A Technique for Generating Test Data using Genetic Algorithms

Authors Name/s per 1st Affiliation *(Author)*
line 1 (of *Affiliation*): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: e-mail address if desire

Authors Name/s per 2nd Affiliation *(Author)*
line 1 (of *Affiliation*): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: e-mail address if desired

***Abstract*—Automatic path coverage test data generation is an undecidable problem and genetic algorithm (GA) has been used to test data generation since 2000. This paper presents a method for optimizing GA efficiency by identifying the most critical path clusters in a test function. We do this by using the static program analysis to find all the paths having the path conditions with low probability in generating coverage data, then basing on these path conditions to adjust the procedure of generating new populations in GA. The proposed approach is also applied some test functions. Experimental results show that improved GA which can generate suitable test data have higher path coverage than the traditional GA.**

***Keywords*—generic algorithm; path coverage testing; automatic test data generation**

I. 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 cannot 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 cannot 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 paths in a test function. It combines 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 these difficult paths, adjusts the procedure of generating new populations in GA.

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

II. BACKGROUND

This section describes to the two content is the theoretical background for the proposal of this paper, path coverage testing and genetic algorithm.

A. Path coverage testing

The objective of path coverage 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 coverage 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.

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). 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) 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.

B. *Generic algorithm*

The basic concepts of genetic algorithm (GA) were developed by Holland [17]. GA is commonly applied to a variety of problems involving search and optimization. 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 generates 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 to the 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, crossover and mutation.

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).

III. RELATED WORK

The path coverage literature using GA 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, neighborhood 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 their 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 analyzed to determine which the best was. 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.

IV. PROPOSED APPROACH

This section describes details of our proposed approach, to test data generation using GA. In order to generate test data which can cover the paths having the lowest coverable probability, we propose 2 step approaches as in the above flow chart:

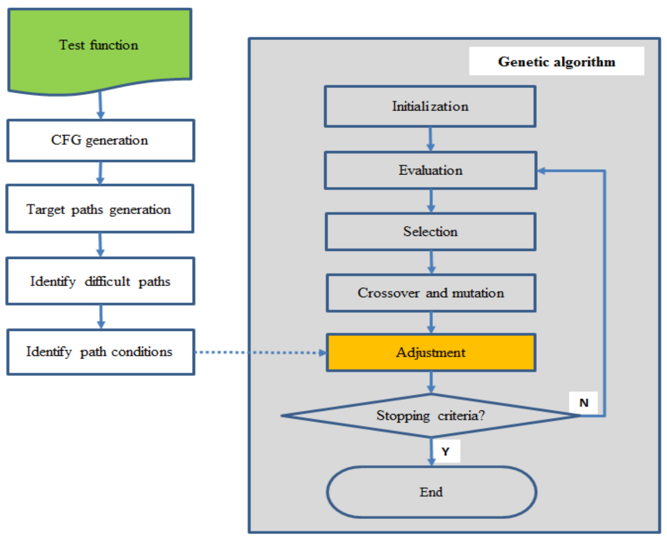

Fig. 1. Flow chart of our proposed approach

A. *Step 1: Perform static program analysis*

The goal of this step is from a given test function, identify the path condition of the difficult paths, to serve as conditions of adjustment for GA in the next step. Step 1 is executed by below procedures:

1) CFG generation: Currently there are many tools to construct a CFG from a given function, but we choose Dr. Garbage plugins[1] because of its popularity.

2) Target paths generation: Traverse the CFG to obtain all feasible test paths using the depth-first search and backtracking algorithm as follows. At each decision, a path from the initial vertex to this decision is checked whether it is feasible or not. If this path is feasible, it means that all test paths contain the path may be feasible. All branches from the decision are therefore continued traversing. If it is not, the path is infeasible so that all test paths traverse this path are always infeasible. This result makes the traverse process from this decision terminate. All feasbile paths which were obtained from this traversal will stored in target paths.

3) Identify difficult paths: Identify difficult paths basing on the probability of generating test datas which cover each path of the test function. Paths having the lower probability of generating test data are difficult ones.

4) Identify path condition: For each difficult path, identify its path conditions. Those path conditions will be used as conditions to adjust the procedure of generating new population in GA.

B. *Step 2: Execute GA*

To automatically generate test cases, using GA with below procedures:

1) Representation: Depend on the type of input parameters of test function, GA uses a real or integer vector as a chromosome $x = (x_1, x_2, \ldots, x_n)$ 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.

2) Initial population: At first, it needs to identify a fixed popsize number is the number of chromosome in a population. Then initialize randome values for all chromosomes in the first population.

3) 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 $eval(x_i)$ for each chromosome x_i ($i = 1, \ldots, popsize$) is calculated as follows:

```
if (executed path target paths)
{
   eval($x_i$) = 0;
   remove this path from the target paths;
}
else
{
   eval($x_i$)= $\frac{Total\ distance\ from\ executed\ path\ to\ each\ target\ path}{Total\ paths\ in\ target\ paths}$
}
```

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

$$distance(path\ i, path\ j) = \frac{Number\ of\ different\ vertices}{Number\ of\ vertices\ in\ the\ shorter\ path}$$

4) Selection: A selection scheme is applied to determine how individuals are chosen for mating based on their fitness. Fitness can be defined as a capability of an individual to survive and reproduce in an environment. Selection generates

[1] http://www.drgarbage.com

the new population from the old one, thus starting a new generation. Each chromosome is evaluated in present generation to determine its fitness value. This fitness value is used to select the better chromosomes from the population for the next generation.

5) Crossover: After selection, the crossover operation is applied to the selected chromosomes. It involves swapping of values of vector x = (x_1, x_2,…, x_n) between two chromosomes. This process is repeated with different parent chromosomes until the next generation has enough chromosomes. After crossover, the mutation operator is applied to a randomly selected subset of the population.

6) Mutation: Mutation alters chromosomes in small ways to introduce new good traits. It is applied to bring diversity in the population.

7) Adjustment: After executing the mutation, 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 this value to satisfy such condition.

V. EXPERIMENTAL RESULTS

In this section, we illustrate our proposal through a common sample, and experiment with different test functions to demonstrate the improvement of our methods.

A. A triangle classifier

Triangle classifier is a widely used test function in the research area of software testing. Consider the test function Tritype tA2008 [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)
{
  int trityp = 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;
}
```

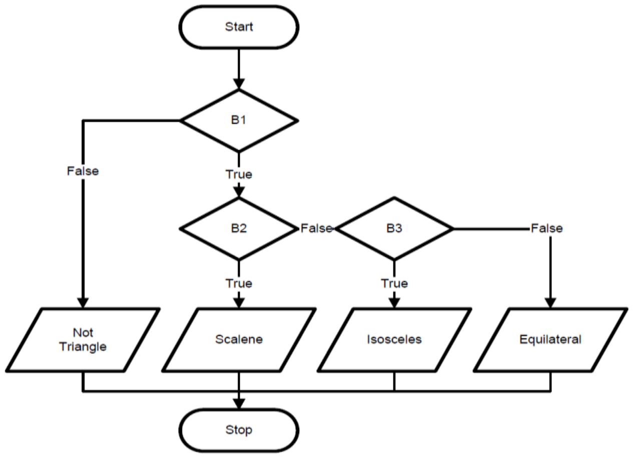

Fig. 2. The CFG of above Tritype function

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 1000^3. There are 1000 triangles 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 $1000/1000^3 = 0.000001$. Also through a similar calculation, we will have a table showing the generation probability of path coverage data as follows:

TABLE I. PATH COVERAGE'S PROBABILITY OF TRITYPE FUNCTION

Classification	**Probability**
Path 4 (Scalene)	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 Tritype function, we cannot generate the test data for path 2 (Equilateral), which is the most difficult path to be covered.

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

TABLE II. PATH CONDITION OF TRITYPE FUNCTION

Classification	Probability	Path condition
Path 4 (Scalene)	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 perform 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 performed by a chromosome, which is a vector x = (x_1, x_2, x_3). The initial target paths list also includes 4 paths of the test function.

Now we have to calculate all distances between each path in given test function. Using above proposed formula, for example, we have some distance as below:

distance(path 1, path 2) = 3
distance(path 2, path 3) = 0.3333

At last we adjust each value of the chromosome for the above identified path condition of difficult path. 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 TABLE V.

B. Experiment with other test functions

In addition to the test function Tritype tA2008 mentioned above, in this paper, we also executed the experiments for the following test functions:

- Tritype function ttB2002 [22] accepts three numbers representing sides of a triangle, classifies its type, and computes its area.
- Triangle function tM2004 [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:

TABLE III. PATH CONDITIONS OF OTHER TEST FUNCTIONS

Test function	Path ID	Target paths	Path conditions
ttB2002 (Tritype) [22]	1	[1,F],[2,F],[3,F],[4,F],[5,F]	b==c
	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 Triangle [22]	1	[1,F],[2,F],[3,F]	
	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]	infeasible path
	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==c
	3	[1,T],[2,F],[3,T]	a==b
	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

Parameter settings of traditional GA and improved GA are as following:

- Length of the chromosome: 3
- Selection method: random
- Two-point crossover probability (p_c): 0.5
- Mutation probability (p_m): 0.1
- Stopping criteria: if target paths list is empty

Each test function still requires other parameters below:

TABLE IV. GA PARAMETERS SETTING FOR EACH TEST FUNCTION

Test function	Type	Range	Max gen	Popsize
ttB2002	integer	[1, 400]	150	250
tM2004	double	[0, 50]	100	250
tA2008	integer	[1, 1000]	100	100
QuadEq2	double	[-50, 50]	150	250

Test function	Max run	Epsilon	Max eval. func. call
ttB2002	1	10	37500
tM2004	1	1	25000
tA2008	1	10	10000
QuadEq2	5	1	187500

- Type: type of input variables
- Range: range of input variables
- 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 eval. func. call: maximum number of evaluation function calls

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

TABLE V. COMPARING TRADITIONAL GA AND IMPROVED GA

Test function	**Target paths**	**Path ID**	**Evaluation function calls**	
			Traditional GA	**Improved GA**
ttB2002 (Tritype) [22]	8	1	2747	988
		2	266	252
		3	cannot cover	4218
		4	16	16
		5	10	10
		6	cannot cover	19322
		7	4	4
		8	1	1
tM2004 Triangle [22]	8	1	1	1
		2	cannot cover	6968
		3	cannot cover	4973
		4	cannot cover	16
		5	cannot cover	266
		7	cannot cover	3328
tA2008 [18, 22]	4	1	3	3
		2	cannot cover	3198
		3	81	81
		4	1	1
QuadEq2	5	1	3	3
		2	cannot cover	43361
		3	1	1
		4	cannot cover	75508
		5	cannot cover	222

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 cannot 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, comparing to 10000 times [18] or 21073 times [24]), proving that our proposed GA is more effective.

VI. 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 improved 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 test functions, and 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 cannot 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 cannot.

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.