Enhancing Coverage Based Verification using Probability Distribution

Essam Arshed Ahmed

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 (Electrical & Computer Engineering)

at

Concordia University

Montréal, Québec, Canada

September 2008

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Essam Arshed Ahmed**

Entitled: **Enhancing Coverage Based Verification using Probability Distribution**

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

Master of Applied Science (Electrical & Computer Engineering)

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. John Xiupu Zhang

_____ Dr. Jamal Bentahar

_____ Dr. Abdelwahab Hamou-Lhadj

_____ Dr. Sofiène Tahar

Approved by _____

Chair of the ECE Department

_____ 2007 _____

Dean of Engineering

ABSTRACT

Enhancing Coverage Based Verification using Probability Distribution

Essam Arshed Ahmed

Functional Verification is considered to be a major bottleneck in the hardware design cycle. One of the challenges faced is to automate the verification cycle itself. Several attempts have been made to automate the verification cycle using Artificial Intelligence (AI) approaches. On the other hand, coverage based verification is an essential part of functional verification where the objective is to generate test vectors that maximize the functional coverage of a design. It uses a random test generator that can be directed by some AI algorithms. This process of adapting AI to direct the test generator according to coverage is called Coverage Directed Test Generation (CDG). CDG is a manual and exhausting process, but it is vital to complete the verification cycle. To increase the coverage, a Cell-based Genetic Algorithm (CGA) is developed to automate CDG. We propose a new approach of using CGA with random number generators based on different probability distribution functions such as Normal (Gaussian) distribution, Exponential distribution, Gamma distribution, Beta distribution and Triangle distribution. We apply the new approach on a 16×16 packet switch modeled in SystemC, where we define appropriately several static and temporal coverage points and study the effect of the probability distribution on the coverage rate using CGA as an optimization tool. Furthermore, we model the same 16×16 packet switch using Verilog and express the same coverage points using SystemVerilog and run the simulation using Verilog simulator and random number generator based on Normal distribution, Exponential distribution and Uniform distribution to show their effect on coverage and compare the results with our approach. Then experiments show that some probability distributions have more effect on the coverage than other distributions.

ACKNOWLEDGEMENTS

First of all, I would like to praise and thank almighty God who gave me the power and strength to complete this work. Secondly, I would like to give my mother a special thanks for her continued encouragement and countless prayers to succeed in my mission. I would like truthfully to thank my supervisor Dr. Sofiène Tahar for his support and guidance to overcome the obstacles I faced during my thesis work. I would like also thank the members of my thesis committee, Dr. Jamal Bentahar and Dr. Abdelwahab Hamou-Lhadj for being my examiners and their feedback on the thesis. Also, I would like to thank Dr. Ali Habibi, from MIPS Inc., who introduced me to the topic of this thesis and encouraged me to carry on with the topic. His help and feedback were very valuable to complete my thesis. Likewise, I would like to thank Dr. Otman Ait Mohamed for his valuable information and discussions. In addition, I would like to thank Sayed Hafizur Rahman who introduced me to the Hardware verification Group (HVG) and Asif Iqbal Ahmed who encouraged me to join HVG and later introduced me to Dr. Ali Habibi. Also, I would like to give special thanks to Naeem Abbasi for his time and valuable discussions related to my thesis work and likewise I would like to thank inclusively all members of HVG for their support, help and encouragement. Finally, I would like to thank my sisters and brother as well as my friends for their encouragements.

To my mother and the memory of my father

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACDG	Automated Coverage Directed test Generation
AI	Artificial Intelligence
CDF	Cumulative Distribution Function
CDG	Coverage Directed test Generation
CGA	Cell-based Genetic Algorithm
CPU	Central Processing Unit
DUV	Design Under Verification
EDA	Electronic Design Automation
FIFO	First In First Out
FSM	Finite State Machine
GA	Genetic Algorithm
GNU	GNU's Not Unix
GPR	General Purpose Register
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
MIPS	Microprocessor without Interlocking Pipelined Stages
MT	Mersenne Twisted
OOP	Object Oriented Programming
OVA	Open vera Assertion
PDF	Probability Distribution Function
PSL	Property Specification Language
RNG	Random Number Generator
RTL	Register Transfer Level
SoC	System-On-Chip
SPSS	Statistical Package for the Social Sciences
SVA	System Verilog Assertions

TLM	Transaction Level Modeling
VCS	Verilog Compiled Simulator
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

1.1 Motivation

In recent years, the semiconductor industry has been growing fast and gaining more profits. It is reported by Semiconductor Industry Association (SIA) that the global chip sales hit $255.6 billion in 2007 with an increase of 3.2% from 2006, and it is expected to grow by 7.7 in 2008 with a jump of 7 percent in 2009 and a rise of 8.5 percent in 2010 [4][31]. The rapid growth in semiconductor devices is due to the rapid demand from the market for computers, mobile phones and other consumer electronics [4]. In addition, the demand is not only for quantity but also for complexity yet simplicity in using consumer electronics devices such as mobile phones, handhelds, laptop computers, and digital cameras. Adding more functions to an electronics device yet making it simple to use is a challenging task and requires a significant amount of time and money to put all the functions and consequently add more logic gates in a single chip. According to Moore's law, the number of transistors in a single chip was almost doubled every two years [7].

On the other hand, consumers would not appreciate electronic devices that fail some of their functions during their normal life and from the manufacturers point view, the cost of finding bugs through consumers is larger than the cost of verification

[44]. Thus, producing a chip that works correctly has become an essential task in the chip developing process.

This rapid growth creates a lot of challenges and pressure on researchers and engineers to design and fabricate workable chips functionally and physically within strict design specifications and a rigid time frame. Research conducted by Collett International shows that the number of chips that work from the first time is less than 50% of the total chips fabricated [39]. It was also found that more than 70% of the defects are due to functional errors rather than physical or other types of error. These functional errors are due to incorrect implementation of the design specification in hardware description language (HDL) code, or so-called *bugs*. Therefore, it has become a vital issue to produce functionally correct and verified designs before the fabrication process. This will minimize the fabrication cost and reduce time-to-market.

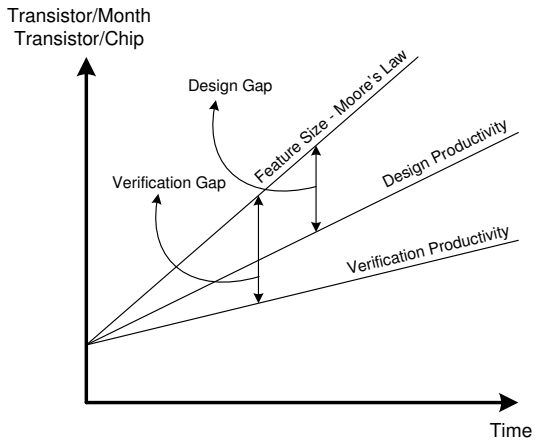

Figure 1.1: Design and Verification Gaps [42]

Verifying the functionality of the design or a chip is called Functional verification. Functional verification is a bottleneck in any chip or System-on-Chip (SoC) development process: it is reported that functional verification takes around 60%-70% of chip development efforts in terms of time, and computer and personal resources

[28]. The challenges of functional verification come from the design complexity (logic gates), design duration, and verification complexity. Figure 1.1 shows that the design productivity growth continues to remain lower than complexity growth, which in return increases the gap between the design and verification. Therefore, functional verification becomes an essential part of any chip development process.

1.2 Functional Verification

The main task of functional verification is to compare the specification of a design with its observed behavior to determine the equivalency between the specification and the actual design, and any differences are reported as bugs. The word "design" refers to Design Under Verification (DUV) [12]. The first step in this process is to identify the main functions that need to be verified and divide them into fragments and also to identify the areas of the design that are prone to bugs [41]. The next step is applying one of the functional verification methodologies to simulate the design and collect the simulation result. These methods can use either a directed test scheme or a random test scheme.

There are several methodologies for tackling functional verification problems and they are divided into simulation and formal methods. Formal methods use mathematical expressions and mathematical reasoning to prove the correctness of the design, but in simulation methods, the design is represented functionally and logically by the semantic of a language which can be simulated to observe the behavior of the design. Simulation methods can be further divided into several methodologies such as simulation-based verification, assertion-based verification, and coverage based verification.

Formal verification focuses on systematic ways to prove or disprove the correctness of the design using mathematical formal methods. Mathematical expressions and symbols are used to express the properties of the design, then use mathematical

reasoning to prove or disprove the correctness of the properties regardless to the input values [20]. There are three main approaches for formal verification: Model Checking, Equivalence Checking, and Theorem Proving. Model checking and equivalence checking are exhaustive techniques and cannot be used for large design due to state-space explosion problem which partially solved by introducing Symbolic Model Checking. On the other hand, Theorem Proving can be used to verify larger designs but it is not very practical due to considerable human effort and expertise needed [19].

Simulation-based verification is the most widely used in verification; almost all Electronic Design Automation (EDA) tools support simulation based verification where the testbench is built to provide valid scenarios to verify the logic behavior of the design. A testbench can provide random, directed and constrained random input over the entire input space of the design. The main advantage of this methodology is that it is easy to control the input signal, it is easy to build, and it is easy to target certain aspects of the design. On the other hands, its main drawback is that it is difficult to control the internal signals and to observe their behavior at the output.

In assertion verification, the designer asserts a certain code or statement in the code of hardware design to verify properties of the design. The asserted statement does not affect the design behavior, but extracts useful information about the properties of the design [11][44]. Assertion based verification is considered a white box verification, where asserted statements are written in HDL or special assertion language such as OpenVera Assertion (OVA), SystemVerlig Assertion (SVA) or Property Specification language (PSL) [11].

Assertion-based Verification checks mainly two types of assertions [11]: (1) Immediate assertion and (2) Concurrent assertion. The *Immediate assertion* is also called *static* or *event assertion* [29]: it detects static events using a simple assertion check to detect if the event occurs immediately or not. While *Concurrent assertion*, also called *temporal assertion* [23], detects a sequence of events over a time period, it

means that a sequence of several events should occur before the final asserted event is checked. There is another type of assertion, which is called *pre-defined assertion building blocks*, which uses pre-defined assertion libraries and is developed by EDA vendors and supported by assertion languages such OVL, PSL and SVA [44].

Coverage-based verification play an important role in functional verification because it is used to assess the progress in the verification cycle and identify the area of the design that has not been tested. The coverage-based verification requires to define coverage tasks (coverage points and coverage group) that are used to quantify coverage progress. Coverage tasks represent various functions and properties of the design. They are classified into two main types as in assertion-based verification: (1) Static coverage points and (2) Temporal coverage points [29]. A coverage group is a set of coverage points. In this thesis, we are going to use the words Static and Temporal to describe both the coverage points and assertions.

Coverage metrics is considered to be a measure of the coverage which answers the question, how much coverage do we get and are we done? [12]. In other words, it measures the completeness of the verification process and directs the verification process towards unexplored areas of the design. Sets of criteria and thresholds are established to compare with coverage metrics to determine whether all activities in the verification process are completed or not. Coverage metrics help in:

1. Quantifying the completeness of the verification by applying heuristic measures
2. Identifying the areas of the design that are not covered and guiding the tes-bench driver

There are several coverage metrics but the most widely used are code coverage, finite state machine coverage, structural coverage and functional coverage. Each metrics is used to assess part of the design and provide helpful information about the area that has not been tested. The code coverage is used to verify the HDL code of the design and it is divided into branch coverage, line coverage, expression

coverage and path coverage. The structural coverage focuses on logical structure of the design. It refers also to toggle coverage and combinational coverage.

The finite state machine (FSM) coverage is a useful metric that provides more information about the functionality of the design. FSM metrics focus on state coverage and transition coverage. The *state coverage* shows the percentage of the visited states and the *transition coverage* shows the percentage of the visited possible transition or paths between the states [43]. Since the FSM of the complete design may be very large and difficult to cover, it can be divided into two categories: (1) Handwritten FSM that captures the behavior of the system at higher level; (2) Automatic extraction of FSM from the design description. Functional coverage uses the concept of coverage events or coverage tasks that define a property or function of the DUV and it is specified in the coverage model to detect its occurrence.

1.3 Coverage Directed Test Generation

In any chip development process, the design's specification serves as input for defining the verification and coverage tasks. The functional coverage process, in particular, can be seen as two steps: (1) defining the cover points; and (2) finding the appropriate test to hit those points. This process is typical of coverage verification processes in simulation-based verification and it is the most challenging and exhausting task due to the manual effort of analyzing coverage information and modifying testbench to enhance the coverage [34]. The functional coverage process is also called Coverage Directed-test Generation (CDG).

Figure 1.2 shows the manual CDG where verification engineers guide the random number generator by setting up directives and constraints. The random number generator generates test vectors for the simulator. The simulator simulates the DUV and the coverage points. At the end of the simulation, a coverage report is generated. The coverage report contains information about the coverage points that are

covered by the generated test patterns during the simulation. The verification engineers analyze the coverage report and modify the constraints to cover the area of the design that are not covered.

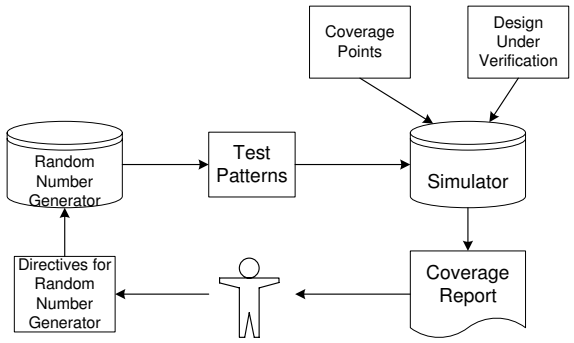

Figure 1.2: Manual Coverage Directed Test Generation

A constrained Random Number Generator (RNG) is the main part of the testbench in coverage-based verification (Figure 1.2). Manual modification of such constrained random number generators is not always feasible due to the poor controllability from the primary inputs over the internal variables. A non-automatic human-based technique, even if it succeeds in deciphering the previously mentioned inputs/internals relationship, will result in a very tedious and time-consuming overall coverage optimization process. Thus, Coverage Directed-test Generation (CDG) is a real problem in functional verification.

The problem lies in finding the best directives or constraints for the random number generator to generate test patterns that achieve maximum coverage in less time. Several attempts were made to solve this problem by automating the CDG and replacing the human effort with an Artificial Intelligence (AI) technique to analyze the coverage report and provide directives for the RNG based on pre-defined knowledge and learning experience. Known AI algorithms that have been explored are Neural Network, Bayesian Network, and Genetic Algorithms.

In general, coverage-based verification and CDG use uniform probability distribution to generate random numbers. Although AI algorithms guide the constrained RNGs to achieve maximum coverage, the time it takes to achieve such coverage depends on the design's inputs and its internal variable relationship. Therefore, it does not guarantee that AI algorithms can achieve maximum coverage in a short time, because the input that achieves maximum coverage may not be distributed uniformly across the input domain. Therefore, other probability distributions can be used to generate random numbers and enhance the coverage further by achieving the maximum coverage in a short time.

In this thesis, we suggested using different probability distribution functions to generate random numbers along with AI algorithms. The suggested distributions are Normal distribution, Exponential distribution, Gamma distribution, Beta distribution, and Triangle distribution

1.4 Related Work

In this section, we present related work in the area of functional coverage-based verification using different methodologies and algorithms. We will focus on the Artificial Intelligence algorithms such as a Bayesian Network, Neural Network and Genetic Algorithms, as well as design's functional properties and different methodologies used to automate the verification cycle. Finally, we will present a brief review of a methodology of cell-based genetic algorithm that is used to automate coverage directed test generation.

Many attempts have been made to automate the verification cycle and close the gap between analyzing the coverage result and modifying the testbench by using Artificial Intelligence algorithms such Neural Network, Bayesian Network and Genetic Algorithms.

In [40], a Markov chain is used to generate test vectors for the DUV. The

parameters of the Markov chain are modified based on coverage analysis. A pre-defined probability distribution, which depends on the current state of the DUV, is used to analyze the coverage and guide the Uniform distribution RNG. This work uses probabilistic and semi-formal analysis of sequential circuit specification to form a model for verification in contrast to the approach where real HDL model or higher abstraction model (SystemC). Along the same line of thought, a Bayesian Network is used in [10] to define the relationship between the directives of a random test generator and coverage space. A learning algorithm trains the data of the Bayesian Network with correct knowledge to direct the random test generator. However, the quality of those data affect the ability of Bayesian Network to encode the correct knowledge. This is in contrast with totally free random data, which is only affected by the seed of the random data generator.

In [35], a Neural Network is used for Priority Directed Test Generation. This is done using two main modules: the Priority Control module and a Neural Learning module. The Neural Learning module analyzes the result and feeds the learning experience to the Priority Control module. The priority control module controls the random test generation by specifying a set of rules and attributes that are set manually. This algorithm uses several pre-selected test vectors with different priorities instead of free random initialization as we do in this thesis. In fact, we follow a similar flow, but use GA as the learning and optimizing algorithm.

Inductive Logic Programming (ILP), which is a subfield of machine learning, is used in [15] as a learning method from examples in the context of CDG. An implementation of ILP, called Progol, is used as case study which is provided with pre-defined knowledge of functional tasks of the system and their relationship rules. The Progol generates test directives for a random stimulus generator that generates test vectors for DUV. The study targets only static coverage tasks.

In [14], an approach for automatic Coverage Directed test Generation (CDG) is proposed where the constraint for the random number generator can be extracted

through simulations over a number of clock cycles. The tool analyzes the simulation data (coverage information) and extracts input constraints automatically that are used to control the internal signals, but the designer needs to specify the internal constraints. The major advantage of this approach is that it optimizes a sequence of the inputs but requires a lot of effort to define the constraints manually and it does not target temporal properties.

Evolutionary algorithms such as genetic algorithms are also introduced in the area of functional verification as an optimization technique to generate input test vectors. For example, genetic algorithm is proposed in [9] where the input test vector is a series of n numbers that are optimized and generated by a uniform random number generator. This approach tackles temporal properties, for example, when a transaction is output to a bus and is acknowledged, then its number appears on the transaction indication bus after 3 clock cycles. In the same way, a genetic algorithm is proposed in [6] to automatically generate biased random instructions to verify microprocessor architecture at Register Transfer Level (RTL). Likewise, a GA is used in [46] to generate a sequence of inputs applied to digital integrated circuits. In this approach, each individual of the population represents a chain of inputs.

Another study, [45], used Genetic Algorithm in the context of CDG using multilayered environment. The verification platform consists of five different layers to make it reusable. The DUV is placed in signal layers (RTL layer) and each layer provides a set of services. The coverage tasks are defined as function of inputs and the DUV is viewed as FSM. The inputs are the generated input and the states are the set of the machine's state that are triggered by the inputs. This work did not tackle the temporal properties of the DUV. The output of the design under verification is checked automatically either by comparing with the behavioral model of DUV or by comparing with the expected result in the scoreboard. The solution is encoded in few chromosomes and only three chromosomes were randomly initialized at the beginning of the simulation. Finally, this work did not show clearly the evolution

process of the GA.

After reviewing the above studies, we found some limitations, such as targeting one property at a time [9], predetermined initial data in contrast with free initialization [10], and targeting static properties [15][14] except in [9]. In addition, the genetic algorithms that were used do not provide a clear information about how the coverage is measured and how the solution is presented by the GA. These limitations were overcome by a recent work [34] that used Cell-based Genetic Algorithm (CGA).

In CGA, the input domain is divided into sequences of inputs called *cells*. Each cell has upper and lower limits and it represents a single input to the DUV. The cells are randomly selected from the range of input domain. The range of the input domain is from (2^0) to $(2^{31} - 1)$. The number of cells and the range of the input domain are defined by the user. The basic operation of CGA starts when a certain number of cells is generated randomly, each cell targets the DUV and the coverage information is collected. Then, CGA evaluates the coverage information using pre-defined evaluation function, which is called *fitness function*. The fitness function selects the most fitted cells based on pre-defined criteria and forwards them to the next generation. The rest of the cells are forwarded for genetic operations to produces new modified cells for the next generation. Experimental results showed better coverage compared to Specman testbench [2] and pure random test generators. On the other hand, CGA uses Uniform random number generator, and targets small sets of static properties.

In this thesis, we are enhancing the work of [34] by adding random number generators based on different probability distribution and study the effect of the probability distribution on the coverage. Moreover, we are targeting larger sets of static and temporal properties.

1.5 Methodology

The general concept of performing coverage in hardware design, as depicted in Figure 1.3, involves both the design and the coverage points. The simulator, with coverage reporting enabled, provides a coverage report including functional coverage (cover groups, assertions, and cover points). The classical technique for improving coverage requires verification engineers to evaluate the report and decide to fine-tune the tests (this may also involve changing the testbench components) in order to maximize the coverage. This link is replaced in Figure 1.3 by the CGA box.

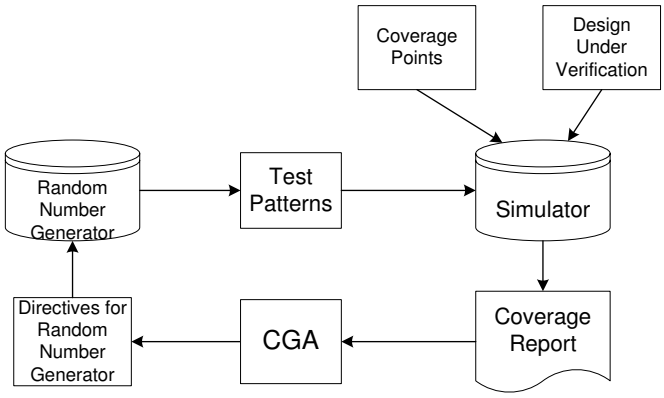

Figure 1.3: Automatic Coverage Directed Test Generation

The CGA analyzes the coverage information and optimizes the search for new directives for the random number generator. In other words, it fine-tunes the input and the system setting in order to hit a set of given coverage points. To perform this operation, we propose to use the random test generators with constrained test generators that support several probability distributions. We propose five probability distributions such as *Normal (Gaussian), Exponential, Gamma, Beta*, and *Triangle* probability distributions. We implemented, tested and integrated the five distributions into the CGA. The random number generator generates random number based on the selected distribution. For example, if Normal distribution is selected then the CGA generates cells based on the mean and the standard deviation of the Normal

distributions over the input domain. The algorithm task is to find the appropriate numbers in the domain to maximize the coverage.

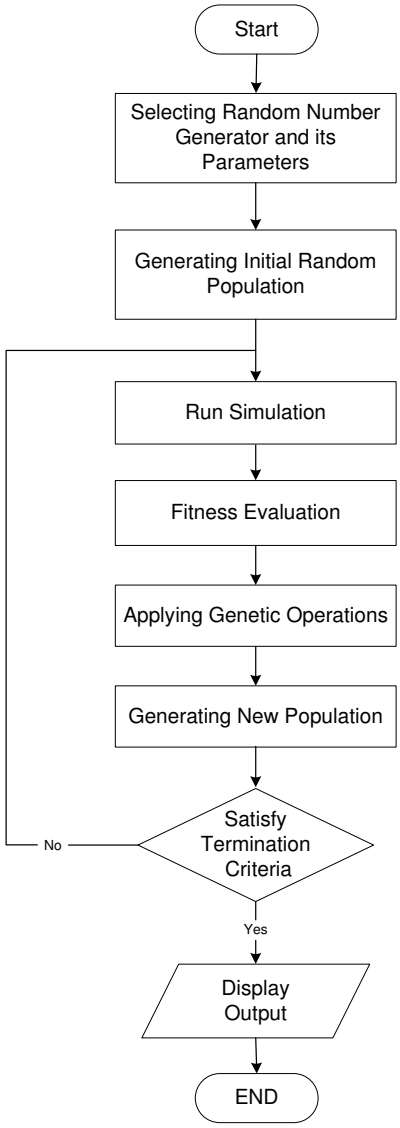

Figure 1.4: Flowchart of Proposed CGA Process

The overall CGA process is modeled in Figure 1.4. First, a random distribution is chosen for the reset of the execution. Then, an initial population is defined. There are several techniques available to define this initial population: randomly, using the same random distribution used for the inputs, or a user-defined technique. After

running the simulation, a fitness function is evaluated. This function evaluates how good the previous population is in terms of coverage. There are several ways to define the fitness. In general, the fitness is calculated based on the percentage of the cover points being hit over the total number of cover points in the design.

The fitness evaluation guides the generation of the next generations. For instance, we preserve some of the best elements in the population, perform cross-over operations in order to generate new elements, add some mutations, and keep some random members in order to preserve diversity. The overall run-evaluation-generation process is performed until a given termination condition is hit (e.g., 95% coverage or fitness is 0.99) or if the times run out.

1.6 Thesis Contribution

In this thesis, we have developed a methodology for automatic CDG attempting to enhance the coverage using different probability distribution functions to generate random numbers. In addition, we apply our methodology to a large number of static and temporal coverage tasks of DUV.
In summary, the contribution of our thesis is as follows:

- We developed five random number generators based on different probability distribution functions. The probability distribution functions we used are: Normal distribution, Exponential distribution, Gamma distribution, Beta distribution, and Triangle distribution. We implemented the five RNGs using different algorithms, then we tested and integrated them into the CGA.

- We applied our methodology to a large set of coverage points. We defined both static and temporal properties of the design, and applied our methodology using CGA and the five random number generators to verify those properties. We used a 16×16 packet switch as a design under verification. We implemented

the 16×16 packet switch as behavioral model and the coverage points using SystemC.

- We also implemented the 16×16 packet switch as an RTL model using Verilog HDL and coverage points in SystemVerilog. We simulated the RTL model and compared the results with our approach.

1.7 Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 provides an introduction to the basic principles and operation of the genetic algorithms, then we present general introduction to random number generators and basic concepts, formulae and graphs of Normal, Exponential, Gamma, Beta and Triangle probability distribution functions. We also introduce the basics of SystemC concepts and architecture, as well as the basics of SystemVerilog language. In addition, we provide an illustrative example of 32-bit CPU to demonstrate coverage metrics and extraction of coverage points. In Chapter 3, we describe our detailed methodology and proposed genetic algorithm, then we present random number generator and the algorithms to generate random numbers based on Normal, Exponential, Gamma, Beta, and Triangle probability distribution functions. In Chapter 4, we describe the operation and specification of the 16×16 packet switch as a case study, then we represent the SystemC and the Verilog models of the packet switch. Also, we describe the static and temporal coverage points of the packet switch, then we represent the experiments and discuss the simulation results. Finally, in Chapter 5, we present our conclusion and the future work.

Chapter 2

Preliminaries

This chapter describes briefly the main components on which we are going to build our work in this thesis. The main components are Genetic Algorithms, SystemC, SystemVerilog, coverage metrics, random number generator, and probability distribution functions.

2.1 Genetic Algorithms

Genetic Algorithm is an adaptive heuristic search technique that is used to find an optimum solution to a problem. Genetic Algorithm is based on evolutionary algorithms, which use techniques inspired by the evolutionary theory of Darwin in which the best (fittest) individuals survive. Since its introduction in the 1960's by John Holland [26], it has been experimented and applied to many areas of science and engineering for search, optimization and machine learning problems where search space is very large for traditional optimization methods to find the best solution.

GA is an iterative process implemented as computerized procedures. In each iteration, it searches in different direction for good individuals (potential solution) in the entire population. It performs a comparison among the potential solutions and forwards the best solution over the next generation until it finds the optimum

solution and then terminates the iteration process.

GA does not guarantee a single best solution for the problem, but it always provides a set of optimum solutions more efficiently compared to traditional search techniques. GA considers multiple search points in a population at the same time, and thus reduces the possibility to stuck in local minima during the simulation. This is one of the main advantages of GA.

Simple Genetic Algorithm's functionality starts with constant randomly generated population of size N. The individuals in the population are represented as a binary string of length l. Each individual in the initial population is evaluated in order to generate a new generation. The generation of a new generation is repeated until the best solution is found or other criteria are met. The new generation consists of highly-fitted individuals that are produced by applying mutation to the offsprings. The individuals are evaluated by the fitness function [32].

The individual in the population is called a genome or chromosome of the potential solution. The genome is the basic element of the final solution and there are different ways to represent or encode a genome such as trees, hashes, linked lists, etc., and it always depends on the problem. In general, GA uses a fixed length of bits string to encode the genome as shown in Figure 2.1.

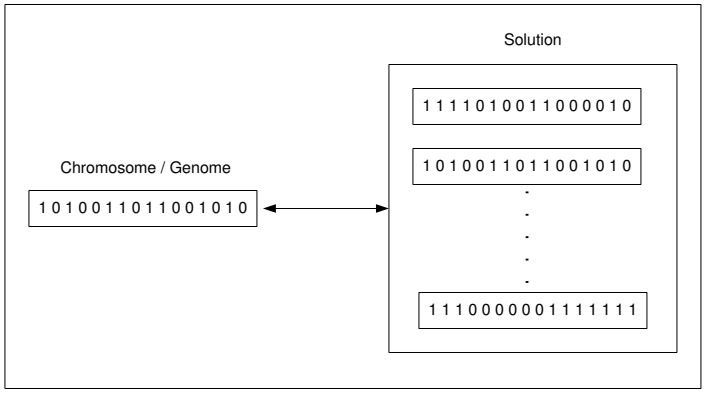

Figure 2.1: Chromosome / Genome presentation

After defining the genome, the population of potential solutions is defined. The population is initially generated randomly most of the time, but it can be generated using some heuristic algorithms. The population can also be loaded from the previous generation. The size of the population is dependant on the complexity of the problem and the size of search space. Population size is an important factor in GA and it could be critical in some applications because it affects population diversity and selective processes and hence it affects the evaluation process. The size of the population in general remains constant over the generations but there are some implementations of GA in which varying or dynamic population sizes are used [24].

Genetic Algorithm has two main operators: crossover and mutation. *Crossover* operation is applied to two selected individuals by exchanging part of their genome to form a new individual. The location of the point where crossover occurs is selected randomly, and the crossover can be at a single point or at two points for same size of individuals. Figure 2.2 illustrates (a) single point and (b) two point crossover operations. The crossover is applied to the individual based on crossover probability Pc, a random number r where $r \in [0, 1]$ is generated. If $r < Pc$ then the chromosome will be selected for the crossover, otherwise it will carryover to the next generation.

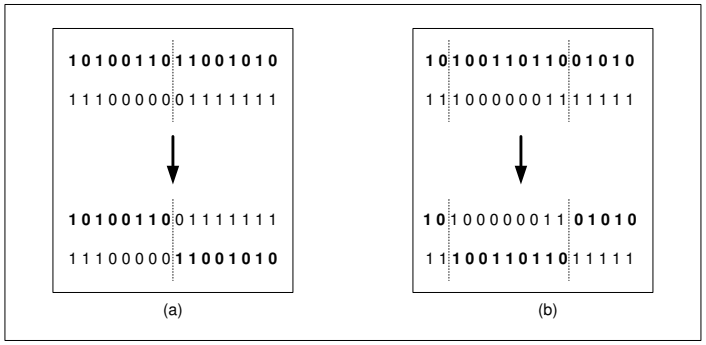

Figure 2.2: Crossover Operation

Mutation operation is applied to one individual by changing an arbitrary bit or

bits from its chromosome. The bits are chosen randomly. The main purpose of the mutation is to avoid slowing or stopping the evolution by minimizing similar chromosomes. Mutation keeps the diversity of the population and helps to explore the hidden areas in the search space. Mutation can occur at any bit in the chromosome string with some probability *Pm*. Figure 2.3 illustrate the mutation operation.

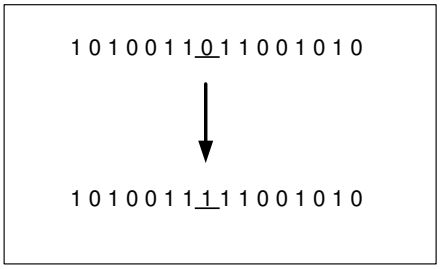

Figure 2.3: Mutation Operation

2.1.1 Selection

Selection is a process where chromosomes are selected from a population for reproduction. During the process of evolution, a portion of the population is selected to produce new offspring by applying certain methods. The new-born offspring replaces the discarded one. The selection process and method should prevent premature convergence and should be applied to a large enough diverse population. There are several selection methods, but the most known are the *Roulette Wheel* selection method, the *Tournament* selection method, and the *Ranking* selection methods.

In Roulette Wheel selection method, a chromosome is selected based on its fitness value and the value of selection probability. Random probability number is generated and compared to the chromosome's fitness value, if the fitness value is equal or less than the probability value, then the chromosome is selected. In Tournament selection, few chromosomes are selected randomly to compete with each other to remain in the population, and finally the most fitted one is selected

for next generation.

2.1.2 Evaluation

Evaluation is very important in the genetic algorithm to find the optimum solution. It is a process to evaluate the potential solution by using evaluation function or fitness function. Fitness function is always dependant on the problem and it is very important for an efficient evolution. There is no specific fitness function or method to evaluate the potential solution. For example, assume the function F is depend on three independent variables x, y, and z, $F=f(x, y, z)$. The values of x, y, and z will lead the value of function F to zero or close to zero ($F=f(x, y, z) \rightarrow 0$). The closer the value of F is to zero, the higher the fitness value. The above example seems simple but in the real world where complex problems exist, it is very difficult to formulate a fitness function that can express the effectiveness of the potential solution. The fitness function should have all the information needed by GA for guidance through search space to determine the correct potential solution.

The operation of of simple Genetic Algorithm is described in the following steps :

- Initial Population
  - create initial population
  - evaluate individual in initial population
- Create new population
  - select fit individual for reproduction
  - generate offspring with genetic operator crossover
  - mutate offspring
  - evaluate offspring

2.2 Random Number Generator

Increase in the design complexity in terms of inputs and functionalities make it almost impossible to build testbenches or test cases that completely verify a design's features and its functions. Therefore, the solution is to create test vectors randomly using a random number generator [36].

Random number generators are widely used in many verification environments and are heavily adapted by many simulation-based hardware verification tools such as Specman [37], VCS [17], and QuestaSim environments [12]. Also, they are used by Neural Network and Genetic Algorithms during the process of learning and evolution. Random number generators that are used in simulation-based verification should be powerful enough to not produce sequences of repeated numbers within short cycles that may be shorter than the simulation cycle. In general, poor random number generators lead to misguided results. The random numbers generated by verification tools are not completely random, but for practical purposes RNGs are considered to be random if they follow the following properties:

1. Repeatability: the sequence of the generated numbers is the same for all seeds.

2. Randomness: the generated random numbers should be pure random and pass all statistical tests for randomness such as frequency test, gap test, serial test, and permutation test [18].

3. Long period: the random numbers should be generated for a period that is much longer than the simulation time.

4. Insensitive to seeds: randomness and period of the generated numbers should not depend on the initial seeds.

In addition, random numbers are generated based on different probability distribution functions. The most common distribution is Uniform distribution.

2.3 Probability Distribution Function

Probability distribution is a fundamental concept in statistics, and the Probability Distribution Function (PDF) is a function that describes values and their frequencies of occurrence at random events. The values must cover all possible random events and the total sum of probability is equal to 100%. It is defined in term of Probability Density Function (PDF) and Cumulative Distribution Function (CDF). The probability distribution is classified into two types: discrete probability distribution functions and continuous probability distribution functions. Like discrete distribution function, continuous probability distribution functions are used in many applications, and one of those applications is to generate random numbers based on different probability distribution functions. Examples of such distribution functions are Uniform distribution, Normal distribution, Beta distribution, Gamma distribution, Exponential distribution, Rice distribution, Triangular distribution, Wigner Semicircle distribution, Weibull distribution, and Hyperbolic distribution [38].

2.3.1 Uniform Distribution Function

Uniform distribution is defined by two parameters: a (lower limit) and b (upper limit), and the probability of any value that occurs between a and b is equal (Figure 2.4). A random number is said to be uniformly distributed in $a \leq x \leq b$ if its probability density function is described in Equation 2.1, where the probability of x to be generated is equal along the interval [a, b].

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq x \leq b \\ \\ 0 & \text{for } x < a \text{ or } x > b, \end{cases} \tag{2.1}$$

A Uniform distribution is widely used in generating random numbers, which is used in many applications. The Mersenne Twisted (MT) pseudo algorithm is described in [22] and it generates a 32-bit long random integer. There are other

random number generators that can use uniform RNG to generate random number based on their distributions, such as Normal Distributions, Gamma Distributions, Beta Distributions, Exponential Distributions, and Triangle Distribution.

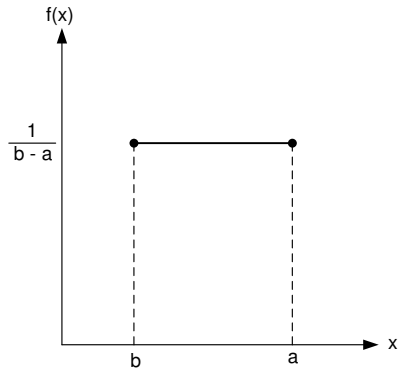

Figure 2.4: The Uniform Distribution Function

2.3.2 Normal Distribution Function

Normal distribution, also known as Gaussian distribution, is a continuous distribution function that is defined by two parameters: *mean* (μ) and standard deviation (σ) of the distribution. The probability density function (pdf) of Normal Distribution of variable x is defined as:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} \tag{2.2}$$

Where: x = variable μ = mean (average) σ^2 = variance

The Normal distribution is symmetric around its mean, as shown in Figure 2.5. The typical standard Normal distribution is obtained when $\mu = 0$ and $\sigma = 1$. It is noted that there are two controlling parameters for Normal distribution: by controlling those parameters we can plot different shapes of Normal probability density function.

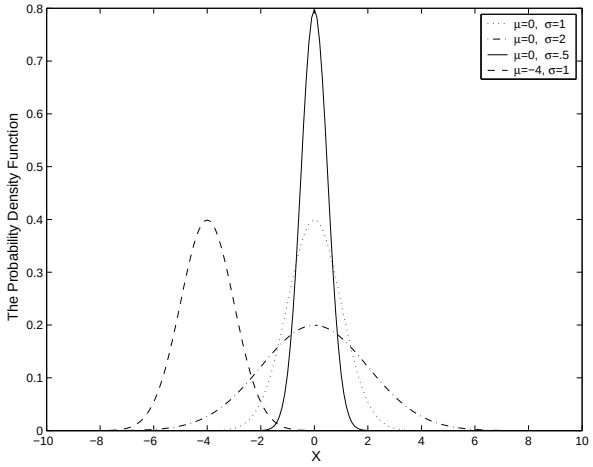

Figure 2.5: The Normal Distribution Function

2.3.3 Exponential Distribution

Exponential distribution is another continuous distribution function that is defined by the following equation:

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & , x \geq 0 \\ \\ 0 & , x < 0, \end{cases} \tag{2.3}$$

where: $\lambda > 0$ and $x \in (1, \infty)$

Exponential distribution describes time intervals between events that occur continuously and independently at a constant average rate (Figure 2.6).

2.3.4 Gamma Distribution

Gamma distribution belongs to non-symmetric continuous probability distribution that has two non-integer parameters, Scale factor θ and Shape factor k (Figure 2.7. Gamma distribution can be used to drive other distributions such as Exponential distribution and Uniform distribution by changing the values of its parameters. The probability density function of the Gamma distribution can be expressed in terms

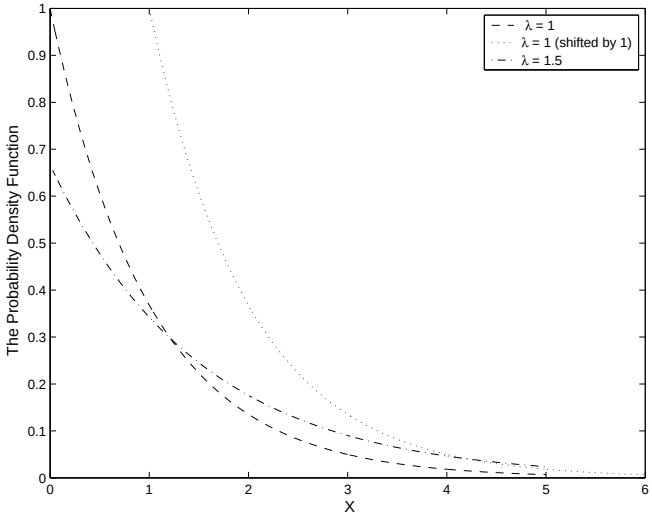

Figure 2.6: Exponential Distribution Function

of the gamma function Γ as follows:

$$f(x;k,\theta) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)} \tag{2.4}$$

where: k and $\theta > 0$

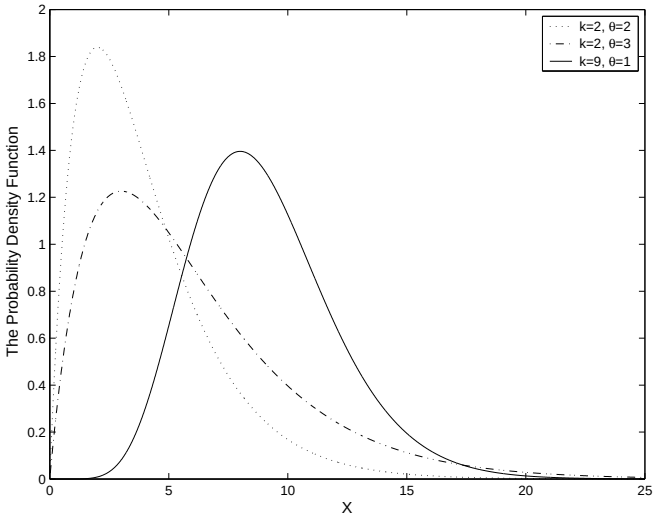

Figure 2.7: Gamma Distribution Function

2.3.5 Beta Distribution

Beta distribution also belongs to continuous probability distributions and it is defined on the interval [0, 1] and parameterized by two parameters, typically denoted by α and β whose values are greater than 0. The probability density function of the beta distribution is described by following equations:

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\int_0^1 u^{\alpha-1}(1-u)^{\beta-1}\,du} = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \tag{2.5}$$

where: $0 < x < 1 \quad \alpha$ and $\beta > 0$

There are different shapes of Beta distributions depending on the values of its parameters (Figure 2.8). For example, if $\alpha < 1$ and $\beta < 1$ then we get a U-shaped curve, for $\alpha = \beta = 1$ we get a Uniform distribution.

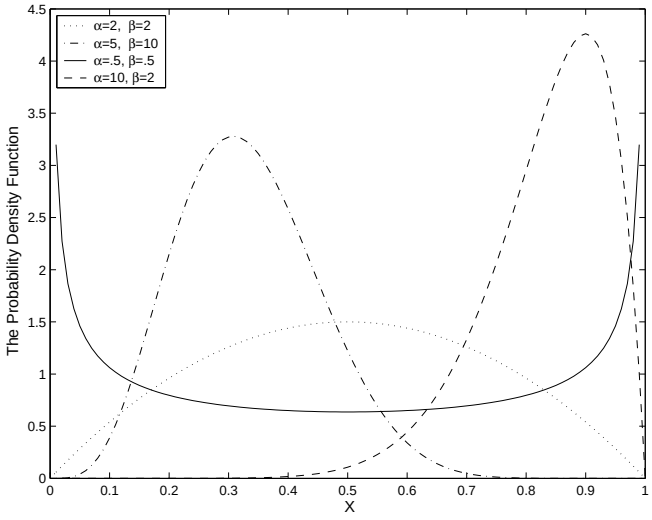

Figure 2.8: Beta Distribution Function

2.3.6 Triangle Distribution

Triangular distribution is defined by three parameters: lower limit a, mode c, and upper limit b (Figure 2.9). Its probability distribution function (pdf) is given in

the equation 2.6 and its Cumulative distribution function (cdf) is given in equation 2.7, from which we can extract the inverse cdf which is presented in equation 2.8.

$$f(x;a,b,c) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x \leq c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c \leq x \leq b \\ 0 & \text{for any other cases} \end{cases} \tag{2.6}$$

$$F(x;a,b,c) = \begin{cases} \frac{(x-a)^2}{(b-a)(c-a)} & \text{for } a \leq x \leq c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{for } c \leq x \leq b \end{cases} \tag{2.7}$$

$$F_X^{-1}(x) = \begin{cases} \sqrt{x(b-a)(c-a)} + a & , 0 \leq x < (c-a)/(b-a) \\ b - \sqrt{(1-x)(b-a)(b-c} & , (c-a/(b-a) \leq x \leq 1 \\ 0 & , x < 0 \\ 1 & , x > b, \end{cases} \tag{2.8}$$

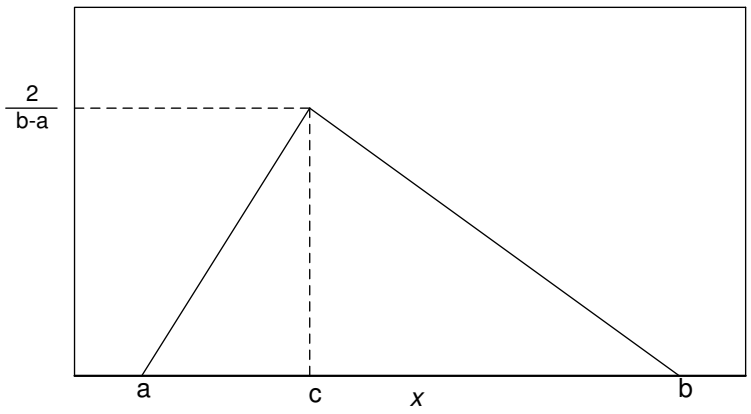

Figure 2.9: Triangle Distribution Function

2.4 SystemC

SystemC is an open source system level design language based on C++. It consists of a C++ class library and other libraries and a simulation scheduler that supports clock, concurrent behaviours, and timed event simulations and descriptions of hardware. In addition, SystemC provides a construct that helps to design hardware such as signals, ports and modules. Signals, ports and modules are similar to corresponding terms in HDL. SystemC can be run using most of the standard C++ developing environment such as Microsoft Visual C++ or GNU gcc, and its compilation process is shown in Figure 2.10. In 1999, the first version of system level language called SystemC was released. SystemC was a result of contributions from many research groups and EDA companies, and after few years of improvement, SystemC has been standardized by IEEE in 2005 [5].

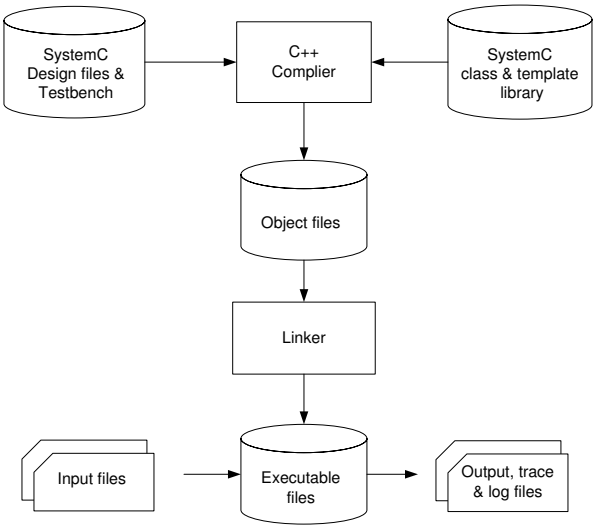

Figure 2.10: SystemC Compilation Process

SystemC is a powerful and general-purpose language, and it is very suitable for describing complex and sophisticated systems. It can be used to build reusable verification components and scalable testbenches [12]. Moreover, SystemC can be

used to describe assertions for assertion based verifications such as in FSM machine. Complete verification environments can be built from SystemC not only for design written in SystemC but also for designs written in other HDL [5].

2.4.1 SystemC Architecture

The main architecture of SystemC is shown in Figure 2.11. The shaded area represents the SystemC main class library (core layer) which is built on the top of C++ standard programming language. The layer shown immediately above the SystemC class library represents proprietary SystemC. C++ libraries are associated with specific design or verification methodologies or specific communication channels. The user may use them and can add more libraries to them, and it is not a part of standard SystemC. The main SystemC class library is divided into four categories:

1. The core language
2. The SystemC data types
3. The predefined channels
4. The utilities

The core language and SystemC data type are more essential and are typically used together even though they may be used independently of one another. In addition, the core of SystemC contains a process scheduler which makes the core a simulation engine of the SystemC.

Processes are executed in response to the notification of events. Events are notified at specific points in simulated time. In the case of time-ordered events, the scheduler is deterministic. In the case of events occurring at the same point in simulation time, the scheduler is non-deterministic [3].

In the core language, modules are basic structural blocks or classes which represent a system in a hierarchical order. A module holds a structure that represents

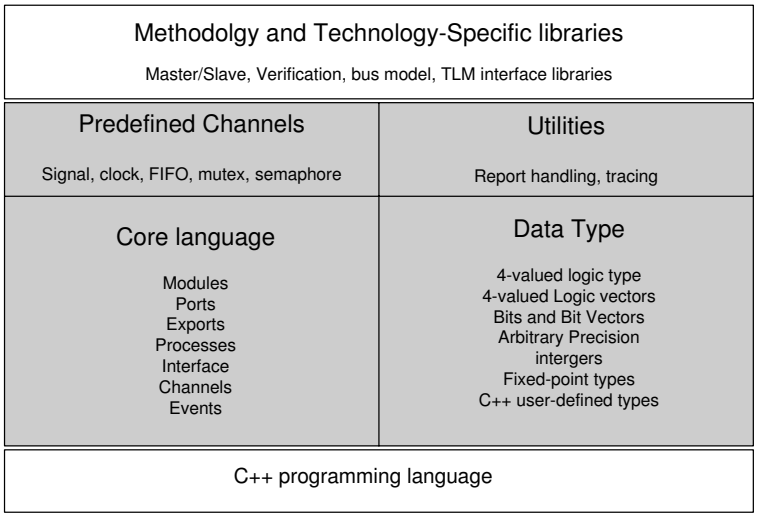

Figure 2.11: SystemC Architecture

a function of a hardware or software or both. Each module may contain variables, ports, signals, channels, process, events and other core language elements and data types. The ports are used to communicate with the outside environment of the module while the processes perform the functionality of the module in a concurrent way (Figure 2.12).

There are three kinds of processes: *methods* (SC_METHOD), *threads* (SC_THREAD), and *clocked threads* (*cthread*) (SC_CTHREAD). They run concurrently when triggered by clock or events listed in the sensitivity list. Channels contains communication mechanisms and can be used to connect processes together. Channels are considered to be an implementation of the interfaces. There are two types of channels: *hierarchical channels* and *primitive channels*. Hierarchal channels are modules while primitive channels are not modules.

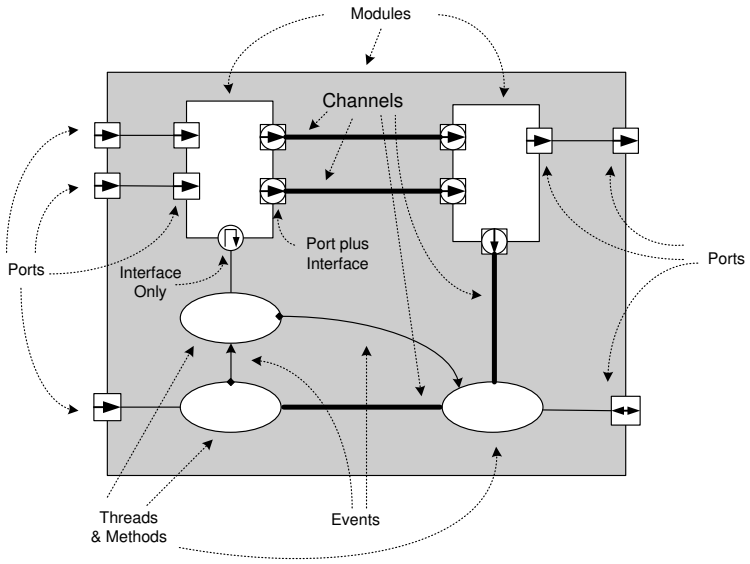

Figure 2.12: SystemC Components [5]

2.5 SystemVerilog

SystemVerilog is a major extension of Verilog HDL, it was developed originally by Accellera, a consortium of EDA companies and users, who wanted to create next generation of Verilog. SystemVerilog became IEEE standard in 2005 [36].

SystemVerilog is considered as a unified language for specification, design, and verification. It provides syntax and semantics for assertion-based verification or what is called SystemVerilog Assertion (SVA) and, coverage-based verification. This is considered to be a major advantage of SystemVerilog [1]. Another advantage of SystemVerilog is the Object Oriented Programming (OOP) capabilities that helps to model a design and testbench at higher level of abstraction. Moreover, it allows engineers to build reliable, repeatable, and flexible advanced verification environment that can be used in verifying multiple designs [36].

SystemVerilog introduced many new features that can be used in design and

verification of electronics circuits. Some of these features are: new data types, procedures and program blocks, interfaces, classes and inheritance, constraints random number generators, queues and lists, assertion and coverage, synchronization, and other improvements to existing Verilog features.

2.6 Illustrative Example: 32-bit CPU

In this section, we use an illustrative example to demonstrate the notion of functional coverage. We use 32-bit CPU (MIPS I) as an example because it has various functions and components to demonstrate the concept of functional coverage. MIPS I is one family member of the MIPS processor which is a RISC microprocessor architecture developed by MIPS Technologies, and it uses 32-bit Instruction Set Architecture (ISA) [16]. There are 32 general purpose 32-bit registers *r0-r31* where register *r0* is hardwired to the constant 0. MIPS has five pipelined stages, starting with the Fetch stage, then the Instruction decode stage, the Execution stage, the Memory stage and the Write back stage. The instructions are divided into three types: R, I, and J, and every instruction starts with the opcode.

2.6.1 Functional Coverage Verification

Functional verification of a CPU is one of the most challenging and complex tasks to perform. The major issue lies in identifying the functional coverage metrics [25]. We focus on functional coverage and present four verification models of different functions in a pipelined microprocessor.

We use the terminology of Static coverage point to describe static property and Temporal coverage point to describe temporal properties. *Static* property refers to the validity of certain conditions applied to the value of variables at a specific event or clock cycle. For example, detecting the condition where ADD instructions produce a carry. On the other hand, a *temporal* property refers to the validity of

certain conditions on the value of the variables across several clock cycles or events. For example, when an interrupt instruction is detected, the action should be followed with 2-3 clock cycles.

Several coverage metrics can be used to verify different functions of a CPU, such as finite state machine metrics, instruction set metrics, internal register metrics and exceptions metrics.

Finite State Machine Metrics

The MIPS architecture consists of five pipelined stages : Fetch, Decode, Execute, Memory, and Writeback as shown in Figure 2.13. Instructions pass through these stages in a synchronized clock cycle. One clock cycle is needed for each stage. The pipelined stages can be expressed as a Finite Sate Machine (FSM) and define the datapath for all the instructions. In order to verify the FSM of the pipelined stage, the coverage points should represent the paths and the states of the FSM.

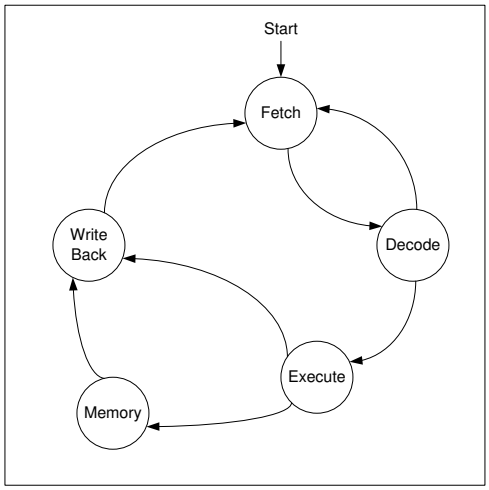

Figure 2.13: MIPS I CPU Finite State Machine

To fully verify the FSM, we have to verify every path and the transition of every state. This can be done using assertion-based verification or coverage-based verification. In assertion verification, each state can be represented as an static

property and the transition between stages can be represented as temporal property. An example of static property could be if an ADD instruction arrives at the Execute state and the operands of the ADD instruction are detected at the input of Execute state at certain clock cycle. An example of temporal property could be "if at a certain clock cycle the output of the Fetch state is an ADD instruction, then at the next clock cycle, the output of the Decode state should be an ADD instruction". Since assertion-base verification is consider a white box verification method, there are many properties that can be extracted from design specification and design HDL implementation.

In the coverage based verification, the functions of the design is expressed using the coverage tasks concept, where each function is defined as a coverage point. The idea is to cover every state and path of the FSM. The following finite paths are extracted from the FSM:

1. Fetch - Decode - Fetch
2. Fetch - Decode - Execute - Writeback
3. Fetch - Decode - Exceute - Memory - Writeback

The first path can be verified if branch instructions with branch taken or jump instructions are detected. In case of branch instructions *beq $rs, $rt, imm* and *bnq $rs, $rt, imm*, three coverage points can be declared, Op_Code, RS, and RT. In case of jump instructions *j destination* and *jr $rs*, two coverage points can be declared, Op_Code and RS. The SystemVerilog syntax of the coverage points is described as follows:

```
covergroup Path1_Branch @(ClockEvent)
     coverpoint Op_Code { bins OpCode = {4,5}; }
     coverpoint RS { bins RSValue = {0, 8:15}; }
     coverpoint RT { bins RTValue = {0, 8:15}; }
```

```
 corss Op_Code, RS, RT;
endgroup
```

In the above code, the cover point `Op_Code` is covered when the value of the input `OpCode` is detected to be 4 or 5 at the `ClockEvent`. Similarly, the cover points `RS` and `RT` are covered when their associated variables are equal to 0 or any number from 8 to 15. The `corss Op_Code, RS, RT` is the cross coverage of all the three points and if all the points are covered at certain clock event then the whole group is covered. Same explanation is applied the following SystemVerilog syntax:

```
covergroup Path1_Jump @(ClockEvent)
     coverpoint Op_Code { bins OpCode = {2,8}; }
     coverpoint RS { bins RSValue = {20:23}; }
     corss Op_Code, RS;
endgroup
```

The second path can be easily verified with any arithmetic or logic instructions and with branch-not-taken instruction. The coverage point can be expressed easily with different types of instructions operands. We can use more than one coverage group but since we need a high rate of coverage we will use only one coverage group. The SystemVerilog syntax of the coverage points is described as follows:

```
covergroup Path2 @(ClockEvent)
     coverpoint Op_Code { bins OpCode = {32:39, 8:15}; }
     coverpoint RS { bins RSValue = {8:15, 16:25}; }
     coverpoint RT { bins RTValue = {8:15, 16:25}; }
 corss Op_Code, RS, RT;
endgroup
```

The third path can be verified with *load* and *store* instructions since both can access the memory. The SystemVerilog syntax of the coverage points as follows:

```
covergroup Path2 @(ClockEvent)
     coverpoint Op_Code { bins OpCode = {32, 35, 40, 43}; }
     coverpoint RS { bins RSValue = {8:15, 16:25}; }
     coverpoint Imm { bins ImmValue = {0:65535}; }
 corss Op_Code, RS, Imm;
endgroup
```

Instruction Set Metrics

MIPS has more than 32 instructions excluding the floating point instructions. Those instructions are divided into three main types. In order to fully verify the function of the CPU, all possible instructions with all possible combination of their operand must be applied. In a real verification environment, it is quite an impossible job. Therefore, a selected set of instructions are applied as test vectors. We define three main coverage groups for each instruction type. It is also possible to define more than three coverage groups based on the functions of the instructions such as a group for arithmetic instruction and another for shift, logic, set, load, store and branch instructions.

The coverage group as it is written in SystemVerilog syntax is as follows:

```
covergroup RType @(ClockEvent)
     coverpoint Op_Code {
     bins A = { [0:9]};
     bins B = { [16:19], [24:27]};
     bins C = { [32:39]};
     bins D = { [42, 43];
          }
     coverpoint RS {
          bins RS_A = { [8:15], [16:25] };
          }
```

```
    coverpoint RT {
        bins RT_A = { [8:15], [16:25] };
        }
    coverpoint RD {
        bins RD_A = { [8:15], [16:25] };
        }
    cross Op_Code, RS, RT, TD;
endgroup
```

Similarly, we can write the coverage group for I-type and J-type instructions.

Internal Register Metrics

MIPS has 32-bit General Purpose Register (GPR) labeled as R0, R1,..., R32. The register R1 is hardwired to logic 0 and its value is always 0. Some of the GPR are reserved for special purposes. For example, R31 is used to store the return address, R29 is used by stack pointer, the R26 and R27 are used by the OS kernel [16].

To verify the GPR we need to verify the Read/Write operation to the registers. Verifying 32 registers is not an exhausting process and it can be done using direct testing or random testing. In the random testing, we detect any of the 32 registers in any instruction at the position of source or target operand for read operation and destination operand for write operation. For example, the ADD instruction required 3 operands as ADD $rd, $rs, $rt, if rd = 10001 and rs = 11000, and rt = 01000, then we cover R17 for write operation and R25 and R8 for read operation. The coverage group that verifies the Read/Write operation is similar to the instruction set coverage group except we have two cover points for instruction and four cross coverage points.

```
covergroup Read_Write @(ClockEvent)
    coverpoint ROp_Code {
```

```
      bins A = { [32:39]};
        }
      coverpoint IOp_Code {
      bins B = { [8, 15];
           }
      coverpoint RS {
           bins RS_A = { [8:15], [16:25] };
           }
      coverpoint RT {
           bins RT_A = { [8:15], [16:25] };
           }
      coverpoint RD {
           bins RD_A = { [8:15], [16:25] };
           }
      cross ROp_Code, RD;      // write operation
      cross IOp_Code, RT;      // write operation
      cross ROp_Code, RS, RT;  // Read operation
      cross IOp_Code, RS;      // Read operation
endgroup
```

Flag Coverage GPR

In computer architecture, flags refers to one or more bit registers that are part of the Status Register. The function of the flags or the status register is to indicate the post-operation conditions; these flags are: zero flag, carry flag, overflow flag and negative flag. For example, the carry flag is a single bit register that is set to 1 if the add instruction produces a carry. It is essential to verify the flag register because some of them if not all of them, such as carry flag, are considered to be among corner cases in coverage verification [37].

The flags register can be verified using assertion based methodology or coverage based methodology. In assertion based verification, the flags can be expressed as an static cover point which can be detected at the clock event. Similarly, other flags can be expressed using SystemVerilog assertion syntax.

```
property flag_carry
          @(posedge Clock) (Carry_Flag == 1);
endproperty
```

In the above code, the property `flag_carry` is covered if the `Carry_Flag` is equal to 1 at the positive edge of the clock.

In coverage based verification our objective is to find suitable instructions that produce carry. Thus, the coverage group will consist of instructions, operands and the flag register. The coverage group can be expressed using SystemVerilog syntax as follows:

```
covergroup Cov_Carry @(ClockEvent) {
    coverpoint Op_Code { bins OpCode = {32, 33, 8, 9}; }
    coverpoint RS { bins RSValue = {8:15, 16:25}; }
    coverpoint RS { bins RSValue = {8:15, 16:25}; }
    coverpoint RS { bins RSValue = {8:15, 16:25}; }
    coverpoint Imm { bins ImmValue = {0:65535}; }
    coverpoint carry { wildcard bins A = 2'b1}; }
corss Op_Code, carry;
endgroup
```

Chapter 3

Improving Coverage using a Genetic Algorithm

In this chapter, we present our detailed methodology for RNG based on different probability distributions such as Normal, Exponential, Gamma, Beta, and Triangle probability distribution. Also, we describe our proposed Cell-based Genetic Algorithm and how we represent the cells based on different probability distributions. Then, we describe various methods and techniques to implement and test the random number generators based on different probability distribution such as Normal, Exponential, Gamma, Beta and Triangle distribution. Also, we present histograms that show the results of our implemented RNGs.

3.1 Methodology

The proposed methodology consists of several steps (Figure 3.1). First, we define and model the coverage tasks of the DUV as Static and Temporal coverage tasks. Each model of the coverage tasks has several coverage points that can act as individual coverage points or as group of coverage points. Second, we select one of the probability distributions and its parameters and run the simulation, we repeat

the simulation for the same distribution but with different parameters of the probability distribution. Third, we repeat the simulation for all distributions and their parameters, and finally we compare the results for all the simulations. In addition, we repeat the above steps for a selected number of coverage points or sets of coverage points in order to study the effect of different distributions on the number of coverage points.

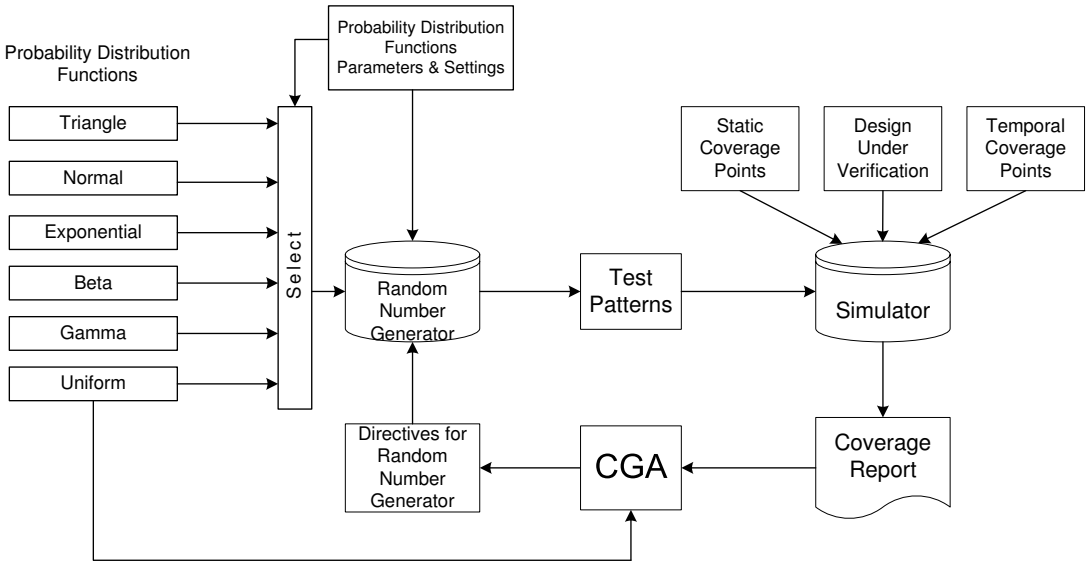

Figure 3.1: CGA Methodology with Multiple Probability Distributions

Figure 3.1 shows different probability distribution functions that are used to generate random numbers. The user selects an appropriate probability distribution function and its parameters values. Only one distribution along with its parameters values is selected during the simulation. In this thesis, we choose five probability distribution functions, which are *Normal (Gaussian), Exponential, Gamma, Beta*, and *Triangle* distributions. Uniform, Normal and Exponential distribution are chosen because they are widely used in many hardware design and simulation tools, while Gamma and Beta distributions generate different probability curves based on values of their parameters. Triangle distribution is chosen because its parameters

are directly related to the input domain of DUV and can be used to identify range in which maximum coverage can be achieved by controlling only one parameter, the mode. For example, if a DUV has input domain in the range of 0 to 1000, then the triangle distribution parameters will setup as follows: lower limit a=0, upper limit b=100 and the value of mode c will be controlled by the CGA within the range 0 to 100 to find the optimum coverage. Uniform distribution is also used by the CGA to generate a random number for its operations such as mutation and crossover.

The overall flow of the proposed CGA is described in Figure 3.2. Initially, a random distribution is selected by the user along with the values of its parameters. For example, if Normal distribution is selected then the values of mean μ and standard deviation σ are set to predefined values. Next, the values of GA parameters are read from an input file. Some of the GA parameters are: maximum population size, maximum number of generations, number of cell, number of simulation cycles, tournament size, probability values for crossover, mutation and elitism, selection type, fitness definition, fitness evolution formula and other related parameters. After that, the initial population is generated by using either a free random initialization technique or fixed random initialization techniques.

The process runs as long as the number of generations is less than the maximum generation number that is set by the user. For each population size, the program resets the coverage counters which are variables that count the number of hits for each coverage points. The simulation for the DUV is run for pre-defined cycles which are equal to number of simulation cycles multiplied by number of clock tics. At the end of the simulation, coverage information is evaluated by a fitness function and the coverage result is stored in an output file for each population.

In the next step, we apply genetic operations such as selection, crossover, mutation and elitism on the selected chromosomes to produce new and more fitted chromosomes. Finally, the process generates a new population of the next generation and repeat the process for each generation as shown in Figure 3.2. The program

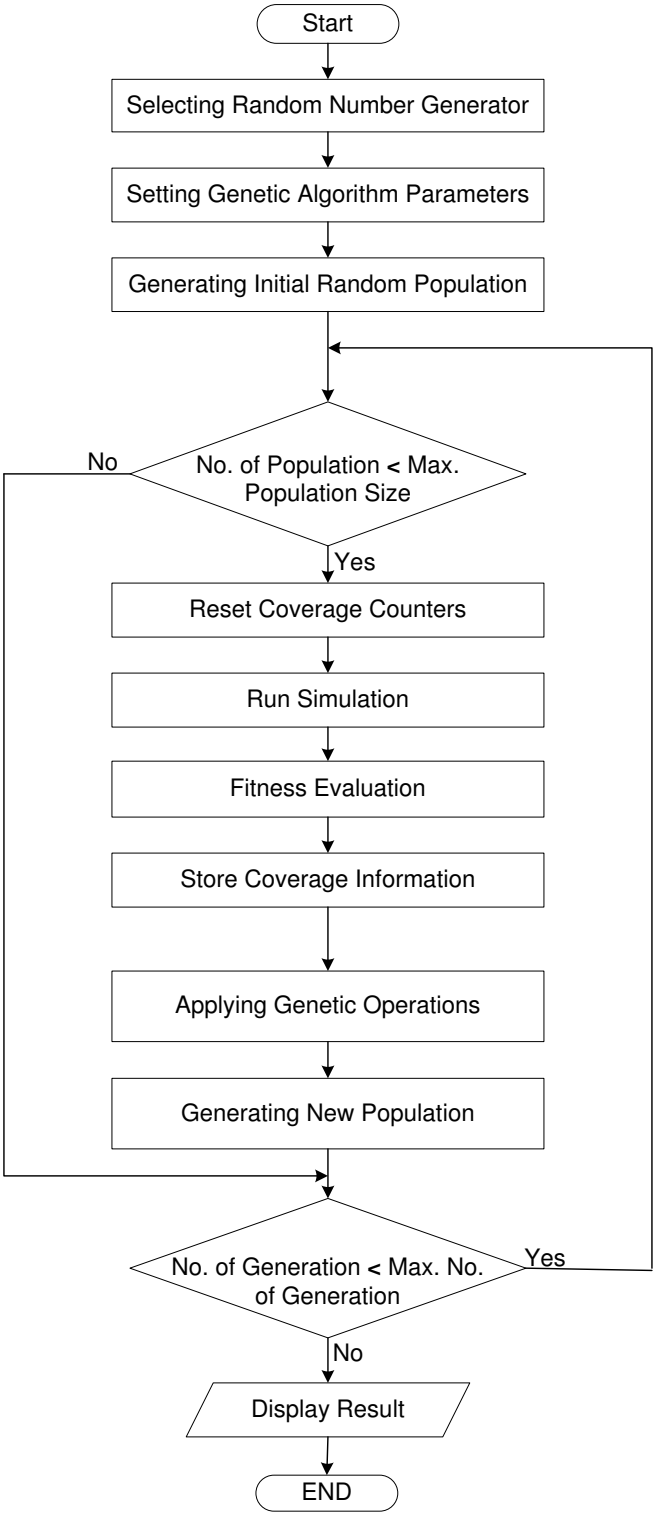

Figure 3.2: CGA Process Flowchart

terminates when the number of generations reaches the maximum setup by the user. At the end, the results are displayed and stored in an output file.

3.2 Proposed Genetic Algorithm

The proposed Cell-based Genetic Algorithm [34] is a search and optimization algorithm used to enhance coverage results. CGA is based on a Genetic Algorithm where the solution is encoded using the concept of cells. Each cell has certain numbers generated randomly between two limits. In general, a Genetic Algorithm has the following components:

1. Solution Representation
2. Initial Population
3. Fitness Evaluation
4. Genetic Operators (crossover and mutation)
5. Other parameters (probabilities, selection methods, and termination criteria)

3.2.1 Solution Representation

Usually, a Genetic Algorithm uses a fixed-length bit string to encode a single value solution. However, the solution of the CDG problem is represented by complex and rich cells. A *cell* is the fundamental unit in a CGA and each cell represents a weighted uniform random distribution over two limits: an upper limit and a lower limit. Moreover, the near optimal random distribution for each test generator may consist of one or more cells according to the complexity of that distribution.

The initial population contains a number of cells, where each cell has three parameters, lower limit (L), upper limit (H) and weight (W). These parameters are generated randomly within the range of input domain ($0 - 2^n$-1), where n is number

of inputs to DUV, and the maximum number of cells in the initial population is defined by the user. The numbers in each cell are generated randomly based on the selected probability distribution as shown in Figure 3.3 with a Normal (3.3(a)) and a Triangle (3.3(b)) distribution.

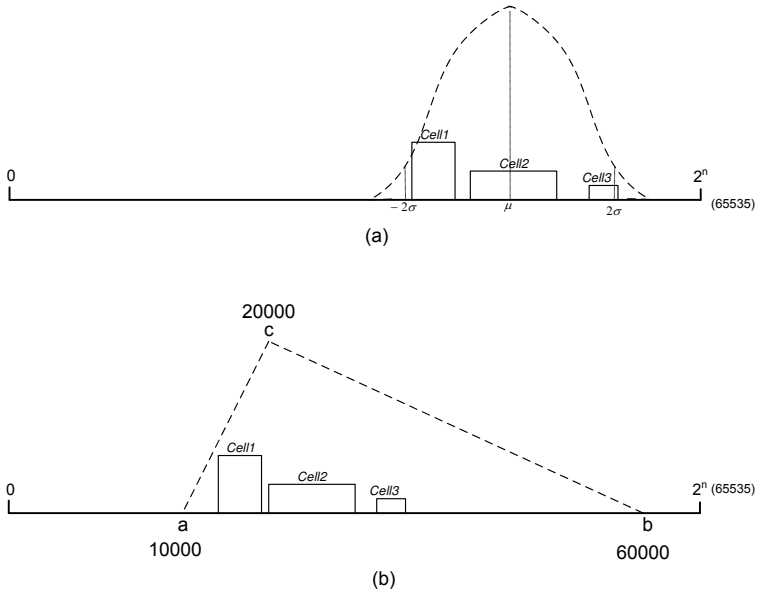

Figure 3.3: Cell Definition

We call the list of cells that represent this distribution a *Chromosome* and the number of chromosomes that represent the whole solution is called a *Genome*. Each chromosome encapsulates many parameters used during the evolution process including the maximum useful range L_{max} for each test generator and the total weight of all cells.

Each cell in the chromosome contains a set of random numbers. Those numbers are considered as inputs to a DUV. During the simulation, the DUV is targeted by every cell in the chromosome, and coverage information is collected for every cell in the chromosome.

3.2.2 Initialization

The CGA uses two initialization schemes:

Fixed period random initialization

In this scheme, we divide the range [0,L_{max}] into n_i equal sub-ranges and then generate random initial cells within each sub-range. This scheme is biased and not random, but ensures that an input with a wide range will be represented by many cells.

Random period random initialization

In this scheme, the cells are generated within the range [0,L_{max}] randomly. The first cell is generated over the range [L_{i0},H_{i0}], then the successive cells are generated within [H_{ij},L_{max}]. In other words, the lower limit of the next cell must come after the end of the previous cell.

3.2.3 Selection

The CGA uses two methods of selection: (1) Roulette Wheel, and (2) Tournament selection [26]. The selection method is chosen by the user at the beginning of the simulation. We use Tournament selection in our experiments because we can control the selection of highly fitted cells by changing the tournament size.

3.2.4 Crossover

Crossover operators are applied on chromosomes with a probability P_c for each chromosome. In order to determine whether a chromosome will undergo a crossover operation, a uniformly random number $p \in [0,1]$ is generated and compared to P_c. If $p < P_c$, then the chromosome will be selected for crossover, or else it will be forwarded without modification to the next generation. The value of P_c is defined

by the user. Crossover is important to keep useful features of good genomes of the current generation and to forward that information to the next generation.

We define two types of chromosome-based crossover: (1) single point crossover, where cells are exchanged between two chromosomes, and (2) inter-cell crossover, where cells are merged together to produce a new offspring. Moreover, we assign two predefined constant weights: $W_{cross-1}$ for *single point crossover* and $W_{cross-2}$ for *inter cell crossover*.

The selection of either type depends on these weights; we generate a uniform number $N \in [1, W_{cross-1} + W_{cross-2}]$ and accordingly we choose the crossover operators as follows:

- Single Point Crossover: $1 \leq N \leq W_{cross-1}$
- Inter-Cell Crossover: $W_{cross-l} < N \leq W_{cross-l} + W_{cross-2}$

3.2.5 Mutation

Mutation operators introduce new features to the evolved population, which are important to keep a diversity that helps the GA to escape from a local minima and to explore hidden areas of the solution space.

Mutation is applied to individual cells with a probability P_m for each cell, in contrast to crossover operators, which are applied to pairs of chromosomes. Moreover, the mutation rate is proportional to the complexity of chromosomes such that more complex chromosomes will be more sensible to mutation.

Due to the complex nature of the genome, we propose many mutation operators that are able to mutate the low limit, high limit, and the weight of cells. According to the mutation probability P_m, we can decide whether a cell will be selected for mutation or not. In the case a cell is chosen for mutation, we apply one of the following mutation operators:

- Insert or delete a cell.

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

The selection of the mutation operator is based on predefined weights associated with each of them. The weight of each operator is setup initially at beginning of the simulation in the same way as other GA parameters. Crossover operators are selected similarly.

3.2.6 Fitness Evaluation

The evaluation of solutions represented by fitness values is important to guide the learning and evolution process in terms of speed and efficiency. The potential solution of the CDG problem is a sequence of weighted cells that constrain a random test generator 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 then 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 be considered simultaneously. For instance, we may achieve 100% for some coverage points while leaving other points totally inactivated. Accordingly, we have designed a four stage fitness evaluation that aims to activate all coverage points before tending to maximize the total coverage rate as follows:

1. Find a solution that activates all coverage points at least one time regardless of the number of activations.
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 *CovRatel*.

4. After achieving these three goals, we consider the average number of activation of each coverage point. Either a linear or a square root scheme will be used to favor solutions that produce more hits of coverage points above the threshold coverage rate.

3.2.7 Termination Criterion

The CGA termination criterion checks for the existence of a potential solution that is able to achieve 100% or other predefined values of coverage rate that is acceptable for all coverage groups. If the CGA terminates without achieving the predefined coverage rate, it terminates when the maximum number of generation is reached and reports the best potential solution of the final generation run.

3.3 Random Number Generator

A random number generator (RNG) is a computational program that generates a sequence of numbers without any specific pattern based Pseudo-Random Number Generator (PRNG) algorithms. Random numbers uniformly distributed between 0 and 1 can be used as a base to generate random numbers of any selected probability distribution. Random number generators are frequently used in simulation based verification. Besides, they are used by GAs and other evolutionary techniques during the process of evolution and learning. Thus, it is required to have a good random number generator. This is why we adopt the Mersenne Twister (MT) algorithm [22] to generate a Uniform random number generator and to use it as a base for other distribution random number generators. A Mersenne Twister is a pseudo-random number generating algorithm designed for fast generation of very high quality pseudo-random numbers. The algorithm has a very high order (623) of

dimensional equidistribution and a very long period of $2^{19937} - 1$. We implemented five different random number generators based on Normal distribution, Exponential distribution, Gamma distribution, Beta distribution and Triangle distribution.

There are a number of methods and techniques that can be used to extract random number algorithms. Some of those techniques are [21]:

1. Inverse CDF technique
2. Acceptance-Rejection technique
3. Monte Carlo method

The inverse method involves finding the inverse cumulative distribution function of PDF and uses a Uniform RNG to generate random values that are substituted in an inverse CDF and generates random numbers. In the acceptance-rejection method two samples are produced randomly x from *F(x)* and y from *U(0,1)*, then whether the function of x is greater than the value of y is tested. If it is, the value of x is accepted. Otherwise, the value of x is rejected and the procedure is repeated again. The Monte Carlo method is more complicated and involves extensive integral computation on the summation of random numbers in certain domain.

There are other techniques for generating random numbers that are specific to some probability distributions such as Box-Muller, Ziggurat algorithm and Polar techniques for Normal distribution [33].

In this thesis, we applied different techniques and methods to formulate algorithms to implement the five distribution RNGs. We used the Inverse Cumulative Distribution Function (CDF) technique to formulate the algorithm for Exponential and Triangle distributions [21]. The Acceptance-Rejection method is used to formulate Gamma and Beta distribution algorithm [30]. The Box-Muller method is used to formulate the Normal distribution [33]. Since all RNG algorithms require one or more independent uniformly distributed random numbers, we use the MT algorithm to generate uniform random numbers for all the other distributions.

All the random number generators were implemented using the C++ programming language and the results were tested with Histograms using SPSS. The results are tested with a large number of data (approximately one hundred thousand of data) and the histograms were compared with plotted probability distribution functions.

3.3.1 Normal Distribution RNG

There are several methods for generating normally distributed random numbers such as the Inverse CDF method, Box-Muller method and Ziggurat method. None of these methods are error free, but they generate random numbers whose mean and standard deviation are close to the expected one. Based on a performance metrics, it is found that Box-Muller methods provide better performance [33]. The Ziggurat method is a more accurate method but its implementation is quite complex. We used the Box-Muller method to generate random numbers based on Normal distribution.

Box-Muller Method

This method converts a pair of independent uniformly distributed numbers into a pair of random number with Normal distribution. The basic formulae for the conversion are described in Equations 3.1 and 3.2.

$$y_1 = \sqrt{-2\ln(x_1)}\cos(2\pi x_2) \tag{3.1}$$

$$y_2 = \sqrt{-2\ln(x_1)}\sin(2\pi x_2) \tag{3.2}$$

We begin with two independent random numbers, x_1 and x_2 in the range from 0 to 1, which are randomly generated from a Uniform distribution. Then apply the above equations to obtain two new independent random numbers which are used to

generate Normal distribution random number. To generate x_1 and x_2 randomly, the MT algorithm is used with some modification to ensure a good quality generator.

Algorithm 1 Pseudo-code of Normal Distribution RNG

1: **if** n is odd **then**
2: $x_1 \leftarrow Uniform\_Random\_Number(0, 1)$
3: $x_2 \leftarrow R8\_Uniform(R32)$
4: $y_1 = \sqrt{-2\ln(x_1)}\cos(2\pi x_2)$
5: $y_2 = \sqrt{-2\ln(x_1)}\sin(2\pi x_2)$
6: **else**
7: $y_1 \leftarrow y_2$
8: **end if**
9: $n \leftarrow n + 1$
10: **return** $Normal\_Random\_Number \leftarrow \mu + \sigma \times y_1$

In Algorithm 1, the *Uniform_Random_Number(0,1)* is a function that generates random numbers between 0 and 1, and *R8_Uniform(R32)* is another function that generates 32-bit long integer random numbers. Both *Uniform_Random_Number(0,1)* and *R8_Uniform(R32)* use the MT algorithm independently to generate the random numbers. The Algorithm 1 is implemented and tested using histogram as shown in Figure 3.4.

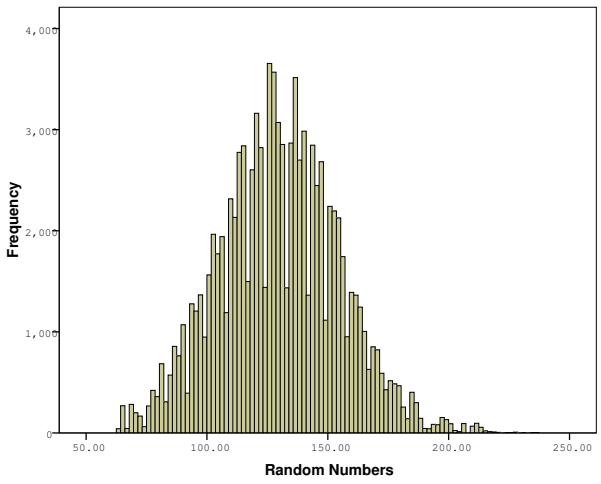

Figure 3.4: Histogram of Normal Distribution Function ($\mu = 130, \sigma = 25$)

3.3.2 Exponential Distribution RNG

To generate exponentially distributed random numbers we use the inverse function of the probability distribution function of exponential distribution (Equation 3.3). *Rand* is a random number generated by the MT uniform random number.

$$Exp\_Rand\_Number = ScalingFactor \times -\ln(1 - Rand) \tag{3.3}$$

Algorithm 2 generates exponentially distributed random numbers and is adapted with minor modifications from a program developed by Eliens [8].

Algorithm 2 Pseudo-code of Exponential Distribution RNG

```
1: H=MT32 ÷ var1
2: L =MT32 % var2
3: T = Const2×L - Const3×H
4: if T is greater 0 then
5:    seed = T
6: else
7:    seed = Const1 + T
8: end if
9: newseed = seed ÷ Const4
10: return Exp_Random_Number ← λ × −log(1 − newseed)
```

In Algorithm 2, *MT32* is a uniform Mersenne Twister random number, *var1*, *var2*, *Const1*, *Const2*, *Const3*, and *Const4* are constant integer. λ is the scaling factor for exponential distribution.

Scale and Shift factor

It is noted that the numbers generated by log*(1-newseed)* are between 0 and 1 and it is required that the number to be generated is an integer greater than 1. Therefore, two parameters are introduced, the *scaling* and *shifting* parameters. The scaling parameter extends the exponential line on the x-axis while the shifting parameter shifts the starting point of the exponential graph towards the positive side of the x-axis by the amount of the shift factor. The effect of scaling and shifting factors is

shown in Figure 3.5. These two factors are important to increase the flexibility of the random number generator and to target different coverage points in the input space.

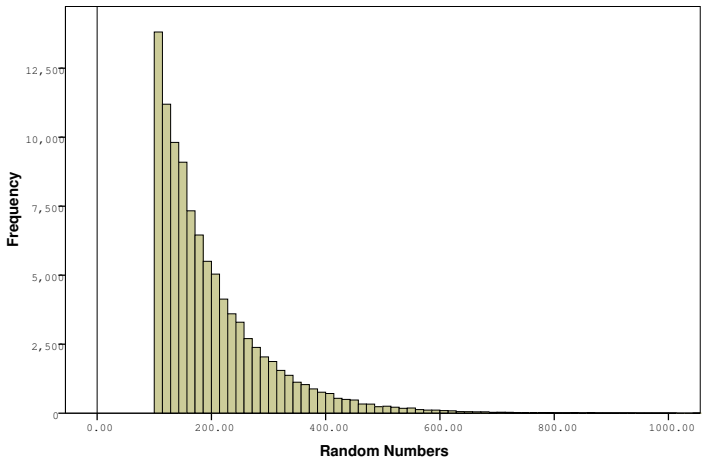

Figure 3.5: Histogram of Exponential Distribution Function with Shift Factor of 100

3.3.3 Gamma Distribution RNG

The Rejection method is used to generate random numbers based on a Gamma distribution. The method and C implementation are adapted with modification from [30]. Algorithm 3 describes the method where two independent random numbers are generated, *value1* and *value2* in the range [0,1]. Then, these two numbers are used to calculate the *Average* and G (Gamma) values. Finally, G value is compared with another random number as shown in line 15 of Algorithm 3, if G is less then the generated random number then we calculate and return Gamma random number.

There are different shapes that can be plotted for a Gamma distribution function based on the value of k and θ, *Gamma(k,θ)*. We chose only three sets of values for Gamma distribution function, Gamma(2,2), Gamma(2,3), and Gamma(9,1). The testing histograms for Gamma(9,1) is in Figure 3.6.

Algorithm 3 Pseudo-code of Gamma Distribution RNG

if K = 1.0 **then**
  return $ScalingFactor \times exponential(1) \div \theta$
end if
if K > 1.0 **then**
  repeat
    repeat
      repeat
        $value1 \leftarrow 2 \times RandReal() - 1$
        $value2 \leftarrow 2 \times RandReal() - 1$
      until (value1 × value1) + (value2 × value2) > 1
      $y \leftarrow value2/value1$
      $Average \leftarrow \sqrt{2 \times (K-1)+1} \times y + (K-1)$
    until Average ≤ 0
    $G \leftarrow (1+y^2) \times \ln(K - 1 \times \log Average/(K-1) - y\sqrt{2(K-1)+1})$
  until $RandReal() > G$
  return $ScalingFactor \times Average/\theta$
end if

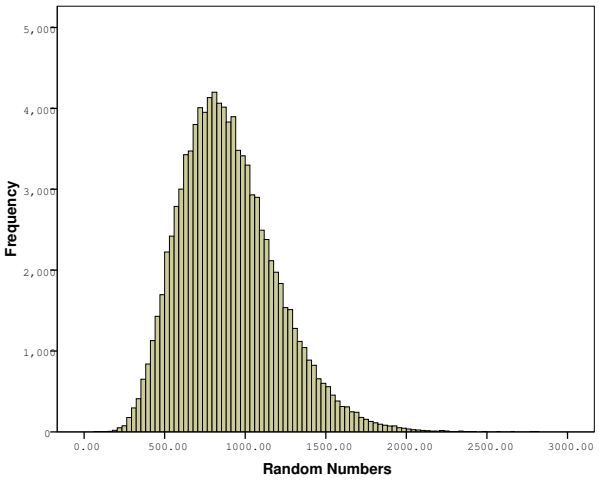

Figure 3.6: Histogram of Gamma Distribution Function ($K = 9, \theta = 1$)

3.3.4 Beta Distribution RNG

Random numbers based on Beta distribution can be generated using the Gamma distribution function that is symbolized as $\Gamma(k, \theta)$. According to the Beta distribution properties, if X=Gamma(α, θ) and Y=Gamma(β, θ), then X/(X+Y) is equal to the Beta(α, β) distributed function where θ is a constant value. Therefore, the Beta distribution random number generator is generated using the following formula.

$$BetaRNG = ScalingFactor \times (\Gamma(\alpha, 1) \div (\Gamma(\alpha, 1) + \Gamma(\beta, 1)))$$

There are different shapes that can be plotted for the Beta distribution function based on the value of α and θ, *Beta(α,θ)*. We choose only three sets of values Beta(2,2), Beta(5,10), and Beta(10,2). The testing histograms for Beta(2,2) and Beta(10,2) are plotted in Figure 3.7 and Figure 3.8, respectively.

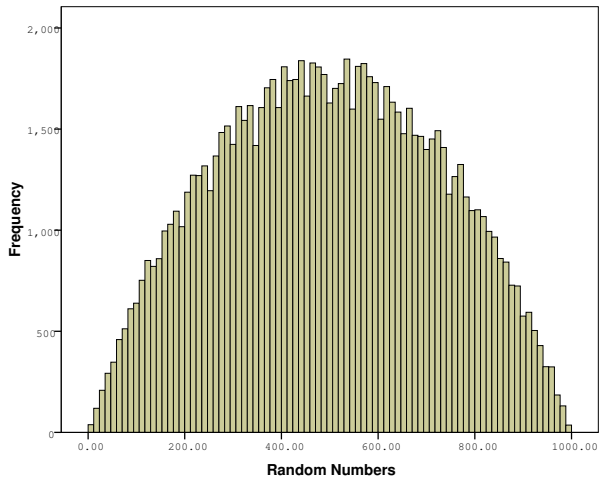

Figure 3.7: Histogram of Beta Distribution Function ($\alpha = 2, \beta = 2$)

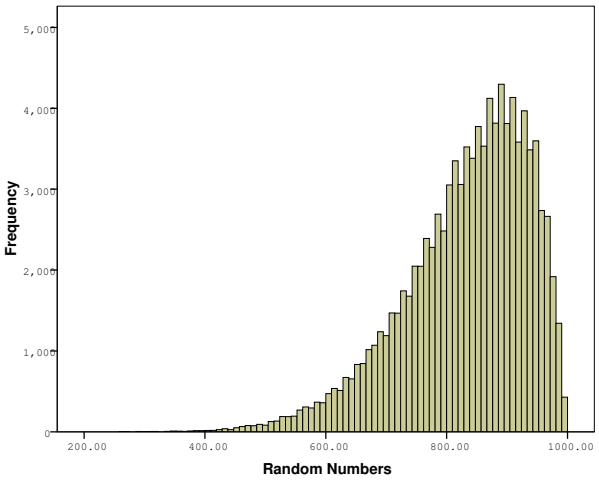

Figure 3.8: Histogram of Beta Distribution Function ($\alpha = 10, \beta = 2$)

3.3.5 Triangle Distribution RNG

Triangular distribution is used where limited sample data is available and the relationship between the variables is known but the data is scarce. It is a very useful distribution for modeling processes where the relationship between variables is known. Random numbers can be generated based on triangle distribution using the Inverse CDF method. The inverse of the CDF is described in Equation 3.4, where x is a uniform random number, it could be greater than 1 or between 0 and 1, *f(x)=(0,1)*. In our program since we need to generate a random integer number that is greater that 1 ($x > 1$), the MT algorithm is used to generate x.

$$F_X^{-1}(x) = \begin{cases} \sqrt{x(b-a)(c-a)} + a & , 0 \leq x < (c-a)/(b-a) \\ b - \sqrt{(1-x)(b-a)(b-c} & , (c-a/(b-a) \leq x \leq 1 \\ 0 & , x < 0 \\ 1 & , x > b \end{cases} \tag{3.4}$$

In Algorithm 4, we calculate the ratio h. Then, we generate random number x in the range between 0 to 1. The function *Rand32* generates a 32-bit long uniform

Algorithm 4 Pseudo-code of Triangle Distribution RNG

1: $h \leftarrow (c-a)/(b-a)$
2: $x \leftarrow Rand32 \times (1.0/4294967296.0)$
3: **if** x $\geq$ 0 AND $\leq$ h **then**
4: **return** $result \leftarrow \sqrt{x \times (b-a) \times (c-a))} + a$
5: **end if**
6: **if** ((x $\geq$ h) AND (x $\leq$ 1)) **then**
7: **return** $result \leftarrow b - \sqrt{(1-x) \times (b-a) \times (b-c))}$
8: **end if**
9: **if** (x > b) **then**
10: **return** $result \leftarrow 1$
11: **end if**
12: **if** (x < 0) **then**
13: **return** $result \leftarrow 0$
14: **end if**

random number and then it is divided by 2^{32}. The value of the *result* represents triangle distributed random number. The testing histograms for Triangle(60,75,90) and Triangle(10,30,150) are plotted in Figure 3.9 and Figure 3.10, respectively.

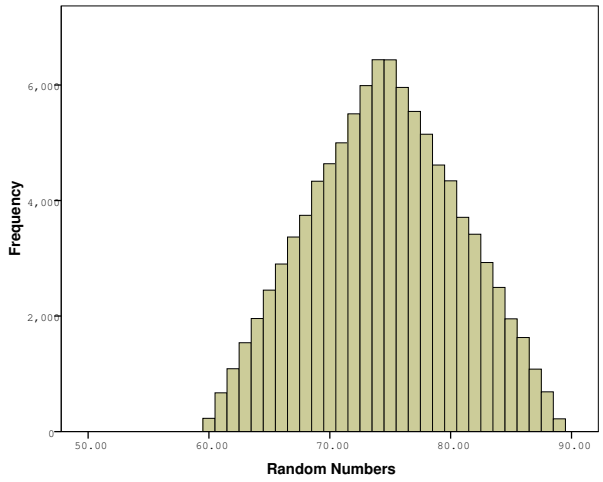

Figure 3.9: Histogram of Triangle Distribution Function (a=60, c=75, b=90)

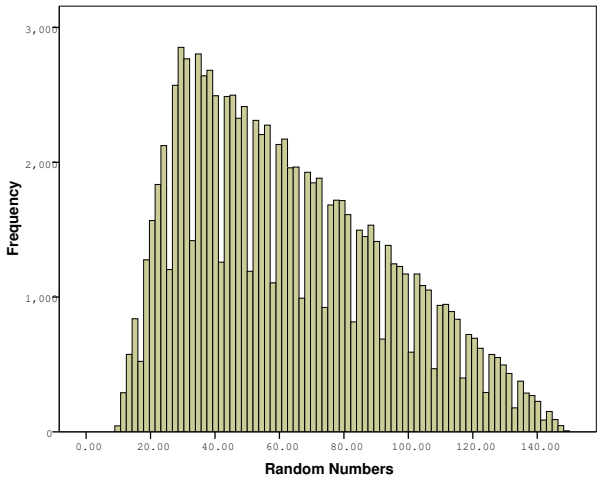

Figure 3.10: Histogram of Triangle Distribution Function (a=10, c=30, b=150)

3.4 Summary

In this chapter, we described in details our methodology for improving the coverage using different probability distribution functions. Then, we described the proposed Cell-based genetic Algorithm where we explained how the probability distributions employed to represent the solution, and how the cells are mapped to DUV. Thereafter, we described the other operations of CGA such initialization, selection, crossover, mutation, fitness evaluation, and termination criteria.

In this chapter, we also introduced random number generators and different methods and techniques that are used to build them. In addition, we described the algorithms and methods that we used to implement random number generators based on Normal, Exponential, Gamma, Beta and Triangle distributions. Also, we presented several histograms that show the distributions of our implemented RNG.

Chapter 4

Case Study Packet Switch

In this chapter, we will apply the methodology and algorithms developed in this thesis on a 16×16 packet switch model. We will first describe the specification of the packet switch and its model in SystemC and Verilog. Then, we will describe the functional coverage points of the packet switch. Finally, we will show the experimental results and discussion.

4.1 Design Description

The 16×16 packet switch we use as a case study is based on the SystemC model of a 4×4 multi-cast packet switch provided by SystemC library [27]. The packet switch uses a pipelined self-routing ring of shift registers to route the incoming packets to their destination. Figure 4.1 shows the general structure of the packet switch where each input and output port has a FIFO buffer depth of 16 each. The input port is connected to a sender process, and the output port is connected to the receiver process. There are two clocks controlling the operation of the switch: the external clock and the internal clock. The internal clock is 16 times faster than the external clock. The shift register is triggered by the internal clock. The packets arrive at the switch randomly and are then routed to destination based on the destination ID

[27].

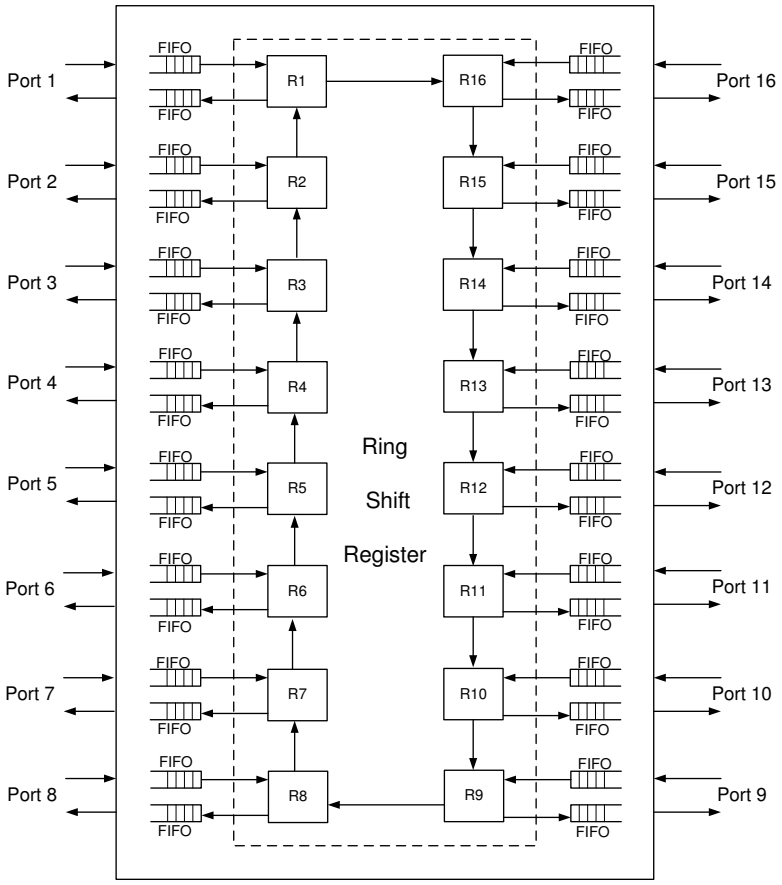

Figure 4.1: Packet Switch Structure

The 16-bit packet contain data (8-bit), sender ID (4-bit) and destination ID (4-bit) as shown in Figure 4.2.

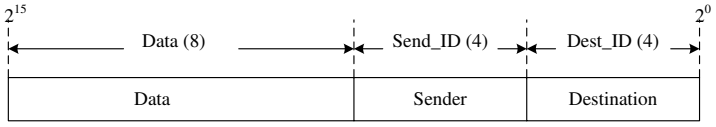

Figure 4.2: Packet Structure

4.1.1 Specification

The packet is considered to be valid if any of its sections has a non-zero value, otherwise the packet is invalid and it is ignored by the packet switch. When the packet is unchanged over few or several clock cycles, then the output of the switch core is zero-value.

The packet switch has 16 input and 16 output ports, and each port is connected to a FIFO. The FIFOs are triggered by the external clock. Each FIFO connected to a register in the ring of shift registers. Each register has two inputs and two outputs. One input and one output are connected to the FIFOs, one input is connected to the output of the previous register in the ring, and one output is connect to the input of the next register in the ring. The operation of shift registers is triggered by the internal clock.

Ideally, the sender sends the packet randomly to input port of the switch. If the FIFO connected to the input is full, then the packet is dropped; otherwise the packet is stored in the FIFO, and in the next positive edge of the external clock cycle, the packet reaches the register. The register receives the packet from the FIFO and sends it to the next register in the ring. The register also receives packets from the previous register in the ring at the positive edge of the internal clock cycle. The register also, checks the destination ID of the arrived packet and directs the packet to the output FIFO if the destination ID matches the register ID.

There is no specific protocol that governs the operation of the packet switch. The packet is routed to its destination based on the destination ID, when the sender ID matches the destination ID, the packet needs 16 internal clock cycles to reach its destination.

4.1.2 SystemC Model

The SystemC model of the packet switch is an abstract model where the sender, receiver, FIFOs and switch core are represented as SC_MODULE processes as shown in Figure 4.3.

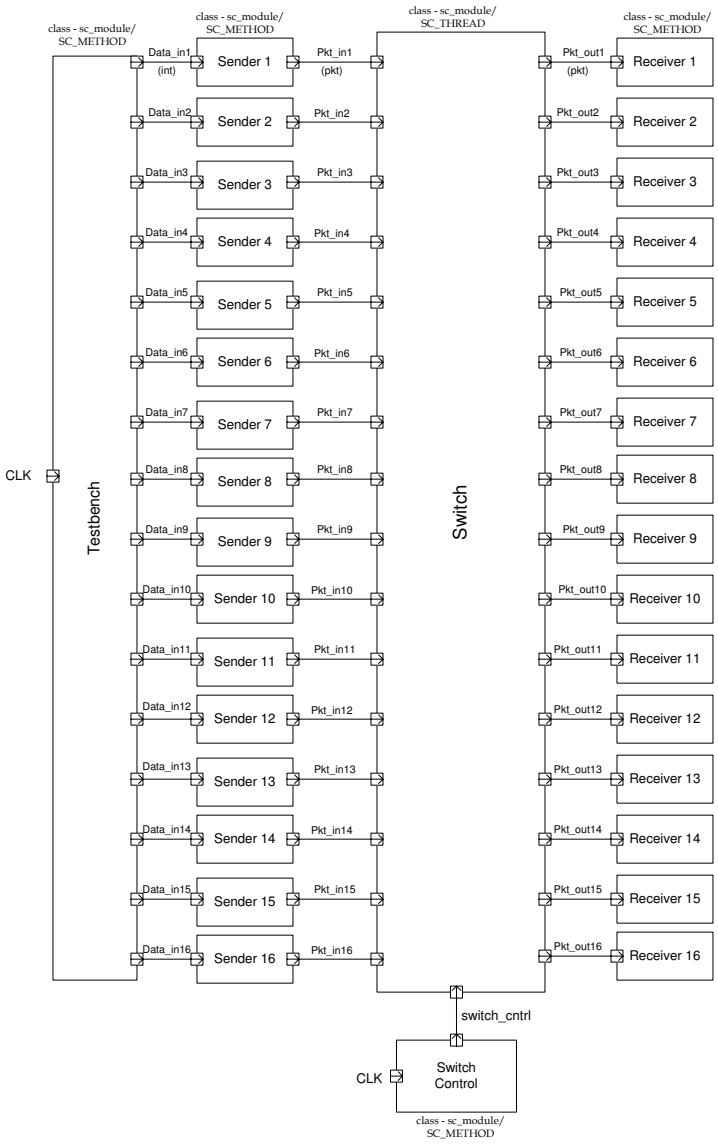

Figure 4.3: SystemC Model - Block Diagram

The switch core has four main tasks: receive the packet, store the packet in the FIFOs, rotate the packet in the ring registers, and deliver the packet to its destination. The senders receive integer number (packtes) from the the genetic algorithm (testbench) and convert them into 16-bit packets and then send them to switch core. The receivers receive the packets and checks if the packets arrive in time. The *switch_cntrl* is an internal clock which is 16 times faster than *CLK*. Figure 4.4 shows the class diagram of the SystemC model for the packet switch.

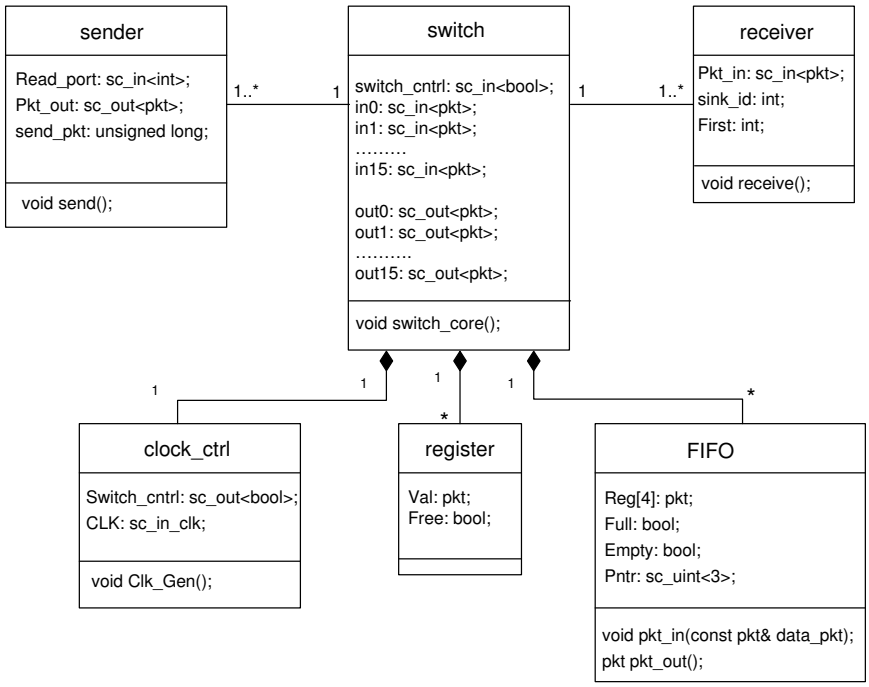

Figure 4.4: SystemC Model - Class Diagram

4.1.3 Verilog Model

The Verilog model of the packet switch is divided into four major parts as shown in Figure 4.5: sender, receiver, FIFO and register modules. The register and FIFO modules are used to make the core of the switch. The sender is used to construct the packet, while the receiver modules are used to display and monitor the received

packet. The Verilog module is designed at RTL.

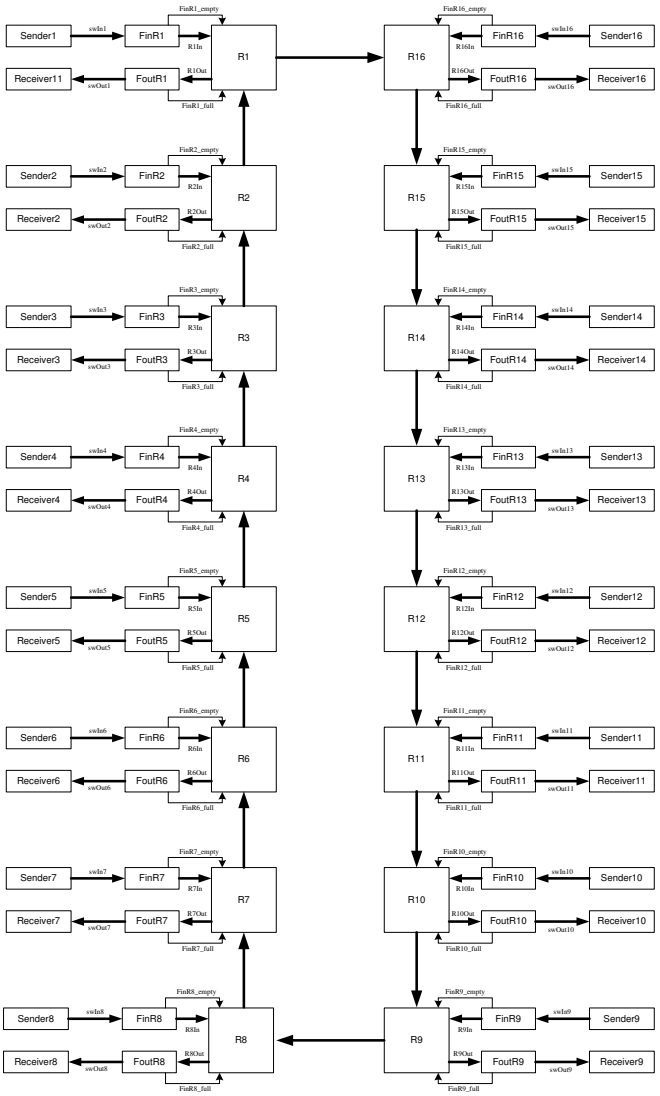

Figure 4.5: Verilog Model - Block Diagram

4.2 Functional Coverage Points

We divided the coverage points into two categories as in [13]: static coverage point, and temporal coverage point. In the *Static* coverage point, the function that needs to be verified is described as a procedural statement and it is evaluated immediately. In the *Temporal* coverage point, the function is time dependent and it is described including the time notation. It is also evaluated at different stages on the time line.

4.2.1 Static Coverage Point

In the packet switch model, static coverage points describe the operation of the external input and external output of each register in the shift ring register. The pseudo-code of static coverage point can be expressed as:

At the Input of a register:
if *(FIFO_In not empty) AND (Register is free)* ***then***
Read the packet from the FIFO
Set the register to not free
Increment No. of hits of coverage point 1
end if

At the output of a register:
if *(Register not free) AND (dest ID equal register ID) AND (FIFO_Out not full)* ***then***
Send the packet to FIFO_Out
Set the register free
Increment No. of hits of coverage point 2
end if

SystemVerilog has a specific syntax to describe static assertion. It defines the

property between two keywords *property* and *endproperty* as follows:

```
property p1;
@ (posedge M_Clk) not $stable(R0In) && (R0In != 16'h0000);
endproperty
```

At every positive edge of the external clock cycle (*@(posedge M_Clk)*), the property *p1* checks if the input to register 1 *(R0In)* is not stable (its current value is different than the previous value), and its value not equal to zero. If this property is detected true then it is covered, otherwise it is not covered.

4.2.2 Temporal Coverage Point

In the packet switch model, we define one temporal coverage point that verifies if the packet arrives at its destination in 16 internal clock cycles if the sender ID is equal to the destination ID. The pseudo-code of Temporal coverage point can be expressed as:

At the input port of a Register
if *(Sender ID = Destination ID))* ***then***
  Increment count1
  Stamp the packet with current time or clock
end if

At the output port of the a Register (input of an Output FIFO):
if *((count1 > count2)* ***then***
  Clock difference = ClockCount - Packet arrival Time
  Increment count2
  if *(Clock difference = 16 Internal clock cycle)* ***then***
    Increment No. of hits of coverage point 1

end if

end if

In the above pseudo-code, when the sender ID is equal to the destination ID of the packet at the input of a register, a counter which is initially equal to 0 is incremented. Then the value of the current time is added to the packet since it is needed to calculate the arrival time at the destination. At the output of the register, if the *count1* is greater than *count2* indicating that sender ID is equal to destination ID, then the arrival time is calculated and *count2* is incremented to become equal to *count1*. Then, if the time difference is equal to 16 internal clock cycles, then the coverage point is hit.

In SystemVerilog, this temporal assertion is described as follows:

```
property p2;
reg [15:0] Reg_R0In;
@(posedge SW_Clk) (R0In[7:4]==R0In[3:0] && R0In[15:8]!=8'h00,
    Reg_R0In=R0In) ##16 (R0Out[7:4]==R0Out[3:0] && R0Out==Reg_R0In);
endproperty
```

The property *p2* is checked at every positive edge of the internal clock cycle (*@(posedge SW_Clk)*). The simulator checks if sender ID is equal to destination ID (*R0In[7:4]==R0In[3:0]*) and the data is not equal to zero (*R0In[15:8]!=8'h00*), then it stores the input value into temporary variable (*Reg_R0In=R0In*). Then, after 16 clock cycles the simulator checks if the output of the register received the same data or not. If this property is detected true, then it is covered, otherwise it is not covered.

4.3 Experimental Results

In this section, we describe the simulation results on the 16 × 16 packet switch written in SystemC and Verilog HDL. The SystemC model is run with CGA and

the Verilog model run under a VCS environment. Also, we will show the effects of different distribution on the coverage rate.

4.3.1 Description of Experiments

We implemented the CGA with different probability distribution functions in C++. The coverage points are expressed in SystemC. The program is run in Microsoft Visual C++ 6.0 with SystemC 2.0.1 under Windows XP SP2 operating system on AMD Turion 64x2 processor of 1.6GHz, and with 1GB of memory. The RNGs based on Uniform distribution (*Uniform*), Exponential distribution (*(Exponential)*), Beta distribution (*Beta*(α, β)), Gamma distribution (*Gamma(K,*θ*)*), Normal distribution (*Normal*(μ, σ)) and Triangle distribution (*Triangle(a,c,b)*) are coded as separate C++ classes and are integrated into the CGA. The parameters of Normal and Triangle distributions are selected within the range of 1 (*2^0*) to 65535 (*2^{16}-1*). For Normal distribution, (*Normal*(μ, σ)), we selected three values for the mean (μ) while we kept the standard deviation (σ) constant. In a similar way, we choose a set of values for the three parameters of Triangle distribution, *Triangle(a,c,b)*. For *Beta*(α, β) and *Gamma(K,*θ*)*, we selected three sets of values for each based on their effect on the distribution.

The Verilog model of the 16×16 packet switch is implemented in a synthesisable Verilog HDL of the same specification as the SystemC model, and the Verilog model is run using the Synopsys VCS tool under Solaris 9.0 operating system on Sun Ultra SPARC-IIi processor of 650MHz, with 512MB memory. Only three RNGs based on uniform distribution *$dist_uniform(seed, a, b)*, normal distribution *$dist_normal(seed, mean, std)* and exponential distribution *$dist_exponential(seed, mean)* were used due to lack of support for Beta, Gamma, and Triangle distribution in VCS.

In following, we describe several experiments where SystemC model of the 16×16 packet switch is simulated with CGA and the implemented five distributions

RNGs while the Verilog model is simulated with RNGs provided by Synopsys VCS libraries.

CGA has several parameters that are predefined and their values are set up by the user. Some of those parameters were kept unchanged from their default values and some were changed due to their expected effect on the coverage. Some of the parameters and their values are listed in Table 4.1.

Parameter	**value**
Population Size	100
Number of Generations	100
Number of Cell	25
Number of Simulation Cycles	50
Clock Tic	5
Tournament Size	5
Crossover Probability	95
Mutation Probability	20
Elitism Probability	3

Table 4.1: GA Parameters

4.3.2 Experiment 1: 32 Static Coverage Points

In this experiment, we study the effect of different probability distributions on the coverage rate using CGA as an optimization and search program with the 16×16 packet switch modeled in SystemC as DUV. The coverage tasks are defined in the SystemC model as 32 static coverage points. The function of the coverage tasks is to detect the external input and external output of each register in the ring of shift registers. When the register receives the packet from an *Input_FIFO*, the input coverage task is triggered and when the register sends the packet to the *Output_FIFO*, the output coverage task is triggered. In SystemC, these static coverage tasks are written as follows:

```
//At the input of a register
    if((!q0_in.empty)&&(R0.free))
        }
            R0.val = q0_in.pkt_out();
            R0.free = false;
            InCovTask17++;
        }

    //At the output of a register
        if((!R0.free)&&(R0.val.did == 0)&&(!q0_out.full))
        }
            q0_out.pkt_in(R0.val);
            R0.free = true;
            OutCovTask1++;
        }
```

We start the simulation by selecting the probability distribution RNG and its parameters which remains constant during the simulation. We run the simulation for 100 generations where *PopulationSize=100* and maximum number of cells is set to *NoOfCell=25*. The result of the simulation is summarized in Table 4.2. The first column shows the probability distributions and their parameters. The second column shows the values of the maximum fitness occurring during the simulation, while the third column shows the generation number where maximum fitness occurred. We run the simulation several times for each distribution. We obtain similar results each time with slight variation due to the random nature of the CGA. The fitness value reflects the coverage rate; if the fitness value exceeds 1.6, then all coverage points are hit at least once and 100% coverage is achieved. The maximum fitness

refers to the maximum value total average hits of all coverage points when all coverage points must be hit. The result shows 100% coverage for all distributions. It is noted that with some distributions, the maximum fitness is reached at an earlier generation than others such as in the case of Exponential distribution (Figure 4.6).

Probability Distribution RNGs	**Max. Fitness**	**Generation No. at Max. Fitness**	**CPU Time at Max. Fitness**
Uniform (MT)	2.06	98	315.12s
Exponential	1.98	27	86.8s
Beta (2-2)	1.87	87	289.14s
Beta(5-10)	1.77	77	254.57s
Beta (10-2)	2.055	82	279.82s
Gamma (2-2)	1.998	70	229.68s
Gamma (2-3)	1.976	51	166.73s
Gamma (9-11)	2.075	76	244.62s
Normal (10000-2000)	2.04	52	167.54s
Normal (30000-2000)	2.017	83	264.56s
Normal (50000-2000)	2.005	82	261.1s
Triangle (0-10000-65535)	2.254	58	188.4s
Triangle (0-30000-65535)	2.075	73	231.31s
Triangle (0-50000-65535)	1.996	98	308.7s
Triangle (0-15000-30000)	2.062	60	188.62s
Triangle (30000-45000-65535)	2.075	62	249.23s

Table 4.2: Results of 32 Static Coverage Points

The same experiment is run with different values for population size (*PopulationSize=20*) and number of cells (*NoOfCell=5*). In this simulation we obtain almost similar values to the previous simulation: the coverage rate is 100% except for Gamma(9-11) where we get 96% coverage because we could not cover one coverage point. The fitness values are lower than the values in the previous results due to the lower value of number of cells.

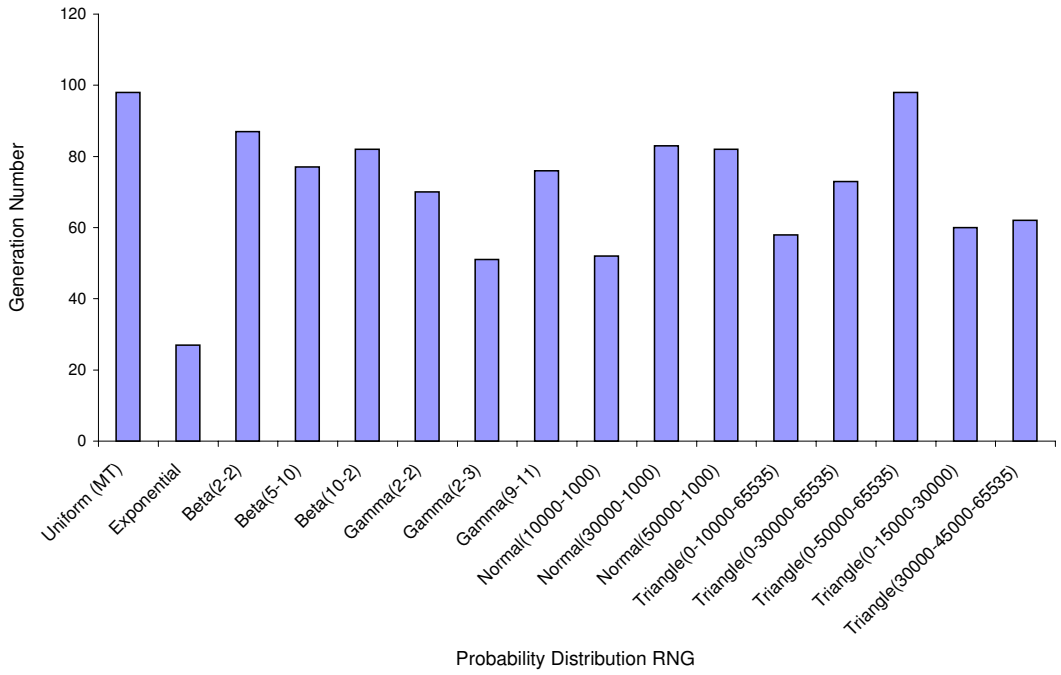

Figure 4.6: Maximum Fitness of 32 Static Coverage Points

4.3.3 Experiment 2: 16 Static Coverage Points

In this experiment, we repeat Experiment 1 with 16 static coverage points to detect only the external inputs of each register. In addition, we select only one set of parameters for each distribution. The objective is to study the effect of reducing the number of coverage points on generation number where the fitness value reaches its maximum. Table 4.3 summarizes the results while Figure 4.7 compares the results with the results of Experiment 1.

The results show that maximum fitness values are obtained at a lower number of generation when the number of coverage points is decreased in general. In Experiment 2 each generated packet must hit one coverage point, while in Experiment 1 it must hit 2 coverage points; this will help to achieve maximum coverage in a short time for all coverage points.

Probability Distribution RNGs	**Maximum Fitness**	**Generation No. at Max. Fitness**	**CPU Time at Max. Fitness**
Uniform (MT)	2.20859	38	56.406s
Exponential	2.1005	9	14.343s
Beta (10-2)	1.73994	18	29.844s
Gamma (2-2)	2.318	57	81.298s
Normal (30000-1000)	2.20859	17	25.156s
Triangle (0-30000-65535)	2.23994	33	47.39s

Table 4.3: Results of 16 Static Coverage Points

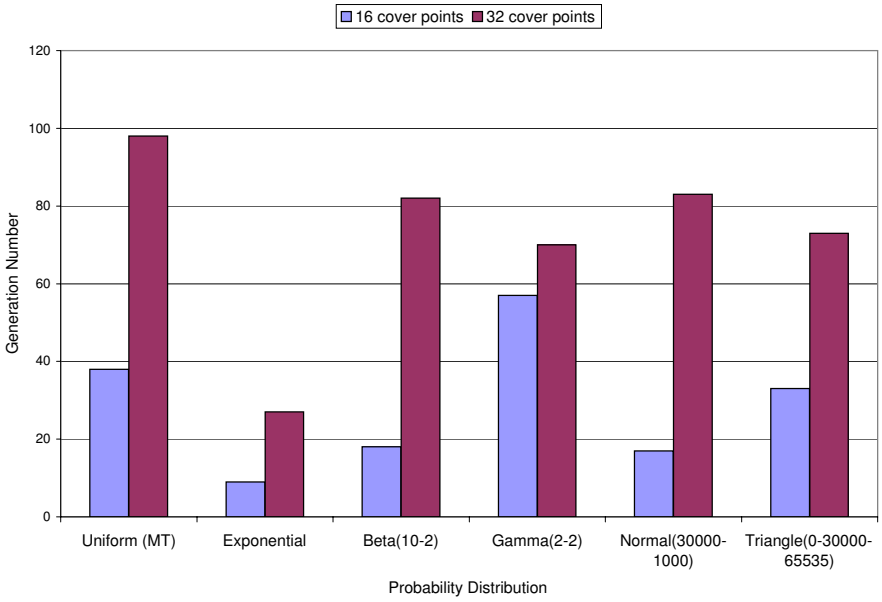

Figure 4.7: Maximum Fitness: 32 vs. 16 Static Coverage

4.3.4 Experiment 3: Group Coverage Points

In this experiment, we repeat Experiment 1 but on a group of coverage points in which we put every four coverage points into one group. Each group is seen by the algorithm as one coverage point. For example, we select the first four coverage points as one group, and if any of those coverage points is hit then the coverage group is also hit. Thus, 32 coverage points are divided into eight coverage groups as listed in Table 4.4. In addition, we select only one set of parameters for each distribution. The objective is to study the effect of coverage group on generation number where the fitness value reaches its maximum. In this simulation, the Genetic Algorithm recognizes only eight coverage groups and tries to optimize for eight coverage points instead of 32.

Coverage Group	**Coverage points**
Group 1	point 1 - point 4
Group 2	point 5 - point 8
Group 3	point 9 - point 12
Group 4	point 13 - point 16
Group 5	point 17 - point 20
Group 6	point 21 - point 24
Group 7	point 25 - point 28
Group 8	point 29 - point 32

Table 4.4: Coverage Groups

Probability Distribution	**Max. Fitness**	**Gen. No. at Max. Fit.**	**CPU Time at Max. Fit.**
Uniform (MT)	11.744	92	237.4s
Exponential	11.0868	54	167.5s
Beta (10-2)	9.585	88	279.3s
Gamma (2-2)	11.085	96	317.2s
Normal (30000-2000)	11.244	75	226.5s
Triangle (0-30000-65535)	11.574	69	205.9s

Table 4.5: Results of 8 Static Coverage Groups

The results in Table 4.5 show an increase in the fitness value above 9.0 due to the increase in the number of hits for each group. For example, if any of the coverage points in the coverage group is hit, then the coverage group is hit. But, this does not indicate that all coverage points of the group are hit. Figure 4.8 shows the generation number where maximum fitness is obtained for Experiment 1 and Experiment 3. It is noted that the maximum fitness reached in Experiment 1 is closer to that for the number of generations in Experiment 3 except for Exponential and Gamma distributions. But in case of the Exponential distribution, the maximum fitness is achieved at a lower number of generations than other distributions.

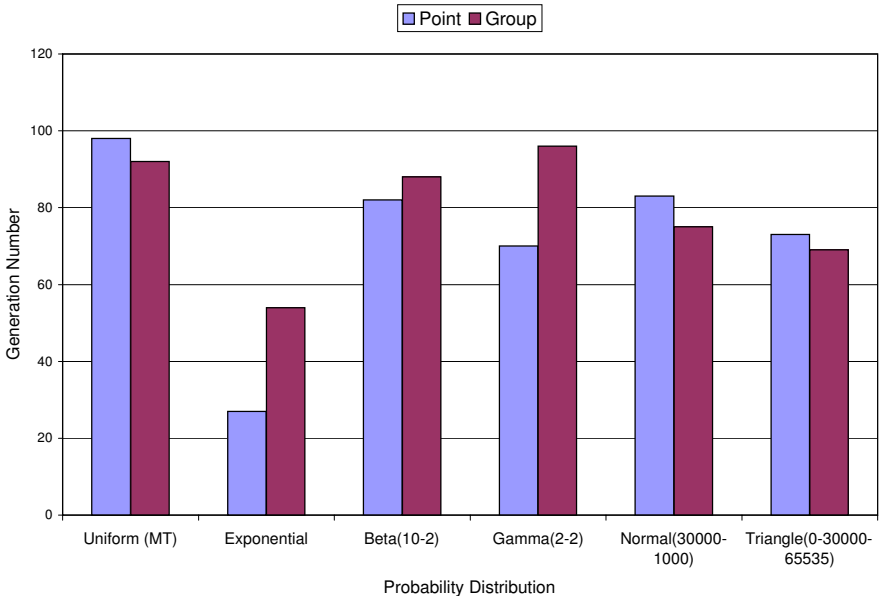

Figure 4.8: Maximum Fitness: Points vs. Group Static Coverage

4.3.5 Experiment 4: 16 Temporal Coverage Points

In this experiment, we study the effect of different probability distributions on 16 temporal coverage points. The function of the coverage task is to detect if the packet arrives at its destination in 16 internal clock cycles when the packet sender ID is

equal to its destination ID. Since there are 16 ports, 16 coverage points are defined. The C code of the temporal coverage task can be written as follows:

```
//At the input of a register
    if((!q0_in.empty)&&(R0.free)){
        R0.val = q0_in.pkt_out();
        if(R0.val.sid == R0.val.did){
            hit01++;
            TotalHit++;
            R0.val.pkttime = SWClkCount;
        }
    R0.free = false
    }

//At the output of a register
  if((!q0_out.empty){
        outpkt0 = q0_out.pkt_out();
        if(hit01 > hit02){
            hit02++;
            tt = SWClkCount − outpkt0.pkttime;
            if(tt == 16){
                cov1++;
                TotalCov++;
            }
            else;
                TotalNotCov++;
        }
        out0.write(outpkt0);
```

}

In this experiment, we use another formula to calculate the fitness where 100% coverage is reached if the fitness value exceeds 995. The results of the simulation is summarized in Table 4.6.

Probability Distribution RNGs	**Max. Fitness**	**Gen. No. at Max. Fit.**	**Coverage Rate**	**CPU Time at Max. Fit.**
Uniform (MT)	499.48	31	50	94.62s
Exponential	561.98	3	56	12.22s
Beta(2-2)	561.98	47	56	145.4s
Beta (5-10)	561.88	41	56	127.42s
Beta (10-2)	561.88	33	56	104.2s
Gamma (2-2)	561.98	23	56	75.25s
Gamma(2-3)	499.48	61	50	204.1s
Gamma(9-11)	499.37	51	50	167.42s
Normal (10000-2000)	561.98	77	56	228.95s
Normal (30000-2000)	561.88	14	56	44.2s
Normal (50000-2000)	561.98	16	56	47.2s
Triangle (0-10000-65535)	499.48	58	50	171.64s
Triangle (0-30000-65535)	499.48	76	50	223.72s
Triangle (0-50000-65535)	499.48	37	50	108.92s
Triangle (0-15000-30000)	499.48	23	50	68.35s
Triangle (30000-45000-65535)	499.48	27	50	79.73s

Table 4.6: Results of 16 Temporal Coverage Points

The results show almost 50% coverage for all distributions even when the simulation ran for more than 100 generations. The main reason is due to the fact that the coverage point is only hit when the sender ID is equal to the destination ID when the data arrives at its destination in 16 internal clock cycles. The results that the CGA produce are based on the best population result of the entire generation. In other words, if a single generation produces 50 populations and one population produce best test vectors that gives highest coverage in the generation, then that

population is reported as best population even thought it does not give 100% coverage. It is observed that if we look at the entire generation, we find that all points are covered in different populations.

4.3.6 Experiment 5: 32 Static Assertion (SVA)

Besides SystemC experiments, we simulated the Verilog model of a 16×16 packet switch with static assertions using three different random number generators. 32 coverage points are defined and coded using SystemVerilog Assertion. The three RNGs are based on Uniform distribution *$dist_uniform(seed, a, b)*, Normal distribution *$dist_normal(seed, mean, std)* and Exponential distribution *$dist_exponential(seed, mean)* that are supported by Verilog HDL. The simulation ran for 200 thousand cycles using a VCS tool, and 1300 packets were generated and sent to the packet switch.

The objective of this study was the same as in Experiment 1, namely to study the effect of probability distribution on the coverage, and to compare it with the results in Experiment 1.

The results are summarized in Table 4.7. The seed value is set to 0, and we experimented with the random value of the seed, which ranges randomly from 0 to 8 (*SEED=urandom*()%8). We obtained 100% coverage (here, coverage in terms of the simulator semantics, which refers to hitting every cover point at least once) of static assertion properties for all distributions after running for at least 200 thousand cycles. A similar result is obtained with random seed, but with less hits for some assertion points. The Uniform and Exponential distributions using random seeds show less coverage at certain parameters as displayed in Table 4.7. The high coverage with static assertions is due to the fact that each packet must hit two assertion points.

Probability Distribution	Coverage Rate
Uniform (0,0,65535)	100
Uniform (0,0,32500)	100
Exponential (0,5000)	100
Exponential (0,500)	100
Normal (0,10000,2000)	100
Normal (0,30000,2000)	100
Normal (0,50000,2000)	100
Uniform (rand(0-8),0,65535)	53
Exponential (rand(0-8),5000)	96
Normal (rand(0-8),30000,2000)	100

Table 4.7: Results of 32 Static Assertions (SystemVerilog)

4.3.7 Experiment 6: 16 Temporal Assertion (SVA)

In this experiment, we use the Verilog model of the 16×16 packet switch with temporal assertion to simulate with three different distributions RNGs. 16 temporal properties are defined and three random number generators were used and they are based on Uniform distribution *$dist_uniform(seed, a, b)*, Normal distribution *$dist_normal(seed, mean, std)* and Exponential distribution *$dist_exponential(seed, mean)*. The simulation run for 200 thousand cycles using VCS, where 1300 packet were sent.

The results are summarized in Table 4.8, which seem to be similar to the results of Experiment 5.

4.3.8 Experiment 7: Coverage-base Verification

In this experiment, we simulate the 16×16 packet switch Verilog model with coverage points enabled using the concept of coverage-based verification where the property that needs to be verified is expressed as coverage point or coverage group. Similarly, we use the same three random number generators as in Experiment 6. In addition, 16 coverage groups were defined where each group has two coverage points. The

Probability Distribution	coverage rate
Uniform (0,0,65535)	100
Uniform (0,0,32500)	100
Exponential (0,5000)	100
Exponential (0,500)	93
Normal (0,10000,2000)	100
Normal (0,30000,2000)	100
Normal (0,50000,2000)	100
Uniform (rand(0-8),0,65535)	0
Normal (rand(0-8),30000,2000)	37
Exponential (rand(0-8),5000)	6

Table 4.8: 16 Temporal Assertions (SystemVerilog)

function of each coverage group is to detect at each positive edge of external clock cycle if the sender ID is equal to the destination ID. We define the coverage points as follows:

```
covergroup CovGp1 @(posedge Clock);
        coverpoint R0In_SID
        { wildcard bins A = {4'b0001}; }
        coverpoint R0In_DID
        { wildcard bins B = {4'b0001}; }
        corss R0In_SID , R0In_DID;
endgroup
```

Table 4.9 summarizes the simulation results. It is noted from the results that when the simulation runs for 100 thousand clock cycles, we achieve a lower coverage rate, but if the simulation runs for 300 thousand cycles, we achieve 100% coverage. Moreover, with Uniform and Normal distributions we achieve a higher coverage than Exponential distribution with less simulation cycles.

Probability Distribution	Coverage rate at different clock cycles		
	100 000	150 000	300 000
Uniform(0,0, 65535)	93.75	100	100
Exponential(0,5000)	50	75	100
Normal(0,30000,2000)	81.25	100	100

Table 4.9: Results of Coverage Groups

4.3.9 SystemC vs. Verilog

Table 4.10 compares the results of SystemC and Verilog simulations with the RNG of constant seed *(SEED=0)* and of random seed *(SEED=0-100)*. In case of random seed for Verilog model, we the experiments until we achieve 100% in order to compare with SystemC result. In case of Exponential and Normal distribution we achieve 100%, while in case of Uniform distribution we only achieve 43% even though we run the experiment for long time. In the case of SystemC, the coverage increased to a certain value and did not pass that value. This issue is more related to the DUV. The coverage result reported by the CGA is the best result obtained in a population in the entire generation and not as overall coverage of the entire generation. For example, the Exponential distribution at third (3^{rd}) generation reported 56% coverage but the overall coverage of the entire (3^{rd}) generation was 100%. The overall coverage is reported indirectly and needs to be calculated based on the number of hits for each coverage point. The overall coverage for SystemC simulation with CGA is calculated and reported in the last column of Table 4.10. Since the nature of the CGA (SystemC) simulation is similar to the nature of VCS (SystemVerilog), the results should provide a fair comparison. Therefore, we can obtain 100% coverage using SystemC and CGA in a shorter time than SystemVerilog if proper distribution (Exponential) is used.

Probability Distribution RNGs	Coverage Rate - CPU time in Sec.		
	SystemVerilog		SystemC
	Constant Seed	Random Seed	CGA (Gen. No.)
Uniform	100 - 3.83s	43 - 104.5s	100 (31) - 94.62s
Exponential	100 - 3.81s	100 - 17.8s	100 (3) - 12.22s
Normal	100 - 3.95s	100 - 34.9s	100 (14) - 44.2s

Table 4.10: SystemVerilog and SystemC Comparison (Temporal Assertions)

4.4 Discussion

We run several experiments using different parameters of RNG and CGA. The results show that not all distributions produce the same result. Maximum fitness or maximum coverage is achieved in shorter time or with a smaller number of generations for some probability distributions such as exponential distribution. This finding can be used to set up a termination criteria for the simulation. Also, the maximum coverage is reached in shorter time if the number of coverage points is decreased or grouped.

On the other hand, we found that the results were not consistent for all SystemC simulations due to the random nature of the CGA. It might be useful to run the simulation for each probability distribution RNG several times or even many more times and then apply statistical methods to determine more accurate effects of the probability distribution on the coverage. Also, we found that the coverage did not improve after some generations, and it fluctuates around a certain value. This result is due to the nature of the packet switch design and the structure of the generated packet.

It is important to note that considering the relatively small size of the design, the simulation time difference between the SystemC and Verilog implementations was pretty minor. This explains why both simulations were able to reach high coverage numbers in relatively comparable execution times. We do believe, however, that for large designs the SystemC simulation at higher abstraction level would

run much faster than the VCS simulation at RTL (hundred of cycles per second). Therefore, it should be appropriated to optimize coverage at the higher level (e.g., TLM–Transaction Level Modeling) and reuse the tests for RTL (certainly, considering some minor modifications for the tests).

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we presented a new approach of using Genetic Algorithms and random probability distributions in order to improve coverage for hardware design. We implemented, tested, and integrated random number generators based on *Exponential*, *Normal*, *Gamma*, *Beta* and *Triangle* probability distributions with cell-based genetic algorithm to automate the coverage directed test generation. We applied our approach to SystemC and Verilog models of a 16×16 Packet Switch in which we defined static and temporal coverage points.

It was found that each probability distribution has an effect on the coverage; where in some distributions the coverage reaches its maximum within a small number of generations, while with others it reaches it after several generations. The experiments show a consistency with the results for some probability distributions when the experiments were repeated such as Exponential distribution, while in the results of other distributions show slight differences due to the random nature of the Genetic Algorithm.

It might be useful to run the simulation for each probability distribution RNG

several times or even many more times and then apply statistical methods to determine more accurate effects of the probability distribution on the coverage. Moreover, we found that the coverage did not improve after some generations and fluctuates around a certain value. This result is due to the nature of the packet switch design and the structure of the generated packet.

We implemented 16×16 packet switch written in Verilog HDL and simulated with with three different random number generator based on Uniform, Exponential, and Normal probability distribution and studied their effect with constant seed and random seed and compared it with SystemC simulation results.

It was found that the simulation time difference between the SystemC and Verilog implementations was pretty minor due to the relatively small size of the design. This explains why both simulations were able to reach high coverage numbers in a relatively comparable execution times. We do believe, however, that for large designs the SystemC simulation at higher abstraction level would run much faster than the VCS simulation at RTL. Therefore, it should be appropriated to optimize coverage at the higher level (e.g., Transaction Level Model - TLM) and reuse the tests for RTL (certainly, considering some minor modifications for the tests).

5.2 Future Work

The RNG presented in this thesis generates acceptable, accurate results but further enhancement can be made by using more accurate and complex algorithms. RNG should generate integer numbers within a range of [0, $2^{32} - 1$]. Some RNGs based on Exponential, Gamma and Beta distribution that we implemented in this thesis generate numbers within the range of [0, 1] and then a scaling factor was used to scale the numbers in the range of [0, 9999]. We believe that different algorithms can be used to generate random numbers based on Exponential, Gamma and Beta distributions.

In addition, as a future work, from the practical point of view, we think it is valuable to apply our approach to complex designs where regular simulators fail to hit specific coverage points in a pure random execution mode. For example, we can target our approach to explore designs such as a 32-bit microprocessor or an IP Router.

Cell-based Genetic Algorithm plays the main role in our approach, therefore, it is essential, from the theoretical point of view, to investigate several aspects of the algorithm. For instance, we would like to explore any possible link between the implementation of CGA and the best random distribution to use (or the parameters of the random distribution). Another algorithm could be used to measure the progress of the coverage based on random distributions or their parameters.

Another aspect of the algorithm that can be investigated is a sequence of inputs over time rather than a static randomization over time. CGA generates a series of inputs to the DUV, but the sequence of the inputs is not observed and optimized by the CGA. Observing and optimizing sequences of inputs adds another powerful feature to the CGA to target certain functions in a design.

Finally, we believe this thesis is an important milestone towards building a complete environment for automatic coverage enhancing methodology. Therefore, it is important to develop algorithms besides the GA to fully automate the verification cycle taking into consideration the nature of the DUV.

Bibliography

[1] IEEE Standard Association. IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. www.ieee.org, 2005.

[2] IEEE Standard Association. IEEE Standard for the Functional Verification Language e. www.ieee.org, 2006.

[3] IEEE Standard Association. IEEE Standard SystemC Language Reference Manual. www.ieee.org, 2006.

[4] Semiconductor Industry Association. Global Chip Sales Hit 255.6 Billion in 2007. www.sia-online.org/pre_release.cfm?ID=464, 2008.

[5] D. C. Black and J. Donovan. *SystemC: from the Ground Up.* Springer, 2004.

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

[7] Intel Corporation. Moores Law Timeline. http://download.intel.com/pressroom/kits/ events/moores_law_40th/MLTimeline.pdf, 2008.

[8] A. Eliens. hush-4.0, Department of Mathematics and Computer Science, Vrije University, Amsterdam, The Netherlands. http://www.cs.vu.nl/pub/eliens/archive/hush-4.0.tar.gz, 2007.

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

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

[11] H. D. Foster, A. C. Krolnik, and D. J. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, 2004.

[12] M. Glasser, A. Rose, T. Fitzpatrick, D. Rich, and H. Foster. *Advance Verification Methodology Cookbook*. Mentor Graphics Corporation, 2007.

[13] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User Define Coverage - A Tool Supported Methodology for Design Verification. In *Proc. of Design Automation Conference*, pages 158–163, San Francisco, California, USA, 1998.

[14] O. Guzey and L. Wang. Coverage Directed Test Generation Through Automatic Constraint Extraction. In *Proc. of the IEEE International High Level Design Validation and Test Workshop*, pages 151–158, Irvin, California, USA, 2007.

[15] H. Hsueh and K. Eder. Test Directive Generation for Functional Coverage Closure Using Inductive Logic Programming. In *Proc. of IEEE International High-Level Design Validation and Test Workshop*, pages 11–18, Monteray, California, 2006.

[16] MIPS Technologies Inc. MIPS32 Instruction Set Quick Reference. www.mips.com, 2007.

[17] Synopsys Inc. Synopsys Verilog Compiler Simulator. http://www.synopsys.com, 2008.

[18] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesely, 1998.

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

[20] W. K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005.

[21] D. T. Lang. Approaches for Random Number Generation. Class notes of Statistical Computing: STAT 141, http://eeyore.ucdavis.edu/stat141/Notes/RNG.pdf, University of California at Davis, USA, 2006.

[22] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistance Uniform Pseudo-Random Number Generator. *ACM Transaction on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[23] A. Meyer. *Principles of Functional Verification*. Elsvier Science, 2004.

[24] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996.

[25] P. Mishra and N. Dutt. Functional Coverage Driven Test Generation for Validation of Pipelined Processors. In *Proc. of the Design Automation and Test in Europe*, volume 2, pages 678–683, Munich, Germany, 2005.

[26] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT press, 1999.

[27] Open SystemC Initiative OSCI. www.SystemC.org, 2006.

[28] C. Pixley, A. Chittor, F. Meyer, S. McMaster, and D. Benua. Funtional Verification 2003: Technology, Tools and Methodology. In *Proc. 5th International Conference on ASIC*, volume 1, pages 1–5, Beigin, China, 2003.

[29] A. Piziali. *Functional Verification Coverage Measurement and Analysis*. Kluwer Academic Publishers, 2004.

[30] W. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[31] The Register. Consumer Electronics Fuel Chip Sales. http://www.theregister.co.uk/2007/11/16/electronics_boosts_chip_sales, 2008.

[32] F. Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Springer, 2006.

[33] R. Roy. Comparison of Different Techniques to Generate Normal Random Variables. Technical report, The State University of New Jersey, USA, 2004.

[34] A. Samara, A. Habibi, S. Tahar, and N. Kharma. Automated Coverage Directed Test Generation Using Cell-Based Genetic Algorithm. In *Proc. of IEEE International High Level Design Validation and Test Workshop*, pages 19–26, Monterey, California, USA, 2006.

[35] H. Shen and Y. Fu. Priority Directed Test Geneation for Functional Verification using Neural Networks. In *Proc. of the Conference on Design Automation - Asia South Pacific*, volume 2, pages 1052–1055, Shanghai, China, 2005.

[36] C. Spear. *SystemVerilog for Verification A Guide to Learning the Testbench Language Features*. Springer, 2007.

[37] Specman Elite. Specman Elite Tutorial 4.3.6. www.verisity.com, 2006.

[38] M. Spiegel, J. Schiller, and R. Srinivasan. *Theory and Problems of Probability and Statistics*. McGraw-Hill, 2000.

[39] G. S. Spirakis. Opportunities and Challenges in Building Silicon Products in 65nm and Beyond, keynote speech. In *Design Automation and Test in Europe*, pages 2–3, Paris, France, 2004.

[40] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer. A Funtional Validation Technique: Biased-Random Simulation Guided by Observabilty-Based Coverage. In *Proc. International Conference on Computer Design*, pages 82–88, Los Alamitos, California, USA, 2001.

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

[42] A. H. Warren. Introduction: Special Issue on Microprocessor Verifications. *Formal Methods in System Design*, 20:135–137, 2002.

[43] P. Wilcox. *A guide to Advance Funtional Verification*. Springer, 2004.

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

[45] R. Yang, L. Wu, J. Guo, and B. Liu. The Research and Implement of an Advanced Function Coverage Based Verification Environment. In *Proc. of 7th International Conference on ASIC*, pages 1253–1256, Guilin, China, 2007.

[46] X. Yu, A. Fin, F. Fummi, and E. Rudnick. A Genetic Testing Framework for Digital Integrated Circuit. In *Proc. of the 14th IEEE International Conference on Tools with Artificial Intelligence*, pages 521–526, Washington, DC, USA, 2002.