Applying Genetic Algorithm to Optimize Path Testing and to Achieve Total Code Coverage

Mohd Athar, Nargis Parveen
Research Scholars
Department of Computer Science, Shri Venkateshwara University
Gajraula, India
m.atharjmi@gmail.com, nargis.parveen@gmail.com

***Abstract*— The objective of programming is not to accomplish a result, but relevance and correctness of the result also required. Correctness can be tested by applying software testing on the developed program. Testing is most important practice which is performed for supporting better quality product. Efficient ways can reduce percentage of cost and time incurred in testing. In spite of scads of theoretical work in field of Software Testing, its advancement is slow towards automation. In this approach, Genetic Algorithm (GA), which is a meta-heuristic algorithm, is employed for optimizing path testing to achieve total code coverage. Random testing is used as a comparison of the efficiency and effectiveness of test data generation using Genetic Algorithms. The advantage of GAs is that through the search and optimization process, test sets are improved such that they are at or close to the input sub domain boundaries. The GAs gives most improvements over random testing when these sub domains are small. Optimization of software testing is achieved by employing GA and the process is automated. It results in formulation of test suite for a module that gives 100 % code coverage. The process of code analysis to find all modules in a program, generation of CFG, finding cyclomatic complexity, determination of all independent paths and GA steps are automated.**

***Index Terms*— Genetic Algorithm, BBT, SUT.**

I. INTRODUCTION

Software testing is important but it possesses some fundamental challenges. It poses two essentially arduous jobs; selecting tests and assessing test results. Selecting test cases are hard as there is enormous number of potential test inputs in varied sequences but only some of them unwrap failures. In Evaluation/assessment, the real output of test run is compared with expected result. This evaluation is done in opaque-testing. Test Suite (TS) generation from operational profile can be automated but it poses substantial hardheaded problems. Time plays a foremost constraint in case of testing. Another vital component is cost. Due to these two constraints, it is intricate task to execute all test cases. When coverage is taken as optimization parameter then target is formulation of TS that could give 100 % code coverage. Optimization problems can be unbridled by GA which can be regarded as computer model of biological evolution. It works on principle of evolution, where superior chromosomes (having greater fitness value) are chosen for mutation and crossover operations. Evolution continues until the optimized solution is achieved. Good results are found astoundingly speedily when GA is implemented. Generating optimized TS is meta-heuristic problem which can be resolved by GA. Testing tools can be put in two class; dynamic & static.

It’s important to have adequate test cases for accomplishment of testing and making software more dependable [2]. Making system reliable is vital as flunking it could sustain massive losses. In [2], BBT is optimized by applying GA. It’s implemented in Matlab [version no 7]. Test cases of SUT are heavily influenced by GA. GA depends on various parameters. Population size is vital parameter. Bigger population size brings variation in initial populace at cost of more function evaluations and longer completing times.

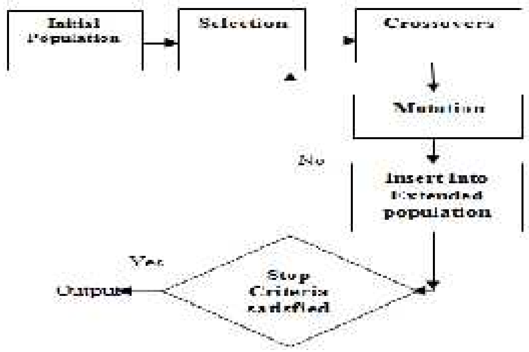


Figure 1 Proposed solution

Figure 1 shows the procedure adopted in [2]**.** Fitness function is defined. Threshold fitness is set**.** Population size is set, crossover rate taken 0.6, mutation rate taken .001**.**While halt condition is not met**,** reproduce by crossover and mutation

Testing job is reckoned to be optimization problem whose intent is maximization of noticing errors with minimization of effort. GAs with specification can obtain results with superior quality in lesser time [2]. Figure 6 shows, experiments have been done on “program to find largest number” Inputs are obtained by program’s specification.

A
B
C
Program to find largest number
Largest number

Figure 2 Block representation of input to the program and output obtained

Figure 2 shows block representation of program designed to generate test cases. The program is to finding largest number among inputted numbers. It is having multiple inputs and one output variable.

Two central issues in process of evolution of genetic search are population diversity and selective pressure. There may be chances of converging early to local optimum solutions if strong pressure is given on selection. On other hand, weaker selection pressure may leads to unproductive search and optimization [3]. Fitness is the key for selection of parents. Fitter chromosomes have more likelihood to get selected. Crossover works on two chromosomes which are chosen as parents to construct two children. In this research work [3], point crossover is employed; two parents are divided partly, with both children getting half of each of parents. Mutation is applied to amend genes in chromosome. For example, let a chromosome be: 'abdfag', Mutation may pick gene at position 2 and transform it to a 'z', thus, ensuing a new chromosome: 'azdfag'.

In the work done by Dhawan et.al.[3], test data for input set is delimitated in terms of stipulations which illustrate valid and invalid data values. Stipulations are determined from program's specification.

Role of fitness function to improve WBT

GA is used by WBT to search for precise test data which give high coverage of SUT. Fitness function is necessary feature of GA and is engineered on basis of SUT [10]. Objective function is used to construct fitness function which is applied to sequent genetic ops. Intent of GA is to maximize fitness function. If fitness function is modeled well, probability of reaching higher coverage is enhanced considerably. Based on CFG and requisite test aim, test criteria are separated in different classes [10]:

• Node-oriented methods: It requires traversal of particular nodes in CFG. Statement test and condition test can be categorized in this class. Accomplishment of partial aim of this method isn't reliant on path executed in CFG.

• Path-oriented methods: It requires traversal of definite path in CFG. This class comprise of every variation of path test. Finding fitness functions for this class of test is less sophisticated compare to node-oriented method.

Other test criteria can be node-path-oriented method and node-node-oriented method [10]. Baresel et.al [10] separates test into partial objectives and fitness functions are defined for each partial objective, i.e., each statement corresponds partial objective when applying coverage criterion. Ultimate goal of fitness function can be summarized as:

➢ Substantially enhance chance of detecting solution and attain improved coverage of SUT

➢ Reduce count of iterations to achieve optimization.

Intention of Baresel et.al [10] is to better formulation of fitness functions, so that, evolutionary testing could be enhanced by getting prominent coverage. It is difficult to investigate reasons behind unsuccessfulness of optimizations because of large search space and existence of many dimensions.

Operators of GA

Execution of GA commence with stochastic population of chromosomes. Fitness function assist evaluation of population and reproductive chances is allowed to population which symbolizes a more adept solution to problem. Chromosomes having superior fitness value are selected by Selection operator. Selection operator is crucial in GA. Jadaan et.al. [12] has proposed altered Roulette Wheel Selection(RWS) to reduce incertitude in selection process. RWS probabilistically choose individuals based on their fitness values (F). Fittest chromosome takes largest share within roulette wheel and chromosome with least fitness value takes smallest share. A random number is generated in interval $[0, S]$ where S is ΣF, chromosome whose segment is closer to random number is picked.

II. Motivation

Software testing is a principal technic which is employed for bettering quality attributes of Software Under Test(SUT), particularly reliability and correctness but is also regarded to be tedious. This is also supposed to be intricate work. Software testing suffers from the cognitive biasing of the testers. Automation of testing is a proficient way which can foreshorten time taken and cost incurred in software development. It can also notably better the quality of software.

III. Problem Statemnt

"To automate generation of TS for each module of SUT by applying GA that could give 100% code coverage"

Random generation of TS does not certify the traversal of all segments of code. There may be odds that the code segment which is not checked could end the program abnormally. So, to obtain optimized TS from set of many feasible TSs, GA is applied.

IV. Approach

Intent is to optimize TS which could give 100 % code coverage. This optimization which is grounded on total code coverage needs that inner composition of program is well-known. Inner composition of program can be discovered by Path testing in which a set of test-paths are selected in a program. The different independent paths in the program could be determined through CFG. An independent path is that path in CFG that has one novel set of processing statements or novel conditions. Test cases carrying the information of the path covered by them are grouped together to form initial population of chromosomes and GA is applied. In the end, TS is obtained for each module that gives hundred per-cent code coverage.

V. Methodology

Generating TS that guarantees full coverage of statements in program, is complex task. There are also odds that more than one test case in TS are checking same path. This redundancy is not appreciated. It is imperative to have optimized test data sets. In this section, GA is employed for optimizing Path testing.

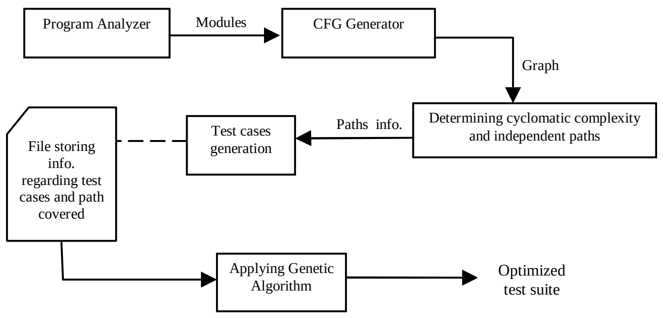

Program Analyzer
Modules
CFG Generator
Graph
Determining cyclomatic complexity and independent paths
Paths info.
Test cases generation
File storing info. regarding test cases and path covered
Applying Genetic Algorithm
Optimized test suite

Figure 3 Block diagram of methodology

Figure 3 illustrates approach applied in this work to accomplish the objective. Program analyzer analyzes the java program and discovers all the modules in it. CFG generator generates the CFG for each module. CFG is used to find CC and total independent paths. Test cases are generated and paths followed by them are found. The data regarding test cases and path followed are put in a file. This file is utilized when GA is employed. Each of the blocks is explicated fully in this chapter.

Methodology is divided into two approaches:

- Testing
- Applying Genetic Algorithm

VI. SURVEY

Generation of chromosomes

Chromosomes are constituted by grouping genes. In GA, chromosomes are optimized. In this work, TS need to be optimized; TS is taken as chromosome, so, test cases are genes. Test cases are randomly grouped to form chromosomes. This grouping form feasible solutions which are optimized to acquire best solution. Initially, size of chromosome is equal to count of total independent paths.

Selection and Crossover operator

Chromosomes having superior fitness value are selected by Selection operator. Reproduction of chromosomes is done by mutation and crossover operator.

Selection operator

This operator is employed to choose parents for mating pool. In this work, roulette selection wheel (RSW) is used as selection operator.

Steps of RSW:

Step 1: Total fitness of all chromosomes in population is calculated.

Step 2: Calculated accumulated frequency for each chromosome.

Step 3: Spawn a random number between zero & total sum of fitness

Step 4: **[Loop]** Find the sum of accumulated fitness of chromosomes from 0. When sum is greater than random number, return the chromosome.

In this way, parents are selected for mating pool.

Crossover operator

In this work, single point crossover is used.

Steps of single point crossover:

Step 1: Random points are selected in both the parents. This divides the chromosomes into two haves each.

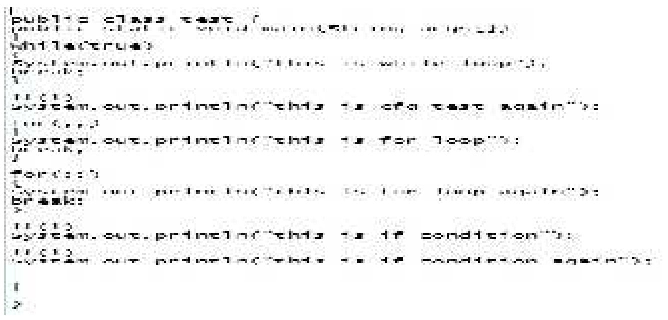
Step 2: Replace second half of first chromosome with second half of second chromosome.

Step 3: Similarly, replace second half of second chromosome with second half of first chromosome.

VII. RESULTS AND ANALYSIS

This section discusses regarding implementation of problem statement and goals attempted to accomplish by employing GA. Results got are analyzed graphically. Snapshots have been taken of a sample problem resolved by employing methodology discoursed in chapter 5. It also gives clear picture of implementation.

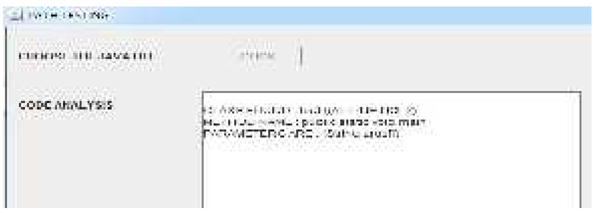
A sample problem is taken which is a java program. Since WBT is concerned with code structure not functionality, the module is doing simple task of displaying some statements.



```
{
public class test {
[illegible]
}
while(true)
{
[illegible]
[illegible]
}

if(1)
System.out.println("this is cfg test again");
[illegible]
}
System.out.println("this is for loop");
[illegible]
}

for(;;)
{
[illegible]
break;
}

if(1)
System.out.println("this is if condition");
if(1)
System.out.println("this is if condition again");

}
}
```

Figure 4 Sample problem

In first step, program is analyzed to discover the modules in it. Figure 4 is showing the result of code analysis.



```
[illegible]
METHOD NAME: public static void main
PARAMETERS ARE: String args[]
```

Figure 5 Output of Code Analyzer in TextArea

As depicted in figure 5, path of java file is given by clicking on “click” button. Class and methods information are displayed in “*TextArea*”. Information includes “class name”, ”methods in class”, “parameters of methods”. *Code Analyzer* also writes output in a “*.txt” file, which is used to fetch line numbers at which method definition exists.

Modules find by *code analyzer* is used by CFG generator to build CFG. *CFG generator* fetches the line number from where module begins from text file generated by *code analyzer*.

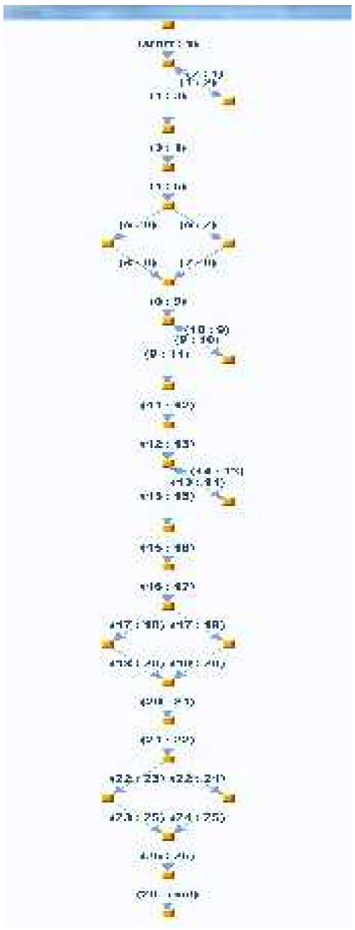
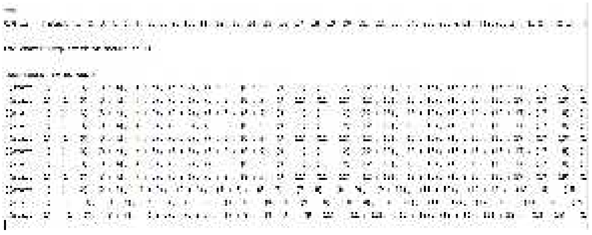


Figure 6 CFG of main module

Figure 6 shows the CFG of *main* module of sample problem. Orange buttons are vertices of CFG and arrows are edges of CFG. Arrows are labeled like “1:2” showing the flow of control from vertex “1” to vertex “2”.

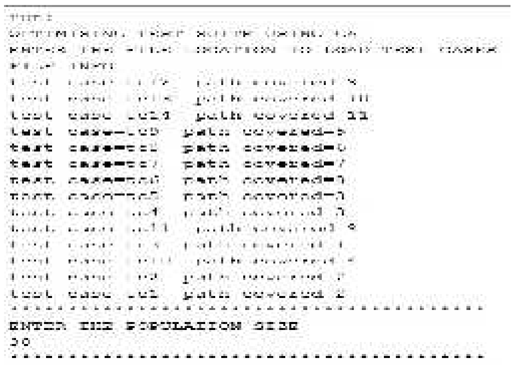


```
[illegible]
```

Figure 7 Displaying cyclomatic complexity and independent paths of “main” module

As depicted in Figure 7 cyclomatic complexity of “main” module is 11 and all the independent paths are displayed.

Next step is generation of random test cases. For each test case, corresponding path in CFG is determined. Information regarding the paths covered by test cases is stored in file.

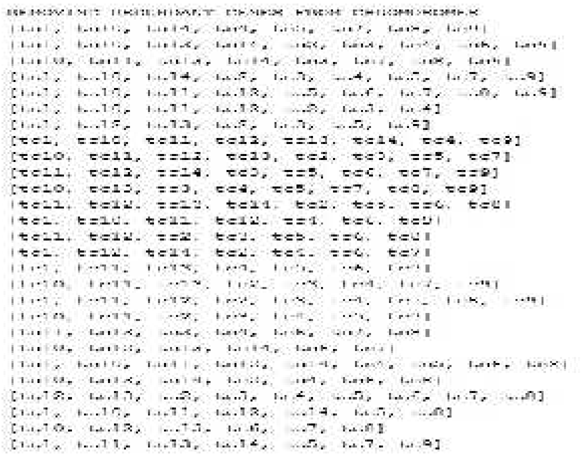


```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
test case tc13  path covered 11
test case=tc5  path covered=5
test case=tc1  path covered=6
test case=tc7  path covered=7
test case=tc6  path covered=3
test case=tc5  path covered=3
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
test case tc1  path covered 2
*********************************************
ENTER THE POPULATION SIZE
30
*********************************************
```

(a) Test cases and path information are loaded from file and initial population is set 30

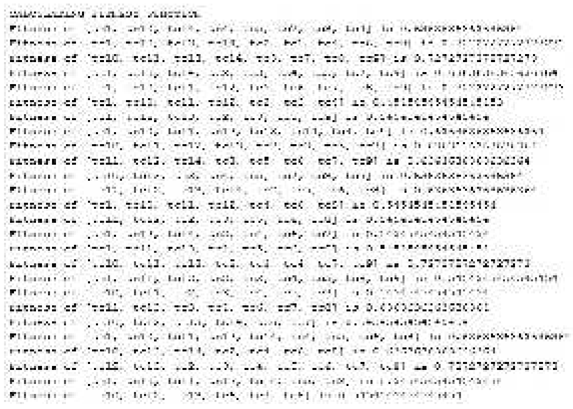
```
[illegible]
```

(b) Initial population of chromosomes are generated. Generation 1



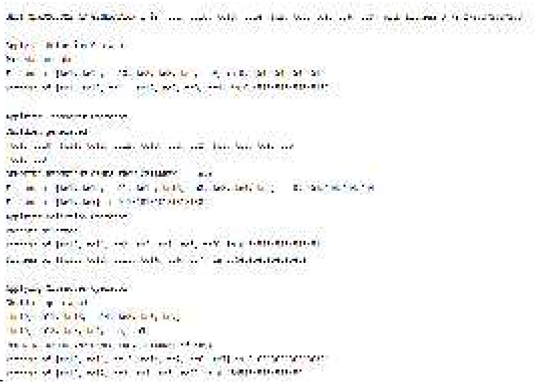
```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[tc1, tc10, tc11, tc12, tc13, tc14, tc4, tc9]
[tc10, tc11, tc12, tc13, tc2, tc3, tc5, tc7]
[tc11, tc12, tc14, tc3, tc5, tc6, tc7, tc9]
[tc10, tc13, tc3, tc4, tc5, tc7, tc8, tc9]
[tc11, tc12, tc13, tc14, tc2, tc5, tc6, tc8]
[tc1, tc10, tc11, tc12, tc4, tc6, tc9]
[tc11, tc12, tc2, tc3, tc5, tc6, tc8]
[tc1, tc12, tc14, tc2, tc3, tc6, tc7]
[illegible]
```

(c) Redundant genes are removed



[illegible]

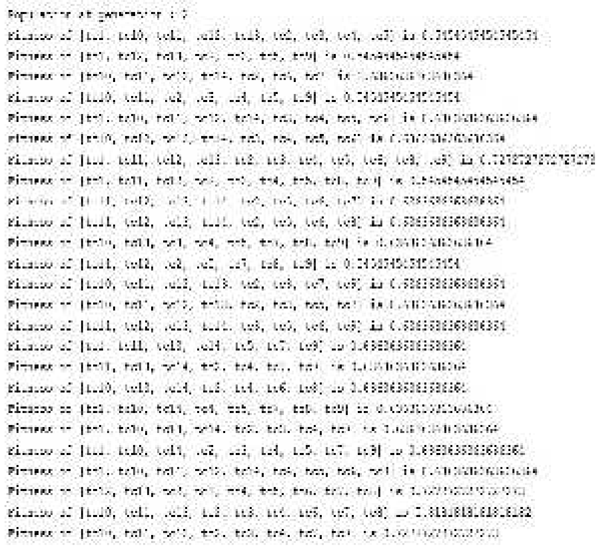
(d) Fitness values are calculated



[illegible]

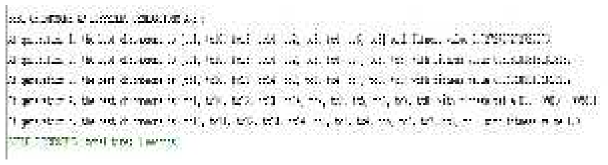
1

(e) Displaying best chromosome at generation 1. Selection and crossover are applied on population 1



[illegible]

(f) Displaying population at generation 2



[illegible]

(g) Displaying best chromosomes at different generations.

Figure 8 GA steps to sample problem (a,b,c,d,e,f,g)

Figure 8 shows the implementation of GA on the sample problem. In figure (a) information regarding genes is taken from a file and initial population size is defined. In figure (b) the initial populations of chromosomes are formulated from the genes. In figure (c) redundancy of chromosomes are removed. In figure (d), fitness of chromosomes are displayed. Fitness is calculated by the fitness function as proposed in methodology chapter. In figure (e), working of selection operator and crossover operator are shown. In figure (f), generation 2 is displayed obtained from generation 1 and by applying the GA operators. Figure (g) shows best chromosome of different generations.

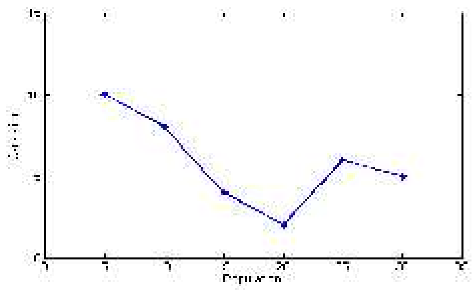


Figure 9 Initial population vs Generation graph

Figure 9 shows graph of initial population taken and number of generations taken to get an optimized solution. Here initial chromosomes size is 11 and number of test cases provided are 14. The graph shows if population is less, then, GA takes more generations to optimize. But after certain limit, if the population is increased then also GA doesn't perform well.

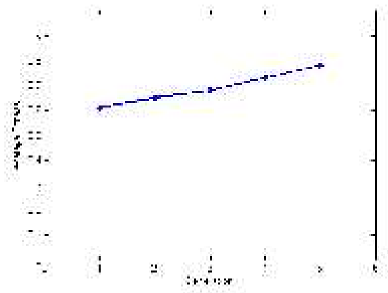


Figure 10 Generation vs Average fitness graph

Figure 10 shows graph between generation and average fitness of population. Initial chromosome size is 11 and test cases provided are 14 and population size is 30. It is noted that with every passing generation, average fitness of population is improving.

The same methodology is obtained on other problems also.

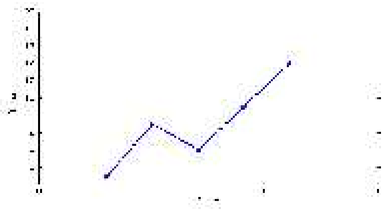


Figure 11 Chromosome size vs. Generation graph

Figure 11 depicts graph between initial size of chromosomes (x-axis) and number of generations (y-axis) taken to get an optimized solution. Initial population is fixed to 5. Chromosome's size is unswervingly proportional to intricacy of module. So, as the intricacy of module increases, it takes more generations to obtain an optimized solution.

VIII. Conclusion and Limitations

In this work, optimization of software testing is achieved by employing GA and the process is automated. It results in formulation of test suite for a module that gives 100 % code coverage. The process of code analysis to find all modules in a program, generation of CFG, finding cyclomatic complexity, determination of all independent paths and GA steps are automated. GA is employed on a set of different software programs and analyses are done on results obtained which decide performance of GA.

In this work, test cases are created manually and paths followed by them are manually determined. RSW selection operator is employed for selecting parents and single point crossover is employed as crossover operator. In future, test case generation from operational profile and path followed by them in CFG can be automated. Other selection operators and crossover operator can be applied and comparison can be drawn between performances of different operators.

Acknowledgment

This paper could not be written to its fullest without the guidance of Dr. Avdhesh Gupta, who served as my supervisor, as well as one who challenged and encouraged me throughout my time spent studying under him. He would have never accepted anything less than my best efforts, and for that, I thank him.

References

[1] Bayliss, D. and Taleb-Bendiab, A.: 'A global optimisation technique for concurrent conceptual design', Proc. Of ACEDC'94, PEDC, University of Plymouth, UK., pp. 179-184, 1994

[2] BCS SIGIST (British Computer Society, Specialist Interest Group in Software Testing): Glossary of terms used in software testing, 1995

[3] DeMillo R. A. and Offutt A. J.: 'Experimental results from an automatic test case generator', ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 2, pp. 109-127, April 1993

[4] Feldman, M. B. and Koffman, E. B.: 'ADA, problem solving and program design', Addison-Wesley Publishing Company, 1993

[5] Frankl P. G. and Weiss S. N.: 'An experimental Comparison of the effectiveness of branch testing and Data Flow Testing', IEEE Transactions on Software Engineering, Vol. 19, No. 8, pp. 774-787, August 1993

[6] Gallagher M. J. and Narasimhan V. L.: 'A software system for the generation of test data for ADA programs', Micro processing and Microprogramming, Vol. 38, pp. 637-644, 1993

[7] Gutjahr W.: 'Automatische Testdatengenerierung zur Unterstuetzung des Software tests', Informatik Forschung und Entwicklung, Vol. 8, Part 3, pp. 128-136, 1993

[8] Hills, W. and Barlow, M. I.: 'The application of simulated annealing within a knowledge-based layout design system', Proc. of ACEDC'94, PEDC, University of Plymouth, UK., pp. 122-127, 1994

[9] Holmes, S. T., Jones, B. F. and Eyres, D. E: 'An improved strategy for automatic generation of test data', Proc. of Software Quality Management '93, pp. 565-77, 1993

[10] Jin L., Zhu H. and Hall P.: 'Testing for quality assurance of hypertext applications', Proceedings of the third Int. Conf. on Software Quality Management SQM 95, Vol. 2, pp. 379-390, April 1995

[11] Korel B.: 'Dynamic method for software test data generation', Software Testing, Verification and Releliability, Vol. 2, pp. 203-213, 1992

[12] Lucasius C. B. and Kateman G.: 'Understanding and using genetic algorithms; Part 1. Concepts, properties and context', Chemometrics and Intelligent Laboratory Systems, Vol. 19, Part 1, pp. 1-33, 1993

[13] Müllerburg, M.: 'Systematic stepwise testing: a method for testing large complex systems', Proceedings of the third Int. Conf. on Software Quality Management SQM 95, Vol. 2, pp. 391-402, April 1995

[14] O'Dare, M. J. and Arslan, T.: ' Generating test patterns for VLSI circuits using a genetic algorithm', Electronics Letters, Vol. 30, No. 10, pp. 778-779, February 1994

[15] Parmee, I. C. and Denham, M. J.: 'The integration of adaptive search techniques with current engineering design practice', Proc. of ACEDC'94, PEDC, University of Plymouth, UK., pp. 1-13, 1994

[16] Parmee I. C., Denham M. J. and Roberts A.: 'evolutionary engineering design using the Genetic Algorithm', International Conference on Design ICED'93 The Hague 17-19, August 1993

[17] Rayward-Smith, V. J. and Debuse, J. C. W.: 'Generalized adaptive search techniques', Proc. of ACEDC'94, PEDC, University of Plymouth, UK., pp. 141-145, 1994

[18] Reeves, C., Steele, N. and Liu, J.: 'Tabu search and genetic algorithms for filter design', Proc. of ACEDC'94, PEDC, University of Plymouth, UK., pp. 117-120, 1994

[19] Roberts, A. and Wade, G.: 'Optimization of finite wordlength Filters using a genetic algorithm', Proc. of ACEDC'94, PEDC, University of Plymouth, UK., pp. 37-43, 1994

[20] Roper, M.: 'Software testing', International software quality assurance Series, 1994

[21] Schultz A. C., Grefenstette J. J. and DeJong K. A.: 'Test and evaluation by Genetic Algorithms', U.S. Naval Res. Lab. Washington D.C. USA, IEEE Expert, Vol. 8, Part 5, pp. 9-14, 1993

[22] Sthamer, H.-H., Jones, B. F. and Eryes, D. E.: 'Generating test data for ADA generic Procedures using Genetic Algorithms', Proc. of ACEDC'94, PEDC, University of Plymouth, UK., pp. 134-140, 1994