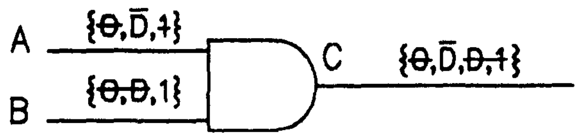


Figure 3.3 Inverse images for a 2-input *AND* gate

Example 3.4: If $\{\overline{D}\}$ must be *justified* at the output of the *AND* gate in Fig 3 3 (signified in the figure by crossing out the alternate output values), then the value of input A must be $\{\overline{D}\}$ and of input B must be $\{1\}$. If either input carried some other value, then the set of possible values at the output of the gate would not include $\overline{D}$ and it would not be possible to justify the required value.

The process of backward implication can be formalized using the concept of inverse images of set functions [Raj88]:

Definition 3.2: Let $f : X \times Y \rightarrow Z$ be a function of two variables, and A, B, and C be nonempty subsets of X, Y, and Z, respectively, $A \subseteq X$, $B \subseteq Y$, and $C \subseteq Z$. The inverse image of C on coordinate X under f, restricted to $A \times B$, which we denote $f^{-X}_{|A\times B}(C)$, is the set of all $x \in A$ such that $f(x, y) \in C$ for some $y \in B$, and similarly for coordinate Y. In set builder notation:

$$f^{-X}_{|A\times B}(C) = \{\, x \in A \mid f(x,y) \in C \text{ for some } y \in B \,\}$$
$$f^{-Y}_{|A\times B}(C) = \{\, y \in B \mid f(x,y) \in C \text{ for some } x \in A \,\}.$$

Example 3.5: In a 2-input *AND* gate with inputs A and B and output C, consider the combinations of input/output values in which $A = \{\overline{D}\}$ participates: *AND*($\overline{D}$,0) = 0, *AND*($\overline{D}$,$\overline{D}$) = $\overline{D}$, *AND*($\overline{D}$,D) = 0, and *AND*($\overline{D}$,1) = $\overline{D}$. In order for $\overline{D}$ to appear in A', the reduced value of input A, it must appear in the original value of input A and some combination of $\overline{D}$ on input A with a value at input B must produce a value which appears in C', the reduced value at the output of the gate. Using the bitwise encoding B_2^4, the inverse image for a gate can be described by four characteristic equations, as was done in the previous section for images. For the *AND* gate, the inverse image A' on input A of the reduced or restricted set C' ($C' \subseteq C$) is:

$$\begin{aligned}
a'_0 &= a_0 c'_0 \\
a'_{\overline{D}} &= a_{\overline{D}}(b_1 c'_{\overline{D}} + b_D c'_0 + b_{\overline{D}} c'_{\overline{D}} + b_0 c'_0) \\
a'_D &= a_D(b_1 c'_D + b_D c'_D + b_{\overline{D}} c'_0 + b_0 c'_0) \\
a'_1 &= a_1(b_1 c'_1 + b_D c'_D + b_{\overline{D}} c'_{\overline{D}} + b_0 c'_0).
\end{aligned}$$

Lemma 3.1: Let the power set of set S, denoted $P(S)$, be the set composed of S and all of its subsets (including the empty set). Given A, B, C, $f^{-X}_{|A\times B}(C)$, $f^{-Y}_{|A\times B}(C)$ defined in $P(S)$, then:

$$\begin{aligned}
f^{-X}_{|A\times B}(C) &= A \cap f^{-X}_{|S\times B}(C) \\
&= A \cap \{\, x \in S \mid f(x,y) \in C \text{ for some } y \in B \,\} \\
f^{-Y}_{|A\times B}(C) &= B \cap f^{-Y}_{|A\times S}(C). \\
&= B \cap \{\, y \in S \mid f(x,y) \in C \text{ for some } x \in A \,\}.
\end{aligned}$$

Proof: From the definition of the power set, for any $X \in P(S)$, $X \subseteq S$ and $X \cap S = X$. For any general set function g with inverse image g^{-1}, $g^{-1}[C \cap D] = g^{-1}[C] \cap g^{-1}[D]$ [HrbJec84]. Thus:

$$\begin{aligned}
f^{-X}_{|A\times B}(C) &= f^{-X}_{|(A\cap S)\times B}(C \cap S) \\
&= \{\, x \mid x \in (A \cap S) \text{ and } f(x,y) \in (C \cap S) \text{ for some } y \in B \,\} \\
&= \{\, x \mid x \in A \text{ and } f(x,y) \in S \text{ for some } y \in B \,\} \cap \\
&\quad \{\, x \mid x \in S \text{ and } f(x,y) \in C \text{ for some } y \in B \,\} \\
&= A \cap \{\, x \mid x \in S \text{ and } f(x,y) \in C \text{ for some } y \in B \,\} \\
&= A \cap f^{-X}_{|S\times B}(C)
\end{aligned}$$

and similarly for $f^{-Y}_{|A\times B}(C)$. ∎

Example 3.6: Using lemma 3.1, Tables 3.2**a** and **b** are computed for 2-input *AND* and *OR* gates, respectively. The table gives the generalized inverse image $f^{-X}_{|\{0,D,\overline{D},1\}\times B}(C)$ given C (column address) and B (row address). For example, using Table 3.2**a** to find the inverse image on input A of output $\{\overline{D}\}$ for the *AND* gate in Fig. 3.3: table$[\{\overline{D}\},\{0,D,1\}] \cap \{0,\overline{D},1\} = \{\overline{D}\} \cap \{0,\overline{D},1\} = \{\overline{D}\}$ (table$[2,13] \cap 11 = 2 \cap 11 = 2$). To increase readability, elements 0 (inconsistency) and 15 (no implication) are replaced by blanks and periods,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1		.														
2		5	10	.												
3		.	10	.												
4		3			12	.										
5		.			12	.										
6		7	10	.	12	.	14	.								
7		.	10	.	12	.	14	.								
8		1	2	3	4	5	6	7	8	9	10	11	12	13	14	.
9		.	2	.	4	.	6	.	8	.	10	.	12	.	14	.
10		5	10	.	4	5	14	.	8	13	10	.	12	13	14	.
11		.	10	.	4	.	14	.	8	.	10	.	12	.	14	.
12		3	2	3	12		14	.	8	11	10	11	12	.	14	.
13		.	2	.	12	.	14	.	8	.	10	.	12	.	14	.
14		7	10	.	12	.	14	.	8	.	10	.	12	.	14	.
15		.	10	.	12	.	14		8	.	10	.	12	.	14	

a) *AND* gate

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1		1	2	3	4	5	6	7	8	9	10	11	12	13	14	
2			3						12		.					
3		1	3	3	4	5	7	7	12	13	.	.	12	13	.	.
4					5				10				.			
5		1	2	3	5	5	7	7	10	11	10	11	.		.	.
6			3		5		7		14		.		.		.	
7		1	3	3	5	5	7	7	14	.	.	.	.	.	.	.
8									.							
9		1	2	3	4	5	6	7	.	.	.	.	.	.	.	.
10			3						.		.					
11		1	3	3	4	5	7	7	.	.	.	.	.	.	.	.
12					5				.				.			
13		1	2	3	5	5	7	7	.	.	.	.	.	.	.	.
14			3		5		7		.		.		.		.	
15		1	3	3	5	5	7	7	.	.	.	.	.	.	.	.

b) *OR* gate

Table 3.2 Inverse images in B_{16}

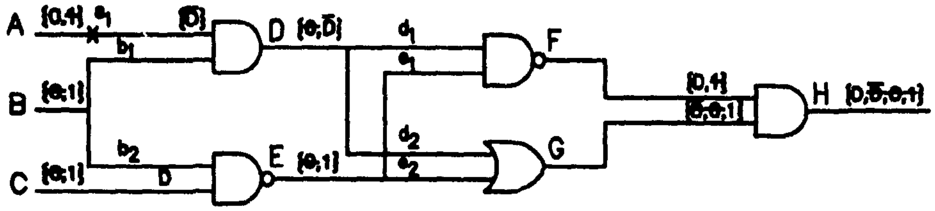


Figure 3.4 Backward implication in the circuit from Fig. 3.2

respectively. Similar tables can be computed for other gate types and backward implication can be performed using table lookup.

Example 3.7: Fig. 3.4 illustrates backward implication for fault a_{s_1}, continued from example 3.3. In order to observe $\{D\}$ at the primary output, line F must carry $\{D\}$ and line G must carry $\{1\}$. In order for F to carry $\{D\}$, line $e1$ must carry $\{1\}$ and line $d1$ $\{\overline{D}\}$, which in turn implies that $b1$ must be $\{1\}$. Finally, if input B is $\{1\}$, then C must be assigned to $\{0\}$ in order to produce $\{1\}$ at G.

3.3 Limitations of set functions

In the definition of the image and inverse image set functions for a gate, the inputs to the gate are assumed to be independent. If the inputs are correlated by a common subcircuit, then the sets of possible values at circuit nodes obtained using images and inverse images may be pessimistic in that not all the values in the sets (and, in particular, not all the combinations of values) can actually be produced.

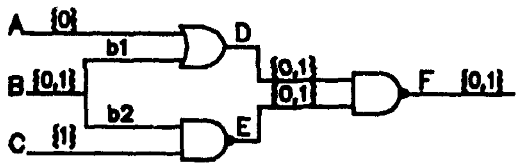
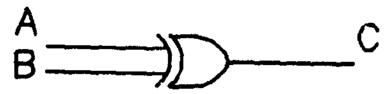


Figure 3.5 Pessimism in the forward implication step

Example 3.8: The circuit of Fig. 3.5 is an implementation of a 2-input *MUX* with data inputs $\overline{A}$, C and select B. The select input can be either 0 or 1, but since both data inputs are 1, the output value will be 1 regardless of which is selected. However, the set of possible values of F found during forward propagation using images of set functions is pessimistic, containing both 0 and 1. The pessimism arises as a result of the implicit assumption made in forward propagation that the values of nodes D and E are independent when, in fact, the are closely related—they cannot be 1 simultaneously.

As images and inverse images were defined for 2-input gates whose inputs are independent, the characteristic equations extracted from multi-gate circuits containing reconvergence may not be exact.

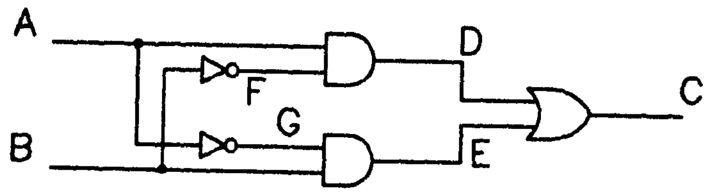
Example 3.9: Fig. 3.6 illustrates that the characteristic equations derived for three different implementations of the *XOR* function are not the same When forward propagation is performed in circuits containing reconvergent fanout, spurious terms in the characteristic equations brought about by reconvergent fanout cause values which cannot actually be produced to appear at network nodes. For example, inspection of the equation for c_0 for



a) 2-input *XOR* gate

$$
\begin{aligned}
c_0 &= a_0b_0 + a_{\overline{D}}b_{\overline{D}} + a_Db_D + a_1b_1\\
c_{\overline{D}} &= a_0b_{\overline{D}} + a_{\overline{D}}b_0 + a_Db_1 + a_1b_D\\
c_D &= a_0b_D + a_{\overline{D}}b_1 + a_Db_0 + a_1b_{\overline{D}}\\
c_1 &= a_0b_1 + a_{\overline{D}}b_D + a_Db_{\overline{D}} + a_1b_0
\end{aligned}
$$

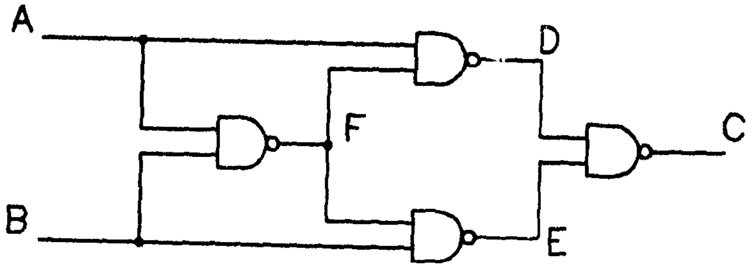
b) Characteristic equations for **a**



c) *AND-OR* implementation

$$
\begin{aligned}
c_0 = {} & a_0b_0 + a_{\overline{D}}b_{\overline{D}} + a_Db_D + a_1b_1 +\\
& a_0a_1 + b_0b_1\\
c_{\overline{D}} = {} & a_0b_{\overline{D}} + a_{\overline{D}}b_0 + a_Db_1 + a_1b_D +\\
& a_{\overline{D}}a_D(b_{\overline{D}} + b_D) + b_{\overline{D}}b_D(a_{\overline{D}} + a_D)\\
c_D = {} & a_0b_D + a_{\overline{D}}b_1 + a_Db_0 + a_1b_{\overline{D}} +\\
& a_{\overline{D}}a_D(b_{\overline{D}} + b_D) + b_{\overline{D}}b_D(a_{\overline{D}} + a_D)\\
c_1 = {} & a_0b_1 + a_{\overline{D}}b_D + a_Db_{\overline{D}} + a_1b_0 +\\
& b_0b_1(a_{\overline{D}} + a_D) + a_0a_1(b_{\overline{D}} + b_D)
\end{aligned}
$$

d) Characteristic equations for **c**



e) Four *NAND* implementation

$$
\begin{aligned}
c_0 = {} & a_0b_0 + a_{\overline{D}}b_{\overline{D}} + a_Db_D + a_1b_1 +\\
& a_0a_1(b_{\overline{D}} + b_D) + b_0b_1(a_{\overline{D}} + a_D)\\
c_{\overline{D}} = {} & a_0b_{\overline{D}} + a_{\overline{D}}b_0 + a_Db_1 + a_1b_D +\\
& a_{\overline{D}}a_D(b_{\overline{D}} + b_D) + b_{\overline{D}}b_D(a_{\overline{D}} + a_D) +\\
& a_0a_{\overline{D}}a_1b_1 + a_1b_0b_{\overline{D}}b_1\\
c_D = {} & a_0b_D + a_{\overline{D}}b_1 + a_Db_0 + a_1b_{\overline{D}} +\\
& a_{\overline{D}}a_D(b_{\overline{D}} + b_D) + b_{\overline{D}}b_D(a_{\overline{D}} + a_D) +\\
& a_0a_Da_1b_1 + a_1b_0b_Db_1\\
c_1 = {} & a_0b_1 + a_{\overline{D}}b_D + a_Db_{\overline{D}} + a_1b_0 +\\
& a_1(a_0 + b_{\overline{D}}b_D) + b_1(b_0 + a_{\overline{D}}a_D)
\end{aligned}
$$

f) Characteristic equations for **e**

Figure 3.6 Three implementations of the exclusive-or function and corresponding characteristic equations

the circuit in Fig. 3.6c indicates that 0 will appear in the output value if input A carries $\{0, 1\}$ and input B $\{D, \overline{D}\}$, $\{D\}$, or $\{\overline{D}\}$. Methods to overcome the problem of reconvergent stem correlation are discussed in section 4.4.

Chapter 4 Reduction list calculation: a method to identify necessary assignments

The key to the necessary assignment identification technique presented in this chapter is the concept of *reduction* which defines the relation between assignments in the circuit under test and the values which must be justified. A general theory of reduction based on the mathematical properties of images and inverse images of set functions is developed. Applied to deterministic test pattern generation, the calculation of *reduction lists* provides a systematic means to identify necessary assignments using set operations and to store this information in a concise form.

Test generation for a particular target fault can be represented by a search tree whose nodes represent the state of the test at each instant and whose edges represent assignments. Leaf nodes represent inconsistent states (backtracks) or valid test patterns. If the fault is untestable, then there are only non-solution leaf nodes; if the fault is testable, then there may be solution and non-solution leaf nodes, depending on the order in which the space is searched. If inconsistent requirements are detected during test generation, a backtrack is performed and the state of the circuit is rolled back to that which existed prior to the most recent arbitrary assignment; the alternate choice (if any) is then explored. The fault is untestable if there are no arbitrary assignments which can be reversed.

Using a 16-valued logic system for test generation, the only operation performed is justification (section 2.1). The state of the test process at any point (the current node in the search tree) is represented by a set of node/value combinations which must be justified in order for the conditions of the test to be satisfied and a test pattern identified. Given an initial set of *justification points*, others can be derived in two ways:

- A necessary assignment to an internal node of the circuit under test is identified and applied. Since the assignment is necessary, there is no need to search the space defined by alternate assignment(s) to the node, as no test patterns exist there.
- An arbitrary decision is made to search the tree in a particular direction and a *branch node* is assigned to a particular value. Unlike a necessary assignment, the decision may not be correct and must be reversed (backtracked) if a conflict is detected.

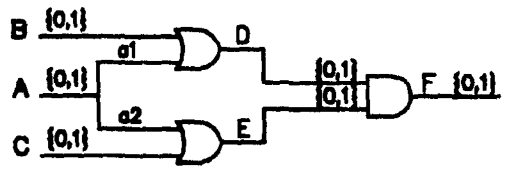


Figure 4.1 Test pattern generation for f_{s_1}

Example 4.1: In order to sensitize a s_1 fault on line f of the circuit shown in Fig. 4.1, the output of the *AND* gate must be driven to $\{0\}$. Since F is a primary output of the circuit, propagating the fault effect is trivial. If input A were assigned to $\{1\}$, then the value of both lines D and E would be $\{1\}$—thus, the *AND* gate output would be $\{1\}$, and it would not be possible to test the fault. Thus, a necessary assignment (and a second justification point) in this example is node A assigned to $\{0\}$.

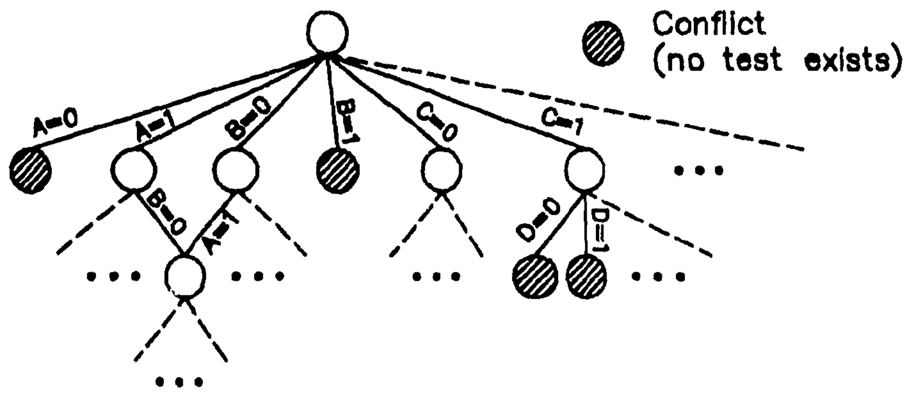


Figure 4.2 The search forest

Necessary assignment identification through reduction list calculation can be understood through the *search forest*—the graph composed of all possible search trees for a particular target fault. Edges represent assignments made in the circuit under test and vertices represent the state of the test generation process after the assignments have been made. From the root, many initial assignments are possible—for example, in the search forest depicted in Fig. 4.2, input A can be assigned to $\{0\}$ or to $\{1\}$. However, input B could equally well be assigned initially, as could any other primary input or internal node of the circuit. Once an initial assignment is made, there are again many choices for the next assignment, and so on. Individual search trees may overlap, as the test generation process lands in the same state (at the same node) after a particular set of assignments is made, regardless of the order in which they are assigned—for example, the same node in the search forest is reached if the test generator assigns first $A = 1$ and then $B = 0$ or first $B = 0$ then $A = 1$.

Reduction list calculation is equivalent to searching one level deep from the current node in the search forest to identify the *first-order necessary assignments*. In other words, from the node in the search forest representing the current state of the test, reduction lists identify those assignments which immediately terminate at non-solution leaf nodes. For example, in Fig. 4.2, the reduction lists would identify that $A_{\{0\}}$ and $B_{\{1\}}$ are non-solution leaf nodes, and thus that $A_{\{1\}}$ and $B_{\{0\}}$ are first-order necessary assignments. On the other hand, $C_{\{0\}}$ would not be identified by the reduction lists as it is a *second-order necessary assignment* (found by searching two levels deep in the search forest)—after $C_{\{1\}}$ is assigned, any assignment to D leads to a conflict. Reduction analysis determines the effect of a single assignment made to a node in the circuit and does not recognize conflicts which appear only after multiple assignments are made. However, applying necessary assignments moves the process into a new node in the search forest, from which necessary assignments which were not recognized from the previous state may be identified.

The result of necessary assignment identification is to create a reordered search tree where non-solution leaf nodes are placed as near the root as possible, preventing the test generator from wasting time searching areas of the tree in which there are no solutions. Ideally, the test generator would identify all necessary assignments rapidly, and thus find a test pattern or prove untestability without backtracks. , /ever, the test generation problem is *NP*-complete [FujToi82] so, in general, test generation algorithms are able to identify only some of the necessary assignments (see section 4.5).

4.1 Reduction lists and necessary assignment identification

Reduction lists capture global information about the function of the circuit under test through local computations of lists at the inputs and outputs of each gate. Gl[illegible]al analysis is achieved through the indexation of the assignments which appear on the reduction lists. Necessary assignment identification through reduction list calculation is unifying in the sense that all other proposed techniques, including backward implication, dominator identification, and learning are special cases of this general method.

Definition 4.1: An *assignment* is a pair consisting of a node identifier and a value. The assignment of node S to value v is denoted S_v.

Definition 4.2: For each line l in the circuit and for each possible value v which it could take, the *reduction list* R_v^l contains those assignments to nodes in the circuit which would cause value v to vanish from the set of possible values of l. An assignment which reduces v at l is called a *reduction assignment for* l_v. An assignment which reduces the required value at a justification point is called a *reduction assignment*.

For each possible value of each line in the circuit, the corresponding reduction list gives the set of reduction assignments for that line and value. Using a 16-valued logic system for test generation, each line in the circuit has four reduction lists associated with it, one for each possible value, 0, 1, D, and $\overline{D}$.

Example 4.2: In the circuit shown in Fig. 4.1, when A is assigned to $\{1\}$, the value of F becomes $\{1\}$; stated equivalently, assigning A to $\{1\}$ causes 0 to vanish from the set of possible values at F. In other words, $A_{\{1\}}$ (read "node A assigned to value $\{1\}$")

reduces $F_{\{0\}}$—$A_{\{1\}}$ is a reduction assignment for $F_{\{0\}}$. Thus, reduction list R_0^F contains assignment $A_{\{1\}}$.

Necessary assignments are derived from the reduction lists at the justification points. If $C_{\{z\}}$ must be justified, then all assignments which appear on reduction list R_z^C must be *eliminated*. That is, if $P_{\{v\}}$ is an assignment which would reduce $C_{\{z\}}$, then value v must be removed from the set of possible values of point P since assigning P to v will cause a conflict (it will no longer be possible to justify $C_{\{z\}}$). If the set of possible values at P remaining after v is removed is empty (represented by $\{\}$ in $P(B_2^2)$ from Table 2.5), then a conflict exists under current assignments and a backtrack must be performed.

Example 4.3: From the previous example, if $F_{\{0\}}$ is a justification point (for example, in generating a test for f_{s_1}), then assignment $A_{\{1\}}$ must be eliminated from the set of possible assignments at A since assigning A to $\{1\}$ will lead to a conflict. Thus, $A_{\{1\}}$ is a reduction assignment and $A_{\{0\}}$ is a necessary assignment.

The set of justification points can be represented by an *AND-OR* graph, whose *AND*-nodes represent assignments all of which must be justified in order to find a test and *OR*-nodes represent assignments at least one of which must be justified. For example, in order to generate a test for a fault, the point of the fault must be driven to a value opposite that caused by the fault (sensitization) and D or $\overline{D}$ must be observed on at least one primary output (propagation). A justification point is satisfied if the forward propagated value of the corresponding gate is the same as the required value. A test pattern is generated when all justification points are satisfied. Conversely, the justification point cannot be satisfied if its forward propagated and required values are disjoint. If an *AND*-node cannot be justified, then no test patterns exist in the space defined by current assignments and a backtrack must be performed to reverse the last arbitrary assignment. The fault is untestable if there are no arbitrary assignments which can be reversed. If an *OR*-node cannot be justified, then it is dropped—if none of the *OR*-nodes can be satisfied, then a backtrack must be performed Lemma 4.1 follows from this reasoning.

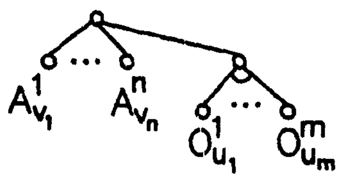


Figure 4.3 The *AND-OR* graph of justification points

Lemma 4.1: Given the set of justification points $\{A^1_{v_1}, \ldots, A^n_{v_n}\}$ all of which must be satisfied (*AND*-nodes in the *AND-OR* graph) and points $\{O^1_{u_1}, \ldots, O^m_{u_m}\}$ at least one of which must be satisfied (*OR*-nodes in the *AND-OR* graph), then the set of reduction assignments is:

$$\left(\bigcup_{i=1}^{n}\left(\bigcap_{\forall x \in v_i} R_x^{A^i}\right)\right) \cup \left(\bigcap_{j=1}^{m}\left(\bigcap_{\forall y \in u_j} R_y^{O^j}\right)\right)$$

Example 4.4: The required value of an *AND* or *OR* node may not be unique. For example, in order to justify $\{D\}$ at the output of an *AND*gate whose input values are $\{0, D\}$ and $\{0, 1, D\}$, it is necessary to justify $\{D\}$ at the first input and $\{1, D\}$ at the other. In order to be a reduction assignment with respect to the second input, an assignment would have to reduce *both* D and 1 there.

Operations *intersection* ($\cap$), *union* ($\cup$) and *difference* ($\setminus$) are performed on the reduction lists.

Example 4.5: Given lists of assignments $L_1 = \{A_{\{0\}}, B_{\{0\}}, C_{\{1\}}\}$, and $L_2 = \{A_{\{1\}}, B_{\{0\}}, C_{\{0,D\}}\}$:

$$L_1 \cap L_2 = \{B_{\{0\}}\}$$
$$L_1 \cup L_2 = \{A_{\{0,1\}}, B_{\{0\}}, C_{\{0,D,1\}}\}$$
$$L_1 \setminus L_2 = \{A_{\{0\}}, C_{\{1\}}\}.$$

In order to reduce a value from the output of a gate, it is necessary to eliminate all combinations of input assignments which could be combined to produce that output value. For examp!-, in order to reduce value z from the set of assignments at the output of a 2-input gate, one or the other (or both) input value from all input combinations which produce z at the output must vanish. An assignment which would cause this to happen belongs to R_z^C.

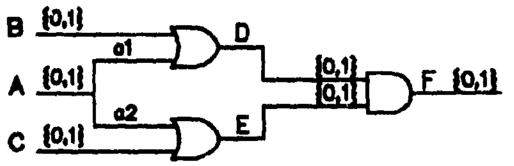
Example 4.6: From the characteristic equations for a 2-input *AND* gate given in example 3.2, value 0 is included in the set of possible values at the output of the gate if 0 is present at either input, or if D at one input can be combined with $\overline{D}$ at the other. Thus, for an assignment to reduce 0 at the output of the gate, it must reduce: 0 at both inputs, either D at input A or $\overline{D}$ at input B, and either $\overline{D}$ at input A or D at input B. This can be represented as the intersection of reduction lists from the inputs of the gate: in order to appear on $\vec{R}_0^C$, an assignment must appear on $\vec{R}_0^A$, $\vec{R}_0^B$, $\vec{R}_D^A$ or $\vec{R}_{\overline{D}}^B$, and $\vec{R}_{\overline{D}}^A$ or $\vec{R}_D^B$. In addition, if the output value of the gate were assigned to some value other than 0, then that would have the effect of reducing 0 at the output of the gate as well. The reduction equations for output C of a 2-input *AND* gate with inputs A and B are:

$$\begin{aligned}
\vec{R}_0^C &= (\vec{R}_0^A \cap \vec{R}_0^B \cap (\vec{R}_{\overline{D}}^A \cup \vec{R}_D^B) \cap (\vec{R}_D^A \cup \vec{R}_{\overline{D}}^B)) \cup \{C_{\{\overline{D},D,1\}}\}\\
\vec{R}_{\overline{D}}^C &= ((\vec{R}_1^A \cup \vec{R}_{\overline{D}}^B) \cap (\vec{R}_{\overline{D}}^A \cup \vec{R}_1^B) \cap (\vec{R}_{\overline{D}}^A \cup \vec{R}_{\overline{D}}^B)) \cup \{C_{\{0,D,1\}}\}\\
\vec{R}_D^C &= ((\vec{R}_1^A \cup \vec{R}_D^B) \cap (\vec{R}_D^A \cup \vec{R}_1^B) \cap (\vec{R}_D^A \cup \vec{R}_D^B)) \cup \{C_{\{0,\overline{D},1\}}\}\\
\vec{R}_1^C &= \vec{R}_1^A \cup \vec{R}_1^B \cup \{C_{\{0,\overline{D},D\}}\}.
\end{aligned}$$

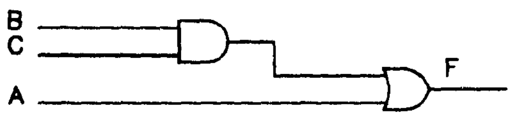
Example 4.7: Similarly, the reduction equations for output C of a 2-input *XOR* gate with inputs A and B are:

$$\begin{aligned}
\vec{R}_0^C &= ((\vec{R}_0^A \cup \vec{R}_0^B) \cap (\vec{R}_{\overline{D}}^A \cup \vec{R}_{\overline{D}}^B) \cap (\vec{R}_D^A \cup \vec{R}_D^B) \cap (\vec{R}_1^A \cup \vec{R}_1^B)) \cup \{C_{\{\overline{D},D,1\}}\}\\
\vec{R}_{\overline{D}}^C &= ((\vec{R}_0^A \cup \vec{R}_{\overline{D}}^B) \cap (\vec{R}_{\overline{D}}^A \cup \vec{R}_0^B) \cap (\vec{R}_D^A \cup \vec{R}_1^B) \cap (\vec{R}_1^A \cup \vec{R}_D^B)) \cup \{C_{\{0,D,1\}}\}\\
\vec{R}_D^C &= ((\vec{R}_0^A \cup \vec{R}_D^B) \cap (\vec{R}_{\overline{D}}^A \cup \vec{R}_1^B) \cap (\vec{R}_D^A \cup \vec{R}_0^B) \cap (\vec{R}_1^A \cup \vec{R}_{\overline{D}}^B)) \cup \{C_{\{0,\overline{D},1\}}\}\\
\vec{R}_1^C &= ((\vec{R}_0^A \cup \vec{R}_1^B) \cap (\vec{R}_{\overline{D}}^A \cup \vec{R}_D^B) \cap (\vec{R}_D^A \cup \vec{R}_{\overline{D}}^B) \quad (\vec{R}_1^A \cup \vec{R}_0^B)) \cup \{C_{\{0,\overline{D},D\}}\}.
\end{aligned}$$

Similar sets of reduction equations can be defined for other gate types, including simple gates such as *OR*, *NAND*, *NOT*, *etc.* as well as for more complex blocks such as *MUX*es, adders, *etc.* More generally, provided that its function can be described in terms



a) Non-minimal example circuit



c) Minimized example circuit

Line	List	Contents
A	$\vec{R}_0^A$	$\{A_{\{1\}}\}$
	$\vec{R}_1^A$	$\{A_{\{0\}}\}$
B	$\vec{R}_0^B$	$\{B_{\{1\}}\}$
	$\vec{R}_1^B$	$\{B_{\{0\}}\}$
C	$\vec{R}_0^C$	$\{C_{\{1\}}\}$
	$\vec{R}_1^C$	$\{C_{\{0\}}\}$
D	$\vec{R}_0^D$	$\{A_{\{1\}}, B_{\{1\}}, D_{\{1\}}\}$
	$\vec{R}_1^D$	$\{D_{\{0\}}\}$
E	$\vec{R}_0^E$	$\{A_{\{1\}}, C_{\{1\}}, E_{\{1\}}\}$
	$\vec{R}_1^E$	$\{E_{\{0\}}\}$
F	$\vec{R}_0^F$	$\{A_{\{1\}}, F_{\{1\}}\}$
	$\vec{R}_1^F$	$\{D_{\{0\}}, E_{\{0\}}, F_{\{0\}}\}$

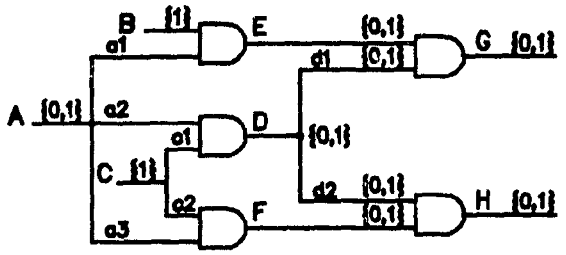
b) Reduction lists for a

Figure 4.4 Reduction list calculation in a circuit

of images and inverse images of set functions, reduction equations can be defined for any module.

Example 4.8: The circuit from Fig. 4.4a, taken from [SchAut89], illustrates reduction list calculation in a network of simple gates. As discussed in example 4.2, $A_{\{1\}}$ appears on $\vec{R}_0^F$. If $F_{\{0\}}$ must be justified during test generation, then $\{1\}$ must be eliminated from stem A. Note that the circuit from Fig. 4.4a is nonminimal, implementing the same function as the circuit shown in Fig. 4.4c. In general, if an assignment to stem S appears on a reduction list at one of its reconvergence gates G when all inputs are assigned to $\{0,1\}$, then S controls the output value of G. The circuit can be redesigned to make S a direct input to G and the intervening unnecessary logic deleted, making the circuit both smaller and easier to test (some reconvergence has been removed).

The reduction lists are completely defined by the forward propagated values of network nodes, which are determined by the injected target fault and the assignments which have been made in the circuit under test. Given a target fault and set of justification points, there is one and only one corresponding set of reduction lists. Example 4.9 illustrates reduction list calculation when circuit values are partially determined.



a) Example circuit

Line	List	Contents
A	$\vec{R}^A_0$	$\{A_{\{1\}}\}$
	$\vec{R}^A_1$	$\{A_{\{0\}}\}$
B	$\vec{R}^B_1$	$\{\}$
C	$\vec{R}^C_1$	$\{\}$
D	$\vec{R}^D_0$	$\{A_{\{1\}}, D_{\{1\}}\}$
	$\vec{R}^D_1$	$\{A_{\{0\}}, D_{\{0\}}\}$
E	$\vec{R}^E_0$	$\{A_{\{1\}}, E_{\{1\}}\}$
	$\vec{R}^E_1$	$\{A_{\{0\}}, E_{\{0\}}\}$
F	$\vec{R}^F_0$	$\{A_{\{1\}}, F_{\{1\}}\}$
	$\vec{R}^F_1$	$\{A_{\{0\}}, F_{\{0\}}\}$
G	$\vec{R}^G_0$	$\{A_{\{1\}}, G_{\{1\}}\}$
	$\vec{R}^G_1$	$\{A_{\{0\}}, D_{\{0\}}, E_{\{0\}}, G_{\{0\}}\}$
H	$\vec{R}^H_0$	$\{A_{\{1\}}, H_{\{1\}}\}$
	$\vec{R}^H_1$	$\{A_{\{0\}}, D_{\{0\}}, F_{\{0\}}, H_{\{0\}}\}$

b) Reduction lists for a)

Figure 4.5 Reduction list calculation with partially determined circuit values

Example 4.9: In the circuit from Fig. 4.5, $A_{\{1\}}$ reduces $G_{\{0\}}$ and $H_{\{0\}}$ if $B = \{1\}$ and $C = \{1\}$ have been determined by other assignments during test pattern generation. However, if $A = \{0,1\}$, $B = \{0,1\}$, and $C = \{0,1\}$ (*i.e.* no assignments have been made), then $A_{\{1\}}$ does not reduce either $G_{\{0\}}$ or $H_{\{0\}}$.

Reduction lists identify necessary assignments in the region of the circuit reached by the D symbol. In addition to other necessary assignments, the reduction lists identify dominators (section 2.2) as nodes whose necessary assignment is to $\{D\}$, $\{\overline{D}\}$, or $\{D, \overline{D}\}$. Static and dynamic learning and structure-based sensitization (described in [SchAut89]) are

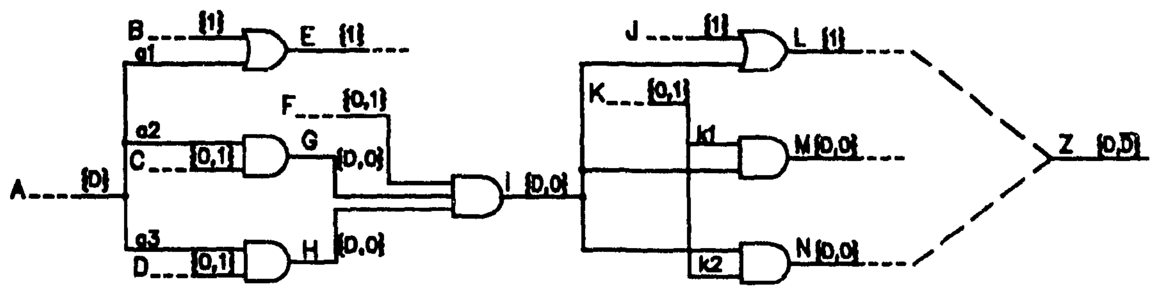


Figure 4.6 Necessary assignments in the region of the D symbol

also special cases of necessary assignments identified by the reduction lists, as illustrated by the following example.

Example 4.10: The subcircuit in Fig. 4.6 illustrates the generality of necessary assignment identification by reduction list calculation. Here, if $B = C = D = J = \{0,1\}$, then $I_{\{0\}}$ reduces $I_{\{D\}}$, $L_{\{D\}}$, $M_{\{D\}}$, $N_{\{D\}}$, and $Z_{\{D,\overline{D}\}}$ (hence I is a dominator with respect to output Z). In addition, $K_{\{0\}}$ reduces $M_{\{D\}}$ and $N_{\{D\}}$. No assignments are necessary, however, as the fault effect does not need to be observed at Z in order to test the fault (the fault effect may propagate to another primary output through gate E). However, if $B_{\{1\}}$ is assigned during test generation, $Z_{\{D,\overline{D}\}}$ becomes a justification point and $I_{\{D\}}$ is a necessary assignment. After $I_{\{D\}}$ is assigned, then $F_{\{1\}}$ can be identified as a necessary assignment by backward implication. Finally, if $J_{\{1\}}$ is assigned during test generation, $K_{\{1\}}$ is a necessary assignment, as $K_{\{0\}}$ reduces both D and $\overline{D}$ at Z.

Assignment analysis through reduction list calculation is equivalent to making assignments in the circuit under test and finding the implication of those assignments on the values of all other nodes in the circuit. Assignments are analysed in parallel using list (set) operations, rather than serially, as is done by the learning techniques proposed in [SchTriSar88, SchAut89]. The key differences between reduction list computation and necessary assignment identification by other techniques are the parallel nature of the calculation and the use of a 16-valued logic system Test generation algorithms employing a

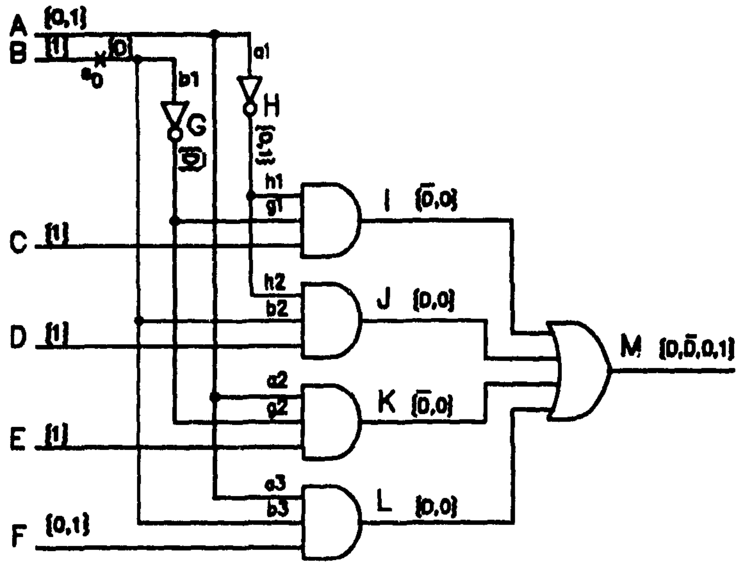


Figure 4.7 Test generation for B_{s_0}

5-valued logic system cannot identify certain necessary assignments in the region of the circuit reached by the fault effect, even if structure-based sensitization techniques are used.

Example 4.11: The 4 × 1 multiplexor from Fig. 4.7 illustrates necessary assignment identification in the region of the network which can be reached from the fault site. Previous assignments have set $C = D = E = \{1\}$. The reduction lists indicate that $A_{\{0\}}$ reduces $L_{\{D\}}$, $K_{\{\overline{D}\}}$, $J_{\{0\}}$, and $I_{\{0\}}$, and therefore reduces both D and $\overline{D}$ at M. Thus $A_{\{1\}}$ is a necessary assignment. After $A_{\{1\}}$ is assigned, backward implication from $M_{\{\overline{D}\}}$ identifies $F_{\{0\}}$ as another necessary assignment. Using a 5, 9, or 11-valued logic system, $A_{\{1\}}$ cannot be identified as a necessary assignment because the values of lines I, J, K, and L cannot be resolved.

To partially overcome problems caused by the poor resolution of a 5-valued logic system, common logic modules (adders, multiplexors, *etc.*) whose logic dependencies are predetermined, have been added to the library of building blocks recognized by a modular version of SOCRATES [SaMaTrSc89]. However, dependencies between modules and in unrecognized structures continue to be overlooked In addition, before each new module can be recognized, implication, unique sensitization, and multiple backtrace procedures which take the signal dependencies of the module into account must be derived manually

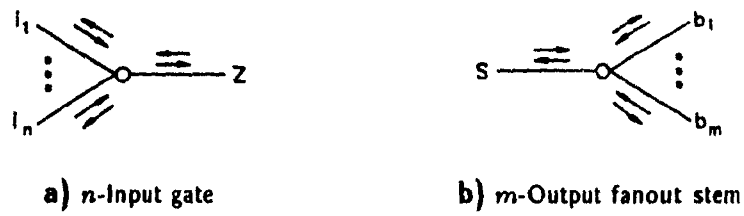


Figure 4.8 Circuit nodes with associated reduction lists

and added to the system. Based on the concepts of images and inverse images of set functions, reduction lists enjoy complete value resolution and are able to identify necessary assignments automatically, without resorting to modular circuit descriptions.

4.2 Logical constraints and propagation of implications

By formulating the test pattern generation problem in terms of images and inverse images of set functions rather than in terms of logical assignments and their implications, the test generation algorithm is able to extract information about the function of the circuit under test. This is important, as the logical constraints imposed by assignments propagate unconditionally in the circuit under test—from inputs toward outputs and from outputs toward inputs. The result of full implication propagation is to determine all implications of each assignment, both forward and backward in the circuit.

The circuit under test can be viewed as a graph, with gates represented by nodes and lines as edges. Primary inputs and outputs are special types of gates, with no inputs and no outputs, respectively. Each edge in the graph has a set of reduction lists associated with it, one reduction list for each possible value of the line. Since the constraints imposed by assignments propagate both forward and backward in the circuit, it is natural to distinguish "forward" ($\overrightarrow{R}$) and "backward" ($\overleftarrow{R}$) components of the reduction lists, shown as directed arrows in Fig. 4.8. Circuit nodes relate the reduction lists of the edges connected to them. For each edge connected to a node, the outward bound reduction list component (forward or backward if the line is an output from or input to the corresponding gate, respectively) is a function of the inward bound reduction list components of all other edges connected to the node. The operation performed when the reduction list components are combined depends on the function of the corresponding gate.

The propagation of reduction lists from the output of a gate to its inputs can be described by a set of reduction equations. In order to reduce a value from an input of a gate, an assignment must reduce all combinations of values of the output and other inputs of the gate in which the reduced input value participates.

Example 4.12: From the inverse-image characteristic equations for a 2-input *AND* gate with inputs A and B and output C given in example 3.5, value 1 remains in the set of possible values at input A of the gate if 0, 1, D, or $\overline{D}$ is present at input B and remains in the implied value of the output. In order for an assignment to reduce 1 at input A, it must simultaneously reduce: either 0 at input B or 0 at output C, either 1 at input A or 1 at output C, either D at input B or D at output C, and either $\overline{D}$ at input B or $\overline{D}$ at output C. These conditions can be stated in terms of the intersection of reduction lists from the output and other input of the gate: in order for an assignment to appear on $\overleftarrow{R}^A_1$, it must appear on $\overrightarrow{R}^B_0$ or $\overleftarrow{R}^C_0$, $\overrightarrow{R}^B_1$ or $\overleftarrow{R}^C_1$, $\overrightarrow{R}^A_D$ or $\overleftarrow{R}^C_D$, and $\overrightarrow{R}^B_{\overline{D}}$ or $\overleftarrow{R}^C_{\overline{D}}$. As in example 4.6, an assignment of input A to a value other than 1 also reduces 1 at the input. The reduction equations for input A of a 2-input *AND* gate are:

$$\overleftarrow{R}^A_0 = \overleftarrow{R}^C_0 \cup \{A_{\{1,\overline{D},D\}}\}$$
$$\overleftarrow{R}^A_{\overline{D}} = ((\overrightarrow{R}^B_1 \cup \overleftarrow{R}^C_{\overline{D}}) \cap (\overrightarrow{R}^B_D \cup \overleftarrow{R}^C_0) \cap (\overrightarrow{R}^B_{\overline{D}} \cup \overleftarrow{R}^C_{\overline{D}}) \cap (\overrightarrow{R}^B_0 \cup \overleftarrow{R}^C_0)) \cup \{A_{\{0,D,1\}}\}$$
$$\overleftarrow{R}^A_D = ((\overrightarrow{R}^B_1 \cup \overleftarrow{R}^C_D) \cap (\overrightarrow{R}^B_D \cup \overleftarrow{R}^C_D) \cap (\overrightarrow{R}^B_{\overline{D}} \cup \overleftarrow{R}^C_0) \cap (\overrightarrow{R}^B_0 \cup \overleftarrow{R}^C_0)) \cup \{A_{\{0,\overline{D},1\}}\}$$
$$\overleftarrow{R}^A_1 = ((\overrightarrow{R}^B_1 \cup \overleftarrow{R}^C_1) \cap (\overrightarrow{R}^B_D \cup \overleftarrow{R}^C_D) \cap (\overrightarrow{R}^B_{\overline{D}} \cup \overleftarrow{R}^C_{\overline{D}}) \cap (\overrightarrow{R}^B_0 \cup \overleftarrow{R}^C_0)) \cup \{A_{\{0,\overline{D},D\}}\}.$$

Example 4.13: Similarly, the reduction equations for input A of a 2-input *XOR* gate are:

$$\overleftarrow{R}^A_0 = ((\overleftarrow{R}^C_0 \cup \overrightarrow{R}^B_0) \cap (\overleftarrow{R}^C_{\overline{D}} \cup \overrightarrow{R}^B_{\overline{D}}) \cap (\overleftarrow{R}^C_D \cup \overrightarrow{R}^B_D) \cap (\overleftarrow{R}^C_1 \cup \overrightarrow{R}^B_1)) \cup \{A_{\{\overline{D},D,1\}}\}$$
$$\overleftarrow{R}^A_{\overline{D}} = ((\overleftarrow{R}^C_0 \cup \overrightarrow{R}^B_{\overline{D}}) \cap (\overleftarrow{R}^C_{\overline{D}} \cup \overrightarrow{R}^B_0) \cap (\overleftarrow{R}^C_D \cup \overrightarrow{R}^B_1) \cap (\overleftarrow{R}^C_1 \cup \overrightarrow{R}^B_D)) \cup \{A_{\{0,D,1\}}\}$$
$$\overleftarrow{R}^A_D = ((\overleftarrow{R}^C_0 \cup \overrightarrow{R}^B_D) \cap (\overleftarrow{R}^C_{\overline{D}} \cup \overrightarrow{R}^B_1) \cap (\overleftarrow{R}^C_D \cup \overrightarrow{R}^B_0) \cap (\overleftarrow{R}^C_1 \cup \overrightarrow{R}^B_{\overline{D}})) \cup \{A_{\{0,\overline{D},1\}}\}$$
$$\overleftarrow{R}^A_1 = ((\overleftarrow{R}^C_0 \cup \overrightarrow{R}^B_1) \cap (\overleftarrow{R}^C_{\overline{D}} \cup \overrightarrow{R}^B_D) \cap (\overleftarrow{R}^C_D \cup \overrightarrow{R}^B_{\overline{D}}) \cap (\overleftarrow{R}^C_1 \cup \overrightarrow{R}^B_0)) \cup \{A_{\{0,\overline{D},D\}}\}.$$

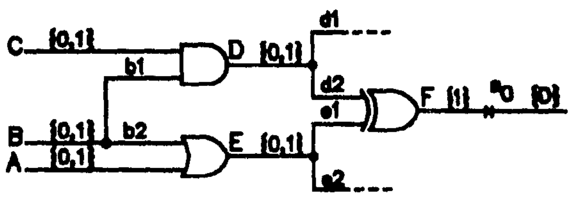


Figure 4.9 Test generation for fault f_{s_0}

Dependencies between circuit nodes may cause logical constraints at one justification point to appear at another. The reduction lists are able to capture these constraints and identify additional necessary assignments.

Example 4.14: In order to test the fault f_{s_0} in the subcircuit shown in Fig. 4.9, $F_{\{1\}}$ must be justified. We note that $E_{\{0\}}$ requires $B_{\{0\}}$, as $B_{\{1\}}$ reduces $E_{\{0\}}$; thus, $\overleftarrow{R}_1^{b2}$ includes $E_{\{0\}}$. Since $E_{\{0\}}$ appears on $\overleftarrow{R}_1^{b2}$, it also appears on $\overrightarrow{R}_1^{b1}$, and thus propagates to stem D and appears on $\overrightarrow{R}_1^{D}$. Since $E_{\{0\}}$ appears on both $\overrightarrow{R}_1^{d2}$ and $\overrightarrow{R}_1^{e1}$, it reduces $F_{\{1\}}$. However, $F_{\{1\}}$ is a justification point—thus $E_{\{1\}}$ is necessary. A similar argument applies to $D_{\{0\}}$, which is also necessary. It is important to observe that an assignment to stem E appears on a reduction list at D despite the fact that D is neither driven by nor drives E.

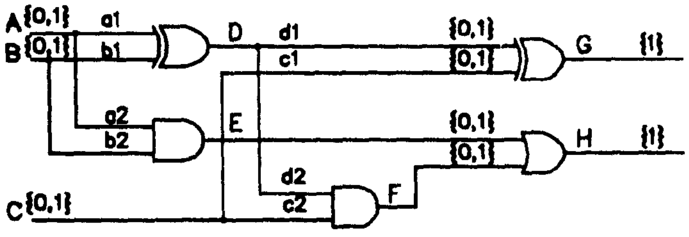


Figure 4.10 Value justification of a full adder

Example 4.15: In justifying $\{1\}$ on both the sum and carry outputs of the full adder from Fig. 4.10, no necessary assignments are identified by backward implication despite that consideration of the function reveals that inputs A, B, and C must all be assigned to $\{1\}$ However, assignment $C_{\{0\}}$ appears on $\overrightarrow{R}_1^{F}$; since $H_{\{1\}}$ must be justified, an assignment which reduces $F_{\{1\}}$ implies that E must carry $\{1\}$ (*i.e.* $C_{\{0\}}$ appears on $\overleftarrow{R}_0^{E}$) Hence $C_{\{0\}}$ requires that $A = B = \{1\}$ (*i.e.* appears on both $\overleftarrow{R}_0^{a2}$ and $\overleftarrow{R}_0^{b2}$), as $E = \{1\}$ is required if $C = \{0\}$ in order to satisfy $H_{\{1\}}$. Thus, $C_{\{0\}}$ appears on $\overrightarrow{R}_1^D$, since it appears on both $\overrightarrow{R}_0^{a1}$ and $\overrightarrow{R}_0^{b1}$. Finally, since $C_{\{0\}}$ appears on $\overrightarrow{R}_1^D$ and $\overrightarrow{R}_1^C$, it appears on $\overrightarrow{R}_1^G$. $G_{\{1\}}$ is a justification point, so $C_{\{0\}}$ is a reduction assignment and $C_{\{1\}}$ is necessary. A similar argument can be made for assignments $A_{\{0\}}$ and $B_{\{0\}}$, both of which are also reduction assignments.

4.3 General theorem of reduction

Reduction lists can be calculated for a general set function f using theorem 4.1.

Theorem 4.1: Let $f : X \times Y \to Z$ be a function, and A, B, and C be nonempty subsets of X, Y, and Z, respectively, $A \subseteq X$, $B \subseteq Y$, $C \subseteq Z$. Let R_x^X for each $x \in A$, R_y^Y for each $y \in B$, be the set of assignments which cause values x, y to vanish from sets A, B at coordinates X, Y, respectively. Then the set of assignments which cause value z to vanish from set C at coordinate Z, denoted by $\overrightarrow{R}_z^Z$, is the intersection of all assignments which cause x and/or y to vanish from sets A and/or B for every $x \in A, y \in B$ such that $f(x,y) = z$. For each $z \in C$:

$$\begin{aligned}\overrightarrow{R}_z^Z &= (Z_C \setminus Z_{\{x\}}) \cup \bigcap_{x \in f^{-X}_{|A\times B}(z)} \left(\overrightarrow{R}_x^X \cup \Big(\bigcap_{y \in f^{-Y}_{|x\times B}(z)} \overrightarrow{R}_y^Y\Big)\right)\\ &= (Z_C \setminus Z_{\{x\}}) \cup \bigcap_{y \in f^{-Y}_{|A\times B}(z)} \left(\overrightarrow{R}_y^Y \cup \Big(\bigcap_{x \in f^{-X}_{|A\times y}(z)} \overrightarrow{R}_x^X\Big)\right).\end{aligned}$$

The set of assignments which cause value x to vanish from set A, denoted by $\overleftarrow{R}_x^A$, is the intersection of all assignments which cause y and/or z to vanish from sets B and/or C, respectively, for every $y \in B, z \in C$ such that $x \in f^{-X}_{|A\times B}(z)$. For each $x \in A$:

$$\overleftarrow{R}_x^X = (X_A \setminus X_{\{x\}}) \cup \bigcap_{z \in f(x,B)} \left(\overleftarrow{R}_z^Z \cup \Big(\bigcap_{y \in f^{-Y}_{|x\times B}(z)} \overrightarrow{R}_y^Y\Big)\right).$$

Similarly, for each $y \in B$:

$$\overleftarrow{R}^{Y}_{y} = (Y_B \setminus Y_{\{y\}}) \cup \bigcap_{z \in f(A,y)} \left(\overleftarrow{R}^{Z}_{z} \cup \left(\bigcap_{x \in f^{X}_{|A \setminus y}(z)} \overrightarrow{R}^{X}_{x} \right) \right).$$

Proof: (By construction) In order to reduce a value at the output of a gate, an assignment must reduce all input combinations which cause that value to appear. To reduce z at output Z, it is sufficient that an assignment reduce all values x at input X which could be combined with some value at input Y to produce z at Z. By definition, the inverse image on X of z given $Y = B$ is that set of values at X. If the assignment does not reduce a value x in that set, then it must reduce all values y at input Y which could be combined with x to produce z at Z—that is, all values in the inverse image on Y of z given that $X = x$. A similar argument can be made starting with all values at input Y which can be combined with a value at X to produce z.

To reduce a value at the input of a gate, an assignment must reduce all input/output combinations in which that value participates. For example, to reduce x at input X, it is sufficient for an assignment to reduce all values z at output Z which can be produced by combining x with a value at input Y—by definition, those values in the image on Z of x given $Y = B$. If the assignment does not reduce some z in that set, then it must reduce all values y at input Y which can be combined with x at input X to produce z at the output—all values in the inverse image on Y of z given $X = x$. Similarly, to reduce y at input Y. ∎

Lemma 4.2: For each fanout branch s_i of stem S carrying value v (Fig. 4.8b),

$$\overrightarrow{R}^{s_i}_{x} = \overrightarrow{R}^{S}_{x} \cup (S_v \setminus S_{\{x\}}) \cup \left(\bigcup_{j \neq i} \overleftarrow{R}^{s_j}_{x} \right) \qquad \text{for each } x \in v$$

For the fanout stem itself,

$$\overleftarrow{R}^{S}_{x} = (S_v \setminus S_{\{x\}}) \cup \left(\bigcup_{i=1}^{n} \overleftarrow{R}^{s_i}_{x} \right) \qquad \text{for each } x \in v$$

Example 4.16: Using theorem 4.1 to derive the reduction equations for an *AND* gate, with inputs $A = \{0,1,D,\overline{D}\}$, $B = \{0,1,D,\overline{D}\}$ and output $C = \{0,1,D,\overline{D}\}$:

$$
\begin{aligned}
\vec{R}^{C}_{0} &= (C_{\{0,\overline{D},D,1\}} \setminus C_{\{0\}}) \cup (\vec{R}^{A}_{0} \cup \overbrace{(\vec{R}^{B}_{0} \cap \vec{R}^{B}_{D} \cap \vec{R}^{B}_{\overline{D}} \cap \vec{R}^{B}_{1})}^{=\phi^{7}}) \cap (\vec{R}^{A}_{\overline{D}} \cup (\vec{R}^{B}_{0} \cap \vec{R}^{B}_{D})) \\
&\quad \cap (\vec{R}^{A}_{D} \cup (\vec{R}^{B}_{0} \cap \vec{R}^{B}_{\overline{D}})) \cap (\vec{R}^{A}_{1} \cup \vec{R}^{B}_{0}) \\
&= (C_{\{\overline{D},D,1\}}) \cup \vec{R}^{A}_{0} \cap (\vec{R}^{A}_{\overline{D}} \cup \vec{R}^{B}_{D}) \cap (\vec{R}^{A}_{D} \cup \vec{R}^{B}_{\overline{D}}) \cup (\vec{R}^{B}_{0} \cup \overbrace{(\vec{R}^{A}_{\overline{D}} \cap \vec{R}^{A}_{D} \cap \vec{R}^{A}_{1})}^{=\phi^{8}}) \\
&= (C_{\{\overline{D},D,1\}}) \cup \vec{R}^{A}_{0} \cap \vec{R}^{B}_{0} \cap (\vec{R}^{A}_{\overline{D}} \cup \vec{R}^{B}_{D}) \cap (\vec{R}^{A}_{D} \cup \vec{R}^{B}_{\overline{D}}) \\
\vec{R}^{C}_{\overline{D}} &= (C_{\{0,D,1\}}) \cup (\vec{R}^{A}_{\overline{D}} \cup (\vec{R}^{B}_{\overline{D}} \cap \vec{R}^{B}_{1})) \cap (\vec{R}^{A}_{1} \cup \vec{R}^{B}_{\overline{D}}) \\
\vec{R}^{C}_{D} &= (C_{\{0,\overline{D},1\}}) \cup ((\vec{R}^{A}_{D} \cup (\vec{R}^{B}_{D} \cap \vec{R}^{B}_{1})) \cap (\vec{R}^{A}_{1} \cup \vec{R}^{B}_{1}) \\
\vec{R}^{C}_{1} &= (C_{\{0,\overline{D},D\}}) \cup \vec{R}^{A}_{1} \cup \vec{R}^{B}_{1}.
\end{aligned}
$$

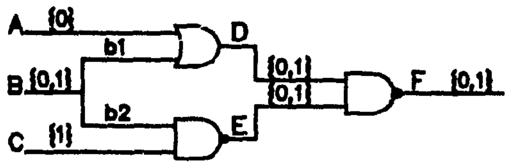
4.4 Stem correlation

The values at the inputs to a gate may be related to each other through the reconvergence of a fanout stem. In this case, the set of values at the output of the gate determined using images of set functions may be pessimistic in that certain values present in the set cannot be obtained. Reduction lists capture the relation between circuit nodes and can be used to eliminate this pessimism, reducing branching and backtracking as well as the total CPU time required to generate a test.

Example 4.17: The circuit of Fig. 4.11 is an implementation of a 2-input *MUX* with select B and inputs $\overline{A}$, C. Since both data inputs are $\{1\}$, output F will be $\{1\}$ regardless of which is selected Unfortunately, the forward propagated value of F erroneously includes both 0 and 1 (see example 3 8) However, reduction list $\vec{R}^{F}_{0} = \{B_{\{0,1\}}\}$ identifies that either assignment to input B reduces 0 at F, indicating that 0 is not an attainable assignment at

[7] Since any such assignment would cause $B = \{\}$

[8] Since any such assignment would cause $A = \{0\}$, and thus $C = \{0\}$—a contradiction



Line	List	Contents
A	$\vec{R}_0^A$	{}
B	$\vec{R}_0^B$ $\vec{R}_1^B$	$\{B_{\{1\}}\}$ $\{B_{\{0\}}\}$
C	$\vec{R}_1^C$	{}
D	$\vec{R}_0^D$ $\vec{R}_1^D$	$\{B_{\{1\}}\}$ $\{B_{\{0\}}\}$
E	$\vec{R}_0^E$ $\vec{R}_1^E$	$\{B_{\{0\}}\}$ $\{B_{\{1\}}\}$
F	$\vec{R}_0^F$ $\vec{R}_1^F$	$\{B_{\{0,1\}}\}$ {}

a) 2-input *MUX* b) Reduction lists for a

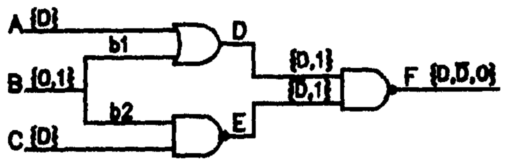
Figure 4.11 Correlation of assignments

line F: the forward propagated value of F can be restricted to $\{1\}$ from $\{0,1\}$, eliminating the pessimism. A *stem correlation* exists for assignment $F_{\{0\}}$ caused by B.

Stem correlation is similar to the identification of uniquely implied signal values proposed in [SchAut89], except that the use of a 16-valued logic system enables the reduction lists to identify stem correlations in the region of the circuit reached by the fault effect. Examples 4.18 and 4.19 illustrate two important cases of stem correlation in the D-region which cannot be identified by 5, 9, or 11-valued logic systems.

Example 4.18: The set of possible values of node F in Fig. 4.12 determined using images of set functions is $\{0, D, \overline{D}\}$. However, as in the previous example, a stem correlation exists for assignment $F_{\{0\}}$ caused by B. The forward propagated value of F can be restricted to $\{D, \overline{D}\}$. No further assignments are required to cause the fault effect to propagate to the output of the *MUX*.

The reduction lists are determined by the current sets of possible values in the circuit under test, taking into account assignments made earlier in the test generation process Stem correlation identified by these reduction lists indicates that it is not possible to both



a) 2-input *MUX*

Line	List	Contents
A	$\vec{R}^{A}_{D}$	{}
B	$\vec{R}^{B}_{0}$	$\{B_{\{1\}}\}$
	$\vec{R}^{B}_{1}$	$\{B_{\{0\}}\}$
C	$\vec{R}^{C}_{D}$	{}
D	$\vec{R}^{D}_{D}$	$\{B_{\{1\}}\}$
	$\vec{R}^{D}_{1}$	$\{B_{\{0\}}\}$
E	$\vec{R}^{E}_{\bar{D}}$	$\{B_{\{0\}}\}$
	$\vec{R}^{E}_{1}$	$\{B_{\{1\}}\}$
F	$\vec{R}^{F}_{0}$	$\{B_{\{0,1\}}\}$
	$\vec{R}^{F}_{\bar{D}}$	$\{B_{\{0\}}\}$
	$\vec{R}^{F}_{D}$	$\{B_{\{1\}}\}$

b) Reduction lists for a)

Figure 4.12 Stem correlation in the D-region—propagation of the fault effect

satisfy the current set of justification points and produce the correlated value. However, if the set of justification points were different, it might be possible to obtain the correlated value.

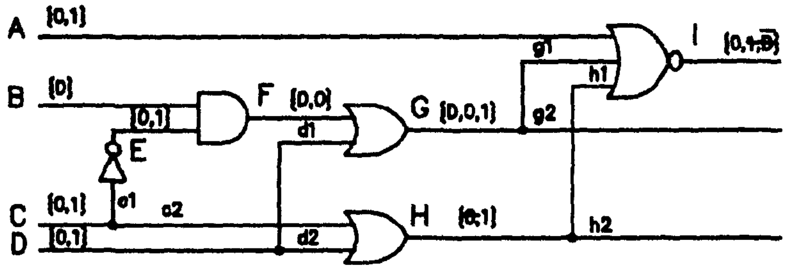


Figure 4.13 Stem correlation in the D-region—no propagation of the fault effect

Example 4.19: Stem correlation can be used to determine that the fault effect cannot propagate to the output of the encoder illustrated in Fig 4.13. Although the value of node G found using images of set functions is $\{D, 0, 1\}$, the reduction lists show that $G_{\{D\}}$ is correlated by stem C—thus, the forward propagated value of G can be restricted to $\{0, 1\}$. In this example, it is not possible to both justify $H_{\{1\}}$ and produce D on the output of gate G. However, if $I_{\{0\}}$ were the only justification point, then D could be produced at the output of gate G by assigning $C = D = \{1\}$.

4.5 Complexity of test pattern generation and the computation of reduction lists

During test generation, node values in the circuit under test are progressively refined as the process converges to a test vector. That is, the cardinality of the sets of possible values of circuit nodes is a monotonically non-increasing function Thus, reduction lists can only grow during test generation—assignments can be added, never deleted—since an assignment must cause a reduction in a more refined system if it caused a reduction previously.[9] Since the total number of assignments which can appear on any reduction list cannot be greater than the number of nodes in the circuit, the reduction lists can be computed in polynomial time using selective trace.

In a circuit containing l lines and n nodes, there are a total of $4l$ reduction lists and $4n$ node assignments. An individual reduction list cannot contain more than $4n$ assignments, and at least one assignment must be added to at least one reduction list in order for further computation to be scheduled. The time required to update a reduction list is proportional to its length. Therefore, the worst case time required to compute the reduction lists is $O(ln^2)$.

As discussed above, reduction analysis is equivalent to searching one level deep in the search forest to identify first-order necessary assignments. Second-order necessary assignments can be identified if double assignments are placed on the reduction lists, rather than single assignments as has been discussed heretofore In general, reduction lists will identify *all* necessary assignments only if *all* assignments are analysed (single, double, triple, *etc.*). The exponential complexity of test generation arises through the exponential number of assignment combinations which must be considered in order to identify all

[9] The reduction lists may decrease in size on a backtrack, as the test generation process is rolled back from a later stage to an earlier one where node values were less refined

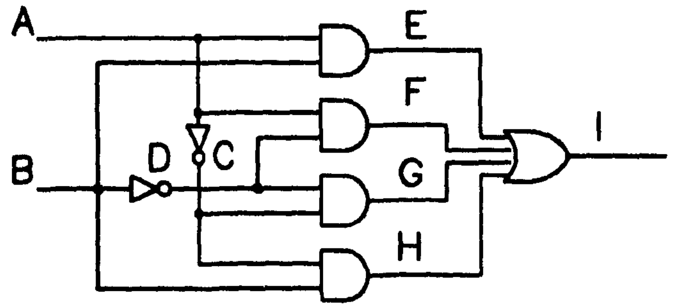


Figure 4.14 A 2-input trivial function

necessary assignments and through backtracks which may occur if an incomplete analysis is done and some necessary assignments are overlooked.

Example 4.20: The circuit of Fig. 4.14 implements the function $AB+\bar{A}B+A\bar{B}+\bar{A}\bar{B}=1$. All primary inputs are assigned to $\{0,1\}$; the set of possible values of all nodes determined during forward propagation is $\{0,1\}$ (not shown in the diagram to increase readability). Although it is not possible to produce $\{0\}$ at the output of the circuit, no correlation is detected by the reduction lists (see the preceding section). The uncorrected pessimism may result in unnecessary branching and backtracking during test pattern generation for faults in a circuit containing this module. Note, however, that the reduction equations are able to prove that i_{s_1} is untestable without branching. (In order to produce $\{0\}$ at node I, $E=F=G=H=\{0\}$ are required. $A_{\{1\}}$ requires $B_{\{0\}}$ in order to satisfy $E_{\{0\}}$, but $B_{\{0\}}$ requires $A_{\{0\}}$ in order to satisfy $F_{\{0\}}$. A conflict is detected since A cannot simultaneously be assigned to both 0 and 1.)

The previous example indicates that, at the cost of additional processing, a more exact analysis of stem correlation can be done by attempting to justify each of the possible output values of each gate in the circuit at which some fanout stem reconverges and then dropping all values which cannot be justified. This technique is equivalent to analysing the effect of double assignments (*i.e.* identifying second-order necessary assignments) It is not difficult to design a circuit in which the only necessary assignments are third (or higher) order and cannot be identified even if double assignments are considered

Necessary assignment identification can also be performed in terms of *conduction analysis* in which the assignments that produce values at network nodes are captured. Reduction analysis is more concise, however, as 2^n conduction lists are required to capture the same information as is done with n reduction lists. For example, conduction list $C^A_{\{0\}}$ contains those assignments which cause node A to become $\{0\}$, but says nothing about assignments which cause A to become a nonunique set of values containing 0 (for example $\{0,1\}$ or $\{D,0,1\}$—a unique conduction list is required for each possible value, 2^n in all. On the other hand, one reduction list is required for each element of the base set of values, n in all. Conduction lists can be formed by the intersection of the corresponding reduction lists.

Chapter 5 Exploiting circuit topology in efficient identification of necessary assignments

The processing required to compute the reduction lists is proportional to the number of assignments which must be analysed and the area of the circuit in which they must be propagated. This chapter discusses structural properties of the circuit under test which can be used to restrict both while guaranteeing that no first-order necessary assignments are overlooked. This information can be found in a preprocessing step and then reused for each target fault.

Certain necessary assignments can be identified by backward implication. Since backward implication can be performed in linear time, an efficient test generator should identify as many necessary assignments as possible using backward implication before going to more expensive and sophisticated techniques such as reduction list computation. The key observation is that only potentially necessary assignments which would not be identified using backward implication should be analysed using reduction lists.

5.1 Structural analysis of reconvergence in combinational circuits

This section presents definitions which are used throughout the chapter to describe the topology of the circuit under test in order to formalize reduction list computation. Necessary assignment identification represents an application of the *stem region concept* [MaaRaj90] to a general problem involving signal propagation along potentially reconvergent paths.

Definitions 5.1 through 5.3 are taken from [MaaRaj90].

Definition 5.1: If there are two or more disjoint paths between stem A and gate B, then A is a ***reconvergent fanout stem***, and B is a ***primary reconvergence gate*** of stem A.

- If there are no reconvergent fanout stems on the paths from reconvergent fanout stem A and its primary reconvergence gates, then A is a ***narrow reconvergent fanout stem***. Otherwise, A is a ***wide reconvergent fanout stem***.
- Let C be a narrow reconvergent fanout stem. 1) If C is located on a path between reconvergent fanout stem A and a primary reconvergence gate of A, then all the primary reconvergence gates of stem C that are not primary reconvergence gates of stem A are ***secondary reconvergence gates*** of stem A. 2) If stem D is located on a path between reconvergent fanout stem A and a primary reconvergence gate of A, then all the primary and secondary reconvergence gates of D that are not primary reconvergence gates of stem A are ***secondary reconvergence gates*** of stem A.

Primary and secondary reconvergence gates of a stem are referred to collectively as ***reconvergence gates*** of that stem.

Definition 5.2: The ***stem region*** of reconvergent fanout stem A is composed of all the circuit nodes (stems and gates) that are both reached by stem A and reach a reconvergence gate of stem A, and all the output lines of these nodes.

Definition 5.3: Let x be a line in the stem region of A. x is an exit line of stem A if x belongs to the stem region of A (x is an output line of a node in the stem region of A), and x in an input to a node which is not in the stem region of A.

The algorithm given in [MaaRaj90] can be used to identify the reconvergence gates, stem region, and exit lines for all reconvergent fanout stems in the circuit under test

5.2 Candidate assignment identification

Definition 5.4: An assignment to a network node which may reduce the value of some justification point is called a *candidate assignment* if the corresponding necessary assignment cannot be identified by backward implication. The node is called a *candidate node*.

The time required to compute the reduction lists is proportional to their length. Since reduction list calculation is more costly than conventional backward implication, an efficient test generator should not use reduction lists to identify necessary assignments which can be found by other means. This section discusses the impact of reconvergent fanout on necessary assignment identification and presents properties related to the structure of the circuit under test which can be used to restrict the number of candidate assignments which are analysed using the reduction lists. Assignments which do not match the criteria of any of the following properties are not candidate assignments and need not be considered.

Definition 5.5: For gate G performing function $f : X \times Y \rightarrow Z$, with inputs A, B nonempty subsets of X and Y, respectively, $A \subseteq X$, $B \subseteq Y$, $x \in A$ is a *controlling input value* of G if $\|B\| > 1$ and $f^{-Y}_{|x \times B}(f(x, B)) = B$. Similarly, $y \in B$ is a controlling input value of G if $\|A\| > 1$ and $f^{-X}_{|A \times y}(f(A, y)) = A$. $C \subseteq Z$ is a *controlling output value* of G if $\|f^{-X}_{|A \times B}(C)\| > 1$ and $\|f^{-Y}_{|A \times B}(C)\| > 1$.

In other words, a controlling value at the input to a gate forces the gate output to value v, regardless of the value(s) at the othe input(s) of the gate. A controlling output value of a gate is one which can be produced by two or more combinations of values from its inputs. Note that definition 5.5 is not static, but takes into account the current input and output values of the gate.

Lemma 5.1: If no additional necessary assignments can be identified by backward implication, the required value of each justification point is a controlling output value of the corresponding gate.

Proof: If there is only a single input combination which could produce the required output value, then backward implication can be used to replace the justification point with other requirements closer to primary inputs. Similarly, requirements at the output of any single-input gate can be replaced by requirements at the input of the gate. ∎

If an assignment to a node in the network causes a reduction at a justification point—*i.e.* there is a necessary assignment to the node—then that assignment can be identified by conventional backward implication if there is only a single path from the node to the justification point. However, if there are two or more disjoint paths from some node to the justification point, the effect of an assignment to the node may propagate along multiple paths and cause a reduction at the justification point. Conventional backward implication does not take reconvergent fanout into account and thus could not identify the necessary assignment.

As discussed in the previous chapter, additional necessary assignments may be identified in the restricted space after the first set of necessary assignments is applied to the circuit under test. Thus, backward implication and reduction analysis are performed iteratively until no more necessary assignments can be found. To further reduce computation, reduction analysis is restricted to those assignments which would not be found after several iterations of processing. That is, if an assignment to node A is necessary only if an assignment to node B is also necessary, then the processing of A can be put off until the status of B is determined.

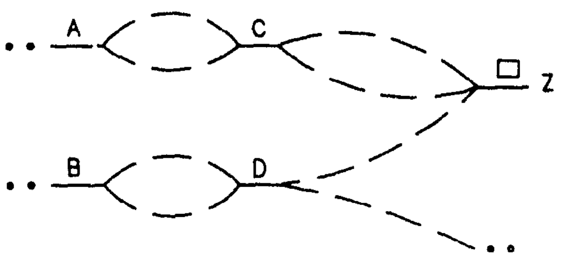


Figure 5.1 Candidate assignment identification

Example 5.1: Fig. 5.1 depicts a subcircuit containing one justification point, Z, and four stems, A, B, C, and D, three of which are reconvergent. There is a single path from stem D to Z, so any necessary assignment to D can be found by backward implication from Z. On the other hand, stem C reconverges at Z. The effect of an assignment to C may propagate along disjoint paths to Z causing a reduction which could not be identified by backward implication. Although stem A is reconvergent, all paths from A to Z pass through C. If an assignment to A is necessary, then an assignment to C is also necessary—there is no reason to analyse A until necessary assignments to C (if any) have been made. Similarly, all paths from B to Z pass through D, so assignments to B need not be analysed as their effect is registered at D. Thus, stem C is the only candidate stem in Fig. 5.1.

Property 5.1 formalizes these conditions:

Property 5.1: Assignments to stems which reconverge at any justification point are candidate assignments.

Proof: Assume that backward implication cannot be used to identify any additional necessary assignments from justification point J and that an assignment to node N reduces the required value of J. From lemma 5.1, the required value at J is a controlling output value of J. The reduction assignment to N controls J in the sense that it reduces the r.quired value at J regardless of other assignments in the network. There are two cases to consider:

1. A single path from N to J. The assignment to N must cause controlling value to appear at the input to J which is reached from N—otherwise the required value at J would not be reduced. This implies that the required value of J can be reduced by an assignment to that input, and thus that backward implication can be used to identify a necessary assignment to that input—violating the original assumption. This case cannot occur.
2. Multiple paths from N to J An assignment to N may change the value of several inputs to J. This case cannot be captured by conventional backward implication,

which does not consider reconvergence. Thus, assignments to N must be explicitly analysed.

3. Multiple paths from N to J which are not disjoint. Since the paths are not disjoint, there is a common section. The reduction assignment to N must control (in the sense of case 1, above) the gates along the common part of the path, and in particular, the common gate closest to J. Call this gate G. If there is a reduction assignment to N then there is also a reduction assignment to G—the value assumed by G when the reduction assignment to N is made. However, since G is simply another node in the circuit under test, candidate assignments to G are also considered. From G, there may be either a single path (case 1) or multiple disjoint paths (case 2) to J; in either event, there is no reason to explicitly analyse candidate assignments to N until necessary assignments to G (if any) are made. After necessary assignments to G are made, then necessary assignments to N will be identified in the next iteration of either backward implication or reduction analysis, depending on whether or not there are multiple disjoint paths from N to G. In either case, first-order necessary assignments to N will not be overlooked. ∎

In a preprocessing step, lists of stems which reconverge at each node in the circuit are recorded. During test generation, the union of the lists of reconvergence stems for each *AND*-node of the justification list are marked as candidate stems.

The set of values which must be justified in order to test the fault can be represented by an *AND-OR* graph, as discussed in chapter 4. *OR*-nodes arise because the fault may be detectable on two or more primary outputs. It is not necessary to justify any individual *OR*-node—the only requirement is that at least one of them be satisfied. From lemma 4 1, an assignment must reduce the desired values of all *OR*-nodes (*i.e.* both D and $\overline{D}$ at all the reached primary outputs) in order to be a reduction assignment. Thus, only assignments to stems which reach all of the *OR*-nodes of the justification list are potential reduction assignments.

Property 5.2: Candidate assignments with respect to the *OR*-nodes of the justification list are to those stems which reach them all.

Proof: An assignment to a stem cannot affect the value of an *OR*-node which it does not reach. Thus, if a stem does not reach all of the *OR*-nodes, it cannot affect all of them. Therefore, assignments to that stem could not be necessary and need not be analysed. ∎

As the only *OR*-nodes which arise during test pattern generation are primary outputs to which the fault effect may propagate, the list of stems which logically drive each of the primary outputs is recorded in a preprocessing step. During test pattern generation, the list of reached ·tems for each *OR*-node of the justification list are intersected to find the set of stems which reach them all; these stems are candidates.

Certain stems are candidates despite that there are only single paths from them to justification points. Logic constraints propagate unconditionally in the circuit—both from outputs toward inputs and from inputs toward outputs. As was demonstrated in example 4.14, an assignment to a node may cause a reduction in an area of the circuit which is neither driven by nor drives the node. Assignments which reach one justification point may correlate the values of other nodes and thus may cause a reduction. These cases arise only when two or more justification points lie in the stem region of the node.

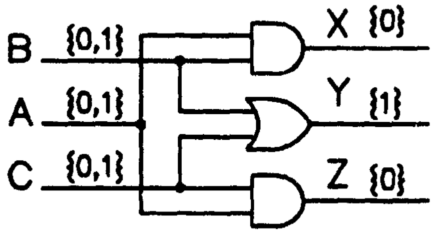


Figure 5.2 Transfer of requirements from one justification point to another

Example 5.2: There are three justification points in the circuit shown in Fig. 5.2: $X_{\{0\}}$, $Y_{\{1\}}$, and $Z_{\{0\}}$. $A_{\{1\}}$ requires $B_{\{0\}}$ (in order to justify $X_{\{0\}}$); $B_{\{0\}}$ implies $C_{\{1\}}$ (in order to satisfy $Y_{\{1\}}$); $C_{\{1\}}$ requires $A_{\{0\}}$ (in order to satisfy $Z_{\{0\}}$). Since $A_{\{1\}}$ leads to a conflict, $A_{\{0\}}$ is a necessary assignment.[10] $A_{\{0\}}$ cannot be identified by backward implication, thus assignments to stem A must be analysed using the reduction lists.

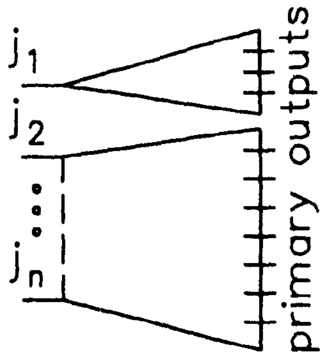


Figure 5.3 Premise. No stems exist whose stem regions include $J1$ and at least one other justification point

Property 5.3: Stems whose stem region includes two or more justification points are candidates.

Proof: (By contradiction) Assume that there exists stem S which has no reconvergence gates among the justification points and whose stem region includes only one justification point, and that an assignment to S is necessary. Without loss of generality, assume that the justification point included in the stem region of S is $J1$, as shown in Fig. 5.3. The primary outputs reached by $J1$ are disjoint with those reached by the other justification points—otherwise a reconvergent stem driving $J1$ and at least one other justification point must exist. Since the set of outputs reached by $J1$ is disjoint with those reached by the other justification points, an assignment to S which makes it impossible to observe the fault effect on the outputs reached by $J1$ cannot impede propagation to any of the other outputs—S does not reach them. The assignment to S is not necessary, contradicting the original hypothesis. ∎

[10] $A_{\{0\}}$ is necessary in the sense that there is no test pattern in the space defined by $A_{\{1\}}$. Of course, there may be no test pattern in the space defined by $A_{\{0\}}$ either, in which case the current set of justification points cannot be satisfied, and a backtrack must be performed

Assignments to certain non-stem nodes are also candidates. Assignments to the outputs of multi-input gates may correlate the value of several stems, the effect of which may propagate further in the network than a single assignment to any of the stems. In this case, the correlated stem assignments are subsumed by the non-stem assignment(s) and no longer need be considered.

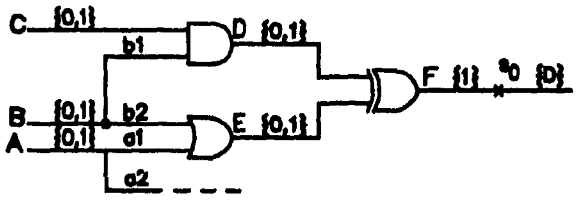
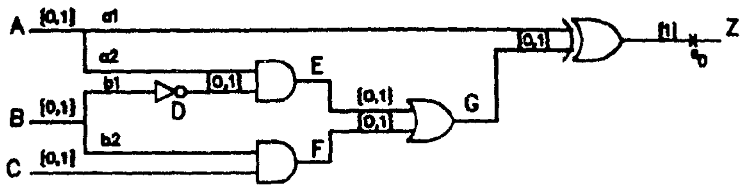


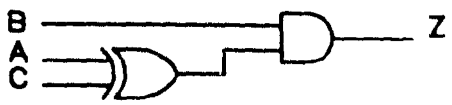
Figure 5.4 Propagation of gate assignments

Example 5.3: In Fig. 5.4, there are no stem assignments which propagate to the output of gate F. However, $E_{\{0\}}$ causes both stems A and B to become $\{0\}$, the combined effect of which "bounces back" to F from stem B through line $b1$ and gate D, reducing $F_{\{1\}}$. The necessary assignment to E may have an impact on justification points elsewhere in the circuit, as its effect propagates forward from $a2$.

Non-stem candidate assignments can be found using property 5.4 in linear time in a levelized forward trace from candidate stems.

Property 5.4: The set of candidate assignments to the output of gate G performing function f, with inputs $I1,...,In$ carrying values $v1,...,vn$ with candidate assignments $v1',...,vn'$, respectively ($v1' \subseteq v1,...,vn' \subseteq vn$), is $Z = \{z \in f(v1,...,vn) \mid \|f^{-Ij}_{|v1\times \quad \times vn}(z)\| = 1$ and $f^{-Ij}_{|v1\times \quad \times vn}(z) \subseteq vj'$ for some j, $1 \leq j \leq n\}$. The subsumed input candidate assignments, vj'' for input Ij ($vj'' \subseteq vj'$), are those input assignments uniquely implied by the candidate output assignments: $vj'' = \{x \in vj' \mid f^{-Ij}_{|v1\times \quad \times vn}(z) = x$ for some $z \in Z\}$.





c) Minimized circuit

a) Non-minimal circuit

Line	List	Contents	List	Contents
A	$\overleftarrow{R}_0^A$ $\overrightarrow{R}_1^A$	$\{E_{\{1\}}\}$ $\{A_{\{0\}}\}$	$\overleftarrow{R}_1^A$	$\{A_{\{0\}}\}$
$a1$	$\overrightarrow{R}_0^{a1}$ $\overrightarrow{R}_1^{a1}$	$\{E_{\{1\}}\}$ $\{A_{\{0\}}\}$	$\overleftarrow{R}_1^{a1}$	$\{A_{\{0\}}\}$
$a2$	$\overleftarrow{R}_0^{a2}$ $\overrightarrow{R}_1^{a2}$	$\{E_{\{1\}}\}$ $\{A_{\{0\}}\}$	$\overleftarrow{R}_1^{a2}$	$\{A_{\{0\}}\}$
B	$\overleftarrow{R}_0^B$	$\{F_{\{1\}}\}$	$\overleftarrow{R}_1^B$	$\{E_{\{1\}}\}$
$b1$	$\overrightarrow{R}_0^{b1}$	$\{F_{\{1\}}\}$	$\overleftarrow{R}_1^{b1}$	$\{E_{\{1\}}\}$
$b2$	$\overleftarrow{R}_0^{b2}$	$\{F_{\{1\}}\}$	$\overrightarrow{R}_1^{b2}$	$\{E_{\{1\}}\}$
C	$\overleftarrow{R}_0^C$	$\{F_{\{1\}}\}$		
D	$\overleftarrow{R}_0^D$	$\{E_{\{1\}}\}$	$\overrightarrow{R}_1^D$	$\{F_{\{1\}}\}$
E	$\overrightarrow{R}_0^E$	$\{E_{\{1\}}\}$	$\overleftarrow{R}_0^E$	$\{E_{\{1\}}\}$
F	$\overrightarrow{R}_0^F$	$\{F_{\{1\}}\}$	$\overleftarrow{R}_0^F$	$\{F_{\{1\}}\}$
G	$\overrightarrow{R}_0^G$	$\{E_{\{1\}}, F_{\{1\}}\}$	$\overleftarrow{R}_0^G$	$\{A_{\{0\}}\}$
Z	$\overrightarrow{R}_1^Z$	$\{E_{\{1\}}\}$		

b) Reduction lists for **a** (empty reduction lists not shown)

Figure 5.5 Test generation for z_{s_0}

Proof: From the definition of the image and inverse image of set functions. ∎

Example 5.4: The candidate assignments to test z_{s_0} in the circuit from Fig 5 5a found using property 5.4 are $A_{\{0\}}$, $E_{\{1\}}$, and $F_{\{1\}}$—$E_{\{1\}}$ subsumes $A_{\{1\}}$ and $B_{\{0\}}$, and $F_{\{1\}}$ subsumes $B_{\{1\}}$. $E_{\{1\}}$ reduces $Z_{\{1\}}$, a justification point—thus $E_{\{0\}}$ is a necessary assignment If non-stem candidate assignments are not analysed, then necessary assignment $B_{\{1\}}$ would be overlooked. Note that the circuit in Fig 5.5a performs the same function as the minimized circuit in Fig. 5.5c, in which all necessary assignments can be directly identified by backward implication.

5.3 Region of propagation

The second major issue in the efficient identification of necessary assignments is the area in which reduction analysis must be performed. It is important to restrict the region of the circuit in which reduction lists are computed to avoid calculating lists which will not be used to identify necessary assignments.

Definition 5.6: The *shadow* of the justification points is the region of the network reached by tracing backward from the justification points.

Definition 5.7: The *cone* of the candidate stems is the region reached by tracing forward from the candidate stems (found using properties 5.1 through 5.3).

The justification points and candidate stems mark boundaries beyond which reduction lists need not be computed. However, due to the unconditional nature of reduction list propagation, candidate assignments may appear on reduction lists, anywhere in the region of propagation. In particular, stem assignments may propagate outside of their stem region.

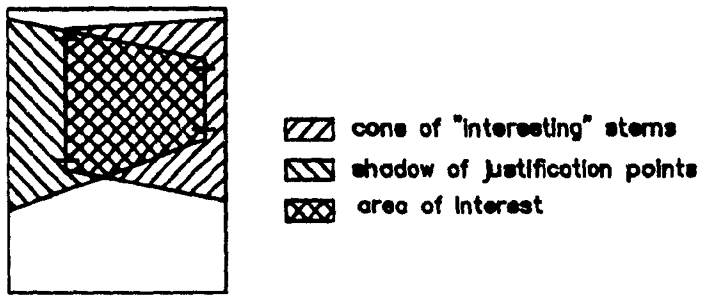

▨ cone of "interesting" stems
▧ shadow of justification points
▩ area of interest

Figure 5.6 Property 5 5 the region of reduction list propagation

Property 5.5: Reduction lists need not be computed outside the region where the shadow of the justification points overlaps the cone of the candidate stems.

Proof: By definition, there are no justification points outside the shadow at which assignments which propagate outside the shadow could cause a reduction or from which such assignments could "bounce back" to reenter the shadow. By definition, there are no candidate assignments outside their cone. Assignments need not be propagated backward outside of the cone as there are no stems from which such assignments could "bounce forward" to re-enter the cone (any such stem must reconverge at a justification point and thus would be a candidate). ■

During the course of test generation, justification points are added as assignments are made, deleted as they are satisfied, and changed as backtracks occur. The candidate assignments and region in which they must be analysed is related to the set of justification points, and change as the justification points change. Each time the set of justification points is modified, the candidate assignments and region of propagation must be updated. The overhead to update the set of candidate assignments is related to the topology of the circuit under test. The region of propagation can be updated in time linear in the size of the circuit.

Chapter 6 A method to identify nonconflicting assignments based on Boolean function monotonicity

Nonconflicting assignments have the desirable property of vastly and irrevocably reducing the space which must be searched for a test vector while guaranteeing that they will never need to be reversed. If a test pattern exists in the search space before the nonconflicting assignment is made, then at least one test pattern exists in the search space after the assignment is made (Fig. 6.1**a**). Conversely, if there is no test pattern in the search space after the assignment is made, then there was no test pattern in the original space (Fig. 6.1**b**).

a) Test pattern exists **b)** No test pattern exists

Figure 6.1 Pruning assignments

In this chapter, *tendency lists* are defined for general Boolean algebras in terms of images and inverse images of set functions, providing a systematic means to analyse function monotonicity and identify nonconflicting assignments during deterministic test generation.

6.1 Generalized function monotonicity and tendency lists

A Boolean function is said to be *unate* in variable x if x appears in only true ("positive unate") or only complemented ("negative unate") form in the sum-of-products or product-of-sums form of the function [BrHaMcSa84]. Function monotonicity or unateness can be exploited during test generation.

Example 6.1: In order to sensitize the s_0 fault at the output of a 2-input *MUX* performing function $f = AB + \bar{B}\bar{C}$, it is necessary to justify a "1" at the circuit output. Justification is equivalent to satisfying the Boolean equation $AB + \bar{B}\bar{C} = 1$—which can be satisfied by making assignments in such a way that one or the other minterm evaluates to 1. To this end, assignments $A = 1$ and $C = 0$ lead in the direction of the goal and cannot conflict with it. On the other hand, either assignment to input B, while leading one term toward the goal, leads the other term away from the goal, so there is no nonconflicting assignment to B.

Function montonicity in networks which do not contain faults can be described in terms of the two-element Boolean algebra $B_2^1 = \{0, 1\}$ and is equivalent to unateness of the Boolean function implemented by the network. There is a direct correspondence between the variables in the sum-of-products (product-of-sums) form of the Boolean equations describing network node functions and the values assumed by these nodes when test vectors are applied.

In circuits which may contain faults, however, monotonicity cannot be defined in terms of B_2^1 as it is not possible to represent the possible presence of faults. One solution is to use B_2^1 and record function monotonicity separately for the fault-free and faulty circuit. This solution is poor, however, as it does not exploit the close relationship between the fault-free and faulty circuits in the region reached by the fault effect. On the other hand, $B_2^2 = \{0, \overline{D}, D, 1\}$ is sufficiently precise to describe circuit values in the presence of potential faults and recognizes the relation between the fault free and faulty circuits, offering a better solution. In the region of the circuit not reached by the fault effect, monotonicity defined in terms of B_2^2 reduces to that defined in terms of B_2^1

For general Boolean algebras, theorem 6.1 describes the construction of *tendency lists* at the output of a logic block performing function f, given the tendency lists at its inputs. Assignments which appear on tendency list T_z^C at node C lead C toward value z (an element in the base set of the power set which is isomorphic to the Boolean algebra in question)—after those assignments are made, z will be in the set of possible values at C if it was possible to produce z at C before the assignments were made. In other words, tendency lists give, for each element of the base set of the Boolean algebra (power set), those assignments which lead in the direction of that element and do not conflict with the goal of producing it

Theorem 6 1. Let f $X \times Y$ - Z be a function, X, Y, and Z Boolean algebras, and A, B, and C be nonempty subsets of X, Y, and Z, respectively, $A \subseteq X$, $B \subseteq Y$, $C \subseteq Z$. Let T_x^X, T_y^Y for each $x \in A$, $y \in B$, be the set of assignments which tend toward values x, y at input coordinates X, Y, respectively. The set of assignments which tend toward value z in set C at coordinate Z, denoted by T_z^Z, is the union of all assignments which tend toward x at coordinate X, $x \in A$, if $f(x,y) = z$ for some $y \in B$ and $\|f(A,y)\| > 1$ combined with all assignments which tend toward y at coordinate Y, $y \in B$, if $f(x,y) = z$ for some $x \in A$ and $\|f(x,B)\| > 1$. In set builder notation, for each $z \in C$:

$$A' = \{\, a \in A \mid f(a,b) = z \text{ for some } b \in B \text{ and } f^{-X}_{|A\times b}(z) \neq A \,\}$$
$$B' = \{\, b \in B \mid f(a,b) = z \text{ for some } a \in A \text{ and } f^{-Y}_{|a\times B}(z) \neq B \,\}$$
$$T_z^Z = \bigcup_{x\in A'} T_x^X \cup \bigcup_{y\in B'} T_y^Y .$$

Proof. (By construction) In order for an assignment appearing on a tendency list at input coordinate X to cause Z to tend toward z, that assignment must cause input X to tend toward a value x which can be combined with a value b at input Y to produce z at Z. However, if the image of A at input X and b at input Y on output Z is uniquely z, then b at input Y is the only requirement to satisfy [illegible] at X is not required A similar argument applies to assignments appearing at input [illegible] ∎

Theorem 6.1 is used to derive tendency equations for a 2-input *AND* gate with inputs A and B and output C for different combinations of input values in the following example.

Example 6.2: For an *AND* gate, with inputs $A = \{0, 1, D, \overline{D}\}$, $B = \{0, 1, D, \overline{D}\}$ and output $C = \{0, 1, D, \overline{D}\}$:

$$
\begin{aligned}
T_0^C &= T_0^A \cup T_{\overline{D}}^A \cup T_D^A \cup T_0^B \cup T_{\overline{D}}^B \cup T_D^B \\
T_{\overline{D}}^C &= T_1^A \cup T_{\overline{D}}^A \cup T_1^B \cup T_{\overline{D}}^B \\
T_D^C &= T_1^A \cup T_D^A \cup T_1^B \cup T_D^B \\
T_1^C &= T_1^A \cup T_1^B .
\end{aligned}
$$

Similarly, for an *AND* gate with inputs $A = \{0, \overline{D}, D, 1\}$, $B = \{0, \overline{D}\}$ and output $C = \{0, \overline{D}\}$:

$$
\begin{aligned}
T_0^C &= T_0^A \cup T_D^A \\
T_{\overline{D}}^C &= T_{\overline{D}}^A \cup T_1^A \cup T_{\overline{D}}^B .
\end{aligned}
$$

6.2 Tendency lists and nonconflicting assignments

Tendency lists can be used to identify nonconflicting assignments during automatic test pattern generation. The key observation is that assignments to primary inputs can always be justified, whereas assignments to internal nodes may not themselves be satisfiable Therefore, primary input assignments only are analysed using tendency lists.

The tendency lists at primary input nodes are initialized with the corresponding input assignments. $Ij_{\{0\}}$ is added to list T_0^{Ij} and $Ij_{\{1\}}$ is added to list T_1^{Ij} at each unassigned primary input Ij. Tendency lists in the rest of the circuit can be computed in linear time using theorem 6.1 in a levelized forward pass For each possible value of every node in the network, the corresponding tendency list contains those input assignments which lead in the direction of that value at the node Unique input assignments which appear on that tendency list are nonconflicting assignments with respect to that node and value

The tendency lists at the justification points are used to identify nonconflicting assignments. A *vote* is collected by finding the union of the tendency lists corresponding to the desired value of all justification points. Unique assignments appearing in this set are nonconflicting with respect to all justification points and can be applied to the network under test.[11]

Lemma 6.1: Given the set of justification points $\{A^1_{v_1}, \ldots, A^n_{v_n}\}$ all of which must be satisfied (*AND*-nodes in the *AND-OR* graph) and $\{O^1_{u_1}, \ldots, O^m_{u_m}\}$ at least one of which must be satisfied (*OR*-nodes in the *AND-OR* graph), unique assignments appearing in

$$\left(\bigcup_{i=1}^{n}\Big(\bigcup_{\forall x \in v_i} T_x^{A^i}\Big)\right) \cup \left(\bigcup_{j=1}^{m}\Big(\bigcup_{\forall y \in u_j} T_y^{O^j}\Big)\right)$$

are nonconflicting assignments.

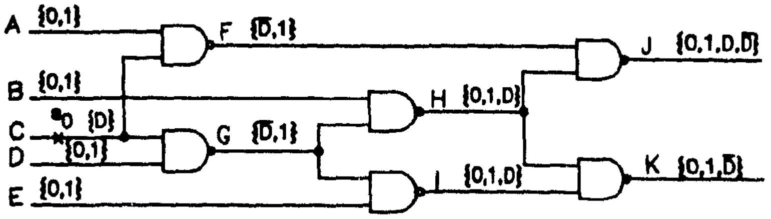
Proof: From theorem 6.1, such an assignment could not lead any justification point away from its desired value(s). ■

The following examples illustrate nonconflicting assignment identification during test pattern generation.

Example 6.3: $C_{\{1\}}$ is the only necessary assignment to generate a test for fault C_{s0} in Fig. 6.2. Voting at justification points $J_{\{D,\overline{D}\}}$ and $K_{\{\overline{D}\}}$, the set of desirable assignments is $\{A_{\{0\}}, B_{\{1\}}, D_{\{1\}}\} \cup \{A_{\{1\}}, B_{\{0\}}\} \cup \{B_{\{1\}}, D_{\{1\}}, E_{\{1\}}\}$, resulting in a unanimous vote on inputs D and E, yielding nonconflicting assignments $D_{\{1\}}$ and $E_{\{1\}}$, a test vector for the fault. Note that $C_{\{1\}}$ is a nonconflicting assignment in addition to being necessary.

Example 6.4: Nonconflicting assignments may be identified when the target fault is untestable. In that instance, they restrict the remaining search space, enabling the test generator

[11] All nonconflicting assignments can be made simultaneously as the unateness properties of the inputs are independent of each other However assigning them all may result in an over-specified test vector—a vector which contains fewer don't care input assignments than might otherwise be the case That is, if the nonconflicting assignments are made serially rather than simultaneously the test generator may recognize that the fault is tested with fewer inputs assigned To minimize the number of assigned primary inputs while avoiding unnecessary branching and backtracking, it may be desirable to make nonconflicting input assignments serially only if no necessary assignments can be identified



a) Example circuit

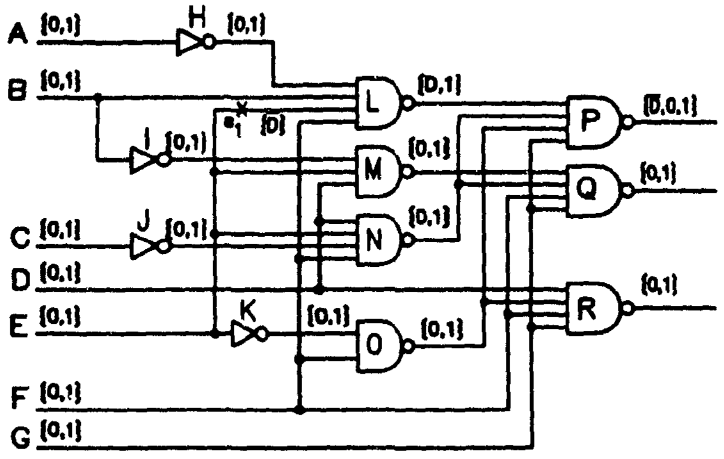
Line	List	Contents	Line	List	Contents
A	T_0^A	$\{A_{\{0\}}\}$	H	T_0^H	$\{B_{\{1\}}, D_{\{0\}}\}$
	T_1^A	$\{A_{\{1\}}\}$		T_D^H	$\{B_{\{1\}}, D_{\{1\}}\}$
B	T_0^B	$\{B_{\{0\}}\}$		T_1^H	$\{B_{\{0\}}\}$
	T_1^B	$\{B_{\{1\}}\}$	I	T_0^I	$\{D_{\{0\}}, E_{\{1\}}\}$
C	T_0^C	$\{C_{\{0\}}\}$		T_D^I	$\{D_{\{1\}}, E_{\{1\}}\}$
	T_1^C	$\{C_{\{1\}}\}$		T_1^I	$\{E_{\{0\}}\}$
D	T_0^D	$\{D_{\{0\}}\}$	J	T_0^J	$\{A_{\{0\}}, B_{\{0\}}\}$
	T_1^D	$\{D_{\{1\}}\}$		$T_{\bar{D}}^J$	$\{A_{\{0\}}, B_{\{1\}}, D_{\{1\}}\}$
E	T_0^E	$\{E_{\{0\}}\}$		T_D^J	$\{A_{\{1\}}, B_{\{0\}}\}$
	T_1^E	$\{E_{\{1\}}\}$		T_1^J	$\{B_{\{1\}}, D_{\{0\}}\}$
F	$T_{\bar{D}}^F$	$\{A_{\{1\}}\}$	K	T_0^K	$\{B_{\{0\}}, E_{\{0\}}\}$
	T_1^F	$\{A_{\{0\}}\}$		$T_{\bar{D}}^K$	$\{B_{\{1\}}, D_{\{1\}}, E_{\{1\}}\}$
G	$T_{\bar{D}}^G$	$\{D_{\{1\}}\}$		T_1^K	$\{B_{\{1\}}, D_{\{0\}}, E_{\{1\}}\}$
	T_1^G	$\{D_{\{0\}}\}$			

b) Tendency lists for **a**

Figure 6.2 Test generation for C_{s_0}

to identify a conflict more quickly (*i.e.* with fewer branches and backtracks). The circuit in Fig. 6.3 illustrates the identification of nonconflicting assignments during redundancy identification for $e0_{s_1}$. Voting at justification points $E_{\{0\}}$ (fault sensitization) and $J_{\{\bar{D}\}}$ (fault propagation) identifies nonconflicting assignments $A_{\{0\}}$, $B_{\{1\}}$, $C_{\{1\}}$, $D_{\{0\}}$, $E_{\{0\}}$, and $G_{\{1\}}$, reducing the remaining search space to $\frac{1}{2^6}$ of its previous size

The tendency lists correspond to the values in the network, reflecting the assignments which have been made and the values which must be justified. Assignments made during



a) Test generation for $e0_{s_1}$

Line	List	Contents	Line	List	Contents
A	T_0^A	$\{A_{\{0\}}\}$	J	T_0^J	$\{C_{\{1\}}\}$
	T_1^A	$\{A_{\{1\}}\}$		T_1^J	$\{C_{\{0\}}\}$
B	T_0^B	$\{B_{\{0\}}\}$	K	T_0^K	$\{E_{\{1\}}\}$
	T_1^B	$\{B_{\{1\}}\}$		T_1^K	$\{E_{\{0\}}\}$
C	T_0^C	$\{C_{\{0\}}\}$	L	T_D^I	$\{A_{\{0\}}, B_{\{1\}}, F_{\{1\}}\}$
	T_1^C	$\{C_{\{1\}}\}$		T_1^I	$\{A_{\{1\}}, B_{\{0\}}, F_{\{0\}}\}$
D	T_0^D	$\{D_{\{0\}}\}$	M	T_0^M	$\{B_{\{0\}}, E_{\{1\}}, D_{\{1\}}\}$
	T_1^D	$\{D_{\{1\}}\}$		T_1^M	$\{B_{\{1\}}, E_{\{0\}}, D_{\{0\}}\}$
E	T_0^E	$\{E_{\{0\}}\}$	N	T_0^N	$\{D_{\{1\}}, E_{\{1\}}, C_{\{0\}}, F_{\{1\}}\}$
	T_1^E	$\{E_{\{1\}}\}$		T_1^N	$\{D_{\{0\}}, E_{\{0\}}, C_{\{1\}}, F_{\{0\}}\}$
F	T_0^F	$\{F_{\{0\}}\}$	O	T_0^O	$\{E_{\{0\}}, F_{\{1\}}\}$
	T_1^F	$\{F_{\{1\}}\}$		T_1^O	$\{E_{\{1\}}, F_{\{0\}}\}$
G	T_0^G	$\{G_{\{0\}}\}$	P	$T_{\overline{D}}^P$	$\{A_{\{0\}}, B_{\{1\}}, C_{\{1\}}, D_{\{0\}}, E_{\{0\}}, F_{\{0,1\}}, G_{\{1\}}\}$
	T_1^G	$\{G_{\{1\}}\}$		T_0^P	$\{A_{\{1\}}, B_{\{0\}}, C_{\{1\}}, D_{\{0\}}, E_{\{0,1\}}, F_{\{0\}}, G_{\{1\}}\}$
				T_1^P	$\{C_{\{0\}}, D_{\{1\}}, E_{\{0,1\}}, F_{\{1\}}, G_{\{0\}}\}$
H	T_0^H	$\{A_{\{1\}}\}$	Q	T_0^Q	$\{B_{\{1\}}, C_{\{1\}}, D_{\{0\}}, E_{\{0\}}, F_{\{0,1\}}, G_{\{1\}}\}$
	T_1^H	$\{A_{\{0\}}\}$		T_1^Q	$\{B_{\{0\}}, C_{\{0\}}, D_{\{1\}}, E_{\{1\}}, F_{\{0,1\}}, G_{\{0\}}\}$
I	T_0^I	$\{B_{\{1\}}\}$	R	T_0^R	$\{D_{\{1\}}, E_{\{1\}}, F_{\{0,1\}}, G_{\{1\}}\}$
	T_1^I	$\{B_{\{0\}}\}$		T_1^R	$\{D_{\{0\}}, E_{\{0\}}, F_{\{0,1\}}, G_{\{0\}}\}$

b) Tendency lists for a)

Figure 6.3 Nonconflicting assignment identification in redundancy identification

test generation, whether necessary, nonconflicting, or arbitrary, constrain the operation of the circuit under test and may lead to the identification of additional nonconflicting assignments which could not be identified previously. The tendency lists are updated to reflect the possible values in the network each time the values change.

As the maximum tendency list length for each network node can be predetermined (the number of primary inputs which reach it), efficient tendency list computation can be implemented easily. Tendency lists need be calculated only in the *shadow* of the justification points (see section 5.3). Incremental updating of the tendency lists is simple as changes originate at and propagate forward from nodes whose value was changed.

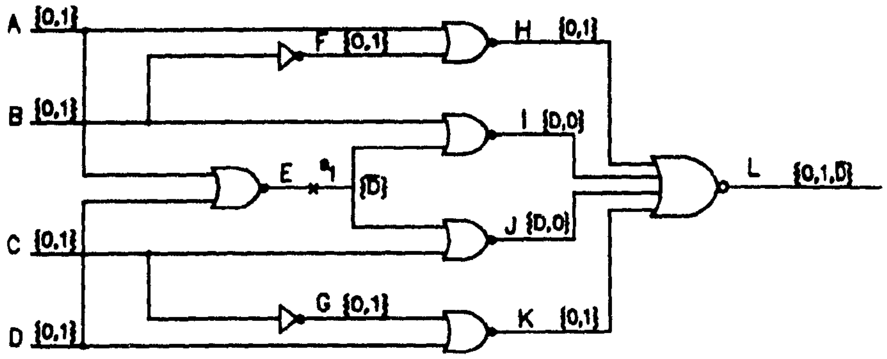
Nonconflicting assignment analysis can take the place of heuristic techniques used by other test generation algorithms. For example, required values of "head lines" [FujShi83] can be justified by nonconflicting assignments to primary inputs, so there is no need to put off their justification. Other techniques such as [SilSpi88] which rank branch assignments by their likelihood to cause conflicts can be replaced, since those assignments are often nonconflicting.

6.3 Pessimism in the tendency analysis

Tendency list calculation performed using theorem 6.1 is pessimistic in that some nonconflicting assignments which can be identified by exact analysis of the Boolean equations implemented by the network may be overlooked. However, an assignment which could lead to a conflict will never incorrectly be identified as nonconflicting.

The following examples illustrates a circuit in which the tendency lists correctly identify all nonconflicting assignments in one gate-level circuit (example 6.5), but miss some in a slightly different gate-level implementation of the same function (example 6 6)

Example 6.5: Although no necessary assignments are identified when targeting fault E_{s1} in Fig. 6.4, voting at justification points $E_{\{0\}}$ and $L_{\{\overline{D}\}}$ determines that desirable input



a) Example circuit

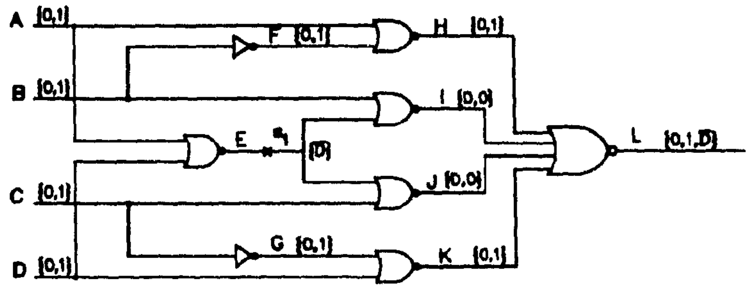
Line	List	Contents	Line	List	Contents
A	T_0^A T_1^A	$\{A_{\{0\}}\}$ $\{A_{\{1\}}\}$	B	T_0^B T_1^B	$\{B_{\{0\}}\}$ $\{B_{\{1\}}\}$
C	T_0^C T_1^C	$\{C_{\{0\}}\}$ $\{C_{\{1\}}\}$	D	T_0^D T_1^D	$\{D_{\{0\}}\}$ $\{D_{\{1\}}\}$
E	T_0^E T_1^E	$\{A_{\{1\}}, D_{\{1\}}\}$ $\{A_{\{0\}}, D_{\{0\}}\}$	F	T_0^F T_1^F	$\{B_{\{1\}}\}$ $\{B_{\{0\}}\}$
G	T_0^G T_1^G	$\{C_{\{1\}}\}$ $\{C_{\{0\}}\}$	H	T_0^H T_1^H	$\{A_{\{1\}}, B_{\{0\}}\}$ $\{A_{\{0\}}, B_{\{1\}}\}$
I	T_0^I $T_{\bar{D}}^I$	$\{B_{\{1\}}\}$ $\{B_{\{0\}}\}$	J	T_0^J $T_{\bar{D}}^J$	$\{C_{\{1\}}\}$ $\{C_{\{0\}}\}$
K	T_0^K T_1^K	$\{C_{\{0\}}, D_{\{1\}}\}$ $\{C_{\{1\}}, D_{\{0\}}\}$	L	T_0^L $T_{\bar{D}}^L$ T_1^L	$\{A_{\{0\}}, B_{\{1\}}, C_{\{1\}}, D_{\{0\}}\}$ $\{A_{\{1\}}, B_{\{0\}}, C_{\{0\}}, D_{\{1\}}\}$ $\{A_{\{1\}}, B_{\{0,1\}}, C_{\{0,1\}}, D_{\{1\}}\}$

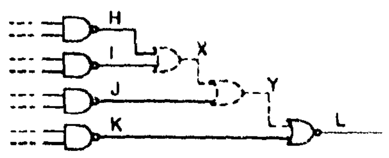
b) Tendency lists for **a**

Figure 6.4 Test generation for E_{s_1}

assignments are $\{A_{\{1\}}, D_{\{1\}}\} \cup \{A_{\{1\}}, B_{\{0\}}, C_{\{0\}}, D_{\{1\}}\}$, resulting in nonconflicting assignments $A = D = \{1\}$, $B = C = \{0\}$, a test pattern for the fault.

Example 6.6: The function of the circuit in Fig. 6.5a is not affected by expanding the four-input *NOR* gate at its output into a cascade of two-input gates (Fig. 6.5**b**). However, the





b) Output gate cascade

a) Schneider's counter-example

Line	List	Contents
H	T_0^H	$\{A_{\{1\}}, B_{\{0\}}\}$
	T_1^H	$\{A_{\{0\}}, B_{\{1\}}\}$
I	T_0^I	$\{B_{\{1\}}\}$
	T_D^I	$\{B_{\{0\}}\}$
J	T_0^J	$\{C_{\{1\}}\}$
	T_D^J	$\{C_{\{0\}}\}$
K	T_0^K	$\{C_{\{0\}}, D_{\{1\}}\}$
	T_1^K	$\{C_{\{1\}}, D_{\{0\}}\}$
X	T_0^X	$\{A_{\{1\}}, B_{\{0,1\}}\}$
	T_1^X	$\{A_{\{0\}}, B_{\{1\}}\}$
	T_D^X	$\{A_{\{1\}}, B_{\{0\}}\}$
Y	T_0^Y	$\{A_{\{1\}}, B_{\{0,1\}}, C_{\{1\}}\}$
	T_1^Y	$\{A_{\{0\}}, B_{\{1\}}\}$
	T_D^Y	$\{A_{\{1\}}, B_{\{0,1\}}, C_{\{0\}}\}$
L	T_0^L	$\{A_{\{0\}}, B_{\{1\}}, C_{\{1\}}, D_{\{0\}}\}$
	T_D^L	$\{A_{\{1\}}, B_{\{0,1\}}, C_{\{0\}}, D_{\{1\}}\}$
	T_1^L	$\{A_{\{1\}}, B_{\{0,1\}}, C_{\{0,1\}}, D_{\{1\}}\}$

c) Tendency lists for **b**

Figure 6.5 Test generation for E_{s_1}

tendency lists are modified (compare T_D^L in Figs 6 4b and 6.5c), resulting in a pessimistic vote and a missed nonconflicting assignment to input B

Pessimism in monotonicity analysis through tendency list computation is brought about by incomplete treatment of fanout-free regions. The problem is that the tendency lists, by treating individual gates, does not determine the effect of individual assignments—the effect is that some input lists are added to the output list more than once. A solution, not developed further in this thesis, is to extract the fanout-free regions of the circuit and perform tendency analysis for the entire region at once.

Chapter 7 The QUEST test pattern generation algorithm

The objective of the QUEST algorithm is to reduce or eliminate backtracking during automatic test pattern generation by delaying arbitrary branching as long as possible. At every step, necessary and nonconflicting assignments are applied iteratively until either a conflict is detected and a backtrack initiated[12], no more algorithmic assignments can be identified and an arbitrary branch is made, or a test vector is generated. The algorithm is complete; the process continues until a test pattern is generated or the target fault is proven untestable.

This chapter also discusses the use of preprocessed information during test generation. There are tradeoffs between the time taken to identify necessary assignments and the number which are overlooked. Unidentified necessary assignments may cause unnecessary branching and backtracking during test generation; however, dynamic reduction list calculation may be costly, even when the techniques described in chapter 5 are used to reduce the amount of processing performed. Thus, from the point of view of CPU time, it may be desirable to obtain information which can be used to identify some necessary assignments quickly and can be reused for each target fault, avoiding the dynamic calculation of reduction lists completely.

Finally, experimental results obtained when the QUEST algorithm was used to generate tests for faults in a variety of benchmark circuits are presented

[12] Necessary and nonconflicting assignments cannot cause a backtrack unless there is no test pattern in the search space defined by the most recent arbitrary branch

7.1 Organization of the test pattern generation system

```
for each target fault
    inject fault;
    do
        do
            forward propagation; /* section 3.1 */
            update justification points; /* section 4.1 */
            if conflict detected
                backtrack;
            else
                calculate reduction lists; /* chapter 4 */
                calculate tendency lists; /* chapter 6 */
                make necessary assignments;
                make nonconflicting assignments;
            endif
        until no algorithmic assignments identified
        if test vector not yet found
            make arbitrary branch (heuristic);
        endif
    until test pattern found or fault proven untestable;
endfor
```

Figure 7.1 The QUEST test pattern generation algorithm

Test pattern generation begins with the insertion of the target fault. To account for the presence of the injected fault, the line (or fanout stem) associated with the fault is broken and a pseudo-primary input/primary output pair is created, as described in section 3.1. Forward propagation is performed to determine the sets of possible values in the circuit in the presence of the injected fault, and test pattern generation begins with the initial set of justification points that the fault must be sensitized and the fault effect must be observed on at least one primary output.

The reduction and tendency lists are computed (updated) to reflect the forward propagated circuit values, and necessary and nonconflicting assignments are identified from the reduction and tendency lists at the justification points. Necessary assignments become new justification points, since their values must be justified in order for the fault to be tested. Forwara propagation is performed to update circuit values to reflect the assignments which were made, and the cycle is repeated until a conflict is detected or no more necessary or nonconflicting assignments are found.

Necessary and nonconflicting assignments may be identified in the restricted space after an arbitrary assignment which were not necessary or nonconflicting before the branch On a backtrack, all assignments which were made after the most recent arbitrary branch must be retracted along with the branch assignment itself. A convenient method to do this is to keep track of the set of justification points (including assigned primary inputs) which existed prior to the arbitrary branch and roll back to it. A change to the justification points initiated by a branch or backtrack can be treated in the same manner as any other justification point modification—node values and reduction and tendency lists must be updated to reflect the new set of justification points.

7.2 Preprocessed information

Two types of reusable information can be recorded during preprocessing and used to reduce the amount of computation performed by the test generator when targeting a particular fault. *Topological information* can be used to limit the work done to calculate reduction lists while guaranteeing that no first-order necessary assignments are overlooked (chapter 5). However, the dynamic computation of reduction lists may be costly, even if these techniques are used. *Generic reduction information* can be used to identify a subset of the necessary assignments, avoiding dynamic reduction list calculation. Two classes of generic reduction information, *generic reduction lists* and *propagate assignments*, are presented in this section. *Dominator identification*, although not generic, is used to trigger the application of propagate assignments and is also discussed.

7.2.1 Generic reduction lists

Generic reduction lists are analogous to "static learning" defined in [SchTriSar88] The generic reduction lists are computed with all the primary inputs assigned to {0, 1} and no injected faults. Due to the conditions under which the lists are computed, the assignments they contain are reduction assignments only for justification points and nodes not reachable from the point of the fault.

For each stem in the circuit, assignments to both 0 and 1 are added to the generic reduction lists, along with candidate gate assignments found using property 5.4. Subsumed stem assignments are not dropped, however, as the gate may be inside the D-region while the subsumed stem is outside for some target fault—there would be no reduction assignment to the gate, despite that the stem assignment is still valid. If subsumed stem assignments are dropped, necessary assignments may be overlooked.

Similar to the discussion presented in chapter 5, memory requirements can be reduced by pruning the generic reduction lists to remove implications which can be captured by backward implication. Property 7.1 formalizes the conditions under which the implication of an assignment to a network node (either a fanout stem or gate) must be stored on the generic reduction list of another node.

Property 7.1: Let $f : X \times Y \rightarrow Z$ be a function, and A, B, and C be nonempty subsets of X, Y, and Z, respectively, $A \subseteq X$, $B \subseteq Y$, $C \subseteq Z$. For each $z \in C$, generic reduction list G_z^C contains all assignments S_v such that $S_v \in \overrightarrow{R}_z^C$ and either $\|f^{-X}_{|A\times B}(z)\| \neq 1$ or $\|f^{-Y}_{|A\times B}(z)\| \neq 1$.

Proof: From theorem 4.1 and the definition of inverse images, only these implications cannot be identified by backward implication. ∎

In order to avoid storing implications which can be found by several iterations of backward implication and generic reductions (see property 5.1), stem assignments need be retained at a network node only if there are multiple disjoint paths from the stem to the node—in other words, the node must be a reconvergence gate of the stem. This implies that, in computing generic reduction lists, assignments to a stem need not be propagated outside its stem region. Similarly, candidate gate assignments need not be propagated outside the area defined by the union of the stem regions of stems for which the gate subsumes at least one assignment.

7.2.2 Circuit dominators

Dominators—nodes through which the fault effect must propagate in order to reach any primary output—are necessary assignments to $\{D, \overline{D}\}$, and can be identified either by backward implication or reduction analysis. To avoid overlooking necessary assignments if the reduction lists are not updated dynamically, dominators not found by backward implication can be identified in linear time using the algorithm from [CoxRaj88], outlined below.

Viewing the circuit as a graph, with vertices and edges corresponding to gates and lines respectively, dominators can be identified by considering only the subgraph whose vertices: *a)* reach a justification point whose required value includes D and/or $\overline{D}$, and *b)* whose set of possible values includes D and/or $\overline{D}$. Vertices whose removal would result in an unconnected subgraph correspond to dominators.

One method to find such gates is to trace backward in reverse levelized order from justification points whose required value includes D and/or $\overline{D}$. The inputs to a gate are placed on a list if D or $\overline{D}$ at that input can be combined with the other input values to produce D or $\overline{D}$ at the gate output. Gates are removed from the list when their inputs are checked. If the list contains a single gate at any point, then that gate is a dominator. Individual justification points are scheduled when their level is processed (*OR*-nodes are treated as a single justification point).

7.2.3 Propagate assignments

The basis of the identification of ***propagate assignments*** is that, in order for the fault effect to propagate from a reconvergent stem to a primary output, it must first reach at least one of the stem's exit lines.

Example 7.1: If stem S from Fig. 7.2 is a dominator, the fault effect must propagate through $e1$, $e2$ or $e3$ (its exit lines) in order to be observed at any primary output. If assignments made earlier in the test generation process make it impossible to propagate

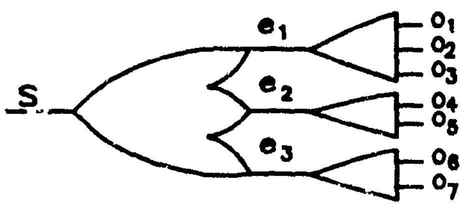


Figure 7.2 Propagate assignment identification

the fault effect through $e2$ (*i.e.* neither D nor $\overline{D}$ is in the set of possible values at $e2$), then it must propagate through $e1$ and/or $e3$—an assignment which eliminates D and $\overline{D}$ from both $e1$ and $e3$ is a reduction assignment.

Definition 7.1: An exit line of a dominant stem is *active* if it is within the shadow of the justification points, and its set of possible values includes D and/or $\overline{D}$.

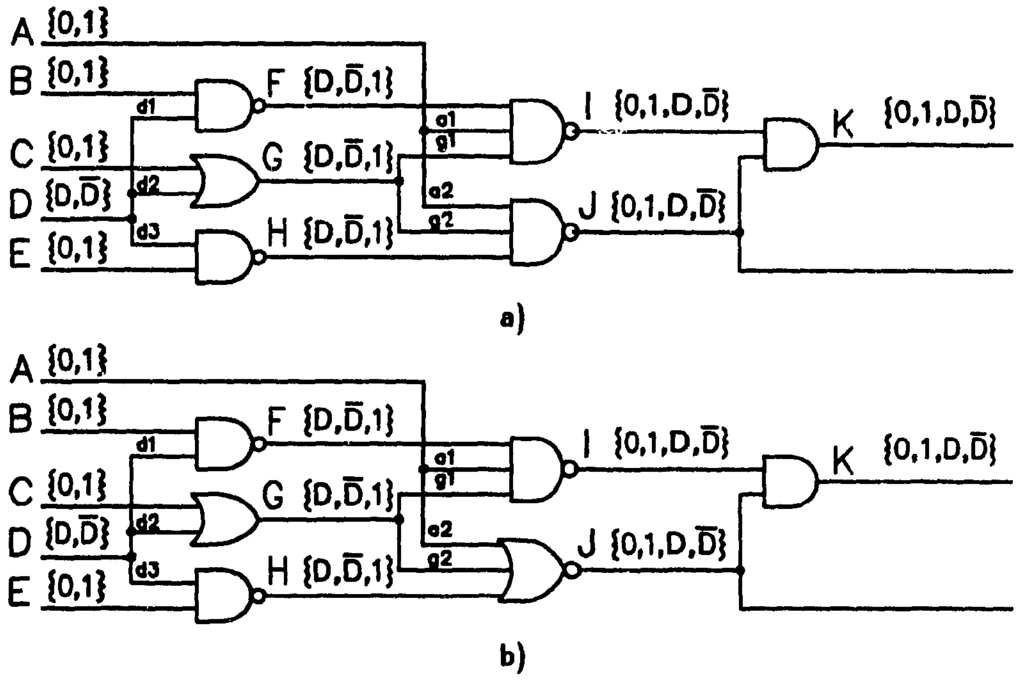


Figure 7.3 Propagate assignments example

Example 7.2: In order to propagate the fault effect from D, an identified dominator, to any primary output in the subcircuit shown in Fig. 7.3a, it must propagate through $j2$ or K, the exit lines of D. A necessary condition to observe the fault effect on either exit line is

$A = \{1\}$. If *NAND* gate J from Fig. 7.3a is replaced by a *NOR* gate (Fig. 7.3b), there are no necessary assignments—the required value of A to propagate the fault effect through exit lines K is $A = \{1\}$ and through $j2$ is $A = \{1\}$. If, however, $B = \{0\}$ and $C = \{1\}$ are assigned earlier during test generation, then the only active exit line of D is $j2$, so $A = \{0\}$ is a necessary assignment.

Propagate assignments for each exit line of each reconvergent fanout stem in the circuit can be found during preprocessing by:

1. Injecting $\{D, \overline{D}\}$ at the stem and calculating the reduction lists inside its stem region. Only assignments to fanout stems are placed on the reduction lists, and only "forward" reduction lists are computed.
2. Recording the intersection of reduction lists $\overrightarrow{R}_D$ and $\overrightarrow{R}_{\overline{D}}$ at each exit line of the stem.

When a reconvergent stem is identified as a dominator during test generation, the propagate assignments for each of its active exit lines are intersected to find the set of assignments which make fault propagation from that stem impossible.

Propagate assignments for nonreconvergent fanout stems can be defined similarly. If a nonreconvergent stem is a dominator, the fault effect must propagate to at least one primary output reached by the stem in order for the fault to be tested. In effect, the "exit lines" of a nonreconvergent stem are its reached primary outputs. Making this assumption, its propagate assignments can be identified during preprocessing and used during test generation as described above.

Propagate assignments are similar to those found by SOCRATES using "instructions 1 and 2" [SchTriSar88] and "dynamic instructions 1 and 2" [SchAut89], except that they are identified once during preprocessing, rather than dynamically during test generation.

7.3 Experimental results

The algorithms presented in this paper were implemented to investigate the behavior of a test generation system which uses necessary and nonconflicting assignments and, in particular, to determine the extent to which backtracking can be reduced using algorithmic rather than heuristic techniques.

Faults can be divided into classes depending on the relative difficulty of finding a test pattern for them. For example, many faults are "random testable" because it is easy to find a test vector for them by fault simulating random input vectors. In addition, experimental results indicate that a test can be generated for many faults using only preprocessed information. However, faults exist—particularly certain untestable faults—which are abandoned unless all necessary assignments are identified using dynamically updated reduction lists. This observation leads to the conclusion that no test generation system is ideal for all target faults. An algorithm which finds all necessary assignments may take excessive CPU time because the dynamic calculation of reduction lists is costly, yet an algorithm which uses only preprocessed information will abandon or spend excessive time branching and backtracking needlessly on some difficult faults. On the other hand, a multi-phase algorithm can combine the best features of all its components.

Complete test pattern generation experiments were run on the ISCAS'85 benchmark circuits [BrgFuj85]. Although the circuits are small, they contain examples of interesting faults which are abandoned by many test generation systems [Goel81, FujShi83, KirMer87, Cheng88]. In order to test the deductive power of the QUEST algorithm, all faults were explicitly targeted—no fault simulation was performed. Experimental results were generated as follows:

Deterministic test pattern generation: A two-phase algorithm was used, with a backtrack limit of 10 for each pass. In order to determine the effect of algorithmic assignments on test generation in the absence of "intelligent" heuristics, the results presented in this section were produced by assigning to $\{0\}$ the first unassigned primary input which could have an effect in the final test pattern

a) *Phase 1:* Test generation using preprocessed information only, but using dominators and propagate assignments.

b) *Phase 2:* Test generation using dynamically updated reduction lists.[13]

Circuit	Testable			Untestable		CPU Time (s)*		
	Flts	Abd.	Undet.	Flts	Abd	Avg.	Max.	Pre.
C432	520	0	0	4	0	0.22	0.45	6.63
C499	750	0	0	8	0	0.21	0.66	3 58
C880	942	0	0	0	-	0.13	0.41	4 51
C1355	1566	0	0	8	0	0.73	1.36	11.81
C1908	1870	0	0	9	0	0.61	1.49	31 75
C2670	2630	32	0	117	0	0.44	2.61	26 88
C3540	3291	2	0	137	0	0.95	2.58	104.68
C5315	5291	2	0	59	0	0 34	1.67	48 02
C6288	7710	71	0	34	0	3.45	6.74	432 18
C7552	7419	28	0	131	0	0.79	3.35	86.24

* Sun 4/SLC

Table 7.1 Experimental results

Table 7.1 summarizes the results obtained, differentiating between testable and untestable faults in the benchmark circuits. For example, C2670 contains 2630 testable faults (after prime fault collapsing [Cha79]), of which 32 were abandoned after the backtrack limit was exceeded. These faults were covered by test patterns generated for other faults, however, so no faults were undetected in the experiment. Further, in all circuits, all of the faults which were abandoned after phases 1 and 2 of the test pattern generation algorithm were covered when less than 1000 random vectors were fault simulated, and thus would not have been targeted in a conventional test pattern generation experiment which begins with random vector fault simulation. C2670 also contains 117 undetectable faults, all of which were proven to be untestable in the experiment. The table also reports the average time required to target a fault and the maximum time required for any target fault in the experiment. The preprocess time includes the time required to perform all operations up to the injection of the first target fault—read the netlist and create data structures, analyse the topology of the network, extract generic reduction information and identify propagate assignments, *etc.*

[13] The implementation used here does not take advantage of the techniques described in chapter 5

No untestable faults were abandoned. Furthermore, phase 2 of the test generation algorithm (full reduction list propagation) was required to prove redundancy only in C432—three faults were abandoned (after 10 backtracks each) by phase 1, after which two of the three were proven untestable with no backtracks and the third was proven untestable with one backtrack in phase 2. Circuit C2670 contains eight faults which each required one backtrack (in phase 1) to prove redundancy; circuit C7552 contains 6 faults which were backtracked, the "worst" of which required 3 backtracks (phase 1). This result indicates that QUEST is particularly efficient at redundancy identification, which is often a problem for conventional test pattern generation algorithms.

Table 7.2a details the branching and backtracking activity during the test generation experiments summarized in Table 7.1. For each phase of the test generation experiment, the table gives the number of faults for which branches (backtracks) were performed, the total number of branches (backtracks) performed in that phase, and the number of faults abandoned in that phase after the backtrack limit was exceeded. Branches and backtracks are not counted for faults which were abandoned. All faults (both testable and untestable) were targeted by phase 1; faults abandoned in phase 1 were targeted in phase 2. For example, 26 of 5350 total faults in C5315 were abandoned by phase 1. Of 5324 faults successfully targeted, arbitrary branching occurred for 4053 of them. 2 faults were abandoned by phase 2; a test pattern was generated for the remaining 24.

Table 7.2b summarizes the results of a second series of experiments, performed to determine the effect of nonconflicting assignments on branching and backtracking during test pattern generation. All faults were targeted a second time using the conditions described above, but without identifying nonconflicting assignments. Applying nonconflicting assignments reduces the amount of branching and backtracking performed during test pattern generation. In addition, one untestable fault in C432 and 11 untestable faults in C2670 are abandoned if nonconflicting assignments are not identified.

Despite the simplistic heuristic used to choose arbitrary assignments, a test is generated for the vast majority of faults without backtracking. Of particular interest is the

Circuit	Phase 1					Phase 2					Un-
	Branch		Backtrack		Abd.	Branch		Backtrack		Abd.	Det
	Flts	Tot.	Flts.	Tot.	Flts.	Flts.	Tot.	Flts.	Tot.	Flts.	Flts.
C432	458	5064	0	0	3	1	1	1	1	0	0
C499	686	22131	31	34	0	-	-	-	-	-	0
C880	706	8423	0	-	0	-	-	-	-	-	0
C1355	1566	50368	86	109	0	-	-	-	-	-	0
C1908	1810	37453	23	29	7	5	68	0	0	0	0
C2670	2074	60225	184	238	34	2	112	2	12	32	0
C3540	2747	43243	250	356	19	17	265	12	29	2	0
C5315	4053	65248	201	318	26	24	538	22	44	2	0
C6288	7624	133835	2155	2587	74	2	19	0	0	71	0
C7552	7085	231477	783	1352	75	17	370	14	14	28	0

a) Nonconflicting assignments identified

Circuit	Phase 1					Phase 2					Un-
	Branch		Backtrack		Abd.	Branch		Backtrack		Abd.	Det.
	Flts.	Tot	Flts	Tot	Flts	Flts	Tot	Flts.	Tot.	Flts.	Flts
C432	520	9802	0	0	3	0	0	0	0	1	1
C499	686	22817	33	38	0	-	-	-	-	-	0
C880	867	16952	0	-	0	-	-	-	-	-	0
C1355	1566	52517	94	125	0	-	-	-	-	-	0
C1908	1860	41399	88	100	7	7	70	0	0	0	0
C2670	2199	73791	184	238	45	2	114	2	12	43	11
C3540	3070	49192	258	383	19	17	300	12	29	2	0
C5315	4754	84044	229	346	27	24	563	22	50	3	0
C6288	7631	137483	2155	2587	74	2	19	0	0	71	0
C7552	7119	291952	797	1358	77	44	592	16	18	28	0

b) Nonconflicting assignments not identified

Table 7.2 The effect of nonconflicting assignments

number of faults for which a test is generated without branching, a result which is independent of heuristics and is heavily influenced by nonconflicting assignment identification.

Missed necessary assignments cause backtracking in test generation. For example, for several faults abandoned by phase 1 of the algorithm, a test was generated without branching by phase 2. Although the number of abandoned faults is reduced by nonconflicting assignment identification, the greatest gains come from the identification of necessary assignments. If all necessary assignments were identified by full reduction list propagation, even fewer backtracks would be performed This approach is practical if reduction analysis is implemented efficiently, exploiting the properties developed in chapter 5.

reduction analysis is implemented efficiently, exploiting the properties developed in chapter 5.

In practice, the heuristic used to choose arbitrary branches has a huge impact on the number of branches and backtracks performed for both testable and untestable faults and affects the number of abandoned faults. It would be interesting to investigate the effect of various heuristics on QUEST. In addition, new heuristics based on algorithmic measures (reduction list lengths, *etc.*) are promising.

Chapter 8 Conclusions

This thesis characterizes three types of assignments made during the course of deterministic test pattern generation. Necessary assignments are those which must be made in order to find a test pattern—the search is guaranteed to fail if they are not made. Nonconflicting assignments lead in the direction of a test and never need to be backtracked, vastly and irrevocably reducing the space which must be searched for a test pattern. Remaining assignments are arbitrary—they may or may not lead to a test pattern and must be reversed is a test cannot be found after they are assigned.

A complete mathematical basis for the identification of necessary and nonconflicting assignments has been developed and algorithms to identify them presented. Issues relating to the efficient implementation of these algorithms have been discussed from both a theoretical and practical point of view. Structural properties of the circuit under test are used to reduce the processing performed to identify necessary and nonconflicting assignments. In addition, several classes of generic reduction information are exploited to identify necessary assignments while avoiding dynamic reduction list computation.

The identification of necessary and nonconflicting assignments is the core of the QUEST test pattern generation algorithm. Experimental results show that QUEST is able to reduce or eliminate backtracking in test pattern generation through algorithmic rather than heuristic means. Results also indicate that QUEST is particularly efficient at *redundancy identification*, which is often a problem for conventional test pattern generation algorithms.

Chapter 9 References

[AbrKul85] M. Abramovici, J.J. Kulikowski, P.R. Menon, and D.T. Miller, "Test Generation in LAMP2: Concepts and Algorithms," *Proceedings International Test Conference*, Philadelphia, PA, Sept. 1985, pp. 49–56.

[AbBrFr90] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, W.H. Freeman and Co., New York, 1990

[Akers76] S.B. Akers, "A Logic System for Fault Test Generation," *IEEE Transactions on Computers*, vol C-25, no. 2, June, 1976, pp. 620–630.

[Ando80] H. Ando, "Testing VLSI With Random Access Scan," *Proceedings COMPCON, Spring 1980*, 1980, pp. 50–52.

[BaMcSa87] P.H. Bardell, W.H. McAnney, and J. Savir, *Built-In Self-Test for VLSI*, Wiley-Interscience, New York, 1987.

[BoHsPu71] W.G. Bouricius, E.P Hsieh, G.R. Putzolu, J.P. Roth, P R Schneider, and C J Tan, "Algorithms for Detection of Faults in Logic Circuits," *IEEE Transactions on Computers*, vol. C-20, no. 12, Nov. 1971, pp. 1258–1264

[BrHaMcSa84] R.K. Brayton, G D Hachtel, C. McMullen, and A Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston MA, 1984.

[BrgFuj85] F. Brglez, and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," *Proceedings International Symposium on Circuits and Systems, Special Sesson on ATPG and Fault Simulation*, Kyoto, Japan, June, 1985.

[ChaDonÖzg78] C. Cha, W. Donath, and F Özgüner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits" *IEEE Transactions on Computers*, vol C-27, no. 3, March 1978, pp 193–200

[Cha79] C.W Cha, "Multiple Fault Diagnosis in Combinational Networks" *Proceedings 16th Design Automation Conference* San Diego CA, June 1979, pp 149 155

[Cheng88] W.T Cheng, "Split Circuit Model for Test Generation" *Proceedings 25th Design Automation Conference*, Anaheim, CA, June, 1988, pp 96–101

[CoxRaj91] H. Cox, and J. Rajski, "On Necessary and Nonconflicting Assignments in Algorithmic Test Pattern Generation," *IEEE Transactions on Computer-Aided Design*. under review

[CoxRaj87] H Cox, and J. Rajski, "A Method of Test Generation and Fault Diagnosis in Very Large Combinational Circuits" *Proceedings International Test Conference*, Washington, D.C., September, 1987, pp. 932–943

[CoxRaj88] H. Cox, and J. Rajski, "A Method of Test Generation and Fault Diagnosis," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 7, July, 1988, pp 813–833.

[EicWil77] E.B. Eichelberger, and T.W. Williams, "A Logic Design Structure for LSI Testing," *Proc. 14th Design Automation Conf.*, June 1977, pp 462-468.

[Eld59] R.D. Eldred, "Test Routines Based on Symbolic Logical Statements," *Journal of the ACM*, vol. 6, 1959, pp. 33-36.

[FujToi82] H. Fujiwara, and S. Toida, "The Complexity of Fault Detection Problems in Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C-31, no. 6, June, 1982, pp. 555–560

[FujShi83] H. Fujiwara, and T Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, vol. C-32, no 12, December, 1983, pp 1137–1144

[FuKaYa89] S. Funatsu, M. Kawai, and A Yamada, "Scan Design at NEC," *IEEE Design and Test of Computers*, vol 6, no. 3, June 1989, pp. 50–57

[GarJoh78] M.R. Garey, and D S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W H Freeman and Co , New York, NY, 1978.

[Goel81] P Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C-30, no 3, March, 1981, pp 215–222

[Golomb87] S.W. Golomb, *Shift Register Sequences*, Holden-Day inc., San Francisco, CA, 1967.

[Harel86] D Harel, "A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems," *Proceedings of the 18th ACM Symposium on the Theory of Computing*, 1986, pp. 185–194

[HorMcL89] P.D Hortensius, R D McLeod, W. Pries, D.M. Miller, and H C. Card, "Cellular Automata-Based Pseudorandom Number Generators for Built-In Self-Test," *IEEE Tranactions on Computer-Aided Design*, vol. 8, no 8, Aug. 1989, pp 842–859.

[HrbJec84] K. Hrbacek, and T Jech, *Introduction to Set Theory, 2nd ed*, Pure and Applied Mathematics, Marcel Dekker, inc , New York, 1984

[JaMoChHa89] R Jacoby, P Moceyunas, H Cho, and G Hachtel, "New ATPG Techniques for Logic Optimization," *Proceedings International Conference on Computer-Aided Design*, Santa Clara, CA, November 1989, pp 548–551

[Kautz68] W H Kautz, "Fault Testing and Diagnosis in Combinational Digital Circuits," *IEEE Transactions on Computers*, vol C-17, no 4, April 1968, pp 352–366

[KirMer87] T. Kirkland, and M R. Mercer, "A Topological Search Algorithm for ATPG," *Proceedings 24th Design Automation Conference*, Miami Beach, FL, June, 1987, pp 502–508.

[Klug88] H.P. Klug, "Microprocessor Testing by Instruction Sequences Derived From Random Patterns," *Proceedings IEEE International Test Conference*, Washington DC, Sept. 1988, pp. 73–80

[LaiSie83] K.W. Lai, and D.P. Siewiorek "Functional Testing of Digital Systems," *Proceedings 20th Design Automation Conference*, Miami FL, June 1983, pp. 207–213.

[Lar89] T. Larrabee, "Efficient Generation of Test Patterns Using Boolean Difference," *Proceedings International Test Conference*, Washington DC, August 1989, pp. 795–801.

[MaaRaj90] F. Maamari, and J Rajski, "A Method of Fault Simulation Based on Stem Regions," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 2, February 1990, pp. 212–220.

[McCMou87] E.J. McCluskey, and S Mourad, "Comparing Causes of IC Failure," *Developments in IC Testing*, edited by D.M. Miller, Academic Press, London, 1987, pp. 13–46.

[Mei74] K.C.Y. Mei, "Bridging and Stuck-At Faults," *IEEE Transactions on Computers*, vol. C-23, no. 7, July 1974, pp. 720–727.

[MinRog89] H.B. Min, and W.A. Rogers, "Search Strategy Switching: An Alternative to Increased Backtracking," *Proceedings International Test Conference*, Washington DC, August 1989, pp. 803–811.

[MuAgND90] F. Muradali, V.K Agarwal, and B. Nadeau-Dostie, "A New Procedure for Weighted Random Built-In Self-Test," *Proceedings IEEE International Test Conference*, Washington DC, Sept. 1990, pp. 660–669.

[Muth76] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Transactions on Computers*, vol. C-25, no. 6, June 1976, pp. 630–636

[NaThAb78] R. Nair, S M Thatte, and J.A. Abraham, "Efficient Algorithms for Testing Semiconductor Random-Access Memories," *IEEE Transactions on Computers*, vol. C-27, no. 6, June 1978, pp 572–576.

[RajCox86a] J. Rajski, and H. Cox, "On the Application of a Transition Logic System to VLSI Fault Analysis " *Proceedings International Symposium on Circuits and Systems*, San Jose, CA, May, 1986, pp. 1265–1268

[RajCox86b] J. Rajski, and H Cox, "Stuck-Open Fault Testing in Large CMOS Networks by Dynamic Path Tracing," *Proceedings International Conference on Computer Design*, Rye Brook, NY, October, 1986, pp 252–255

[Raj88] J. Rajski, "GEMINI A Logic System for Fault Diagnosis Based on Set Functions," *Digest 18th International Symposium on Fault Tolerant Computing Systems* Tokyo, Japan, June, 1988, pp 292–297

[RajCox90] J. Rajski, and H. Cox, "A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation," *Proceedings International Test Conference*, Washington, D.C., September, 1990, pp. 25–34.

[Ravi87] K.W. Ravi, *Imperfections and Imputities in Semiconductor Silicon*, John Wiley and Sons, New York, NY, 1987.

[RobRaj88] M. Robinson, and J. Rajski, "An Algorithmic Branch and Bound Method to PLA Test Pattern Generation," *Proceedings International Test Conference*, Washington, D.C., September, 1988, pp. 784–795.

[Roth66] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, July, 1966, pp. 278–291.

[SaMaTrSc89] T.M. Sarfert, R. Markgraf, E. Trischler, and M.H. Schulz, "Hierarchical Test Pattern Generation Based on High-Level Primitives," *Proceedings International Test Conference*, Washington DC, August 1989, pp. 470–479.

[ScLiCa75] H.D. Schnurmann, E. Lindbloom, and R.G. Carpenter, "The Weighted Random Test Pattern Generator," *IEEE Transactions on Computers*, vol. C-24, no. 7, July 1975, pp. 695–700.

[SchTriSar88] M.H. Schulz, E. Trischler, and T. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Transactions on CAD*, vol. 7, no. 1, January 1988, pp. 126–137.

[SchAut89] M.H. Schulz, and E. Auth, "Improved Deterministic Test Pattern Generation With Applications to Redundancy Identification," *IEEE Transactions on CAD*, vol. 8, no. 7, July 1989, pp. 811–816.

[SeHsBe68] F.F. Sellers, M.Y. Hsiao, and L.W. Bearnson, "Analysing Errors With Boolean Difference," *IEEE Transactions on Computers*, vol. C-17, no. 7, July 1968, pp. 676–683.

[ShMaFe85] J.P. Shen, W. Maly, and F.J. Ferguson, "Inductive Fault Analysis of MOS Integrated Circuits," *Design and Test of Computers*, Dec. 1985, pp. 13–26.

[SilSpi88] G.M. Silberman, and I. Spillinger, "G-Riddle: a Formal Analysis of Logic Designs Conducive to the Acceleration of Backtracing," *Proceedings International Test Conference*, Washington, D.C., September, 1988, pp. 764–772.

[Smith79] J.E. Smith, "Detection of Faults in Programmable Logic Arrays," *IEEE Transactions on Computers*, vol. C-28, no. 11, Nov. 1979, pp. 845–852.

[Stew77] J.H. Stewart, "Future Testing of Large LSI Circuit Cards," *Proceedings 1977 Semiconductor Test Symposium*, Oct. 1977, pp. 6–15.

[Tarjan74] R. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal on Computing*, vol. 3, no. 1, 1974, pp. 61–89.

[ThaAbr80] M.S. Thatte, and J.A. Abraham, "Test Generation for Microprocessor," *IEEE Transactions on Computers*, vol. C-29, no 6, June 1980, pp 429–441

[Wad78] R.L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell System Technical Journal*, vol. 57, May–June, 1978, pp. 1449–1474.

[WaLiRoly87] J.A. Waicukauski, E. Lindbloom, B.K. Rosen, and V.S. Iyengar, "Transition Fault Simulation," *IEEE Design and Test of Computers*, vol. 4, April 1987, pp. 32–38.

[WilAng73] M.J.C. Williams, and J.B. Angell, "Enhancing Testability of Large-Scale Integrated Circuits Via Test Points and Additional Logic," *IEEE Transactions on Computers*, vol. C-22, no. 1, Jan. 1973, pp. 46–60.

[WilBro81] T.W. Williams, and N.C. Brown, "Defect Level as a Function of Fault Coverage," *IEEE Transactions on Computers*, vol. C-30, no. 12, Dec. 1981, pp. 987–988.

[WilPar83] T.W. Williams, and K.P. Parker, "Design for Testability—A Survey," *Proceedings IEEE*, vol. 71, Jan. 1983, pp. 98–112.

[Wun88] H.J. Wunderlich, "Multiple Distributions for Biased Random Test Patterns," *Proceedings IEEE International Test Conference*, Washington DC., Sept. 1988, pp. 236–244.