Comparison of Genetic and Random Techniques for Test Pattern Generation*

E. Ivask[1], J. Raik[1], R. Ubar[2]

[1]*Department of Computer Engineering, TTU, Raja 15, EE0026 Tallinn, Estonia, E-mail: ieero@va.ttu.ee*
[2]*Currently on the leave at the Institut National Polytechnique de Grenoble/CSI, France*

ABSTRACT: Current paper presents a test pattern generation approach based on genetic algorithms. The algorithm is designed so that it allows direct comparison with random methods. Experimental results on ISCAS'85 benchmarks [6] show that the proposed algorithm performs significantly better than similar approach published in [1]. In addition, the test sets generated by the algorithm are more compact.

1 Introduction

In current paper we introduce a test generation approach based on genetic optimization. In Section 2, the algorithm for test pattern generation is explained. Section 3 provides experimental results, which show that the genetic approach outperforms random test generation. According to the experiments, our approach achieves better fault coverages and more compact test sizes than the CRIS genetic test pattern generator reported in [1].

2 A Genetic Algorithm for Test Generation

John Holland, the founder of the field of genetic algorithms, pointed out the ability of simple representations (bit strings) to encode complicated structures and the power of simple transformations to improve such structures [3]. He showed that with a proper control structure, rapid improvements of bit strings could occur (under certain transformations), so that a population of bit strings 'evolves' as populations of animals do. An important formal result stressed by Holland was that even in large and complicated search spaces, given certain conditions on the problem domain, genetic algorithms would tend to converge on solutions that were globally optimal or nearly so.

In order to solve the problem, the following components are must in genetic algorithm:

1) a chromosomal representation of solution to the problem,
2) a way to create an initial population of solutions,
3) an evaluation function that plays the role of the environment, quality rating for solutions in terms of their "fitness"
4) genetic operators that alter the structure of "children" during reproduction
5) values for the parameters that genetic algorithm uses (population size, probabilities of applying genetic operators)

2.1 Representation

In a genetic framework, one possible solution to the problem is called an *individual*. Like we have different persons in society, we also have different solutions to the problem (one is more optimal than the other). All individuals together form a *population* (society).

In context of test generation, test vector (test pattern) will be the individual and the set of test vectors will correspond to population.

2.2 Initialization

Initially, a random set of test vectors is generated. This set is subsequently given to a simulator tool for evaluation. For research purpose random initializing a population is good. Moving from a randomly created population to a well adapted population is a good test of algorithm, since the critical features of final solution will have been produced by the search and recombination mechanisms of the algorithm rather than the initialization procedures.
Following steps of algorithm are carried out repeatedly.

2.3 Evaluation of test vectors

Evaluation is used to measure the fitness of the individuals, i.e. the quality of solutions, in a population. Better solutions will get higher score. Evaluation function directs population towards progress because good solutions (with high score) will be selected during selection process and poor solutions will be rejected.

We use fault simulation with fault dropping in order to evaluate the test vectors. The best vector in the population is determined and added to the selected *test vector depository*. The depository consists of test vectors that form the final set of test vectors. By adding only one best vector to the depository, we assure that the final test set will be close to minimal.

* This work has been supported by the Estonian Science Foundation grant G-1850

2.4 Fitness scaling

As a population converges on a definitive solution, the difference between *fitness* values may become very small. Best solutions can not have significant advantage in reproductive selection. We use square values for test vector's fitness values in order to differentiate good and bad test vectors.

2.5 Selection of candidate vectors

Selection is needed for finding two (or more) candidates for crossover. Based on quality measures (weights), better test vectors in a test set are selected. Roulette wheel selection mechanism is used here. Number of slots on the roulette wheel will be equal to population size. Size of the roulette wheel slots is proportional to the fitness value of the test vectors. That means that better test vectors have a greater possibility to be selected. Assuming that our population size is N, and N is an even number, we have N/2 pairs for reproduction. Candidates in pair will be determined by running roulette wheel twice. One run will determine one candidate. With such a selection scheme it can happen that same candidate is selected two times. Reproduction with itself does not interfere. This means the selected vector is a good test vector and it carries its good genetic potential into new generation.

2.6 Crossover

Exchanges corresponding genetic material from two parent chromosomes allow useful genes on different parents to be combined in their offspring. Crossover is the key to genetic algorithm's power. Most successful parents reproduce more often. Beneficial properties of two parents combine.

From pair of candidate vectors selected by roulette wheel mechanism, two new test vectors are produced by one-point crossover as following:

1) we determine a random position m in a test vector by generating a random number between 1 and L, assuming that L is the length of the test vector
2) first m bits from the first candidate vector are copied to the first new vector
3) first m bits from second candidate vector are copied to the second new vector
4) bits $m + 1 \ldots L$ from first candidate vector are copied to second new vector (into bits $m + 1 \ldots L$)
5) bits $m + 1 \ldots L$ from the second candidate vector are copied to the first new vector (into bits $m + 1 \ldots L$)

2.7 Mutation

Random mutation provides background variation and occasionally introduces beneficial material into a species' chromosomes. Without the mutation all the individuals in population will sooner or later be the same (because of the exchange of genetic material) and there will be no progress anymore. We will be stuck in a local maximum.

In order to encourage genetic algorithm to explore new regions in space of all possible test vectors, we apply mutation operator to the test vectors produced by crossover. In all the test vectors, every bit is inverted with a certain probability p. It is also possible to use a strategy where only predefined number of mutations are made with probability p=1 in random bit positions. This should reduce the computational expense. However, experiments showed decrease in fault coverage. Therefore, this method is not used here.

Steps 2.2 – 2.5 are repeated until all the faults from the fault list are detected or a predefined limit of evolutionary generations is exceeded. Test generation terminates also when the number of noncontributing populations exceeds a certain value. The value depends on the circuit size and is equal to *Number of inputs / const*, where const is a constant that can be set by the user. The smaller the value of *const*, the more thoroughly we will search.

In current implementation, the test generation works in two stages, with different mutation rates.

1) In the first stage, when there are lots of undetected faults and fitness of vectors is mostly greater than zero (in each evolutionary generation many faults are detected), a smaller mutation rate is used (p = 0.1).
2) In the second stage, when there are only few undetected faults and none of the vectors in population detects these faults, the weights of the vectors will all be zeros. We can not say which vector is actually better than others. Now the mutation rate is increased (p = 0.5) to bring more diversity into population, in order to explore new areas of the search space.

3 Experimental Results

The experiments were partly aimed at showing how much is the genetical approach better than random. In order to achieve that, same simulation procedures were used for random and genetic test generation. Population size for the genetic test generator was set to 32. It is a tradeoff between speed and fault coverage. In each (evolutionary) generation, or step, one vector from 32 is selected and put into final test set (vector depository). The random test generator performs in a similar way. It generates patterns in packages of 32 vectors. The best vector from the package (based on simulation results) will be selected, if it detects some previously not detected faults. Therefore, we can compare the two methods adequately. Both of the test generation tools belong to the diagnostics software package Turbo Tester [8]. All of the experiments were run on a Sun SparcStation 20 computer.

The experiments were carried out on ISCAS'85 benchmarks[6]. In first experiment, minimum number of test vectors was determined to detect all detectable faults. It comes out that genetic method requires always less test vectors (patterns) to yield the same fault coverage than random method. For the 'hard-to-test' circuits c2670 and c7552, equal number of test vector simulations for both

methods was taken and then the fault coverage reached was estimated. Genetic method discovers 118 faults more than random in the case of c2670 and 33 faults more in the case of c7552. Execution times for the random method were slightly shorter for smaller circuits like c432 and c499.

Subsequently, we investigated fault detection in time for random and genetic generators. Result graphs are presented in Figure 1 and Figure 2. Random generator achieves good fault coverage sooner but genetic generator detects additional faults in the end. Except for the smallest circuits c432 and c499 as we see in Tables 1 and 2. Effectiveness of genetic generator comes evident in case of circuits that have a large number of inputs. We compared our results to the ones achieved in [2]. The key feature there was keeping certain inputs together (in order to better propagate fault effects) during reproduction process. The method detected all faults for c7552 and c2670. However, the approach given here uses (up to 2 times) less of test vectors for all circuits. We could not compare execution times, because they were not revealed.

In addition, our results were compared to the genetic approach in [1]. The key feature of the latter method is monitoring circuit activity. Namely, information about the activity of internal nodes during fault simulation is collected, and points in the circuit where fault propagation was blocked are identified. Based on that information fitness values for test vectors are given. The comparison between our approach and [1] is presented in Table 3. It is evident that such a monitoring used in [1] is not effective.

Simple approach given here detects all detectable faults with a smaller time for all circuits and generates 1,6 – 6,5 times less test vectors than [1]. Comparison was not adequate for circuits c2670 and c7552 because in [1] the test generation was terminated too early.

References

[1] D. G. Saab, Y. G. Saab, J. A. Abraham, “CRIS: A test Cultivation Program for Sequential VLSI Circuits”, Intl. Conf. Computer-Aided Design, Nov. 1992, pp. 216-219.

[2] I. Pomeranz, S. M. Reddy, “On improving Genetic Optimization based Test Generation”, Proc. of European Design and Test Conference 1997, pp. 506-511

[3] J.H. Holland, “Adaptation in Natural and Artificial Systems”, University of Michigan Press, 1975

[4] E. M. Rudnick, J. H. Patel, G. S. Greenstein, T. M. Niermann “Sequential Circuit Generation in a Genetic Framework”, 31st ACM/IEEE Design Automation Conference

[5] Goldberg “Genetic algorithms”, Addison-Wesley USA,1991

[6] F. Berglez, H. Fujiwara, “A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran”, Proc. of the Int. Test Conf., pp. 785-794, 1985.

[7] S. R. Ladd, “Genetic Algorithms in C++”, M&T Books, 1996

[8] R.Ubar, J.Raik, P.Paomets, E.Ivask, G.Jervan, A.Markus. “Low-Cost CAD System for Teaching Digital Test”, .Microelectronics Education. World Scientific Publishing Co. Pte. Ltd. pp. 185-188, Grenoble, France, Feb. 1996