Experimenting with Evolutionary Meta-heuristics for State Justification in Sequential ATPG

Syed Zafar Shazli

COMPUTER ENGINEERING

June 2001

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by

SYED ZAFAR SHAZLI

under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee

____
Dr. Aiman H. El – Maleh (Chairman)

____
Dr. Sadiq M. Sait (Co – chairman)

____
Dr. Alaaeldin Amin (Member)

____
Department Chairman

____
Dean of Graduate Studies

____
Date

Dedicated to all those who cared for me

Acknowledgements

All praise be to Allah, Subhanahu-wa-ta-a'la, for his limitless blessing and guidance. May Allah bestow peace on his prophet, Muhammad (peace and blessings of Allah be upon him), and his family. I acknowledge the support and facilities provided by King Fahd University of Petroleum and Minerals, Dhahran, Kingdom of Saudi Arabia.

All my family members, especially my parents, were a constant source of motivation and support. Their prayers, guidance and inspiration lead to the successful accomplishment of this work.

I would like to express my profound gratitude and appreciation to my thesis committee chairman Dr. Aiman El-Maleh, for his guidance, patience, and sincere advice throughout this thesis. He was helpful in every stage of the work and donated a huge amount of time. I also acknowledge the co-chairman of my thesis committee, Dr. Sadiq M. Sait, who introduced me to the area of evolutionary strategies. Thanks are also due to Dr. Alaaeldin Amin for serving on my thesis committee.

I am also thankful to my house mates in NC, Minhas, Junaid, Murtaza and Aamir, who relieved me of many household chores. I also acknowledge the support of other friends at the campus, especially Arshad, Noman, Rais, Salman and Shahzada Bhai who provided a wonderful company. Thanks are also due to Khalid Al-Utaibi for providing me an excellent company in the department and helping me in the Arabic abstract of this thesis.

Contents

List of Tables

List of Figures

THESIS ABSTRACT

Name: SYED ZAFAR SHAZLI

Title: Experimenting with Evolutionary meta-heuristics for State justification in Sequential ATPG

Major Field: COMPUTER ENGINEERING

Date of Degree: June 2001

Sequential circuit test generation using deterministic, fault-oriented algorithms is highly complex and time consuming. New approaches are needed to enhance the existing techniques, both to reduce execution time and improve fault coverage. Evolutionary algorithms have been effective in solving many search and optimization problems. Since test generation is a search process over a large vector space, it is an ideal candidate for evolutionary algorithms. A common search operation in sequential ATPG is to justify a desired state assignment on the sequential elements. State justification using deterministic algorithms is a difficult problem and is prone to many backtracks, which can lead to high execution times. Significant speedups can be obtained with simulation-based approaches. Untestable faults however, cannot be identified using these approaches and deterministic algorithms are needed. In this work, we propose a hybrid approach which uses a combination of evolutionary and deterministic algorithms for state justification. A new method based on Genetic Algorithms is proposed, in which we engineer state justification sequences vector by vector. This is in contrast to previous approaches where GA is applied to the whole sequence. The drawback of previous approaches lies in their inability to justify hard-to-reach states because of fixed-length sequences. Moreover, they do not take into account the quality of intermediate states reached and evaluate a chromosome only on the basis of the final state reached. Our proposed technique overcomes these drawbacks, as it generates the sequences vector by vector. The proposed method is compared with previous GA-based approaches. Significant improvements have been obtained for ISCAS 89 benchmark circuits in terms of state coverage and CPU time.

MASTER OF SCIENCE DEGREE

King Fahd University of Petroleum and Minerals, Dhahran.
June 2001

ملخص الرسالة

الاسم: سيد ظفر شاذلي
العنوان: دراسة تطبيقية لطريقة مركبة لتعيين حالة الدارة أثناء الإنتاج التلقائي لأنماط الاختبار في الدوائر المتسلسلة
المجال الرئيس: هندسة الحاسب الآلي
تاريخ الدرجة: يونيو 2001م

إن إنتاج أنماط الاختبار (Test Pattern Generation) للدارة المتسلسلة باستخدام الطرق الموجهة (Deterministic Methods) أو الخوارزميات التي تعتمد على تعيين الأخطاء مسبقا (Fault-oriented Algorithms)؛ يعتبر غاية في التعقيد، ويستغرق الكثير من الوقت. لذلك نحتاج إلى نهج جديد للارتقاء بالتقنيات المتوفرة لتقليص وقت التنفيذ، وزيادة نسبة اكتشاف الأخطاء.

تعتبر الخوارزميات الارتقائية (Evolutionary Algorithms) فعالة في مجال البحث عن حلول مثلى لمشكلة معينة كإنتاج أنماط الاختبار، التي تعتبر عملية بحث ضمن مجال واسع من الأنماط. كما تعتبر عملية تعيين حالة الدارة (State Justification) في عناصر التسلسل من ضمن العمليات المشتركة بين طرق الإنتاج التلقائي لأنماط الاختبار (ATPG) في الدوائر المتسلسلة.

إن تعيين حالة الدارة باستخدام الخوارزميات الموجهة مشكلة معقدة تتضمن العديد من القرارات التراجعية التي تزيد وقت التنفيذ، حيث يمكن تقليص هذا الوقت باستخدام طرق مبنية على المحاكاة، لكن هذه الطرق تعجز عن اكتشاف الأخطاء غير المستقرة التي يمكن اكتشافها باستخدام الخوارزميات الموجهة. أم في هذه الرسالة فإننا نعرض نهجا مركبا يستخدم مزيجا من الخوارزميات الارتقائية والخوارزميات الموجهة لتعيين حالة الدارة. هذا النهج يعتمد على تطبيق الخوارزميات الجينية (Genetic Algorithms) على سلاسل الحالات المرد تعيينها النمط تلو الآخر على خلاف الخوارزميات الجينية السابقة التي تعمل على سلسلة كاملة. إن مشاكل الطرق السابقة تكمن في عدم قدرتها على تعيين الحالات التي يصعب الوصول إليها بسبب استخدام سلسلة محدودة الطول، بالإضافة إلى أن هذه الطرق لا تأخذ في حسبانها جودة الحالات الوسيطة التي تصل إليها الدارة أثناء عملية التعيين، حيث تقوم بتقييم الكروموزوم بناء على آخر حالة وصلتها الدارة. إن النهج الجديد يتغلب على هذه المشاكل في أثناء إنتاج أنماط الاختبار واحدا تلو الآخر، وهذا النهج تمت مقارنته مع الخوارزميات السابقة، حيث دلت نتائج الدراسة التي أجريت على مجموعة من الدوائر من علامة المقايسة ISCAS 89 على تحسن كبير في نسبة الحالات التي تم تعيينها بإضافة تقليص وقت التنفيذ.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن، الظهران.

يونيو 2001 م

Chapter 1

Introduction

With today's technology, it is possible to build very large systems containing millions of transistors on a single integrated circuit. Designing such large and complex systems while meeting stringent cost and time-to-market constraints requires the use of computer-aided-design (CAD) tools. Increasing complexity of digital circuits in very large scale integration (VLSI) environment requires more efficient algorithms to support the operations performed by CAD tools [1].

1.1 Background

Testing of integrated circuits is an important area which nowadays accounts for a significant percentage of the total design and production costs of ICs. For this reason, a large amount of research efforts have been invested in the last decade in the development of more efficient algorithms for the Automatic Test Pattern Generation

(ATPG) for digital circuits [2]. In order to obtain acceptably high quality of tests, design for testability (DFT) techniques are in use [3]. The first technique, called *full-scan design*, can be used to reduce the sequential test generation problem to a less difficult combinational test generation problem. In this technique, all memory elements are chained into shift registers so that they can be set to desired values and observed by shifting test patterns in and out. In large circuits however, this technique adversely affects the test application time as all the test vectors have to be scanned in and out of the flip-flops. Moreover, all of the memory elements may not be scanable in a given circuit [4]. In order to alleviate the test complexity, a second technique, called *partial-scan design*, is employed. This involves scanning a selected set of memory elements. Both these methods can add 10-20% hardware overhead. In case of a full scan design, a combinational test generator can be used to obtain tests. However, a sequential test generator is necessary in case of a partial scan or no-scan design [4]. In this work, we assume either no-scan or partial scan designs. Generating test sequences for synchronous sequential circuits is a more challenging problem than that of combinational circuits for several reasons. They may be itemized as follows:

- Each fault must be first excited by presenting a given value not only on the primary inputs but also on the flip-flop outputs.
- The difference existing on the fault source between the values of the fault-free

and the faulty circuit must be then propagated to the primary outputs. This is accomplished normally in the next time frames.

- The test sequence is composed of three parts: excitation vector, propagation sequence and state-justification sequence.
- The lengths of state-justification and propagation sequences are not known before hand as they depend on the starting state and the Finite State Machine (FSM).
- Untestable faults require a large amount of time to be identified.

Different approaches have been used to solve the problem of sequential ATPG [5]. They are listed below:

- Using structural ATPG in which learning techniques and heuristics are used to guide the search.
- Using the symbolic approach which exploits techniques which are based on the extraction and manipulation of Boolean functions implemented by the circuit. Typically, these techniques involve Binary Decision Diagrams (BDDs) and are applied at the functional level. They are however, completely inapplicable when dealing with circuits having more than some tens of flip-flops.
- Using a simulation-based approach which consists of generating pseudo-random sequences, fault-simulating them, and then modifying their characteristics to

obtain the required fault coverage.

1.2 Proposed Work

The goal in this work is to use Genetic Algorithms (GAs) for generating sequences that will help the Automatic Test Pattern Generator (ATPG) in detecting more faults by reaching specific states. GAs are very well suited for optimization and search problems [6]. Several ATPGs have been reported which use genetic algorithms for simulation-based test generation. A good comparison is given in [3]. The main advantage of GA-based ATPGs as compared to other approaches, is their ability to cover a larger search space in lower CPU time. This improves the fault coverage and makes these ATPGs capable of dealing with larger circuits. On the other hand, the main drawback consists in their inability to identify untestable faults [2]. Deterministic algorithms for combinational circuit test generation have been proven to be more effective than genetic algorithms [7]. Higher fault coverage are obtained, and the execution time is significantly reduced. However, this is not the case for sequential circuits. These circuits involve justifying state assignments on sequential elements. State justification using deterministic algorithms is a difficult problem, especially if design and tester constraints are considered [8]. In simulation-based ATPGs, the search proceeds in the forward direction only. Hence there are no backtracks and state justification is easier as compared to deterministic ATPGs. In this work, a hybrid state justification approach is proposed, where both deterministic and genetic-based algorithms are employed. In evaluating this approach, we will conduct experiments in which a deterministic test generator will be employed initially. Untestable faults will be identified. The states which could not be reached in this phase, will be attempted in a genetic phase for state justification. Since Genetic Algorithms have been used successfully for combining useful portions of several candidate solutions to a given problem [6], we will try to genetically engineer sequences which justify the leftover states. In the past, Genetic Algorithms have been used for state justification [9]. The length of the sequence was a function of the structural sequential depth of the circuit, where sequential depth is defined as the minimum number of flip-flops in a path between the primary inputs and the farthest gate. In case of feed-back loops, the structural sequential depth may not give a correct estimate of the number of vectors required for justifying a given state. Thus, if a state requires longer justification sequence, it will not be justified. The approach also does not take into account the quality of intermediate states reached and evaluates a chromosome only on the basis of the final state reached. In this work, we will use an incremental approach in which the length of the sequences will be dynamic. State justification sequences will be genetically engineered vector by vector. Even if some state remains unjustified after the genetic phase, the best sequence obtained in a given number of generations will be viewed as a partial solution. The determinisitc ATPG will be seeded with this sequence so that it may become able to reach previously unvisited regions of the search space.

1.3 Organization of the thesis

A brief overview of sequential ATPG is presented in Chapter 2. Chapter 3 discusses Genetic Algorithms in general and their use in Sequential ATPG. The proposed approach of using Genetic Algorithms for state justification is presented in Chapter 4. Experiments performed and discussion on the results obtained are given in Chapter 5. Finally, Chapter 6 offers the conclusions and directions for future research.

Chapter 2

Sequential ATPG

Although scan-based design for testability techniques can convert a sequential circuit into a combinational one for testing purposes, in some cases, the cost of full scan can be prohibitive in both area overhead and performance degradation. Therefore, efficient sequential circuit test generation algorithms are very important for producing high quality VLSI circuits.

2.1 Fault models

An instance of an incorrect operation of the unit under test (UUT) is referred to as an *error*. The causes of the observed errors may be fabrication errors, fabrication defects and physical failures. These are collectively referred to as *physical faults*. Examples of fabrication errors are wrong components, incorrect wiring etc. Fabrication defects usually result from an imperfect manufacturing process. These can include shorts or opens in MOS circuits, and mask alignment errors. Physical failures occur during the lifetime of a system due to component wear-out and environmental factors. Physical faults can be broadly classified as

- *permanent faults*, i.e., always being present after their occurence;
- *intermittent faults*, i.e., existing only during some intervals; and,
- *transient faults*, i.e., a one-time occurence caused by a temporary change in some environmental factor.

In general, physical faults do not allow a direct mathematical treatment of testing. The solution is to deal with *logical faults*, which are a convinient representation of the effect of physical faults on the operation of the system. The basic assumptions regarding the nature of logical faults are referred to as a *fault model*.

2.1.1 Advantages of fault modeling

By modeling a physical fault, the problem of fault analysis becomes a logical rather than a physical problem. Its complexity is greatly reduced since many different physical faults may be modeled by the same logical fault. Some logical fault models are technology-independent in the sense that the same fault model is applicable to many technologies. Hence, testing methods developed for such a fault model remain valid despite changes in technology. Moreover, tests derived for logical faults may be used for physical faults whose effect on circuit behavior is too complex to be analyzed.

2.2 Fault detection in sequential circuits

The goal of sequential circuit ATPG using the single stuck-at fault (SSF) model is to derive an input vector sequence such that, upon application of this input vector sequence, we obtain different output responses between the fault-free and faulty circuits. The SSF model is an abstraction of defects in a circuit which cause a single line connecting components to be permanently stuck either at logic 0 or logic 1 [10]. In this work, we assume the SSF model. In a typical ATPG process, a test or test set is evaluated by the length of the test sequence or the number of test vectors in it, and by the number of faults covered by the test, known as fault coverage. Fault coverage is defined as the fraction of faults in the circuit that is detected by the test sequence [11].

Three separate tasks are necessary to generate a test to detect a fault in a sequential circuit. In the first task, the values of the machine state and the primary input values which excite the fault must be determined. Next, a justification sequence must be derived in order to attain the value of the excitation state on the state bits. Finally, the fault effects must be propagated to a primary output. This goal can seldom be reached in the same time frame which excites it. The fault is

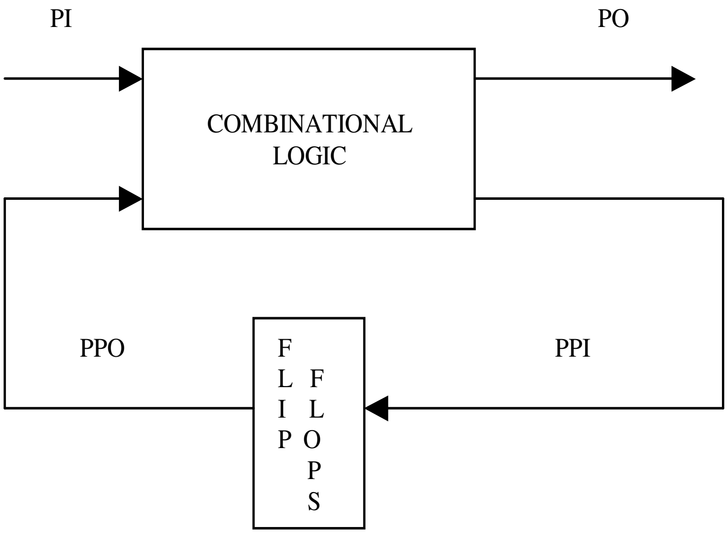

Figure 2.1: A model of synchronous sequential circuit.

mostly first propagated to the flip-flop inputs (Pseudo-primary outputs), and a given number of clock cycles is required to propagate the fault effects from the flip-flops to the primary outputs. This phase is termed as fault propagation [11]. A model of a synchronous sequential circuit is shown in Figure 2.1.

2.3 Terminology

Metrics exist to measure both the resultant quality level which a test set will attain, and the completeness of test generation. *Fault coverage* is defined as the percentage of faults which are detected. *Fault efficiency* is defined as the percentage of faults that are either detected or declared untestable. A fault is said to be *detectable*, if there exists an input sequence such that, for every pair of initial states of the fault-free and faulty circuit, the response of the fault-free circuit is different from that of the faulty circuit. A fault is untestable if it is not detectable. A fault is *partially testable*, if there exists an initial state S^f of the faulty circuit and an input sequence I, such that, for every fault-free initial state S, the response $Z(I, S)$ is different from $Z^f(I, S^f)$. Every fault that is not partially testable, is a redundant fault [12]. Figure 2.2 summarizes the difference between various classes of faults.

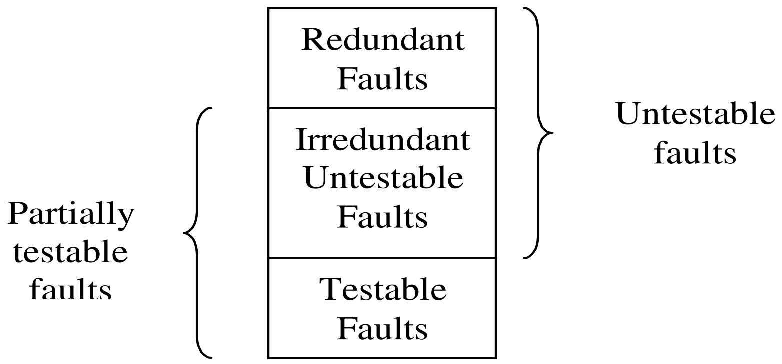

Figure 2.2: Classes of faults.

In case of sequential circuits, the value on the sequential elements (state bits) describes the state of the circuit (machine). To generate tests for a sequential circuit, it is necessary to drive that circuit to a known unique state. A *synchronizing sequence* is a sequence of primary inputs which initializes the state bits to some known state regardless of the prior state of the circuit. The final state reached at the end of a synchronizing sequence is called a *reset state*.

A *cycle* or loop exists in a sequential circuit, when the same node can be revisited after starting from that node and traversing the circuit, while not traversing any other node more than a single time within that traversal. The *length of a cycle* is said to be the number of sequential elements encountered while traversing the cycle [13].

2.4 Complexity of sequential ATPG

Sequential ATPG is a much more complex process than combinational ATPG due to signal dependencies across multiple time frames [13]. It has been shown in [14] that the test generation problem for combinational circuits is NP-complete. The search space is of the order of 2^n, where n is the number of inputs. For sequential circuit ATPG, the worst-case search space is 9^m, where m is the number of flip-flops. This exponential search space makes exhaustive ATPG search computationally impractical for large sequential circuits [4]. In the last years, one of the main goals of researchers was to develop effective algorithms for sequential circuit test pattern generation [15]. A lot of work has been done in the area of sequential circuit test generation using both deterministic and simulation based algorithms. The bottleneck in deterministic algorithms is line justification and backtracking. In simulation-based approaches, no backtracking is required but their quality in terms of fault coverage is generally lower [15]. Several issues contribute to the complexity of sequential ATPG [16]:

- *Number of time frames.* Consider a 20-bit counter. Detecting the s-a-0 fault at the MSB would require a sequence of 2^{19} vectors (to set the MSB to 1). Thus, ATPG would need 2^{19} time frames. No existing logic-level ATPG tool could handle this circuit. Partial-scan techniques may be necessary in such cases.

- *Propagation delay.* It is difficult to accurately model the propagation delay and incorporate it in ATPG because it causes problems for circuits with combinational asynchronous feedback loops and for designs with multiple phases or clocks. A practical solution is to use a simulator, which can model delay and timing accurately, to verify the generated tests and to filter out the tests that cause races and hazards.

- *Existence of loops or cycles.* Sequential circuits have feedback loops. This can result in endless loops during test pattern generation if no special measures are taken. The complexity of sequential ATPG increases in the presence of loops.

- *Sequential depth.* The sequential depth of a circuit refers to the greatest number of sequential elements encountered in a traversal from any primary input to any primary output [13]. The complexity of sequential test generation is directly propotional to the sequential depth of the circuit.

- *Density of Encoding.* Density of encoding, describes the fraction of total number of possible states that are valid. The lower the density of encoding, the higher the probability that the ATPG will select, and subsequently waste time trying to justify, an invalid state [13].

The task of exciting the fault is on the order of complexity of combinational test generation. However, the tasks of state justification and fault propagation both involve traversing distinct states of the circuit. The complexity of state traversal is correlated to the size of the state space which is to be traversed. The total number of possible states in a circuit is 2^{DFF}, where DFF refers to the number of flip-flops in the circuit. However, not all states are necessarily valid. The presence of invalid states is known to increase the difficulty of state traversal [17]. The smaller the percentage of states which are valid, the greater the chance that the test generator will spend time attempting to traverse an invalid state. Thus, the smaller the fraction of the total number of states which are valid, the greater the difficulty of state traversal, and the higher the complexity of sequential ATPG. Traversing all valid states does not guarantee 100% fault efficiency. However, as shown in [17], there is a trend that when the ATPG is able to traverse a majority of the valid states it attains a level of fault efficiency of nearly 100%.

2.5 Classes of sequential ATPG

Several different approaches have been presented for sequential circuit test generation. These can be divided into the following different classes as given in [3].

2.5.1 Structure-based approach

In this approach, tests are generated by activating faults and sensitizing paths for fault propagation through the multiple copies of the combinational circuit. Generally forward time processing (FTP) is used for fault propagation and reverse time processing (RTP) is used for initialization. HITEC [18] is an example of a sequential ATPG that uses both FTP and RTP, and utilizes several techniques to improve the performance of test generation. It also uses a fast circuit simulator, PROOFS [19], which combines the advantages of concurrent, differential and parallel fault simulation algorithms. D-algorithm, PODEM, FAN and BACK are examples of structure-based algorithms. A discussion about these algorithms can be seen in [3], [11] and [20].

2.5.2 State-based approach

The state-based approach uses an abstract model called finite-state machine (FSM), describing the behavior of the circuit. A state transition graph-based (STG-based) algorithm is used to assist the search in the circuit state space. STALLION [21] is an STG-based ATPG which extracts the STG for the fault-free circuit. It finds an activation state S and a fault propagation sequence T for a given fault. Using the STG, STALLION finds a state transfer sequence to bring the circuit from the initial state S_0 to the activation state S. Fault simulation is performed to check the validity of the sequence, and if the generated sequence is not valid, an alternative transfer sequence is generated. Since the extraction of STG is not feasible for large circuits, STALLION constructs a partial STG to overcome this shortfall. STEED is another algorithm which is based on state-transition graphs. STEED has the ability to handle larger circuits than STALLION. A description of the test generation strategy used in STEED can be seen in [3].

2.5.3 Simulation-based approach

The simulation-based approach uses the simulation under the guidance of cost functions. In a typical simulation-based test generation process, a trial sequence is generated. It is then evaluated using logic simulation or fault simulation. A cost function which determines how far the state of the circuit is from the required state for the detection of the fault is computed. If the cost is reduced, the trial vector is included in the final test sequence, otherwise, it is discarded. Simulation-based ATPGs differ in the way they generate new trial vectors and the definition of cost functions for guiding the search. The major advantage of simulation-based ATPGs is their ability to work with more realistic models of the circuit and their simplicity of implementation. However, such ATPGs cannot identify undetectable faults. Their test sets are generally longer and they may fail to generate tests for hard-to-detect faults if they are not guided by a suitable cost function [3]. CONTEST [22], CRIS [23], GATEST [24] and GATTO [25] are some well-known simulation based ATPGs.

A number of test generators use a combination of the methods listed above. The methods may use low cost test generation step, such as random vector generation, or a hybrid of deterministic test generation and fault simulation.

Chapter 3

Use of Genetic Algorithms in Sequential ATPG

3.1 Introduction

In this chapter, we first give an overview of Genetic Algorithms (GA). We then discuss various approaches used in applying GA to sequential Automatic Test Pattern Generation. The last section describes different strategies used for state justification in sequential ATPG using GA.

3.2 An overview of Genetic Algorithms

In this section, we introduce various terminologies used in describing Genetic Algorithms. A structure of the basic Genetic Algorithm is also presented.

3.2.1 The Inspiration of Nature

Genetic Algorithms attempt to mimic nature by evolving solutions to problems rather than designing them. They are a long way from the power of Natural Evolution! Populations of tens or hundreds are common, rather than the millions used in nature; tens or hundreds of generations elapse rather than millions. As a rough estimate, it might be said that Genetic Algorithms are about a hundred million times less effective than Natural Evolution - and this ignores the fact that Natural Evolution builds on its previous discoveries as it creates more complexity [26].

Genetic Algorithms work by analogy with Natural Selection as follows. First, a population pool of chromosomes is maintained. The chromosomes are strings of symbols or numbers. They might be as simple as strings of bits - the simplest type of strings possible. The chromosomes are also called the genotype (the coding of the solution). These chromosomes must be evaluated for fitness. Poor solutions are purged and small changes are made to existing solutions. The gene pool thus evolves steadily towards better solutions.

3.2.2 GA Terminology

The following terms are frequently encountered in GA literature [27].

Chromosome

The structure that encodes how the organism is to be constructed is called a chromosome. In most combinatorial optimization problems, a single chromosome is generally sufficient to represent a solution. Mostly binary encoding is used to represent a solution but some character encodings have also been used [6].

Fitness

The fitness value of an individual (chromosome) is a measure of its closeness to the optimal. The fitness is determined by a *fitness function*. This function generally indicates the cost of the solution.

Selection

In each generation, parents are selected to produce new children. The selection of parents is biased by fitness, so that fit parents produce more children; very unfit solutions produce no children. This is known as selection. The genes of good solutions thus begin to proliferate through the population. In this work, we use Roulette-wheel selection mechanism. In *roulette-wheel method* [26], a wheel is constructed on which each member of the population is given a sector whose size is propotional to the relative fitness of that individual. To select a parent, the wheel is spun and whichever individual comes up becomes the selected parent. Therefore, individuals with lower fitness values also have a finite but lower probability of being selected for crossover. Figure 3.1 shows the roulette wheel for the population whose fitness values and their relative percentages are as follows:

$C_1 = 100 = 10\%$ $C_2 = 250 = 25\%$ $C_3 = 300 = 30\%$ $C_4 = 350 = 35\%$

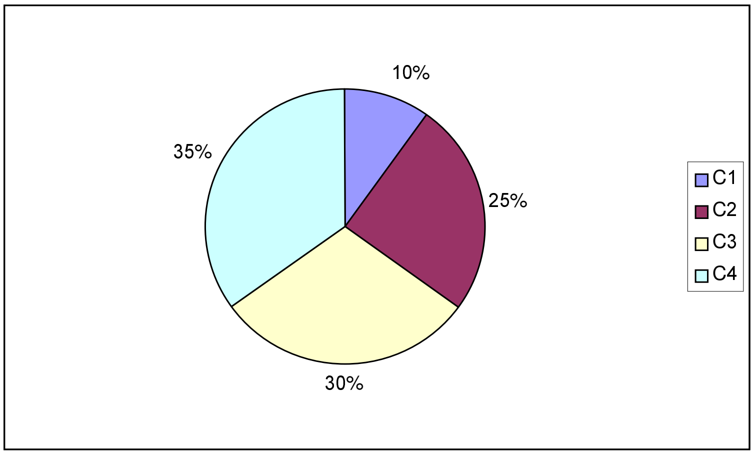

Figure 3.1: A roulette-wheel.

Crossover

Crossover is performed on the parents selected. Useful characteristics of two parents are combined to produce a better individual. There are a number of crossovers reported in literature, and the performance differs in different problems. A good overview of different crossover operators is given in [28]. The following two crossovers have been used more often:

- One point uniform crossover: In a *one-point uniform crossover*, an integer position is randomly selected within a chromosome. Each of the two parents are divided into two parts at this random cut point. The offspring is then generated by catenating the segment of one parent to the left of the cut point with the segment of the second parent to the right of the cut point [27]. For example, consider two parent chromosomes

P_1 : 101101 P_2 : 001100

Let the random cut point be after the third gene from the left. The resulting chromosome after crossover will be

C: 1 0 1 1 0 0

We have used one-point crossover in this implementation.

- Two-point uniform Crossover: In a *two-point uniform crossover*, two random cut points are selected. The offspring is generated by joining together the extreme segments of one parent and the middle segment of the other parent. This has been used in [8].

Mutation

Small changes (mutations) are made to at least some of the newly created children. Some of these mutations may be harmful (in the Natural World, the vast majority of mutations either have no effect or are harmful). However, this doesn't matter as bad mutations will soon be purged by selection. Good mutations, on the other hand, will succeed, causing further increases in fitness.

Generation and Replacement

A *generation* is an iteration of GA, where individuals in the current population are selected for crossover and offsprings are created. Due to the addition of offsprings, the size of population increases. In order to keep the number of members in a population fixed, a constant number of individuals are selected from this set which consists of both the individuals of the initial population, and the generated offsprings. If M is the size of initial population and N_o is the number of offsprings created in each generation, then, before beginning the next generations we select M new members from a population of $M + N_o$ members [27]. Various replacement policies are used for selecting the next generation:

- Tournament Selection without replacement: In *tournament selection without replacement*, two individuals are randomly chosen and removed from the population, and the better of the two is selected. The two individuals are not replaced into the original population until all other individuals have also been removed. Since two individuals are removed from the population for every individual selected, and the population size remains constant from one generation to the next, the original population is restored after the new population is half-filled. Therefore, the best individual is selected twice, and the worst individual is not selected at all. The number of copies selected of any other individual cannot be predicted except that it is either zero,one or two [8].

- Tournament Selection with replacement: In *tournament selection with replacement* the above procedure is repeated except that, the individuals selected are not removed from the old population. Therefore, individuals with higher fitness have a greater probability of being selected again [6].

3.2.3 The Basic Genetic Algorithm

During each generation of the genetic algorithm, a set of offsprings are produced by the application of the crossover operator. This operator ensures that the offsprings generated have a mixture of parental properties. Mutation is also applied with a small probability to introduce diversification in the population. From the entire pool consisting of both the parents and their offsprings, a fixed number of individuals are chosen that form the population of the new generation. The randomly assigned initial pool is presumably pretty poor. However, successive generations improve as we discard unhealthy chromosomes in the population [27]. The structure of a simple genetic algorithm is given in Figure 3.2. This has been taken from [27].

3.3 Literature Review

Several approaches to test generation using genetic algorithms have been proposed in the past [2, 5, 7, 8, 9, 15, 23, 24, 25, 29, 30, 31, 32, 33, 34, 35, 36]. Fitness functions were used to guide the GA in finding a test vector or sequence that maximizes given

Algorithm (Genetic_Algorithm)
  (N_p = Population Size)
  (N_g = Number of Generations)
  (N_o = Number of Offsprings)
  (P_i = Inversion Probability)
  (P_μ = Mutation Probabilty)
  Begin
    (Construct initial population)
    Construct_Population(N_p);
    For j = 1 to N_p
      Evaluate_Fitness (Population[j])
    EndFor;
    For i = 1 to N_g
      For j = 1 to N_o
(Choose parents with probability proportional to fitness value)
        (x,y) ← *Choose_parents*;
        (Perform crossover to generate offsprings)
        offspring[j] ← *Crossover(x,y)*
        For k = 1 to N_p
          With probability P_μ apply *Mutation* (Population[k])
          With probability P_i apply *Inversion* (Population[k])
        EndFor;
        Evaluate Fitness(offspring[j])
    EndFor;
    Population ← Select(Population, offspring, N_p)
  EndFor;
  Return highest scoring configuration in population
End. (Genetic Algorithm)

Figure 3.2: Structure of a simple genetic algorithm.

objectives for a single fault or a group of faults. A major difference in various GA-based approaches lies in the way the fitness is computed. Some techniques use logic simulation for evaluation of candidate vectors or sequences, while other techniques use fault simulation. In addition, there are certain other techniques which target different objectives in various phases of test generation. These techniques typically, use both logic and fault simulation in evaluating candidate sequences. The main advantage of GA-based ATPGs, as compared to other approaches, is their ability to cover a larger search space in lower CPU time. Traversing a larger number of states, improves the fault coverage of sequential ATPGs as mentioned in [17]. However, untestable faults cannot be identified using these approaches [2]. For combinational circuit test generation, deterministic algorithms have proven to be more effective than genetic algorithms [7]. Sequential test pattern generation requires state justification. State justification using deterministic algorithms is a difficult problem, especially if design and tester constraints are considered [8]. A reasonable approach is to use hybrid techniques which include the deterministic algorithm for fault excitation and propagation within a single time frame, and perform state justification using a GA.

3.3.1 Logic Simulation based Test Generators

Genetically engineered sequences were produced by test generators reported in [23], [29] and [30]. The objective in these approaches was fault detection. However, the test generators were unable to drive the fault effects to the primary outputs for hard-to-detect faults.

CRIS

CRIS [23] was the first ATPG to use GAs. The objective function was based on balanced circuit activity (found from logic simulator). Sequences which produced more activity were termed to be more fit. The aim was to detect a higher number of faults. Balanced circuit activity creates randomness in the circuit model. The process of creating balanced randomness has a close relationship with fault coverage as is shown in [30]. Logic activities were monitored during simulation and were used to rank and select the candidate sequences. Vectors were spliced to produce a new candidate sequence. Randomly selected bits were flipped during mutation. The results obtained for ISCAS 89 benchmark circuits were comparable to those of a deterministic ATPG HITEC [18]. On the average, the proposed technique ran 25 times faster than traditional deterministic techniques with very competitive test length and fault coverage for large sequential circuits.

FESTA

Fault detection by state traversal and activity(FESTA) used a fitness function based on logic simulation [29]. The objective of this technique was fault detection. The technique aimed at increasing the number of reached states during the test generation process. GA was applied on sequences and one point crossover was applied at boundaries of the vectors. During the fitness evaluation phase, more importance was given to sequences which were able to explore new states and produce a higher flip-flop activity. Experimental results showed that the approach attained a better fault coverage than state-of-the-art GA-based ATPG in a much reduced time, especially for large circuits. This is because for such circuits logic simulation is far less expensive than fault simulation. The test sets produced by FESTA were also more compact as compared to other GA-based ATPGs.

3.3.2 Fault simulation based test generators

A fault simulator was used for evaluating fitness in [24], [25] and [31]. The fitness functions were biased towards maximizing the number of faults detected and the number of fault effects propagated to the flip-flops.

GATEST

GATEST was implemented around the PROOFS [19] fault simulator and was reported in [31]. Test sequences were used rather than test vectors. The GA generated candidate test sequences, and the fitness was computed by a sequential circuit fault simulator. Fitness was a function of the number of faults detected by the sequence and the number of faults propagated to flip-flops. The objective was detection of a group of hard-to-detect faults. Both binary and non-binary encodings were experimented with. Binary encoding was found to give better results. Crossover was applied at vector boundaries. Uniform crossover gave better results than two-point crossover. Several genetic parameters were experimented with. Best selection scheme was found to be tournament selection without replacement. The number of generations was limited to 8. A population size of 32 was used for all circuits during test sequence generation. Generation gap was also experimented. If the population size is N, and the GA generates g offsprings, then g/N is referred to as the generation gap [26]. A generation gap of $2/N$ gave better fault coverage. In general, fault coverage was higher than CRIS [23] for 17 out of 18 benchmark circuits but the execution time was 6 to 40 times longer.

GATTO

The objective of the test generation run was detection of a specific fault in GATTO [25]. An evaluation function estimated how close the fault is to being detected. The fault with maximum value of evaluation function was selected as the target fault. An individual in the population corresponded to a sequence composed of a variable number of vectors to be applied after the reset state. Each sequence was fault simulated with respect to a target fault. Fitness function was the sum of weighted number of gates with different values in good and faulty circuits and the weighted number of flip-flops with different values in good and faulty circuits. A weight associated with a gate or flip-flop is a measure of its observability. Stopping criterion was the fixed number of generations or the detection of fault. Roulette wheel selection [27] with simple crossover and mutation was used. Length of sequences increased with increasing fault coverage. Sequences with considerable test size were obtained for highly sequential benchmark circuits.

3.3.3 ATPGs using Logic and Fault Simulation

A logic simulator is used along with a fault simulator to evaluate candidate test vectors and sequences in test generators reported in [7] and [32].

ALT-TEST

ALT-TEST repeatedly shifted between GA-based and deterministic test generation [32]. The population size was a function of the length of the sequence. A multiple of the structural sequential depth of the circuit was used as the test sequence length. It used a fault simulator initially to evaluate candidate test sequences and 31 faults were targeted for each test sequence generated. The fitness function then favored visiting more states when the fault detection count dropped significantly. In particular, the fitness of a sequence depended on

- number of faults detected;
- number of flip-flops that carried the fault effects;
- number of new states visited; and,

- number of hard-to-control flip-flops set to specific values.

The fault coverage improved by more than 40% for some benchmark circuits. Moreover, it succeeded in identifying the redundant faults by using deterministic test generation algorithm.

Simple Genetic Algorithm

Test vectors were represented by chromosomes in the population in the ATPG reported in [7]. It was the first scheme in which test sequences were cultivated dynamically. GA was applied on individual vectors. Initial population was randomly constructed. The fitness of a candidate vector was a measure of the number of flip-flops set to 0 or 1 in the first phase. The number of flip-flops that changed value since the last time frame were also included in evaluating the fitness. In phase 2, the objective was to detect the maximum number of faults. Hence, the fitness of a candidate vector was a measure of the number of previously undetected faults detected by the vector. A fault simulator PROOFS, was used for fault detection. Several schemes were applied for selection and crossover. Roulette wheel selection with uniform crossover gave the best results. The test sets obtained by this technique, were smaller than CRIS [23] but fault coverage was lower for highly sequential circuits.

3.3.4 Parallel GA-based implementations

In this section, we discuss two parallel implementations of GA that have been reported for test pattern generation [33], [34].

GATTO*

GATTO* is based on the GATTO tool, and is reported in [34]. It exploits the power of a parallel or distributed system in order to improve the result quality, rather than to reduce the CPU time requirements. PVM library was used and fault coverage were reported for several benchmark circuits. The approach consisted of three phases as in GATTO [25]. In the first phase, test sequence partitioning was adopted, so that every processor fault simulated a different sequence. Every processor executed the same GA experiment in the second phase, aiming at finding a test sequence for the same target fault starting from the same population of sequences. Processors were organized in groups, and the flip-flop weights used by the processors in different groups were different. This orients the search towards different areas of the search space. The best results obtained in every group were analyzed periodically and processors were moved from the most unsuccessful groups to the most successful ones. Thus, the algorithm aims at dynamically evaluating the most suitable parameter values and at increasing the computational resources allocated to the most successful ones. In phase 3, a fault partitioning technique was used, according to which all the processors simulate the same sequence on a subset of the whole fault list. GATTO* was able to produce the best fault coverage reported in the literature for large ISCAS 89 benchmark circuits.

ProperGATEST

Another parallel implementation ProperGATEST, was described in [33]. It consisted of three different stages. GA consisted of a population of individuals, in which each individual represented a sequence of vectors. Each individual was simulated using a group of 31 faults, for the purpose of evaluating the fitness. The following parameters affected the fitness:

- Number of faults in the given fault group detected.
- Number of new states visited.
- Number of flip-flops that carry fault effects at the end of simulation.

In the first stage, the aim was to detect as many faults as possible with short sequences and minimal time. Thus, more weight was given to the number of faults detected. In the second stage, the goal was to maximize visitation of new states and fault effect propagation to flip-flops. In the final stage, the focus was once again shifted towards targeting the remaining hard-to-detect faults. Hence, fault detection and new state identifications were weighted more heavily in this stage. ProperGATEST consisted of three different parallel implementations of ATPG using GA.

In the first algorithm, the fitness evaluation phase was parallelized. Each processor maintains its own copy of the entire population. Fitness evaluation was statically distributed over the processors. The fitness values computed by different processors were communicated to a single processor, which collected the information and broadcasted it to all processors. Each processor then evolved the next generation and computed the best individual in the population for that generation. After a fixed number of generations, the best individual was added to the test set. Fault simulation was then done by all processors using the best-fit individual. The results remained unchanged from the uniprocessor run and execution times were reduced significantly.

The processors exchanged the fittest individual among each other at pre-determined intervals in the second algorithm. The same number of individuals were assigned to each processor and independent fault lists were maintained. The processors traversed different areas of the search space and hence converged to their results faster with the added advantage that the quality of results improved in certain cases.

In the third algorithm, each processor works on a sub-population, and therefore the population size was small. Also, due to migration of fit individuals from one processor to another, each processor detected faults faster than if they were to run independent GAs with a reduced population size. The algorithm provided excellent execution times, but the test sizes were larger than the previous algorithms.

3.4 Conclusion

In this chapter, we have reviewed various GA-based approaches for test generation. We have also presented a brief discussion of Genetic Algorithms. This heuristic is a powerful stochastic iterative heuristic for general combinatorial optimization problems. The test pattern generators developed using GA can be broadly classified in three categories. The first category contains those ATPGs which use logic simulation for evaluation of candidate sequences or vectors. ATPGs which use fault simulation for candidate evaluation come in the second category. There are certain other ATPGs, which use both logic and fault simulation for computing the fitness. These come in the third category. Various test pattern generators which come under these categories, have been discussed in this chapter. In addition, parallel GA-based generators have also been discussed.

Chapter 4

State Justification using GA

4.1 Introduction

State justification is one of the most time-consuming tasks in sequential Automatic Test Pattern Generation (ATPG). For states that are difficult to justify, deterministic algorithms take significant CPU time without much success most of the time. In this chapter, we describe a hybrid approach for state justification. A new method based on Genetic Algorithms is proposed, in which we engineer state justification sequences vector by vector. Previous approaches for state justification using GA are also described.

4.2 Problem Description

Despite improved efficiency, GA-based ATPG algorithms require large amounts of CPU time when dealing with very large sequential circuits. Hence, the less expensive logic simulation has been used to traverse the search space in the test generation techniques proposed in [29], [15], [8] and [10]. Storing the complete state information for large circuits is impractical. Similarly, keeping a list of sequences capable of reaching each reachable state is also infeasible. State justification is therefore performed by using GAs which are very well suited for optimization and search problems. In this work, we propose a hybrid state justification approach, employing both deterministic and genetic-based algorithms. Since Genetic Algorithms have been used successfully for combining useful portions of several candidate solutions to a given problem [6], we try to genetically engineer sequences, vector by vector, to justify the hard-to-reach states.

4.3 Approaches used for state justification using GA

The hard-to-activate faults in sequential circuits sometimes require specific states and justification sequences in order for them to be activated. GA-based test generators discussed in the previous chapter, failed to drive the circuit to these specific states for fault excitation, resulting in low fault coverage for many circuits. GA was used for state justification in [8, 9, 15, 35, 36].

4.3.1 GA-HITEC

A hybrid test generator GA-HITEC was reported in [8]. Deterministic algorithms were used for fault excitation and propagation, and a GA was used for state justification. Sequences were evolved over several generations. The fitness of each individual was a measure of how closely the final state reached matched the desired state. The test generator makes several passes through the fault list, with different conditions and time limits imposed in each pass. Detected faults are removed from the fault list. State justification is performed using a GA in the first pass. A small population size and few generations were used to reduce the execution time in this pass. The search space is expanded in the second pass and GAs are again used for state justification. In the third phase, the deterministic test pattern generator, HITEC, is used for state justification and identifying untestable faults.

4.3.2 IGATE

Methods using finite-state-machine (FSM) sequences were used for fault-effect propagation and state justification in IGATE as mentioned in [9]. GA was used to engineer complete test sequences. A test generator IGATE, based on GA, was developed which targeted individual faults in two phases. The first phase focused on activating the target fault, while the second phase tried to propagate the faults to the primary outputs. The targeted fault is said to be detected at the primary outputs when the faulty machine state is distinguished from the fault-free machine state.

The fitness function used in IGATE depends upon

- fault detection by the individual chromosome;
- sum of dynamic controllabilities;
- matching flip-flop values between the final state reached by the sequence and the target state;
- weighted faulty circuit activity induced by the individual; and,
- number of new states visited by the individual.

The weights given to these parameters were different in the two stages. The faults covered and the length of the test sets were compared with other GA-based approaches for various ISCAS 89 benchmark circuits and significant improvements were reported.

4.3.3 STRATEGATE

A test generator STRATEGATE, was reported in [35]. It was further modified in [36]. It used the linear list of states obtained dynamically during the derivation of test vectors to guide the search during state justification. GA-based techniques were used to engineer valid state justification sequences and the objective was detection of a particular fault. STRATEGATE used several passes through the fault list.

Single time-frame activation was performed in the first phase. The aim in this phase was to engineer a vector, composed of PI and flip-flop values, capable of activating the target fault in a single time-frame. A GA was employed for this stage and dynamic fitness objectives were set up for each target fault. The justification frontier for a SSF at a given line consists of values necessary for justifying a desired value at that line. During the single time-frame fault activation, the fitness function tried to maximize the number of justification frontier values justified. Once a target fault is excited, its fault-effects need to be propagated to at least one PO or flip-flop. The values required at different lines to propagate this fault effect are called the propagation frontier. The fitness function aimed to dynamically advance the propagation of fault effects beyond the current propagation frontier. Once a vector (PI and flip-flop values) was successfully derived, the FF values (state) were relaxed to one that had as many don't cares as possible and could still activate the target fault. The order in which the flip-flops were relaxed was determined in a greedy fashion: from the least controllable to the most controllable flip-flop. State relaxation improves the success rate of state justification which was attempted in the next phase. State justification was performed by using a GA with an initial population consisting of random sequences and any useful state-transfer sequences. Genetic engineering of several sequences was performed to try to justify the target state. Candidate sequences in the GA population were simulated, starting from the current state. The objective was to engineer a state justification sequence that justifies the required state by genetically combining the candidate justification sequences. If a sequence was found that justified the target state, the sequence was appended to the test set and a fault simulator was used to remove any additional faults detected by the sequence. Otherwise, the current target fault was aborted and test generation continued for the next fault in the fault list. The parameters that affected the fitness of an individual in the GA during test generation were as follows:

- *Fault detection*: It is included in the fault activation phase to cover faults that propagate directly to the POs in the time frame in which they are excited.
- *Sum of dynamic controllabilities*: Maximizing it during single time-frame fault activation makes the state more easily justifiable.
- *Matches of flip-flop values*: Matching flip-flop values between the final state reached by the sequence and the target state.
- *Sum of distinguishing powers*: Maximizing this value, increases the probability that the fault effects reach the flip-flops having more powerful distinguishing sequences.
- *Induced faulty circuit activity*: This parameter measures the number of events

generated in the faulty circuit, with events on more observable gates weighted more heavily.

- *Number of new states visited*: This is used to expand the search space and is given a high weight in the final stages.

Parallel-pattern fault simulation [20] was used to speed up the process. 32 candidate sequences from the population were simulated simultaneously during fitness evaluation. Fault-free simulation was performed initially, followed by faulty circuit evaluation, in which events start exclusively from the faulty gate. HITEC [18] was used after the first GA stage to identify untestable faults. The fault coverage improved for most of the ISCAS 89 benchmark circuits when compared with other GA-based ATPGs and HITEC. For circuits where HITEC required long execution times, STRATEGATE reported higher fault coverage and lower execution times.

4.4 Tabu Search

Tabu Search is a general iterative meta-heuristic that is used for solving combinatorial optimization problems. The heuristic is based on selected concepts from Artificial Intelligence. The rules used in Tabu Search are broad enough to make it applicable separately or as a guide for other heuristic procedures applied to combinatorial optimization problems [37].

A key feature of Tabu Search is that it imposes restrictions on the search process preventing it from moving in certain directions to drive the process through regions desired for investigation. An important component that enables Tabu Search to achieve the above-mentioned feature is the use of an adaptive flexible memory. This distinguishes Tabu Search from other memory-less optimization heuristics. Tabu Search is a generalization of a local search. It searches for the best move in the neighborhood of the current solution [27].

4.4.1 Tabu List

At any given point of time during the operation of the algorithm, there are n possible directions for future investigation of the search space, where n is the number of possible moves at that point, and many directions might be overlapping. One of the objectives of Tabu Search is to prevent cycling back to previous solutions [37].

For the purpose of not cycling back to recently visited solutions, certain attributes of the move are made tabu for a specific number of coming iterations called Tabu Tenure or *Tabu List size* [37]. To implement this, a queue of attributes can be used. Tabu Tenure depends primarily on the size of the problem as well as the objective of the search. Generally, the size can be determined using experimental runs.

4.5 Proposed GA-based state justification technique

GA-based state justification approaches described in the previous sections, have a common characteristic of engineering sequences. Deterministic algorithms were used for fault excitation and propagation, and a GA was used for state justification in the test generators reported in [8] and [9]. Sequences were evolved over several generations. The fitness of each individual was a measure of how closely the final state reached matched the desired state. A chromosome was represented by a sequence of vectors. Candidate sequences were simulated starting from the last state reached at the end of the previous test sequence. The objective was to engineer a test sequence that justified the required state. The length of the sequence was a function of the structural sequential depth of the circuit, where sequential depth is defined as the minimum number of flip-flops in a path between the primary inputs and the farthest gate. In case of feed-back loops, the structural sequential depth may not give a correct estimate of the number of vectors required for justifying a given state. Thus, if a state requires longer justification sequence, it will not be justified. The approach also does not take into account the quality of intermediate states reached and evaluates a chromosome only on the basis of the final state reached.

To overcome these deficiencies, we propose an incremental approach in which the length of the sequences is dynamic. State justification sequences are genetically engineered vector by vector. Even if some state remains unjustified after the genetic phase, the best sequence obtained in a given number of generations is viewed as a partial solution and is appended to the justification sequence. Hence, the quality of intermediate states reached while searching for a target state, is also considered while building the justification sequence.

In this work, we use GA for traversing from one state to another. Individual vectors are represented by chromosomes in the population. Deterministic ATPG is run for every target fault. First, the fault is activated and propagated to a primary output. Next, state justification is attempted. If the required state is justified by the deterministic ATPG, then the derived sequence is fault simulated and all detected faults are dropped from the faultlist. Otherwise, our GA-based algorithm attempts to justify the required state. A block diagram of the methodology is shown in Figure 4.1.

4.5.1 Encoding of the chromosome

During generation of individual test vectors, each character of a chromosome in the population is mapped to a primary input. A binary encoding has been used in this implementation.

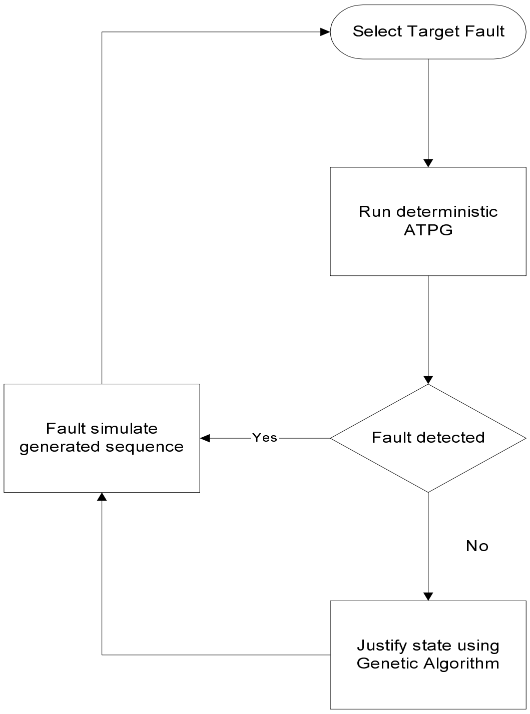

Figure 4.1: A block diagram of the methodology.

4.5.2 Fitness Function

Fitness function is the most important parameter of the GA. A solution is considered to be better than another if its fitness is higher. Each vector (chromosome) is logic simulated to give the state reached. This state is compared with all the flip-flop assignment values of the target state. The fitness $f(v_i)$ of a vector v_i is computed as follows:

$$f(v_i) = \frac{m(s_i,s_j)}{B(s_j)}$$

where s_i is the state reached by vector v_i, s_j is the target state and $m(s_i, s_j)$ is the number of matching specified bits in s_i and s_j. $B(s_j)$ gives the number of specified bits in s_j (i.e., those which are not 'x').

4.5.3 Parent Selection

The choice of parents for crossover from the set of indivduals that comprise the population is probabilistic. It is assumed that stronger indivduals, that is those with higher fitness values, are more likely to mate than the weaker ones. Hence, we select parents with a probability that is directly propotional to their fitness values; the larger the fitness of a certain chromosome, the greater is its chance of being selected as one of the parents for crossover. To accomplish parent selection, we have used the Roulette-wheel selection scheme.

4.5.4 Crossover

Crossover provides a mechanism for the offspring to inherit the characterstics of both the parents. Several crossover operators have been proposed in the literature. Depending on the combinatorial optimization problem being solved some are more effective than others. A good comparison is given in [28]. In this work, we have used one-point crossover.

4.5.5 Mutation

Mutation produces incremental changes in the offspring by randomly changing values of some genes. In this work, mutation corresponds to changing single bit positions. Mutation has the effect of perturbing a certain chromosome in order to introduce new characteristics not present in any element of the population. In our work, bit flipping is used to implement mutation.

4.5.6 Forming a new generation

A generation is an iteration of GA where individuals in the current population are selected for crossover and offsprings are created. Due to the addition of offsprings, the size of population increases. In order to keep the number of members in a population fixed, a constant number of individuals are selected from this set for the new generation. The new population thus consists of both, members from the initial generation and the offsprings created. In this work, we have used three replacement strategies.

$(n+1)$ **replacement strategy**

In this strategy, we change one chromosome in every generation. One crossover is performed in every generation. If the child is more fit than the worst member of the previous generation, it is introduced into the population. Hence, we select the best $n-1$ members from a population of n, and the worst member gets replaced if its fitness is less than the fitness of the offspring.

Random Elitist strategy

We perform $n/2$ crossovers on a population of n chromosomes. This produces n offsprings. The best $n/2$ members are transferred to the next generation. The remaining members of the new generation are selected randomly from the leftover chromosomes.

Roulette Elitist strategy

In this strategy, n offsprings are produced after $n/2$ crossovers in a population of n chromosomes. The best $n/2$ members are transferred automatically to the next generation. Roulette wheel decides the remaining $n/2$ members of the new generation. This gives an advantage to the relatively more fit members of the population to be transferred to the next generation.

The stopping criteria used is the number of generations. Another criteria which has been experimented with is that the algorithm stops searching for the desired state when there is no improvement in the average fitness of the population for a specified number of generations.

4.5.7 Traversing from a state to a state

The algorithm is run for a fixed number of generations. If the state reached is the desired state, the algorithm stops and picks the next state from the list. However, if the algorithm is unable to reach the desired state, it picks the best chromosome found until then and adds it to the test set. Then, it continues the process again by calling the GA starting from this reached state. This state is nearer to the target state in terms of the Hamming distance, which increases the likelihood of reaching the desired state in the next GA runs. The following parameters are used to guide the search

Tabu List Size

To prevent the algorithm from re-visiting recently visited states, we propose a Tabu List containing the last visited states. The length of this list is a user-defined parameter. On reaching a state, the algorithm looks into the Tabu list. If the state reached is present, the next fit vector is chosen and its fitness is evaluated.

Backtrack limit

When all the chromosomes in the population are unable to reach a new state, (a state which is not in the Tabu List), we move to a previously visited state. This is termed as *backtracking*. We impose an upper limit on this parameter and the algorithm stops searching for a state when this parameter exceeds.

Nlimit parameter

The algorithm traverses *at least* Nlimit number of states before it gives up the search for the desired state. If the fitness of the currently visited state is less than the average fitness of the last Nlimit states, the algorithm stops further searching of the desired state; otherwise the search is continued.

A flowchart of the state justification process used in this work is shown in Figure 4.2.

4.5.8 Removing the reached states from the list of desired states

Once a sequence is generated by the algorithm, we compare the states reached by the sequence with the list of desired states. All the desired states reached by the sequence are removed. This prevents us from searching again for those states which we have already reached while searching for some other target state.

4.6 Conclusion

In this chapter, we discussed various approaches in which Genetic Algorithm has been used for state justification. We have proposed a hybrid approach for state justification in which we engineer state justification sequences vector by vector. This overcomes the problem of fixed length sequences associated with earlier ap-

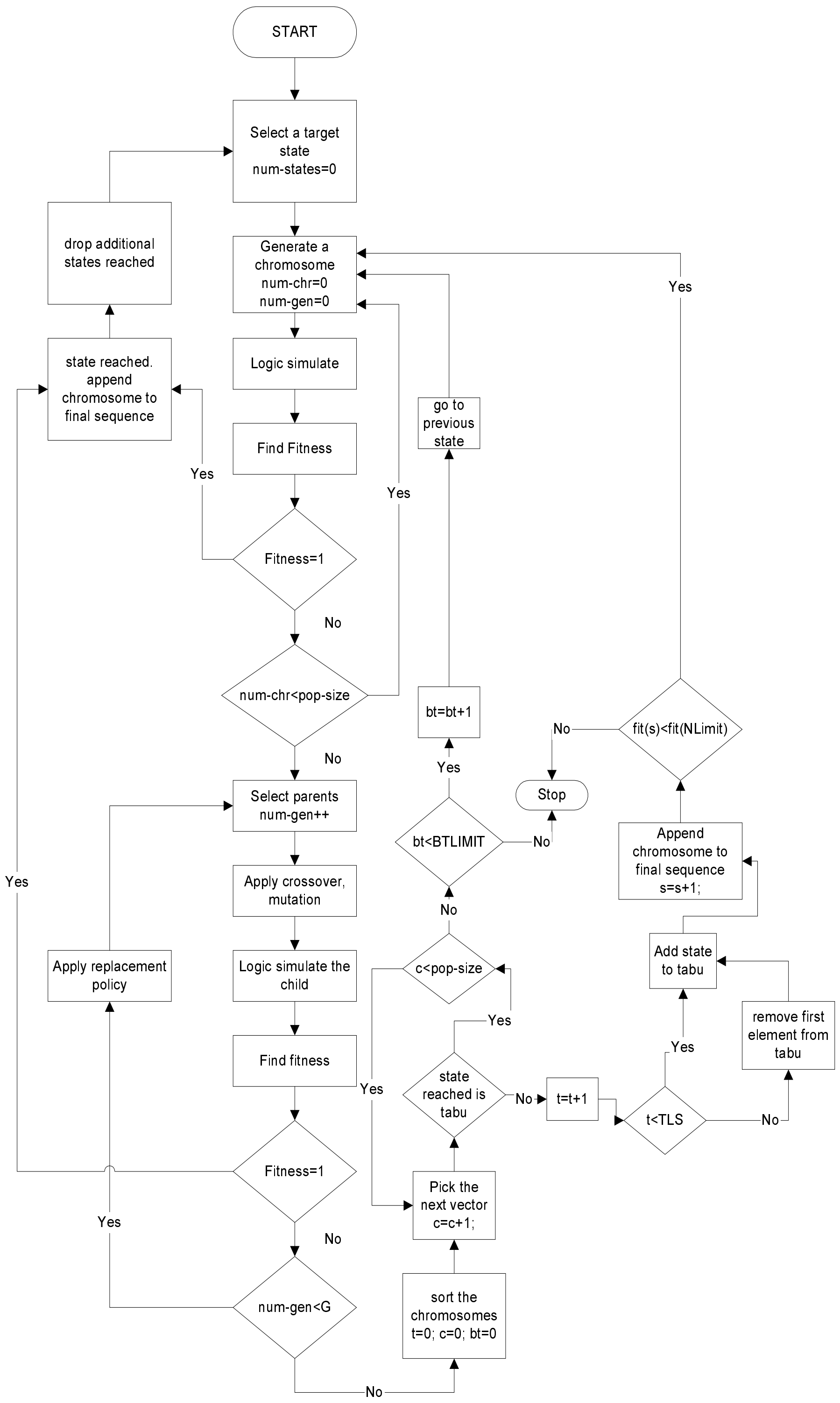

Figure 4.2: A flowchart of the algorithm used.

proaches. Moreover, the proposed approach considers the quality of intermediate states reached while searching for a target state. This is in contrast to earlier approaches, where a sequence was evaluated only on the basis of the final state reached. Various parameters used in our methodology were explained. The fitness function was elaborated and a flowchart of the proposed approach was explained.

Chapter 5

Experiments and Results

5.1 Introduction

In this chapter, we will discuss the experiments performed to evaluate the proposed GA-based state justification technique. The chapter is organized as follows. We will explain the process of obtaining the set of states that will be used to evaluate the effectiveness of the proposed approach in Section 5.2. In Section 5.3, we perform a sensitivity analysis of the parameters used. A comparison of the three replacement strategies mentioned in the previous chapter is also included in this section. Section 5.4 gives a comparison of the proposed approach with the one used in IGATE [9]. A discussion of the results obtained is also given. A conclusion is given in Section 5.5.

5.2 Generation of the desired states for benchmark circuits

In order to test the effectiveness of the proposed GA-based state justification technique, we need to generate a set of states for every benchmark circuit, that are hard to justify by a deterministic sequential ATPG. Typically, an ATPG aborts its search for a test if the backtrack limit is reached. Hence, by stretching the ATPG to a large backtrack limit, we can get a list of hard-to-detect faults. In general, these faults will be the ones that require specific hard-to-justify states. A list of such states is maintained for each circuit under consideration. This step involves a one-time effort and is performed at the beginning of the experimental runs.

5.2.1 Benchmarks used

In this work, we have conducted experiments on ISCAS 89 benchmark circuits. Only those circuits were used for which the deterministic ATPG HITEC [18], was unable to detect the faults after exhausting the backtrack limit of 10^9. In addition, four re-timed circuits given in [13] were considered. Re-timing changes the sequential nature of the circuit, while preserving both the combinational nature and the functionality [13]. The four re-timed circuits used were the ones on which HITEC performed poorly even after spending astronomical amount of CPU time. Thus, re-timing offered a vehicle to analyze the behaviour of complex sequential circuits. Table 5.1

lists the number of primary inputs, primary outputs and D flip-flops of the circuits used in this work.

circuit	# of PI	# of PO	# of DFF
s1423	17	5	74
s3271	26	14	116
s3384	43	26	183
s5378	35	49	179
s6669	83	55	239
scfRjisdre	27	54	20
s832jcsrre	18	19	31
s510Rjcsrre	20	7	30
s510Rjosrre	20	7	32

Table 5.1: The benchmark circuits used.

5.2.2 Obtaining a list of desired states

A list of target states is obtained for each circuit as follows:

- A deterministic test pattern generator HITEC [18] is stretched to a backtrack limit of 10^9 to identify the redundant faults.
- The aborted faults are taken and are converted to their full-scan equivalents
- HITEC then produces a test for each of these faults.
- The required state is then relaxed using PROOFS [19].
- The desired states are merged if they are compatible, as shown in Figure 5.1.

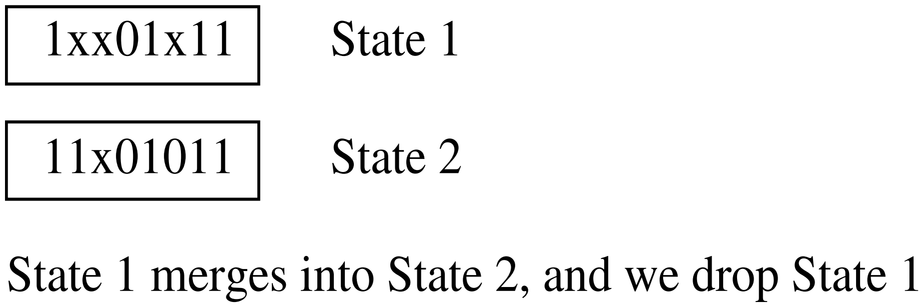

Figure 5.1: Merging of desired states.

Table 5.2 lists the target states obtained for each of the circuits. It is worth mentioning that some of the states obtained may correspond to redundant faults as the ATPG aborted the search of a sequence for detecting the target fault after a given number of backtracks.

circuit	# of target states
s1423	135
s3271	45
s3384	102
s5378	524
s6669	32
scfRjisdre	267
s832jcsrre	57
s510Rjcsrre	114
s510Rjosrre	114

Table 5.2: The number of target states obtained.

5.3 Application of GA to state justification of the desired states

The proposed GA-based state justification algorithm was implemented inside a fault simulator HOPE [38]. This involved an additional 2000 lines of code in C. The experiments were run on SUN ULTRA 10 stations. The target states were provided as an input to the algorithm. The processing times were computed and the output was a list of reached states and the state justification sequence derived. The state justification algorithm used in IGATE [8] [9] was also implemented inside HOPE for comparison purposes. Time taken and the list of states reached by the algorithm were reported. The following sets of experiments were carried out:

1. Comparing three replacement policies for the GA.

2. Sensitivity analysis of the different parameters used.

5.3.1 Comparison of replacement strategies

The initial population is randomly generated. Fitness is computed for each chromosome and parents are selected based on the Roulette-wheel selection scheme. Crossover and mutation operators as discussed in Section 4.5.4, are applied. More fit children replace the members of the previous generation [26]. Three replacement policies were experimented with. A discussion of these is given in Section 4.5.6.

The $(n+1)$ replacement strategy, replaces one chromosome of the previous population after every generation. In Random Elitist strategy, two chromosomes are created in every crossover and $N/2$ crossovers are performed in every generation, where N is the number of chromosomes in the population. The best half of these are inserted in the new population. Half of the members of the new population are selected randomly in Random Elitist strategy, while they are selected based on a roulette wheel mechanism in Roulette Elitist strategy. The results of the simulations carried out using these three replacement policies are shown in Table 5.3.

circuit	CHR	GEN	BT	NLimit	TLS	$(n+1)$		Random Elitist		Roulette Elitist	
						SR	Time	SR	Time	SR	Time
s1423	16	100	10	120	150	58	126	19	508	32	748
	32	100	10	120	150	64	365	31	778	49	3586
	64	100	10	120	150	64	572	49	11300	68	13704
s3271	16	800	20	225	150	20	4592	11	5023	15	11214
	32	100	20	225	150	21	6244	18	11805	20	18113
	256	100	20	225	150	21	10625	19	12976	21	109612
s3384	16	800	10	375	150	65	11849	23	14912	34	17445
	64	800	10	375	150	66	23115	51	24905	41	30023
	256	800	10	375	150	66	41225	65	68428	50	100615
s5378	16	400	10	275	150	64	25294	22	84225	45	112610
	32	400	10	275	150	113	29274	53	100324	61	141251
	64	400	10	275	150	115	34893	55	117520	61	161225
s6669	16	10	10	375	150	19	130	19	871	22	914
	16	100	10	375	150	27	503	19	5151	22	8681
	16	400	10	375	150	30	1664	22	17905	22	24668
scfRjisdre	16	100	10	40	150	18	25	17	285	26	836
	64	100	10	40	150	19	42	34	832	43	6700
	256	100	10	40	150	20	114	46	5055	50	48820
s832jcsrre	16	400	100	100	150	7	79	6	77	6	82
	256	400	100	100	150	7	190	7	1946	7	2126
	1024	400	100	100	150	9	360	8	3441	9	4956
s510Rjcsrre	16	400	10	45	150	12	14	8	120	6	140
	256	400	10	45	150	16	132	23	523	23	1208
	512	400	10	45	150	23	260	31	2340	31	5038
s510Rjosrre	16	800	10	45	150	12	92	5	233	4	305
	64	800	10	45	150	19	661	13	1171	11	2841
	256	800	10	45	150	19	2740	17	9870	19	19342

Table 5.3: Comparison of the selection schemes.

In Table 5.3, the number of states reached (SR) and the time taken to reach those states are given for each replacement strategy described above. It was observed that $(n+1)$ replacement strategy was the best in terms of execution time. It also reached a comparable number of states for most of the circuits. This strategy changes only one member of the previous generation and hence the number of operations in one generation of $(n+1)$ replacement strategy requires less time as compared to other strategies. Moreover, changes in the characteristics of the population do not occur as abruptly as in the other two schemes. Figure 5.2 shows the average and best fitness of the population against the number of generations for one of the target states that is justified by the algorithm using the $(n+1)$ replacement strategy. It can be seen that the average fitness increases monotonically with the number of generations. This is due to the fact that we are always preserving the best chromosome in each generation.

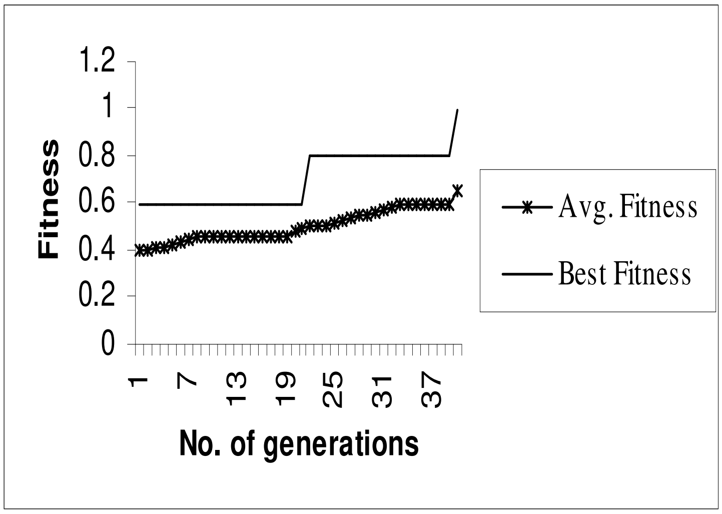

Figure 5.2: Average and best fitness plotted against the number of generations for $(n+1)$ replacement strategy.

In Figure 5.3, we show the state traversal for one of the states that has been reached by the algorithm. It can be seen that we progress towards better states in terms of the Hamming distance as the algorithm runs for more iterations. Less fit states are reached if we we are unable to reach a better state because of the Tabu restriction. Moreover, we move towards the best state among all alternatives, even if that state is worse than the current state. This helps in avoiding the local minima. The example is for one of the target states of s1423 circuit.

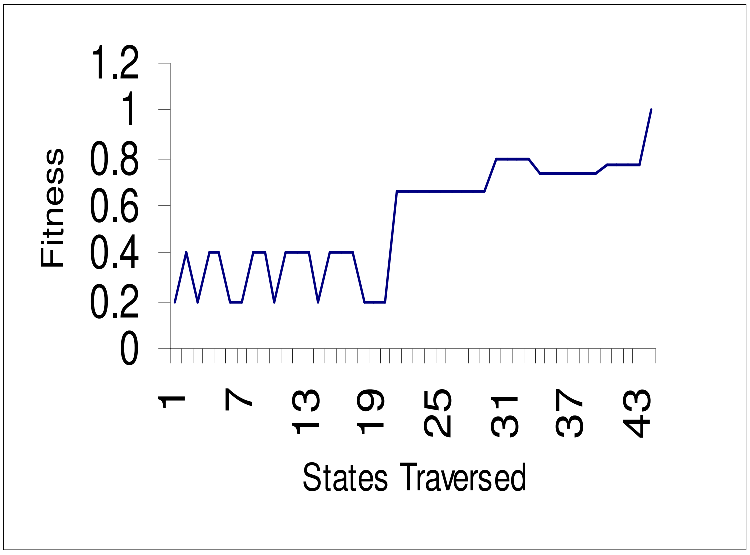

Figure 5.3: State traversed versus the fitness of reached states for a target state of s1423 circuit.

In Figure 5.4, state traversal for one of the unreached states is shown. It can be observed from the figure, that the quality of states reached is better in terms of Hamming distance as the algorithm runs for more iterations.

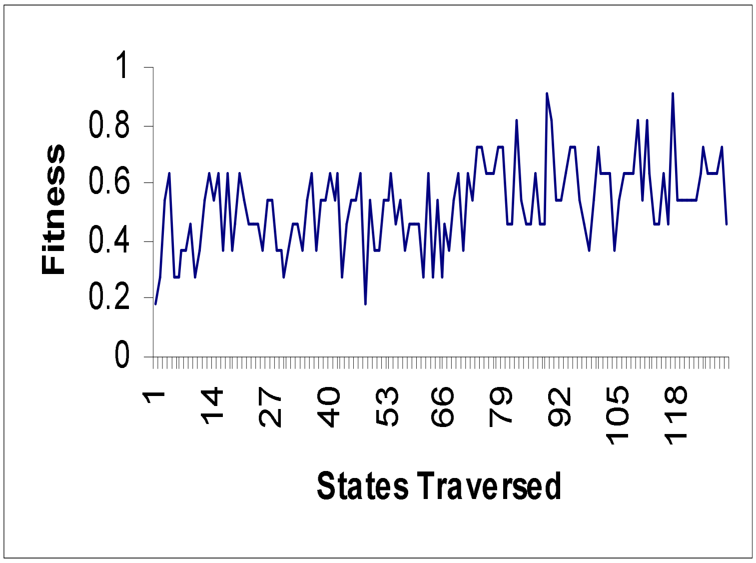

Figure 5.4: State traversed versus the fitness of reached states for one of the unreached state of s1423 circuit.

The Random Elitist strategy was comparable to the $(n+1)$ replacement strategy in terms of the quality of solution. However, as more operations were required in one generation of this strategy, it was expensive in terms of CPU execution time. Sorting had to be performed on the children produced after crossover, to select the best half of them. This took additional execution time as compared to the $(n+1)$ replacement strategy.

The Roulette Elitist strategy gave better results in terms of the number of justified target states than the Random Elitist strategy for most of the circuits. Additional operations like applying the roulette-wheel to select half of the candidates, made it worse than the Random Elitist and $(n+1)$ replacement strategies in terms of the execution time. However, it was comparable to the $(n+1)$ replacement scheme as far as the number of reached states is concerned.

It can be observed from Table 5.3 that $(n+1)$ selection strategy was far better than the other two strategies in terms of execution time. It gave comparable results for the number of reached states for most of the circuits. In the remainder of this thesis, the $(n+1)$ replacement strategy is employed.

5.3.2 Sensitivity analysis of parameters

It has been observed that Genetic Algorithms are quite sensitive to variations in the control parameters [27]. The parameters that affect the proposed algorithm include size of the population, the number of generations, size of the Tabu list, value of the Nlimit parameter and the number of backtracks. A sensitivity analysis of these parameters was performed and we discuss it in this section. In addition, the type of encoding used (binary or integer), crossovers used, selection and replaceement policies and rates of crossover and mutation affect the quality of solution. As each character of the chromosome was mapped to a primary input, a binary encoding was used. As observed in Section 5.3.1, the $(n+1)$ replacement strategy gave better results in terms of execution time while producing solutions of comparable quality. Hence, it was decided to use this replacement strategy in the simulations. The crossover rate was kept at 1 and the roulette-wheel selection scheme as described in Section 3.2.2 was used for parent selection. One point uniform crossover as mentioned in Section 3.2.2 was used in this implementation because of its low complexity. Bit-flipping was used in mutation, and the mutation rate was kept at 0.01.

Effect of population size

The effect of population size on the quality of solution was viewed in this experiment. The number of chromosomes was kept constant in each generation. As the population size increased, the time taken by the algorithm for each generation increased. Tabu List size was kept at 150 and the Nlimit was 1.5 times the number of flip-flops for each circuit. The values of other parameters, were the best values obtained for each circuit. The results are given in Table 5.4.

It can be seen from the table that there was a general trend of improvement in the number of reached states as the population size increased. However, the total run time increased appreciably with the increase in the population size. As can be seen from the table, a population size of 32 seems sufficient considering both the reached states and required CPU time.

Effect of the number of generations

In this experiment, the effect on the quality of solution was observed for the number of generations the algorithm was run while moving from a state to a state. Because of the replacement policy used, each generation replaced the worst chromosome from

circuit	Chromes	Gen	BT	NLimit	TLS	Reached	Time(sec)
s1423	16	100	10	120	150	58	126
	32	100	10	120	150	64	365
	64	100	10	120	150	64	572
s3271	16	100	10	225	150	19	831
	32	100	10	225	150	21	1244
	256	100	10	225	150	21	2625
s3384	16	800	10	375	150	65	11849
	32	800	10	375	150	66	18122
	64	800	10	375	150	66	23115
	256	800	10	375	150	66	41225
s5378	16	400	10	275	150	64	25294
	32	400	10	275	150	113	29274
	64	400	10	275	150	115	34893
s6669	16	100	10	375	150	22	503
	32	100	10	375	150	23	788
	64	100	10	375	150	23	1349
scfRjisdre	16	100	10	40	150	18	25
	32	100	10	40	150	19	33
	64	100	10	40	150	19	42
	256	100	10	40	150	20	114
s832jcsrre	16	400	100	100	150	7	79
	32	400	100	100	150	7	118
	256	400	100	100	150	7	190
	1024	400	100	100	150	9	360
s510Rjcsrre	16	400	10	45	150	12	14
	32	400	10	45	150	12	77
	256	400	10	45	150	16	132
	512	400	10	45	150	23	260
s510Rjosrre	16	800	10	45	150	12	92
	32	800	10	45	150	16	334
	64	800	10	45	150	19	661
	256	800	10	45	150	19	2740

Table 5.4: Effect of the change in population size.

the last generation. Hence, the average fitness monotonically increased with every generation. The population size which gave the best results was used for every circuit. The results are shown in Table 5.5.

It can be seen from the table that there was a general trend of improvement in the number of reached states with the number of generations. The increase in execution time was not as much as in the case of increase in population size.

circuit	Chromes	Gen	BT	NLimit	TLS	Reached	Time(sec)
s1423	32	10	10	120	150	46	91
	32	100	10	120	150	64	365
	32	200	10	120	150	68	1457
	32	400	10	120	150	71	2912
	32	800	10	120	150	71	6303
s3271	16	10	10	225	150	12	105
	16	100	10	225	150	19	831
	16	400	10	225	150	21	2455
	16	800	10	225	150	21	4592
s3384	16	10	10	375	150	29	1104
	16	100	10	375	150	45	2687
	16	200	10	375	150	45	4744
	16	400	10	375	150	49	7794
	16	800	10	375	150	65	11849
	16	1200	10	375	150	69	15887
s5378	32	10	10	275	150	51	3125
	32	400	10	275	150	113	29274
	32	800	10	275	150	115	49257
s6669	16	10	10	375	150	19	130
	16	100	10	375	150	22	503
	16	400	10	375	150	30	1664
scfRjisdre	16	100	10	40	150	18	25
	16	400	10	40	150	28	108
	16	800	10	40	150	42	183
	16	1400	10	40	150	45	233
s832jcsrre	16	10	10	45	150	5	3
	16	100	10	45	150	6	11
	16	400	10	45	150	6	49
	16	800	10	45	150	7	73
s510Rjcsrre	256	100	10	45	150	14	30
	256	400	10	45	150	16	132
	256	800	10	45	150	20	210
	256	1400	10	45	150	20	1063
s510Rjosrre	32	10	10	45	150	3	5
	32	100	10	45	150	7	81
	32	400	10	45	150	12	167
	32	800	10	45	150	16	334
	32	1200	10	45	150	16	611

Table 5.5: Effect of the change in the number of generations.

Effect of change in Nlimit parameter

The algorithm traverses *atleast* Nlimit number of states before it gives up the search for the desired state. If the fitness of the currently visited state is less than the average fitness of the last Nlimit states, the algorithm stops further searching of the desired state. Experiments were carried out to tune this parameter and the results are given in Table 5.6. The best parameters obtained from Table 5.4 and

Table 5.5 for population size and number of generations respectively, were used for each circuit.

It can be observed from the table, that there was a general trend of improvement in the number of reached states with the increase in Nlimit. The algorithm searches more intensely for an objective state when the Nlimit is increased. However, increasing this parameter brings about an increase in the execution time. It was observed that an Nlimit value which is 1.5 times the number of flip-flops gave good state coverage in reasonable time.

circuit	Chromes	Gen	BT	NLimit	TLS	Reached	Time(sec)
s1423	32	400	10	120	150	71	2912
	32	400	10	500	150	74	9447
s3271	16	800	10	225	150	21	4592
	16	800	10	500	150	21	10654
	16	800	10	800	150	23	22096
s3384	16	800	10	50	150	53	2289
	16	800	10	375	150	65	11849
	16	800	10	500	150	67	16219
s5378	32	400	10	275	150	113	29274
	32	400	10	600	150	115	57135
s6669	16	400	10	50	150	19	130
	16	400	10	375	150	30	1664
	16	400	10	500	150	30	3716
scfRjisdre	16	800	10	40	150	42	183
	16	800	10	100	150	43	533
s832jcsrre	16	400	10	45	150	6	49
	16	400	10	100	150	6	73
	16	400	10	500	150	7	664
s510Rjcsrre	256	400	10	45	150	16	132
	256	400	10	100	150	16	410
	256	400	10	200	150	20	1063
s510Rjosrre	32	800	10	45	150	16	334
	32	800	10	100	150	16	1243
	32	800	10	500	150	19	3717

Table 5.6: Effect of the change in Nlimit.

Effect of change in Tabu List size

To prevent the algorithm from visiting recently visited states, a Tabu list containing the last visited states was formed. On reaching a list, the algorithm looks into the Tabu list. If the state reached is present, the algorithm goes to the state reached by the next fit vector. A large value of Tabu list size, restricts us from undoing a bad move which had been taken previously. Several experiments were carried out to find a reasonable value. It was observed that a Tabu List size of 15, gave the best results. The best values obtained for population size and the number of generations from Table 5.4 and Table 5.5 respectively, were used for all the circuits. The Nlimit was kept at 1.5 times the number of flip-flops. The results are shown in Table 5.7.

circuit	Chromes	Gen	BT	NLimit	TLS	Reached	Time(sec)
s1423	32	400	10	120	150	71	2912
	32	400	10	120	50	71	2966
	32	400	10	120	15	74	3119
s3271	16	800	10	225	150	21	4592
	16	800	10	225	50	21	4627
	16	800	10	225	15	21	4894
s3384	16	800	10	375	150	65	11849
	16	800	10	375	50	65	11703
	16	800	10	375	15	67	16121
s5378	32	400	10	275	150	113	29274
	32	400	10	275	50	113	31127
	32	400	10	275	15	115	31281
s6669	16	400	10	375	150	27	503
	16	400	10	375	50	27	715
	16	400	10	375	15	30	1466
scfRjisdre	16	800	10	40	150	42	183
	16	800	10	40	50	42	355
	16	800	10	40	15	46	533
s832jcsrre	16	400	10	45	150	6	49
	16	400	10	45	50	6	77
	16	400	10	45	15	7	106
s510Rjcsrre	256	400	10	100	150	16	410
	256	400	10	100	50	16	492
	256	400	10	100	15	19	663
s510Rjosrre	32	800	10	45	150	16	334
	32	800	10	45	50	16	416
	32	800	10	45	15	18	441

Table 5.7: Effect of the change in Tabu List size.

Effect of change in the number of Backtracks

When a state is reached after which no further traversal is possible, we move back to a previously visited state. This is termed as *backtracking*. Increase in the backtrack limit causes an increase in execution time. The increase in the number of visited states was not very significant except for some cases. The best values obtained for population size and the number of generations from Table 5.4 and Table 5.5 respectively, were used for all the circuits. The Nlimit was kept at 1.5 times the number of flip-flops. The size of the Tabu List was set to 15. The results are shown in Table 5.8.

It was observed that a backtrack limit of 10 gave good results in reasonable time.

5.3.3 Recommended parameters for the proposed approach

Based on the experiments performed and the observed results, certain parameters are found to perform better than others for most of the circuits. A list of recommended parameters is given in Table 5.9.

A population size of 32 and 400 generations gave good quality results in reasonable time. A backtrack limit of 10 and Tabu List size of 15 are suggested. Nlimit is suggested as 1.5 times the number of flip-flops in the circuit. The algorithm was run with the parameters suggested above for all circuits. Results are shown in Ta-

circuit	Chromes	Gen	BT	NLimit	TLS	Reached	Time(sec)
s1423	32	400	200	120	15	75	4217
	32	400	100	120	15	75	4212
	32	400	10	120	15	74	3119
s3271	16	800	200	225	15	21	5042
	16	800	100	225	15	21	4992
	16	800	10	225	15	21	4894
s3384	16	800	200	375	15	68	17714
	16	800	100	375	15	68	17703
	16	800	10	375	15	67	16121
s5378	32	400	200	275	15	115	31311
	32	400	100	275	15	115	31302
	32	400	10	275	15	115	31281
s6669	16	400	200	375	15	30	1723
	16	400	100	375	15	30	1715
	16	400	10	375	15	30	1466
scfRjisdre	16	800	200	40	15	48	735
	16	800	100	40	15	48	735
	16	800	10	40	15	46	533
s832jcsrre	16	400	200	45	15	7	189
	16	400	100	45	15	7	187
	16	400	10	45	15	7	106
s510Rjcsrre	256	400	200	100	15	21	951
	256	400	100	100	15	21	949
	256	400	10	100	15	19	663
s510Rjosrre	32	800	200	45	15	18	460
	32	800	100	45	15	18	462
	32	800	10	45	15	18	441

Table 5.8: Effect of the change in Backtrack Limit.

Population size	32
No. of Generations	400
Nlimit	1.5 times # of FF
TLS	15
Backtrack limit	10

Table 5.9: Recommended parameters for the proposed approach.

ble 5.10. In Table 5.11, we present the best results obtained for each of the circuits. The parameters are listed for every circuit. It can be seen from Table 5.10 and Table 5.11, that the parameters proposed in Table 5.9 gave good quality results when compared with the best results obtained for each circuit.

circuit	Reached	Time(sec)
s1423	74	3119
s3271	21	6015
s3384	67	18314
s5378	115	31281
s6669	30	1764
scfRjisdre	48	803
s832jcsrre	8	139
s510Rjcsrre	16	163
s510Rjosrre	16	181

Table 5.10: Results obtained from the suggested parameters.

circuit	Chromes	Gen	BT	NLimit	TLS	Reached	Time(sec)
s1423	32	400	100	120	15	75	4212
s3271	16	400	10	225	150	21	2455
s3384	16	800	100	375	15	68	17703
s5378	32	400	10	275	15	115	31281
s6669	16	400	10	375	15	30	1466
scfRjisdre	16	800	100	40	15	48	735
s832jcsrre	1024	400	100	100	150	9	360
s510Rjcsrre	512	400	10	45	150	23	260
s510Rjosrre	64	800	10	45	150	19	661

Table 5.11: Best results obtained for each circuit.

5.4 Comparison with previous approaches

GA has been used previously for state justification [8] [9]. It has been applied to a sequence of vectors as opposed to individual vectors in the proposed approach. The fitness of each individual was a measure of how closely the final state reached by the sequence matched the desired state. If any sequence was found which produced the desired state, the sequence was added to the test set. The GA was run for a specific number of generations if the sequence failed in reaching the desired state.

5.4.1 Limitations of the technique

There were several drawbacks in the technique proposed by Patel et. al. in [8][9].

Fitness Function

The fitness function used, matched only the last state reached by the sequence with the desired state. It did not take into account any more closely matched state which it reached while simulating the vectors in the sequence. The fitness was hence a measure of only how closely the last state reached by the sequence matched the desired state.

Fixed number of vectors

The number of vectors in a sequence was fixed at 4 times the structural sequential depth of the circuit. Hence, there were a fixed number of vectors to simulate for reaching a desired state. In case of feed-back loops, the structural sequential depth may not give a correct estimate of the number of vectors required for justifying a given state. Thus, if a state requires longer justification sequence, it will not be justified by the algorithm.

5.4.2 Parameters used in comparison

The parameters used were the same as those mentioned in [8][9]. The number of vectors in each sequence was kept equal to 4 times the structural sequential depth of the circuit. 32 chromosomes were used and the algorithm was run for 8 generations. Tournament selection and two-point uniform crossover were used. The crossover and mutation probabilities of one and 1/64 respectively, were used in the simulations.

For our proposed approach, we found better results with a population size of 32, 400 generations, an $(n+1)$ replacement strategy, and one point crossover. The parameters used were as given in Table 5.9. Results are demonstrated in Table 5.12

			our approach		approach in [9]			approach in [9]		
Name	# of FF	Target states	states reached	time(sec)	gens	states reached	time(sec)	gens	states reached	time(sec)
s1423	74	135	74	3119	8	50	2743	50	61	3953
s3271	116	45	21	6015	8	15	1664	200	18	6319
s3384	183	102	67	18314	8	31	3794	250	45	21161
s5378	179	524	115	31281	8	45	3133	100	48	225160
s6669	239	32	30	1764	8	23	1701	50	24	2289
scfRjisdre	20	267	48	803	8	25	501	100	31	5196
s832jcsrre	31	57	8	139	8	7	120	100	7	2170
s510Rjcsrre	30	114	16	163	8	12	61	100	13	504
s510Rjosrre	32	114	16	181	8	9	62	100	13	583

Table 5.12: Comparison of the two techniques.

The first column in the table shows the circuit name. In the second and third columns, the number of flip-flops (FFs) and the number of target states respectively is given for each circuit. The states reached and CPU time obtained by our algorithm are mentioned in the next two columns. For comparison purposes, we ran the algorithm proposed in [9], for several number of generations and the results are shown in the next columns.

It can be observed from the results, that the number of desired states reached by our technique are more than those reached by the technique used in IGATE [9] for all the circuits. Furthermore, our proposed technique reached a higher number of states than IGATE [9] in all the cases even when the latter was run for greater amount of CPU time.

5.4.3 Fault coverage comparisons

In order to verify the effectiveness of the generated state justification sequences in detecting hard-to-detect faults, we seeded them to a deterministic test pattern generator HITEC [18]. HITEC makes use of previously visited states while doing state justification. The faults detected by an initial run of HITEC with 1000000 backtracks were removed from the fault list. The results are shown in Table 5.13.

		faults detected			reached states	
Name	TF	approach in [9]	our approach	TS	approach in [9]	our approach
s1423	926	312	578	135	61	74
s3271	61	34	41	45	18	21
s3384	376	91	116	102	45	67
s5378	1221	103	285	524	48	115
s6669	40	29	31	32	24	30
scfRjisdre	1920	1397	1802	267	31	48
s832jcsrre	293	38	147	57	7	8
s510Rjcsrre	374	45	85	114	13	16
s510Rjosrre	459	232	431	114	13	16

Table 5.13: Faults detected by the two state-justification techniques.

It can be observed that a large number of hard-to-detect faults are detected when we seed HITEC with the state justification sequence obtained by the proposed strategy. The number of faults detected are significantly higher than the faults detected when the ATPG is seeded with the state justification sequences generated by the technique used in IGATE [9]. Apart from justifying more states, our technique takes advantage of the partial justification sequences generated.

5.5 Conclusion

In this chapter, we presented and discussed various experiments that were performed using our GA-based algorithm. Various replacement policies were experimented with and results obtained have been compared. It was observed that $(n+1)$ replacement strategy gave the best results in terms of the number of states reached and execution time. Sensitivity analysis of different parameters like number of chromosomes in the population, number of generations, etc has also been performed. An Nlimit parameter has been introduced and its effect on the quality of solution has been analyzed. The effect of Tabu List size has also been studied. It was observed that a population size of 32, 400 generations and backtrack limit of 10 gave good results. An Nlimit value of 1.5 times the number of flip-flops and a Tabu List Size of 15 gave good results in reasonable time. The performance of the proposed algorithm has been compared with another GA-based state justification algorithm reported in the literature. It was observed that the number of desired states reached by our proposed technique is more than the previous approach for all the circuits. Furthermore, our proposed technique reached a higher number of states for all the circuits considered, even when the previous algorithm was run for greater amount of CPU time. The state justification sequence was seeded into a deterministic test pattern generator and significant improvements in the number of faults detected were obtained over previous approaches.

Chapter 6

Conclusion

6.1 Summary

Once a digital circuit is designed and fabricated, it needs to be tested for the presence or absence of physical defects or faults. Generating test patterns for testing digital circuits consumes a significant portion of the design time. Automatic test pattern generation (ATPG) deals with this problem automatically for a given circuit description. The process of test pattern generation for digital logic involves a search through all possible input values or sets of input values to find one that causes the output of a good circuit to differ from that of a faulty circuit. The problem is far more complex for sequential circuits as compared to combinational circuits. A common search operation in sequential ATPG is to justify a desired state assignment on the sequential elements. State justification using deterministic algorithms is a difficult problem and is prone to many backtracks, which can lead to high execution times. Significant speedups can be obtained with the simulation-based approaches. Untestable faults however, cannot be identified using these approaches and deterministic algorithms are needed. In this work, we propose a hybrid approach which uses a combination of evolutionary and deterministic algorithms for state justification. Genetic Algorithms (GAs) were used for generating sequences that will help the Automatic Test Pattern Generator (ATPG) in detecting more faults by reaching specific states. The main advantage of GA-based ATPGs as compared to other approaches, is their ability to cover a larger search space in lower CPU time. This improves the fault coverage and makes these ATPGs capable of dealing with larger circuits. The approach used was compared with other GA-based approaches and significant improvements were observed.

The contributions of this work can be itemized as follows:

- A hybrid ATPG approach for sequential circuits, where, both deterministic and GA-based state justification are involved.
- A novel state justification procedure based on GA.
- Genetic engineering of a sequence vector by vector. This has the advantage of dynamically determining the length of the justification sequence. Furthermore, this has the benefit of taking the fitness of intermediate states into account.
- A comparison of three replacement strategies. The $(n+1)$ replacement strategy gave better results.

- The use of a Tabu list to prevent the algorithm from visiting previously visited states. Having a reasonably small Tabu list size allows the algorithm to get out of bad moves.

- Sensitivity analysis of the parameters was carried out and a list of parameters is recommended.

- A comparison with other approaches based on engineering sequences. Better results obtained in terms of the number of states reached and the CPU time. Higher fault coverage achieved than the previous techniques.

Chapter 1 briefly describes the Sequential ATPG problem. Various approaches presented in the literature for solving the problem have been presented in Chapter 2. In Chapter 3, Genetic Algorithms and their application to the sequential ATPG problem have been reviewed. We have described the proposed approach of using Genetic Algorithms for state justification in Chapter 4. An overview of Tabu Search is also given in the same chapter. In Chapter 5, experimental results have been presented. Sensitivity analysis of different parameters like number of chromosomes in the population, number of generations, etc has been performed. An Nlimit parameter has been introduced and its effect on the quality of solution has been analyzed. The effect of Tabu List size has also been studied. The performance of the proposed algorithm has been compared with another GA-based algorithm reported in the literature.

6.2 Future Research

Our work can be extended to address the following issues:

- Designing other meta-heuristics (for e.g., Tabu Search) for state justification and their comparison with the proposed scheme.
- Parallelization of the algorithm, especially the evaluation of the fitness function.

Bibliography

[1] Aiman El-Maleh, M. Kassab, and J. Rajski. A Fast simulation-based learning technique for real sequential circuits. In *Design Automation Conference*, 1998.

[2] F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. A Parallel Genetic Algorithm for aoutomatic generation of test sequences for digital circuits. In *International Conf. on High Performance Computing and Networking, Belgium*, April 1996.

[3] Y. C. Kim and K. K. Saluja. Sequential test generators: past, present and future. *INTEGRATION, the VLSI journal*, 26:41–54, 1998.

[4] M. H. Konijnenburg, J. T. van der Linden, and A. J. van de Goor. Sequential test generation with advanced illegal state search. In *International Test Conference*, 1997.

[5] F. Corno, M. Rebaudengo, and Sonza Reorda. Experiences in the use of evo-

lutionary techniques for testing digital circuits. In *Application and Science of Neural Networks, Fuzzy Systems and Evolutionary computation, SPIE*, 1998.

[6] Srinivas M. and L. M. Patnaik. Genetic Algorithms : A Survey. *IEEE Computer Magazine*, pages 17–26, June 1994.

[7] E. M. Rudnick, J. G. Holm, D. G. Saab, and J. H. Patel. Application of Simple Genetic Algorithm to sequential circuit test generation. In *European Design and Test Conference*, pages 40–45, February 1994.

[8] Elizabeth M. Rudnick and Janak H. Patel. State justification using Genetic Algorithms in sequential circuit test generation. *A survey report from CRHC Univ. of Illinois, Urbana*, January 1996.

[9] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Application of genetically engineered finite-state-machine sequences to sequential circuit ATPG. *IEEE Transactions on CAD of Integrated circuits and systems*, 17:239–254, March 1998.

[10] X. Chen and M. L. Bushnell. Sequential circuit test generation using dynamic justification equivalence. *Journal of Electronic Testing*, 8:9–33, 1996.

[11] Hideo Fujiwara. *Logic Testing and Design for Testability.* The MIT Press, 1991.

[12] M. A. Iyer, D. E. Long, and M. Abramovici. Identifying sequential redundancies without search. In *33rd Design Automation Conference*, pages 457–462, 1996.

[13] T. E. Marchok, Aiman El-Maleh, W. Maly, and J. Rajski. A complexity analysis of sequential ATPG. *IEEE Transactions on CAD of Integrated circuits and systems*, 15:1409–1423, November 1998.

[14] P. H. Ibarra and S. K. Sahni. Polynomially complete fault detection problems. *IEEE Transactions on Computing*, 24:242–249, 1975.

[15] M. S. Hsiao. Use of Genetic Algorithms for testing sequential circuits. *Ph.D. Dissertation, UIUC*, December 1997.

[16] Kwang-Ting Cheng and A. Krstic. Current directions in automatic test-pattern generation. *IEEE Computer Magazine*, pages 58–64, November 1999.

[17] T. Marchok, Aiman El-Maleh, W. Maly, and J. Rajski. Complexity of Sequential ATPG. In *European Design and Test Conference*, pages 252–261, 1995.

[18] T. Niermann and J. H. Patel. HITEC: a test generation package for sequential circuits. In *European Test Conf.*, pages 214–218, February 1991.

[19] T. Niermann, W. T. Cheng, and J. H. Patel. PROOFS: A fast memory efficient sequential circuit fault simulator. In *Design Automation Conf.*, pages 535–540, June 1990.

[20] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital System Testing and Testable Design*. Computer Science Press, 1990.

[21] H. K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentalli. Test generation for sequential circuits. *IEEE Transactions on CAD*, 7:1081–1093, 1988.

[22] V. D. Agrawal, K. Cheng, and P. Agrawal. A directed search method for test generation using a concurrent simulator. *IEEE Transactions on Computer Aided Design*, 10(5):652–667, 1989.

[23] D. G. Saab, Y. G. Saab, and J. A. Abraham. CRIS: A test cultivation program for sequential VLSI circuits. In *International Conf. on Computer-aided Design*, pages 216–219, November 1992.

[24] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann. Sequential circuit test generation in a genetic algorithm framework. In *Design Automation Conference*, pages 698–704, June 1994.

[25] F. Corno, P. Prinetto, M. Rebaudengo, and Sonza Reorda. GATTO: A genetic algorithm for automatic test pattern generation for large synchronous sequential circuits. *IEEE Transactions on CAD of Integrated circuits and systems*, 15:991–1000, August 1996.

[26] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[27] Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms with appli-*

cations in Engineering: Solving combinatorial optimization problems. IEEE Computer Society, 1999.

[28] William M. Spears and Vic Anand. A study of crossover operators in genetic programming. In *Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems ISMIS 91*, pages 409–418, 1991.

[29] F. Corno, M. Rebaudengo, Sonza Reorda, and M. Violante. Exploiting logic simulation to improve simulation-based sequential ATPG. In *Sixth IEEE Asian Test Symposium, Arta, Japan*, November 1997.

[30] D. G. Saab, Y. G. Saab, and J. A. Abraham. Automatic test vector cultivation for sequential VLSI circuits using genetic algorithms. *IEEE Transactions on CAD*, 15:1278–1285, October 1996.

[31] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann. A Genetic algorithm framework for test generation. *IEEE Transactions on CAD of Integrated circuits and systems*, 16:1034–1044, September 1997.

[32] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Alternating strategies for sequential circuit ATPG. In *European Design and Test Conference*, pages 368–374, March 1996.

[33] D. Krishnaswamy, M. S. Hsiao, V. Saxena, E. M. Rudnick, and J. H. Patel. Par-

allel Genetic Algorithms for simulation-based sequential circuit test generation. In *IEEE VLSI Design Conference*, pages 475–481, 1997.

[34] F. Corno, P. Prinetto, M. Rebaudengo, Sonza Reorda, and E. Veiluva. A PVM tool for automatic test generation on parallel and distributed systems. In *International Conf. on High Performance Computing and Networking, Italy*, pages 39–44, 1995.

[35] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Sequential circuit test generation using dynamic state traversal. In *European Design and Test Conference*, pages 22–28, March 1997.

[36] M. S. Hsiao, E. M. Rudnick, and J. H. Patel. Dynamic state traversal for sequential circuit test generation. *ACM Transactions on Design Automation of Electronic Systems*, 5, July 2000.

[37] Ahmed A. Al-Yamani. A Parallel Tabu Search Algorithm for VLSI Standard Cell Placement. *MS Thesis, KFUPM*, April 1999.

[38] H. K. Lee and D. S. Ha. HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15:1048–1058, September 1996.

Vitae

- Syed Zafar Shazli
- Born in Karachi, Pakistan.
- Received Bachelor of Science (Hons.) degree from University of Karachi, Karachi, Pakistan in February 1995.
- Received Masters in Computer Science degree from University of Karachi, Karachi, Pakistan in February 1997.
- Joined Computer Engineering Department, KFUPM, as a research assistant in September 1998.
- Received Master of Science (M.S.) degree in Computer Engineering from KFUPM, Dhaharan, Saudi Arabia, in June 2001.