Analysis of Evolutionary Optimization Algorithms in Automated Test Pattern Generation for Sequential Circuits

By:

Majed Mohammad Alateeq

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Engineering

Department of Electrical and Computer Engineering

University of Alberta

Abstract

In automated test pattern generation (ATPG), test patterns are automatically generated and tested against all specific modeled faults, such as stuck-at fault, which is most commonly used in fault modeling. Testing of sequential circuits can be performed exhaustively, randomly or algorithmically. Exhaustive and random test pattern generators consume a high percentage of resources which make them impractical solutions, especially for large sequential circuits. Moreover, the testing time increases rapidly as the number of inputs, or the circuit's complexity increases, which means that these types of tests are ineffective and cannot be fully adapted. Since the test pattern generation is a search process completed over a large search space, algorithmic test pattern generation is a favorable option because of its ability to reduce the size of the search space, which leads to lowering the number of test patterns and reducing the testing time. The objective of this work is to present a complete analytical study of ATPG for sequential circuits using algorithmic test pattern generators. Three optimization algorithms, namely: genetic algorithm (GA), particle swarm optimization (PSO) and differential evolution (DE), were analytically studied for the purpose of generating optimized test sequence sets. Furthermore, this work investigated the broad use of evolutionary algorithms and swarm intelligence in automated test pattern generation to expand the analysis of the subject. The obtained experimental results demonstrated the improvement in terms of testing time, number of test vectors, and fault coverage compared with previous optimization-based test generators. In addition, the experiments highlight the weakness of each optimization algorithm in the test pattern generation (TPG) and offer some constructive methods of improvement. We present several recommendations and guidelines regarding the use of optimization algorithms as test pattern generators to improve the performance and increase their efficiency. Moreover, the recommendations will allow for faster convergence toward optimal solution sets when being implemented for similar applications.

Acknowledgement

Thank you “Allah” for all your blessings. Thank you for helping me to achieve one of my goals.

I would like to dedicate this thesis to my parents, my father “Mohammad” and my mother “Aljawhara”, who passed away before completing this work. May “Allah” grant them Jannah.

I would like to express my sincere appreciation and gratitude to my supervisor Professor: Witold Pedrycz, for his guidance during my research. His support and inspiring suggestions have been precious for the development of this thesis content. I am so fortunate to have Dr. Pedrycz as a supervisor during my study because of his high-level academic/research experience. I appreciate all his contributions of time, ideas and support. His endless positive support will be never forgotten.

I must express my gratitude to my wife for her support and encouragement. She has been extremely supportive throughout my study. Many thanks to her and to my kids for accepting being away from them while working on my thesis.

I would like to convey my gratitude to all my brothers and sisters for their best wishes and positive supports. No words can express my gratitude, but I pray that Allah will bless and reward them.

Table of Contents

List of Tables

List of Figures

List of Abbreviations and Acronyms

ATPG: Automated Test Pattern Generation.

BED: Binary Differential Evolution.

CPU: Central Processing Unit.

CUT: Circuit Under Test.

DE: Differential Evolution.

DDE: Discretized Differential Evolution.

EC: Evolutionary Computing.

EA: Evolutionary Algorithm.

GA: Genetic Algorithm.

IC: Integrated Circuit.

NP: Population Size in BDE and DDE.

PR: Permutation Rate in BDE and DDE.

PM: Mutation Rate in BDE and DDE.

PSO: Particle Swarm Optimization.

BPSO: Binary Particle Swarm Optimization.

Pbest: Personal Best. In PSO, it is the best solution found by the *ith* particle.

Gbest: Global Best. In PSO, it is the best value that is obtained by any particle in a population.

CR: user-defined crossover constant.

SAF: Stuck at Fault.

S-A-1: Stuck at One.

S-A-0: Stuck at Zero.

SI: Swarm Intelligence.

TPG: Test Pattern Generation.

PI: Primary Input.

PO: Primary Output

PPI: Pseudo-primary Input.

PPO: Pseudo-primary Output.

VLSI: Very Large Scale Integration.

Chapter 1: Introduction

As the development of integrated circuits is rapidly increasing, due to the growth of technology needs, testing and validating chip functionality has become more challenging. An integrated circuit (IC) can be tested by applying a series of test vectors that can detect any defect that might occur in the manufacturing process. The fully automated process of generating the test vectors is crucial to gain the minimum number of test vectors possible in the shortest time period. The problem of automated test generation belongs to the class of NP-complete problems and it is becoming progressively more difficult as the complexity of very large scale integration (VLSI) circuits increase. Recently, automated test pattern generation has given full fault coverage on almost all combinational circuits. However, none of the algorithmic test pattern generations can fully handle the real-world sequential circuits due to either the occurrences of untestable faults, or the complexity of the sequential circuit itself which requires more dedicated efforts to solve the problem.

Algorithmic testing approaches for sequential circuits can be categorized into three categories: deterministic algorithms, simulation-based algorithms, and hybrid test algorithms. In the simulation-based test generator, where processing occurs in the forward direction only, complex sequential circuits are easily tested. A simpler type of simulation-based automated test pattern generation (ATPG) is the random test generation, which has several drawbacks that limit its capability in terms of testing time and fault coverage. Other advanced simulation-based algorithms could significantly reduce central processor unit (CPU) time and improve the fault coverage. However, hard-to-detect faults remain a major problem in all the simulation-based algorithms. Deterministic algorithms and hybrid algorithms are necessary for most cases of hard-to-detect faults because of their advanced capabilities of fault testing. However, complexity and testing time of those algorithms are very high, which leads to increased overall testing costs.

Optimization algorithms such as genetic algorithms and particle swarm algorithms belong to simulation-based algorithms and are categorized as evolutionary optimization algorithms under guided random search methods, as shown in Figure 1.1. Optimization algorithms can generate and optimize efficient test vectors for combinational and sequential digital circuits. The basic concept behind the evolutionary optimization algorithms is as follows: they start with an initial population of individuals (strings of bits), each bit is mapped to a primary input and an

individual is evaluated with a fitness function. Better individuals will evolve through several generations until a stopping criteria is reached. Several genetic algorithm (GA)-based test generators have been presented in the field and the results are promising. Optimization algorithms have been used in combination with some other ATPG techniques to detect more faults and reduce CPU resources.

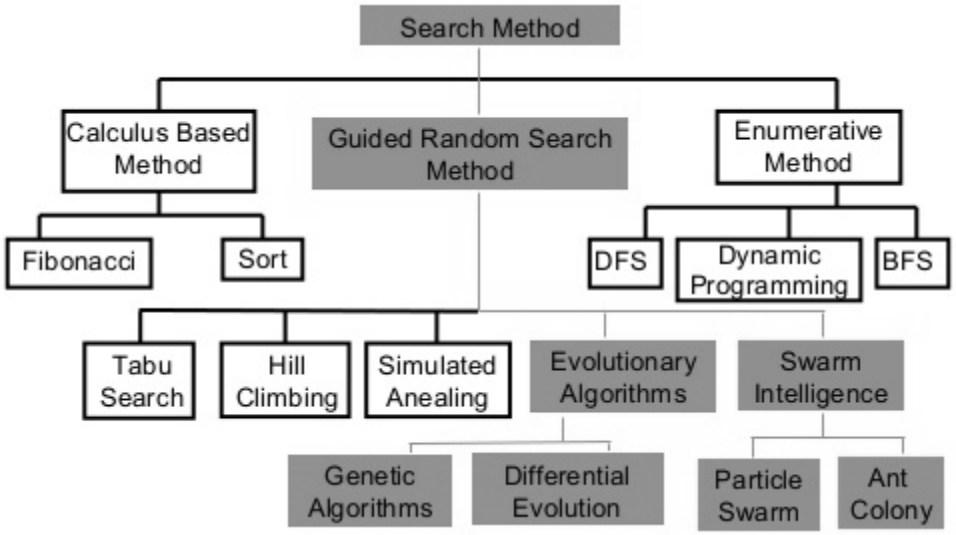

Figure 1.1: Classes of search methods [79].

Economically, many challenges in the manufacturing process had to be solved to achieve high performance systems. One of the steps in this process, namely testing, poses the most significant challenge to contemporary and future integrated circuit manufacturing. This is a continuing trend due to decreasing silicon cost and increasing complexity of integrated circuits, testing constitutes a significant portion of the IC manufacturing cost. (IC) testing for quality assurance is approaching 50% of the manufacturing costs for some complex circuits [1]. Testing might diminish final profits if it consumes long periods of time and does not produce high fault coverage. Low fault coverage will increase the rejection rate of chips, while long testing times will increase production cost. This relationship shows how significant the cost of VLSI testing is. Shortening testing time while increasing fault coverage will lead to higher circuit design quality, profit and a low rejection rate. As complexity of sequential circuits grows rapidly, the testing time must be short. This implies that, the testing time should not be affected by any other factors. Fault simulators are an essential part in the testing process. The primary role of fault simulators is to determine which faults are detected by a specific test vector in a circuit under test (CUT). They also determine the quality of each sequence being applied to a circuit. The response is

observed and evaluated by comparing it to an expected response of a known good circuit. A fault is said to be detected by a test vector if there is a difference between the output of a good circuit and the output of the faulty circuit. There are other several tasks realized by fault simulators such as:

- Good circuit simulation,
- Fault list generation,
- Circuit structure modification (Gate Injection),
- Faulty Circuit Simulation (Fault propagation and detection).

Fault simulation methods play a major role in reducing testing time and attain high testing efficiency. Flexibility, efficiency and versatility are three aspects that differentiate between different fault simulation methods. For example, the advantages of the concurrent fault simulator, which is one of the oldest fault simulation methods, lay in its flexibility and versatility and it can easily adopt several delay and functional models.

1.1 Motivation

Complexity of testing is proportional to complexity of sequential circuits, and they both follow Moore's law which states that the number of transistors on a chip increases at a rate of roughly one, and doubles every 18 to 24 months [2]. This is clearly explained in Figure 1.2, which shows the incremental increase of transistors on chips every year.

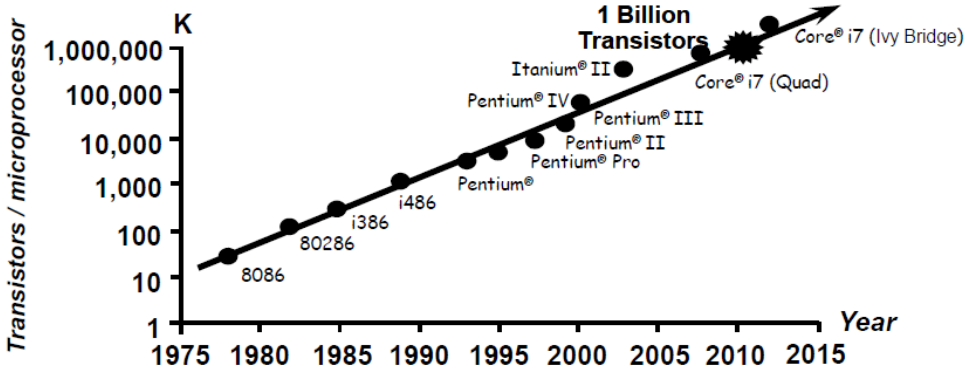

Figure 1.2: Moore's Law [65].

The incremental complexity of VLSI due to a gradual increase in the number of transistors per chip places great emphasis on the importance of finding close-to-perfection algorithms to test sequential circuits. Exhaustive testing for sequential circuits, which means using all possible binary combinations for testing, may take years as indicated in the following:

$$\text{Testing Time} = \frac{2^n + 2^m}{\text{Testing rate (test per second)}} = {}^{2^{n+m}}/_{\text{Testing rate (test per second)}},$$

where *n* is the number of inputs and *m* is the number of flip-flops.

As an example, the sequential circuit s35932 [50] has 35 primary inputs (PI's) and 1728 flip-flops. Therefore, the circuit needs 1.65×10^{518} years to be fully tested at a testing rate of one million test vectors per second. Consequently, testing of this circuit implies that the need for another technique one that can reduce the testing time to several seconds is extremely urgent.

Any algorithm must be able to detect faults at the manufacturing level at the earliest point possible. If the faulty components find their way into higher levels of integration, the cost of production will increase dramatically. It is cheaper to find and fix an IC than to find and fix a board in a system. Understanding the reasons for the costs associated with testing is another essential aspect to realize the necessity for a perfect algorithmic test generation strategy. It is in the chip manufacturer's best interest to minimize the number of bad devices shipped to the customer. A bad device is an IC that fails to meet one or more specifications at any point in the manufacturing process. Poorly designed tests, or parts that are not designed for testability, can result in bad devices appearing as good parts, or good devices failing tests and appearing as bad. The shipment of bad devices leads to incurred replacement costs, loss of reputation, and possible loss of market share. The other side of this problem is not much better. When good parts are represented as bad, it reduces the chip yield, and correspondingly, it reduces the earnings of the chip manufacturer [1]. Finally, effective and efficient testing for all types of IC's is a must and the testing must be performed in a very short time period otherwise the produced IC will be out of date and will lose its targeted market share.

1.2 Research Objectives

The overall objective when dealing with ATPG is to find the minimum number of test sequences that detect all testable faults in the shortest test time possible. There are several objectives for this specific research. These objectives are as follows:

- Analyze genetic algorithm (GA), differential evolution (DE), and particle swarm optimization (PSO) algorithms in ATPG for synchronous sequential circuits.
- Compare the GA, DE and PSO algorithms in terms of their performance and effectiveness.

- Determine the optimum parameter values for each of the optimization algorithms.
- Introduce improved versions of currently implemented algorithmic test pattern generators.
- Show the effectiveness of using optimization algorithms as stand-alone algorithms to solve the problem of ATPG for sequential circuits.
- Determine which optimization algorithm would perform better among all other optimization algorithms.

1.3 Research Originality

This research provides a different viewpoint than other previously implemented algorithms by focusing on optimizing the search space and analyzing the searching process of the optimization algorithms in test pattern generation. Since none of the optimization-based test generators presented in the literature could fully optimize the three criteria, namely, testing time, number of test vectors and fault coverage all together; hence, we use an analytical approach to reshape the use of optimization algorithms in generating test sequences by studying all the parameters in details to understand the type of interactions that exist between them. The results of the comparison are more accurate than the presented results in the literature because we use the same fitness function and fault simulator for all three optimization algorithms. By achieving the research objectives, we could reach the following key contributions:

- Identified optimal parameter values for each optimization algorithm by analyzing them individually to see their effects on the search process for a solution.
- Optimized overall results such as fault coverage/testing time to their higher/lower values.
- Identified and classified optimization algorithms based on their effectiveness in solving ATPG for sequential circuits.

1.4 Thesis Organization

The thesis chapters are organized as follows:

Chapter 2 presents a focused literature review of the three ATPG categories: deterministic-based algorithms, simulation-based algorithms and hybrid algorithms. Optimization algorithms used in automated test pattern generation were also covered in this chapter, specifically GA, DE and PSO.

Chapter 3 introduces the commonly used fault simulation algorithms. The HOPE fault simulator is presented since it is adopted in this research. The stuck-at fault model is explained in detail.

Chapter 4 introduces the three optimization algorithms used to generate test vectors. Binary versions of DE and PSO are also presented, where we try to investigate each optimization's parameters.

Chapter 5 presents the implementation of GA, BDE and BPSO to generate test vectors/test sequences. Parameters, search space and results are presented and analyzed. The performance of each implemented algorithm is presented by observing the testing time and number of test sequences which reflects the quality of each algorithm. Results are assembled at the end for comparison purposes.

Chapter 6 presents the summary and conclusions of the thesis. Several directions and recommendations are suggested for future work.

Chapter 2: Literature Review

In this chapter, we comprehensively reviewed different test pattern generation techniques for sequential circuits that have been presented in the past but with more concentration on optimization-based testing algorithms as they are a pivotal part of this work while other categories are out of scope for this research. The test generation techniques have been classified in the literature into three categories and several sub-categories, as illustrated in Figure 2.1. The classification is principally based on fault excitation, propagation and state justification, which determines the advantages and drawbacks of an algorithm.

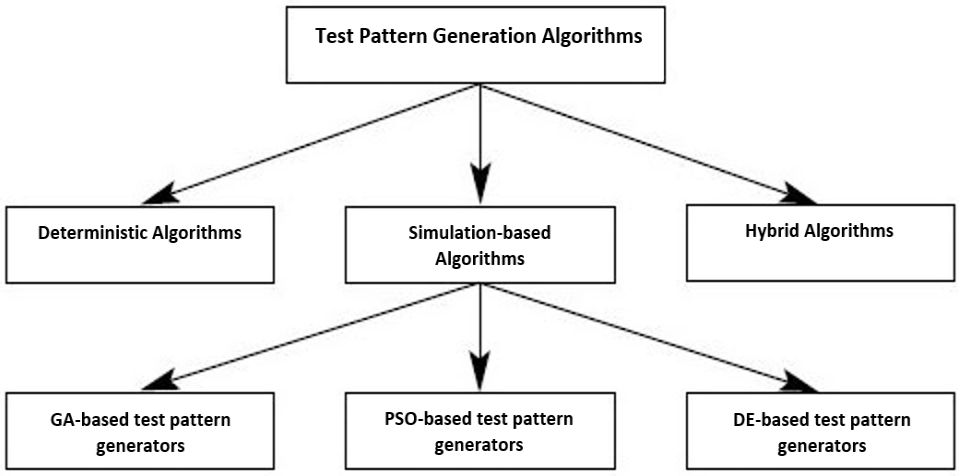

Figure 2.1: Test pattern generation categories.

2.1 Introduction

Testing IC's is an essential step when dealing with designing and engineering an IC product. Due to increasing size and complexity of real-world sequential circuits, testing is automatically fully generated and implemented. Because of the automated process, several factors must be taken into consideration, such as testing time and resource consumption. An intensive research effort has been proposed in the literature to solve for ATPG with varying degrees of success. Starting with very basic TPG algorithms such as the random test pattern generation [12], to more complex TPG algorithms such as the hybrid test pattern generation, significant progress has been made towards solving the problem of sequential circuit test generation, and yet the problem remained unresolved. The ATPG algorithms for sequential circuits can be classified into three categories: deterministic algorithms [4-6], [8-11], simulation-based algorithms [3], [12-20], and hybrid test generators [54-58]. Each approach has its own merits in terms of fault coverage and

rate of convergence. Moreover, each approach has several weaknesses which leaves the topic open for further studies.

2.2 Deterministic algorithms

Deterministic algorithms used to solve the problem of test generation for sequential circuits are complex and time consuming. However, fault coverage in deterministic algorithms is relatively high. The need for improved techniques arises to reduce the execution time and improve the fault coverage while keeping the complexity low. This is where simulation-based algorithms become good candidates to solve automated test pattern generation for sequential circuits. In deterministic fault-oriented algorithms, each targeted fault must be excited and the fault effects need to be propagated to a primary output (PO). Fault effects may propagate directly to a PO in the same time frame, in which the time frame is excited or through flip-flops to the PO's in subsequent time frames. The required state must then be justified through reverse time processing. Values required at the flip-flops are backtracked time-frame by time-frame until a time-frame is reached in which all flip-flops have do-not-care (Xs) values. The development of a deterministic test generator is very time consuming and the test sets can be large [3].

The full-scan method is often used for converting sequential circuits to combinational circuits. Hence, a test generator for combinational circuits can be used to fully test a sequential circuit. Full-scan works as follows: all flip-flops are arranged in a scan chain and the flip-flops are scanned in to read current values, then the flip-flops are scanned out after each test vector is applied. This method is not attractive due to the additional area and performance overhead. Hence, the partial-scan is more desirable since it requires less overhead and only a subset of the flip-flops is scanned.

The HITEC algorithm [4] is a state-of-the-art algorithm when discussing deterministic algorithms. This technique was used in later research with simulation-based algorithms to reach optimum results. The test generation mechanism is divided into two phases. The first phase involves a fault being activated and propagated to a PO in the forward direction only. The second phase occurs in reverse time processing to justify the initial state determined in the first phase. Despite the long testing time, fault coverage was promising when it was presented and it opened the door for further improvements in ATPG.

ATOMS [5], which is an improved version of a deterministic ATPG system for combination circuits called ATOM [6], is a structure-based technique to speed up the test vector generation for sequential circuits. It uses the PODEM algorithm [7] for time frame processing. This algorithm proposed a new method to discover unjustifiable states. Starting from a specific state, several states are visited during traversing the state transition graph of the sequential circuit in the backward direction. This algorithm uses the technique of reducing the size of the search space that needs to be explored by eliminating the 0/1 and 1/0 logic value assignment for any flip-flop in the excitation time frame. This algorithm significantly reduced the testing time and could increase the fault coverage in comparison with other deterministic algorithms described in [8] – [10].

In [11], state justification in backward time processing is listed as cube structures which make it possible to directly backtrack to the point where a decision related to a flip-flop assignment was made and a smaller number of flip-flops can be used to find a set of cubes. The results show some improvements when compared with some other deterministic algorithms.

Table 2.1 shows results obtained for two deterministic algorithms in terms of the number of vectors and fault coverage. ATOMS could obtain relatively high fault coverage in most of circuits but the number of vectors is considerably high. Moreover, the algorithms require a large testing time because of the backtracking mechanism, which is required by all deterministic algorithms. Lowering the test vectors will surely lower the testing time which the deterministic algorithms have failed to achieve.

Circuit	Total number of faults	ATOMS		HITEC	
		Fault Detected	Number of Vectors	Fault Detected	Number of Vectors
S298	308	265	312	265	322
S344	342	329	127	324	115
S349	350	335	118	332	128
S444	474	424	4433	373	1761
S641	467	404	184	404	209
S713	581	476	172	476	128
S832	870	818	998	817	1137
S1196	1242	1239	373	1239	435
S1238	1355	1283	409	1283	475
S1494	1506	1453	1162	1453	1245
S5378	4603	3379	839	3231	912
S35932	39094	34908	272	34901	496

Table 2.1: Results of deterministic-based test generators [5].

2.3 Simulation-based Algorithms

Random test pattern generation is the simplest type of simulation-based ATPG. Generating test vectors randomly is a preferred technique for only small-scale circuits with easy-to-detect faults. However, hard-to-detect faults and untestable faults will not be detected when using only the random test generator. Random test vectors can cover a significant number of faults but the fault coverage is uncertain even with a very large test set [12]. Several simulation-based algorithms have been proposed over the years. The first simulation-based test generator was proposed by Seshu and Freeman in [13]. This method uses randomly generated test patterns and any vector that increases the fault coverage is added to the test set. While this method was successful for combinational circuits, it cannot process hard to test circuits or sequential circuits. Remarkable progress in the development of simulation-based algorithms has been made since then. The basic idea behind simulation-based TPG for sequential circuits is as follows: one starts with trial test sequences. Fault simulation will be performed for those sequences and a cost function will determine how close the sequence is from being a good solution.

One of the best earlier simulation-based algorithms is the CONcurrent TEST (CONTEST) algorithm [14]. The test sequences generation process starts by initializing all flip-flops and concurrently simulating a list of faults. It ends by targeting a single fault at a time until it reaches adequate fault coverage. The fitness function uses a testability measure by estimating the number of PI's of the current vector that must be changed.

This algorithm basically uses the mutation process, one-bit change heuristic, by mutating a single bit from the previously accepted test vector. A new mutated test vector is generated for each generation. Consequently, CONTEST generates more test vectors which leads to incremental increases in the testing time. This strategy is considered evolutionary because of the evolving test sequence, per a specific fitness function. The improvement that evolutionary algorithms have brought to ATPG is very noticeable in terms of CPU time, test sequence length and fault coverage. Some of the well-known evolutionary and optimization algorithms are addressed in detail in the following sections.

2.3.1 Genetic Algorithm

The genetic algorithm (GA) was first used in automated test generation for sequential circuits by Saab et al, and the strategy is called CRIS [15]. This strategy takes advantage of the fitness function used in phase 2 of CONTEST, which is based on the distance of the fault effects to the

PO. The second fitness function selects test sequences that increase the signal activity in a circuit. The results were not encouraging. The fault coverage was low and the test sequences were long because of the insufficient fitness functions used. Another version of CRIS [16] improves the fault coverage by using a fault simulator instead of a logic simulator to evaluate candidate test vectors. The results were better in terms of fault coverage, but the execution time was longer.

GATTO [17] is another GA-based test generator. A group of 64 faults is chosen when they propagate furthest in the circuit. If a test sequence can propagate the fault effect to the PO, the test sequence will be added to a test set. The fitness function aims to maximize the circuit activity. The fault coverage is higher than the fault coverage of CRIS in some cases, but it is lower in other cases. The implementation of GATTO required the reset state to set all flip-flops to a known state before test generation. GATTO+ [18] has made several improvements on GATTO by modifying the evaluation function and mutation operator to reduce the test length. Moreover, the crossover operator was modified to work in a vertical manner.

Results in terms of number of vectors were improved significantly in GATEST, developed by Rudnick [3][19-20]. GATEST can be considered as a reference, or state-of-the-art, when dealing with GA-based ATPG. The overall performance was satisfactory with all data-dominant circuits. CPU time was reduced in comparison with deterministic algorithms. The number of test vectors was well optimized and the fault coverage was high. This strategy starts with randomly generated individuals. The process of evaluating and generating new test vectors/sequences go through four phases: (1) Flip-flop's are initialized, (2) Test vectors are generated to detect as many faults as possible, (3) The vectors that create high activity levels in the good and faulty circuits are selected, and (4) Similar to phase 2, test sequences are generated to detect as many faults as possible.

Table 2.2 shows the results for different GA-based test generators which have been presented in the literature in previous years. GATEST has superior results over other GA-based generators in terms of the number of test vectors. However, the algorithms show poor performance in fault coverage and testing time because it requires multiple visits to the flip-flips in several steps. Moreover, the technique moves between generating test vectors and test sequences which adds computation overhead and leads to increased testing time. The results indicate GATEST consumed a large amount of time to reach a reasonable fault coverage for the s35932 circuit, as

well as other large sequential circuits. Since testing time correlates greatly with the size of a circuit, the testing time for the s35932 circuit is the largest among all GA-based test generation algorithms.

Circuit	Total number of faults	CRIS			GATTO+			GATEST		
		Fault Detected	Number of Vectors	Time	Fault Detected	Number of Vectors	Time	Fault Detected	Number of Vectors	Time
S298	308	253	476	16.2s	273	215	23s	264	161	6.05m
S344	342	328	115	19.9s	319	103	1s	329	95	5.85m
S349	350				325	104	1s	335	95	5.83m
S510	564				564	968	58s			
S641	467	398	628	1.45m	406	266	22s	404	139	8.24m
S713	581	475	1124	1.58m	480	281	26s	476	128	9.41m
S832	870	370	1328	1.75m	586	401	80s	539	150	12.3m
S1196	1242	1180	2477	1.62m	1236	1091	118s	1232	347	11.6h
S35932	39094	34481	1525	2.65h	18030	3249	115.4m	35003	200	106.7h

Table 2.2: Results of several GA-based test generators [3], [17-18].

Each algorithm in Table 2.2 has optimized the test sets differently. While GATEST reduced the number of vectors, CRIS and GATTO+ reduced the testing time, and GATTO+ attained higher fault coverage in more sequential circuits than other algorithms. The optimal solution has never been obtained by any of the previous GA-based test generator since none of the algorithms could concurrently optimize the testing time, number of test vectors and fault coverage.

2.3.2 Particle Swarm Optimization

The PSO shares several similarities with other evolutionary algorithms. However, fewer parameters must be adjusted and analyzed, besides PSO is simpler and easier to implement. Recently, researchers' eyes have turned to PSO to find solutions for ATPG due to its promising results in other applications. PSO adjusts the path of particles based on information gained about each particle's best performance of its neighbors. PSO was first used in ATPG in [21]. The main concept is similar to GA-based ATPG. However, initializing all flip-flops, selecting targeted faults and generating test vectors are all performed by PSO. The technique uses two binary coded discrete PSO's [22] because inputs, outputs and all other signals are discrete values that may be zero or one. This technique starts with initializing all flip-flops logically by simulating the circuit

with the 3-valued logic simulator that can correctly compute the known state. It ends with a test set compaction phase which is important to reduce the number of test vectors, which will lead to reduction in the cost of testing. Generating test sequences is well explained in Figure 2.2. The results of this strategy were slightly improved compared to GA-based test generation algorithms, in terms of CPU time and fault coverage.

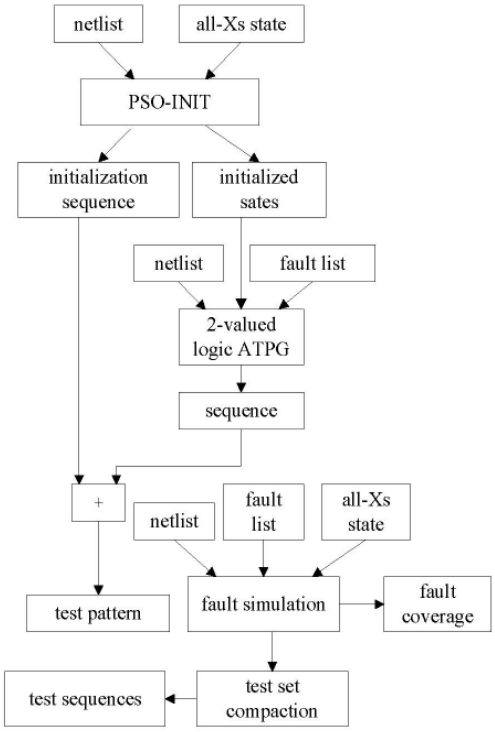

Figure 2.2: PSO flowchart for ATPG [21].

A similar strategy was used in [23] which shows a faster convergence rate than that for the GA-based ATPG. The implementation covers several steps starting from setting the initial position and velocity until it finds the best optimal solution. This method suggests one should consider using PSO in testing small-scale sequential circuits. However, this method is expected to be one of the best candidates to solve ATPG for all sequential circuits since it fully uses the circuit structure information. Moreover, there are less parameters, compared with GA, to manage when using PSO which gives PSO a higher preference over other evolutionary algorithms. Lastly, properly setting up PSO will result in faster convergence and higher fault coverage.

2.3.3 Differential Evolution

Differential evolution (DE) was recently used for generation of test vectors. The strategy selects four random individuals and mutates them based on a mutation strategy. Then, a recombination phase is performed on the new population between an individual from the old population and an individual from the new generation. Lastly, a selection phase is implemented by comparing the old population with the new population. The fitness is expected to increase from one population to the next because the evolution is biased toward more highly fit individuals. In DE-based test generator [24], each individual has an associated fitness, which measures the test sequence quality in terms of fault detection, dynamic controllability and observability measures, among other factors. The evolutionary processes of mutation and crossover are used to generate an entirely new population from the existing population. In the mutation phase, four individuals are selected randomly from the existing population, and new individuals are generated according to the mutant strategy. Then, they are selected as the members of a trial population. The mutant operator for ATPG is redefined as follows: the circuit is partitioned according to the definition of input fanout cone, that is, search from a primary input PI_j to primary outputs along the circuit, all parts linked logically to PI_j belong to the PI_j input fanout cone such that the circuit is partitioned into N sub-circuits which are logically related.

The possibility that faults inside the PI_j cone (sub-circuit) are affected by the logical value of PI_j is maximal, so the number of faults detected inside the PI_j cone is taken as the parameter of the mutant equation. A crossover (recombination) operator is applied on the trial population where one individual is selected from the old population and the other is selected from the trial population, with selection biased toward more highly fit individuals. The two individuals are crossed to generate a new individual.

Table 2.3 shows the overall results for the DE-based test generator which has achieved high fault coverage in several sequential circuits. However, more research is needed on this algorithm because of its promising results in ATPG. In comparison with other optimization algorithms, DE can detect a high number of faults. However, the number of test vectors is noticeably high as well as the length of test sequence. Testing time is not reported in [24] but DE is known for its fast convergence compared to GA-type algorithms [82].

Circuit	Total number of faults	Fault Detected	Number of Vectors
S298	308	265	132
S344	342	329	82
S382	350	364	1492
S526	474	451	4920
S641	467	404	126
S713	581	476	116
S1196	1242	1238	524
S35932	39094	35101	302

Table 2.3: Results of DE algorithm implemented in [24].

2.4 Hybrid Test Pattern Generator

The need for a combination of testing algorithms is to eliminate the shortcomings of deterministic-based algorithms and simulation-based algorithms. The basic idea behind hybrid test generators is that the testing process starts with a fast run of simulation-based test generators and then uses a deterministic-based test generator to improve the fault coverage and to identify untestable faults [3]. In [54], the CRIS-Hybrid algorithm uses a simulation-based technique [15] until there is no further improvement in fault coverage. The algorithm switches to a deterministic phase to identify untestable faults and to compute a fault cluster. A targeted set of candidate faults from the cluster is tested with a deterministic ATPG and the resulting test sequence is used to restart the search process of the simulation-based technique. The deterministic search procedure is implemented in two phases: (1) the forward time processing phase, and (2) the state justification phase. In the first phase, an undetected fault is activated and propagated to a primary output based on a PODEM-like search [7] and on a single time frame expansion. During this phase, a state may need to be justified to activate the fault. In this case, the second phase is activated to justify the state. This technique uses the logic simulator to evaluate sequences based on good circuit activity. The results show improvements in fault efficiency in comparison with the HITEC test generator. ALT-TEST [55], is another hybrid test generator which alternates between a GA-based and deterministic test generator. The test generation process is divided into three stages. The first stage attempts to detect as many remaining faults as possible from the fault list. The second stage attempts to maximize the number of visited states and propagated fault effects to flip-flops. The third stage attempts to detect the final remaining faults and to visit a new state [3]. ALT-TEST improves the fault coverage and reduces the test set length compared to results obtained from other test generators. The GA-HITEC test generator [56] was able to

relatively reduce the test vectors because of the effectiveness of GA's for state justification. It combines deterministic algorithms for fault excitation and propagation with the GA for state justification. Deterministic procedures for state justification are used if the genetic approach is unsuccessful, to allow for identification of untestable faults and to improve the fault coverage. A similar approach, called MIX, was presented in [57]. It is composed of four main procedures: circuit synchronization, state-driven test generation, deterministic test generation, and GA-based test generator. The test generation operates from the least computationally intensive to the most computationally intensive approach, such that the faults go through several test generation strategies before they are marked as aborted. The MIX-PLUS [58] test generator concentrates on areas of the search space by removing unnecessarily specified values of state variables. Identical values in the fault free and the faulty circuits during the fault propagation phase of test generation help increase the number of mandatory assignments. Thus, the search space is reduced and the fault efficiency is improved, while the run time is reduced compared to MIX. Table 2.2 shows the results of several hybrid test generators.

Circuit	Total number of faults	GA-HITEC		CRIS-Hybrid		ALT-TEST	
		Fault Detected	Number of Vectors	Fault Detected	Number of Vectors	Fault Detected	Number of Vectors
S298	308	265	415	266	331	265	221
S344	342	328	169	329	308	329	75
S349	350	335	188	334	367	335	104
S400	426	346	704	372	1461	384	2196
S526	555	367	873	433	1191	453	2109
S641	467	404	292	404	990	404	140
S713	581	467	294	475	918	476	185
S1196	1242	1239	377	1239	2467	1239	948
S5378	4603	3238	683	3503	3227	3506	7270
S35932	39094	34862	425	34899	1204	35095	825

Table 2.4: Results of several hybrid test generators [3].

From Table 2.3, ALT-TEST shows good performance in terms of fault coverage. Conversely, there is a perceptible increase in the number of vectors in several sequential circuits, such as s400 and s5378, which is considered as a tradeoff between fault coverage and the number of vectors. In comparison with simulation-based test generators, test generators that are based on

optimization algorithms have shown superior results for most of the circuits except the s35932 circuit which clearly shows the power of these algorithms if they are implemented properly.

2.5 Conclusion

In conclusion, adopting evolutionary algorithms in ATPG started years ago either as stand-alone algorithms or combined with deterministic algorithms. The promising results of implementing GA in ATPG allowed researchers to explore other evolutionary algorithms to attain global optimal solutions. A global optimal solution implies the sequence length is as short as possible; the number of test vectors is low and the testing time is significantly reduced. Optimum solutions have been found by several algorithms. However, we cannot assure that the global optimal solution has been accomplished because a global maximum solution in one algorithm becomes a local maximum in another algorithm due to several factors, such as algorithm parameter modifications and improving the implemented fitness function. Since ATPG for sequential circuits is a search process over a large vector space, the problem will remain open for further improvements and for new heuristics.

By looking at the results of using several evolutionary algorithms in ATPG, there have been positive progress since the very first adapted algorithm and the advancement continues. GA shows slowness in reaching optimal solutions with comparatively low fault coverage while PSO and DE have relatively faster convergence toward better results. On the other hand, the continued reduction in the number of test vectors affirms that the optimal solution sets have never been found for sequential circuits, thus, this issue will remain open for further improvements. The comparison between GA, DE, and PSO reflects the results obtained from several publications [3], [17-18], [21], and [23-24] that have been presented in this chapter.

Chapter 3: Fault Simulation

This chapter briefly presents the fault model used in this research and the most dominant fault simulation algorithms. The HOPE [59] fault simulator is also presented at the end of this chapter since it is the adapted fault simulator in this work and it has significant impact on the overall results.

3.1 Introduction

Simulation is the process of predicting the behavior of a circuit before it is physically built. For digital circuits, simulation has two purposes. It helps the designer to verify that the design conforms to the functional specification (called logic simulation). It is also used to simulate faulty circuits during test development (called fault simulation) [25]. A fault simulator determines whether a fault is detected or not by a given test vector. In addition, the fault simulators speed up the test generation process. The basic idea behind all fault simulators is as follows: a fault free circuit and copies of the same circuit with faults injected, such as stuck-at fault, are simulated. The same test vector is applied to all the copies of the circuit with and without the fault. The outputs of the faulty circuits are compared in the comparators, with the output of the fault free circuit. If a mismatch is reported, a fault is detected by the test vector being simulated. Several fault simulation algorithms have been presented in the literature over the years with each having its advantages and disadvantages.

3.2 Stuck-at Fault Model

The single stuck-at fault model has been successfully used in many contemporary fault simulators. Therefore, only single stuck-at faults are adopted in this research. In a single stuck-at fault, the circuit is assumed to be modeled as an interconnection (called netlist) of Boolean gates. A stuck-at fault is assumed to affect only the interconnection between gates. Each connecting line can have two types of faults: stuck-at-1 and stuck-at-0 (commonly written as s-a-1 and s-a-0, respectively). Thus, a line with a stuck-at-1 fault will always have a logic state 1 irrespective of the correct logic output of the gate driving it. A circuit with n lines can have $3^n - 1$ possible stuck line combinations. This is because each line can be in one of the three states: s-a-1, s-a-0, or fault-free. All combinations except one having all lines in fault-free states are counted as faults. Three properties characterize a single stuck-at fault [26]:

- Only one line is faulty.
- The faulty line is permanently set to either 0 or 1.
- The fault can be at an input or output of a gate.

For example, Figure 3.1 shows a circuit with a single stuck-at fault in which node D is tied to logic 0 (D is s-a-0). It is assumed that only a single fault is present in the circuit to simplify the problem. A logic one must be applied to node D if there is to be a difference between the faulty and fault-free circuits. Also, a logic zero must be applied to node C so that if the fault is present, it can be detected at the output E. In addition to the s-a-0 fault at node D, several other faults must be considered during the test generation process: s-a-1 at D, s-a-0 at A, s-a-1 at A, s-a-0 at B, etc. Some of these faults are logically equivalent, and no test can be obtained to distinguish between them. For example, in Figure 3.1, s-a-0 at A, s-a-0 at B, and s-a-0 at D, are equivalent since they are detected by the same tests. Equivalent fault collapsing is often used by test generators to identify equivalent faults to reduce the number of faults that must be targeted [3].

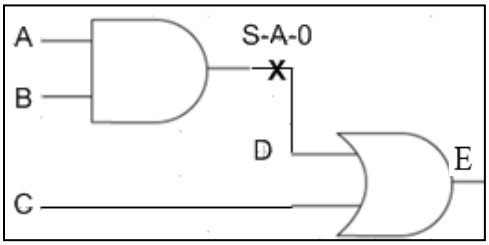

Figure 3.1: Single stuck-at fault [3].

A fault can be considered as either testable or untestable Untestable faults are faults for which there exists no test pattern that can both excite the fault and propagate its fault-effect to a primary output. In sequential circuits, untestable faults may result from the presence of unreachable states or impossible state transitions [25]. In contrast, testable faults are faults which there exists at least one test vector that can propagate its fault effect to a primary output.

3.3 Parallel Fault Simulation

In the parallel fault simulator [13] and [24], the fault free circuit and the faulty circuits are simulated simultaneously. The number of faulty simulated circuits is determined by the machine word size. If the word size is 32 bits, then 31 faulty circuits can be simulated plus the fault free circuit, simultaneously. All faulty circuits are identical to the fault free circuit except a line where a stuck-at fault is present. The parallel fault simulator lacks the capability to model accurate rise and fall delays of signals since all signal changes corresponding to several circuits

must be computed together. In general, a signal may rise in one circuit while it falls in another. In a parallel simulator, sequential logic is modeled with unit-delay [27]. More complicated delay models cannot be modeled because several faults are evaluated at the same time. Furthermore, a simulation pass cannot terminate unless all the faults in this pass are dropped. Parallel fault simulation is best used for simulating the beginning of the test pattern sequence, when a large number of faults are detected by each pattern. Two types of fault simulations use the parallelism technique to simulate faults. In **parallel-pattern single-fault propagation,** the computer word parallelism is used for parallel simulation of several faulty circuit states in the same row of the table. In **parallel-pattern parallel-fault propagation,** the computer word parallelism is used for parallel simulation of several faulty circuit states in multiple rows of the table [28].

3.4 Deductive Fault Simulation

The deductive fault simulator was first introduced by Armstrong in [29]. In this method, only the fault-free circuit is simulated. All signal values in each faulty circuit are deduced from the fault-free circuit values and the circuit structure. All deductions are performed simultaneously because the circuit structure is the same for all faulty circuits. Deductive fault simulation gains tremendous speed from processing all faults in a single pass of true-value simulation, augmented with the deductive procedures [17]. Deductive fault simulation involves allocating a fault list to each gate. The fault list contains one entry for each fault which is detectable on the output of that gate, plus one entry containing the number of faults in the list. The fault list on a gate's output can be computed from the fault lists associated with its inputs [30]. Several drawbacks relate to the deductive fault simulator; the unknown values are not easily handled. Both cases, controlling and noncontrolling values, must be considered. Moreover, deductive fault simulation is only suitable for the zero-delay timing model, because no timing information is considered during the fault propagation process. Lastly, it has a potential memory management problem. Since the size of the fault lists cannot be predicted in advance, there can be a large variation in memory requirements during algorithm execution [25].

3.5 Concurrent Fault Simulation

In concurrent fault simulation [31], every gate is associated with a concurrent fault list that consists of a set of bad gates. The concurrent fault simulator is based on the good and faulty circuits differing in a small region, which is the fan-out cone from the fault site. The concurrent

fault simulator simulates only the differential parts of the whole circuit. Concurrent fault simulation is based on the event driven simulation paradigm [5] where a change in the logic value of a node (in the good or the faulty machine) constitutes an event and causes that node to be placed on an "event queue". The simulation progresses through discrete time steps by handling all the events at the "current time" and then advancing the simulation clock. The simulation starts by applying a vector to the primary input nodes of the circuit which causes a subset of these nodes to be placed on the event queue. When an event is removed from the event queue, it is processed as follows:

- If the event results from a change in the state of a node in the good machine (good event), then all the elements gates having that node as input are evaluated. A change in an output node of any such element causes that node to be scheduled at the appropriate time (the current time plus the delay of the element).
- An event from a faulty machine (faulty event) is handled similarly with the state of that node taken from the fault effect list.
- When evaluating an element activated by a good event, any fault effect on the input nodes of the element is propagated to the output if the fault causes the state of the output to differ from its fault-free value.
- If the state of a node in the good machine becomes identical to that in a faulty machine, then the corresponding fault effect is dropped from the fault effect list on that node.

The advantage of concurrent fault simulation is its speed which results from considering only the active faults in the circuit. However, if the number of active faults is relatively large, then the speed degrades due to the overhead incurred from the maintenance of the fault effect list. Another drawback of concurrent fault simulation is its unpredictable memory requirements [29].

3.6 Differential Fault Simulation

In differential fault simulation (DSIM) [33-34], the simulator operates by combining the merits of concurrent fault simulation and single fault propagation techniques. DSIM reduces the memory requirement and the overhead of memory management in concurrent fault simulations by simulating the good machine and each machine separately rather than simulating all machines simultaneously. Furthermore, DSIM simulates each machine by reprocessing its difference from the previously simulated machine which improves its efficiency. The name derives from its use

of the difference between any two circuits. DSIM requires very little memory because it stores only one copy of all the line values of the circuit and the differences between adjacent circuits. However, it cannot drop detected faults easily because subsequent faulty circuits rely on differences from previous faulty circuits [28]. The algorithm is illustrated in Figure 3.2.

```
For every test vector, Vi,
{  /* compute good machine circuit status */
   if (Vi is the first vector )
       initialize the circuit status;
   else
       remove the previously injected fault;
   recover current states;
   set Vi pattern at the primary inputs;
   do event-driven simulation;
   store primary output values;
   set the sensitized output counter to be 0;

   for every undetected faulty machine, Bj,
   {
      remove the previous injected fault;
      recover current states;
      inject current fault;
      do event-driven simulation;
      if the sensitized output counter has positive value
          drop the fault;
   }
}
```

Figure 3.2: Differential fault simulation algorithm [25].

The problem with differential fault simulation is that the order of events caused by fault sites is not the same as the order of the timing of their occurrence. If the circuit behavior depends on the gate delay of the circuit, the timing information of every event must be included. This solution, however, can potentially require high memory consumption [25].

3.7 HOPE Fault Simulator

The HOPE [59] fault simulator is adopted in this work due to its high performance. HOPE is an efficient parallel fault simulator for synchronous sequential circuits that employs the parallel version of the single fault propagation technique. HOPE is based on an earlier fault simulator called PROOFS [60], which employs several heuristics to efficiently drop faults and to avoid simulation of many inactive faults. The following heuristics are incorporated in PROOFS:

A. Reduction of Faults to be Simulated in Parallel:

This strategy aims to reduce the number of single event non-stem faults simulated in parallel in two phases. In the first phase, all single event non-stem faults inside fanout-free regions are mapped to the corresponding stem faults by local fault simulation of the non-stem faults.

In the second phase, mapped stem faults that are sensitive are further examined for possible elimination from parallel simulation.

B. Functional Fault Injection:

The function of a gate changes to reflect the presence of a fault when it is introduced to an input, or an output of the gate. This suggests that injection of a fault into a circuit can be accomplished by introducing a new type of gate whose function reflects the behavior of the fault. Once all the faults are injected, all the gates are given a special number (code). Values 1 - 9 are assigned for fault-free gates. Faulty gates are set to "20 + faulty bit position" as their index code. Now, the lowest level gate is retrieved from the vent queue and the gate function is examined using switch and case statements which define the functionality for AND, OR and the other gates. This method does not incur an overheard in CPU time as the fault-free gates are examined in the switch-case statements. No extra gates are needed and no added events occur. But one shortcoming is that it requires a separate evaluation procedure for the faulty gates which is more complex than that for fault-free gates.

C. Static and Dynamic Fault Grouping Methods:

HOPE proposes the combination of two new fault ordering methods, a static fault ordering method performed during preprocessing, followed by a dynamic fault ordering method performed during fault simulation.

3.7 Conclusion

In this chapter, we presented several fault simulation algorithms which are essential to determine the effectiveness and performance of test vector generators. Parallelism in fault simulation has a significant positive impact on testing time by taking advantage of parallel simulation of several faulty circuit states in the same row of the table. Several heuristics were added to some algorithms to raise the performance. The HOPE fault simulator has shown superior performance in the literature by employing parallel fault simulation with several heuristics to reduce fault simulation time. It is expected that HOPE will lower testing time of the implemented algorithms in this work considerably, and it will positively participate in achieving this work's objectives, outlined in section 1.2.

Chapter 4: Optimization Algorithms

In this chapter, a detailed explanation of the three adapted optimization algorithms, namely, GA, DE and PSO is presented. In addition, binary versions of PSO and DE are also presented since this research is only concerned with binary-coded solutions.

4.1: Introduction

The adapted optimization algorithms in this thesis belong to the class of evolutionary algorithm (EA) and swarm intelligence (SI). All the algorithms share a similar strategy as they are all population-based algorithms, but the implementation is different. In EA, the environmental pressure causes natural selection, or survival of the fittest, which causes a rise in the fitness of the population. Given a function to be evaluated, we can randomly create a set of candidate solutions, i.e., elements of the function's domain, and apply the quality function as an abstract fitness measure - the higher the better. Based on this fitness measure, some of the better candidates are chosen to seed the next generation by applying recombination and/or mutation to them. Recombination is an operator applied to two or more selected candidates, the so-called parents, and results in one or more new candidates, i.e., the children. Mutation is applied to one candidate and results in one new candidate. Executing recombination and mutation leads to a set of new candidates, the offspring, that compete based on their fitness, and possibly age, with the older ones for a place in the next generation. This process can be iterated until a candidate with sufficient quality (a solution), is found or a previously set computational limit is reached [61].

SI concerns the collective, emerging behavior of multiple, interacting agents that follow some simple rules [74]. Each agent may be considered as unintelligent, while the whole system of multiple agents may show some self-organizational behavior and thus can behave like some sort of collective intelligence. Many algorithms have been developed by drawing inspiration from the SI systems in nature. The main properties for SI-based algorithms can be summarized as follows:

- Sharing of information among the multiple agents.
- Agents have self-organization and co-evolution.
- It is highly efficient for its co-learning.
- It can be easily parallelized for practical and real-time problems [75].

4.2: Genetic Algorithm

Genetic algorithms were presented by Holland in 1975 in his book *Adaptation in Natural and Artificial Systems* [64]. GA is an evolution-inspired algorithm for optimization and machine learning. It starts with an initial population of chromosomes Figure 4.1. A chromosome can be binary-coded or it might contain a character from a larger alphabet (non-binary-coded). The initial population, which is typically generated randomly, evolves into better populations (solutions) by using a kind of "natural selection" together with the genetics−inspired operators of crossover and mutation. The process of selection and reproduction is repeated until a complete new generation is generated and then the old generation will be discarded. A fitness function is required to measure the quality of each generated solution. The fitness is expected to increase from one population to another since the selection is biased towards highly fit individuals. The fittest individual will survive over consecutive generations for solving a problem.

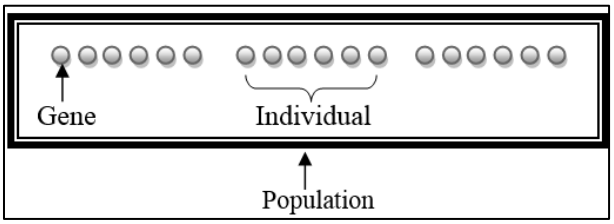

Figure 4.1: GA Population.

The GA is summarized in the pseudo-codes provided in Figures 4.2 and 4.3.

```
Begin
      Generate a random population of solution;
      Calculate the fitness of the initial population;
      Repeat
      {
         Select a pair of parents based on fitness
         Create two offspring using crossover
         Apply mutation to each child
         Evaluate the mutated offspring
         All the off spring will be the new population and the parent will die
         Stop if the stopping criteria is met.
      }
End
```

Figure 4.2: (A) Pseudo-code for GA.

```
Generate Initial Population
Evaluate (Population);
bestIndivdual = best of (Population);
For I = 1 to NumGenerations Do
       NewPopulation = 0;
       For j=1 to PopulationSize/2 Do
              Select P1 and P2 from Population;
              Crossover (P1, P2) => The results (c1, c2);
              Mutate (c1);
              Mutate (c2);
              Add c1 and c2 to NewPopulation;
       Evaluate (NewPopulation);
       Population = NewPopulation;
       bestIndividual = bestOf (Population U bestIndividual);
Solution = bestIndividual;
```

Figure 4.3: (B) Pseudo-code for GA

4.2.1: Genetic Operators

4.2.1.1: Selection

The meaning of selection is to pass an individual from one generation to another. An individual is selected based on a fitness function which will determine the quality of each individual. Selection ensures that survival of the fittest individual is achieved through several techniques to carry individuals from one generation to another. The following selection techniques are widely used in several applications and they are briefly explained in the following sections:

- **Roulette Wheel Selection**

  The most straightforward implementation of the selection rule is the so-called roulette-wheel selection [35]. The conspicuous characteristic of this method is to give each individual *i,* a probability *p(i),* to be selected. It is also known as fitness proportionate selection. Each individual in a population is allocated a segment in a roulette wheel and the size of the segment is proportional to its fitness value. Figure 4.4 illustrates how an individual is selected by using the roulette wheel method. Since the size of a segment depends on the fitness value, individuals with higher fitness values have more probability of being selected, which may lead to biased selection towards high fit individuals. However, there is no guarantee that good individuals will be passed to the next generation.

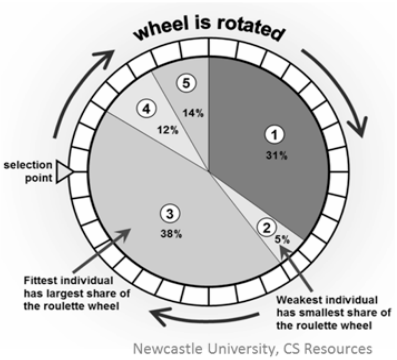

Figure 4.4: Roulette wheel selection.

The selection probability is given as:

$$p(i) = \frac{f(i)}{\sum_{j-1}^{n} f(j)},$$

where n donates the population size, and $f(i)$ is the fitness of each individual.

Since the selection is directly proportional to fitness, it is possible that strong individuals may dominate in producing offspring, thereby limiting the diversity of the new population. In other words, proportional selection has high selective pressure [36].

- **Rank Selection**

  Rank selection sorts the individuals according to their fitness values, where the rank N is assigned to the best individual and rank 1 is assigned to the worst individual. Ranking selection was proposed to eliminate disadvantages of proportionate selections and to overcome the drawback of premature convergence to a local optimum [37]. The selection probability in ranking selection is proportional to relative fitness rather than absolute fitness. This type of selection tends to avoid premature convergence by tempering selection pressure for large fitness differentials that occur in early generations.

- **Tournament Selection:**

  This type of selection is another widely-used selection strategy in the GA. The idea of tournament selection is simple. Select some number of individuals randomly from a population (with or without replacement), select the best individual from this group for further genetic processing with fixed probability p, and repeat as often as desired (usually until the mating pool is filled). Tournaments are often held between pairs of individuals (tournament size s), although larger tournaments can be used and may be analyzed [38].

4.2.1.2: Fitness Function:

A mathematical representation is used to determine the ability of an individual to survive from one generation to another and it is used to quantify how good the solution is represented by a chromosome [38]. The fitness function should be chosen in such a way that a chromosome that is closer to the optimal solution in the search space should have a higher fitness value. The fitness function is the only information, also called the payoff information, that GA's use while searching for possible solutions [40].

4.2.1.3 Crossover

Crossover, or recombination, is simply a matter of replacing some of the genes in one individual by genes of the corresponding individual. Crossover combines (mates) two chromosomes (parents) to produce a new chromosome (offspring). The idea behind crossover is that the new chromosome may be better than both parents if it assumes the best characteristics from each of the parents. Crossover occurs during evolution according to a user definable crossover probability [62].

A. One-Point Crossover:

The parental chromosomes are split at a randomly determined crossover point. Subsequently, a new child genotype is created by appending the first part of the first parent with the second part of the second parent [39], as shown in Figure 4.5.

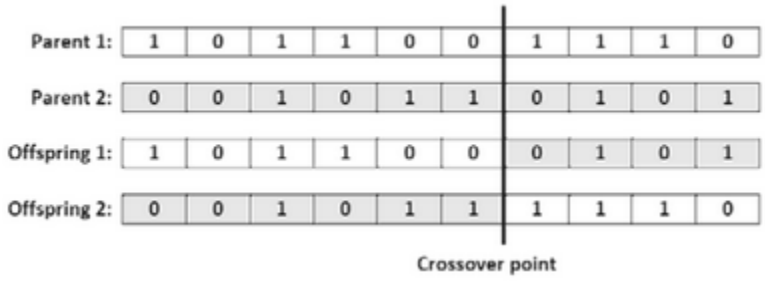

Figure 4.5: One-point crossover [40].

B. Two-Point Crossover

In two-point crossover, or m-point crossover, two points are randomly selected between 1 and L-1, where L is the length of the chromosome. The contents between these points are exchanged between two mated parents, as shown in Figure 4.6. Adding more crossover points reduces the performance

of the GA because building blocks are more likely to be disrupted. However, an advantage of having more crossover points is that the problem space may be searched more thoroughly.

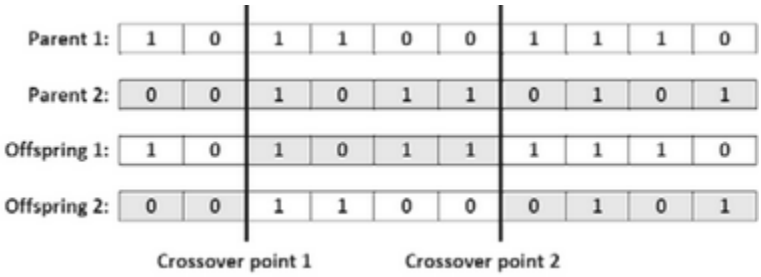

Figure 4.6: Two-point crossover [40].

C. Uniform Crossover:

Two chromosomes are combined to produce new offspring. With the same probability *p,* bits are copied from either the first parent or the second parent to make a new offspring [40], as shown in Figure 4.7.

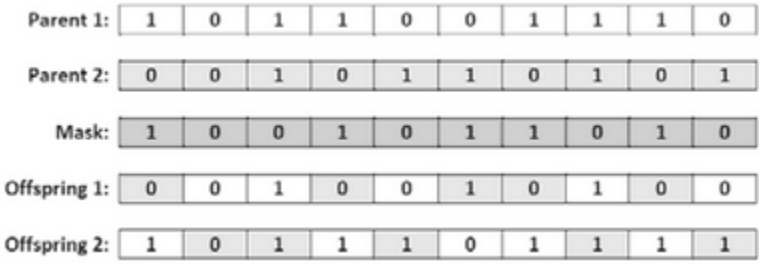

Figure 4.7: Uniform crossover [40].

4.2.1.4 Mutation

In binary-coded chromosomes, mutation means flipping a gene in a chromosome with probability *p,* where *p* is the probability that a single gene is modified. Since each gene has two states: zero or one, the size of the mutation step is always one and it happens less frequently because it is a divergence operation to discover a better minimum/maximum space by breaking one or more members of the population out of the local minimum/maximum space. The gene to be mutated is mainly randomly selected. However, there are other mutation techniques for a given string such as:

- **Inversion of a single bit**: One randomly chosen bit is inverted with probability *p*. See Figure 4.8.
- **Bitwise inversion**: The entire string is inverted bit by bit with probability *p*.

- **Random Inversion**: The string is replaced by a randomly chosen one with probability *p*.

Before Mutation	1	0	1	1	0	0	0	1
After Mutation	1	0	1	1	0	1	0	1

Figure 4.8: Single bit inversion.

4.3: Differential Evolution

Differential evolution (DE) [47-48] is a population-based algorithm that has been successfully employed to solve a wide range of global optimization problems. The method of DE, as illustrated in Figure 4.9, is nearly identical to the GA's approach. DE allows each successive generation of solution to 'evolve' from the previous generations' strengths.

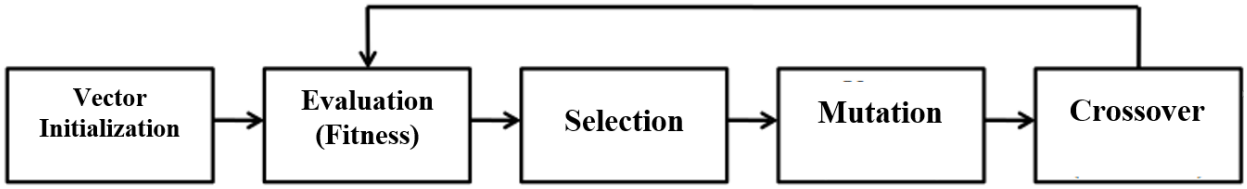

Figure 4.9: Differential evolution algorithm.

The DE method can be applied to real-valued problems over a continuous space with much more ease than the GA. In DE, a parent vector from the current generation is called a *target* vector, a mutant vector obtained through the differential mutation operation is known as a *donor* vector, and finally an offspring formed by recombining the donor with the target vector is called a *trial* vector [70]. The idea of the DE method is that the difference between two vectors yields a difference vector which can be used with a scaling factor to traverse the search space. As in the GA, DE begins with a random population which is chosen equally over the problem space, and the next generation creates an equal number of donor vectors (mutant vectors) that are created through means of:

$$\forall i \in n : D_i = X_3 + F\,(X_1 - X_2).$$

The "mutation" step is shown in Figure 4.10, where x and y are the axes of the decision space, X_1 is chosen either randomly or as one of the best members of the population (depending on individual encodings), X_2 and X_3 are randomly chosen and F is the scale factor.

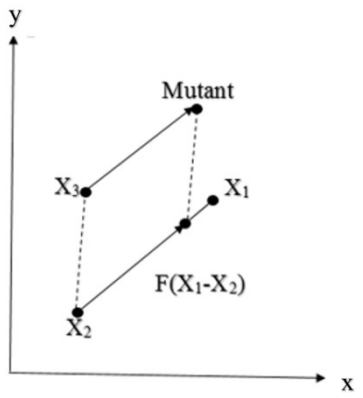

Figure 4.10: A simple DE mutation scheme [80].

A trial vector $T_{i,j}$ is created by selecting between the donor vector and the previous generation for each element (j) according to the crossover rate (*CR*) 0–1. For each element in the vector we choose either the corresponding element from the previous generation vector or from the donor vector, such that:

$$\forall i, j : \text{if}(\text{ rand } < CR \parallel j = J_{rand}) \text{ then } T_{i,j} = D_{i,j} \text{ otherwise } T_{i,j} = X_{i,j,}$$

where J_{rand} is randomly chosen for each iteration through i and ensures that no T_i is the same as the corresponding X_i. Then the trial vector's fitness is evaluated, and for each member of the new generation, X'_i, we choose the better performing of the previous generation, *Xi*, or the trial vector, T_i [49]. DE is considered to be completely self-organized since it adds the weighted difference between two population vectors to a third vector. DE is summarized in the pseudo-code provided in Figure 4.11.

```
Begin
   Generate randomly an initial population of solutions.
   Calculate the fitness of the initial population.
   Repeat
      For each parent, select three solutions at random.
      Create one offspring using the DE operators.
      Do this a number of times equal to the population size.
      For each member of the next generation
         If offspring(x) is more fit than parent(x)
            Parent(x) is replaced.
   Until a stop condition is satisfied.
End.
```

Figure 4.11: Pseudo-code for DE.

Since the ordinary DE is incapable of working with problems with binary-valued parameters, several heuristics have been applied to DE to solve binary-coded problems. Discretized differential evolution (DDE) [71], uses a sigmoid function to discretize a normalized solution vector to form a bit string. DDE works as follows: an initial random population is created with

initial individuals and their initial fitness is calculated. Through the number of generations previously set, a trial population is created using the mutation and crossover processes. This new population is discretized by the sigmoid function which assigns the values 1 or 0, depending on whether the continuous dimension of the individual is greater than 0 or not. The fitness of this trial and discretized population is calculated and if the trial individual fitness is greater than the previous one, the new individual is incorporated into the new population. Figures 4.12 and 4.13 present the pseudo-code of the DDE algorithms [71].

```
Function f(x) = DE (range, NP, CR, F)
x ⇐ random (range, NP)
fit_x ⇐ f(x)
while not done do
  for i = 1 to NP do
    v_{i,G+1} ⇐ mutation (x_{i,G}, F)
    u_{i,G+1} ⇐ crossover (x_{i,G}, v_{i,G+1}, CR)
  end for
  if sigmoid (u_{i,G+1}) > 0 then
    u_{i,G+1} ⇐ 1
  else
    u_{i,G+1} ⇐ 0
  end if
  fit_u ⇐ f(u)
  for i = 1 to NP do
    if fit_u(i) > fit_x(i) then
      x_{i,G+1} ⇐ u_{i,G+1}
    else
      x_{i,G+1} ⇐ x_{i,G}
    end if
  end for
end while
```

Figure 4.12: DDE algorithm [71].

```
Parameters: range, NP, PM, PR
Initial Population x_i (i = 1, ..., range)
Evaluate fitness f(x_i) of each individual
while not done do
  for i = 1 to NP do
    if rndreal(0,1) < PR ou j = j_rand then
      if rndreal(0,1) < PM then
        InverterBit(y_j)
      end if
      Crossover(y_j)
    end if
  end for
  Calculate fitness f(y)
  if f(y) > f(x_i) then
    x_i ⇐ y
  end if
  Evaluate x*
end while
```

Figure 4.13: BDE Algorithm [71].

Here, *NP* is the population size, *CR* is the crossover rate, *F* is the mutation rate, *PR* is the permutation rate, *PM* is the mutation rate and the *range* is the dimension of each individual.

Binary Differential Evolution (BDE) was presented in [69], which uses homomorphous mapping and the interpretation of the continuous solution vector as a vector of probabilities. An older version of BDE was introduced in [73] which uses an angle modulation. It consists of a generation of a bit string using a trigonometric generating function. Another version of BDE was presented in [72] which is only feasible in binary search space and the original mutation process of DE is replaced by a random bit inversion, which is inspired by GA. The perturbation process is a new parameter introduced to measure how many individuals of the population will pass

through the mutation and crossover processes and it will ensure there is at least one individual that will be mutated. The crossover process remains unchanged.

4.4: Particle Swarm Optimization

Particle swarm optimization (PSO) is another population-based search algorithm that simulates the social behavior of agents that interact with each other by acting on their local environment. It was designed and presented in 1995 by Kennedy and Eberhart [42]. The algorithm starts with an initial population of the solution, called particles, and searches for the optimum solutions by updated generations. The particle swarm concept originated as a simulation of a simplified social system. The original intent was to graphically simulate the graceful but unpredictable choreography of a flock of birds. Initial simulations were modified to incorporate nearest-neighbor velocity matching, eliminate ancillary variables, and incorporate multidimensional search and acceleration by distance [41-43]. At some point in the evolution of the algorithm, it was realized that the conceptual model was, in fact, an optimizer. Through a process of trial and error, a number of parameters extraneous to the optimization were eliminated from the algorithm, resulting in the very simplified original implementation [44].

In PSO, each particle is treated as a point in the *D*-dimensional problem space. The *ith* particle is represented as $X_i=(x_{i1},x_{i2},\ldots\ldots,x_{iN})$, the best previous position (the position giving the best fitness value) of the *ith* particle is recorded and represented as $P_i=(p_{i1},p_{i2},\ldots\ldots,p_{iN})$, the index of the best particle among all the particles in the population is represented by the symbol *g*. The rate of position change (velocity) for particle *i* is represented as $V_i=(v_{i1},v_{i2},\ldots\ldots,v_{iN})$, the particles are manipulated according to the following equations:

$$v_{ij}(t+1) = w\,.\,v_{ij}(t) + c_1\,.\,\text{rand}()\left(\text{pBest}_{ij}(\text{t}) - x_{ij}(t)\right) + c_2\,.\,\text{rand}()\ \left(\text{gBest}_{ij}(t) - x_{ij}(t)\right),$$

$$x_{id} = x_{id} + v_{id}.$$

Here, c_1 and c_2 are positive constants and the random variable (*rand()*) is a uniform distribution between 0 and 1, *w* is the inertia weight which shows the effect of the previous velocity vector on the new vector. A larger inertia weight facilitates global exploration (searching new areas) while a smaller inertia weight tends to facilitate local exploration to free-tune the current search area. Suitable selection of the inertia weight provides a balance between global and local exploration abilities and thus requires fewer iterations on average to find the optimum [4], [45].

A modified version of the PSO algorithm was introduced in 1997 [22], that allows PSO to work in binary space. In the binary PSO (BPSO), the particle's personal best and global best is updated as in the continuous version. The major difference between BPSO with the continuous version is that velocities of the particles are defined in terms of probabilities that a bit will change to one or zero. Using this definition, a velocity must be restricted within the range: 0-1. Updating the velocity vector of the particle is performed using the similar equation from PSO. The normalization procedure is performed by a sigmoid function:

$$\text{sig}\left(v_{ij}(t+1)\right) = \frac{1}{1+\text{e}^{-\left(v_{ij}(t+1)\right)}},$$

and the new position of the particle is obtained using the equation below:

$$x_{ij}(t+1) = \begin{cases} 0 & \text{rand} \geq \text{sig}\left(v_{ij}(t+1)\right) \\ 1 & \text{other} \end{cases}.$$

The BPSO can be used in a variety of applications, especially when the values of the search space are discrete like decision-making, solving the lot sizing problem, the traveling salesman problem, scheduling and routing. BPSO can be effectively employed to solve ATPG for sequential circuits because of its effectiveness in solving several problems such as those mentioned in [76-78].

4.5: Conclusion

The optimization algorithms included in this chapter are considered the most suitable candidates to generate test sequences for sequential circuits. Although, optimization algorithms share a similar principle as they are all population-based algorithms, there are several differences in terms of the searching mechanism for an optimal solution and updating the individual's position. Throughout this research, we investigate each algorithm's parameters to show their significance in an algorithm implementation to search for an optimum in ATPG. Moreover, some algorithms are used differently than originally intended by adding heuristics for performance improvements which will be explained in detail in the following chapter.

Chapter 5: Test Generation - Implementation and Results

This chapter presents a detailed explanation of the implementation of GA, DE and PSO in generating test sequences for sequential circuits. A complete investigation of the performance of each algorithm is also presented, followed by a comparison between each algorithm in terms of implementation, performance and results. Boxplots [51] were chosen to represent the results because they help to identify the variation of data and understand an algorithm's performance. It is worth mentioning that we use the same fitness function for all optimization algorithms in this chapter for equitable comparison. The following fitness function measures the quality of a generated test sequence:

$$\text{Fitness} = \text{number of faults detected} + \frac{\text{number of faults propogated to flip\_flops}}{(\text{number of faults})(\text{number of flip\_flops})(\text{sequnce length})}.$$

An analysis of the random test generator was also included in this chapter to see the advantages and drawbacks, if any, of adopting optimization algorithms for test sequences generation.

5.1 Circuits Description

ISCAS89 sequential benchmark circuits [50] were tested in this research because of their high adoption in the literature. The chosen circuits range from small-scale circuits to large-scale circuits. Table 5.1 presents a brief description of the several selected circuits, where the PI column is the number of primary inputs in the circuit, and PO is the number of primary outputs of the circuit. Sequential depth is defined as the maximum structural sequential depth of all flip-flops in the circuit, where the structural sequence of a flip-flop is defined as the minimum number of flip-flops that must be passed through to reach that flip-flop from a PI [2]. In this research, we considered the single stuck-at fault model in synchronous sequential circuits under the zero-gate delay model. The HOPE [59] fault simulator was used to simulate each test vector and compute its fitness.

	Circuit	PIs	Sequential Depth	Total Faults	POs
1	S298	3	8	308	6
2	S344	9	6	342	11
3	S349	9	6	350	11
4	S510	19	2	564	7
5	S641	35	6	467	24
6	S713	35	6	581	23
7	S953	16	3	1079	23
8	S1196	14	4	1242	14
9	S35932	35	35	39094	320

Table 5.1: Description of sequential circuits [3].

5.2 Random Test Generation

Random test generation is the simplest type of simulation-based test generation where test vectors are applied to a sequential circuit and a fault simulator compares the faulty circuit and good circuit and determines the existence of a fault. For the analysis, we started by randomly testing all targeted sequential circuits to classify the type of detected faults. Usually, some faults are easily detected by randomly generated test vectors which are classified as easy-to-detect faults. The downside of the random test vectors generator is that it requires an enormous number of test vectors for fault detection and it consumes a high percentage of CPU resources. While random test generation is recommended for relatively small-scaled sequential and combinational circuits, its efficiency is noticeably reduced as the size of the circuits increase.

In this work, the test runs several times, each time with a different number of test vectors, with each increment is a factor of 10. In sequential circuits fault detection, a sequence of test vectors is needed because the output of a circuit is dependent on both the present and previous (past) inputs to the CUT. A test vector plays two roles: drive the circuit under test into a proper state and detect the given fault from that state. All flip-flops in each circuit are assumed to be in a known state, either all zeros or all ones, before the test runs. Hence, we assume that fault detection will start after applying a few test vectors. This assumption holds true for all sections of this research.

Each circuit has a breakout point where no more faults will be detected no matter how many more test vectors are randomly applied. For example, fault coverage became nearly constant when the test vectors reached 100,000 vectors for the s298 circuit. Figure 5.1 shows the improvement of fault coverage as the test vectors were randomly applied to the s298 circuit.

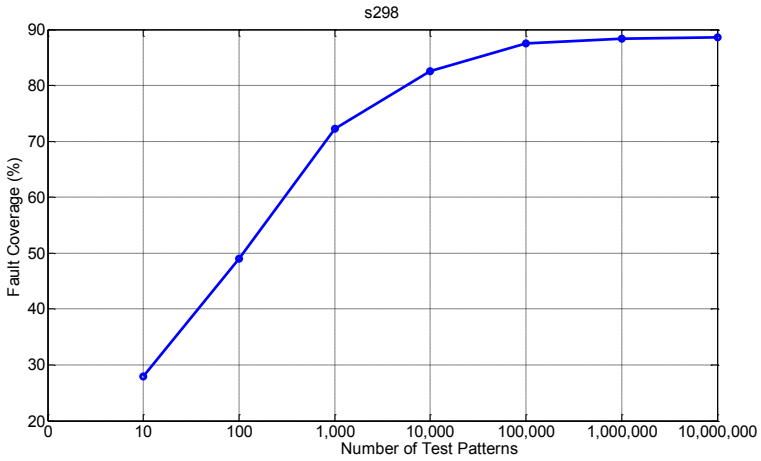

Figure 5.1: Fault coverage for s298 circuit.

The fault coverage stays below 90% even after one million test vectors. The remaining faults (undetected faults) are considered as hard-to-detect faults which means that an advanced algorithm is needed to detect the faults. Figure 5.2 shows similar behaviour of fault detection using random test pattern generation for a larger circuit. Fault coverage increases as the test vectors are fed to the fault simulator until the fault coverage becomes unchanged. Determining the stopping point, the point where no more faults are detected, is not possible due to several reasons such as the nature of a circuit's faults and the size of a circuit. The s27 circuit, shown in Figure 5.3, gained full coverage with less than 200 test vectors randomly generated because all stuck-at faults in the circuit are easy-to-detect faults, which makes the circuit amenable to full coverage using random test pattern generation.

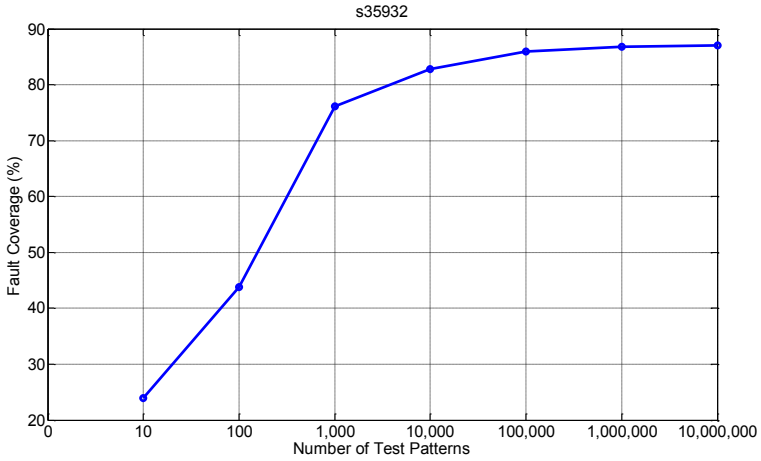

Figure 5.2: Fault coverage for the s35932 circuit.

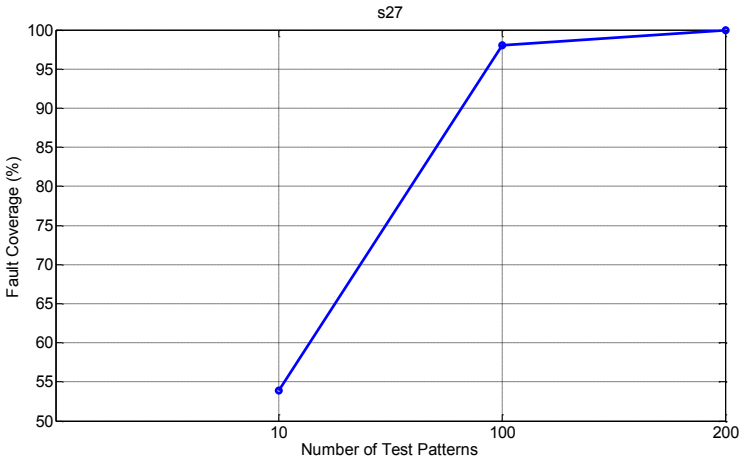

Figure 5.3: Fault coverage for the s27 circuit.

Figure 5.4 clearly shows that random test generators can only be used to detect easy-to-detect faults. In the s400 circuit, fault coverage could not reach 20% due to the nature of the faults. Consumption of the CPU resources increased as we increased the testing limit. Fault detection is dependent on fault nature (type) and it does not completely depend on the number of test vectors nor the sequence length. Other graphs for other circuits are shown in Appendices A-D.

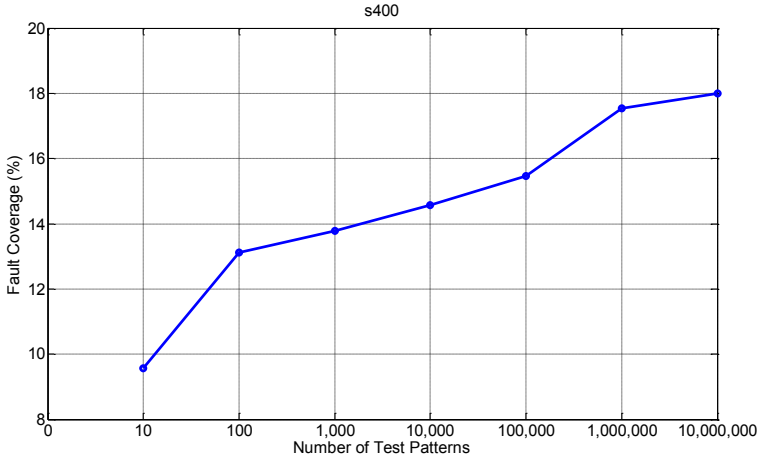

Figure 5.4: Fault coverage for the s400 circuit.

The behaviour of all graphs illustrates that an appropriate stopping criteria is highly needed. The stopping criteria cannot be easily determined because the faults differ from one circuit to another. However, random test pattern generation is highly recommended as a first testing tool, but it is not capable of reaching full fault coverage independently of other advanced testing tools for most of the sequential circuits. Random test pattern generators perform better on data-

dominant designs of small-scale circuits. If untestable faults are present, another advanced technique is highly recommended.

Testing time can be obviously seen with larger circuits such as s35932. The sequential circuit s35932 gained 86.993% fault coverage in 40.5 minutes by one million test vectors applied, which is considered a very high testing time. Random test pattern generator is not an effective solution, nor a practical test generator candidate to solve ATPG problems for sequential circuits even with full fault coverage when it is reachable. The need arises to find another testing algorithm to optimize the search space in the random test generator because that will lead to optimized testing time with the number of test vectors.

Searching for a solution by using random test generators is not effective nor efficient due to two major reasons: the search space is promotional to the size of a circuit and to the sequence length, which means that search space becomes enormous with large circuits and ultra large circuits. The second reason is that the search mechanism is not directed by any means which leads to visiting a state several times regardless how good the state is. The consequence of these two reasons is that resource consumption in terms of time and memory becomes maximum. To make the use of random test generators effective and efficient, two points must be considered. Firstly, a circuit must be small in size. As the size of a circuit increases, the efficiency of the test generations decrease. Secondly, defining a stopping criteria for test generation in terms of a fault coverage percentage will thus increase the efficiency.

5.3 Genetic Algorithm

The GA was explained in detail in section 5.1. I only used binary coding where every chromosome is a string of bits, 0 or 1. Each character of a string is mapped to a PI. Thus, all PI's are set to known states, 0 or 1, and the unknown state X is discarded.

The sequential GA-based test generator is divided into two processes and several sub-processes:

1. GA Pre-process:
    - Initialize all flip-flops.
    - Preprocess and partition the circuit.
2. GA Process:
    - Generate a random sequence (vector series) as an initial population.
    - Compute the fitness of each sequence.

- The evolutionary processes of GA are used to generate a new population from the existing population.

Initialization of sequential circuits is implemented in the pre-process state where all flip-flops in a circuit will be set to Zero (off-set). The first "time frame" will have a pseudo-primary Output (PPO) of zeros. This helps to shorten the test sequence since several leading bits will be discarded (Figure 5.5). Note that this assumption is valid if an external reset signal is implemented by resettable flip-flops. Several test vectors might be used to ensure that all flip-flops are set to Zero. These vectors will be applied to the fault-free circuit. This step is not obligatory. However, it will be important if a circuit contains some portions that are hard to initialize and it can take place within the pre-process step. Most of the work concentrated on test vector generation for fault detection (GA Process step).

In sequential circuits, a circuit must be in a specific state (time frame) to detect a specific fault. A sequential circuit is duplicated into several copies to represent the circuit in a different state. A sequential circuit should be driven to a specific state; all flip-flops should have specific values, in order to arrive at a specific primary output.

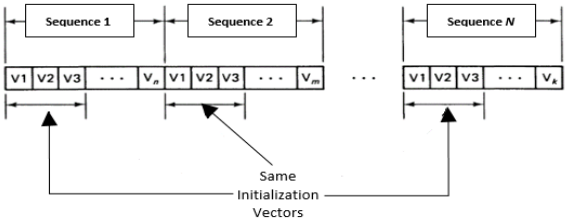

Figure 5.5: Test sequences.

5.3.1 Parametric Analysis

Several GA parameters should be investigated carefully to achieve satisfactory fault coverage. The initial population is the first element that needs to be addressed. Most of the previous GA-based test pattern generators ignore the effects of the initial population on later populations and several publications did not consider the initial population as a "Parameter". In GA-based generators, each population is constructed of individuals. Each individual represents a sequence of test vectors. A vector within a sequence is supposed to either detect a fault or drive a circuit to a specific state, called a time frame. A vector (gene) is said to be a relatively strong vector (gene) if it can detect a fault, or faults, or it causes the effects of a fault, or faults, to propagate to a flip-flop, or flip-flops. A population should be large enough to ensure adequate diversity. In addition,

it is important that the initial (first) population is strong enough to increase the rate of convergence. Since the first (initial) population is generated randomly and applied to a sequential circuit, the initial population's performance becomes unpredictable unless it is calculated afterwards. The results of the initial population are a measure of the later populations.

It is obvious that the quality of the initial population correlates with positive evolution of the population unless good genes were lost during the sequence process. To ensure that an initial population is good enough before proceeding to the next generation, the quality of the initial population is calculated before deciding whether to move to the next generation or to create another initial population. If the evaluated initial population is equal to or greater than a specific value, which is a predefined value, the next generation will be generated, otherwise another new initial population will be randomly generated. By performing this step, computation is reduced in later generations alongside gaining a faster rate of convergence. Moreover, selection operator effects on the results have been minimized. In this work, the algorithmic condition, shown in Figure 5.6, was added to the initial population generation.

```
Generate Initial Population;
μ = ∑( Evalulate Initial Population);
If  μ ≥ Specific Value
Proceed to Next Generation;
Else;
Generate another Initial Population;
```

Figure 5.6: Initial population algorithm.

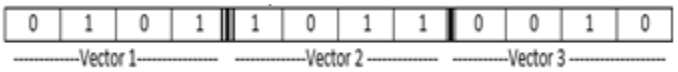

Figure 5.7: Test sequence.

Another key parameter is the population size. It has been mentioned earlier in this section that a population should have a sufficient size to avoid computational overhead. The population size is not fixed for all circuits. The size is related to vector length (see Figure 5.7), and primary inputs, to ensure a higher degree of diversity. Table 5.2 [3] was found to be useful in determining the appropriate population size for sequential circuits. However, there are some limitations on applying these population size values to GA. As the vector length increases (> 99), the initial population will cover a very small area of the whole search space. To overcome this issue, the number of generations needs to be increased to allow the individuals to explore the maximum

possible area of the search space. However, computational overhead will increase because of the incremental increase in the number of generations.

Vector Length	Population Size
< 4	8
4 – 16	16
17 – 49	16
50 – 63	24
64 – 99	24
> 99	32

Table 5.2: Population size value [3].

Another important parameter is the number of generations. The program will stop when it reaches full coverage, or when it reaches the maximum number of generations. It is possible to reduce the computation overhead by lowering the number of generations. However, we might end up having poor fault coverage. Several applications [3], [19-20] decided to limit the number of generations for time reduction purposes. This technique works well, with large sequential circuits; only if the population size is highly optimized with the existence of good-enough individuals.

Crossover and mutation are other major and crucial parameters. In this work, I used a probability of 1 for crossover which means that two individuals are always crossed to generate new individuals. Generating new test sequences through the process of crossover will lower the probability of applying identical sequences to the sequential circuit. As the crossover probability decreases, the probability of applying similar test sequences in the following generation increases. The crossover scheme occurs between vectors within two sequences. It is basically swapping two vectors (genes) of two different sequences to generate a new set of sequences. A negative consequence of crossover is that a good vector (a key individual) might be lost which may lead to a negative change of search direction.

Mutation probability is a problem-dependent value. Having a high mutation rate value will cause the search space to be maximized, while a very low mutation value will cause premature convergence. The mutation selects a position, either randomly or previously-defined, and complements it. Many publications suggest that the mutation rate ranges between 0.005 – 0.01 [63]. However, since the length of test sequences varies from one circuit to another, it is better to modify the mutation rate accordingly. In this work, a mutation rate of 0.01 was used by default and it was modified according to vector length. Mutation might cause the search direction to be completely changed if it complements a key individual within a test sequence. Test sequences

processing caused by mutation and crossover operators have relatively higher effects on the results than the selection operator.

The fitness function is the major parameter when discussing optimization algorithms. In GA-based test pattern generations, the fitness function will measure the ability of a test sequence to either detect a fault or propagate a fault to a flip-flop. Since the main goal of ATPG is to detect the maximum number of faults, the fitness function will calculate how many faults were detected by each test vector. Furthermore, the fitness function will determine the ability of a test vector to propagate fault effects to a flip-flop. Then, the overall quality of a test sequence will be determined.

5.3.2 Results

The results of implementing GA in ATPG are demonstrated below. Figures 5.8, 5.9 and 5.10 show the results of the s298 circuit with two different selection strategies and different crossover approaches. Two-point crossover attained higher fault coverage with lower data distribution. The uniform crossover has higher distribution with lower fault coverage because the random gene swapping led to an increase in the probability of losing good genes and the search for a solution is randomized. The overall results show that one-point and two-point crossover techniques have firm and explicable results, unlike uniform crossover. It has been mentioned earlier that the selection technique does not have a major impact on the results. However, rank selection causes the competition to occur between strong sequences because of the ordering scheme. The outcomes of using rank selection show that strong sequences live longer until later generations. The sensitivity of a test vector (gene) within a sequence is high which means that any change within a sequence may cause the search direction to change dramatically, then the fault coverage becomes unpredictable. Each test vector causes a sequential circuit to move from one time frame to another, a change in a test vector will cause the circuit to arrive at a different time frame and the search direction will change because of a change in the test vector. The mutation operator has little effect on fault coverage for most of the circuits. Figure 5.11 shows the effects of different mutation rate values on the s298 circuit. Results have little variation from one value to another. It is recommended that mutation occurs either at the beginning, or at the end of a population to explore much more search space. The default mutation rate of 0.01 shows good-enough results for most of the sequential circuits.

The number of test vectors is a crucial parameter. The s298 circuit has its highest fault coverage with about 144 test vectors in less than two seconds. These test vectors could cover nearly 88% of all faults. Random test pattern generation could find the same fault coverage with more than 100,000 test vectors. The s35932 circuit has its highest fault coverage with about 197 test vectors in 4.3 minutes. These test vectors could cover nearly 87% of all faults. GA reduced the number of test vectors significantly which implies that GA-based test pattern generation is a perfect candidate to replace random test pattern generation for all types of sequential circuits.

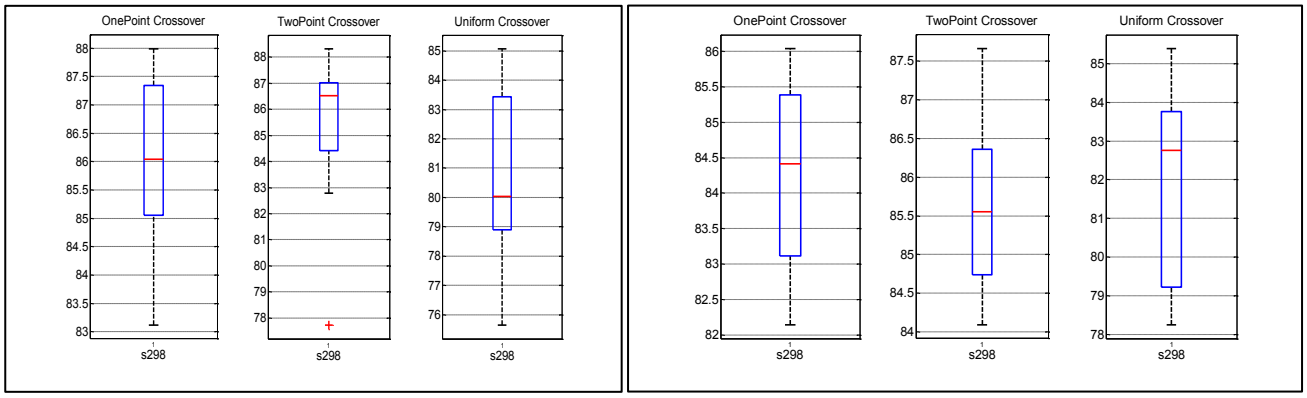

Figure 5.8: Results of s298 circuit using rank selection (Left) and roulette wheel selection (Right). Different crossover schemes were used.

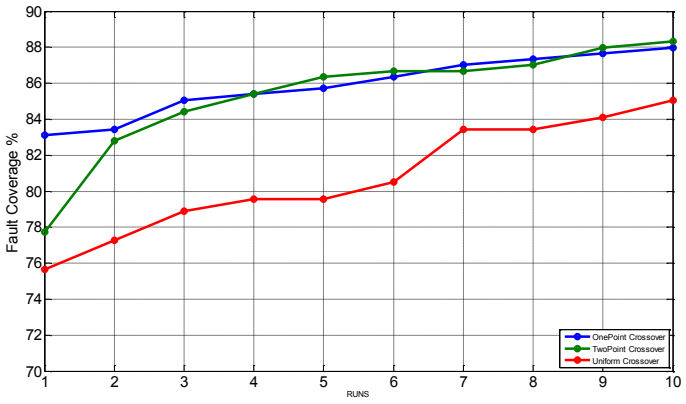

Figure 5.9: Fault Coverage for s298 circuit, rank selection.

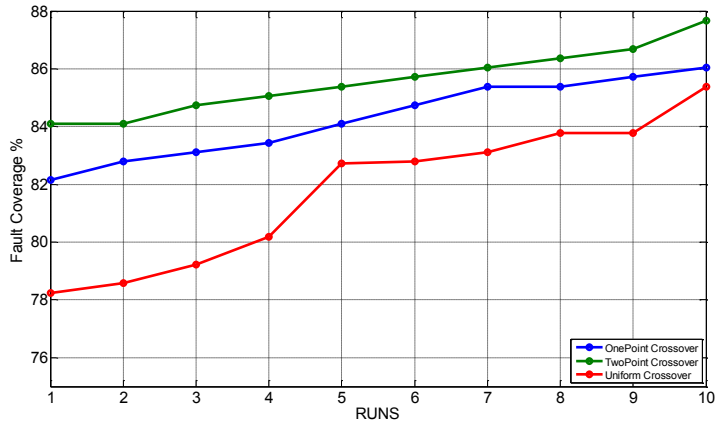

Figure 5.10: Fault coverage for s298 circuit, roulette wheel selection.

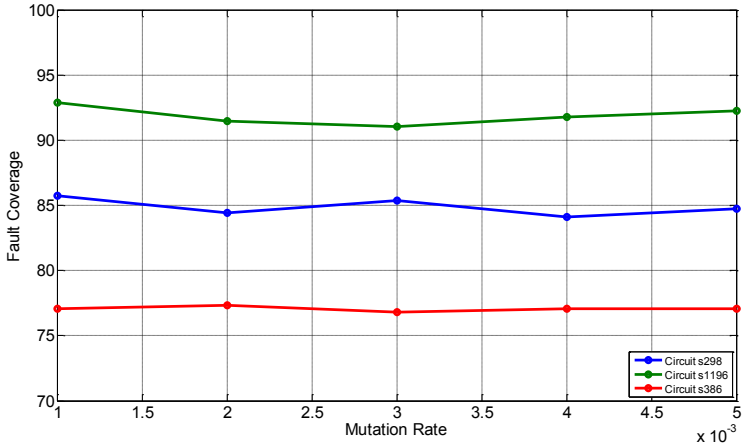

Figure 5.11: Mutation effects on several sequential circuits.

5.4 Particle Swarm Optimization

This section explains the implementation of BPSO in generating test sequences for sequential circuits. It presents an analysis of each parameter and its effects on testing time and fault coverage. In comparison with GA and BDE, BPSO has fewer parameters to optimize to reach the desired solution. The initial population is made up of randomly initialized particles. Particles revise their own velocity and position based on a predefined fitness function of its own and other particles in a population. In other words, particles modify their movement according to their own experience and their neighboring particle experience. Position and velocity are to be modified in each iteration of the PSO algorithm to find the optimum solution. Inertia weight w, is a key role in the process of providing balance between the exploration and exploitation processes. The inertia weight determines the contribution rate of a particle's previous velocity to its velocity at the current time step. The basic PSO, presented by Eberhart and Kennedy in 1995 [42], has no inertia weight. In 1998, Shi and Eberhart [52] first presented the concept of inertia weight by introducing constant inertia weight [53]. Further improvements on the concept of inertia weight have been introduced afterwards.

In binary coding PSO, each binary string represents a particle which is constructed of a series of test vectors. Velocity is defined as the probability of a bit to change from zero to one, or vice versa, which will help a particle to move to another location. The location might be a global optimum solution or another local optimum solution. The search space in BPSO is limited between zero and one. However, the length of a test sequence increases the size of the search space accordingly.

5.4.1 Parametric Analysis

Since the inputs, outputs and all other internal signals are discrete values, the following evolutionary equation, which is the two-binary coded discrete PSO, has been used:

$$v_{ij}(t+1) = w \,.\, v_{ij}(t) + c_1 \,.\, \text{rand}() \left(\text{pBest}_{\text{ij}}(\text{t}) - x_{ij}(t)\right) + c_2 \,.\, \text{rand}() \left(\text{gBest}_{\text{ij}}(t) - x_{ij}(t)\right),$$

$$w = w_0 - \left(w_0 - w_f\right) . {}^{t}/_{T}.$$

The sigmoid function is used to normalize the original velocity to be a value between 1 and 0:

$$\text{sig}\left(v_{ij}(t+1)\right) = \frac{1}{1 + e^{-\left(v_{ij}(t+1)\right)}},$$

$$x_{ij}(t+1) = \begin{cases} 0 & \text{rand} \geq \text{sig}\left(v_{ij}(\text{t}+1)\right) \\ 1 & \text{other} \end{cases},$$

where w is the inertia weight, w_0 is the initial value of the inertia weight, w_f is the final value of the inertia weight, T is the maximum iteration, *rand* is a random number between 0-1 with uniform distribution, and c_1 and c_2 are positive acceleration constants. V_i (v_{i1}, v_{i2}, …, v_{id}) is the velocity of the particle, X_i (x_{i1}, x_{i2},…, x_{id}) is the current particle of the *ith* particle, P_i (p_{i1}, p_{i2}, …, p_{id}) is the best visited position for the *ith* particle and G_i (g_{i1}, g_{i2}, …, g_{id}) is the best position explored so far. The subscripts *i,*and *j* refer to particle number *i* and the particular bit *j* of that particle's velocity, respectively.

Inertia weight is used to balance the global and local search capabilities. It shows the effects of the previous velocity on the new velocity. A large weight facilitates a global search. That is, as velocity becomes larger, the particles move and search in more space. Thus, the ability to explore more regions of the search space increases. Conversely, a smaller weight facilitates a local search which means that the velocity becomes smaller and that benefits the current solution space to find a good solution. The search will be concentrated on a promising area to find a solution. Inertia weight *w*, needs to be well optimized to achieve balance and it is an application-dependent value.

Acceleration coefficients, c_1 and c_2, are better to be well adjusted to quantify the performance relative to the experience and neighbors, respectively. If $c_1 = 0$, then we will have a social-only model which means that a particle does not have its own past performance. If $c_2 = 0$, then we will have a cognition-only model which means that the neighbors' experience is unknown and there is no sharing of information between particles. In this work, several values of c_1 and c_2 in the range 0–4 have been used and we reached the best solution set by using equal values for c_1 and c_2 of 2.

The initial population affects the convergence rate. Generating test sequences randomly might increase the area of search space because of the difference in fitness value between particles. Since there is no selection mechanism in PSO, the population size is equal to the neighborhood size and all neighbors are fully connected with each other which means that the velocity is dynamically adjusted according to the particle's personal best performance achieved so far and the best performance achieved so far by all the other particles. Lastly, the number of iterations depends on the size of a circuit and sequence length. In each iteration, PSO updates a set of

previously generated/evaluated test sequences, called particles, in terms of fitness and velocity. The particles will move to new position if they attain higher fitness value than previous ones.

5.4.2 Results

PSO for ATPG works as follows: an initial population of test sequences is generated randomly and applied to a sequential circuit. Initial velocities are generated in the range of 0-1 for each test sequence in the initial population. Each sequence is updated to fly to another position within a search space. If the fitness of the updated sequence is better than “gBest” or “pBest”, the updated sequence will replace the previous particle position with either “gBest” or “pBest”, and the velocity will be updated as well. When the stopping condition is met, which is either a full fault coverage or maximum iteration number, the set of global/local solutions for the targeted faults are the end results.

The probability of losing a good gene within a sequence is very low because of the comparison/replacement scheme of particles. Each updated particle will move according to its own experience and the whole group experience within the search space. Consequently, the search mechanism is highly directed and guided. Thus, a particle, or particles, will move to areas of possible solutions quickly, efficiently and effectively. Figure 5.12 shows a search space in 2D which has the axes y_1 and y_2, i.e., the place we are going to look for the optimum solution, of a few particles after a few iterations for the sequential circuit s298. Each green square represents a particle at its local solution. A blue circle represents a best solution, or the global optimal solution. The oval shape represents areas of possible global optimal solutions. All particles within these oval shapes have high fitness values. Most of the particles outside these two circles have relatively small fitness values. The length of arrows delineates how much a particle’s fitness value has improved.

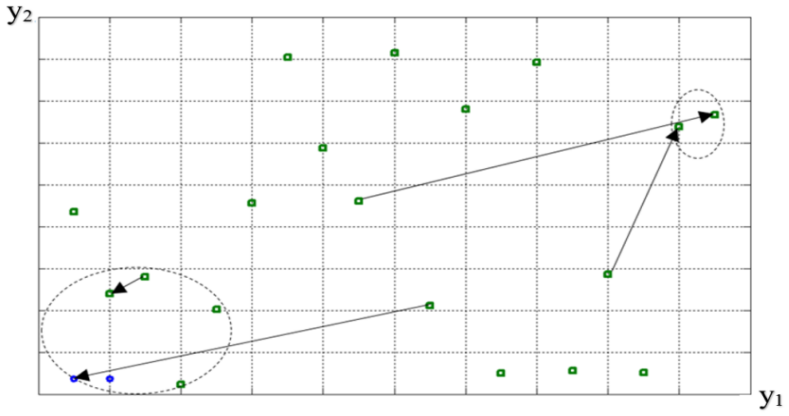

Figure 5.12: A part of the search space for the s298 circuit

The overall results of using PSO in ATPG are very encouraging and promising. Faults coverage was high in most of the tested sequential circuits and testing time was advantageous over other evolutionary algorithms. Figure 5.13 shows the fault coverage of the s298 circuit by implementing PSO. The variation in results is low and the difference between the highest detected fault and the lowest detected fault is only a few faults which implies that the reliability of this method is high. Figure 5.14 shows fault coverage for a larger circuit. The fault coverage obtained by PSO for all CUTs in this work is high and all easy-to-detected faults were covered in a short time period with the fewest possible test vectors. PSO detects 86.3% of the faults for the s35932 circuit in 1 minute 14 seconds, which is considered a major improvement and advantage compared to other ATPG algorithms. Since PSO relies on sequence manipulation, increasing the sequence length results in increasing the fault simulation time. As an example, the sequential s349 circuit has 9 inputs, PSO-based test generator detects 98% of the total faults in 0.1 seconds with a sequence length of 9. As the sequence length was doubled to 18, the fault coverage did not improve and it stayed at 98% while the fault simulation time increased to 0.445 seconds. Sequence length must be kept as low as possible because updating the bit string will double the testing time a factor of 2 or even higher. However, sequence length needs to be well determined to allow the circuit to arrive at the targeted state and thus the fault effects arrive at either a PPO or PO. Figure 5.15 shows the effects of increasing sequence length on testing time and the fault coverage on the circuit s1196. The sequence length was doubled while keeping other parameters at their lower values. On the other hand, reducing the sequence length to be equal to the PIs will reduce the fault simulation time by approximately 2%. Initial population is another key parameter that has a direct impact on testing time which is shown in Figure 5.16. However, the effects on testing time that come from initial population is lower than the effects that come from sequence length and number of iterations.

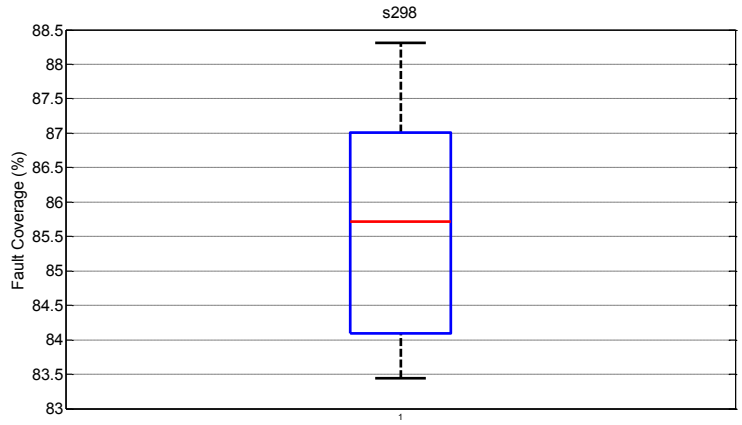

Figure 5.13: Fault coverage of the s298 circuit using the PSO-based test generator, max detected fault = 272, min detected fault = 257.

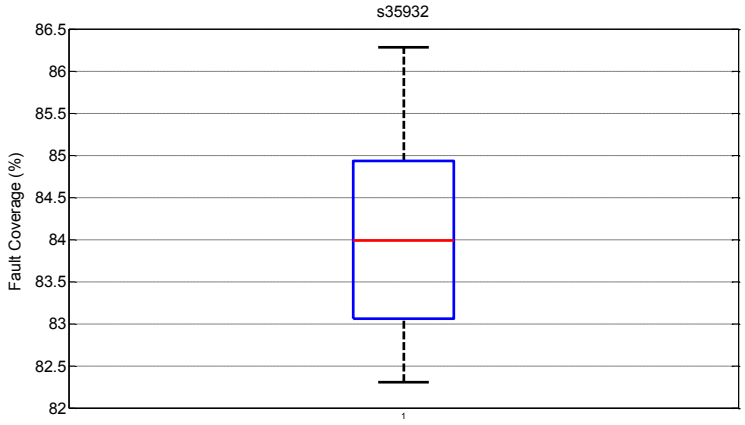

Figure 5.14: Fault coverage of the s35932 circuit using the PSO-based test generator, max detected fault = 33730, min detected fault = 32178.

It is recommended to start with a low iteration value and increase it proportionally with the size of a circuit to avoid unnecessary computational overhead. The doubled sequential depth value was used as an initial iteration value, and then it was modified as needed. In fact, setting the sequential depth value of each circuit as a progressive limit and doubling the sequence length shows high, but not the highest, fault coverage results. Figure 5.17 shows the effects of increasing the number of iteration of fault coverage and testing time

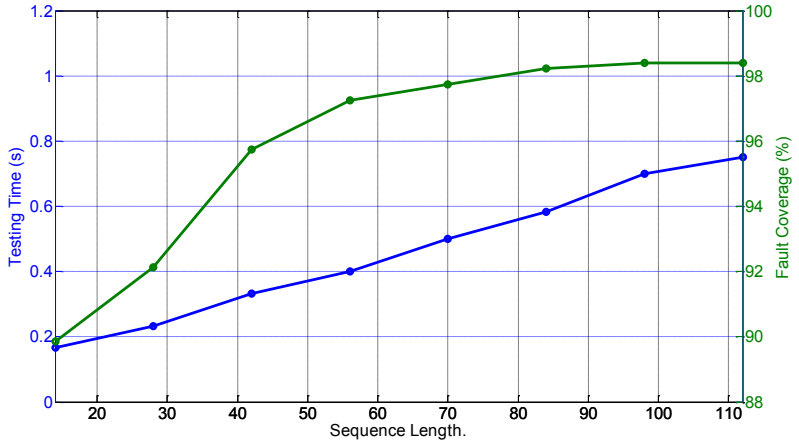

Figure 5.15: Sequence length effects on testing time and fault coverage for the s1196 circuit.

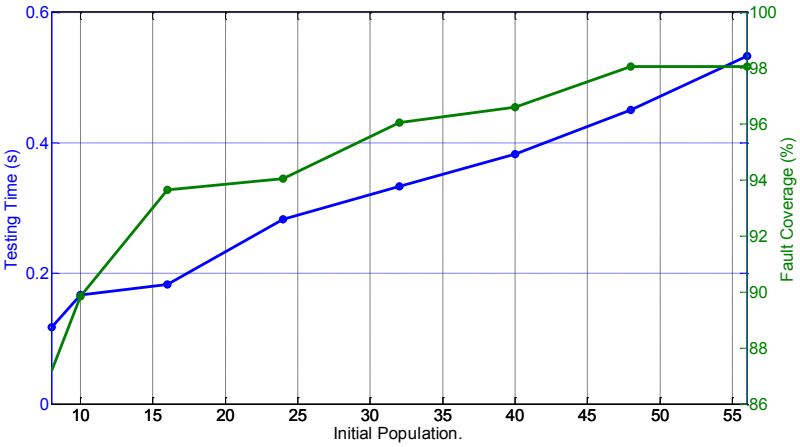

Figure 5.16: Initial population effects on testing time and fault coverage for the s1196 circuit.

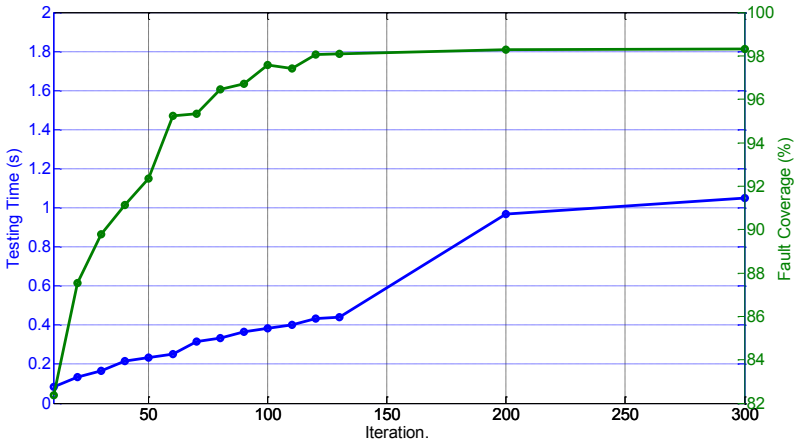

Figure 5.17: Number of iteration effects on testing time and fault coverage for the s1196 circuit.

5.5 Differential Evolution

Binary coded differential evolution (BDE) has been implemented in several applications [66-69], and [71]. BDE in ATPG for sequential circuits aims to limit the search space in the range [0-1]. It starts with a randomly generated initial population and each individual is evaluated through a fitness function. A new population is then generated through the process of crossover and mutation according to a mutation strategy. The BDE algorithm used in this work is explained in Figure 5.18.

```
Begin
      Generate a random population of solutions;
      Calculate the fitness of the initial population
      Set control parameter values;
      Repeat
      {
         Select a parent vector.
         Select three individuals for mutation.
         Produce one offspring.
         Crossover the mutant vector with the parent vector.
         Evaluate the resultant vector.
         If offspring vector is better than parent vector
             Replace parent vector with the offspring
             Else
             Retain parent

         Stop if the stopping criteria is met.
      }
End
```

Figure 5.18: Binary DE algorithm (BDE).

The binary mutation strategy is driven from the original differential mutation in addition to the mutation operator from GA. Figure 4.11 shows the original differential mutation and we treat each variable as follows:

- X_3 is the parent vector that has the highest fitness value in the current generation.
- X_1, X_2 are chosen randomly from the current population.
- F is the scale factor which has an effect on the difference between the particles (X_1, X_2).

The resultant mutant vector comes from the following:

$$\text{Mutant} = X_3 + F\ (X_1 - X_2)$$

The binary mutation strategy is slightly different since there are eight combinations of three binary variables. The mutation operation is performed as follows [81]:

- If X_3 equals zero and X_1 equals X_2, then the result of mutation equals zero.
- If X_3 equals one and X_1 equals X_2, then the result of mutation equals one.
- If X_3 equals zero and X_1 is different from X_2, then X_3 will mutate to 1 - X_3 with some probability.

- If X_3 equals one and X_1 is different from X_2, then X_3 will mutate to 1 - X_3 with some probability.

Table 5.3 summarizes the four mutation rules.

X_3	X_2	X_1	Results
0	0	0	0
0	0	1	1- X_3 with some probability
0	1	0	1- X_3 with some probability
0	1	1	0
1	0	0	1
1	0	1	1- X_3 with some probability
1	1	0	1- X_3 with some probability
1	1	1	1

Table 5.3: Mutation strategy.

The crossover operator is used to build the trial vector from the mutant vector and the candidate vector. It is used to produce a fitter offspring. There are two main types of crossover in DE: binomial and exponential. Binomial is similar to uniform crossover used in GA where at least one component is taken from the mutant vector. This type of crossover is neglected in this work. On the other hand, exponential crossover is similar to two-point crossover used in GA where the first-cut point is randomly selected and the second-cut point is determined such that L consecutive components are taken from the mutant vector.

In this work, mutation and crossover is repeated for all NP members of the current generation. Next, we evaluate the fitness function for each of the trial vectors, test sequences, and compare each vector's fitness value with the fitness value of the candidate from the previous generation. If the trial fitness function value is higher than that produced by the candidate vector, then the trial vector replaces the candidate vector, otherwise the candidate vector will survive to the next generation.

5.5.1 Parametric Analysis

In binary DE, there are a few parameters to analyze, including NP, CR and PF. It should be noticed that NP should be at least four individuals because three test sequences are needed to create a mutant vector and one test sequence is needed for the crossover operation. Increasing population size, NP, or test sequence length, leads to increasing the search space which leads to more exploration and the probability of finding a global optimum solution increase as a result. However, testing time increases in DE due to increasing the search space. In BDE, search space

is limited in the range 0-1. However, since the sequential circuits need a series of test sequences of different lengths for fault detection, the search space increases accordingly.

As mentioned earlier, three test sequences are needed for mutation and they can be chosen randomly or one test sequence can be one of the best test sequences in a population. Choosing one of the best test sequences for mutation helps strong genes to survive to next generations, while randomness in choosing the test sequences increases the probability of losing strong genes because weak test sequences might compete together. This work always assigned the best individual of the current population to be one of the three test sequences to help guide the search process towards areas of possible solutions. Choosing a suitable mutation strategy is what determines the effectiveness of BDE in ATPG. Several mutation strategies have been discussed in chapter 3. The adopted strategy in this work has shown good results with the literature, beside its simple implementation [81].

A predefined rate *PF*, was added to the mutation strategy to determine which bit will be mutated in some cases. This rate was kept random to allow for diversity. It is important to not allow copying of a test sequence by mutation or crossover. This case has rarely happened after crossing the mutant vector with the target vector where the resultant vector was the exact same previous target vector.

The fitness function is a major parameter in all evolutionary algorithms. It aims to measure the quality of a candidate vector to detect faults or propagate fault effects to flip-flops. The problem of measuring the fitness of a candidate vector in ATPG is that it is not a straightforward calculation; each test vector/test sequence must be fully simulated before calculating its fitness. The testing time is the total simulation time plus calculation time. The fitness time used in this work is similar in all the sections of this paper. However, it is recommended to modify the fitness function to reach higher coverage.

5.5.2 Results

BDE was able to detect most of the easy-to-detect stuck-at faults. The overall performance was not as expected. The testing time was noticeably high, especially in large circuits such as the s35932 circuit with 6.6 minutes and the fault coverage was not maximized. On the other hand, the fault coverage in small-scaled sequential circuits was relatively high as demonstrated with the s298 circuit in Figure 5.19. In this work, the testing time, which is proportional to the sequence length, was mostly consumed by the mutation strategy. The adapted mutation strategy

increased the computational overhead with larger circuits while the computational overhead remained within the expected range with small-scaled circuits.

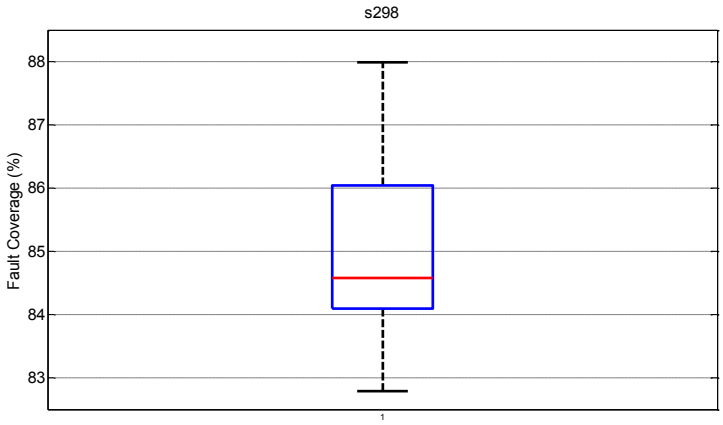

Figure 5.19: Fault coverage of the s298 circuit using the DE-based test generator. Maximum faults detected = 271 (87.987%), minimum faults detected = 255 (82.792%).

The implementation of BDE is easier than other evolutionary algorithms with fewer parameters to set. Setting parameter values appropriately was not a difficult issue. It was better to keep the *CR* value as high as possible and make the scale factor *F* close to 0.5, if needed. The challenge comes from having a binary mutation strategy that is able to reduce the testing time.

5.6 Conclusion

This chapter focused on representing a complete analysis of several optimization algorithms on ATPG for sequential circuits. All the three test generators were implemented around the sequential circuit fault simulator HOPE [59] in the C language. Test results were achieved assuming that the initial state of all flip-flops were known and set to zero. The results showed the best fault coverage obtained to illustrate the capability of an optimization algorithm to explore a search space. Graphs of fault coverage of several sequential circuits showing the result variations are in Appendices A-D. Table 5.4 shows the number of faults detected and the number of test vectors for all three optimization algorithms. PSO was able to detect more faults than the other algorithms in most of the sequential circuits, while GA and DE have similar results. Testing time, which was reported in a previous chapter, was lower in PSO than in both GA and DE.

Circuit	Total number of faults	GA		PSO		DE	
		Fault Detected	Number of Vectors	Fault Detected	Number of Vectors	Fault Detected	Number of Vectors
S298	308	272	144	272	142	271	154
S344	342	337	91	335	90	333	89
S349	350	340	90	343	90	338	89
S510	564	564	263	564	239	564	317
S641	467	404	134	407	138	400	142
S713	581	475	119	480	118	474	119
S1196	1242	1188	362	1220	348	1134	567
S35932	39094	34011	197	33730	201	33230	328

Table 5.4: Results for all three optimization algorithms.

Testing time has a significant and direct impact on the final product profits, which is a major concern in the industry. It must be reduced to a minimum without affecting any other factors, and it takes the highest priority over other factors. PSO has shown better performance in the shortest time. Fault coverage obtained depends on the nature of faults, fault simulator and the ATPG algorithm. The optimization algorithms, in general, show high effectiveness in detecting all testable faults in low testing time with a relatively small number of test vectors. Figure 5.20 shows the relationship between fault detection and test vector generation over time *t*.

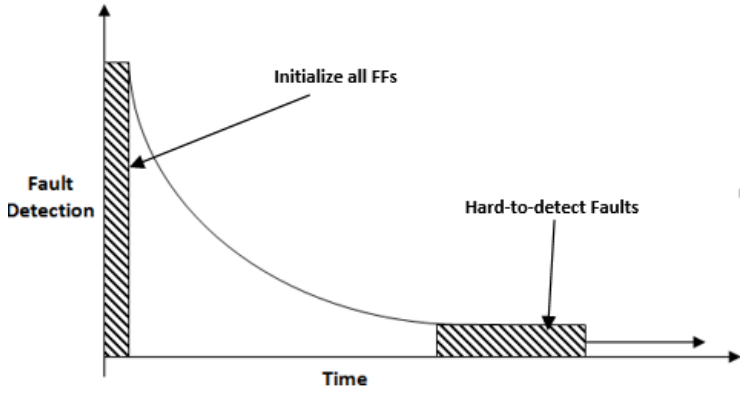

Figure 5.20: Fault detection as a function of time.

The first test vectors/sequences applied to a sequential circuit are supposed to initialize all flip-flops to a known state. In this work, all flip-flops are initialized before by a control signal and no test sequences were generated for initialization purposes. The fault detection starts at its maximum rate, then the detection rate decays over time because of the fault dropping, which means any detected fault will be dropped out of the fault list. Fault detection will stop at time *t*

once it reaches the hard-to-detect faults region. Optimization algorithms are the best candidates to operate in easy-to-detect faults regions. However, the performance was weak in hard-to-detect faults regions which require non-simulation-based algorithms to effectively search for test sequences.

PSO shows superiority in performance over other optimization algorithms, as seen in Figure 5.21, and it is expected that SI algorithms will show similar results because of similarities in the search mechanism which is highly guided since it lets a particle rely on its own experience as well as the whole group experience. In contrast, the GA search mechanism lets individuals move to optimum solutions as a group which leads to incremental increases in testing time which concludes that GA is slower than PSO. In GA, the fitness value of the whole population is necessary to keep the fitness value of the successive generations increasing. In this work, a condition was added to the initial generation to measure its overall fitness before proceeding to the next generation. Although DE is nearly similar to GA, DE would perform better with a suitable mutation strategy. DE parameters do not require significant modification, which is necessary in GA, and is considered an advantage of DE over GA. Lastly, this work emphasizes the substantial performance of PSO to generate test vectors/sequences for synchronous sequential circuits.

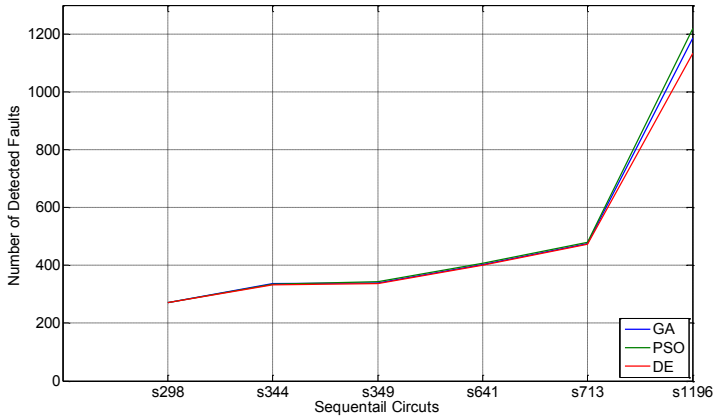

Figure 5.21: Comparison between several circuits of optimization algorithms performance.

Chapter 6: Conclusions and Future Work

This thesis facilitated developing a solution that can reduce the extensive efforts to find effective solutions to generate test sequences for sequential circuits. Optimization algorithms offer attractive results in generating test vectors/sequences for sequential circuits by optimizing the search space and guiding the searching process to effectively search for solutions in short time periods. Although this research obtained high fault coverage for most of the sequential circuits by carefully analyzing and modifying the algorithms' parameters, rather methods will not completely solve ATPG for sequential circuits because of the continuous advancement being made in technology. However, optimization algorithms, especially PSO, show a noteworthy indication that it has a significant capability to search for the optimum test sequence set. The conclusion of this thesis emphasizes the high efficiency of the PSO algorithm over other evolutionary algorithms to solve ATPG for sequential circuits. Several recommendations were mentioned in the earlier chapter to optimize the results of PSO in generating test vectors which can be carried out in future work. The GA has a slower searching mechanism and its parameters require a lot of alternation. We found that one-point crossover and two-point crossover increased the fault coverage for most of the circuits, while uniform crossover caused the searching process to be more randomized. The mutation operator in GA had little effect on the overall results and it is recommended to choose a low mutation value. One advantage of using GA in ATPG is that it gives more controllability of the searching process since it has many parameters. As an example, the mutation operator may allow searching, if implemented properly, to move from one local optimum to another local optimum to explore more search space. DE must have a proper binary mutation strategy that does not add significant computation overhead when it operates with long test sequences in large sequential circuits. The mutation strategy needs to have a mutant rate that excludes a part of the test sequence from being mutated to reduce the testing time. Otherwise, the testing time will increase rapidly as the length of the test sequence increases.

Evolutionary algorithms (EA's) show an increase in testing time for all sequential circuits while swarm intelligence (SI) shows an optimized testing time and higher fault coverage due to the nature of the searching process. The results of PSO, as a representative of SI, in ATPG raise the significance of implementing other SI-based algorithms in ATPG, such as Ant Colony.

6.1 Future Work

Based on the results of this thesis, some of the studies and experiments that can be performed in the future are summarized in the following:

- Use a Hybrid PSO to improve the accuracy and increase the fitness of the early generations. I suggest implementing GA-PSO to take advantage of the selection operator to eliminate weak particles in the early generations.
- Implement parallelism in PSO to increase convergence rate. Two groups of faults can be detected at the same time. This will add complexity to the PSO implementation but it will significantly reduce the testing time.
- Improve DE with another improved mutation strategy. Always consider the test sequence length since it may increase the computational overhead and reduce the convergence rate.
- Since the parameters of the GA require a large amount of modification, it is crucial to find optimum values for all parameters that can work with the majority of sequential circuits. This could be achieved through either clustering or categorizing sequential circuits based on specific standards and then match each cluster/category with each set of parameter values.
- Evaluate the fitness functions to see their effectiveness in measuring the quality of a test sequence should be performed by future researchers because of the major impact of the fitness functions on the quality of the testing algorithms.

Bibliography

[1] A. Grochowslj, D. Bhauacharya, T. R. Viswanathan, and K. Laker. "Integrated Circuit Testing for Quality Assurance in Manufacturing: History, Current Status and Future Trends", *IEEE Trans. Circ. Syst. II*. vol. 44, no. 8, pp. 610-633, Aug. 1997.

[2] G. E. Moore, "Progress in Digital Integrated Electronics", In *Proc. Of the Int'l. Electron Device Meeting*, pp. 11-13. Moore's Law, Dec. 1975.

[3] P. Mazumder and E. M. Rudnick, "Genetic Algorithms for VLSI Design, Layout and Test Automation", Prentice Hall, New Jersey, 1999.

[4] T. Niermann and J. Patel, "HITEC: A test generation package for sequential circuits", *Proc. European Conf. Design Automation (EDAC),* pp. 214-218, Feb. 1991.

[5] I. Hamzaoglu and J. Patel, "Deterministic test pattern generation techniques for sequential circuits", *Proc. Int'l Conf. Computer-Aided Design*, pp. 538-543, 2000.

[6] I. Hamzaoglu and J. H. Pate1, "New Techniques for Deterministic Test Pattern Generation", *Proc. 16th, IEEE VLSI Test Symp.,* pp. 446-452, 1998.

[7] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", *IEEE Trans. on Computers*, vol. C-30, no. 3, pp. 21-222, March 1981.

[8] A. Dargelas, C. Gauthron and Y. Bertrand, "MOSAIC: a multiple-strategy oriented sequential ATPG for integrated circuits", *Proc. of European Design & Test Conf.*, pp. 29-36, France, March 1997.

[9] A. Gosh, S. Devadas, and A. R. Newton, "Test generation and verification for highly sequential circuits", *IEEE Trans. Computer Aided Design*, vol. 10, no. 5, pp. 652-667. May 1991.

[10] N. Gouders and R. Kaibel "Test generation techniques for sequential circuits", *Proc. IEEE VLSI Test Symposium,* pp. 221-226, 1991.

[11] D. H. Lee and S. M. Reddy, "A New Test Generation Method for Sequential Circuits", *Proc. Int. Conf. on Computer-Aided Design*, pp. 446-449, November 1991.

[12] S. Kang and S. A. Szygenda, "The simulation automation system (SAS); concepts, implementation, and results", *IEEE Transactions on VLSI Systems*, Vol. 2, pp. 89-99, March 1994.

[13] S. Seshu and D. N. Freeman, "The diagnosis of asynchronous sequential switching systems", *IRE Trans. Electronic Computing*, vol. 11, pp. 459-465, Aug. 1962.

[14] V.D. Agrawal, K.T. Cheng, P. Agrawal, "A directed search method for test generation using a concurrent simulator," *IEEE trans. Computer-Aided Design*, vol. 8, no. 2, pp. 131-138, Feb. 1989.

[15] D. Saab, Y. Saab, and J. Abraham, "CRIS: A test cultivation program for sequential VLSI circuits", *Proc. Int. Conf. Computer-Aided Design,* pp. 216-219, Nov. 1992.

[16] D. G. Saab, Y. G. Saab, and J. Abraham, "Automatic test vector cultivation for sequential VLSI circuits using genetic algorithms", *IEEE Trans. Computer-Aided Design,* vol. 15, pp. 1278–1285, Oct. 1996.

[17] P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "An Automatic Test Pattern Generator for Large Sequential Circuits Based on Genetic Algorithms", *Proc. Int. Test Conf.,* pp. 240-249, October 1994.

[18] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, and R. Mosca, "Advanced techniques for GA-based sequential ATPGs", *in Proc. Eur. Design Test Conf*., pp. 375–379, Mar. 1996.

[19] E. Rudnick, J. Patel, G. Greenstein, and T. Niermann, "Sequential Circuit Test Generation in a Genetic Algorithm Framework", In *Proc. 31st Annual Design Automation Conference,* pp. 698–704, 1994.

[20] E. M. Rudnick, J. H. Patel, G. S. Greestein, and T. M. Niermann, "A genetic algorithm framework for test generation", *IEEE Trans. Computer-Aided Design*, vol. 16, n. 9, pp. 1034-1044, Sep. 1997.

[21] H. Yanli, Z. Chunhui, and L. Yanping "A new method of test generation for sequential circuits", IEEE *in Int. Conf. on Communications, Circuits and Systems*, pp. 2181-2185, June 2006.

[22] J. Kennedy and R. C. Eberhart, "A discrete binary version of the particle swarm algorithm", *IEEE International Conference on Systems, Man, and Cybernetics,* vol. 5, pp. 4104-4108, Oct 1997.

[23] W. Xu and Y. Gu, "Study on automatic test generation of digital circuits using particle swarm optimization", pp 324-328, Oct. 2011.

[24] X. Chuanpei, L. Zhi, and M. Wei, "Study of differential evolution on ATPG", In *2006 Int. Conf. on Communication, Circuits and Systems Proc.*, vol. 3, pp. 2084-2087, Guilin, 2006.

[25] L. T. Wang, C. Wu, and X. Wen, "VLSI Test Principles and Architectures: Design for Testability", Morgan Kaufmann Publishers Inc., USA, 2006.

[26] M. L. Bushnell, V. D. Agrawal, "Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI circuits", Kluwer Academic Publishers., 2000.

[27] S. Seshu, "On an Improved Diagnosis Program", *IEEE Transactions on Electronic Computers*, vol. EC-14, no 1, pp 76-79, Feb 1965.

[28] N. K. Jha and S. Gupta, "Testing of Digital Systems", Cambridge University Press, NY, USA, 2002.

[29] D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits", IEEE Transactions on Computers, vol. C-21, no.5, pp. 464-471, May. 1972.

[30] H. Y. Chang, S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "Comparison of parallel and deductive fault simulation methods", *IEEE Trans. Comp.*, vol. C-23, pp. 1132-1138, Nov. 1974.

[31] E. G. Ulrich and T. Baker, "Concurrent Simulation of nearly Identical Digital Networks", Computer, vol. 7, pp. 39-44, Apr 1974.

[32] D. G. Saab, "Parallel concurrent fault simulation", *IEEE Trans. VLSI Syst.*, vol. 1, pp. 356–363, Sept. 1993.

[33] W. T. Cheng and M. L. Yu, "Differential Fault Simulation for Sequential Circuits", Journal of Electronic Testing: Theory and Applications, vol. 1, no. 1, pp. 7-13, Feb. 1990.

[34] W. T. Cheng and M. L. Yu, "Differential Fault Simulation - A Fast Method Using Minimal Memory", *Design Automation Conf.*, pp. 424-428, 1989.

[35] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning", (Addison-Wesley, Reading, Massachusetts, 1989).

[36] A. Engelbrecht, "Computational intelligence: an introduction", Chichester, England; Hoboken, NJ: John Wiley & Sons, 2007.

[37] D. Whitley, "The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best", In J. David Schaffer, editor, *Proc. of the 3rd Int. Conf. on Genetic Algorithms,* pages 116-121, San Mateo, CA, 1989, Morgan Kaufmann Publisher.

[38] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms", in Foundation of Genetic Algorithms, G. Rawlins (ed.), Morgan Kaufman, pp. 69-93, San Mateo, CA, 1991.

[39] C. R. Reeves and J. E. Rome, "Genetic Algorithms Principles and Perspectives", Kluwer Academic Publishers. Netherlands, 2003.

[40] A. Mukhopadhyay, U. Maulik, and S. Bandyopadhyay, "Multiobjective Genetic Algorithms for Clustering", Springer Publishing Company, Sep 2011,

[41] R. Eberhart, and J. Kennedy, "A New Optimizer Using Particles Swarm Theory", Proc. Sixth International Symposium on Micro Machine and Human Science (Nagoya, Japan), IEEE Service Center, Piscataway, NJ, pp. 39-43, 1995.

[42] J. Kennedy, and R. Eberhart, "Particle Swarm Optimization", *IEEE International Conference on Neural Networks*, IEEE Service Center, Piscataway, NJ, IV, pp. 1942-1948, Perth, Australia, 1995.

[43] J. Kennedy and R. Eberhart. Swarm Intelligence. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2001.

[44] R. Eberhart and Y. Shi, "Particle swarm optimization: developments, applications and resources", *Proceedings of IEEE Congress on Evolutionary Computation*, vol. 1, pp.27-30, May 2001.

[45] R. Eberhart and Y. Shi, "Comparison between genetic algorithms and particle swarm optimization", *In Proc. Of 7th Int. Conf. on Evolutionary Programming VII,* pp. 611-616, March 1998.

[46] M. A. Khanesar, "A Novel Binary Particle Swarm Optimization", *2007 Mediterranean Conference on Control and Automation*, pp. 1-6, Athens, Greece, 2007.

[47] R. Storn and K. Price, "Differential evolution a simple and efficient heuristic for global optimization over continuous spaces", *J. Global Optimization*, vol. 11, no. 4, pp. 341-359, 1997.

[48] K. Price, "Differential evolution: A fast and simple numerical optimizer", *In Proc. Biennial Conf. North Amer. Fuzzy Info. Processing Soc.*, pp. 524–527, Berkeley, CA, USA, 1996.

[49] B. Hegery, C. C. Hung, and K. Kasprak, "A comparative study on differential evolution and genetic algorithms for some combinatorial problems", *in Proc. of 8th Mexican International Conference on Artificial Intelligence*, 2009.

[50] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits", *Int. Symposium on circuits and systems*, pp. 1929-1934, May 1989.

[51] J. W. Tukey, "Exploratory Data Analysis", Addison-Wilson Publishing Company, January, 1977.

[52] Y. Shi and R. Eberhart, “A modified particle swarm optimizer”, In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence, the 1998 IEEE International Conference*, pp. 69–73, 2002.

[53] J. C. Bansal, P. K. Singh, M. Saraswat, A. Verma, S. S. Jadon, and A. Abraham, “Inertia Weight strategies in Particle Swarm Optimization”, *in Proc. IEEE Nature and Biologically Inspired Computing (NaBIC)*, Salamanca, pp. 19-21 Oct. 2011.

[54] D. G. Saab, Y. G. Saab, and J. A. Abraham, “Iterative [simulation-based genetics + deterministic techniques] = complete ATPG”, *Proc. Int. Conf. Computer-Aided Design,* pp. 40-43, 1994.

[55] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, “Alternating Strategies for Sequential Circuit ATPG”, *Proc. European Design and Test Conf.,* pp. 368-374, March 1996.

[56] E. M. Rudnick, and J. H. Patel, “Combining deterministic and genetic approaches for sequential circuit test generation”, *Proc. Design Automation Conf.,* pp. 183-188, June 1995.

[57] X. Lin, I. Pomeranz, and S. M. Reddy, “MIX: A Test Generation System for Synchronous Sequential Circuits”, *Proc. Int. Conf. on VLSI Design,* pp. 456-463, January 1998.

[58] X. Lin, I. Pomeranz, and S. M. Reddy, “Techniques for Improving the efficiency of Sequential Circuit Test Generation”, *Proc. Int. Conf. on Computer-Aided Design,* pp. 147-151, November 1999.

[59] 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,* vol. 15, no. 9, pp. 1048–1058, 1996.

[60] T. M. Niermann, W.-T. Cheng, and J. H. Patel, “PROOFS: A fast, memory efficient sequential circuit fault simulator”, in *Proc. Design Automation Conf.,* pp. 535-540, June 1990.

[61] A. E. Eiben and J. E. Smith, “Introduction to Evolutionary Computing”, New York, Springer, 2003.

[62] Y. Kaya, M. Uyar, and R. Tekdn, “A Novel Crossover Operator for Genetic Algorithms: Ring Crossover”, *Computing research repository, CORR*, vol. abs/1105.0, 2011.

[63] J. Grefenstette, “Optimization of control parameters for genetic algorithms”, *IEEE Transactions on Systems, Man and Cybernetics,* Vol. 16, No. 1, 122–128, 1986.

[64] J. Holland, "Adaption in natural and artificial systems. An introduction analysis with applications to biology, control, and artificial intelligence", Univ. of Michigan, Ann Arbor, 1975.

[65] B. F. Cockburn and J. Han, “Review of Classical Sequential Logic Design,” www.eclass.srv.ualberta.ca/portal/, 2013.

[66] T. Gong and A. Tuson, “Differential evolution for binary encoding”, *In soft computing in industrial applications,* pp. 251-262, Springer, 2007.

[67] X. He and L. Han, “A novel binary differential evolution algorithm based on artificial immune system”, *In Proc. of 2007 IEEE congress on evolutionary computation*, *IEEE*, pp. 2267-2272, 2007.

[68] Y. Chen, W. Xie, and X. Zou, “A binary differential evolution algorithm learning from explored solutions”, *Neurocomputing*, vol. 149, 1038-1047, Feb. 2015.

[69] E. Andries and P. Gary, “Binary Differential Evolution Strategies”, *In Proc. of 2007 IEEE Congress on Evolutionary Computation,* IEEE, pp.1942-1947. 2007.

[70] S. Das and P. N. Suganthan, “Differential Evolution: A Survey of the State-of-the-Art”, *IEEE Trans. On Evolutionary Comp*. vol. 15, pp. 4-31, Feb. 2011.

[71] J. Krause and H. S. Lopes, “A comparison of differential evolution algorithm with binary and continuous encoding for the MKP”, *In Proc. 2013 BRICS Congress on Computational Intelligence. IEEE Computer Society*, pp. 381–387, 2013.

[72] J. Krause, R. Parapinelli, and H. Lopes, “Proposta de um algoritmo inspirado em Evolução Diferencial aplicado ao Problema Multidimensional da Mochila”, in Anais do Encontro Nacional de Inteligencia Artificial. Curitiba, PR: SBC, Oct. 2012.

[73] A. Engelbrecht, N. Franken, and G. Pampara, “Binary differential evolution”, *In IEEE Congress on Evolutionary Computation,* pp. 1873-1879, 2006.

[74] I. Fister, X-S. Yang, J. Brest, and D. Fister, "A Brief review of Nature-Inspired Algorithms for Optimization”, *Elektrotrhniski, Vestnik,* vol. 80(3), pp. 116-122, 2013.

[75] A. Hassanien, and E. Emary, “Swarm Intelligence, Principle, Advances, and Appliactions”, CRC Press, 2015.

[76] A. Marandi, F. Afshinmanesh, M. Shahabadi, and F. Bahrami, "Boolean Particle Swarm Optimization and Its Application to the Design of a Dual-Band Dual-Polarized Planar Antenna", *IEEE Congress on Evolutionary Computation*, pp. 3212-3218, Canada, July 2006.

[77] N. Franken and A. Engelbrecht, "Particle swarm optimization approaches to coevolve strategies for the iterated prisoner's dilemma", *IEEE Trans. on Evolutionary Computation*, pp.562-579, 2005.

[78] C. Zhang and H. Hu, "Using PSO algorithm to evolve an optimum input subset for a SVM in time series forecasting", *IEEE International Conference on Systems, Man and Cybernetics,* pp. 3793-3796, 2005.

[79] D. Dawar, "Evolutionary Computation and Applications," www.slideshare.net/ddawar/evolutionary-computation-andapplications , 2015.

[80] A. Moraglio, "The Geometry of Evolutionary Algorithms: Unification and Theory-Laden Design," http://www.slideshare.net/AlbertoMoraglio/cec-2013-tutorial , June 2013.

[81] C. Deng, C. Ling, Y. Ynag, B. Zhao, and Haizhang, "Binary differential evolution algorithm with new mutation operator," *In Int. Conf. on Intelligent Computing and Intelligent Systems*, pp. 498-501, Oct. 2010.

[82] H. Bersini, M. Doringo, S. Langerman, G. Seront, and L. Gambardella, "Result of the first international contest on evolutionary optimization (1st ICEO)," In IEEE Int. Conf. on Evolutionary Computation (ICEC'96), pp. 611-615, Japan, May 1996.

Appendix A

Random Test Generator

The following graphs show the performance of adapting random test generators in ATPG. It is clear that the number of test vectors will easily exceed one million to obtain a reasonable fault coverage value. Hence, this type of test generator is inefficient and unreliable.

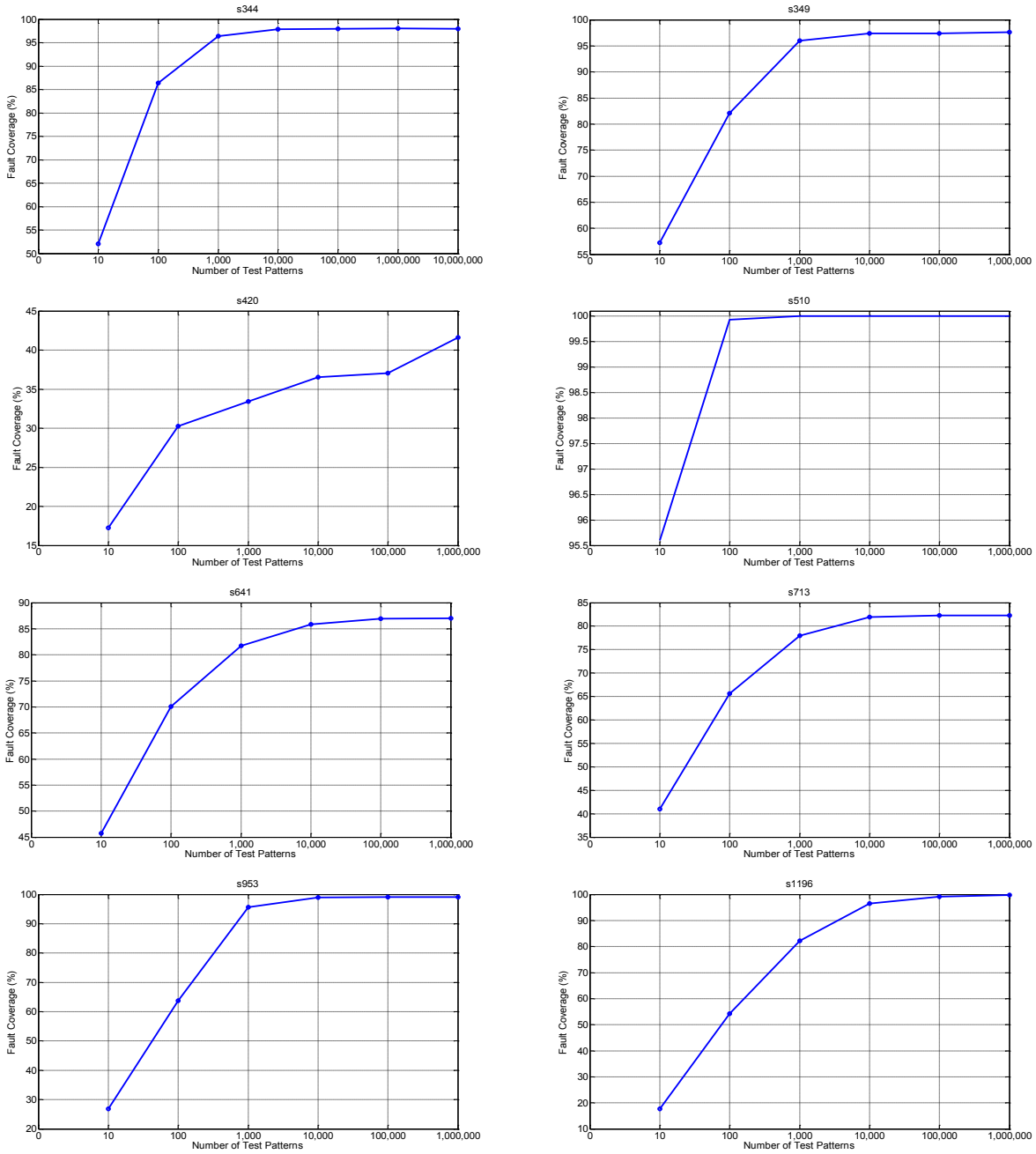

Figure A.1: Fault coverage (Average) for several sequential circuits. (a) Min = 39.5%, Max = 98.5%. (b) Min = 50%, Max = 98%. (c) Min = 6.3%, Max = 42.1%. (d) Min = 40.2%, Max = 100%. (e) Min = 37%, Max = 86.9%. (f) Min = 35.6%, Max = 82.3%. (g) Min = 23.7%, Max = 99.1%. (h) Min = 10%, Max = 99.8%.

Appendix B

GA-based Test Generator

The following graphs represent the performance of GA-based test generators in ATPG. Each graph shows the fault coverage of one circuit with three different crossover schemes and rank selection operator. The boxplots help to show the data vibrations of 10 runs of a test. Several outliers occurred because of the effects of losing good genes on the earlier generations.

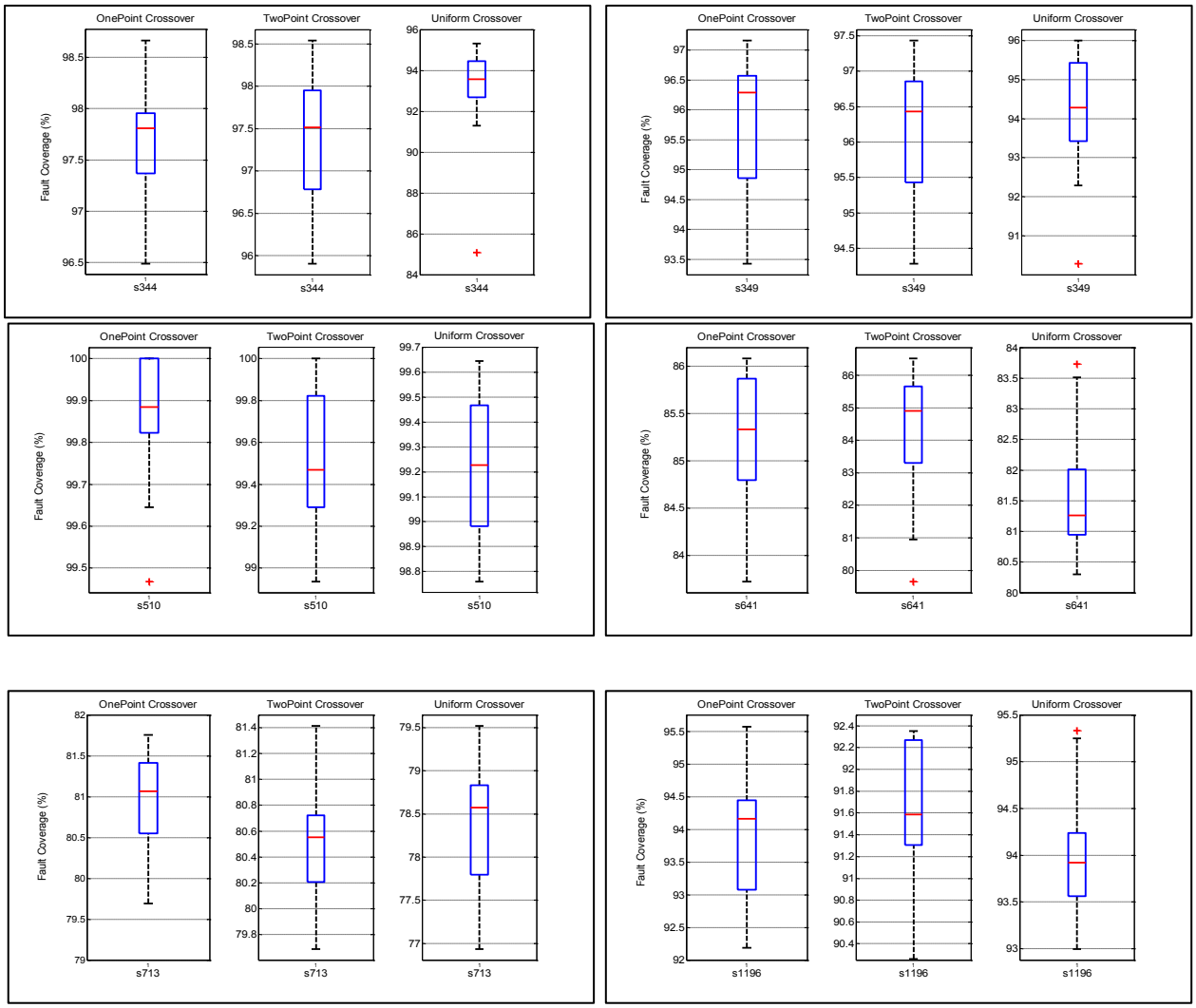

Figure B.1: Fault coverage for several sequential circuits using GA-based test generator – rank selection with different crossover methods.

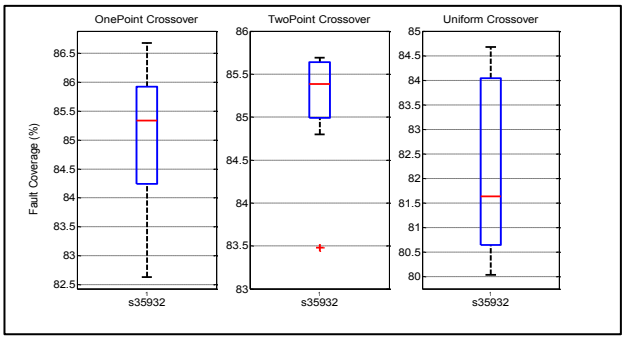

Figure B.2: Fault coverage for s35932 using GA-based test generator – Rank Selection with different crossover methods.

The following graphs represent the performance of GA-based test generators in ATPG. Each graph shows the fault coverage of one circuit with three different crossover schemes and roulette wheel selection operator. The boxplots help to show the data vibrations of 10 runs of a test. Several outliers occurred because of the effects of losing good genes on the earlier generations.

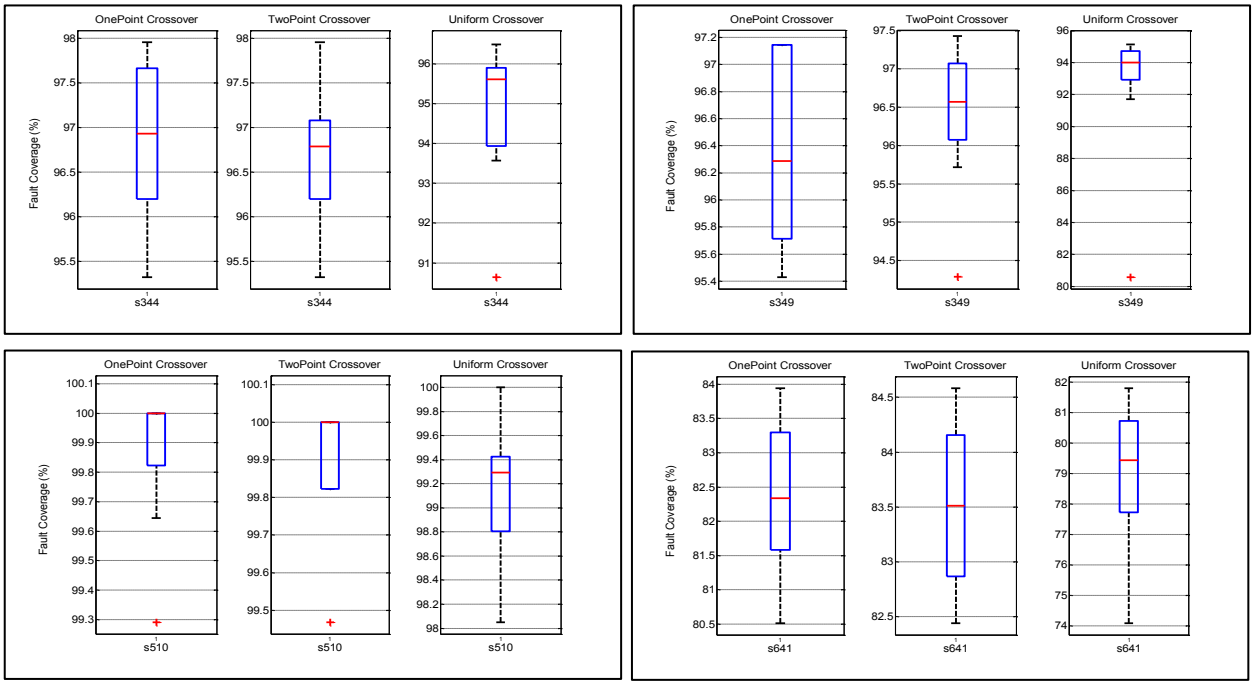

Figure B.3: Fault coverage for several sequential circuits using GA-based test generator – roulette wheel selection with different crossover methods.

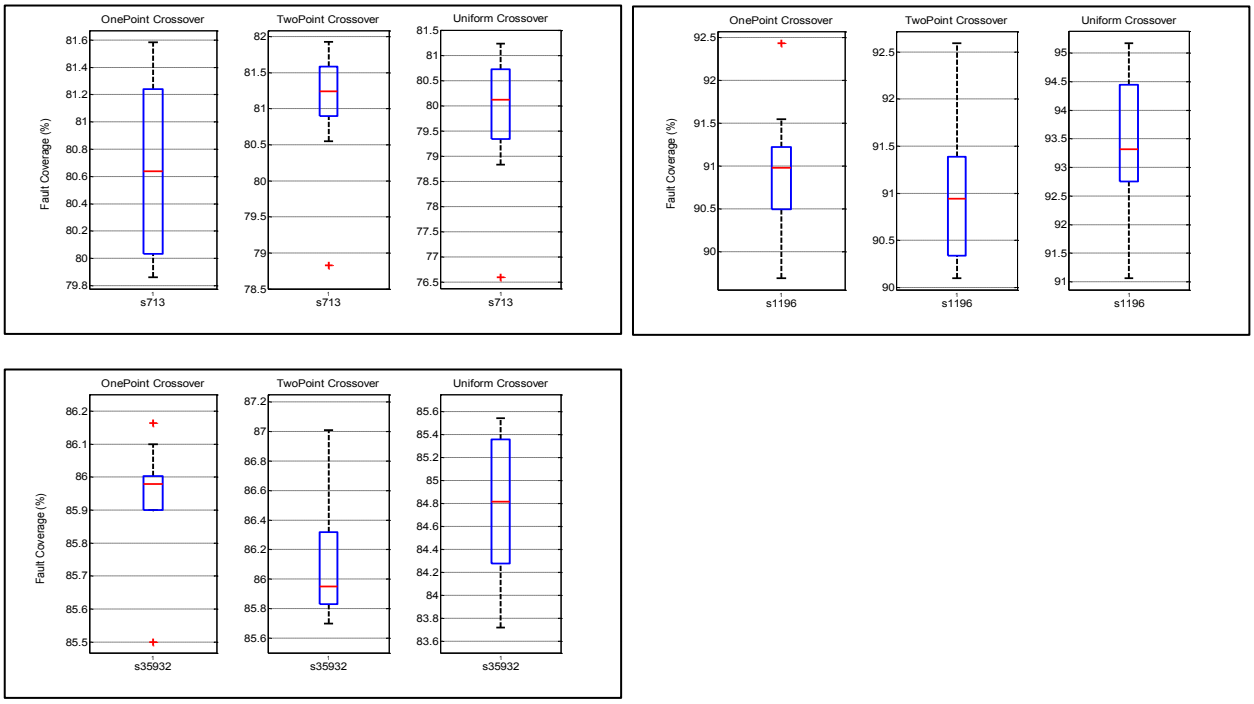

Figure B.4: Fault coverage for several sequential circuits using GA-based test generator – roulette wheel selection with different crossover methods.

The following graphs aim to show the performance of one-point crossover, two-point crossover and uniform crossover. The uniform crossover has the least performance in all circuits except s1196. Although we used two different selection operators, the comparison between crossover techniques was identical.

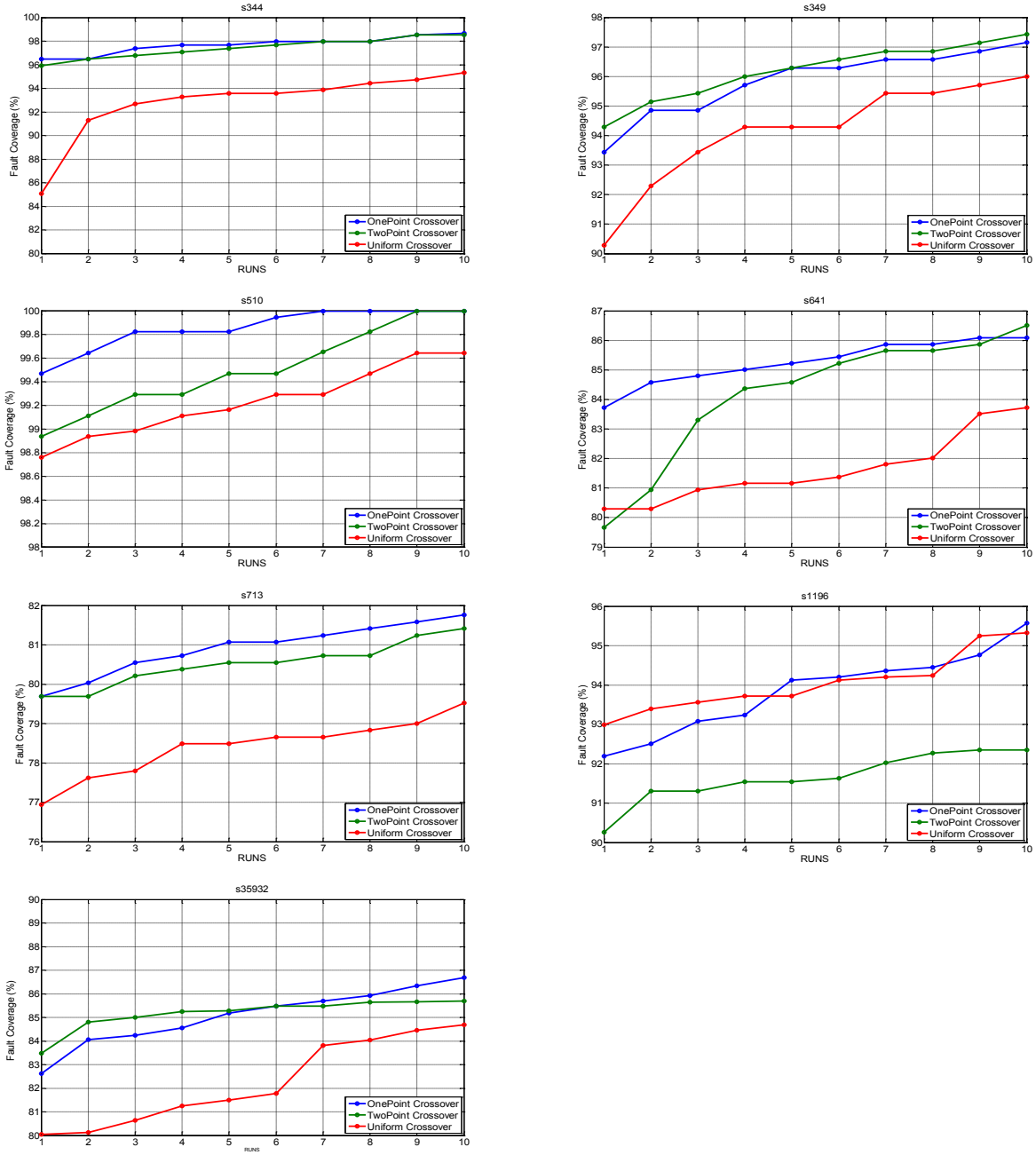

Figure B.5: Comparison between three crossover methods with rank selection (Results are in ascending order).

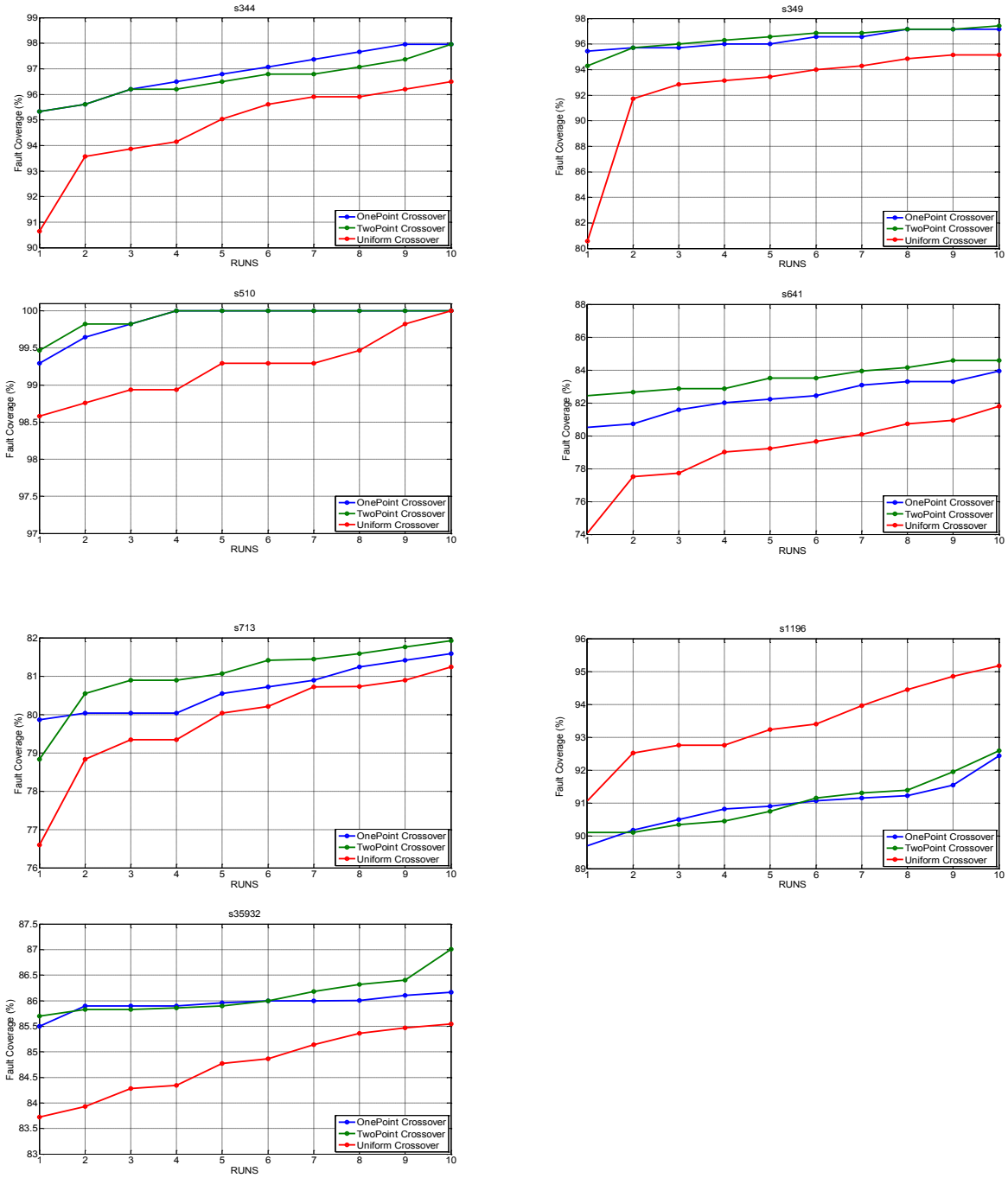

Figure B.6: Comparison between three crossover methods with roulette wheel selection (Results are in ascending order).

Appendix C

PSO-based Test Generator

The following graphs show the fault coverage of several sequential circuits. High fault coverage was obtained without any outliers which implies the accuracy and efficiency of search process in PSO.

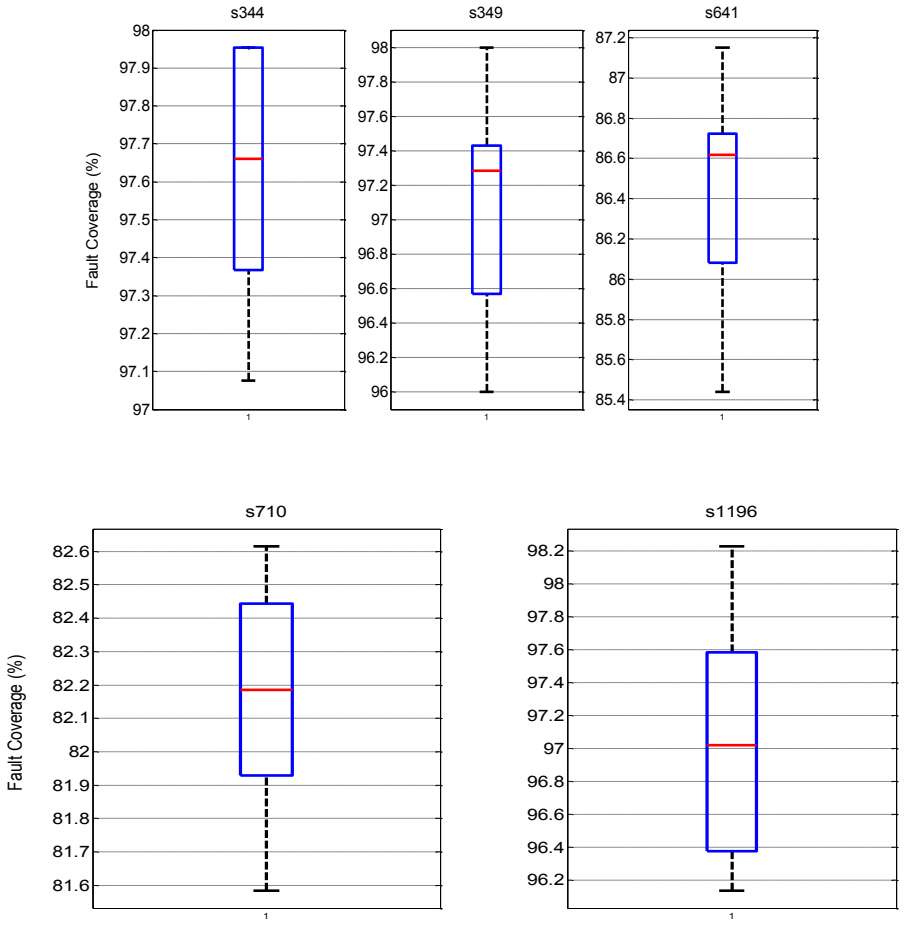

Figure C.1: Fault coverage of several sequential circuits using PSO-based test generator.

Appendix D

DE-based Test Generator

The following graphs show the performance of DE-based test generator in ATPG. The obtained fault coverage was lower than other methods. However, improvements can be obtained by implementing higher effective mutation strategy.

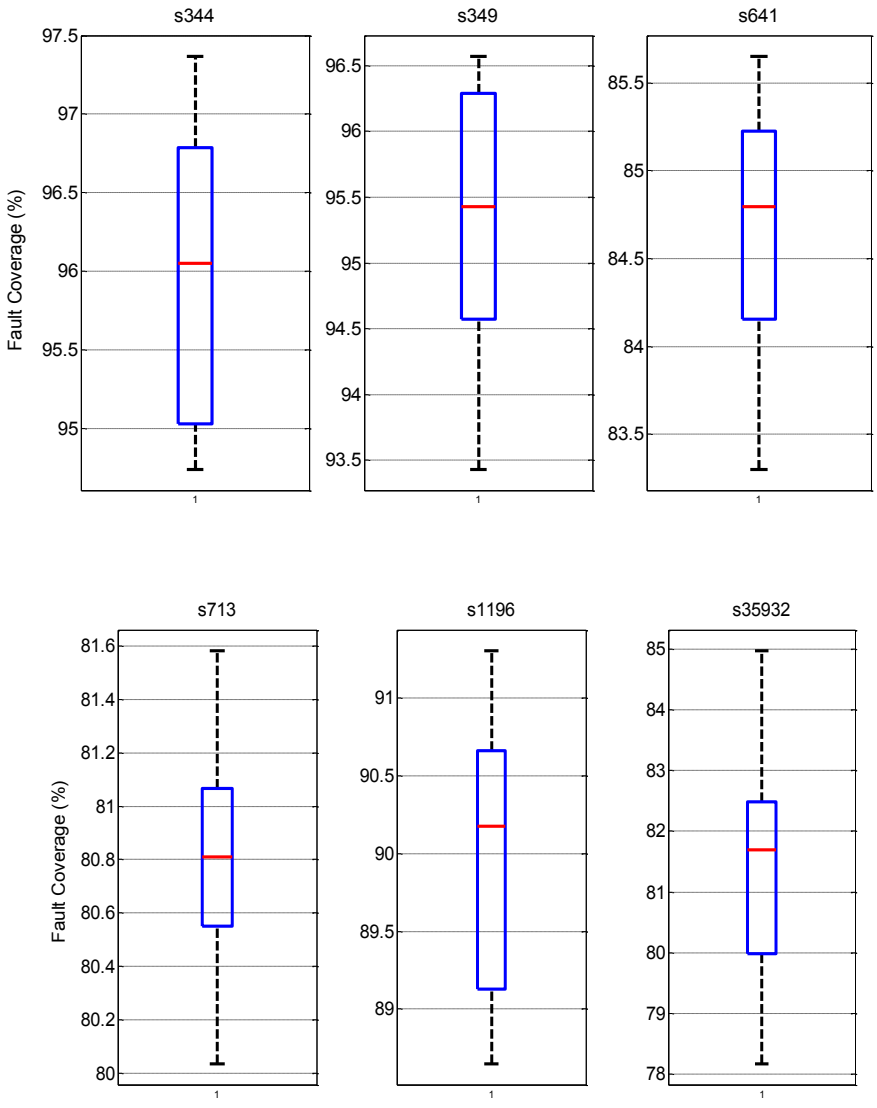

Figure D.1: Fault coverage of several sequential circuits using DE-based test generator.