Test Set Generation and Optimisation using Evolutionary Algorithms and Cubical Calculus

Jasbir S. Takhar

A thesis submitted in partial fulfillment of the requirements of

Sheffield Hallam University

for the degree of Doctor of Philosophy

December 2003

Acknowlegements

I would like to thank, first and foremost, my supervisor Dr. Daphne Gilbert for her inspiration, encouragement and support throughout my PhD studies. There were many occasions on which her uplifting words would force me to think about a problem in another way or help me overcome the writer's block that many a thesis-writer inevitably experiences. Without her hard work and dedication, this work would not have been possible.

I would like to acknowledge other members of my supervisory and advisory team for their support including Dr. Hugh Porteous and Paul Davies of Sheffield Hallam University and Dr. Anthony Hall of Hull University. I would also like to thank Dr. Mike Thomlinson, who as a senior member of The School of Science and Mathematics, provided help in both administrative and academic matters, including the funding that enabled my participation in the IEEE Midwest Symposium on Circuits and Systems in California, USA. I must also mention Prof. Bob Green and Dr. Hamid Dehghani both, of Sheffield Hallam University, for providing an excellent laboratory environment in which I conducted much of this work.

The test set minimization algorithm was proven to be successful by minimizing test sets generated by Prof. Raimund Ubar's team at Tallin Technical University, Estonia. In particular, I would like to thank Jaan Raik and Annti Markus for actually generating the test sets and providing technical support. This collaboration was one of the highlights of this research project as it validated the minimization work.

Dedication

To Mum, Dad, Sam and Didi.

Abstract

As the complexity of modern day integrated circuits rises, many of the challenges associated with digital testing rise exponentially. VLSI technology continues to advance at a rapid pace, in accordance with Moore's Law, posing evermore complex, NP-complete problems for the test community. The testing of ICs currently accounts for approximately a third of the overall design costs and according to the Semiconductor Industry Association, the per-transistor test cost will soon exceed the per-transistor production cost. Given the need to test ICs of ever-increasing complexity and to contain the cost of test, the problems of test pattern generation, testability analysis and test set minimisation continue to provide formidable challenges for the research community. This thesis presents original work in these three areas.

Firstly, a new method is presented for generating test patterns for multiple output combinational circuits based on the Boolean difference method and cubical calculus. The Boolean difference method has been largely overlooked in automatic test pattern generation algorithms due to its cumbersome, algebraic nature. It is shown that cubical calculus provides an elegant and economical technique for solving Boolean difference equations. Formal mathematical techniques are presented involving the Boolean difference and cubical calculus providing, a test pattern generation method that dispenses with the need for costly circuit simulations. The methods provide the basis for test generation algorithms which are suitable for computer implementation.

Secondly, some of the core test pattern generation computations outlined above also provide the basis of a new method for computing testability measures such as controllability and observability. This method is effectively a very economical spin-off of the test pattern generation process using Boolean differences and cubical calculus.

The third and largest part of this thesis introduces a new test set minimization algorithm, GA-MITS, based on an evolutionary optimization algorithm. This novel approach applies a genetic algorithm to find minimal or near minimal test sets while maintaining a given fault coverage. The algorithm is designed as a post-processor to minimise test sets that have been previously generated by an ATPG system and is thus considered a static approach to the test set minimisation problem. It is shown empirically that GA-MITS is remarkably successful in minimizing test sets generated for the ISCAS-85 benchmark circuits and hence potentially capable of reducing the production costs of realistic digital circuits.

Keywords: *Digital testing, automatic test pattern generation, ATPG, Boolean difference, cubical calculus, testability analysis, controllability, observability, test set minimization and compaction, combinatorial optimization, evolutionary algorithms, genetic algorithms.*

Chapter 1. Digital Testing

1.1 Introduction

Over the last three decades, the dramatic advances in VLSI technology have produced tremendous challenges for the test community. As circuits become larger and new fabrication techniques allow increased gate density, the complexity of digital testing increases exponentially. As this work will show, many of the problems in digital testing are NP-hard and as a result, their solution requires a multi-disciplinary approach. Many of these hard problems have been tackled with traditional, established techniques as well as some of the more recent innovations within the fields of computer science, engineering, and mathematics. Historically, digital testing has been a game of 'catch-up' with circuit design and manufacture technology. As soon as one test problem is adequately solved, the blistering pace of VLSI technology introduces new, even more complex problems. This can be seen today with the rapid approach of nanotechnology which seemingly renders traditional testing techniques, such as I_{ddq} testing, as ineffective [1].

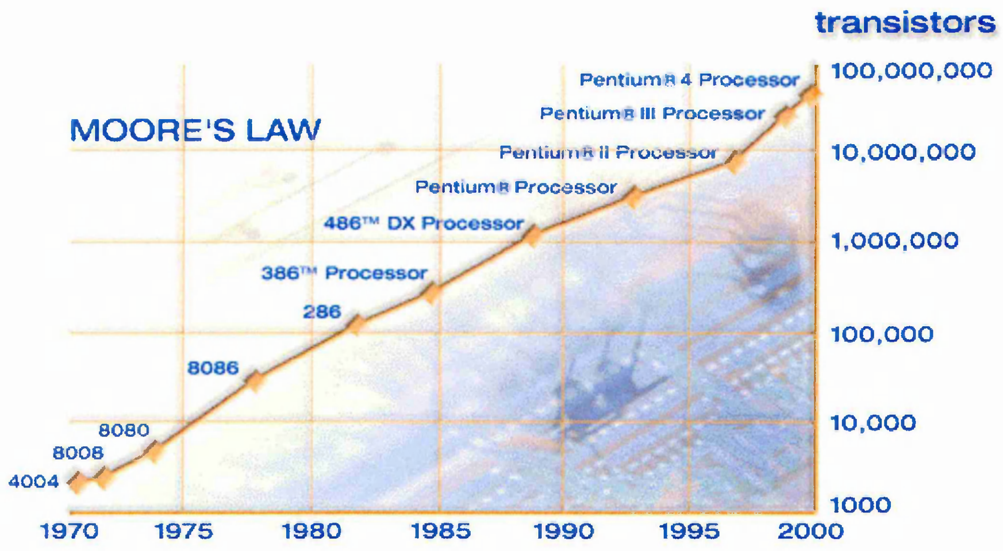
The microprocessor is the most important class of digital circuit and is, by far, the most complex. The microprocessor has changed the very fabric of everyday life in most parts of the world and continues to do so. As digital technology advances, giving rise to higher levels of integration, microprocessors increasingly find themselves in our everyday lives. In the late 1960s and early 1970s, integrated circuits and microprocessors superseded the slide-rule. In the 1980s they gave rise to the personal computer that today, provides more computing power on our desks than was used to put man on the moon. Over the last decade or so, advances in communications, including the internet and mobile technology, have changed the way people interact with one another. Looking ahead, the future for digital devices seems to hold promises of nanotechnology that may be as revolutionary as the microprocessor itself. One of the common threads in technology research has been to make things smaller and faster. Smaller and faster integrated circuits enable smaller and more powerful mobile devices for example. However, it is just this goal that gives rise to the enormous challenges in digital testing.

It is important at this point to understand the seemingly relentless pace of change in VLSI technology as it will offer the reader some perspective as to why digital testing remains such a fertile topic for researchers. This continual change in the semiconductor industry will be discussed in the context of the microprocessor and more specifically, those microprocessors developed by the Intel Corporation. The astounding evolution of this device over the past thirty years will help the reader understand the nature of the industry and the tremendous challenges it poses for the test community.

Gordon Moore, founder of the worlds largest microprocessor company Intel Corporation, made a very famous observation about the pace of change in the development and manufacture of integrated circuits. In his now famous paper [2], Moore predicted an exponential growth in transistor density on an integrated circuit and that this trend would continue for some decades. His observation, which was christened *Moore's Law* states,

The number of transistors on an integrated circuit doubles approximately every eighteen months, while the cost of the circuit decreases by half.

Moore made this prediction four years after the very first integrated circuit was developed in 1961 and over forty years later, his observation still holds true. Figure 1.1 illustrates Moore's Law at work by graphically depicting the number of transistors in Intel's family of microprocessors over the last thirty years or so.



(a)

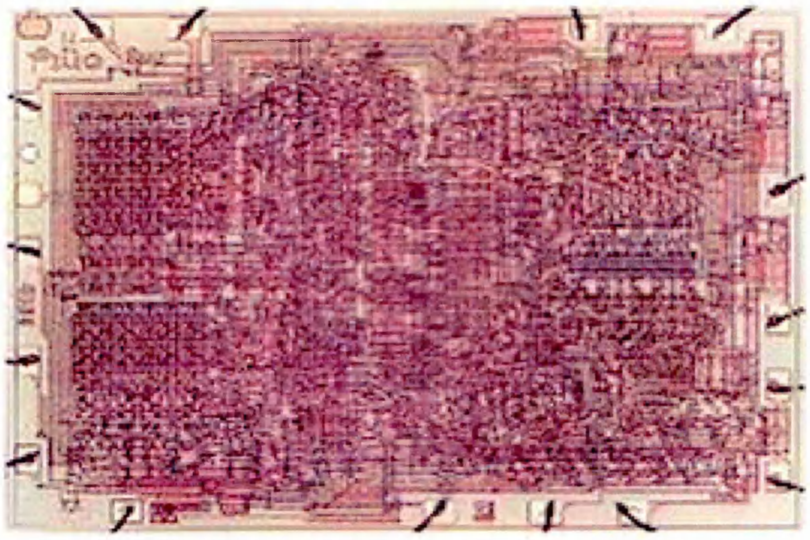
Processor Name	Year of introduction	Transistors
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386™ processor	1985	275,000
486™ DX processor	1989	1,180,000
Pentium® processor	1993	3,100,000
Pentium II processor	1997	7,500,000
Pentium III processor	1999	24,000,000
Pentium 4 processor	2000	42,000,000

(b)

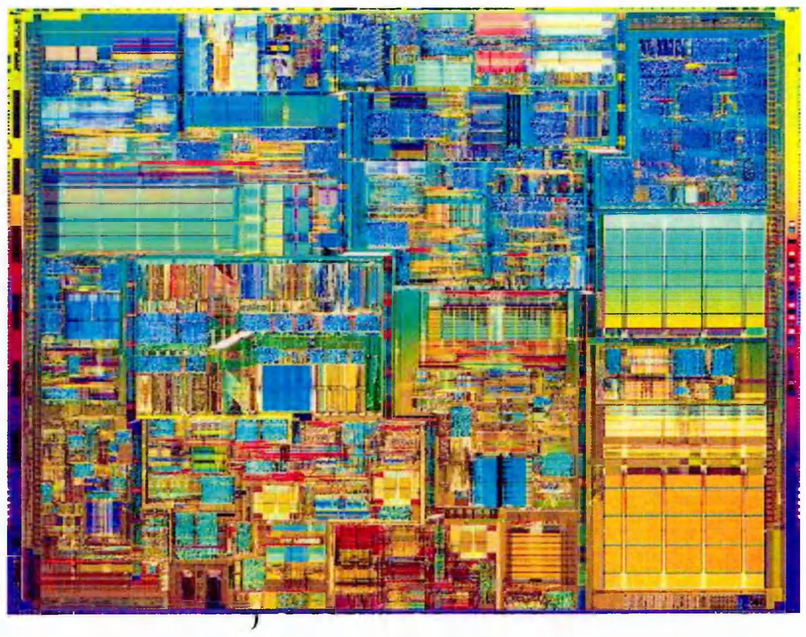
Figure 1.1 *Moore's Law illustrated by the history of Intel microprocessors. Figure (a) is a curve depicting the data points in (b) and shows the exponential growth in the number of transistors on an integrated circuit. Reproduced courtesy of Intel Corporation.*

The world's first microprocessor, the Intel 4004 introduced in November 1971, contained 2,250 transistors, ran at a clock speed of 108k Hertz and was able to perform 60,000 calculations per second. The 4004 is shown in diagram 1.2(a). As can be seen from the above curve, the number of transistors increase exponentially over the years and in 2000, Intel introduced the Pentium® 4 processor, which contained 42 million transistors and ran at an initial clock-speed of 1.5G Hertz. The Pentium 4 is shown in Figure 1.2(b). Intel states that, if over the same period, increases in car speeds kept pace with the increases in microprocessor speeds, one could cover the distance from San Francisco to New York (a distance of approximately 3000 miles) in 13 seconds. This fact alone illustrates the astounding developments in VLSI technology and gives some indications of the challenges faced by designers and testers alike.

Moore's law cannot continue forever as we are approaching the limits of physics as we know it. However, there is confidence that integrated circuits will continue to follow the curve for at least another decade with innovations in process and fabrication technology. In fact, Moore himself, as recently as February 2003 gave a presentation entitled, "No exponential is forever... but we can delay it forever" [4], in which he expresses his confidence that the semiconductor industry will overcome many of the challenges that face it over the next decade and continue to pack ever more transistors on an integrated circuit. In fact, in early 2003, Intel introduced the latest in its line of microprocessors, the Pentium 4-M, containing over 70,000,000 transistors.



(a)



(b)

Figure 1.2 (a) *Intel 4004 microprocessor containing 2,250 transistors, introduced in 1971.* **(b)** *Intel Pentium® 4 microprocessor containing 42,000,000 transistors introduced in 2000. Reproduced Courtesy of Intel Corporation.*

All of this VLSI research and development at the outer edges of our knowledge in such fields as chemistry, physics, engineering and mathematics is to be celebrated. However, these developments pose huge challenges for the test community who are in a continuous race to keep up. The costs associated with testing integrated circuits are huge and are of great concern to the semiconductor industry. In his paper [5] addressing many of the key issues in test, Kenneth Thompson of Intel, estimates his company spends a third of its capital expenditure on test and test equipment and he does not see this percentage decreasing any time soon. As further evidence of the increasing complexity and cost of test, the Semiconductor Industry Association (SIA), a well respected authority on the semiconductor industry, estimates that the cost of testing integrated circuits will actually surpass the cost of their production. The graph given in Figure 1.3 illustrates this.

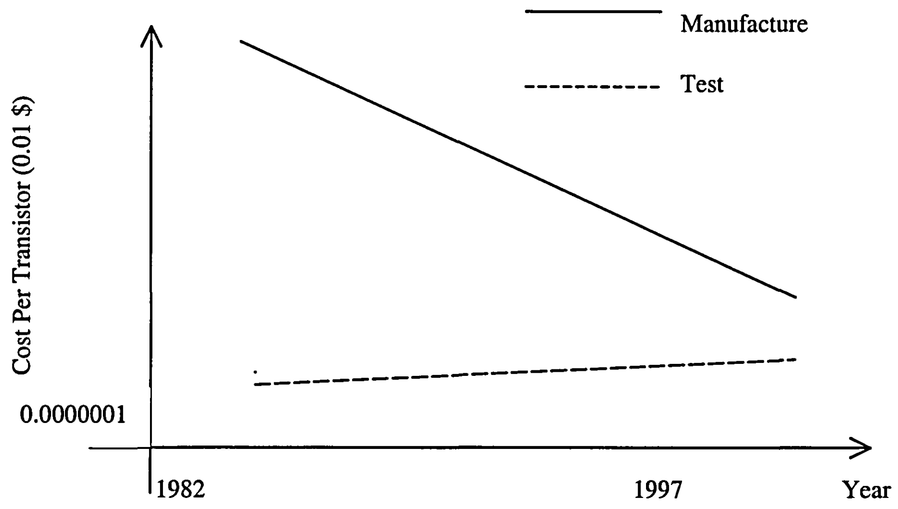


Figure 1.3 *Fabrication and cost trends*

Integrated circuits are tested using *automatic test equipment* (ATE), also referred to as *testers*. It is the cost of these testers, according to Thompson [5], that represents a large proportion of the overall cost of test. He further states that Intel tests over 50 million microprocessors a year using 300 testers that consume 7.5 megawatts of electricity, enough to power a small town. Testers are very large, complex devices and to give the reader and idea of their size, one from a leading manufacturer of test equipment, Advantest Corporation, is shown in Figure 1.5

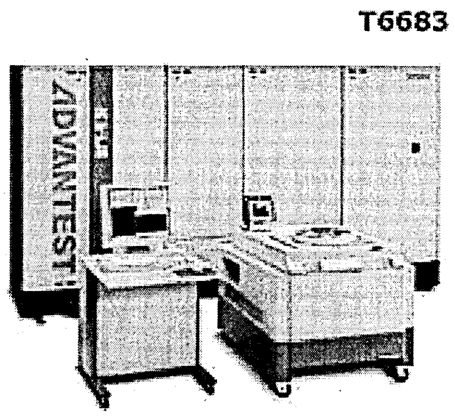


Figure 1.5 *Automatic Test Equipment, model T6683, Advantest Corporation.*

The purpose of test is, of course, to determine whether a circuit is defect-free and functions as intended. Under fault-free conditions, a given set of inputs to a circuit produces a corresponding set of outputs. In many cases, defects in the circuit will result in a deviation from the expected outputs. It is the detection of these deviations that is the central objective of post-fabrication digital testing. A circuit is tested by applying a given set of inputs, known as a *test vector*, and observing the output. For a given test vector, the fault-free output will be known and if the output deviates from this, a fault ewsawill have been detected in the circuit. The generation of these test vectors is known *as test pattern generation*. Test patterns are generated with a given fault (or faults) in mind and by generating a set of test vectors, known as a *test set*, a circuit can be tested for a given percentage of possible faults, known as the *fault coverage*. Once a test set has been generated, it is applied to an integrated circuit by *automatic test equipment* (ATE), which essentially observes the output in response to an input vector to determine whether the behaviour is as expected.

The task of generating test vectors for a circuit is one of the final stages of the overall circuit design life-cycle. Once a circuit design is almost complete, using the description of a circuit and a *fault model*, one is able to generate test vectors. Circuits are often designed using high-level description languages that, much like high-level software language such as C, C++, describe the functionality of a circuit in both machine and human readable form. One of the most widely used languages is VHDL (Very high speed integrated circuit **H**ardware **D**escription **L**anguage) [6] and became an IEEE standard in 1987. Languages such as these enable designers to design and model a circuit. A fault model is an abstraction of physical faults in a circuit and enables engineers to generate tests for these faults (fault models will be discussed later). A description of a circuit and a given fault model are the main inputs to a test pattern generation (ATPG) algorithm. Once a test set has been generated, they are applied to the manufactured circuits by testers to determine whether they function correctly. A high-level depiction of this process is given in Figure 1.4.

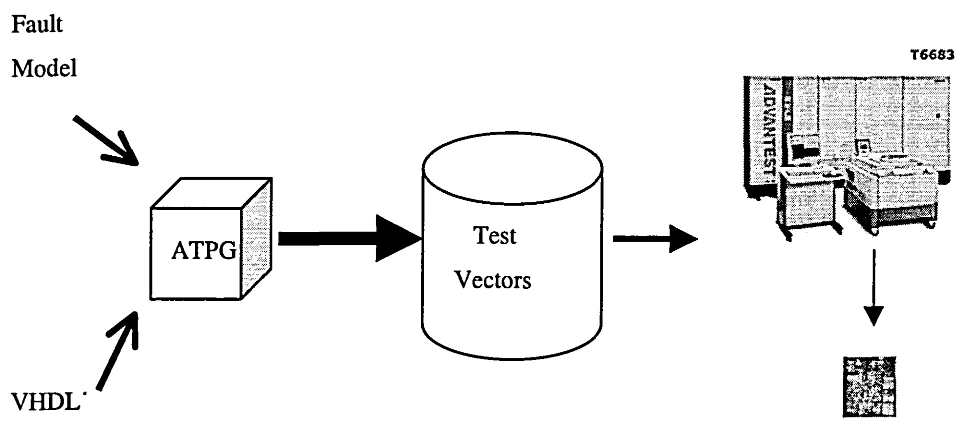


Figure 1.4 *High-level description of test pattern generation and test set application for integrated circuits.*

As stated above, one of the roles of ATE is to apply test vectors to an integrated circuit and compare the output vectors with the expected outputs for a fault free-circuit. This test may only take a fraction of a second, but when faced with testing many millions of ICs a year, this is a huge cost burden for IC manufacturers. ATE equipment is itself getting quicker, but as ICs increase in complexity and gate density, so do the number of possible faults. This in turn often implies that a larger test set has to be applied by the test equipment to achieve the same level of fault coverage and for a given tester, this inevitably translates to greater test set application time. Again, the test community is faced with the challenges of Moore's Law. Higher test set application times often translate into the need for more test equipment and therefore higher test costs. So, the need to reduce this test application time is critical for IC manufacturers in their continual quest for cost control.

Digital testing is a very large and diverse subject area, encompassing many disciplines. This thesis will describe the original work conducted by the author in three key areas of test; test pattern generation, test set minimisation and testability analysis.

Test Pattern Generation and Test Set Minimisation

Test pattern generation [7, 8] and test set minimisation [9] are two of the most important areas within the field of digital testing. Of course the generation of test vectors is necessary in order to actually test a circuit but the generation of high quality test vectors can contribute to lowering the cost of test. Small, optimised test sets, containing high-quality test vectors that achieve high fault coverage will obviously take less time to apply to a circuit than ones that contain more test vectors. Given the complexity of integrated circuits, the process of actually generating the test patterns, is extremely challenging. The author will present a new technique for generating test patterns for combinational circuits that combines the Boolean difference [10] and cubical calculus [11]. The technique applies to multiple output, combinational circuits using the single-stuck-at fault model. Both the Boolean difference and the cubical calculus have been in existence for a number of decades but the Boolean difference technique has been overlooked within test pattern generation because of its cumbersome, algebraic nature. Cubical calculus is shown to overcome this problem and provides a very competent solution to this problem. Chapter Two will introduce cubical calculus and the Boolean difference and will provide a rigorous discussion on this new test pattern generation technique.

The increasing importance of minimised test sets in lowering test application time has already been discussed above. With this objective in mind, the author has successfully applied an evolutionary algorithm [12] to obtain minimised test sets. Chapter Three presents a detailed survey and analysis of a particular class of evolutionary algorithm known as a *genetic algorithm*. Once the reader has obtained an understanding of this optimisation method, Chapter Four goes on to describe the general problem domain of test set minimisation and the application of a genetic algorithm to solve the problem. The author developed genetic algorithm software to solve this problem for real-world test sets generating by a research group at Tallinn Technical University, Estonia. The data provided by this group and the software written by the author is described in Appendix A.

Testability Analysis

Testability analysis provides a means of determining how difficult a circuit would be to test before it is actually manufactured. By performing this analysis during the design stage, circuit designers are able to catch features in a circuit that would make certain faults difficult (or impossible) to detect, resulting in design modifications at an early (and less expensive) stage in the life-cycle of a circuit. Following the work on test pattern generation using cubical calculus and the Boolean difference, a new technique for measuring testability was devised. It was soon realised that much of the core computations in the test pattern generation algorithm could be applied to calculate measures such as *controllability* and *observability* [7]. This work is presented in the final part of Chapter Two and will be shown to make novel use of the Boolean difference and cubical calculus.

The remainder of this chapter will introduce some basic concepts of digital testing and more specifically, test pattern generation, as this area forms the nucleus of the work in this thesis. Once the terminology of the field is introduced, some of the basic ideas behind test pattern generation will be described, some of which have already been mentioned above. Once these preliminary topics have been covered, the final section in this chapter will introduce three of the most important test pattern generation algorithms that form the basis of many of the commercially available ATPG tools in use today.

1.2 Basic Terminology

As in most subject areas, there is much jargon and terminology in digital test. Consider the combinational, digital circuit in Figure 1.5. This circuit contains five logic gates; three AND gates, labelled G1 to G3, and two OR gates, labelled G4 to G5. The connections to and from the gates are known as *circuit lines*, *line*s or *nodes*. There are 5 primary inputs, PI1 - PI5 to the circuit and two primary outputs, PO1 and PO2. The *primary inputs/outputs* to the circuit provide a means of connecting the circuit to the external environment. If the circuit were designed as an integrated circuit (IC), it would be contained in a plastic case and the primary inputs/outputs would be the pins of the IC, providing direct access to them.

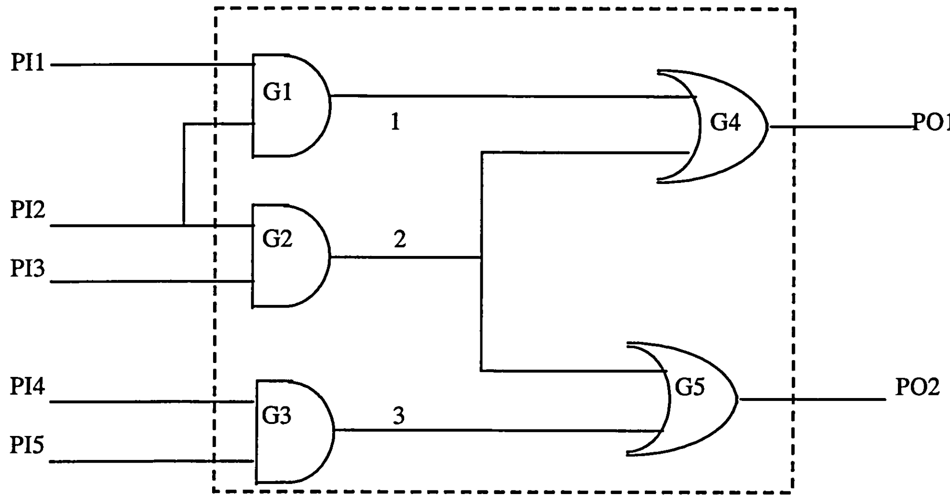


Figure 1.5 *Combinational, digital circuit. The hashed line around the circuit illustrates the casing or packaging of an IC.*

This circuit also contains three *internal lines*, labelled 1, 2 and 3 and are the outputs of gates G1, G2 and G3 respectively. There is no direct, external access to these lines as they are completely enclosed within the IC packaging. In some complex IC designs it is deemed necessary to provide external access to some internal lines, to aid the test process, but they are still known as internal lines. A feature present in many digital designs is known as *fan-out* and is when a circuit line is routed into two or more gates. Line 2 is an example of fan-out. The primary input PI2 is another example of fan-out but differs slightly to the fan-out of line 2. In this case one path of the signal PI2 proceeds through gates G2 and G4 and another path is through G1 and G4. The original signal at PI2 recombines at the output of G4, at primary output PO1. This phenomenon of recombination is known as *reconvergent fan-out* and, as will be explained later holds special significance from a digital testing perspective.

Testing a digital circuit such as that given above involves applying sets of values to the primary inputs and comparing the corresponding outputs with the expected behaviour of the circuit. Each set of input values applied to the circuit is known as a *test vector* or *test pattern* and is often referred to as just a *test*. For a given test vector the corresponding, expected output is known as the *fault-free output*. In the above consider applying the values PI1=0, PI2=0, PI3=1, PI4=0, PI5=1. This set of input values produce fault free values at outputs PO1 and PO2 of 0 and 0 respectively. These inputs and the corresponding fault-free outputs constitute the test vector 00101|00. The values 00101 to the left of the vertical line correspond to the inputs values at PI1, PI2 and so on and the values to the right correspond to the corresponding output values, PO1 and PO2.

1.3 Fault Models

Circuit defects due to the manufacturing process for example, manifest themselves electrically as faults. In order to compile tests for faults one must establish a fault model that defines the relationship between a defect and a fault. A number of fault models exist and the popular ones will now be discussed.

1.3.1 Stuck-at Faults

One of the most popular fault models is the stuck-at fault model. In this model a line in a circuit is either permanently set or stuck-at-1 or stuck-at-0. Regardless of the logic value that should be present at the line under normal working conditions, a defect in the circuit has resulted in the line being stuck-at a given logic value. The abbreviations s-a-1 and s-a-0 denote stuck-at-1 and stuck-at-0 respectively. Consider the two input OR gate given in Figure 1.6. Input 1 is stuck-at-0 and when the test vector 10|1 is applied to this gate, the actual output is 0 due the fault. In what conditions would a manufacturing defect result in a stuck-at fault?

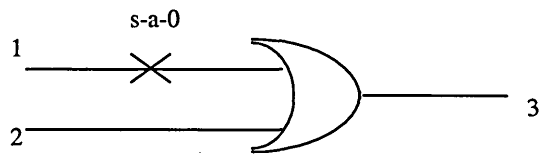


Figure 1.6 *Two input OR gate.*

Figure 1.7 shows the CMOS (Complementary Metal-Oxide Semiconductor) implementation of a two input NAND gate [8]. The supply line (power line) V_{dd} is equivalent to logic 1 and the ground line is equivalent to logic 0. The symbols *T1* to *T4* represent transistors.

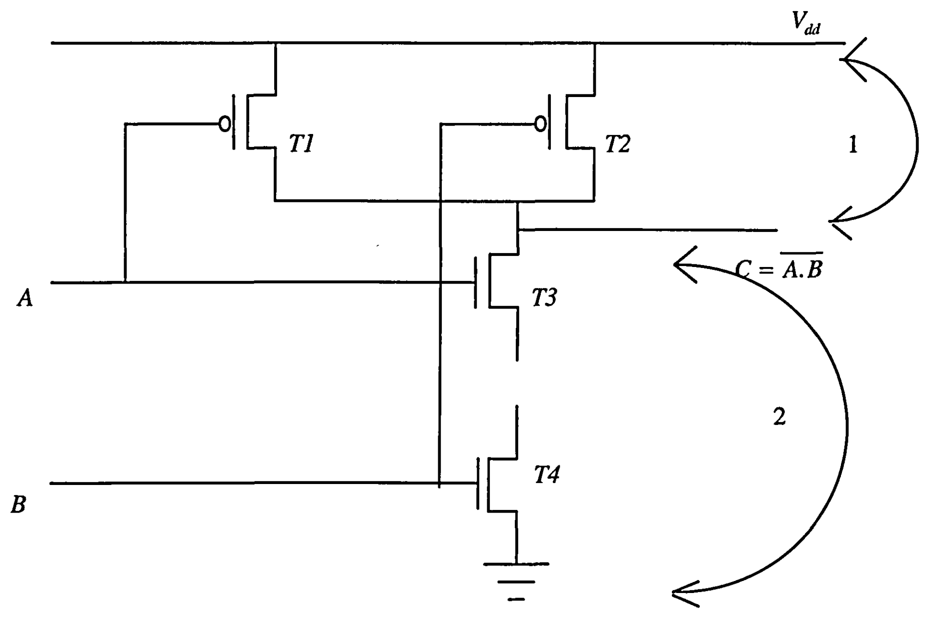


Figure 1.7 *CMOS implementation of a two input NAND gate.*

The numbers 1 and 2 signify two distinct defects in the circuit. Defect 1 is that the output of the gate has been short circuited with the power rail. This defect will result in the fault *C* s-a-1. Defect 2 on the other hand is a short circuit between *C* and the earth rail, resulting in the fault *C* s-a-0. These short circuit defects are common due to today's manufacturing processes and can be conveniently modelled using the stuck-at fault model.

There are two different stuck-at models. A single stuck-at fault model in which it is assumed that only a single stuck-at fault is present in any given circuit and the multiple stuck-at model [8] which assumes multiple faults in a circuit. The model adopted in the present work is the single stuck-at model and this discussion will therefore be limited to this variant.

1.3.2 Bridging Faults

As integrated circuits become more densely packed with transistors, the probability of short circuits between circuit lines increases. These short circuits produce permanent faults known as *bridging faults* that cannot be modelled as stuck-at faults. Three main types of bridging fault exists. The first is an *input bridging fault* where the primary inputs if a circuit are shorted together. A *feedback bridging fault* occurs if there is a short circuit between the primary output(s) and inputs(s) of a circuit. A *non-feedback bridging fault* is a short circuit that does not fall into one of the two aforementioned categories. Figure 1.8 illustrates the different type of faults.

It must be noted that in general bridging faults are layout dependent and only occur between adjacent circuit lines. They differ from short circuits at the transistor level as described in section 1.3.1.

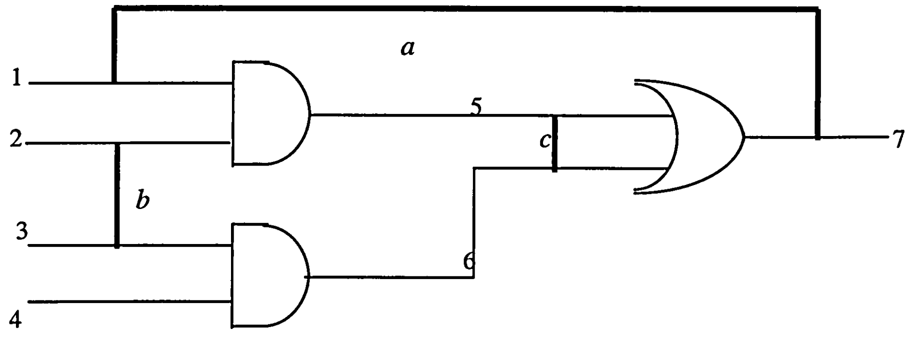


Figure 1.8 *Combinational circuit with an input bridging fault between inputs 2 and 3, a feedback bridging fault between output 7 and input 1 and a non feedback bridging fault between lines 5 and 6.*

Bridging faults cannot be tested using the stuck-at fault model since the bridged lines are able to assume both logic levels. If under fault free conditions two bridged lines assume the same logic value, then the circuit operation is unaffected. If however they assume complementary values a conflict arises. The actual logic values adopted in such a case depends on the technology used to fabricate the circuit as one logic value will dominate the other. In TTL logic (transistor-transistor logic) logic 0 dominates and if two bridged lines need to assume complementary values, both will be set to 0. In CMOS however, such a concept does not always apply and bridging faults have to be analysed at the transistor level [9, 10]. As this is beyond the scope of the present discussion, the reader is directed towards the references.

For s circuit lines, there is a total of $s(s-1)/2$ single bridging faults and obviously many more multiple bridging faults. Since the probability of bridging faults is higher for physically adjacent circuit lines, in general only faults between these lines will be tested for as it would be impractical to try and locate all bridging faults.

1.3.3 DELAY FAULTS

A number of circuit defects, such as bridging faults, result in faults that affect the logical behaviour of a circuit. Smaller defects such as only partial short circuits, although they may result in correct logical behaviour, often result in disrupting the timing of the circuit. In digital circuits, logic signals flow through the circuit in synchronisation with the clock signal. The correct timing of the signals through the circuit are imperative for the circuit to behave in the desired manner. Any delays in the transition of a signal from logic 0 to 1 or vice versa may disrupt the circuit. It is these transitional delays that are known as *delay faults.* There are two main types of delay faults, gate delay faults [15] and path delay faults [16]. The main difference between the two models is that the gate delay model can only cope with delays due to single, isolated defects whereas the path delay model can deal with the affects of distributed delays due to a number of defects. Delay faults cannot be tested using the stuck-at fault model as the fault behaviour does not affect the logical operation of a circuit. Other methods for testing delay faults exist and the reader is directed towards the references for further details [8], [17].

1.4 The Basics of Test Pattern Generation for Combinational Logic Circuits.

In general, electronic components and in particular integrated circuits, are very reliable devices. However due to imperfections in the manufacturing process that produce ICs, such as the presence of dust particles in the fabrication plant, faults will occur. Once the IC design has been finalised, the design engineer compiles a test set that may be applied to a device after it has been manufactured, to test for any defects.

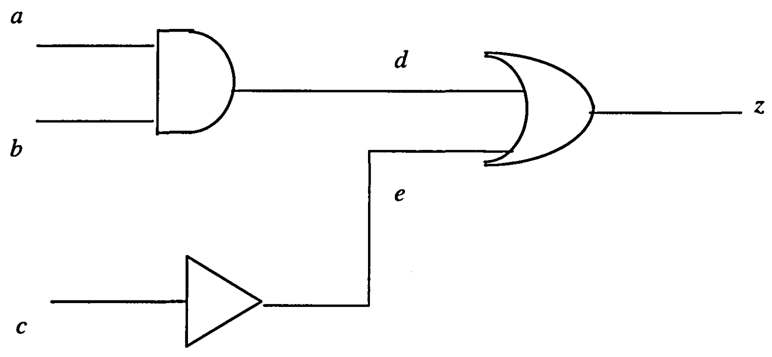
For a non-redundant (meaning the function realised by the circuit under fault-free and fault conditions are different) combinational circuit containing n-inputs all faults may be tested by applying all 2^n possible test patterns. This process of *exhaustive testing,* may be realised for small circuits but is impractical for circuits of say, 30 or more inputs. Exhaustive test pattern generation is obviously NP-hard since the number of test patterns increase as two to the power of n, the number of primary inputs. For example to exhaustively test a circuit containing 60 inputs there are 2^{60} possible test vectors. If it were possible to apply 10,000 tests per second it would take approximately 3.5 million years to test a single circuit [8]. In practice however, it is unnecessary to apply all possible test vectors since a single test vector can cover a number of faults. The process of *fault simulation* [18] is used to determine which faults are covered by a given test vector. When the fault coverage of the test patterns has been generated it is possible to calculate the fault coverage of the test set. If there are x possible faults in a circuit and f faults can be detected by the test set, the fault coverage f_c is given by,

$$f_c = \frac{f}{x}$$

and is often expressed as a percentage. Many test pattern generation algorithms exist and will be discussed later in this chapter. The majority of them assume the single stuck-at fault model and that the circuit is non-redundant. In the following discussion, the single stuck-at fault model for non-redundant circuits will be adopted. At the simplest level, the process of testing a circuit consists of applying successive sets of values to the primary inputs, and of observing the resulting values appearing at the primary outputs. In order to assess the test, the outputs from the tests are compared to the fault-free outputs.

1.4.1 The Sensitive Path Concept

The first task of all test procedures is to compile a fault list, consisting of all the possible faults that can occur. As we are adopting the single stuck-at-fault model, it is assumed that these are the only faults that can occur. A test will then be written which will detect each fault in the fault list. Although each test may be written with the intention of covering one fault, it will invariably turn out in practice that it covers other faults in the list. To illustrate the basic principles of test, the circuit in Figure 1.9 (a) will be used as an example.



(a)

Fault	Test Pattern	Fault Coverage
a/0	111/1	*a*/0, *z*/0, *b*/0
a/1	011/0	*a*/1, *z*/1
b/1	101/0	*b*/1, *z*/1
etc.	etc.	etc.

(b)

Figure 1.9 (**a**) *Circuit realising the Boolean function* $z = a\,b\; + \overline{c}$ (**b**) *Partial fault list, test pattern and fault coverage table for (a)*

The Fault List

The first task when testing any circuit is to compile a fault list. The fault list for the above circuit is (the abbreviation *x*/0 means node *x* is stuck at logic value 0),

a/0 *a*/1 *b*/0 *b*/1 *c*/0 *c*/1 *d*/0 *d*/1 *e*/0 *e*/1 *z*/0 *z*/1

Now we have a fault list, tests must be generated to cover each fault.

Testing isolated logic gates, where one has access to its primary inputs and outputs is a trivial task with the use of logic probes. However, it is often the case with combinational circuits that they contain internal nodes that one cannot directly access because they are housed in I.C. casings. In such situations the *sensitive path* concept is central to the testing procedure. The concept ensures that a fault at a node appears at the output of the circuit. If we assume there is an error at input *a* in Figure 1.9, a path has to be sensitised between it and the output, making the output dependent only on the value of input *a*. This is achieved by manipulating the values of *b*, *c*, *d* and *e*.

So, to sensitise the path between *a* and *z*, we must first transmit the value at input *a* to the output of the AND gate (node d) i.e. enable the gate. This is achieved by setting *b* to 1. Now, to transmit the value of node *d* through to *z*, we must enable the OR gate. This is done by setting node *e* to 0. To set e to 0 we have to work backwards and set node *c* to 1. So by setting, *b*=*c*=1 we have sensitised a path between the input *a* and the output *z*.

Fault Cover and Test Pattern Generation

Once the fault list has been compiled, tests have to be written to cover each fault. There are two requirements when writing a test.

i) Establish the fault free condition. So if we are testing whether a node is stuck at 1, we have to set the node at logic value 0.
ii) Establish a sensitive path between the faulty node and the output.

Let us now write a test pattern for the first fault in the fault list, *a*/0. The sensitive path has already been calculated for input *a*, so to test for *a*/0 the test pattern is *a*=*b*=*c*=1, giving a value of 1 at the output *z*. The test pattern is written as, 111/1. It was mentioned earlier that some test patterns will cover more than one fault and this is one of them. As we are seeking the logic value 1 at the output *z*, we are also in effect testing *z*/0. In a similar manner *b*/0 is covered. The procedure is now repeated for the remaining faults on the fault list. The table given in Figure 1.9(b) is a partial list of faults and their corresponding test vectors along with each test vector's fault coverage.

Once the test patterns have been derived for all of the faults a minimal test set is compiled which consists of the minimal number of tests that will cover all of the faults. It is this test set that is used in the final test procedure. As a further example consider a stuck-at-1 fault on line *f* for the circuit in Figure 1.10. To establish the logic value 0 (complementary to the fault condition) on *f*, logic values *a* = *b* = 1 are required. We must now sensitise a path between line *f* and the primary output *i*. To achieve this the we must set *g* = *h* = 0. The primary output *i* is now dependent on the logic value at *f*. There is one final stage left in the test pattern generation process that was not required in the previous example (due to the simplicity of the circuit) and that is to perform *back-propagation* (also referred to as *back justification*) to set the values of the primary inputs *c*, *d* and *e* required to set the appropriate logic values at *g* and *h*. The required logic values are; *c* = 1 or 0, *d* = 0, *e* = 0, giving the test patterns 11100 | 1 and 11000 | 1.

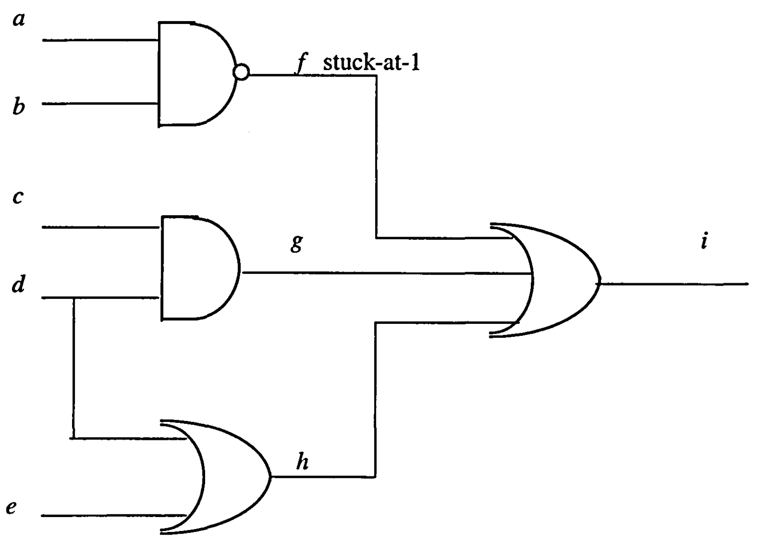


Figure 1.10. *Combinational logic circuit with a stuck-at-1 fault at line f.*

Reconvergent Fan-Out and Undetectable Faults

Although most faults that may exist in a circuit can be detected, there are certain topological features that can make faults undetectable or difficult to test. As previously mentioned, testability analysis is a means to allow circuit designers to identify such features in their design before tape-out (i.e. a design is finalised).

One such feature is known as reconvergent fan-out and is illustrated in Figure 1.11. The signal at input *b* leads to the two NAND gates i.e it 'fans-out'. The outputs of these two gates then lead to the single NAND gate preceding the output i.e they reconverge. This causes problems as faults are often masked and cannot be detected. To illustrate this point, consider the effect of the fault *b*/0. Because of the *fan-out,* the fault will affect the inputs of both gates 1 and 2, and, because of the values assigned to input *a* and *c*, the fault is transmitted through both gates, and hence affects both inputs of gate 3, the place of reconvergence. Since gate 3 is a NAND gate, a change of input from 00 to 11 will produce a change of output from 1 to 0, and hence the fault is transmitted through gate 3. In this case, although neither input change alone would have produced a change at the output (since 01 and 10 both give the output as 00), the two acting together result in fault transmission.

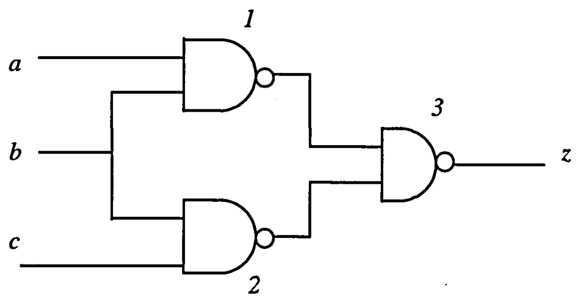


Figure 1.11: $z = \overline{\overline{ab}.\overline{bc}}$

Redundancy in a circuit can also create undetectable faults [8]. Often when one is seeking to sensitise a path, the situation arises where there are conflicting requirements e.g. a single gate needs to be set at the two logic values. In such cases testing difficulties have arisen because of the failure to minimise a circuit in the design stage.

Although they are not part of the simple stuck-at fault model, bridging faults, where two circuit nodes are accidentally connected together, are often included. Each node in a bridging fault can assume either logic value, the result depends on the type of technology used to implement the circuit. For example, in TTL (Transistor-Transistor Logic), the low node dominates and a high node will be pulled low. If the fault free value of each node is the same, then the circuit operation is unaffected. If however they are different, the fault needs to be detected and rectified. Similar techniques to those used for detecting stuck-at faults such as *sensitised path* are used to detect bridging faults. Unfortunately, as with reconvergent fan-out, some bridging faults are impossible to detect. Not all bridging faults can be included in the fault model. For a few thousand nodes there will be perhaps millions of node pairs. So in practice, only bridging faults involving adjacent nodes or tracks are included.

1.4.2 The Boolean Difference Method

The Boolean difference [8], [10], [11] method of test pattern generation relies on Boolean algebraic descriptions of circuit lines. The Boolean difference is essentially an XOR of two closely related Boolean functions. If *g* and *h* are functions then, in the notation of Boolean algebra,

$$g \oplus h \equiv g\bar{h} + \bar{g}h \qquad (1)$$

where $\oplus$ denotes the XOR operation. Consider a Boolean function *F*(X) of a single output circuit, where $X = (x_1, \ldots\ldots, x_n)$ and the variables $x_1, \ldots, x_n$ represent the primary inputs. The Boolean difference of *F*(X) with respect to x_i is defined by

$$\frac{dF(X)}{dx_i} = F(x_1, \ldots, x_i, \ldots, x_n) \oplus F(x_1, \ldots, \bar{x}_i, \ldots, x_n) \tag{2}$$

It must be noted that the left hand side of the above equation is not a derivative, it is simply notation to represent the Boolean difference with respect to the primary input x_i. The most important property of the Boolean difference which forms the basis of its use in test pattern generation is that

$$\frac{dF(X)}{dx_i} = 1 \tag{3}$$

if and only if the output of the function *F*(*X*) is different for normal and erroneous settings of the primary input x_i, in which case a fault at the primary input will be detectable, or *observable*, at the primary output. Conversely if

$$\frac{dF(X)}{dx_i} = 0$$

is true then *F*(*X*) is logically invariant under normal and erroneous settings of x_i and a fault in x_i cannot be detected at the primary output.

The solutions of equation (3) provide the input vectors that propagate a fault on line x_i to the primary output. A test for x_i exists if the other inputs can be chosen so that a change of logic value at x_i produces a change of logic value at the primary output. To actually generate a test vector for x_i stuck-at-0/1, this primary input must first be set to 1/0 and then the fault has to be propagated to the primary output. By performing a logical AND operation between the logic value at the at-fault-line opposite to the fault condition and equation (3) one is able to generate test patterns for stuck-at faults at x_i. Thus the test vectors for x_i stuck-at-0 and stuck-at-1 are given by the solutions of equations (4) and (5) respectively.

$$x_i \cdot \frac{dF(X)}{dx_i} = 1 \tag{4}$$

$$\bar{x}_i \cdot \frac{dF(X)}{dx_i} = 1 \tag{5}$$

The above equations (4) and (5) generate test sets for faults at the primary inputs only. However, the ability to generate tests for faults at the internal lines of a circuit is of greater interest. For an internal circuit node, s_j say, the Boolean difference with respect to s_j becomes

$$\frac{dF(X,s_j)}{ds_j} = F(x_1,\dots,x_n,s_j) \oplus F(x_1,\dots,x_n,\bar{s}_j)$$

where s_j is regarded as a pseudo primary input [6]. The solution of the Boolean equation

$$dF(X,s_j)/ds_j = 1 \tag{6}$$

provides all the input vectors for which a stuck-at fault on s_j alters the primary output.

As in the previous discussion, to generate a test vector for s_j stuck-at-0/1, the node must first be set to 1/0 and the fault propagated to a primary output. An internal node, s_j, can be expressed as a function of the primary inputs, viz. $s_j(X) = s_j(x_1,\dots\dots,x_n)$, and the solution of the Boolean equation

$$s_j(X) = k \tag{7}$$

yields the input vectors that set s_j to k for $k = 0,1$. The input vectors required to propagate a fault at s_j to a primary output are given by the solutions to (6) above. To generate test patterns for a fault on s_j, it is therefore necessary to solve both equations (6) and (7) simultaneously. Hence, for a circuit with n inputs and m outputs, the test sets T_0 and T_1 for s_j stuck-at-0 and stuck-at-1 respectively are given by the solutions of

$$T_0: \qquad s_j(X) \,.\sum_{i=1}^{m}\frac{dF_i(X,s_j)}{ds_j} = 1 \tag{8}$$

$$T_1: \qquad \overline{s_j(X)} \,.\sum_{i=1}^{m}\frac{dF_i(X,s_j)}{ds_j} = 1 \tag{9}$$

where $F_i(X)$ denotes the ith output, for i=1,.....,m.

In order to actually perform the above Boolean calculations, there are a number of Boolean properties which are required. A selection of properties taken from [10] are given below.

Aside

property (1):

$$\frac{d[F(X)+G(X)]}{dx_i} =$$
$$\overline{F}(X)\frac{dG(X)}{dx_i} \oplus \overline{G}(X)\frac{dF(X)}{dx_i} \oplus \frac{dF(X)}{dx_i}.\frac{dG(X)}{dx_i}$$

property (2):

$$\frac{d[F(X)G(X)]}{dx_i} = F(X)\frac{dG(X)}{dx_i} \oplus G(X)\frac{dF(X)}{dx_i} \oplus \frac{dF(X)}{dx_i}.\frac{dG(X)}{dx_i}$$

property (3):

$$\frac{dF(X)}{dx_i} = 1 \qquad F(X)\ \textit{depends only on}\ x_i$$

Example 1.1

Under what conditions will an error in x_1 cause the output to be in error if $f(x) = x_1x_2 + x_3$?

Since,

$$F(x) = x_1x_2 + x_3$$

$$\begin{aligned} \frac{d}{dx_1} &= \overline{x}_3\frac{d(x_1x_2)}{dx_1} && \textit{by property (1) above} \\ &= \overline{x}_3x_2\frac{d(x_1)}{dx_1} && \textit{by property (2)} \\ &= \overline{x}_3x_2 && \textit{by property (3)} \end{aligned}$$

where we have used $\frac{dx_i}{dx_j} = 0$ if x_i and x_j are independent (or if $i \neq j$). The above result means that an error in x_1 will ensure the output is in error if and only if $\overline{x}_3x_2 = 1$, i.e. x_3=0 and x_2=1. Hence if x_1 is stuck-at-1, we need to set $x_1 = 0$ and use $x_2 = 1, x_3 = 0$ to detect the fault.

1.5 Test Pattern Generation Algorithms

Like all algorithms, the singular goal of test pattern generation algorithms is to apply the fundamental understanding of the domain to create efficient, automatic solutions for generating test patterns. The fundamental understanding of the domain encompasses the behaviour of digital circuits, the different types of defects within a circuit and of course the abstractions and faults models that have been created to conceptualise and test a defect. Given this important understanding of the structure of the problem, it is left to the researchers to find efficient solutions by use of effective and robust algorithms coupled with efficient organisation of the underlying data through the use of novel data structures.

Given the size and density of integrated circuits, test pattern generation can be a very complex process and is a very active area of research. Many different approaches have been used to efficiently generate test patterns. These approaches include *random* test pattern generation [19], [20] in which the fault coverage for randomly generated test patterns, using fault simulation, is determined and used to form test sets. Pseudo-exhaustive [21] test pattern generation is a technique that tries to generate test patterns by trying to minimize the time required to exhaustively test a circuit by making use of circuit topology and input/output dependencies. Mathematical techniques such as graph methods [22] and statistical methods such as Monte Carlo [23] have also been used. In addition to algorithms based on some of the aforementioned and more traditional areas of mathematics, newer approaches have also been used in test pattern generation. These include evolutionary algorithms [24], [25] and cellular automata [26].

Many of the above mentioned approaches are underpinned by the basic processes of digital test pattern generation as described earlier in this chapter. Path sensitisation, simulation and the use of fault models are central to many ATPG algorithms regardless of their approach. These basic principles, as well as one or two others, were developed over the past two or three decades and form the basis of the early and now fundamental test pattern generation algorithms. Many of the techniques described in the previous paragraph therefore, also find themselves using these basic principles. The three algorithms described below, *The D-Algorithm, PODEM* and *FAN,* are widely recognized as the *gold standard* within the field of automatic test pattern generation and as such, must be described in any work on test pattern generation.

1.5.1 The D-Algorithm

The D-algorithm [7], [11], published by John Roth in 1960, is by far the most famous test pattern generation algorithm for combinational circuits and the single stuck-at fault model. Given its age, it still remains as the center piece of the field and other algorithms, including PODEM and FAN are essentially extensions of this seminal work. Roth used many important concepts in his work including the use of cubical complex notation, backtracking, error propagation and line justification. He also employed a five-valued composite logic system where,

$$\begin{aligned} \mathbf{X} &= x/x \\ \mathbf{1} &= 1/1 \\ \mathbf{0} &= 0/0 \\ \mathbf{D} &= 1/0 \\ \overline{\mathbf{D}} &= 0/1 \end{aligned}$$

In the above notation, *a/b* implies that *a* is the value of a line under fault-free conditions and *b* is the value of the line under a fault condition. **X** represents 'dont care' or unspecified values. The most interesting notation is the **D** notation, which represents a fault on a line, and is central to the algorithm. To detect a stuck-at-0 error on a line one must first set the line to 1, represented by a **D**. Given the definition of **D** above, this implies the value at the line under fault-free conditions will be 1 and under the fault condition it will be 0. In a similar manner, a stuck-at-1 fault can be represented by a $\overline{\mathbf{D}}$.

In order to generate a test for a particular fault, the fault line is represented by either **D** or $\overline{\mathbf{D}}$ (depending on the fault the test is being generated for) with all other lines initially set to **X**. The next step is to sensitise a path from this line to one or more primary outputs of the circuit by setting the unspecified values from **X** to either 1 or 0. This process is known as the *D-drive*. Then there is a *backward implication* process, starting from the fault line, back to the primary inputs of the circuit. In a similar manner to the D-drive stage, the circuit lines leading to the primary inputs are set to 1 or 0 in order to set the **D** value at the faulty line. If one is able to set the primary inputs of the circuits to either 1 or 0, without conflict, then a test for the fault has been generated.

Before a detailed explanation of the algorithm is given, it is important to examine further, through example, the composite notation and the notion of *singular covers.* A singular cover is a compact representation of a truth table and each row in the cover is known as *a singular cube.* The singular cover for a two input NAND gate is given in Table 1. The truth table in (a) shows that when either (or both) of the inputs is set to 0, the output of a NAND gate is always 1, and when the inputs are both 1, the output is 0. An extended version of this table, using composite logic, is shown in Table 1(b). It illustrates some examples of backward implication. For example, in the *before* table, one row has *c* set to 0, *b* set to 1 and *a* unspecified. Through backward implication, it is obvious that for a NAND gate, *a* must also be set to 1. Another example in this table shows through backward implication, that *b* must be set to 0 if both *a* and *c* are set to 1. The final truth table in (c) shows that **D** and $\overline{\mathbf{D}}$ can imply both backward and forward implication.

A	*b*	*c*
0	x	1
x	0	1
1	1	0

(a)

before

A	*b*	*c*
x	1	0
1	x	1

after

A	*b*	*c*
1	1	0
1	0	1

(b)

a	*B*	*C*
1	**D**	$\overline{D}$
D	1	$\overline{D}$
D	**D**	$\overline{D}$
D	$\overline{D}$	0

(c)

Table 1.11 (a) *Truth table for a NAND gate* **(b)** *truth table illustrating backward implication and (c) forward and backward implication.*

The D-algorithm also employs two key concepts; the *J-frontier* and the *D-frontier*, each being a list of gates that meet given criteria and are used to keep track of forward and backward implications. The J-frontier contains gates for which the output is assigned a logic value that is not implied by its inputs and for which no unique backward implication exists. For example, using the NAND gate as defined above, if *a*=*b*= x and *c* = 1 there are three ways to satisfy this output. That is, either *a* = 1, *b* = 0, or a = 0, b = 1, or *a*=*b*=0. Thus, no unique backward implication exists and these gates are candidates for line justification or backward implication. The D-frontier contains gates whose outputs are **X** and one or more of their inputs are D or $\overline{\mathbf{D}}$. These gates are candidates for D-drive as introduced above. A procedure *of imply-and-check* is executed each time a line is set to a new value of 1 or 0 to ensure no conflicts have occurred. This procedure carries out all forward and backward implications based on the topology of the circuit. To explain the algorithm further, it will be used to generate a test for the stuck-at fault in the circuit given in Figure 1.12

To aid the discussion in the examples, the following notation will be used;

- P_0 will denote the circuit line *P* stuck-at-0
- $V(i)$ will be used to represent a line value $V \in \{0,1\}$ assigned at a particular step *i* of the algorithm

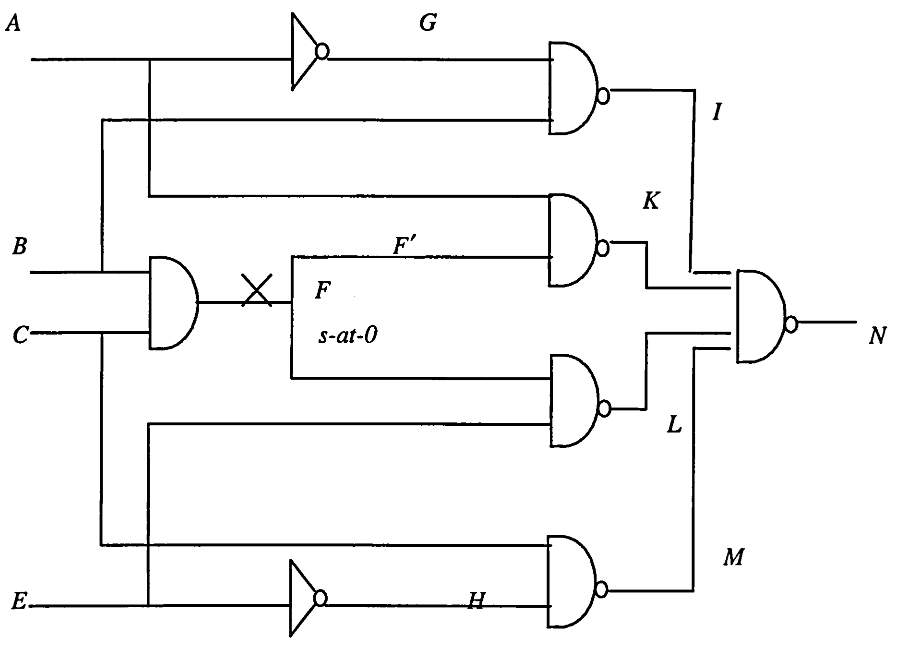


Figure 1.12 *Combinational circuit with a stuck-at-0 fault at line F*

Example 1.2 Using the D-algorithm to generate a test for line F stuck-at-0 in Figure 1.12

The fault will be represented as F_0 .

Step 1. Given the stuck-at-0 fault on *F*, we must set *F* = 1 and perform the imply-and-check based on this setting. The backward implication of this is that we must set $B = C = 1$. This stage produces the D-frontier {*K, L*} and the J-frontier ϕ, the empty set.

Step 2. Select a gate from the D-frontier through which to drive the value at F'. Select gate K.

To get D through gate *K* we need to assign $A = 1(1)$ and we get K = $\overline{D}$. Performing the 'imply-and-check' of these settings we see that,

$$G = 0(1)\,, I = 1(1).$$

- this step produces the D-frontier { *L, N*} and the J-frontier ϕ.

Step 3. Now need to drive the error $\overline{D}$ from *K* to *N* by selecting a gate from the D-frontier. We select gate *N*.

To drive the value through the primary output, we need to set $L = 1(2)$ and $M = 1(2)$. Carrying out the imply-and-check we see we require $H = 0(2)$ and $I = 1(2)$ which implies $L = \overline{D}$ (2) which is a conflict as L has already been set to 1 ! Given this conflict, the D-algorithm now has to perform its back-tracking step. That is, to unset all the values set in the current step (i.e. step 2) and select a different gate from the D-frontier $\{L, N\}$.

Step 4. Select gate L.

To drive the value through L we need to assign $E = 1$. Performing the imply-and-check, we see that,

$$E = 1(2),\ H = 0(2),\ M = 1(2)$$

- this step produces the D-frontier {N} and J-frontier ϕ.

Step 5. Drive $\overline{D}$ through N. This implies $I = 1(3)$.Performing the imply-and-check, we see that,

$$A = 1(3),\ G = 0(3),\ N = \mathbf{D}$$

At this point we see that no conflicts exist, the D and J-frontiers are empty which in turn imply that all primary inputs have been set and that the fault has been driven through to the primary output. So a test vector for F stuck-at-0 is,

$$A = B = C = E = 1$$

As can be seen from the above example, the D-algorithm traverses the circuit, continuously driving faults through gates and performing the imply-and-check procedure to ensure no conflicts have occurred or been implied by the D-drive process. . If conflicts have occurred, an attempt is made to resolve them through back-tracking, which is just a systematic way of undoing the previous D-drive step and selecting another gate from the D-frontier and attempting the drive process through this new gate. For a given stage in the algorithm, if all gates in the D-frontier result in conflicts then no test exists for that fault. The above example is merely a description of how a test can be generated for one particular fault in one particular circuit. It gives some hints to the algorithm but a formal description of the algorithm is given is Figure 1.13.

1.5.2 PODEM – Path Oriented DEcision Making

The PODEM algorithm, conceived by Goel [27] and published in 1981, is based largely on the D-Algorithm (uses the same notation) and is also a circuit based, fault-oriented test pattern generation algorithm. Goel viewed test pattern generation as a finite space search problem and he staged the problem, "as a search of an *n*-input combinational 0-1 state space of primary input patterns of an *n*-input combinational logic circuit". PODEM is *an implicit enumeration* algorithm in which all primary input patterns are exhaustively selected and then determined to see whether they are tests for stuck-at faults in a circuit. The goal of PODEM was to reduce the heavy computational load of the D-Algorithm and achieved it by re-staging the problem in terms of a finite search space.

The D-Algorithm considers every node in a circuit to be part of the search space when trying to locate a test vector for a particular fault. Goel reduced the search space by confining it to include only the primary inputs since all other nodes may be expressed as functions of these.

Suppose we have a set of primary inputs that have been assigned either logic 1 or 0 and we set another primary input *p*, to logic value 1. As we propagate these primary inputs settings through to the primary output(s), much like the D-drive, and we encounter a conflict, we would only have to set the primary input *p* to 0 and see whether the conflict has been resolved. If this complimentary value also results in a conflict, this input is removed as a candidate for a test pattern, thus reducing the search space.

```
D-alg
begin
if implyandcheck() = FAILURE then return FAILURE
if(error not at Primary Output (PO) then
  begin
      if D-frontier = $\phi$ then return FAILURE
      repeat
         begin
              select an untried gate (G) from the D-frontier
              $c$ = controlling value at G
              assign $\bar{c}$ to every input of G with value x
              if D-alg() = SUCCESS then return SUCCESS
         end
      until all gates from D-frontier have been tried
      return FAILURE
  end
/* Error has been propagated to a primary output */
if J-frontier = $\phi$ then return SUCCESS
select a gate (G) from the J-frontier
$c$ = controlling value at G
repeat
   begin
       select an input (j) of (G) with value x
       assign c to j
       if D-alg() = SUCCESS then return SUCCESS
       assign $\bar{c}$ to j  /*reverse decision*/
   end
   until all inputs of G are specified
   return FAILURE
end
```

Figure 1.13 *High-level flow diagram of The D-Algorithm*

The back-tracking process in the D-Algorithm is very costly and PODEM reduces the amount of back-tracking that needs to be performed. Undoing previous decisions and performing the *imply-and-check* process with another gate from the D-frontier can be very costly in itself and in the worst case, all possible choices will have to be tried to resolve a conflict. PODEM attempts to resolve conflicts by resetting primary inputs only and then performing the D-drive process. It is this reduction in search space that gives PODEM the performance advantages over the D-Algorithm which will continually search all nodes in a circuit. In PODEM, as soon as a conflict is encountered, only a subset of the primary inputs need to be reset before the D-drive process is started once more.

When attempting to generate a test, PODEM begins by assigning all primary inputs the value **X**. It then aims to achieve what is known as an *initial objective*, which is to set the at-fault node to the opposite value to the fault-condition. The next stage of the algorithm is the *backtrace*, and this stage aims to obtain primary input assignments given the initial objective. It must then determine whether these primary input assignments have resulted in a conflict with the initial objective through the process of implication (PODEM uses circuit simulation to do this). If no conflicts have been generated, PODEM then selects another primary input and assigns a value to it and performs the simulation again to ensure no conflicts have occurred. If a conflict has occurred through this new setting, this primary input is then set to the complement of the initial setting to see whether this too causes a conflict (again through simulation). If a conflict occurs once more, then this primary input is removed from the search space and is no longer considered in the test generation process. If however, no conflict has occurred with either settings, the process of assigning another primary input repeats until a test has been generated or a conflict occurs or there is no path to propagate the fault to a primary output. The actual propagation to a primary output is performed in a similar manner as the D-Algorithm by searching for a path from the current D-frontier to one or more of the primary outputs. As an illustration of the algorithm, an example from [8] will now be discussed that uses PODEM to generate a test for a stuck-at-0 fault in the circuit in Figure 1.14.

Example 1.3. The use of PODEM to generate a test for line 5 stuck-at-0 for the circuit in Figure 1.14

In the following discussion, the notion of a *net* will be used to describe the circuit topology. A net is essentially a circuit line that either feeds the input of a gate or leads from the output of a gate.

The initial objective is to set the output of gate *A* to 1. Then it is necessary to backtrace to one or more of the primary inputs. By backtracing, it can be seen that the input x_1 has to be set to 0 (x_2 could also have been selected). This input feeds net 1 so we set net 1 to logic 0.

1	2	3	4	5	6	7	8	9	10	11	12
0	X	X	X	X	X	X	X	X	X	X	X

Because 0**XXX** is not a test for the fault (determined through simulation), a second iteration of the algorithm is performed and the primary input x_2 is also set to 0 in a similar manner and results in **D** as the output of gate A and hence the value at net 5.

1	2	3	4	5	6	7	8	9	10	11	12
0	0	X	X	D	X	X	X	X	X	X	X

Since net 5 is now specified, PODEM will now try to find a gate with D as its input and X as its output towards the primary outputs, in a similar manner to the D-drive of the D-Algorithm. Gates G and H satisfy this condition. Selecting gate G and the subsequent initial condition results in the primary input x_3 being set to 0:

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	X	D	1	X	0	X	$\overline{D}$	X	X

$x_1\ x_2\ x_3\ x_4$ = 000X is still not a test, so the PODEM must proceed. Given gate J has $\overline{D}$ on its input net 10, and Xs on input nets 9 and 11, the initial objective is now to set net 12 to logic 1 but we need to select net 9 as the next objective. This results in the primary input x_4 being set to 0 as follows:

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	D	1	1	0	0	$\overline{D}$	$\overline{D}$	D

Hence, now all primary inputs have been set and the fault can be propagated to a primary output, we have a test vector, $x_1\ x_2\ x_3\ x_4$ = 0000, for this particular stuck-at-0 fault.

The same test could of course have been found by the D-Algorithm but substantially more trial-and - error would have been required due to the number of propagation paths and other consistency operations [8], [27]. Also, for untestable faults, there is much wasted effort when compared to PODEM. It is these features of PODEM and the search space reduction feature described above that gives it significant performance improvements over the D-Algorithm. In some cases, these improvements are an order of magnitude better in terms of both processor time and memory usage [27]. For a deeper explanation of the algorithm, the readers are directed to Goel's paper and a number of undergraduate texts [7], [11].

1.5.3 FAN - Fanout-oriented Test Generation

In 1983, Fujiwara and Shimono [28] published their research which sought to accelerate the test generation algorithms of the day. Their work resulted in a test generation algorithm they named FAN, which was an efficient extension of PODEM. The efficiency is largely due to the fact that FAN uses the concept of multiple backtrace. That is, when backtracking, it does so across multiple paths as opposed to single-path backtracking performed by PODEM. Multiple backtracing reduces the number of backtracks that have to be performed, hence the reduction in computational load.

The explanation of the FAN algorithm requires the introduction of new terminology. A *bound* line is the output of a gate that is part of a reconvergent fan-out loop. A line that is not bound is considered to be *free*. A *headline* is a free line that drives a gate that is part of a reconvergent fan-out loop. Let us consider the circuit in Figure 1.15 below. Lines *H, I* and J are bound lines; *A, B, C, D, E, F* are free lines and *G, H* and *F* are headlines. Because by definition, headlines are free lines, they are considered as primary inputs and can be set arbitrarily. So, during backtrace, if a headline is reached, it as though a primary input has been reached and the backtrace ceases.

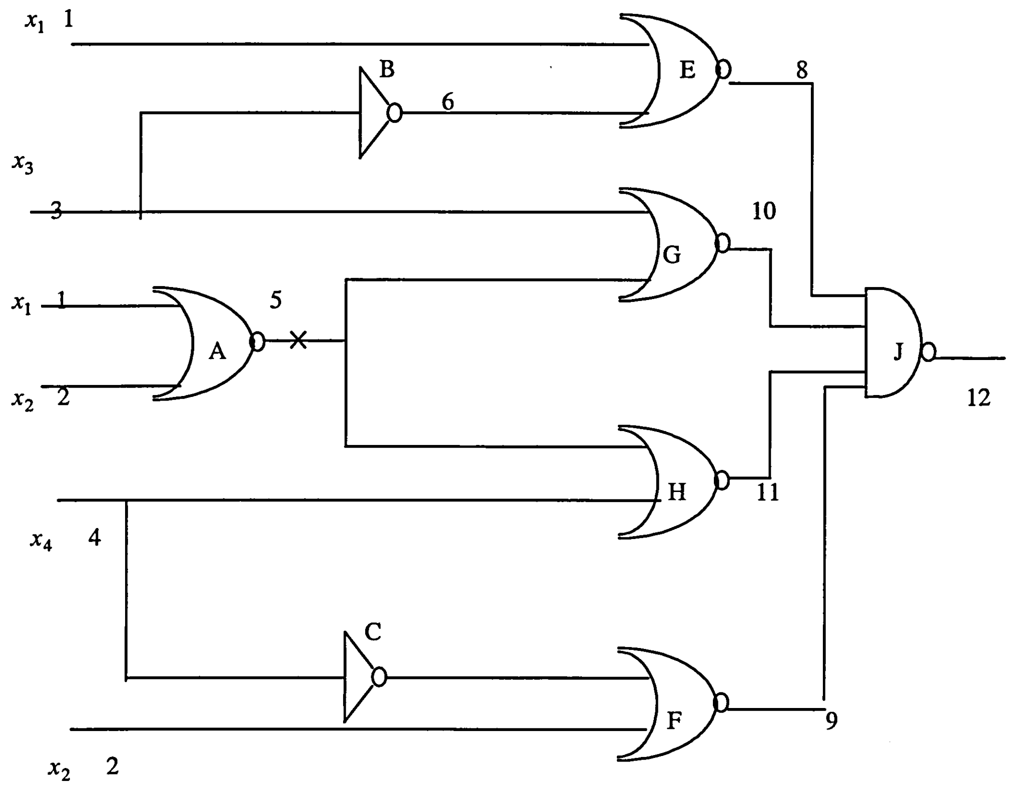


Figure 1.14 *Combinational circuit with stuck-at-0 at line 5*

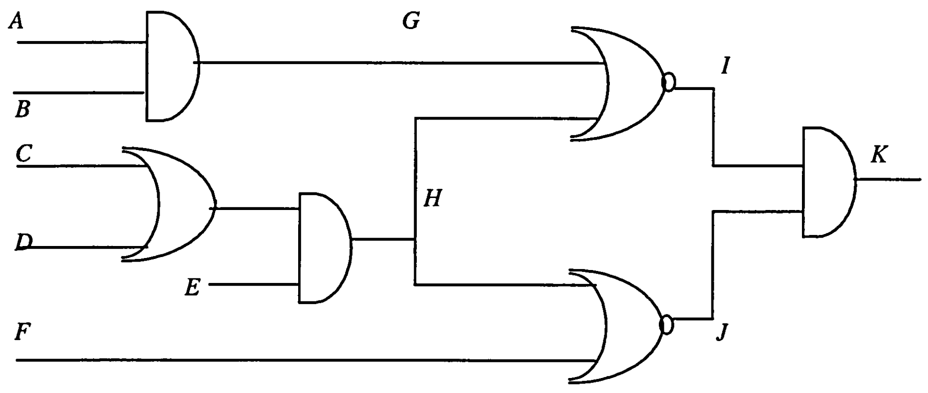


Figure 1.15 *Combinational circuit [8]*

Consider the circuit in Figure 1.16 [8]. Lets assume an initial objective to set line *H* to logic 1, thus testing for *H* stuck-at-0. PODEM would backtrace to *C, D* or *E*. Lets assume the backtrace is done via the path *H-E-C* which sets *E* to 1. This would mean *C* = 0. But, this would result in *F* being set to 1, *G* to 0 and *H* to 0, which fails the initial objective. Now if the backtrace is performed along *H-G-F-C* instead, the initial objective is achieved. Thus, possibly two or more backtraces would be required by PODEM to achieve the initial objective.

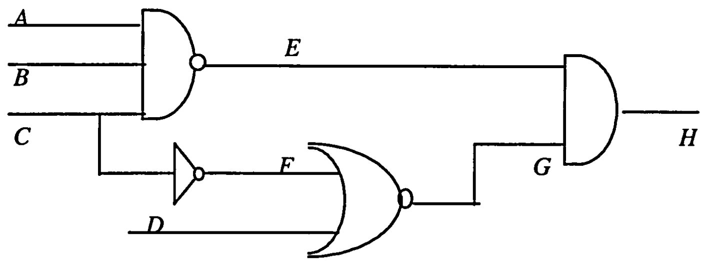


Figure 1.16 *Combinational circuit with stuck-at-0 fault on line H*

FAN however, backtraces along multiple paths to the fan-out point, along say *H-E-C* and *H-G-F-C* ensuring the value 1 would be set at C, while all along, the initial objective is kept in mind.

Reconvergent fan-outs cause many conflicts when trying to backtrace to primary inputs. Only when FAN has traced all paths to a particular fan-out point, will the actual fan-out stem be assigned a value. PODEM on the other hand will backtrace from an initial objective all the way to a primary input, perform simulation and then detect a conflict if one is present. It is this additional effort that is avoided by FAN. By not proceeding with the backtrace until all paths have been traced to it, FAN is able to avoid conflicts before possibly reaching a primary input and without the need for costly simulation, as required by PODEM.

1.5.4 A brief comparison of the D-Algorithm, PODEM and FAN

The three algorithms above were described in chronological order, PODEM and FAN each being refinements and improvements on their predecessor. The D-Algorithm considers all circuit nodes when trying to generate a test and uses much backtracing and forward justification. The algorithm blindly and stubbornly performs these simulations with little regard for what it has already encountered in terms of conflicts and without foresight of what may lay in ahead in for example, the D-frontier. PODEM tries to address these shortcomings by considering the test generation process as a search in a finite space. By only considering only the primary inputs and by discarding those that do not contribute to a test pattern, effectively reducing the search space, PODEM is able to remember and learn about the circuit topology as it performs test pattern generation. The problem with both of these algorithms is the early detection of conflicts and the wasted computational effort in determining this condition. Reconvergent fanouts are often the culprits of these conflicts and FAN tries to eliminate these early on the backtrace process by tracing along multiple paths towards the fanout stem. Only when all paths have been traced to these points and it has been determined that no conflict has arisen, will FAN continue simulating further along the circuit paths. This heuristic itself is largely responsible for the computational efficiencies FAN is able to achieve over PODEM and the D-Algorithm.

Performance comparisons have been made of the three algorithms [29] by collating the results from publications written by the researchers of each algorithm. Unfortunately performance comparisons are difficult as the algorithms have not been given the same problem set to solve. Not only this, but the implementation of these algorithms were in different languages for different hardware platforms (hardware in the 1960s cannot be compared to that of the 1980s!) so it is difficult to make direct and truly meaningful comparisons. But some results are available for PODEM and FAN and and do give some indication of the relative merits of each approach. Goel discussed comparisons between PODEM and a random pattern generator [27]. Testing 50,000 gates, in a time of approximately 1372 minutes on an IBM 370/168 machine, PODEM was able to achieve 89% fault coverage of approximately 90,000 faults, which was a marked improvement over the random approach. The largest example reported by the authors of FAN was a 20,000 gate circuit which it, in a time of 291 minutes on an NEC System-1000, was able to achieve 95% fault coverage of approximately 33,000 faults.

Although the above results compare 'apples to oranges' it seems apparent from the results presented in the respective papers that FAN certainly seems a marked improvement over PODEM in terms of computational performance. For further details of the results the reader is directed towards the references given above.

As was mentioned earlier, test pattern generation is a complex and continuous effort given the pace of development of ICs. The above three algorithms form the basis of many ATPG algorithms and continue to do so. This is evident is some of the more recent work that has emerged; SOCRATES [30] is based on the strategies within FAN, ATOM [31] improves on PODEM and STAR-ATPG [32] and SPIRIT [33] both incorporate and improve upon, a number of the heuristics used in both PODEM and FAN. It would seem that as ICs incrementally follow Moore's Law, ATPG algorithms also improve incrementally to maintain this pace.

1.6 Testability Analysis

The term *testability* refers to the ease and effectiveness with which a circuit can be tested [34], [35]. The process of test generation is a costly one, both financially and in terms of time. It was for this reason that testability measures were developed. Testability depends on the components and the topology of the circuit i.e. its design. The testability of a circuit is normally regarded as a function of two measures, *controllability* and *observability* [36]. Some of the more popular definitions are defined at each node of a circuit as follows:

> **Controllability:** This is the ability to control the fault-free logic value at a node from the primary inputs, so that any logic value can be placed on the node by manipulating the values of the primary inputs.
>
> **Observability:** This is the ability to propagate the value of a node to one of the primary outputs, so that if there is a change of value at a node, there will be a corresponding change at a primary output.

In Section 1.4 above, the sensitised path concept was used to generate test patterns. The basic requirements for writing the tests were to first establish a fault free value at the node (i.e. *control* the value at the node) and then to transmit this value to the primary output (i.e. to *observe* the value at the output). If it is difficult to set a node to a particular logic value (i.e. it is not controllable) or the value cannot be propagated to a primary output (i.e. it is not observable) then the circuit is seen to be difficult to test. It is for this reason that controllability and observability are seen as intuitive measures of testability.

Measuring testability is a precursor to the expensive procedure of test pattern generation. It is best used at the design stage as a guide to the ease with which a circuit can be tested. Difficult nodes or areas of a circuit can be identified, enabling the design to be modified before actual testing takes place.

Testability programs quantify the testability of a circuit in one of two ways; scoring and by algorithmic methods. Scoring methods [34] assign points to certain features within a circuit. So for example, features that decrease testability (e.g. reconvergent fan-out) are given low points and those that enhance testability (e.g. large number of primary outputs) are given high points. The program identifies each feature within a design and keeps a 'score', a high score in this case signifying a design that is easily tested.

Algorithmic methods on the other hand such as CAMELOT [37] and TMEAS [38] produce testability ratings on a node-by-node basis, each node being considered independent. Many of the algorithmic methods are based on the concepts of controllability and observability.

Testability measures are useful as a comparative tool for circuit design. However, two major problems associated with testability are their inability to address reconvergent fan-out and to find faults in a circuit which are impossible to test. The overall testability rating is often calculated by a simple function of controllability and observability such as their product which can often obscure the actual testability of a design. This is also highlighted in Savir's paper [39], "Good controllability and observability do not guarantee good testability", so care has to be taken as to their interpretation. In Chapter Two, a new method for evaluating observability and controllability will be presented based on Boolean difference method and cubical calculus.

1.7 References

[1] MacMillen D., et al, "An industrial view of electronic design automation", IEEE transactions on Computer Aided Design of Integrated Circuits and Systems, Vol 19, no. 12, pp 1428-1446, Dec. 2000

[2] Moore, G., "Cramming more components onto integrated circuits", Electronics, Vol. 38, No. 8, April 1965.

[3] Intel Corporation, Online Museum, "Microprocessor Hall of Fame", www.intel.com.

[4] Moore G., "No exponential is forever ... but we can delay forever", Presented at International Solid State Circuits Conferences (ISSCC), February, 2003.

[5] Thompson K., "Intel and the Myths of Test", IEEE Design and Test of Computers, pp 79 – 81, Spring 1996.

[6] Yalamanchili, S., "VHDL STARTERS GUIDE", Prentice Hall, Sept. 1997.

[7] Abramovici M., et al., "DIGITAL SYSTEMS TESTING AND TESTABLE DESIGN", IEEE Press, 1990.

[8] Lala, P., "DIGITAL CIRCUIT TESTING AND TESTABILITY", Academic Press, 1997.

[9] Ravi S., et. al., "High-level test compaction techniques", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 21., No. 7, pp 827 – 840, July 2002.

[10] Sellers F.F., M.Y Hiss, L.W. Bearnson, "Analysing errors with the Boolean difference", IEEE Transactions on Computers, Vol-C17, No. 7, July 1968.

[11] Roth J.P. "COMPUTER LOGIC, TESTING AND VERIFICATION", Computer Science Press, Maryland USA, 1980.

[12] Holland, J, "ADAPTATION IN NATURAL AND ARTIFICIAL SYSTEMS", The MIT Press, Cambridge, Massachusetts, USA, 1992.

[13] Wadsack R.L., "Fault modelling and logic simulation in CMOS and MOS integrated circuits", Bell Systems Technical Journal, pp1149-1475, May/June 1978.

[14] Visweswaran G.S., et al., "The effects of transistor source-to-gate bridging faults in complex CMOS gates", IEEE Journal of Solid State Circuits, vol. 26, no. 6, pp893-896, June 1991.

[15] Kishida K.F., et al, "A delay test system for high speed logic LSI's", Proceedings of the 23rd Design Automation Conference, pp786-790, July 1986.

[16] Smith G.L., "Model for delay faults based upon path", Proceeding of the International Test Conference, pp342-349, 1985.

[17] Lin C.J., Reddy S.M., "On delay fault testing in logic circuits", Proceedings of the International Conference on CAD, pp 148-151, 1986.

[18] Levendel Y.H. and Menon P.R., "Fault simulation methods - extension and comparison", Bell Systems Technical Journal, pp2235-2258, November 1981.

[19] Savir, J., "Random pattern testability of control and address circuitry of an embedded memory with feed-forward data path connections", Journal of Electronic Testing Theory and Applications, pp 279 – 296, Vol. 15, No.3, Dec. 1999.

[20] Schubert A., Anheier W., "On random pattern testability of cryptographic VLSI cores", Journal of Electronic Testing Theory and Applications, pp 279 – 296, Vol. 16, No.3, Jun. 2000.

[21] Srinivasan, R., "Novel test pattern generators for pseudoexhaustive testing", IEEE Transactions on Computers, Vol. 49., No. 11, Nov. 2000.

[22] Ubar, R., "Test synthesis with alternative graphs", IEEE Design and Test of Computers, pp. 48-57, Spring 1996.

[23] Gupta S., et. al., "Test pattern generation based on arithmetic operations", International Conference on Computer Aided Design, pp117-124, 1994.

[24] Pomeranz, I, Reddy, S., "On improving genetic optimization based test generation", Proc. European Design and Test Conference, pp506-511, 1997.

[25]. Drechsler R., et. al., "EVOLUTIONARY ALGORITHMS IN CIRCUIT DESIGN", Kluwer Academic Publishers, October 2002.

[26] Chiusano S., et. al., "Cellular automata for sequential test pattern generation", 15th IEEE VLSI Test Symposium, Monterey, CA (USA), pp. 60-65, April 1997.

[27] Goel P., "An implicit enumeration algorithm to generate tests for combinational logic circuits", IEEE Transactions Computers, pp 215-222, Mar. 1981.

[28] Fujiwara H., Shimono T., "On the acceleration of test generation algorithms", IEEE Transactions Computers, pp 1137-1144, Dec. 1983.

[29] Kirkland T., Mercer M., "Algorithms for automatic test pattern generation", IEEE Design and Test of Computers, pp43 – 55, June 1988.

[30] Schulz M., "SOCRATES: A highly efficient automatic test pattern generation system", IEEE Transactions on Computer-Aided Design, Vol. 7, No. 1, pp126-137, Jan 1988.

[31] Hamzaoglu, I., Patel, J., "New techniques for deterministic test pattern generation", ", Journal of Electronic Testing Theory and Applications, Vol. 15, No.3, pp 63 – 73, Dec. 1999.

[32] Tsai K., et. al., "Star-TEST: The theory and its applications", IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems, Vol. 19, No. 9, pp1058-1063, Sep. 2000.

[33] Gizdarski E., Fujiwara H., "SPIRIT: A highly robust combinational test generation algorithm", IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems, Vol. 21, No. 12, pp14461457, Dec. 2002.

[34] C.T. Wood, "The quantative measure of testability", Proc. IEEE Autoscan, pp.286-291, 1979.

[35] Chang, S., "TAIR: Testability Analysis by implication reasoning", IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems, Vol. 19, No. 1, pp152-160, Jan. 2000.

[36] L.H. Goldstein, "Controllability/Observability analysis of digital circuits", IEEE Trans. Circuits and Systems, Vol. CAS-26, No. 9, pp685-693, Sept 1979.

[37] R.G. Bennetts et al., "CAMELOT: A computer aided measure for logic testability", Proc. IEE., Vol 128-E, pp177-189, Sept. 1981

[38] J. Grason, "TMEAS a testability measurement program" in Proc. 16th IEEE Design Automation Conference., San Diego, CA, pp156-161, June 1979.

[39] J. Savir, "Good controllability and observability do not guarantee good testability", IEEE Trans. on Computers, Vol. **C-32**, pp1198-1200, Dec 1983.

Chapter 2. Test Pattern Generation for Multiple Output Circuits using Cubical Calculus and the Boolean Difference

2.1 Introduction

The use of the Boolean difference [1-3] to generate test patterns is very widely cited in undergraduate text books and the overall technique has been well understood since the mid 1960's [4][5]. Unfortunately, for such an established technique, the use of the Boolean difference in practical ATPG tools is far from widespread due to the cumbersome nature of its algebraic properties [2]. To solve the Boolean equations, (8), (9), (11), (12) below, laborious manipulations based on these properties are often required, a process which itself can be difficult to automate.

The purpose of this chapter is to introduce original work by the author which couples the Boolean difference and Roth's Cubical Calculus [6] to generate test patterns for multiple output combinational circuits. The chapter opens with rigorous discussions of the Boolean difference and cubical calculus along with some general properties of Boolean functions. Once these mathematical foundations have been laid, an existing test pattern generation algorithm, developed by Xue and Zhang [7], for single output combinational circuits will be described. The Boolean difference, cubical calculus and the concepts introduced in Xue and Zhang's algorithm will then be extended to introduce an original test pattern generation algorithm. Some slight modifications to both Roth's work and Xue and Zhang's will be required to arrive at this new algorithm and set theory together with Boolean algebra will be used along the way to aid the discussion.

By the end of the chapter it is hoped the reader will be cognisant of the merits of coupling Roth's cubical calculus and the Boolean difference for both single and multi-output circuits. It will be shown that the advantages of the Boolean difference, namely that it is a systematic method of solution for test generation equations for all possible test vectors, can be retained whilst avoiding its disadvantages of lengthy and cumbersome manipulations, by using the cubical calculus.

2.2 Test Pattern Generation using Boolean Differences

2.2.1 Single Output Case

The Boolean difference is essentially an XOR of two closely related Boolean functions. If g and h are functions then, in the notation of Boolean algebra,

$$g \oplus h \equiv g\bar{h} + \bar{g}h \tag{1}$$

where $\oplus$ denotes the XOR operation. Consider a Boolean function *F*(X) of a single output circuit, where $X = (x_1, \ldots\ldots, x_n)$ and the variables $x_1, \ldots, x_n$ represent the primary inputs. The Boolean difference of *F*(X) with respect to x_i is defined by

$$\frac{dF(X)}{dx_i} = F(x_1, \ldots, x_i, \ldots, x_n) \oplus F(x_1, \ldots, \bar{x}_i, \ldots, x_n) \tag{2}$$

or equivalently,

$$\frac{dF(X)}{dx_i} = F(x_1, \ldots, 1, \ldots, x_n) \oplus F(x_1, \ldots, 0, \ldots, x_n) \tag{2'}$$

It must be noted that the left hand side of the above equation is not a derivative in the usual sense, it is simply a notation to represent the Boolean difference with respect to the primary input x_i. The most important property of the Boolean difference, which forms the basis of its use in test pattern generation, lies in the interpretation of the equation

$$\frac{dF(X)}{dx_i} = 1 \tag{3}$$

If the above equation is satisfied it means that the output of the function *F(X)* is different for normal and erroneous settings of the primary input x_i. Therefore a fault at the primary input will be detectable, or *observable*, at the primary output. Conversely if

$$\frac{dF(X)}{dx_i} = 0$$

then *F(X)* is logically invariant under normal and erroneous settings of x_i so a fault at x_i cannot be detected at the primary output.

The solutions of equation (3) provide the input vectors that propagate a fault on line x_i to the primary output. Thus, a test for a fault on x_i exists if a change in logic value at x_i produces a change in logic value at the primary output. To actually generate a test vector for x_i stuck-at-0/1, this primary input must first be set to 1/0 and then the fault has to be propagated to the primary output. Mathematically this requirement is equivalent to simultaneously solving equation (3) and the logic value at the at-fault-line (which is opposite to the fault condition). Thus the test vectors for x_i stuck-at-0 and stuck-at-1 are given by the solutions of equations (4) and (5) respectively.

$$x_i \cdot \frac{dF(X)}{dx_i} = 1 \tag{4}$$

$$\bar{x}_i \cdot \frac{dF(X)}{dx_i} = 1 \tag{5}$$

Equations (4) and (5) generate test sets for faults at the primary inputs only. However, the ability to generate tests for faults at the internal lines of a circuit is of greater interest. In the analysis below, an internal line is treated as though it were a primary input and is referred to as a *pseudo primary input* [2]. For an internal circuit line, s_j say, the Boolean difference with respect to s_j becomes

$$\frac{dF(X,s_j)}{ds_j} = F(x_1,\ldots,x_n,s_j) \oplus F(x_1,\ldots,x_n,\bar{s}_j)$$

or equivalently,

$$\frac{dF(X,s_j)}{ds_j} = F(x_1,\ldots,x_n,1) \oplus F(x_1,\ldots,x_n,0)$$

where s_j is regarded as a pseudo primary input. The solution of the Boolean equation

$$dF(X,s_j)/ds_j = 1 \tag{6}$$

provides all the input vectors for which a stuck-at fault on s_j alters the primary output.

As in the previous discussion, to generate a test vector for s_j stuck-at-0/1, the node must first be set to 1/0 and the fault propagated to a primary output. An internal node, s_j, can be expressed as a function of the primary inputs, viz. $s_j(X) = s_j(x_1,\ldots\ldots,x_n)$, and the solution of the Boolean equation

$$s_j(X) = k \tag{7}$$

yields the input vectors that set s_j to k for k=0,1. The input vectors required to propagate a fault at s_j to a primary output are given by the solutions to (6) above. To generate test patterns for a fault on s_j, it is therefore necessary to solve both equations (6) and (7) simultaneously. Hence, the test sets T_0 and T_1 for s_j stuck-at-0 and stuck-at-1 respectively are given by the solutions of

$$T_0: \quad s_j(X) \cdot \frac{dF(X,s_j)}{ds_j} = 1 \tag{8}$$

$$T_1: \qquad \overline{s_j(X)}\,.\frac{dF_i(X,s_j)}{ds_j} = 1 \tag{9}$$

2.2.2 Multiple Output Case

The formulation of the multiple output case is very similar to that for the single output case. It is still necessary to set the at-fault line to either logic 1 or 0. The only difference from the single output case is that the fault condition has to be propagated to at least one of the primary outputs. To ensure this, the Boolean difference equation should be solved for all primary outputs. Therefore, for n inputs and m outputs, solutions of the Boolean equation,

$$\sum_{i=1}^{m}\frac{dF_i(X,s_j)}{ds_j} = 1 \tag{10}$$

provide all the input vectors for which a stuck-at fault on an internal line s_j alters one or more of the primary outputs. Equation (10) only holds true if one or more of the terms in the sum equals 1, since in order for the Boolean equation ,

$$\frac{dF_1(X,s_j)}{ds_j}+\frac{dF_2(X,s_j)}{ds_j}+ \qquad +\frac{dF_m(X,s_j)}{ds_j} = 1$$

to hold, at least one of the terms on the left hand side must equal 1. Thus, for n inputs and m outputs the test sets T_0 and T_1 for s_j stuck-at-0 and stuck-at-1 respectively are given by the solutions of

$$T_0: \qquad s_j(X).\sum_{i=1}^{m}\frac{dF_i(X,s_j)}{ds_j} = 1 \tag{11}$$

$$T_1: \qquad \overline{s_j(X)}\,.\sum_{i=1}^{m}\frac{dF_i(X,s_j)}{ds_j} = 1 \tag{12}$$

since equations (11) and (12) are only satisfied if $s_j(X)=1$ or 0 respectively and a fault at s_j is observable at one or more of the primary outputs. The test set equations for a fault at a primary input are derived similarly, with obvious adjustments.

2.3 Properties of Boolean Functions with Applications to the Boolean Difference

Every Boolean function

$$f(X) = f(x_1,\ldots\ldots,x_n)$$

satisfies the property

$$f(X) = \overline{x_n}f_0(X) + x_n f_1(X) \tag{a}$$

where

$$f_0(X) = f(x_1,......,x_{n-1},0)$$
$$f_1(X) = f(x_1,......,x_{n-1},1)$$

[8]. This may be shown by examining the sum-of-products form of Boolean functions. Every Boolean function $f(X)$ can be written as,

$$f(X) = \sum_{i=0}^{2^n-1} a_i \Pi_i$$

where for each $i = 0 , .. , 2^n - 1$, $a_i = 0$ or 1 and Π_i is a product of *n* terms, each of which is one of the variables $x_1,....,x_n$ or its negation. Comparing

$$f_0(X) = f(x_1,......,x_{n-1},0)$$
$$f_1(X) = f(x_1,......,x_{n-1},1)$$

with

$$f(X) = \overline{x_n} f_0(X) + x_n f_1(X)$$

$f_0(X)$ may be obtained from $f(X)$ by removing all terms in $f(X)$ in which x_n is not negated and deleting $\overline{x}_n$ from the remaining terms. Similarly, $f_1(X)$ may be obtained from $f(X)$ by removing all terms in $f(X)$ in which x_n is negated and deleting x_n from the remaining terms. This may be seen from the following simple example. If

$$\begin{aligned} f(X) &= f(x_1,x_2) \\ &= \overline{x}_1\overline{x}_2 + x_1\overline{x}_2 + x_1x_2 \end{aligned}$$

then

$$f_0(X) = \overline{x}_1 + x_1$$
$$f_1(X) = x_1$$

and

$$\begin{aligned} f(X) &= \overline{x_n} f_0(X) + x_n f_1(X) \\ &= \overline{x}_1\overline{x}_2 + x_1\overline{x}_2 + x_1x_2 \end{aligned}$$

The application of property (a) is important within test pattern generation when a fault is present at an internal circuit line, as was discussed earlier in section 2.2. If for example there is a fault on line '*g*', where $g = g(x_1,...,x_n)$ and it is treated as a pseudo-input to the circuit, then

$$F(X,g) = \overline{g}F(X,0) + gF(X,1)$$

so that by setting $F_0 = F(X,0)$ and $F_1 = F(X,1)$

$$\begin{aligned}
\frac{dF(X,g)}{dg} &= F(X,g) \oplus F(X,\bar{g}) \\
&= F(X,g).\overline{F(X,\bar{g})} + \overline{F(X,g)}.F(X,\bar{g}) \\
&= (\bar{g}F_0 + gF_1)(\overline{gF_0 + \bar{g}F_1}) + (\overline{\bar{g}F_0 + gF_1})(gF_0 + \bar{g}F_1) \\
&= (\bar{g}F_0 + gF_1)(\bar{g} + \bar{F}_0)(g + \bar{F}_1) + (g + \bar{F}_0)(\bar{g} + \bar{F}_1)(gF_0 + \bar{g}F_1) \\
&= \bar{g}F_0\bar{F}_1 + gF_1\bar{F}_0 + gF_0\bar{F}_1 + \bar{g}\bar{F}_0F_1 \\
&= (g + \bar{g})(F_0\bar{F}_1 + F_1\bar{F}_0) \\
&= F_0\bar{F}_1 \oplus F_1\bar{F}_0 \\
&= F_0 \oplus F_1 = F(X,0) \oplus F(X,1)
\end{aligned}$$

This means $\frac{dF(X,g)}{dg}$ is independent of g, so g may be omitted from any calculations involving $\frac{dF(X,g)}{dg}$. So if solving an equation involving the Boolean difference by cubical calculus or any other means, one need only work with the primary inputs of the circuit.

Moving now to the multiple output case, revisiting equation (10) will yield a similar result as for the single output case. Denoting, $\sum_{i=1}^{m} F_i(X,s_j)$ by $H(X,s_j)$

$$\frac{d\left(\sum_{i=1}^{m} F_i(X,s_j)\right)}{ds_j} = \frac{dH(X,s_j)}{ds_j}$$

it follows that,

$$\begin{aligned}
\frac{dH(X,s_j)}{ds_j} &= H(X,0) \oplus H(X,1) \\
&= (H(X,0) + H(X,1)).(\overline{H(X,0)} + \overline{H(X,1)})
\end{aligned} \qquad \text{(b)}$$

It should be noted that since $\frac{d}{ds_j}$ is not linear over the OR operation [1], it is not true that

$$\frac{d\left(\sum_{i=1}^{m} F_i(X,s_j)\right)}{ds_j} = \sum_{i=1}^{m} \frac{d}{ds_j} F_i(X,s_j) \qquad (13)$$

However, it will be shown that equation (13) does hold if a so-called *disjointness condition* is imposed on the output coordinates, and this enables property (b) to be used in the generation of test patterns using cubical calculus for multi-output circuits. This will be discussed fully later in the chapter.

2.4 The Calculus of Cubes

Cubical calculus [6] provides an alternative view of Boolean functions and is attributed to the work of J.P. Roth during the 1960s. It uses the usual binary notation along with the 'don't care' symbol, x, and allows for a very economical and compact representation of a function. Although he remarks that perhaps the first use of it was by Boole [9] and Shannon [8], it was Roth who laid the formal mathematical foundations. He was then able to apply it to areas such as logic minimisation and digital circuit testing. One such application was for NASA's Venus probe [10]. So great was the need to minimise the weight of on-board circuitry that absolute logic minimisation was required and Roth's *Extraction Algorithm* achieved this. More famous however, is his *D-Algorithm* [6] which has achieved widespread usage in the area of digital circuit testing and is essentially a formal specification of the path sensitisation method for generating test vectors.

Before the application of the cubical calculus can be discussed, it is necessary to introduce the foundations of this field. This is done below, largely through formal definitions and examples.

2.4.1 Cubical Definitions and Operations

A cube

A *cube* defines a relationship between the input variables and the output variables of a function. It is written using the notation,

$$a_1 a_2 \ldots a_r | b_1 b_2 \ldots b_s$$

$$\text{or} \qquad a|b$$

where $a_1 \ldots a_r \in \{0,1,\mathrm{x}\}$ are the input variables and $b_1 \ldots b_s \in \{0,1,\mathrm{x}\}$ are the output variables. The 'x' is known as the 'don't care' symbol and is interpreted differently depending on whether it appears on the input or output side of the cube. When a variable on the input side has the value x, it means that this variable can take the value 1 or 0. When an output variable takes the value x, it means that this variable is unspecified. Note, that this definition of x is different to that provided by Roth in [6]. The symbol | (known as 'slash') separates the inputs and the outputs. Variables of a cube are also known as *coordinates* and they can be either *bound* or *unbound.* A *bound* coordinate has the value '1' or '0' and an unbound coordinate takes the value 'x'. It is to be further noted that unbound coordinates are also known as *free* coordinates.

For example, using the above notation,

$$1\mathrm{x}0 \,|\, 1\mathrm{x}$$

is a cube interpreted as; when $a_1 = 1$ and $a_3 = 0$, regardless of the value of a_2, the output $b_1 = 1$ and the output b_2 is unspecified i.e. a don't care state. Given the different interpretation of x depending on whether it appears on the input or output side of the cube, it may be inferred from this cube that whenever $a_1 = a_2 = 1$, $a_3 = 0$, or $a_1 = 1$, $a_2 = a_3 = 0$, then $b_1 = 1$. But since $b_1 = x$, nothing can be inferred about the value of b_2 for either of the possible input sets.

Cover

A *cover* is a set of cubes that unambiguously defines a function. As an example, the truth-table of a two-input (a_1, a_2), single output b_1 AND function is (note that each row constitutes a cube),

a_1	a_2	b_1
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1. *Boolean AND function of two input variables* $(a_{1,}, a_2)$ *and one output* b_1 .

The cubes that represent this function most economically are,

0x|0

x0|0

11|1

Thus, these three cubes define a cover of the AND function. This alternative representation is more compact and economical than the original truth-table. It is to be noted that this notion of economy will appear throughout this discussion and refers not only to a reduced number of cubes but also to a reduced number of bound coordinates.

For Boolean functions there two types of covers, a 1-*cover* and a 0-*cover*. A 1-cover defines a function whose output is equal to 1 and a 0-cover defines a function whose output is 0. For the AND function, the cubes 0x|0 and x0|0 define the 0-cover and the cube 11|1 defines the 1-cover.

Vertex

A *vertex* is a special cube in which all input coordinates are bound (1 or 0) and all but one of the output coordinates are x. This means all input coordinates are defined but only one of the outputs is defined. A vertex therefore, refers to just one output. For example,

101|xx1

is a vertex and refers to the third output.

Contain

A cube $a \mid b$ is said to *contain* vertex $c \mid d$ if $a \mid b$ can be transformed into $c \mid d$ by the appropriate change of free input coordinates into bound and bound output coordinates, except one, into free. For example, if

$$a \mid b = \text{x01} \mid 01 \quad \text{and} \quad c \mid d = 101 \mid \text{x1}$$

then by changing the first input coordinate from an x to a 1 and by changing the first bound output coordinate from a 0 to an x, we obtain the cube $c \mid d$ from $a \mid b$. The cube $a \mid b$ therefore contains all the information of the vertex $c \mid d$. If $a \mid b$ contains all the vertices of a cube $e \mid f$, then $a \mid b$ is said to *contain* $e \mid f$.

Face

If a cube $a \mid b$ contains the cube $c \mid d$ and $a \mid b$ can be transformed into $c \mid d$ by changing just one free input coordinate into a bound one, then $c \mid d$ is said to be a *face* of $a \mid b$. For example, if

$$a \mid b = 10\text{x} \mid 110 \quad \text{and} \quad c \mid d = 100 \mid \text{x}10$$

then $c \mid d$ is a face of $a \mid b$.

An *output face* of a cube is obtained by changing one or more bound output coordinates into free and an *input face* is obtained by changing just one free input coordinate into a bound one. For example, 10x | xx0 is an output face and 100 | 110 is an input face of 10x | 110.

Interface

Before introducing the concept of interface for cubes, the *interface* of single coordinates is defined as follows, where the symbol 'I' denotes interface.

For input coordinates,

$$0 \text{ I } 0 = 0 \text{ I x} = \text{x I } 0 = 0 \qquad \text{x I x} = \text{x}$$

$$1 \text{ I } 1 = 1 \text{ I x} = \text{x I } 1 = 1 \qquad 1 \text{ I } 0 = 0 \text{ I } 1 = q = \text{conflict}$$

As can be seen above, the interface operation finds common input coordinates. As x can be either 1 or 0, a bound coordinate interfaced with a free one results in the bound coordinate.

For output coordinates,

$$0 \text{ I } 0 = 0, \;\; 1 \text{ I } 1 = 1, \;\; 1 \text{ I x} = \text{x I } 1 = 0 \text{ I x} = \text{x I } 0 = \text{x}$$

The interface of two cubes with the same dimensions is the cube formed from the interface of their individual coordinates. For example,

$$(10\text{x} \mid \text{xx}) \text{ I } (1\text{x}1 \mid \text{x}0) = 101 \mid \text{xx}$$

The Consistency of Cubes

A set of cubes which defines a function (a cover) must be *consistent*. This ensures that if different cubes in the set have common input parts, these do not produce different outputs, i.e. identical inputs have the output as defined by the function. Cubes are consistent if a conflict in the output part of their interface implies a disagreement, q, in the input part.

For example, the cubes

x11 | 1x and 00x | x0 have the interface 0q1 | xx and are consistent
x0x | x0 and 100 | 11 have the interface 100 | 1q and are inconsistent
01 | 1 and 11 | 0 have the interface q1 | q and are consistent

The notion of consistency, in more general terms, is to exclude the possibility of common inputs having conflicting outputs, since inconsistent cubes cannot contribute to the cubical representation of the same function. In the second example above the two cubes are inconsistent since it is implied that for the single input 100, the second output takes the values 0 and 1.

An interface having the coordinate 'q' is said to be a *degenerate* cube. For consistent cubes, two cubes are said to be *disjoint* if their interface is degenerate.

Cover and Function

The definition of a cover given earlier can now be made more precise. A set of pairwise consistent, non-degenerate cubes all referring to the same input and output variables is known as a *cover*, denoted by C. A cover defines a function, F, which is defined for each vertex v (recall a vertex refers to a single output), contained in each cube of C, and only those. For example, referring back to the AND function in Table 2.1, a cover is given by,

0x | 0
x0 | 0
11 | 1

The function is defined by this cover for all possible inputs and the interface of any pair of cubes in this cover will show that they are consistent, in accordance with the definition. i.e.

x0 | 0 I 11 | 1 = 1q | q
x0 | 0 I 0x | 0 = 00 | 0
0x | 0 I 11 | 1 = q1 | q

CARE and DON'T CARE Vertices

CARE vertices are those vertices that have an output value of 1 and *DON'T CARE* vertices are those that have an output value of 0. For example, $110 | \mathrm{x}\,1$, $100 | 1\mathrm{x}$ are CARE vertices and $001 | \mathrm{xx}0$, $101 | \mathrm{x}0\mathrm{x}$ are DON'T CARE vertices.

Interface of Covers

The *interface* of two covers *C* and *D*, written $C \text{ I } D = \{c \text{ I } d : c \in C, d \in D\}$. Where possible, $C \text{ I } D$ is reduced to an optimal set of non-degenerate cubes using the CONTAIN operation and/or SHRINK algorithm specified below. If *c* I *d* is disjoint for all $c \in C$ and $d \in D$, then *C* and *D* are said to be disjoint and we write this $C \text{ I } D = \varnothing$, where $\varnothing$ denotes the empty set.

As an example, consider the covers

$$C = \begin{Bmatrix} 1\mathrm{x}0 | x1 \\ \mathrm{xx}1 | \mathrm{xx} \end{Bmatrix}, \; D = \begin{Bmatrix} \mathrm{xx}1 | 0\mathrm{x} \\ 00\mathrm{x} | \mathrm{xx} \\ 1\mathrm{x}1 | 01 \end{Bmatrix}$$

whose interface is given by,

$$C \text{ I } D = \begin{Bmatrix} 1\mathrm{xq} | \mathrm{xx} \\ \mathrm{q}00 | \mathrm{xx} \\ 1\mathrm{xq} | \mathrm{x}1 \\ \mathrm{xx}1 | \mathrm{xx} \\ 001 | \mathrm{xx} \\ 1\mathrm{x}1 | \mathrm{xx} \end{Bmatrix}$$

which after removing degenerate cubes reduces to

$$\mathrm{C\,I\,D} = \{\mathrm{xx}1 | \mathrm{xx}\}$$

But since $\mathrm{xx}1 | \mathrm{xx}$ contains no vertices, we have finally

$$\mathrm{C\,I\,D} = \varnothing$$

Although Roth does not describe how to treat degenerate cubes in the CONTAIN and SHRINK algorithms, it is logical to remove them as they provide no additional information about the cover in addition to the non-degenerate cubes in the cover.

Adjoin

The *adjoin* of two cubes $a|b$ and $c|d$ written $(a|b)$ V $(c|d)$ is any set *S* of cubes which is equivalent to $\{a|b, c|d\}$ in the sense that *S* and $\{a|b, c|d\}$ contain precisely the same input/output information. It is usual to optimise the set *S* so that it contains the minimum number of cubes and a maximum number of free variables.

For example,

$$(1x0x|1) \text{ V } (0x0x|1) = \{1x0x|1, 0x0x|1\} = \{xx0x|1\}$$

The adjoin of two covers *C* and *D* is defined similarly to be any set of cubes which is equivalent to the combined set $\{c \in C\} \cup \{d \in D\}$. Again, where possible an optimal set is chosen.

Coface

A *coface* of a cube is a generalised representation of that cube. A cube *e* is a coface of cube *c* , if *c* is a face of *e*. The coface of a cube will contain more vertices than the original cube (the face), not all of which will necessarily be consistent with the cover.

Let *c* be a cube in the cover *C*. Cube *e* is said to be a *coface* of *c*, with respect to *C*, if *c* is a face of *e* (*e* contains *c*) and *e* itself is consistent with the cubes of *C*. To obtain a coface of a cube , one changes a bound input coordinate to an x or changes a free output coordinate to a bound one.

Consider the cube, 010 | 0 and its coface x10 | 0, obtained by altering the input side. The coface can be seen to be a generalised representation as it contains two vertices, 110 | 0, 010 | 0 i.e. more information than was contained in the original cube. Now consider the cube 010 | 0xx and its coface 010 | 01x , generated from the output side. This again is a more general representation as this coface contains the vertices, 010 | x1x and 010 | 0xx.

Additionally, it is necessary to interface this new cube (the coface) with each other cube in the cover to ensure consistency. For example, the cubes,

$$001|1 \quad 101|1 \quad 000|1 \quad 100|1 \quad 011|0$$

define a cover *C*. Taking *c* = 001 | 1 and changing one of the bound input coordinates to a free, we form the (potential) coface *e* = x01 | 1. For this truly to be a coface, it must be consistent with each of the other cubes of the cover. So, checking the consistency,

$$\begin{aligned} x01|1 \;\text{ I }\; 101|1 &= 101|1 \\ x01|1 \;\text{ I }\; 000|1 &= 00q|1 \\ x01|1 \;\text{ I }\; 100|1 &= 10q|1 \\ x01|1 \;\text{ I }\; 011|0 &= 0q1|q \end{aligned}$$

Hence *e* is a coface of *c* with respect to *C*.

The CONTAIN Operation

Let *C* be a cover. The *CONTAIN* operation (not to be confused with *contain* which refers to a relationship between cubes) deletes from *C*, all cubes that are faces of other cubes of *C*. A coface of a cube is a generalised form of the original cube (which is a face of the coface). So, by deleting faces of a cube in *C*, we are in fact removing specific instances of the more general coface. Since the CONTAIN operation is removing this redundancy, there is no loss of information in the resulting cover. For example, consider a cover *C*, consisting of the following cubes,

x010|0xx   1010|0xx   xx10|xx0   x110|xx0

The result of applying the CONTAIN operation to *C* would result in the removal of the cubes 1010|0xx and x110|xx0 from this cover as they are faces of the cubes x010|0xx and xx10|xx0 respectively.

The #-Product ('sharp' product)

The #-product is a cubical differencing operation. If cube *a* is 'sharped' with cube *b*, *a # b*, the result is a cover containing all vertices of *a* that are not contained in *b*.

The #-product of two cubes $a|b$ and $c|d$ written $(a|b)$ # $(c|d)$ satisfies

(i) if $a|b$ and $c|d$ are disjoint then $(a|b)$ # $(c|d) = a|b$
(ii) if $b = d$, then $(a|b)$ # $(c|d)$ is a set of cubes of the form $e|b$ whose input parts constitute a cover of the inputs contained in *a* but not in *c*.
(iii) if all coordinates in *b* are bound and all coordinates in *d* are free then $(a|b)$ # $(c|d)$ = $a|b$ since $c|d$ contains no information.

Note that in (ii) and (iii) the differencing operation # affects the input coordinates only and is a departure from Roth's definition. The following examples illustrate these rules:

10|x0 # 00|x0 = 10|x0 , since these cubes are disjoint.
x0x|1x # 001|1x ={10x|1x , x00|1x} , on the basis of rule (ii)
x10|1x # 01x|1x =110|1x , on the basis of rule (ii)
x00|10 # 000|xx =100|10 , on the basis of rule (iii)

The sharp product $a\ \#\ C$ of a cube a and a cover $C=\{c_1,c_2,....,c_k\}$ is defined by $a\#C$ = $(((a\#c_1)\#c_2...)\#c_k)$ and the #-product of the covers $C\#D$ is given by, $C\#D = \vee_{i=1}^{k}(c_i\#D)$. As before the #-product operation is expressed as an optimal set of cubes where possible. As an example consider the covers,

$$C_1 = \begin{Bmatrix} \mathrm{x1}x\,|\,1 \\ \mathrm{xx1}\,|\,1 \end{Bmatrix} \text{ and } C_2 = \{11x\,|\,1\ \}$$

$$C_1\#C_2 = \begin{Bmatrix} \mathrm{x1}x\,|\,1\#11\mathrm{x}\,|\,1 \\ \mathrm{xx1}\,|\,1\#11\mathrm{x}\,|\,1 \end{Bmatrix} = \begin{Bmatrix} \mathrm{01x}\,|\,1 \\ \mathrm{0x1}\,|\,1 \\ \mathrm{x01}\,|\,1 \end{Bmatrix}$$

Redundancy

A cube r in a cover E is said to be *redundant* if all of its vertices are contained in other members of E.

Proposition A cube r of the cover E is redundant *iff*,

$$[r\#(E-r)] = \varnothing \text{ (empty)}$$

where $(E-r)$ is the cover which is obtained from E by removing r and $\varnothing$ is the empty set.

To clarify, the term,

$$[r\#(E-r)] = E'$$

is a cover containing the vertices of r that are not contained in the other cubes of E. If it is empty, then all the vertices are contained in other cubes of E, and it follows that r is redundant.

Cubical Complex

Let C be a cover. The *Cubical Complex*, $K(\ C\ )$ is the set of cubes in cover C along with the faces and cofaces (with respect to C) of each said cube.

Prime Cubes and Covers

A cube r, is said to be *prime* with respect to a cover or complex E, if none of the vertices of r are contained in other cubes of E.

2.4.2 Geometrical Visualisation

To reinforce the definitions given thus far, a geometrical visualisation of the ideas is given. Figure 2.1 depicts a 3-dimensional cube which can be used to illustrate a cubical complex and function of three input variables and one output variable. The output part of the function is omitted for clarity. Please note to avoid confusion, the term '3-dimensional cube' will be used to refer to the cube that is drawn in the figure. The word 'cube' alone refers to cubes such as x01 | 1x.

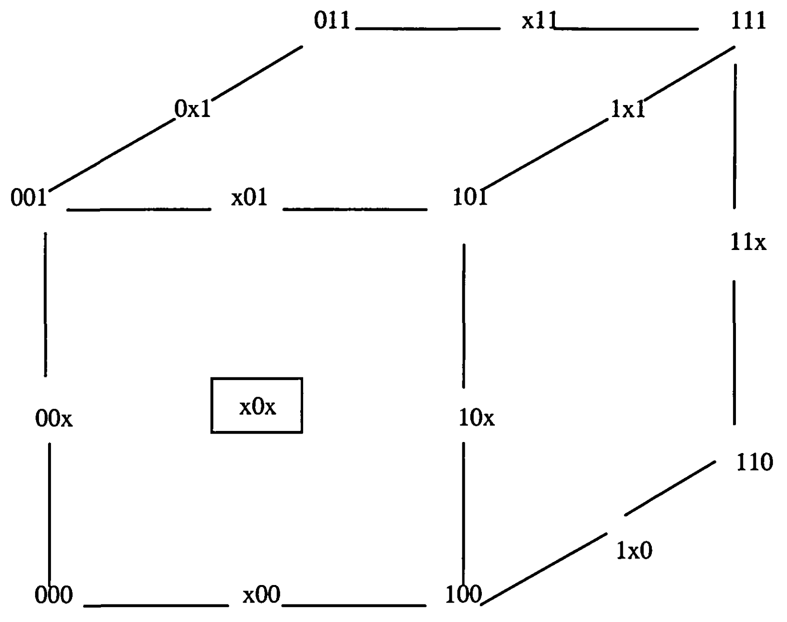


Figure 2.1. *Geometrical visualisation of a cubical complex*

The vertices of the cubes represent the input part of vertices of the function. The vertices are placed such that each vertex differs from its immediate neighbours by only 1 bit (as in Gray coding). This gives rise to certain simplifications and aids the minimisation of cubes and bound coordinates in a cover.

The cube joining a pair of vertices is a coface of each vertex. For example, the cube x00 joining the vertices 000 and 100. In turn, 000 and 100 are both faces of x00. The front panel of the 3-dimensional cube contains a small square containing the cube c = x0x (the others have been omitted for clarity). This is a coface of the cubes x00, 10x, x01 and 00x. They in turn are faces of x0x.

2.4.3 Example to Deduce a Minimum Cover of a Function

The motivation behind finding a minimum cover is to form a more compact representation of a function, with fewer cubes and bound coordinates than were in the original cover. With the use of don't care coordinates we can simplify the cover by using the fact that a single cube with fewer bound coordinates can represent several cubes. This is really the idea behind cofacing, as described earlier.

Let the following cubes define the on-states (i.e. the inputs for which the output is 1) of a three input, one output function.

011|1 010|1 111|1 110|1 101|1

The remaining cubes are either 'Don't Care' (have undefined outputs) or off-states (i.e. inputs for which the output is 0). The above cubes are the *CARE* vertices (inputs that result in a 1 on the output) defining the function and its cubical complex. Figure 2.2 is a geometrical representation of the cover including the 0-cubes (those with 0 free coordinates) and the 1-cubes (those with 1 free coordinate) and a single 2-cube.

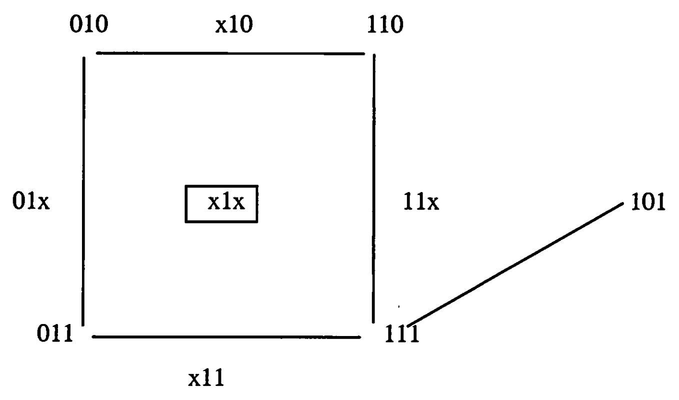


Figure 2.2. *Geometric representation for example cover*

The pair of 0-cubes 011 and 111 define the 1-cube (and coface) x11 which 'covers' the same vertices as this pair but more economically. The other cofaces follow similarly. This representation is more economical since four vertices have been replaced by four co-faces and only two of these cofaces are required to cover all four vertices. Furthermore, the cubes x11, 11x, 01x and x10 define the 2-cube x1x (which is a coface of these cubes). So the four vertices 011, 111, 110 and 010 are 'covered' by the cube x1x , reducing the number of cubes from four to one and the number of bound coordinates from twelve to one.

The remaining vertex, 101 is 'covered' by the 1-cube 1x1 and although the number of cubes remains the same, when 101 is replaced by 1x1, the number of bound coordinates is reduced by one.

Thus, the cover has been minimised from the original five cubes with 15 bound variables to just two cubes,

$$x1x,\ 1x1$$

with three bound variables.

2.4.4 Approximate Optimisation Algorithm: SHRINK

SHRINK is an approximate optimisation/minimisation algorithm by Roth [1]. It is approximate because it does not guarantee to find a minimal cover for a function. The result of SHRINK is a cover M containing all the vertices of a cover C and is composed solely of non-redundant (see earlier definition) prime cubes. The algorithm in the single output case comprises the following steps.

Step 1. Perform CONTAIN on C resulting in C'. This operation removes all faces in C of other cubes contained in C.

Step 2. Select a cube c in C' and find/obtain a coface z of c w.r.t. C'. (If c has no coface w.r.t. C', select another cube in C').

Step 3. Replace c in C' by z to form C''

Step 4. Remove from C'' all cubes contained in z to form C'''.

Step 5. Set $C' := C'''$ and repeat steps 2 -4 until the resulting cover P consists solely of prime cubes (i.e. until no further cofacing is possible.

Step 6. Remove redundant cubes from P to produce a non-redundant prime cover M.

Example 2.1 Note: We are only minimising the cubes that give an output of 1 for this function.

Consider the cover C, with

$$C=\begin{Bmatrix} 1x11|1 & x010|0 \\ xx00|1 & 0011|0 \\ 11xx|1 & \end{Bmatrix}$$

Step 1.1 Remove faces

None to be removed.

$$C' =\begin{Bmatrix} 1x11|1 & x010|0 \\ xx00|1 & 0011|0 \\ 11xx|1 & \end{Bmatrix}$$

Step 1. 2. Choose c =1x11|1 from C'. Its cofaces are xx11|1, 1xx1|1 and 1x1x|1. Now let us check each for consistency with respect to C'. The cofaces xx11|1 and 1x1x|1 are inconsistent with 0011|0 and x010|0 respectively since,

$$(xx11\,|\,1)\ \text{I}\ (0011\,|\,0) = 0011\,|\,q$$

and

$$(1x1x\,|\,1)\ \ \mathrm{I}\ \ (x010\,|\,0) = 1010\,|\,q$$

The final coface, 1xx1|1 is consistent since,

$$\begin{aligned}
(0011\,|\,0)\ \ &\mathrm{I}\ \ (1xx1\,|\,1) = q011\,|\,q\\
(x010\,|\,0)\ \ &\mathrm{I}\ \ (1xx1\,|\,1) = 101q\,|\,q\\
(xx00\,|\,1)\ \ &\mathrm{I}\ \ (1xx1\,|\,1) = 1x0q\,|\,1\\
(11xx\,|\,1)\ \ &\mathrm{I}\ \ (1xx1\,|\,1) = 11x1\,|\,1
\end{aligned}$$

Therefore, 1xx1 | 1 is a valid coface. We shall call this coface *z*.

Step 1.3 Replace *c* with *z* in C' to form C''

$$C'' = \begin{Bmatrix} 1xx1|1 & x010|0 \\ xx00|1 & 0011|0 \\ 11xx|1 & \end{Bmatrix}$$

Step 1.4 Remove from C'' all cubes contained in z to form C'''.

None to be removed.

Step 1.5 Set $C' := C'''$ and repeat steps 2 -4 until the resulting cover P consists solely of prime cubes.

Step 2.2 Select cube *c* = xx00|1. It has cofaces xxx0|1 and xx0x|1. Checking each for consistency w.r.t. C',

$$\begin{aligned}
xxx0\,|\,1\ \ \mathrm{I}\ \ x010\,|\,0 &= x010\,|\,q \quad \therefore \text{ inconsistent}\\
xx0x\,|\,1\ \ \mathrm{I}\ \ x010\,|\,0 &= x0q0\,|\,q \quad \therefore \text{ consistent}\\
xx0x\,|\,1\ \ \mathrm{I}\ \ 0011\,|\,0 &= 00q1\,|\,q \quad \therefore \text{ consistent}\\
xx0x\,|\,1\ \ \mathrm{I}\ \ 1xx1\,|\,1 &= 1x01\,|\,1 \quad \therefore \text{ consistent}\\
xx0x\,|\,1\ \ \mathrm{I}\ \ 11xx\,|\,1 &= 110x\,|\,1 \quad \therefore \text{ consistent}
\end{aligned}$$

So *z* = xx0x | 1 is a valid coface.

Step 2.3 Replace *c* with *z* in C' to form C'' .

$$C'' = \begin{Bmatrix} 1xx1|1 & x010|0 \\ xx0x|1 & 0011|0 \\ 11xx|1 & \end{Bmatrix}$$

Step 2.4 None to be removed.

$$C''' = \begin{Bmatrix} 1xx1\,|\,1 & x010\,|\,0 \\ xx0x\,|\,1 & 0011\,|\,0 \\ 11xx\,|\,1 & \end{Bmatrix}$$

Step 3.2 The cube 11xx | 1 has cofaces 1xxx | 1 and x1xx | 1 .

$$\begin{array}{ll}
1xxx|1 \ \mathrm{I} \ x010|0 = 1010|q & \therefore \text{ inconsistent} \\
x1xx|1 \ \mathrm{I} \ x010|0 = xq10|q & \therefore \text{ consistent} \\
x1xx|1 \ \mathrm{I} \ 0011|0 = 0q11|q & \therefore \text{ consistent} \\
x1xx|1 \ \mathrm{I} \ 1xx1|1 = 11x1|1 & \therefore \text{ consistent} \\
x1xx|1 \ \mathrm{I} \ xx00|1 = x100|1 & \therefore \text{ consistent}
\end{array}$$

So, x1xx | 1 is a valid coface.

Step 3.3

$$C'' = \begin{Bmatrix} 1xx1|1 & x010|0 \\ xx0x|1 & 0011|0 \\ x1xx|1 & \end{Bmatrix}$$

Step 3.4 None to be removed.

$$C''' = \begin{Bmatrix} 1xx1|1 & x010|0 \\ xx0x|1 & 0011|0 \\ x1xx|1 & \end{Bmatrix}$$

There are no redundant cubes to be removed so the minimised cover is

$$M = \begin{Bmatrix} 1xx1|1 & x010|0 \\ xx0x|1 & 0011|0 \\ x1xx|1 & \end{Bmatrix}$$

The cubes in this cover are prime since none of the cubes in *M* is contained in any other member of *M*. Although we have not decreased the number of cubes in the above cover, we have significantly decreased the number of bound coordinates.

2.4.5 The *P** Algorithm

The *P** Algorithm (pronounced PI-STAR) which is also due to Roth [6], generates a cover (either a 1-cover or a 0-cover) for any line in a circuit in terms of the primary inputs. The circuit line may be internal or a primary output. The algorithm is best explained with the aid of a simple example which is taken from [6]. Consider the circuit in Figure 2.3(a) and the corresponding truth table in Figure 2.3(b). This circuit has two primary inputs *a*, *b*, internal lines *c*, *d*, *e* and *f* and a single primary output *g*. The *P** Algorithm is now used to generate the 1-cover for the primary output *g*. Each stage of the algorithm is labelled on the left hand side of the table.

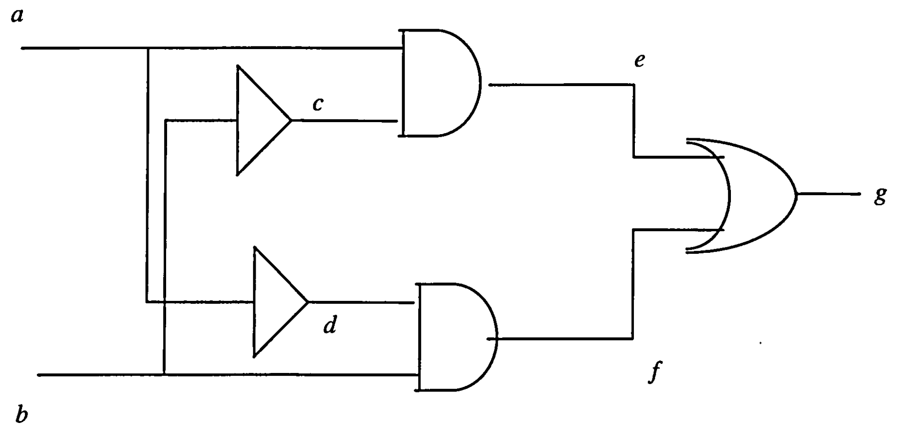
The *P** Algorithm starts from the primary output *g* of the circuit and proceeds backwards to the primary inputs. As the algorithm traverses the table, each stage assigns logic values to the circuit variables and a cover for the condition $g = 1$ (the *covering condition*) is gradually built. It is seen in stage 1 of this example that the value of 1 is inserted in the column corresponding to variable g. In stage 2, we substitute the cover that achieves this condition. The operator

$$P^*: g = \mathrm{OR}(e,f)$$

expresses the cover required for the variable g. From the circuit diagram it is apparent that g is given by the Boolean sum (OR) of the variables e and f. The (optimised) cover that sets $g = 1$ contains two cubes, e=1,f=x and e=x,f=1. The algorithm now has two new covering conditions since two cubes were generated in stage 2. The new covering conditions are always given by the right-most variable in the cover so far. The right-most variable in stage 2 is f and for the first cube in this cover, f = x. The operator $P^*: f = \mathrm{AND}(b,d)$ expresses the required cover which is b=x, d=x since f itself is unspecified. For the second cube in stage 2 the covering condition is f = 1 and the cover b=1, d=1 satisfies this. The two cubes are written in stage 3 and P^* continues recursively generating covers for the left most variable until the right most variable is a primary input. When this stage is reached, a cover for the original covering condition at stage 1 has been generated, that is , a 1-cover for the line g is {10|1, 01|1}.

As the P^* algorithm works its way backwards through a circuit, some circuit variable w of a cube v say, may have been set to a particular logic value by a previous operation. As the algorithm progresses, if in the cube v the value of the variable w coincides with the new value required of w, then no conflict exists and that value of w remains. If however a conflict occurs between the previously assigned value of w and that required by the new P^* operator, then that cube must be dropped from the cover as it represents an inconsistency.

As a further example, covers for outputs h, i and j for the circuit given in Figure 2.4 will be derived using the P* algorithm. In this particular case, we shall treat line g as a pseudo-primary input. That is, it shall be treated just as the primary inputs a, b, c and d. This will be explained further later in this chapter but this technique is required when deriving test sets for stuck-at faults at internal lines of a circuit.

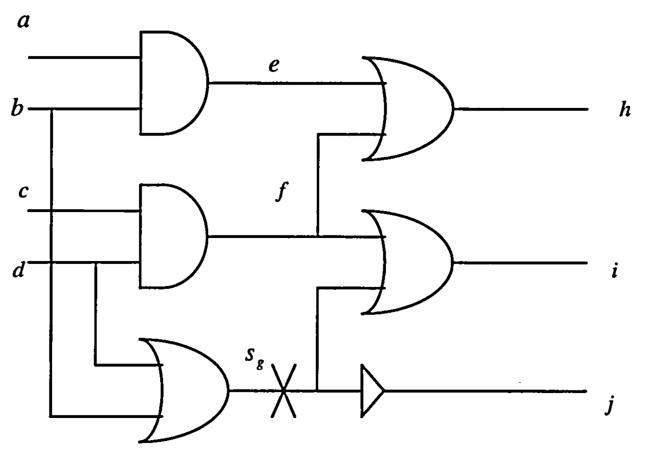


(a)

Stage	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*Operation*
1							1	
2					1	x		P^*:g = OR(e,f)
					x	1		
3		x		x	1			P^*f = AND(d,b)
		1		1	x			
4	1	x	1	x				P^*:e = AND(a,c)
	x	1	x	1				
5	1	x	1					P^*:d = NOT(a)
	0	1	x					
6	1	0						P^*:c = NOT(b)
	0	1						

(b)

Figure 2.3 (a) *Combinational circuit representing the XOR function,* $g = a.\bar{b} + \bar{a}.b$. **(b)** *Corresponding table showing the stages required to generate a cover for g =1 using the P* Algorithm.*



(a)

Stage	*a*	*b*	*c*	*d*	*e*	*f*	s_g	*h*	*i*	*j*	*Operation*
1								1			*h=1*
2					1	x					*P**:*h* = OR(*e*,*f*)
					x	1					
3			x	x	1						
			1	1	x						
4	1	1	x	x			x				*P**:*f* = AND(*c*,*d*)
	x	x	1	1			x				*P**:*e* = AND(*a*,*b*)
5									1		*I=1*
6						1	x				*P**:*i* = OR(*f*, s_g)
						x	1				
7	x	x	1	1			x				*P**:*f* = AND(c,*d*)
	x	x	x	x			1				s_g *=1*
8										1	*J = 1*
9	x	x	x	x			0				*P**:*j*= NOT(s_g)

NB: Cubes in stages 4, 7 and 9 represent final covers In the final stages, any unspecified co-ordinates for primary inputs and the pseudo-primary input are set to x.

(b)

Figure 2.4 (a) *Combinational circuit.* **(b)** *Corresponding truth table showing the stages required to generate the 1-covers for lines h, i, j using the P* Algorithm.*

2.4.6 Covers of Composite Functions

This following operations have been adapted from Roth's work [6] so that they may be used in a situation that involves the concept of pseudo inputs and the combining of functions, as is required to generate test vectors for single and multiple output circuits..

The three main cubical operations are interface, adjoin and sharp product (# product). These are analogues of the set theoretic operations $\cap$, $\cup$, \ (intersection, union and difference) respectively. Thus if C_1 and C_2 respectively are covers of two functions $F_1(X)$ and $F_2(X)$ having the same inputs and outputs, the interface of C_1 and C_2 is a cover for the product function, $F_1(X).F_2(X)$. Similarly application of the adjoin and sharp product operators to C_1 and C_2 results in covers for the functions $F_1(X)+F_2(X)$ and $F_1(X).\overline{F_2(X)}$ (or $\overline{F_1(X)}.F_2(X)$) respectively.

2.5 Test Pattern Generation Using Boolean Differences and Cubical Calculus

2.5.1 Derivation of Covers for Test Pattern Generation

Algebraically deriving the Boolean difference [8] is often seen as a tedious and manual process, requiring the use of many formulae and Boolean properties. This traditional method is unsuitable for computer implementation so a different approach is required. In their paper, Xue and Zhang [7] propose a test pattern generation algorithm using the Boolean difference based on cubical calculus and set theory. Their work is applicable to multiple input, single output combinational circuits only. The underlying theory of their method is now described and an algorithm based on this theory will be described in the following section.

Recall the definition of the Boolean difference given by equation (1),

$$g \oplus h \equiv g\bar{h} + \bar{g}h$$

For the single output case, the set theoretic analogue of equation (1) is,

$$g \Delta h = (g \setminus h) \cup (h \setminus g) = (g \cup h) \setminus (g \cap h) \qquad (14)$$

which is sometimes referred to as the symmetric difference. Equation (14) is illustrated graphically in Figure 2.5 below.

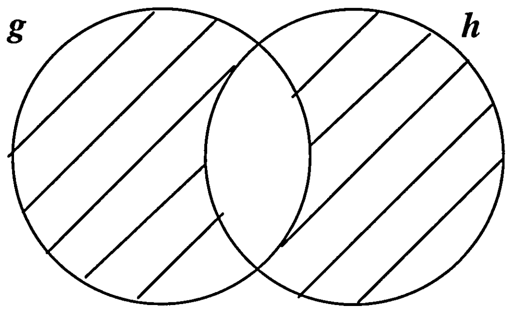


Figure 2.5 *Venn diagram illustrating equation (14). The shaded area represents the symmetric difference of the sets g and h.*

Let $F(X, s_j)$ be the Boolean function of a single output circuit with internal line s_j, as in section 2.2.1. Recalling that a cover is a set of cubes that unambiguously defines a function, let C_0 , C_1 and D be the covers for the functions $F(X, s_j = 0)$, $F(X, s_j = 1)$ and $dF(X, s_j)/ds_j$ respectively. The covers C_0 , C_1 can be obtained using Roth's P^* algorithm along with the algebraic descriptions of the circuit lines.

It follows from equation (14) above that the Boolean difference in terms of covers and cubical operations is given by,

$$\begin{aligned} D &= (C_0 \# C_1) \vee (C_1 \# C_0) \\ &= (C_0 \vee C_1) \# (C_0 \sqcap C_1) \end{aligned} \tag{15}$$

So, if S_j is a cover for the function $s_j(X)=1$, then the test sets for s_j stuck-at-0 and stuck-at-1 are given by the covers,

$$T_0 = D \sqcap S_j \tag{16}$$

$$T_1 = D \# S_j \tag{17}$$

respectively. The above equations are the cubical analogues of equations (4) and (5).

2.5.2 Test Set Generation Algorithm using Cubical Calculus - Single Output Case

In what follows, the test pattern generation algorithm developed by Xue and Zhang [7] will be described together with two examples. In each example, the at-fault line is identified using the notation s_j and test sets for s_j stuck-at-1 and stuck-at-0 will be generated. It is to be noted that the algorithm is suitable for multiple input, single output circuits only.

Test set generation algorithm (Xue and Zhang)

Step 1. Derive the covers C_0 , C_1 for the Boolean functions $F(X,s_j=0)$ and $F(X,s_j=1)$.

Recalling the equation,

$$\begin{aligned} \frac{dF(X,s_j)}{ds_j} &= F(x_1,\ldots,x_n,s_j) \oplus F(x_1,\ldots,x_n,\bar{s}_j) \\ &= F(X,s_j=0) \oplus F(X,s_j=1) \end{aligned}$$

it is necessary to derive the covers C_0 and C_1 for $F(X,s_j=0)$ and $F(X,s_j=1)$ respectively. We derive the covers C_0 and C_1 by generating a 1-cover for the primary output of the circuit. The 1-cover for the primary output is derived using the P^* algorithm and if line s_j is an internal line, it is treated as a pseudo primary input. For this 1-cover, the cubes in which the s_j coordinate is 0 are placed in C_0 and those cubes in which s_j is 1 are placed in C_1 . For those cubes in which s_j = x, two cubes are created for which $s_j = 0$ and s_j = 1 and these new cubes are placed in C_0 and C_1 respectively

Given that $\frac{dF(X,s_j)}{ds_j}$ is independent of s_j, the coordinates corresponding to s_j are then removed from each cube in both C_0 and C_1 .

Step 2. Derive the cover S_j for the function $s_j=1$.

This cover is also derived using the $P*$ algorithm.

Step 3. Derive the cover D.

This cover is derived using $D = (C_0 \text{ V } C_1) \# (C_0 \text{ I } C_1)$.

Step 4. Derive the covers T_0 and T_1.

The covers T_0 and T_1 contain the test vectors that respectively detect stuck-at-0/1 faults on the line s_j. Each cover is derived from D and S_j using the cubical equations (16) and (17).

Step 5. Remove redundant cubes in T_0 and T_1.

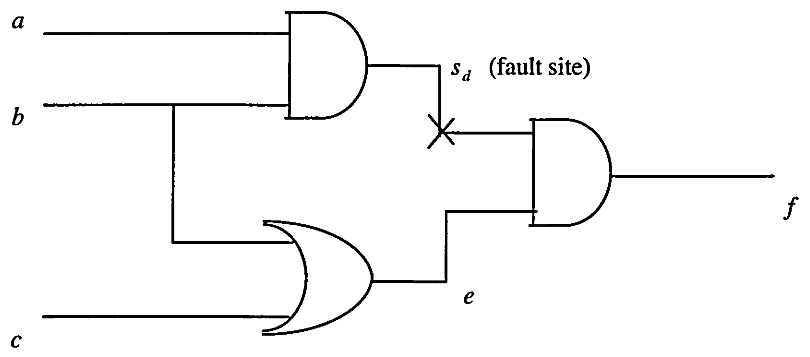
The covers T_0 and T_1 are minimised using the SHRINK algorithm.

End.

To illustrate the above algorithm, two examples shall be given.

Example 2.2

For the circuit given in Figure 2.6(a), the test sets T_0 and T_1 will be generated for a fault on the internal line *d*. So, let $s_j = s_d$.



(a)

Stage	*a*	*b*	*c*	s_d	*e*	*F*	*Operation*
1						1	
2				1	1		*P*:f* = AND(s_d,*e*)
3	x	1	x	1			*P*:e* = OR(*b,c*)
	x	X	1	1			

(b)

Figure 2.6 (a) *Combinational circuit.* **(b)** *Corresponding truth table showing the stages required to generate a 1-cover for line f using the P* Algorithm.*

Step 1. Derive the covers C_0 *,* C_1 *for the Boolean functions* $F(X, s_d = 0)$ and $F(X, s_d = 1)$.

The *P** algorithm for this example is given in Figure 2.6(b). In the final stage 3 of the *P** algorithm there are two cubes which constitute the 1-cover for the output line *f*. Since s_d=1 for both cubes, the cover C_0 is empty and

$$C_1 = \begin{Bmatrix} \text{x1x}|1 \\ \text{xx1}|1 \end{Bmatrix}.$$

Note that the value of the pseudo-input s_d is not explicitly included in the cubes of C_1 , but it is implicitly recorded in the suffix 1 of C_1.

Step 2. Derive the cover S_d *for the function* s_d=1

In this simple case it can be deduced by inspection that

$$S_d = \text{11x}|1$$

Step 3. Derive the cover D.

Since the cover C_0 is empty,

$$D = (C_0 \text{ V } C_1) \# (C_0 \text{ I } C_1) = C_1$$

Step 4. Evaluate the covers T_0 *and* T_1 *using equations (12) and (13).*

$$T_0 = D \text{ I } S_j = \begin{Bmatrix} \text{x1x}|1 \\ \text{xx1}|1 \end{Bmatrix} \text{I} \{\text{11x}|1\} = \begin{Bmatrix} \text{x1x}|1 \text{ I } \text{11x}|1 \\ \text{xx1}|1 \text{ I } \text{11x}|1 \end{Bmatrix} = \begin{Bmatrix} \text{11x}|1 \\ \text{111}|1 \end{Bmatrix}$$

$$T_1 = D \# S_j = \begin{Bmatrix} \text{x1x}|1 \# \text{11x}|1 = \text{01x}|1 \\ \text{xx1}|1 \# \text{ 11x}|1 = \text{0x1}|1, \text{x01}|1 \end{Bmatrix} = \begin{Bmatrix} \text{01x} | 1 \\ \text{0x1} | 1 \\ x\text{01} | 1 \end{Bmatrix}$$

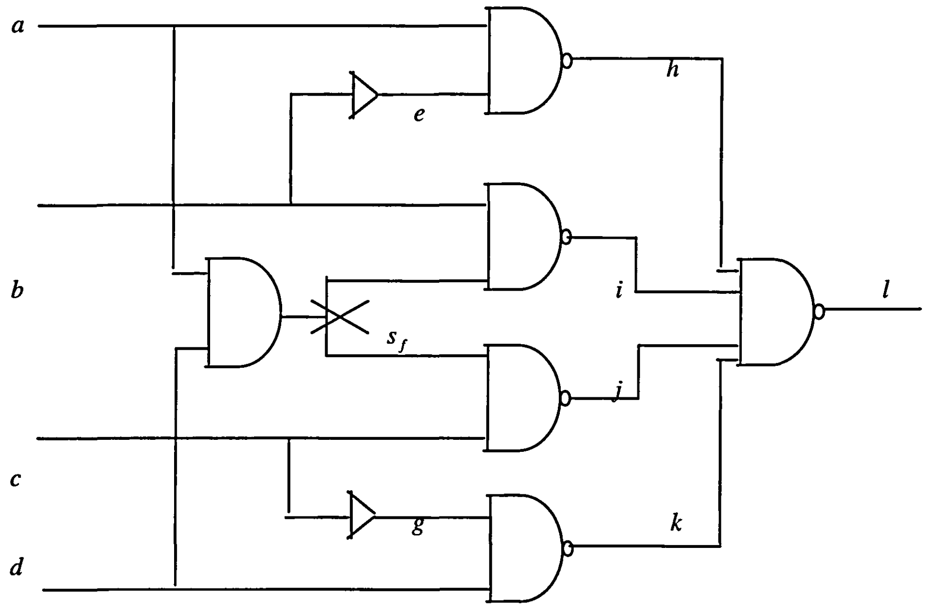
Step 5. Eliminate redundancy in covers T_0 and T_1.

In this simple case it may be verified by inspection that 111|1 is contained in 11x|1 and that 0x1|1 is contained in $\{01x|1,\ x01|1\}$. Hence the minimal covers are $T_0 = 11x|1$ and $T_1 = \{01x|1,\ x01|1\}$.

So from the above cover T_0, the input vectors 11x will detect a stuck-at-0 fault on line s_d. This test vector may be verified using the sensitive path technique for test pattern generation. So to test for a stuck-at-0 fault at line s_d, it must first be set to logic value 1. This is only achieved by setting the primary inputs *a* and *b* to logic 1. To propagate the value at line s_d to the primary output, one must set line *e* to logic value 1. Since the gate preceding line *e* is an OR gate, the fact that input *b* is already set to 1 will ensure this condition. The logic value on input *c* is therefore irrelevant and this fact is confirmed since this coordinate in the cube 11x|1is unspecified. A similar process may be followed to verify the test vectors given in the cover T_1.

Example 2.3 [taken from 7]

Figure 2.7 (a) gives a combinational circuit for which the stuck-at-1/0 faults at line s_f are to be tested. For brevity only the covers at each step will be given



(a)

Stage	*a*	*b*	*c*	*d*	*e*	s_f	*g*	*h*	*I*	*j*	*k*	*l*	*Operation*
1												1	
2								x	x	x	0		*P*:l* = NAND(*h,i,j,k*)
								x	x	0	x		
								x	0	x	x		
								0	x	x	x		
3	X	x	x	1	x	x	1	x	x	x			*P*:k* = NAND(*d,g*)
4	X	x	1	x	x	1	x	x	x				*P*:j* = NAND(s_f ,*c*)
5	X	1	x	x	x	1	x	x					*P*:l* = NAND(s_f ,*b*)
6	1	x	x	x	1	x	x						*P*:h* = NAND(*a,e*)
7	X	x	0	1	x	x							*P*:g* = NOT(c)
8	1	0	x	x		x							*P*:e*= NOT(b)

(b)

Figure 2.7 (a) *Combinational logic circuit taken from [7].* **(*b*)** *Results of* P^* *algorithm for circuit in* (*a*).

Step 1. Derive the covers C_0 *,* C_1 *for the Boolean functions* $F(X, s_f = 0)$and $F(X, s_f = 1)$.

The *P** algorithm for this example is given in Figure 2.7(b).

Stages 4, 5, 7, 8 provide the final 1-cover for the output line *l*. This may be summarised as,

$$\left\{\begin{array}{ccccc} a & b & c & d & s_f \\ \mathrm{x} & \mathrm{x} & 1 & \mathrm{x} & 1 \\ \mathrm{x} & 1 & \mathrm{x} & \mathrm{x} & 1 \\ \mathrm{x} & \mathrm{x} & 0 & 1 & \mathrm{x} \\ 1 & 0 & \mathrm{x} & \mathrm{x} & \mathrm{x} \end{array}\right\}$$

Note that only two cubes for which s_f =1 are included in C_1 only, whereas the two cubes for which s_f =x are included in both C_1 and C_0 to give,

$$C_1 = \begin{Bmatrix} \mathrm{xx01}|1 \\ \mathrm{xx1x}|1 \\ \mathrm{x1xx}|1 \\ \mathrm{10xx}|1 \end{Bmatrix} \qquad C_0 = \begin{Bmatrix} \mathrm{xx01}|1 \\ \mathrm{10xx}|1 \end{Bmatrix}$$

Step 2. Derive the cover S_f *for the function* $s_f = 1$

In all cases this can be generated using the *P** algorithm by using s_f as the starting point. In this simple case however it can be deduced by inspection that the cover is given by

$$S_f = 1\mathrm{xx}1|1$$

Step 3. Derive the cover D.

Since $C_0 \, \mathrm{V} \, C_1 = C_1$ and $C_0 \, \mathrm{I} \, C_1 = C_0$ this cover is derived using $D = C_1 \# C_0$.

$$D = \begin{Bmatrix} \mathrm{xx01}\,|\,1 \\ \mathrm{xx1x}\,|\,1 \\ \mathrm{x1xx}\,|\,1 \\ \mathrm{10xx}\,|\,1 \end{Bmatrix} \# \begin{Bmatrix} \mathrm{xx01}\,|\,1 \\ \mathrm{10xx}\,|\,1 \end{Bmatrix}$$

$$= \begin{Bmatrix} (\mathrm{xx01}\#\,\mathrm{xx01})\#\mathrm{10xx} \\ (\mathrm{xx1x}\#\,\mathrm{xx01})\#\mathrm{10xx} \\ (\mathrm{x1xx}\#\,\mathrm{xx01})\#\mathrm{10xx} \\ (\mathrm{10xx}\#\,\mathrm{xx01})\#\mathrm{10xx} \end{Bmatrix} = \begin{Bmatrix} \mathrm{0x1x}\,|\,1 \\ \mathrm{x11x}\,|\,1 \\ \mathrm{x1x0}\,|\,1 \end{Bmatrix}$$

Step 4. Evaluate the covers T_0 and T_1.

$$T_0 = D \, \mathrm{I} \, S_f = \begin{Bmatrix} \mathrm{0x1x}|1 \\ \mathrm{x11x}|1 \\ \mathrm{x1x0}\,|\,1 \end{Bmatrix} \mathrm{I} \{1\mathrm{xx}1|1\} = \{1111|1\}$$

$$T_1 = D \# S_f = \begin{Bmatrix} \mathrm{0x1x}\#1\mathrm{xx}1 \\ \mathrm{x11x}\#1\mathrm{xx}1 \\ \mathrm{x1x0}\#1\mathrm{xx}1 \end{Bmatrix} = \begin{Bmatrix} \mathrm{0x1x}\,|\,1 \\ \mathrm{011x}\,|\,1 \\ \mathrm{x110}\,|\,1 \\ \mathrm{x1x0}\,|\,1 \end{Bmatrix}$$

Step 5. Eliminate redundancy in covers T_0 and T_1.

The cover T_0 contains no redundancy. By inspection 011x|1 is contained in 0x1x|1 and x110|1 is contained in x1x0|1 so the minimal cover $T_1 = \{\mathrm{0x1x}|1, \mathrm{x1x0}|1\}$.

2.5.3 Test Set Generation using Cubical Calculus - Multiple Output Case

For the multiple output case, calculation of the Boolean difference requires a slightly different approach. No previous work has been developed for the multiple output case using the Boolean difference and cubical calculus and what follows is wholly original and was developed by the author.

Recall from section 2.2.2 that for a circuit with n inputs and m outputs, the Boolean difference with respect to a line s_j is given by,

$$\sum_{i=0}^{m} \frac{dF_i(X,s_j)}{ds_j}$$

where $X = (x_1,...,x_n)$ and $F_i(X)$ denotes the *i*th output for i =1,...,*m*. Given the above form of the Boolean difference for the multiple output case, it is necessary to develop the cubical analogue, which will be denoted as $\tilde{D}$. Given an equation to evaluate $\tilde{D}$, we can proceed to develop the cubical equivalents of equations (11) and (12) which yield the test sets for s_j stuck-at-0/1 respectively.

In the algorithm developed by Xue and Zhang, the covers C_0 and C_1 were used to calculate the cover

$$D = (C_0 \, \mathrm{V} \, C_1) \, \# \, (C_0 \, \mathrm{I} \, C_1)$$

for the single output case where C_0 and C_1 denote the 1-covers for $F(X, s_j = 0)$ and $F(X, s_j = 1)$ respectively. In the case of *m* outputs, there will exist *m* one covers for s_j =1 and *m* one covers for s_j= 0, one for each primary output.. First consider a cubical cover $\tilde{D}$ in the multiple output case which is calculated in a similar manner to D by grouping each of the '1' and '0' covers together and using the equation

$$\tilde{D} = (\tilde{C}_0 \, \mathrm{V} \, \tilde{C}_1) \, \# \, (\tilde{C}_0 \, \mathrm{I} \, \tilde{C}_1)$$

where $\tilde{C}_0 = C_0^1 \, \mathrm{V} \, C_0^2 \, \mathrm{V}.. \, ..\mathrm{V} C_0^m$, $\tilde{C}_1 = C_1^1 \, \mathrm{V} \, C_1^2 \mathrm{V}.. \, ..\mathrm{V} C_1^m$ and C_k^i is the 1-cover for $F_i(X, s_j = k)$, *k*= 1, 2. If this is done we would effectively be calculating the cubical analogue of the Boolean expression,

$$\frac{d\left(\sum_{i=1}^{m} F_i(X,s_j)\right)}{ds_j} \tag{18}$$

However what is really required is the cubical analogue of

$$\frac{dF_1(X,s_j)}{ds_j} + \frac{dF_2(X,s_j)}{ds_j} + \ldots \qquad \ldots + \frac{dF_m(X,s_j)}{ds_j} \tag{19}$$

The value of (19) is 1 if a stuck-at fault alters the value of 1 or more of the primary outputs, whereas the value of (18) is 1 if a stuck-at fault alters the Boolean sum of the values of all of the primary outputs. Thus in general, the result of setting these Boolean expressions equal to 1 and solving will not be the same. However, by introducing the concept of a *disjointness condition* for the covers C_k^i, it will be shown that the above expression for $\tilde{D}$ can be used to calculate a cubical cover for (19).

For simplicity, consider the two output case. First we will develop the cubical form of

$$\frac{dF_1(X,s_j)}{ds_j}+\frac{dF_2(X,s_j)}{ds_j} \tag{20}$$

Let C_0 , C_1 C_0' , C_1' denote the covers for the functions $F_1(X,0)$, $F_1(X,1)$, $F_2(X,0)$, $F_2(X,1)$ respectively. Then the cubical equivalent of (20) is,

$$D' = \left[(C_0 \text{ V } C_1) \# (C_0 \text{ I } C_1)\right] \text{ V } \left[(C_0' \text{ V } C_1') \# (C_0' \text{ I } C_1')\right] \tag{21}$$

For ease of manipulation we 'translate' this equation into its equivalent Boolean form using the correspondence between the cubical operators V, I, # and the Boolean operators +, . ,- respectively. Let $C_0 = a$, $C_1 = b$, $C_0' = a'$, $C_1' = b'$ and $D' = d$. Equation (21) then becomes

$$d = (a+b).\left(\overline{a.b}\right)+(a'+b').\overline{a'b'}$$

so that from De Morgans's theorem

$$d = a\overline{b}+b\overline{a}+a'\overline{b'}+b'\overline{a'}$$

and 'translating' back into cubical form gives

$$D' = (C_0\#C_1) \text{ V } (C_1\#C_0) \text{ V } (C_0' \# C_1') \text{ V } (C_1' \text{ V } C_0') \tag{22}$$

D' is the cubical analogue of equation (19) for m = 2. Now a similar form for equation (18) will be developed. Again for the two output case, we develop the cubical form of

$$\frac{d\left(F_1(X,s_j)+F_2(X,s_j)\right)}{ds_j}$$

The cover for the above function is given by

$$\tilde{D} = (C_0 \text{ V } C_1 \text{ V} C_0' \text{ V} C_1') \;\#\; \left((C_0 \text{ V} C_0') \;\text{ I }\; (\text{C}_1 \text{ V } C_1')\right)$$

Translating as before into Boolean algebraic form gives

$$\begin{aligned}\tilde{d} &= (a+b+a'+b') \,.\, \overline{(a+a').(b+b')}\\ &= (a+b+a'+b') \,.\, \left(\overline{(a+a')}+\left(\overline{b+b'}\right)\right)\\ &= \left(a.\overline{b}.\overline{b'}+b\overline{a\;a'}+a'\overline{b}\;\overline{b'}+b'\overline{a}\;\overline{a'}\right)\end{aligned}$$

Translating back into cubical form gives

$$\tilde{D} = \left((C_0\#C_1)\#\text{C}_1'\right) \text{ V } \left((C_1\#C_0)\#\text{C}_0'\right) \text{ V } \left((C_0'\#C_1)\#\text{C}_1'\right) \text{ V } \left((C_1'\#C_0)\#\text{C}_0'\right) \tag{23}$$

Comparing (22) and (23) it may be seen that D' contains $\tilde{D}$ and therefore $\tilde{D}$ in its current form may not contain all the cubes in D' . However, if the covers corresponding to different outputs are disjoint so that the following *disjointness condition*,

$$C_0 \cap C_0' = C_0 \cap C_1' = C_1 \cap C_0' = C_1 \cap C_1' = \varnothing \tag{24}$$

is satisfied where $\varnothing$ denotes the empty cube, then equation (23) is equivalent to (20). To explain further let us examine the first bracketed term in (23),

$$\left(\left(C_0 \# C_1\right) \# C_1'\right)$$

Since C_1' is disjoint with respect to C_0 and C_1 (the 1-covers of $F_1(X,0)$ and $F_1(X,1)$ respectively for the first output) then this term reduces to

$$\left(\,C_0 \# C_1\right)$$

Following this argument for the remaining three terms in equation (23),

$$\begin{aligned}\tilde{D} &= \left(C_0 \# C_1\right) \vee \left(C_1 \# C_0\right) \vee \left(C_0' \# C_1'\right) \vee \left(C_1' \# C_0'\right) \\ &= D'\end{aligned} \tag{25}$$

provided equation (24) is satisfied. Hence, provided the disjointness condition is satisfied, we may work with the cover $\tilde{D}$ instead of D'. It is straightforward to extend this analysis to the general m-output case so that,

$$\tilde{D} = \left(\bigvee_{k=0}^{l} \bigvee_{i=0}^{m} C_k^i\right) \# \left(\left(\bigvee_{i=1}^{m} C_0^i\right) \cap \left(\bigvee_{i=1}^{m} C_1^i\right)\right)$$

provided the disjointness condition

$$C_k^p \cap C_l^q = \varnothing, \; p \neq q$$

is satisfied for all $p, q \in \{1, \ldots, m\}$, $k, l \in \{0,1\}$.

Now we have found the cubical form for cover $\tilde{D}$ we are able to find the cubical forms for the test sets T_0 and T_1. In a similar manner to the single output case the test sets for s_j stuck-at-0 and stuck-at-1 are given by the covers

$$T_0 = \tilde{D} \cap S_j \tag{26}$$

$$T_1 = \tilde{D} \# S_j \tag{27}$$

respectively where S_j is the cover for s_j=1. These two equations are the 'cubical' equivalents of equations (11) and (12).

Test set generation algorithm: multiple outputs

The algorithm required for the multiple output case remains almost identical to that developed by Xue and Zhang. Each step for the multiple output case will now be given, indicating any divergence from the single output case. At each step the cover(s) obtained are optimised using SHRINK.

Step 1. For i=1,...,m, derive the 1-covers C_0^i, C_1^i for the Boolean functions $F_i(X, s_j = 0)$, and $F_i(X, s_j = 1)$ respectively. Deduce the covers C_0 and C_1 using the relations

$$C_0 = C_0^1 \vee C_0^2 \vee \ldots \vee C_0^m$$
$$C_1 = C_1^1 \vee C_1^2 \vee \ldots \vee C_1^m$$

The P^* algorithm is used to derive the covers C_k^i; if line s_j is an internal line, it is treated as a pseudo primary input in each case. It is at this point that the disjointness condition is brought into the algorithm. When using the P^* algorithm to derive a 1-cover for output i, a 1 in the *i*th output coordinate of a cube in C_0^i or C_1^i signifies in the usual way that the value of output i is 1 for the given input. Zeros are inserted in the remaining $m-1$ output coordinates of these cubes to ensure that cubes corresponding to distinct outputs are disjoint, so that $\tilde{D}$ satisfies (25).

When in the derivation of 1-covers for $F_1(X,s_j), F_2(X,s_j), \ldots\ldots, F_m(X,s_j)$ the coordinate corresponding to s_j is undefined i.e. is equal to x, then this single cube is transformed into two new cubes with one cube containing s_j=0 and the other containing s_j=1. These new cubes are then inserted into C_0 and C_1 respectively (cf. Section 2.5.2, Example 2.2).

Step 2. Derive the cover S_j for the function $s_j = 1$.

Same as the single output case, using the P^* algorithm.

Step 3. Derive the cover $\tilde{D}$.

The cover $\tilde{D}$ is given by the operation

$$\tilde{D} = (C_0 \vee C_1) \# (C_0 \mathrm{I}\, C_1)$$

Step 4. Derive the covers T_0 and T_1.

The covers are given by,

$$T_0 = \tilde{D} \,\mathrm{I}\, S_j$$

$$T_1 = \tilde{D} \# S_j$$

Step 5. Remove redundant cubes in T_0 and T_1.

The covers T_0 and T_1 are minimised using the SHRINK algorithm.

End.

Two examples will now be presented to illustrate this algorithm.

Example 2.4

Consider a stuck at fault on line s_g in the circuit given in Figure 2.4a.

Step 1. Derive the 1-cover C, for the Boolean functions $F_1(X, s_j), F_2(X, s_j), \ldots\ldots, F_m(X, s_j)$ *corresponding to each of the primary outputs in a circuit and find the covers* C_0 , C_1 .

Referring back to Figure 2.4b, we can see the derivation of the 1-covers for each output of the circuit. From these covers and by removing redundant cubes using SHRINK we have the covers,

$$C_0 = \begin{Bmatrix} \text{xx11}\,|\,\text{1xx} & \text{xx11}\,|\,\text{x1x} \\ \text{11xx}\,|\,\text{1xx} & \text{xxxx}\,|\,\text{xx1} \end{Bmatrix},\ C_1 = \begin{Bmatrix} \text{xx11}\,|\,\text{1xx} & \text{xxxx}\,|\,\text{x1x} \\ \text{11xx}\,|\,\text{1xx} & \text{xx11}\,|\,\text{x1x} \end{Bmatrix}$$

Step 2. Derive the cover S_j *for the function* $s_j = 1$

Again, using the P* algorithm or by inspection,

$$s_g = \{\text{xxx1|xxx, x1xx|xxx}\}$$

whose input parts provide a 1-cover for line s_g .

Step 3. Derive the cover $\tilde{D}$.

Given the covers C_0 and C_1 $C_0 \vee C_1 = \begin{Bmatrix} \text{xx11}\,|\,\text{1xx} & \text{xxxx}\,|\,\text{x1x} \\ \text{11xx}\,|\,\text{1xx} & \text{xxxx}\,|\,\text{xx1} \end{Bmatrix}, C_0 \mathrm{I} C_1 = \begin{Bmatrix} \text{xx11}\,|\,\text{1xx} & \text{xx11}\,|\,\text{x1x} \\ \text{11xx}\,|\,\text{1xx} & \end{Bmatrix}$

the Boolean difference $\tilde{D}$ can be evaluated.

$$\tilde{D} = (C_0 \vee C_1)\#(C_0 \mathrm{I} C_1) = \begin{Bmatrix} \text{xx0x}\,|\,\text{x1x} \\ \text{xxx0}\,|\,\text{x1x} \\ \text{xxxx}\,|\,\text{xx1} \end{Bmatrix}$$

This is a cover for the sum (OR) of the Boolean differences of the three output functions with respect to line s_g. The cube xx0x|x1x in $\tilde{D}$ signifies that if input *c* is set to 0, then regardless of the value of the other inputs, a change of value in line s_g (treated as a primary input) results in a change of value in output *i*. The interpretation of the other cubes in $\tilde{D}$ is similar; as with C_0 and C_1, the zeros in the output part of the cubes in $\tilde{D}$ carry no significance.

Step 4. Derive the covers T_0 and T_1.

Using the interface and sharp product of $\tilde{D}$ and s_g gives

$$T_0 = \tilde{D} \sqcap S_g = \begin{Bmatrix} \text{xx01}\,|\,010 & \text{xxx1}\,|\,001 \\ \text{x1x0}\,|\,010 & \text{x1xx}\,|\,001 \\ \text{x10x}\,|\,010 & \end{Bmatrix}$$

$$T_1 = \tilde{D} \# S_g = \{\text{x0x0}\,|\,010 \;,\; \text{x0x0}\,|\,001\}$$

which provide the test sets for line s_g stuck-at-0 and stuck-at-1 respectively. Note that $T_0 \vee T_1 = \tilde{D}$, and that the presence of a 1 in the output side of a cube in T_0 or T_1 signifies that the fault is observable at the corresponding output. Thus the cube x1x0|010 in T_0 signifies that if inputs *b* and *d* are set to 1 and 0 respectively, then regardless of the values of the other inputs, line s_g stuck-at 0 is observable at output *i* under these input tests.

Example 2.5

Consider a stuck-at fault at line s_d in the circuit given in Figure 2.8.

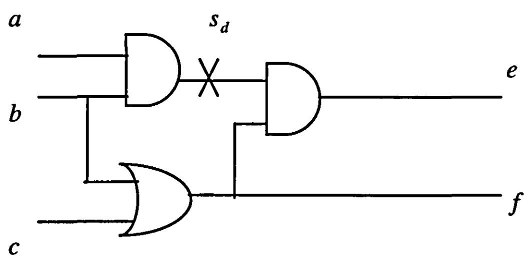


Figure 2.8. *Multiple output circuit with 'stuck-at' fault on line* s_d

Step 1. Derive the 1-cover C, for the Boolean functions, $F_1(X, s_j), F_2(X, s_j), \ldots\ldots, F_m(X, s_j)$ *corresponding to each of the primary outputs in a circuit and find the covers* C_0 , C_1 .

If line s_d is treated as a primary input then the 1-covers of outputs *e* and *f* respectively are {x1x1|10, xx11|10} and {x1xx|01, xx1x|01}. The covers C_0 and C_1 are deduced from the following table.

a	*b*	*c*	s_d	*e*	*f*	*Operation*
					1	*f=1*
x	1	x	x			*P**:*f* = OR(*b*,*c*)
x	x	1	x			
				1		*e=1*
			1		1	*P**:e = AND(s_d.*f*)
x	1	x	1			*P**:*f* = OR(*b*,*c*)
x	x	1	1			

This gives

$$C_0 = \begin{Bmatrix} \text{x1x|01} \\ \text{xx1|01} \end{Bmatrix}, \quad C_1 = \begin{Bmatrix} \text{x1x|01 x1x|10} \\ \text{xx1|01 xx1|10} \end{Bmatrix}$$

Step 2. Derive the cover S_j for the function $s_j = 1$

Using the P* algorithm,

$$S_d = \{\text{11x|xx}\}$$

whose input parts provide a 1-cover for line s_d and the output coordinates have been unspecified for consistency.

Step 3. Derive the cover $\tilde{D}$.

Given the covers C_0 and C_1

$$C_0 \text{ V } C_1 = \begin{Bmatrix} \text{x1x|01 x1x|10} \\ \text{xx1|01 xx1|10} \end{Bmatrix}$$

$$C_0 \text{ I } C_1 = \begin{Bmatrix} \text{x1x|01} \\ \text{xx1|01} \end{Bmatrix}$$

the Boolean difference $\tilde{D}$ can be evaluated.

$$\tilde{D} = (C_0 \text{ V } C_1) \# (C_0 \text{ I } C_1) = \begin{Bmatrix} \text{x1x|10} \\ \text{xx1|10} \end{Bmatrix}$$

Step 4. Derive the covers T_0 and T_1.

Using the interface and sharp product of $\tilde{D}$ and s_d gives

$$T_0 = \tilde{D} \sqcap S_d = \{11\text{x} \mid 10\}$$
$$T_1 = \tilde{D} \# S_d = \begin{Bmatrix} 01\text{x} \mid 10 \\ \text{x}01 \mid 10 \end{Bmatrix}$$

which provide the test sets for line s_d stuck-at-0 and stuck-at-1 respectively. Thus the cube 11x|10 in T_0 signifies that if inputs *a* and *b* are both set to 1, then regardless of the values of input *c*, line s_d stuck-at 0 is observable at output *e*.

2.6 Testability Analysis using Cubical Calculus

Testability analysis [11, 12] of digital circuits quantifies the ease of testing a circuit design without performing the computationally expensive process of test pattern generation. The difficulty of test generation can vary considerably for different implementations of the same digital function. The ability to assess the testability of a particular design can help identify areas of the circuit that are particularly difficult to test and help in the selection of competing designs. Testability analysis therefore is of real value during the design stages of a circuit when the designer requires a quick yet accurate estimation of testability without having to submit a design to full test pattern generation. It is imperative then that any testability tool be computationally quicker than the test generation process. Two main classes of testability analysis tools exist.

Early testability tools were based on scoring methods [13, 14]. These methods allocate a score to a circuit based on certain features that are seen to either improve or degrade circuit testability. Each feature is allocated a predetermined score in direct proportion to whether it improves or reduces overall circuit testability. For example, features detrimental to testability may be the inability to set the circuit to a predetermined state and reconvergent fan-outs. Conversely, a feature that may be seen to improve testability is the presence of a large number of primary inputs. To calculate the final testability score of a design, all the features of a design are identified and their corresponding testability scores are combined using simple arithmetic. Although these scoring methods are relatively easy to implement, they only provide a crude measure of testability. The measures may crudely distinguish one design as being, on average, more testable than another but the final scores are unable to identify regions of poor testability. In this respect they are very limited in their usefulness and as a result have been superceded by other methods.

Algorithmic methods such as CAMELOT [15] and TMEAS [16] are far superior to scoring methods and produce testability analysis based on circuit topology. In these methods, the testability of each circuit line is quantified. An overall testability measure is again calculated by some combination of the testability of individual lines. There has been much controversy over the exact calculation of overall testability highlighted in [17]. The exact approach used to calculate the testability of a line differs from one algorithm to another. However the majority are based on the mathematical models of two important operations in test pattern generation. The first operation when generating a test vector for a stuck-at-1/0 fault on line s is to set that line to logic 0/1. A measure of the ease with which s can be set to 0/1 is known as its *controllability.* The next operation required to generate a test is to propagate the fault at s to a primary output. A measure of the ease with which this is achieved is known as its *observability.* Overall circuit controllability and observability are then calculated by, for example, averaging the controllabilities and observabilities of the individual lines. An overall testability measure is then calculated by, for example, multiplying the overall controllability and observability. A number of techniques exist and are not without their critics [17].

Different testability algorithms employ different measures for controllability and observability and a number of comparative studies exist [18]. The SCOAP [19] algorithm for example, characterises testability using different measures at each line. Controllability is defined by the proportion of lines that have to be be set to logic 1 or 0 for the line to be set to 1/0. Observability is measured by the proportion of lines that need to be set to 1/0 in order to propagate a given line's logic value to a primary output.

The concept of observability has already been encountered in this chapter when the framework for test pattern generation using cubical calculus was explained. In what follows a new method for calculating testability measures is proposed using cubical calculus, developed wholly by the author.

Probabilistic techniques for measuring both controllability and observability have been described in [17] based on the assumption that all input vectors are equally likely. This method ties in very well with some of the covers required for the test pattern generation techniques described earlier. We therefore assume that all input vectors are equally likely and also, for simplicity, confine our attention to the single output case.

Definitions

The *i-controllability* $C_i(s_j)$ of a line s_j is a measure of the ease with which s_j can be set to 0 or 1 and is given by

$$C_i(s_j) = \text{ proportion of input vectors that set } s_j \text{ to } i$$

Observability is a measure of the ease with which a change of value at a line can be observed at the primary output. The observability of a line s_j at the primary output k is given by

$$O_k(s_j) = \text{proportion of input vectors for which a change of value at } s_j \text{ results in a change of output value}$$

Using the above definitions, the cover corresponding to the solution of the Boolean equation $s_j(X) = 1$ can be used to quantify controllability. The solution of the Boolean difference equation (3) provides all the input vectors that sensitise a path between the fault site at line s_j and the primary output , i.e. for which the value of s_j is observable at the primary output. Hence we can use the 1-cover of the Boolean difference $dF(X, s_j)/ds_j$ to quantify observability. So, returning to the discussion of Section 2.5.2, the covers S_j and D can be used to calculate the controllability and observability respectively of the node s_j. The calculations are a simple matter of counting the total number of input vectors in each of the aforementioned covers. As an example, a circuit from [17] will be analysed.

Example 2.6

We calculate the observability and 1-controllability of line 3 for the single output circuit in Fig. 2.9

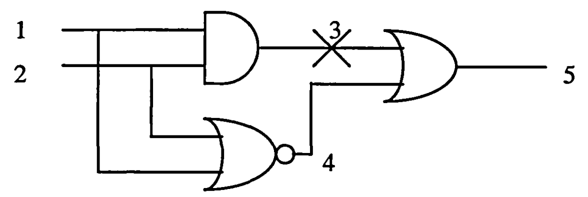


Fig. 2.9. ***Combinational Circuit with 'stuck-at' Fault on Line 3***

Since $C_0 = \{00|1\}$, $C_1 = \{xx|1\}$, giving $D = \begin{Bmatrix} 1x|1 \\ x1|1 \end{Bmatrix}$ the total number of cubes in the above cover , D is 3 and the total number of input vectors is 2^2 so $O_5(s_3)$ $= 3/4$. Also the cover $S_3(X) = \{11|x\}$ contains a single input vector so $C_1(s_3)$ = ¼. This example shows that testability measures are very easily calculated from the relevant covers. Final controllability, observability and testability measures for the circuit would be obtained by combining, in a predetermined manner, the measures corresponding to all the lines in the circuit.

2.7 Summary

This chapter has introduced a new test pattern generation algorithm for multiple output circuits using Boolean Differences and cubical calculus. This chapter provided the reader with an overview of the fundamentals of cubical calculus and an explanation of the Boolean difference technique for test pattern generation. It was explained that, although the Boolean difference is an elegant technique for test pattern generation, its use unfortunately was limited because of the algebraically cumbersome methods required for its evaluation. If an alternative method could be found to solve the Boolean difference, then surely this technique could be suitable for wider adoption in ATPG systems. The central aim of this chapter was to outline such a solution. The solution was provided by Roth's cubical calculus, albeit with some minor modifications to his original work.

The algorithm provided by Xue and Zhang for single output combinational circuits certainly laid some of the foundations for this work. But a number of cubical calculus operations were redefined by the author in order to cope with the multiple output case. The 'disjointness condition', as introduced in section 2.5 and described by equation (24), is central to the application of cubical calculus to the solution of the Boolean difference. Once this was proven algebraically, a test pattern generation algorithm was developed that was shown to efficiently generate test vectors for a limited number of examples. By overcoming the problems associated with the solution of the Boolean difference, the author has developed an original test pattern generation algorithm that retains the advantages of the Boolean difference while at the same time, avoiding some of its disadvantages.

In the wider context of test pattern generation techniques, it is felt that this new test pattern generation algorithm could lay the foundations for a new class of test pattern generation tools. The vast majority of today's ATPG algorithms are based on topological traversal techniques. For example Roth's D-algorithm, PODEM and FAN are such algorithms. Given the computational expense of such topological algorithms, the author feels there may be some significant value in an algorithm, such as the one described above, that is able to perform some of the work of test pattern generation in a more algebraic manner.

The chapter ended with some very interesting, original work on testability measures. This work was developed after the test pattern generation algorithm and fits in well with existing, probabilistic definitions of testability measures. Given today's complex circuits, testability measures are more important than ever as they provide a means of predicting the ability to test a circuit at the initial design stages. Again, taking an algebraic rather than topological approach may have many advantages and cubical calculus could prove a very useful approach in this area.

This work presented in this chapter has certainly opened up the topic of algebraic test pattern generation, based on the Boolean difference and cubical calculus, for further study. It is felt by the author that there are some key areas of investigation. Firstly, the relative merits of this technique versus traditional topological algorithms must be investigated. This will first involve the efficient software implementation of the algorithm described in section 2.5.3 above. This will enable benchmarking (speed of execution, memory usage) against existing algorithms. Secondly, it will be important to evaluate the quality of the test sets generated by the algorithm. How large are the test sets relative to

PODEM, FAN etc? Also, it may be possible to achieve some reduction in test effort during test pattern generation when using the Boolean difference. In the earlier discussion, it was mentioned that the solving the Boolean difference provides all possible tests for a given fault. But what if only 95% fault coverage was required or only one test vector was required for each fault? Defining these parameters during the test pattern generation phase may result in realizing some significant economies. A final area for further study would be to investigate the application of cubical calculus and the Boolean difference to sequential circuits. Given the current nature of the microelectronics industry, sequential test pattern generation is an important and prominent field of study within digital testing.

In conclusion, the author is encouraged by the findings and the results presented in this work. It has been shown that cubical calculus is a valuable tool for the solution of the Boolean difference as applied to test pattern generation. The algorithms developed in this chapter, prima facie, seem to lend themselves well to computer implementation, mitigating the need for cumbersome algebra when solving test pattern generation equations. An added bonus of using cubical calculus as described here is that it can also be used to design minimised logic functions. Therefore, it may be possible to produce a digital design suite, which could incorporate circuit design , testability measurement and test pattern generation in a single digital design tool. Given the power of modern software engineering techniques and the relentless pace of Moore's Law, such a suite could possibly be run on today's desk-top computers.

2.8 References

[1] Sellers F.F., M.Y Hsiao, L.W. Bearnson, "Analysing errors with the Boolean difference", IEEE Transactions on Computers, Vol-C17, No. 7, July 1968.

[2] Russell G. et al, "CAD FOR VLSI", Van Nostrand (UK) 1985.

[3] Abramovici M., Breuer M.A., Friedman A.D., "DIGITAL SYSTEMS TESTING AND TESTABLE DESIGN", IEEE Press, 1990.

[4] Bearnson L.W., "ARITHMETIC ERROR DETECTION IN DIGITAL COMPUTERS", M.S. Thesis, Dept. of Electrical Engineering, Syracuse University, Syracuse, N.Y., U.S.A., May 1965.

[5] Amar V. and Condulmari N., "Diagnosis of large combinational networks", IEEE Transactions on Electronic Computers (Correspondence), Vol. **EC-16**, pp. 675-680, Oct. 1967.

[6] Roth J.P. "COMPUTER LOGIC, TESTING AND VERIFICATION", Computer Science Press, Maryland USA, 1980.

[7] Xue H.X, Zhang Y.N., "A test generation algorithm based on Boolean differences and cubical operations", New Advances in Computer Aided Design and Computer Graphics, Vol 1 and 2, Ch 166, p634-637, 1993.

[8] Shannon, C.E., "A symbolic analysis of relay and switching circuits", Transactions, AIEE, vol. 57, pp. 713-723, 1940.

[9] Boole G., "AN INVESTIGATION OF THE LAWS OF THOUGHT", The Open Court Publishing Company (1916). Reprinted by Dover Publications, 1951.

[10] Roth J.P., "Programmed logic optimisation", IEEE Trans. Computers, vol. C-27, No.2, February 1978.

[11] Miller K.W., "Testability - an introduction for COMPASS94", COMPASS 1994, Proceedings of the Ninth Annual Conference on Computer Assurance, 1994, Chapter 26, p173-174.

[12] Agrawal V.D., Mercer M.R., "Testability Measures - What do they tell us?" Proceedings of the. IEEE Test Conference, pp401-406, 1982.

[13] Dejka W.J., "Measure of testability in device and system design", Proceedings of the 20th Midwest Symposium on Circuits and Systems, pp38 - 52, August 1977.

[14] Wood C.T., "The quantitative measure of testability", Proceedings of IEEE Automatic Test Conference, pp286-291, 1979.

[15] Bennetts R.G. et al., "CAMELOT: A computer aided measure for logic testability", Proc. IEE., Vol 128-E, p177-189, Sept. 1981

[16] Grason J., "TMEAS a testability measurement program" in Proc. 16th IEEE Design Automation Conference., San Diego, CA, p156-161, June 1979.

[17] Savir J., "Good controllability and observability do not guarantee good testability", IEEE Trans. on Computers, Vol. **C-32**, p1198-1200, Dec 1983.

[18] Roberts, M.W. and Lala P.K., "Testability measures in digital circuits - A critique", Proceedings of the 29th Midwest Symposium on Circuits and Systems, pp347 - 351, August 1986.

[19] Goldstein L.H., and Thigpen E.L., "SCOAP: Sandia controllability/observability analysis program", Proccedings of the 17th IEEE Design Automation Conference, pp190-196, June 1980.

Chapter 3. Genetic Algorithms

3.1 Introduction

Over the last fifteen years Genetic Algorithms (GAs) have proven themselves to be a robust optimisation and search tool and have successfully been used across a wide variety of applications such as machine learning, music generation and engineering design. They draw inspiration from the Darwinian ideas of evolution and natural selection and closely mimic a number of biological reproduction operations. GAs use directed, probabilistic search techniques to locate global optima in large, complex search spaces and are well suited to solving NP-hard problems.

GAs are largely attributable to the work of John Holland in the mid-sixties. Although the marriage of Computer Science and evolution was first proposed in the late 1950s, it was he who placed the field on a firm mathematical footing. When his book, "Adaptation in Natural and Artificial Systems" [1] was first published in 1975, research interest in the area was confined to a handful of people, "my students and their colleagues" . In the early 1980s interest in GAs began to rise rapidly for a number of reasons; a change of focus in Artificial Intelligence, the realisation that they could be used as an analytic tool for complex adaptive systems and the appearance of favourable, comparative studies between GAs and other optimisation techniques. However, it has only been over the last ten years or so that GAs have been embraced by the wider academic and industrial community. GAs have matured into a popular and robust optimisation technique and are now regarded as part of the mainstream in computer science, engineering and mathematics.

Darwin's theories of evolution were first outlined in his book, "Origin of Species" [2] and his ideas of *natural selection* and *survival of the fittest* are key themes within all GAs. In nature the ability of individuals to attract mates and produce offspring is directly related to their ability to survive and adapt to their environment. The competition for food, shelter and other resources will be won by the stronger or fitter individuals while the weaker ones may well die away. These relatively fit individuals will produce proportionally more offspring than the weaker ones. Since the characteristics of individuals (eye colour, strength, ability to detect danger etc.) are encoded in their genes, survival of the fittest actually implies survival of the fittest genes. Sets of genes are known as chromosomes and these determine the entire make-up of an individual. The combination of good characteristics (or good genes) from two mates will sometimes produce super-fit individuals and this is the process by which species evolve and adapt to their environment.

By loosely emulating nature GAs evolve populations of potential solutions to a problem. By devising an appropriate coding scheme, a solution may be represented as a chromosome and it is to populations of chromosomes that the ideas of natural selection and survival of the fittest are applied. Associated with each chromosome is a fitness value which rates its competence as a solution. The greater the fitness of a chromosome, the better it is as a solution and the greater the probability of it being selected as a mate - the natural selection analogy. Therefore the relatively fit solutions will produce proportionally more offspring than the weaker ones and just as in nature this implies that the fit genes survive and the weaker die - the survival of the fittest analogy. So as generations arise, the good solutions will mate, combine their genes with other solutions and hopefully produce better and better solutions.

The complexity of NP-hard problems, in terms of computer time and resources render classical methods of optimisation futile. The Traveling Salesperson Problem (TSP), a classic NP-hard combinatorial optimisation problem, is an excellent illustration of the magnitude of such an optimisation task. The goal is to find the shortest possible route between a number of cities, *n*, that an imaginary salesperson has to visit. The brute force method would be to evaluate the distance travelled for every possible route. However, as *n* increases the number of possible routes increases as *n*!. For the 4 city problem there are 24 possible routes. For 25 cities the solution of the problem is equivalent to finding a single raindrop in all the world's oceans. Problems such as this would take desktop computers millions of years to solve, yet such problems do exist (for example an oil tanker visiting a number of petrol stations) and we require reasonable solutions to be calculated in minutes. Using GAs a team at British Telecom PLC. was able to find a solution to a 3000 city problem in just 25 minutes which was within a few per cent of the optimal solution [3].

There are many other examples of complex real world problems to which we require 'good' solutions. In many cases the emphasis is not on finding the optimal solution but rather on finding a good solution in an acceptable period of time. Although no golden panacea exists for the application of GAs to optimisation problems, they do however provide a proven technique that has been successful across a wide class of problems. By evolving from initial mediocre solutions, GAs are able to cleverly locate reasonable solutions to large problems in relatively short periods of time.

3.2 GA Terminology

GAs are largely inspired by biological mechanisms and entities and as a result much of the associated terminology is borrowed from nature. All living organisms consist of cells which in turn consist of *chromosomes*. Sets of chromosomes determine the entire make-up of an organism and thus can be regarded as the biological 'blue-print' for that organism. Chromosomes consist of *genes* and each gene represents a particular characteristic of that organism, hair colour for example. The different possible values of each gene are known as *alleles* and in the case of human hair colour may take the value, black, brown, blonde and so on. The position of each gene within a chromosome is known as its *locus*.

In GAs, candidate solutions are encoded as bit strings known as chromosomes. Each bit within a chromosome is a gene and its allele depends on the type of coding scheme that has been devised for the chromosomes. In the case of binary encoded chromosomes, each gene has an allele of either 1 or 0.

3.3 Search Spaces and Fitness Landscapes

The goal of many optimisation techniques is to locate an optimal solution amongst a number of candidate solutions. The set of candidate solutions is often referred to as the search space in which we are searching for a particular solution. The search space may be visualised as a landscape containing features such as hills and valleys upon which solutions to the problem are located. Just as a conventional landscape may be mapped in terms of coordinates, each solution may be regarded as having a unique coordinate within the search space. In the context of maximising an objective function it is usual to search the landscape for the highest peak as it is here that the optimal solution will be located.

GAs may be thought of as continually roaming such landscapes in an effort to locate the highest peaks or the deepest valleys. By evaluating the fitness of each candidate solution the GA is able to rate the position (height or depth) of a solution on the landscape. In terms of maximisation, those that are located on relatively high ground (those of high fitness) mate with other solutions sometimes creating solutions that are located on even higher peaks. As generations evolve the population of solutions will begin to converge on a small number of peaks. The GA cannot guarantee that it has found the global optimum but it will have improved on the initial, randomly generated solutions.

To prevent premature convergence on local (rather than global) optima, the mutation operator ensures the GA keeps an open mind about areas of the landscape that have yet to be explored. By randomly altering genes, mutation allows exploration of uncharted territory that would not have been visited through the recombination of existing chromosomes alone. Throughout this chapter, the concept of searching landscapes will be revisited as it provides a convenient means of visualising the operation of GAs.

3.4 Genetic Algorithm Fundamentals

3.4.1 GA Overview

To apply a GA one must have a clearly defined problem and be able to represent candidate solutions as strings known as *chromosomes*. From the problem definition one must also be able to formulate a means of assessing a solution by, for example, evaluating a fitness function.

GAs evolve sets of *N* candidate solutions, each set known as a *population*. Each evolutionary time step is known as a *generation* and the entire number of evolved generations is known as a *run* of the GA. New generations are produced by a process known as *crossover*, whereby selected parent chromosomes exchange genetic material to produce child chromosomes. To ensure continuing genetic diversity, a proportion of each new generation of chromosomes is subjected to *mutation*, in which a randomly chosen gene is altered. There are a number of ways of terminating a run, the popular methods being; evolving a given number of generations, *G* and stopping a run when a given population has converged on a particular solution. When a GA terminates it will present the fittest individual(s) found during the run. Since there are many random and probabilistic mechanisms within a GA, given different seeds, two runs of a GA will often produce different generational results. The final result may be the same but the manner in which each run gets to the best chromosome will differ from run to run.

Given then, that the criteria for applying a GA have been matched, the overall structure of a typical GA is illustrated by the high-level code given in Figure 3.1.

```
Set GA parameters
 population size, N
 number of generations to be evolved, G
 crossover probability, c
 mutation probability, m

Generate initial random population of chromosomes
For each generation
    {
        Evaluate fitness of each chromosome
        Select N/2 parent pairs
          For each parent pair
            {
              randomly generate a crossover point
                  For each gene
                    {
                      apply crossover and mutation operators
                    }
            }
    }
Return fittest chromosome found after G generations
```

Fig. 3.1. *High level description of GA. Note that population size 'N' is even.*

The following steps outline the various operations within a GA.

Step 1

Initial parameters are set for the GA such as population size, maximum number of generations that will evolve, crossover rate and mutation rate.

Step 2

The GA randomly generates a population of *N* chromosomes.

Step 3

The fitness of each chromosome is calculated using the fitness function.

Step 4

The next generation of *N* chromosomes is created from the current generation. This is achieved by performing the following processes *N* /2 times.

i. Probabilistically select (with replacement) 2 chromosomes from the current generation to act as parents to produce 2 child chromosomes. The probability of chromosome being selected is proportional to its fitness. The greater the fitness the greater the likelihood of selection.
ii. With a given probability, the crossover probability p_c, the two parent chromosomes are crossed over at a randomly selected loci (position) to form two child chromosomes.
iii. Mutate each gene of the child chromosomes with a given probability, p_m , the mutation rate.
iv. Place the child chromosomes in the new population.

Step 5

Has the termination criterion been met? For example, have *G* generations evolved? If 'no' return to step 3.

Step 6

Terminate the GA and present the fittest chromosome(s).

3.4.2 The Simple GA - An Example

As an example, Holland's original 'Simple GA' (SGA) [4] will be applied to a function optimisation problem which has the objective or fitness function ,

$$f(x) = 2x \ , \quad \text{where} \, x \in \mathrm{N} \ \text{and} \, 0 \leq x \leq 31$$

The Coding Scheme

Fundamental to the success of a GA in solving a particular problem is the suitable encoding of candidate solutions. The coding scheme must map the parameters of the problem to a unique binary or real-valued string. The fixed-length binary string is the most common encoding scheme, used in the majority of GA applications. Much of Holland's pioneering work concentrated on such schemes and as a result the GA community followed suit. The strings themselves are the chromosomes and the bits of the strings are the genes. The alleles (values) of each gene depend on the type of coding scheme chosen. For a binary scheme the genes can take values of either 1 or 0.

The above function is defined for integers in a finite range so a convenient encoding scheme would be to use binary chromosomes of length 5. The chromosomes are decoded into integers using the usual binary/decimal conversion process, with the left most bit being the most significant. For example the chromosome,

[0 0 1 0 1]

represents the integer 5. Table 3.1 below illustrates a selection of chromosomes for the present problem and their associated fitness values.

Chromosome	Integer x	$f(x)$
00000	0	0
00001	1	2
00110	6	12
01010	10	20
11000	24	48

Table 3.1. *A selection of binary encoded chromosomes of length 5 together with their associated fitness values.*

Now that an encoding scheme has been established the genetic operators; selection, crossover and mutation must be chosen.

Selection

In order for the generations to evolve, individuals from the current population of chromosomes mate to produce off-spring for the next generation. The process of choosing which individuals will mate is known as selection. Selection mimics Darwin's phenomenon of natural selection which states that the greater the fitness of an individual, the greater the probability of it being selected as a mate. The SGA uses a scheme known as *roulette wheel selection.*

Roulette wheel selection is an example of a fitness proportionate selection scheme, in which the expected number of times a chromosome will be selected from a set of *n* chromosomes in *n* trials, is that chromosomes fitness divided by the average fitness of the population.

To implement this scheme each chromosome is allocated a slot of a notional roulette wheel, the size of each slot being proportional to the chromosome's fitness. The spin of the wheel is simulated and the chromosome beneath the wheel's marker is selected as a parent. As an example consider the population of chromosomes given in Table 3.2.

No.	Chromosome	Integer	Fitness = 2 x Integer
1	00001	1	2
2	00011	3	6
3	01000	8	16
4	01001	9	18
			$\sum = 42$

Table 3.2. *Population of four binary chromosomes representing unsigned integers and their associated fitness values.*

The roulette wheel corresponding to this population is given in Figure 3.2. The sum of the fitnesses of the population can be calculated and is 42 in the present case. The spin of the wheel can therefore be simulated by randomly generating a number between 0 and 42 and the chromosome which occupies the slot containing that number will be selected. If, for example, the number 20 is generated chromosome 3 will be selected.

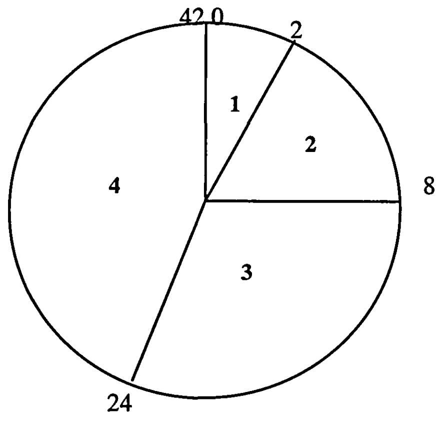


Figure 3.2. *Roulette wheel corresponding to population of chromosomes in Table 3.2*

Crossover

Once two parents have been selected, it is necessary for them to exchange genetic material. Crossover describes this process of combining two parent chromosomes to produce child chromosomes. It is widely acknowledged that crossover is the main search mechanism within a GA. By combining parent chromosomes the GA often produces new chromosomes and as a result new areas of the search space are explored.

In the natural world, two parent chromosomes do not always exchange genetic material (for example one of the parents may be infertile or two parents do not mate). This phenomenon is carried over into GAs and parent chromosomes exchange genes with a given probability known as the crossover probability, p_c or crossover rate. Since crossover enables quick exploration of the search space, crossover should take place with probability greater than 0.5. The SGA uses single-point crossover and is a good starting point when applying a GA to a new problem. Consider two chromosomes of length *l*. A crossover point between the first and last gene is randomly generated, creating a head and tail segment in each chromosome. By exchanging the tail segments of the chromosomes two child chromosomes are created. Figure 3.3 illustrates two parent chromosomes exchanging genetic material about a crossover point located after the third gene (from the left).

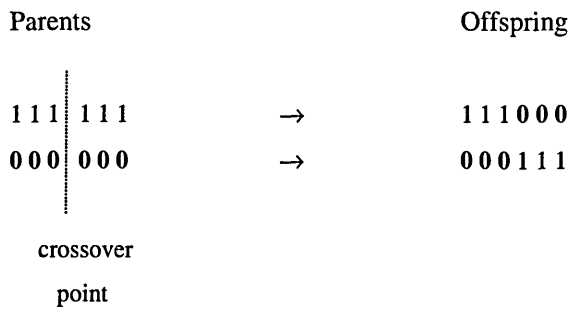


Figure 3.3 *Single point crossover of two binary chromosomes about a crossover point after the third gene.*

Mutation

Mutation is the sporadic, random alteration of genes. For binary encoded chromosomes mutation results in the genes flipping their value from 1 to 0 or vice versa. Mutation occurs with a given probability, p_m known as the mutation probability or mutation rate. In nature mutation is a relatively rare occurrence and this fact is reflected in GAs, where p_m is typically less than 5%.

Mutation is applied to the child chromosomes once they have been created through crossover. Figure 3.4(a) shows a binary chromosome before mutation and 3.4(b) gives the same chromosome once mutation has been applied to the third gene (from the left).

1 0 0 1 0 1	1 0 1 1 0 1
(a)	(b)

Figure 3.4(a) *Chromosome before mutation.* **(b)** *Chromosome after mutation of the third gene.*

Now we have discussed the main component of the SGA a dry run will be performed, evolving 2 generations of chromosomes from an initial, randomly generated population. Each population will contain 4 chromosomes and the crossover rate and mutation rate are set at 100% and 5% respectively. Each parent pair will produce 2 child chromosomes.

Generation 0				**Mating**	**Process**	**Generation 1**			
No	Chromosome	x	Fitness $f(x)$	Parents	crossover point	No	Chromosome	x	Fitness $f(x)$
1	01001	9	18	[3,2] 00110,10001	1	1	10110	22	44
2	10001	17	34	[3,2] 00110,10001	1	2	00001	1	2
3	00110	6	12	[4,2] 01101,10001	4	3	01101	13	26
4	01101	13	26	[4,2] 01101,10001	4	4	10001	17	34
	$\Sigma f(x)$		= 90				$\Sigma f(x)$		= 106
	Max. fitness		= 34				Max. fitness		= 44
	Average fitness		= 22.5				Average fitness		= 26.5

Table 3.3. *Evolution of Generation 1 from the Randomly Created Generation 0.*

As can be seen from Table 3.3, Generation 0 contains some fairly mediocre solutions to the optimisation problem. This is to be expected given that the chromosomes were randomly generated. The maximum fitness in the population is 34, corresponding to $x = 17$ and the average fitness is 22.5. By generation 1, the maximum fitness has been raised to 44 and the average to 26.5. By Generation 2, given in Table 3.4, the GA seems to be making good progress. The average fitness has been raised to 35.5 and the fittest chromosome has a fitness of 58. The mating of chromosome 3 and 4 has resulted in the fittest chromosome and the weakest. This is a common feature within genetic algorithms. In the evolution of Generation 3 from Generation 2 the chances are that the weakest individual will not be selected due to the overwhelming strength of the other chromosomes in the generation. Just as in nature, the genetic material contained in this individual will die out. Another feature to be noted in generation 2 is the mutation of the fourth gene (from the left) of chromosome 4.

Generation 1				Mating Process		Generation 2			
No	Chromosome	x	Fitness $f(x)$	Parents	crossover point	No	Chromosome	x	Fitness $f(x)$
1	10110	22	44	[3,4] 01101,10001	1	1	11101	29	58
2	00001	1	2	[3,4] 01101,10001	1	2	00001	1	2
3	01101	13	26	[4,1] 10001,10110	4	3	10111	23	46
4	10001	17	34	[4,1] 10001,10110	4	4	10010	18	36
	$\Sigma f(x)$		$=106$				$\Sigma f(x)$		$=142$
	Max. fitness		$=44$				Max. fitness		$=58$
	Average fitness		$=26.5$				Average fitness		$=35.5$

Table 3.4. *Evolution of Generation 2 from Generation 1.*

From these tables, it can be seen that the GA is definitely evolving stronger chromosomes from an initial population of weaker ones. Since the population size in each generation is small, the expected values of selection, crossover and mutation are far from the actual ones. This dry run however, certainly illustrates the mechanics and search strategy adopted by a typical GA.

Even from the small example presented above, it can be seen that through a number of random and probabilistic operations, a GA seems to be a good optimisation tool. Starting from a randomly generated population, the GA is able to explore a search space and home in on promising regions. All this is achieved by sampling just a few points in the space. Furthermore, the operations taking place, such as crossover and mutation are very simple compared to other optimisation techniques that require for example, derivatives of the objective function to be calculated. The above example has certainly illustrated *what* a GA a does but it has provided little insight into *how* and *why* a GA works so well. These issues will now be addresses in the following section.

3.5 The Mathematical Foundations of Genetic Algorithms

It was John Holland who, during the 1970s, laid down the mathematical foundations of GAs [1]. He introduced the idea that GAs process *schemata* rather than individual chromosomes and therefore are able to sample large regions of the search space. At first glance a population of n chromosomes would seem to sample, at most, n points in the search space. However, the schemata contained in each individual enable the GA to actually process many more regions through what is known as *implicit parallelism.*

Holland's Schema Theorem [1][5] describes the way in which schemata are processed under the processes of selection, crossover and mutation. Throughout he assumes binary encoding and single-point crossover, although the theory has been extended to accommodate other encodings and crossover operators [6]. The theorem describes how instances of schemata either increase or decrease from one generation to another, given their form and average fitness.

All of these ideas will now be discussed in more detail in the context of finite-length, binary chromosomes.

3.5.1 Schemata (Similarity Templates)

A schema is a pattern matching device which gives rise to a compact notation for describing similarities between finite-length strings (chromosomes). Schemata are described using the symbols {1, 0, *}, where 1 and 0 are so-called fixed bits and the asterisk denotes the "don't care' symbol (it can represent either 1 or 0). Thus, for example the schema H = 11** contains two fixed bits (both 1's) and represents four bit strings, 1100, 1101, 1110 and 1111. The number of fixed bits in a schema H is known as its order and is denoted by $o(H)$. Another feature of schemata is their defining length $d(H)$. Given a schema $H = s_1 s_2 \dots s_l$ suppose that the first fixed bit (from the left) is s_i and the final fixed bit is s_{i+r}, where $i \geq 1$, $r \geq 0$ and $i + r \leq l$. Then the defining length $d(H)$ of H is r, the distance between the two outermost fixed bits in a schema. To illustrate these ideas, for the schema H = *1*00*0, $o(H)$ = 4 and $d(H)$ = 5.

Given a string length l, or equivalently a chromosome with l genes, the total number of different possible bit strings is 2^l. The total number of different possible schemata in this bit string is 3^l, since each element of a schema can take the value 1, 0 or x. These results are important since they give an indication of the amount of information a GA actually processes.

So why are schemata important? At first glance a GA processes a given number of individuals in a generation and based on their relative fitness, selects and mates the fittest to (hopefully) produce even fitter individuals. However, by considering only the chromosomes and their fitness, the search process is based on only a limited amount of information. By investigating features of strings that yield high fitness, one can incorporate much more information into the search process. Consider Table 3.5 below which gives four binary strings of length 5 representing unsigned integers and their associated fitnesses under the fitness function $f(x) = x^2$.

Chromosome	Integer	Fitness
00010	2	4
10000	16	256
01001	9	81
10010	18	324

Table 3.5. *A selection of binary encoded chromosomes representing unsigned integers and their associated fitness.*

Examination of the above table helps one examine features amongst the chromosomes which lead to high fitness values. So, what distinguishes the chromosomes of high fitness? It seems clear that having a 1 in the first bit position gives rise to strong chromosomes and the schemata 1**** and 100*0 represent this feature (hence the alternative name for a schema, 'similarity template').

The amount of additional information a GA incorporates into the search process is equal to the number of different schemata present in a population. For a population of n chromosomes of length l the number of actual schemata present is between 2^l and $n.2^l$, since each fixed bit may be replaced by a * to form schemata. For a given population, Holland showed (taking into account crossover and mutation) that the number of schemata actually processed by the GA is approximately equal to n^3. This is an important result and is known as implicit parallelism. Although a GA explicitly calculates the fitnesses of chromosomes in a population, it also implicitly calculates the average fitness of a large number of schemata present in a given population. No additional processing time or computer memory is required for this implicit processing, it is simply part of the normal operations within a GA.

3.5.2 The Schema Theorem

The Schema Theorem describes the dynamics of a GA in terms of the increase and decrease of schema instances as populations evolve. Given the number of instances of schema H in a population at time t the schema theorem approximates the expected number of instances of the schema at time t+1, given the effects of selection (fitness proportionate), crossover (single point) and mutation on schema survival.

Suppose at time t there are m instances of schema H where,

$$m = m(H,t) \ .$$

The effects of selection are such that the probability, p_i of a string being selected as a mate is given by,

$$p_i = \frac{f_i}{\sum f_j}$$

where f_i is the fitness of the string and $\sum f_j$ is the sum of fitnesses of all members of a population. Similarly, the probability of a schema being selected as mate is given by, $m(H,t).\frac{f(H)}{\sum f_j}$ where $f(H)$ is the average fitness of the strings containing schema H. At time t+1 therefore the expected number of instances of schema H in a population of n strings is given by,

$$m(H,t+1) = n.m(H,t)\frac{f(H)}{\sum f_j} \qquad (1)$$

The average fitness of the entire population may be written $\overline{f} = \frac{\sum f_j}{n}$ therefore $\sum f_j = n.\overline{f}$. Substituting this into (1) gives

$$m(H,t+1) = m(H,t).\frac{f(H)}{\overline{f}} \tag{2}$$

From the above equation it may be seen that instances of a particular schema grow from one generation to another as a ratio of average schema fitness to average fitness of a population. In other words the number of schemata of above average fitness is likely to increase in the next generation. By further investigation of equation (2) it is possible to gain an insight to the mathematical form of this schema growth/decay.

Assume that a schema H is of above average fitness by an amount $c\overline{f}$, where c is a constant. The average fitness of this schema is then $\overline{f} + c\overline{f}$. Substituting this into (2) we obtain,

$$m(H,t+1) = m(H,t).\frac{\overline{f}+c\overline{f}}{\overline{f}} = m(H,t).(1+c)$$

Starting at time $t = 0$ and assuming c to be fixed we obtain

$$m(H,t+1) = m(H,0).(1+c)^t$$

which is the discrete analogue of exponential form. It may be concluded therefore that selection allocates an exponentially increasing (decreasing) number of instances of an above (below) average schema into successive generations.

However, the effects of crossover have yet to be considered. In single point crossover, a random crossover point is selected between 1 and $l-1$. A schema of large defining length is more likely to be disrupted or destroyed than a schema of shorter defining length. For example, in the schema *1***0, of defining length 4 and total length 6, there are 4 crossover points which potentially result in schema destruction. In contrast the schema *10*** has only a single crossover point which potentially destroys it. In calculating the probability of schema destruction not only is it necessary to consider the defining length but also the number of destructive mates. The schema *1***0 has four possible mates i.e. *1***1, *0***0, *0***1 and itself. Half of all these possible matings preserve the original schema. Therefore, the probability of schema destruction $\leq \frac{d(H)}{l-1}.(\text{probability of destructive matings})$.

For the schema *1***0, probability of schema destruction $\leq \frac{4}{5}.\left(1-\frac{1}{2}\right)$

$$\leq \frac{2}{5}$$

and for the schema *10***, the probability of schema destruction $\leq \frac{1}{10}$. This probability is an upper bound since probabilistic selection introduces bias towards strong schemata and therefore not all matings are equally probable.

However, for simplicity the schema theorem treats the ratio $\frac{d(H)}{l-1}$ as the upper bound for schema destruction, the inequality masking the effects of different mates and the bias introduced through selection. Furthermore, given that crossover takes place with a probability, p_c, the probability of schema destruction is, $\leq p_c \frac{d(H)}{l-1}$. From this result, a lower bound on the probability of schema survival, p_s is given by,

$$p_s \geq 1 - p_c \frac{d(H)}{l-1} \qquad (3)$$

Hence adjusting equation (2) to take account of the effects of crossover gives,

$$m(H,t+1) \geq m(H,t).\frac{f(H)}{\overline{f}}\left[1 - p_c \frac{d(H)}{l-1}\right] \qquad (4)$$

The final operator that needs to be considered is mutation. Mutation takes place with a given probability, p_m. For a schema of order *o(H)*, the probability of schema survival under mutation is given by,

$$(1-p_m)^{o(H)} \approx 1 - o(H).p_m \qquad (5)$$

if $p_m << 1$. Adjustment of equation (4) to incorporate the effects of mutation gives,

$$m(H,t+1) \geq m(H,t).\frac{f(H)}{\overline{f}}\left[1 - p_c \frac{d(H)}{l-1}\right]\left[1 - o(H)p_m\right]$$

which, by neglecting small cross products approximates to,

$$m(H,t+1) \geq m(H,t).\frac{f(H)}{\overline{f}}\left[1 - p_c \frac{d(H)}{l-1} - o(H)p_m\right] \qquad (6)$$

Equation (6) is the mathematical formulation of Holland's Schema Theorem which states,

> "short, low order schema of above average fitness receive exponentially more instances in subsequent generations than those with below average fitness".

From his schema analysis Holland has shown that selection seems to focus the GA on areas of the search space that have above average fitness. These regions are defined by the above average schemata that increase exponentially in subsequent generations. Crossover combines these highly fit schemata in the hope of homing-in on even fitter individuals. In other words, this process is exploiting existing information. Mutation on the other hand ensures that the GA keeps an open mind about regions of the search space that have yet to be explored. It acts as an insurance policy since not all schemata can be produced through crossover alone. Mutation therefore aids in the exploration of the search space and helps preserve genetic diversity.

Adaptation by any organism to an unpredictable environment is seen to create a tension between exploration and exploitation. This fine balance occurs since one takes away from the other. Exploration of untried schemata for instance, takes away from the exploitation of tried and tested, highly fit schemata. The search strategy must prevent premature convergence on sub-optimal solutions by continually exploring uncharted regions of the search space. However, it must also incorporate and utilise, or exploit, existing information. Holland proposed his original GA as a strategy for achieving an optimal balance between exploitation and exploration by allocating exponentially more instances of fit schemata to subsequent generations relative to instances of the weaker ones. But why is this allocation strategy a good one? This question leads to an important problem in statistical decision theory, the Two-Armed Bandit problem. Its solution indicates why the above allocation strategy, viz. allocating exponentially more instances of above average fitness schemata to successive generations, is successful and was used by Holland to justify his arguments.

3.5.3 The Two-Armed and K-Armed Bandit Problem.

The Two Armed Bandit problem is a convenient means of modelling the trade-off between exploration and exploitation. Analysis of the problem helps in deciding the allocation of resources in the face of uncertainty. The problem is as follows.

Figure 3.5 below illustrates a two-armed slot machine that is a popular gambling device (the conventional machines often have one arm and are called ‘one arm bandits’). The gambler can play either arm 1 or arm 2 which are independent of one another. The arms are labeled A_1 and A_2 and have a mean payout per trial of μ_1 and μ_2 with variances of σ_1^2 and σ_2^2 respectively. The gambler has no prior knowledge of these means or variances, he can estimate them through experimentation alone. The goal of the gambler is of course to maximise his winnings during N trials, by first gaining knowledge of which arm seems to payout the most (exploration) and then to *exploit* this knowledge to maximise the payout. What allocation strategy should the gambler adopt?

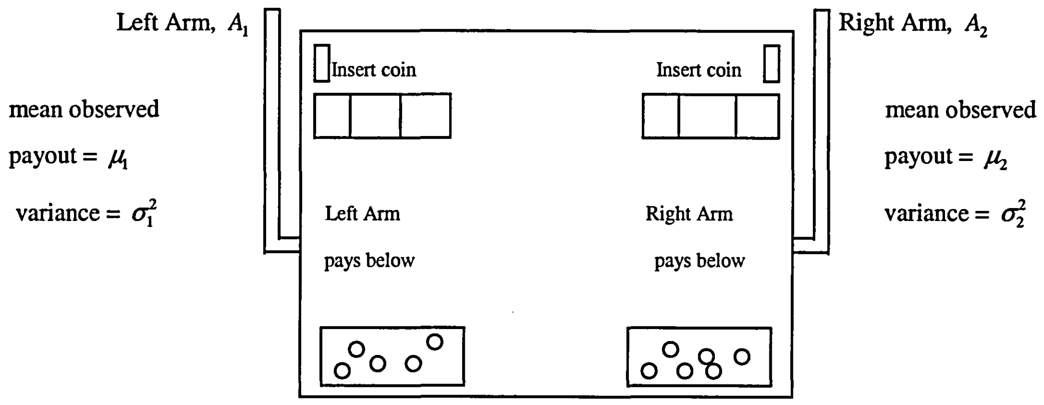


Figure 3.5. *A two armed bandit*

By collecting information about each arm, the gambler must decide which arm seems to pay most. The dilemma is how to search (exploration) for the highest paying arm while simultaneously using that information (exploitation). Although the precise mathematical details of the solution [1] lies outside the present discussion, the following gives an adequate overview.

Suppose the gambler has N coins and therefore a total of N trials to allocate between the two arms. The first step is to allocate an equal number of trials n ($2n < N$) to each arm. After examining the respective payouts the remaining (N - $2n$) trials are allocated to the observed better arm Let A_1 be the actual better arm and A_2 be the actual worse arm. Also, let $A_l(N, N-n)$ denote the arm with the observed higher payout and $A_h(N,n)$ denote the arm with the observed lower payout.

What is the value of $n = n^*$ that maximises expected payouts or minimises the loss? There are two potential sources of loss for the gambler.

1. The observed worse arm $A_l(N,n)$ is actually the better arm, A_1. Therefore the gambler has lost expected profits on $(N-n)$ trials that were wrongly allocated to $A_l(N, N-n)$. In this case the loss is given by

$$\text{Loss 1} = (N-n).(\mu_1 - \mu_2)$$

2. The observed worse arm $A_l(N,n)$ is actually the worse arm. Therefore the gambler has lost expected profits on the allocation of n trials, during the exploration phase and the loss is given by

$$\text{Loss 2} = n.(\mu_1 - \mu_2)$$

If q is the probability that the observed worse arm, $A_l(N,n)$ is actually the better arm, A_1 i.e.

$$q = \mathbf{Pr}(A_l(N,n) = A_1)$$

then the losses, $L(N-n,n)$ over n trials are,

$$L(N-n,n) = q.(N-n).(\mu_1 - \mu_2) + (1-q)(\mu_1 - \mu_2).n \tag{7}$$

where q depends on n^2 . By taking the derivative of $L(N-n,n)$ with respect to n we can find an expression for $n = n^*$ that minimises equation (7) i.e. the loss. The exact details of this calculation will be omitted as they are not relevant to the discussion. The result of the calculation is ,

$$n^* \approx b^2 \ln\left[\frac{N^2}{8\pi b^4 \ln N^2}\right] \tag{8}$$

where b is a constant. Equation (8) implies that,

$$\exp\left(\frac{n^*}{b^2}\right) \approx \frac{N^2}{8\pi b^4 \ln N^2} \tag{9}$$

Since $\ln N^2 = 2\ln N$ and $\exp\left(\frac{n^*}{b^2}\right) = \exp\left[2\left(\frac{n^*}{2b^2}\right)\right] = \left[\exp\left(\frac{n^*}{2b^2}\right)\right]^2$ by setting

$2b^2 = \left(\frac{1}{c^2}\right)$ equation (9) can be re-written as,

$$c^2 N \approx 2\sqrt{\pi \ln N}\exp\left(cn^*\right) \tag{10}$$

From equation (10) since, $\dfrac{N}{\sqrt{\ln N}} \to \infty$ as $N \to \infty$ then,

$$\exp\left(cn^*\right) \approx \frac{c^2 N}{2\sqrt{\pi \ln N}} \to \infty \text{ as } N \to \infty$$

i.e. $n^* \to \infty$ as $N \to \infty$. Also from (10) as N increases, $c^2 N >> \exp\left(c^2 n^*\right)$ so,

$$\frac{N}{n^*} >> \frac{\exp\left(c^2 n^*\right)}{c^2 n^*} \to \infty \quad \text{as } N \text{ (and hence also } n) \to \infty \tag{11}$$

Since $\dfrac{N-n^*}{n*} = \dfrac{N}{n^*} - 1 \approx \dfrac{N}{n^*}$ for large N equation (11) implies,

$$\frac{N-n^*}{n^*} >> \frac{\exp\left(c^2 n^*\right)}{c^2 n^*} \tag{12}$$

The gambler's optimal strategy for minimising his/her gambling loss can be seen from equation (12). This equation implies that for large N, as the total number N of trials increases, the number of trials allocated to the better arm, $\left(N - n^*\right)$ grows more than exponentially relative to the number of trials

[2] As the number of trials n increase, the probability q , that the observed worse arm is actually the better arm, decreases.

allocated to the worse arm n^*. This interpretation reveals more than a coincidental similarity between the gamblers optimal strategy and a GAs search strategy outlined by the Schema Theorem.

3.5.4 The Schema Theorem and the Two-Armed Bandit Problem

The Two-Armed Bandit problem is a convenient model of the *exploration versus exploitation* dilemma faced by adaptive systems such as GAs. Its solution provides a strategy for allocating resources in the face of uncertainty. Holland's Schema Theorem suggest that a GA allocates schemata from generation to generation in a manner very similar to the optimal strategy given by the solution of the Two-Armed Bandit problem. Recalling equation (3) we can see the exponentially increasing number of trials given to above average fitness schemata relative to the weaker ones. Holland argued that a GA implements this optimal search strategy through implicit parallelism, where the GA is actually processing n^3 schemata in a population of n chromosomes. So returning to the question raised towards the end of section 3.4 as to whether the allocation strategy of a GA is a good one, the solution of the Two-Armed Bandit problem certainly suggests that it is.

Originally, Holland suggested that a GA adopts the optimal allocation strategy and seems to play a 3^l-armed bandit with all 3^l possible schemata in a population competing as arms. This original theory has been widely debated and modified. The problem is that unlike the Two-Armed Bandit, the different schemata (or arms) in a GA interact. It is now thought [8] that the GA is actually playing a 2^k - armed bandit in each order-*k schema partition.* A schema partition is defined as a division of the search space into 2^k competing schemata. For example the partition **d*** consists of the two schemata *1** and *0** and the partition **dd** contains *00*, *01*, *10* and *11*. It is the schemata within each partition that are competing with each other as in a 2^k-armed bandit problem. Popular understanding suggests that the best observed schema within each partition receive exponentially more trials than the second and so on. However, the *k*-armed bandit strategy will only be adopted by partitions in which the competing schemata have relatively uniform fitnesses (i.e. low fitness variance).

As mentioned earlier, the topic of GAs is an active and continually evolving field of research. There is much debate as to how and why GAs work [7,8] and the Schema Theorem coupled with the *k*-armed bandit analogy provide some insight. In light of the extent of the literature and the controversy surrounding GA theory the reader at this stage is directed to the references for further information.

3.6 GA Implementation Issues

The GA outlined in Section 3.4 is Holland's original Simple GA. It uses a fixed-length binary encoding scheme, fitness proportionate, roulette wheel selection, single point crossover and the standard mutation operator. Much of the original work on GAs and GA theory was based on this implementation. However, as the GA community grew, so did the number of different implementations of the algorithm. Take for example, the issue of encoding schemes. As GAs were applied across a growing number of problems, it became apparent that alternative representations to the binary coding schemes were required. For example, many real world problems required the optimisation of real valued parameters. A binary scheme can become awkward when trying to represent multiple real values to a reasonable number of decimal places. It was found that chromosomes containing real number worked far better. The encoding issue has just been taken as a brief example. This and other issues will now be discussed in more detail.

3.6.1 Encoding Schemes

As mentioned earlier, the success of a GA in solving a particular problem is largely dependent on devising a suitable scheme for encoding candidate solutions. The coding scheme must map the parameters of the problem to a unique binary or real-valued string. Binary encoding schemes are by far the most popular, largely due to the early GA pioneers.

A significant problem with binary encodings is that they do not provide the flexibility to solve a large class of problems that have multiple, real-valued parameters. A common method for encoding some of these problems is to map the parameter values to integers, which can then be encoded as in Table 3.1 above. The binary strings for each variable are then concatenated to produce the chromosome. As an example consider a three variable optimisation problem in which each variable can take values in the range 0 to 2.55. By multiplying the value of each variable by 100, we can represent each variable, to an accuracy of 2 decimal places, by an 8 bit binary string. Concatenation of three, 8 bit strings will lead to each candidate solution being represented by a 24 bit binary string , as shown in Figure 3.6a.

However, this technique becomes unwieldy for larger problems, such as evolving weights for neural networks, and for problems requiring greater accuracy. In such cases, real valued encoding schemes have been adopted [9],[10] and the chromosomes are collections of real numbers as given in Figure 3.6b.

[00010001 00011110 11100111]	[12256.00, 23.45, 0.0023, 100.50]
(a)	(b)

Figure 3.6. (a) *Three-variable binary chromosome representing the values 0.17, 0.30, 2.31 respectively.* **(b)** *Four variable, real valued chromosome representing the numbers 12256.00, 23.45, 0.0023 and 100.50.*

Initial reluctance by the GA community to adopt real valued schemes was largely due to Holland's Schema Theorem, which suggested that binary encodings displayed far better performance than any alternative scheme. However, over recent years there has not only been controversy over the schema theorem [11] but also favourable comparative evidence supporting real-valued schemes over binary ones in certain applications [12], [13]. The key phrase in the previous sentence is 'certain applications'. In some cases binary encodings will perform better than other schemes and in others real valued schemes will work best. Currently there are no hard and fast rules which govern the choice of encoding scheme and the performance of the different schemes depends solely on the problem.

3.6.2 The Fitness Function

At the heart of all GAs is the fitness function and its appropriate formulation is a key factor in the successful application of the algorithm. The fitness function rates the competence of each chromosome as a solution to the problem.

The fitness function is often a mathematical formulation of the problem to be solved and must reflect the criteria by which a candidate solution is judged. In the case of function optimisation problems, the fitness function is often (but not necessarily) identical to the objective function. As an example, consider the optimisation of the function,

$$f(x) = x^2$$

in the interval [0, 15]. If chromosomes are encoded as binary strings which map to unsigned integers, the fitness of each solution is the value of $f(x)$ where x is the integer represented by the given chromosome. Table 3.6 displays a selection of four gene chromosomes and their corresponding fitness values.

For combinational optimisation problems such as the Travelling Salesperson Problem, an obvious fitness function (but not the only one) evaluates the total distance travelled given the order in which each city is visited. The two classes of problems just mentioned reduce fitness evaluations to a single figure of merit (value of objective function, distance travelled by salesperson). However this is not always possible in situations such as engineering design where there are multiple, sometimes conflicting, criteria. It would be unwise in these cases to try and reduce the ideas of optimality to a single figure of merit and we need to evaluate solutions with respect to each of the different criteria.

Chromosome	Integer	Fitness Value
0001	1	1
0011	3	9
1010	10	100

Table 3.6. *The fitness of a selection of 4 bit chromosomes evaluated using the fitness function,* $f(x) = x^2$

The concept of Pareto Optimality [4] aids in the evaluation of solutions to multi-objective problems [14]. To illustrate the idea an example taken from [4] will be discussed. Consider a widget manufacturer who wishes to minimise both widget cost and on-the-job accidents. There are five possible scenarios, A, B, C, D and E in which the plant may be run which result in the following cost accident count.

A = (2, 10)	(cost, accident count)
B = (4, 6)	
C = (8, 4)	
D = (9, 5)	
E = (7, 8)	

Which scenario should the manufacturer choose? By plotting scenarios A to E on an Accident versus Cost graph, given in Figure 3.7, one is able to assess the relative merits of each scenario.

To minimise both cost and accident count, we require scenarios that lie as near to the origin of the graph as possible. Therefore scenarios A, B and C are good choices. In all three cases none of the three points is best along both axes so it is a matter of assessing trade-offs as to which scenario is better. The profit hungry manufacturer will select scenario A whereas others may prioritise safety and select scenario C.

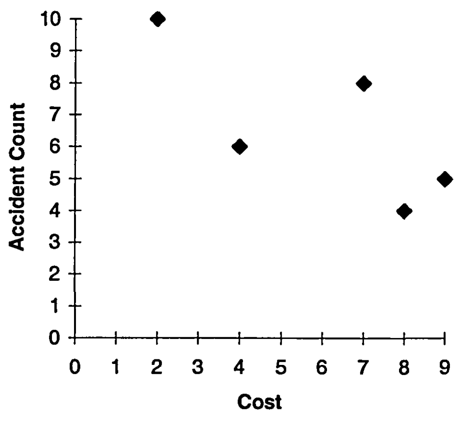


Figure 3.7 *Widget Cost vs. Accident Count scatter graph*

The concept of pareto optimality does not provide a single 'best' answer, instead it provides a set of possibilities, known as the Pareto Optimal (P-Optimal) Set. The final choice of solution is reserved for the human decision maker who is able to weigh the pros and cons of each solution in the P-Optimal set.

To whatever class of problem a GA is applied, correct formulation of the fitness function is of paramount importance. The accurate assessment of a potential solution is only as good as the quality of the fitness function. The assessment of potential aircraft engine designs, for example, will only be as good as the mathematical model in which they are simulated. An inaccurate fitness function may lead a GA away from the *real* optimal solution. Therefore much time and effort should be devoted to developing these fitness functions as they play a large part in the ultimate success of this optimisation technique.

3.6.3 Selection

The purpose of selection within a GA is to increase the average fitness of populations of chromosomes as successive generation arise. By favouring fit individuals in the current generation to act as parents, it is hoped that subsequent generations will contain chromosomes of even higher fitness. Selection can therefore be seen to concentrate the search procedure on promising regions of the search space.

Over recent years, selection has become the subject of much research and a number of comparative studies have emerged [15][16][17]. An important property of selection schemes common in much of the literature is *selection pressure* [17][18][19]. Although there are several definitions it broadly describes the degree to which fitter individuals are favoured over the weaker ones for selection. Too high selection pressure will result in strong individuals in early generations dominating the mating process and they will soon take over subsequent populations. This is not to be encouraged as although these individuals will be relatively fit in early generations they will very often be of sub optimal global fitness. In the presence of too high selection pressure the GA will therefore be primarily involved in the exploitation of sub-optimal solutions, resulting in premature convergence. Too weak selection pressure on the other hand will significantly slow down evolution as it will favour as many weak solutions as it does strong solutions. In this situation, the GA will primarily be involved in exploration of the search space and in extreme cases may even resemble a random walk strategy among the initial population.

It is apparent from the above that selection is a driving force within a GA and contributes significantly to the exploration/exploitation balance. The search process must be directed towards promising regions of the search space, this is obvious. But it is also important not to completely neglect relatively weak individuals as they may contain valuable genetic material, which when combined with fitter individuals will produce the super-fit solutions that are being sought. Good selection procedures must therefore address the problem of achieving a balanced search strategy.

The research community has devised a number of other properties that aid the comparative study of selection schemes, selection pressure being just one. In what follows a number of the most popular schemes will be described together with a high-level discussion of their relative merits. For detailed analyses of these and other less popular schemes the reader is directed toward the references.

3.6.3.1 Fitness Proportionate Schemes

Fitness proportionate selection (FPS) was used in Holland's original GA and in the formulation of the Schema Theorem. In this scheme, the expected number of times a chromosome will be selected as a parent is given by the individual's fitness $f(i)$ divided by the average fitness of the population, $\bar{f}_j$, i.e.

$$\text{Expected no. of reproduction trials for individual } i = n.\frac{f(i)}{\bar{f}_j}$$

where n is the number of trials.

This scheme is easily implemented using the roulette wheel selection technique as described earlier. Each individual is allocated a slot on a roulette wheel proportional to its fitness. A spin of the wheel is then simulated and the individual beneath the pointer is selected as a mate. A problem with this method is that for relatively small populations and a small number of trials the actual number of times an individual is selected is far from the statistical expected value. Because each parent is selected during an independent trial, it is statistically possible that the best individuals are never selected. To overcome this problem, a slight variation of the above method, known as stochastic universal sampling (SUS), has been devised [20]. In this method, if it is necessary to select N parents, the roulette wheel has N equally spaced markers and is spun only once. The individuals lying beneath the markers are selected. The technique is not only simple but eliminates the sampling errors associated with the original roulette wheel scheme.

All fitness proportionate schemes suffer from a common problem however, viz. that the rate of evolution depends on the variance of the fitnesses in the population (the selection pressure therefore is not constant throughout a run). During early generations of the GA, the fitness variance in the populations will be high, that is there will be a small number of relatively super-fit chromosomes. Under fitness proportionate selection these individuals and their off-spring will dominate the evolutionary process, possibly resulting in the premature convergence of the GA on sub-optimal solutions. In later generations when the fitness variance is low evolution will cease since very few chromosomes stand-out for selection.

In an attempt to overcome these problems GA researchers have devised several scaling methods for mapping absolute fitness values to expected fitness values. Fitness Windowing [17][21], Linear Scaling and Sigma Scaling [4] are examples of scaling methods. Although these schemes go some way to removing the dependency of selection pressure on fitness variance, a significant problem still remains. The presence of exceptionally fit/unfit individuals will result in performance degradation since premature convergence or a halt in evolution is likely.

Because of the problems associated with FPS schemes their use is not recommended by many GA practitioners. Other selection schemes have been produced to move away from the dependency of selection pressure on absolute fitnesses.

3.6.3.2 Rank Selection

A selection scheme based on rank was first proposed by Baker [22]. In this scheme, the chromosomes in a population are ranked according to their fitness and the expected number of reproductive trials for each chromosome is proportional to its rank. Absolute fitnesses are masked by rank and therefore premature convergence due to sub-optimal individuals can be avoided. Two main ranking schemes [17] are currently in common use.

Linear Ranking

The fittest individual (which has rank 1) in a population is given a fitness, s where $1 \le s \le 2$ and the weakest (which has rank N) is given the fitness $2-s$ and intermediate strings are allocated fitnesses according to,

$$f(i) = s - \frac{2(i-1)(s-1)}{N-1}$$

where i is the rank of each individual. The above formula will give an average fitness of 1 and the fitness values correspond to the expected number of offspring for a given individual of rank i.

The parameter s can be seen to control the selection pressure, the higher the value of s, the greater the selection pressure. If s is set to 2 for example, the worst string has zero probability of being selected. Unfortunately, it is difficult with linear ranking to achieve higher selection pressures while still giving the weaker individuals some chance of selection.

Exponential Ranking

In this scheme, the best string is given a fitness of 1, the second best a fitness of s (typically around 0.99), the third best is allocated a fitness of s^2 and so on. The expected number of trials for each individual is obtained by dividing each fitness value by the population average. The selection pressure for this scheme is proportional to $1-s$.

The difference between exponential and linear ranking is that the former provides a greater chance of selection for the worse individuals in a population (at the expense of the above average ones). For equivalent selection pressures, exponential ranking should therefore result in greater diversity amongst the populations.

The main disadvantage of ranking schemes is the additional computation time required to sort the population and ascribe a rank fitness to each individual. This of course is in addition to computing the 'raw' fitness from the problem specific, fitness function. It is to be noted that for all ranking schemes, once the expected value of each chromosome has been calculated, the parents have to be sampled and this can be done using the SUS method described earlier.

3.6.3.3 Tournament Selection

This selection scheme is relatively simple and computationally efficient. In binary tournaments, two chromosomes are picked randomly from the population and the one with highest fitness is selected. A slight variation on the method is to probabilistically select the fittest chromosome in a tournament. That is, to only select the fitter of the two chromosomes 70% of the time and select the weaker one the remaining 30% of the time. The tournaments can also be extended to randomly picking p chromosomes and selecting the fittest q as mates, where $p > q$. The selection pressure for this scheme can be adjusted by changing the tournament size or win probability.

In practice Tournament Selection suffers from the same sampling errors as roulette wheel selection. Each tournament takes place individually and therefore it is possible that the best individual is never selected. However, its simplicity and the fact it is well suited to parallel implementation ensures that the scheme remains popular.

3.6.3.4 Elitism

Elitism ensures that the fittest individuals in the present generation are retained and copies into the next generation. Chromosomes in one generation can be lost through mating and lack of selection in subsequent generations. As the fittest chromosomes contain good genetic information on which later generations are to be based it is important to keep these individuals. An implementation of elitism may ensure that the fittest 5% of chromosomes are always carried over into the next generation.

3.6.3.5 Brief Comparison of Schemes

Throughout the published literature, it is apparent that 'raw' fitness proportionate schemes are inadequate due to their sensitivity to fitness variance within a given population. Although scaling methods go some way to reducing this dependency, it seems that extraordinary individuals disrupt evolution and introduce premature convergence. Ranking schemes disassociate absolute fitness and the expected number of trials for a given fitness score. Exponential ranking seems to offer the better scheme since the rate of loss of the worst individual is less than for linear ranking. Although this may slow down the rate of convergence, it is hoped that the greater diversity will improve the quality of the final solution. Tournament selection offers a computationally simple scheme without the problems associated with fitness proportionate schemes. However one must bear in mind the sampling errors that are inherent with this technique.

In Goldberg and Deb's [15] comparative studies they conclude that by appropriate adjustment of parameters, many of the selection schemes exhibit the same behaviour. For example a binary tournament is similar to linear ranking with $s = 2$. If tournaments are made stochastic, the results are similar to those for exponential ranking. From these and other results it can be inferred that no single scheme is the absolute best, different fitness functions will favour different schemes.

A criticism of much of the comparative studies is that they do not address the interactions between selection schemes and the crossover and mutation operators. How do low selection pressures interact with high mutation rates for example? They both help to increase the diversity of genetic material. There must be many other questions such as this which have yet to be analysed. It is felt that they must be answered by the GA community with rigorous theoretical analysis so others are able to make a more informed choice of selection scheme.

Given the current theoretical bounds, which scheme should one use when designing a GA? This question can only be answered after experimenting with a number of schemes and by considering as much problem specific knowledge as possible. It is the view of the author that selection pressure is a very important property as is the ability to easily control it. Exponential ranking therefore is the favoured option particularly for non time-sensitive applications.

3.6.4 Crossover

Once two parents have been selected, it is necessary for them to exchange genetic material. Crossover describes this process of combining two parent chromosomes to produce child chromosomes. It is widely acknowledged that crossover is the main search mechanism within a GA. By combining parent chromosomes the GA often produces new chromosomes and as a result new areas of the search space are explored.

In the natural world, two parent chromosomes do not always exchange genetic material (for example one of the parents may be infertile or two parents do not mate). This phenomenon is carried over into GAs and parent chromosomes exchange genes with a given probability known as the crossover probability, p_c or crossover rate. Since crossover enables quick exploration of the search space, crossover should take place with probability greater than 0.5. Just as with selection, there are many different crossover techniques and some of the more popular ones will now be discussed. It must be noted that each crossover scheme has its relative merits and a brief evaluation will follow.

3.6.4.1 Single-Point Crossover

Single point crossover has already been discussed in Section 3.4.2. Genetic information is exchanged about a single crossover point. Although simple, this method has a number of shortcomings. Firstly, it cannot combine all possible schemata that are present in the parents. For example, the 00xxxx1 and xxx11xx cannot be combined to form 00x11x1. Also, schemata of large defining length have a greater probability of destruction than those of short defining length. This could be a significant hurdle to the evolutionary process if for example, the schema 1xxxxx0 was required to form the optimal solution(s). Furthermore, single point crossover suffers from *end-point bias* meaning that the end-points of chromosomes will always be exchanged between parents.

3.6.4.2 Two Point and Multi-Point Crossover

Two point crossover is very similar to the single point case except two points are randomly chosen between which the chromosomes exchange genetic information. Figure 3.8 illustrates two point crossover for chromosomes of length 8.

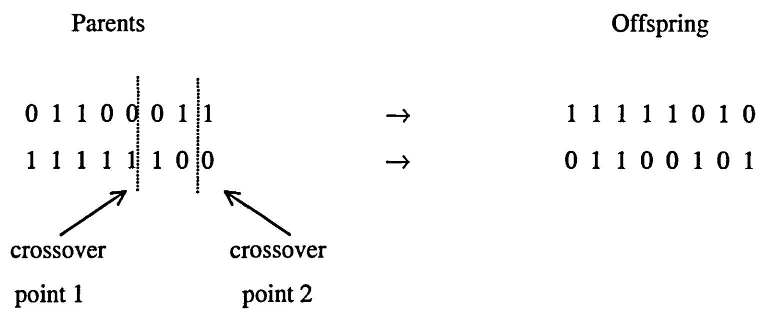


Figure 3.8. *Illustration of two-point crossover.*

Since two crossover points are selected, this scheme reduces the likelihood of disrupting schemata of large defining length and can also combine more schemata than single point crossover. End-point bias is also reduced.

Multi-point crossover (which also encompasses two-point crossover) exchanges genetic information about *p* crossover points. The study by Spears and De Jong [23] suggests that two crossover points is the optimum number and that more than this results in GA performance degradation.

3.6.4.3 Uniform and Parameterised Uniform Crossover

Uniform crossover [24] does not involve exchanging segments of chromosomes but rather it involves exchange on a gene by gene basis. The exchange of genes is performed according to a randomly generated crossover mask which consists of 1's and 0's and is equal in length to the chromosomes. Wherever a 1 appears in this mask, the corresponding gene in the first parent is used to form the offspring and wherever a 0 appears, the gene from the second parent is used. This scheme is illustrated in Figure 3.9 below.

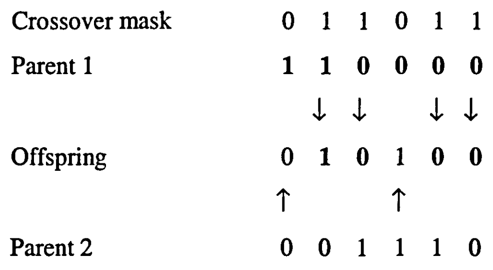


Figure 3.9 *Illustration of uniform crossover*

The second offspring is created by swapping the parents and repeating the process. Since the crossover mask is generated randomly with the 1's and 0' having equal probability of occurrence (0.5), the number of crossover points using uniform crossover will average $l/2$, where l is the chromosome length.

Parameterised uniform crossover [25] is similar to the above method except that the exchange of genes occurs with a probability p_x which usually takes a value in the range $0.5 \le p_x \le 0.8$.

Uniform crossover in general has no positional bias so any schema contained in the two parents can be recombined in the offspring. However, since this technique can be very disruptive of schemata, it can prevent coadapted alleles being preserved from generation to generation.

3.6.4.4 Brief Comparison of Schemes

As with so many other GA issues there is no simple answer to the question, "which crossover scheme is the best ?". Much of the research is based on empirical evidence and on only a handful of test functions. Different studies have also produced different results [8]. Although a number of properties such as positional bias and probability of schema disruption have been devised to aid the study of crossover schemes, it seems that even they do not provide definitive guidelines.

Spears and De Jong [24] strongly favour parametrised uniform crossover and are very critical of multi-point crossover for more than two crossover points. Very few researchers however, have addressed the interactions between the different crossover schemes and for example, encodings, fitness functions and so on. It seems that these types of studies will provide further insight into which variant of crossover to use and why.

So which scheme does one use? It is the view of the author that two-point crossover provides the most balanced technique in terms of simplicity of implementation, schema disruption and positional bias.

3.6.5 Mutation

As already mentioned mutation is the sporadic, random alteration of genes. For binary encoded chromosomes mutation flips gene values from 1 to 0 or vice versa. In real valued encodings mutation may result in a randomly generated number replacing the existing allele of a gene.

The crossover operator enables rapid exploration of the search space. After a number of generations however, with crossover alone, the GA will converge on a number of optima and exploration of uncharted territory will cease. The GA is said to be exploiting existing chromosomes. The mutation operator provides a mechanism for further exploration of the search space by creating chromosomes that may not have been created through crossover alone.

3.6.6 Selection of GA Operator Probabilities

As already mentioned, the search strategy of a GA is a balance between exploration and exploitation. Population size, selection, crossover probability and mutation probability are key parameters in helping to achieve this balance. High crossover probabilities increase the incidence of recombination of good schemata but also can disrupt good strings. Increasing mutation rates tend to transform the GA into a random search but also helps to introduce lost genetic material and ensures some exploration takes place during the latter stages of the search. Large populations can ensure greater genetic diversity and help to prevent premature convergence, but the run-time of the algorithm is obviously increased.

Choice of these interacting control parameters is obviously an optimisation problem in itself and is an active area of study [26],[27] . Several researchers have proposed sets of control parameters that have performed well on test functions. Two distinct sets which are particularly effective in the case of large and small populations respectively have emerged.

1. crossover probability = 0.6
   mutation probability = 0.001
   population size = 100

This scheme places emphasis on crossover in large populations rather than on mutation [26].

2. crossover probability = 0.9
   mutation probability = 0.01
   population size = 30

This scheme favours very high crossover rates as high disruption is seen to be needed in small populations [27].

Another approach proposed by the GA community involves so-called adaptive schemes, in which the operator probabilities change with time. Fixed parameter settings have the disadvantage of perhaps working better early on in a run but then not so well later on. One strategy is to exponentially decrease mutation as generations evolve to decrease exploration and increase exploitation [28], [29]. The rationale behind this plan is that in later generations the GA should not be thrown off the scent of good solutions. A completely opposite approach is to exponentially increase mutation as generations evolve since crossover will no longer produce further variety amongst solutions [30], [31]. Another adaptive approach is based on the success of an operator at improving the fitness of offspring [32]. Credit is given to an operator when it produces the fittest chromosome in the population. A weighting score is then given to each operator based on its success over the past 50 (or any other given number) of matings. For each subsequent mating each operator is selected probabilistically based on its score over the previous 50 matings. During the course of the GA the operator probabilities vary and adapt specific to the problem in hand.

As is the case with many topics in GA theory, there is lively debate about which strategy should be adopted and why. Many of the arguments are based on empirical evidence and have been proven only for a handful of test functions [33], [34]. When applying a GA to a problem, the user must be willing to experiment and find appropriate operator probabilities and strategies for the particular case. The ideal GA would not only adapt chromosomes to a particular fitness function but would also adapt parameter settings such as population size, selection pressure, and crossover and mutation rate. As population evolve, these parameters would be dynamically altered based upon the performance of the algorithm. At present it seems that very little seems to have been done on this front which is likely to be a very important area for future research.

3.7 GA Applications

Since their inception, GAs have been embraced across a wide range of application areas. The GA's abilities as an optimisation tool have been recognised as providing a viable and in some cases, an only option in many problems areas.

Numerical Function Optimisation. Function optimisation was one of the earliest applications of GAs. Conventional optimisation techniques do not always work well on multi-modal and noisy functions but GAs have excelled in this area [35].

Scheduling Problems. This class of problems require the efficient allocation of resources within some given constraints. Examples include the school time-tabling problem [36], efficient allocation of processor time in a multiprocessor computer system [37][38] and the optimisation of the manufacturing processes within an industrial plant to optimise profit, production time and so on [39].

Aerospace and Automotive Design. Engine designs have been optimised to reduce noise [40], aerodynamic optimisation for aircraft has been tackled [41] as has controller design for complex fighter aircraft [42] and a GA has been applied to the design of an anti-lock braking system [43].

Machine Learning and Artificial Intelligence. GAs have been successfully used in a number of robotic/intelligent machine applications such as motion planning [44][45]. Efficient neural networks have been designed with the help of GAs [46].

Genetic Programming. Genetic programming [47][48] is a branch of evolutionary computing that evolves computer programs to solve a specific task. Given a set programming functions (sorting, addition, multiplication, etc.) an initial population of programs is generated containing a random sample of the functions. The fitness is assessed by the ability of a given program to solve a certain problem, such as sorting a set of numbers and finding the mean and standard deviation. Subsequent generations evolve in the usual manner.

3.8 Summary

It is apparent from this brief introduction that GAs form a very broad and rich subject area. From the subject's infancy to the present day, researchers have been developing new implementations of the algorithm as more and more application areas are tackled. Although the list of applications in the previous section is just a sub-set of the areas in which they have been applied, it is apparent that GAs are able to solve problems across a wide range of disciplines. The main criticisms of past GA research is that much of it has been empirical, in particular with regards to parameter settings. There are many papers that conclude, "Our findings are based on a limited number of test functions....and the users must be willing experiment with their fitness functions". This however may be changing, as the need for a firm theoretical framework is being recognised.

Nonetheless, the success of GAs over the last 20 years or so has been promising and has resulted in several international conferences and journals. Many Artificial Intelligence/Machine Learning paradigms have made a spectacular entrance into academia and industry. Unfortunately, early promises often fail to materialise and the techniques die as quick as they became the vogue. GAs on the other hand seem to have passed the rigours of time, their success has if anything, exceeded initial claims. Although they may never provide a panacea for all, they have certainly established themselves in the mainstream as robust optimisation tools.

3.9 References

[1] Holland, J, "ADAPTATION IN NATURAL AND ARTIFICIAL SYSTEMS", The MIT Press, Cambridge, Massachusetts, USA, 1992.

[2] Darwin, C., "The Origin of Species", Orion Books, 1872.

[3] Matthews, R., "Hard Maths ? No problem", New Scientist, No. 2001, Vol. 148, pp. 40 - 43, 28 October 1995.

[4] Goldberg D.E., "GENETIC ALGORITHMS IN SEARCH, OPTIMISATION AND MACHINE LEARNING", Addison-Wesley , Reading, MA, 1989.

[5] R. Poli., "Why the Schema Theorem is Correct also in the Presence of Stochastic Effects", Proceedings of the Congress on Evolutionary Computation (CEC 2000), San Diego, USA, July 2000

[6] Vose, M. D., "Generalising the notion of schema in genetic algorithms", Artificial Intelligence, No. 50, p350-396. , 1991

[7] Grefenstette J. J., Baker J. E. , "How genetic algorithms work: A critical look at implicit parallelism", In J. D. Schaffer, ed. , Proceedings of the International Conference on Genetic Algorithms, Morgan Kaufmann, 1989.

[8] Mitchell M., "AN INTRODUCTION TO GENETIC ALGORITHMS", The MIT Press, Cambridge, USA, 1996.

[9] Montana D.J., Davis L.D., "Training feedforward networks using genetic algorithms", Proceedings of the International Joint Conference on Artificial Intelligence", Morgan Kaufmann, 1989.

[10] Schulze-Kremer S., "Genetic algorithms for protein tertiary structure prediction", In R. Manner and B.Manderick, eds., Parallel Problem Solving from Nature 2, North-Holland, 1992.

[11] Antonisse J., "A new interpretation of schema notation that overturns the binary encoding constraint", In Schaffer J.D. ed., Proceedings of the Third International Conference Algorithms, Morgan Kaufmann, 1989.

[12] Janikow C.Z, Michalewicz Z., "An experimental comparison of binary and floating point representations in genetic algorithms", In R.K. Belew and L.B. Booker eds. , Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 151 - 157, Morgan Kaufmann, 1991.

[13] Wright A.H., "Genetic algorithms for real parameter optimisation", I G. Rawlins, ed., Foundations of Genetic Algorithms, Morgan Kaufmann, 1991.

[14] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M., da Fonseca, V.G., "Performance assessment of multiobjective optimizers: an analysis and review", IEEE Transactions on Evolutionary Computation, pp. 117- 132, Vol. 7, Issue 2, Apr. 2003.

[15] Goldberg D. E. , Deb K., "A comparative analysis of selection schemes used in genetic algorithms", In Rawlings J.G.E. ed., Foundations of Genetic Algorithms, pp. 69 - 93, Morgan Kaufmann, 1991.

[16] Blickle T., Thiele L., "A comparison of selection schemes used in genetic algorithms", Technical Report Nr. 11, Swiss Federal Institute of Technology, Zurich, Switzerland, December 1995.

[17] Hancock P.J.B., "An empirical comparison of selection methods in evolutionary algorithms", In Fogarty T.C. ed. Evolutionary Computing: AISB Workshop, 1994.

[18] de la Maza M., Tidor B., "An analysis of selection procedures with particular attention paid to proportional and boltzmann selection", In Forrest S. ed., Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, 1993.

[19] Blickle T., "Theory of Evolutionary Algorithms and Applications and Application to System Synthesis", Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, 1996.

[20] Baker J.E., "Reducing bias and inefficiency in the selection algorithm", In Grefenstette J.J. ed., Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Earlbaum, 1987.

[21] Beasley D., Bull D.R., Martin R.R., "An overview of genetic algorithms: Parts 1 and 2", Inter-University Committee on Computing, Dept. of Computing Mathematics, University of Cardiff, Cardiff, UK, Dept. of Electrical and Electronic Engineering, University of Bristol, Bristol, UK, 1993.

[22] Baker J.E., "Adaptive selection methods for genetic algorithms", In J.J. Grefenstette, ed., Proceedings of the First International Conference on Genetic Algorithms and their Applications, Erlbaum, 1985.

[23] Spears W.M., De Jong K.A., "An analysis of multi-point crossover", In G. Rawlins ed., Foundations of Genetic Algorithms, pp301 - 315, Morgan Kaufmann, 1991.

[24] Syswerda G., "Uniform crossover in genetic alogorithms", In R.K. Belew and L.B. Booker eds. , Proceedings of the Fourth International Conference on Genetic Algorithms, pp2 - 9, Morgan Kaufmann, 1991.

[25] Spears W.M., De Jong K.A., "On the virtues of parameterised uniform crossover", In J.D. Schaffer, ed., Proceedings of the Third International Conference on Genetic Algorithms, pp230 - 236, Morgan Kaufmann, 1989.

[26] De Jong K.A. and Spears W.M., "An analysis of the interacting roles of population size and crossover in genetic algorithms", Proc. First Workshop Parallel Problem Solving from Nature, pp. 38-47, Springer-Verlag, Berlon, 1990.

[27] Grefenstette J.J., "Optimisation of control parameters for genetic algorithms", IEEE Trans. Systems, Man and Cybernetics, Vol SMC-16, No.1 , Jan/Feb 1986.

[28] Srinivas M, Patnaik L.M., "Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms", IEEE Transactions on Systems, Man and Cyberspace, April 1994.

[29] Michalewicz Z., Janikow C.Z., "Handling constraints in genetic algorithms", In R.K. Belew and L.B. Booker eds. , Proc. of the Fourth International Conference on Genetic Algorithms, pp. 151 - 157, Morgan Kaufmann, 1991.

[30] Davis L., "Adapting operator probabilities in genetic algorithms", In J.D. Schaffer, ed., Proceedings of the Third International Conference on Genetic Algorithms, pp61 - 69, Morgan Kaufmann, 1989.

[31] Syswerda G., "Schedule optimisation using genetic algortihms", In L. Davis ed., HANDBOOK OF GENETIC ALGORITHMS, pp. 332 - 349, Van Nostrand Reinhold, 1991.

[32] L. Davis, "HANDBOOK of GENETIC ALGORITHMS", Van Nostrand Reinhold, 1991.

[33] Bramlette M.F., "Initialization, mutation and selection methods in genetic algorithms for function optimisation", In R.K. Belew and L.B. Booker eds. , Proc. of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, 1991.

[34] Schaffer J.D., Caruana R.A., Eshelman L.J., Das R., "A study of control parameters affecting online performance of genetic algorithms for function optimisation", In J.D. Schaffer, ed., Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, 1989.

[35] De Jong K.A., "An analysis of the behaviour of a class of genetic adaptive systems", Ph.D. Thesis, University of Michigan, Ann Arbor, USA, 1975.

[36] Burke E.K., Elliman D.G. and Weare R.F., "A Genetic Algorithm Based University Timetabling System" East-West Conference on Computer Technologies in Education, Crimea, Ukraine pp35-40, 1994.

[37] Ahmad I., Dhodhi M.K., "Multiprocessor scheduling using a problem-space genetic algorithm", Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 152 - 157, University of Sheffield, Sheffield, UK, Sept. 1995.

[38] Baxter M.J., Tokhi M.O., Fleming P.J., "Task-processor mapping for real-time parallel systems using genetic algorithms", Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 158 - 163, University of Sheffield, Sheffield, UK, Sept. 1995.

[39] Hindi, K.S., Hongbo Yang, Fleszar, K., "An evolutionary algorithm for resource-constrained project scheduling, IEEE Transactions on Evolutionary Computation, pp. 512- 518, Vol. 6, Issue 5, Oct. 2002.

[40] Fisher K.A., "The application of genetic algorithms to optimising the design of an engine block for low noise", Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 18 -22, University of Sheffield, Sheffield, UK, Sept. 1995.

[41] Obayashi S., Takanashi S., "Genetic algorithm for aerodynamic inverse optimisation problems", Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 7 -12, University of Sheffield, Sheffield, UK, Sept. 1995.

[42] Chipperfield A., Fleming P.J., "Gas turbine engine controller design using multiobjective genetic algorithms", Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 214 - 219, University of Sheffield, Sheffield, UK, Sept. 1995.

[43] Yonggon Lee, Zak S.H , "Designing a genetic neural fuzzy antilock-brake-system controller", IEEE Transactions on Evolutionary Computation, pp. 198- 211, Vol. 6, Issue 2, Apr. 2002.

[44] Chen M., Zalzala A.M.S., “Safety considerations in the optimisation of paths for mobile robots using genetic algorithms”, Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 299 - 306, University of Sheffield, Sheffield, UK, Sept. 1995.

[45] Rana A.S., Zalzala A.M.S., “An evolutionary algorithm for collision free motion planning of multi-arm robots”, Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 123 - 130, University of Sheffield, Sheffield, UK, Sept. 1995.

[46] Tang K.S., Chan C.Y., Man K.F., Kwong S., “Genetic structure for NN topology and weights optimisation”, Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 250 -255, University of Sheffield, Sheffield, UK, Sept. 1995.

[47] Koza J.R., “GENETIC PROGRAMMING: ON THE PROGRAMMING OF COMPUTERS BY MEANS OF NATURAL SELECTION, MIT Press, 1992.

[48] Kishore, J.K.; Patnaik, L.M.; Mani, V.; Agrawal, V.K., “Application of genetic programming for multicategory pattern classification”, IEEE Transactions on Evolutionary Computation, pp. 242-258, Vol. 4, Issue 3, Sep. 2000.

Chapter 4. The Derivation of Minimal Test Sets Using Genetic Algorithms

4.1 Introduction

The design of today's integrated circuits (ICs) and microprocessors is a very large and complex task. For example, Intel's Pentium II processor, containing 7.5 million transistors, took approximately 1600 man years to design [1]. To remain competitive, microprocessor manufacturers must continually satisfy Moores Law[1], striving to develop faster, more complex designs. This increase in complexity requires substantial increases in design effort which in turn requires the continual development of sophisticated computer aided design (CAD) and test algorithms.

Digital design by nature requires many large and NP-hard problems to be solved. Many tasks involve finding solutions to multi-modal, multi-objective optimisation problems. Over recent years researchers have applied a variety of mathematical techniques to these problems, including Genetic Algorithms. Although GAs do not provide solutions for every optimisation task within digital design, they have certainly excelled in a number of key areas.

Before the concept of a minimal test set and its derivation using a genetic algorithm is discussed, this chapter will begin by introducing the motivation behind this work and then with a brief overview of GAs as applied to the CAD of integrated circuits. This is then followed by a survey of existing test set minimisation techniques.

4.2 Motivation

The motivation behind the derivation of minimal test sets is two-fold. A number of integrated circuit manufacturers are faced with the task of testing millions of units per annum. As mentioned in Chapter One, Intel, the microprocessor manufacturer produces in the region of 50 million units each year and the testing of these can account for up to one third of their total manufacturing budget [2]. Post production testing of ICs requires each unit to be placed in an automatic test equipment (ATE) which applies a set of test vectors to the circuit under test. Smaller test sets will result in smaller test application times in the ATE, in turn reducing the per unit cost of testing.

Another very important reason for reducing test set sizes is due to the increased use of built-in self test (BIST) [3] structures that are becoming prevalent in the current generation of ICs. In this scheme, the

[1] Moore's Law states that microprocessor complexity (power, speed, transistor density) roughly doubles every 2 years or so.

test vectors required to test a circuit or sub-circuit within an IC are stored on the circuit itself. A small area of circuit will contain memory elements (in most cases read only memory) which are exclusively used for storing the test vectors. Obviously, as test sets become smaller so does the amount of on-board memory required to store the test vectors. This in turn reduces the amount of silicon required to produce the IC, which in turn reduces the cost of each circuit and increases the yield during manufacture.

4.3 Application of GAs in Computer Aided Design and Test of Integrated Circuits.

Many of the problems in CAD of ICs including the derivation of minimal test sets, share the following characteristics [4].

- Global multi-modal, multi-objective optimisation tasks
- Large, NP-hard problems
- Mutually dependent problems, artificially divided into (NP-hard, large) sub-problems
- Highly constrained
- 'Noisy' cost/objective functions (estimations)

If GAs are to be accepted in the CAD community they must compete with the current state-of-the-art algorithms for solving problems of this nature. They should not be used because they are novel or the current vogue. They must offer performance improvements over the current, established techniques in a given problem domain.

One of the earliest applications of GAs to IC design was the NP-hard problem of circuit layout or floorplan design [5]. The shape of various modules in a microprocessor for example (memory blocks, ALU, etc.), can be approximated as rectangles. The goal of the GA is to place and orient *n* given rectangles such that no rectangles overlap and the area of the rectangle enclosing all *n* rectangles is minimised. The earliest work using GAs dates back to the mid to late eighties [6][7], but was not competitive with what was then the state-of-the-art. Over recent years however, GA based approaches have produced some very competitive results, finding near optimal solutions [8][9][10]. Once the constituent blocks of an IC have been placed, the various connections between them have to be established. This process of *routing* has also benefited from competitive GA approaches [11][12].

More recently, GAs have been applied to some of the earlier stages in the digital design process. For example they have been used to minimise combinational and sequential logic expressions [13][14]. In the domain of logic synthesis GAs have been able to produce good solutions. The goal of the optimisation process in this domain is to produce Boolean representations with, for example, low power consumption and small circuit areas. Two state-of-the-art data structures for representing digital circuits are Binary Decision Diagrams (BDDs) and Ordered Kronecker Functional Decision Diagrams (OKFDDs). GAs have been applied successfully to both BDD minimisation [15] and OKFDD minimisation [16].

An exciting development in circuit design has been the recent advent of the field of *evolvable hardware* [17]. This term describes the process of employing an evolutionary algorithm to design electronic circuits or components. Genetic programming has been used to design both passive and active analogue circuits [18]. Genetic algorithms have been used to design efficient operational amplifiers [19], and also to optimise transistor size in VLSI circuits [20]. A field that has attracted much attention since the mid 1990s has been evolutionary electronics [21]. Given a circuit function, GAs evolve candidate circuits which are mapped onto Field Programmable Gate Arrays (FPGAs). The gate arrays contain standard logic gates such as AND, OR, NOT and so on. The final results produced by the GA not only employ these gates but also the intrinsic properties of silicon, from which the gate arrays are composed [22]. The circuits are very often far more compact than those designed by humans using traditional techniques. Also, traditional techniques are most unlikely to adopt the capacitative/inductive properties of silicon in the design, something that the GA seems able to exploit. The researchers in this field are as yet unable to fully understand why the final evolved design functions correctly. When seemingly redundant circuit features are disconnected from the circuit, they often cease to operate correctly.

From the above discussion, GAs have been used across a broad range of digital/analogue design tasks. Many of the applications have required the design of new variants of the genetic algorithm. For example new encodings and new ways of implementing the GA operators (e.g. crossover, mutation) will have been devised. It is fair to state that many GA based CAD algorithms do not compete with existing state-of-the-art techniques on the basis of time. They do however compete on solution quality. Table 4.1 below, taken from [23], illustrates this point for placement/floorplan design. But it is felt that as computing power continues to increase and pure GA research matures, their use will become prevalent in many other design areas where they are currently uncompetitive on time.

Algorithm	Result Quality	Speed
Simulated Annealing	Near Optimal	Very Slow
Genetic Algorithm	Near Optimal	Very Slow
Force Directed	Medium....Good	Slow....Medium
Numerical Optimisation	Medium....Good	Slow....Medium
Minimum Cut	Good	Fast
Clustering/constructive placement	Poor	Fast

Table 4.1 *Comparison of VLSI Cell Placement Algorithms taken from [23]*

One of the more successful applications of GAs is in the area of digital testing and in particular automatic test pattern generation. Test Pattern generation (TPG) is an NP-hard problem [24], well suited to a GA approach. One of the earliest GA based test generation schemes was presented by Srinivas and Patnaik [25]. The scheme uses a random test pattern generator to create the initial population of test vectors. Fault simulation is then invoked and using the search capabilities of a GA, test sets are created with a given fault coverage. The scheme combines two traditional TPG methods viz. the directed search approach and random test pattern generation. This combined approach results in a smaller number of test vectors having to be fault simulated than in a purely random scheme. The algorithm has been used to generate test sets for the ISCAS-85 benchmark circuits, where it produced considerably better results than the random approach in terms of both test set size and the number of test vectors which have to be fault simulated. O'Dare and Arslan [26] have presented a similar TPG scheme using fault simulation and an initial, randomly generated population of test vectors. They claim to have generated compact test sets, although no real evidence is presented in the paper and no results for any benchmarks circuits are included.

Both of the above test pattern generation schemes are for combinational circuits. Over recent years much work has appeared for sequential circuits [27][28][29] and the reader is directed towards the references for further details.

4.4 A Survey of Test Set Minimisation Techniques and Algorithms

Many of the popular test pattern generation algorithms are capable of achieving very high fault coverages for both sequential and combinational circuits [3][30-33]. Achieving high fault coverage is a well researched and understood domain. Over recent years however, the research community has focused on the issue of maintaining high fault coverage while reducing the number of test vectors in the test set. Many ATPG algorithms do not address this issue of test set minimisation [34][35][36][37] but, increasingly, papers outlining test set minimisation techniques are appearing in the literature. In what follows these techniques and algorithms will be discussed in more detail.

Random test set generation [38] is a test set minimisation technique that may be viewed as incorporating a 'random walk' optimisation strategy. From an initial test set, generated by an ATPG, further test sets are formed by randomly selecting test vectors from the initial set. If there are t test vectors in the initial test set then the randomly generated test sets will comprise v test vectors where $v \leq t$. The fault coverage of each test set is obtained by performing fault simulations. The procedure is halted after a given number of test sets are generated and the smallest test set with the required fault coverage is returned. Random test set generation does achieve some reduction in test set size but the quality of the final test sets does not compete with those generated by more 'directed' means. Results obtained by such means will be presented later in this chapter.

Reverse order fault simulation [39][40][41][42] is a relatively simple and popular technique. An ATPG derives an initial test set. The order in which the test vectors were generated is retained and is reversed and fault simulation is then invoked. A reduction in test set size normally occurs, since it is often the case that test vectors generated late in the test generation process detect hard-to-detect faults in addition to some of the easier to detect faults which may already have been covered by earlier test vectors.

Although simple, this technique does require additional fault simulations, which is a costly process (both in terms of processor speed and memory requirements). Reductions are certainly achieved in test set size, but it is felt that they are not competitive in comparison with more sophisticated methods.

PODEM-X [43] is a well established test pattern generation algorithm, comprising three test generation schemes, a fault simulator and a test set compactor. The test set compactor merges as many different test vectors as possible into a single test pattern. The merging of patterns is achieved by making use of the unassigned primary inputs in a given test pattern. Obviously, the assigned primary inputs of the test patterns to be merged must match to avoid any conflicts. For example consider the test patterns given in Table 4.2. The first and the final patterns can be merged since the unassigned values in pattern 1 can be replaced by the corresponding assigned values in pattern 3 and the assigned primary values in both patterns are equal, therefore avoiding conflict. Test patterns 2 and 3 cannot be merged however since the assigned values of primary inputs *a* and *e* conflict.

Test				*Primary*	*Inputs*			
Vector	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
1	x	x	0	x	0	0	x	0
2	0	x	0	x	1	0	0	x
3	1	1	x	1	0	0	x	0

Table 4.2 *Three test vectors and their primary input assignments. Test vectors 1 and 3 can be merged as their assigned primary inputs are non-conflicting.*

The above scheme is an example of static compaction. That is, the compaction algorithm is invoked after the test patterns have been generated. PODEM-X also contains a dynamic compactor which attempts to reduce test set sizes as the test vectors are generated. When a test vector is generated for a given fault, it is often the case that not all primary inputs will be assigned. When this occurs, PODEM-X selects a different fault and attempts are made to cover this new fault with the assigned values of the original vector and by experimenting with assignments to the unassigned primary inputs. Further faults are selected and simulated until as many primary inputs as possible of the of the original test vector are assigned.

As with all test generation and compaction schemes, the above process is a collection of NP-hard problems. As the number of inputs increase, experimenting with the unassigned values of each test vector becomes a costly process.

GATE [44] is a scheme based on genetic algorithms that not only generates test vectors for combinational circuits but also applies a number of heuristics to achieve compact test sets. The test generation algorithm within GATE is known as SOFE [40] and it comprises a random test pattern generator and a fault simulator. SOFE itself performs test set minimisation and begins with a randomly generated, initial test set with a given fault coverage. The fault simulator then simulates a given fault until a test vector (from the initial test set) is found to cover that fault. This is known as 'Stop On First Error' (SOFE). Further faults are simulated, reducing the size of the initial test set since only test vectors that cover a new fault are retained. A second trial (SOFE/2) is then performed by reversing the order of the test vectors in the test set, in the hope of achieving a further reduction in test size as discussed in section 4.5.2. Further SOFE trials are then performed with other test set orderings in the hope of reducing the test size even further. Carter et al. [40] used SOFE/6 in conjunction with a covering heuristic (simulating groups of vectors and eliminating vectors if there is an overlap in fault coverage) to generate relatively compact test sets.

GATE starts by randomly generating a population of *n* test sets and pre-processes them in a manner very similar to SOFE (reverse order fault simulation and other permutations of test vectors in a test set). GATE then evolves subsequent populations of test sets in the usual manner using selection, crossover and mutation. To achieve a further level of compaction and algorithmic efficiency, the evaluation of the fitness function for each test set is achieved using a 'covering heuristic'. Repeated fault simulation of a test vector is expensive. In an effort to reduce this cost, GATE compiles a so-called fault matrix (see Section 4.5) so each test vector is only simulated once. To achieve the additional compaction, a test vector is only included in the test set if it covers the greatest number of remaining faults, which may be ascertained from the fault matrix.

GATE therefore can be seen to perform three distinct levels of test set compaction.

i. SOFE-like pre-processing of the initial population of test sets.
ii. GATE's own covering heuristic to eliminate test vectors that cover previously detected faults.
iii. Search space exploration capabilities of a GA to locate minimal test sets.

The effectiveness of the GATE algorithm can be seen from results in Table 4.3 below, which compares the results of a 'pure' SOFE approach with those of GATE.

Circuit	**SOFE/6** Test set size		**GATE** Test set size		**Reduction using GATE** (%)	
	Average	Best	Average	Best	Average	Best
C432	45.0	45.0	39.4	37.0	12.4	17.8
C3540	149.0	149.0	128.0	128.0	14.1	14.1

Table 4.3 *Comparison of test set sizes achieved by SOFE/6 and GATE for two ISCAS-85 benchmark circuits, taken from [53]. The results are an average over 20 trials of each algorithm.*

As can be seen from the above table, the additional processes of GATE achieve test set reductions of between 14.1% and 17.8% over the SOFE/6 approach.

COMPACTEST [45] is a test set generation and compaction algorithm which also employs a variety of minimisation heuristics. Before the test vectors are actually generated (using PODEM [31]), there is a pre-processing phase which reorders the fault list. In such a fault oriented test pattern generator, the order in which the faults are presented to the algorithm can play a significant role in determining the size of the final test set. When a test vector is generated for a fault appearing at the top of a fault list it is fault simulated and is often found to cover faults found nearer the bottom of the list. This process is continued until all faults are covered. COMPACTEST includes a heuristic for ordering the faults in such a way that tests covering faults at the top of the list cover a maximal number of faults found later in the list. The heuristic relies on the concept of *independent faults.* As this is beyond the scope of the current discussion, the reader is directed to the reference for the details of the reordering algorithm.

A second heuristic employed by COMPACTEST is known as *maximal compaction* which aims to increase the fault coverage of a test vector by determining which assigned, primary inputs are not essential to cover the fault for which the test was generated. By identifying these primary inputs and experimenting with assignments, further faults are (hopefully) covered. The process is similar to that used in PODEM-X.

A final heuristic used within COMPACTEST is known as *rotating backtrace,* the aim being to sensitise different paths each time a value on a line has to be justified. In doing so, it is hoped that different faults, propagating along different paths may be detected by a given test vector. An example, taken from [45] will be discussed to clarify the concept. Figure 4.1 below illustrates a combinational circuit. If a fault from the set {5/1, 6/1, 7/1, 8/1} is to be covered it is necessary to set line 12 and therefore line

10 to logic 0. This in turn will require at least one of the primary inputs {1, 2, 3, 4} to be set to logic 0. In Pomeranz et al. [45] they outline the fact that in existing ATPGs the selection of which one of these primary inputs that is set to 0 is fixed. So for example when generating tests for the faults {5/1, 6/1, 7/1, 8/1}, primary input 1 will always be set to 0. Thus a primary fault will be covered, taken from the set {5/1, 6/1, 7/1, 8/1} in addition to a secondary fault, viz. 1/1. In COMPACTEST however, for each fault in the aforementioned set, a different primary input (from primary 1 to primary input 4) will be selected. So for example, to cover 5/1 primary input 1 will be set to 0, for 6/1 primary input 2 will be set to 0 and so on, thus increasing the number of secondary faults covered by the test vectors.

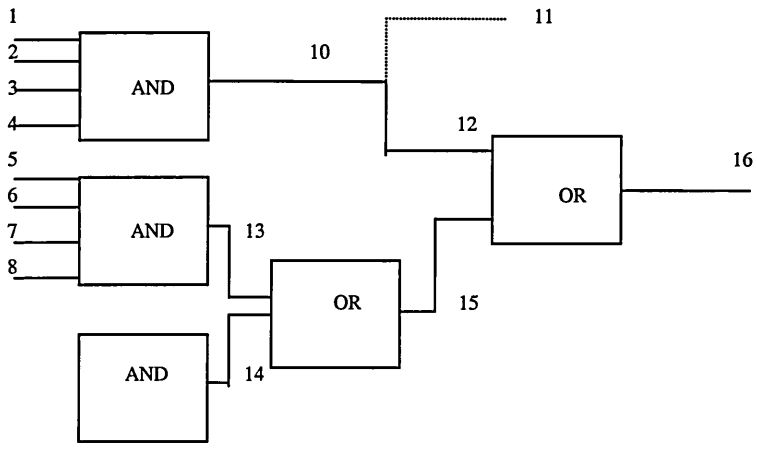


Figure 4.1 *Combinational logic circuit taken from [54].*

By employing the three compaction heuristics just described, COMPACTEST is able to generate very compact test sets, competing with many existing schemes in terms on final solution quality i.e. test set size.

The test set minimisation procedures described above have been largely developed for combinational circuits. Over the last three or so years some work has appeared addressing the compaction issue for sequential logic circuits [46-49]. Since the present discussion is relevant to combinational circuits, the reader is directed to the references for further details.

4.5 The Minimal Test Set Problem

For a combinational logic circuit and the single stuck-at fault model, a minimal test set can be defined as follows:

For a given set S of detectable faults a set of test vectors is said to be a minimal test set if it contains the least number of test vectors required to cover all the faults in S.

It must be noted that in general, for a given circuit, a minimal test set is not always unique as other test sets of the same size and fault coverage may exist. Using test generation algorithms such as those mentioned earlier, test vectors can be derived for most if not all faults in a combinational circuit. These test vectors therefore comprise a test set for the circuit under test. Consider the circuit given in Figure 4.2 below. The circuit contains three primary inputs and a single primary output. It contains five circuit lines and since each line can be either stuck-at 1 or stuck-at 0, the circuit has ten possible faults. Since there are three primary inputs, there are 2^3 possible input vectors, each of which is displayed in the *fault matrix* given in Table 4.3. A fault matrix is a convenient means of displaying the fault coverage of each test vector. The notations *k*/0 and *k*/1 denote the fault line *k* stuck-at-0 and stuck-at-1 respectively and a √ indicates each fault that is covered by a given test vector. For example, the test vector 000 covers the faults *c*/1, *d*/1 and *e*/1.

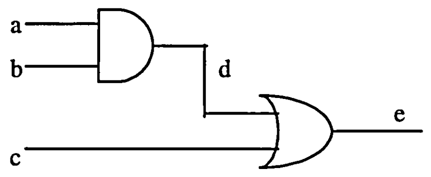


Figure 4.2 *A three input, single output, combinational logic circuit*

Test	Fault									
abc	*a*/0	*a*/1	*b*/0	*b*/1	*c*/0	*c*/1	*d*/0	*d*/1	*e*/0	*e*/1
000						√		√		√
001					√				√	
010		√				√		√		√
011					√				√	
100				√		√		√		√
101					√				√	
110	√		√				√		√	
111									√	

Table 4.3 *Fault matrix for the circuit given in Fig. 4.2*

Due to the nature of combinational test pattern generation (path sensitisation, back-tracking [3]) it is often the case that a single test vector covers several faults and this can be seen in Table 4.3. Also apparent is the fact that most of the faults are covered by more than one test vector e.g. the fault *e*/1 is covered by each of the test vectors 000, 010, 100. It is this overlap in fault coverage that can be exploited to reduce the number of test vectors in the test set. For example, the test vector 111 covers the single fault *e*/0. By inspection of the fault matrix, this vector can be removed from the original test set since the vectors 001, 011, 101 and 110 cover this fault in addition to other faults.

From the above discussion, the derivation of minimal test sets is an example of a multi-objective optimisation problem. There are two interacting parameters which have to be optimised simultaneously, viz.

i. the number of test vectors in a test set have to be minimised and,
ii. fault coverage of the test set, which has to attain a predetermined level set by the ATPG tool.

So what is a minimal test set for the above circuit? For this simple example we expect all the faults to be covered and there are three minimal test sets, viz. (010, 100, 110, 001), (010, 100, 110, 011) and (010, 100, 110, 101). These sets are minimal, in the sense that they both cover all the faults and no smaller subset of the test vectors covers all the faults. Each set contains four test vectors, half the size of the original test set. A 50% reduction in test set size could well translate to a 50% reduction in the post production test overhead. Although simple, this example has illustrated the concept of a minimal test set and the reductions in test set size which can be achieved. In practice this reduction is often greater but even small reductions in test set size could help reduce the large test costs incurred by manufacturers such as Intel [2].

The size and complexity of the above circuit and its corresponding fault matrix are far removed from those encountered in practical designs. For a fault matrix containing eight test vectors the minimal test set can be found either through exhaustive test set evaluation or by inspection. But what about fault matrices containing hundreds of test vectors and possibly thousands of stuck-at faults? How would one find a global or near global minimal test set? As the number of test vectors, n in the original test set increases, the task of finding a minimal test set grows as 2^n i.e. the problem is NP-hard. Exhaustive evaluation of every possible test vector becomes unreasonable for values of say, $n > 20$, so a more sophisticated approach is required. From the discussion of Chapter 2, genetic algorithms are well suited to such an optimisation task and in what follows, such an algorithm will be shown to perform very well in finding minimal or near minimal test sets.

4.6 GA-MITS : Genetic Algorithm based MInimisation of Test Sets

GA-MITS is a wholly original test set minimisation algorithm developed by the author and forms a large part of this PhD. It has been written using the 'C' programming language under the UNIX operating system and uses a non-standard GA technique designed specifically for this application.

GA-MITS is a static, test set minimisation algorithm designed as a post-processor for existing test pattern generation algorithms. As a post processor, GA-MITS will be presented with the results of an

ATPG tool that is, a test set consisting of a given number of test vectors which covers a given number of faults. Let the original test set *T* generated by the test pattern generator consist of $N(T)_{\max}$ test vectors and cover a total of $N(C)_{\max}$ faults. It has been decided by the author that in GA-MITS a constrained optimisation method will be adopted, i.e. the algorithm is minimising the test set subject to a predetermined level of fault coverage. This predetermined level of fault coverage is normally equal to the original fault coverage attained by the ATPG. The algorithm has been designed to easily accommodate other fault coverage strategies and it is simply a matter of changing a single parameter in the source code (it can search for test sets covering say, 90% of the faults covered by the original test set). The goal therefore of GA-MITS is to optimise two separate interacting and often conflicting parameters; minimising the number of test vectors in a test set while ensuring that the fault coverage of a test set is the same as the original fault coverage, or set to a prescribed level. In what follows, it is assumed for simplicity that the original fault coverage is to be preserved.

The data required by GA-MITS is the fault matrix data generated by the ATPG tool, similar to that given in Table 4.3. Given this, the algorithm 'searches' the fault matrix for minimal or near minimal test sets covering all $N(C)_{\max}$ faults. This approach may be regarded as an instance of the more general, set covering problem [50]. The algorithm in its current form is applicable to combinational logic circuits and the single stuck-at fault model, although it is felt by the author that it may be extended to accommodate sequential logic.

The pseudo-code given in Figure 4.3 outlines the overall structure of GA-MITS. The algorithm begins by generating a fault matrix from the test vector/fault coverage data produced by the ATPG tool. The test pattern generation algorithm, be it random, deterministic or whatever, is of no relevance to the minimisation algorithm. It is the resultant vectors and accompanying parameters, that are required for the minimisation process. These parameters are;

1 the number of test vectors $N(T)_{\max}$ present in the test set *T*, generated by the ATPG tool.

2 the total number of possible faults *f*, in the circuit under test

3 the total number of faults $N(C)_{\max}$ covered by the original test set where,

$$N(C)_{\max} \leq f .$$

4 the fault coverage of each test vector.

```
Read fault matrix data from ATPG system
Set GA parameters
    population size, N
    no. generations, G
    crossover probability, c
    mutation  rate, m

Generate initial random population of test sets
For each generation
  { Evaluate fitness of each test set
     Select N/2 parent pairs
         For each parent pair
            {
                  randomly generate 2 crossover points
                  For each gene
                      { apply crossover and mutation operators
                      }
            }
  }
Return fittest, minimal test set found after G generations
```

Figure 4.3 *High level pseudo code for GA-MITS*

Given the above parameters, a fault matrix can be generated and the required fault coverage, $N(C)_{\max}$ of the minimal test set can be established as a goal for the algorithm. The next few lines of the pseudo code set the necessary GA parameters of population size, number of generations to be created, and crossover and mutation rates.

The algorithm then commences in the usual GA manner by first generating a random, initial population of *N* test sets. Each 'random' test set will either be a sub-set of the original test set *T* or *T* itself. Since there are $N(T)_{\max}$ test vectors present in *T*, there are $2^{N(T)_{\max}} - 1$ possible test sets [2] that can be randomly generated or created at a later stage by the GA. The algorithm then enters the main loop, iterating through $(G-1)$ generations. Once the fitness of each candidate test set has been evaluated, selection, crossover and mutation are used to create the subsequent generation of test sets. When all generations have been created, GA-MITS returns the fittest, minimal test set found.

From the discussion of Chapter Two, it is apparent that the application of a GA to a given problem requires careful consideration of many design issues. For example, which crossover operator should one use and what should be the crossover rate? The answers to these and many similar questions can only be given once the GA designer has considerable knowledge of the application and after thorough experimentation. The remainder of this section will outline *which* operators, selection schemes etc. have been used within GA-MITS but will not give *why* each has been chosen. The reasoning behind the design decisions will be presented later in this chapter when theoretical and experimental justification will be provided.

4.6.1 ATPG Data and the Generation of a Fault Matrix

The fault matrix data used to evaluate GA-MITS and its results, were generated using four different test generation methods: an implementation of PODEM, a GA based TPG, a random TPG and Turbo-Tester [51] , an ATPG tool developed by Prof. Raimund Ubars's group at Tallinn Technical University, Estonia. The format of the test data provided by Turbo-Tester is given in Table 4.4. In this example, three test vectors have been generated for a circuit with 4 nodes and therefore 8 possible stuck-at faults. As can be seen, the form of the test/fault coverage data is not the same as that given in Table 4.3.

Test vector	Node			
	a	*b*	*c*	*d*
1	1	0	X	1
2	0	0	X	X
3	1	1	0	X

Table 4.4 *Output data supplied by Turbo-Tester. Each row corresponds to a test vector and the columns a, b, c and d represent circuit nodes. A '1', '0' and 'X' in a given column signify that the corresponding node is stuck-at-1 testable, stuck-at-0 testable and not testable at all, respectively.*

Instead, referring back to the Table 4.4, each test vector is given along with the type of fault it covers at each circuit node. For a given test vector, a '1' or '0' in a column signifies that the vector covers a stuck-at 1 or a stuck-at 0 fault respectively at the corresponding node. An 'X' in a column signifies that the vector covers no fault at that node. Test vector 1 therefore covers a stuck-at 1 fault at node *a*, a stuck-at 0 fault at node *b* and a stuck-at 1 fault at node *d*. GA-MITS takes the data in the above form and generates a fault matrix, similar to that given in Table 4.3. The corresponding fault matrix for the above data is given in Table 4.5.

[2] There are $2^{N(T)_{max}} - 1$ possible test sets since the empty set does not constitute a test set as it contains no test vectors and covers no faults.

Test	Faults							
Vector	a/0	a/1	b/0	b/1	c/0	c/1	d/0	d/1
1		√	√					√
2	√		√					
3		√		√	√			

Table 4.5. *Corresponding fault matrix for data given in Table 4.4*

4.6.2 Encoding Scheme and Chromosome Structure

As mentioned in Chapter 2, developing an encoding scheme to represent candidate solutions as chromosomes is one of the primary steps when applying a GA to a given problem. The goal of GA-MITS is to minimise test sets so the 'currency' of the GA, the chromosome, must represent a test set. The coding scheme devised for this algorithm was a binary encoded chromosome of length v, where v is the number of test vectors in the original test set, generated by the test pattern generator. A chromosome therefore contains v binary genes, each gene taking the value '1' or '0'. Figure 4.4 illustrates this chromosome structure for the circuit given in Figure 4.3.

$$[\;1\;\;0\;\;1\;\;0\;\;0\;\;0\;\;1\;\;1\;]$$

Figure 4.4. *A typical GA-MITS chromosome for the circuit in Figure 4.2 representing the test set consisting of the vectors (000, 010, 110, 111).*

This circuit contains 3 primary inputs and therefore has 8 possible input vectors. The chromosome given above has 8 genes, each gene corresponding to a test vector. The left-most locus (gene position) in the chromosome corresponds to the first test vector 000, the next locus, the vector 001 and so on. A gene value (allele) of 1 signifies the presence of the test vector corresponding to that locus and an allele of 0 signifies its absence. The above chromosome represents the test set consisting of the vectors (000, 010, 110, 111).

This above ordering of test vectors in a chromosome in terms of binary word value (000, 001, 010 etc.) works very well for small circuits where test patterns may be generated in such an orderly and exhaustive manner. Unfortunately today's test pattern generation tools do not generate test vectors in such a manner and they do not necessarily present test vectors in binary word order. The SOFE and GATE algorithms for example, permute the order of test vectors to achieve test pattern compaction. In such cases the order of test vectors in a chromosome is exactly the same as the order in which they are presented to GA-MITS by a particular ATPG tool. The first vector in the fault matrix will be represented by the first gene (from the left) in the chromosome, the second vector by the second gene and so on.

It is obvious from the chromosome structure that the total number of 'derivative' test sets that may be formed from the original test set, and therefore the size of the search space, is given by $2^{N(T)_{\max}} - 1$. This gives an indication of the size of the optimisation task, which rises exponentially with the size of the original test set presented to the algorithm.

4.6.3 The Fitness Function

The appropriate formulation of the fitness function is a key factor in the successful application of all genetic algorithms. The fitness function of GA-MITS must accurately measure the competence of a test set as a minimal test set, that is one which covers a maximal number of faults with a minimal number of test vectors. Let $N(T)$ be the number of test vectors contained in an arbitrary test set and let $N(C)$ be its fault coverage. As GA-MITS is attempting to optimise these two parameters, its fitness function must be formulated to maximise the number of faults covered $N(C)$ by a test set while minimising the number of test vectors $N(T)$. As the algorithm has been designed as a post processor to ATPG tools, the fault coverage must be maximised to a predetermined level, $N(C)_{\max}$. However, the number of test vectors in minimal or near minimal test sets is not known in advance.

It was found through experimentation that the fitness function F given below accurately reflects the competence of a test set as a minimal test set.

$$F = \begin{cases} N(C) & \text{for } N(C) < N(C)_{\max} \\ k.\dfrac{N(C)}{N(T)} & \text{for } N(C) = N(C)_{\max} \end{cases}$$

where k is a constant and satisfies the condition $k > N(T)_{\max}$.

This function has two forms. The form allocated to a particular test set is dependent on the fault coverage of the test set. For a test set that achieves sub optimal fault coverage i.e. $N(C) < N(C)_{\max}$, the fitness F of a test set is given by,

$$F_1 = N(C).$$

For a test set that achieves the desired fault coverage, the fitness F of a test set is given by

$$F_2 = k.\frac{N(C)_{\max}}{N(T)}.$$

For a given fault coverage $N(C)$ and fixed value of k where $k > 0$, it is clear that the value of F_2 increases as $N(T)$ decreases. The goal of GA-MITS is therefore to maximise the fitness function, F.

A test set that achieves a fault coverage of $N(C)_{\max}$ must always be allocated a higher fitness than one which covers less faults. GA-MITS is only searching for test sets which cover the desired number of faults. With this in mind, the magnitude of F_2 must always be greater than F_1. However because of the $N(T)$ term in the denominator of F_2, this may not always be the case. This is the reason why the multiplier k is included in F_2. To ensure F_2 is always greater than F_1, k must be greater than $N(T)_{\max}$. In practice k is equal to 2 x $N(T)_{\max}$.

Table 4. 6 shows a selection of test sets with their corresponding fitness values, for the circuit in Figure 4.2. In this example, $N(C)_{\max}$ = 10, $N(T)_{\max}$ = 8 and k = 20. From this table, test set 2 achieves maximal fault coverage with 6 test vectors and its fitness is 33.33. However, in Test Set 4, the same fault coverage is achieved but with 4 test vectors and its fitness is evaluated as 50. So, the fitness function is behaving as required and is allocating higher fitnesses to smaller test sets, for a given fault coverage.

Test Set	$N(C)$	$N(T)$	Fitness
1. (000, 111)	4	2	$F = N(C) = 4$
2. (000, 001, 010, 011, 100, 110)	10	6	$F = 20.{}^{10}\!/_{6} = 33.33$
3. (000, 001, 010, 101, 110)	10	5	$F = 20.{}^{10}\!/_{5} = 40$
4. (010, 100, 101, 110)	10	4	$F = 20.{}^{10}\!/_{4} = 50$

Table 4.6 *A selection of test sets and their associated fitness values for the circuit in Fig. 4.2.*

4.6.4 Parent Selection Scheme used in GA-MITS

In both natural and artificial domains, selection is one of the driving forces of evolution. Evolution relies on achieving a balance between exploitation and exploration and when designing a GA, a selection scheme that is able to provide this balance should be chosen. The relative merits of various selection schemes have already been discussed in the previous chapter, highlighting an important property of all selection schemes viz. selection pressure. With selection pressure in mind and through experimentation it was decided to use an exponential ranking selection scheme within GA-MITS.

Once the 'raw' fitness of each chromosome has been calculated using the fitness function described above, the population of N chromosomes is sorted into ascending order of rank, where Rank(N) is the fittest individual, Rank(N -1) is the second fittest down to Rank(1) which is the weakest chromosome in the population. Once these ranks have been established the values of the following exponential ranking function $f(i)$ [52], are ascribed to each chromosome according to rank.

$$f(i) = \frac{(s-1)}{\exp(N.\ln s) - 1} \times \exp((N-i).\ln s)$$

where,

N = number of chromosomes in population

s = selection constant, $0 < s < 1$

i = rank

The constant s is a very important parameter as it determines the selection pressure in the above scheme. The selection pressure is proportional to (1 - s) therefore the greater the value of 's' the greater the likelihood of the weaker individuals in the population being selected. Table 4.7 gives the rank fitness values for the function $f(i)$ for different values of s and a population size of 10. It is to be noted that the sum (over every member of the population) of the rank fitness values for each value of s is 1.

Rank	$f(i)$						
(i)	s = 0.3	s = 0.4	s = 0.5	s = 0.6	s = 0.7	s = 0.8	s = 0.9
1	0.000014	0.000157	0.000978	0.004056	0.012458	0.030073	0.059482
2	0.000046	0.000393	0.001955	0.006759	0.017797	0.037591	0.066091
3	0.000153	0.000983	0.003910	0.011266	0.025424	0.046988	0.073435
4	0.000510	0.002458	0.007820	0.018776	0.036321	0.058735	0.081594
5	0.001701	0.006145	0.01564	0.031293	0.051887	0.073419	0.090660
6	0.005670	0.015362	0.031281	0.052155	0.074124	0.091774	0.100734
7	0.018900	0.038404	0.062561	0.086926	0.105891	0.114718	0.111926
8	0.063000	0.096010	0.125122	0.144876	0.151273	0.143397	0.124363
9	0.210001	0.240025	0.250244	0.241460	0.216104	0.179246	0.138181
10	0.700004	0.600063	0.500489	0.402433	0.308721	0.224058	0.153534

Table 4.7 *Exponential rank fitness values for function,* $f(i)$ *over a range of values of* s.

As can be seen from the above table, the value of s controls the selection pressure in this exponential ranking scheme. For $s = 0.3$, there is little chance of the weakest individual being selected relative to the strongest chromosome. The probability of the chromosome of Rank(1) being selected as a parent is four orders of magnitude less than that of chromosome of Rank(10). Conversely, for $s = 0.9$ the weakest individual has a significantly greater chance of selection as there is only single order of magnitude between the its probability of selection and that of the fittest individual. But as mentioned earlier one must select a scheme or value of s that achieves an acceptable balance between exploration and exploitation. With this in mind, it is the view of the author that the values $s = 0.4$, $s = 0.5$ and $s = 0.6$ provide this balance, the exact choice of this parameter being made only after experimentation with the particular application. After much experimentation, the value of s chosen for GA-MITS was 0.65. This choice will be discussed later in the chapter. Once exponential rank fitnesses are assigned to chromosomes, parents are selected using roulette wheel selection, with replacement.

4.6.5 Selection of GA Parameters

GA-MITS uses two-point crossover and the crossover rate is set at 95%. The mutation operator used is simple bit-wise mutation with a mutation rate of 0.7%. These selections were largely made through experimentation although theoretical issues were used as a guide. The results of the experiments will be presented later in this chapter. The population size was set at 100 chromosomes (regardless of circuit size) in each generation and the number of generations in each run of GA-MITS was 100. Again, these parameters were set largely through experimentation and with some theoretical guidance. The reasoning behind the selection of these settings will be discussed later in this chapter.

4.6.6 The Use of Inoculation and Elitism.

The vast majority of GA literature advocates the use of a purely randomly generated initial population of chromosomes. Over recent years however the process of inoculation [53], whereby the initial population includes some non-random chromosomes as well as purely random ones, has emerged from the literature. The exact nature of these non-random chromosomes is problem specific and requires a thorough understanding of the problem to be optimised and the fitness function. In the case of test set minimisation it was decided that to speed-up the algorithm, a single chromosome containing gene values of 1 throughout would be inserted into an otherwise purely random initial population. Reductions in run time of up to 50% have been achieved by adopting this method. This inoculation procedure was designed specifically for GA-MITS.

Elitism is a popular technique of ensuring that the fittest individual in the current population is carried forward into the next generation. This copying process ensures that this valuable genetic information is not lost through the crossover and mutation operators. In GA-MITS, two copies of the fittest chromosome are carried over into the next generation.

4.7 Circuits used to Generate Fault Matrices for GA-MITS

All test set minimisation results were obtained using fault matrices generated by the team at Tallinn Technical University. There are two distinct groups of fault matrices, for two different classes of combinational circuits. The first group of fault matrices correspond to a family of simplified RISC (Reduced Instruction Set Computer) processors. These circuits were developed at Tallinn as a test bed for their test pattern generation algorithms. Table 4.7a gives each circuit's profile along with the number of possible faults. For each RISC processor, two fault matrices were generated using two different test pattern generation algorithms. The first ATPG is based on *alternative graphs* [51] and the second based on the concept of random pattern generation [33].

The second group of fault matrices used in this work correspond to the ISCAS-85 benchmark circuits [54]. They were developed as a test bed for combinational test pattern generation algorithms and were presented at the *International Symposium on Circuits and Systems*, 1985. They have become the international standard for evaluating ATPG algorithms, cited by the vast majority of ATPG papers over the last 18 years or so. Table 4.7b displays the circuits along with their fault statistics.

RISC Processor	Number of Gates	Number of Inputs	Number of Outputs	Total Number of stuck-at faults
4 bit	603	42	5	612
8 bit	1195	74	9	1168
16 bit	2379	138	17	2240
32 bit	4747	266	33	4402

(a)

ISCAS-85 Benchmark Circuit	Number of Gates	Number of Inputs	Number of Outputs	Total Number of Stuck-at Faults
c432	160	36	7	616
c499	202	41	32	1202
c880	383	60	26	994
c1908	880	33	32	1732
c2670	1193	233	140	2626
c3540	1669	50	22	3296
c5315	2307	178	123	5424
c6288	2406	32	32	7744
c7552	3512	207	108	7104

(b)

Table 4.7 (a) *Simplified family of RISC processors - circuit size and fault data.* **(b)** *ISCAS-85 benchmark circuits - fault data.*

For each ISCAS-85 circuit three different sets of fault matrices, derived by three different ATPGs, were used to obtain minimal test sets by GA-MITS. The three ATPG algorithms were: an implementation of PODEM, a random test pattern generator and one based on genetic algorithms.

GA-MITS therefore was tested on a relatively wide range of fault matrices and on an internationally recognised set of benchmark circuits. For a given circuit, different test pattern generation techniques produced significantly different fault matrices. For the RISC circuits for example, the test sets produced by the random generator were notably larger than those generated by the functional method.

4.8 Typical Performance of GA-MITS

An example test set minimisation problem for the ISCAS-85 circuit c2670 will now be examined. This circuit contains 2626 possible stuck-at faults and the PODEM based ATPG tool was used to generate the fault matrix, producing the following parameters,

$$N(T)_{\max} = 160 \text{ test vectors}$$

$$N(C)_{\max} = 2508 \text{ faults.}$$

Figure 4.5 is a graph representing the results of the minimisation process as GA-MITS creates 100 generations of chromosomes, each generation containing 100 test sets. Tabulated results, for the first 47 generations, corresponding to this graph are given in Table 4.8

Recall that the objective of GA-MITS is to minimise the number of test vectors in each test set subject to a predetermined level of fault coverage. Inoculation of the initial population ensures at least one chromosome achieves the required fault coverage and it is highly likely that this relatively fit individual will propagate its genes through subsequent generations. The evolution of subsequent minimal test sets (in this context 'minimal' refers to the smallest test set achieving the required fault coverage in a given generation) seems to suggest that they are the offspring of this biased chromosome. From generation to generation the reduction in test set size of the minimal test set is, on the whole, steady and incremental. That is, only a small reduction in test set size, typically from 1 to 5 test vectors is achieved from one generation to another. In later generations, GA-MITS seems to get stuck on local maxima[3], only achieving small improvements after a large number of generations.

From the graph and table below, it may be seen that in early generations, GA-MITS is steadily reducing the size of the minimal test sets found from generation to generation. The improvements in test set sizes are most dynamic in the first 30 generations or so, as can be seen from the gradient of the curve in Figure 4.4. Although GA-MITS does not reduce the test set size from every generation to another large reductions occur from time to time. For example, from generation 0 to 1 a reduction in test set size of 5 test vectors was achieved and between generation 8 to 9 a reduction of 3 vectors was achieved. Between generations 19 and 36, the curve begins to flatten and improvements in test set size only occur by 1 or 2 test vectors at a time. From generation 40 to 43 GA-MITS seems to have prematurely converged on a test set containing 120 test vectors and it is not until generation 44 that a smaller a test

[3] Although GA-MITS is searching for test sets of *minimal* size, it is actually *maximising* the fitness function, *F*.

set, containing 119 test vectors, has been located. In the remaining generations no improvement in test set size is achieved, strongly indicating that the GA has converged.

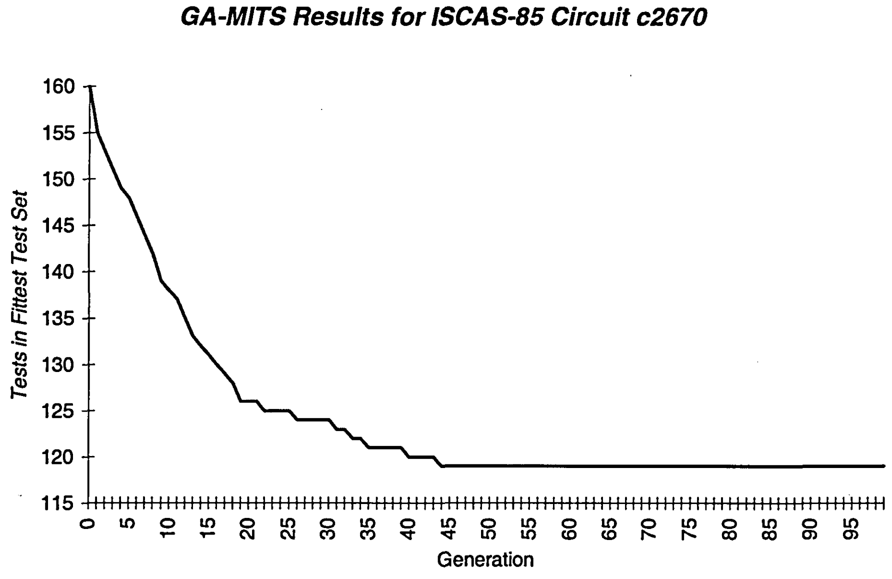


Figure 4.5 *Graph of number of test vectors in the minimal test set vs. generation for the ISCAS-85 circuit c2670. The fault matrix data was generated by PODEM. The minimal test set was located at generation 44 and comprised 119 test vectors covering all 2508 testable faults.*

Generation	Test Vectors in Minimal Test Set	Generation	Test Vectors in Minimal Test Set	Generation	Test Vectors in Minimal Test Set
0	160	**16**	130	**32**	123
1	155	**17**	129	**33**	122
2	153	**18**	128	**34**	122
3	151	**19**	126	**35**	121
4	149	**20**	126	**36**	121
5	148	**21**	126	**37**	121
6	146	**22**	125	**38**	121
7	144	**23**	125	**39**	121
8	142	**24**	125	**40**	120
9	139	**25**	125	**41**	120
10	138	**26**	124	**42**	120
11	137	**27**	124	**43**	120
12	135	**28**	124	**44**	119
13	133	**29**	124	**45**	119
14	132	**30**	124	**46**	119
15	131	**31**	123	**47**	119

Table 4.8. *Corresponding minimal test set data for Figure 4.4. Only the first 47 generations are given above.*

The above is typical behaviour of GA-MITS with inoculation. The majority of reductions achieved in test set size take place in the first 20 to 30 generations. The remaining generations seem to refine the final solution by approximately 1 to 5%. If this run of GA-MITS was left for another 100 generations perhaps it would achieve further reductions in test set size, but this is not guaranteed as it may have prematurely converged on a sub-optimal maximum or it may have located a global maximum. In general GAs do not guarantee that they have located one of the globally optimal solutions. In the case of GA-MITS, the initial test sets have been significantly reduced in size, whether these are *the* optimal solutions in open to debate.

As a further example, the curve given in Figure 4.6 illustrates the results achieved by GA-MITS for another ISCAS-85 circuit, c432. Again, the behaviour of the algorithm is similar to that for the circuit c2670, with an initial steep negative gradient indicating steady reductions in test set size. The curve begins to flatten by around generation 20 until finally the GA converges on a test set of size 55 at generation 29.

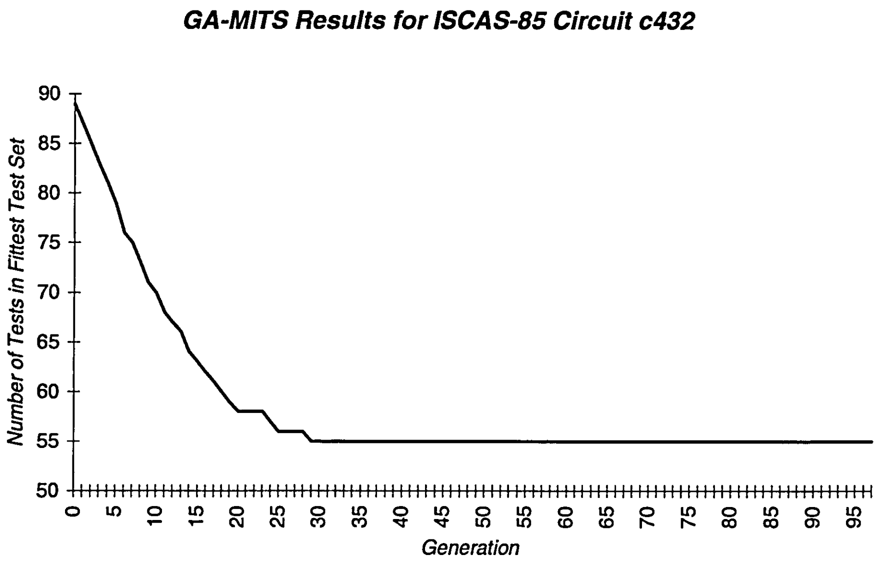


Figure 4.6 *Graph of number of test vectors in the minimal test set vs. Generation for the ISCAS-85 circuit c432.*

The final output of GA-MITS to the user is an on-screen message indicating the size of the final minimal test set, the fault coverage of this test set and the chromosome corresponding to the minimal test set. As an illustration, the final output for the above test set minimisation problem for the circuit c2670 whose fault data was generated by PODEM is given below.

```
Fittest   chromosome   properties:   tests=119    fault
coverage=2508

The test is as follows:

10100000010100101011001111101110010111001010111 1001010
110101011011011111111111110111010111110011111111111
11111111111111111011100101101011111111111111111111
1
```

It must be remembered that a value of '1' at the n^{th} gene (or locus) means that that the n^{th} test vector in the original test set is present in the final test set. A '0' signifies the absence of the corresponding test vector.

4.9 Minimisation Results for a Family of Simplified RISC Processors

Table 4.9 and 4.10 below give the minimisation results for the RISC processors using a functional ATPG tool based on Alternative Graphs [51] and a random test pattern generator respectively. In Table 4.9 it may be seen that for each of the RISC processors, the ATPG produced test sets containing 63 test vectors (the fault coverage for each of these test sets is given in the third column). From the final column in this table, GA-MITS is seen to achieve a reduction test set size of 62% for the 4 bit processor and a reduction of 55% for each of the other three processors.

RISC Processor	Total No. Faults	No. Faults Covered by ATPG	ATPG Test Set Size	Test Set Size achieved by GA-MITS	Reduction in test set size %
4 bit	612	611	63	24	61.905
8 bit	1168	1167	63	28	55.556
16 bit	2240	2239	63	28	55.556
32 bit	4402	4401	63	28	55.556

Table 4.9 *RISC processor minimisation data. The corresponding fault data was generated using Alternative Graphs [51].*

The fault data associated with Table 4.10, derived by a random test pattern generator, can be seen to be significantly smaller than those generated by the functional ATPG (62% smaller for the 4 bit processor). For these test sets, GA-MITS achieves reductions in test sets between 15 and 7%, which are significantly smaller than for the functionally generated test sets.

RISC Processor	Total No. Faults	No. Faults Covered by ATPG	ATPG Test Set Size	Test Set Size achieved by GA-MITS	Reduction in test set size %
4 bit	612	611	24	22	8.333
8 bit	1168	1167	34	29	14.706
16 bit	2240	2239	39	36	7.692
32 bit	4402	4401	44	41	6.818

Table 4.10 *RISC processor minimisation data. The corresponding fault data was generated using a random test pattern generator.*

A point to note from the above two tables is that, for a given circuit, the test sets generated by the functional ATPG are significantly larger than the those generated by the random ATPG. However after minimisation, all but one of the 'functional' test sets are actually smaller than the minimised, randomly generated ones. The average reduction in test set size for the functionally generated test sets is approximately 57% and approximately 10% for the randomly generated test sets. Why are the original 'functional' test sets larger than the 'random' ones and why do the 'functional' test sets minimise significantly more than the random ones?

The reason why the randomly generated test sets are smaller than the functional ones is because there is inherent minimisation taking place in the test pattern generation process. As test vectors are generated a new vector is only accepted into the test set if it covers a fault that is not covered by the tests in the test set so far. The functional ATPG in its barest form (as used in here) generates tests on a node by node basis without necessarily taking note of whether a previous test has already covered that fault. Although a smaller, randomly generated test set may be far from a global minimum for a given circuit, it will contain far less overlap in fault coverage, or *redundancy*, than its functionally generated counterpart. Since GA-MITS is using this redundancy to reduce the size of the test sets and there is not much to exploit in the 'random' case, it is highly unlikely that the final minimised test set will be a global minimum.

Since the functionally generated test sets are relatively large, it is safe to suggest that there is much more overlap in fault coverage. Since there are more test vectors in this test set, it is more probable than for the 'random' case that the test vectors comprising a minimal test set will be present. There is more redundancy for GA-MITS to exploit and it therefore locates smaller test sets than for the 'random' case.

The observations made in the previous section are only for a limited number of simple cases. These simplified RISC processors were developed for the initial testing of ATPG algorithms and are not representative of practical designs. The nature of minimisation results for more realistic circuits will now be explored.

4.10 Minimisation Results for the ISCAS-85 Benchmark Circuits

The ISCAS-85 benchmark circuits approach the size and complexity found in practical circuit designs. If GA-MITS is able to minimise test sets corresponding to these circuits, it may be confidently assumed that the algorithm has a genuine role alongside today's ATPG tools.

In what follows, the term *original test set* refers to a test set generated by some ATPG that has not undergone minimisation by GA-MITS or any other static minimisation algorithm. The term *minimised test set* refers to a test set that has undergone minimisation by GA-MITS or other algorithm.

Three different ATPGs have been used to generate test sets for GA-MITS. The minimisation results for each ATPG are tabulated below. It must be noted that in these results tables an Asterisk denotes that the minimised test set is known to be a global minimum given the original test set generated by some ATPG method. In some cases the team at Tallinn were able to prove that a particular test set, whether minimised or not, was a global minimum by examining the test vectors present in that set. They used the concept of *unique patterns* which are defined as test vectors that cover faults uncovered by any other vectors present in the test set (the terms *test patterns* and *test vectors* will be assumed to be interchangeable). These patterns are obviously essential to a test set which has to cover all $N(C)_{\max}$ faults. Therefore if a test set, be it an original or minimised test set, comprises vectors that are all unique patterns then that test set is a global minimum. Furthermore, they are able to prove that, a test set minimisation algorithm has reached a global minimum if the number of patterns in the minimised test set is less than three patterns larger than the number of unique patterns in the original test set [56].

ISCAS-85 Circuit	Total No. Faults	No. Faults Covered by ATPG	ATPG Test Set Size	Test Set Size achieved by GA-MITS	Reduction in test set size %
c432	616	573	89	55	38.202
c499	1202	1194	140	100	28.571
c880	994	994	70	52*	25.714
c1908	1732	1722	144	122*	15.278
c2670	2626	2508	160	119	25.625
c3540	3296	3146	201	145	27.861
c5315	5424	5364	178	108	39.326
c6288	7744	7693	41	33	19.512
c7552	7104	6973	276	198	28.261
			Ave=144.33	*Ave. = 103.55*	*Ave. = 27.594*

Table 4.11 *Minimisation results for ISCAS-85 benchmark circuits. The corresponding fault data was generated using an implementation of PODEM.*

ISCAS-85 Circuit	Total No. Faults	No. Faults Covered by ATPG	ATPG Test Set Size	Test Set Size achieved by GA-MITS	Reduction in test set size %
c432	616	573	51	46*	9.804
c499	1202	1194	86	85*	1.163
c880	994	994	46	38	17.391
c1908	1732	1723	121	110*	9.090
c2670	2626	2508	112	87	22.321
c3540	3296	3149	155	138	10.968
c5315	5424	5364	115	99	13.913
c6288	7744	7693	21	21*	0
c7552	6937	6867	192	156	18.750
			Ave. = 99.89	*Ave. = 86.67*	*Ave. = 11.49*

Table 4.12 *Minimisation results for ISCAS-85 benchmark circuits. The corresponding fault data was generated using an ATPG based on genetic algorithms.*

ISCAS-85 Circuit	Total No. Faults	No. Faults Covered by ATPG	ATPG Test Set Size	Test Set Size achieved by GA-MITS	Reduction in test set size %
C432	616	573	51	46*	9.804
C499	1202	1194	86	86*	0
C880	994	994	63	45*	28.571
c1908	1732	1723	132	112	15.152
c2670	2626	2389	107	75	29.907
c3540	3296	3149	167	143	14.371
c5315	5424	5364	132	106	19.700
c6288	7744	7693	24	22*	8.333
c7552	6937	6851	249	164	34.137
			Ave. = 112.33	*Ave. = 88.78*	*Ave. = 17.78*

Table 4.13 *Minimisation results for ISCAS-85 benchmark circuits. The corresponding fault data was generated using a random ATPG.*

From the above three tables it may be seen that for a given circuit and fault coverage, the fault data produced by different ATPGs varies considerably. In descending order of original test set size, on average, PODEM produces the largest test sets, then the random generator followed by the GA based generator. All three ATPGs contain some test set minimisation heuristics, each with its own degree of success. Closer examination of the implementation of each ATPG will help explain the variance in the test set size.

The test sets produced by PODEM contained, on average, 144.33 vectors. Within this particular implementation of PODEM, once each test vector has been generated for a given fault (known as the *primary* fault) fault simulation is invoked to determine whether other faults (*secondary* faults) are also detected by the vector. If any secondary faults are covered, PODEM will not attempt to explicitly generate vectors for these faults, thus helping to reduce the number of test vectors in the final test set. Without the additional fault simulation step which aids test set minimisation, much larger test sets would be produced as test pattern generation would be attempted for every fault in a circuit.

The random ATPG produced test sets containing, on average, 112.33 test vectors. The minimisation heuristic within random ATPGs has already been discussed in Section 4.4. Briefly, test vectors are randomly generated and fault simulation is used to determine the fault coverage of the vector. As each new vector is generated, it is only accepted as part of the test set if it covers faults that have not been covered by the test vectors in the test set so far. Therefore *dynamic* test set minimisation (minimisation as each test vector is generated) occurs in an attempt to reduce as much overlap in fault coverage as possible. This method of test pattern generation and minimisation does however result in a fair amount of redundancy. The reason for this is that a vector will be accepted into the test set if it covers a least one new fault. It is highly likely that this vector will also cover faults already covered by other vectors in the test set so far. Thus, there will exist a certain amount of redundancy so the test set will not be as compact as it could be.

The smallest test sets were generated by the GA based ATPG which yielded test sets containing on average 99.89 vectors. The GA puts greater effort into finding vectors that detect most new faults than any of the other schemes mentioned above. The fitness function [55] is formulated such that it rewards those test vectors in proportion to the number of new faults that it covers and penalises those that only cover previously covered faults. This coupled with the strong searching capabilities of a GA result in significantly smaller test sets than both PODEM and the random generator.

For a given circuit and fault coverage, it can be seen from the above tables that different test pattern generators will produce different test sets. GA-MITS will never improve on the fault coverage of these original test sets as it does not generate any new vectors or change, in any way, the constituent vectors. What GA-MITS will do is search amongst the original test sets for any redundancy that may exist in the hope of locating the smallest test set covering all $N(C)_{\max}$ faults. A reduction in test set size by GA-MITS is not always guaranteed as there may be no redundancy contained in a test set. The level of redundancy present in a test set not only depends on circuit design but also on the particular ATPG tool used. Some tools 'doctor' the test vectors more than others in an attempt to reduce test set size.

For the fault data produced by the PODEM generator GA-MITS was able to reducee the test sets on average by 27.59%. The greatest reduction in test set size was for the circuit c5315, for which a reduction of 39% was achieved. The smallest reduction achieved was 15% corresponding to the circuit c1908. For the random generator, an average reduction in test set size of 17.78% was achieved, with the greatest reduction of 30% for the circuit c2670 and the minimum reduction in test set size of 0% for the circuit c499. This original test set was proven to be the global minimum, containing no redundancy. For the GA based generator the average reduction in test set size was 11.49%. For this group of fault data, the test set corresponding to circuit c2670 was minimised the most with a reduction of 22%. The smallest reduction was 0% for the test set corresponding to the circuit c6288 which again was proven to be the global minimum.

It is apparent therefore that there is a strong correlation between the original test set size and the amount of minimisation achieved by GA-MITS. On the whole, the larger the original test sets, the larger the percentage reduction in test set size. The test sets generated by PODEM were larger than those generated by the other two methods and on average these test sets were reduced by 27.59%. The smallest test sets generated were by the GA based ATPG. The average reduction in test set size achieved by GA-MITS was 11.49%, significantly smaller than the reductions achieved for fault data corresponding to both PODEM and the random generator. These results follow on from those obtained in Section 4.9. Just as was stated for the RISC processors, the above results can be explained by the greater degree of redundancy present in the larger, original test sets. This greater scope for minimisation is exploited by GA-MITS resulting in larger percentage reductions in test set size.

Within each test set there may exist a number of globally minimal test sets[4]. As already stated, the degree to which GA-MITS is able to minimise an original test set depends on the amount of redundancy within it. Since it has been shown that for a given circuit different ATPGs produces different test sets, a minimised test set is only minimal relative to the original test set. It may not be the smallest possible test set for that circuit per se. The only way to guarantee one has generated *the* globally minimal test set for a circuit, regardless of ATPG tool, is to exhaustively generate test vectors for every possible fault in the circuit and then to exhaustively evaluate every possible test set to determine the minimal test set. This is obviously an impractical approach.

Given then the three ATPG tools coupled with GA-MITS, what are the smallest minimised test sets and which tool generated the original test set? Let us examine the results obtained for the circuits c5315 and c1908 given below in Table 4.14. The size of the smallest minimised test set for the circuit c5315 is 99 vectors for the test set originally generated by the GA based TPG. The fault matrices corresponding to the PODEM and the random ATPG were minimised to 108 and 106 tests respectively. For the circuit c1908 the smallest test set found by

[4] It is often the case that there are several globally, minimal test sets of equal size and fault coverage but obviously containing different test vectors.

	PODEM			Random TPG			GA TPG		
Circuit	ATPG Test Set Size	Minimal Test Set found by GA-MITS	% reduction due to GA-MITS	ATPG Test Set Size	Minimal Test Set found by GA-MITS	% reduction due to GA-MITS	ATPG Test Set Size	Minimal Test Set found by GA-MITS	% reduction due to GA-MITS
c1908	144	122*	15.278	132	112	15.152	121	110*	9.091
c5315	178	108	39.326	132	106	19.697	115	99	13.913

Table 4.14. *Minimisation results for circuits c1908 and c5315.*

GA-MITS was within the fault matrix data generated by the GA based ATPG. This test set, comprising 110 vectors, was 12 and 2 vectors smaller than the those corresponding to the PODEM and random generators respectively. The smallest minimised test sets all corresponded to those originally generated by the GA based ATPG. This holds true for the test sets of all but one circuit, c432. For the circuit c432 the random and GA-based generators produced exactly the same test vectors which reduced down to exactly the same test set, containing 46 vectors.

Examining the percentage reductions in test set sizes, overall the smallest reductions correspond to the test sets originally generated by the GA based TPG. This is counter to the results obtained for the RISC circuits. In those minimisation results, larger original test sets resulted in the greatest percentage reduction and also, on the whole, in the smallest minimised test sets for a given circuit. The conclusion drawn from this result is that there would seem to be greater redundancy in the larger, less doctored, original test sets. This turns out not to be the case for the ISCAS-85 circuits and the three ATPGs used, which again reflects favourably the relative quality of the GA based ATPG method. This ATPG simply puts more effort into finding high quality vectors (that each cover as many faults as possible) and more compact original test sets. The GA based ATPG was far superior than both PODEM and the random method.

If there is any redundancy to be exploited within a test set it can be confidently expected that GA-MITS will do so to find the smallest possible test sets for the required fault coverage. This confidence is due to the fact that GA-MITS did locate the smallest test sets in each of the cases where they were known. This strongly suggests, although does not prove, that GA-MITS will find the smallest test sets in all cases. Returning to circuit c1908, it has been proven that the minimised test set corresponding to the PODEM generator, containing 122 vectors, is the globally minimal test set(s) for that original set containing 144 vectors. For the fault data corresponding to the same circuit but generated by the GA based method, the minimal test set found by GA-MITS contains 110 vectors and has also been proven to be a global minimum amongst the original test set of size 121 vectors. Similar results can be seen for the circuits c499 and c880. For c880 amongst the original GA generated fault data, GA-MITS located a test set containing 38 vectors. For the random fault data a minimum test set containing 45 test vectors was located by GA-MITS which was again proven to be a global minimum for that fault data. It must be reiterated that GA-MITS is only as good as the TPG tool that generated the original test set and global minima will exist amongst all fault matrices.

Out of the 27 fault matrices presented to GA-MITS (three ATPG tools each generating test sets for 9 ISCAS circuits), 10 of the minimised test sets were proven to be global minima. Of the remaining 17 minimised test sets, the Tallinn team were unable to prove whether these were global minima because each minimised test set contained three test patterns or more than the number of unique patterns. These minimised test sets may well be global minima but at this stage it cannot be proven.

So how does GA-MITS perform relative to other test set minimisation algorithms? The only way to make valid comparisons is to present identical test sets to different algorithms. The Tallinn team have also developed a test set minimisation algorithm which is based on the concept of Bipartite graphs [56]. Throughout the development of their algorithm the results obtained by GA-MITS have been used a benchmark for their algorithm. During the early stages of development the Bipartite based algorithm performed poorly compared to GA-MITS. As the results obtained by GA-MITS were improved and fed back to Tallinn their algorithm was continually refined until eventually their results were identical to those obtained by GA-MITS. This developmental process has been fully acknowledged in the above stated reference.

A popular minimisation technique mentioned in Section 4.4 was 'Reverse Order Fault Simulation' (ROFS), used in a number of popular ATPG tools [39 - 42]. Table 4.15 compares the minimisation results obtained by GA-MITS and reverse order faults simulation. The original test sets corresponding to this fault data were generated using the PODEM TPG, again by the Tallinn team.

Circuit	Original test set size	Minimised test set size obtained by ROFS.	Minimised test set size obtained by GA-MITS	% reduction using GA-MITS in comparison to ROFS
c432	89	89	55	38.202
c499	140	137	100	27.007
c880	70	70	52	25.714
c1908	144	142	122	14.085
c2670	160	160	119	25.625
c3540	201	198	145	26.677
c5315	178	173	108	37.572
c6288	41	40	33	17.500
c7552	276	275	198	28.000
		Ave. = 142.67	*Ave. = 103.55*	*Ave. = 26.71*

Table 4.15. *Comparison of test set minimisation results obtained by reverse order fault simulation and GA-MITS.*

As can be seen from this table, GA-MITS is a far superior minimisation algorithm than the traditional reverse order fault simulation method. Although simple in its implementation, the results obtained by ROFS are very poor. GA-MITS was able to minimise test sets by up to 38.20% more than ROFS and the smallest relative reduction was a significant, 14.08%. In a similar manner to ROFS, GA-MITS is designed as a 'bolt-on' post processor to existing test pattern generation tools and although more sophisticated, it is felt that GA-MITS is as easy to incorporate into existing ATPG tools.

The most important factor of all test set minimisation algorithms is the size of the absolute minimal test set. The absolute minimal test set in this context is defined as the size of the smallest test set found for a given circuit, regardless of ATPG tool and minimisation heuristic(s). It is the size of the absolute minimal test set that will after all, determine the choice of CAD tool adopted by a manufacturer. It has already been mentioned that exhaustive ATPG and minimisation is impractical but is the only known way to guarantee one has generated the minimal test set for a circuit. However, comparisons can be made between the results of different TPGs and/or minimisation algorithms.

The ISCAS-85 benchmark circuits have been in the public domain for the last 18 years or so. The vast majority of test pattern generation papers and technical reports use these circuits, enabling comparisons to be made. So how does GA-MITS perform relative to the results given in the literature? Unfortunately direct, meaningful comparisons between GA-MITS and other algorithms cannot be made. The only possible way is to present identical test sets to GA-MITS and other minimisation algorithms or to present test sets generated by ATPGs containing minimisation heuristics to GA-MITS to see whether any redundancy exists that can be exploited. Several research groups were approached including the authors of [25] and [44], requesting fault data from their particular ATPG tools. Unfortunately, due to personnel leaving the groups or the lack of man power to reproduce the results, the author was unable to receive any data from these researchers. Another difficulty in making comparisons without identical fault data is due to different fault coverages obtained and different ways of pre-processing the faults. The Tallinn group use fault collapsing giving rise to a different number of faults in the fault list for the ISCAS circuits whereas other groups do not perform this initial step.

Nonetheless, it is felt that comparing the sizes of test sets obtained by other methods with those obtained by GA-MITS is still a useful exercise. Table 4.16 below illustrates these comparisons. Although the different fault coverages of each method are not given in the table, they are more or less identical. The reader is directed towards the literature for further details.

Circuit	[57]	SOCRATES [41]	GATE [44]	SMART + FAST [33]	COMPACTEST [45]	GA-MITS (best)
c432	48	60	37	n/a	n/a	46
c499	53	55	n/a	n/a	n/a	86
c880	31	63	n/a	39	30	45
c1908	157	122	n/a	151	142	112
c2670	110	122	n/a	78	67	75
c3540	154	173	122	178	111	143
c5315	126	150	n/a	97	56	106
c6288	37	32	n/a	40	16	22
c7552	n/a	235	n/a	143	87	164

Table 4.16 *Comparison of test set sizes for ISCAS-85 benchmark circuits generated by different ATPGs/minimisation algorithms. The shaded figures denote the smallest test set sizes found in the literature. n/a denotes that this result was unavailable in the literature.*

A thorough literature search revealed the ATPG methods/minimisation algorithms in the above table as being some of the most competitive in terms of the size of the absolute minimal test set. Many of these ATPGs contain some very sophisticated test pattern generation techniques in addition to some very competent minimisation heuristics. GATE for example employs three levels of compaction. On the whole, COMPACTEST produces the smallest test sets but GA-MITS coupled with Tallinn's GA based ATPG actually produced the smallest test set for the circuit c1908.

It must be reiterated that the performance of GA-MITS is limited to the quality of the ATPG that generated the original test set. Based on the results so far the author is confident that given any redundancy in the test sets generated by the methods given in Table 4.16, GA-MITS would be able to search out more compact test sets. It is unfortunate at this stage that this data was unavailable and that these experiments could not be performed. It is hoped however, that such results will be obtained in the near future.

An important property of any new algorithm is its computational complexity, i.e. processor time and computer memory required for its implementation. Increasingly, the cost of computer memory and processor time has been decreasing while the processing power of modern day computers has increased the rapid pace predicted by Moore's Law. Therefore the limits of computational complexity are continually being challenged as algorithms which were previously considered too costly are now easily implemented. However, the adoption of a new algorithm must compete with what is currently the state-of-art either in terms of solution quality or complexity. The context in which an algorithm is to be employed is also a deciding factor when it is being considered for use. Is the application domain time sensitive as, for example algorithms employed in daily weather prediction software?

And so onto the competitiveness of GA-MITS and the context in which it may be applied. GA-MITS is a static minimisation algorithm (applied once a test set has been generated) designed as an easy-to-incorporate post processor for existing ATPG tools. The sole purpose of test set minimisation is to save circuit test and fabrication costs. Smaller test sets require less post production test time and help reduce the amount of memory required to store test vectors in a BIST scheme. The application of GA-MITS is a one-off cost after the circuit design has been finalised (cheaper methods can be employed during the design phase to measure testability etc.). Since manufacturers may produce thousands and possibly millions of units of a particular design, even the smallest reduction in test set size may help cut test and fabrication expenditure.

In the context of run-time, the fact that GA-MITS is a relatively slow algorithm should not hinder its possible role in today's CAD world. For the largest ISCAS-85 circuit, c7552, and an original test set generated by PODEM GA-MITS took a maximum of approximately 2.5 hours to run and achieved a reduction of 28%. The minimisation effort is a one-off cost and this execution time would be very quickly recouped by manufacturers faced with producing and testing thousands of circuits. The execution time of the algorithm is likely to drop as processing power of computers increases. All experiments carried out with GA-MITS have been run on a slow, 100 MHz Pentium (processor clock speed) based personal computer. This hardware platform is far removed from what is today's state-of-the-art technology such as powerful engineering workstations which are optimised for mathematically intensive tasks. The upper execution time of around 2.5 hours will no doubt be reduced if GA-MITS were run on such hardware and will continue to drop as processing speeds of computers increase. Additional reductions in run time could be achieved as a result of optimising the various algorithms within GA-MITS and the source code. No real effort has been placed in making the algorithm more efficient and run-time savings are certainly attainable. Furthermore in terms of pure GA research, the development of parallel implementations of genetic algorithms [58] could also be used to help reduce run-time.

Within GA-MITS itself for a given population size and number of generations in a run, the feature dominating the execution time of the algorithm is the evaluation of the fitness function which requires checking the fault matrix for the fault coverage of each test vector. This time is proportional to the size of the fault matrix which is of size,

$$N(C)_{\max} \times N(T)_{\max}$$

where $N(C)_{\max}$ and $N(T)_{\max}$ is the fault coverage and number of constituent test vectors of the original test set respectively. The standard GA operations of the algorithm such as crossover and mutation are proportional to $N(T)_{\max}$. The memory requirements for GA-MITS are also largely proportional to $N(C)_{\max} \times N(T)_{\max}$ when a new generation over-writes the previous generation in memory.

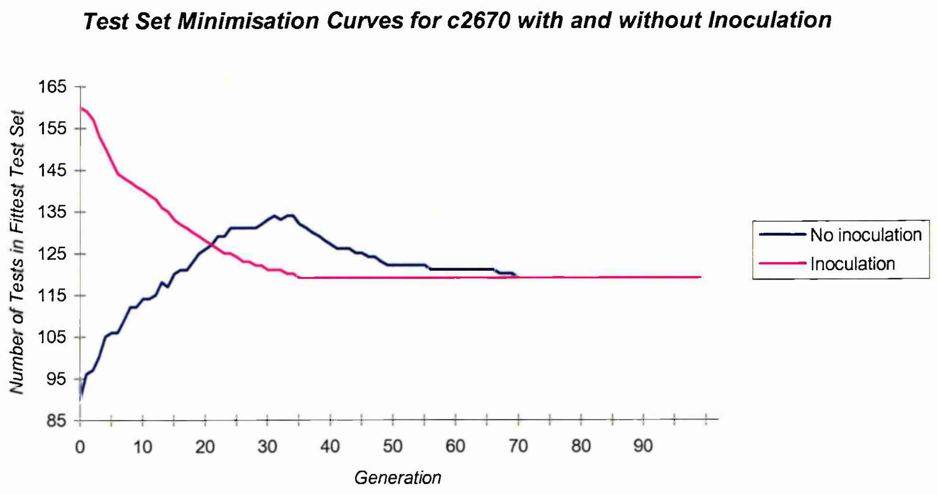
It is increasingly the view of the CAD community [4, 59] that the computational complexity of algorithms is becoming less of an issue with the pace of change in the computer world. The emphasis is on solution quality when comparing different algorithms. It is on this important front that GA-MITS is certainly excelling and will compete with current methods.

4.11 GA-MITS Design Issues

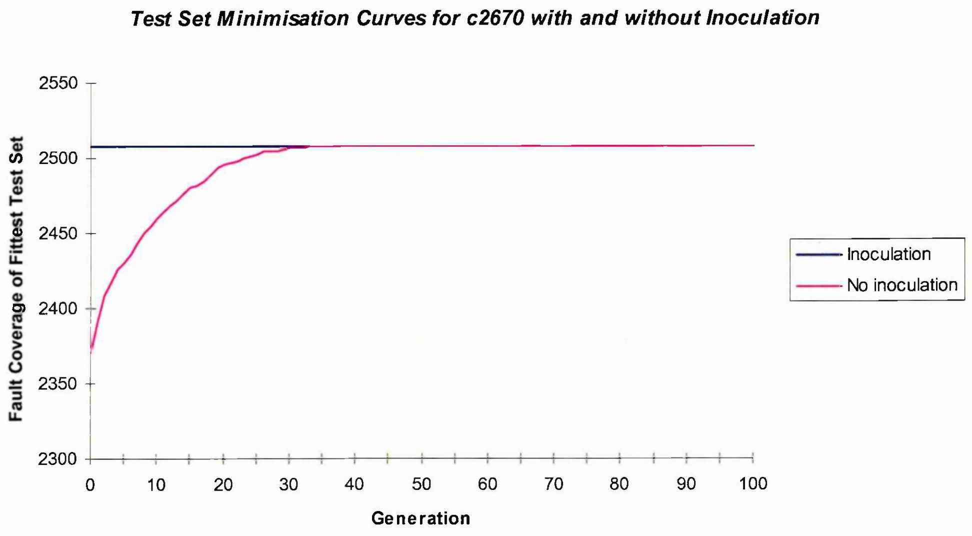
4.11.1 Inoculation of the Initial Population

In Section 4.8, the typical behaviour of GA-MITS was discussed. Broadly speaking, the minimisation process begins with a steady reduction in test set size, slowing down during the later generations as GA-MITS approaches minimal solutions. During the remainder of the run the algorithm is essentially refining the solutions until it converges on what is the smallest test set it is able to locate. In Figure 4.7, the red curve illustrates this behaviour. The behaviour of GA-MITS without inoculation is considerably different, as can be seen from the blue curve in Figure 4.7(a). In earlier generations the size of the fittest test sets actually increase until such a point, or generation, is reached that the required fault coverage, $N(C)_{\max}$, is achieved by at least one chromosome in a population. This may be seen in Figure 4.7(b). As can be seen, there is no instant reduction in test set size as in the case with inoculation. The fact that GA-MITS with inoculation instantly reduces test set sizes from generation zero onwards is largely due to the initial population being inoculated with a chromosome containing gene values of '1' throughout. So why does this serve to improve the behaviour of the GA?

Without inoculation the initial generation would consist of randomly generated chromosomes containing no biased individuals. Because of this random nature it is highly unlikely that chromosomes would be present which achieved the required fault coverage (this fact has been borne out in experiments). In such a case GA-MITS would perform two distinct phases of optimisation. The first phase would be to achieve maximal fault coverage by maximising the function F_1. As this form of the fitness function disregards test set size, the sizes of the optimal test sets in these early generations increase with fault coverage. Once maximal fault coverage has been achieved by one or more chromosomes, the second phase commences and GA-MITS maximises the second form of the fitness function F_2 for the fittest test sets in addition to F_1 for those achieving less than maximal fault coverage. During this second phase the size(s) of the optimal test set(s) in each generation start decreasing while maintaining maximal fault coverage. Figure 4.8 below shows a schematic test set minimisation curve of Test Set Size versus Generation without inoculation. The broken line separates the two phases of optimisation.



(a)



(b)

Figure 4.7(a) *Curves showing number of test vectors in optimal test set versus generation for the circuit c2670. The blue curve is for GA-MITS without inoculation. It is not until generation 34 that the fittest test set achieves maximal fault coverage. The red curve is for GA-MITS with inoculation. The final minimal test set found using both approaches contains 119 test vectors. This maximum was located at generation 37 with inoculation and at generation 72 without.* **(b)** *Corresponding Fault Coverage versus Generation curves.*

In this curve a test set achieving maximal fault coverage has been located at generation g'. To the right of the broken line GA-MITS begins reducing the size of the optimal test set found so far, from generation to generation. GA-MITS requires a large number of generations to find a test set achieving maximal fault coverage. The rationale behind inoculating the initial population is to eliminate the first, fault coverage maximisation phase and reducing the total time required by GA-MITS to find the minimal test set. The advantages of inoculation will now be presented by analysing the minimisation of test sets for the ISCAS-85 circuit c2670 with and without inoculation.

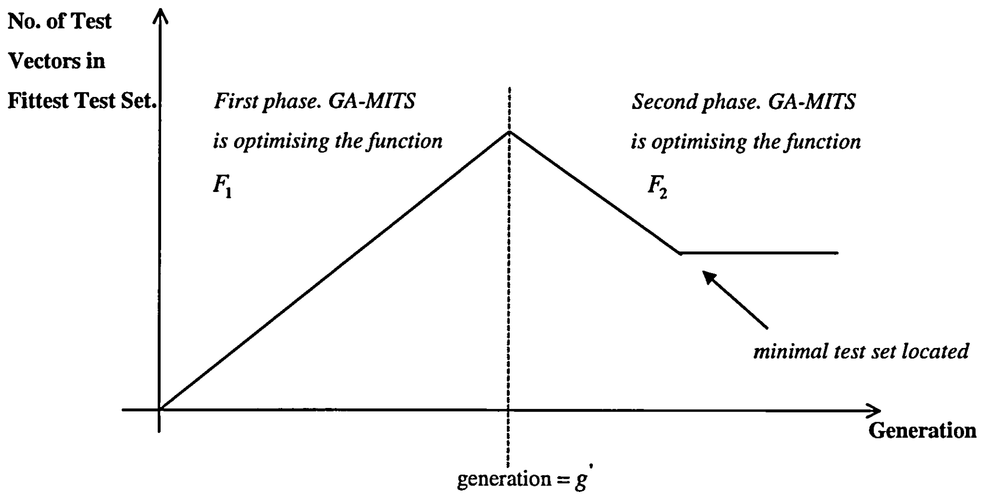


Figure 4.8 *Test Vectors in Minimal Test Set versus Generation curve for hypothetical test set optimisation case. The broken line separates the two phases of optimisation where GA-MITS first optimises F_1 and then F_2. At generation g' a test set achieving maximal fault coverage has been located.*

Returning to Figure 4.7, the blue curve shows the number of test vectors in the optimal test set versus generation for the no-inoculation case and the red curve corresponds to the inoculation case. As expected, it may be seen from this curve that the fittest test set in the initial population does not achieve maximal fault coverage but covers 2371 faults with 90 test vectors. GA-MITS proceeds to optimise the fault coverage without regard for the test set size. The size of the optimal test set in each of these early generations increases with fault coverage. Maximal fault coverage is finally achieved by generation 34 with a test set containing 134 test vectors. The curve then proceeds with a sharp, negative gradient as the test set sizes are rapidly reduced (much like the initial behaviour of GA-MITS with inoculation). By around generation 49, the algorithm only improves the quality of the minimal test set by a small percentage, over a relatively large number of generations. By generation 72 GA-MITS without inoculation has found the optimal test set containing 119 test vectors. This test set was located by GA-

MITS with inoculation by generation 37 which is almost 50% quicker than the no-inoculation case, clearly illustrating the superiority of the inoculation scheme.

Thus the attempt to speed-up the algorithm and increase the quality of the final solution by eliminating the initial, fault coverage optimisation phase was successful. To give GA-MITS this 'head-start', the process of doping or inoculating was devised. Unsure of the correct term for this process it was decided by the author to ask the GA research community (by posting a query to the internet based, GA-LIST[5]) whether this was existing and 'acceptable' practice. The overwhelming verdict was that, although initially frowned upon by GA purists (after all, nature did not have a head-start !), inoculation or the incorporation of problem specific knowledge was essential for GAs to be competitive with other more established optimisation techniques.

The inoculation process had to guarantee that a test set achieving maximal fault coverage, for every possible circuit, was inserted into the initial population. The easiest method for meeting this criterion was to insert the test set containing all test vectors. Although in most cases it was far from the minimal test set, GA-MITS instantly began reducing test set sizes, dispensing with the initial fault coverage optimisation phase. It could be argued that without inoculation, the first test set to achieve maximal fault coverage would be significantly smaller than the test set containing all possible test vectors. However, the searching capabilities of GA-MITS soon negates this perceived advantage as the final, minimal test set is located much sooner than the case without inoculation. This can be seen from the red curve Figure 4.7.

On the whole, the minimal tests found for a given circuit using GA-MITS with inoculation were smaller than those found without inoculation. This seems to be due to the fact that when GA-MITS with inoculation starts reducing test set sizes i.e. from generation 1 onwards, there were more schemata present than in the case without inoculation. Without inoculation, the test set sizes are only reduced at generation g', where $g' > 0$, when a test set achieving the required fault coverage has been located. At this point the population at generation g' will be less diverse, i.e. contain less schemata, than the initial population with inoculation. This reduced diversity at the point of maximal fault coverage means there is less exploration that can be performed and the emphasis seems to be placed on exploiting the genetic information contained in generation g' and onwards.

[5] GA-LIST is an Internet based mailing list for GA practitioners, run by researchers at the US Navel Research Centre, Washington. It is widely subscribed by leading GA researchers and is the first 'port-of-call' for questions, journal/book announcements and conference paper calls. It is also a valuable resource for GA related research, and software. It can be found at http://www.aic.nrl.navy.mil/galist .

4.11.2 The Use of an Exponential Ranking Scheme within GA-MITS

An exponential ranking scheme [60, 61] was chosen as the parent selection method within GA-MITS. Selection is widely viewed as providing the driving force for evolution within both the natural and artificial worlds and must therefore be chosen with much care and consideration. A careful balance must be achieved between exploration and exploitation and a critical parameter for controlling such a balance is selection pressure (this was discussed in much detail in Chapter 3). Too little selection pressure will result in too much exploration and too much selection pressure will result in too much exploitation. The ability to control the selection pressure is an important property of any selection scheme and exponential rank selection provides this level of control. The exponential rank fitnesses are given by the function,

$$f(i) = \frac{(s-1)}{\exp(N.\ln s) - 1} \times \exp((N-i).\ln s)$$

where,

N = number of chromosomes in population
s = selection constant, $0 < s < 1$
i = rank

This again was discussed in more detail in Section 4.6.4 of this chapter. The parameter s controls the selection pressure and selection pressure is proportional to (1 - s).

So what is the correct setting for the parameter s ? The research undertaken by Blickle [60] suggests a setting of s in the range $0.4 \leq s \leq 0.6$ as it is within this range that a reasonable balance between exploration and exploitation is achieved. However, the designers of a GA must experiment with their application and fitness function before a fixed setting of s can be made. The curves in Table 4.9 and 4.10 illustrate the test set minimisation results for the circuits c499 and c3540 (both of the original test sets were generated by PODEM) for a range of values of s in the exponential fitness function $f(i)$.

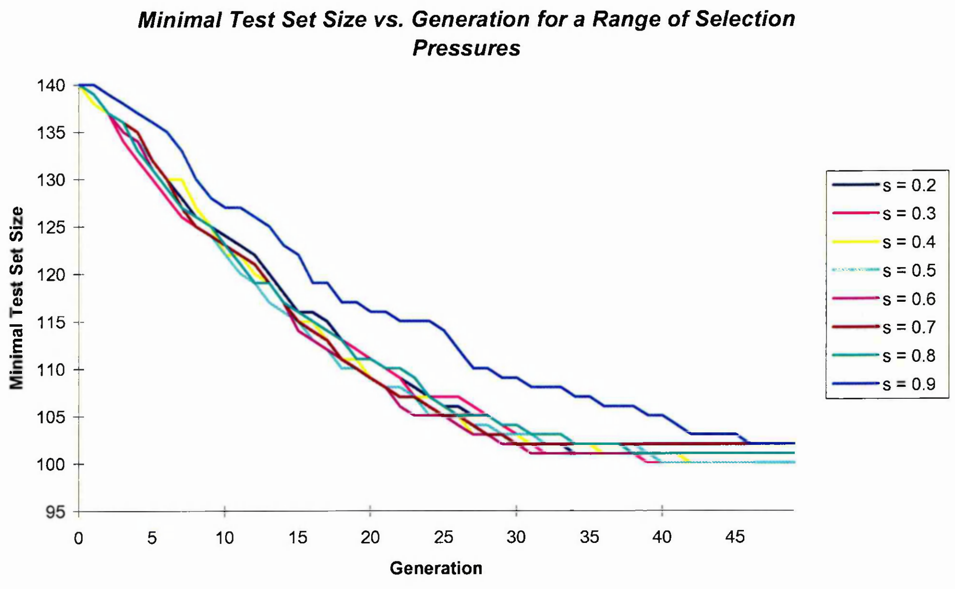


Figure 4.9. *Minimisation curves for the circuit c499 for a range of values of s. The colour of each curve corresponds to a value of s given in the box to the right of the curve.*

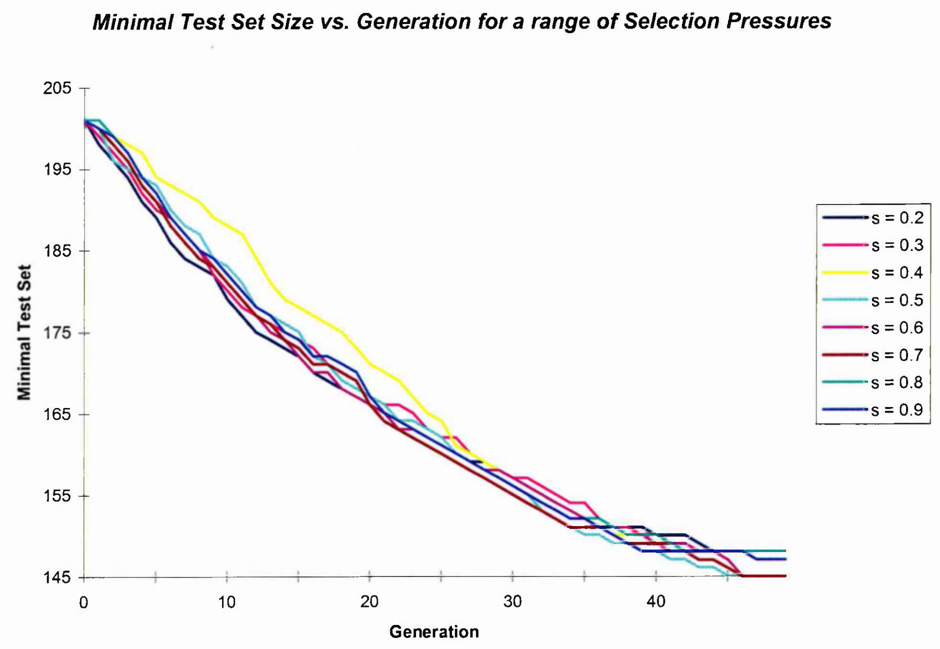


Figure 4.10. *Minimisation curves for the circuit c3540 for a range of values of s.*

For the curves in Figure 4.9 corresponding to the circuit c499, the smallest minimal test sets found for all the values of s contained 100 vectors. GA-MITS was able to locate this minimum value for three different settings of s: 0.3, 0.4, and 0.5. For a selection pressure of $s = 0.2$, the minimal test set found contained 101 test vectors suggesting that it prematurely converged on this sub-optimal minimal test set. For the values of s, 0.6, 0.7, 0.8 and 0.9 the sizes of the minimal test sets were 101, 102, 101 and 102 respectively. These values of *s* provided too little selection pressure and not enough exploitation of good solutions was undertaken by GA-MITS. For the curves corresponding to the circuit c3540, GA-MITS was able to locate the minimum test set, containing 145 test vectors for the values of *s*: 0.5, 0.6 and 0.7. For values of *s* less than 0.5 and greater than 0.7 too much and too little selection pressure was present in the selection scheme respectively. The minimal test sets found for these values of *s* contained only one or two more test vectors than those obtained with the *s* equal to 0.5, 0.6, 0.7.

The final setting for the parameter *s* was chosen as 0.5. For all minimisation results, this setting enabled GA-MITS to locate the smallest test set for a given circuit. In many cases, just as the two given in the above two figures, other values of *s* also enabled GA-MITS to locate the smallest test sets for a given circuit but $s = 0.5$ was a consistently good performer in all experiments. Theoretically speaking this setting provided the ideal balance between exploration and exploitation. The inclusion of relatively weak chromosomes in the mating processes ensured an adequate diversity in genetic material, or schemata, for the GA to continue to explore the search space. Exploitation of strong schemata also continued, ensuring that promising areas of the search space were exploited to the full. The fact that all of the known global minima were located by GA-MITS vindicates the choice of setting of *s*.

Initially a fitness proportionate selection scheme was used in GA-MITS due to its simplicity of implementation. In comparison to the rank scheme described above, the fitness proportionate scheme produced very poor results. Figure 4.11 compares the minimisation results obtained for the two schemes. As can be seen, with the fitness proportionate scheme, GA-MITS is unable to locate the minimal test set, containing 55 test vectors, within the designated 50 generations. Instead it was found much later at generation 61. This result was the same for the vast majority of original test sets. Furthermore, in many cases, the fitness proportionate scheme did not enable GA-MITS to find the smallest test set that was found using rank selection. Table 4.17 gives the minimisation results where the fitness proportionate scheme did not locate the test sets with the smallest number of test vectors that were located using the rank scheme. Out of the 27 cases, there were 7 sets of fault data for which the fitness proportionate scheme did not locate the smallest test set. At most, the difference in test set size was 4 vectors for the test set corresponding to the circuit c7552 (the original test set was generated using the GA based TPG). Although on the whole these differences in test set size only represent small percentage differences, they are still significant when choosing a selection scheme.

Much of the published literature [60, 61, 62] suggests that the fitness proportionate schemes suffer from premature convergence if there are ultra-fit individuals in the initial populations. Subsequent generations will largely contain off-spring of these individuals due to the selection pressure being too strong. Since GA-MITS employs inoculation there will almost definitely be at least one super-fit individual in the first generation so fitness proportionate selection within this application will result in premature convergence. The use of rank selection masks this phenomenon of the super-fit individual by selecting parents on the basis of rank. In more general terms, rank selection maintains even selection pressure throughout the run of a GA, helping to avoid premature convergence. In fitness proportionate schemes the selection pressure is proportional to the variance of the fitnesses in the population. So in earlier generations, due to the biased chromosome, the selection pressure will be very high resulting in too much exploitation and not enough exploration. The even selection pressure within a rank scheme certainly favours, in the case of GA-MITS, a balanced evolutionary strategy.

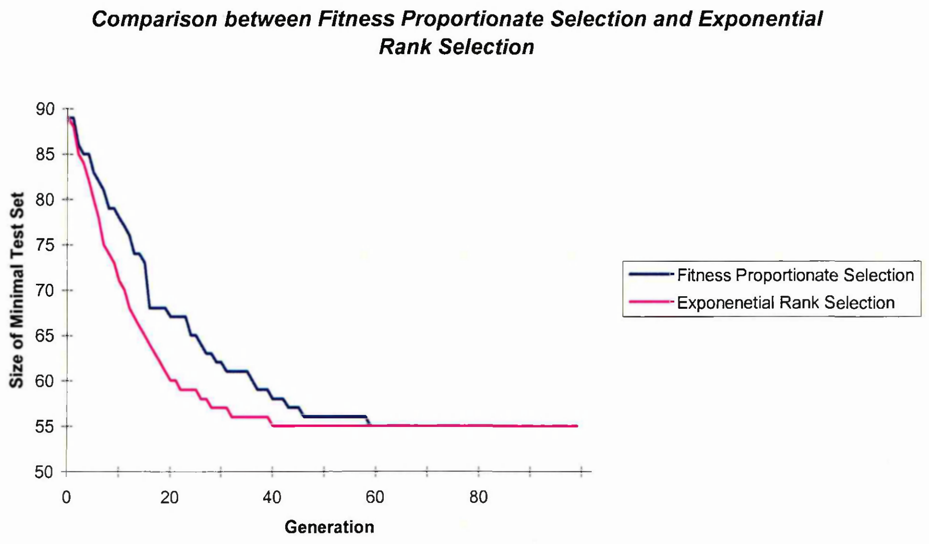


Figure 4.11 *Comparison of minimisation curves for fitness proportionate selection and rank selection. The fault data was generated by PODEM for the circuit c432.*

PODEM ATPG			GA based ATPG			Random ATPG		
Circuit	Fitness Proportionate Selection	Exponential Rank Selection	Circuit	Fitness Prop. Selection	Exponential Rank Selection	Circuit	Fitness Prop. Selection	Exponential Rank Selection
C499	102	100	**c1908**	112	110	**c7552**	166	164
c2670	120	119	**c2670**	88	87	-	-	-
c7552	201	198	**c7552**	160	156	-	-	-

Table 4.17 *Minimisation results for which the fitness proportionate scheme did not locate the smallest test sets as found by GA-MITS using the exponential rank selection scheme. The figures given in the table represent the number of test vectors in the smallest test sets located by GA-MITS.*

4.11.3 Selection of Crossover Operator and Crossover Rate

Along with selection, crossover is a very important feature within evolutionary algorithms. For successive generations of chromosomes to be fitter than their parents, the crossover operator must combine strong schemata with each other while at the same time causing as little disruption amongst the good schemata themselves. As in nature, crossover within evolutionary algorithms occurs with a high probability, known as the crossover probability, p_c, which is typically greater than 70%. Crossover was dealt with in some detail in Chapter Three of this thesis and the reader is directed there for a more detailed discussion.

Many crossover schemes have been developed over recent years. In Holland's original GA, single point crossover was the scheme of choice, and as a result is still popular amongst GA designers. Two point crossover and uniform crossover have developed a large following over recent years and have probably surpassed single-point crossover in popularity. Other crossover schemes [44] have been developed and are largely concerned with eliminating repetition in the chromosomes, for example, when routes are combined in the Travelling Salesperson Problem, the same city must not be visited more than once. During the design of GA-MITS single point, two point and uniform crossover [63] schemes were examined to determine which best suited the test set minimisation problem.

Figures 4.12 and 4.13 illustrate the test set minimisation curves for the circuits c432 and c499 using single point crossover. Each curve within the figures corresponds to a different setting of crossover probability. For the circuit c432 all settings of p_c greater than and including 30% enabled GA-MITS to locate the global minimum containing 55 test vectors. For the settings of $p_c > 60\%$, the convergence was relatively rapid, the minimum test set being found in 26 generations for the setting of p_c equal to 90% and 95%. Test set minimisation for the circuit c499 was far more sensitive to crossover probability than many of the ISCAS 85 circuits. In Figure 4.13 it may be seen that only settings of p_c equal to 90% and 95% enabled GA-MITS to locate the global minimum test set after 40 and 38 generations respectively. A crossover probability of 100% was also used during the experiments and in all cases it gave almost identical results to those obtained using 95%. For clarity, the curves corresponding to this setting were omitted from the figures. All other settings caused GA-MITS to prematurely converge on sub-optimal solutions because not enough combination of strong schemata was occurring.

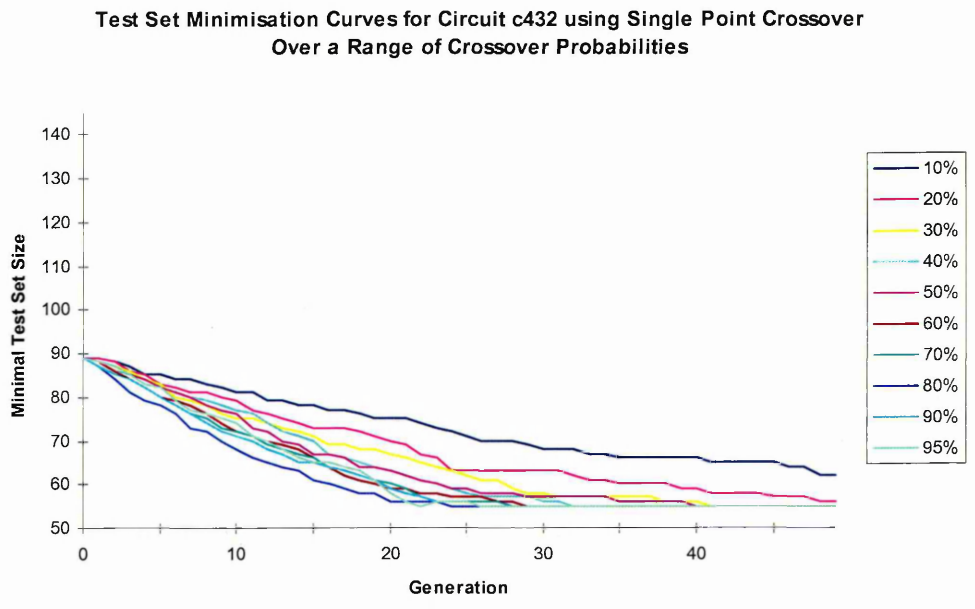


Figure 4.12 *Test set minimisation curves for circuit c432 using single point crossover.*

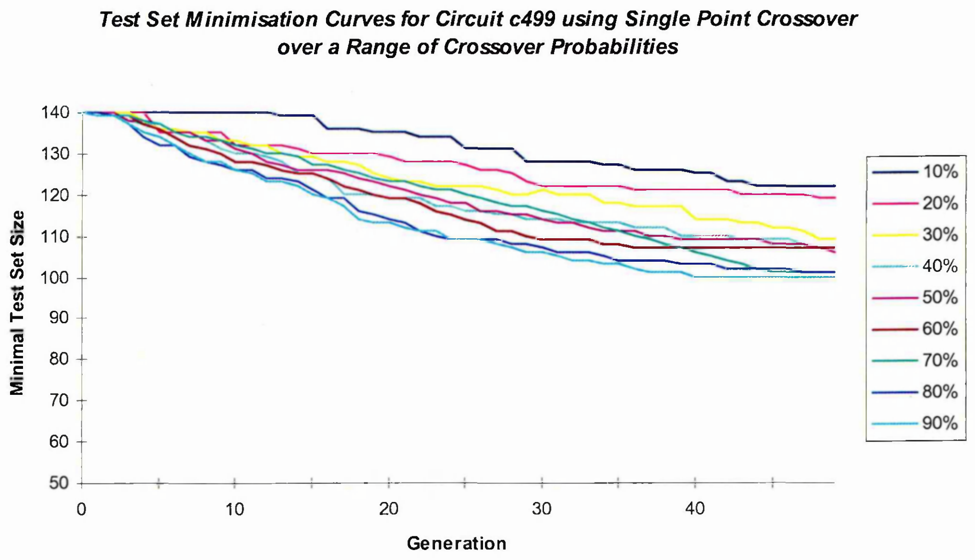


Figure 4.13 *Test set minimisation curves for circuit c499 using single point crossover.*

For two point crossover the results for circuits c432 and c499 are very similar to those obtained using single-point crossover and can be seen in Figures 4.14 and 4.15 respectively. For c432 the crossover probabilities of 10% and 20% resulted in premature convergence onto some very poor solutions while settings of p_c 70% and greater caused very rapid convergence on the global minimum. For c499 only the settings of p_c equal to 90% and 95% resulted in the global minimum being located.

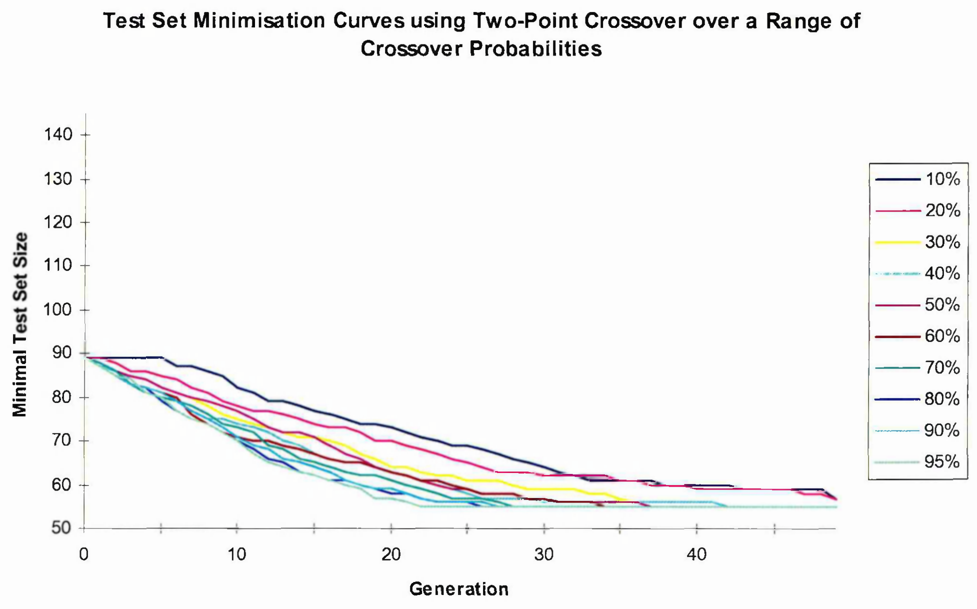


Figure 4.14 *Test set minimisation curves for circuit c432 using two point crossover.*

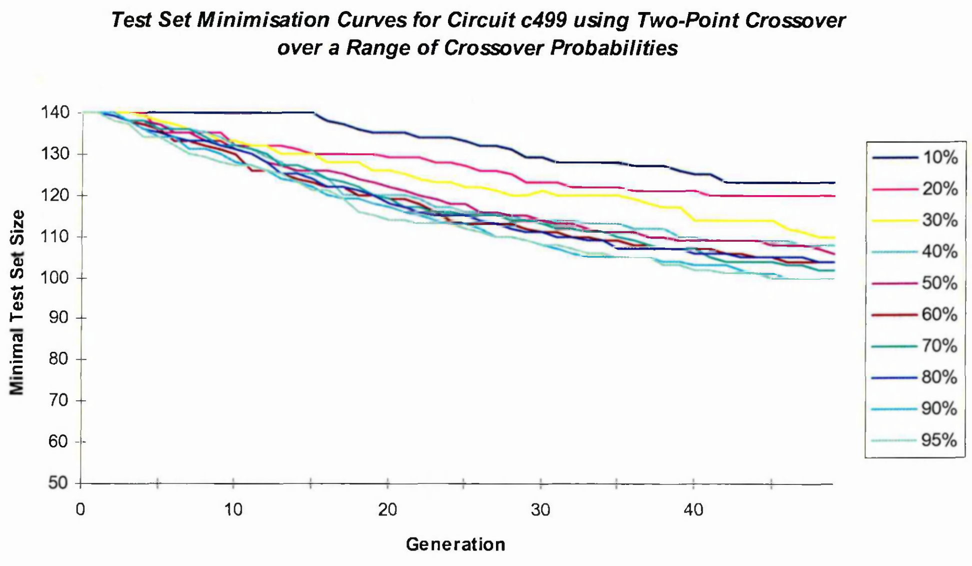


Figure 4.15 *Test set minimisation curves for circuit c499 using two point crossover.*

The curves given in Figure 4.16 and 4.17 correspond to the minimisation results obtained using uniform crossover. The results for the circuit c432 are similar to those obtained for both single point and two point crossover in terms of final solution quality. For the circuit c499 uniform crossover only found the global minimum test set for settings of p_c equal to 40% and 50%. As crossover occurs on a gene-by-gene basis this implies that GA-MITS locates the global minimum test set when approximately half the genes in the parent chromosomes are crossed (at random).

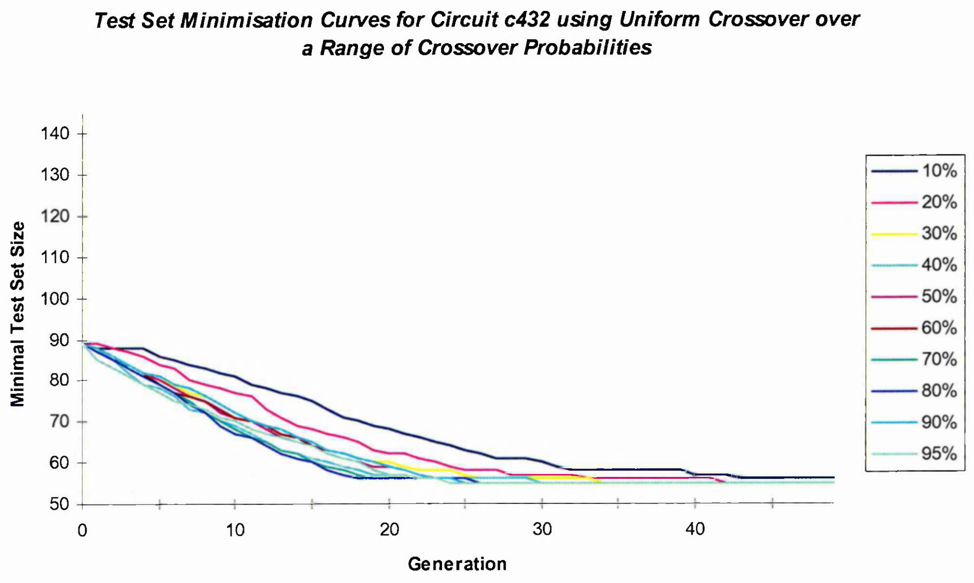


Figure 4.16 *Test set minimisation curves for circuit c432 using uniform crossover.*

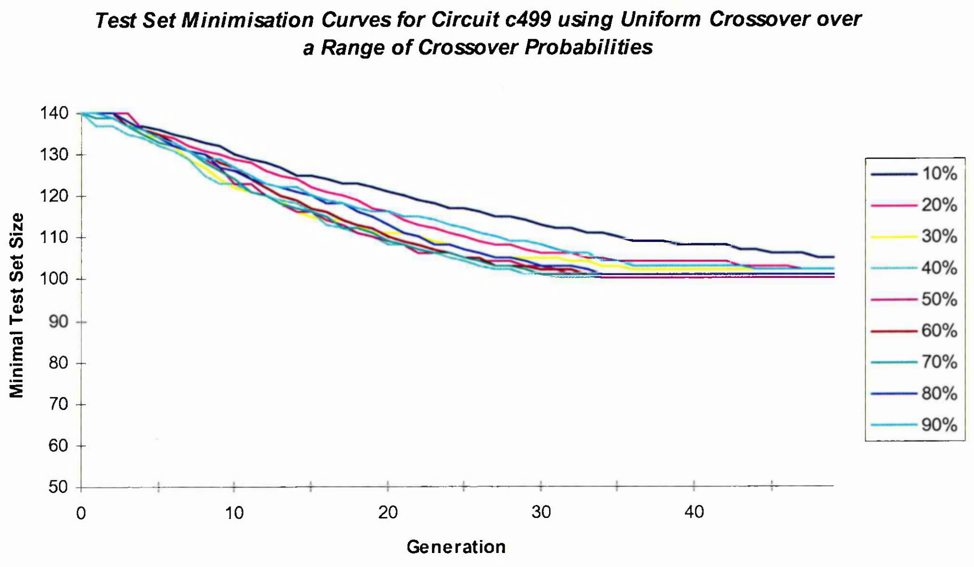


Figure 4.17 *Test set minimisation curves for circuit c499 using uniform crossover.*

So given the above results, which are typical for all the other circuits in the ISCAS-85 family, which crossover scheme produces the best results and which setting of p_c should be used ?

It may be seen from the above results, given the correct setting of p_c, all three crossover schemes enabled GA-MITS to locate the globally minimal test sets. Much theoretical analysis of crossover schemes has been undertaken [64, 65] and the relative merits of different schemes have been established. It is known that single point crossover cannot recombine all schemata and schemata of large defining length have a greater probability of disruption. In addition, this scheme suffers from 'end point bias' which means that the genes at the ends of the chromosomes will always be exchanged between parent chromosomes. Two point crossover addresses some of the problems associated with single point crossover by reducing end point bias and reducing the probability of disrupting schemata of large defining length. The uniform crossover scheme has the potential to be very disruptive of schemata of any length although on a positive note it is able to recombine all schemata present in the parents.

After consideration of the above known facts about each crossover scheme, two point crossover was implemented within GA-MITS with p_c equal to 95%. Two point crossover seems superior to single point crossover in this application because of its reduced end point bias and the lower likelihood of long schemata disruption. Having examined the results of GA-MITS there are many cases where the fittest chromosome contained schemata of large defining length. Although uniform crossover offers the advantage of being able to recombine all present schemata, it can be very disruptive especially for genes that coexist and which are essential for test sets of minimal size. This phenomenon of genes that coexist is known as *coadaption* in evolutionary terms [61]. For example, if alleles values of 1 are essential for a minimal test set at loci *n* and *n*+1 (where $1 \le n+1 \le N(T)_{\max}$) then there is a higher probability of this schema being disrupted using uniform crossover than for both single and two point crossover. Also in the case of a minimal test set containing unique patterns (vectors), the allele of 1 at the corresponding loci will also be a coadapted gene along with all the other necessary test vectors. Since 10 of the minimal test sets located by GA-MITS were proven as being global minima, it suggests that the phenomenon of the unique patterns, and hence of the coadapted gene(s), is frequently encountered within test set minimisation problems. It was for this reason and the slow convergence rate that a uniform crossover scheme was dismissed for use in GA-MITS.

Another observation of the above results is that the higher the probability p_c in the two point crossover scheme, the better the final results in terms of convergence and final solution quality. So why was the value of $p_c = 95\%$ selected as opposed to 100% ? The reasoning behind this decision was to ensure that a small proportion of the current population were copied over into the next generation. Although elitism may be seen as performing a similar role, it only ensures that the fittest chromosome in the generation is copied over. Setting the crossover probability to 95% increases the probability that relatively weaker chromosomes in the current generation are also copied over into the next generation to help increase genetic diversity in each generation.

4.11.4 Selection of Mutation Operator and Mutation Rate

The mutation operator used within GA-MITS is the simple bit-wise scheme as used in Holland's original GA. In this scheme each bit is mutated with a given probability. The problem faced by the designer of a GA is to select the exact setting of this mutation probability, p_m. Mutation provides a mechanism for exploring uncharted areas of the search space by creating new schemata that are not otherwise present within a population of chromosomes. With selection and crossover providing the exploitation of good schemata, mutation ensures some exploration of new schemata occurs thus helping to achieve a balanced optimisation strategy. Too much mutation however results in the GA doing too much exploration and not enough exploitation of good solutions, resulting in convergence on sub optimal solutions (in extreme cases, the GA could be seen as performing a random walk optimisation strategy). So how much mutation should occur ?

In nature, mutation is a relatively rare event, taking place less than 1 or 2% of the time, a fact that is echoed within GAs. But the only effective method for deciding on the probability p_m is through experimentation. The curves in Figure 4.18 are the test set minimisation curves for the circuit c2670 whose fault data was originally generated by PODEM. Each curve represents a different mutation rate from $p_m = 5\%$ down to $p_m = 0.1\%$.

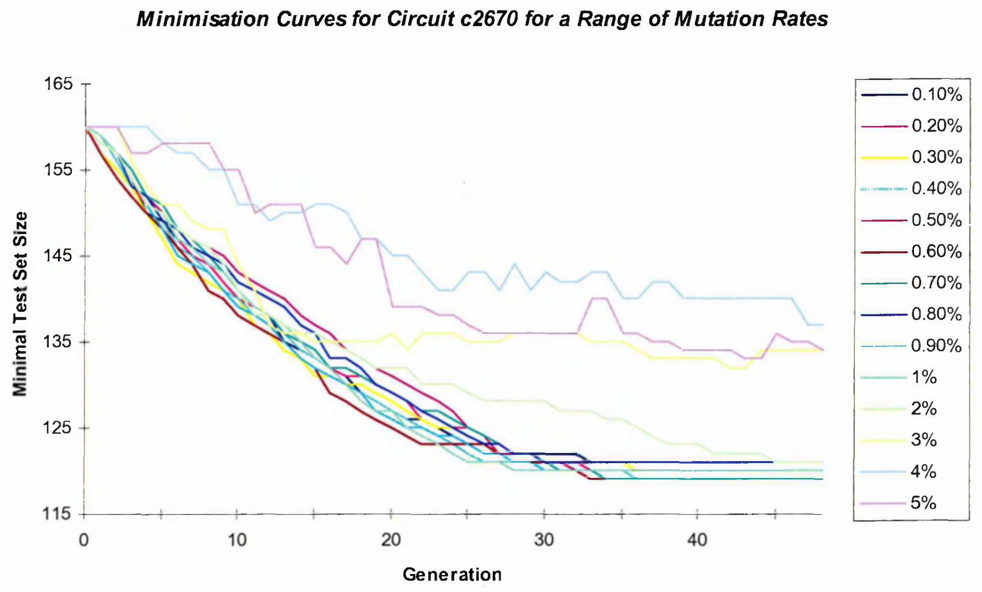


Figure 4.18. *Test set minimisation curves for the circuit c2670 for a range of mutation rates.*

As can be seen the settings of p_m equal to 5%, 4% and 3% are very disruptive on the minimisation process. Instead of a steady reduction of test set size as generations arise, as some generations are created the minimal test sets are actually getting larger. For example, given a mutation rate of 4%, from generation 13 to 14 the size of the minimal test set increases from 149 vectors to 150 vectors. The process of elitism should ensure that the fittest test set in a generation is at least the size of the fittest test set in the previous generation but since the mutation rate is so high and so disruptive, it results in the loss of this valuable genetic information. A similar result is seen for the mutation rate of 5% in which an increase in test set size of 3 vectors occurs from generation 18 to 19.

It is only with mutation rates of 2% and below that GA-MITS achieves the best results. For mutation rates of 2%, 1%, 0.9%, 0.3%, 0.2% and 0.1% GA-MITS locates minimal test sets containing either 121 or 120 test vectors. The smallest minimal test set containing 119 test vectors was only found for values of p_m equal to 0.7%, 0.6%, 0.5% and 0.4%. This result was similar for the remaining circuits in the ISCAS family. The exact value of p_m used in GA-MITS was 0.7% as it was the highest rate that consistently enabled the algorithm to locate the global minima. The highest rate was chosen as it is the view of the author that mutation is a very important property of evolutionary algorithms and as much of it should occur as frequently as is possible. Although exponential selection balances out the effects of having super-fit chromosomes in the initial population (due to inoculation) these individuals none the less drive the evolutionary process. Much exploitation is occurring as a result of the presence of these individuals. Having as much mutation as possible provides an opportunity for GA-MITS to also keep in mind other areas of the search space, instead of blindly following the lead of the inoculated individual and its offspring.

4.11.5 Selection of Population Size and Number of Generations in a Run

The final two parameters that have to be selected for GA-MITS are population size and the number of generations in each run of GA-MITS. The product of population size and generation count gives the total number of points that will be sampled in the search space and problems involving larger numbers of chromosomes should intuitively require a greater amount of the search space to be sampled. But what is required is an adequately sized population and generations in a run for the range of problems that may be encountered by GA-MITS. So what are these settings?

Returning again to Holland's original GA and that described in [66] (the 'simple GA') a general framework of 100 chromosomes and 100 generations seemed to be favoured. In many of the papers and books cited in this work population sizes and generation counts rarely exceeded these figures. Because of the robust nature of GAs and their strong searching capabilities the aforementioned settings of population size and generation count seem to provide very good optimisation results, regardless of the size of the search space. There is very little guidance on how to set these two parameters, let alone any theoretical correlation between the population size/generation count product and the size of the search space in a particular problem. Just as was the case with many of the other parameter choices for GA-MITS, it was only through experimentation that the final settings were decided upon.

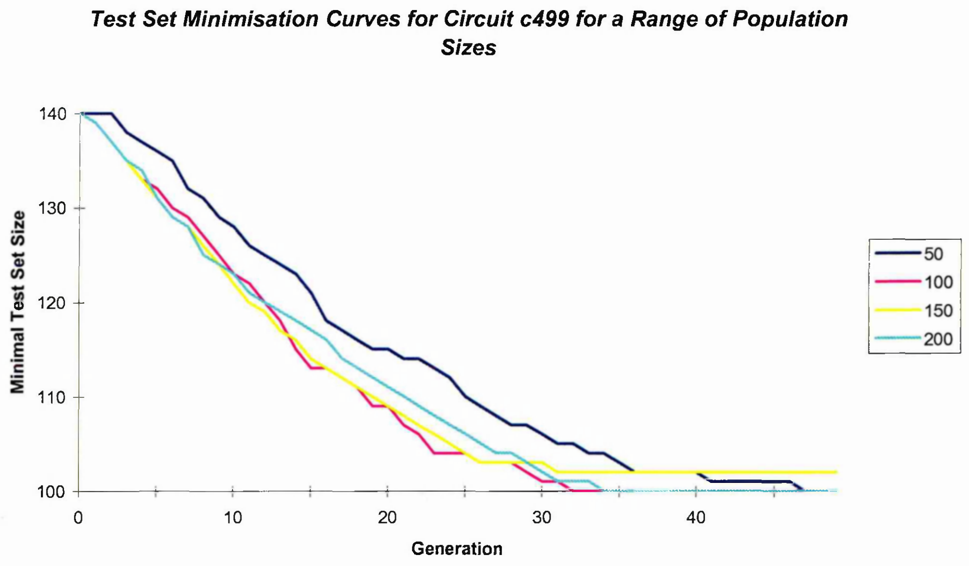


Figure 4.19 *Test set minimisation curves for the circuit c499 using a range of population sizes.*

Below in Figures 4.19 and 4.20 are the test set minimisation curves for the circuits c499 (fault data generated by PODEM) and c1908 (fault data generated by the GA based ATPG) over a range of population sizes. Four different population sizes were used. They were 50, 100, 150 and 200 chromosomes in each generation. The number of generations in each run was 50. For c499 all population sizes except 150, resulted in the GA finding the smallest minimal test set containing 100 test vectors. The quickest convergence was achieved with a population size of 100 chromosomes. For the circuit c1908 all four population sizes resulted in the minimal test set containing 110 test vectors being found, the population size of 150 chromosomes achieving the most rapid convergence. The results therefore for these examples, and indeed the remaining ISCAS circuits, are very similar for all population sizes. The actual setting for population size selected was 100 chromosomes to ensure an adequate balance between genetic diversity and the execution time of GA-MITS. It was stated earlier that the evaluation of the fitness of each chromosome dominates the overall execution time of the algorithm so it was felt by the author that population sizes of 150 and 200, although providing greater genetic diversity, do not improve either the convergence rate or the quality of the final solution by a significant amount.

The number of generations in each run of GA-MITS was selected as 50. All minimisation results quoted in this chapter were achieved using this figure. In addition, for all cases, GA-MITS was also run for 100 and more generations to see whether the algorithm had in fact converged. Given the GA operators, selection technique and parameter settings, all minimal test sets were found within the specified 50 generations. Again the execution time increases with generation count as more chromosome fitnesses will have to be evaluated, so the lower the generation count the better in terms of final execution time. An alternative method for determining the number of generations in a run would be to build in some convergence criteria into the GA itself [62]. A popular convergence criterion would be for the GA to halt once the average fitness of the current population was very similar to the maximum fitness in that generation, i.e. when the average was within 5% say, of the maximum fitness. Another criterion is to halt the GA when the maximum fitness has not improved over the last *n* generations. When GA-MITS was first designed it was run for a fixed number of generations. GA-MITS was able to find test sets containing the smallest number of test vectors in less than 50 generations for all cases and therefore this setting was retained in the final design of the algorithm.

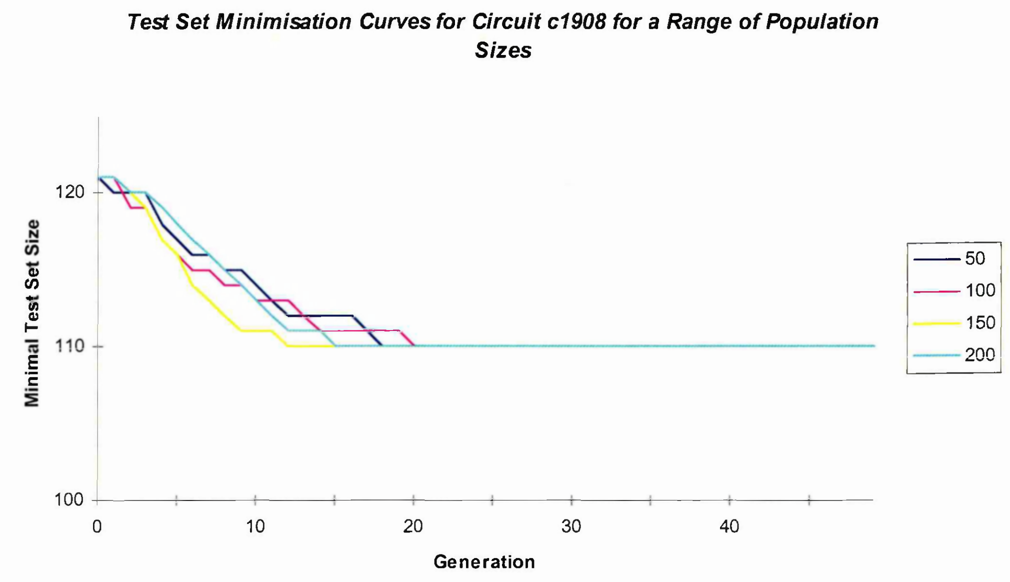


Figure 4.20 *Test set minimisation curves for the circuit c1908 using a range of population sizes.*

4.12 Summary

This chapter has introduced a novel and wholly original algorithm, GA-MITS, that solves the NP-hard problem of deriving minimal test sets for combinational, digital circuits. The algorithm has been designed as a post processor to existing ATPG tools and as a result may be easily incorporated in to such tools. The problem of generating minimal test sets for digital circuits is of real concern to today's integrated circuit manufacturers [2]. The complexity of modern electronics has dictated that many designs must now be able to test themselves using BIST techniques. Minimal test sets require less silicon on which to store the test vectors in such schemes which in turn reduces per unit manufacturing costs. For circuits that require testing by external stimuli, e.g. by the application of test vectors, minimal test sets reduce the per unit testing time and therefore testing costs. Since minimal test sets tackle, head-on, the most important factor within any profit making organisation viz. reducing costs, their derivation is of significant interest to all manufacturers particularly now as microprocessor designs become increasingly complex and approach prohibitively expensive testing costs [67].

Genetic Algorithms, which form the basis of GA-MITS, have proven themselves to be robust optimisation tools, successful across a wide range of application areas. A thorough literature search in this chapter shows their wide reaching applicability in the domain of integrated circuit design and test. Although they do not always provide the quickest optimisation method in this and many other fields, they do provide a method that competes with existing state-of-the-art algorithms in terms of final solution quality. In areas such as circuit layout and test pattern generation for example, they are providing solutions to problems that are far superior to a number of traditional methods.

The results given in this chapter show that GA-MITS does indeed have a major role to play amongst today's digital test tools in reducing manufacturer's costs. It has been applied to derive minimal test sets for a family of simplified RISC processors in addition to an internationally recognised set of benchmark circuits which are far more representative of industrial designs. The results obtained by GA-MITS, for all cases, illustrate that the algorithm is successful in locating minimal test sets from test sets that have been previously generated by a particular ATPG tool. If there is any redundancy in the original test set, GA-MITS is shown to have exploited it and located smaller test sets with the same fault coverage. For the simplified RISC processors, reductions in test set size of up to 62% were achieved. For the ISCAS-85 benchmark circuits reductions of up to 39% were achieved, with an average reduction of approximately 18%. For the benchmark circuits, out of the 27 sets of fault data presented to GA-MITS, 10 of the minimal tests generated were shown to be *the* global minima amongst the original test sets. Although this is by no means solid proof, it does indicate that GA-MITS has the ability to locate the absolutely minimal test sets within the original data.

This chapter has also described all the steps that are required to successfully design a GA for a particular application. The design process of any GA is largely based upon empirical evidence and this has been rigorously presented. All design decisions were guided by theoretical evidence, but many of the final parameter settings could only be chosen after much experimentation and careful consideration of the results. The majority of papers on the theoretical issues of GAs cited in this work state that their particular findings are valid only for the fitness function(s) described in those papers. The final words of wisdom in many of these papers are often along the lines,

> "....despite limited empirical success in using this method or that, a general answer remains elusive." [68]

> "...one must weigh up the cost of embellishments on a single application with their general applicability but I leave it to others to decide whether or not the techniques described here have any wider value" [69]

So it was only after much experimentation that the author could confidently arrive at all the design decisions and parameter settings that were required for GA-MITS.

This work has resulted in the publication of a paper [70] presented at the *40th Midwest Symposium on Circuits and Systems*, held in Sacramento, USA, August 1997. In addition, the author received an award for the work in the annual *Best Student Paper Contest* , shown in Appendix C. This work therefore has received international academic recognition as being valid and applicable in modern world electronic design and test. This suggests that GA-MITS has achieved its goal of test set minimisation and has a role in the future development of ATPG tools.

4.13 References

[1] Gibbs W.W., "The law of more", Solid State Century, Special Issue Scientific American, Vol. 8, No. 1, pp 62 - 63, November 1997.

[2] Thompson K.M., "Intel and the myths of test", IEEE Design and Test of Computers, pp 79-81, Spring 1996.

[3] Abramovici M., Breuer M.A., Friedman A.D., "DIGITAL SYSTEMS TESTING AND TESTABLE DESIGN", IEEE Press, 1990.

[4] R. Drechsler and N. Drechsler, "EVOLUTIONARY ALGORITHMS IN CIRCUIT DESIGN", Kluwer Academic Publishers, 2002.

[5] Cohoon J., and Paris W., "*Genetic Placement,*" Proc. IEEE ICCAD, pp.422-425, 1986.

[6] Stockmeyer L., "Optimal orientation of cells in slicing floorplan designs", Information and Control, 57(2), pp.91–101, 1983.

[7] Wong D. F., Liu C. L., "A new algorithm for floorplan design", Proc Design Automation Conference, pp 101–107, 1986.

[8] Esbensen, H., and Kuh E., "Design Space Exploration Using the Genetic Algorithm," Proceedings of the *1996 IEEE International Symposium on Circuits and Systems*, pp. 500-503, May, 1996.

[9] Nakaya S., et. al., "An adaptive genetic algorithm for VLSI floorplanning based on sequence-pair", Proc. IEEE International Symposium on Circuits and Systems, Vol.3, pp.65-68, 2000.

[10] Valenzuela, Christine L., Wang, Pearl Y., "*A* Genetic Algorithm for VLSI Floorplanning", Proc. Parallel Problem Solving from Nature VI (PPSN VI), pp 671-680, 2000.

[11] H. Esbensen. "A Macro-Cell Global Router Based on Two Genetic Algorithms", Proc. of European Design Automation Conf. Euro-DAC, pp 428–433, Sept. 1994.

[12] Krashinsky, R., "GRAPE: Genetic Routing And Placement Engine" , Embodied Intelligence Project, Massachusetts Institute of Technology, 2000.

[13] Drechsler R., Gockel N., Becker B., "A genetic algorithms for minimisation of fixed polarity Reed-Muller expansions", Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms, pp 392 - 395, 1995.

[14] Miller J.F., Thomson P., "Combinational and sequential logic optimisation using genetic algorithms", Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA, pp 18 -22, University of Sheffield, Sheffield, UK, Sept. 1995.

[15] Drechsler R., Becker B., Gockel N., "Heuristics for OBDD minimisation by evolutionary algorithms", In Parallel Problem Solving from Nature, Springer-Verlag, pp 730 - 739, 1996.

[16] Drechsler R., Gockel N., Becker B., "Minimisation of OKFDDs by genetic algorthms", International Symposium on Soft Computing, p B:271 - 277, 1996.

[17] Zebulum R.S. et. al., "EVOLUTIONARY ELECTRONICS: AUTOMATIC DESIGN OF ELECTRONIC CIRCUITS AND SYSTEMS BY GENETIC ALGORITHMS", CRC Press, 2001.

[18] Koza J.R., "GENETIC PROGRAMMING IV: ROUTINE HUMAN-COMPETITIVE MACHINE INTELLIGENCE", Kluwer Academic Publishers, 2003.

[19] Kruiskamp W., Leenaerts D., "DARWIN: CMOS op-amp synthesis by means of genetic algorithm", Proceedings of the 32nd Design Automation Conference, Association for Computer Machinery, pp 698 - 703, 1995.

[20] Rogenmoser R., Kaeslin H., Blickle T., "Stochastic methods for transistor size optimisation of CMOS VLSI circuits", in Parallel Problem Solving from Nature IV, pp 849 - 858, Springer-Verlag, 1996.

[21] Thomson A., "Hardware evolution: Automatic design of electronic circuits in reconfigurable hardware by artificial evolution", Ph.D. Thesis, University of Sussex, 1996.

[22] Thomsom A., "An evolved circuit, intrinsic in silicon, entwined with physics", Proceedings of the 1st International Conference on Evolvable Systems (ICE90), Tsukuba, Japan, pp 390 - 405, 1996.

[23] Shahookar P., Mazumder P., "VLSI cell placement techniques", ACM Computing Surveys, Vol.23, No.2, pp. 143 - 220, 1991.

[24] Fujiwara H. and Toida S., "The complexity of fault detection: An approach to design for testability", Proceedings of the 12th International Symposium on Fault Tolerant Computing, pp 101-108, June 1982.

[25] Srinivas M. and Patnaik L.M., "A simulation based test generation scheme using genetic algorithms", Proceedings of the 6th International Conference on VLSI Design, pp 132 - 135, Jan. 1993.

[26] O'Dare M.J. and Arslan T., "Generating test patterns for VLSI circuits using a genetic algorithm", Electronics Letters, Vol. 30, No.10, May 1994.

[27] Prinetto P., Rebaudengo M., Sonza Reorda M., "An automatic test pattern generator foe large sequential circuits based on genetic algorithms", Proceedings of the International Test Conference, pp 240 - 249, 1994.

[28] Rudnick E.M., Patel J.H., "A genetic approach to test application time reduction for full scan and partial scan circuits", Proceedings of the 8th International Conference in VLSI Design, pp 288 - 293, Jan. 1995.

[29] Corno F., Prinetto P., Rebaudengo M., Sonza Reorda M., "GATTO: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuits", IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems, Vol. 15, No. 8, Aug. 1996.

[30] Roth J.P., "Diagnosis of automata failures: A calculus and a method", IBM Journal of Development, Vol. 10, pp 278-291, July 1966.

[31] Goel P., "An implicit enumeration algorithm to generate tests for combinational logic circuits", IEEE Transactions on Computers, Vol. C-30, pp 215-222, March 1981.

[32] Abramovici M., Breuer M.A., Friedman A.D., "DIGITAL SYSTEMS TESTING AND TESTABLE DESIGN", IEEE Press, 1990.

[33] Abramovici M. et. al., "SMART and FAST: Test generation for VLSI scan-design circuits", IEEE Design and Test of Computers, Vol. 3, pp 43 -54, Aug. 1986.

[34] Fujiwara H. and Shimono T., "On the acceleration of test generation algorithms", IEEE Transactions on Computers, Vol C-32, pp 1137-1144, Dec. 1983.

[35] Cheng K.T., and Agrawal V.D., "A simulation based direction-search method for test generation", Proceedings of the International Conference of on Computer Aided Design, pp 48-51, Oct. 1987.

[36] Takamatsu Y. and Kinoshita K., "CONT: A concurrent test generation algorithm", FTCS-17, pp 22-27, July 1987.

[37] Kirkland T., Mercer M.R., "A topological search algorithm for ATPG", Proceedings of the 24th ACM/IEEE Design Automation Conference, pp 502-508, June 1987.

[38] Raik, J., Markus A., Personal Communication, Tallin Technical University, 1997.

[39] Reddy L.N., Pomeranz I., Reddy S.M., "ROTCO: A reverse order test compaction technique", Proceedings of EURO-ASIC 92, June 1992.

[40] Carter J.L., Dennis S.F., Iyengar V.S., Rosen B.K., "ATPG via Random Pattern Simulation", Proc. of the 1985 International Symposium on Circuits and Systems, pp 683 - 686, 1985.

[41] Schulz M., et. al. "SOCRATES: A highly efficient automatic test pattern generation system", IEEE Transactions on Computer Aided Design, pp 126 -137, Jan. 1988.

[42] Waicukauski J.A., Shupe P.A., Giramma D.J., Matin A., "ATPG for ultra large structured designs", Proceedings of the International Test Conference, pp 44 -51, 1990.

[43] Goel P., Rosales B.C., "PODEM-X: An automatic test generation system for VLSI logic structures", Proceedings of the 18th Design Automation Conference, pp 260 - 268, 1981.

[44] Aylor J.H., Cohoon J.P., Feldhousen E.L., Johnson B.W., "GATE - A genetic algorithm for compacting randomly generated test sets", International Journal of Computer Aided Design VLSI Design 3, pp 259 - 272, 1991.

[45] Pomeranz I., Reddy L.N., Reddy L.N., "COMPACTEST: A method to generate compact test sets for combinational circuits", IEEE Transactions on Computer Aided design of Integrated Circuits and Systems, Vol. 12. No. 7, pp. 1040 - 1049, July 1993.

[46] Guo R., Pomeranz I., Reddy S.M., "Procedures for static compaction of test sequences for synchronous sequential circuits based on vector restoration", ", Proceedings Design Automation and Test in Europe, pp583-589, 1998.

[47] Hsiao M.S., Chakradhar S.T., "State relaxation based subsequence removal for fast static compaction in sequential circuits", Proceedings Design Automation and Test in Europe, pp577-582, 1998.

[48] Hsiao M.S., Rudnick E.M., Patel J.H., "Fast algorithms for static compaction of sequential circuit test vectors", Proceedings of the VLSI Test Symposium, pp 188-195, April 1997.

[49] Corno F., Prinetto P., Rebaudengo M., Sonza Reorda M., "New static compaction techniques of test sequences for sequential circuits", Proceedings of the European Design and Test Conference, pp 37-43, March 1997.

[50] Christofedes, N., and Korman K., "A computational survey of methods for the set covering problem," Management Science, Vol. 21, No. 5, pp. 591-599, Jan. 1975.

[51] Ubar R., "Test Synthesis with alternative graphs", IEEE Design and Test of Computers, pp 48 - 57, Spring 1996.

[52] Blickle T., "Theory of Evolutionary Algorithms and Applications to System Synthesis", Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, 1996.

[53] Surrey, P.D., Radcliffe, N.J., "Inoculation to initialise evolutionary search", Evolutionary Computing: AISB Workshop, Ed. T. Fogarty, Springer-Verlag, 1996.

[54] Brglez F. and Fujiwara H., "A neutral netlist of 10 combinational benchmark designs and a special translator in FORTRAN", International Symposium on Circuits and Systems, June 1985.

[55] Aero H., Personal Communiction, Tallinn Technical University, Estonia, 1997.

[56] Markus A., Raik J., Ubar R., "Fast and efficient compaction of test sequences using bipartite graph representation", To appear in the Proceedings of the Asian Test Conference, 1998.

[57] Akers S. B., Jansz W., "Test Set Embedding in a built-in self-test environment", Proceedings of the IEEE International Test Conference, pp. 257 -263, Aug. 1989.

[58] Alba E., Tomassini M., "Parallelism and evolutionary algorithms", IEEE Transactions on Evolutionary Computation, Vol. 6, Issue 5, pp.443-462, October 2002.

[59] Esbensen H., Design Optimisation Group Manager, personal communication, Avant! Corporation, Fremont, California, USA, 1998,

[60] Blickle T., Thiele L., "A comparison of selection schemes used in genetic algorithms", Technical Report Nr. 11, Swiss Federal Institute of Technology, Zurich, Switzerland, December 1995.

[61] Hancock P.J.B., "An empirical comparison of selection methods in evolutionary algorithms", In Fogarty T.C. ed. Evolutionary Computing: AISB, pp80-94, 1994.

[62] MitchelL M., "AN INTRODUCTION TO GENETIC ALGORITHMS", The MIT Press, Cambridge, Massachusetts, USA, 1996.

[63] Spears W.M., De Jong K.A., "An analysis of multi-point crossover", In G. Rawlins ed., Foundations of Genetic Algorithms, pp301 - 315, Morgan Kaufmann, 1991.

[64] Syswerda G., "Uniform crossover in genetic algorithms", In R.K. Belew and L.B. Booker eds. , Proceedings of the Fourth International Conference on Genetic Algorithms, pp2 - 9, Morgan Kaufmann, 1991.

[65] Spears W.M., De Jong K.A., "On the virtues of parameterised uniform crossover", In J.D. Schaffer, ed., Proceedings of the Third International Conference on Genetic Algorithms, pp230 - 236, Morgan Kaufmann, 1989.

[66] Goldberg D.E., "GENETIC ALGORITHMS IN SEARCH, OPTIMISATION AND MACHINE LEARNING", Addison-Wesley , Reading, MA, 1989.

[67] Chin C., Manager, Design for Testability Group, - Personal Communication, Sun Microsystems, Mountain View, California, USA, 1997.

[68] Goldberg D.E., Deb K., "A comparative analysis of selection schemes", In G. Rawlins ed. , Foundations of Genetic Algorithms, Morgan Kaufmann., 1991.

[69] Fogarty T., "Varying the probability of mutation in a genetic algorithm", Proceedings of the Third International Conference on Genetic Algorithms", pp104-109, Morgan-Kaufmann, 1989.

[70] Takhar, J.S., Gilbert D.J., "The derivation of minimal test sets for combinational logic circuits using genetic algorithms", Proceedings of the 40th Midwest Symposium on Circuits and Systems, pp 40-44, Sacramento, USA, August 1997.

Chapter 5. Conclusion and Further Work

The work presented in this thesis provides original and novel approaches to three important areas in combinational digital testing viz. test pattern generation, test set minimisation and testability analysis. Chapter One gave a brief overview of each of these areas and a discussion on the rapid pace of change in VLSI technology by introducing Moore's Law. After placing the field of test within this real-world, industrial context, basic digital testing terminology and fundamental test pattern generation techniques were presented.

Chapter Two presented an original test pattern generation technique which combines cubical calculus and the Boolean difference. The chapter began with a detailed discussion on both the Boolean difference method and John Roth's seminal work on the calculus of cubes. It was noted that the elegant Boolean difference technique was not used in ATPG systems due to its cumbersome, algebraic nature. It was shown that the cubical calculus provides a means to solve the Boolean difference and an outline test pattern generation algorithm was presented. The chapter concluded by introducing an original technique for evaluating testability measures again, using the Boolean difference and cubical calculus. It is felt by the author that the work presented in this chapter opens up the field of algebraic test pattern generation which, for many years, has been overlooked by topological techniques (such as those presented in Chapter One). Given that todays ICs are designed using high-level description languages, test pattern generation for some classes of digital circuits based on algebraic techniques may become more prominent. It was shown that some of the core computations required for test pattern generation can also be used to evaluate testability measures. It is felt by the author that by using this work, these two important digital design and test functions could be incorporated into a single design suite.

Chapter Three introduced the general area of evolutionary computation and more specifically, genetic algorithms. By emulating Darwinian evolution and natural selection, it was shown that genetic algorithms provide a very powerful technique for providing solutions to NP-complete problems. The algorithm is not deterministic and does not guarantee finding the optimal solution, but it does provide very good solutions to very large problems. Digital circuit design and test is littered with such problems and genetic algorithms have been applied in many cases. Chapter Four describes one such problem, the minimisation of test sets, and presents original work by the author in optimizing test sets using a genetic algorithm. A new algorithm was developed, GA-MITS, that was shown to be very successful in locating minimal test sets for the ISCAS '85 benchmark circuits. The test sets were provided by the Design and Test Centre at Tallinn Technical University based on one of their test pattern generation algorithms. The size of the optimised test sets produced by GA-MITS, is bound by the search space represented by the original test set. Therefore, raw comparisons with other minimisation techniques, using different test sets generated by different ATPG systems are not valid. But, the Tallinn team did have their own test set minimisation algorithm which GA-MITS consistently out-performed. In fact, during the development phases of both algorithms, results were shared and once the Tallinn team realised they had been out-performed, they would then use the results of GA-MITS to refine their algorithm. In conclusion, the genetic algorithm proved very successful in generating minimal test sets and is capable of optimising large test sets. Given the discussion of Chapter One about the need to control test costs by reducing test application time, GA-MITS provides an excellent solution to a real-world problem.

Given the finite time allocated for all academic research, the work presented in this thesis opens some interesting avenues for future work. The work presented in Chapter Two outlines the theoretical basis of a test pattern generation algorithm. This work needs to be implemented in a high-level computer language and needs to be *bench-marked* against other, existing test pattern generation algorithms. The goal of every algorithm designer is to take the domain knowledge, together with efficient data structures to represent the data, and produce a generalised, repeatable solution for a problem. The work to implement this test pattern generation technique would be to efficiently represent and solve the cubical calculus. The implementation must be mindful of both memory and processor usage, as today's circuits are represented by enormous amounts of data. It is felt by the author that the key to a successful implementation will be the efficient management and processing of this data. Another avenue for future work would be to investigate whether the calculus of cubes and the Boolean difference is applicable to sequential circuit testing, as this a very wide and active area within digital testing.

The test set minimisation work also has some potential for future work. Much work has been done in the general area of evolutionary algorithms which may be applicable to GA-MITS. For example, the parallel implementation of GAs is a very interesting area to improve the time-performance of genetic algorithms in general. Another example would be to explore dynamic parameter settings. Many of the run-time parameters of GA-MITS, such as crossover and mutation rates, are static and set at compile-time. It would be interesting to explore the area of dynamically altering these, and other parameters, as generations evolve. Perhaps a high mutation rate in later generations when the algorithm has converged on a solution may enable it to locate fitter, more isolated solutions. GA-MITS as it is implemented, provides the single best solution to the test set minimisation problem that it is able to locate. One possible area for future development could be to incorporate the notion of Pareto Optimality. The test set minimisation problem is essentially a multi-objective optimisation problem; to minimise the test set size while maximising the fault coverage of the test set. The use of Pareto Optimality could enable GA-MITS to offer multiple solutions to the user who ultimately would decide which test set to use. For example, consider if the current implementation of GA-MITS produces a single solution; a test set comprising 25 test vectors giving 100% fault coverage. What if in a previous generation, GA-MITS encountered a test set containing 10 test vectors but offering 90% fault coverage? This is not an impossible scenario when one considers test vectors that only cover one or two *hard-to-test* faults. Given a 60% reduction in test size and therefore test application time, would a test engineer forgo 10% in fault coverage? For non-critical applications, he may well opt for lower fault coverage. Pareto Optimality would allow GA-MITS to offer such choices and could be a very interesting area of research.

The last point of discussion above, opens up the topic of compromising fault coverage for performance and may well be applicable to much of the test pattern generation and optimisation work presented in this thesis. The work presented here assumes the need for 100% fault coverage. But what if the goal of the test pattern generation algorithm for example, was to achieve 95 or 99% fault coverage? If this could be achieved it would certainly result in significant reductions in the time and memory requirements of the algorithm. There are many non-critical applications for IC's such as cheap, mass produced consumer devices that may well benefit by lower fault coverage testing. It is felt by the author that this particular work could produce some very interesting and important results.

In closing, the test pattern generation, testability measures and test set minimisation work was presented at *40th Midwest Symposium on Circuits and Systems*, held in Sacramento, USA and published in the symposiums proceedings (see Appendix B). In addition, the author received an award for the test set minimisation work in the symposium's annual *Best Student Paper Contest* (see Appendix C). The work presented in this thesis therefore, has received some international academic recognition as being valid and applicable in modern world electronic design and test.

Appendix A. Software Listing: GA-MITS.

The listing below is specific to a given test set minimization problem. In the case below, it relates to the ISCAS c499 circuit. For each test set minimization problem, there are specific parameters that need to be set in the code. For example, test set size, fault coverage, maximum number of possible faults in a circuit. Comments in the code specify these settings. Other than these settings, the code remains the same for every test set minimization problem.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

/*-----------------------------------------------------------------------------------------------------------------
                                        GA-MITS
                         Genetic Algorithm – MInimisation of Test Sets
                                Author: Jasbir S. Takhar,
                           Sheffield Hallam University, 1996 – 2003

                                Test Set Miminimisation of:
                              ISCAS c499 CIRCUIT: GENETIC ATPG
-------------------------------------------------------------------------------------------------------------------*/


#define MBIG 1000000000        /*   PARAMETERS FOR RAN3          */
#define MSEED 161803398
#define MZ 0
#define FAC (1.0/MBIG)


float ran3(long *idum);
void pop_gen();          /* generates initial population        */
void fitness();
void stats(int gen_count);
void new_gen();          /* generates new population from old one */

int  select(void);
int  mutate(int allele, int count, int child);
int  crossover(int mate1, int mate2,int count);
void stats_f(int no_gens);

int line_count,data_count,count; /* For data extraction              */

/* CHANGE FOR EACH MODEL
    The parameters below define the parameters of the test set to be minimized. Fault count,
     Total number of tests etc.
*/
int (*data)[1202];         /* Subscript = total no. of faults         */
int (*pop)[86];               /* All subscripts = total  no. of            */
int (*new_pop)[86];           /*               possible tests              */
int *tests,*faults;           /* Arrays containing no. of t's and f's      */
int *global_fit;              /* Array containing fittest chromosome   */
int *global_tests;            /* pointer to fittest no. tests              */
int *global_faults;           /* pointer to fittset no. faults             */
int *high_fit;                /* Subscript of the fittest chrom in gen.   */
int *chrom_l;                 /* Length of chromosome = no.tests          */
int *fault_l;                    /* Total number of faults in data            */
int *pop_size;
int *no_gens;
double *fit;
double *glob_fit;             /* Fittness value of global fit chromosome */
double *sum_fit;
double *gen_ave;              /* Array of ave fittnes for each generation        */
double *gen_max;              /*   "  "  max "     "  "  "       */
long idum;
int muts=0;

void main()
{
    int i,j,k;
    int gen_count=0;
    int gens=300;
    int h_fit=0;
    int g_t=0;            /* initialiser for global_tests  */
    int g_f=0;            /*   "       "  global_faults */

/* --------------------- CHANGE FOR EACH TEST SET -----------------------
------------------------------------------------------------------------------------------------ */

            int ch_l=86;              /*  Chromosome length=no. of tests   */
            int f_len=1202;           /*            Total no of faults          */

            int p_size=150;
            double s_fit=0.0;
            double gfit=0;
            char dat[610];          /* must change to accomodate each line of test data */
                                  /* i.e. min. = total no. faults/2               */
          time_t t1;
         FILE *f1;
        chrom_l=&ch_l;
        fault_l=&f_len;
        pop_size=&p_size;
        glob_fit=&gfit;
        no_gens=&gens;
        high_fit=&h_fit;
        sum_fit=&s_fit;
        global_tests=&g_t;
        global_faults=&g_f;
```

```
      t1=time(NULL);
      idum=-(t1%100);

/* ------------------------CHANGE FOR EACH  TEST SET ------------------
  ---------------------------------------------------------------------------------------*/

         pop=malloc(sizeof(int[150][86]));            /* Subscripts=              */
         new_pop=malloc(sizeof(int[150][86]));    /*      [popsize][chrom_l]  */
         data=malloc(sizeof(int[86][1202]));         /* Size of fault matrix      */
         fit=malloc(sizeof(double)* *pop_size);
         global_fit=malloc(sizeof(int)* *chrom_l);
         gen_max=malloc(sizeof(double)* *no_gens);
         gen_ave=malloc(sizeof(double)* *no_gens);
      tests=malloc(sizeof(int)* *pop_size);
      faults=malloc(sizeof(int)* *pop_size);
         printf("\n Processing......\n");
         /* &&&&&&&&&&&        GENERATING DATA              &&&&&&&&&&& */

         if( (f1=fopen("c499gen.txt","r"))==NULL )
          {
                   perror("cannot open file c499gen.txt");
                   exit(EXIT_FAILURE);
          }

          line_count=0;
          while(fgets(dat,sizeof(dat),f1)!=NULL)
          {  /*  printf("\n line_count=%d",line_count);*/
                    count=0;
                  while (dat[count]=='X' || dat[count]=='1' || dat[count]=='0' )
                   {
                           data_count=count*2;

                           if(dat[count]=='X')
                            { data[line_count][data_count]=0;
                                    data[line_count][data_count+1]=0;
                            }
                  else
                           if(dat[count]=='1')
                             { data[line_count][data_count]=1;
                                    data[line_count][data_count+1]=0;
                             }
                             else
                           if(dat[count]=='0')
                                    { data[line_count][data_count]=0;
                                            data[line_count][data_count+1]=1;
                             }
                           count++;
                   }
                           line_count++;
```

```
            }printf("\n line_count=%d count=%d \n",line_count,count);

  fclose(f1);

/* 8888888888888888  GENERATE 2-D DATA ARRAY      8888888888888888 */
            pop_gen();                /*   generate initial population   */
            fitness();          /*   calculate fitness           */
            stats(gen_count);    /*   calculate statistics        */

            for(j=1;j<*no_gens;j++)
            {
         printf("\n %d",gen_count);
               gen_count++;
               new_gen();              /*   generate new population */
               fitness();
               stats(gen_count);
            }
            stats_f(*no_gens);

            for(j=0;j<*no_gens;j++)
                        {  printf("\n gen %d   max_fit=%f  ave_fit=%f ",j,gen_max[j],
                     gen_ave[j]);
                        }
             getchar();

printf("\n the fittest chromosome: tests=%d  faults=%d\n",*global_tests,*global_faults);

for(j=0;j<*chrom_l;j++)
            {
             printf("%d",global_fit[j]);
            }
printf("\n  muts=%d Done!",muts);
getchar();
}

void pop_gen()
{
   float r_num;
   int i,seed,j;

  for(i=0;i<*chrom_l;i++)
   {
      pop[0][i]=1;
   }

     for(i=1;i<50;i++)
      {
        for(j=0;j<*chrom_l;j++)
              {
          r_num=ran3(&idum);
```

```
            if(r_num<0.2)
             {  pop[i][j]=0;
               }
             else{
                 pop[i][j]=1;
               }
              }
        }
     for(i=50;i<*pop_size;i++)
       {
         for(j=0;j<*chrom_l;j++){
               r_num=ran3(&idum);
            if(r_num<0.5)
                        {  pop[i][j]=0;
                        }
                 else
                        {  pop[i][j]=1;
                        }
               }
       }
   return;
}

void fitness()
{
 int chrom,allele,fault_c,i,k,j;
 int fault_counter=0;
 int test_counter=0;
 int *fault_cov=malloc(sizeof(int)* *fault_l);
 int num,denom;
 double frac;

for(chrom=0;chrom<*pop_size;chrom++)        /* loop through population */
 {
            test_counter=0;
            fault_counter=0;
            for(fault_c=0;fault_c<*fault_l;fault_c++) /* set fault_cov[] to zero*/
            {  fault_cov[fault_c]=0;}
            for(allele=0;allele<*chrom_l;allele++)   /* loop through each allele */
              {
                if(pop[chrom][allele]==1){
             test_counter++;               /* test exists, allele=1*/
                for(fault_c=0;fault_c<*fault_l;fault_c++)   /* loop through database*/
                {
                  if( (data[allele][fault_c]==1) && (fault_cov[fault_c]!=1) )
                   {
                      fault_cov[fault_c]=1;
                      fault_counter++;
                    }
                }   /* end of fault counter loop*/
            }
          }                    /* end of allele loop*/
        /*     printf("\n faults = %d   tests=%d",fault_counter,test_counter);*/
        /*  denom=(*fault_l+1)-fault_counter;*/
        fit[chrom]=fault_counter;
        tests[chrom]=test_counter;
        faults[chrom]=fault_counter;
          if( fault_counter==1194)
             fit[chrom]= (100.0*fit[chrom] )/test_counter;
        }
      retum;
  }

 void stats(int gen_count)
{
  int j,x;
  double max_fit=fit[0];
  double min_fit=fit[0];
  double ave_fit=0.0;
  *high_fit=0;
   *sum_fit=fit[0];
   *global_tests=tests[0];
   *global_faults=faults[0];
   for(j=1;j<*pop_size;j++){
      *sum_fit+=fit[j];
       if(fit[j]>max_fit){
         max_fit=fit[j];  /* max_fit is the fitness VALUE of fittest  */
            *high_fit=j;     /* high_fit is the subscript of the fittest */
          *global_tests=tests[j];
          *global_faults=faults[j];
   }

 if(fit[j]<min_fit)
    min_fit=fit[j];
}

ave_fit=*sum_fit/ *pop_size;
gen_max[gen_count]=max_fit;
gen_ave[gen_count]=ave_fit;
if(max_fit>*glob_fit){
    *glob_fit=max_fit;
     for(j=0;j<*chrom_l;j++){
            global_fit[j]=pop[*high_fit][j];
     }
 }
 printf("\n global fit=%f tests=%d faults=%d ",*glob_fit,*global_tests,*global_faults);
 retum;
}

void new_gen()
```

```
{
 int k,j,mate1,mate2,q,r;
 /* peforming elitism: copying high_fit as the top two
     individuals in the new population, new_pop         */
 for(q=0;q<10;q++)
            {
             for(r=0;r<*chrom_l;r++)
                         { new_pop[q][r]=pop[*high_fit][r];
                         }
            }
 for(j=10;j<*pop_size;j=j+2)
  {
   mate1=select();           /* select two mates     */
   mate2=select();
   crossover(mate1, mate2,j);       /* CROSSOVER  MUTATION */
  }
 for(k=0;k<*pop_size;k++){
     for(j=0;j<*chrom_l;j++){
             pop[k][j]=new_pop[k][j];
             }
 }
}
int select()
{
   double partsum;
    int j;
    int s_fitt;
    int mum_s;
    int r_num;
    s_fitt=*sum_fit/1;
    r_num=(int)100000000* ( ran3(&idum) );
    mum_s=r_num%s_fitt;  /* this causes an error if all fitneses are zero */
    if(mum_s==0)
       return 0;
    else{
      j=0;
      partsum=0;
      while(partsum<mum_s){
        j=j+1;
        partsum=partsum+fit[j-1];
      }
           return (j-1);
     }
}

int crossover(int mate1, int mate2, int count)
{
     int xpoint1,xpoint2,r1,r2,j,i;
     int *parent1=malloc(sizeof(int)* *chrom_l);
     int *parent2=malloc(sizeof(int)* *chrom_l);
    float cross_ran;

    /* FUNCTION TO IMPLEMENT 2-POINT CROSSOVER   */
    /* BY GENERATING 2 RANDOM CROSSOVER POINTS.  */
    /* IF THESE ARE EQUAL, EFFECTIVELY PERFORMING */
    /* ONE POINT CROSSOVER                       */

     for(j=0;j<*chrom_l;j++){
       parent1[j]=pop[mate1][j];
       parent2[j]=pop[mate2][j];
       new_pop[count  ][j]=parent1[j];
       new_pop[count+1][j]=parent2[j];
     }

    cross_ran=ran3(&idum);
    if(cross_ran>0.1) {   /* CROSSOVER PROBABILITY  */
      r1= (int)100000000* ( ran3(&idum) );
      r1= (r1 % (*chrom_l-1) )+1;
      r2= (int)100000000* ( ran3(&idum) ) ;
      r2= (r2 % (*chrom_l-1) )+1;
      if(r1<r2){
        xpoint1=r1;
        xpoint2=r2;
      }
      else{
            xpoint1=r2;
          xpoint2=r1;
       }
    for(j=0;j<*chrom_l;j++){
     if( j>=xpoint1 && j<xpoint2){
       new_pop[count  ][j]=mutate(parent2[j],j,1);
          new_pop[count+1][j]=mutate(parent1[j],j,2);
     }
     else if(xpoint1==xpoint2 && j<xpoint1){
       new_pop[count  ][j]=mutate(parent2[j],j,1);
       new_pop[count+1][j]=mutate(parent1[j],j,2);
     }
     else
     {
        new_pop[count  ][j]=mutate(parent1[j],j,1);
        new_pop[count+1][j]=mutate(parent2[j],j,2);
     }
    }
  return 0;
}

int mutate(int allele,int count,int child)
{
 int mut;
 float m_prob;
 mut=allele;
 m_prob=ran3(&idum);
  if(m_prob<0.02){
      muts++;
      if(allele==1){
          mut=0;
      }
     if(allele==0){
        {mut=1;
      }
 }
return mut;
}

void stats_f(int no_gens)
{
               int j;
               FILE *fp;
               if( (fp=fopen("c499res.txt","wt") )==NULL)
                   printf("error can't open file");
               else
               {
                 for(j=0;j<no_gens;j++)
                   {
                     fprintf(fp,"%d\t%f\t%f\n",j,gen_ave[j],gen_max[j]);
                   }
            fprintf(fp,"\n the fittest chromosome: tests=%d
                   faults=%d\n",*global_tests,*global_faults);
            for(j=0;j<*chrom_l;j++){
                 fprintf(fp,"%d",global_fit[j]);
            }
        }
        fclose(fp);
       return;
}

float ran3(long *idum)
{
 static int inext, inextp;
 static long ma[56];
 static int iff=0;
 long mj,mk;
 int i,ii,k;
 if(*idum <0 || iff == 0)
   {
     iff=1;
     mj=MSEED-(*idum < 0 ? -*idum : *idum);
     mj %= MBIG;
     ma[55] = mj;
     mk=1;
```

```
    for(i=1;i<=54;i++)
     {
      ii=(21*i) % 55;
      ma[ii]=mk;
      mk=mj-mk;
      if (mk < MZ) mk += MBIG;
      mj=ma[ii];
     }

    for(k=1;k<=4;k++)
     for(i=1;i<=55;i++)
        {
         ma[i] -= ma[1+(i+30) % 55];
         if(ma[i]<MZ) ma[i]+=MBIG;
        }
      inext=0;
      inextp=31;
      *idum=1;
     }

    if (++inext == 56) inext=1;
    if (++inextp == 56) inextp=1;
    mj=ma[inext]-ma[inextp];
    if (mj < MZ) mj += MBIG;
    ma[inext]=mj;
    return (mj*FAC);
}
```

Appendix B. Publications

The Derivation of Minimal Test Sets for Combinational Logic Circuits using Genetic Algorithms

Jasbir S. Takhar and Daphne J. Gilbert
School of Science and Mathematics
Sheffield Hallam University
Sheffield, S1 1WB, UK

Abstract - *To reduce the post-production cost of testing digital circuits, the derivation of minimal test sets is an important issue. The technique presented here applies a Genetic Algorithm to find minimal or near minimal test sets. The algorithm aims to minimise test sets that have been previously generated by an ATPG system and as such has been designed as a post-processor. The algorithm has been applied to a family of RISC (Reduced Instruction Set Computer) processors and a selection of ISCAS-85 benchmark circuits.*

I. INTRODUCTION

For today's complex digital circuits, the derivation of high coverage, minimal test sets is an important issue. Although capable of achieving high fault coverages, VLSI test generation systems such as TOPS [1] and Turbo-Tester [2] do not address this issue. However, after a test set with adequate fault coverage has been generated by ATPGs (Automatic Test Pattern Generators) or other methods, test set minimisation algorithms may be applied as a separate process. As such, this is a one-off cost and may result in significant time/cost savings in post production unit testing.

This paper describes the application of a Genetic Algorithm (GA) [3-6] to derive minimal test sets for digital, combinational circuits. Based loosely upon the Darwinian ideas of evolution and natural selection [7], they have proven themselves to be practical, robust optimisation tools, well suited to the NP-hard problem of deriving optimal test sets. They have been applied successfully in a variety of VLSI design/test contexts [8-9]. O'Dare and Arslan [10] have applied them to generate test vectors and high coverage test sets and Aylor et al [11] have addressed the compaction issue but using fault simulations in addition to a 'covering heuristic'. The technique described here optimises test sets that have previously been generated by ATPG systems. As such it may be seen as a 'bolt-on' function for such an ATPG system.

II. OUTLINE OF THE MINIMAL TEST SET PROBLEM

The techniques described in this paper apply to combinational logic circuits and the single stuck-at fault model [12]. A minimal test set can be defined as follows:

For a given set S of detectable faults a set of test vectors is said to be a minimal test set if it contains the least number of test vectors required to cover all the faults in S.

Using techniques such as path-sensitisation, test vectors can be derived for most, if not all the given faults in a circuit. These test vectors therefore comprise a test set for the circuit. However, it often occurs in practice that a single test vector can cover several faults and that many of the faults are covered by more than one test vector; it is this overlap that can be exploited to reduce the size of the test set. A *fault matrix* [12] such as that given in Table 1, displays the fault coverage of each test vector (*k*/0 , *k*/1 denote the faults *k* stuck-at-0 and *k* stuck-at-1 respectively and √ indicates each fault that is covered by a given test vector).

To illustrate these ideas and the algorithm, consider the simple combinational logic circuit shown in Figure 1. It has 2^3 possible test vectors and since each line can be stuck at either one or zero, the circuit has 10 possible faults. The fault matrix for this circuit is given in Table I.

From the matrix it is readily seen that there are two minimal test sets, viz. (010, 100, 101, 110) and (001, 010, 101, 110) since each of these covers all ten faults. Both test sets are 50% of the original size and this translates into a similar reduction in the test overhead. In practice the reduction is often greater, but even small reductions in test set size can result in significant savings for the manufacturer who is faced with testing millions of units per annum [13].

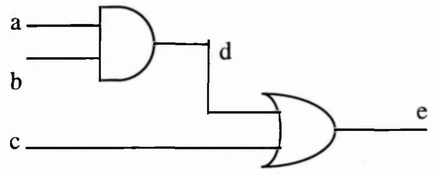


Fig. 1. A 3 input combinational logic circuit

TABLE I: FAULT MATRIX FOR CIRCUIT IN FIG. 1.

Test	Fault									
abc	a/0	a/1	b/0	b/1	c/0	c/1	d/0	d/1	e/0	e/1
000						√		√		√
001					√				√	
010		√				√		√		√
011					√				√	
100				√		√		√		√
101					√				√	
110	√		√				√		√	
111									√	

This simple example was designed as an illustration of the problem and its solution. In this case all of the minimal test sets could be identified by inspection. However, for a circuit of any reasonable size, it may not be realistic or even possible to obtain a complete solution to the minimisation problem. Instead a significant reduction in the size of the test set, together with reasonable fault coverage is sought. GAs are extremely well suited for such combinatorial optimisation problems.

III. GENETIC ALGORITHMS

Genetic Algorithms (GAs) draw inspiration from the biological and evolutionary processes of *selection*, *crossover* and *mutation*. They use directed, probabilistic search techniques to find globally optimal solutions in large, complex search spaces. GAs contain populations of N candidate solutions called *chromosomes*. In a binary encoded GA, chromosomes consist of bit strings, each bit being referred to as a *gene*. Associated with each chromosome is a fitness value (dependent on the application) which rates its competence as a solution. After an initial population has been randomly generated, subsequent generations (each containing N chromosomes) evolve by *mating* fit individuals from the current population.

In nature individuals compete for food, mates etc. resulting in the fittest surviving to produce offspring, a phenomenon called "survival of the fittest". Similarly, in GAs the probability of selecting a chromosome as a mate is proportional to its fitness. Once selected, the two parents exchange genetic material by crossover to produce two children. Application of the mutation operator results in the random alteration of genes which enables other candidate solutions, which might not have arisen through crossover alone, to be explored.

IV. THE POST- PROCESSING MINIMAL TEST SET PROBLEM IN A GA CONTEXT

The objective of the GA is to evolve test sets that are successively more minimal from an initial, randomly generated population (population size N, ranging from 100 to 200 chromosomes depending on the circuit). The fitness of each test set is based on its competence as a minimal test set, i.e. as one which covers a given number of faults with the least possible number of test vectors. The fitness of each chromosome is determined by the fitness function (discussed in more detail below) and is essentially a mathematical formulation of the aforementioned criteria. The fittest chromosomes are chosen as mates to produce the next generation since they contain good genetic building blocks from which the subsequent generations evolve.

As the generations arise the GA will produce even fitter chromosomes than were present in the previous generation, until eventually the fitness of the fittest chromosome in the population ceases to increase. It is at this point that we have our minimal or near minimal test set. To emphasise the scale of this problem, the simple circuit in Figure 1 would require exhaustive assessment of $2^8 - 1$ possible test sets for their fitness in order to determine the minimal test set(s). The size of the problem obviously increases exponentially as the number of test vectors increases. The execution time of the GA however, does not increase exponentially with circuit size.

A. The Chromosome Structure

The first stage in applying a GA to any problem is translating it into the GA framework. This is achieved by defining a coding scheme from which we can produce the chromosomes. In the present context it is appropriate to choose each chromosome to represent a test set; each chromosome is then of length l, where l is equal to the number of test vectors produced by the ATPG system, preceding minimisation. A gene value (allele) of 1 in the chromosome represents the presence of a test and a 0 the absence of it, with the first gene representing test vector 1, the second test vector 2, and so on. A typical chromosome for our example circuit is

$$(1\ 0\ 1\ 0\ 0\ 0\ 1\ 1)$$

which represents the test set (000, 010, 110, 111).

B. The Fitness Function

At the heart of all GAs is the Fitness Function and its appropriate formulation is a key factor in the successful application of the algorithm. In our problem the fitness function must reflect the definition of a minimal test set. The definition given earlier is for the general case in which no prior knowledge of the maximal fault coverage attainable or of the size of a minimal test set is assumed. However, since our GA has been designed as a post-processor for ATPG systems the number of testable faults to be covered, and hence the maximal fault coverage that can be attained using our algorithm, is known for a given circuit. All that remains to be found is the size of the minimal test set.

It is apparent that the GA must optimise two parameters; the fault coverage *N*(*C*) (to a known value) and the number of test vectors *N*(*T*) (to an unknown value). The search for the minimal test set(s) is therefore a two stage problem: the first being to achieve the known, maximal fault coverage and the second being to find the minimal number of test vectors that achieves this level of fault coverage. It was found through experimentation that the fitness function, *F*, given below satisfies the aforementioned criteria.

$$F = \begin{cases} N(C) & N(C) < \text{max fault coverage} \\ k \times \dfrac{N(C)}{N(T)} & N(C) = \text{max fault coverage} \end{cases}$$

where *k* is a constant whose value is greater than the total number of test vectors generated by the ATPG system for the given circuit.

To increase the speed of the algorithm it was decided to 'inoculate' [14] the initial population with a single chromosome containing alleles of 1 throughout. This ensures that we have a (relatively unfit) chromosome with maximum coverage. The value of *k* is set so that the fitness of this chromosome is only slightly greater than those that do not achieve maximal fault coverage. This ensures this inoculated individual does not dominate the reproduction process. The genetic material contained in the remaining randomly generated population is necessary for the minimal test sets to evolve.

C. Selection, Crossover and Mutation

The standard biased roulette-wheel selection method [3] was used for selecting the parents. In this method each chromosome is allocated a slot on a notional roulette wheel, the size of each slot being proportional to the chromosome's fitness. A randomly generated number (to simulate a spin of the wheel) decides which slot has been chosen. The greater the fitness of an individual, the greater the probability of it being selected - the survival of the fittest metaphor.

The GA community has devised several forms of crossover, each with its merits [6]. The method employed in the present case is two point crossover. Two crossover points, each lying between the first and last gene are chosen at random. The segments of the parent chromosomes lying between these points are then exchanged to create two child chromosomes. Once selected, parent pairs do not always exchange genetic information, but do so with a given probability known as the crossover probability, *c*. This was set at 90%.

Mutation describes the random alteration of genes within chromosomes. In nature this is a relatively rare event and this is reflected in our algorithm. The mutation rate, *m* was set at 2%, that is, each gene has a 2% chance of flipping its value from a 1 to a 0 or vice versa.

V. Implementation

The Genetic Algorithm was written using the 'C' programming language, developed under UNIX. The hardware platform used was a 100MHz Pentium Personal Computer.

The implementation of the algorithm itself is outlined by the high-level description given in Fig. 2. As can be seen, once the fault matrix data has been read from the ATPG system, a population of test sets is randomly generated. The algorithm then enters the main loop, iterating through *G* generations. The first function in this loop evaluates the fitness of the current population of test sets. Then using selection, crossover and mutation, the next generation is created and the loop continues. Once all *G* generations have evolved, the algorithm returns the fittest, minimal test set found. The values of the parameters *N*, *G*, *c* and *m* were found through experimentation.

VI. Results

Experiments have been carried out on two classes of digital circuits; the first being a family of simplified RISC processors (4, 8, 16 and 32 bit, containing only the arithmetical and logical parts) and the second being a selection of ISCAS-85 benchmark circuits. For each RISC processor, two sets of test patterns were generated, one using a functional generator and the other using a random test generator. A random test generator alone was used for the ISCAS-85 circuits. The results are given in Tables II and III.

As can be seen, the GA is able to minimise the test sets with good results. For the RISC processors, the test sets generated by the functional generator have been compressed, on average by approximately 55%. For the same circuits but test sets generated by the random generator, the compression rates range from 7% to 15%. From Table III it can be seen that the GA is also able to minimise the test set sizes for the ISCAS-85 benchmark circuits. The compression rate varies from 13% for circuit c432 to over 33% for circuit c7552.

Further discussion of the results for the RISC processors is worthwhile as they indicate the types of scenarios in which the GA is able to perform best. Although the sizes of the original functionally generated test sets are greater than those generated by the random test generator, the final, minimised test sets for the functional generator are significantly smaller than those corresponding to the randomly generated test sets. This suggests that the greater the overlap in fault coverage by the original test vectors, the greater the scope for minimisation by the GA. The random generator inherently performs a type of test set minimisation and this seems to limit the amount of compression that the GA achieves.

```
Read fault matrix data from ATPG system
Set GA parameters
    population size, N
    no. generations, G
    crossover probability, c
    mutation rate, m

Generate initial random population of test sets
For each generation
  { Evaluate fitness of each test set
    Select N/2 parent pairs
        For each parent pair
          {
            randomly generate 2 crossover points
                For each gene
                  {     apply crossover and mutation operators
                  }
          }
  }
Return fittest, minimal test set found after G generations
```

Fig. 2. High level description of GA

TABLE II
TEST SET MINIMISATION RESULTS FOR FAMILY OF SIMPLIFIED RISC PROCESSORS

		Functionally Generated Test Sets				Randomly Generated Test Sets			
Proc.	Total No. Faults	Original Test Set Size	No. Faults Covered	Test Set Size After Minimisation	% Reduction	Original Test Set Size	No. Faults Covered	Test Set Size After Minimisation	% Reduction
4 bit	612	63	611	24	62	24	611	22	8
8 bit	1168	63	1167	28	55	34	1167	29	15
16 bit	2240	63	2239	28	55	39	2239	36	8
32 bit	4402	63	4401	28	55	44	4401	41	7

TABLE III
TEST SET MINIMISATION RESULTS FOR ISCAS-85 BENCHMARK CIRCUITS USING RANDOM TEST SET GENERATOR

Circuit	Total No. Faults	Original Test Set Size	No. Faults Covered	Test Set Size After Minimisation	% Reduction
c432	974	54	928	47	13
c1908	2788	138	2775	114	17
c3540	5568	180	5308	137	24
c7552	11590	187	10956	125	33

VII. Conclusion

In each case considered, the GA was able to minimise the test sets with good results. Compression rates of up to 33% were achieved for the ISCAS-85 circuits and up to 62% for the RISC processors.

The results indicate that there is a strong correlation between compression rate and the method used to generate the original test sets. From Table II, for a given circuit and fault coverage, the original functionally generated test sets are larger than the randomly generated ones. However, the functionally generated, minimised test sets are usually smaller than the minimised, randomly generated ones. There is a natural explanation for this. Because there is greater overlap in fault coverage in the functionally generated test sets there is a larger choice of test sets with maximal coverage and hence greater scope for minimisation.

The test process can account for up to one third of manufacturing budgets [13]. The incorporation of an algorithm such as the one presented here into existing ATPGs could contribute significantly to the reduction of such large test overheads.

Acknowledgments

The authors wish to thank Prof. Raimund Ubar and his team, in particular Jaan Raik and Antti Markus, at Tallinn Technical University, Estonia, for generating all the fault matrices which were used to obtain the results described in this paper.

References

[1] Kirkland T., Mercer M.R., "A topological search algorithm for ATPG" Proceedings of the 24th ACM/IEEE Design Automation Conference, pp. 502-508, June 1987.

[2] Ubar, R., "Test synthesis with alternative graphs", IEEE Design and Test of Computers, Spring 1996, pp 48-57.

[3] Goldberg D., *Genetic Algorithms in Search, Optimisation & Machine Learning*, Addison-Wesley, USA, 1989.

[4] Srinivas M., Patnaik, L.M., "Genetic algorithms: a survey", IEEE Computer, 1994

[5] Davis L., ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[6] Mitchell M., *Introduction to Genetic Algorithms*, The MIT Press, Cambridge, USA, 1996.

[7] Holland J.H. , *Adaption in Natural and Artificial Systems*, The MIT Press, Cambridge, USA, 1993.

[8] Rudnick E.M., Holm J.G. , Saab D.G, Patel J.H., "Application of simple genetic algorithms to sequential circuit test generation", Proc. European Design and Test Conf. , 1994, pp 40-45.

[9] Srinivas M., Patnaik, L.M., "A simulation based test generation scheme using genetic algorithms", 6th International Conference on VLSI design, 1993, pp 132-135.

[10] O'Dare M.J., Arslan T., "Generating test patterns for VLSI circuits using a genetic algorithm", Electronic Letters, May 1994, Vol. 30, No. 10, pp 778-779.

[11] Aylor J.H., Cohoon J.P., Feldhousen E.L., Johnson B.W., "GATE- a genetic algorithm for compacting randomly generated test sets", The International Journal of Computer Aided VLSI Design 3, 1991, pp 259-272.

[12] Wilkins B. R. , *Testing Digital Circuits*, Chapman and Hall, UK, 1986.

[13] Thomson K. M., "Intel and the myths of test", IEEE Design and Test of Computers, Spring 1996, pp 79-81.

[14] Surry P.D., Radcliffe N.J., "Inoculation to initialise evolutionary search", Evolutionary Computing: AISB Workshop, Ed. T. Fogarty, Springer-Verlag,1996.

Test Pattern Generation for Multiple Output Digital Circuits using Cubical Calculus and Boolean Differences

Jasbir S. Takhar and Daphne J. Gilbert
School of Science and Mathematics
Sheffield Hallam University
Sheffield, S1 1WB, UK

Abstract - *A new method is presented for generating test patterns for multiple output combinational circuits. Formal mathematical techniques, involving the cubical calculus and Boolean differences, are used to generate test patterns thus dispensing with the costly process of fault simulations. The methods provide the basis for test generation algorithms which are suitable for computer implementation, and also enable testability measures such as observability and controllability, to be computed with relative ease.*

I. INTRODUCTION

As the complexity of present day circuits rises, many of the problems associated with their test, particularly test pattern generation, increase exponentially. Although various design strategies exist, such as 'design for testability' [1], the issue of test pattern generation itself is still very much alive.

This paper describes a method for generating tests for multiple output combinational circuits, based on formal mathematical techniques, thus avoiding costly circuit simulations. The use of Boolean differences [2] to generate tests is often referred to in undergraduate texts [3,4] but is far from widespread in real test systems. This is due to the difficulties associated with computing the Boolean difference. The cubical calculus [5] offers a method for computing it with relative ease. This paper furthers the work of Xue and Zhang [6], which describes a single output test generation algorithm and is based on the single stuck-at fault model.

In addition to test pattern generation, the methods described in this paper enable certain testability measures [7,8] to be easily calculated. The relationship between various covers and controllability/observability measures is very close and can be computed quickly and simply.

II. TEST GENERATION USING BOOLEAN DIFFERENCES

A. The Boolean Difference

The Boolean difference is essentially an XOR of two closely related Boolean functions. If g and h are functions then, in the notation of Boolean algebra,

$$g \oplus h \equiv g\bar{h} + \bar{g}h \qquad (1)$$

where $\oplus$ denotes the XOR operation. Consider a Boolean function F(X) of a single output circuit, where $X = (x_1, \ldots\ldots, x_n)$ and the variables $x_1, \ldots, x_n$ represent the primary inputs. The Boolean difference of F(X) with respect to x_i is defined by

$$\frac{dF(X)}{dx_i} = F(x_1, \ldots, x_i, \ldots, x_n) \oplus F(x_1, \ldots, \bar{x}_i, \ldots, x_n)$$

For an internal circuit node, s_j say, the Boolean difference with respect to s_j becomes

$$\frac{dF(X, s_j)}{ds_j} = F(x_1, \ldots, x_n, s_j) \oplus F(x_1, \ldots, x_n, \bar{s}_j)$$

where s_j is regarded as a pseudo primary input. The solution of the Boolean equation

$$dF(X, s_j)/ds_j = 1 \qquad (2)$$

provides us with all the input vectors for which a stuck-at fault on s_j alters the primary output [2].

B. Test Pattern Generation

To generate a test vector for s_j stuck-at-0/1, the node must first be set to 1/0 and the fault propagated to a primary output. An internal node, s_j, can be expressed as a function of the primary inputs, viz. $s_j(X) = s_j(x_1, \ldots\ldots, x_n)$, and the solution of the Boolean equation

$$s_j(X) = k \qquad (3)$$

yields the input vectors that set s_j to k for k=0,1. The input vectors required to propagate a fault at s_j to a primary output are given by the solutions to (2) above. To generate test patterns for a fault on s_j, it is therefore necessary to solve both equations (2) and (3) simultaneously. Hence, for a circuit with n inputs and m outputs, the test sets T_0 and T_1 for s_j stuck-at-0 and stuck-at-1 respectively are given by the solutions of

$$T_0: \quad s_j(X) . \sum_{i=1}^{m} \frac{dF_i(X, s_j)}{ds_j} = 1 \tag{4}$$

$$T_1: \quad \overline{s_j(X)} . \sum_{i=1}^{m} \frac{dF_i(X, s_j)}{ds_j} = 1 \tag{5}$$

where $F_i(X)$ denotes the ith output, for i=1,....,m

III. Cubical Calculus

A. *Introduction and Definitions*

Cubical calculus provides an alternative approach to Boolean functions and is largely attributed to the work of J.P. Roth [5], [9]. In addition to the usual binary values, 0, 1, the 'don't care' symbol, x, is used and allows for very compact and economical representations of Boolean functions. In the context of combinational logic circuits, a 'cube' defines a relationship between the primary inputs and primary outputs. It is written using the notation,

$$a_1 a_2 \ldots . a_n | b_1 b_2 \ldots b_m$$

where $a_1, a_2, \ldots.., a_n$ are the values of the primary inputs and $b_1, b_2, \ldots, b_m$ are the values of the corresponding primary outputs. A *cover* for the function realised by a circuit is a set of cubes that unambiguously defines that function. A single output function can have a 1-*cover*, for which the output is equal to 1 and similarly a 0-*cover*. For example, a two input, one output AND gate is represented by the

$$0-\text{cover}: \begin{Bmatrix} 0\text{x}|0 \\ \text{x}0|0 \end{Bmatrix} \text{ and } 1\text{-cover: } \{11|1\}$$

Cubes consist of coordinates, which can be bound (1/0's) or free (x's). Each 'x' in the input part a cube can be replaced by a '1' or a '0'; an 'x' in the output part of a cube means that the corresponding variable is unspecified. Thus under the usual interpretation the cube x01x|1x0 signifies that if inputs 2 and 3 are set to 0 and 1 respectively, then regardless of the values of the remaining inputs, the values of outputs 1 and 3 respectively are 1 and 0, with no value specified at output 2.

This paper uses and adapts the method of Roth in a situation which involves the concept of pseudo inputs and the combining of functions with different primary outputs. To accommodate this situation, some departure from the usual rules is necessary from time to time, and this is indicated when appropriate.

B. *Cubical Operations Required for Test Pattern Generation*

The three main cubical operations are interface, adjoin and sharp product (# product). These are analogues of the set theoretic operations $\cap$, $\cup$, \ (intersection, union and difference) respectively. Thus if C_1 and C_2 respectively are covers of two functions $F_1(X)$ and $F_2(X)$ having the same inputs and outputs, the interface of C_1 and C_2 is a cover for the product function, $F_1(X) . F_2(X)$. Similarly application of the adjoin and sharp product operators to C_1 and C_2 results in covers for the functions $F_1(X)$ + $F_2(X)$ and $F_1(X) . \overline{F_2(X)}$ (or $\overline{F_1(X)} . F_2(X)$) respectively. We now briefly describe those aspects of these three operations which are needed in the present context.

Adjoin

The adjoin of two cubes $a|b$ and $c|d$, written $(a|b)$ V $(c|d)$ is any set S of cubes which is equivalent to $\{a|b, c|d\}$ in the sense that S and $\{a|b, c|d\}$ contain precisely the same input/output information. It is usual to choose S optimally to have a minimum number of cubes each containing a maximum number of free variables. For example, (1x0x|1) V (0x0x|1) = {1x0x|1, 0x0x|1} = {xx0x|1}. The adjoin of two covers C and D is defined similarly to be any set of cubes which is equivalent to the combined set $\{c \in C\} \cup \{d \in D\}$; where possible an optimal set is chosen.

Interface

The interface of single coordinates is defined as

$$0 \text{ I } 0 = 0 \text{ I x} = \text{x I } 0 = 0 \quad : \quad 1 \text{ I } 1 = 1 \text{ I x} = \text{x I } 1 = 1$$

$$\text{x I x} = \text{x} \quad : \quad 1 \text{ I } 0 = 0 \text{ I } 1 = q$$

where q denotes a conflict. The interface of two cubes is formed using the interface of their individual coordinates. For example, (10|x) I (0x|1) = q0|1. Two cubes are said to be disjoint if their interface contains a conflict. The interface of two covers C and D, written C I D, is given by,

$$C \text{ I } D = \{c \text{ I } d\text{: } c \in C,\ d \in D\}$$

Where possible C I D reduced to an optimal set of cubes as is explained in connection with the adjoin operator. If c I d is disjoint for all $c \in C$ and $d \in D$, then C and D are said to be disjoint, and we write this C I D = $\varnothing$, where $\varnothing$ denotes the empty cube.

Sharp Product

The #-product of two cubes $a|b$ and $c|d$ written $(a|b)$ # $(c|d)$ satisfies

(i) if $a|b$ and $c|d$ are disjoint then $(a|b)$ # $(c|d)$ = $a|b$

(ii) if $b = d$, then $(a|b)$ # $(c|d)$ is a set of cubes of the form $e|b$ whose input parts constitute a cover of the inputs contained in a but not in c.

(iii) if all coordinates in b are bound and all coordinates in d are free then $(a|b)$ # $(c|d)$ is as in (ii).

Note that in (ii) and (iii) the differencing operation # affects the input coordinates only. The following examples illustrate these rules:

10|x0 # 00|x0 = 10|x0 , since these cubes are disjoint.

x0x|1x # 001|1x ={10x|1x , x00|1x} , on the basis of rule (ii)

x10|1x # 01x|1x =110|010 on the basis of rule (iii)

The sharp product a # C of a cube a and a cover $C = \{c_1, c_2, \ldots, c_k\}$ is defined by $a\#C$ = $((a\#c_1)\#c_2\ldots)\#c_k)$ and the #-product of the covers $C\#D$ is given by, $C\#D = \mathrm{V}_{i=1}^{k}(c_i\#D)$. As before the #-product operation is expressed as an optimal set of cubes where possible.

IV. Test Set Generation Using Cubical Calculus

A. Single Output Case

The set theoretic analogue of equation (1) is

$$g\Delta h = (g \setminus h)\cup(h \setminus g) = (g \cup h)\setminus(g \cap h) \qquad (6)$$

which is sometimes referred to as the symmetric difference. Recalling that a cover is a set of cubes that unambiguously defines a function, let C_0 , C_1 and D be the covers for the functions $F(X, s_j = 0)$, $F(X, s_j = 1)$ and $dF(X, s_j)/ds_j$ respectively. The covers C_0 and C_1 can be obtained by using Roth's 'PI Star' algorithm [5] along with the algebraic descriptions of the circuit functions.

It follows from equation (6) above that

$$\begin{aligned} D &= (C_0 \# C_1) \mathrm{V} (C_1 \# C_0) \\ &= (C_0 \mathrm{V} C_1) \# (C_0 \mathrm{I} C_1) \end{aligned} \qquad (7)$$

So, if S_j is a cover for the function $s_j(X) = 1$, then the test sets for s_j stuck-at-0 and stuck-at-1 are given by the covers

$$T_0 = D \mathrm{I} S_j \qquad (8)$$

$$T_1 = D \# S_j \qquad (9)$$

respectively.

B. Multiple Output Case

For the multiple output case, calculation of the Boolean difference requires a slightly different approach. For simplicity, let us consider the three output case. From equations (4) and (5), we need to obtain a cover $\tilde{D}$ for

$$\sum_{i=1}^{3} \frac{dF_i(X, s_j)}{ds_j}$$

For i = 1, 2, 3, let C_0^i , C_1^i denote the covers for the functions $F_i(X, s_j = 0)$, $F_i(X, s_j = 1)$ respectively. Evidently,

$$\tilde{D} = V_{i=1}^{3} \; (C_0^i \mathrm{~V~} C_1^i) \# (C_0^i \mathrm{~I~} C_1^i)\;) \qquad (10)$$

Using elementary set theory, it is straightforward to deduce from (10) that if the covers corresponding to distinct outputs are disjoint, so that

$$C_0^i \mathrm{I} C_0^j = C_0^i \mathrm{I} C_1^j = C_1^i \mathrm{I} C_1^j = \varnothing \quad , i \neq j \qquad (11)$$

is satisfied, then

$$\tilde{D} = (C_0 \mathrm{~V~} C_1) \# (C_0 \mathrm{~I~} C_1) \qquad (12)$$

where

$C_0 = C_0^1 \mathrm{~V~} C_0^2 \mathrm{~V~} C_0^3$, $C_1 = C_1^1 \mathrm{~V~} C_1^2 \mathrm{~V~} C_1^3$. The form (12) of $\tilde{D}$ is more convenient in the present context, and a mechanism for ensuring that the disjointness condition (11) is satisfied, is described in Example 1.

The test sets for s_j stuck-at-0 and stuck-at-1 are given by the covers $T_0 = \tilde{D} \mathrm{I} S_j$ and $T_1 = \tilde{D} \# S_j$ respectively. These two equations are the 'cubical' equivalents of equations (4) and (5).

Example 1

Consider a stuck at fault on line 7 in the circuit given in Fig. 1. If line 7 is treated as a primary input which is set to 0, then the 1-covers of outputs 8, 9 and 10 respectively are {xx11|100,11xx|100}, {xx11|010} and {xxxx|001}, and their adjoin constitutes C_0 . Similarly, C_1 is obtained by assuming line 7 is set to 1, and finding the corresponding 1-cover of the primary outputs, to give

$$C_0 = \begin{Bmatrix} \text{xx11|100} & \text{xx11|010} \\ \text{11xx|100} & \text{xxxx|001} \end{Bmatrix} , \; C_1 = \begin{Bmatrix} \text{xx11|100} & \text{xxxx|010} \\ \text{11xx|100} & \end{Bmatrix}$$

Note that a 1 in the output part of a cube in C_0 or C_1 signifies in the usual way that the value of the corresponding output is 1 for the given input (including line 7). On the other hand, the zeros in the output part of these cubes are simply a device to ensure that cubes corresponding to distinct outputs are disjoint, so that $\tilde{D}$ satisfies (11). Taking the adjoin and interface of C_0 and C_1 gives

$$C_0 \mathrm{V} C_1 = \begin{Bmatrix} \text{xx11}|100 & \text{xxxx}|010 \\ \text{11xx}|100 & \text{xxxx}|001 \end{Bmatrix}, C_0 \mathrm{I} C_1 = \begin{Bmatrix} \text{xx11}|100 & \text{xx11}|010 \\ \text{11xx}|100 & \end{Bmatrix}$$

from which

$$\tilde{D} = (C_0 \mathrm{V} C_1) \# (C_0 \mathrm{I} C_1) = \begin{Bmatrix} \text{xx0x}|010 & \text{xxxx}|001 \\ \text{xxx0}|010 & \end{Bmatrix}$$

which is a cover for the sum (AND) of the Boolean differences of the three output functions with respect to line 7. The cube xx0x|010 in $\tilde{D}$ signifies that if input 3 is set to 0, then regardless of the value of the other inputs, a change of values in line 7 (treated as a primary input) results in a change of value in output 9. The interpretation of the other cubes in $\tilde{D}$ is similar; as with C_0 and C_1, the zeros in the output part of the cubes in $\tilde{D}$ carry no significance.

To deduce the test sets T_0 and T_1 it is now only necessary to restrict the input parts of the cubes in $\tilde{D}$ so that line 7 is set to 1 and 0 respectively. This is achieved using S_7 ={xxx1|xxx, x1xx|xxx} whose input parts provide a 1-cover for line 7. The values of the output entries here are immaterial, and for consistency have been set to x. Using the interface and sharp product of $\tilde{D}$ and S_7 gives

$$T_0 = \tilde{D} \mathrm{I} S_7 = \begin{Bmatrix} \text{xx01}|010 & \text{xxx1}|001 \\ \text{x1x0}|010 & \text{x1xx}|001 \\ \text{x10x}|010 & \end{Bmatrix}$$

$$T_1 = \tilde{D} \# S_7 = \{\text{x0x0}|010 \ , \ \text{x0x0}|001\}$$

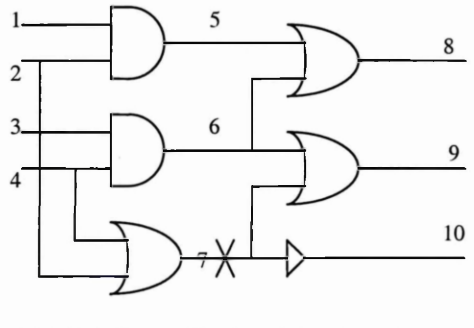


Fig. 1 Multiple Output Circuit with 'stuck-at' Fault on Line 7

which provide the test sets for line 7 stuck-at-0 and stuck-at-1 respectively. Note that $T_0 \mathrm{V} T_1 = \tilde{D}$, and that the presence of a 1 in the output side of a cube in T_0 or T_1 signifies that the fault is observable at the corresponding output. Thus the cube x1x0|010 in T_0 signifies that if inputs 2 and 4 are set to 1 and 0 respectively, then regardless of the values of the other inputs, line 7 stuck-at 0 is observable at output 9 under these input tests.

V. Testability Measures

The ability to assess a circuit's testability has long been considered important. Testability measures such as controllability and observability [10,11] enable the ease (or difficulty) of testing a circuit to be taken into account at the design stage. Many systems have been developed to compute testability measures and are not without their critics [14]. There are many ways of quantifying these measures and what follows is probabilistic, based on the assumption that all input vectors are equally likely.

The *i*-controllability $C_i(s_j)$ of a node s_j is a measure of the ease with which s_j can be set to 0 or 1 and is given by
$C_i(s_j)$ = proportion of input vectors that set s_j to i

Observability is a measure of the ease with which a change of value at a line can be observed at a primary output. For a circuit with m primary outputs, the observability of a line s_j at output k is given by
$O_k(s_j)$ = proportion of input vectors for which a change of value at s_j results in a change of value at k

For a single output circuit, the solution of the Boolean difference equation (1) provides all the input vectors that sensitise a path between the fault site at line s_j and the primary output [2], i.e. for which the value of s_j is observable at a primary output. Hence we can use the 1-cover of the Boolean difference to quantify observability. Also, the solution of the Boolean equation $s_j(X) = 1$ can be used to quantify controllability. So, returning to the discussion of Section IV, the covers S_j and D can be used to calculate the controllability and observability respectively of the node s_j. The calculations are a simple matter of counting the total number of input vectors in each of the aforementioned covers. As an example, a circuit from [14] will be analysed.

Example 2

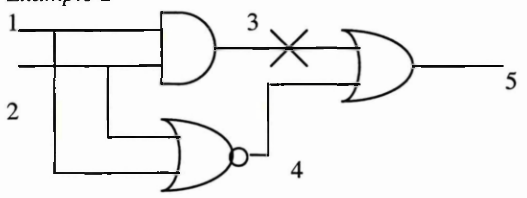


Fig. 2. Combinational Circuit with 'stuck-at' Fault on line 3

We calculate the observability and 1-controllability of line 3 for the single output circuit in Fig. 2.

Since $C_0 = \{00|1\}$, $C_1 = \{xx|1\}$, giving $D = \begin{Bmatrix} 1x|1 \\ x1|1 \end{Bmatrix}$

the total number of cubes in the above cover , D is 3 and the total number of input vectors = 2^2 so $O_5(s_3) = 3/4$. Also the cover $S_3(X) = \{11|x\}$ contains a single input vector so $C_1(s_3) =$ ¼. This example shows that testability measures are very easily calculated from the relevant covers.

VI. Conclusions

In this paper an efficient method has been described for generating test patterns for multiple output circuits. Cubical calculus and set theory provide us with the solution to the Boolean difference equations and the methods lend themselves well to computer algorithms. In addition, testability measures are easily calculated using the covers of various functions.

A significant advantage with the techniques described in this summary is that they can also be used to design minimised logic functions. Therefore, it is possible to produce a complete digital design suite, which could incorporate initial design stages, testability analysis and finally generate test patterns for the completed design. With modern day computing power, such a suite could easily be run on a desk-top machine.

References

[1] T.W. Williams and K.P.Parker, "Design for testability - a survey" in "VLSI TESTING and VALIDATION TECHNIQUES" ed. H.K. Reghbati, North-Holland, p383-395, 1985.

[2] F.F. Sellers, M.Y Hiss, L.W. Bearnson, "Analysing errors with the Boolean difference", IEEE Transactions on Computers, Vol-C17, No. 7, July 1968.

[3] V.P. Nelson et al, "DIGITAL LOGIC CIRCUIT ANALYSIS AND DESIGN", Prentice-Hall, New Jersey, 1995

[4] G. Russell et al, "CAD FOR VLSI", Van Nostrand (UK) 1985.

[5] J.P. Roth, "COMPUTER LOGIC, TESTING AND VERIFICATION" , Computer Science Press, 1980.

[6] H.X. Xue, Y.N. Zhang, "A test generation algorithm based on Boolean differences and cubical operations", New Advances in Computer Aided Design and Computer Graphics, Vol 1 and 2, Ch 166, p634-637, 1993.

[7] K.W. Miller, "Testability - an introduction for COMPASS94", COMPASS 1994, Proceedings of the Ninth Annual Conference on Computer Assurance, 1994, Chapter 26, p173-174.

[8] C.T. Wood, "The quantitative measure of testability", Proc. IEEE Autoscan, p286-291, 1979.

[9] J.P. Roth , "Programmed logic array optimisation", IEEE Transactions on Computers, Vol-**C-27**, Feb. 1978.

[10] L.H. Goldstein, "Controllability/Observability analysis of digital circuits", IEEE Trans. Circuits and Systems, Vol. CAS-26, No. 9, p685-693, Sept 1979.

[11] V.D. Agrawal, M. R. Mercer, "Testability Measures - What do they tell us?" Proc. 1982 IEEE Test

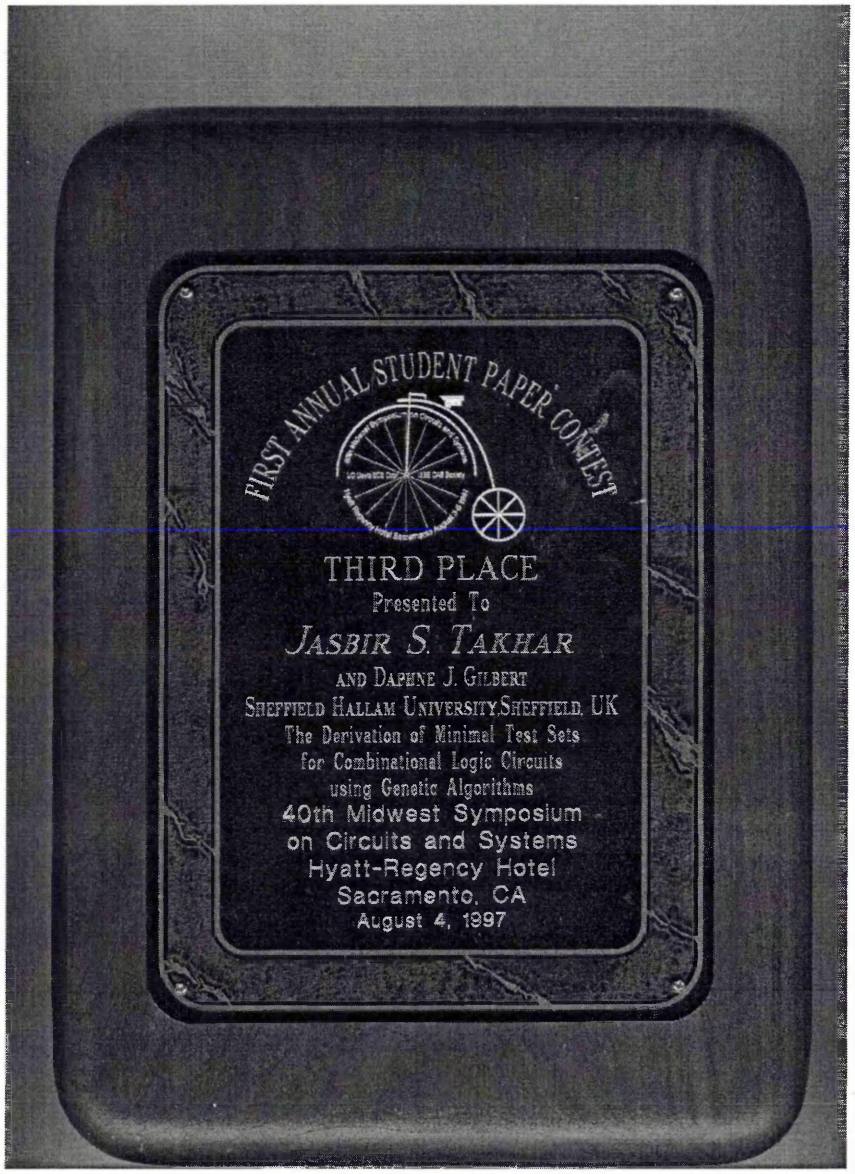
[12] R.G. Bennetts et al., "CAMELOT: A computer aided measure for logic testability", Proc. IEE., Vol 128-E, p177-189, Sept. 1981

[13] J. Grason, "TMEAS a testability measurement program" in Proc. 16th IEEE Design Automation Conference., San Diego, CA, p156-161, June 1979.

[14] J. Savir, "Good controllability and observability do not guarantee good testability", IEEE Trans. on Computers, Vol. **C-32**, p1198-1200, Dec 1983.

Appendix C. Awards

The author was awarded Third Place in the 'Annual Student Paper Contest' at the 40th Midwest Symposium on Circuits and Systems, Sacramento, USA, 1997, for his paper entitled, "The Derivation of Minimal Test Sets for Combinational Logic Circuits using Genetic Algorithms".



FIRST ANNUAL STUDENT PAPER CONTEST

THIRD PLACE
Presented To
JASBIR S. TAKHAR
AND DAPHNE J. GILBERT
SHEFFIELD HALLAM UNIVERSITY,SHEFFIELD, UK
The Derivation of Minimal Test Sets
for Combinational Logic Circuits
using Genetic Algorithms
40th Midwest Symposium
on Circuits and Systems
Hyatt-Regency Hotel
Sacramento, CA
August 4, 1997