Automatic Test Case Generation Using Genetic Algorithm

Rakesh Kumar, Surjeet Singh, Girdhar Gopal

Abstract— *Software testing is most effort consuming phase in software development. One would like to minimize the efforts and maximize the number of faults detected. Hence test case generation may be treated as an optimization problem. One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. Generating test cases automatically will reduce cost and efforts significantly. In this paper, test case data is generated automatically using Genetic Algorithms and results are compared with Random Testing. It is observed that Genetic Algorithms outperforms Random Testing.*

Index Terms— Automatic Test Case Generation, Boundary Value Analysis, Evolutionary Algorithms, Genetic Algorithm, Random Testing, Software Testing.

1 Introduction

Software development consists of various phases like Requirement analysis, Design, Coding and Testing. Out of these testing consumes maximum efforts. The software is tested with enough set of test cases, to make a judgment about quality or acceptability and to discover errors. Two fundamental techniques used to identify test cases are functional and structural testing. Testing of software using these two approaches is very time consuming. So to reduce the efforts there should be a mechanism to generate test cases automatically. A number of techniques are available to generate automatic test cases like random testing, anti-random testing etc. Objective of all these techniques is to find minimal number of test cases to test the software fully. This can be considered as an optimization problem. To solve optimization problems there are a number of techniques and one of them is Genetic Algorithms. Genetic algorithms are population based search based on the Darwin's principle of *survival of the fittest.* GA is basically an evolutionary technique inspired by biological evolution. It was developed in 1970's by J. Holland and his colleagues and his students at University of Michigan's [1]. It mimics the process of natural evolution. GA starts with a initial population and then apply genetic operators like selection, crossover, mutation and replacement on that population to evolve better and better individuals. GA can be terminated in either of two cases: maximum number of generations achieved or optimum value found [2].

2 Problem Statement

The most significant weakness of testing is that the postulated functioning of the tested system can, in principle, only be verified for those input situations which were selected as test data. Testing can only show the existence but not the non-existence of errors, [3]. Proof of correctness can only be produced by a complete test, i. e. a test with all possible input values, input value sequences, and input value combinations under all practically possible constraints. In practice, complete testing is usually impossible because of the vast amount of possible input situations. Testing can therefore only be a sampling method. Accordingly, the selection of an appropriate sample containing the most error-sensitive test data is essential to testing. If test data relevant to the practical deployment of the system are omitted, the probability of detecting errors within the software declines. Of all the testing activities – test case design, test execution, monitoring, test evaluation, test planning, test organization, and test documentation – essential importance is thus attributed to test case design, [4].

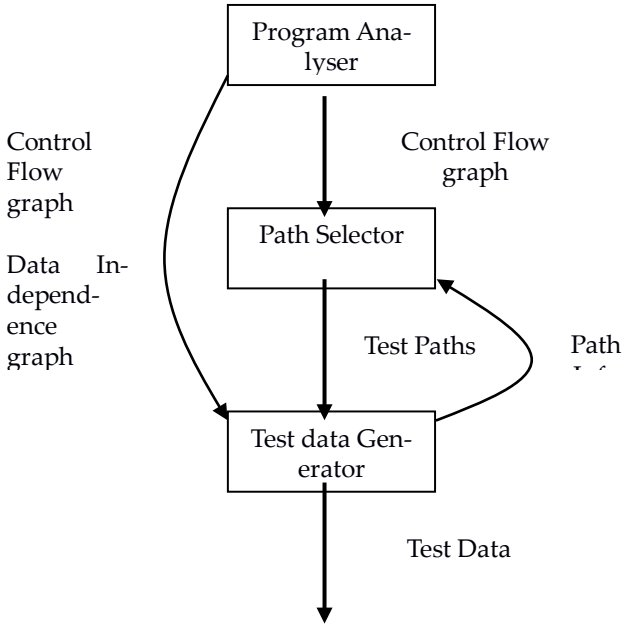

Fig.1: Architecture of a test data generator system

Figure 1 models a typical test data generator system, it consists of three parts: program analyzer, path selector and test data generator. The source code is run through a program ana-

- *Rakesh Kumar is currently Professor at DCSA, Kurukshetra University, Kurukshetra, Haryana-136119, INDIA,E-mail: rakeshkumar@kuk.ac.in*
- *Surjeet Singh is currently Assistant Professor at GMN College, Ambala Cantt., Haryana, INDIA,E-mail: surjeetsagwal@gmail.com*
- *Girdhar Gopal is currently pursuing Ph.D. in Computer Acience and Applications in Kurukshetra University, Kurukshetra, Haryana-136119, INDIA, PH-+91-9896482704, E-mail: girdhar.gopal@kuk.ac.in*

lyzer, which produces the necessary data used by the path selector and the test data generator. The selector inspects the program data in order to find suitable paths. Suitable can for instance mean paths leading to high code coverage. The paths are then given as argument to the test data generator which derives input values that exercise the given paths. The generator may provide the selector with feedback such as information concerning infeasible paths.

Software Testing accounts for approximately 50% of total software cost, [5]. This cost could be reduced if the process of testing is automated. In the past, a number of different methods for generating test date have been presented. These methods are divided in three classes: *Random, path-oriented and goal oriented* test data generation [6].

Due to the non-linearity of software (if-statements, loops, etc.), the conversion of test problems into optimization tasks usually results in complex, discontinuous, and non-linear search spaces. Neighborhood search methods such as hill climbing are not suitable in such cases. Therefore, meta-heuristic search methods, such as evolutionary algorithms, are employed. The suitability of evolutionary algorithms for testing is based on their ability to produce effective solutions for complex and poorly understood search spaces with many dimensions. The dimensions of the search spaces are directly related to the number of input parameters of the system under test. The execution of different program paths and the nested structures in software systems lead to multi-model search spaces when testing.

In order to automate software tests using evolutionary algorithms, the test aim must itself be transformed into an optimization task. A numeric representation of the test aim is necessary, from which a suitable fitness function for the evaluation of the generated test data can be derived. Depending on which test aim is pursued, different fitness functions emerge for test data evaluation. If an appropriate fitness function can be defined for the test aim, and evolutionary computation is applied as the search technique, then the Evolutionary Test proceeds as follows. The initial set of test data is generated, usually at random. In principle, if the test data has been obtained by a previous systematic test, this could also be used as an initial population, [7]. The Evolutionary Test could thus benefit from the tester's knowledge of the system under test. Each individual within the population represents a test datum with which the system under test is executed. For each test datum the execution is monitored and the fitness value determined for the corresponding individual.

Next, test data with high fitness values are selected with a higher probability than those with a lower value and are subjected to combination and mutation processes to generate new offspring test data. It is important to ensure that the test data generated are in the input domain of the test object. The main idea behind evolutionary testing is the combination of interesting test data in order to generate offspring test data that truly fulfill the test objectives. The offspring test data are evaluated by executing the system under test. A new population of test data is formed by merging offspring and parent individuals according to the survival procedures laid down. From here on, the process repeats itself, starting with selection until the test objective is fulfilled or another given stopping condition is reached. Evolutionary testing can be summarized as following algorithm:

```
testCaseGeneration()
  allTargets = targets(programUnderTest);
  initPop = generateInitialPop(popSize);
  curpop = initpop;
  while Not isEmpty(allTargets)
        p = selectTarget(allTargets);
        attempt = 0;
        while notcovered(p) and attempts < maxAttempts
              execute testCases in curPop;
              update allTargets;
              if covered(p)
                    break;
              compute fitness[p] for testCases in curPop
              extract newPop from curPop      according
              to fitness[p]
              crossover newpop;
              mutate newPop;
              curPop = newPop;
              attempts = attempts + 1;
        end while;
  end while;
end;
```

3 RELATED WORK

Random testing is the simplest technique of test data generation. Actually it could be used to generate data for any type of program since; ultimately, every data is a string of bits. But random testing mostly does not perform well in terms of coverage. Since it merely relies on probability it has quite low chances in finding semantically small faults [8], and thus accomplishes high coverage. A fault that is only revealed by small percentage of program input is called semantically small fault. For example, in following code:

```
void function1(int x, int y)
{       if (x ==y)        print("ONE");   // statement 1
        else              print("ZERO");  // statement 2
}
```

The probability of executing *statement 1* is 1/n, where n is the maximum integer, since to execute *statement 1*, both x and y must be same. So random testing can generate these types of test data at very low probability.

Random testing selects test data randomly from the input domain and then test the program with these test cases. The automatic production of random test data, drawn from a uniform distribution, should be the default method by which other systems should be judged, [9]. Statistical testing is a test case design technique in which the tests are derived according to the expected usage distribution profile.

The distribution of selected input data should have the

same probability distribution of inputs which will occur in actual use in order to estimate the operational reliability, [10].

There is not much difference between partition and random testing in terms of finding faults, [Hamlet and Taylor (1990)]. Hamlet showed that random testing is superior to partition testing with regard to human effort especially with more partitions and if confidence is required. For a small number of sub-domains partition testing will perform better than random testing.

Random number generators are ineffective in that they rarely provide the necessary coverage of the program, [12]. This comment was strengthened and new opinion that random testing is probably the poorest methodology in testing was given, [13].

However, many errors are easy to find, but the problem is to determine whether a test run failed. Therefore, automatic output checking is essential if large numbers of tests are to be performed, [14], [15]. They also said that partition testing is more expensive than performing an equivalent number of random tests which is more cost effective because it only requires a random number generator and a small amount of software support. The change of range for random testing has a great effect, [14]. Further they mentioned a disadvantage of random testing which is to satisfy equality values which are difficult to generate randomly.

The advantage of random testing is normally that it is more stressing to the program under test than hand selected test data, but on the other hand random inputs may never exercise both branches of a predicate which tests for equality, [16]. Even in the case that random testing is cheaper than partition testing, the slight advantage of random testing could be compensated for by using more random tests and there is no assurance that full coverage can be obtained, e.g. if equality between variables are required. And secondly it may mean examining the output from thousands of tests.

Random testing was especially recommended for the final testing stage of software by Tsoukalas (1993) and Girard and Rault (1973) [17], [18].

Duran and Ntafos (1984) recommended a mixed final testing, starting with random testing, followed by a special value testing method (to handle exceptional cases) [14]. Ince (1986) reported that random testing is a relatively cheap method of generating initial test data [9].

4 RESEARCH METHODOLOGY

Testing is a process which needs to be done effectively. Exhaustive testing is not possible due to limitation of sources. In past, it is observed that test cases lies in different classes. And programmers commit more mistakes in some classes of test cases. Most important class of these is boundary values, there are more chances for a software to fail at boundaries. So automatic test case generation must focus on these kind of test case classes which are more crucial for a software. Boundary values of a program are described as input boundaries of the variables used in program. For the sake of drawing the problem a function F is used. F is a function of two variables, x1 and x2. When the function F is implemented as a program, these input variables will have some boundaries: a <= x1 <= b; c <= x2 <= d;

The input space of function F is shown in Figure 2. Any point within the shaded rectangle is a legitimate point to the function F.

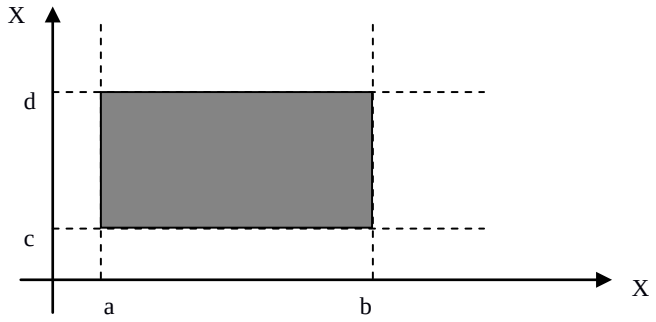

Figure 2: Input domain of a function of two variables

Boundary value analysis focuses on the boundary of the input space to identify test cases. So the basic idea is to select five values, minimum, just above the minimum, nominal value, just below the maximum and the maximum. So the test case values lying for Boundary value analysis are shown in Figure 3.

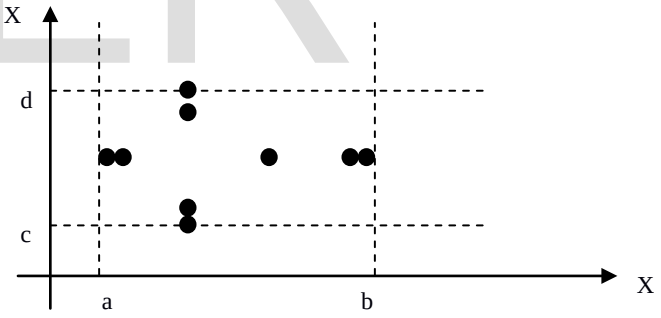

Figure 3: Boundary value analysis test cases for a function of two variables

Hence it is clear from the figure 3, where boundary value analysis tests. Boundary value analysis does not make sense for boolean variables. Because the extreme values for these variables are TRUE and FALSE. So the other three slots can't fill for boundary value analysis. Boundary value analysis works well in programs where the program is function of several independent variables that represent bounded physical quantities.

In this research, the researcher carried out the identification of these boundaries automatically through Genetic algorithm and random testing and then compares the results of both techniques. Genetic algorithm and random testing both starts with some random initial population and then

GA use the fitness of individuals to progress towards the optimums, whereas random testing works randomly throughout the run. For this experiment, the distance from the boundaries is taken as the fitness of the individual chromosome. The algorithms are coded in MATLAB 2011a.

Genetic Algorithm for test case generation:

The proposed genetic algorithm for test case generation for boundary value analysis is presented here. Firstly, The major components of GA are discussed and then overall algorithm is presented.

Representation

The proposed GA uses real values to taken the values of input variables x1,x2,.... of the program. The length of the chromosome depends on the number of variables.

Suppose we wish to generate test cases of a program P with input variables x1 and x2. Each of which has their value ranges within some domain say [c,d]. Then the values representing the chromosome are [a1, a2], where a1 and a2 are ranges within the respective variables domain [c, d].

Initial Population

Initial population is generated randomly as discussed in Representation section. pop_size vectors of c_size length are generated randomly, where pop_size is the size of population, and c_size is the number of variables. For the search space values, initial population is generated with 5 less than lower bound of variable, and 5 more than upper bound values, i.e. if lower bound and upper bound are 5,15 respectively, then initial values are generated from 0 (5-5) and 20 (15+5). The appropriate value of pop_size is experimentally determined.

Fitness Function

Fitness of each chromosome is determined by its difference from the boundaries of the variable. The more a variable is close to the boundaries the more it is declared fit.

```
Fitness(popsize, chromLength, curpop)
  lBound = lower boundary of the variable;
  uBound = upper boundary of the variable;
  for I = 1 to popsize
        for j = 1 to chromLength
              diffLower = lBound – curpop(I,j);
              diffUpper = uBound – curpop(I,j);
              moreClose = min(diffLower, diffUpper);
              fitness(i) = moreClose;
        end
  end
end;
```

Selection

After computing the fitness of each test case in the current population, the algorithm selects test cases from the effective members of the current population that will be parents of the new population. In the selection process the GA uses the *roulette wheel method* [2], whereas Random Testing use *random selection method*. These two methods are described below.

(i) Roulette wheel: For the selection of a new population with respect to the probability distribution based on fitness values, a roulette wheel with slots sized according to fitness is used. Such roulette wheel is constructed as follows:

- Calculate the fitness value $fit(v_i)$ for each chromosome v_i $(i=1,2,\ldots, pop\_size)$.
- Find the total fitness of the population $F = \sum fit(v_i)$.
- Calculate the probability of a selection p_i for each chromosome v_i ($i = 1, \ldots, pop\_size$):

  $p_i = fit(v_i)/F$.
- Calculate a cumulative probability q_i for each chromosome v_i ($i = 1, \ldots, pop\_size$):

  $q_i = \sum p_j$.

The selection process is based on spinning the roulette wheel pop_size times; each time we select a single chromosome for a new population in the following way:

- Generate a random (float) number r from the range [0..1].
- If $r < q_1$ then select the first chromosome (v_1); otherwise select the i-th chromosome v_i ($2 \le i \le pop\_size$) such that $q_{i-1} < r \le q_i$.

Obviously, some chromosomes would be selected more than once.

(ii) Random selection: In this method, the selection of parents is made randomly, so that every effective member of the current population has an equal chance of being selected for recombination.

Assume that l members of the current population were effective, where $l \le pop\_size$.

The parents are selected as follows:

Isolate the effective members and number them from 1 to l;
For i=1 to *pop_size* do
Begin
Generate an random integer number j from the range [0..l];
Select chromosome v_j from the effective members;
End For;

Crossover:

It operates at the individual level. During crossover, two parents (chromosomes) exchange sub string information (genetic material) at a random position in the chromosome to produce two new strings (offspring). The objective here is to create better population over time by combining material from pairs of (fitter) members from the parent population. Crossover occurs according to a crossover probability. The probability of crossover pc gives us the expected number $pc \cdot pop\_size$ of chromosomes, which undergo the crossover operation. which is as follows:

For each chromosome in the (new) population:

- Generate a random (float) number r from the range [0..1];
- If $r < pc$ then select given chromosome for crossover.

Now selected parents are randomly mated. For each pair of selected parents arithmetic crossover is used with crossover probability 0.7. Arithmetic crossover operator defines a linear combination of two chromosomes [19]. Two chromosomes are selected randomly for crossover and produce two offspring's which are linear combination of their par-

ents as per the following computation:
$C_{igen}+1 = a.C_{igen} + (1-a).C_{jgen}$
$C_{jgen}+1 = a.C_{jgen} + (1-a).C_{igen}$
where C_{gen} an individual from the parent generation , $C_{gen}+1$ an individual from child generation, a (alpha) is the weight which governs dominant individual in reproduction and it is between 0 and 1.

Following parameters are used in experiments:-

- *Population Size*: various population sizes are tried and best ones are taken for comparison i. e. 10, 20, 50 & 100.
- *Generations*: program is executed with different number of generations and analysis of less number of generations and more number of generations is also taken into consideration i. e. 100, 200, 500 & 1000.
- *Encoding*: chromosomes (test cases) are coded in real values, so Value encoding scheme of GA is used.
- *Selection*: Roulette wheel selection is used for Genetic Algorithm, and Random selection is implemented for Random Testing.
- *Crossover*: number of crossovers available for real value coding, out of which arithmetic crossover is applied with 0.7 probability.
- *Mutation:* uniform mutation is applied in experiments with 0.1 probability.
- *Replacement*: Simple genetic algorithm replacement takes place, in which whole new population replaces the old one.

4 RESULTS & OBSERVATIONS

All inputs are taken from user, so that testing with different parameters can be done easily. User interface while running in MATLAB is as follows: -
INPUTS:
No. of individuals in population : 20, No. of Variables : 2, No. of Generations : 200
limits of 1st variable : Lower limit : 5 & Upper limit : 15
limits of 2nd variable : Lower limit : 6 & Upper limit : 16
OUTPUTS: With Genetic algorithm, test cases generated as follows:

Variables / Runs	Variable 1	Variable 2
1	15.20	6.92
2	4.56	14.58
3	7.67	8.11
4	13.19	15.02
5	7.88	6.52

While for Random testing, following test cases are resulted as output:

Variables / Runs	Variable 1	Variable 2
1	13.98	12.67
2	9.75	14.78
3	11.23	10.85
4	8.81	13.06
5	10.93	8.64

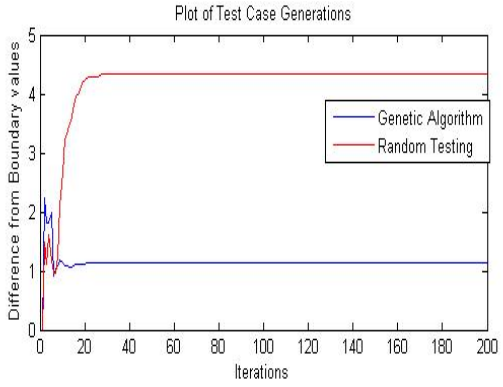

Figure 4

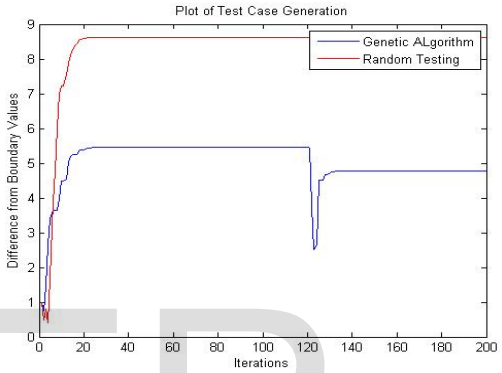

Figure 5

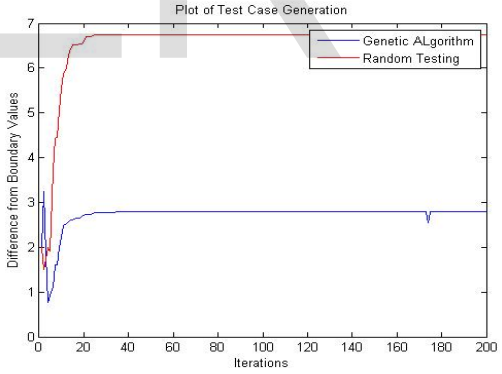

Figure 6

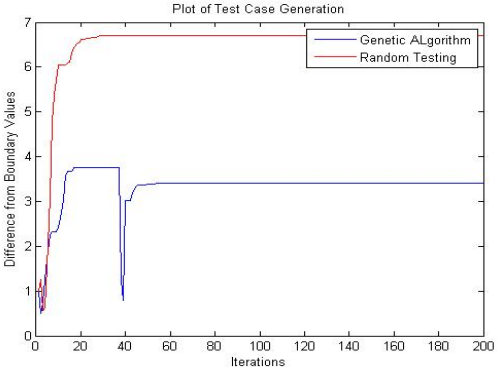

Figure 7

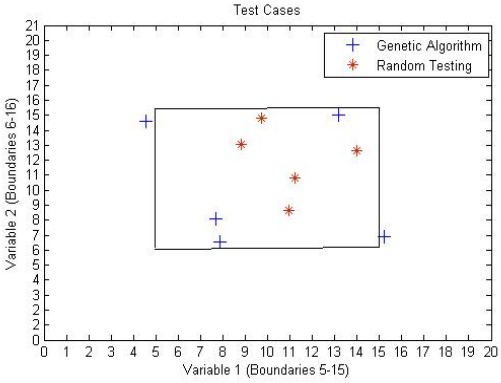

Figure 8: Results showing the final test cases analysis

Another run is carried out with following inputs and outputs:
INPUTS:
No. of individuals in population : 20, No. of Variables : 2, No. of Generations : 200
Limits of 1st variable : Lower limit : 5 & Upper limit : 15
limits of 2nd variable : Lower limit : 10 & Upper limit : 20
OUTPUTS: With Genetic algorithm, test cases generated as follows:

Variables / Runs	Variable 1	Variable 2
1	14.14	12.79
2	6.68	9.84
3	16.63	15.88
4	16.96	10.98
5	6.50	9.52

While for Random testing, following test cases are resulted as output:

Variables / Runs	Variable 1	Variable 2
1	14.00	14.93
2	10.95	12.39
3	10.34	15.34
4	11.88	16.96
5	8.11	12.72

Figure 9 to Figure 14 explains these executions:

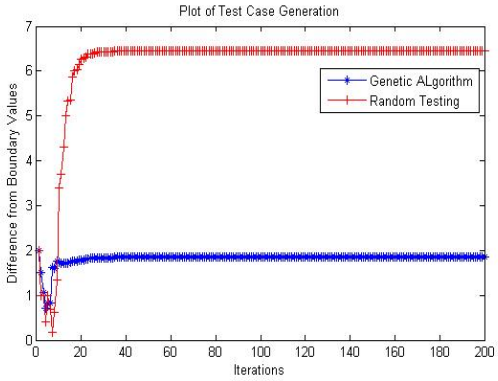

Figure 9

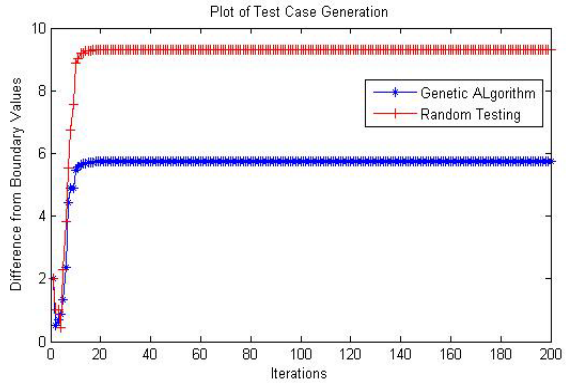

Figure 10

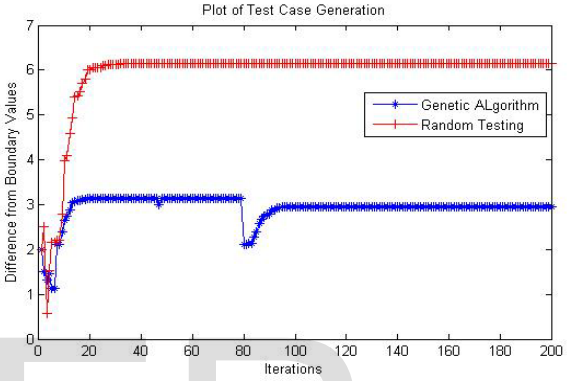

Figure 11

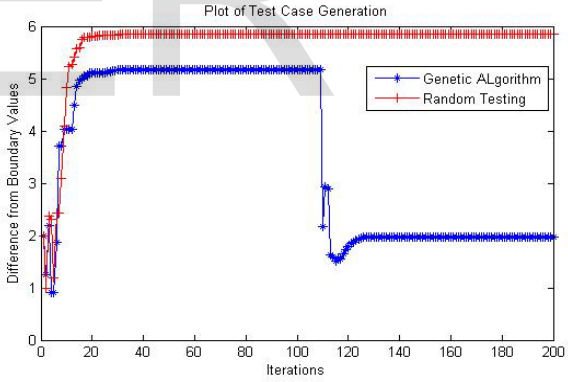

Figure 12

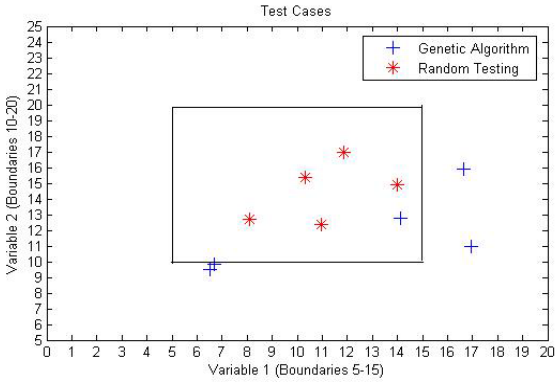

Figure 13: Results showing the final test cases analysis

5 CONCLUSION

The overall results show evolutionary testing to be a promising approach for fully automating test case design for boundary value analysis technique of testing. To increase the efficiency and effectiveness, and thus to reduce the overall development cost for software-based systems, a systematic and automatic test case generator is required. Genetic algorithms search for relevant test cases in the input domain of the system under test. Due to the full automation of test case generation, the overall quality of software is also enhanced in comparison to using random testing. The application scope of evolutionary test case generation can go further than the work described above. Additional application fields can be control flow graphs, path testing, stress testing etc. Actually, every technique of testing can be implemented using Genetic Algorithms automatic test case generation.

REFERENCES

[1] Holland J., "*Adaptation in natural and artificial systems*", University of Michigan Press, Ann Arbor, 1975.

[2] Goldberg D. E., "*Genetic algorithms in search, optimization, and machine learning*", Addison Wesley Longman, Inc., ISBN 0-201- 15767-5, 1989.

[3] Dijkstra, E. W., Dahl, O. J., Hoare, C. A. R.: "*Structured programming*", Academic Press., 1972.

[4] Wegener, J. and Pitschinetz, R.: "*TESSY – Yet Another Computer-Aided Software Testing Tool?*" Proceedings of the Second International Conference on Software Testing, Analysis and Review, Bruxelles, Belgium, 1994.

[5] Beizer B. "*Software Testing Techniques*". Van Nostrand Reinhold, 2nd edition, 1990.

[6] Ferguson R. and Korel B. "*The chaining approach for software test data generation*". IEEE Transactions on Software Engineering, 5(1):63-86, January 1996.

[7] Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H. and Jones, B.: "*Systematic Testing of Real-Time Systems*". Proceedings of the Fourth European International Conference on Software Testing, Analysis & Review, Amsterdam, Netherlands, 1996.

[8] Offutt J. and Hayes J., "*A semantic model of program faults*". In International Symposium on Software Testing and Analysis (ISSTA 96), pages 195{200. ACM Press, 1996.

[9] Ince, D. C.: "*The automatic generation of test data*", The Computer Journal, Vol. 30, No. 1, pp. 63-69, 1987.

[10] Taylor R.: '*An example of large scale random testing*', Proc. 7th annual Pacific North West Software Quality Conference, Portland, OR, pp. 339-48, 1989.

[11] Hamlet D. & Taylor R., "*Partition testing does not inspire confidence*", IEEE Transactions on Software Engineering, Vol. 16, 1990, pp. 1402-1411.

[12] Deason, W. H., Brown, D. B., Chang, K. H., and II, J. H. C. (1991). "*A Rule-Based Software Test Data Generator*". *IEEE Trans. on Knowl. and Data Eng.*, 3(1):108–117.

[13] Myers, G. J. (1979). *Art of Software Testing*. John Wiley & Sons.

[14] Duran J. W. and Ntafos S. C.: 'An Evaluation of Random Testing', IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, pp. 438-444, July 1984

[15] Duran, J. W. and Ntafos S., 'A report on random testing', Proceedings 5th Int. Conf. on Software Engineering held in San Diego C.A., pp. 179-83, March 1981

[16] Bertolino, A.: 'An overview of automated software testing', Journal Systems Software, Vol. 15, pp. 133-138, 1991

[17] Tsoukalas M. Z., Duran J. W. and Ntafos S. C.: 'On some reliability estimation problems in random and partition testing', IEEE Transactions on Software Engineering, Vol. 19, No. 7, pp. 687-697, July 1993

[18] Girard, E. and Rault, F. C.: 'A programming technique for software reliability', IEEE Symp. Computer Software Reliability, pp. 44-50, 1973

[19] Michalewicz Z., "Genetic Algorithms + Data Structures = Evolution Programs", Springer-Verlag, 2nd edition, 1994.