AN INVESTIGATION INTO A PROBABILISTIC DIGITAL INTEGRATED CIRCUIT TESTABILITY ANALYSIS

Piero Franco

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, 1988

DECLARATION

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

31st day of December 1988

ABSTRACT

Testing has become a major factor in the design of VLSI circuits, and ways to reduce testing costs are needed. This work is an evaluation of a probabilistic definition of testability for scan-designed digital circuits, as an alternative to deterministic test pattern generation. It is shown that probabilistic testing is desirable as it eliminates the need for circuit-specific test pattern generation, and circuits can easily be made self-testable. Probabilistic testing is shown to be feasible since random test lengths are realistic due to the exponential decrease in testing uncertainty with linear increase in cost, and there are usable probabilistic definitions of fault cover. The probability that an arbitrarily chosen Boolean function is testable with equiprobable input probabilities is shown to increase with the size of the function, and for worst-case circuits the test length grows linearly with gate size if optimised input probabilities are used. An exact algorithm is proposed for calculating fault detection probabilities using Shannon's expansion theorem, and a prototype is implemented in Lisp. Although the algorithm is of the same order of complexity as test pattern generation, it replaces test pattern generation, eliminates fault simulation, and since it operates on Boolean expressions, can help formal design verification.

ACKNOWLEDGEMENTS

I am grateful to Professor H. E. Hanrahan for supervising the project, and providing both the necessary computing facilities and software packages. I am grateful to Manfred von Willich for his discussions, particularly in connection with the program OP.LSP in Appendix H. Finally I would like to thank The University of the Witwatersrand and the CSIR for financial support throughout the project.

ACKNOWLEDGEMENTS

I am grateful to Professor H. E. Hanrahan for supervising the project, and providing both the necessary computing facilities and software packages. I am grateful to Manfred von Willich for his discussions, particularly in connection with the program OP.LSV in Appendix H. Finally I would like to thank The University of the Witwatersrand and the CSIR for financial support throughout the project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

Expressions containing both Boolean and algebraic terms are used together in this work. The convention adopted is that uppercase symbols refer to node names or Boolean variables for nodes, and the corresponding lowercase symbols refer to the probability of nodes being logically true, i.e.

a = Pr{node A is true}

The shorthand notation $a' = (1 - a)$ is used for probabilities.

The + and - operators are reserved for algebraic expressions, and the Boolean operators used are:

AND : implied
OR : v
NOT : '
XOR : ⊕

CHAPTER 1

OVERVIEW

1.1 Background

The microelectronics revolution has had a major impact on society and shows no signs of stopping. At the heart is the infamous silicon chip, and the design and manufacture of better and faster VLSI circuits is a major industry.

Ideally digital systems design consists of a specification and a working finished product. Everything covering this vast conceptual gap is just a means to an end, and therefore automation of the design cycle is the subject of much research. The final goal is behavioural synthesis, but the original promise of silicon compilers that produce physical layouts from behavioural descriptions is at present only realized for specific classes of circuits using restricted layouts.

Design verification and testing are two areas in particular where work needs to be done, since the time and cost of testing are now recognized as bottlenecks in VLSI. Due to VLSI complexity, it is becoming increasingly difficult to determine in a cost-effective way whether a logic design performs the required function, and whether a physical device is an error-free implementation of a logic circuit. Moreover, due to the widespread application of microelectronics in critical applications, consumers are now expecting levels of reliability unheard of in other systems of similar complexity, magnifying the importance of thorough testing. Shortcomings of conventional test pattern generation have led to the exploration of alternative design philosophies to reduce test generation and application costs. This work is an investigation into the feasibility of a probabilistic approach to testability analysis, with emphasis on design verification and testing.

1.2 Scope of Work

Testing the whole spectrum of digital circuits is a vast field, and all aspects cannot be covered here. The problems of relevance in testing are dictated by technology.

However the technology is changing so rapidly that it is not uncommon to design for a particular technology and finally implement in another. By analysing circuits at the logical level a balance is achieved between too much technology-dependent detail, and loss of structural detail present at the behavioural level. Therefore the logical level is used throughout this work, and specific technological issues such as parametric testing will not be discussed.

The decision was also made not to include asynchronous sequential circuits explicitly in the analysis. The reason is that asynchronous circuits are essentially untestable, and therefore are used much more rarely than synchronous circuits in VLSI designs.

An investigation into a testability analysis such as this one is by its very nature multi-faceted where instead of a single idea being explored, many interrelated concepts must be developed and evaluated. Probabilistic testing is therefore evaluated by comparing it to deterministic testing from different perspectives.

1.3 Outline of Work

This work is logically and physically divided into three almost self-contained parts, which discuss probabilistic testing at different levels of abstraction. The reason that the three parts are separated is that different techniques are used to investigate each part. As a result Chapter 2 covers fairly broad issues in testing and is written for a general audience, whereas Chapter 3 includes *theoretical issues in testing, and Chapter 4* discusses the development of algorithms. The progression of the work is firstly to show that probabilistic testing is desirable, then feasible, and finally practical.

Chapter 2 starts with a literature survey of the state of the art in the digital integrated circuit design cycle covering test pattern generation, evaluation and application. Based on the review, a probabilistic method is proposed for dealing with VLSI testing problems. The choice of structured design for testability techniques to scan design sequential circuits is motivated, and the tradeoffs are noted. Thereafter, the work focuses on probabilistic testing of combinational circuits.

Chapter 3 addresses the testing problem at a lower level of abstraction. Here design for testability techniques are assumed to have been adopted to scan design sequential circuits so that only combinational testing needs to be considered. The feasibility of probabilistic testing is shown by developing the necessary theory relating test lengths, fault detection

probabilities and fault cover, and theoretically analysing two examples. The first example finds the testability distribution of all Boolean functions to estimate the testability of an arbitrarily chosen circuit. The second example is a variation on probabilistic testing where a worst-case AND gate is tested with optimised input probabilities.

Chapter 4 is once again at a lower level of abstraction and therefore more detailed than the previous parts. Although probabilistic testing was shown to be feasible by general theoretical considerations, here it is also shown to be practical. Algorithms are developed for calculating fault detection probabilities, and optimising input signal probabilities. Various modes of operation of the testability analysis algorithm are described with examples, and the implementation of a prototype written in Lisp is discussed. The programs shown in Appendix H are the result of several iterations to improve efficiency, and constitute a significant fraction of the duration of the work.

Since the calculation of exact fault detection probabilities is known to be a complex problem in general, but approximations can have unacceptable errors, the idea explored in this work is to use the testability analysis to replace other intractable problems in the design cycle. Although the resulting algorithms are of the same order of complexity as deterministic test pattern generation, it will be shown that probabilistic testing also has other benefits in terms of design verification and automation.

CHAPTER 2

DESIRABILITY OF PROBABILISTIC TESTING

2.1 Introduction

The first aim of this chapter is to review the state-of-the-art in the digital integrated circuit design-manufacture-test cycle. Although the problems of importance depend on current technology, advances occur so rapidly that for a review to not be outdated before it is finished, fundamental theoretical constraints must not be overlooked. The second, and more important aim is to use the literature review in the light of modern trends towards design automation, to develop a proposed testing methodology for dealing with the problems in VLSI testing.

The background below covers the design cycle, design verification, and testing. Testers are mentioned and the central testing problem is introduced. Design for testability techniques which attempt to reduce the testing problem are reviewed. Application specific ICs (ASIC) are considered, and finally the limits of VLSI complexity are discussed.

As in most engineering design, the testing methodology proposed makes certain tradeoffs. It is believed that the compromises made are analogous to those made by the users of high level software programming languages. It is now standard practice to write software systems in structured high level languages, and only use assembler when absolutely necessary. Similarly the proposed testing methodology advocates the use of structured scan techniques which should be used whenever possible to simplify testing sequential circuits.

2.2 Design Cycle

The complexity of VLSI has changed the integrated circuit design cycle. Classical design objectives such as minimizing logic gates or flip-flops have been replaced by factors like pin-count, testability, and easily modifiable modular designs.

VLSI design has parallels with writing large software systems, but the problems are generally worse (Sequin, 1983). High level programming languages impose structured design techniques to facilitate software development, but a piece of silicon is essentially an unstructured domain which forces no partitioning or modularisation. Another consideration is that ICs are two-dimensional, and require physical interconnections between parts (i.e. no software "jump to subroutine").

Figures 2.1 and 2.2 show the conventional integrated circuit design and testing cycles. The design engineer is responsible for the verified logical design, while the test engineer generates test patterns and calculates the estimated fault cover. Test patterns can be generated by the designer, but there are manufacturers which insist that a circuit cannot be properly tested by its designer.

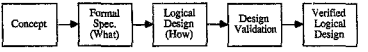


Figure 2.1 : Integrated Circuit Design

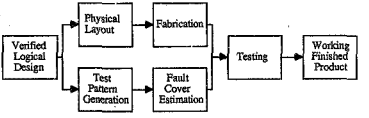


Figure 2.2 : Integrated Circuit Manufacture and Testing

In practice design verification and test pattern generation are iterative processes, and there are two different types of testing. Prototype testing is used to diagnose and locate faults so that errors in the manufacturing process (e.g. lithographic masks) can be corrected. In production testing, however, pass/fail testing is normally adequate and fault location is not required. The testing cost per IC is a major factor in production testing.

It has been found that it is possible to design VLSI circuits that are virtually untestable, thereby making test generation in Figure 2.2 impractical. The only way to solve this problem is to modify the logical design taking testing constraints into account, thereby linking design and test. It is now generally agreed that it is the designer's responsibility to design logic in a manner that is testable. Figure 2.3 shows the integration of design and test in the VLSI design cycle. Design for testability techniques are used to try to anticipate testing problems in the design phase, in order to minimize the need to redesign a circuit due to its poor testability.

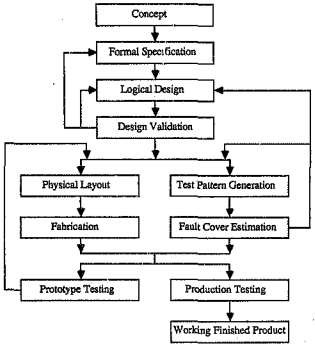


Figure 2.3 : Integrated Design-Test Cycle

Design verification, testing and design for testability are reviewed in more detail below.

2.3 Design Verification

Design verification and testing are two distinct activities. Design verification is aimed at proving that the logical design actually performs its expected function, whereas testing is aimed at showing that a physical device is an error-free implementation of a logic circuit. Therefore bugs in a design should be discovered during design validation and not during simulated test runs.

It is clear that some form of automated design verification is becoming necessary as VLSI complexity increases and synthesis tools are used to aid circuit design. There are many ways to validate a logical design. A simple approach is to simulate all possible inputs to a device in all possible conditions, and exhaustively prove that the device is performing the correct function (perfect induction). For large circuits, however, this method becomes prohibitively time-consuming.

Two extreme approaches to showing that a device "works" are to run the part in its intended application environment, or to formally prove the correctness of the design from its specifications (Sequin, 1983). The drawback with the former method is that it can generally reveal the presence of bugs, but not their absence. When a device is run or simulated in its application environment it is usually difficult to determine which parts of the device have not been tested yet.

Design verification by formally verifying the correctness of a design from its specifications is conceptually sound. However there are many unresolved problems and this method is not practical for arbitrary circuits at present (Sequin, 1983). A formal method of software verification using assertions in predicate calculus has been devised and found useful for small, critical sections of programs. For example, it can be used to prove that a particular loop will terminate, and that upon termination the required results have been achieved.

A simpler kind of verification also needs to be performed. This is often implemented as a rule-based system which check technology-dependent constraints such as fan-in, fan-out, delays etc. It is conceivable that in the long run "silicon compilers" will generate and formally prove designs, but according to Sequin (1983), formal proofs of circuit correctness should not be expected in this century.

Another aspect of formal design verification is that a description of the circuit's function must be expressed in an unambiguous formal notation. Since complex circuits perform

complex functions, formally specifying the circuit's behaviour is in itself a major task. For example the design specifications for the Motorola MC68020 microprocessor are 700 pages long (Daniels and Bruce, 1985).

The conclusion can be drawn that although formal design verification is not possible for general circuits at present, its usefulness especially in automated systems implies that it is a consideration in any design automation or analysis system. The link with the testability analysis is that the proposed testability analysis operates the circuit's Boolean equations, which can form the basis for a formal design verification system.

2.4 Testers

The final element in the design-manufacture-test cycle is the tester. Two categories of tests are performed for logic circuits. The first is logic testing, and the second is parametric testing of circuit speed, voltage and current tolerances, and power consumption. Tests are done both on wafers and packaged ICs after burn-in. Only logic testing will be considered here.

A major testing problem is the high cost of VLSI tester equipment (McCluskey, 1984). Furthermore, according to Barber and Satre (1987) the simple, yet devastating truth about modern automatic test equipment, is that it has not been able to keep up with the advancing technology of the VLSI circuits it must test.

In the near future, three major challenges face automatic tester equipment (ATE) manufacturers. The first challenge is the problem of testing Very High Speed bipolar and Gallium-Arsenide integrated circuits (VHSIC). It is estimated that 500 MHz testers are needed for GaAs integrated circuits, although the fastest testers available at present operate at about 100MHz. Wafer Scale Integration is posing many problems, including the testing and connecting of redundant modules.

The third problem is the transmission line effects that exist between the IC under test and the detection circuits. Coaxial cables leading to the detection circuits load the test IC's outputs, and cause reflections due to impedance mismatches. Transmission line effects can be reduced by lowering the output impedance of the IC's output drivers or shortening the coaxial cables. However the former causes excess power dissipation in CMOS circuits, and the latter is difficult due to the large number of test probes necessary for VLSI circuits.

It can be seen that not only is high operating speed a requirement for high speed testers, but resolution, timing flexibility and driver integrity are also required. Furthermore there is always the problem that by definition testers will never be fast enough to test the highest speed ICs at their operating speed. In developing a testing methodology for VLSI circuits, the above discussion has shown that there are significant advantages to being able to eliminate the need for expensive testers.

2.5 The Central Problem in Digital Testing

In an area growing as rapidly as VLSI circuits, the theoretical computational complexity of testing must be considered since it imposes a bound on the efficiency of any testing methodology. Simply stated, testing digital circuits is hard.

In order to qualify the previous statement, some results of complexity theory of algorithms are needed. In combinatorial computing, a problem is said to be easy if a polynomially-bounded algorithm has been found for its solution (Carre, 1979). Conversely a problem is said to be hard if the existence of a fast algorithm for its solution implies that the NP-complete problems are easy. A good account of the theory of NP-completeness can be found in Garey and Johnson (1979).

The theory of NP-completeness is perhaps the most important theoretical development in recent algorithmic research (Abadir and Reghbati, 1983). Experience shows that many problems have best algorithms for their solution which fall into two groups. The first class (denoted P) consists of problems whose solution is bounded by a polynomial of small degree (e.g. searching, sorting). The second class (denoted NP) consists of problems which can be solved in polynomial time with a non-deterministic algorithm.

A non-deterministic algorithm is one for which a state may determine many next states, and all the next states are explored simultaneously (Golumbic, 1980). NP-problems have the property that if a solution is given, it can be checked deterministically in polynomial time (Beineke and Wilson, 1983).

The class P is contained in the class NP (i.e. $P \subset NP$), but an important open question in the theory of computation is whether the two classes are equal (i.e. is $P = NP$?) (Golumbic, 1980). It is generally believed that $P \neq NP$ (Beineke and Wilson, 1983) due to the existence of a subset of difficult NP-problems called NP-complete problems. A

problem is NP-complete if it is both a member of NP and it is NP-hard. A problem is NP-hard if the existence of a deterministic polynomial-time algorithm for it, would imply the existence of a polynomial-time algorithm for every problem in NP (Golumbic, 1980).

Many results are known for NP-complete problems. A property of NP-complete problems is that any problem in the class is polynomially transformable into any other problem in the class. Therefore if any NP-complete problem can be solved in polynomial time, all other NP-complete problems can also be solved in polynomial time (Beineke and Wilson, 1983).

As yet only exponential time algorithms are known for the solution of NP-complete problems (Golumbic, 1980). However many NP-complete problems have been studied intensively for decades, and no fast algorithms have been found. Therefore it seems likely that no such algorithms exist (Carre, 1979; Beineke and Wilson, 1983).

Returning to digital circuits, Ibarra and Sahni (1975) have shown that the problem of generating tests to detect single stuck-at faults in a combinational circuit is an NP-complete problem. Since there are no known polynomial-time algorithms for solving NP-complete problems, any algorithm generating test patterns for faults in general combinational circuits has a limiting exponential complexity. Therefore using the terminology above, the problem of detecting single stuck-at faults in combinational circuits is hard.

2.6 Testing

The reliability levels expected of modern electronic systems has made thorough testing of integrated circuits imperative. The current state of affairs in VLSI testing is summarized below.

Since tests can be generated and applied for any non-redundant fault in a circuit, all testing problems relate to cost. The time required for exhaustive testing increases exponentially with circuit size, as does the complexity of algorithms generating test sets. Automatic test pattern generation (ATPG) and testing are now recognized as major cost factors in the manufacture of integrated circuits (Wunderlich, 1985; McCluskey, 1984).

problem is NP-complete if it is both a member of NP and it is NP-hard. A problem is NP-hard if the existence of a deterministic polynomial-time algorithm for it, would imply the existence of a polynomial-time algorithm for every problem in NP (Golumbic, 1980).

Many results are known for NP-complete problems. A property of NP-complete problems is that any problem in the class is polynomially transformable into any other problem in the class. Therefore if any NP-complete problem can be solved in polynomial time, all other NP-complete problems can also be solved in polynomial time (Beineke and Wilson, 1983).

As yet only exponential time algorithms are known for the solution of NP-complete problems (Golumbic, 1980). However many NP-complete problems have been studied intensively for decades, and no fast algorithms have been found. Therefore it seems likely that no such algorithms exist (Carre, 1979; Beineke and Wilson, 1983).

Returning to digital circuits, Ibarra and Sahni (1975) have shown that the problem of generating tests to detect single stuck-at faults in a combinational circuit is an NP-complete problem. Since there are no known polynomial-time algorithms for solving NP-complete problems, any algorithm generating test patterns for faults in general combinational circuits has a limiting exponential complexity. Therefore using the terminology above, the problem of detecting single stuck-at faults in combinational circuits is hard.

2.6 Testing

The reliability levels expected of modern electronic systems has made thorough testing of integrated circuits imperative. The current state of affairs in VLSI testing is summarized below.

Since tests can be generated and applied for any non-redundant fault in a circuit, all testing problems relate to cost. The time required for exhaustive testing increases exponentially with circuit size, as does the complexity of algorithms generating test sets. Automatic test pattern generation (ATPG) and testing are now recognized as major cost factors in the manufacture of integrated circuits (Wunderlich, 1985; McCluskey, 1984).

Circuits are expanding faster than testing ability (Bending, 1984), and ATPG is becoming harder with increasing circuit size (Williams and Parker, 1982). Techniques that worked well for MSI such as the D-algorithm do not cope well with VLSI (Abadir and Reghbati, 1983). Developers of ATPG algorithms are cautious, for example Genrad (1985) state that their HILO-3 test generator operates most effectively with structural-level models of simple logic circuits.

Test pattern generation and testing rely heavily on models of possible failures in a circuit. For the tests to be effective the fault model used must be an accurate representation of physical failures. By far the most common fault model used is the single stuck-at (SA) model (Muehldorf and Savkar, 1981). The assumption is made that most failures result in nodes being permanently tied to either logic level. Although there are other fault models such as multiple stuck-at and bridging faults, the single stuck-at model has proved useful in practice, and covers most cases of multiple faults (Muehldorf and Savkar, 1981).

Testing can be either functional or structural, the former based on a behavioural description of the circuit, and the latter on a gate level description. Exhaustive functional tests are impractical. A MC6800 microprocessor, for example, would take 2 million years to test at its operating frequency (Daniels and Bruce, 1985). Exhaustive tests are not feasible for combinational circuits either. To exhaustively test a VLSI combinational circuit with 50 inputs would take over 3 years at 10MHz and 60 inputs would take about 3600 years.

Functional tests which exercise the circuit's major functions generally yield low gate-level fault detection probabilities (Thomas, 1977), and the efficiency of the test is difficult to estimate without much fault simulation. Structural testing usually provides better fault cover with less computation.

Built-In Self-Test

An important area in digital testing is built-in self-test (BIST). BIST schemes use some extra hardware on chip to allow the chip to test itself (see Section 2.7.3). Built in testing can be exhaustive, pseudoexhaustive or random, but each method can suffer from complexity problems.

Exhaustive testing consists of applying all possible inputs to a circuit in all possible states. For combinational circuits the test time grows exponentially with number of

inputs, and for sequential circuits test time also grows at least exponentially with internal latches. As noted above, test times are impractical for VLSI circuits.

Pseudoexhaustive testing involves partitioning a logic circuit by the use of path sensitization, and then exhaustive testing of each partition (McCluskey and Bozorgui-Nesbat, 1981). The weakness with this test methodology in general is that the algorithm to perform the partitioning is computationally complex, since the problem is NP-complete (Illman, 1985).

The third kind of testing is random pattern testing. The test lengths are much shorter than for exhaustive testing (Illman, 1985), but it is difficult to predict the fault cover. The problem of computing the signal probabilities at a node has been shown to be NP-hard by Wunderlich (1985).

To conclude, TPG and testing of VLSI chips is an increasing problem, and ways to simplify testing are needed. An effective approach is to attempt to anticipate testing problems in the design phase in order to design logic in a manner that is testable, as discussed in the next section.

2.7 Design for Testability

The ease with which practically untestable VLSI chips can be designed has moved testing from an afterthought to an integral part of the design cycle (Williams and Parker, 1982). It is now agreed that it is largely the designers' responsibility to design logic in a manner that is testable (Muehldorf and Savkar, 1981). Design for testability techniques are used to "link design and test".

Testability is that property of a circuit which gives an indication of the ease or cost of testing the circuit. Although various definitions of testability exist, Bennetts' (1984) definition below is popular:

> "A circuit is "testable" if a set of test patterns can be generated, evaluated and applied in such a way as to satisfy pre-defined levels of performance, defined in terms of fault-detection, fault-location and test application criteria, within a pre-defined cost budget and timescale."

The three major approaches used to design for testability are testability analysis, ad-hoc techniques, and structured techniques. These methods are described below.

2.7.1 Ad-hoc Techniques

The ad-hoc techniques solve problems for particular designs, but do not generally solve the testing problem for all designs. In some cases general guidelines are given, for example avoiding redundancy and asynchronous logic (Bennetts, 1984), while in other cases rules for partitioning circuits are used.

Common ad-hoc techniques are to partition circuits by breaking connections between modules or feedback paths; introducing extra test points to aid controllability or observability; using bus structures; and compressing test responses with signature analyzers (Williams and Parker, 1982). Signature analysis bridges the gap between the ad-hoc and structured techniques, and is discussed below.

2.7.2 Testability Analysis

Testability analysis consists of applying an algorithm to a circuit to obtain numerical values for its testability, based on the controllabilities and observabilities of the circuit's nodes.

The two most important testability measures are SCOAP and CAMELOT (Illman, 1985). Others, including TMEAS, TEST/80 and TESTSCREEN are briefly reviewed in Bennetts (1984). SCOAP (Goldstein, 1979) characterizes testability in terms of six cost functions, separated into combinational and sequential components. The controllabilities and observabilities are calculated using an iterative algorithm. CAMELOT (Bennetts, 1984) uses two cost functions, the controllability and observability, and a "Transfer Factor" approach to relate changes in values across components.

A weakness of algorithmic testability measures is that they are not a true reflection of the ease of testing a circuit (Savir, 1983). The problem is that in general no conclusion about the testability of a circuit can be reached by using a function of the controllability and observability values. (Common definitions use the product, square root of sum of squares, or geometric mean of the controllability and observability.)

For example, consider testing an internal node in a combinational circuit for a stuck-at 0 fault. Firstly the node must be forced to the opposite value of the fault. Therefore the controllability of the node is the fraction of the possible input vectors that set the node to 1. Secondly a path from the test node to an output must be sensitized so that the value of the node can be observed. The observability of the node is the fraction of input vectors that sensitize a path in the circuit.

Similarly the testability of the node is the fraction of possible input vectors that detect the fault at the node. Since testability depends on both forcing the node to a logic value and sensitizing a path, it is the intersection of the controllability and observability sets. Even if the controllability and observability sets are large, if the two sets are almost disjoint, the testability is very low (Savir, 1983).

In general if the sum of the controllability and observability is less than or equal to unity, the testability can be 0. The testability ranges are summarized in Table 2.1.

Table 2.1 : Testability ranges given Controllability and Observability

Controllability and Observability	Testability
$CY + OY \leq 1$	$0 \leq TY \leq \min(CY,OY)$
$CY + OY > 1$	$CY + OY - 1 \leq TY \leq \min(CY,OY)$

Although the algorithmic testability measures discussed in this section are not exact, they can provide useful information if interpreted with care and good engineering judgement (Agrawal and Mercer, 1982). However, in the trend towards automated design systems, it makes more sense to use an intrinsic measure of testability that is more accurate.

2.7.3 Structured Techniques

In contrast to the ad-hoc approach, the structured techniques are directed at solving the general sequential circuit testing problem (Williams and Parker, 1982). These methods have wider application, but higher cost than the ad-hoc techniques. The objective is to reduce the sequential complexity of a circuit to aid test generation. The method adopted

is that if the circuit's latches can be controlled and observed, the testing problem reduces to one for combinational circuits.

Conventionally, circuits designed with structured design for testability techniques have extra control lines to connect the internal latches together into a shift register. In test mode a vector is shifted into the latches serially, then clocked through the combinational part of the circuit, and finally shifted out of the circuit serially to be observed. Variations in the implementation of the shift register latches include Level Sensitive Scan Design, Scan Path, Scan/Set Logic and Random Access Scan (Williams and Parker, 1982).

A disadvantage of the structured techniques mentioned above is that testing is serialized. For L internal latches, L+1 clock cycles are required for each vector in a test set. This problem can be solved by generating the test vectors and compressing the results on chip. Linear feedback shift registers (LFSR) are commonly used in BIST circuits both as pseudorandom number generators and signature analyzers as seen below.

A representative example of a BIST technique is Built in Logic Block Observation (BILBO) (Bennetts, 1984). The circuit is divided into two BILBO registers and two combinational circuits as shown in Figure 2.4. A typical four bit BILBO register is shown in Figure 2.5.

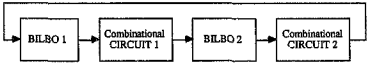
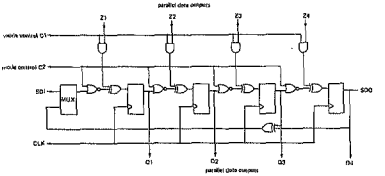


Figure 2.4 : General BILBO Structure

The BILBO element has four modes of operation selected with control inputs. In normal mode the latches are independent and function as an ordinary register, and in shift register mode the latches form a register as in Level Sensitive Scan Design. In LFSR mode the BILBO element is configured to act as a signature analyzer or a pseudo-random number generator.



Control Inputs: C1=1 C2=1 : Normal Mode
C1=1 C2=0 : LFSR Mode
C1=0 C2=1 : Reset Mode
C1=0 C2=0 : Shift Register Mode

Figure 2.5 : Structure of a BILBO element (Source Bennetts, 1984)

Combinational Circuit 1 in Figure 2.4 can be tested in two ways. The first method is similar to Level Sensitive Scan Design. A test vector is shifted serially into BILBO 1, then clocked through the combinational circuit into BILBO 2, and then shifted serially out of BILBO 2. The second method is to configure BILBO 1 as a pseudo-random number generator and BILBO 2 as a signature analyzer. A number of random vectors are clocked through Combinational Circuit 1, and then the contents of BILBO 2 are compared with the expected fault-free "signature". Combinational Circuit 2 is tested by reversing the roles of registers BILBO 1 and BILBO 2.

Scan design techniques have been discussed in detail because it is felt that they are an effective means of addressing the sequential circuit testing problem. Therefore the testability analysis developed and implemented in Chapters 3 and 4 deals exclusively with combinational circuits.

A tradeoff in using the structured design for testability techniques is that the increased controllability and observability has a hardware overhead. However scan design techniques have been used in mainframe computers, and are viable for VLSI circuits. A limitation of the structured techniques is that although sequential circuits are reduced to

combinational circuits, the problem of testing the large combinational circuits possible with VLSI is not addressed.

2.8 Application Specific Integrated Circuits

The present importance of design for testability measures in digital systems is a result of many factors. Two factors in particular are the growth in usage of Application Specific Integrated Circuits (ASIC) and the complexity attainable with VLSI. ASICs are considered in this section, and the physical limits of miniaturization are covered in the next section.

Historically digital designs were implemented in standard logic families using LSI or MSI components. As a result of the microprocessor revolution of the 1970s, many designs were implemented with a microprocessor and a few peripherals. However experience has shown that the microprocessor is not the "solve-all" of digital systems, and other forms of logic implementation have emerged (see Figure 2.6). The term application specific integrated circuit normally refers only to gate arrays and standard-cells, but the original distinctions have become blurred, so the definition is sometimes extended to include programmable logic devices (PLD), thereby covering the whole semi-custom range.

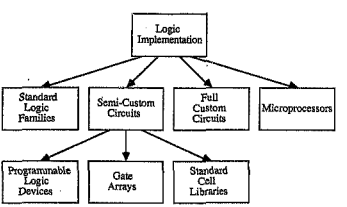


Figure 2.6 : Implementation of Logic Circuits

Two of the main reasons that semi-custom or full-custom ICs have replaced microprocessors and standard logic families in many applications are probably speed and integrity. To realize their true potential, VLSI systems should be designed for as much concurrent action as possible (Sequin, 1983). In custom ICs, concurrent tasks are performed naturally in parallel, at much higher speeds than is possible with essentially serial microprocessors. Large circuits implemented in standard logic families tend to be bulky and unreliable due to many board interconnections between components.

The number of logic designs implemented in custom circuits (especially ASICs) is increasing rapidly. The number of new worldwide ASIC designs is expected to increase from 6000 in 1985 to 90 000 in 199[illegible] (Canyon, 1987). Estimates of the total market share taken by ASICs in the early 1990s range from 20% (Moore, 1984) to 50%, with some consensus at 30% (Waller, 1987). The large ASIC market share has resulted in all major semiconductor houses making large investments in ASICs, particularly in proprietary cell-based ICs including 32-bit microprocessors.

An important trend in ASIC users has been revealed by a formal Intel survey (Waller, 1987). The latest technology and high density are not the most important considerations for prospective ASIC buyers. At the top of the list are quick turnaround times and early working prototypes. In order to be competitive in the cut-throat digital systems business, ASIC users need a short development time for working prototypes, and are prepared to pay for it.

Application specific ICs have highlighted the use of design for testability techniques in two ways. Firstly there is now a greater need for testability measures, and secondly good testability measures are essential to design large testable ASICs.

Designers of microprocessor/standard logic circuits were not greatly concerned with testability since the microprocessors were pre-tested by the suppliers and standard logic circuits have many observability points. Testability was left to a handful of circuit designers involved in full-custom logic design. The advent of semi-custom ICs, however, has brought VLSI design within the reach of many digital systems designers as a cost effective and quick alternative to other logic implementations. Therefore the need for design for testability techniques is now more widespread.

The structured design for testability techniques have been used for standard cells and are particularly useful in gate array designs. The reason is that the size of the gate array on a chip is fixed regardless of how many transistors are actually used in the logic design.

Since extra transistors are manufactured anyway, it makes sense to use them to improve testability.

2.9 Complexity

The fundamental physical limitations of integrated circuits were once only theoretical considerations, but with the ongoing exponential growth in circuit complexity, many of the issues will soon be of practical importance. The most important limitations are discussed below to determine their effects on the longer term prospects for VLSI circuit testing.

The maximum size of a VLSI chip is limited by the probability of a manufacturing fault becoming prohibitively high. However there are techniques for increasing the effective chip area, for example Wafer Scale Integration (WSI) consists of duplicated modules on a wafer which are interconnected after testing. On the other hand, the miniaturization of components on the chip is also limited. There are economic constraints and problems with manufacturing processes, but more importantly, fundamental thermodynamic and quantum mechanical limits are being approached (Keyes, 1975).

Two of the most important considerations are dielectric breakdown and electromigration. Dielectric breakdown by tunneling or avalanche mechanisms limits the minimum thickness of depletion layers that support potential differences, and electromigration damage to conductors limits the maximum current density that can be carried.

The third consideration is the removal of heat generated by each device on a chip. In terms of the energy required to store one bit, conventional semiconductors use about 4×10^{-11} Joules. This is 10^{10} kT in thermodynamic terms, where k is the Boltzmann constant and T is the absolute temperature.

Table 2.2 : Energy used to store one bit

	Energy to store one bit
Conventional semiconductors	10^{10} kT
Practical limits on conventional semiconductors	10^{6} kT
Josephson junction superconductors	5×10^{4} kT
Quantum mechanical limit	~ kT

Table 2.2 was derived from Keyes (1975) , and shows the energy required per logical operation for different processes. Superconductors require five orders of magnitude less energy than conventional semiconductors, but are still far above the quantum mechtanical limit. Major advances in ceramic superconductor materials have been made recently, and high temperature superconducting ICs are becoming possible.

At present the smallest feature size on commercially available VLSI is about 1μm. For example, the Intel and Motorola state-of-the-art 32-bit microprocessors use 1,5μm and 2μm lithography respectively (Gelsinger, 1987; Daniels and Bruce, 1985). The fundamental physical limits noted above become important at about 0,25μm.

From a designer's perspective, the net result is that the number of components per chip is expected to continue growing exponentially for another decade (Moore, 1984), due to a combination of increased miniaturization, wafer scale integration, and new physical implementations of logic functions.

Another aspect of complexity is the usefulness of the devices produced. There is a growing concern that the potential of VLSI cannot be exploited fully unless new ways of managing the vast amount of information needed are developed. For example Gordon Moore (1984), chairman of Intel, is of the opinion that if a million component commercial technology existed, we probably would not know what to do with it.

Therefore complexity of VLSI will eventually be limited both by the ability to design useful circuits, and by physical constraints. However current testing problems can still be expected to worsen for another decade, making investments in structured testing methodologies which are useful now, virtually essential in the future.

2.10 Conclusion and Recommendations

Sections 2.2 to 2.9 have reviewed some of the stepping stones between a specification for a logic circuit and a finished working product. The review shows that testing has become a major factor in integrated circuit design due to VLSI complexity, and automatic test pattern generation is increasingly difficult. Although various design for testability techniques exist to help linking design and test of digital circuits, the structured techniques have the greatest hardware overhead, but are the most general and lend themselves to automation. Moreover, structured scan design techniques can be used for built-in self-test, and therefore solve many tester problems.

Based on the conclusions drawn in this chapter, the following general methodology for testing digital circuits is proposed:

1. The single stuck-at fault model can be used since it is sufficient in most cases. Extensions to other fault models should be possible if necessary.

2. Ideally design verification should be performed especially in an automated design cycle. An exact formal description of the circuit is needed for design validation. It is reasonable to store a description of the circuit's behaviour and structure in Boolean expressions, since they are familiar to the designer.

3. Structured design for testability techniques should be used wherever possible to convert sequential circuits into combinational circuits, and the testability of the combinational circuits should then be analysed with an algorithmic measure.

4. Circuits should be designed with built-in self-test features where the extra hardware overhead is available, so that the chips can partially test themselves both on the wafer before dicing, and during burn-in, eliminating the need for expensive logic testers.

5. BIST circuits should be tested with pseudo-random patterns since exhaustive testing takes too long.

6. By calculating fault detection probabilities accurately, there is no need to do fault simulation for random testing.

Even though the testing problem is complex in general, it is essential that VLSI chips are tested. The approach to testing proposed above is to use random pattern testing, with accurate calculation of fault detection probabilities to find the fault cover. Calculating fault detection probabilities is as difficult as generating tests for faults, but it is more useful for built-in self-test. The reason is that in most cases only random patterns can be generated on chip, so even if deterministic test sets are found they would be difficult to apply. Given the characteristics of the pseudo-random number generator on chip, fault detection probabilities can be used to determine the length of the test necessary to have a certain confidence of detecting faults in the circuit.

Furthermore an analytical method using Boolean expressions is proposed for finding the fault detection probabilities. A disadvantage is the symbolic manipulation of expressions

needed, but the ability to manipulate expressions combined with a Boolean algebra representation of the circuit are surely prerequisites for formal design verification.

The next chapter investigates the theory of probabilistic testability analysis, and Chapter 4 discusses the implementation of the algorithm for calculating fault detection probabilities.

Finally one cannot resist the temptation to draw analogies between digital circuit testing and modern physics. In a quest to successively refine a deterministic theory of matter, it was found necessary to revert to a probabilistic model based on uncertainty principles. Is it not possible then, that in trying to test increasingly complex circuits, a probabilistic approach might not only be useful, but also the only practical solution?

needed, but the ability to manipulate expressions combined with a Boolean algebra representation of the circuit are surely prerequisites for formal design verification.

The next chapter investigates the theory of probabilistic testability analysis, and Chapter 4 discusses the implementation of the algorithm for calculating fault detection probabilities.

Finally one cannot resist the temptation to draw analogies between digital circuit testing and modern physics. In a quest to successively refine a deterministic theory of matter, it was found necessary to revert to a probabilistic model based on uncertainty principles. Is it not possible then, that in trying to test increasingly complex circuits, a probabilistic approach might not only be useful, but also the only practical solution?

CHAPTER 3

FEASIBILITY OF PROBABILISTIC TESTING

3.1 Introduction

In Chapter 2 it was concluded that probabilistic testing is desirable by considering the advantages over deterministic testing. The most important benefits include:

- No circuit-specific test pattern generation is required.
- No fault simulation is needed to determine the fault cover.
- Simpler testers can be used since the test vectors can be generated during testing, and do not need to be stored.
- Built-in self-test (BIST) schemes can be used since random patterns are easily generated on chip.

The main disadvantage of random pattern testing is the difficulty in determining the length of the test needed to obtain a specified fault cover. It will now be shown that probabilistic testing is theoretically feasible, and a practical algorithm for calculating fault detection probabilities and test lengths is presented in Chapter 4.

The argument for the feasibility of probabilistic testing is based on five considerations:

1. In practice all testing methodologies are probabilistic.
2. It can be shown that for all practical circuits, there are essentially equivalent deterministic and probabilistic definitions of fault cover.
3. For random testing, there is an exponential decrease in testing uncertainty for a linear increase in test length, therefore testing with high confidence intervals can be achieved with reasonable test lengths.

4. For an arbitrarily chosen Boolean function, the probability that the function is testable with equiprobable input probabilities increases with the size of the function.

5. If optimised input probabilities are used, the test length grows linearly with gate size even for worst case circuits such as large AND gates.

These five points are considered in turn in this chapter. Most of the mathematical derivations have been included in Appendices A to E.

3.2 Deterministic versus Probabilistic Testing

Both the deterministic and the probabilistic testing disciplines have their adherent followers, and it has been my experience that many heated discussions arise in assessing the relative merits of the two testing philosophies. Before comparing deterministic and random pattern testing, it is necessary to dispel the misconception that deterministic testing can be used to determine with complete certainty whether a logic circuit is operating correctly.

Consider the following absurd situation of an attempt to test an arbitrary combinational circuit with absolute certainty.

> Testing strategies based on stuck-at fault models are inadequate since some failures in the circuit cannot be modeled as stuck-at faults.
>
> Therefore the obvious testing strategy to determine if the circuit is operating correctly, is to test it for every possible input combination. Test times grow exponentially (about 1 year for 50 inputs and over 1000 years for 60 inputs at 30MHz), but at least all cases are considered.
>
> However in this testing procedure (and all others) the assumption is made that faults were present when testing started, since faults which occurred during the test might not be detected by the remainder of the test.
>
> A simple attempt to rectify the problem might be to repeat the exhaustive test seven times.

This test uses a lot of tester time, but still does not test the circuit with absolute certainty. The reason is that there are some bridging faults which convert the combinational circuit into a sequential circuit. Therefore to test for these *faults it is not sufficient to try* all possible input combinations, but also all combinations in different sequences. Since the circuit can have an arbitrary sequential depth and the internal state is not known, it is not possible to find a test set which will detect these faults.

To make matters even worse, only permanent faults have been considered so far. Intermittent faults can occur at arbitrary times, or can occur in specific transient states of the circuit (e.g. CMOS open circuits which act as antennas and behave as *pattern sensitive faults*), *and cannot be modeled in* general. Some technologies are also prone to timing (delay) faults, and "soft faults" due to electromagnetic interference (e.g. gamma rays).

The scenario just described shows that phrases like "these test vectors achieve 100% stuck-at *fault cover" should not be interpreted as more* than a probabilistic statement about the test. In this work the term probabilistic testing is conventionally used to denote random pattern testing unless otherwise specified.

Neither deterministic testing nor random pattern testing can give the assurance of *absolute testing, therefore in practice* all testing is probabilistic. (This raises the ethical question about the reliability and safety of electronic equipment in critical tasks, but this issue cannot be discussed here. Note that reliability techniques such as redundancy only reduce the uncertainty, but do not solve the problem. In general there is no such thing as a "fail-free" or even a "fail-safe" circuit, since no remedial action can be taken for *undetected* faults.)

The set of all possible faults in a circuit can be represented by a pie chart as in Figure 3.1.

There are three areas in the pie chart in Fig.re 3.1. Sector A represents the fraction of faults in a circuit which are detectable by the single stuck-at model. Sector B is the fraction of faults in the circuit which are not detectable by the single stuck-at fault model, but are *detectable using more complex* models like the multiple stuck-at model. Thirdly, Sector C is the fraction of faults in the circuit which are not detectable by any fault model.

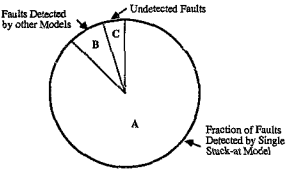


Figure 3.1 : Pie Chart of all possible faults in a logic circuit

Although the exact sizes of the sectors are not known, the theory and practice of digital circuit testing is based on the assumption that Sector C is small, and testing using single stuck-at fault models also assumes that Sector B is small.

Since the single stuck-at fault model has proved useful in practice and is used much more commonly than any other model (McCluskey et.al., 1988), in most cases Sectors B and C are small. There are, however, examples in the literature which show that the single stuck-at model is not always valid.

For the remainder of this work, the single stuck-at fault model will be used. The analysis can be extended to the multiple stuck-at fault model at the cost of increased complexity.

3.3 Deterministic and Probabilistic Fault Cover

Possible probabilistic and deterministic definitions of fault cover are given below. The two definitions are compared with respect to the probability of failing a circuit that does not work. The single stuck-at fault model is used, and it is assumed that all faults are equally likely in the derivations in Appendix A.

For deterministic testing it is common to define the fault cover (c) as the fraction of stuck-at faults in a circuit which are tested by a specific test set. The probabilistic definition of fault cover used here, however, is slightly different. Each stuck-at fault in the circuit is tested with a confidence interval c, therefore the probability of failing a node

that does not work is c. If the process is repeated for all faults, and the length of the random test set is chosen such that all faults in the circuit are tested with at least confidence c, then c may be defined as the fault cover.

For testing it is more important to know the probability of *failing a circuit that does not* work than the fault cover. Therefore the deterministic and probabilistic definitions of fault cover must be compared with respect to the corresponding probability of failing a circuit that does not work.

For a circuit containing n stuck-at faults with a probability w that the node at each fault site works, the equations for the probability of failing a circuit that does not work are derived in Appendix A and repeated below.

$$\Pr\{\text{circuit failing/circuit faulty}\}_{\text{determ}} = \frac{1 - w^{cn}}{1 - w^{n}} \qquad \ldots (3.1)$$

and

$$\Pr\{\text{circuit failing/circuit faulty}\}_{\text{prob}} = \frac{1 - (1 - c(1-w))^{n}}{1 - w^{n}} \qquad \ldots (3.2)$$

where c is the fault cover defined above.

Although equations (3.1) and (3.2) have different forms, it is shown in Appendix A that for all practical circuits (i.e. w very close to 1), the equations are essentially equivalent. Therefore the same probability of failing a circuit that does not work is obtained whether a fraction c of the nodes in the circuit are tested deterministically, or all the nodes are tested with confidence c probabilistically. Therefore within the framework of the single stuck-at fault model, the deterministic and probabilistic definitions of fault cover are practically identical.

3.4 Relation between Test Length and Fault Detection Probabilities

A mathematical relationship is derived in this section between the fault detection probability of a node, and the length of the random test set necessary to detect the fault with a specified confidence interval. The relationship depends both on assumptions about the test setup, as well as the type of testing strategy adopted.

There are many possible testing strategies.

The approach used in a final year design (Franco, 1986) was to find the probability of the output of the circuit being true under both normal conditions, and with a specific fault at a node. The difference in probabilities was related to the number of test vectors necessary to distinguish between the two modes of circuit operation with a given confidence. The fundamental assumption made in this testing strategy, is that for given input probabilities to a circuit, an internal fault will cause the output's probability to change.

This testing approach is very simple to apply in practice, but it is shown below to not be a very efficient form of random pattern testing. However it was investigated because it is worth consideration, both as a comparison with other random pattern testing strategies, and for its connection with the deterministic concepts of testing by transition counting and syndrome testing. From now on this type of testing will be referred to as "probabilistic syndrome" testing. The details of probabilistic syndrome testing are included in Appendix B, which contains both a brief derivation for the test length required to detect a fault with a specified confidence, as well as a parallel of the AND gate example considered in Section 3.6.

Probabilistic syndrome testing suffers from two drawbacks. The first is that the test is not very effective since only long-term probabilities are measured, and not short-term changes resulting from particular random patterns. The second drawback is that some faults cannot be detected by considering only long-term probabilities. Functions with this property are referred to as syndrome untestable in deterministic syndrome testing.

It is easy to find circuits with faults that cannot be detected by probabilistic syndrome testing. For instance consider a circuit where the output is the XOR of two internal nodes *X* and *Y*. *Assuming that the probabilities of nodes* X and Y are 1/2 and independent, the output probability under normal operation will also be 1/2. It can be shown that for faults X or Y stuck-at 0 or 1, the output probability remains 1/2, thereby making these faults undetectable by the analysis above.

However the output does change, because different faults make it true for different inputs, even though the long-term probability remains the same. To cater for this type or problem, a more effective testing strategy is needed. To improve the situation, the "test output" should be a function of the inputs and outputs of the circuit, and not the outputs of the circuit alone. The conceptual test layout is depicted in Figure 3.2.

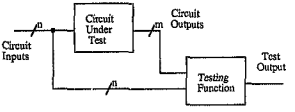


Figure 3.2 : Circuit testing using a testing function

By a suitable choice of the testing function in Figure 3.2, all non-redundant faults will change the signal probability of the test output. In general, the greater the number of circuit inputs used in the testing function, the more useful it is. The trade-off is that the complexity of the circuit to be analysed increases.

The best testing function is an identical working replica of the circuit under test (commonly called the gold unit), with the corresponding outputs of the two circuits XORed together. All the outputs of the XOR gates can then be ORed to give a single test output. The advantage of this testing function is that if a single circuit output differs from its expected value once, it can be detected, which is clearly the most efficient form of testing. Therefore testing no longer relies on the long-term probabilities of the outputs changing under fault conditions, but only the effect of a fault altering an output once. The corresponding test setup shown in Figure 3.3 is frequently used in practice for both deterministic and random pattern testing, and will therefore be the model used in the derivations below. (The working replica of the circuit under test is sometimes eliminated for practical reasons, and the circuit's outputs are compared to stored fault-free responses, but the setups are identical in terms of logic testing.)

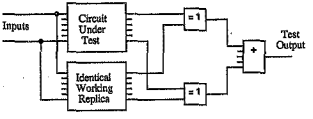


Figure 3.3 : Testing function using an identical working replica of the circuit under test

For random inputs to the circuit in Figure 3.3, any non-redundant fault in the circuit under test will cause the test output to have a finite probability of being true. Thus the testability of the node with the fault will be defined as

> "the number of test vectors necessary to have a predefined confidence that the fault will cause the test output to be true at least once".

The relationship defining the nodal testability or fault detection probability value will now be derived.

3.4.1 Derivation of the Test Length Equation

For a particular fault F in a circuit, let the probability that the test output is true be f. Therefore f will be called the fault detection probability for the fault F. (The term fault detectability is sometimes used to denote the number of inputs which detect the fault = $f \cdot 2^L$.) Statistically, if the event is an input vector to the circuit, the probability of a favourable outcome (i.e. fault detected) is f. The number of times N the event has to be repeated in order to have a confidence c that at least one favourable outcome has occurred, needs to be found. The derivation only uses elementary probability theory and appears often in the literature.

If the probability of a favourable outcome of the event is f, then (1-f) is the probability of an unfavorable outcome. Since for random tests the events are independent, for N events

$$\Pr\{\text{no favourable event has occurred}\} = (1-f)^N$$

therefore

$$\Pr\{\text{at least one favourable outcome has occurred}\} = c$$

$$= 1 - (1-f)^N \qquad \ldots(3.3)$$

Equation (3.3) defines the relationship between the length N of the random test set and the fault detection probability f for the test setup in Figure 3.3.

For comparison, the corresponding equations for probabilistic syndrome testing derived in Appendix B are repeated. For mode of circuit operation A with output probability a,

and mode E with output probability b, the number of test vectors required N is given by equation (3.4).

$$N = \frac{r + Z\sqrt{r(1-b)}}{b} = \frac{r - Z\sqrt{r(1-a)}}{a} \quad \ldots (3.4)$$

where

$$r = \left(\frac{Z(a\sqrt{1-b} + b\sqrt{1-a})}{b-a}\right)2 \quad \ldots (3.5)$$

with b > a, and for a confidence interval c, Z is found from the error function c = Erf(Z).

3.4.2 Assumptions Implicit in Deriving the Test Length

Equation (3.3) is accurate if the input vectors to the circuit are truly random, and a single 1 in the test output is detected. This model was used because it is the most general. However for many of the BIST techniques described in Section 2.7.3, linear feedback shift registers (LFSR) are used both to generate pseudorandom test vectors, and to compress the successive outputs of the circuit under test for signature analysis. The errors introduced into the equation for the test length by both LFSRs are noted below.

Pseudorandom Tests

The difference between random tests and pseudorandom tests generated with a LFSR is that in the latter the patterns do not repeat until all patterns have occurred. In statistical terminology random patterns correspond to Bernoulli trials or "sampling with replacement", while pseudorandom testing corresponds to "sampling without replacement" (McCluskey et.al., 1988). The effect is that the test lengths predicted by the random model are slightly longer than necessary for given confidence intervals if LFSR generated pseudorandom patterns are used for testing. (Models for pseudorandom testing are discussed in (McCluskey et.al., 1988).)

Aliasing in LFSRs

Whereas the previous error resulted in a pessimistic estimate of the test length, aliasing in LFSRs results in an optimistic estimate. Signature analysis is based on the assumption that after the test is completed, the contents of the LFSR signature analyzer

will differ for working and faulty circuits. However, since the LFSR can only be in one of a finite number of states, there is a non-zero probability that the faulty circuit will have the same signature as the fault-free circuit. This condition is referred to as aliasing. Bounds on aliasing errors are found in (Williams et.al., 1988), and by suitable choice of LFSR length and feedback polynomial, aliasing errors can be reduced.

3.5 Testability Distribution of Boolean Functions

The aim of this section is to calculate the testability of all logic circuits with a certain number of inputs, in order to obtain a distribution of the difficulty of testing logic circuits. From the testability distributions (profiles) general conclusions can be drawn about whether circuits are random pattern testable on average.

The exact testability distribution of logic circuits cannot be calculated, therefore it was estimated from the testability distribution of the inputs of Boolean functions using certain assumptions.

3.5.1 Defining the Testability Distribution of Boolean Functions

To determine the testability of a circuit, the fault detection probabilities of all possible faults in the circuit must be found. The testability of the circuit is defined as the lowest fault detection probability, since if the random test set is long enough to detect the most difficult fault with a certain confidence, the other faults will also be detected with at least the same confidence. The single stuck-at fault model is used in this section.

When determining the testability of a circuit using the single stuck-at fault model, only failures in the circuit representable by stuck-at 0 and 1 faults for every node in the circuit are considered. Furthermore it can be shown that only faults at the circuit's inputs and internal fanout nodes need to be tested, since the other nodes are taken into account by *fault collapsing*.

The fault detection probability of internal fanout nodes depends on the gate level realization of the circuit, but the fault detection probability of faults at the circuit's inputs depends only on the Boolean function of the circuit. However, since there are infinitely many realizations of the same function, it is impossible to find the testability of every logic circuit. Therefore it is proposed to estimate the testability distribution of i-input

circuits (see Figure 3.4) by finding the testability distribution of faults at the input nodes of i-input Boolean functions. (It is only an estimate since *internal faults at the fanout nodes in the circuit are not considered.*) The number of possible i-input Boolean functions is finite, therefore the testability of every function can theoretically be found.

Note that the testability distribution of the inputs of Boolean functions is essentially a realization-independent distribution of the testability of logic circuits. Therefore the approximation used corresponds to the well known functional testing approach.

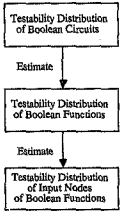


Figure 3.4 : Model for Testability Estimate

As an illustrative example, the testability of all 2 input Boolean functions based on equiprobable 0/1 *random test patterns* has been found exhaustively, and the results are shown in Table 3.1. The fault detection probability and testability distributions are plotted in Figure 3.5.

However the 6 entries marked with an asterisk in Table 3.1 have at least one input with a fault detection probability of 0, which means *that they are not strictly* 2-input functions. Therefore the distributions shown in Figure 3.5 depend on whether all 2-input functions or only strictly 2-input functions are used in the definition of the testability distribution. This discrepancy is shown below to decrease quickly for larger functions by considering the fraction of i-input functions depending strictly on i inputs.

Table 3.1 : Testability of 2 input gates with equiprobable input signal probabilities

	Function	F(A)⊕F(A')	F(B)⊕F(B')	FDP-A	FDP-B	Testability	
0	0	0	0	0	0	0	*
1	AB	B	A	1/4	1/4	1/4	
2	AB'	B'	A	1/4	1/4	1/4	
3	A	1	0	1/2	0	0	*
4	A'B	B	A'	1/4	1/4	1/4	
5	B	0	1	0	1/2	0	*
6	A⊕B	1	1	1/2	1/2	1/2	
7	AvB	B'	A'	1/4	1/4	1/4	
8	A'B'	B'	A'	1/4	1/4	1/4	
9	(A⊕B)'	1	1	1/2	1/2	1/2	
10	B'	0	1	0	1/2	0	*
11	AvB'	B	A'	1/4	1/4	1/4	
12	A'	1	0	1/2	0	0	*
13	A'vB	B'	A	1/4	1/4	1/4	
14	(AB)'	B	A	1/4	1/4	1/4	
15	1	0	0	0	0	0	*

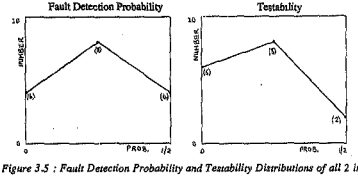


Figure 3.5 : Fault Detection Probability and Testability Distributions of all 2 input Boolean functions

Let a(i) be the number of functions depending on strictly i inputs.

Therefore a(i) is the total number of i-input functions minus all possible combinations of strictly i-1 input functions, minus all possible combinations of strictly i-2 input functions, etc.

The number of strictly j input functions is a(j), and the number of combinations of j-input functions in a function of i inputs is $\binom{i}{j}$ (binomial coefficient).

Therefore since the total possible number of functions is 2^{2^i}, we have a recursive definition for a(i) given by equation (3.6).

$$a(i) = 2^{2^i} - \sum_{j=0}^{i-1} \binom{i}{j} a(j) \qquad \ldots (3.6)$$

The ratio of a(i) to 2^{2^i} is tabulated in Table 3.2 below.

Table 3.2 shows that from 5-input functions onwards, it makes virtually no difference whether the definition of the testability distribution is based on all i-input or only strictly i-input functions.

Table 3.2 : Fraction of functions depending on all inputs

i	a(i)	2^{2^i}	Ratio
1	2	4	0.50000000000
2	10	16	0.62500000000
3	218	256	0.85156250000
4	64594	65536	0.98562622070
5	4294642034	4294967296	0.99992426904
6	18446744047940775978	18446744073709551616	0.99999999860

3.5.2 Calculating the Testability Distribution of Boolean Functions

The distributions of fault detection probability and testability of i-input Boolean functions defined above are found in this section for larger circuits.

The distributions were found for 2-input functions in Table 3.1 by exhaustively considering every possible case. However since the total number of functions increases super-exponentially this method is only practical up to 4-input functions. (For example, there are more 8-input functions than electrons in the universe!)

The distributions of larger functions were estimated by two different methods and the results are compared with a chi-square goodness-of-fit test in Appendix C. In the first method a Monte Carlo simulation was done, and in the second method various asymptotic theoretical approximations were derived.

Monte Carlo Simulation

For large functions where exhaustive calculation of the testability distribution is impractical, the algorithm below was used to randomly select a number of functions from the set of possible functions, and calculate their fault detection probabilities and testabilities.

Repeat
  Arbitrarily choose a Boolean function
  Find the fault detection probabilities of all input nodes
  Find the testability of the function
  Update the distribution estimates
Until enough trials for required confidence interval.

According to the Monte Carlo Method, the results obtained were used to estimate the actual testability distribution within certain confidence intervals. Unfortunately the uncertainty in the Monte Carlo Method decreases as the reciprocal of the square root of the number of trials, and therefore long simulations were necessary for good confidence intervals. (Since the simulation time increases exponentially with the number of inputs, one million trials were only done up to 14-input functions.)

Theoretical Fault Detection Probability Distribution

The fault detection probability and asymptotic approximations for the testability distributions of the inputs of Boolean functions are found in this section. The theoretical distributions were derived in order to have a mathematical description of the testability

distributions which could be used for functions with too many inputs for accurate Monte Carlo simulations.

The fault detection probability distribution was found by considering the number of times the output of a function differs from its expected value when there is a fault at an input node. The distribution is shown in Appendix C to be a purely combinatorial problem, and has a binomial distribution.

The approximation to the testability distribution is more complicated, since the fault detection probabilities of the different inputs to the function are interrelated. The testability distribution of an i-input function is shown in Appendix C to be basically the first order statistic of i fault detection probabilities, with the dependencies between inputs having to be taken into account. (A brief summary of order statistics is given in Appendix D.) The four approximations used to estimate the testability distribution are shown in Figure 3.6.

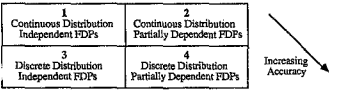


Figure 3.6 : Four asymptotic approximations to the testability distribution of Boolean functions

Although all four approximations to the testability distribution were shown to be asymptotically correct in Appendix C by a chi-square comparison with the Monte Carlo simulation, for small functions the fit is worse for the approximations that assume that the fault detection probabilities are continuous and independent.

None of the approximations are exact because only pairwise correlations between different inputs are considered in approximations 2 and 4 in Figure 3.6. Correlations between groups of 2, 3, 4, ... inputs need to be taken into account to improve the fit for small functions. This slight increase in accuracy at much higher complexity did not seem necessary, since it must be remembered that the testability distributions of the *inputs* of Boolean functions is only an estimate of the testability distributions of logic circuits.

distributions which could be used for functions with too many inputs for accurate Monte Carlo simulations.

The fault detection probability distribution was found by considering the number of times the output of a function differs from its expected value when there is a fault at an input node. The distribution is shown in Appendix C to be a purely combinatorial problem, and has a binomial distribution.

The approximation to the testability distribution is more complicated, since the fault detection probabilities of the different inputs to the function are interrelated. The testability distribution of an i-input function is shown in Appendix C to be basically the first order statistic of i fault detection probabilities, with the dependencies between inputs having to be taken into account. (A brief summary of order statistics is given in Appendix D.) The four approximations used to estimate the testability distribution are shown in Figure 3.6.

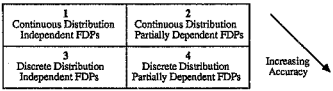


Figure 3.6 : Four asymptotic approximations to the testability distribution of Boolean functions

Although all four approximations to the testability distribution were shown to be asymptotically correct in Appendix C by a chi-square comparison with the Monte Carlo simulation, for small functions the fit is worse for the approximations that assume that the fault detection probabilities are continuous and independent.

None of the approximations are exact because only pairwise correlations between different inputs are considered in approximations 2 and 4 in Figure 3.6. Correlations between groups of 2, 3, 4, ... inputs need to be taken into account to improve the fit for small functions. This slight increase in accuracy at much higher complexity did not seem necessary, since it must be remembered that the testability distributions of the inputs of Boolean functions is only an estimate of the testability distributions of logic circuits.

Graphs of the fault detection probability and testability distributions for 2 to 12 input functions are shown in Figures 3.7 and 3.8 respectively. The vertical axes are scaled so that the "area" under each of the graphs is the same.

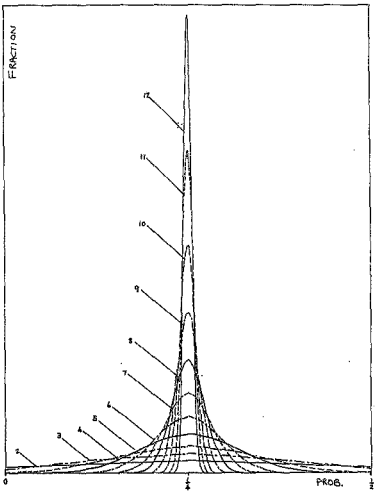


Figure 3.7 : Fault Detection Probability Distribution of Boolean functions

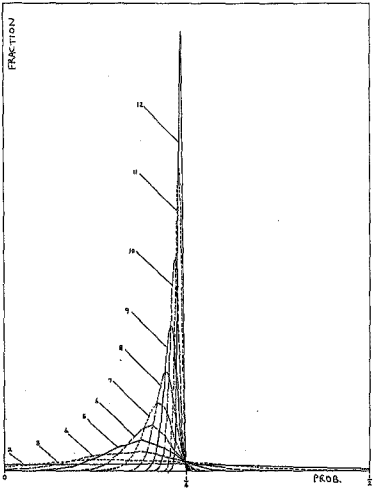


Figure 3.8 : Testability Distribution of Boolean functions

The testability distribution in Figure 3.8 summarizes this section succinctly. As the number of inputs to a function increases, a larger fraction of the functions have testabilities grouped in the vicinity of 1/4. Since functions with testabilities of about 1/4 require very few random tests to achieve testing with high confidence intervals, it can be concluded that for arbitrarily chosen functions, the probability of a function being testable increases with the size of the function. This point is elaborated in the conclusion to this chapter in Section 3.7.

Although on average faults at the inputs of large functions are testable with short random pattern test sets, there are still many functions with testabilities close to zero. These functions would require very long equiprobable 0/1 random tests for reasonable fault covers. The effect of optimising input probabilities for so-called random resistant faults is investigated in the next section by analysing a worst-case circuit.

3.6 Example : Analysis of an i-input AND G[illegible]imised Probabilities

A detailed testability analysis for an i-input AND gate with variable input probabilities is done in this section. This example pre-empts the discussion in Section 4.7 on optimising probabilities for arbitrary combinational circuits. It is assumed that the probabilities of the inputs to the AND gate are regulable and independent. The testability measure defined by equation (3.3) is used below, and the analysis is repeated in Appendix B for probabilistic syndrome testing. (An example similar to the first part of this section can be found in (Breuer and Friedman, 1976) for NAND gate networks.)

An AND gate was chosen for the example because large AND gates are worst case circuits with regard to random pattern testing, and are notoriously difficult to test probabilistically due to their high fan-in.

The analysis also applies to NAND, OR and NOR gates by replacing the probabilities of inputs being true by false where necessary.

The test circuit shown in Figure 3.9 consists of the AND gate being tested and a gold unit, with the outputs XORed together to give the test output.

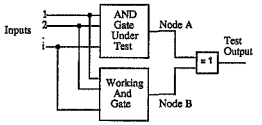


Figure 3.9 : Circuit for measuring the testability of an i-input AND gate

Due to symmetry, all the inputs to the AND gate have the same probability of being true, say p. For an input stuck-at 0 in the AND gate under test, the signal probability of node A in Figure 3.9 is zero. The signal probability at node B is p^i. Under these conditions the XOR gate acts as a buffer, so the signal probability of the test output (i.e. the fault detection probability) is also p^i. The number of test vectors N_0 necessary to detect an input stuck-at 0 is found from equation (3.3).

$c = 1 - (1 - p^i)^{N_0}$

or

$$N_0 = \frac{\ln(1 - c)}{\ln(1 - p^i)} \qquad \text{... (3.7)}$$

where c is the confidence interval.

For a single input stuck-at 1 in the AND gate under test, the signal probability at node A in Figure 3.9 is p^{i-1}. The signal probability at node B is as before, and the probability of the test output can be shown to be $(1-p)p^{i-1}$.

(Intuitively, the condition necessary to detect an input stuck-at 1 for an AND gate is to have that input false and all other inputs true. The probability of an input being false is (1-p) and the probability of i-1 inputs being true is p^{i-1}.)

The number of test vectors N_1 necessary to detect an input stuck-at 1 is

$$N_1 = \frac{\ln(1 - c)}{\ln(1 - (1-p)p^{i-1})} \qquad \text{... (3.8)}$$

Equations (3.7) and (3.8) are plotted in Figures 3.10 and 3.11 for various sizes of AND gates and a confidence interval of 99%.

The graph for the SA0 fault decreases monotonically with increasing input probabilities. This is expected since in the limiting case of the inputs always true, only one vector is needed to detect an input node stuck-at 0. The graph for the SA1 fault has a minimum and increases to infinity for input probabilities tending to either 0 or 1.

To optimise the input probabilities for the shortest test possible to test the AND gate, the minimum of the greater of N_0 and N_1 must be found. Since N_0 has no minimum, N_1 is differentiated to find its minimum, and then checked to see if it is greater than N_0.

Differentiating equation (3.8) and setting the derivative to zero with the constraints $0 < c < 1$ and $0 < p < 1$ gives

$$p = 1 - 1/i \qquad \ldots (3.9)$$

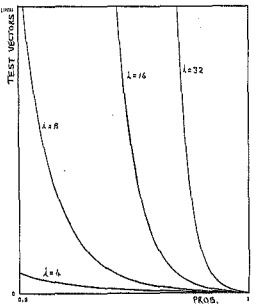


Figure 3.10 : Test vectors versus input probabilities for AND gate stuck-at 0 fault

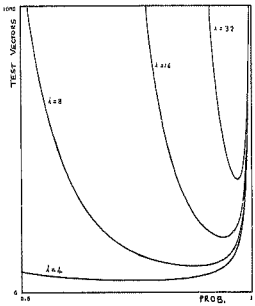


Figure 3.11 : Test vectors versus input probabilities for AND gate stuck-at 1 fault

Comparing equations (3.7) and (3.8), we have that $N_1 \geq N_0$ for $p \geq 1/2$, and from equation (3.9) $p \geq 1/2$ for $i \geq 2$. Therefore for all AND gates $N_1 \geq N_0$ and equation (3.8) corresponds to the minimum sought.

The number of test vectors needed to test an AND gate is plotted in Figure 3.12 for both the analysis above and probabilistic syndrome testing (from Appendix B) at 99% confidence, as well as exhaustive testing.

The number of test vectors in exhaustive testing grows exponentially with AND gate size, probabilistic syndrome testing grows quadratically, and the analysis above grows roughly linearly. This confirms the result that probabilistic syndrome testing is not a very efficient form of random pattern testing.

The asymptotic growth in the test lengths using optimised probabilities is shown to be linear in Appendix E by substituting the optimal value for the input probabilities into the

expression for N_1 and using various limiting relations. For large AND gates tested with 99% confidence, the test length is about 12,5 random vectors per input.

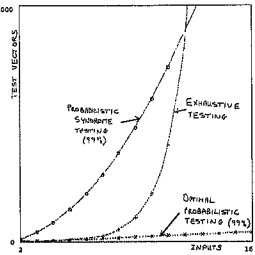


Figure 3.12 : Test vectors versus number of AND gate inputs for optimal probabilistic testing, probabilistic syndrome testing, and exhaustive testing

Note that equations (3.7) and (3.8) can be written in the form

$$(1 - X)^N = (1 - c) \qquad \ldots (3.11)$$

where X is a function of the gate size and input probabilities.

If c is the confidence interval, then (1-c) is a measure of uncertainty. From equation (3.11) it is seen that there is an exponential decrease in uncertainty with linear increase in cost (i.e. test vectors). Figure 3.13 shows this relationship between test vectors and uncertainty for an 8-input AND gate.

(An alternative way of looking at the minimization of test vectors done above, is to find the factor (1-X) in equation (3.11) such that the uncertainty decays as quickly as possible with N.)

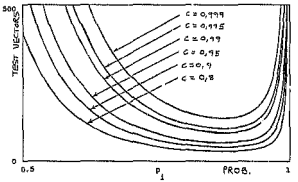


Figure 3.13 : Test vectors and uncertainty versus input probabilities for an 8-input AND gate

A final point worth noting in the AND gate example is the limiting probability of the output of a working AND gate being true when tested with optimised input probabilities.

we have

$$Pr\{output\} = (1 - 1/i)^i$$

using equation (3.9) for optimal testing, and

$$\lim_{i \to \infty} Pr\{output\} = 1/e \qquad \ldots (3.12)$$

Equation (3.12) has been derived rigorously, and differs from the result of 1/2 based on information theory (Agrawal, 1981). This shows that the assumption of maximum information throughput is not entirely justified. (For example Agrawal (1981) states that a 10-input AND gate is expected to be tested most efficiently with input probabilities of 0,933, whereas based on equation (3.9) the optimum input probabilities should be 0,9.)

3.7 Conclusion

In this chapter probabilistic testing has been shown to be feasible by considering various important concepts.

Firstly it was shown that in practice all testing is essentially probabilistic, since no fault model can take into account all possible failures which can occur in a circuit. The necessary theory for determining probabilistic fault covers and test lengths was derived, and applied in two cases at the end of the chapter.

In a section comparing deterministic and probabilistic fault cover for the single stuck-at fault model, it was shown that both definitions yield virtually identical probabilities of failing a circuit that does not work. It was also shown in Appendix A that the probability of failing a circuit that does not work is always at least as great as the fault cover.

An expression for the length of a random test set required to test a node with a specified confidence, given the fault detection probability, was derived in Section 3.4. The limitations of probabilistic syndrome testing were shown to be avoided by using a more efficient test setup. An exponential decrease in testing uncertainty for linear increase in test length was shown, which means that, if necessary, fault covers can be increased significantly for relatively small increases in test lengths.

Probabilistic testing was applied both to the whole spectrum of Boolean functions with equiprobable 0/1 probabilities, and to a specific circuit with optimised input probabilities.

In section 3.5 the testability distribution of Boolean functions was defined and estimated. The distributions for different size circuits plotted in Figure 3.8 were found to group near 1/4 for an increasing fraction of larger circuits. Using equation (3.3) relating the test length to the fault detection probability, functions with testabilities greater than 0,1 can be tested with 99,9% confidence in less than 100 vectors. This leads to the startling conclusion that faults at the inputs of virtually every logic circuit with more than 8 or 9 inputs, can be tested with more than 99,9% confidence with 100 equiprobable 0/1 random test vectors. Of course this is only a statistical observation assuming that all functions are equally distributed, and there are still many functions with very low testabilities, but it is nonetheless a noteworthy result.

Finally a complete testability analysis of an i-input AND gate was done as an example. The results obtained confirmed that if optimised input probabilities can be used, then even worst-case circuits such as AND gates can be efficiently tested probabilistically. For large gates the growth in test length was shown to be a linear function of gate size, thereby yielding test lengths of the same order as deterministic testing.

It must be noted that the AND gate example is a special case, since the fault detection probabilities and optimised input probabilities were easily obtained. In general the problem of calculating the fault detection probabilities is NP-hard and the subject of the next chapter.

Optimising the input probabilities for general combinational networks is also difficult, and not always practical since in some cases arbitrary signal probabilities might be difficult to generate. There are instances, however, when optimised input probabilities are useful, so the problem is addressed in the next chapter.

CHAPTER 4

CALCULATION OF FAULT DETECTION PROBABILITIES

4.1 Introduction

The benefits of random pattern testing were discussed in Chapter 2, and it was shown in Chapter 3 that random testing is feasible. The relationship between the fault detection probability of a fault and the random test length was derived, and test lengths were found to be much shorter than exhaustive. This chapter deals with the final element in the testability analysis: the calculation of accurate values for the fault detection probabilities.

Unfortunately there are no simple algorithms for calculating signal probabilities for Boolean expressions, since the problem is NP-hard (Section 2.5). This means that any algorithm to calculate signal probabilities will have limiting exponential complexity for some circuits, but it is exactly those circuits for which any test pattern generation algorithm (e.g D-algorithm) will have exponential complexity. However the growth in computation time for practical circuits has been observed to be roughly quadratic (Savir et.al., 1982), which seems to indicate that for most useful circuits test generation will not attain its limiting exponential complexity.

In accordance with the testing philosophy described in Chapter 2, only combinational circuits are considered here. It is assumed that sequential circuits have been designed using the structured design for testability techniques, and therefore appear to be combinational circuits when operating in test mode.

In principle finding fault detection probabilities is equivalent to calculating the signal probabilities for the test output of the combined circuit under test and the gold unit as shown in Figure 3.3 in the previous chapter.

A standard Boolean method of finding signal probabilities is presented in the first part of this chapter, followed by a discussion of Boolean Differences. An algorithm is proposed for calculating fault detection probabilities, and possible modes of operation of the algorithm are discussed. A prototype fault detection probability algorithm

implemented in Lisp is described, with program listings included in Appendix H. The problem of finding optimum input signal probabilities for testing arbitrary combinational circuits is addressed, and finally a complete example is presented.

4.2 Standard Boolean Method

This is a straightforward algorithm for finding the signal probability of a Boolean expression. The first step is to write the expression in sum-of-products form. The expression is then expanded as a canonical sum of fundamental products (i.e. disjunctive normal form, where each term corresponds to a 1 entry in the Karnaugh map). A property of expressions written in this form is that each term is mutually exclusive of the others, since only one of the terms can be true for a particular input combination. Therefore the signal probability of the expression is simply the arithmetic sum of the probabilities of each term in the expansion.

The disadvantage with the standard Boolean method is that although the algorithm is conceptually simple, the expansion easily reaches exponential complexity. The reason is that for a function of i variables, there are 2^i possible minterms in the canonical sum of fundamental products expression. There are many simple functions that have *exponential complexity when expanded (e.g. a parity generator), which make this* algorithm unusable in practice.

The signal probability of the output of the circuit in Figure 4.1 is found as an illustrative example.

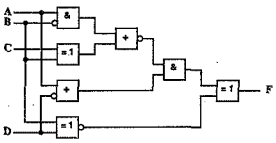


Figure 4.1 : Combinational circuit example

The Boolean expression for the output F of the circuit is

$$F = \{ [AB' \vee (B \oplus C)]' [A \vee D'] \} \oplus (B \oplus D)' \quad \ldots (4.1)$$

and writing F as a canonical sum of fundamental products we get

$$F = A'B'C'D \vee A'B'CD \vee A'BCD \vee A'BCD \vee AB'CD \vee ABC'D \vee ABCD \vee ABCD$$

Using the convention that uppercase letters denote Boolean variables, and lowercase letters denote probabilities, since all terms are mutually exclusive, if the inputs are assumed to be statistically independent, the probability of the output is

$$f = a'b'c'd' + a'b'cd' + a'bcd' + a'bcd + ab'cd' + abc'd' + abcd + abcd$$

The numerical value of f can be found by an algebraic substitution for the signal probabilities a,b,c and d.

4.3 Boolean Difference

Boolean Difference is a direct analytical method for finding tests for stuck-at faults in combinational circuits. It is considered here because the probabilistic algorithm derived below uses the same concepts. Boolean Difference is characterized as an algebraic or functional procedure in that circuit equations are utilized to generate tests, as opposed to a topological gate level description (e.g. D-algorithm, path sensitization).

Let F be a function of i variables including A, written F(A) for sh rt,

$$F(A, X_1, X_2, \ldots, X_{i-1}) \equiv F(A)$$

Consider testing for the fault A stuck-at 0 in F(A). As described in Chapter 3 the tests T_0 that detect the fault are

$$T_0 = F(A) \oplus F(0) \quad \ldots (4.2)$$

With some manipulation (Breuer and Friedman, 1986) equation (4.2) can be written in the following form

$$T_0 = A\,[\,F(0) \oplus F(1)\,] \qquad \text{... (4.3)}$$

and similarly to detect the fault A stuck-at 1, the tests are

$$T_1 = A'\,[\,F(0) \oplus F(1)\,] \qquad \text{... (4.4)}$$

The factor $F(0) \oplus F(1)$ is referred to as the Boolean Difference of F with respect to A, and is denoted by dF/dA:

$$dF/dA = F(0) \oplus F(1) \qquad \text{... (4.5)}$$

dF/dA represents all conditions for which F(A) and F(A') are different, i.e. the conditions for which F is sensitive to the value of A alone.

Identities have been developed to aid in the calculation of dF/dA (Breuer and Friedman, 1986), four of which are repeated in equations (4.6).

$$\left.\begin{aligned}
&\frac{d(FG)}{dA} = F\frac{dG}{dA} \oplus G\frac{dF}{dA} \oplus \frac{dF}{dA}\frac{dG}{dA} && \text{(a)}\\
&\frac{d(F \vee G)}{dA} = F'\frac{dG}{dA} \oplus G'\frac{dF}{dA} \oplus \frac{dF}{dA}\frac{dG}{dA} && \text{(b)}\\
&\frac{d(F \oplus G)}{dA} = \frac{dF}{dA} \oplus \frac{dG}{dA} && \text{(c)}\\
&\frac{dG}{dA} = \frac{dG}{dF}\frac{dF}{dA} && \text{(d)}
\end{aligned}\right\} \qquad \text{... (4.6)}$$

Equation (4.6 d) is a simple chain rule for Boolean Differences with reference to Figure 4.2. The combinational circuit in Figure 4.2 is composed of two sub-circuits F and G. It is assumed that variables $X_1, X_2, \ldots X_{i-1}$ and $Y_1, Y_2, \ldots Y_j$ are independent of each other.

The name primary node is used here to describe node F in Figure 4.2 due to its similarity with the circuit's primary inputs from a computational viewpoint. A primary node in a circuit is one which can be treated as an input node in subsequent symbolic manipulations of the circuit equations (i.e. the variable assigned to a primary node can be used as itself rather than as its value in terms of previous nodes in the circuit).

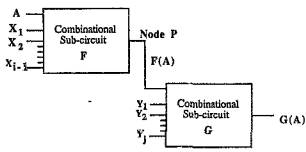


Figure 4.2 : Chain rule for Boolean Difference and Primary node P

The usefulness of primary nodes is evident since G(A) is reduced from a function of i+j variables to a function of j+1 variables. The reduction could be repeated for the second sub-circuit in Figure 4.2 if other primary nodes existed, thereby simplifying the calculation even further.

Primary nodes are necessary in efficient implementations of algorithms implementing both Boolean Differences and calculating fault detection probabilities.

4.4 Fault Detection Probability Algorithm

The algorithm developed for the calculation of signal probabilities is exact and therefore based on the same equations as Boolean Differences. The difference, however, is that instead of finding deterministic tests, probabilities of detecting faults are found. The calculation of signal probabilities consists of successive function decompositions until all the variables are processed, and is based on Shannon's well known expansion theorem below.

$$F(A) = A\,F(1) \vee A'\,F(0) \qquad \ldots (4.7)$$

Shannon's expansion is a nondisjunctive decomposition and is referred to as trivial function decomposition, since every Boolean function can be partitioned in this way. Equations (4.8) and (4.9) are special cases of equation (4.7) which were found useful in simplifying Boolean expressions.

$$A\,F(A) = A\,F(1) \qquad \ldots (4.8)$$

$$A \vee F(A) = A \vee F(0) \qquad \ldots (4.9)$$

The connection between Shannon's theorem and signal probabilities is found by noting that the two terms on the right hand side of equation (4.7) are mutually exclusive.

Therefore in probabilities

$$Pr\{ F(A) \} = Pr\{ A\, F(1) \} + Pr\{ A'\, F(0) \}$$

and since F(1) and F(0) are independent of the variable A, the final expression becomes

$$Pr\{ F(A) \} = a\, Pr\{ F(1) \} + a'\, Pr\{ F(0) \} \qquad \ldots (4.10)$$

where a'= 1-a is the probability of node A being false.

Equation (4.10) can be thought of as a "probabilistic expansion theorem", and defines a recursive means of finding the signal probability of a function F by successive decompositions.

The algorithm for finding the signal probability of a Boolean function is summarized below:

Step 1 : Use simplification theorems (4.8) and (4.9) wherever possible for obvious simplifications.

Step 2 : Depending on which version of the algorithm is being used (Section 4.4.1). either replace all variables represented only once by their signal probabilities, or leave them in the expression.

Step 3 : When no more obvious simplifications remain, choose a variable in the expression for expansion. The function is then decomposed with respect to this variable by applying the probabilistic expansion theorem.

Step 4 : Repeat from Step 1 until no more unprocessed variables remain.

An exponential upper bound is noted in the growth of the probability expression by using the probabilistic expansion theorem. In many cases the growth in the probability expression depends on the choice of variable in each successive decomposition.

$$A \vee F(A) = A \vee F(0) \qquad \ldots (4.9)$$

The connection between Shannon's theorem and signal probabilities is found by noting that the two terms on the right hand side of equation (4.7) are mutually exclusive.

Therefore in probabilities

$$Pr\{ F(A) \} = Pr\{ A\, F(1) \} + Pr\{ A'\, F(0) \}$$

and since F(1) and F(0) are independent of the variable A, the final expression becomes

$$Pr\{ F(A) \} = a\, Pr\{ F(1) \} + a'\, Pr\{ F(0) \} \qquad \ldots (4.10)$$

where a'= 1-a is the probability of node A being false.

Equation (4.10) can be thought of as a "probabilistic expansion theorem", and defines a recursive means of finding the signal probability of a function F by successive decompositions.

The algorithm for finding the signal probability of a Boolean function is summarized below:

Step 1 : Use simplification theorems (4.8) and (4.9) wherever possible for obvious simplifications.

Step 2 : Depending on which version of the algorithm is being used (Section 4.4.1), either replace all variables represented only once by their signal probabilities, or leave them in the expression.

Step 3 : When no more obvious simplifications remain, choose a variable in the expression for expansion. The function is then decomposed with respect to this variable by applying the probabilistic expansion theorem.

Step 4 : Repeat from Step 1 until no more unprocessed variables remain.

An exponential upper bound is noted in the growth of the probability expression by using the probabilistic expansion theorem. In many cases the growth in the probability expression depends on the choice of variable in each successive decomposition.

Different heuristics can be used to estimate the best choice of variable. An intuitive approach to making F(1) and F(0) as simple as possible is to choose the variable which appears the most times in the expression. However this simple heuristic was not found very useful in practice since the depth of the variables in the expression must be taken into account.

A more rigorous approach is to decompose the function with respect to another variable if the first variable chosen does not simplify F(1) and F(0) adequately. Another alternative is decomposing the function with respect to all variables in Step 3 of the algorithm and choosing the simplest result for further decomposition. This amounts to a breadth-first search of the tree of all decomposition paths. The heuristic which was finally implemented, was to use a weighted sum of the occurrences of each variable depending on the variable's depth in the expression. Top level variables were assigned a weight of 1, and for each lower level, the weight was halved.

4.4.1 Two versions of the algorithm

The algorithm for calculating signal probabilities can be used in two ways, depending on whether values are substituted for single variables in Step 2 or not. The algorithm is simpler if numerical values are substituted for variables because the outcome is just a numerical value for the signal probability of the function. In the more complicated version of the algorithm, values are not substituted for variables, and the outcome is an algebraic expression for the output signal probability in terms of the input signal probabilities. The two uses for the complicated version of the algorithm are described below.

When numerical values are substituted for variables, it is assumed that the variables are independent of each other and their signal probabilities are known. In hierarchical design it is common to construct modules which are then used as building blocks at higher levels. To avoid having to derive the probability expression each time the module is used, a symbolic probability expression can be found once using the complicated version of the algorithm, and associated with the module. The symbolic equations can then be incorporated into the rest of the circuit when the module is used.

The complicated version of the algorithm is also useful for calculating optimum values for the input signal probabilities, which requires an analytical expression for the function

in terms of its input signal probabilities. Optimising input signal probabilities is considered in Section 4.7.

4.4.2 Application of Algorithm to Logic Minimization

Although logic minimization was not the primary purpose of the signal probability algorithm, its ability to manipulate Boolean expressions is a useful tool for logic minimization and some verification. In fact, there are examples in the literature of graph based minimization algorithms using exclusively Shannon's expansion theorem.

Traditionally logic minimization algorithms have concentrated on minimal two-level logic implementations. The first step is to find the prime implicants of the function, and the second is to find a minimal or near minimal cover subject to certain cost criteria. These algorithms are either exact (based on Quine-McCluskey simplification), or use heuristics in determining the essential prime implicants.

Two level logic implementations, however, are not normally as efficient in terms of gate count as multilevel implementations. But synthesis of multilevel minimum cost circuits is an exceedingly difficult problem (Breuer, 1972, p59). An important area of research is to synthesize canonical multilevel logic circuits with identical components which are usually more complicated than single gates. These implementations are called iterative cell arrays or cellular arrays. The fault detection probability algorithm presented above can be used to factor functions into a canonical implementation using multiplexer building blocks. Each decomposition of the function adds a multiplexer. Once again the circuit in Figure 4.1 will serve as an example. Figure 4.3 shows one of the possible multiplexer decompositions for the circuit.

4.5 Modes of Operation of the Algorithm

The two envisaged modes of operation of the testability analysis algorithm are now discussed with examples.

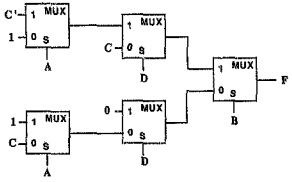


Figure 4.3 : Possible Canonical Implementation of Circuit in Figure 4.1

4.5.1 Interactive Signal Probability Mode

The first mode of operation of the testability analysis algorithm executes concurrently with the designer entering the circuit schematic. At this stage it is not possible to do a complete testability analysis since the circuit is incomplete. However, if the circuit is entered from its inputs to its outputs, the signal probabilities at existing nodes in the circuit can be calculated. The input probabilities to the circuit can be set to desired values or default to 1/2.

In this mode of operation the testability analysis algorithm is essentially calculating the controllabilities of nodes. Since the algorithm is exact, it can be used to aid circuit verification. Two examples of circuit verification are given:

1. Signal probabilities of 0 or 1 at nodes indicate a logical error (or a glitch generator) in the circuit, which can be reported to the designer immediately.

2. For some circuits the designer knows the expected probabilities, which can be compared to the actual signal probabilities of the designed circuit. For instance the signal probability of the leap year enable in a clock must equal the known frequency of leap years.

As an example, the *complicated version of the fault detection* probability algorithm is used below to find the signal probability of the output of the circuit in Figure 4.1. From equation (4.1) we have

$$F = \{ [AB' \vee (B \oplus C)]' [A \vee D'] \} \oplus (B \oplus D)'$$

Variable C is used only once, but as described in Section 4.4.1 it is not substituted by its numerical value in the complicated version of the algorithm . Variable B is chosen for the decomposition, and using the *probabilistic expansion theorem* we get

$$f = b\ Pr\{ [C(A \vee D')] \oplus D \} + b'\ Pr\{ [(A \vee C)'(A \vee D')] \oplus D' \}$$

Further decomposing both terms on the right hand side of the expression with respect to *D* gives

$$Pr\{ [C(A \vee D')] \oplus D \} = d\ Pr\{ (AC)' \} + d'\ Pr\{ C \}$$
$$= d(ac' + a') + cd'$$

and similarly

$$Pr\{ [(A \vee C)'(A \vee D')] \oplus D' \} = d\ Pr\{ A (A \vee C)' \} + d'\ Pr\{ A \vee C \}$$
$$= d'(a+a'c)$$

and substituting back into f we have

$$f = b(d(ac' + a') + cd') + b'd'(a+a'c)$$

The last equation is the final expression for the signal probability of the output of the circuit in terms of its input probabilities. Note that numerical values can simply be substituted for all variables, even though some variables occur more than once in the expression. This is possible since the different parts of the expression are now mutually exclusive as a result of applying *the probabilistic expansion theorem.*

4.5.2 Batch Fault Detection Probability Mode

The second mode of operation of the testability analysis algorithm is used once the circuit is completed, to determine the fault detection probabilities of single stuck-at faults in the circuit.

There is a well-known result in deterministic testing which reduces the number of nodes which have to be tested in a circuit. Only the fault set consisting of faults at the primary inputs and fanout branches need to be considered, since a test set that covers all members of this set is a complete set for all detectable faults in the circuit (Muehldorf and *Savkar, 1981). The above result* for fault collapsing is also true for random testing (Savir et.al., 1984).

The proposed algorithm calculates the fault detection probabilities for the required fault set. The method used is to find the signal probabilities of the set of tests that detect the fault as defined in Section 3.3 on Boolean Differences.

The circuit in Figure 4.1 will again serve as an example. The simple version of the algorithm is used below to find the fault detection probability for the fault A stuck-at 0. All input probabilities are assumed to be *1/2 and independent.*

Using equation (4.3), the tests which detect the fault A stuck at 0 are T = A [F(0) ⊕ F(1)], and the probability t is required. *Substituting for the function F from equation* (4.1) gives

T = A [{ (E ⊕C)' ⊕ (B⊕D)'} ⊕ {(B' v B⊕C)' v D'] ⊕ (B⊕D)'}]

Decomposing with respect to B using the probabilistic expansion theorem, and writing in terms of signal probabilities gives

t/a = b Pr{ [C⊕D] ⊕ [(C v D')⊕D] } + b' Pr{ [C'⊕D'] ⊕ [D'⊕D'] }

Substituting numerical values for a and b, and simplifying, yields

4t = Pr{ C ⊕ (C v D') } + Pr{ C' ⊕ D' }

Repeating the above process by expanding with respect to variable C,

$$4t = c\ Pr\{\ 0\ \} + c'\ Pr\{\ D'\ \} + c\ Pr\{\ D'\ \} + c'\ Pr\{\ D\ \}$$

and substituting for variables c and d gives the final result for the fault detection probability of the fault A stuck-at 0,

$$t = 3/16$$

which corresponds to a test set of 22 equiprobable 0/1 random vectors to detect the fault with 99% confidence.

4.6 Description of Prototype Implemented

The implementation of a testability analysis system can be considered at different conceptual levels. From a systems point of view, the main requirement is compatibility with other computer aided design systems in order for the results of the testability analysis to be useful. Especially in the signal probability mode of the algorithm (Section 4.5.1), an interactive user interface with the schematic capture system is needed. The database of the schematic capture package must be accessed and modified so that problems can be reported to the designer immediately. Writing a complete testability analysis system requires a large software investment, and is beyond the scope of this work.

The approach to the problem adopted was to write a simple interface between a standard circuit netlist format and the testability analysis program. In this way existing software could be used to move circuit schematic data to the testability analysis program. Figure 4.4 shows a block diagram of the data path to the testability analysis program. This certainly does not meet the requirements for an interactive system, and therefore should be seen as a simulation rather than an implementation of the complete testability analysis system. The connection between the schematic capture, logic simulator and testability analysis program was the HILO-3 netlist format.

The second level at which the implementation of the testability analysis program can be considered is the development of algorithms to perform the testability analysis itself. This is the level mainly implemented here.

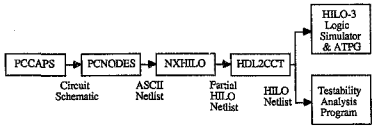


Figure 4.4 : Netlist conversions from Schematic Capture to Testability Analysis

HILO-3 is a trademark of Genrad, Inc.

PCCAPS, PCNODES and NXHILO are trademarks of Personal CAD Systems, Inc.

The main consideration when automating the signal probability algorithm described in this chapter is the large amount of symbolic manipulation that is needed. Available programming languages are compared in Appendix G, and Lisp was chosen to implement an experimental prototype of the proposed algorithm. The main advantages of Lisp over other languages in this application are that Lisp was especially designed for symbolic computation, has flexible list data structures, and an interactive programming environment.

The testability analysis programs are shown in Figure 4.5. HILO netlists are converted into dynamic data structures by the program NET.LSP, and the circuit description is maintained and modified by the program MAIN.LSP. Of significance are the three programs OP.LSP, SHANNON.LSP, and CALC.LSP which perform the required symbolic computations and calculate signal probabilities. The latter three programs are described briefly below, and program listings are included in Appendix H. A substantial portion of the work in this project was devoted to deriving the final data structures and programs mentioned.

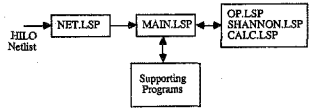


Figure 4.5 : Testability Analysis Programs

4.6.1 *BNF for Boolean Expressions in Testability Analysis*

Since Lisp has powerful functions for accessing the *first* and *rest* elements of dynamic lists, it was advantageous to express intermediate Boolean functions in a prefix notation where the first element of the list is the operator, and the other elements are the operands. More formally, the syntax accepted by the testability analysis program is given in Backus-Naur form below:

```
<expr> ::= <atom>                          ;Variable
           (~ <expr>)                      ;NOT
           (& <expr1> <expr2> ... )        ;AND
           (+ <expr1> <expr2> ... )        ;OR
           (⊕ <expr1> <expr2> ... )        ;XOR
           (~& <expr1> <expr2> ... )       ;NAND
           (~+ <expr1> <expr2> ... )       ;NOR
           (~⊕ <expr1> <expr2> ... )       ;XNOR
```

One of the factors affecting the efficiency of the program is the number of alternatives that have to be tested when a function is called. The above syntax has many operators and a simpler syntax is needed. The syntax finally implemented has the following BNF description:

```
<expr> ::= ( {= / ~} {<atom> / <and-expr>} )
<and-expr> ::= ( <expr>* )
```

In the internal syntax, lists of expressions are implied ANDs, and at the top level, an expression is either an AND or a NAND. The programming of the algorithm was simplified by introducing the syntatically symmetric operators = and ~, denoting the identity and the complement operations respectively.

The syntax above was chosen because it is compatible with both the Boolean and algebraic probability expressions encountered in the algorithm. The program OP.LSP performs easy simplifications, SHANNON.LSP contains heuristics to choose a variable for Shannon's expansion theorem, and CALC.LSP calculates numerical values for the signal probabilities of nodes in the circuit. The heuristic used in SHANNON.LSP is to choose the variable with the highest overall weight, with weights of 1 assigned to variables at the top level, and the weight halving at every lower level in the expression.

The reason that OP.LSP and SHANNON.LSP are two separate programs, is that the cause of the limiting exponential complexity of the algorithm - NP-completeness, has been separated from the "easy" simplifications. Therefore it is *beneficial to do as much* simplification as possible before using Shannon's theorem in order to reduce its use as much as possible. The programs OP.LSP, SHANNON.LSP and CALC.LSP are now described in more detail.

4.6.2 Program OP.LSP

OP.LSP is the heart of the testability analysis algorithm, since the efficiency of the algorithm depends on the number of times the expensive SHANNON.LSP is used. (Blindly using Shannon's expansion theorem often results in no savings over the canonical Boolean method of Section 4.2.)

OP.LSP converts expressions into the internal format, and does some simplification. *The main function in the Lisp program* is the simplification of implied AND lists, the result of which is then modified for either AND or NAND expressions. AND lists are simplified by recursively calling OP.LSP to simplify each expression in the AND list at a time. *Processed expressions simplify either to a numeral, an atom,* or a complex expression. Therefore a numeral, a list of atoms, a processed list of expressions, and an unprocessed list of expressions exist during the operation of the algorithm. A step of the simplification consists of *simplifying the first expression in the unprocessed* list. The next step depends on the simplified expression.

1. If the simplified expression is a numeral, *it is multiplied with the numeral in the* processed list. If the numeral is zero, then the result of the whole AND list is zero.

2. If the simplified expression is an atom, *it is added to the atom list. Furthermore* equation (4.8) is applied to all complex expressions in both the processed and unprocessed lists, to remove duplicates of the atom. All complex expressions are put in the unprocessed list since they are not necessarily simplified any more.

3. If the simplified expression is a complex list, it is simply added to the processed list.

The AND list is simplified once the unprocessed list is empty. The simplification of an AND expression is shown graphically in Figure 4.6.

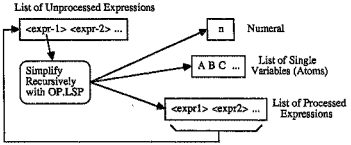


Figure 4.6 : Simplifying an AND expression with OP.LSP

4.6.3 Program SHANNON.LSP

The program SHANNON.LSP is divided into two parts. The first part accepts an expression and a variable name, and applies Shannon's expansion theorem to the expression with respect to the variable. The second part of the program heuristically finds the best variable to use for the decomposition, since the complexity of the resulting expression can depend strongly on the choice of variable.

One of the functions in SHANNON.LSP traverses the expression and compiles data on the number and depth of occurrence of each variable, which is used by another function to heuristically find the best variable. The different heuristics tried are described in Section 4.4.

4.6.4 Program CALC.LSP

The program CALC.LSP calculates numerical values for the nodal signal probabilities by substituting values for variables occurring only once in the expression (i.e. the simple version of the algorithm in Section 4.4.1.) The method used for calculating the signal probabilities is a direct implementation of the algorithm described in Section 4.4 above, and consists mainly of repeated calls to OP.LSP and SHANNON.LSP.

4.7 Calculating optimum input signal probabilities

An aspect related to the calculation of fault detection probabilities which still needs to be considered, is the optimisation of input signal probabilities for minimizing test lengths for arbitrary combinational circuits. This section is intended as an analysis of the problem and a comparison of several techniques implemented for its solution.

The analysis of the AND gate example in Section 3.6 confirmed the expected result that there are circuits which are poorly tested with equiprobable 0/1 input signal probabilities. The same example also showed that dramatic reductions in random test lengths can be achieved by testing with optimised signal probabilities.

In some classes of BIST circuits optimised signal probabilities are difficult to generate, and these circuits would have to be designed to be equally weighted random pattern testable. However, if the test vectors are generated externally or nonlinear feedback shift registers are used (Wunderlich, 1985), arbitrary signal probabilities at the test circuit's inputs can be achieved. The calculation of optimum signal probabilities is discussed below, possible algorithms are surveyed, and finally three promising methods are implemented and tested.

4.7.1 Nonlinear Programming Problem

A general statement of the problem of optimising input probabilities is presented below. Consider testing for n faults in a circuit with i inputs (Note that in VLSI circuits n may be very large). The fault detection probability of the kth fault is calculated by the fault detection probability algorithm described in Section 4.4, and is a function of the input signal probabilities given by $f_k(x_1,x_2,...,x_i)$. For input vectors with signal probabilities $x=[x_1\ x_2\ ...\ x_i]^T$, the length of the random test set is determined by the lowest fault detection probability of all n faults.

Therefore in order to achieve the minimum test length, the input probabilities x must be chosen such that the lowest of $f_1, f_2, ..., f_n$ is maximized, i.e.

Need $\max_x \{ F \}$

where $F = \min_k \{ f_k(x) \}$ $k = 1,2, ... ,n$ } .. (4.11)

Equation (4.11) is a nonlinear programming problem, where the objective function to be maximized F, is the lowest of a number of nonlinear functions. Figure 4.7 illustrates the maximization problem for 5 fault detection probabilities in 1 dimension.

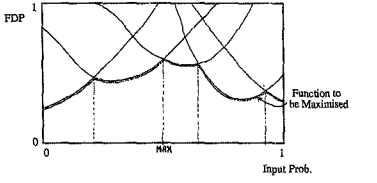


Figure 4.7 : Maximization of the minimum of 5 fault detection probabilities showing local maxima

The objective function could also be defined as the number of test vectors (inversely related to the fault detection probability) which needs to be minimized, but then confidence intervals have to be introduced. For the rest of this section the convention used in the literature on finding the minimum of a function will be adopted, but simply changing the sign of the objective function will convert it into a maximization problem.

A complication is that the optimisation of signal probabilities is a constrained optimisation problem, since the elements of the input vector represent probabilities and can only take on a limited range of values.

Algorithms for both constrained and unconstrained nonlinear optimization are discussed in Appendix F. The use of penalty functions reduces the problem to an unconstrained optimisation. Appendix F also contains a classification of representative unconstrained nonlinear programming algorithms, and a discussion of the two most promising - Powell's method without derivatives and the Hooke and Jeeves method. The most important consideration in the choice of algorithm is that derivatives cannot be used, since the objective function is not differentiable. Both methods were implemented in Pascal from non-commercial FORTRAN and BASIC programs in the literature (see

Appendix F). The performance of these methods is compared for test cases below, after discussing an alternative method of solving the optimisation problem.

4.7.2 Simulated Annealing

One of the difficulties in optimising a function with many local minima is that most nonlinear programming algorithms (including those discussed in Appendix F) tend to be "greedy" and only move "downhill" in directions that minimize the objective function. The effect is that the methods tend to work efficiently when they work at all - but they can easily be trapped at points which are not optimal or only locally optimal.

Two ways of trying to alleviate this problem are to do multiple optimisations from different starting points, or to use random methods such as arbitrarily evaluating the function at different points.

Simulated annealing is a probabilistic technique that has been successfully applied to multivariate optimisation (Kirkpatrick et.al., 1983), including integrated circuit design problems such as partitioning, placement and routing. The principle of simulated *annealing is based on the connection* between statistical mechanics and combinatorial optimisation (see Kirkpatrick et.al. (1983) for a discussion), and draws an analogy between the search for a minimum in an objective function and changes of state in materials.

To solidify a liquid into a regular crystal, the material must be cooled slowly so that it is in equilibrium at all temperatures. If the material is cooled too quickly, it does not have time to reach equilibrium and defects become frozen into the crystal. Greedy algorithms that only move to lower fupction values are like rapidly quenching a physical system to zero temperature, resulting in metastable states analogous to local minima of functions (Greene and Supowit, 1986).

Therefore when using simulated annealing techniques in optimisation, moves which increase the objective function are accepted with a probability dependent on a temperature parameter. The Metropolis method is a commonly used procedure for deciding the acceptance of the "uphill" move (Greene and Supowit, 1986). If the present position is x with objective function value F(x) and the move is to y with objective function F(y), then the move is accepted with a probability given by equation (4.12).

$$\Pr\{\text{move } x \text{ to } y\} = \begin{cases} e^{-(F(y) - F(x))/kT} & \text{for } F(y) \geq F(x) \\ 1 & \text{for } F(y) < F(x) \end{cases} \quad \ldots (4.12)$$

The parameter T corresponds to temperature in physical annealing, and decreases from a large value exponentially. In the simulated annealing algorithm implemented, the direction of the move was chosen randomly, and the distance moved in n-dimensional space was set by the steplength. If at a particular temperature, a predefined number of moves fail, the steplength was reduced. The function value after several successive reductions in steplength is assumed to be the minimum.

4.7.3 Test Function and Results

As mentioned in Appendix F test functions exist (e.g. Himmelblau, 1972, p195) for the evaluation of unconstrained nonlinear programming algorithms, but they are not generally representative of the problems encountered in optimising signal probabilities. These test functions are difficult to optimise due to the nature of their gradients, but typically they are functions of few variables, and only have a few minima which need to be found very accurately.

The characteristics of the type of function that is to be optimised are that there can be many variables, and the fault detection probabilities are in an algebraic sum-of-products form with many crossterms. Evaluation of the optimisation algorithms for particular circuits is too specific and not desirable. Therefore a new test function is used here with the expected characteristics of typical problems, where the number of minima and their relative differences can be controlled by parameters. The test function is defined by equation (4.13) for 1 dimension.

$$T(x) = (1 - a)x^2 + a/2[1 - \cos(2\pi bx)] \quad \ldots (4.13)$$

with $0 \leq a \leq 1, \quad b \geq 0$

The behaviour of the test function is shown below for the 1 dimensional case and various values of the parameters a and b. The effect of increasing a is to decrease the difference between function values at successive minima. The number of minima in the interval [0,1] is directly controlled by the second parameter b.

In the n dimensional case, the test function is the product of test functions in each dimension. The product was used instead of the sum so that crossterms would be present. A scaling factor must be added to the function in n dimensions, otherwise T = 0 for any $x_i = 0$.

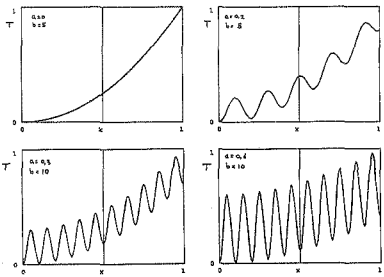


Figure 4.3 : Behaviour of test function for different values of a and b

Powell's method, Hooke and Jeeves' method and simulated annealing were tried on the test function varying the initial point, steplength, number of variables, and the parameters a and b. It must be remembered that the test function is itself only a single case, therefore the results must be interpreted qualitatively rather than quantitively.

The main advantages of the conventional nonlinear programming algorithms is their speed relative to random methods like simulated annealing (for most tests speedups of an order of magnitude were observed), but unless the algorithms are modified, simply searching for a minimum is not sufficient. For example even with a = 0,05 in the test function, the global minimum was rarely found. Even when searching for a single minimum, both Powell's method and Hooke and Jeeves' methods are not as efficient as they might be. The reason is that Powell's method assumes that the function will be approximated by a quadratic near its minimum, but this assumption does not hold

because the function is not differentiable. Hooke and Jeeves' method searches each variable separately, and therefore has difficulties with functions with many crossterms.

Simulated annealing on the other hand, converges at a rate determined by the temperature. Since the cooling schedule can be externally controlled, the tradeoff between the certainty of the solution and run times can be *adjusted dynamically*. The disadvantages of simulated annealing are that global solutions are not guaranteed, and run times and function evaluations are high. A further problem with simulated annealing is that there are many parameters which have to be selected, and there is no general way of determining the values of the parameters in advance. A positive feature of simulated annealing is that for the tests performed, *the number of function evaluations* is not a strong function of the dimension of the problem, which could be advantageous for large nonlinear optimisations.

4.7.4 *Observations and Further Suggestions for* Optimising Probabilities

Nonlinear optimisation is a difficult problem especially in cases where there are many local minima and a point close to the global minimum is required. Some observations and suggestions for further work are noted below.

Most algorithms are very sensitive to the steplength and choice of starting point for multimodal functions. Methods to alleviate this problem such as repeated runs with different initial conditions or random methods such as *simulated annealing increase the* execution time significantly, and while better results are often produced, it is still difficult to determine how close to the global optimum point they have reached.

All the techniques implemented to find optimal input probabilities indicate that it would be *very useful to have a* good starting point for the optimisation. Approximate testability analysis algorithms which run fast by neglecting reconvergent fanout nodes, might be useful in estimating the starting point for the optimisation.

An extension to the above discussion on optimisation is to test the circuit with two sets of random patterns with different signal probabilities. There are circuits for which the sum of the lengths of the two tests is less than the optimum length for one set. An example is testing a circuit with both a large AND and OR gate. It is known that the input probabilities should be close to 1 to test the AND gate, and *close to 0 to test the* OR gate.

By developing the above concept further to more random tests, one eventually gets to many tests of unity length, which corresponds to deterministic testing. This means that in principle there is a spectrum of testing possibilities from a single random test set to deterministic testing.

In the optimisation discussed above, the signal probability at each input node in the circuit was allowed to have an arbitrary value. A simplified problem of practical importance results if all input probabilities are restricted to the same value. For example if a single nonlinear feedback shift register is used, the signal probability can be changed from 1/2, but all flip-flops will have the same probability. The AND gate example in Section 3.6 is a special case of this, where all optimal input probabilities are equal.

Below are suggestions for further work which might be beneficial in the optimisation of signal probabilities but lie outside the scope of this work.

1. Specifically tailored parameters in simulated annealing (possibly different decay curves for the cooling schedule) might reduce run times significantly.

2. Use might be made of the knowledge that the objective function is piecewise differentiable, but it is probably difficult since in many cases the optimum point will be at the intersection of two fault detection probability functions so the derivative will not vanish.

3. Geometric programming is a special case of nonlinear programming, in which the objective function and constraint equations are all polynomials in non-negative variables with positive coefficients (posynomials). Special optimisations exist for these problems (Masanao, 1971). It is possible to convert each fault detection probability expression into a posynomial by replacing each x_k' with y_k, and introducing the extra constraints $x_k + y_k = 1$. Problems with this approach are that the number of variables is increased, extra constraints are added, and care must still be taken to find the minimum of a number of posynomials.

4.8 Conclusion

Whereas the previous chapters have considered probabilistic testing from general and theoretical viewpoints, specific practical aspects of probabilistic testing have been

discussed in this chapter. The two main considerations were the development and implementation of an algorithm for calculating fault detection probabilities, and the problem of optimising input probabilities to reduce test lengths.

Firstly a standard Boolean method of calculating fault detection probabilities was mentioned, and shown to be impractical since the limiting exponential can easily be reached in 2-level expressions for Boolean functions. Boolean Difference was discussed because it is an exact analytical method for determining test sets. The algorithm developed for calculating fault detection probabilities is based in Boolean Differences because it is argued in this work that in order to replace test pattern generation and help design verification, an exact testability measure is required.

A prototype fault detection algorithm was implemented in Lisp and various operating modes were discussed. The algorithm can be used as an aid to design verification interactively with the designer entering the schematic, or to calculate random pattern test lengths on completed circuits.

The use of optimal values for input signal probabilities was shown in Chapter 3 to significantly decrease test lengths in certain cases. The optimisation of input signal probabilities was shown to be a constrained nonlinear programming problem in general. Both classical algorithms and the more modern random methods (simulated annealing) were attempted to solve the problem. The optimisation of a test function is discussed in Section 4.7.3 and it is noted that it is very difficult to find the optimal solution in the presence of many local minima. Suggestions for simplifying the problem are mentioned in Section 4.7.4 since for large circuits a general solution to the problem does not appear to be practical.

CHAPTER 5

GENERAL CONCLUSIONS

The purpose of this work has been to evaluate a probabilistic definition of testability as a viable approach for dealing with the problems of testing VLSI digital integrated circuits. More specifically, it was found that although the investigation covered many interrelated ideas, the work can be grouped in three main parts, analysing probabilistic testing at different levels of abstraction. Since each part is largely self-contained, detailed conclusions are included at the end of Chapters 2, 3 and 4. To summarize, it was shown that probabilistic testing is desirable, feasible, and finally practical.

The outcome of Chapter 2 was a proposed testing methodology based on the requirements of testing, and the literature review. Structured design techniques were adopted as they have become widely used for dealing with the sequential circuit testing problem, in many cases together with built-in self-test. Therefore the main thrust of Chapter 3 was aimed at showing that probabilistic testing of combinational circuits is theoretically feasible, and the equations relating the test lengths to the fault detection probabilities were derived for two types of random pattern testing. The first type is the most efficient form of random pattern testing, while the second is related to syndrome testing. The discussion in Chapter 3 was augmented with two examples. The first analyses the whole spectrum of Boolean functions, whereas the second considers a *specific worst-case example*, and shows that test lengths are reasonable if optimised input probabilities are used.

The example on the *calculation of the testability distribution* of Boolean functions produced some interesting results. The first was that the input fault detection probabilities of a function are either all "even" or all "odd", forming a binary classification of circuits. The second result was *that the testability of virtually all faults* at the inputs of arbitrarily chosen circuits actually increases with the size of the circuit. Although this example was mainly of theoretical interest since faults at internal fanout nodes were not considered, it does have application in cases such as board testing, where pin connections often contribute to a large percentage of malfunctions.

In essence the central idea in Chapter 4 to showing that probabilistic testing is practical, is that instead of only using the testability measure to estimate the cost of test pattern generation, the testability analysis is made accurate enough so that it can be used with random testing to completely eliminate test pattern generation. It might be said that *nothing is gained* since in order to be exact, the testability measure attains the complexity of test pattern generation. However replacing test pattern generation by an accurate *testability analysis has* many beneficial sideffects. In terms of testing, the most important benefit is that although test generation costs are not necessarily reduced, test application *costs can decrease dramatically* due to the simplicity of random pattern testing. This is particularly attractive in high volume production, and when used with built-in self-test for *on-line testing.*

The benefits are not limited to testing, however. *The fact that the* testability analysis algorithm manipulates the Boolean equations of the circuit can be used to help with other aspects of the design cycle such as design verification and *detecting logical redundancy.* This work has only considered the logical level, but the testability analysis can also be used in physical verification, for example races and hazards. The procedure for detecting possible static and dynamic hazards involves operating on the Boolean equations without allowing certain simplifications, which could be performed by the testability analysis algorithm with minor modifications.

Chapter 4 also included the implementation of a prototype fault detection probability algorithm written in Lisp, and a discussion on optimising input signal probabilities. Several operating modes of the fault detection probability algorithm were proposed with examples done manually to describe the algorithms. Expressions containing a few thousand gates were analysed with the Lisp program, and it was clear that although a significant amount of time was spent on the algorithms, more work is needed to approach the speed of commercial programs like HILO-3. One of the problems is that although Lisp was found useful for prototyping, it did not run as efficiently as conventional procedural languages.

However, even a more efficient testability analysis would have the same disadvantage as automatic test pattern generation algorithms, viz. complexity. Since the problem is that *the algorithms* are NP-complete, it cannot be solved in general. It is felt that as circuits get larger, structured techniques will have to be adopted to simplify combinational circuit *testing. It is also felt that the most* promising method is to extend hierarchical design methodologies to hierarchical test application. The idea behind hierarchical test application is *that very large circuits* are always designed hierarchically, so instead of

designing a 200 input circuit with 1000 latches with a single LFSR pseudorandom number generator and one signature analyzer, each module would have its own built-in self-test. A relatively small hardware overhead is required to manage the testing at the highest level in the chip. Therefore in the hierarchical test application scenario proposed, the designer's knowledge about the topology of the circuit is used, but at the circuit segmentation level and not for generating tests for individual faults.

In conclusion, through the progress of this work various aspects of the digital integrated circuit design cycle have been analyzed and solutions proposed in terms of random pattern testing. It cannot be claimed that the general testing problem has been solved, nor even that all the related issues have been discussed. What has been done, however, is that a generalized methodology for testing circuits probabilistically has been developed which is easily automated and is useful for design verification, as well as greatly reducing test application costs.

APPENDIX A

COMPARISON OF DETERMINISTIC AND *PROBABILISTIC* DEFINITIONS OF FAULT COVER

For deterministic testing, the fault cover is normally defined as the fraction of stuck-at faults in a circuit which are tested with a specific test set. However in random pattern testing, each node is tested with a certain confidence, and the fault cover is defined in Section 3.3 as the confidence interval with which all stuck-at faults are tested with a random test of a given length. Equations are derived in this appendix relating the fault cover to the probability of failing a circuit that does not work, for both deterministic and random pattern testing.

Briefly stated, the problem is to find the difference between testing 99% of the nodes in a circuit *deterministically, and testing* all the nodes with 99% confidence, for example. For manufacturing testing it is important to find the probability of failing a circuit that does not work, *given the fault cover* and the probability of individual nodes working. The probability of failing a circuit that does not work is found below for both the deterministic and probabilistic cases. *It is assumed* in the derivations that all nodes have the same probability of working.

A.1 *Deterministic Fault Cover*

Consider testing a circuit *containing n nodes with* possible stuck-at faults, with a probability w that each node works, i.e.

Pr{node works} = w

Let the fraction of the nodes tested with the deterministic test set be c. Therefore c is defined as the fault cover.

The required quantity is Pr{circuit fails/circuit faulty}, where Pr{A/B} denotes the conditional probability of A given B. Using the equation for total probability gives

$$\Pr\{\text{circuit fails}\} = \Pr\{\text{circuit fails/circuit works}\}\ \Pr\{\text{circuit works}\} + \Pr\{\text{circuit fails/circuit faulty}\}\ \Pr\{\text{circuit faulty}\}$$

But since the probability of failing a circuit that works is ideally zero, we get

$$\Pr\{\text{circuit fails/circuit faulty}\} = \frac{\Pr\{\text{circuit fails}\}}{\Pr\{\text{circuit faulty}\}} \quad \text{... (A.1)}$$

The probability of a non-redundant circuit with n nodes working is w^n, therefore

$$\Pr\{\text{circuit faulty}\} = 1 - \Pr\{\text{circuit works}\} = 1 - w^n \quad \text{... (A.2)}$$

Similarly if a fraction c of the nodes are tested, the probability of a circuit passing the test is equal to the probability that (cn) nodes of the circuit work, i.e.

$$\Pr\{\text{circuit passing}\} = \Pr\{(cn) \text{ nodes work}\} = w^{cn}$$

and

$$\Pr\{\text{circuit fails}\} \approx 1 - w^{cn} \quad \text{... (A.3)}$$

Substituting equations (A.2) and (A.3) in (A.1) gives the probability of failing a faulty circuit for deterministic testing.

$$\Pr\{\text{circuit fails/circuit faulty}\} = \frac{1 - w^{cn}}{1 - w^n} \quad \text{... (A.4)}$$

A.2 Probabilistic Fault Cover

Consider testing the same circuit as above containing n nodes with a probability w that each node works.

Let the minimum confidence interval with which each node is tested be c. Once again c is defined to be the fault cover, and

$$\Pr\{\text{node fails/node faulty}\} \geq c \quad \text{... (A.5)}$$

For worst case analysis, the equality in equation (A.5) will be used.

Using total probabilities to find Pr{node passes} gives

$$\Pr\{\text{node passes}\} = \Pr\{\text{node passes/node faulty}\}\ \Pr\{\text{node faulty}\} + \Pr\{\text{node passes/node works}\}\ \Pr\{\text{node works}\} \quad \ldots (A.6)$$

The probability of passing a node that works is unity, and from equation (A.5)

$$\Pr\{\text{node passes/node faulty}\} = 1 - c$$

Therefore equation (A.6) reduces to

$$\Pr\{\text{node passes}\} = (1 - c)(1 - w) + w = 1 - c(1 - w) \quad \ldots (A.7)$$

The probability of a circuit with n nodes passing, is equal to the probability of all nodes passing, therefore

$$\Pr\{\text{circuit passes}\} = [\Pr\{\text{node passes}\}]^n = [1 - c(1 - w)]^n$$

and

$$\Pr\{\text{circuit fails}\} = 1 - [1 - c(1 - w)]^n \quad \ldots (A.8)$$

The probability that a circuit does not work is as in the deterministic case, therefore substituting equation (A.8) into (A.1) gives the probability of failing a faulty circuit for *random pattern testing.*

$$\Pr\{\text{circuit fails/circuit faulty}\} = \frac{1 - [1 - c(1 - w)]^n}{1 - w^n} \quad \ldots (A.9)$$

Equations (A.4) and (A.9) are the required equations in Section 3.3, defining the probability of failing a circuit that does not work for the deterministic and probabilistic cases respectively.

The two equations have different numerators, but in order to make a comparison, typical values for c and w in practical circuits must be noted.

The fault cover c is close to unity for practical testing, and is usually at least 95%. The probability of a node working w is very close to unity for large circuits in order to have a reasonable circuit yield. For example a circuit containing 10000 nodes with a probability 0,9995 of each node working, has a yield w^n of less than 1%. The fact that w is very close to unity can be used together with the binomial approximation (A.10) to show that equations (A.4) and (A.5) are essentially equivalent.

$$(1 + x)^m \cong 1 + mx, \quad \text{for very small } x \qquad \ldots (A.10)$$

In the probabilistic expression in equation (A.9) above, the factor (1-w) is very small, therefore applying the binomial approximation in reverse gives

$$[1 - c(1 - w)]^n = [(1 - (1 - w)^c]^n = w^{cn} \qquad \ldots (A.11)$$

Comparing equations (A.4) and (A.9) using equation (A.11) shows that there is very little difference between the deterministic and probabilistic definitions of fault cover for practical circuits. Therefore by either testing 99% of nodes in a circuit deterministically, or by probabilistically testing all nodes with 99% confidence, gives the same probability of failing a circuit that does not work.

The probability of failing a faulty circuit is plotted in Figures A.1 and A.2 to show the general characteristics for typical conditions. In Figure A.1 the probability of a node working is kept constant at 0,999, and in Figure A.2 the fault cover is 0,99.

It is important to note from the figures that the probability of failing a faulty circuit is never less than the fault cover.

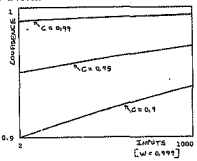


Figure A.1 : Probability of Failing a Faulty Circuit versus Number of Nodes and Fault Cover

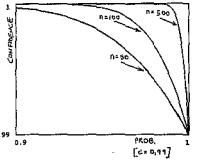


Figure A.2 : Probability of Failing a Faulty Circuit versus Probability of Nodes Working and Number of Nodes

APPENDIX B

TESTABILITY ANALYSIS USING "PROBABILISTIC SYNDROME" TESTING

In Section 3.4 it was noted that there are many possible random pattern testing strategies. The optimal testing strategy is shown in Figure 3.3, and involves a comparison of the output of the circuit under test with a known good response. The relation between test lengths and fault detection probabilities for this type of testing is derived in Section 3.4.1, and given by equation (3.3). The first part of this appendix contains a description of the test setup and the derivation of the test lengths for an alternative testing strategy, where only the long-term output probabilities of circuit are measured. The AND gate example done in Section 3.6 is repeated for this testing procedure at the end of the appendix.

B.1 Description of Test Setup

As mentioned in Section 3.4, the testing strategy described in this appendix is based on previous work (Franco, 1986), but it has been extended and included here both as a comparison with other techniques, and for its connection with syndrome testing.

The syndrome of a Boolean function is defined as the number of minterms realized by the function (Savir, 1980). The principle of syndrome testing is that functions can be implemented in such a way that fault-free and faulty circuits have different syndromes. The test procedure then simply consists of applying every input combination to the function and measuring its syndrome.

The concept of syndrome testing is analysed probabilistically *in this* appendix, hence the use of the term "probabilistic syndrome" testing. The advantage of probabilistic syndrome testing over its deterministic counterpart is that exhaustive tests are not necessarily required to detect faulty circuits with high confidence intervals. Testing is performed by applying random patterns to the circuit under test and observing the signal probability of the output (e.g. by counting ones). The test length is sufficiently long to

APPENDIX B

TESTABILITY ANALYSIS USING "PROBABILISTIC SYNDROME" TESTING

In Section 3.4 it was noted that there are many possible random pattern testing strategies. The optimal testing strategy is shown in Figure 3.3, and involves a comparison of the output of the circuit under test with a known good response. The relation between test lengths and fault detection probabilities for this type of testing is derived in Section 3.4.1, and given by equation (3.3). The first part of this appendix contains a description of the test setup and the derivation of the test lengths for an alternative testing strategy, where only the long-term output probabilities of circuit are measured. The AND gate example done in Section 3.6 is repeated for this testing procedure at the end of the appendix.

B.1 Description of Test Setup

As mentioned in Section 3.4, the testing strategy described in this appendix is based on previous work (Franco, 1986), but it has been extended and included here both as a comparison with other techniques, and for its connection with syndrome testing.

The syndrome of a Boolean function is defined as the number of minterms realized by the function (Savir, 1980). The principle of syndrome testing is that functions can be implemented in such a way that fault-free and faulty circuits have different syndromes. The test procedure then simply consists of applying every input combination to the function and measuring its syndrome.

The concept of syndrome testing is analysed probabilistically in this appendix, hence the use of the term "probabilistic syndrome" testing. The advantage of probabilistic syndrome testing over its deterministic counterpart is that exhaustive tests are not necessarily required to detect faulty circuits with high confidence intervals. Testing is performed by applying random patterns to the circuit under test and observing the signal probability of the output (e.g. by counting ones). The test length is sufficiently long to

determine with a predefined confidence whether the circuit's output probability is the expected fault-free probability (i.e. syndrome) or not.

B.2 Relation between Test Lengths and Syndromes

Consider using probabilistic syndrome testing to distinguish between two modes of circuit operation. The first mode A has an output signal probability a, and the second B has output signal probability b. It is assumed that $a \neq b$, otherwise the circuit is "syndrome untestable" (Savir, 1980). Modes A and B could be the normal and a faulty operation of the circuit, or two distinguishable fault conditions if fault location is required.

The problem is to choose the test length N and number of ones r, such that if after N tests more than r ones occurred, the circuit is in one mode of operation, otherwise it is in the other mode of operation. It must be noted that if r ones have occurred before N tests it is pointless to continue the test, therefore test lengths shorter than N are possible. In statistical terminology this situation is known as semi-curtailed sampling (Guenther, 1977). The expressions for N and r are now derived.

Let Y be the number of tests required until the rth one appears at the circuit's output. It can be shown that the variable Y has a negative binomial distribution (Guenther, 1977, p14) given by

$$Pr\{ Y=y \} = \binom{y-1}{r-1} p^r (1-p)^{y-r} \qquad (y=r,r+1, \ldots) \qquad \ldots (B.1)$$

with mean and variance r/p and $r(1-p)/p^2$ respectively.

We have p = a for mode A and p = b for mode B of circuit operation.

In order to identify the actual mode of circuit operation with a confidence interval c, we need to establish the mutually exclusive events {Y≤N} and {Y>N} such that

$$Pr\{ Y \leq N/p=b \} \cong Pr\{ Y>N/p=a \} \cong c \qquad \ldots (B.2)$$

assuming a < b, else reverse the inequalities.

For reasonably large r (Franco, 1986) equation (B.1) has an approximately normal distribution, and applying the approximation to equation (B.2) gives

$$\Phi\left(\frac{bN-r}{r(1-b)}\right) = 1 - \Phi\left(\frac{aN-r}{r(1-a)}\right) = c$$

where $\Phi(z)$ is the Standard Normal Distribution function

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-x^2/2}\, dx$$

By choosing Z such that $\Phi(Z) = c$ and simplifying (Franco, 1986), we get

$$r = \left(\frac{Z(a\sqrt{1-b} + b\sqrt{1-a})}{b-a}\right)^2 \qquad \text{... (B.3)}$$

and

$$N = \frac{r + Z\sqrt{r(1-b)}}{b} = \frac{r - Z\sqrt{r(1-a)}}{a} \qquad \text{... (B.4)}$$

Equations (B.3) and (B.4) correspond to equation (3.3) for probabilistic syndrome testing.

B.3 Example: Analysis of a i-input AND Gate with optimised input probabilities

The testability analysis of an i-input AND gate done in Section 3.6 is repeated here using probabilistic syndrome testing for comparison.

Consider testing an AND gate with i inputs. Due to symmetry, let all inputs to the AND gate have the same signal probability, say p.

Since the inputs are assumed to be independent, the probability of the output being true in a fault-free AND gate is

$$Pr\{Normal\} = p^i$$

For a single input stuck-at 0, the output probability is

$Pr\{SA\text{-}0\} = 0$, and similarly $Pr\{SA\text{-}1\} = p^{i-1}$

The length of the test required to detect an input stuck-at 0 is found from equations (B.3) and (B.4) by substituting

$a = Pr\{SA\text{-}0\} = 0$, and $b = Pr\{Normal\} = p^i$, giving

$$N_0 = Z^2 \left(\frac{1 + \sqrt{1-p^i}}{p^i} \right) \qquad \text{... (B.5)}$$

where Z is determined from the confidence interval as described above.

Similarly for a single input stuck-at 1 we have

$a = Pr\{Normal\} = p^i$, and $b = Pr\{SA\text{-}1\} = p^{i-1}$, giving

$$N_1 = \frac{Z^2}{(1-p)^2 p^{i-1}} \left(p\sqrt{1-p^{i-1}} + \sqrt{1-p^i} \right) \left(\sqrt{1-p^{i-1}} + \sqrt{1-p^i} \right) \qquad \text{... (B.6)}$$

The input probabilities need to be optimised in order to minimize the greater of N_0 and N_1 in equations (B.5) and (B.6) respectively. The graphs of N_0 and N_1 versus input probabilities follow the same general trends as Figures 3.10 and 3.11 in Section 3.6.

The equation for N_0 decreases monotonically with input probability, but N_1 has a minimum value. N_1 was differentiated numerically for AND gates with 2 to 70 inputs using a 99% confidence interval. A least squares nonlinear regression analysis was preformed on the data obtained, and showed that the minimum number of vectors necessary to test an i-input AND gate grows quadratically with gate size. The coefficients of the quadratic are approximately

$$N_{min} = 33{,}44i^2 - 33{,}44i + 11{,}4 \qquad \text{... (B.7)}$$

The corresponding asymptotic growth for the testability analysis in Section 3.4 is shown to be linear in Appendix E, with $N_{min} = 12{,}5i$ for 99% confidence, which shows that probabilistic syndrome testing is not the most efficient form of random pattern testing.

APPENDIX C

THEORETICAL TESTABILITY DISTRIBUTION OF BOOLEAN FUNCTIONS

The testability distribution of logic circuits was defined in Section 3.5. It was noted that estimates of the testability distribution could be found from the testability distribution of Boolean functions. The testability distribution of Boolean functions was approximated by a Monte Carlo simulation and several asymptotic theoretical distributions. The theoretical asymptotic fault detection probability and testability distributions are found in this appendix. The distributions are compared to the Monte Carlo simulation with a χ^2 goodness-of-fit test at the end of the appendix.

C.1 Theoretical Fault Detection Probability Distribution

The fault detection probability can be found from the number of times that the output is *different to its expected* value. Consider finding the fault detection probability of a fault at the first input of an i-input function with equiprobable 0/1 probabilities at the function's inputs. The truth table is shown in Table C.1 below.

For every pair of consecutive entries in Table C.1 only the first input changes, and the probability of F(0,...) and F(1,...) differing is 1/2. There are 2^{i-1} pairs of terms therefore the number of times the output differs from its expected value ranges from 0 to 2^{i-1}. Therefore using equiprobable input probabilities the fault detection probability ranges from 0 to $(2^{i-1}/2^i) = 1/2$ as expected. The distribution of the number of times the output differs from its expected value is a purely combinatorial problem. The probability that the output differs k times is just the probability of k favourable outcomes out of 2^{i-1} independent events. Therefore the distribution of the fault detection probability is the binomial distribution given in equation (C.1).

$$f_{fdp}(k) = 2^{-i}\binom{2^{i-1}}{k} \qquad \text{... (C.1)}$$

where $\binom{n}{r} = \frac{n!}{r!(n-r)!}$ is the binomial coefficient.

Table C.1 : Finding the Fault Detection Probabilities from the truth table

Inputs i i-1 3 2 1	Boolean function	Probability of two consecutive outputs differing
0 0 ... 0 0 0 0 0 ... 0 0 1	F(0,...) F(1,...)	1/2
0 0 ... 0 1 0 0 0 ... 0 1 1	F(0,...) F(1,...)	1/2
0 0 ... 1 0 0 0 0 ... 1 0 1	F(0,...) F(1,...)	1/2
. . . .	. .	.
1 1 ... 1 1 0 1 1 ... 1 1 1	F(0,...) F(1,...)	1/2

(For large values of $m = 2^{i-1}$ the binomial distribution tends to gaussian with mean $\mu = m/2$ and standard deviation $\sigma = \sqrt{m}/2$. This approximation was used in the calculations for $i \geq 12$.)

C.2 Theoretical Testability Distribution Approximation

The approximation to the testability distribution is more complicated than the fault detection probability (FDP) due to the relationship between the different inputs to the function.

The four asymptotic testability distributions derived are shown in Figure 3.6, and noted below:

1. The FDPs are independent and continuous.
2. The FDPs are partially dependent and continuous.
3. The FDPs are independent and discrete.
4. The FDPs are partially dependent and discrete.

The first approximation to the testability distribution is that the fault detection probabilities of the different inputs to the function are statistically independent. (Used in distributions 1 and 3 above.)

For an i-input Boolean function the testability distribution is the distribution of the lowest of i fault detection probabilities each with a pdf given by equation (C.1). Therefore if the fault detection probabilities are assumed independent, the testability distribution is given by the first order statistic of i fault detection probabilities. If the fault detection probability is assumed continuous (i.e. the binomial distribution is approximated by a gaussian even for small functions) then the first order statistic is given by equation (D.3), and the first order statistic is given by equation (D.4) for discrete distributions. (A brief summary of order statistics is given in Appendix D.)

Although the approximations 1 and 3 to the testability distribution are shown to be asymptotically correct below by a χ^2 comparison with the Monte Carlo simulation, the fit is bad for small functions. The error is due to the assumption of statistical independence.

In reality the fault detection probabilities of the inputs to a Boolean function are not independent and there are correlations between groups of 2, 3, 4, ... fault detection probabilities.

An improved approximation to the testability distribution (used in approximate distributions 2 and 4) is to take into account the major correlation between the different inputs (i.e. pairwise correlation), and neglect all higher correlations. Consider a portion of the truth table for finding the fault detection probabilities shown in Table C.2.

As in Table C.1, the fault detection probability of the first input is found by comparisons A and B differing in Table C.2. Similarly the fault detection probability of the second input is found by comparisons C and D differing.

By considering all possible values of the function F in Table C.2 it is seen that either both fault detection probabilities differ by an even number or both by an odd number. Since the complete truth table consists of a number of sections as in Table C.2, either all the inputs have an "even" or all the inputs have an "odd" fault detection probability. (A fault detection probability is "even" if the output differs from its expected value an even number of times.)

Table C.2 : Correlation between two fault detection probabilities

Inputs i 2 1	F	FDP-1	FDP-2
. .. 0 0	x	A	C
. .. 0 1	x		D
. .. 1 1	x	B	
. .. 1 0	x		

S = Same D = Different

A F	C D
S S	S S/S S
S D	D S/S D
D S	S D/D S
D D	D D/S S

Therefore approximate distributions 2 and 4 were calculated by splitting "even" and "odd" functions, and finding the testability distribution of each group as above (i.e. assuming statistical independence within the group) before combining distributions.

C.3 Comparison of Results with Monte Carlo Simulation

The four theoretical estimates of the testability distribution are compared to the results of the Monte Carlo simulations in Table C.3 by using a χ^2 goodness-of-fit test. The approximations are compared to the exact testability distribution for 2 to 4 input functions, and one million trials of the Monte Carlo simulation for the rest.

As a first estimate χ^2 values of the order of the degrees of freedom indicate a good fit between the two distributions. From Table C.3 it can be seen that all four approximations are asymptotically correct, with the fourth approximation most accurate for small functions. (Note that approximation 4 is exact for 2-input functions since there is only a pairwise correlation between inputs.)

Therefore it is concluded that for large functions the correlations between the different input fault detection probabilities becomes less important, and the theoretical approximations are accurate estimates of the testability distribution of faults at the inputs of Boolean functions.

Table C.3 : χ^2 comparison of testability distribution approximations
(form: χ^2/degrees of freedom)

Inputs	Approx 1	Approx 2	Approx 3	Approx 4
2	0,02/1	-	0,25/1	0,00/1
3	10,28/2	10,83/2	18,45/3	0,69/3
4	8373,96/6	3044,95/6	3908,70/6	521,83/6
5	37583,34/11	20547,99/11	20678,94/11	2120,67/11
6	7462,81/17	6055,45/17	4541,77/16	308,17/16
7	2074,08/23	2085,03/23	1384,96/23	172,83/23
8	726,27/32	787,92/33	490,26/32	45,03/32
9	217,93/44	256,43/44	168,11/43	65,38/43
10	329,71/60	335,31/60	318,87/60	235,54/60
11	227,14/83	221,39/83	233,01/83	230,48/83
12	116,78/113	119,01/113	115,67/113	114,06/113
13	134,20/158	134,99/158	133,54/158	132,23/158

APPENDIX D

DISTRIBUTIONS OF ORDER STATISTICS

A brief overview of order statistics is given below, including the formulae required in Appendix C for the calculation of the testability distribution of Boolean functions. (The formulae are taken from (David, 1970).

Let $X_1, X_2, X_3, \ldots X_n$ denote a random sample from a distribution with a probability density function (pdf) f(x) and cumulative distribution function (cdf) F(x).

Let Y_1 be the smallest X, Y_2 the next X, ... , and Y_n the largest X. Therefore, $Y_1 \leq Y_2 \leq \ldots \leq Y_n$ represent $X_1, X_2, X_3, \ldots X_n$ in order of magnitude.

Then Y_k, k = 1, 2, ... , n, is called the kth order statistic of the random sample $X_1, X_2, X_3, \ldots X_n$

The pdf of the kth order statistic $g_k(x)$ can be found from the cdf given by equation (D.1) which applies for both continuous and discrete cases.

$$G_k(x) = I_{F(x)}(k, n-k+1) \qquad \ldots (D.1)$$

where $I_P(a,b)$ is the incomplete beta function defined below.

$$I_P(a,b) = \frac{\int_0^P t^{a-1}\,(1-t)^{b-1}\,dt}{\int_0^1 t^{a-1}\,(1-t)^{b-1}\,dt}$$

In the continuous case the required pdf $g_k(x) = G'_k(x)$ is given by (D.2), which reduces to (D.3) for the first order statistic.

$$g_k(x) = \frac{n!}{(k-1)!(n-k)!}[1 - F(x)]^{n-1}\, f(x) \qquad \ldots (D.2)$$

and

$$g_1(x) = n\,[1 - F(x)]^{n-1}\,f(x) \qquad \text{... (D.3)}$$

In the discrete case the pdf is the difference between consecutive cdf values (D.4), and since $I_p(1,b)$ is easily integrable the pdf of the first order statistic simplifies to (D.5).

$$g_k(x) = I_{F(x)}(k,n-k+1) - I_{F(x-1)}(k,n-k+1) \qquad \text{... (D.4)}$$

and

$$g_1(x) = [1 - F(x-1)]^n - [1 - F(x)]^n \qquad \text{... (D.5)}$$

Equations (D.3) and (D.5) are the first order statistics for the continuous and discrete cases, used in the estimation of the testability distribution in Appendix C.

APPENDIX E

ASYMPTOTIC TEST LENGTHS FOR OPTIMAL AND-GATE TESTING

A testability analysis of an i-input input AND gate using optimised input probabilities was done in Section 3.6. The i-input *setup* was used, where a single error could be detected. It can be seen from Figure 3.12 that test lengths grow slower than for both exhaustive testing and probabilistic syndrome testing with optimised input probabilities. The *asymptotic growth* in test lengths is shown to be a linear function of AND gate size in this appendix.

The expression for the number of random test patterns needed to detect an input stuck-at 1 fault is given by equation (3.8), which can be written in the form below.

$$(1 - c) = \left(1 - (1-p)p^{i-1}\right)^N \qquad \text{... (E.1)}$$

where i is the number of inputs to the AND gate, p is the input probabilities, c is the confidence interval, and N is the test length.

It was shown in Section 3.6 that the optimum input signal probabilities are given by p = 1 - 1/i. The asymptotic growth in the test length can be found by substituting the optimal value for the input probabilities into equation (E.1), giving

$$(1 - c) = \left(1 - \frac{(1 - 1/i)^i}{i}\right)^N \qquad \text{... (E.2)}$$

As i gets large, the approximation for the exponential can be used, i.e.

$$(1 + x/m)^m \equiv e^x, \quad \text{for large m}$$

Substituting the approximation into equation (E.2) gives

$$N = \frac{\ln(1 - c)}{\ln[1 - 1/e(i-1)]} \qquad \text{... (E.3)}$$

Further applying the approximation

$\ln(1 + x) \equiv x$, for small x

gives

$N_1 = - e (i-1) \ln(1 - c)$, for large i ... (E.4)

Equation (E.4) shows that the number of vectors required to test an AND gate grows linearly with gate size for large gates, if optimised input probabilities are used.

APPENDIX F

NONLINEAR PROGRAMMING ALGORITHMS

The calculation of optimum signal probabilities is discussed in Section 4.7, where it is shown that the optimisation is a nonlinear programming problem. This appendix contains both a technique for transforming a constrained optimisation problem into an unconstrained optimisation problem, and a classification of unconstrained nonlinear programming algorithms. The two most promising algorithms for optimising signal probabilities are discussed separately.

A large number of algorithms have been proposed to solve the general nonlinear programming problem, since classical methods like Cauchy's steepest descent (1847) are unsatisfactory for arbitrary functions of many variables (Mordecai, 1976). Unlike the simplex method of linear programming, no method appears to be superior to all the others (Himmelblau, 1972).

Constrained optimisation is much harder than unconstrained optimisation (Himmelblau, 1972), but the constraints on input probabilities are a special case, since there are only explicit constraints in the form $0 \leq x_j \leq 1$ for j=1,2,...i, but no implicit constraints in the form $g(x) \geq 0$, where $x = [x_j]^T$.

A simple approach for dealing with explicit constraints is to limit the step size when the function is searched outside the permitted ranges. More generally penalty functions transform the problem into an unconstrained optimisation (Mordecai, 1976), and are one of the most successful approaches to solving constrained optimisations. A weighting function is added to the objective function outside the allowed range of the x vector to discourage the algorithm from finding a solution in an unacceptable region. Therefore unconstrained optimisation algorithms are an important area of research in their own right. The classification of some representative algorithms for unconstrained optimisation in Figure F.1 has been derived from (Masanao, 1971; Himmelblau, 1972; and Mordecai, 1976).

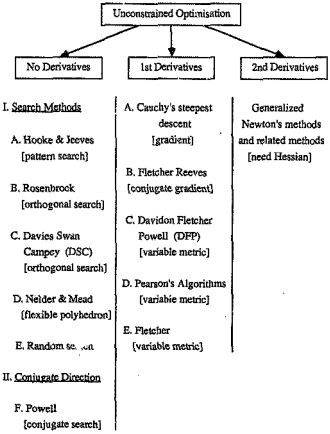


Figure F.1 : Classification of Unconstrained Nonlinear Programming Algorithms

Unconstrained nonlinear programming algorithms can be classified according to the amount of derivative information they use. When gradients or Hessian matrices of second partial derivatives are easily calculated, the methods without derivatives are generally the least efficient (Masanao, 1971). However since in optimising input probabilities the objective function consists of different functions for different input vectors, its derivative does not exist at all points and therefore a method that does not use derivatives must be used.

Himmelblau (1972) has a chapter of extensive comparisons of the above algorithms for 10 specially designed test problems. The methods are evaluated in terms of robustness, number of function evaluations, and execution time. His results indicate as expected that the methods that use derivatives are generally faster, with the exception of Powell's method which has a high ranking. In a test to determine the effect of the number of variables on the algorithm, the Davidon Fletcher Powell and Powell's method without derivatives were superior to the others.

Others confirm that Powell's method without derivatives is generally considered superior to the empirical (search) methods (Mordecai, 1976). Of the search methods, Nelder and Mead is only good for a small number of variables, Hooke and Jeeves is usually better than the orthogonal searches (Masanao, 1971), and random searches are discussed in Section 4.7.2.

There are, however, possible advantages of the search methods over the more sophisticated methods, for example: no explicit knowledge of the objective function is needed; regularity and continuity of the objective function and the existence of derivatives are not required (Himmelblau, 1972); and it is easy to deal with constraints on the individual variables (Masanao, 1971).

The most promising methods are Powell's method and Hooke and Jeeves' method, which were implemented and are described in more detail below.

F.1 Powell's Method

Powell's method without derivatives is a sophisticated algorithm which operates by successive unidimensional searches which locate minima in each direction. The fundamental idea is that for a quadratic function in n dimensions the minimum can be found by n searches in mutually conjugate directions (Masanao, 1971).

Since for many functions only terms up to the quadratic in the Taylor series expansion are significant in the vicinity of minima, and have continuous second derivatives (Mordecai, 1976), Powell's method converges quickly for these functions.

The algorithm implemented for Powell's method is a direct translation into Pascal of the FORTRAN code in (Himmelblau, pp.444-451) using the Golden section unidimensional search.

F.2 Hooke and Jeeves' Method

Hooke and Jeeves' method is a heuristic method for a n-dimensional direct search which is regarded as a reliable method (Mordecai, 1976) and is simple to program (Masanao, 1971).

Initially the method explores the neighbourhood of a chosen point. If a promising direction is found, successive steps are taken in that direction (a pattern of search is established) until the function stops decreasing. If the pattern search fails the step size is decreased. After several consecutive step size decreases, the pattern is destroyed and a new exploratory search is initiated (Masanao, 1971).

In essence the pattern moves can take long steps in the assumed direction of "valleys", and exploratory moves find the way back to these valleys if a pattern move has climbed out (Mordecai, 1976).

The modified Hooke and Jeeves method with a penalty function outside the allowed ranges of the input vector was implemented in Pascal based on a BASIC program in (Bunday, 1984, pp.93-94).

The performance of Powell's method and Hooke and Jeeves' method are compared to a probabilistic optimisation method in Section 4.7.3.

APPENDIX G

COMPARISON OF PROGRAMMING LANGUAGES FOR IMPLEMENTING TESTABILITY ANALYSIS ALGORITHM

The algorithm described in Chapter 3 for calculating fault detection probabilities for combinational circuits is complex. Manual application to even *moderately sized circuits* is laborious and error-prone, thus making it impractical. As a result a computational aid was needed during the development of the testability analysis algorithm, even though the algorithm was not yet finalized.

This appendix provides a brief comparison of the available programming languages (on Apollo DN3000 or *IBM AT compatible*), and the motivation for the choice of Lisp as the prototyping tool. The discussion is limited to the features necessary for programming the algorithm.

The two overriding considerations in the choice of implementation language were the changing specifications and specific nature of the testability analysis algorithm.

Firstly the language should be considered as a software development tool - therefore not only is the compiler important, but also the total environment. *Simply comparing execution* efficiency of the final program is not sufficient in this case. What is needed is a language implementation which supports an integrated system for effectively modifying and debugging programs. *In essence the philosophy* adopted was to get a working prototype as quickly as possible, and once the program was demonstrably correct, to improve efficiency if necessary. The second consideration is that the algorithm mainly *manipulates symbolic expressions*, with little numerical computation in comparison. A language is required which provides a convenient data structure for storing and modifying symbolic expressions, and powerful *tools for applying transformations to* these expressions.

Ghezzi and Jazayeri (1982) in a text on Programming Language Concepts, classify languages in two broad categories: imperative and applicative. Imperative languages are statement-oriented (Algol or Pascal-like languages), while applicative languages are based on mathematical *functions (therefore* referred to as functional programming).

The argument for imperative programming has traditionally centred on its higher execution speed. This is due to the direct correspondence between conventional computer architecture and the high-level concepts of variables, assignments and repetition (Ghezzi and Jazayeri, 1982). On the other hand the fact that imperative languages are merely abstractions built upon the Von Neumann computer architecture has been criticized. The claim is that the language should be based on real programming needs, dictating the machine architecture, and not the reverse. (Recently computers have been designed specifically for applicative programming, for example dedicated Lisp machines which improve execution speed (Pleszkun and Thazhuthaveetil, 1987), but as yet there are no inexpensive versions.)

John Backus now attacks the concepts of assignment statements and variables as the roots of much evil and a curse passed on by the Von Neumann architecture (Ghezzi and Jazayeri, 1982). He advocates the use of a functional style because of its simple, sounder, mathematical foundations and higher expressive power. Backus originator of FORTRAN, the first high level imperative language.

The available imperative languages are Pascal, C and Modula-2. Superficially C resembles Pascal (Kochan, 1983), but since it was developed for writing systems programs, it is richer than Pascal in lower-level instructions such as bit operations. Since the data structures in C are not better than those of Pascal, and the low level instructions are not necessary, the discussion on Pascal below applies to C as well.

Modula-2 is basically a superset of Pascal, and incorporates a few major and some minor improvements (Wirth, 1984). Two fundamental differences are the concept of the module and the addition of concurrency primitives (Paul, 1984). However the data structures and statement constructs are almost identical to those of Pascal, and thus it will be considered together with Pascal.

The applicative languages available are Lisp and Prolog. Lisp originated in the late 1950's as a list-processing language, and is the most used language for artificial intelligence research (Winston and Horn, 1984). Prolog is based on programming in logic, originating from the Marseille Interpreter in 1973 (Campbell, 1984), and has gained popularity since the announcement of the Japanese "Fifth Generation" project in 1982.

In comparing Pascal, Lisp and Prolog with respect to the two considerations noted above, the first difference is the manner of program execution. A Pascal program is

typically compiled and then executed, producing output data from input data. However, Lisp and Prolog are interactive languages, providing a question-answer type dialogue with the user (Lisp evaluates expressions and Prolog attempts to satisfy goals). This is a useful tool in software development since different parts of a program can be interactively tested independently, without having to modify or recompile the program. The interactive environment is also suitable for incremental program development (Pleszkun and Thazhuthaveetil, 1987), which is helpful in the prototyping stage.

A feature of Lisp and Prolog which simplifies debugging is that programs can be compiled or interpreted. An interpreter is less efficient than a compiler, but usually produces better error diagnostics (Ghezzi and Jazayeri, 1982). The Lisp system available also supported a threaded interpreter, where debugged functions could be compiled and run together with interpreted functions.

The second major consideration noted above is that the main requirement of the program is the manipulation of symbolic expressions. Due to their flexibility, lists are convenient data structures for storing expressions of unknown size and complexity. Lisp and Prolog are list based whereas Pascal is record based. To create linked lists in Pascal, lower level concepts such as pointers must be explicitly used, which can be a distraction and cause unwanted problems in program development.

There are no type declarations in Lisp or Prolog, and binding between variables and types is done dynamically. This provides great flexibility in creating and manipulating data structures, but has disadvantages in terms of programming discipline, program correctness and efficiency (Ghezzi and Jazayeri, 1982). In an experimental program the increased flexibility and extensibility probably outweigh the disadvantages of dynamic type binding.

From the preceding paragraphs it can be seen that Lisp and Prolog are superior to Pascal for writing a prototype symbol manipulation program. The main advantages are: the interactive environment; interpretation or compilation; simple list data structures; and dynamic type binding.

Both Lisp and Prolog have powerful dynamic data structures and are well-suited for prototyping (Ghezzi and Jazayeri, 1982; Borland, 1986), but there is a fundamental conceptual difference. Lisp is a functional language while Prolog is a declarative language. Instead of specifying how to solve a problem, a Prolog program consists of

clauses describing the problem. A more formal definition due to Kahn and Carlsson (Campbell, 1984) is as follows:

> "Prolog is a logic programming language based upon the fact that a theorem prover for the subset of first-order logic restricted to Horn clauses can be very efficiently implemented."

Declarative programming has the tempting appearance of giving something for nothing, since the method of solution for the problem is not specified. As the size of the program grows, however, much time is spent searching for solutions consistent with the given facts and rules (Campbell, 1984). Minimizing the size of the search space requires additional procedural programming.

Prolog is more suitable than Lisp for prototyping systems which can be easily represented as clauses, and make use of the built-in inference engine or theorem prover. However since Lisp was specifically designed for symbol manipulation (Winston and Horn, 1984; Ghezzi and Jazayeri, 1982), it is superior to Prolog in this respect and therefore chosen for the development of the prototype testability analysis program.

PROGRAM LISTINGS FOR TESTABILITY ANALYSIS ALGORITHM

Listings for the testability analysis programs developed in Chapter 4 are included in this appendix. As described in Section 4.6, the program OP.LSP operates on Boolean expressions, and performs as much simplification as possible without running in exponential time. SHANNON.LSP does function decompositions using Shannon's expansion theorem, and CALC.LSP is used to find signal probabilities for Boolean expressions.

The programs below are written in the COMMON LISP dialect of Lisp, and assume a full implementation. All the standard functions and features used (e.g. reduce, backquote macro, and optional parameters) might not be available in subsets of COMMON LISP.

Note: In the programs the XOR symbol ⊕ is written as @.

H.1 OP.LSP

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   op.lsp   Boolean Symbolic Manipulation Program for Testability Analysis   ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

#|
Syntax of expr in BNF

  <expr> ::= ( {=|~} {<atom>|<and-expr>} )

  <and-expr> ::= ( <expr>* )

OR

  <expr> ::= ( {=|~} {<atom>|( <expr>* )} )

|#
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;OP normalises and simplifies the expression.
;(op '<expr>) can be called by itself, or used with the simple menu system
;in MENU.LSP

(defun op (expr) (decode-expr              ;rewrites expression in more readable form
                  (simp-expr                ;simplifies expression
                    (norm-expr expr)        ;normalises into standard syntax
         )
```

```
          )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;NORM-EXPR removes extra brackets,
;          checks for correct form,
;        and rewrites OR's and XOR's in standard format.
;(Standard format is described by the BNF above)

(defun  norm-expr  (expr)
   (cond
        ( (null expr)  '(= bad_nil) )
        ( (atom expr)  (list '= expr) )              ;syntax errors in input
        ( (null (cdr expr))
        (case (car expr)
             ( '&  '(= 1) )
             ( '~&  '(= 0) )
             ( '+  '(= 0) )
             ( '~+  '(= 1) )
             ( '@  '(= 0) )
             ( '~@  '(= 1) )
               ( otherwise (norm-expr (car expr)) )
        )
      )
       ( T  (case (car expr)
          ( '~                                            ;NOT
               (if (= (length expr) 2)
                 (if (atom (cadr expr))
                    (if (numberp (cadr expr))
                        (list '= (- 1 (cadr expr)))    ;(~ const) = 1 - const
                   expr
                 )
                      (not-expr (norm-expr (cdr expr)))
               )
                '(= bad_~)                    ;syntax error in input
            )
          )
          ( '&                                            ;AND
                (if (null (cddr expr))
                   (norm-expr (cadr expr))
                     (list '= (mapcar 'norm-expr (cdr expr)))
             )
          )
          ( '~&                                           ;NAND
                 (if (null (cddr expr))
                      (not-expr (norm-expr (cadr expr)))
                      (list '~ (mapcar 'norm-expr (cdr expr)))
             )
          )
          ( '+                                            ;OR
                 (if (null (cddr expr))
                    (norm-expr (cadr expr))
                       (list '~ (mapcar 'not-expr (mapcar 'norm-expr (cdr expr))))
             )
          )
          ( '~+                                           ;NOR
                 (if (null (cddr expr))
                      (not-expr (norm-expr (cadr expr)))
```

```
                (list '= (mapcar 'not-expr (mapcar 'norm-expr (cdr expr))))
        )
      )
      ( '@                                    ;XOR
          (if (null (cddr expr))
              (norm-expr (cadr expr))
              (reduce 'XOR (mapcar 'norm-expr (cdr expr)))
          )
      )
      ( '~@                                   ;XNOR
          (if (null (cddr expr))
                (not-expr (norm-expr (cadr expr)))
                (not-expr (reduce 'XOR (mapcar 'norm-expr (cdr expr))))
          )
      )
        ( otherwise  '(= bad_expr) )          ;syntax error in input
      )
    )
  )
)

;XOR is used by the standard function REDUCE to return the XOR of
;any number of arguments
(defun XOR (&optional first-expr second-expr)
   (cond
        ( (null first-expr)  '(= 0) )
      ( T  (list '=
             (list
                    (list '~ (list (not-expr first-expr) (not-expr second-expr)))
                 (list '~ (list first-expr second-expr))
             )
          )
        )
     )
)

;NOT-EXPR returns the complement of the <expr>
(defun not-expr (expr)
   (cond
        ( (eq (car expr) '~)  (cons '= (cdr expr)) )
        ( (eq (car expr) '=)  (cons '~ (cdr expr)) )
        ( T (list '~ expr) )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;DECODE rewrites the expression in a more readable form.
;Only pretty-prints expressions.

(defun decode-expr (expr)
   (cond
        ( (eq (car expr) '~)
       (cond
            ( (atom (cadr expr)) expr )
            ( (null (cadr expr)) '0 )
            ( T (cons '~& (mapcar 'decode-expr (cadr expr))) )
        )
      )
```

```
   ( (eq (car expr) '=)
   (cond
      ( (atom (cadr expr)) (cadr expr) )
      ( (null (cadr expr)) '1 )
      ( T (cons '& (mapcar 'decode-expr (cadr expr))) )
   )
   )
   ( T '(bad_expr_to_decode))   ;syntax error in input to decode
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;SIMP-EXPR simplifies the expression.
;Checks for trivial cases, and calls reduce to simplify the expression.

(defun simp-expr (expr)
  (cond
     ( (eq (car expr) '=)  (if (atom (cadr expr))
                     expr
                       (top-and (reduce-and (cadr expr)))
                   )
     )
     ( (eq (car expr) '~)  (if (atom (cadr expr))
                   (if (numberp (cadr expr))
                       (list '= (- 1 (cadr expr)))    ;(~ const) = 1 - const
                     expr
                   )
                     (top-nand (reduce-and (cadr expr)))
                 )
     )
     ( T 'bad_to_simp )        ;syntax error in input exprsssion
  )
)

;TOP-AND accepts an <and-expr> and returns an <expr>
(defun top-and (simp-and-expr)
  (cond
      ( (null simp-and-expr) '(= 1) )
      ( (null (cdr simp-and-expr)) (car simp-and-expr) )
      ( T (list '= simp-and-expr) )
  )
)

;TOP-NAND accepts an <and-expr> and returns an <expr>
(defun top-nand (simp-and-expr)
  (cond
      ( (null simp-and-expr) '(= 0) )
      ( (null (cdr simp-and-expr)) (not-expr (car simp-and-expr)) )
      ( T (list '~ simp-and-expr) )
  )
)

;REDUCE-AND works in two steps:
; 1) ORDERs the expression
; 2) Calls REDUN

;;; It is not essential to ORDER before REDUN
;;; However it is often faster
```

```
(defun reduce-and (and-simp-expr)
      (let ((ordered-and-expr (order and-simp-expr '1 '() '())))
       (redun
            (caddr ordered-and-expr)        ;not-done-list
           (car ordered-and-expr)         ;numeral
            (cadr ordered-and-expr)        ;atom-list
         '()                        ;done-list
       )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;ORDER rearranges the <and-expr> into
;  a numeral
;  list of atoms
;  list of complex expressions
;
;ORDER also removes duplicates and subsitiutes for atoms in the atom-list

;NB: DOES NOT RETURN A VALID <and-expr> FOR EFFICIENCY REASONS
;It returns (<numeral> (<atoms>*) <complex-expressions>*)

(defun order (unordered-list numeral atom-list complex-list)
     (if (= numeral 0)
         '(0)
          (if (null unordered-list)
              (list numeral atom-list complex-list)
         (cond
                ( (numberp (cadar unordered-list))
                 (order (cdr unordered-list)
                        (* numeral (ident (car unordered-list)))
                   atom-list
                   complex-list
                 )
               )
                 ( (atom (cadar unordered-list))
                (order
                          (repl-and-0/1 (car unordered-list) '() (cdr unordered-list))
                   numeral
                       (rev-cons (car unordered-list) atom-list)
                          (repl-and-0/1 (car unordered-list) '() complex-list)
                 )
               )
                 ( T  (if (eq (caar unordered-list) '=)
                     (order
                             (append (cadar unordered-list) (cdr unordered-list))
                        numeral
                         atom-list
                         complex-list
                      )
                     (order
                           (cdr unordered-list)
                        numeral
                         atom-list
                              (rev-cons (car unordered-list) complex-list)
                      )
                  )
```

```
            )
          )
        )
      )
)

(defun ident (numerical-expr)
     (if (eq (car numerical-expr) '=)
        (cadr numerical-expr)
         (- 1 (cadr numerical-expr))
     )
)

(defun rev-cons (car-elem cdr-elem)
    (reverse (cons car-elem (reverse cdr-elem)))
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;REDUN simplifies the ordered expression from ORDER
;and returns the <and-expr>

(defun redun (not-done-list numeral atom-list done-list)
     (if (= numeral 0)
         '((= 0))
          (if (null not-done-list)
           (if (= numeral 1)
              (append atom-list done-list)
                (cons (list '= numeral) (append atom-list done-list))
          )
           (single-redun
               (cdr not-done-list)
                 (simp-expr (car not-done-list))
             numeral
              atom-list
             done-list
           )
        )
      )
)

;SINGLE-REDUN works on a single element of <expr>
(defun single-redun (not-done-list done-expr numeral atom-list done-list)
    (cond
        ( (numberp (cadr done-expr))
        (redun
            not-done-list
              (* numeral (ident done-expr))
            atom-list
            done-list
        )
       )
        ( (atom (cadr done-expr))
        (redun
              (repl-and-0/1 done-expr
                    '()
                       (append done-list not-done-list)
              )
            numeral
```

```
          (rev-cons done-expr atom-list)
          '()
      )
    )
   ( T  (if (eq (car done-expr) '=)          ;i.e. can be flattened
        (redun                    ;can also try with SIMP-REDUN
          ;;;  not-done-list            ;(use with simp-redun)
            (append (cadr done-expr) not-done-list)
          numeral
           atom-list
           done-list
        )
        (redun
           not-done-list
          numeral
           atom-list
             (rev-cons done-expr done-list)
          )
        )
      )
    )
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;SIMP-REDUN simplifies the ordered expression
;and returns the <and-expr>

;NB; It does not simplify each element of the AND since the elements were
;    simplified before flattening.

;Using SIMP-REDUN in place of REDUN where noted in SINGLE-REDUN generally
;speeds up the simplification.

(defun simp-redun (not-done-list simp-list numeral atom-list done-list)
    (if (= numeral 0)
        '((= 0))
        (if (null simp-list)
        (redun
            not-done-list
          numeral
           atom-list
          done-list
        )
          (simp-single-redun
            not-done-list
             (cdr simp-list)
             (car simp-list)
          numeral
           atom-list
          done-list
        )
      )
    )
)

(defun simp-single-redun (not-done-list simp-list done-expr numeral atom-list
done-list)
   (cond
```

```
( (numberp (cadr done-expr))
 (simp-redun
   not-done-list
   simp-list
    (* numeral (ident done-expr))
   atom-list
   done-list
 )
)
( (atom (cadr done-expr))
 (redun
      (repl-and-0/1 done-expr
              '()
                   (append done-list simp-list not-done-list)
   )
   numeral
     (rev-cons done-expr atom-list)
    '()
  )
)
( T  (if (eq (car done-expr) '=)          ;i.e. can be flattened
       (simp-redun
          not-done-list
            (append (cadr done-expr) simp-list)
          numeral
           atom-list
          done-list
       )
        (simp-redun
           not-done-list
            simp-list
          numeral
           atom-list
             (rev-cons done-expr done-list)
        )
      )
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;REPL-AND-0/1 and REPL-EXPR-0/1 traverse the tree of the expression,
;replacing for terms and simplifying at the same time.
;REPL-AND-0/1 replaces in an <and-expr>
;REPL-EXPR-0/1 replaces in an <expr>

(defun repl-and-0/1 (old-expr done-and not-done-and)
    (if (null not-done-and)
     done-and
        (let ( (first-expr (repl-expr-0/1 old-expr (car not-done-and))) )
       (cond
            ( (equal first-expr '(= 0))  '((= 0)) )
          ( (equal first-expr '(= 1))
             (repl-and-0/1 old-expr done-and (cdr not-done-and))
          )
            ( T  (repl-and-0/1
                old-expr
                 (cons first-expr done-and)
```

```
                    (cdr not-done-and)
                )
              )
            )
          )
        )
)

(defun repl-expr-0/1 (old-expr expr)        ;returns a replaced <expr>
   (cond
       ( (equal old-expr expr)  '(= 1) )
        ( (equal (cadr old-expr) (cadr expr))  '(= 0) )
       ( (atom (cadr expr))  expr )
         ( T   (let ( (repl-and-expr (repl-and-0/1 old-expr () (cadr expr))) )
                (if (equal (car expr) '=)
                    (top-and repl-and-expr)
                    (top-nand repl-and-expr)
                )
              )
           )
        )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

H.2 SHANNON.LSP

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   shannon.lsp     Program for Shannon's Expansion in Testability Analysis    ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;finds an atom and applies Shannon's Expansion Theorem
;i.e. the identity  f(A,...) = A.f(1,...) + ~A.f(0,...)

(defun shannon (atom-expr expr)
   (top-shannon
      atom-expr
          (simp-expr (repl-expr-0/1   (not-expr atom-expr) expr))
         (simp-expr (repl-expr-0/1   atom-expr expr))
   )
)

(defun and-shannon (atom-expr and-expr)
   (top-shannon
      atom-expr
          (simp-expr (repl-and-0/1   (not-expr atom-expr) '() and-expr))
         (simp-expr (repl-and-0/1   atom-expr '() and-expr))
   )
)

;SHANN is a simplified shannon which does no simplifying except for repl-expr-0/1
and top-shannon
(defun shann (atom-expr expr)
   (top-shannon
      atom-expr
          (repl-expr-0/1   (not-expr atom-expr) expr)
         (repl-expr-0/1   atom-expr expr)
```

```
    )
)

(defun top-shannon (atom-expr f0 f1)
   (if (equal f0 '(= 0))                                          ;0?
        (if (equal f1 '(= 0))
         '(= 0)                                                    ;00
          (if (equal f1 '(= 1))
          atom-expr                                                ;01
            (if (and (eq (car f1) '=) (listp (cadr f1)))          ;0X
                (list '= (cons atom-expr (cadr f1)))
                (list '= (list atom-expr f1))
            )
         )
     )
     (if (equal f0 '(= 1))                                         ;1?
          (if (equal f1 '(= 0))
          (not-expr atom-expr)                                     ;10
            (if (equal f1 '(= 1))
            '(= 1)                                                 ;11
              (if (and (eq (car f1) '~) (listp (cadr f1)))        ;1X
                  (list '~ (cons atom-expr (cadr f1)))
                    (list '~ (list atom-expr (not-expr f1)))
            )
          )
       )
       (if (equal f1 '(= 0))                                       ;X0
            (if (and (eq (car f0) '=) (listp (cadr f0)))
               (list '= (cons (not-expr atom-expr) (cadr f0)))
               (list '= (list (not-expr atom-expr) f0))
         )
           (if (equal f1 '(= 1))
              (if (and (eq (car f0) '~) (listp (cadr f0)))        ;X1
                  (list '~ (cons (not-expr atom-expr) (cadr f0)))
                    (list '~ (list (not-expr atom-expr) (not-expr f0)))
             )
             (list '~
              (list                                                ;XX
                        (if (and (eq (car f0) '~) (listp (cadr f0)))
                        (list '~ (cons (not-expr atom-expr) (cadr f0)))
                           (list '~ (list (not-expr atom-expr) f0))
                 )
                        (if (and (eq (car f1) '~) (listp (cadr f1)))
                         (list '~ (cons atom-expr (cadr f1)))
                         (list '~ (list atom-expr f1))
                 )
               )
             )
           )
         )
       )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;TOP-FIND-SHANNON-VARS takes an <expr> and returns a list of the variables
;and their occurences in the expression.
;The form is (var (depth1 negation1) (depth2 negation2) ... )
```

```
(defun  top-find-shannon-vars  (expr)
    (find-shannon-vars
      (cadr expr)
      (case (car expr)
            ('~  1 )
            ('=  0 )
      )
    )
)

(defun find-shannon-vars (atom-or-and-expr &optional (negations 0) (depth 0)
(vars-list  '()))
   (cond
       ( (null atom-or-and-expr)  vars-list )
       ( (numberp atom-or-and-expr) vars-list )
       ( (atom atom-or-and-expr)
          (add-to-list negations depth vars-list atom-or-and-expr)
       )
       ( T
          (find-shannon-vars
             (cdr atom-or-and-expr)
          negations
          depth
            (find-shannon-vars
              (cadar atom-or-and-expr)
              (case (caar atom-or-and-expr)
                 ('~  (+ negations 1) )
                 ('=  negations )
              )
              (+ depth 1)
              vars-list
            )
          )
       )
   )
)

;ADD-TO-LIST adds the current occurence of the variable to the list of its
;previous occurences in the <expr>
(defun  add-to-list  (negations depth vars-list element)
    (cond
        ( (null vars-list)
          '(list (cons element '(list '(list depth negations))))
        )
        ( (eq element (caar vars-list))
         (cons
           (cons
             element
                (cons '(list depth negations) (cdar vars-list))
           )
                (cdr vars-list)
         )
        )
        ( T
          (cons
                (car vars-list)
                 (add-to-list negations depth (cdr vars-list) element)
```

```
(defun  top-find-shannon-vars  (expr)
    (find-shannon-vars
      (cadr expr)
      (case (car expr)
            ('~  1 )
            ('= 0 )
      )
    )
)

(defun find-shannon-vars  (atom-or-and-expr &optional  (negations 0)  (depth 0)
(vars-list  '()))
  (cond
       ( (null atom-or-and-expr)   vars-list )
       ( (numberp atom-or-and-expr) vars-list )
       ( (atom atom-or-and-expr)
          (add-to-list negations depth vars-list atom-or-and-expr)
       )
       ( T
          (find-shannon-vars
             (cdr atom-or-and-expr)
          negations
          depth
            (find-shannon-vars
               (caar atom-or-and-expr)
               (case (caar atom-or-and-expr)
                     ('~  (+ negations 1) )
                     ('= negations )
               )
               (+ depth 1)
               vars-list
            )
          )
       )
  )
)

;ADD-TO-LIST adds the current occurence of the variable to the list of its
;previous occurences in the <expr>
(defun  add-to-list  (negations depth vars-list element)
    (cond
          ( (null vars-list)
            (list (cons element (list (list depth negations))))
          )
          ( (eq element (caar vars-list))
           (cons
              (cons
                element
                   (cons (list depth negations) (cdar vars-list))
              )
                (cdr vars-list)
           )
          )
          ( T
            (cons
                 (car vars-list)
                  (add-to-list negations depth (cdr vars-list) element)
```

```
        )
      )
    )
)

;TRY-SHANNON tries Shannon's theorem with every variable in the expression.
;Useful when experimenting with different heuristics.
(defun try-shannon (expr)
     (apply-shannon (top-find-shannon-vars expr) expr)
)

(defun apply-shannon (vars-info expr)
     (if (null vars-info)
       '()
        (if (< (length (car vars-info)) 3)
           (apply-shannon (cdr vars-info) expr)
        (cons
          (progn
               (terpri)
                 (print (car vars-info))
                   (print (shannon (list '= (caar vars-info)) expr))
          )
              (apply-shannon (cdr vars-info) expr)
          )
        )

)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;At present HEURISTIC returns for each variable:
;   The number of occurrences.
;   A scaled factor depending on the depth of the occurrences.
;        (each level down is assumed to halve the contribution of the occurrence)
;   The difference between even and odd occurrences of the variable.
;   A scaled even/odd difference depending on the depth of the occurrences.

(defun  heuristic  (vars-info)
     (if (null vars-info)
       ()
      (cons
             (cons (caar vars-info) (heuristic-for-var '0 '0 '0 '0 (cdar vars-info)))
           (heuristic (cdr vars-info))
      )
    )
)

;HEURISTIC-FOR-VAR calculates the heuristics for a single variable in the
expression
(defun  heuristic-for-var  (inst  s-tot  diff-neg  s-neg  var-info)
   (cond
       ( (null var-info)   (list inst s-tot diff-neg s-neg) )
       ( T   (let ( (weight (/ 1.0 (expt2 (caar var-info)))) )
              (heuristic-for-var
              (+ inst 1)
              (+ s-tot weight)
               (if (oddp (cadar var-info))
                 (+ diff-neg 1)
                 (- diff-neg 1)
```

```
              )
              (if (oddp (cadar var-info))
                (+ s-neg weight)
                (- s-neg weight)
              )
              (cdr var-info)
            )
          )
        )
      )
    )
)

(defun expt2 (exp)   ;Convenience for calculating common powers of 2 fast.
  (case exp
     (0  1)      (1  2)
     (2  4)      (3  8)
     (4  16)      (5  32)
     (6  64)      (7  128)
     (8  256)     (9  512)
      (10 1024)     (11 2048)
      (otherwise (expt 2 exp))     ;call exponent function
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;Boolean Difference operations.

;;; This does [f(1) XOR f(0)]
(defun bool-diff (atom-expr expr)
     (let ( (f1 (repl-expr-0/1   atom-expr  expr))
               (f0 (repl-expr-0/1   (not-expr atom-expr) expr))
       )
         (simp-expr `(= (
                                  (~ ((~ (,f1)) (~ (,f0))))
                          (~ (,f1 ,f0))
                        )
                      )
         )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

H.3 CALC.LSP

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   calc.lsp      Program for Calculating Probabilities in Testability Analysis     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
#|

CALC takes a normalised boolean expression and an association-list of the
signal probabilities in the expression and returns the signal probability
for the expression.

It is assumed that all the variables are independent.

The algorithm adopted is outlined below:
```

```
Start
  While variables remain Do   (shannon-calc)
    [
       Repeat   (single-calc)
         [
             Replace all single variables by their signal probabilities
            simplify the expression
         ]
       Until no single variables remain

       Find a suitable variable to apply Shannon's Expansion theorem
           (find-exp-var)

       Apply Shannon's Theorem once with respect to the chosen variable
         (shannon)
    ]
End

| #

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;SHANNON-CALC

(defun shannon-calc (init-expr prob-alist)
      (let ( (expr (single-calc init-expr prob-alist)) )      ;takes care of single vars
             (let* ( (var-heuristics (heuristic (top-find-shannon-vars init-expr))) )
             (if var-heuristics
               (shannon-substitute
                   (choose-variable var-heuristics)
                 prob-alist
                 init-expr
             )
                 (number (simp-expr init-expr))          ; i.e. no variables left
          )
       )
    )
)

;SHANNON-SUBSTITUTE is a probabilistic expansion theorem
(defun shannon-substitute (chosen-var prob-alist expr)
      (let* ( (var-probability (cdr (assoc chosen-var prob-alist)))
              (F1 (simp-expr (repl-expr-0/1 `(= ,chosen-var) expr)))
              (F0 (simp-expr (repl-expr-0/1 `(~ ,chosen-var) expr)))
       )
       (+
              (* var-probability (shannon-calc F1 prob-alist))
              (* (- 1 var-probability) (shannon-calc F0 prob-alist))
       )
    )
)

;CHOOSE-VARIABLE returns the anticipated best variable for performing Shannon's
;expansion theorem.
;The function below reads in the chosen variable, but it can be easily changed
;to be a function of the heuristics contained in var-info.
;The best heuristic found was to choose the variable with the maximum weighted
;sum of occurences.
```

```
(defun choose-variable (var-info)
    (print var-info)
   (read)
)

(defun number (expr)
  (cond
      ( (eq (car expr) '=) (cadr expr) )
       ( (eq (car expr) '~) (- (cadr expr) 1) )
     ( T (print "Number error") 0 )                  ;called incorrectly
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;SINGLE-CALC substitutes directly for all single variables by their signal
;probabilities.

(defun single-calc (init-expr prob-alist)
    (do* ( (expr (simp-expr init-expr) (simp-expr (mult-substitute sing-var-list
prob-alist  expr)))
              (sing-var-list (single-vars init-expr) (single-vars (print expr))) )
        ( (null sing-var-list)  expr)
       (print expr)
  )
)

(defun mult-substitute (sing-var-list prob-alist expr)
  (cond
       ( (null expr)  () )
       ( (atom expr)  (if (member expr sing-var-list)
                  (cdr (assoc expr prob-alist))
                expr
            )
     )
        ( T   (cons (mult-substitute sing-var-list prob-alist (car expr))
                (mult-substitute sing-var-list prob-alist (cdr expr))
          )
      )
   )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Utilities

;SINGLE-VARS returns a list of all the variables used only once.
(defun single-vars (expr)
      (kill-dup-vars (all-vars expr))
)

;LIST-VARS returns a SET of all the variables used in the expression.
(defun list-vars (expr)
     (remove-dup-vars (all-vars expr))
)

;ALL-VARS returns a list of all the variables in the expression.
(defun all-vars (exp)
   (cond
       ( (null exp)  () )
```

```
        ( (atom exp)  (if (or (eq exp '=) (eq exp '~) (numberp exp))
                          ()
                          (list exp)
                      )
        )
          ( T (append (all-vars (car exp)) (all-vars (cdr exp))) )
      )
)

;KILL-DUP-VARS removes all duplicate variables.
(defun  kill-dup-vars  (exp)
  (cond
       ( (null exp)  () )
      ( (atom exp)  exp )
       ( (member (car exp) (cdr exp))
          (kill-dup-vars (remove (car exp) (cdr exp)))
       )
          ( T   (cons (car exp) (kill-dup-vars (cdr exp))) )
      )
)

;alternative implementation of KILL-DUP-VARS
(defun  'kill-dup-vars2  (not-done-list  &optional  (done-list  '()))
  (cond
       ( (null not-done-list) done-list )
       ( (atom not-done-list)  (cons not-done-list done-list) )
        ( (member (car not-done-list) (cdr not-done-list))
           (kill-dup-vars2 (remove (car not-done-list) (cdr not-done-list)) done-list)
       )
          ( T   (kill-dup-vars2 (cdr not-done-list) (cons (car not-done-list) done-list))
)
    )
)

;REMOVE-DUP-VARS replaces all duplicate variables by a single occurence.
(defun  remove-dup-vars  (exp)
  (cond
       ( (null exp)  () )
      ( (atom exp)  exp )
       ( (member (car exp) (cdr exp))
         (cons (car exp) (remove-dup-vars (remove (car exp) (cdr exp))))
       )
          ( T   (cons (car exp) (remove-dup-vars (cdr exp))) )
      )
)


;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
; TEMPORARY functions for entering the probabilities of the variables in the <expr>

(defun  assign-probs  (expr)
     (assign  (list-vars  expr))
)

(defun  assign  (expr)
    (cond
       ( (null expr)  () )
       ( T   (cons (read-prob (car expr)) (assign (cdr expr))) )
```

```
    )
)

(defun  read-prob  (var)
    (format T "Enter probability for variable ~A :" var)
    (cons var (read))
)
```

REFERENCES

Abadir, M.S. and Reghbati, H.K. LSI Testing techniques, *IEEE Micro*, February 1983, pp. 34-50.

Agrawal, V.D. An Information Theoretic Approach to Digital Fault Testing, *IEEE Transactions on Computers*, Vol. C-30, No. 8, August 1981, pp. 582-587.

Agrawal, V.D. and Mercer, M.R. Testability Measures - What Do They Tell Us?, *The Proceedings of the 1982 Test Conference*, 1982, pp. 391-396.

Barber, M.R. and Satre, W.I. Timing Accuracy in Modern ATE, *IEEE Design and Test of Computers*, April 1987, pp. 22-30.

Beineke, L.W. and Wilson, L.W. *Selected Topics in Graph Theory 2*, London:Academic Press, 1983.

Bending, M.J. Hitest : A Knowledge-Based Test Gener[illegible] *IEEE Design and Test of Computers*, May 1984, pp. 83-92.

Bennetts, R.G. *Design of Testable Logic Circuits*, Finland: Addison-Wesley Publishers Limited, 1984.

Borland *Turbo Prolog*, Borland Inc, 1986.

Breuer, M. A. *Design Automation of Digital Systems, Vol 1*, New Jersey:Prentice-Hall Inc, 1972.

Breuer, M.A. and Friedman, A.D. *Diagnosis & Reliable Design of Digital Systems*, California:Computer Science Press, 1976.

Bunday, B. *Non-linear Programming*, London:Edward Arnold, 1984.

Campbell, J. A. (Editor) *Implementations of Prolog*, Chichester, West Sussex:Halsted Press, 1984.

Carre, B. *Graphs and Networks*, Oxford:Claredon Press, 1979.

Daniels, R.G. and Bruce, W.C. Built-In-Self-test Trends in Motorola Microprocessors, *IEEE Design and Test of Computers*, April 1985, pp. 64-71.

David, H. A. *Order Statistics*, New York:Wiley and Sons Inc., 1970.

Franco, P. *PRODICTA - A Probabilistic Digital Integrated Circuit Testability Analysis*, Design Report 12AD/86, Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1986

Garey, M. R. and Johnson D. S. *Computers and Intractability. A guide to the Theory of NP-Completeness*, San Francisco:W. H. Freeman and Company, 1979.

Gelsinger, P.P. Design and Test of the 80386, *IEEE Design and Test of Computers*, June 1987, pp. 42-50.

GenRad, *HILO-3 Product Description*, Milpitas:GenRad, 1985.

Ghezzi, C. and Jazayeri, M. *Programming Language Concepts*, New York:Wiley and Sons, 1982.

Goldstein, L.H. Controllability/Observability Analysis of Digital Circuits, *IEEE Transactions On Circuits and Systems*, Vol. CAS-26, No. 9, September 1979, pp. 685-693.

Golumbic, M.C. *Algorithmic Graph Theory and Perfect Graphs*, New York:Academic Press, 1980.

Greene, J. W. and Supowit,K. J. Simulated Annealing Without Rejected Moves, *IEEE Trans. on Computer Aidad Design*, Vol. CAD-5, No 1, January 1986.

Guenther, W. C. *Sampling Inspection in Statistical Quality Control*, London:Charles Griffin and Company, 1977.

Himmelblau, D. M. *Applied Nonlinear Programming*, New York:McGraw-Hill, 1972.

Ibarra, O.H. and Sahni, S.K. Polynomially Complete Fault Detection Problems, *IEEE Transactions on Computers*, Vol. C-24, No. 3, March 1975, pp. 242-249.

Illman, R.J. Self-tested Data Flow Logic : A New Approach, *IEEE Design and Test of Computers*, April 1985, pp. 50-58.

Keyes, R.W. Physical Limits in Digital Electronics, *Proceedings of the IEEE*, Vol. 63, No. 5, May 1975, pp. 740-767.

Kirkpatrick S., Gelatt, C. D. Jr, and Vecchi, M. P. Optimization by simulated Annealing, *Science*, vol. 220, No. 4598, pp. 671-680, May 1983.

Kochan, S. G. *Programming in C*, New Jersey:Hayden, 1983.

Masanao A. *Introduction to Optimization Techniques. Fundamentals and Applications of Nonlinear Programming*, New York:The Macmillan Company, 1971.

McCluskey, E.J. and Bozorgui-Nesbat, S. Design for Autonomous Test, *IEEE Transactions on Computers*, Vol. C-30, No. 11, November 1981, pp. 866-874.

McCluskey, E.J. Verification Testing - A Pseudoexhaustive Test Technique, *IEEE Transactions on Computers*, Vol. C-33, No. 6, June 1984, pp. 541-546.

McCluskey, E.J. et.al. Probability Models for Pseudorandom Test Sequences, *IEEE Transactions on Computer Aided Design*, Vol. 7, No. 1, January 1988, pp. 68-74.

Moore, G.E. Interviewed by Design and Test, *IEEE Design and Test of Computers*, November 1984, pp. 15-23.

Mordecai A. *Nonlinear Programming Analysis and Methods*, New Jersey:Prentice-Hall, 1976.

Muehldorf, E.I.and Savkar, A.D. LSI Logic Testing-An Overview, *IEEE Transactions On Computers*, Vol. C-30, No. 1, January 1981, pp. 1-16.

Paul, R. J. An Introduction to Modula-2, *Byte*, August 1984, pp.195-210.

Pleszkun, A. R. and Thazhuthaveetil, M. J. The Architecture of Lisp Machines, *Computer*, March 1987, pp.35-44.

Runyon, S. The Great ASIC Wave Gathers Force, *Electronics*, 6 August 1987.

Savir, J. Syndrome-Testable Design of Combinational Circuits, *EEE Transactions On Computers*, Vol. C-29, No. 6, June 1980, pp. 442-451.

Savir, J. Good Controllability and Observability Do Not Guarantee Good Testability, *IEEE Transactions On Computers*, Vol. C-32, No. 12, December 1983, pp. 1198-1200.

Savir, J., Ditlow, G.S. and Bardell, P.H., Random Pattern Testability, *IEEE Transactions on Computers*, Vol. C-33, No. 1, January 1984, pp. 79-90.

Sequin, C.H. Managing VLSI Complexity: An Outlook, *Proceedings of the IEEE*, Vol. 71, No. 1, January 1983, pp. 149-166.

Thomas, J.J. Common Misconceptions in Digital Test Generation, *Computer Design*, January 1977, pp. 89-94.

Waller, L. Can Big Chip Houses Make it in ASICs?, *Electronics*, 6 August 1987.

Williams, T.W. and Parker, K.P. Design for testability - A Survey, *IEEE Transactions on Computers*, Vol. C-31, No. 1, January 1982, pp. 2-15.

Williams, T.W. et.al. Bounds and Analysis of Aliasing Errors in Linear Feedback Shift Registers, *IEEE Transactions on Computer Aided Design*, Vol 7, No. 1, January 1988, pp. 75-83.

Winston, P. H. and Horn, B. K. P. *Lisp*, 2nd ed.,Massachusetts:Addison-Wesley, 1984.

Wirth, N. History and Goals of Modula-2, *Byte*, August 1984, pp.145-152.

Wunderlich, H. PROTEST : A Tool For Probablistic Testability Analysis, *22nd Design Automation Conference*, 1985, pp. 204-211.