2006 IEEE International High Level Design and Test Workshop

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

Amer Samarah, Ali Habibi, Sofiène Tahar, and Nawwaf Kharma
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montreal, Quebec H3G 1M8
Emails:{amer_sam, habibi, tahar, kharma}@ece.concordia.ca

***Abstract*— Functional verification is a major challenge in the hardware design development cycle. Defining the appropriate coverage points that capture the design's functionalities is a non-trivial problem. However, the real bottleneck remains in generating the suitable testbenches that activate those coverage points adequately. In this paper, we propose an approach to enhance the coverage rate of multiple coverage points through the automatic generation of appropriate test patterns. We employ a directed random simulation, where directives are continuously updated until achieving acceptable coverage rates for all coverage points. We propose to model the solution of the test generation problem as sequences of directives or cells, each of them with specific width, height and distribution. Our approach is based on a genetic algorithm, which automatically optimizes the widths, heights and distributions of these cells over the whole input domain with the aim of enhancing the effectiveness of test generation. We illustrate the efficiency of our approach on a set of designs modeled in SystemC.**

I. Introduction

Due to the increasing complexity of hardware design and the never ending pressure of shorter time-to-market, functional verification has become a major challenge in the design development cycle. With unceasing growth in system functionality, the challenge today concerns, in particular, verifying that the logic design obeys the intended functional specification, and that it performs the tasks required by the overall system architecture [16].

Several methodologies have been developed lately in order to tackle the functional verification problem: simulation based verification, assertion based verification and coverage based verification [16]. In simulation based verification, a dedicated test bench is built to functionally verify the design by providing meaningful scenarios. On the other hand, assertion based verification is used to catch errors on the spot, where assertions are written either in a hardware description language or a specialized assertion language (e.g., Property specification Language (PSL) [1] or System Verilog Assertion (SVA) [5]).

The concept of coverage based verification requires the definition of coverage metrics which are used to assess the progress of the verification cycle and to identify functionalities of the design that have not been tested. The most widely used metrics are: code coverage, finite state machine (FSM) coverage and functional coverage. Code coverage evaluates the degree to which the structure of the hardware description language source code has been exercised, while the FSM coverage provides more clues about the functionality of the system. The main problem with FSM coverage is that the generation of large FSMs leads to combinatorial explosion (state explosion problem). In functional coverage, a set of points represent the important behavior and specification of the design are investigated. Accordingly, the coverage could be the number of activations of these points [16].

Random test generators are commonly used for exploring unexercised areas of the design. Coverage tools are used side by side with random test generator in order to assess the progress of the test plans during the verification cycle. The coverage analysis allows for (1) the modification of the directives for the test generators; and (2) the targeting of areas of the design that are not covered well [6]. This process of modifying the directives for the test generator according to feedback based on coverage reports (called Coverage Directed test Generation (CDG)) is a manual and exhausting process, but essential for the completion of the verification cycle. Considerable effort is being invested in finding ways to close the loop connecting coverage analysis to adaptive generation of test directives.

In this paper, we propose an approach for automatic CDG. We aim at (1) constructing efficient test generators for checking the important behavior and specification of the Design Under Test (DUT); (2) finding common directives that activate multiple coverage points; (3) improving the coverage progress rate; and (4) designing directives that can reach uncovered tasks (coverage points). By achieving these goals, we increase the efficiency and quality of the verification process and reduce the time and effort needed to implement a test plan.

Our final objective is to increase the coverage of multiple coverage tasks at the same time. Hence, we propose an algorithm that is capable of targeting complex coverage tasks and groups of correlated coverage tasks while achieving adequate coverage rates. This is done by moving from a blind random test generation over the inputs domains to an optimized generation in relation to the coverage points. We split the input domains into sub ranges each having a specific weight (that represents the probability of generating values from that range). We refer to these ranges as cells. Our proposed algorithm (called Cell-based Genetic Algorithm (CGA)) automatically optimizes the widths, heights and distributions of these cells over the whole domain with the aim

1-4244-0679-X/06/$20.00 ©2006 IEEE

of enhancing the effectiveness of the tests generation process for the considered coverage group.

Our algorithm inherits the advantages of genetic algorithms, which are techniques inspired by evolutionary biology [11]. The evolution starts from a population of completely random abstract potential solutions and continues over generations. In each generation, all individual members of the population are evaluated for fitness using a fitness evaluation function or methods; multiple individuals are stochastically selected from the current population based on their fitness values and are then possibly modified by genetic operators to form a new population for further evolution. The process of evaluation, selection and diversification iterates until a termination criterion is satisfied.

In order to evaluate our algorithm, we consider two designs modeled in SystemC [12]. This latter is the standard for system level design. It is a library and modeling platform built on top of C++ to represent functionality, communications, and software and hardware implementation at various levels of abstraction. The main advantage of SystemC is a faster simulation speed that enables faster convergence of the genetic algorithm. Experimental results illustrate the effectiveness of the proposed algorithm in achieving the goals of CDG.

The rest of this paper is as follows. Section II reviews related work. Section III describes our methodology of integrating the CGA within the design flow. Section IV presents the details of the proposed genetic algorithm. Section V illustrates the experimental results. Finally, Section VI concludes the paper.

II. Related Work

Considerable effort has been expended in the area of functional verification. Several approaches based on simple genetic algorithm, machine learning, and Markov chain modeling have been developed. For instance, the work in [8] uses a simple genetic algorithm to improve the coverage rates for SystemC RTL models. This work tackles a single coverage point at a time, and hence is unable to achieve high coverage rates in the case of complex coverage point. In contrast, our methodology, which is based on multiple sub-range directives, is able to handle multiple coverage points simultaneously. In [4] Bayesian networks are used to model the relation between coverage space and test generator directives. This algorithm uses training data during the learning phase, and the quality of this data affects the ability of the Bayesian network to encode knowledge correctly. In contrast, our algorithm starts with totally random data, where the quality of initial values affects only the speed of learning and not the quality of the encoded knowledge. [15] proposes the use of a Markov chains to model the DUT, while coverage analysis data is used to modify the parameters of the Markov chain which is then used to generate test cases for the design.

Genetic algorithms have been used for many verification and coverage problems. For example, [7] addresses the exploration of large state spaces. This work is based on BDDs (Binary Decision Diagrams) and is restricted to simple Boolean assertions. In [3], genetic programming is used to develop automatic test programs and instruction sequences for microprocessor cores. Besides functional verification, genetic algorithms have been used for Automatic Test Pattern Generation (ATPG) problems in order to improve the detection of manufacturing defects [10].

III. Methodology of Automatic Coverage Directed Test Generation

We propose a verification technique that aims at automatically directing the test generation based on functional coverage metric. Figure 1 illustrates the overall CDG approach, where designers provide the verification team with the design under verification and its functional specification. This latter is given in the form of functional coverage points that represent important behaviors and specifications of the design. After evaluating the coverage reports, the verification engineers continually write and update directives and constraints for the random test generator to activate the coverage points and achieve high coverage rates.

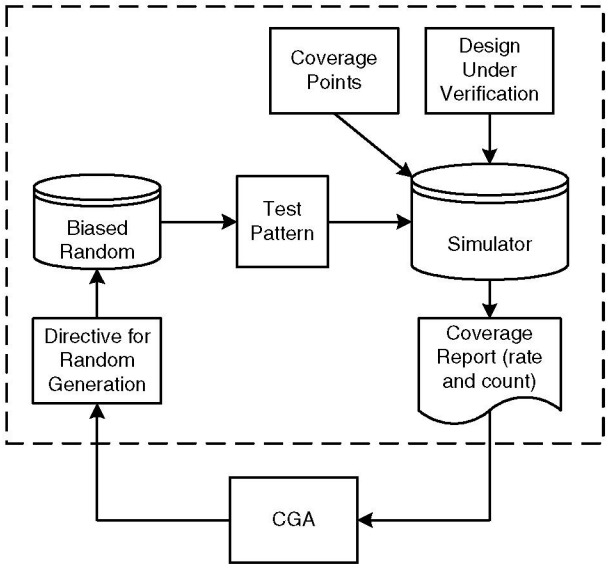

Fig. 1. Automatic Coverage Directed Test Generation (CDG)

To close the loop between the coverage report and directives for test generation, our CGA analyzes the coverage reports, evaluates the current test directives, and acquires the knowledge over time to modify the test directives. This increases the efficiency and quality of the verification process and reduces the time, manual effort, and human intervention needed to implement a test plan.

Figure 2 presents the proposed CGA based CDG methodology. It starts by producing initial random potential solutions, then two phases of iterative processes are performed: a Fitness Evaluation phase and a Selection and Diversification phase. During the Fitness Evaluation phase, we evaluate the quality of a potential solution that represents possible test directives. This process starts with extracting the test directives and generating test patterns that stimulate the DUT. Thereafter,

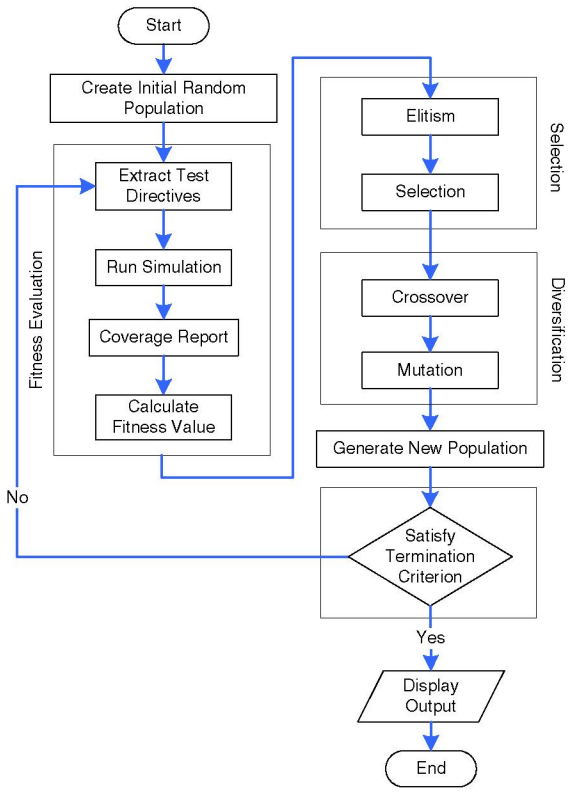

Fig. 2. CDG Methodology using CGA

the CGA collects the coverage data and analyzes them, then assigns a fitness value to each potential solution that reflects its quality. The evaluation criterion is based on many factors including coverage rate, variance of the coverage rate over the same coverage group, and the number of activated points. In the Selection and Diversification phase, several evolutionary operations (Elitism and Selection) and genetic operators (Crossover and Mutation) are applied on the current population of potential solutions to produce a new better population. These two phases will be applied to each population until the algorithm satisfies the termination criterion.

IV. Cell-based Genetic Algorithm - CGA

Like any genetic algorithm, following components of our CGA for CDG should be carefully chosen:

1) a genetic representation for potential solutions.
2) a way to create the initial population of solutions.
3) an evaluation that plays the role of the environment.
4) genetic operators (e.g., crossover and mutation) that alter the composition of new population.
5) values for various parameters that the genetic algorithm uses (e.g., population size, probabilities of applying genetic operators, weight and selection of genetic operators, etc.).

In the next section, we describe in details specification and implementation issues of all components of the CGA including initialization, elitism, selection, crossover, mutation, and termination criteria.

A. Representation (Encoding)

Traditionally, genetic algorithms use a fixed-length bit string to encode a single value solution. However, the solution of the CDG problem is described as sequences of directives that direct the random test generator to activate the whole coverage group. This introduces the need of more complex and rich representations of the potential solution.

A *Cell* is the fundamental unit introduced to represent a partial solution. Each cell represents a weighted uniform random distribution over two limits. Moreover, the near optimal random distribution for each test generator may consist of one or more cells according to the complexity of that distribution. However, we call the list of cells represents that distribution a *Chromosome*. Usually, there are many test generators that drive the DUT, and so we need a corresponding number of chromosomes to represent the whole solution, which we call *Genome*. Figure 3 represents a possible solution for some input i which is composed of 3 cells.

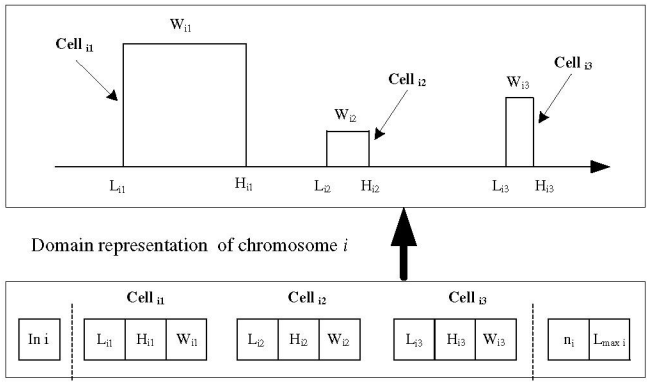

Fig. 3. Chromosome Representation

Let $Cell_{ij}$ be the j^{th} cell corresponding to test generator i that is represented by n_i bits. This cell has three parameters (as shown in Figure 3) to represent the uniform random distribution: low limit L_{ij}, high limit H_{ij}, and weight of generation W_{ij}. Moreover, these parameters have the following ranges:

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

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. Finally, the representation of a chromosome and the mapping between a chromosome and its domain representation is illustrated in Figure 3.

Figure 4 depicts a genome, which is a collection of many chromosomes each representing one of the test generators. Each genome is assigned a fitness value that reflects its quality. A genome also holds many essential parameters required for the evolution process. These include the complexity of chromosomes which equals the number of cells in it, the mutation probability P_m of a cell, the crossover probability

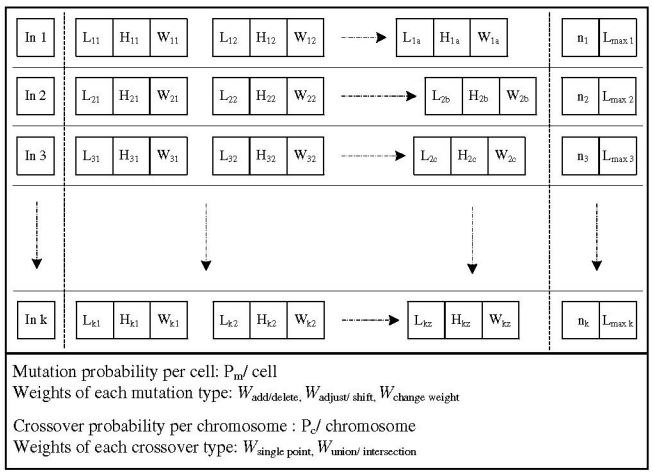

Fig. 4. Genome Representation

P_c of a chromosome, and the weights (W) of the generation of each type of mutation and crossover. The values of the probabilities and weights are constant during the evolution process.

B. Initialization

The CGA starts with an initial random population according to one of the following two initialization schemes:

1) Fixed period random initialization: Given a chromosome $Chrom_i$ that spans over the range $[0, L_{max}]$ and is represented by n_i bits, we divide the whole range of $Chrom_i$ into n_i equal sub-ranges and then generate a random initial cell within each sub-range as shown in Figure 5.

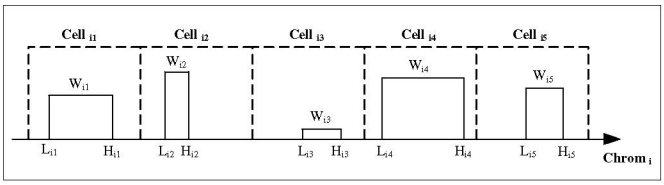

Fig. 5. Fixed Period Random Initialization

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

2) Random period random initialization: Here, we generate a random initial cell within the useful range $[0, L_{max}]$; this new cell will span over the range $[L_{i0}, H_{i0}]$, then we generate the successive cells within the range $[H_{ij}, L_{max}]$ until we reach the maximum range limit L_{max}. In other words, the low limit of each cell must come after the end of the previous cell.

C. Elitism

To ensure continues non-decreasing maximum fitness over generations, we use an elitism mechanism. Accordingly, we copy the best 5% of the fittest individual without any change and forward them to the next generation of solutions. This mechanism guarantees that the best chromosomes are never destroyed by either the crossover or the mutation operators.

D. Selection

For reproduction purposes, the CGA employs the well known roulette wheel fitness proportionate and tournament selection methods [11]. Selection operators and methods must ensure a large diversity of the population and prevent premature convergence on poor solutions while pushing the population towards better solutions over generations. In the case of the tournament selection, we can control the selection pressure of highly fitted individuals by changing the tournament size.

E. Crossover

Crossover operators are applied with a probability P_c for each chromosome. In order to determine whether a chromosome will undergo a crossover operation, a random number $p \in [0, 1]$ is generated and compared to P_c. If $p < P_c$, the chromosome will be selected for crossover, else it will be forwarded without modification to the next generation. Crossover is important to keep useful features of good genomes of the current generation and forward that information to the next generation. It is considered as the wheel of learning for genetic algorithm.

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: $W_{cross-1}$ to Single Point Crossover and $W_{cross-2}$ to 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:

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

(1) Single Point Crossover is similar to a typical crossover operator where each chromosome is divided into two parts and an exchange of these parts between two parent chromosomes is taken place around the crossover point as shown in Figure 6. The algorithm starts by generating a random number $C \in [0, L_{max}]$, then it searches for the position of the point C among the cells of the involved chromosomes in the crossover operation. If point C lies within the range of $Cell_{ij}$ that is $[L_{ij}, H_{ij}]$, then this cell will be split into two cells $[L_{ij}, C]$ and $[C, H_{ij}]$ around point C, as shown in Figure 6. Finally, an exchange of the cells between the two involved chromosomes takes place around point C to produce a new chromosome. At this point, the complexity of the solution as well as the total weights of cells must be computed for future use.

(2) Inter-Cell Crossover is a merging of two chromosomes rather than an exchange of parts of two chromosomes. Given two chromosomes $Chrom_i$ and $Chrom_j$, we define two types of merging as follows:

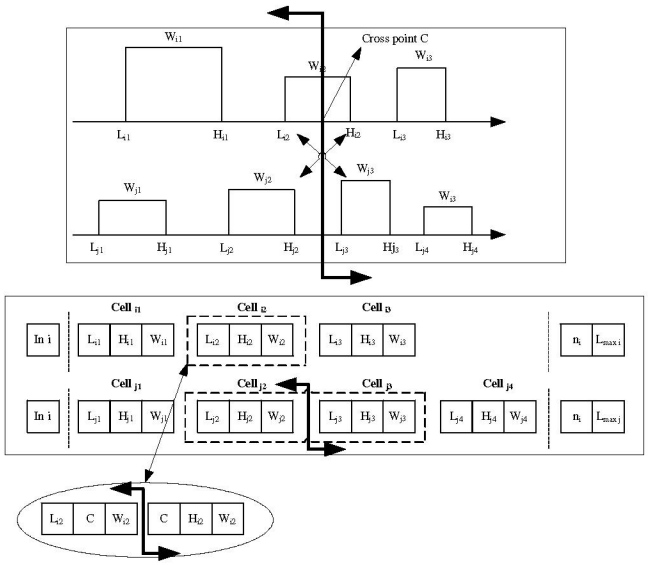

Fig. 6. Single Point Crossover

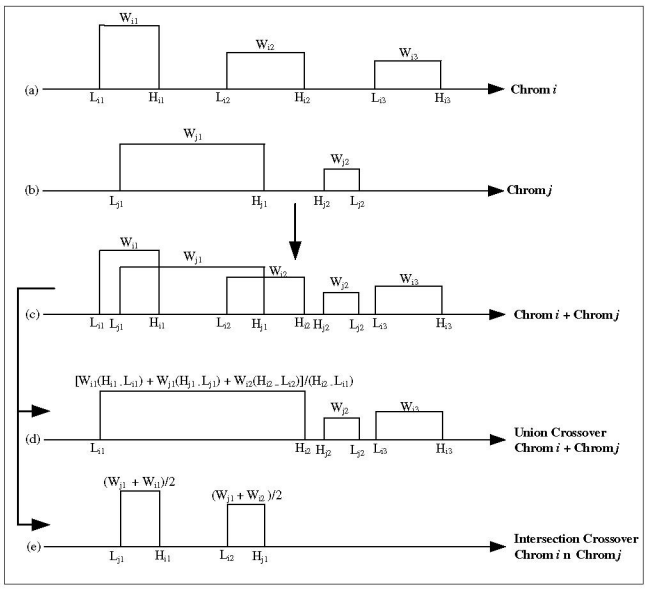

Fig. 7. Inter Cell Crossover

- Merging by Union ($Chrom_i \cup Chrom_j$): Combine $Chrom_i$ and $Chrom_j$ while replacing the overlapped cells with only one averaged weighted cell to reduce the complexity of the solution and to produce a less constrained random test generator (phenotype). The weight of a new merged cell will be proportional to the relative width and weight of each cell involved in the merging process as illustrated in Figure 7.
- Merging by Intersection ($Chrom_i \cap Chrom_j$): Extract average weighted cells of the common parts between $Chrom_i$ and $Chrom_j$, where the new weight is the average of the weights of the overlapped cells. This will produce a more constrained random test generator.

The inter-cell crossover operation is illustrated in Figure 7. This type of crossover is more effective in producing a new useful offspring since it shares information between chromosomes along the whole range rather than at single crossover points as in the first type. Furthermore, inter-cell crossover is able to transfer the good features over generations while presenting distinct cells that enrich the learning process.

We use the same procedure to find out the union and intersection of two chromosomes. Given a $Cell_{ij}$ spanning over the range $[L_{ij}, H_{ij}]$, a random number $N \in [0, L_{max}]$ may lay in one of the following three regions with respect to $Cell_{ij}$ as shown in Figure 8:

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

For example, to find the possible intersection of $Cell_{ik}$ with $Cell_{ij}$, the procedure searchs for the relative position of L_{ik} (low limit of $Cell_{ik}$) with respect to $Cell_{ij}$ starting from $Region2$ down to $Region0$ for the purpose of reducing the computational cost. If L_{ik} lies in $Region2$, then no intersection nor merging is possible, else the procedure searches for the relative position of H_{ik} (high limit of $Cell_{ik}$) with respect to $Cell_{ij}$. According to the relative position of both L_{ik} and H_{ik} with respect to $Cell_{ij}$, the procedure decides whether there is an intersection or a merging or not, and whether it has to check the relative position of $Cell_{ik}$ with respect to the successive cells of $Cell_{ij}$ for possible intersection or merging regions.

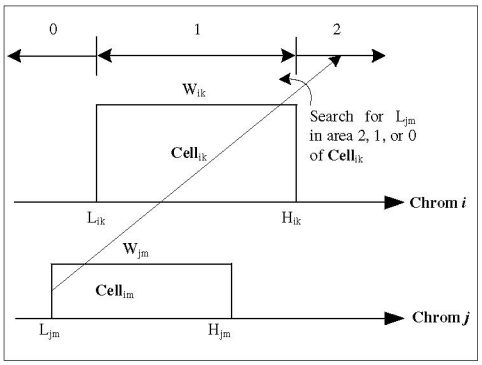

Fig. 8. Inter Cell Crossover Procedure

F. Mutation

Mutation operators introduce new features to the evolved population which are important to keep a diversity that helps the genetic algorithm to escape from a local minimum and explore hidden areas of the solution space. Mutation is applied on individual cells with a probability P_m for each cell in contrast to crossover operators, which is applied to pairs of chromosomes. Moreover, the mutation rate is proportional to the complexity of chromosome such that more complex chromosomes will be more sensible to mutation.

Again, due to the complex nature of our genotype and phenotype, we propose many mutation operators that are able to mutate the low limit, high limit, and the weight of cells. According to the mutation probability P_m, we can decide whether a cell will be selected for mutation or not. In case that 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 cell's weight.

The selection of the mutation operator is based on predefined weights associated with each of them, in a similar manner to the selection of crossover operators.

1) Insert or delete a cell: This mutation operator is to delete a $Cell_{ij}$ or to insert a new one around it. Moreover, we select either insertion or deletion with equal probability. If deletion is chosen, we pop $Cell_{ij}$ out of the chromosome and proceed for the next cell to check the applicability of mutation on it. In case insertion is selected, we insert a new randomly generated cell either behind or next to $Cell_{ij}$. This new random cell must reside within the gap between $Cell_{ij}$ and the previous or next cell according to the relative position of this new cell, with respect to $Cell_{ij}$.

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

3) Change cell's weight: This mutation operation replaces the weight of $Cell_{ij}$ with a new randomly generated weight within the range $[0, 255]$.

G. 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 consider simultaneously. For instance, we may achieve 100% for some coverage points while leaving other points totally inactivated. Accordingly, we designed a four stages fitness evaluation that targets 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 *CovRate1*.
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 [14] will be used to favor solutions that produce more hits of coverage points above the threshold coverage rates.

H. Termination Criterion

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

I. Random Number Generator

Random number generator is frequently used in simulation based verification like the Specman environment. Besides, it is used by genetic algorithm and other evolutionary techniques during the process of evolution and learning. The question of how random is the random process becomes an important issue. A poor random generator will not be useful for simulation based verification where sequences tend to be repeated within a short cycle that may be shorter than the simulation runs. Besides, a poor random generator might drive the genetic algorithm to the same local optima. This is why we adopt the Meanness Twisted algorithm [9] for stochastic decision during the learning and evolution process as well as during the simulation.

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 very long period of 2^{19937} - 1.

V. Experimental Results

The proposed CGA has been implemented in C++ using the STL library on a Windows XP OS. The CGA was tested on a number of hardware designs modeled in SystemC [2]. We compare some results of our CGA with those obtained with Specman Elite [14] to show the effectiveness of our algorithm with respect to such an industrial tools. Table I below summarizes the parameters used in the experiments to follow.

A. Small CPU

Specman Elite [14] and the e-language [13] enable verification engineers to define environmental constraints, to direct the random test generation, and to collect the coverage results. In

Parameter	Value
Population size	50
Tournament size	5
Number of generations	50
Mutation probability	95%
Crossover probability	20%
Weights of crossover	$W_{cross-1} = 1$, $W_{cross-2} = 2$
Weights of mutation	$W_{mut-1} = 2$, $W_{mut-2} = 3$, $W_{mut-3} = 1$
Coverage rate 1	10
Coverage rate 2	25

TABLE I
CGA PARAMETERS

this experiment we compare the results of our algorithm with Specman Elite for a tutorial from Specman that is a model of a simple CPU represented by 4 state machines (see Figure 9).

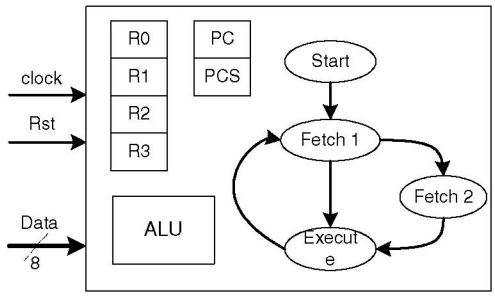

Fig. 9. Simple CPU

We define 4 types of coverage points inherited form the Specman tutorial. The first point covers all state machines and the second one covers all opcodes. The purpose of the third point is to find a suitable input that is able to set the carry flag as much as possible. The last point is a cross coverage point that tries to maximize the cases of setting the carry flag with all addition instructions.

The results in Table II show that our CGA achieved better results in half of the cases. The CGA was able to activate all FSM states and generate all opcodes. Furthermore, the CGA found the best instruction sequences in order to maximally utilize the carry flag.

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

TABLE II
COVERAGE RESULTS (CPU)

B. Router

In the second experiment, we consider a router model. The aim here is to evaluate the performance of our CGA when dealing with coverage points that may raise conflicts while optimizing the input ranges.

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

TABLE III
COVERAGE RESULTS (ROUTER)

The router gets input packets and directs them to one of 16 output channels, according to their address fields. A packet is composed of four fields: destination address (4 bits), packet's length (8 bits), data and parity byte (8 bits). Figure10 shows the block diagram of the router.

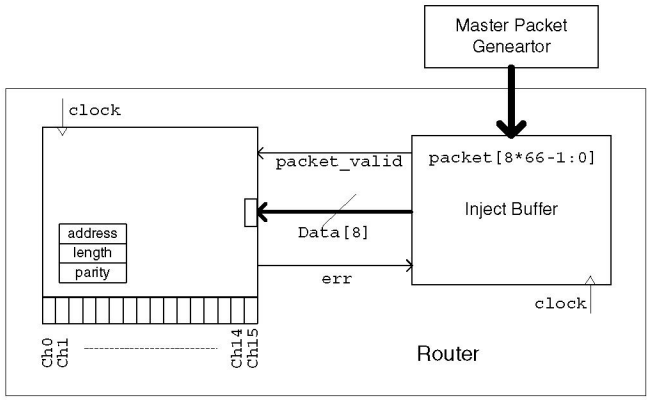

Fig. 10. Router Block Diagram

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

Table III summarizes the coverage results obtained using the CGA. The first column in Table III includes four coverage points to check the output on some specific channels of the router, i.e., the packet's destination address. The third and fourth points, add more constraints by checking not only the packet address but also the packet's length. The experimental results in the second column of the same table confirms that the CGA succeeded to find the appropriate parameters in order to generate only useful packets for toggling the corresponding coverage points. For instance, the CGA takes advantage from the multi-cell model used to represent the input ranges (see Section IV-A).

The above coverage points shown in Table III concern the destination address (4 bits) and packet length (8 bits). Accordingly, we encode only these two fields of the packet

into a genome of a single chromosome of 12 bits range (it could be a genome of two chromosomes, one chromosome to represent the destination address while the other represents the packet length).

Figure 11 shows the evolution progress related to the first coverage point. It highlights the improvement of the average coverage rate of the whole population over many generations of evolution. A noticeable average coverage improvement occurred around the fifth generation. In the same manner, Figure 12 shows the evolution progress related to the second coverage point.

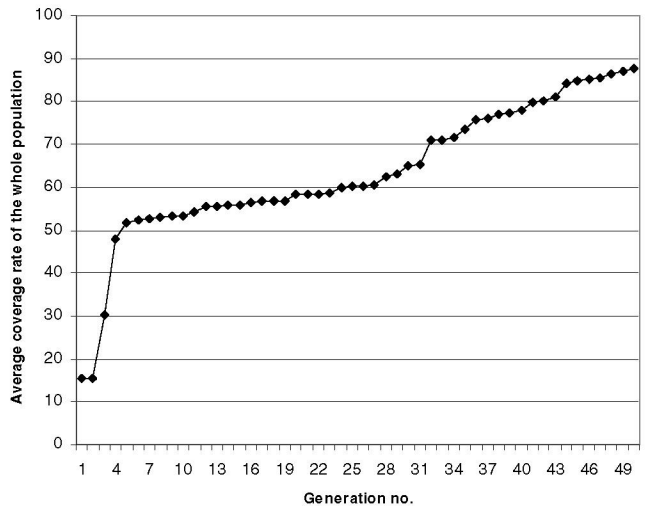

Fig. 11. Coverage Improvement Over Generations (First Coverage Point)

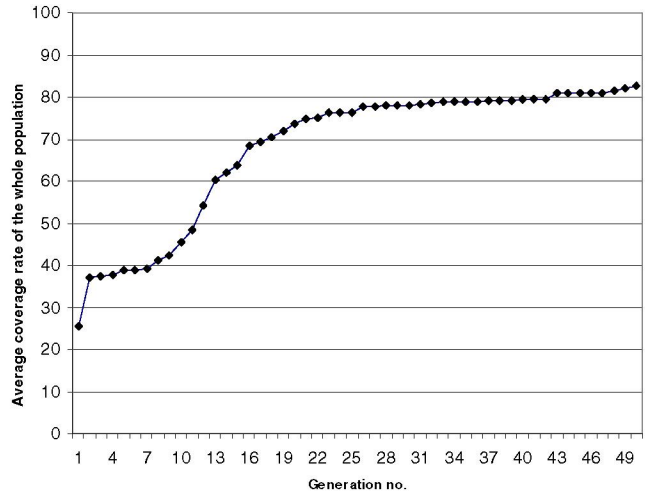

Fig. 12. Coverage Improvement Over Generations (Second Coverage Point)

VI. Conclusion

In this paper, we presented a methodology based on genetic algorithms to enhance the coverage rate for a group of correlated coverage points by closing the feedback path between the coverage space and test directives. The experimental results show the efficiency of our cell-based genetic algorithm in finding useful directives for many coverage points. Moreover, our algorithm shows better results and achieved higher coverage rates than the industrial Specman tool [14]. In addition, our algorithm shows a distinguishable ability in finding proper data and address directives and achieving up to 100% coverage rates for each individual coverage point.

In a future work, we intended to develop a self adaptation scheme of our genetic evolutionary framework, where the weights of mutation and crossover operators as well as the probability of applying each of them can be adapted during the evolution process. This will make our CGA totally evolvable and more flexible.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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