**An improved method of genetic algorithm for path testing**

Dinh Ngoc Thi1, Vo Dinh Hieu1, Nguyen Viet Ha1

1 VNU University of Engineering and Technology, Hanoi, Vietnam

{thidn.di13, hieuvd, hanv}@vnu.edu.vn

**Abstract.** Software testing is a process to identify the quality and reliability of software, which can be achieved through the help of proper test data. However, doing this manually is a difficult task due to the presence of　huge number of predicate nodes in the module. So, this leads towards a problem of NP-complete. Therefore, some heuristic search algorithms have to be used to generate test data. In this paper, a soft computing based approach for generating test data automatically using genetic algorithm (GA) based upon the set of given target paths is proposed. This paper also proposes a method to improve the GA, in which static analysis is used to find paths having the path conditions with low probability in generating coverage data, then basing on these path conditions to adjust the procedure of generating a new population. Finally, the proposed approach is applied some test functions. Experimental results show that improved GA which can generate suitable test data have higher path coverage than the traditional GA.

**1. Introduction**

Software quality becomes more important than ever and software testing is the most significant measure for it. However, software testing is very laborious and costly due to the fact that it is mostly made by manual [1]. In general, software testing accounts for approximately 50 percent of the elapsed time and more than 50 percent of the total cost in software development [2]. Thus, automated software testing is a promising way to cut down time and cost.

Automatic structural test data generation is a crucial problem in software testing automation and its implementation can not only significantly improve the effectiveness and efficiency but also reduce the high cost of software testing. We focus on path coverage test data generation in respect that various structural test data generation problem can be transformed into a path coverage test data generation problem. Furthermore, path coverage testing strategy can detect almost 65 percent of errors in program under test [3].

Although path coverage test data generation is an undecidable problem [4], researchers still attempt to develop various methods and have made some progress. These methods can be classified into two types: static methods and dynamic methods.

Static methods include symbolic execution [5] and domain reduction [6, 7] etc. These methods suffer from a number of problems when it handles indefinite loops, array, procedure calls and pointer references [8].

Dynamic methods include random testing, local search approach [9], goal-oriented approach [10], chaining approach [11] and evolutionary approach [8, 12-14]. Since values of input variables are determined when programs execute, dynamic test data generation can avoid those problems with that static methods are confronted.

As a robust search method in complex spaces, genetic algorithm (GA) was applied to test data generation in 1992 [12] and evolutionary approach has been a burgeoning interest since then. Related works [8, 15, 16] indicate that GA-based test data generation outperforms other dynamic approaches e.g. random testing and local search.

As far as we know, even though GA-based test data generation already proved its efficiency in generating test data for dynamic approaches, it still has to face difficulties when applying to path coverage, which are for the test function having test path with low probability in generating coverable test data, traditional GA can not generate test data which can cover these difficult paths.

This paper gives the proposal to improve traditional GA in generating test data which can cover all the test paths. It is to combine static program analysis with GA. The static program analysis step is applied to find out paths of the test function which are difficult to be covered. After that, basing on the path condition of the difficult to be covered path to adjust the procedure of generating new population in GA.

This paper is organized as follows: Section 2 presents some theoretical background to understanding this research. Section 3 summarizes some related works, and Section 4 gives the proposed approach in detail. Section 5 presents the evaluation and discussion. Section 6 concludes the paper.

**2. Background**

**2.1. Path testing**

The objective of path testing is to search for a collection of test cases (inputs to a program) that between them lead to the traversal of all logical paths through the program. In general, path testing process consists of two major steps: target paths generation and test data generation. Both of them will be mentioned in our proposal of this paper.

*2.1.1. Target paths generation*

Target paths generation means identifying a set of logical execution pathways through the program, that we hope should all be exercised during testing. The source code is needed to construct its logical control flow, which can be presented in a control flow graph (CFG). This graph can be automatically generated by using appropriate programming language grammar in which the program is written.

From the CFG, the different logical paths through the program need to be enumerated. A logical path is a particular flow of execution through the program, which is determined by the decisions made at each decision point between the program’s entry point and its exit point.

*2.1.2. Test data generation*

Generating test data that fulfill path coverage is the main task in path testing. It is the process of creating test data, either heuristically or randomly. In a heuristic approach, the process is guided by some rules to search for required test data; the alternative is that random test data is generated.

**2.2. Generic algorithm**

The basic concepts of genetic algorithm (GA) were developed by Holland [17]. GA are commonly applied to a variety of problems involving search and optimisation. GA search methods are rooted in the mechanisms of evolution and natural genetics. GA draw inspiration from the natural search and selection processes leading to the survival of the fittest individuals. GA generate a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms.

The principle behind GA is that they create and maintain a population of individuals represented by chromosomes (essentially a character string analogous tothe chromosomes appearing in DNA). These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to rules of selection, mutation and reproduction.

Each individual in the environment (represented by a chromosome) receives a measure of its fitness in the environment. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information) and is analogous to the process of sexual reproduction. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The structure of a simple GA is given below.

Simple Genetic Algorithm ()

{

initialize population;

evaluate population;

while stopping criteria not reached

{

select solutions for next population;

perform crossover and mutation;

evaluate population;

}

}

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

**3. Related work**

The path coverage literature started with Lin and Yeh [18] in 2000. They extended Jones et al.'s work [19] from branch coverage to path coverage. The ordinary (weighted) Hamming distance was extended to handle different ordering of target paths that have the same branches. The fitness function is called SIMILARITY, which computes similar items with respect to their ordering within two different paths between actual executed path and the target path. Only one program was used to test the approach, i.e. simple triangle classifier. They reported that the approach outperformed random search. However, in this method, evaluation function must be called many many times in order to generate the test data for the most difficult path to be covered.

Bueno et al. [20] proposed an approach that utilizes control and data flow dynamic information to achieve path coverage testing using GA. In addition, the work also tackled the detection of infeasible paths by monitoring the progress of evolutionary search. The fitness function was formulated by number of coincidence branches and the normalized branch predicate value at which the actual executed path starts to deviate from the target path. Six small test programs were used to validate the approach, with 10 repetitions each to minimize random variations. Two execution modes were used, i.e. one with initialized population and the other with a random initial population. The experiment results were promising.

In 2003, Hermadi and Ahmed [21] presented evolutionary test data generation for path testing using multiple paths. Prior to this work, almost all of the evolutionary test data generators only sought to cover a single target path at a time. The fitness function used the number of matching branches and branch predicate values using Korel's fitness function [9]. It also considered path traversal techniques, neighbourhood influence, weighting, and normalization. Three small programs were used to validate the approach: minimum-maximum finder, triangle classifier, and a combination of both of them. Results were more effective and efficient by tackling multiple paths at a time.

In 2008, Ahmed and Hermadi [23] extended Hermadi and Ahmed's work of 2003 [21]. The extensions were adding a rewarding scheme and using a more efficient test data generator. A total of 32 fitness function variations were tested empirically and analysed to determine which was the best. There were 7 test programs used in the experiments. The results demonstrated that the approach was better compared to other existing work.

In the same year, Chen and Zhong [24] developed a multi-population genetic algorithm for path testing. This work has been improving GA-based path testing as described in Section 2.2. The work reported that the proposed approach outperformed a simple genetic algorithm based approach, using the triangle classifier as the test function. Similar to our approach, this paper also targets finding the test data to cover path conditions of the most difficult path to be covered in test function. As it approached the parallel processing, test data generating time is better than traditional GA, however the number of test data generation is still high (requires 21,073 test data by average).

In [25], Srivastava P.R and Kim T have presented a method for optimizing software testing efficiency by identifying the most critical path clusters in a program. The software under test is converted into a CFG. Weights are assigned to the edges of the CFG by applying 80-20 rule. 80 percentage of weight of incoming credit is given to loops and branches and the remaining 20 percentage of incoming credit is given to the edges in sequential path. The summation of weights along the edges comprising a path determines criticality of path. Higher the summation more critical is path and therefore must be tested before other paths. In this way by identifying most critical paths that must be tested first, testing efficiency is increased.

**4. Proposed approach**



**Fig. 1.** Flow chart of our proposed approach

In order to be able to generate test data which can cover the paths having the lowest coverable probability, we propose 2 steps approach as in the above flow chart:

Step 1: Determine the path conditions of difficult paths to be covered.

Step 2: Execute GA to automatically generate test cases.

**Representation**

Depend on the type of input parameters of test function, GA uses a real or integer vector as a chromosome x = (x1, x2,…, xn) to represent values of the input variables. The length of the vector depends on the required precision and the domain length for each input variable. Besides, it is necessary to perform the target paths which were generated from CFG.

**Initial population**

At first, it needs to identify a fixed popsize number, which is the number of chromosome in a population. After that is the initialization of a randome value for all chromosomes in this initial population.

**Evaluation function**

The algorithm evaluates each test case by executing the program with it as input, and recording the executed path in the program that are covered by this test case. The fitness value eval(xi)for each chromosome xi (i = 1,…, popsize) is calculated as follows:

if (executed path tập target paths)

{

eval(xi) = 0;

remove the executed path from the target paths;

}

else

{

eval(xi) = *;*

}

The distance between 2 different paths of given test function is defined as below:

distance(path i, path j) =

**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. If none of the members of the current population was effective, all the members of current population are considered the parents of the new population. In the selection process the GA uses random selection method whose 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 ≤ popsize. The parents are selected as follows:

Isolate the effective members and number them from 1 to l;

For i = 1 to popsize do

{

Generate an random integer number j from the range [0..l];

Select chromosome xj from the effective members;

}

***Recombination***

In the recombination phase, we use two operators, crossover and mutation, which are the key to the power of GA. These operators create new individuals from the selected parents to form a new population.

*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 acrossover probability *pc*. We proceed inthe following way:

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 we mate selected chromosomes randomly: For each pair of coupled chromosomes we generate a random integer number *pos* from the range [1..*m*-1] (*m* is the number of bit in a chromosome). The number *pos* indicates the position of the crossing point. Two chromosomes (*b1*…*bposbpos+1*…*bm*) and (*c1*…*cposcpos+1*…*cm*) are replaced by a pair of their offspring (*b1*…*bposcpos+1*…*cm*) and (*c1*…*cposbpos+1*…*bm*).

*Mutation*: It is performed on a bit-by-bit basis. Mutation always operates after the crossover operator, and flips each bit with the pre-determined probability. The probability of mutation is *pm*. Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation, i.e., change from 0 to 1 or vice versa. So we proceed in the following way:

For each chromosome in the current (i.e. after crossover) population and for each bit within the chromosome:

• Generate a random (float) number *r* from the range [0..1];

• If *r* < *pm* then mutate the bit.

***Adjustment***

After executing the recombination, in order to generate test cases which can cover the most difficult paths, we need to adjust the values of each chromosome in the population.

The adjustment will be executed as follows:

• Browse the list of path conditions of the difficult paths to be covered.

• If any value of the chromosome can nearly satisfy this path condition, adjust it so that it can satisfy such condition.

**5. Case study**

Consider the Tritype function (simple triangle classifier) [18, 22] which determines whether three given numbers that represent three lengths on a plane form a scalene, isosceles, equilateral, or not a triangle.

int Tritype(int a, int b, int c)

{

inttrityp = 0;

if ((a + b > c) && (b + c > a) && (c + a > b))

{

if ((a != b) && (b != c) && (c != a))

trityp = 1; // Scalene

else

if (((a == b) && (b != c)) || ((b == c) && (c != a)) ||

((c == a) && (a != b)))

trityp = 2; // Isosceles

else

trityp = 3; // Equilateral

}

else

trityp = -1; // Not a triangle

return trityp;

}



**Fig. 2.** The CFG of above Tritype function

This CFG consists of 4 paths as below:

{[1,F]} // path 1 (Not Triangle)

{[1,T], [2,F], [3,F]} // path 2 (Equilateral)

{[1,T], [2,F], [3,T]} // path 3 (Isosceles)

{[1,T], [2,T]} // path 4 (Scalence)

Assuming that the input parameters a, b, c take the positive integer values in the range of a, b, c ∈ [1, 1000]. Each parameter can take 1000 values, so the space of input parameter will be 10003. There are 1000 triagles having equal sides which are (1, 1, 1), (2, 2, 2)…(1000, 1000, 1000). Therefore, the probability that one test data can cover path 2 (Equilateral) is 1000/10003 = 0.000001.

Also through a similar calculation, we will have a table showing the generation probability of path-coverage data as follows:

|  |  |
| --- | --- |
| Classification | Probability |
| Path 4 (Scalence) | 0.4985015 |
| Path 3 (Isosceles) | 0.001998 |
| Path 2 (Equilateral) | 0.000001 |
| Path 1 (Not Triangle) | 0.4994995 |

From the experiments, we found out that if only using the traditional GA to generate data for function tA2008, we can not generate the test data for path 2 (Equilateral), which is the most difficult path to be covered.

GA started....

Path 4: a = 737 b = 241 c = 615

Objective call: 1

Path 1: a = 389 b = 69 c = 133

Objective call: 3

Path 3: a = 764 b = 764 c = 819

Objective call: 81

Objective call: 10000

GA completed./.

Now we apply our proposal for this test function. At first determine the path conditions of difficult paths to be covered.

|  |  |  |
| --- | --- | --- |
| Classification | Probability | Path condition |
| Path 4 (Scalence) | 0.4985015 |  |
| Path 3 (Isosceles) | 0.001998 | a == b || b == c || c == a |
| Path 2 (Equilateral) | 0.000001 | a == b == c |
| Path 1 (Not Triangle) | 0.4994995 |  |

Then execute GA to automatically generate test cases. With the above Tritype function, as input parameter was a set of three numbers a,b,c, each generated test case will be perofrmed by a chromosome, which is a vector x = (x1, x2, x3). The initial target paths will also include 4 paths of the test function.

Then we calculate fitness function for this Tritype function, for example thì ta có khoảng cách giữa các path như sau:

distance(path 1, path 2) = 3

distance(path 2, path 3) = 0.3333

Return to Tritype function, with the above identified path condition, adjustment is executed as follows:

for i = 1 to popsize do

{

if ((abs(chromosome[i].x1 - chromosome[i].x2) <= epsilon) &&

(abs(chromosome[i].x1 - chromosome[i].x3) <= epsilon))

{

// Adjustment for path 2 (Equilateral)

chromosome[i].x2 = chromosome[i].x1;

chromosome[i].x3 = chromosome[i].x1;

}

}

With these adjustments, the returned results after executing GA are as follows:

GA started....

Path 4: a = 737 b = 241 c = 615

Objective call: 1

Path 1: a = 389 b = 69 c = 133

Objective call: 3

Path 3: a = 764 b = 764 c = 819

Objective call: 81

Path 2: a = 900 b = 900 c = 900

Objective call: 3198

GA completed./.

**6. Experimental results and discussion**

In addition to the test Tritype function mentioned above, in this paper, we also executed the experiments for the following test functions:

- Tritype function[22] accepts three numbers representing sides of a triangle, classifies its type, and computes its area.

- Triangle function [22] classifies three numbers representing triangle side lengths into five type triangles: scalene, isosceles, right, iso-right, or equilateral.

- QuadEq2 function finds all roots of a quadratic equation with 3 coefficients a, b and c as input parameters.

The static analysis results of each test function from the respective target paths file and path conditions are as followed:

|  |  |  |  |
| --- | --- | --- | --- |
| Function | PathID | Target paths | Path condition |
| ttB2002 | 1 | {[1,F], [2,F], [3,F], [4,F], [5,F]} | b ==c |
| (Tritype) [22] | 2 | {[1,F], [2,F], [3,F], [4,F], [5,T]} | a == b |
|  | 3 | {[1,F], [2,F], [3,F], [4,T]} | a == b == c |
|  | 4 | {[1,F], [2,F], [3,T], [6,F], [7,F]} |  |
|  | 5 | {[1,F], [2,F], [3,T], [6,F], [7,T]} |  |
|  | 6 | {[1,F], [2,F], [3,T], [6,T]} | a\*a == b\*b + c\*c |
|  | 7 | {[1,F], [2,T]} |  |
|  | 8 | {[1,T]} |  |
| tM2004 | 1 | {[1,F], [2,F], [3,F]} |  |
| (Triangle)[22] | 2 | {[1,F], [2,F], [3,T]} | a\*a == b\*b + c\*c |
|  | 3 | {[1,F], [2,T], [3,F]} | b = c |
|  | 4 | {[1,F], [2,T], [3,T]} | (a\*a == b\*b + c\*c)&&(b == c) |
|  | 5 | {[1,T], [2,F], [3,F]} | a == b |
|  | 6 | {[1,T], [2,F], [3,T]} |  |
|  | 7 | {[1,T], [2,T], [3,F]} | a == b == c |
|  | 8 | {[1,T], [2,T], [3,T]} |  |
| tA2008[18, 22] | 1 | {[1,F]} |  |
|  | 2 | {[1,T], [2,F], [3,F]} | a == b |
|  | 3 | {[1,T], [2,F], [3,T]} | a == b == c |
|  | 4 | {[1,T], [2,T]} |  |
| QuadEq2 | 1 | {[1,F], [2,F], [3,F]} |  |
|  | 2 | {[1,F], [2,F], [3,T]} | b\*b == 4\*a\*c |
|  | 3 | {[1,F], [2,T]} |  |
|  | 4 | {[1,T], [4,F]} | a == 0 |
|  | 5 | {[1,T], [4,T]} | (a == 0)&&(b == 0) |

Settings of traditional GA and improved GA are as following:

• Length of the chromosome: 3

• Selection method: random

• Two-point crossover probability (pc): 0.5

• Mutation probability (pm): 0.1

• Stopping criteria: target paths list is empty

Each test function still requires other parameters below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test function | Type | Range | Max gen | Popsize | Max run | Epsilon | Max aval. func. call |
| ttB2002 | integer | [1, 400] | 150 | 250 | 1 | 10 | 37,500 |
| tM2004 | double | [0, 50] | 100 | 250 | 1 | 1 | 25,000 |
| tA2008 | integer | [1, 1000] | 100 | 100 | 1 | 10 | 10,000 |
| QuadEq2 | double | [-50, 50] | 150 | 250 | 5 | 1 | 187,500 |

• Type: type of input parameters

• Range: range of input parameters

• Max gen: Maximum population generation for each time to run GA

• Popsize: Number of chromosome for each population

• Max run: Maximum runtime of GA

• Epsilon: critical value to adjust chromosome

• Max aval. func. call: The maximum number of times the evaluation call

Results from performing traditional GA and improved GA are as in the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test function | Target paths | Traditional GA | | Improved GA | |
| Path ID | Eval. function call | Path ID | Eval. function call |
| ttB2002 | 8 | 8 | 1 | 8 | 1 |
| (Tritype) [22] |  | 7 | 4 | 7 | 4 |
|  |  | 5 | 10 | 5 | 10 |
|  |  | 4 | 16 | 4 | 16 |
|  |  | 2 | 266 | 2 | 252 |
|  |  | 1 | 2747 | 1 | 988 |
|  |  | \* | 37500 | 3 | 4218 |
|  |  | \* | 37500 | 6 | 19322 |
| tM2004 | 8 | 1 | 1 | 1 | 1 |
| (Triangle)[22] |  | \* | 25000 | 5 | 266 |
|  |  | \* | 25000 | 7 | 3328 |
|  |  | \* | 25000 | 3 | 4973 |
|  |  | \* | 25000 | 2 | 6968 |
|  |  | \* | 25000 | 4 | 21034 |
| tA2008[18, 22] | 4 | 3 | 1 | 3 | 1 |
|  |  | 0 | 3 | 0 | 3 |
|  |  | 2 | 81 | 2 | 81 |
|  |  | \* | 10000 | 1 | 3198 |
| QuadEq2 | 5 | 3 | 1 | 3 | 1 |
|  |  | 1 | 3 | 1 | 3 |
|  |  | \* | 187500 | 5 | 222 |
|  |  | \* | 187500 | 2 | 43361 |
|  |  | \* | 187500 | 4 | 75508 |

From this result table, it can be seen that even if the maximum times of evaluation function was performed, there are still paths which traditional GA can not generate the test data to cover, while improved GA can.

Comparing to [18, 24], the number of times to perform evaluation function in order to generate test case which can cover path 2 (Equilateral) is lower (3198 times, comapring to 10000 times [18] or 21073 times [24]), proving that our proposed GA is more effective.

**7. Conclusion**

In software development life cycle, software testing is one of the critical phases. So generation of test data automatically is a key step which has a great influence on code coverage in software testing. In this paper, we have improve the GA in order to generate test data automatically for feasible target paths.

Our approaching method is to combine the static analysis in order to find path conditions of difficult path to be covered in these functions, then adjust the procedure of generating the new population in GA in order to generate test cases which can cover these paths.

The experimental results on these test functions shows that improved GA can generate test data can cover path having path conditions which can not be covered by test data generated from normal GA.

Besides, comparing to the current methods [18, 24], our proposal is more effective as it can more quickly generate test data which can cover paths that other methods can not.

**8. References**

[1] B. Antonia, "Software Testing Research: Achievements, Challenges, Dreams," in 2007 Future of Software Engineering: IEEE Computer Society, 2007.

[2] G. J. Myers, The Art of Software Testing, 2nd edition: John Wiley & Sons Inc, 2004.

[3] B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, Inc, New York, NY, USA, 1982.

[4] E. J. Weyuker, The applicability of program schema results to programs, International Journal of Parallel Programming, vol. 8, 387-403, 1979.

[5] C. K. James, A new approach to program testing, in Proceedings of the international conference on Reliable software Los Angeles, California: ACM, 1975.

[6] T. Y. Chen, T. H. Tse, and Z. Zhiquan, Semiproving: an integrated method based on global symbolic evaluation and metamorphic testing, in Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis Roma, Italy: ACM, 2002.

[7] S. Nguyen Tran and D. Yves, Consistency techniques for interprocedural test data generation," ACM SIGSOFT Software Engineering Notes, vol. 28, 108-117, 2003.

[8] G. M. C C Michael, M Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering, vol. 27, 1085-1110, 2001

[9] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering, vol. 16, 870-879, 1990.

[10] B. Korel, Dynamic method for software test data generation, Software Testing, Verification & Reliability, vol. 2, 203-213, 1992.

[11] B. Korel, Automated test data generation for programs with procedures, in Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis San Diego, California, United States: ACM, 1996.

[12] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, Application of genetic algorithms to software testing (Application des algorithmes genetiques au test des logiciels), in Proceedings of 5th International Conference on Software Engineering and its Applications Toulouse, France, 625-636, 1992.

[13] J. Wegener, A. Baresel, and H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology, vol. 43, 841-854, 2001.

[14] J. Wegener, B. Kerstin, and P. Hartmut, Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing, in Proceedings of the Genetic and Evolutionary Computation Conference: Morgan Kaufmann Publishers Inc., 2002.

[15] S. Levin and A. Yehudai, "Evolutionary Testing: A Case Study," in Hardware and Software, Verification and Testing, 155-165, 2007.

[16] W. Joachim, Andr, Baresel, and S. Harmen, Suitability of Evolutionary Algorithms for Evolutionary Testing, in Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment: IEEE Computer Society, 2002.

[17] J. H. Holland, Adaptation in Nature and Artificial Systems, Addison-Wesley, Reading, MA, 1975.

[18] Jin-Cherng Lin and Pu-Lin Yeh, Using genetic algorithms for test case generation in path testing, In Proceedings of the 9th Asian Test Symposium 2000 (ATS '00), 241-246, December 2000.

[19] Bryan F. Jones, Harmen-Hinrich Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms, Software Engineering, 11(5):299-306, September 1996.

[20] Paulo Marcos Siqueira Bueno and Mario Jino, Automatic test datageneration for program paths using genetic algorithms, International Journal of Software Engineering & Knowledge Engineering (IJSEKE), 12(6):691-709, 2002.

[21] Irman Hermadi and Moataz A. Ahmed, Genetic Algorithm based test data generator, In Proceedings of the 2003 Congress on Evolutionary Computation (CEC), volume 1, pages 85-91, December 2003.

[22] I. Hermadi, C. Lokan, R. Sarker, Dynamic stopping criteria for search-based test data generation for path testing, Information and Software Technology, 56 (4):395-407, April 2014.

[23] Moataz A. Ahmed and Irman Hermadi, GA-based Multiple Paths Test Data Generator, Computers & Operations Research, 35:3107-3124, October 2008.

[24] Yong Chen and Yong Zhong, Automatic path-oriented test data generation using a multi-population genetic algorithm, In Proceedings of the 4th International Conference on Natural Computation, 2008 (ICNC'08), volume 1, pages 566-570, October 2008.

[25] Srivastava P. R and Kim T, Application of Genetic Algorithm in Software Testing, International Journal of Software Engineering and Its Applications, 3(4), 87-96, 2009.