Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm

Amer Samarah

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

September 2006

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Amer Samarah**

Entitled: **Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm**

and submitted in partial fulfilment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Dr. Rabin Raut

_____ Dr. Peter Grogono

_____ Dr. Amir G. Aghdam

_____ Dr. Sofiène Tahar

Approved by _____

Chair of the ECE Department

_____ 2006 _____

Dean of Engineering

ABSTRACT

Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm

Amer Samarah

Functional verification is a major challenge of the hardware design development and verification cycle. Several approaches have been developed lately in order to tackle this challenge, including coverage based verification. Within the context of coverage based verification, many metrics have been proposed to capture and verify the design functionality. Some of the used metrics are code coverage, FSM coverage, and functional coverage point that capture design specifications and functionalities. Defining the appropriate functional coverage points (monitors) is a quite tricky and non-trivial problem. However, the real bottleneck concerns generating suitable test patterns that can adequately activate those coverage points and achieve high coverage rate.

In this thesis, we propose an approach to automatically generate proper directives for random test generators in order to activate multiple functional coverage points and to enhance the overall coverage rate. In contrast to classical blind random simulation, we define an enhanced genetic algorithm procedure performing the optimization of coverage directed test generation (CDG) over domains of the system inputs. The proposed algorithm, which we call Cell-based Genetic Algorithm (CGA), incorporates unique representation and genetic operators especially designed for CDG problems. Rather than considering the input domain as a single unit, we split it into a sequence of cells (subsets of the whole input's domain) which provide rich and flexible representations of the random generator's directives. The algorithm automatically optimizes the widths, heights and distribution of these cells over the whole inputs domains with the aim of enhancing the effectiveness of using test generation. We illustrate the efficiency of our approach on a set of designs modeled in SystemC.

ACKNOWLEDGEMENTS

First and foremost, my wholehearted thanks and admire are due to the Holy One, Allah, Who has always been with me through out my entire struggle in life wiping out all my fears, for oblation His countless blessings. My sincere appreciation to the Hani Qaddumi Scholarship Foundation (HQSF) that has granted me a postgraduate scholarship at Concordia University.

I would like to express my sincere gratitude to my advisor, Dr. Sofiène Tahar for his guidance, support, patience, and his constant encouragement. His expertise and competent advice have not only shaped the character of my thesis but also my way of thinking and reasoning. Also, I would like to specially thank Dr. Ali Habibi, Concordia University, for his invaluable guidance. Without his insight and steering, needless to say, this thesis would have never got completed.

I would like to thank Dr. Nawwaf Kharma, Concordia University, for his unique approach of teaching and for valuable discussions on my thesis. His contribution played a major role in finalizing this thesis. Many thanks to my thesis committee members, Dr. Peter Grogono and Dr. Amir G. Aghdam, for reviewing my thesis and for their valuable feedback.

To all my fellow researchers in the Hardware Verification Group (HVG) at Concordia University, thank you for encouragement, thoughtful discussions, and productive feedback. Special thanks for Mr. Falah Awwad for introducing me to Dr. Sofiène Tahar. I would also like to express my appreciation for my friends, Saher Alshakhshir, Ize Aldeen Alzyyed, Mohammed Abu Zaid, Mohammed Othman, Alaa Abu Hashem, Hafizur Rahman, Asif Ahmad, Abu Nasser Abdullah, and Haja Moindeen for always supporting and being there for me at the time of my need.

Finally, I wish to express my gratitude to my family members for offering words of encouragements to spur my spirit at moments of depression.

This thesis is lovingly dedicated to

My Mother,
ADLA SHALHOUB
for all I started in her arms

And

My Father,
TAYSIR SAMARAH
for his fervent love for my education

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACDG	Automated Coverage Directed test Generation
APGA	Adaptive Population size Genetic Algorithm
ATG	Automatic Test Generation
ATPG	Automatic Test Pattern Generation
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
CDG	Coverage Directed test Generation
CGA	Cell-based Genetic Algorithm
CPU	Central Processing Unit
DUV	Design Under Verification
EDA	Electronics Design Automatons
FSM	Finite State Machine
GA	Genetic Algorithm
GP	Genetic Programming
HDL	Hardware Description Language
HDS	Hardware Design Systems
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
NN	Neural Networks
PSL	Property Specification Language
OOP	Object Oriented Programming
OSCI	Open SystemC Initiative
OVA	OpenVera Assertion
RTL	Register Transfer Level
SIA	Semiconductor Industry Association
SoC	Systems-on-Chip

TLM	Transaction Level Modeling
STL	Standard Template Library
SVA	System Verilog Assertion
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Motivation

The semiconductor industry has showed continuous and rapid advances during the last decades. The Semiconductor Industry Association (SIA) declared that the worldwide sales of the semiconductor industry in the year 2005 grossed up to an approximate of 227.6 billion USD. A new forecast conducted by SIA projects shows that worldwide sales of microchips will reach 309 billion USD in 2008 [6]. Along with these huge market opportunities, the complexity of embedded systems is increasing at an exponential rate as characterized by Moore's Law [40]. It is estimated that by the year 2010 the expected transistor count for typical System-on-Chip (SoC) solutions will approach 3 billion, with corresponding expected clock speeds of over 100 GHz, and transistor densities reaching 660 million transistors/cm^2 [49]. Concurrently, this increase in complexity will result in an increase in cost and time to market.

According to a study from the Collett International Research, less than half of the designs in North America are bug free the very first time of silicon fabrication [49], as shown in Figure 1.1. This failure, of getting working silicon, dramatically increases the time to market and reduces market shares due to the extreme high

silicon re-spin cost. The same study concluded that 71% of the system failure is due to functional bugs rather than any other type of bugs.

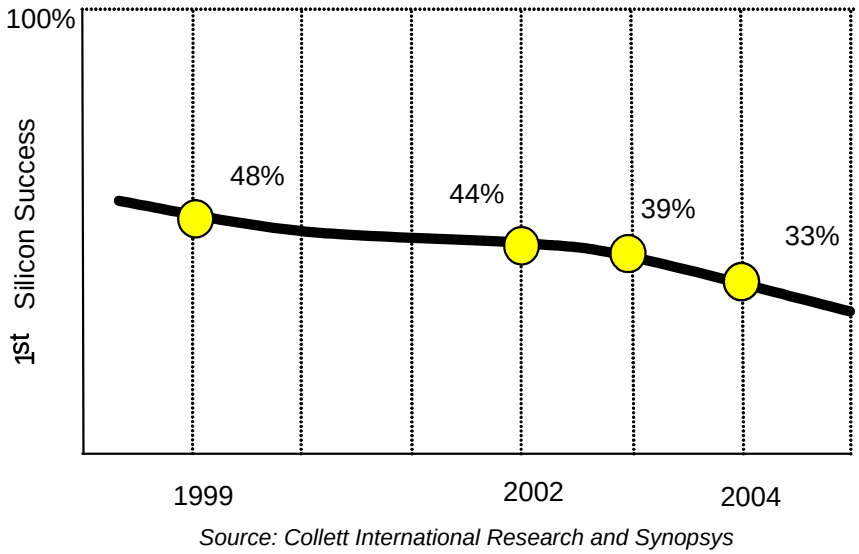


Figure 1.1: North America Re-Spin Statistics [49]

Consequently, functional verification has emerged as an essential part and a major challenge in the design development cycle. With unceasing growth in system functionality, the challenge today concerns, in particular, verifying that the logic design obeys the intended functional specification and performs the tasks required by the overall system architecture [54].

Many studies showed that up to 70% of the design development time and resources are spent on functional verification [17]. Another study highlights the challenges for functional verification. Figure 1.2 shows the statistics of SoC designs in terms of design complexity (logic gates), design time (engineer-years), and verification complexity (simulation vectors) [49]. The study highlights the tremendous complexity faced by simulation based validation of complex SoC's: it estimates that by 2007 a complex SoC will need 2000 engineer-years to write 25 million lines of Register Transfer Level (RTL) code and one trillion simulation vectors for functional

verification. A similar trend can be observed in the high-performance microprocessor era. For example, the number of logic bugs found in the Intel IA32 family of micro-architectures has increased exponentially by a growth rate of 300-400% from one generation to the next [7].

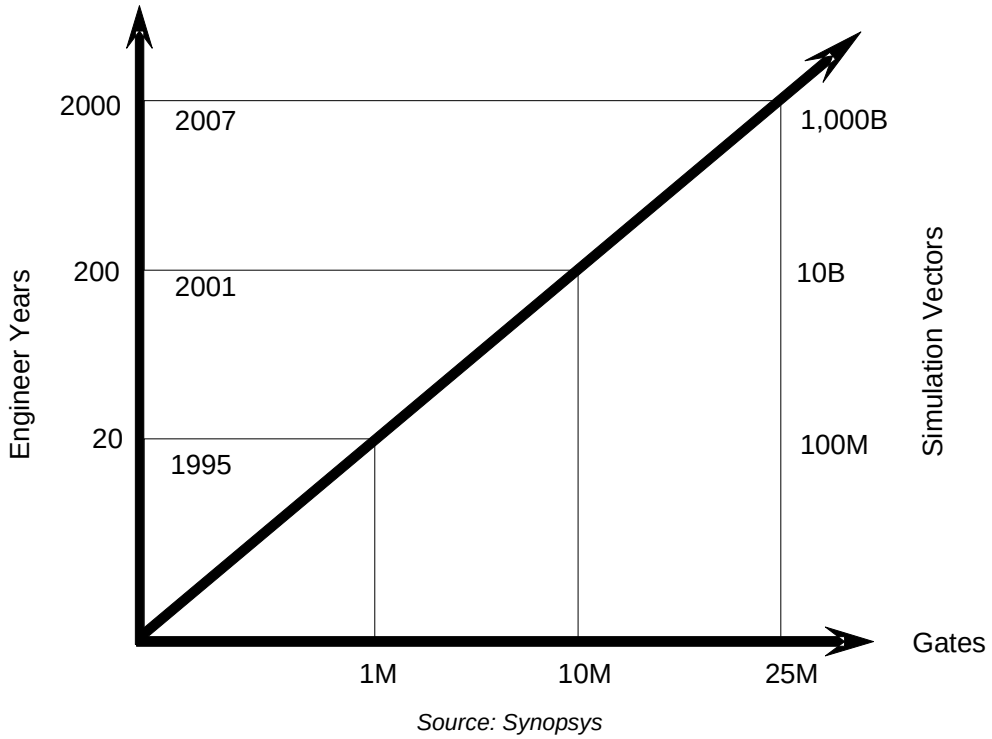


Figure 1.2: Design Verification Complexity [49]

In the next sections, we introduce the different techniques used to tackle functional verification problem. Then we give an overview for the problem of coverage directed test generator where a backward feedback, based on the coverage information, is used to direct the generation of test cases. Then we briefly present the methodology and algorithm we propose in this thesis and comment on related work. Finally, we summarize the contributions of the thesis and its organization.

1.2 Functional Verification

Functional verification, in electronic design automation (EDA), is the task of verifying that the hardware design conforms to specification. This is a complex task that consumes the majority of time and effort in most large electronic system design projects. Several methodologies have been developed lately in order to tackle the functional verification problem such as simulation based verification, assertion based verification, formal verification, and coverage based verification [54].

In simulation based verification, a dedicated test bench is built to verify the design functionality by providing meaningful scenarios. The most widely used form of design validation is a simulation employing random (or directed-random) test cases [52]. In this manner, the verification can be evenly distributed over the entire state space of the design, or can be biased to stress specific aspects of the design. A simulation testbench has high controllability of the input signals of the design under verification (DUV), while it has low controllability of an internal point. Furthermore, a simulation testbench generally offers limited observability in order to manifest errors on the output signals of the DUV.

Assertion based verification is employed to catch errors closer to the source of the bug, in terms of both time and location in the design intent. Assertions can be either properties written in a specialized assertion languages (e.g., Property specification Language (PSL) [2], System Verilog Assertion (SVA) [18], or OpenVera Assertion (OVA) [43]) or integrated monitors, within the simulation environment [54], that are written in a Hardware Description Language (HDL). One significant aspect of specifying assertions is to improve observability. As a result, there is a significant reduction in simulation debug time as reported by [1]. Furthermore, assertions add powerful capabilities to the verification process. They are able to drive formal verification, monitor simulation behavior, control stimulus generation and provide the basis for comprehensive functional coverage metrics. Many prominent

companies adopted the use of assertions. For example, designers at Digital Equipment Corporation reported that 34% of all bugs uncovered with simulation were found by assertions on DEC Alpha 21164 project [32].

In contrast to simulation, formal verification concerns the proving or disproving of the correctness of a system using formal methods of mathematics [35]. Formal techniques use mathematical reasoning to prove that an implementation satisfies a specification and like a mathematical proof the correctness of a formally verified hardware design holds regardless of input values. All possible cases that can arise in the system are taken care of in formal verification. There are three main techniques for formal verification: Equivalence Checking, Model Checking and Theorem Proving [35]. The first two techniques, while exhaustive, do not scale for large designs due to the problem of state-space explosion. Theorem proving, on the other hand, is scalable but needs a considerable human effort and expertise to be practical.

Coverage based verification is an important verification technique used to assess progress in the verification cycle. Moreover, it identifies functionalities of the design that have not been tested. However, it cannot guarantee the completeness of verification process as with formal verification. The concept of coverage based verification requires the definition of coverage metrics that provide quantitative measures of the verification process. The most widely used metrics are: code coverage, finite state machine (FSM) coverage, and functional coverage. Each metric provides specific aspects about the completeness of the verification process. Even though none of these metrics are sufficient to prove a design is error free, they are helpful in pointing out areas of the design that have not been tested.

Code coverage evaluates the degree to which the structure of HDL source code has been exercised. It comes under the heading of white box testing where HDL code is accessible. There are a number of different ways of measuring code coverage. The main ones are [53]: (1) statement coverage, (2) decision coverage, (3) condition coverage, (4) path coverage, (5) toggle coverage, and (6) triggering coverage.

FSM coverage gives more clues about the functionality of the system. There are mainly two ways to measure the FSM coverage [53]: (1) *FSM state coverage*: the percentage of the visited states during the simulation and (2) *FSM transition coverage*: the percentage of the visited possible transition between all states during the simulation. The main problem with FSM coverage is that the generation of large FSMs leads to combinatorial explosion (state explosion problem), where it is impossible to follow all combinations of FSM transitions.

Functional coverage involves defining a set of functional coverage tasks or points (similar to assertion points), which represent the important specification of the DUV. Furthermore, the verification team monitors and verifies the related properties to those specifications during the verification process. The coverage could be simply the number of activation of all coverage points [54].

1.3 Problem Statement and Methodology

The verification cycle starts by providing the verification team with the design to be verified and its functional specifications. Then, the verification engineers translate (interpret) those functional specifications to functional coverage task (or points) that represent the important behavior and specification of the design. Moreover, they write testbenches that are able to activate those coverage tasks and exercise important portions of the design.

Using random test generators is a common method of exploring unexercised areas of the design. Coverage tools are used side by side with a random test generator in order to assess the progress of the test plan during the verification cycle. Coverage analysis allows for: (1) the modification of the directives for the test generators; and (2) the targeting of areas of the design that are not covered well [19]. This process of adapting the directives of test generator according to feedback based on coverage reports is called *Coverage Directed test Generation* (CDG). It is a manual

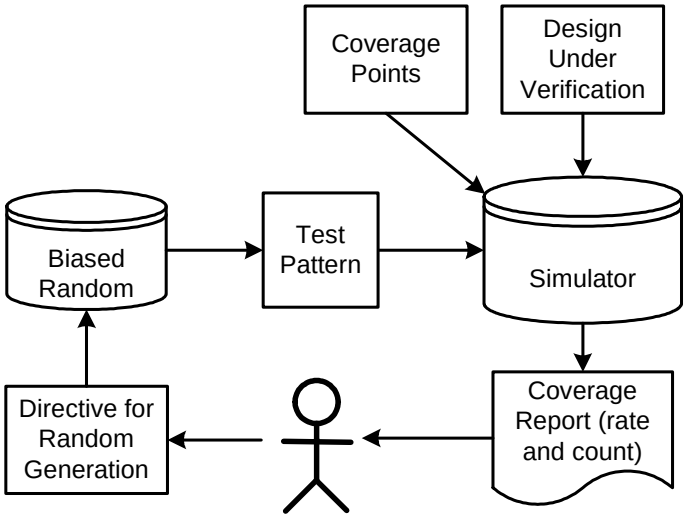


Figure 1.3: Manual Coverage Directed Test Generation

and exhausting process, but essential for the completion of the verification cycle (Figure 1.3).

Considerable effort is being invested in finding ways to close the loop of connecting coverage analysis to adaptive generation of test directives within highly automated verification environment. In this thesis, we propose an approach to automate adaptive CDG called *Cell-based Genetic Algorithm (CGA)*. CGA inherits the advantages of genetic algorithms, which are techniques inspired by evolutionary biology [39]. The evolution starts from a population of completely random abstract potential solutions and continues over several generations. In each generation, all individual members of the population are evaluated for fitness using a fitness evaluation function or methods; multiple individuals are stochastically selected from the current population based on their fitness values and are then possibly modified by genetic operators to form a new population for further evolution. The process of evaluation, selection and diversification iterates until a termination criterion is satisfied.

CGA analyzes the coverage reports, evaluates the current test directives, and acquires the knowledge over time to modify the test directives. We aim at: (1)

constructing efficient test generators for checking important behaviors and specifications of the DUV; (2) finding complex directives for specific coverage tasks; (3) finding common directives that activate groups of correlated coverage tasks while achieving adequate coverage rates; (4) improving the coverage progress rates; and (5) designing directives that can reach uncovered tasks.

By achieving the above goals, we increase the efficiency and quality of the verification process and reduce the time, effort, and human intervention needed to implement a test plan. Moreover, because the CGA increases the number of times a task has been covered during verification, the verification process becomes more robust.

The proper representation of test generator directives has a critical importance. We model the directives as a group of sub ranges, or *cells* as we call them, over the input domains for the test generators. Each cell has specific range (widths) and weight that represents the probability of generating values from that range with respect to weights of other cells. CGA automatically optimizes the range and distribution weights of these cells over the whole domain with the aim of enhancing the effectiveness of the test generation process for the considered coverage group.

In order to evaluate the CGA, we consider several designs modeled in SystemC [42]. SystemC is the latest IEEE standard for system level design languages. It is a class library and modeling platform built on top of C++ language to represent functionality, communications, and software and hardware implementation at various levels of abstraction. The main advantage of using SystemC is to obtain faster simulation speed that enables faster convergence of the CGA. In addition to that, the C++ development environment provides a unified framework for hardware modelings using SystemC, verification procedures, and algorithmic development, which significantly reduce the time to market.

1.4 Related Work

In this section we present the related work in the area of improving functional verification using various algorithms, then we give an overview of the important work for functional processor validation particularly using evolutionary algorithms. An examples of the utilization of genetic algorithms for improving FSM and code coverage are also mentioned. Finally, we present a review of the well known work of using genetic algorithm for automatic test generation (ATG) for combinational, sequential, and SoC chip level testing.

Considerable effort has been spent in the area of functional verification. Methodologies based on *simple genetic algorithm*, *machine learning methods*, and *Markov chain modeling* have been developed. For instance, the work of [27] uses a simple genetic algorithm to improve the assertions coverage rates for SystemC RTL models. This work tackles only a single assertion (coverage point) at a time, and hence is unable to achieve high coverage rates due to the simple encoding scheme used during the evolution process. In contrast, our methodology which is based on multiple sub-range directives is able to handle multiple assertion (coverage points) simultaneously.

The work of [16] proposed a method based on simple genetic algorithm for guiding simulation using normal random input sequences to improve coverage count of property checkers related to the design functionality. The genetic algorithm optimizes the parameters that characterize the normal random distribution (i.e., mean and standard deviation). This work is able to target only one property at a time rather than a group of properties, and it is not able to describe sharp constraints on random inputs. Experimental results on RTL designs, showed that the pure random simulation can achieve the same coverage rate but almost with double number of simulation vectors in case of using genetic algorithm.

In [17], Bayesian networks, which have proven to be a successful technique in machine learning and artificial intelligence related problems, are used to model the

relation between coverage space and test generator directives. This algorithm uses training data during the learning phase, where the quality of this data affects the ability of Bayesian networks to encode correct knowledge. In contrast, our algorithm starts with totally random data, where the quality of initial values affects only the speed of learning and not the quality of the encoded knowledge. Furthermore, [50] proposes the use of a Markov chain that represents the DUV, while coverage analysis data is used to modify the parameters of the Markov chain. The resulting Markov chain is used to generate test-cases for the design.

A recent work of [28] introduces expressive functional coverage metrics which can be extracted from the DUV golden model at a high level of abstraction. After defining a couple of assertions (property checkers), a reduced FSM is extracted from the golden model. The reduced FSM represents the related functionality to the defined assertion. Moreover, the coverage is defined over states and transitions of the reduced FSM to ensure that each assertion is activated through many paths. However, the work of [28] cannot be applied to complex designs where the state explosion problem [20] may arise even for the exploration of the reduced FSM.

Evolutionary algorithm (genetic algorithm and genetic programming) have been used for the purpose of processor validation and to improve the coverage rate based on ad-hoc metrics. For instance, in [10], the authors proposed to use a genetic algorithm to generate biased random instruction for microprocessor architectural verification. They used ad-hoc metrics that utilizes specific buffers (like store queue, branch issue buffer, etc.) for the PowerPC architecture. The approach, however, is limited only to microprocessor verification.

In [13], *genetic programming* rather than genetic algorithm is used to evolve a sequence of valid assembly programs for testing pipelined processor in order to improve coverage rate for user defined metrics. The test program generator utilizes a directed acyclic graph for representing the flow of an assembly program, and an instruction library for describing the assembly syntax.

Genetic algorithms have been used for many other verification and coverage problems. For instances, [20] addresses the exploration of large state spaces. This work is based on BDDs (Binary Decision Diagrams) and hence is restricted to simple Boolean assertions. Genetic algorithms have been used with code coverage metrics to achieve high coverage rate of the structural testing. As an example, the work of [31] illustrate the use of genetic algorithm in order to achieve full branch coverage.

Furthermore, genetic algorithms have been used for Automatic Test Pattern Generation (ATPG) problems in order to improve the detection of manufacturing and fabrication defects [38]. Finally, [11] presents a genetic algorithm based approach to solve the problem of SoC chip level testing. Particularly, the algorithm optimizes the test scheduling and test access mechanism partition for SoC in order to minimize the testing time which in return reduces the cost of testing. This approach shows a superior performance to the heuristic approaches proposed in the literature.

1.5 Thesis Contribution and Organization

In light of the above related work review and discussions, we believe the contributions of this thesis are as follows:

1. We provide an approach to automate the generation of proper directives for random test inputs based on coverage evaluation.

2. We implement a Cell-based Genetic Algorithm (CGA) and incorporate a rich representation and unique genetic operators especially designed for Coverage Directed test Generation problems. A couple of parameters are defined to adapt CGA for different situations and designs.

3. While the convergence of other approaches (e.g., machine learning based techniques) are affected by the quality of the training data, our CGA needs minimum human intervention and can evolve towards optimal solutions starting from a random exploration.

4. We provide effective evaluation functions so that our CGA is able to target the activation of many functional coverage points while achieving minimum acceptable coverage rate for all coverage points.

5. We provide a generic methodology that is applicable for control, data, and corner cases coverage points.

6. We provide an example of unified development environment for design, verification, and algorithmic development as we use Microsoft Visual C++ 6.0 for those purposes.

The rest of the thesis is organized as follows. Chapter 2 provides an introduction to genetic algorithm which is the heart of our proposed methodology. Chapter 3 describes our approach of integrating CGA within the verification cycle, then we present the details of CGA. Chapter 4 discusses the benefit of using SystemC as a modeling language for embedded design, then we illustrate and discuss the experimental results for three case studies written in SystemC. Finally, Chapter 5 concludes the thesis and provides possible improvement of CGA as well as other proposals to solve CDG problems for future work.

Chapter 2

Preliminary - Genetic Algorithms

This chapter gives a brief description of genetic algorithms (GA's) as well as the main thing to be considered during the design of genetic algorithm based solution and the execution flow.

GA's are particular classes of evolutionary algorithms that use techniques inspired by evolutionary biology. They are based on a Darwinian-type competition (survival-of-the-fittest) strategy with sexual reproduction, where stronger individuals in the population have a higher chance of creating an offspring. Since their introduction by Holland in 1975 [14], genetic algorithms have been applied to a broad range of searching, optimization, and machine learning problems. Generally speaking, the GA's are applied to spaces which are too large to be exhaustively searched.

GA's are iterative procedures implemented as a computerized search and optimization procedure that uses principles of natural selection. It performs a multidirectional search by maintaining a population of potential solutions (called *individuals*) and exchange information between these solutions through simulated evolution and forward relatively "good" information over generations until it finds a near optimal solution for specific problem [39].

Usually, GA's converge rapidly to quality solutions. Although they do not

guarantee convergence to the single best solution to the problem, the processing leverage associated with GA's makes them efficient search techniques. The main advantage of a GA is that it is able to consider many points in the search space simultaneously, where each point represents a different solution to a given problem. Thus, the possibility of the GA getting stuck in local minima is greatly reduced because the whole space of possible solutions can be simultaneously searched.

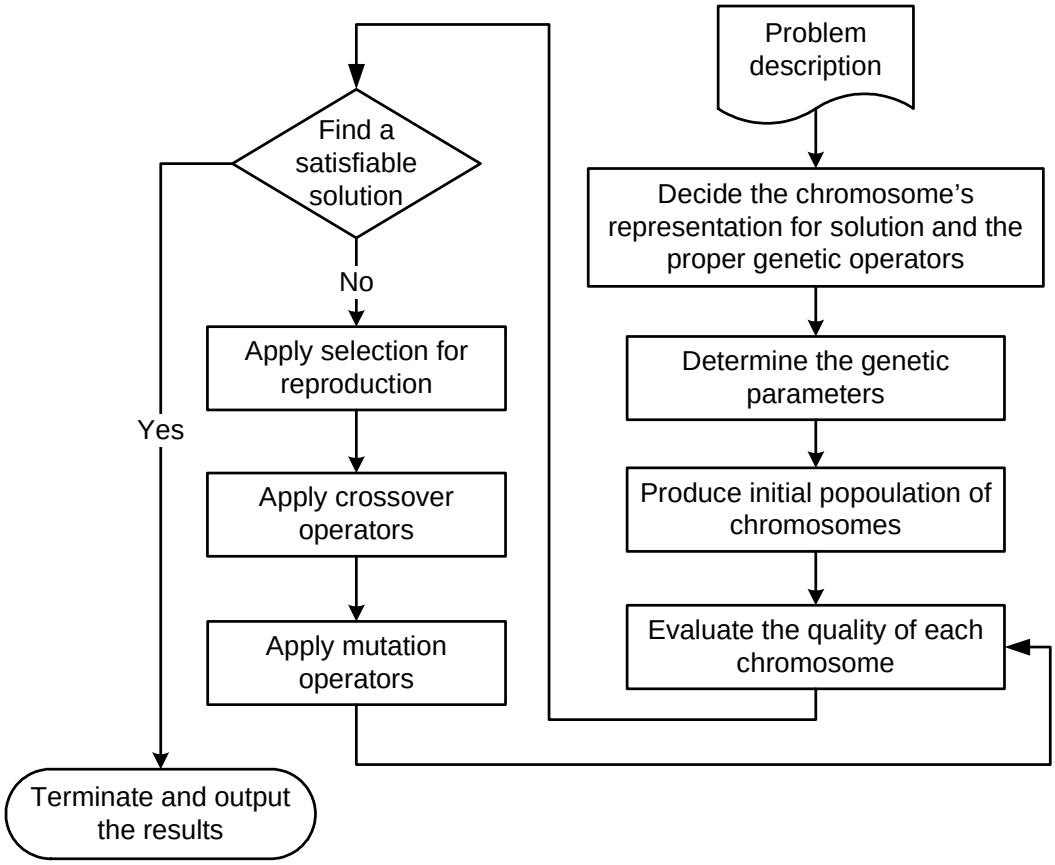


Figure 2.1: Genetic Algorithms - Design and Execution Flow

Given a description of a problem to be solved using GA's, the developers come out with a suitable representation for the potential solution to the problem. Then, they design a few of genetic operators like *crossover* and *mutation* operators to be applied during the phase of evolution from generation to another one. Furthermore,

they tune many genetic parameters that affect the ability and speed of GA's of finding a near optimal solution. Those parameters include population size, probabilities of applying genetic operators, and many more according to each problem. Figure 2.1 shows the design and execution flow of generic GA's.

GA's start by generating initial population of potential solutions. After the initialization phase, an evaluation of the quality of the potential solutions takes place in order to guide the search for optimal solution within the (huge) search space of the problem over generations. As long as the optimal solution has not been found or a maximum allowed time (generations) has not been reached, the evolution process continue over and over. In that case, developers choose some potential solution according to their quality for reproduction purposes, then they apply the genetic operators on the current population to produce a new population for evaluation.

2.1 Representation

During the process of evolution, GA's use an abstract artificial representation (called genome or chromosome) of the potential solution. Traditionally, GA's use a fixed-length bit string to encode a single value solution. Also, GA's employ other encodings like real-valued numbers and data structures (trees, hashes, linked list, etc.) A mapping (encoding) between the points in the search space (*phenotype*) of the problem and instances of the artificial genome (*genotype*) must be defined (Figure 2.2).

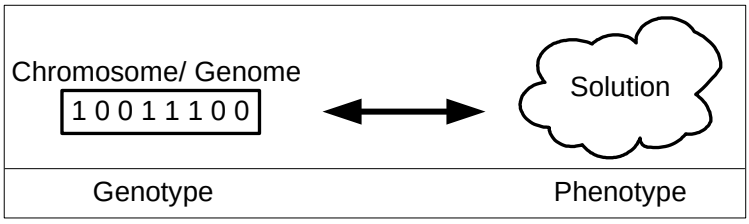


Figure 2.2: Phenotype-Genotype - Illustrative Example

The representation can influence the success of GA's and its convergence to the optimal solution within time limit. Choosing the proper representation is always problem dependent. For instance, [46] state that "If no domain specific knowledge is used in selecting an appropriate representation, the algorithm will have no opportunity to exceed the performance of an enumerative search."

2.2 Population of Potential Solutions

After the proper representation is chosen, a *population* of N potential solutions is created for the purpose of evolution. The set of initial potential solutions can be generated randomly as in most cases, loaded from previous runs of GA's, or generated using heuristic algorithms.

Furthermore, the tuning of the population size N is of great importance. It may depend on the problem complexity and the size of the search space. A large N increases the diversity of the population and so reduces the possibility to be caught in a local optimum. On the other hand, maintaining a large N needs an extensive computational resources for individuals evaluation and for applying reproduction and evolution operators.

Usually the size of a population is kept constant over generations. However, some GA implementations employ dynamic population sizes like APGA, GA's with adaptive population size, and parameter less GA's [14].

2.3 Evaluation Function - Fitness Value

A qualitative evaluation of potential solution's quality and correctness is represented by *fitness values* which may take into consideration many aspects and features of the potential solution. Fitness evaluation is important for speedy and efficient evolution and is always problem dependent.

Fitness values must reflect the human understanding and evaluation precisely. The most difficult thing with this evaluation function is how to find a function that can expresses the effectiveness of potential solution using *single* fitness value that encapsulates many features of the potential solution. That fitness value must be informative enough for the GA to discriminate potential solutions correctly and to guide the GA through the search space.

2.4 Selection Methods

During the evolution process, a proportion of the existing population is selected to produce a new offspring. Selection operators and methods must ensure large diversity of the population and prevent premature convergence on poor solutions while pushing the population towards better solutions over generations. Popular selection methods include *roulette wheel selection*, *tournament selection*, and *ranking selection* [14].

2.4.1 Roulette Wheel Selection

In roulette wheel selection, also known as fitness proportionate selection, a fitness level is used to associate a probability of selection with each individual chromosome as shown in Figure 2.3. Candidate solutions with a higher fitness will more likely dominate, will be duplicated, and will be less likely eliminated which may reduce the diversity of population over generations and may cause a fast convergence to a local optimum that is hard to escape from.

The roulette wheel selection algorithm starts with the generation of a random probability number $p \, \epsilon \, [0, 1]$, then the calculation of the relative fitness value and the cumulative fitness value of current individual. The cumulative fitness values is compared to the random number p till we find an individual with cumulative fitness value less than or equal to that random number.

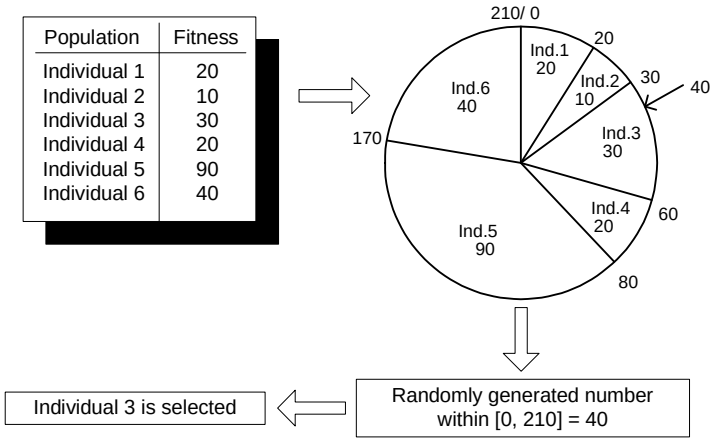


Figure 2.3: Roulette Wheel Selection

2.4.2 Tournament Selection

Tournament selection runs a "tournament" among a few individuals, equal to a tournament size, chosen at random from the population as shown in Figure 2.4. The chosen individuals compete with each other to remain in the population, and finally the one with the best fitness is selected for reproduction. The chosen individual can be selected more than once for the next generation during different "tournament" runs. Moreover, selection pressure can be easily adjusted by changing the tournament size. By using this method, solution with low fitness has a better opportunity to be selected for reproduction and generate a new offspring. However, if the tournament size is high, weak individuals have a smaller chance to be selected.

One of the problems of tournament selection is that the best individuals may not get selected at all especially when the population size N is very large. To solve this problem, an *elitism* mechanism is used, where the best individuals are copied without change to the elitism set. Also, the elitism mechanism guarantees that the best individuals within a specific generation are never destroyed by genetic operators (crossover or mutation operators).

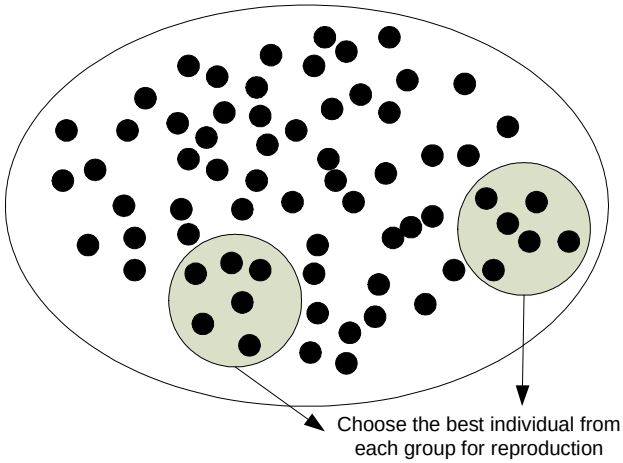


Figure 2.4: Tournament Selection

2.4.3 Ranking Selection

This is a similar method to tournament selection, which is used to reduce the selection pressure of highly fitted individuals when the fitness variation is high. Instead of using absolute fitness values, scaled probabilities are used based on the rank of individuals within their population. Moreover, individuals with lower fitness values give larger weights than the original values in order to improve their chances for selection.

2.5 Crossover Operators

Crossover is performed between two selected individuals by exchanging parts of their genomes (genetic material) to form new individuals. Figure 2.5 shows an illustrative diagram of single point crossover in bit-string chromosomes where genetic materials are exchanged around a crossover point. Crossover is important to keep the useful features of good genomes of the current generation and forward that information to the next generation.

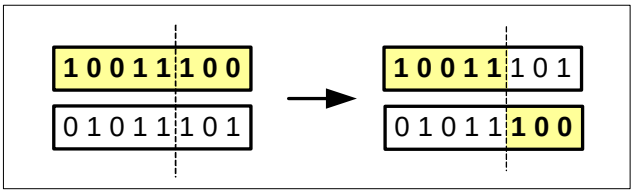


Figure 2.5: Crossover Operator - Illustrative Example

Crossover operators are applied with a probability P_c to produce a new offspring. In order to determine whether a chromosome will undergo a crossover operation, a random number $p \epsilon [0, 1]$ is generated and compared to the P_c. If $p < P_c$, then the chromosome will be selected for crossover, else it will be forwarded without modification to the next generation.

2.6 Mutation Operators

The mutation operator alters some parts of individuals in a randomly selected manner. Figure 2.6 illustrates mutation in a bit-string chromosome where the state of a randomly chosen bit is simply negated. Mutation operators introduce new features to the evolved population which are important to keep diversity that helps GA's escape from a local optimum and explore a hidden area of the search space. We can think of mutation operators as the global search operators, while crossover operators perform the local search. While mutation can explore a hidden region of the global search space, it may destroy the good building blocks (schema) of the selected chromosome. Finally, mutation operators are applied with a probability P_m which tends to be lower than the crossover probability P_c.

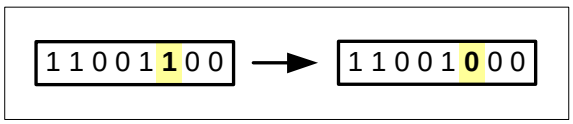


Figure 2.6: Mutation Operator - Illustrative Example

2.7 Summary

The design of a solution based on GA's is problem dependent where representation, genetic parameters, initialization scheme, and genetic operators are chosen differently according to that specific problem. Figure 2.7 provides an illustrative chart for the genetic evolution process of a bit-string chromosome representation. Accordingly, the development of a GA based solution starts with designing a suitable representation for the potential solution to the problem (bit-string in Figure 2.7). Then, a few genetic operators are chosen to be applied during the phase of evolution from generation to another one.

As GA's perform a multi-directional search, the evolution process begins by generating a population of initial potential solutions. Then, we evaluate the quality of each potential solution of the initialization pool for the reproduction of new populations over many generations. The selection of potential solutions, for reproduction purposes, is based on the quality of these solutions.

Next, genetic operators, i.e., crossover and mutation, are applied on the selected potential solution. Afterwards, a new generation of solutions is born. We check whether any of the new potential solution satisfies the termination criterion. If it does, then we stop the running of the GA's cycle, else the GA continues until either reaching the maximum allowed time or finding an optimal solution within the new generations.

In the next chapter, we will illustrate the details of designing our GA based solution for the problem of coverage directed test generation.

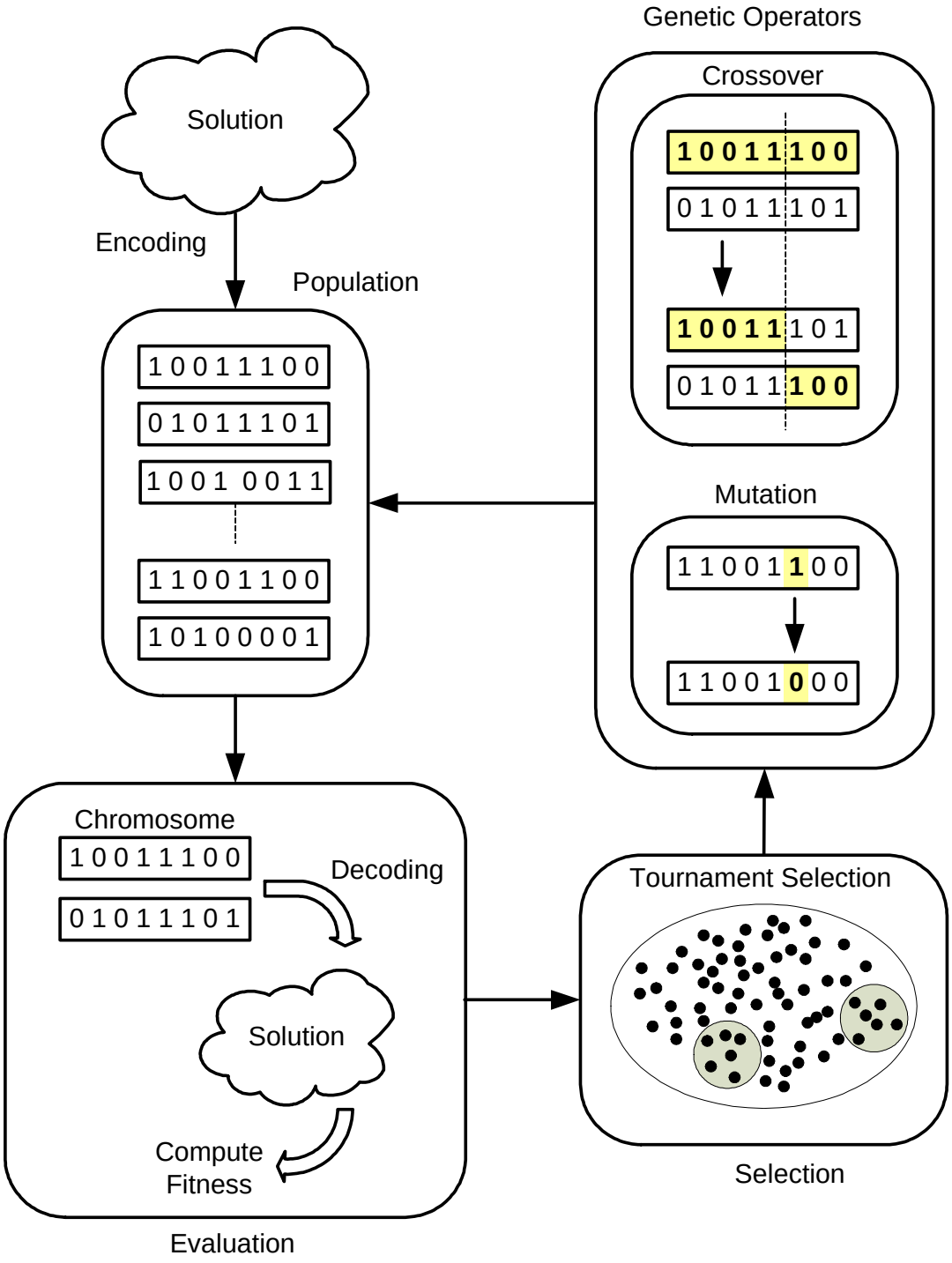


Figure 2.7: Genetic Algorithms - Illustrative Diagram

Chapter 3

Automated Coverage Directed Test Generation Methodology

In this chapter we illustrate the methodology of integrating the Cell-based Genetic Algorithm (CGA) within the design verification cycle in order to provide automated coverage directed test generation (automated CDG) for a group of coverage tasks. Then, we describe the specification and implementation issues of the proposed CGA including representation, initialization, selection, crossover, mutation, and termination criteria. At the end of this chapter, we discuss the properties of random numbers, then we highlight their importance during random simulation and genetic algorithms evolution.

3.1 Methodology of Automated CDG

Classical verification methodologies based on coverage directed test generation usually employ directed random test generators, in order to produce effective test patterns, capable of activating a group of coverage points (tasks), written for a specific DUV. Moreover, analysis of coverage information (reports) is necessary to modify and update the directives of random test generators, in order to achieve verification

objectives and to exercise the important portions of the DUV.

The cell-based genetic algorithm, developed in this thesis, automates the process of analyzing coverage information and modifying the test generators directive, as shown in Figure 3.1.

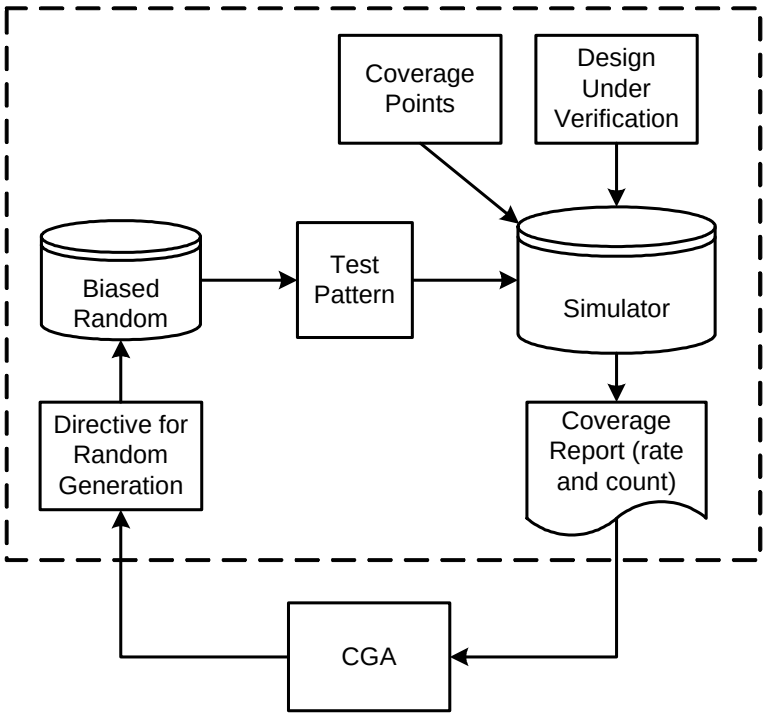


Figure 3.1: Automatic Coverage Directed Test Generation

We integrate our CGA within the design simulation cycle, where we start by producing initial random potential solutions, then two phases of iterative processes are performed: a *fitness evaluation* phase and a *selection and diversification* phase. During the Fitness Evaluation phase, we evaluate the quality of a potential solution that represents possible test directives. This process starts with extracting the test directives and generating test patterns that stimulate the DUV. Thereafter, the CGA collects the coverage data and analyzes them, then it assigns a fitness value to each potential solution that reflects its quality. The evaluation criterion is based on many factors including coverage rate, variance of the coverage rate

over the same coverage group, and the number of activated points. In the selection and diversification phase, several evolutionary operations (elitism and selection) and genetic operators (crossover and mutation) are applied on the current population of potential solutions to produce a new (better) population. These two phases will be applied to each population until the algorithm satisfies the termination criterion. Figure 3.2 presents a flow chart summarizing the proposed CGA.

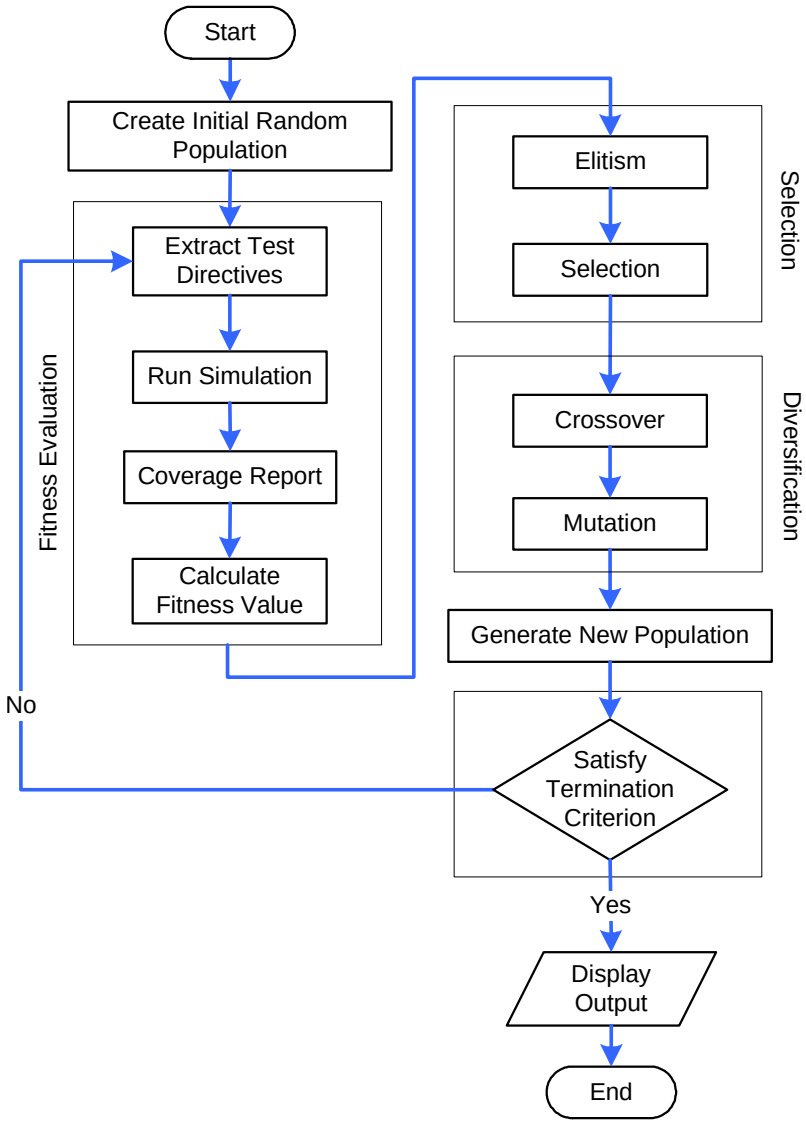


Figure 3.2: Proposed CGA Process

A pseudo-code of the proposed CGA (Algorithm 3.1) is given below. The methodology starts with an empty solution set *Pop*, then it initializes *maxPop* random solutions (lines 3 – 4). The algorithm runs for *maxGen* times. As we mentioned earlier, each run goes through two phases : (1) Fitness evaluation phase (lines 8 – 13), where the whole solutions set, *Pop*, is applied on the DUV for the purpose of stimulation and evaluation of the quality and effectiveness of those solutions; and (2) Selection and diversification phase (lines 16 – 22), where a new population set, *newPop*, is created by selecting individuals from the current population *Pop* and using them as a basis for the creation of the new population *newPop*, via crossover and mutation. The new population, *newPop*, becomes the current population *Pop* for the next run of the algorithm. The operational details of the CGA are provided in the next section.

3.2 Cell-based Genetic Algorithm

In the following sub-sections, we describe the details of our CGA including representation, initialization, selection, crossover, mutation, and termination criteria.

3.2.1 Representation (Encoding)

We represent the description of a random distribution as sequences of directives that direct the random test generator to activate the whole coverage group. Accordingly, we model the directives as a list of sub-ranges over the inputs domains for the test generators. This representation enables the encoding of complex directives. Figure 3.3 represents a potential solution for some input i which is composed of 3 directives.

A *Cell* is the fundamental unit introduced to represent a partial solution. Each cell represents a weighted uniform random distribution over two limits. Moreover, the near optimal random distribution for each test generator may consist of one or

Algorithm 1: Pseudo-code of CGA Methodology

Input : Design Under Verification (DUV).
Input : Genetic parameters and settings.
Output: Near optimal test generator constraints that achieve high coverage rate.

```
1  Pop ⟵ ∅
2  run ⟵ 0
3  for i ← 1 to maxPop do
4      Initialize (Pop[i])
5  end
6  while run < maxGen do
7      Fitness evaluation of possible solutions
8      forall solution of Pop do
9          Extract test directives
10         Run simulation of DUV
11         Collect coverage results
12         Calculate fitness value of current solution
13     end
14     Generating new population of solutions
15     newPop ⟵ ∅
16     while size (newPop) < maxPop do
17         newPop ⟵ Elitism (Pop, best 10%)
18         tempSolns ⟵ Selection (Pop, 2 solutions)
19         tempSolns ⟵ Crossover (tempSolns)
20         tempSolns ⟵ Mutation (tempSolns)
21         newPop ⟵ tempSolns
22     end
23     Pop ⟵ newPop
24     run ⟵ run + 1
25 end
```

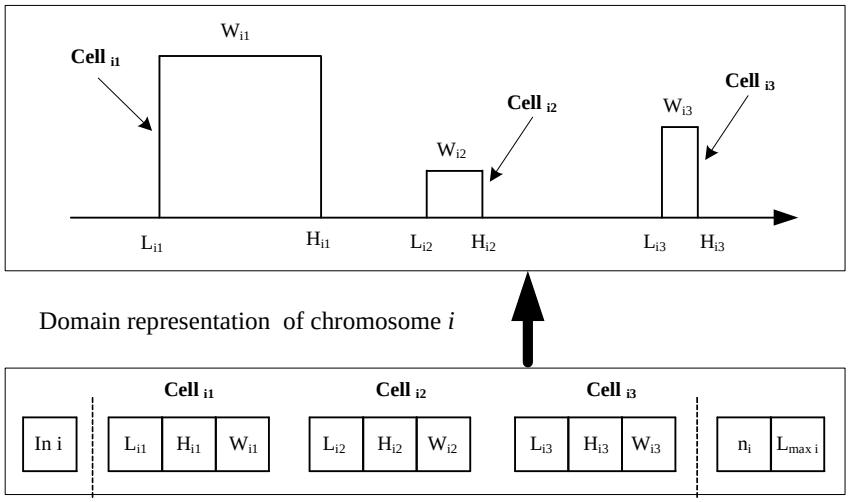


Figure 3.3: Chromosome Representation

more cells according to the complexity of that distribution. However, we call the list of cells representing that distribution a *Chromosome*. Usually, there are many test generators that drive the DUV, and so we need a corresponding number of chromosomes to represent the whole solution to our problem which we call *Genome*.

Let's consider a $Cell_{ij}$ which is the j^{th} cell corresponding to test generator i, which is represented by n_i bits. $Cell_{ij}$ has three parameters (as shown in Figure 3.3) to represent the uniform random distribution: *low limit* L_{ij}, *high limit* H_{ij}, and *weight* of generation W_{ij}. Moreover, these parameters have the following ranges:

- $n_i \epsilon [1, 32]$, number of bits to represent input i
- $w_{ij} \epsilon [0, 255]$, weight of $Cell_{ij}$
- $L_{max} < 2^{n_i}$, valid maximum range of chromosome i
- L_{ij}, $H_{ij} \epsilon [0, L_{max} - 1]$, limits range of $Cell_{ij}$
- $L_{ij} < H_{ij}$

Each chromosome encapsulates many parameters used during the evolution process including the maximum valid range L_{max} for each test generator and the

total weight of all cells. The representation of a chromosome and the mapping between a chromosome and its domain representation is illustrated in Figure 3.3.

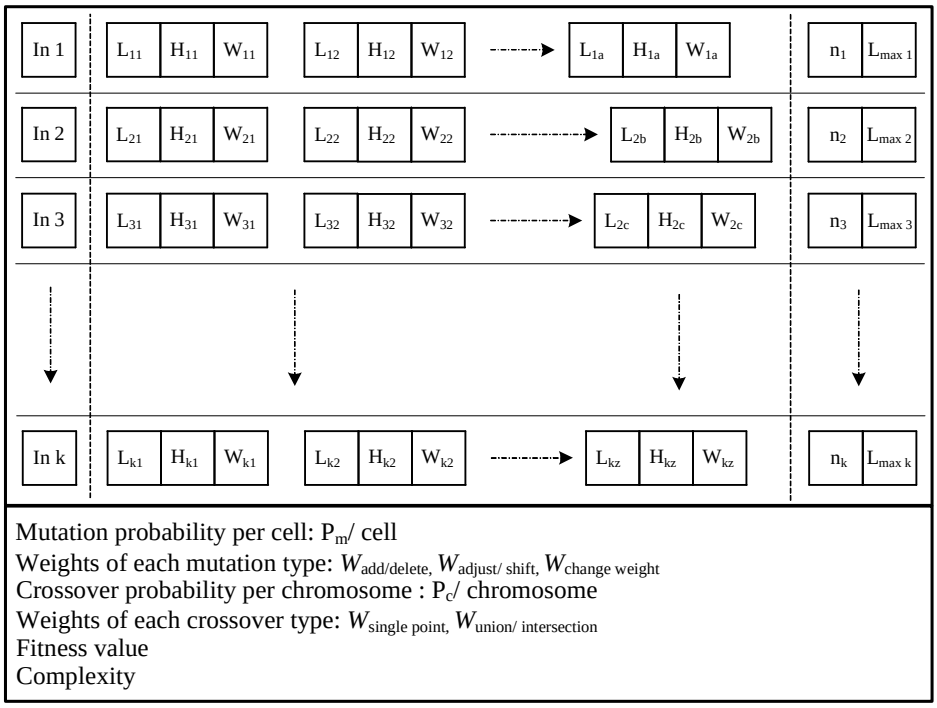


Figure 3.4: Genome Representation

Figure 3.4 depicts a *Genome*, which is a collection of many chromosomes each representing one of the test generators. Each genome is assigned a fitness value that reflects its quality. A genome also holds many essential parameters required for the evolution process. These parameters include the complexity of chromosomes which equals the number of cells in it, the mutation probability P_m of a cell, the crossover probability P_c of a chromosome, and the weights (W) of generation of each type of mutation and crossover. The values of the probabilities and the weights are constant during the evolution process.

3.2.2 Initialization

The CGA starts with an initial random population either by dividing the input range into equal periods where one random cell resides within each period, or by generating a sequence of random cells residing within arbitrary ranges.

Fixed Period Random Initialization

Given that $testGen_i$ spans over the range $[0, L_{max}]$ and is represented by n_i bits, we divide the whole range of $testGen_i$ into n_i equal sub-ranges and then we generate a random initial cell within each sub-range as shown in Figure 3.5.

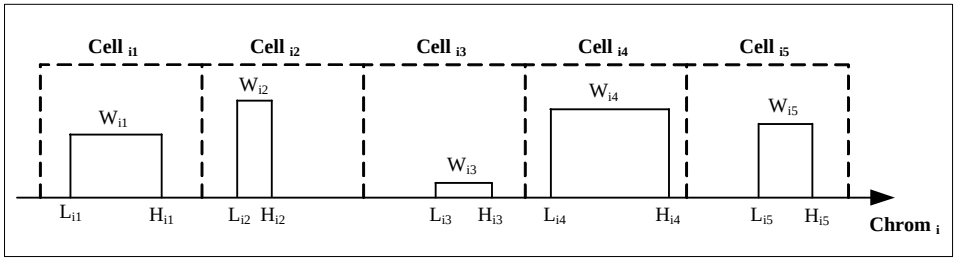


Figure 3.5: Fixed-Period Random Initialization

This configuration is biased (not uniformly random) and as such may not guarantee complete coverage of the whole input space. On the other hand, it ensures that an input with a wide range will be represented by many cells that can describe more complex paradigms.

Random Period Random Initialization

Here, we generate a random initial cell within the useful range $[0, L_{max}]$; this new cell will span over the range $[L_{i0}, H_{i0}]$. Then, we generate the following cells within the range $[H_{ij}, L_{max}]$ until we reach the maximum range limit L_{max}. In other words, the low limit of each cell must come after the end of the pervious cell. The maximum number of allowed cells is defined by the user.

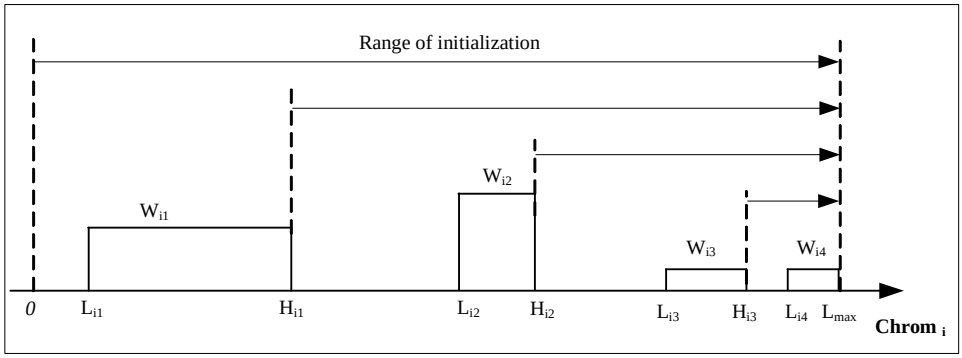


Figure 3.6: Random-Period Random Initialization

3.2.3 Selection and Elitism

For reproduction purposes, the CGA employs the roulette wheel selection (fitness proportionate selection) and tournament selection methods. With the tournament selection, the selection pressure of highly fitted individuals can be controlled by changing the tournament size. Besides, we use elitism to guarantee that the fittest individuals will be copied, unchanged, to the next generation, hence ensuring a continuously non-decreasing maximum fitness.

3.2.4 Crossover

Crossover operators are applied to each chromosome with a probability P_c. We define two types of crossover: (1) *single point crossover*, where cells are exchanged between two chromosomes; and (2) *inter-cell crossover*, where cells are merged together. Moreover, we assign two predefined constant weights: $W_{cross-1}$ to single point crossover and $W_{cross-2}$ to inter cell crossover. The selection of either type depends on these weights. Therefore, we uniformly randomly generate a number N ϵ $[1, W_{cross-1} + W_{cross-2}]$, and select a crossover operators as follows:

$$\left\{\begin{array}{lll} \textbf{If} & N \in [1,\ W_{cross-1}] & \textbf{Then} \\ & \textbf{Type I}\text{: Single Point Crossover} & \\ \textbf{Else If} & N \in (W_{cross-1},\ W_{cross-1} + W_{cross-2}] & \textbf{Then} \\ & \textbf{Type II}\text{: Inter-Cell Crossover} & \end{array}\right\}$$

Single Point Crossover

Single point crossover is similar to a typical crossover operator, where each chromosome is divided into two parts and an exchange of these parts between two parent chromosomes takes place around the crossover point as shown in Figure 3.7. The crossover algorithm starts by generating a random number $C \in [0, L_{max}]$, then it searches for the position of the point C among the cells of the involved chromosomes in the crossover operation. If point C lies within the range of $Cell_{ij}$ that is in $[L_{ij}, H_{ij}]$, then this cell will be split into two cells $[L_{ij}, C]$ and $[C, H_{ij}]$ around point C as shown in Figure 3.7. Finally, an exchange of cells between the two involved chromosomes takes place around point C, to produce a new chromosome. At this point, the complexity of the solution as well as the total weights of the various cells must be computed for future use.

Inter-Cell Crossover

Inter-cell crossover is a merging of two chromosomes rather than an exchange of parts of two chromosomes. Given two chromosomes $Chrom_i$ and $Chrom_j$, we define two types of merging as follows:

- Merging by Union ($Chrom_i \cup Chrom_j$): combines $Chrom_i$ and $Chrom_j$ while replacing the overlapped cells with only one averaged weighted cell to reduce the complexity of the solution and to produce a less constrained random test

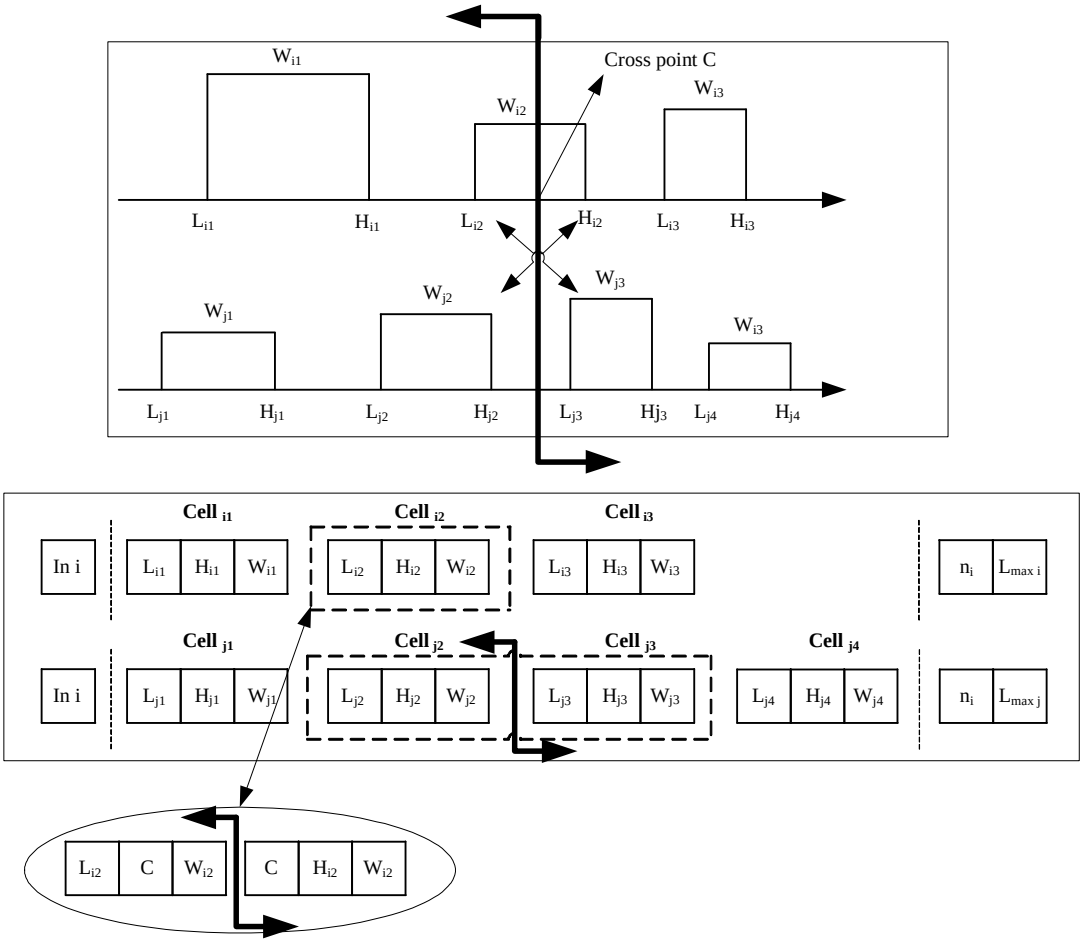


Figure 3.7: Single Point Crossover

generator. The weight of a new merged cell will be proportional to the relative width and weight of each cell involved in the metering process as illustrated in Figure 3.8.

- Merging by Intersection ($Chrom_i \cap Chrom_j$): extracts averaged weighted cells of the common parts between $Chrom_i$ and $Chrom_j$, where the new weight is the average of the weights of the overlapped cells. This will produce a more constrained random test generator.

The inter-cell crossover operation is illustrated in Figure 3.8. This type of crossover is more effective in producing a fitter offspring, since it shares information

between chromosomes along the whole range rather than at a single crossover point.

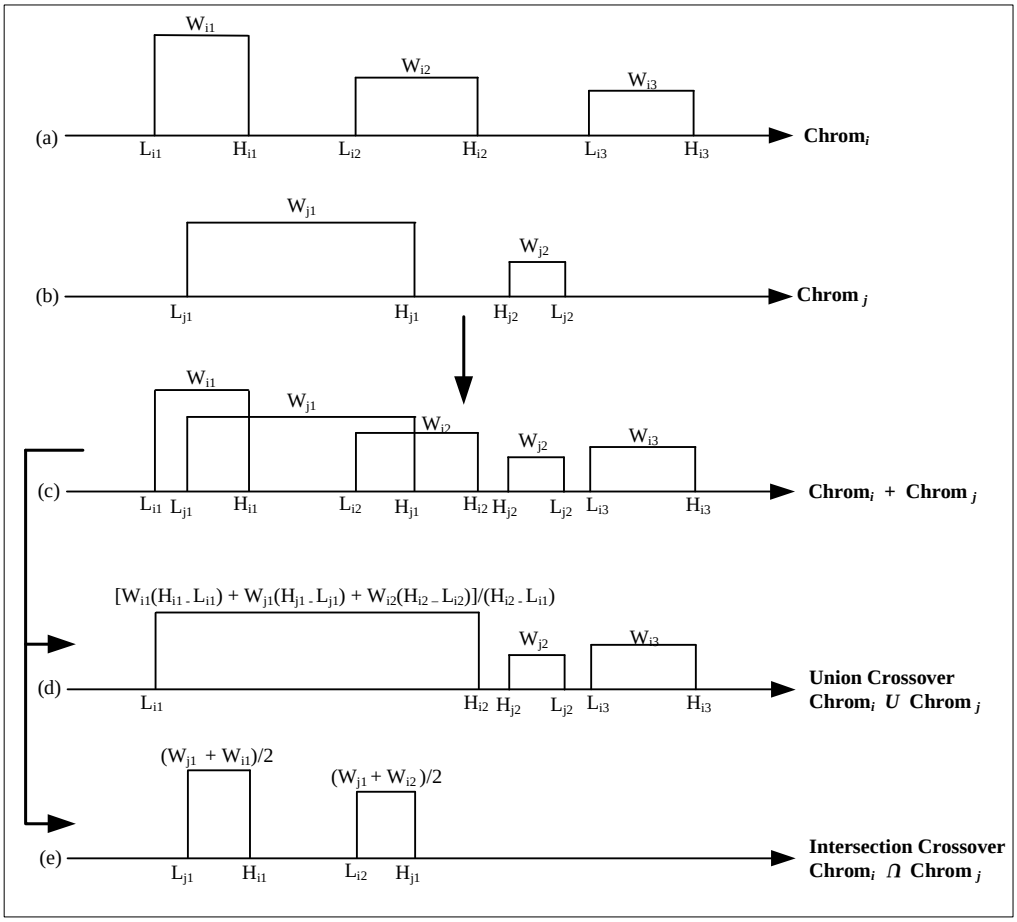


Figure 3.8: Inter-Cell Crossover

Next, we explain the procedure to find out the union and intersection of two chromosomes. Given a $Cell_{ij}$ spanning over the range $[L_{ij}, H_{ij}]$, a random number $N \epsilon [0, L_{max}]$ may lay in one of the following three regions with respect to $Cell_{ij}$, as shown in Figure 3.9:

- $Region0$, if $N < L_{ij}$
- $Region1$, if $(N \geq L_{ij})$ and $(N \leq H_{ij})$
- $Region2$, if $N > H_{ij}$

For example, to find the possible intersection of $Cell_{ik}$ with $Cell_{ij}$, the procedure searches for the relative position of L_{ik} (low limit of $Cell_{ik}$) with respect to $Cell_{ij}$ starting from $Region2$ down to $Region0$ for the purpose of reducing the computational cost. If L_{ik} lies in $Region2$, then no intersection or merging is possible, else the procedure proceeds and searches for the relative position of H_{ik} (high limit of $Cell_{ik}$) with respect to $Cell_{ij}$. According to the relative position of both L_{ik} and H_{ik} with respect to $Cell_{ij}$, the procedure decides whether there is an intersection, a merging or neither, and whether it has to check the relative position of $Cell_{ik}$ with respect to the successive cells of $Cell_{ij}$ for possible intersection or merging regions.

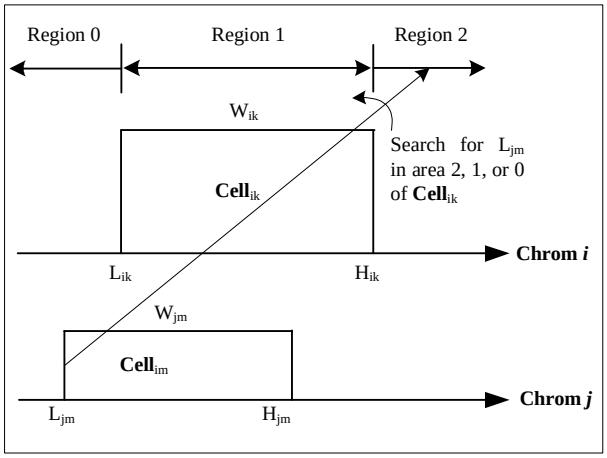


Figure 3.9: Procedures for Inter-Cell Crossover

3.2.5 Mutation

Due to the complex nature of our genotype and phenotype, we propose many mutation operators that are able to mutate the low limit, high limit, and the weight of the cells. Mutation is applied to individual cells with a probability P_m per cell. This is in contrast to crossover operators, which are applied to pairs of chromosomes. Moreover, the mutation rate is proportional to the complexity of the chromosome (number of cells) which make complex chromosomes more likely to be mutated than

simpler ones. When a cell is chosen for mutation, we choose one of the following mutation operators:

- Insert or delete a cell.
- Shift or adjust a cell.
- Change cell's weight.

The selection of which mutation operator to use is based on predefined weights associated with each of the operators. Given W_{mut-1}, W_{mut-2}, and W_{mut-3}, which represent the weights of the above mutation operators, respectively, we generate a uniform random number $N \epsilon [1, W_{mut-1} + W_{mut-2} + W_{mut-3}]$ to choose the mutation operators as follows:

$$\left\{\begin{array}{lll} \textbf{If} & N \epsilon [1,\ W_{mut-1}] & \textbf{Then} \\ & \textbf{Type I}\text{: Insert cell or delete cell} & \\ \textbf{Else If} & N \epsilon (W_{mut-1},\ W_{mut-1} + W_{mut-2}] & \textbf{Then} \\ & \textbf{Type II}\text{: Shift cell or adjust cell} & \\ \textbf{Else If} & N \epsilon (W_{mut-1} + +W_{mut-2},\ W_{mut-1} + W_{mut-2} + W_{mut-3}] & \textbf{Then} \\ & \textbf{Type III}\text{: Change cell's weight} & \end{array}\right\}$$

Insert or Delete a Cell

This mutation operator is either delete $Cell_{ij}$ or insert a new cell around it. Moreover, we select either insertion or deletion with equal probability. If deletion is chosen, we pop $Cell_{ij}$ out of the chromosome and proceed for the next cell to check the applicability of mutation to it. When insertion is selected, we insert a new randomly generated cell either behind or next to $Cell_{ij}$. This new random cell must

reside within the gap between $Cell_{ij}$ and the previous or next cell according to the relative position of this new cell, with respect to $Cell_{ij}$.

Shift or Adjust a Cell

If $Cell_{ij}$ is chosen for this type of mutation, then either we shift or adjust $Cell_{ij}$ with equal probability. This type of mutation affects one or both limits of $Cell_{ij}$, but it does not affect its weight. Moreover, if shifting is selected, then we equally modify both the low and high limits of $Cell_{ij}$ within the range of high limit H_{ij-1} of the previous cell and low limit L_{ij+1} of the next cell to $Cell_{ij}$. On the other hand, if adjusting is selected, we choose randomly either the low or high limit of $Cell_{ij}$, and then we modify the chosen limit within a range that prevents overlapping between the modified cell and other cells of the chromosome.

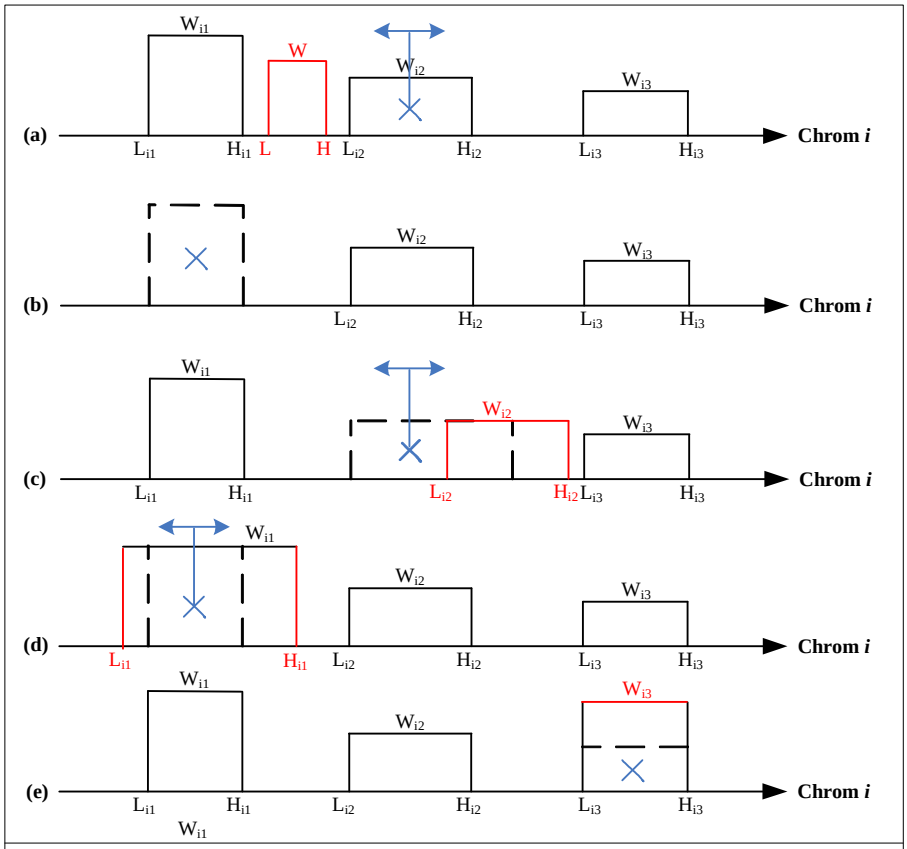
Change Cell's Weight

This mutation operation replaces the weight of $Cell_{ij}$ with a new randomly generated weight within the range $[0, 255]$, as eight bits value is used to represent the weight.

3.2.6 Fitness Evaluation

The potential solution of the CDG problem is a sequence of weighted cells that direct the generation of random test pattern to maximize the coverage rate of a group of coverage points. The evaluation of such a solution is somehow like a decision making problem where the main goal is to activate all coverage points among the coverage group and to maximize the average coverage rate for all points.

The average coverage rate is not a good evaluation function to discriminate potential solutions when there are many coverage points to consider simultaneously. For instance, we may achieve 100% for some coverage points while leaving other points totally inactivated. For example, given a coverage group consisting of three coverage points, according to the average coverage criterion, a potential solution of



(a) Add: Add a randomly generated cell either before or after the current cell.
(b) Delete: Delete the current cell.
(c) Shift: Shift the current cell either to the left or to the right between the gap of previous cell and next cell.
(d) Adjust: Adjust either the low or the high limit of the current cell.
(e) Change weight: Change the weight of the current cell randomly.

Figure 3.10: Mutation Operators

coverage rates (95%, 88%, 0%) is better than another potential solution of coverage rates (44% 39%, 20%) since the first solution has a higher average coverage rate than the second one. However, the first solution activates only two points while the second one activates the whole coverage group with different coverage rates.

We employ two methods for the fitness function evaluation that aims to maximize the average coverage rate, but also achieving an adequate coverage rate for each coverage point. The first method utilizes multi-stage evaluation where a different criterion is used during each stage. This is to ensure that the algorithm will

activate all coverage tasks before tending to maximize the total coverage rate in one direction only. The second method is simply based on maximizing the coverage rate while minimizing the weighted standard deviation between coverage points by using a difference equation.

Multi-Stage Evaluation

We discriminate potential solutions differently over four stages of evaluation (Figure 3.11), where the criterion of stage i cannot be applied before achieving the goal of stage $i-1$. The first three stages aim at achieving a minimum coverage rate for each coverage task within a coverage group according to a predefined threshold coverage rate provided by the verification engineer. In the last stage, we aim at maximizing the average coverage rate for the whole coverage group without ignoring the threshold coverage rates. The four stage evaluation can be described as follows:

1. Find a solution that activates all coverage points at least one time.
2. Push the solution towards activating all coverage points according to a predefined coverage rate threshold *CovRate1*.
3. Push the solution towards activating all coverage points according to a predefined coverage rate threshold *CovRate2* which is higher than *CovRate1*.
4. After achieving these three goals, we consider the average number of activation of each coverage point.

Figure 3.11 provides an illustration of the evaluation of fitness values over each stage. For instance, we assume a maximum fitness value of 10,000. The portion of the first three stages is defined by $stageW$, usually 1000, while the portion of the last stage is equal to 10,000 - $3 * stageW$. Moreover, $stageW$ is equally divided between all coverage points within a coverage group over the first three stages. The last stage scales the average coverage value of the coverage group over its portion.

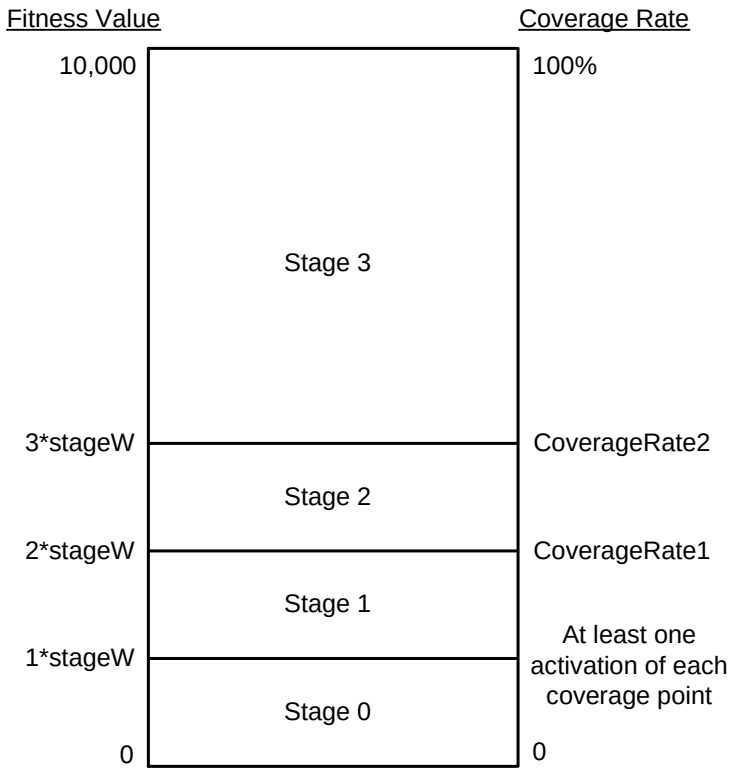


Figure 3.11: Multi Stage Fitness Evaluation

During the last stage of evaluation and in order to favor potential solutions, we use the *linear* or the *square root* scheme [48]. The linear scheme will give the same weight for all activations, while the square root scheme will give lower weights for the successive activation of coverage points. Given a coverage group of $noCov$ coverage points, each with a coverage rate $cov[i]$ and a weight $w[i]$, then the unscaled last stage evaluation can be given by:

- Linear fitness evaluation:

$$fitness = \frac{\sum_{i=0}^{noCov} w[i] * cov[i]}{\sum_{i=0}^{noCov} w[i]} \tag{3.1}$$

- Square root evaluation:

$$fitness = \sqrt{\frac{\sum_{i=0}^{noCov} (w[i] * cov[i])^2}{\sum_{i=0}^{noCov} w[i]}} \tag{3.2}$$

We can enable or disable the second and or the third stage that defines the minimum acceptable coverage rates through the configuration file of the CGA. Algorithm 3.2 below provides a pseudo-code for the multi stage evaluation function in case both the second and the third stages are enabled.

Mean-Weighted Standard Deviation Difference Evaluation

We use a simple equation for fitness evaluation that maximizes the average coverage rate of the whole coverage group and simultaneously minimizes the weighted standard deviation of the coverage rate among the coverage points as follows:

$$fitness = avgCov - k * stdCov \tag{3.3}$$

where k is a constant real number ϵ $[0.5, 1.5]$ and

$$avgCov = \frac{\sum_{i=0}^{noCov} w[i] * cov[i]}{\sum_{i=0}^{noCov} w[i]} \tag{3.4}$$

$$stdCov = \sqrt{\frac{\sum_{i=0}^{noCov} (w[i] * cov[i] - avgCov)^2}{\sum_{i=0}^{noCov} w[i] \ - \ 1}} \tag{3.5}$$

Accordingly, as the average coverage increases and the variations among coverage points decrease, we achieve a better fitness value and obtain a better solution. The weight of standard deviation is tuned by using a constant real number k in the range $[0.5, 1.5]$. A small value of k may not direct the CGA towards an effective solution, while a large value of k may introduce obstacles in the way of finding an effective solution and distracts the CGA from the proper features of the solution.

Algorithm 2: Fitness Evaluation Algorithm

input : $noCov$ Coverage Points and their Rates
input : Coverage Boundaries $CoverageRate1$ and $CoverageRate2$
input : Stage Weight $stageW$
input : $EnableCoverageStage1$ and $EnableCoverageStage2$ are Enabled
input : Maximum Fitness is 10,000
output: Fitness Evaluation

Define Three Stages According to the Coverage Boundaries
stage 1: Coverage point rates ϵ $(0, CoverageRate1]$
stage 2: Coverage point rates ϵ $(CoverageRate1, CoverageRate2]$
stage 3: Coverage point rates ϵ $(CoverageRate2, 100\%]$

Evaluate Fitness
switch *Coverage rates* **do**
    case $\forall$ *Coverage point rates* ϵ *stage 3*
        $fitness = 3.0 * stageW + [\frac{10{,}000-(3*stageW)}{100} * AveragedCoverageRate]$
    end
    case $\forall$ *Coverage point rates* ϵ *stage 2*
        $fitness = 2.0 * stageW$
        foreach *Coverage point rates* ϵ *stage 3* **do**
        $fitness = fitness + \frac{stageW}{noCov}$
    end
    case $\forall$ *Coverage point rates* ϵ *stage 1*
        $fitness = 1.0 * stageW$
        foreach *Coverage point rates* ϵ *stage 2* **do**
        $fitness = fitness + \frac{stageW}{noCov}$
    end
    otherwise
        $fitness = 2.0 * stageW$
        foreach *Coverage point rates* ϵ *stage 1* **do**
        $fitness = fitness + \frac{stageW}{noCov}$
    end
end

3.2.7 Termination Criterion

The termination criterion determines whether the CGA terminates and report the best solution or continues evolving over another generation till obtaining a better potential solution or reaching maximum generations. It checks for the existence of a potential solution that is able to achieve 100% or another predefined value of coverage rate for all coverage groups. If the CGA runs until it reaches the maximum allowed number of generations, it terminates and reports the best solution out of the evolution process regardless of the fitness value and coverage rate.

3.2.8 CGA Parameters

An important problem in our CGA is how to tune various genetic parameters. This is a nontrivial task and frequently requires a separate optimization algorithm to determine the optimal parameters setting. The primary parameters for controlling the CGA are population size N, maximum number of generations to be run G, crossover probability P_c, mutation probability P_m, and elitism probability P_e. Those parameters affect the speed of the evolution towards finding a near optimal solution, escaping from local optimum, and even the convergence of the algorithm.

The Population size N determines the amount of information stored by the CGA. It critically affects the efficiency and solution quality of the CGA. If N is too small, hence insufficient samples are provided (i.e., solution diversity is low), then the genetic evolution will be degenerated or no useful result can be obtained; if N is too large, then the amount of computation time needed may exceed a tolerable limit, and, more importantly, the time for the convergence could be prolonged.

The crossover probability P_c controls the frequency of the crossover operation. If P_c is too large, then the structure of a high quality solution could be prematurely destroyed; if P_c is too small, then the searching efficiency can be very low. Furthermore, the mutation probability P_m is a critical factor in extending the diversity of

the population (hence, helping the optimization search to escape from local optima). In fact, the genetic evolution may degenerate into a random local search if P_m is too large. The elitism probability P_e is usually small as the elitism simply copies the best potential solutions and forwards them to the next generation. Table 3.1 shows the typically used values of above primary parameters during the experiments we conducted.

Parameters	**Typical Values**
Crossover Probability	95%
Mutation Probability	20%
Elitism Probability	3%

Table 3.1: Primary Parameters

Table 3.2 shows the initialization and selection related parameters. When the tournament selection is chosen, we must fix the size of the tournament window which affects the selection pressure. Finally, if we employ free period initialization, we must specify the maximum number of allowed cells (for each input) during the generating of the initial population.

Parameters	**Typical Values**
Creation Type	Free Initialization/Restricted Initialization
Number of Cells	25
Selection Type	Probabilistic Selection/Tournament Selection
Tournament Size	5
Enable Elitism	True/False

Table 3.2: Initialization and Selection Parameters

During the evaluation phase of a potential solutions, the CGA stimulates the DUV for a number of predefined simulation cycles. The number of those simulation cycles depends mainly on the complexity of the DUV as well as the complexity of the associated properties and coverage tasks to be activated.

Parameters	Typical Values
Fitness Definition	Multi-Stage/Mean-Standard Deviation
Standard Deviation Weight	0.5 - 1.5
Fitness Evolution	Linear/Root Square
CoverageRate1	10 - 20
CoverageRate2	10 - 30
Stage Weight	1000.0
Enable CoverageStage1	True/False
Enable CoverageStage2	True/False

Table 3.3: Evaluation Function Parameters

We define many parameters for the evaluation function of the CGA. For instance, the fitness definition parameter specifies whether the multi-stage strategy or the mean-standard deviation strategy is selected for evaluating the effectiveness of potential solutions. When the multi-stage strategy is selected, other parameters must be specified. These includes the enabling of the second and/or third stage of the evolution function, and the specifying of the threshold coverage rates *CoverageRate1* and *CoverageRate2* for both stages. Moreover, if the mean-standard deviation strategy is selected, we only define the weight of the standard deviation of the coverage groups. Table 3.3 provides typical values of fitness evaluation parameters.

3.3 Random Number Generator

Random number generators are heavily used in simulation based verification like the Specman environment [48]. They are also used by genetic algorithms and other evolutionary techniques during the process of evolution and learning. A poor random generator will not be useful for simulation based verification where sequences tend to be repeated within a short cycle that may be shorter than the simulation runs. Besides, a poor random generator might drive genetic algorithms to the same local optima.

Random sequences chosen with equal probability from a finite set of numbers are called uniform random sequences. Also, random sequences generated in a deterministic way are often called *pseudo-random*. These pseudo-random numbers are not completely random, they must fulfill four essential properties to be considered sufficiently random for practical purposes:

1- Repeatability: the same sequence should be produced with the same seeds values. This is vital for debugging and regression tests.

2- Randomness: should produce high quality random sequences that pass all theoretical and empirical statistical tests for randomness like frequency test, serial test, gap test, and permutation test [33].

3- Long period: all pseudo-random number sequences repeat themselves with a finite period that should be much longer than the amount of random numbers needed for the simulation.

4- Insensitive to seeds: period and randomness properties should not depend on the initial seeds.

In our CGA, we adopt the Mersenne Twisted (MT19937) pseudo-random generator [37] for stochastic decision during the learning and evolution processes as well as during the simulation and verification phases. MT19937 is a pseudo-random number generating algorithm designed for fast generation of very high quality pseudo-random numbers. Moreover, the algorithm has a very high order (623) of dimensional equidistribution and a very long period of 2^{19937} - 1.

MT19937 is faster than the most random generators [37] and it overperforms other well-known algorithms like Linear Congruential [33] and Subtractive Method [33] in terms of quality and speed. This is because the MT19937 algorithm mainly uses efficient and fast arithmetic operations (addition, shifting, bitwise and, and bitwise exclusive or) and avoids the use of multiplication or division operations. The following pseudo-code (Figure 3.12) generates uniformly 32 bit integers in the range $[0, 2^{32} - 1]$ using the MT19937 algorithm [37]:

```
%Create a length 624 array to store the state of the generator
1:  var int[0..623] MT

% Initialize the generator from a seed
2:  Function initializeGenerator ( 32-bit int seed )
3:      MT[0] := seed
4:      For i := 1 : 623 // loop over each other element
5:          MT[i] := [last 32bits of (69069 * MT[i-1])]
6:  EndFunction

%Generate an array of 624 untempered numbers
7:  Function generateNumbers()
8:      For i = 0 : 622
9:          y := [32nd bit of (MT[i])] + [last 31bits of(MT[i+1])]
10:         If y even
11              MT[i] := MT[(i + 397) % 624] xor (y ≫ 1)
12:         else if y odd
13:             MT[i] := MT[(i + 397) % 624] xor (y ≫ 1) xor (2567483615)
14:         EndIf
15:     EndFor
16:     y := [32nd bit of(MT[623])] + [last 31bits of(MT[0])]
17:     If y even
18:         MT[623] := MT[396] xor (y ≫ 1)
19:     else if y odd
20:         MT[623] := MT[396] xor (y ≫ 1) xor (2567483615)
21:     EndIf
22: EndFunction

%Extract a tempered pseudo-random number based on the i-th value
23: Function extractNumber(int i)
24:     y := MT[i]
25:     y := y xor (y ≫ 11)
26:     y := y xor ((y ≪ 7) and (2636928640))
27:     y := y xor ((y ≪ 15) and (4022730752))
28:     y := y xor (y ≫ 18)
29:     return y
30: EndFunction
```

Figure 3.12: Pseudo Random Generation Using Mersenne Twisted Algorithm [37]

3.4 Summary

In this chapter, we described our methodology for improving the design verification cycle and automating coverage directed test generation for a group of coverage points (capturing design functionalities). Then, we described the cell-based genetic algorithm which closes the feedback path between the coverage space and the generator directives space within the CDG framework. Also, we discussed the details of the proposed algorithm, which continuously reads and analyzes the coverage reports from many simulation runs and then modifies the test generator directives accordingly.

In this chapter, we also introduced a unique solution representation for the CGD problem based on many directives (cells) associated with each design input. We presented and discussed the employed representation, selection methods, initialization schemes, crossover operators, mutation operators, and the ways of evaluating the potential solutions of the CDG problems. As the random number generation is very important for simulation based verification and for the evolution process of our CGA, we used the Mersenne Twisted pseudo-random generator, and highlighted its randomness properties that ensure its quality.

In the next chapter, we present the application of our methodology on designs modeled in SystemC. We discuss many experiments that show the ability of the CGA to compete with industrial tools. Also, we show the effectiveness and applicability of our CGA for both data and control coverage points.

Chapter 4

Experimental Results

In this chapter we illustrate the effectiveness of our CGA through several experimentations. We use the SystemC language to model and verify three designs: a small CPU, a router, and a master/slave architecture. In the next section, we introduce the emergence need for SystemC as a system level language, then we briefly preview the SystemC architecture. In the second section, we discuss the conducted experimental results.

4.1 SystemC

4.1.1 The Emergence Need for SystemC

The speedy growth of embedded system and the emergence of SoC era are creating many new challenges at all phases of the design flow. It became vital necessity to reconsider the methodologies and tools for specifying, partitioning, implementing, and verifying the embedded systems [23]. In this context, SystemC emerged as one of the best solution for representing functionality, communication, and software and hardware implementations at various levels of abstraction [23]. Furthermore, SystemC can be used as a common language by system designers, software engineers,

and hardware designers and verifiers [23]. This will facilitate hardware software co-simulation and make interoperable system-level design tools, services and intellectual property a reality.

SystemC is an open source standard being developed and maintained under the OSCI organization. It has been recently standardized by IEEE (IEEE 1666) [30] for system-level chip design. SystemC is both an object-oriented system description language and a simulation kernel for the purpose of modeling and simulating embedded systems at various levels of abstraction from the abstract untimed models to cycle-accurate RTL models. The SystemC library extends the capabilities of C++ by introducing important concepts to C++ as concurrency (multiple processes executing concurrently), timed events and hardware data types.

The SystemC library provides constructs that are familiar to a hardware designer such as signals, modules and ports. The model, which is the executable specification, can then be compiled and linked in with the SystemC simulation kernel and SystemC library to create an executable code that behaves like the described model when it is run [9]. Most importantly that standard C++ development environment such as Microsoft Visual C++ or GNU GCC can be used to develop models of hardware, software, and interfaces for simulation and debugging (Figure 4.1.)

SystemC is based on C++, and hence it is powerful enough to describe all kinds of verification environments including testbench generation at all level of abstraction. The testbench, which is also written in SystemC, can be reused to ensure that the iterative refinements of the SystemC model did not introduce any errors. Furthermore, assertions can be described in a SystemC model as FSM machine using C++ syntax. Having a single language will eliminate translation errors and make verification faster as compared to a multi-language environment which may cause slowdown of the verification process [9].

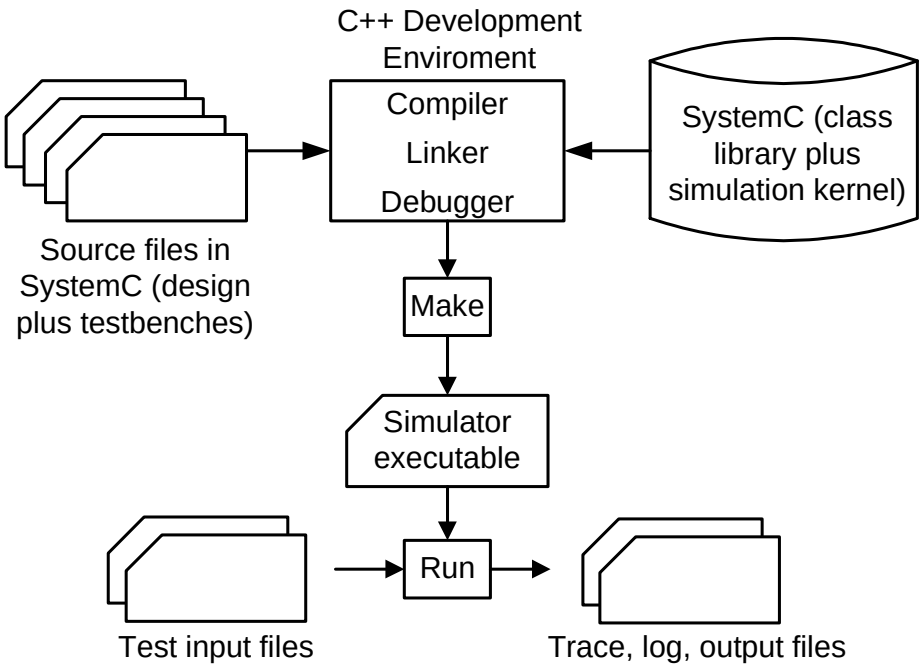


Figure 4.1: SystemC in a C++ Development Environment

4.1.2 SystemC Architecture

The SystemC language architecture is shown in Figure 4.2. The language is built on top of standard C++. The first layer of shaded gray blocks are part of the so-called core layer (or layer 0) of the standard SystemC language. The second shaded gray layer immediately after the kernel layer is the layer 1 of SystemC; it comes with a predefined set of interfaces, ports and channels. Finally, the layers of design libraries above layer 1 are considered separate from the SystemC language. The user may choose to use them or not. Over time other standard libraries may be added and conceivably be incorporated into the standard language [9].

Figure 4.3 gives a description of a simplified model of SystemC design and components. SystemC has a notion of a container class, called *module*, that provides the ability to encapsulate structure and functionality of hardware/software blocks for partitioning system designs. A system is essentially broken down into a containment hierarchy of *modules*. Each module may contain *variables* as simple data members,

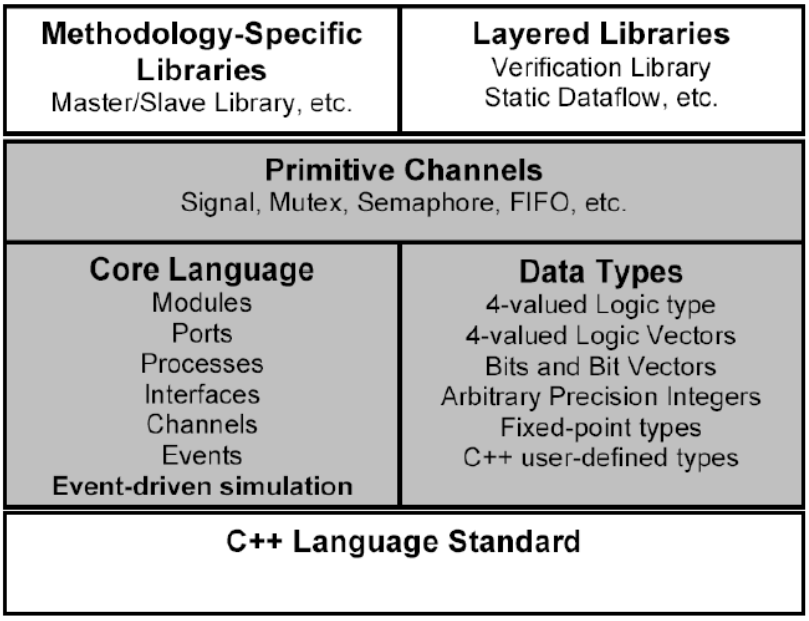


Figure 4.2: SystemC Architecture

ports for communication with the outside environment and *processes* for performing modules functionality and expressing concurrency in the system.

Three kinds of processes are available: *method* processes, *thread* processes, *clocked thread* processes. They run concurrently in the design and may be sensitive to events which are notified by *channels*. A port of a module is a proxy object through which the process accesses a channel interface. The *interface* defines the set of access functions (methods) for a channel, while the channel provides the implementation of these functions to serve as a container to encapsulate the communication of blocks. There are two kinds of channels: *primitive* channels and *hierarchical* channels. Primitive channels do not exhibit any visible structure, do not contain processes, and cannot (directly) access other primitive channels. A hierarchical channel is a module, i.e., it can have structure, it can contain processes, and it can directly access other channels [9].

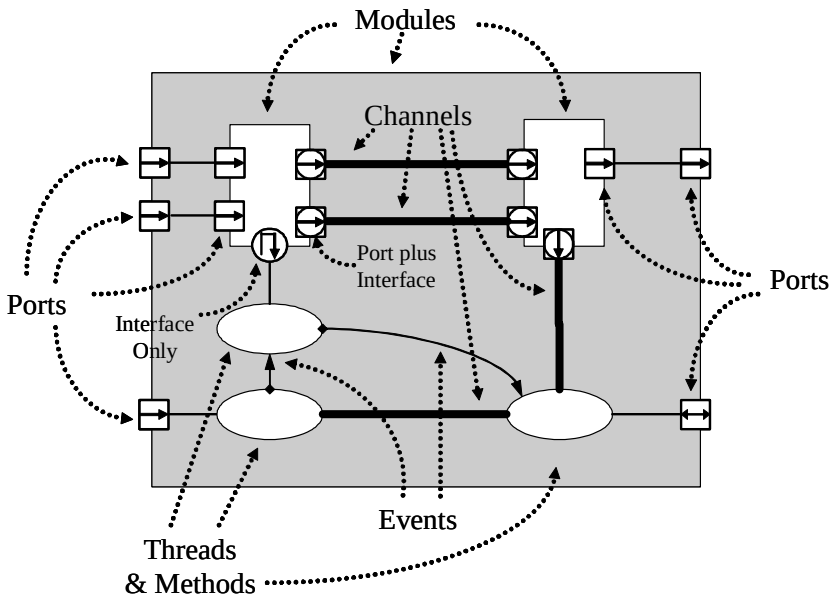


Figure 4.3: SystemC Components

4.2 CGA Implementation and Tuning

4.2.1 CGA Implementation

We implemented the CGA using the C++ programming language and the standard template library (STL). We followed the object oriented programming (OOP) approach. Moreover, we designed many classes for encoding the potential solutions, performing the CGA operations, generating pseudo random numbers, and making the CGA algorithm configurable. The DUV's are written as derived C++ classes from SystemC modules.

The potential solutions are represented using three template classes: Cell class, Chromosome class, and Genome class. Each of these classes can be instantiated with different numbers of bits according to the input domains. The CGA operations are enclosed within a major CGA class. This class supports two initialization schemes, two selection schemes, two crossover operations, and three mutation operations. Also, it offers two methods of fitness evaluation and other miscellaneous functions

for printing and reporting statistical results of the CGA performance.

For pseudo-random generation, we enclosed the Merssene Twisted algorithm within a separate class that can produce 32-bit integer or double precision random numbers. Furthermore, there are three classes for storing, reading, and printing the CGA parameters and configurations. Users of the CGA can modify those parameters and configurations in order to get better results.

The DUV classes are written using the SystemC class library. For each design, at least two classes are needed: one for the DUV and the other for testbench. Within the SystemC main function, we instantiate the DUV and its testbench, make the proper connections between modules, and then run the CGA for many generations till the CGA either reaches the maximum number of generations or satisfies the termination criterion. During the CGA run time, we invoke the SystemC simulator for evaluation of the potential solutions.

4.2.2 CGA Tuning

As we presented early in Section 3.2.8, there are many parameters that control the operation of the CGA and affect its convergence to a proper optimal solution. However, some of these parameters have dominant effects on the CGA performance while others have minor effects and so can be kept constant for all experiments. In this sub-section, we highlight how to choose specific parameters and give some clues and guidelines for the users of the CGA in order to use it efficiently.

We believe that the population size N is the most important parameter that determines the amount of information stored and searched by the CGA, and hence N determines the amount of computation time needed to evolve a new generation and to converge to an optimal solution. The optimal population size can vary during the evolution process. Our rule of thumb is to use a population size proportional to the search space. According to the conducted experiments, we find that a population size of 50 individuals is suitable for a 10 bits search space. While we employ a

population size of 500 individuals for a 16 bit search space to converge rapidly, .

The real bottleneck in the CGA performance is the time required for the evaluation phase of the potential solutions rather than the time required for producing new generations of solutions using the CGA operations. Furthermore, in order to evaluate each potential solution, we need to stimulate the DUV for a number of simulation cycles depending on the complexity of coverage task as well as the details of the implementations. Consequently, we do recommend to use an abstract model using ,e.g. , Transaction Level Modeling (TLM) rather than RTL modeling of the DUV in order to reduce the computation time during the evaluation phase and so achieve faster convergence to an optimal solution.

According to our deep understanding of the CGA framework and after running many experiments, we think that the weights of generating various crossover and mutation operators can be fixed during most experiments. For instance, we favor the generation of "inter-cell crossover" over "single point crossover." The former is able to exchange the proper features over many points, and so it is able affect many cells, rather than around a single point. For mutation operators, the CGA achieves better performance when we favor the "adjust or shift a cell" over "add or delete a cell". The "change cell's weight" has the least importance with respect to other mutation operators. Table 4.1 summarizes the used weights for generating various crossover and mutation operators.

Parameters	**Typical Values**
crossover Weight1	1
crossover Weight2	2
mutation Weight1	2
mutation Weight2	3
mutation Weight3	1

Table 4.1: Genetic Operators Weights

4.3 Small CPU

In this section, we use a small CPU example in order to compare the results of the CGA with the Specman Elite [48], which is a widely used industrial tools. Specman Elite uses the e-language [44] to define environmental constraints, direct the random test generation, and collect coverage results. The small CPU model, we consider, is part of the Specman Elite tutorial. It is an 8-bit CPU with a reduced instructions set (Table 4.2). Figure 4.4 shows its block diagram and Figure 4.5 illustrates its control state machine.

Name	**Opcode**	**Operands**	**Comments**
ADD	0000	register, register	ADD; PC ← PC + 1
ADDI	0001	register, immediate	ADD immediate; PC ← PC + 2
SUB	0010	register, register	SUB; PC ←PC + 1
SUBI	0011	register, immediate	SUB immediate; PC ← PC + 2
AND	0100	register, register	AND; PC ← PC + 1
AND	0100	register, register	AND; PC ← PC + 1
XOR	0110	register, register	XOR; PC ← PC + 1
XORI	0111	register, immediate	XOR immediate; PC ← PC + 2
JMP	1000	immediate	JUMP; PC ←immediate value
JMPC	1001	immediate	JUMP on carry; if carry = 1 PC ← immediate value else PC ← PC + 2
CALL	1010	immediate	Call subroutine; PC ← immediate value; PCS ← PC + 2
RET	1011		Return from call; PC ← PCS
NOP	1100		Undefined command

Table 4.2: Instructions Set of the Small CPU

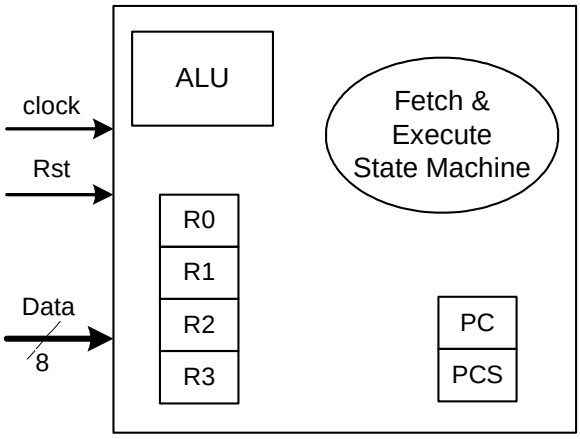


Figure 4.4: Small CPU Block Diagram

All instructions use 4-bit opcode and two operands. The first operand is one of four 4-bit registers internal to the CPU. In the case of arithmetic and logic instructions, the same register stores the result of the operation. The second fetch cycle (Figure 4.5) is only for immediate instructions and for instructions that control the execution flow.

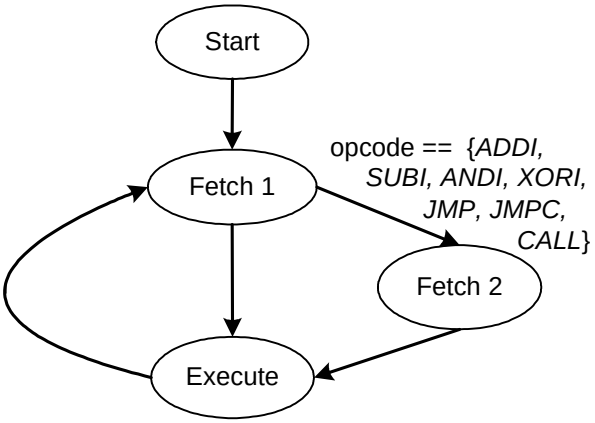


Figure 4.5: Small CPU Control State Machine

We define four types of coverage points inherited form the Specman tutorial. The first point states that the test generator must cover all states in the state machine. The second point concerns covering all possible opcodes. The third point aims to find a suitable instruction sequence in order to set and to utilize the carry flag. The fourth and last point is a cross coverage point that tries to maximize the cases of set carry flag with all addition instructions.

In Table 4.3 we present the coverage results for each of the four previously

discussed coverage points. We notice that: (1) the Specman Elite shows better coverage rates than the blind random simulation; and (2) our CGA shows better coverage results than the Specman Elite. In particular, for the last two coverage points, the CGA shows its efficiency when dealing with corner and cross coverage points that are hard to reach.

Coverage Point	**Random**	**Specman**	**our CGA**
FSM	100	100	100
Opcode	100	100	100
Carry flag is set	13	15	45
Carry flag / Add instruction	30	30-77	85

Table 4.3: Control Coverage Results for the Small CPU

We define another group of coverage points concerning ranges of specific instructions, as shown in Table 4.4. The first point aims at covering arithmetic instructions only (e.g., *ADD*, *ADDI*, *SUB*, and *SUBI*). Similarly, the second and the third points aim at considering logical instructions only and control flow instructions only, respectively. The results show that the CGA is able to achieve 100% coverage rates.

Coverage Point	**Coverage Rate**
Only arithmetic instructions	100
Only logic instructions	100
Only control flow instructions	100

Table 4.4: Range Coverage Results for the Small CPU

4.4 Router

In this section, we consider a router model. The aim here is to evaluate the performance of our CGA when dealing with coverage points that may invoke conflicts

while optimizing the input ranges. The router gets input packets and directs them to one of 16 output channels, according to their address fields. A packet is composed of four fields: destination address (4 bits), packet's length (8 bits), data (multiple of 8 bits) and parity byte (8 bits). Figure 4.6 shows the block diagram of the router.

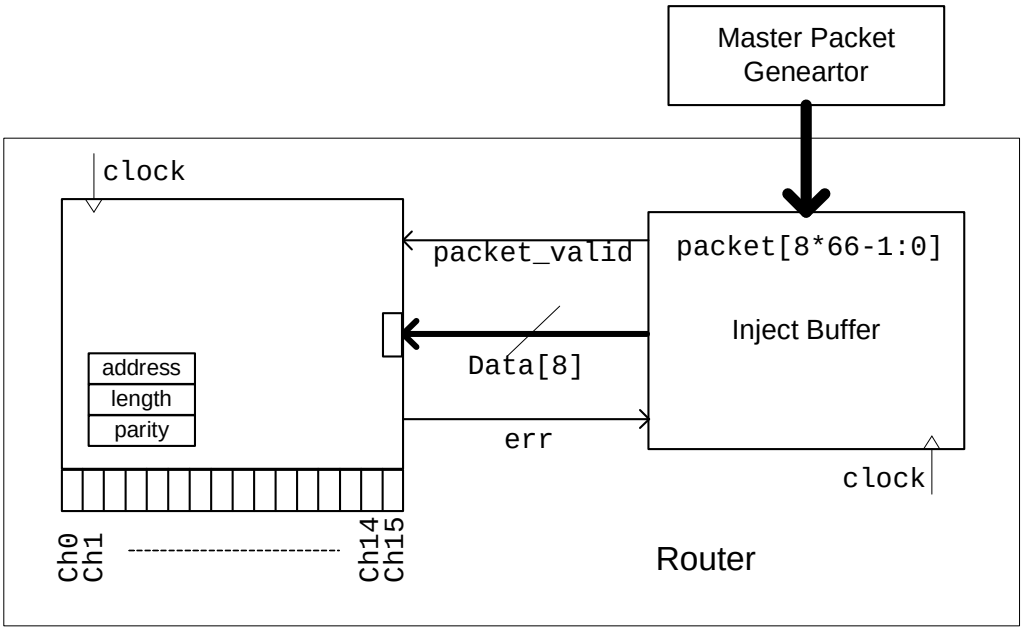


Figure 4.6: Router Block Diagram

The packets are submitted, one at a time, to the Inject buffer, which splits them into a byte stream. The router module latches the first two bytes of the packet and sends the data to the channel specified in the address field. The Inject buffer, asserts the packet_valid signal at the begging of the first byte (the header) and negates it at the beginning of the parity byte. The Master packet generator passes each packet to the Inject buffer and waits until packet_valid signal is negated before submitting the next packet. The following code represents the router interface written in SystemC.

```
#include "systemc.h"
#define ChNo 16
```

```
class router : public sc_module { public:
    sc_in<bool>         clock;
    sc_in<bool>         packet_valid;
    sc_in<sc_uint<8> >  data_in;
    sc_out<sc_uint<8> > channel[ChNo];
    sc_out<bool>        error;

    sc_uint<8>          parity;
    sc_uint<4>          address;
    sc_uint<6>          length;

    void    compParity();
    void    compAdd();
    void    compLength();
    void    compError();

    SC_CTOR(router)
    {
        .....
    }
};
```

Four coverage points have been defined in order to check the appearance of outputs on various channels of the router, i.e., the packet destination address. Furthermore, the last two points add more constraints by checking not only the packet address but also the packet length. Table 4.5 summarizes the coverage results obtained using our CGA algorithm. The experimental results confirm that our CGA succeeded in finding the appropriate directives that only generate useful packets. For instance, the CGA takes advantage from the multi-cell model used to represent the input ranges (cf. Section 3.2.1).

The coverage points shown in Table 4.5 concern the destination address (4 bits) and packet length (8 bits). Accordingly, we encode only those two fields of the packet into a genome of single chromosome of 12 bits range (it could be a genome of two chromosome, one chromosome to represent destination address while the

No.	Coverage Points	Coverage Rate (%)
1	Verify data on channels 1-5 only	100
2	Verify data on channels 4,9, and 15 only	100
3	Verify data on channels 4-7 and ensure that length of packet does not exceed 100 bytes	100
4	Verify data on channels 9 and 10 and ensure that the length of packets does not exceed 100 bytes and is not less than 30 bytes	100

Table 4.5: Coverage Results for the Router Design

other represents the packet length). Moreover, the most significant 4 bits represent destination address while the least significant 8 bits represent the packet length.

Table 4.6 shows the obtained directives (cells) for the coverage points described above. Let's consider the coverage point number 4, which has the directive $[219 : 2345 \rightarrow 2404]$ ($[DB : 929 \rightarrow 964]$ in hexadecimal representation). These values represent the constraints on the packet fields. They state that the destination address must be always 9 while the packet length can be in the range of $[41, 100]$ which satisfies the conditions of coverage point number 4.

Point No.	Coverage Directives $[W$: $L \rightarrow H]$
1	[169: 260 → 1400]
2	[69: 1104 → 1259] [94: 2314 → 2551] [77: 3849 → 4087]
3	[149: 1029 → 1124] [165: 1280 → 1375] [116: 1536 → 1635] [129: 1795 → 1879]
4	[219: 2345 → 2404] [231: 2600 → 2660]

Table 4.6: Output Directives for the Router Design

4.5 Master/Slave Architecture

In this section, we conducted more experiments to test the behavior and performance of the CGA when different parameters and strategies were used. Moreover, the purpose of these experiments is to show the effectiveness of our CGA algorithm in finding a solution for a group of many coverage points that achieve a minimum acceptable coverage rate. We use a Master/Slave architecture consisting of one master block, one middle block, and three slave blocks as shown in Figure 4.7.

For instance, we defined two data coverage groups, each group consists of 3 coverage points, as shown in Table 4.7. Each coverage point specifies the data range on a crossposting slave appearing in Figure 4.7. Also, Table 4.8 summarizes the used parameters and configurations during most experiments in the following sub-sections.

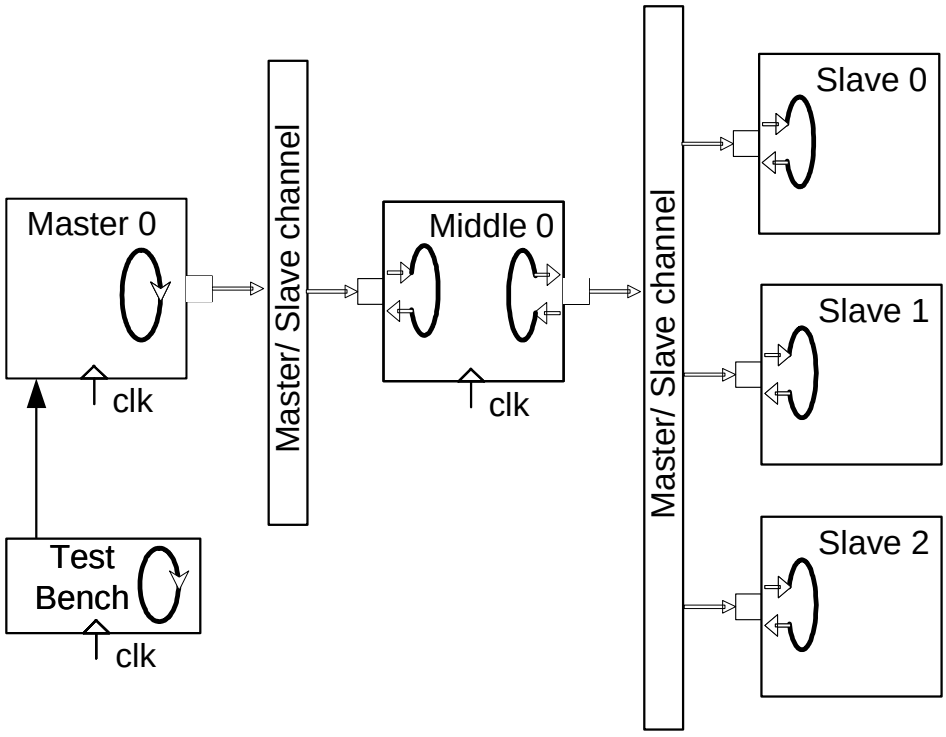


Figure 4.7: Master/Slave Block Diagram

Coverage point		Description
Group 1	cov1	$(x < 400)$ and $(x > 225)$
	cov2	$(x < 447)$ and $(x > 399)$
	cov3	$(x < 265)$ and $(x > 155)$
Group 2	cov4	$[(x < 40)$ and $(x > 25)]$ or $[(x < 60)$ and $(x > 50)]$
	cov5	$(x < 47)$ and $(x > 30)$
	cov6	$(x < 65)$ and $(x > 55)$

Table 4.7: Coverage Points for the Master/Slave Design

Parameters	Value
Population Size	50
Number of Generations	50
Crossover Probability	95%
Mutation Probability	20%
Elitism Probability	3%
Creation Type	Free Initialization
Number of Cell	25
Selection Type	Tournament Selection
Tournament Size	5
Enable Elitism	True

Table 4.8: Primary Parameters for the Master/Slave Design

4.5.1 Experiment 1 - Random Generator vs. CGA

In this experiment, we use multi-stage evaluation (*CovRate1* is equal 10% and *CovRate2* is equal 25%.) and free random initialization. We compare the result of CGA with the random simulation for the same number of cycles. For each group of coverage points, we run both the random simulation and the CGA into two modes: (1) group coverage, where we target all coverage points together as one group; and (2) individual coverage, where we target the coverage points one by one. This is to show the effect of grouping coverage points on the coverage rate and to highlight the ability of CGA of finding satisfactory solutions.

Table 4.9 presents the coverage rate in the case of random test generator. The

individual coverage of each point is slightly higher than the group coverage, while both of them are somehow low.

Cov. Group	**Coverage Rate**					
	Group Cov.			**Individual Cov.**		
	cp1	**cp2**	**cp3**	**cp1**	**cp2**	**cp3**
1	17	12	8	19	14	9
2	25	23	38	26	23	38
3	15	12	7	19	13	7

Table 4.9: Coverage Results of a Random Generator (Experiment 1)

Table 4.10 presents the results of the CGA. In all cases the CGA was able to achieve a coverage rate greater than the minimum coverage rate *CovRate2*. However, the CGA achieves 100% coverage rate for most coverage points when it targets the coverage points one by one.

Cov. Group	**Coverage Rate**					
	Group Cov.			**Individual Cov.**		
	cp1	**cp2**	**cp3**	**cp1**	**cp2**	**cp3**
1	100	66	33	100	100	99
2	50	50	73	100	100	100
3	91	66	28	100	100	95

Table 4.10: Coverage Results of the CGA (Experiment 1)

Figure 4.10 highlights the improvement of the mean group coverage and individual coverage for the three coverage groups using the CGA compared to the random test generators. The progress of evolution for the first coverage group is shown in Figure 4.11 which indicates the mean fitness values for the whole population over 50 generations. The figures show clearly the continuous improvement of fitness values and so the coverage rates over generations.

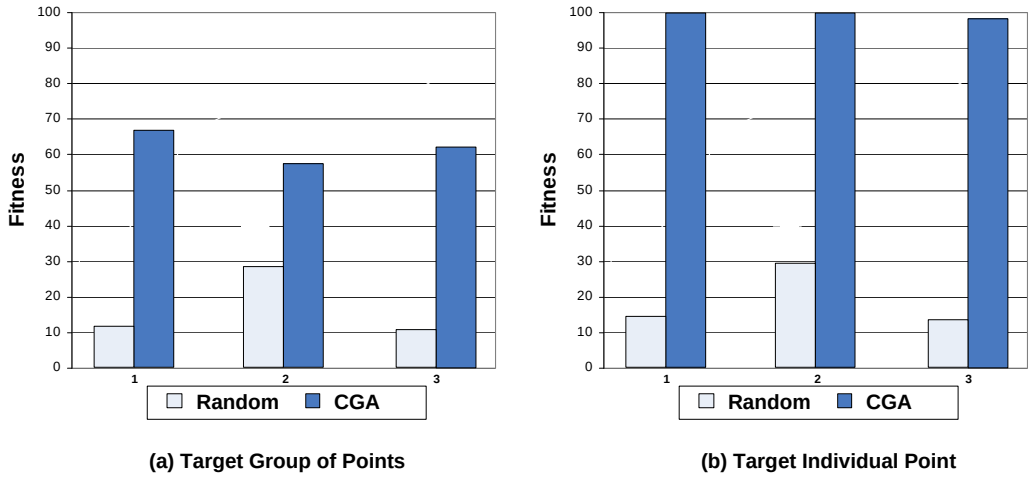


Figure 4.8: Comparison of Mean Coverage Rate (Experiment 1)

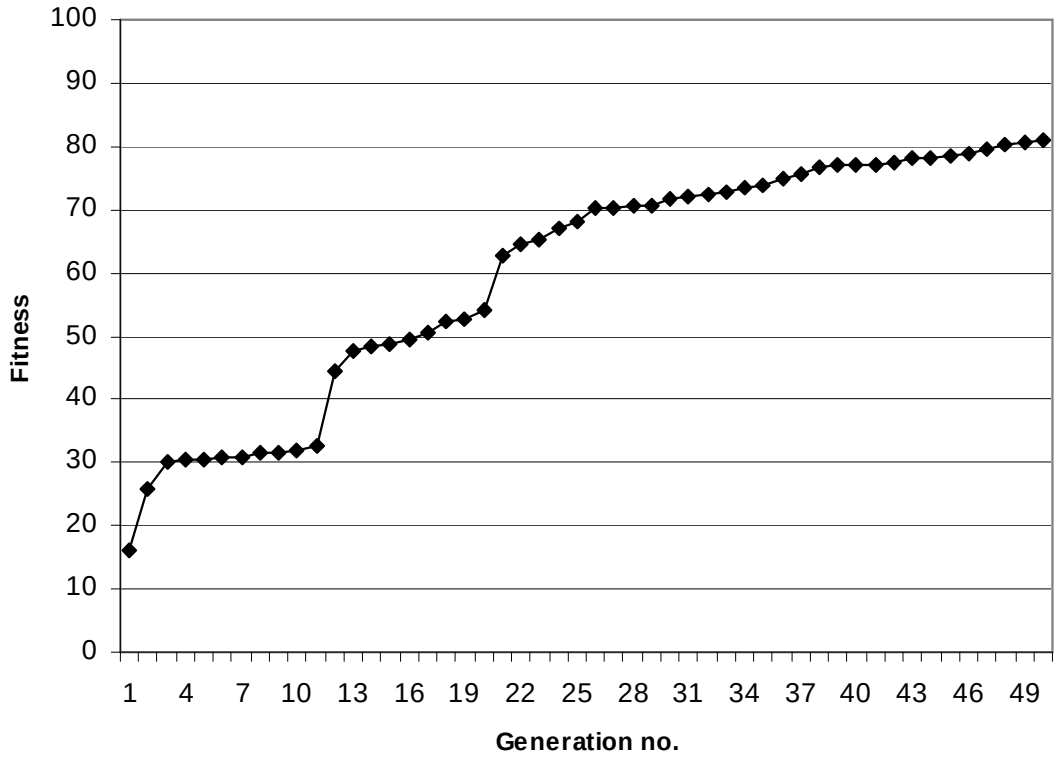


Figure 4.9: Fitness Mean of Coverage Group 1 (Experiment 1)

4.5.2 Experiment 2 - Effect of Population Size

During this experiment, we use the same configuration as Experiment 1, but for another coverage group. However, we change the size of the population parameter to see its effect on the speed of learning. Figure 4.10 illustrates the difference in the evolution process with different population sizes. It shows that the solution for a larger population gives a better result at the final stage. Besides, it shows that when a population with greater size can reach to a near optimal solution at early stage of evolution. Table 4.11 illustrates the best coverage values, and other information about the CPU time. It shows that when the population size doubles, the CPU time is increasing by 2.5 times.

Pop Size	Cov0	Cov1	Cov2	Avrg	StdV	Fitness	CPU Time
50	77	37	32	48.6667	24.6644	6662.15	16.359
100	89	67	26	60.6667	31.9739	7591.21	39.671

Table 4.11: Effect of Population Size on Evolution (Experiment 2)

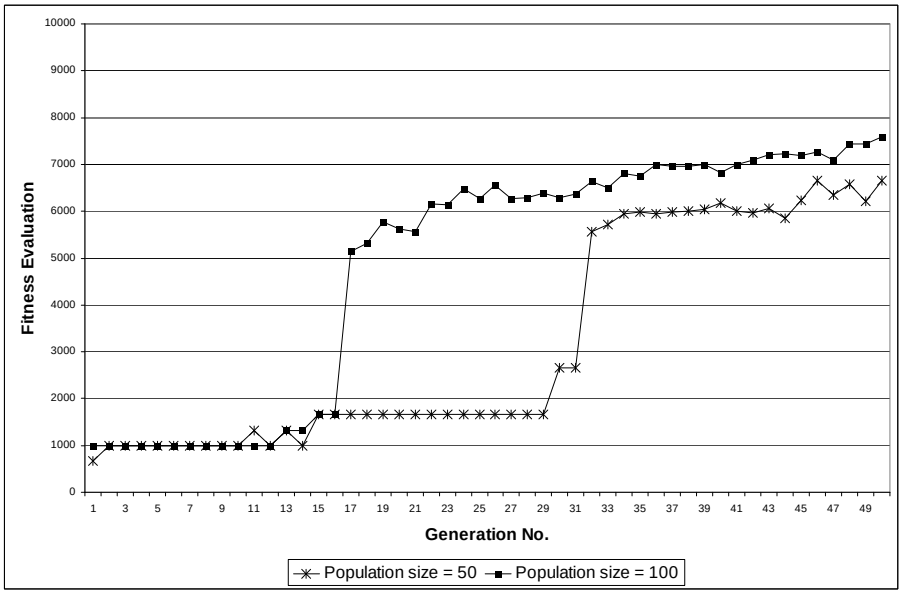


Figure 4.10: Effect of Population Size on Evolution (Experiment 2)

4.5.3 Experiment 3 - Effect of Standard Deviation Weight

In this experiment we use a fitness evaluation based on the difference between the mean and the weighted standard deviation of coverage points rates. Figure 4.11 shows that for small weights the process of evolution is much faster and much better than for high weights. However, if the weight is very small, then it will not take the standard deviation effectively into account and it may generate directives that ignore some of the coverage points as shown in Table 4.12.

Weight	Cov0	Cov1	Cov2	Avrg	StdV	Fitness	CPU Time
0.3333	97	97	0	64.6667	56.003	45.999	47.031
0.5	67	49	49	55	10.3923	49.8038	45.531
1	49	49	46	48	1.73205	46.2679	44.156
2	46	46	47	46.3333	0.57735	45.1786	43.484

Table 4.12: Effect of Standard Deviation Weight (Experiment 3)

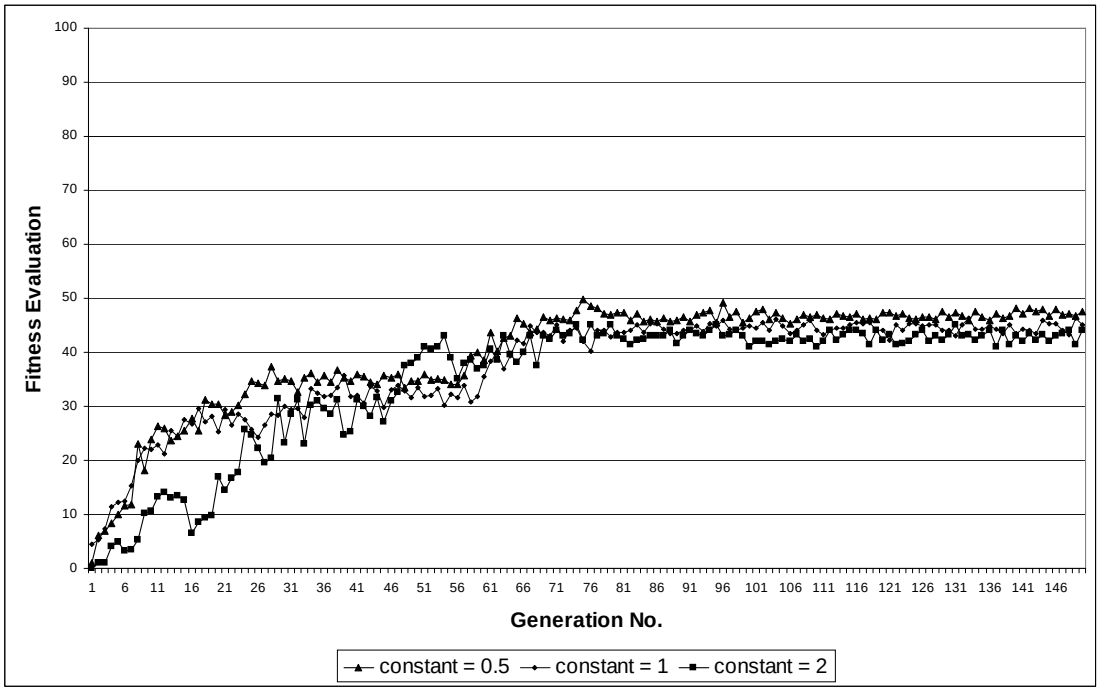


Figure 4.11: Effect of Standard Deviation Weight on Evolution (Experiment 3)

Figure 4.12 shows the evolution of another group of coverage points (group 2) with the same configuration. The results in Table 4.12 and 4.13 show that the standard deviation weight of 0.5 offers the best results in terms of the maximum achievable coverage rates within reasonable time.

Weight	Cov0	Cov1	Cov2	Avrg	StdV	Fitness	CPU Time
0.5	50	50	50	50	0	50	45.156
1	51	49	50	50	1	49	45.437
2	49	49	49	49	0	49	46.922

Table 4.13: Effect of Standard Deviation Weight (Experiment 3)

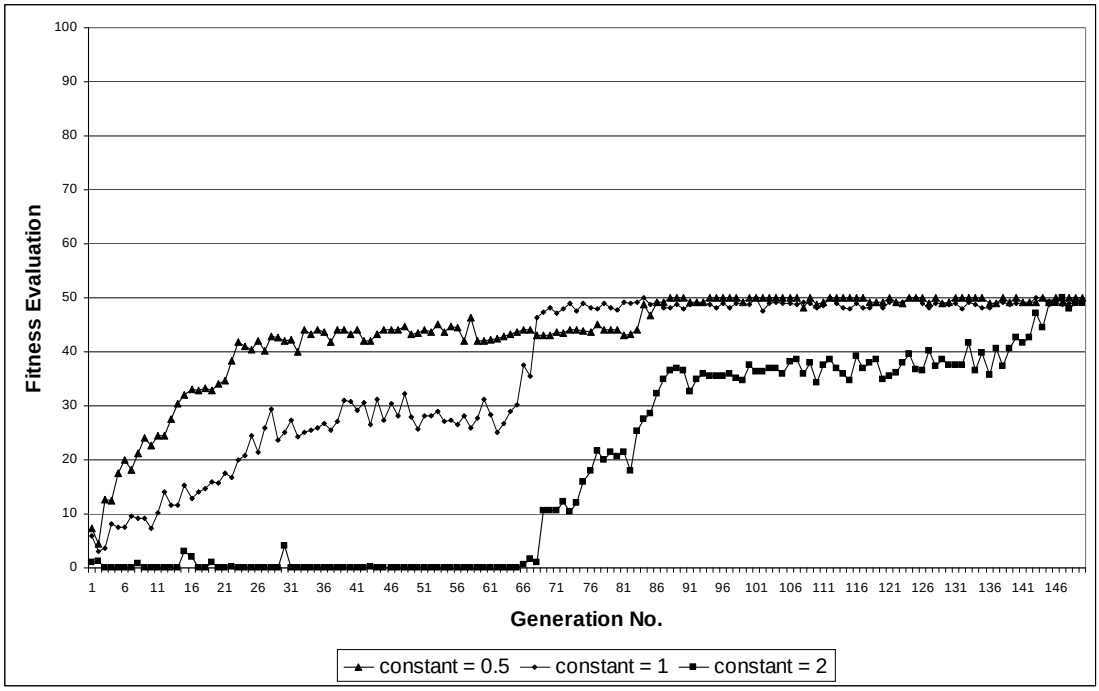
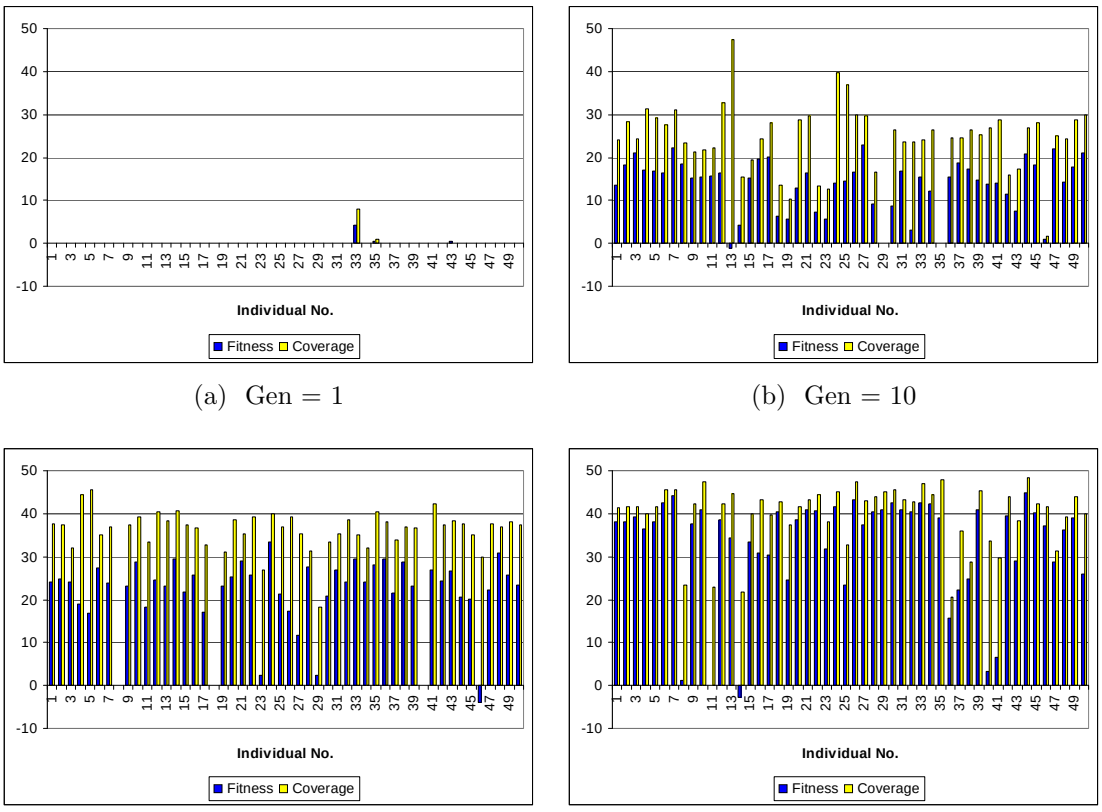


Figure 4.12: Effect of Standard Deviation Weight on Evolution (Experiment 3)

4.5.4 Experiment 4 - Coverage vs. Fitness Evaluation

The relation between average coverage rate and fitness evaluation is sown in Figure 4.13, where the mean-weighted standard deviation method is used. The relation between average coverage rate and fitness evaluation is shown in Figure 4.14 where the multi-stage method is used during evolution.



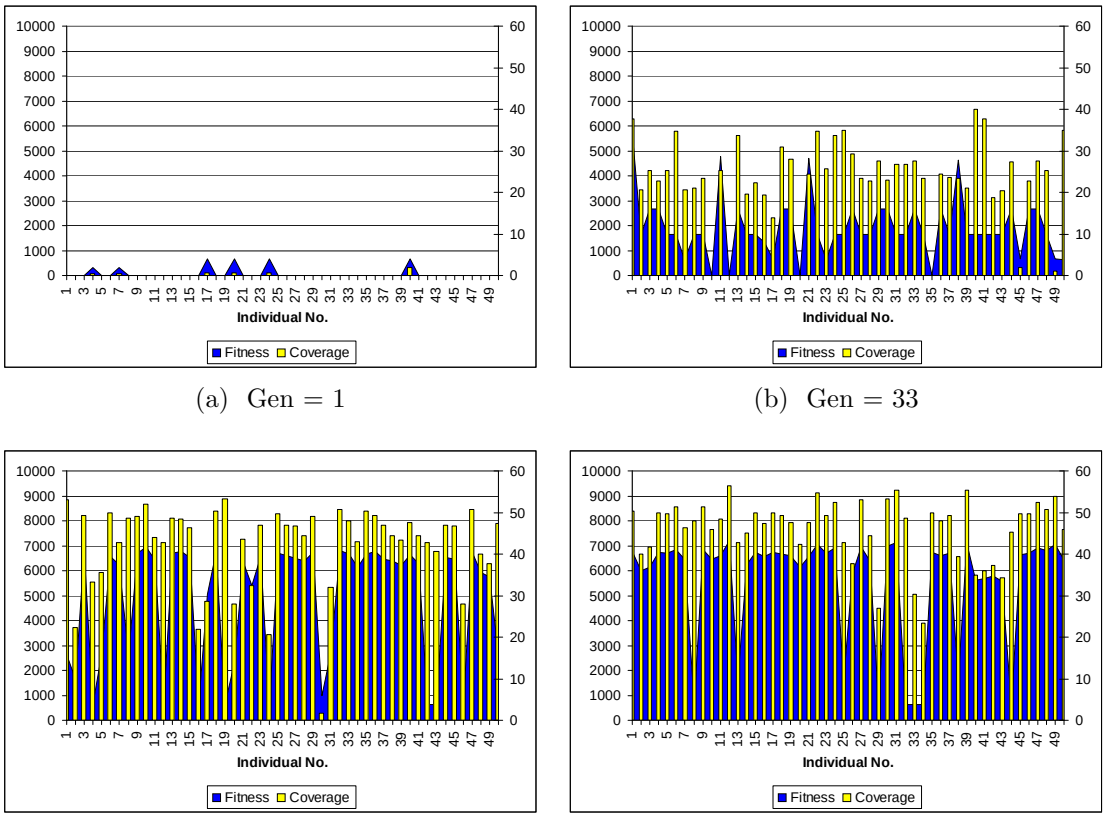
(a) Gen = 1 (b) Gen = 10

(c) Gen = 50 (d) Gen = 100

Figure 4.13: Mean-Standard Deviation Method: Average Coverage Rate vs. Fitness Evaluation (Experiment 4)

We monitor the value of coverage rate and fitness evaluation over many generations to see the improvement on the whole population quality over generations. Both Figures 4.13 and 4.14 show that the evolution starts with minor individuals

that has small fitness values. After 25% of the population has passed, we notice that the population quality becomes better than before. At this stage, we can see the differences between the misleading coverage rate for discriminating individuals and the value of fitness evaluation. Some individuals have very high coverage rates but low fitness evaluations. At the end of the evolution run, most of the population individuals have high fitness values and coverage rates.



(a) Gen = 1

(b) Gen = 33

(c) Gen = 66

(d) Gen = 100

Figure 4.14: Multi-stage Method: Average Coverage Rate vs. Fitness Evaluation (Experiment 4)

4.5.5 Experiment 5 - Shifted Domain

This experiment shows the results for coverage evaluation when a shift of the data domain has taken place in the middle stage of the master/slave architecture (Figure 4.7). The shift operation transforms any data go through the middle stage according to a shift equation. For example, given the following shift equation $3 * x + 5$, then the value of any data x entering the middle stage of the master/slave architecture will be changed according to that shift equation. The shift operation makes the problem of finding an optimal solution more difficult than before. We consider the shift operation as a simulation of complex design, where the mapping between input spaces and coverage spaces is not direct.

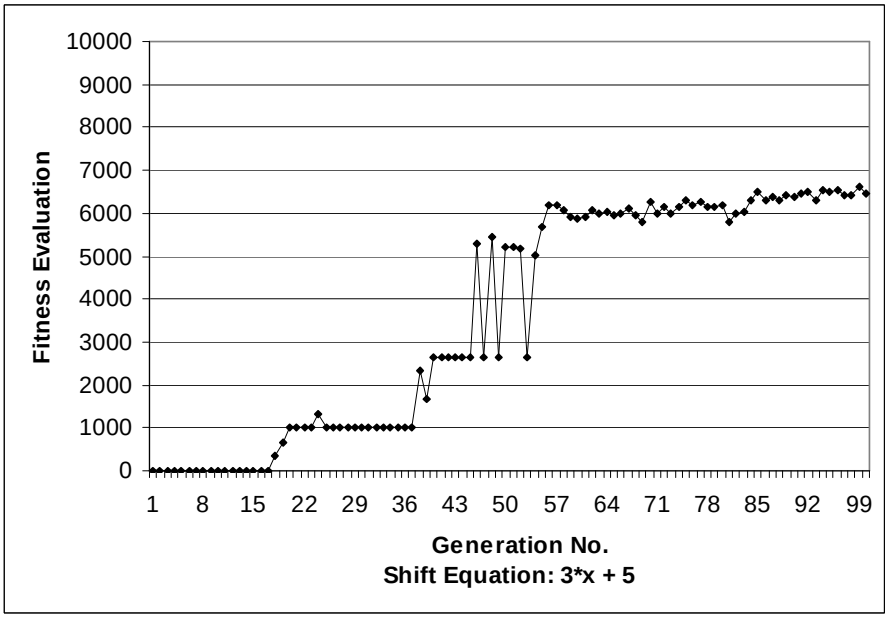


Figure 4.15: Evolution Progress for Shifted Domain (Experiment 5)

Figure 4.15 shows the evolution of best fitness value over many generations, and highlights the ability of the CGA to find an optimal solution even when there is no valid initial solution (zero initial coverage rate). Also, it indicates that the

average coverage rate was zero for 15 generations, which means that the problem of finding an optimal solution when shifting taken place is more difficult than other experiments without shifting operation.

Figure 4.16 shows the evolution progress for another shift operation. We can notice the long time toke the CGA to find an optimal solution. Besides, we notice the fluctuation of the fitness evaluation before reaching an optimal solution. This is also an indication on the difficulty of the shifting problem and the ability of the CGA to perform well even with complex and corner situations. Finally, the coverage rates results for three different shift operations are shown in Table 4.14

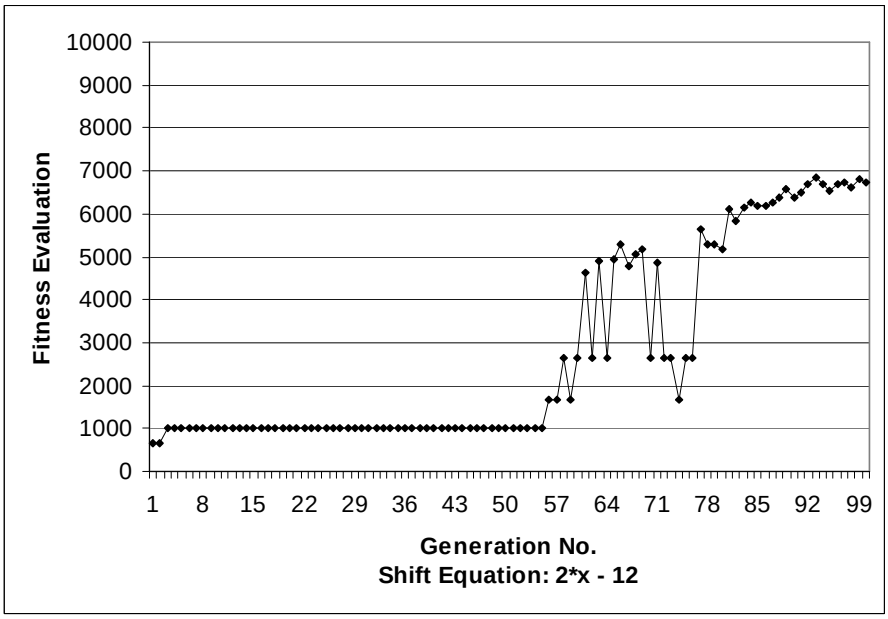


Figure 4.16: Evolution Progress for Shifted Domain (Experiment 5)

Coverage Rate	**Cov0**	**Cov1**	**Cov2**	**Avrg**	**Fitness**	**CPU Time**
3*x + 5	76	40	28	48	6625.77	36.765
2*x - 12	87	31	26	48	6843.82	37.365
x*x - 4	29	57	24	36.6667	5742.85	36.365

Table 4.14: Multi-Stage Evaluation of Shifted Points (Experiment 5)

4.5.6 Experiment 6 - Large Coverage Group

This last experiment concerns the progress of evolution in case of six-points coverage group (group1 and group2). When the mean-weighted standard deviation fitness evaluation is used, we only choose the weight of standard deviation in the minimization equation (cf. Section 3.2.6). A value of 0.5 for that weight gives an adequate solution as shown in Table 4.15.

Weight	**Cov0**	**Cov1**	**Cov2**	**Cov3**	**Cov4**	**Cov5**	**Avrg**	**Fitness**
1/3	50	0	50	50	50	0	33.33	24.73
1/2	25	24	25	36	21	24	25.83	23.24
1	23	11	22	14	22	26	19.67	13.85

Table 4.15: Effect of Standard Deviation Weight (Experiment 6)

On the other hand, if the multi-stage evaluation is used, we have to specify the threshold coverage rate and the weight of first stage of evaluation. Improper setting for those parameters may add constraints to the solution and in turn prevent the CGA from reaching to an optimal solution. Table 4.16 shows that the CGA was not able to reach an optimal solution when high coverage threshold rates ($CovRate1 = 15$ and $CovRate2 = 25$) are used as they over-constrain the solution.

Cov. Rate	**Cov0**	**Cov1**	**Cov2**	**Cov3**	**Cov4**	**Cov5**	**Avrg**	**Fitness**
15/25	26	22	14	19	12	3	16	1658.11
10/20	31	11	23	28	15	12	20	2491.47
10/10	28	17	32	42	33	15	27.8	4216.40

Table 4.16: Effect of Threshold Value on Evolution (Experiment 6)

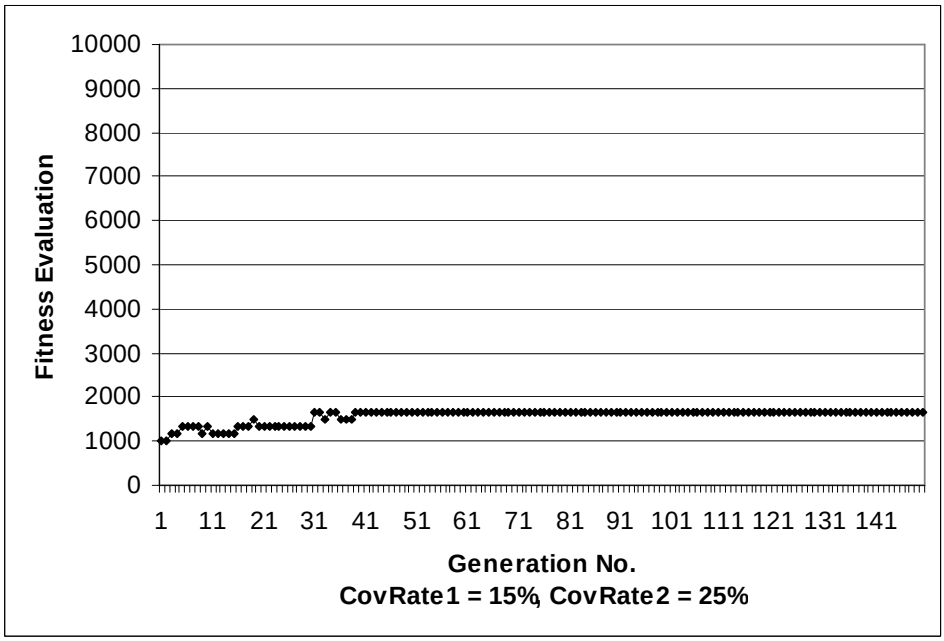


Figure 4.17: Effect of Threshold Value on Evolution (Experiment 6)

Figures 4.17, 4.18, and 4.19 show the evolution progress over 150 generations for different threshold coverage rates when multi-stage evaluation is used. When high coverage rates ($CovRate1 = 15$ and $CovRate2 = 25$) are used, there is no noticeable improvement in the performance of the CGA over generations (Figure 4.17) since these high threshold coverage rates add constraints on the potential solution. Figure 4.20 shows the progress of evolution when low threshold coverage rates (compare to Figure 4.17) are used. While Figure 4.18 shows a better progress, the coverage rates are still low. Finally, Figure 4.19 indicates a considerable improvement of the evolution progress as only one threshold stage is enabled, i.e., less constraints on the CGA to find an optimal solution.

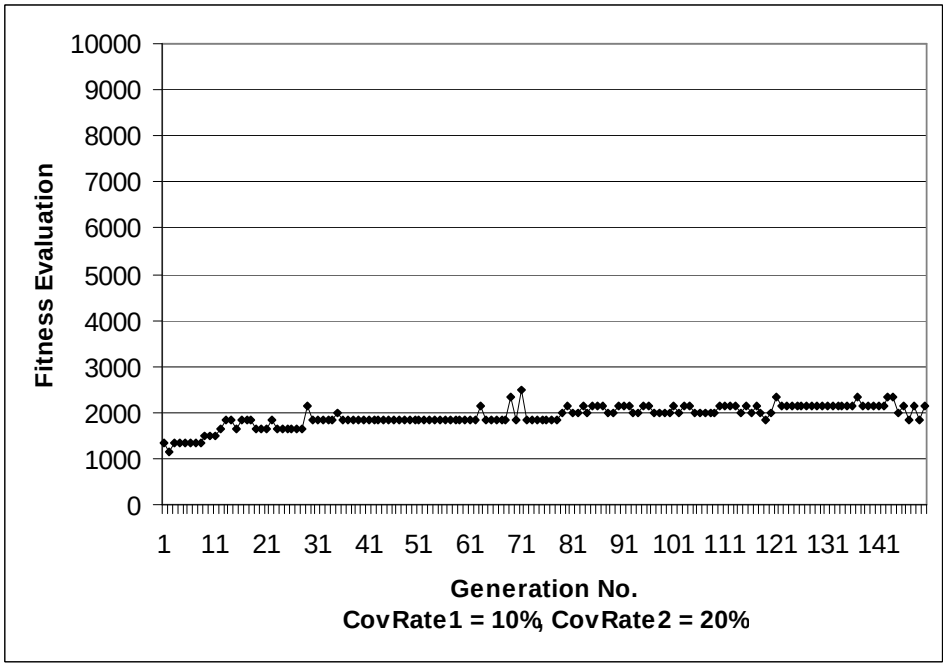


Figure 4.18: Effect of Threshold Value on Evolution (Experiment 6)

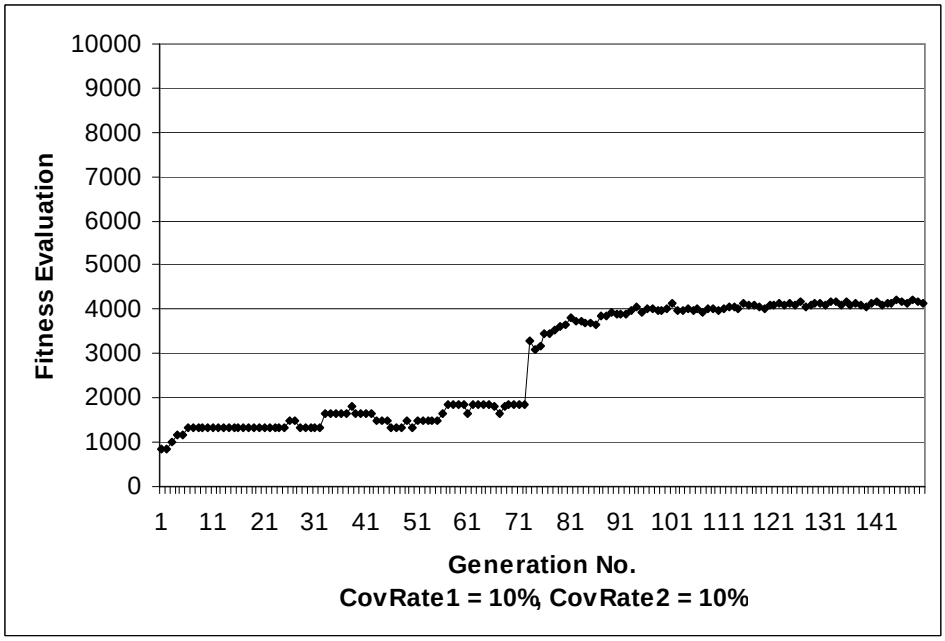


Figure 4.19: Effect of Threshold Value on Evolution (Experiment 6)

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we presented an approach for automated coverage directed random test generation (CDG) within the context of functional verification. To this end, we implemented an algorithm, based on genetic algorithms to form a feedback path between the coverage space and test directives. The introduced Cell-based Genetic Algorithm (CGA) automates the process of analyzing coverage information, associated with functional points, and modifies the random test generators directives. Moreover, we proposed an approach to extract a possibly complex directives for specific coverage task, and to find common directives that activate all coverage points enclosed within the same coverage group, and hence maximize the overall coverage rate.

We proposed to model the potential solution for CDG problem as sequences of cells that encode complex random distributions. Each cell represents a single directive over the DUV inputs, i.e., a cell can be thought of as a weighted uniform random distribution over a range of values. Several genetic operators were developed

to support the evolution process of the proposed representation and to reproduce effective potential solutions. Furthermore, we designed evaluation functions to testify the effectiveness of potential solutions and to discriminate them.

For experimental verification of the proposed methodology, we modeled three designs: a small CPU, a router, and a master/slave architecture written in SystemC language, which is built on the top of C++ language and has the advantage of fast simulation. We wrote different data and control coverage points to be targeted by a directed random test generator during the phase of evaluation.

The experimental results proved the effectiveness of our CGA in finding proper directives for the random test generator. For the small CPU design, the CGA was able to find useful instructions groups in order to utilize the zero and carry flag as well as activate corner jump paths. Similarly, proper specifications of generated packets to verify the router design were obtained and achieved high activation rates. For the master/slave architecture, we specified data coverage points to verify specific sub-ranges of the system outputs. Results showed a fast convergence to an optimal solution that activates many data coverage points using common directives. Furthermore, we run a number of experiments to illustrate the effect of different genetic parameters and sizes of the search space on the convergence to an optimal solution within a time limit.

5.2 Discussion and Future Work

The success of the CGA is highly dependent on the selection and tuning of many control parameters. The process of tuning those parameters is not trivial and needs a firm understanding of the CGA components. It also depends on the DUV complexity as well as the size of the search space. Accordingly, a self adaptation of the CGA

evolution framework is part of our future work. This will make the CGA totally evolvable, more flexible, and easier to use as the human intervention will become minor.

Cells are the main building blocks for a CDG solution. Each cell is identified by three parameters, two of them represent its range while the third one indicates its weight. A more natural representation of the cell with less parameters to optimize may achieve better coverage results within short simulation and learning times. We think that a cell can be represented by some known random distributions like normal distribution, gamma distribution, beta distribution, etc. For example, a cell with normal distribution needs only two parameters: the mean and the standard deviation, to represent its shape and to encode a solution. Genetic algorithms in these cases try to find the best parameters for those distributions in order to achieve high coverage rates. This will emerge the need to redesign our CGA including new representations and genetic operators.

Genetic algorithms have a limited ability to represent complex solutions. Particularly, the CGA utilizes a multi chromosomes (genome) representation of the solution. This representation does not induce parameters to describe different temporal relationships between the DUV inputs. An enhancement to the current representation as well as introducing some genetic operators to deal with temporal relationship may solve the problem. This is, however, is not easy to achieve. Other algorithms like genetic programming [39] and neural networks [29] may be used as a base to develop powerful CDG algorithms. These algorithms employ more complex representations and deeper knowledge encoding than genetic algorithms.

Genetic programming is able to encode more complex data and control directives as well as temporal relationship over a DUV inputs. Moreover, genetic

programming searches for optimal program structure represented internally as a tree where vertices represent program operators and leaves contain the operands. Before utilizing genetic programming based solution, developers specify operator and operands that are enough to form a desirable program. The main disadvantage of genetic programming is the huge search space compared to genetic algorithms.

Neural networks have been used in classification and recognition problems. Consequently, they can be used to classify subsets of the DUV inputs that are suitable to activate the coverage point under consideration. The neural network architecture may contain an input layer that maintains the DUV input, an output layer to represent the logical status for each coverage point, and some hidden layers according to the complexity of the CDG problem. Here, a random generator may be used as primary test generator where the tuning of neurons weights continuously changes over time according to the coverage reports. Backward tracing can be used to extract the useful directives for each coverage point, where the neural network works as a useful test generator.

Bibliography

[1] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *Computer Aided Verification*, LNCS 1855, pages 538–542. Springer-Verlag, 2000.

[2] Accellera Organization. Accellera Property Specification Language Reference Manual, version 1.1. www.accellera.org, 2005.

[3] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design and Test of Computers*, 21(2):84–93, 2004.

[4] B. Aktan, G. Greenwood, and M. Shor. Improving Evolutionary Algorithm Performance on Maximizing Functional Test Coverage of ASICs Using Adaptation of the Fitness Criteria. In *Proc. of Congress on Evolutionary Computation*, pages 1825–1829. IEEE Press, 2002.

[5] S. Asaf, E. Marcus, and A. Ziv. Defining Coverage Views to Improve Functional Coverage Analysis. In *Proc. of Design Automation Conference*, pages 41–44, San Diego, California, USA, 2004.

[6] Semiconductor Industry Association. Chip Sales Will Surpass 300 USD Billion in 2008. http://www.sia-online.org/pre_release.cfm?ID=386, 2005.

[7] B. Bentley. High Level Validation of Next-Generation Microprocessors. In *Proc. of the IEEE International High-Level Design Validation and Test Workshop*, pages 31–35, Cannes, France, 2002.

[8] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models.* Kluwer Academic Publishers, 2003.

[9] D. Black and J. Donovan. *SystemC: From the Ground Up.* Springer, May 2004.

[10] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir. A Genetic Approach to Automatic Bias Generation for Biased Random Instruction Generation. In *Proc. of Congress on Evolutionary Computation*, pages 442-448, Munich, Germany, May 2001.

[11] S. Chattopadhyay and K. S. Reddy. Genetic Algorithm Based Test Scheduling and Test Access Mechanism Design for System-on-Chips. In *Proc. of the International Conference on VLSI Design*, pages 341–346, New Delhi, India, 2003.

[12] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. Fully Automatic Test Program Generation for Microprocessor Cores. In *Proc. of Design Automation and Test in Europe*, pages 11006–11011, Munich, Germany, 2003. IEEE Computer Society.

[13] F. Corno, E. Sanchez, M. S. R., and G. Squillero. Automatic Test Program Generation: a Case Study. *IEEE Design and Test of Computers*, 21(2):102–109, 2004.

[14] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[15] R. Emek, I. Jaeger, Y. Katz, and Y. Naveh. Quality Improvement Methods for System-Level Stimuli Generation. In *Proc. of IEEE International Conference on Computer Design*, pages 204–206, Washington, DC, USA, 2004.

[16] P. Faye, E. Cerny, and P. Pownall. Improved Design Verification by Random Simulation Guided by Genetic Algorithms. In *Proc. of ICDA/APChDL, IFIP World Computer Congress*, pages 456–466, P. R. China, 2000.

[17] S. Fine and A. Ziv. Coverage Directed Test Generation for Functional Verification using Bayesian Networks. In *Proc. of Design Automation Conference*, pages 286–291, New York, NY, USA, 2003. ACM Press.

[18] H. Foster, D. Lacey, and A. Krolnik. *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

[19] L. Fournier, Y. Arbetman, and M. Levinger. Functional Verification Methodology for Microprocessors using the Genesys Test-Program Generator. In *Proc. of Design Automation and Test in Europe*, pages 434–441, Munich, Germany, 1999. IEEE Computer Society.

[20] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces using Genetic Algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 266–280. Springer-Verlag, 2002.

[21] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transaction Computers*, 30(3):215–222, 1981.

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

[23] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[24] A. Habibi, A. Gawanmeh, and S. Tahar. Assertion Based Verification of PSL for SystemC Designs. In *Proc. of International Symposium on System-on-Chip*, pages 177 – 180, Tampere, Finland, November 2004.

[25] A. Habibi, H. Moinudeen, A. Samarah, and S. Tahar. Towards a Faster Simulation of SystemC Designs. In *Proc. of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, pages 418–419, Karlsruhe, Germany, 2006.

[26] A. Habibi and S. Tahar. Towards an Efficient Assertion Based Verification of SystemC Designs. In *Proc. of the IEEE International High-Level Design Validation and Test Workshop*, pages 19–22, Sonoma Valley, California, USA, 2004.

[27] A. Habibi and S. Tahar. Design and Verification of SystemC Transaction-Level Models. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(1):57–68, 2006.

[28] A. Habibi, S. Tahar, A. Samarah, D. Li, and O. A. Mohamed. Efficient Assertion Based Verification Using TLM. In *Proc. of Design Automation and Test in Europe*, pages 106–111, Munich, Germany, 2006.

[29] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, 1995.

[30] IEEE. IEEE ratifies SystemC 2.1 Standard for System-Level Chip Design. http://standards.ieee.org/announcements/pr_p1666.html, 2005.

[31] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal*, 11:299–306, 1996.

[32] M. Katrowitz and L. M. Noack. I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DEC Chip 21164 Alpha Microprocessor. In *Proc. of the Design Automation Conference*, pages 325–330, New York, NY, USA, 1996.

[33] D. E. Knuth. *The Art of Computer Programing*, volume 2: Seminumerical Algorithms. Addison-Wesely, 1998.

[34] T. Kowaliw, N. Kharma, C. Jensen, H. Moghnieh, and J. Yao. CellNet Co-Ev: Evolving Better Pattern Recognizers Using Competitive Co-evolution. In *Genetic and Evolutionary Computation*, LNCS 3103, pages 1090–1101. Springer-Verlag, 2004.

[35] T. Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

[36] O. Lachish, E. Marcus, S. Ur, and A. Ziv. Hole Analysis for Functional Coverage Data. In *Proc. of Design Automation Conference*, pages 807–812, Los Alamitos, California, USA, 2002.

[37] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulations*, 8(1):3–30, 1998.

[38] P. Mazumder and E. Rudnick. *Genetic Algorithms for VLSI Design, Layout and Test Automation*. Prentice Hall Professional Technical Reference, 1999.

[39] Z. Michalewics. *Genetic Algorithm + Data Structures = Evolution Programs*. Springer, 1992.

[40] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):56–59, 1965.

[41] J. Nagda. High Level Functional Verification Closure. In *Proc. of IEEE International Conference on Computer Design*, pages 91–96, Washington, DC, USA, 2002.

[42] Open SystemC Initiative OSCI. IEEE 1666 SystemC Standard, 2006.

[43] Open Vera Assertion. www.open-vera.com, 2006.

[44] S. Palnitkar. *Design Verification with e.* Prentice Hall Professional Technical Reference, 2003.

[45] I. Pomeranz and S. M. Reddy. On Improving Genetic Optimization Based Test Generation. In *Proc. of the European Design and Test Conference*, pages 506–509, Los Alamitos, California, USA, 1997.

[46] N. J. Radcliffe. The Algebra of Genetic Algorithms. *Annals of Maths and Artificial Intelligence*, 10(4):339–384, 1994.

[47] E. M. Rudnick and J. H. Patel. Combining Deterministic and Genetic Approaches for Sequential Circuit Test Generation. In *Proc. of Design Automation Conference*, pages 183–188, New York, NY, USA, 1995.

[48] Specman Elite. www.verisity.com, 2006.

[49] G. S. Spirakis. Designing for 65nm and Beyond, Keynote Speech. In *Design Automation and Test in Europe*, Paris, France, 2004.

[50] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer. A Functional Validation Technique: Biased-Random Simulation Guided by

Observability-Based Coverage. In *Proc. International Conference on Computer Design*, pages 82–88, Los Alamitos, California, USA, 2001. IEEE Computer Society.

[51] S. Tasiran and K. Keutzer. Coverage Metrics for Functional Validation of Hardware Designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.

[52] S. Ur and Y. Yadin. Micro Architecture Coverage Directed Generation of Test Programs. In *Proc. of Design Automation Conference*, pages 175–180, Los Alamitos, California, USA, 1999.

[53] P. Wilcox. *A Guide to Advanced Functional Verification*. Springer, 2004.

[54] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, June 2005.

[55] A. Ziv. Cross-Product Functional Coverage Measurement with Temporal Properties-Based Assertions. In *Proc. of Design Automation and Test in Europ*, volume 01, pages 834–841, Munich, Germany, 2003.