AUTOMATED GENERATION AND INFEASIBLE DETECTION OF TEST PATHS FOR DATA FLOW TESTING USING GENETIC ALGORITHM

***Shaukat Ali Khan and **Aamer Nadeem**

Center for Software DependabilityMohammad Ali Jinnah University (MAJU),Islamabad, Pakistan

*shaukatali74@gmail.com, **anadeem@jinnah.edu.pk

ABSTRACT— *Data flow testing is a code based testing technique that uses the dataflow relations in a program for the guidance of test case selection. Evolutionary testing uses the optimizing search techniques for the generation and selection of test data. This paper discusses the unique approach for data flow testing by applying evolutionary algorithms for the automatic generation of test paths and infeasible path detection by using data flow relations in a program. Our approach starts with a random initial population of test paths and then based on the selected testing criteria new paths are generated by applying a genetic algorithm. A fitness function evaluates each chromosome (path) based on the selected data flow testing criteria and computes its fitness. We have applied one point crossover and mutation operators for the generation of new population. We have also identified infeasible paths using genetic algorithm by executing them with appropriate inputs. The tool ETODF (evolutionary testing of data flow) has been implemented in Java for proof of concept. In experiments with this tool, our approach has very promising results.*

Keywords- Data Flow Testing, Genetic Algorithm, Test Paths, Coverage Criteria, Mutation, Cross over

I. INTRODUCTION

Software testing is the process of executing a program with the intent of finding errors [1]. It is an important and expensive phase of software development life cycle. The purpose of software testing is to provide assurance that the software system is bug free and conforms to its specifications. Software test case design is the most crucial part of software testing. Evolutionary algorithms have been applied to software testing for automated generation of test cases and test data. The application of evolutionary algorithms [4] to software testing for generation of test data is known as evolutionary testing. They include genetic algorithms, particle swarm optimization, ant colony model, etc. [5]. Evolutionary algorithms have been applied in software testing at various levels from procedural to object oriented programming. Application of evolutionary testing is mostly on control flow testing. In this paper, we have extended our previous work [6] that uses the genetic algorithm for the generation for test paths for data flow testing at the unit level. In this paper, we have applied evolutionary approaches to data flow testing for the generation and infeasible detection of test paths. The approach has been implemented in Java language by a prototype tool called E-ETODF for validation. In experiments with this prototype, our approach has much better results as compared to random testing.

The major contributions of this paper are as follows:

- Extension of our previous work [6] for test data generation for data flow testing by adding a new approach for infeasible path detection.
- Extension of our previous implemented tool (ETODF) [24] for test data generation for data flow testing by adding a new component for infeasible path detection.
- GA has been applied for test path generation and infeasible detection of test paths.
- We have performed different experiments on programs ranging from small to medium level for test path generation and infeasible path detection using our proposed approaches.
- Results of the experiments are very encouraging and promising.

The rest of the paper is organized as follows: Section II describes the proposed approach test path generation and infeasible path detection using genetic algorithm for data flow testing and section III represents the tool for ETODF. Section IV represents the experimental results using ETODF Section represents the related work. Section VI concludes the paper and presents the future work.

II. PROPOSED APPROACH

Our approach has two phases; in first phase test paths are generated using genetic algorithm [6] while second phase involves identifying the infeasible paths from the selected path. Infeasible paths are those paths for which there are no input to execute the path. For identification of infeasible paths, we will execute all the paths with some test data. If the execution of path is not possible with input data, then the path is said to be infeasible.

A. Proposed Approach for Test Path Generation

Figure 1 depicts the high level flow of our proposed approach for test path generation. We have proposed a novel approach [6] for the data flow testing of programs using evolutionary approaches at unit level. We have applied genetic algorithm with one point crossover and mutation for generating test paths in our proposed approach. Genetic algorithm is a well-known evolutionary approach successfully applied to a variety of problems for optimization [11,12].

We have applied genetic algorithm with one point crossover and mutation for generating test paths in our proposed approach. Our approach starts with a random population of test paths for a given program and then genetic algorithm has been applied for generating new population. Our proposed approach takes data flow graph of a program, variable whose values to be tested, dynamic data flow coverage criteria , number of iterations , if specified otherwise it uses the default value, and coverage requirement in percent as input. This approach is explained in detail in [6].

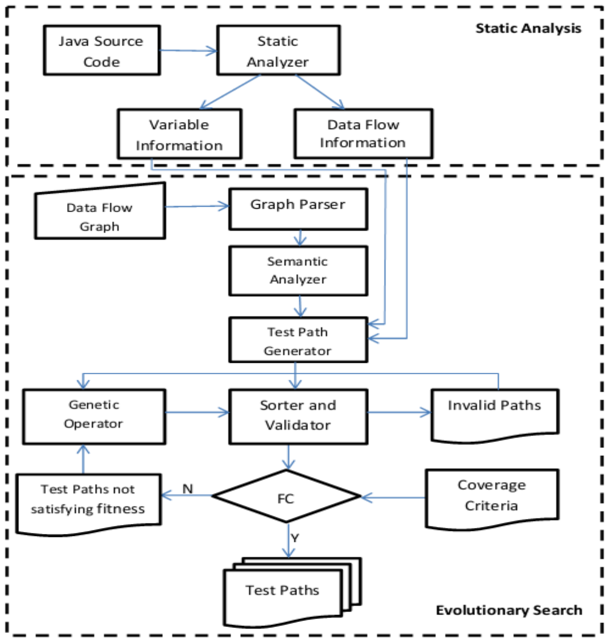

Figure 1. Proposed Approach Flow for Test Path Generation [6]

B. Automated Infeasible Path Detection

Our approach has two phases; in first phase test paths are generated using genetic algorithm while second phase involves identifying the infeasible paths from the generated paths. Infeasible paths are those paths for which there is no input to execute the paths. For identification of infeasible paths, we execute all the paths with some test data. If the execution of path is not possible with input data, then the path is said to be infeasible. Program path is infeasible if there exists no input that executes the path.

In figure 2 there is a sample program showing infeasible statement at line number 7. The main method calls the method ‘an’ only if required argument is positive. In method ‘a’ argument is again checked for negative value in statement at line 6. The statement at line number 7 is not reachable as a method ‘a’ accepts only positive number. In figure 2, the statement at line number 7 is not reachable. If some test path contains astatement at line number 7 shown in figure 2 then this path is infeasible as there is no input data

for execution of statement at line number 7 of figure 2. In our proposed approach we have identified infeasible paths using genetic algorithm.

We have identified infeasible paths using genetic algorithm, we executed each path for a fixed number of iterations, as configured by input, if we found input data for path within fixed number of iterations, configured as input, then path is said to be feasible otherwise path is infeasible. We identified variables in program by static analysis of the code and then input values of the program variables are generated

```
public class Test {
    1.  public void a(int x ){
    2.    if (x<5){
    3.     System.out.println(x);}
    4.   else if(x<10){
    5.    System.out.println(x);}
    6.   else if(x<0){
    7.    System.ou.println("ERROR");}
    8.   }
    9.  public static void main(String args[]){
    10.   int b=10;
    11.   if(b<0){
    12.   System.out.println("Error    Negative
        ");}
    13.  else{
    14.   a(b);}
    15.  }
    16.  }
```

Figure 2. Sample Code Showing Infeasible Statement

Randomly using the input boundaries of the variables. The code is instrumented for generating execution trace of the program. After the iteration has been completed, the executed path is compared with target path, if executed and target path is same, then the input data for this path is preserved and path is said to be feasible.Figure 3 depicts the high level flow of our proposed approach for infeasible path detection. We have used java source code in our proposed approach for experiments. In the first step, source code is analyzed by static analyzer and extracts the variable information and control dependency information from the source code.

The Instrumentor adds the additional lines of code in the source code so that an execution trace can be generated after each execution. The test driver takes the instrumented code and test data, generated by test data generator using variable information extracted in static analysis, for execution of code.

Figure 3 depicts the high level flow of our proposed approach for infeasible path detection. We have used java source code in our proposed approach for experiments. In the first step, source code is analyzed by static analyzer and extracts the variable information and control dependency information from the source code.

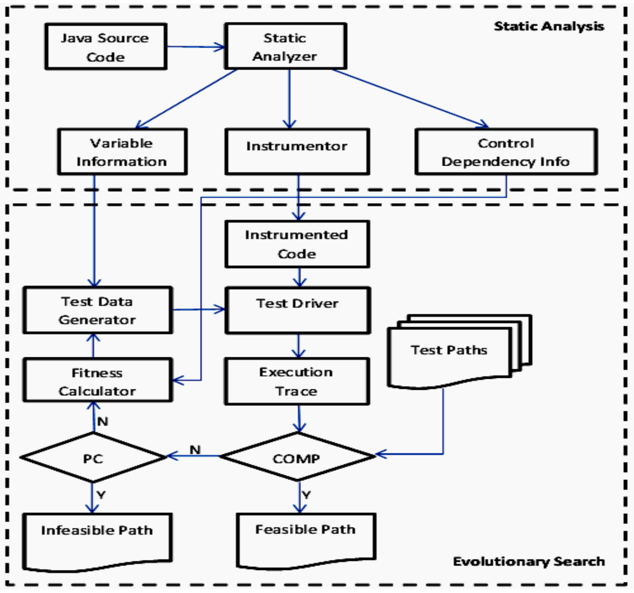

Figure 3. Proposed Approach Flow for Infeasible Path Detection

The Instrumentor adds the additional lines of code in the source code so that an execution trace can be generated after each execution. The test driver takes the instrumented code and test data, generated by test data generator using variable information extracted in static analysis, for execution of code. Test driver executes the instrumented code and information is preserved as execution traces. The Comparator compares the execution trace of the executed path with target path, if same path is followed in execution trace as of target path, then path is called a feasible path and is stored in file as feasible path. If the path followed in execution trace is different from the target path, then population counter (PC) checks, if PC is equal to the maximum number of iterations, then path is said to be infeasible and is stored in the file as infeasible path.

The population counter checks the number of iterations, if the number of iterations is less than the maximum number of iterations, then fitness calculator calculates the fitness value of data using previous data and control dependency information and new data is generated by the test data generator to move into the next iteration. In this way the program is executed, target and executed paths are compared and the paths generated in the first phase of our approach are classified as either feasible or infeasible.

IV. THE TOOL E-ETODF

The proposed approach has been implemented in a prototype tool called E-ETODF (Extended Evolutionary Testing of Data Flow) [25] in Java. The prototype tool uses the genetic algorithm for the generation of test path with single point crossover and mutation for test path generation. The infeasible path detection is through genetic algorithm by mutating the input data in every iteration by calculating the branch distance and approximation level [10]. In this paper, we have extended our tool by implementing a new component for infeasible path detection. The high level architecture of the tool is depicted in Figure 4. ETODF has the following major components:

- Test Path Generator
- Infeasible Path Detector

4.1. Test Path Generator

E-ETODF [24] takes data flow graph as input and stores the graph after analyzing and applying semantic on each node. This approach is explained in detail in [24].

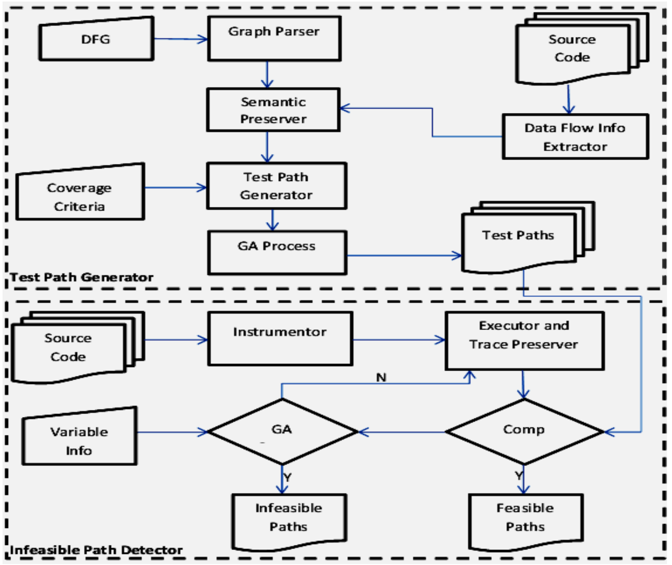

Figure 4. High Level Architecture of ETODF

The test path generator is an important component of ETODF which first generates the test path randomly and on the second iteration it uses genetic algorithms for test path generation. The test path generator uses the genetic algorithm with one point crossover and mutation for the generation of new population. The test paths are sorted by the sorter component of the ETODF and then each path is validated using the semantics of the graph for validness. In this step the invalid paths are sorted out from the test paths. The valid paths are passed to the fitness calculator which calculates the fitness of each path using the data flow coverage criteria. The fitness calculator takes valid paths from Validator components and data flow coverage criteria from user as input and calculate the fitness of the path according to the data flow coverage criteria. After evaluation, path that satisfies the coverage criteria valid are added in the global list of paths while those which do not satisfy the coverage criteria remains in the population and used in the recombination and mutation for the next iteration.

4.2. Infeasible Path Detector

Infeasible path detector detects the infeasible path using genetic algorithm. Each path is executed for a fixed number of times with test data, if the execution of the path is possible, then path is said to be feasible otherwise path is infeasible. The infeasible path detector instruments the code by using Instrumentor component. The instrumented code and test data is provided to the path executor and trace preserver. The path executor executes the path and trace preserver logs the trace. The comparator compares the trace with test paths already generated by test path generator. If executed path and target path are same, the path is said to be feasible and stored infeasible paths. If executed and target paths are different then GA process is initiated for new test data generation based on the fitness value. After certain number of iterations provided as input if the given path is not executed, then path is said to be infeasible and stored in infeasible paths.

V. EMPIRICAL INVESTIGATIONS

To determine the potential effectiveness of our approach and prototype tool, *ETODF*, a case study was performed on five Java programs. All these programs were developed and modified according to the testing requirements; Table II gives information about these programs. For each subject program, the table II lists total number of lines of code, number of methods in each program, number of methods under test in each subject program, and short description of each subject program. For each subject program, *ETODF* runs were performed between 50 and 300. We have input the Data Flow Graph to *ETODF* tool and also different infeasible paths were manually input to ETODF for each subject programs with arange of data input to check the effectiveness of our approach and tool.

Table 1I. Subject Programs

Programs	Lines of Code	MUT	No. of methods	Description
TriType	93	1	2	Given three integers, determines the type of triangle
AccountOP	237	2	4	Given an amount and account number and operation value for to
Mid	84	1	2	Given three integers, determines the middle value of the three integers
Find	69	1	2	Given array S[ ], and index F places all elements less than or equal to
Selection	76	1	2	Given an array of integers, selection sorts the array in ascending order

Results of the experiments are very encouraging and promising by applying ETODF. We found that ETODF efficiently achieved results under given constraints. From experimental results, it is evident that our approach is very efficient for automatic test path generation for data flow testing and infeasible detection of paths. Table III summarizes these results of our experiments on subject programs.

Table III. Results of ETODF runs on subject programs

No of Generations	Coverage Percentage					Avg. coverage
	TriType	AccountOP	Mid	Find	Selection	
50	45%	34%	56%	60%	56%	50.2%
100	53%	43%	61%	62%	64%	56.6%
150	62%	51%	64%	73%	67%	63.4%
200	73%	59%	71%	81%	76%	72%
250	83%	71%	83%	84%	83%	80.8%
300	85%	78%	86%	87%	87%	84.6%
350	87%	87%	88%	88%	88%	87.6%
400	88%	89%	88%	88%	88%	88.2%

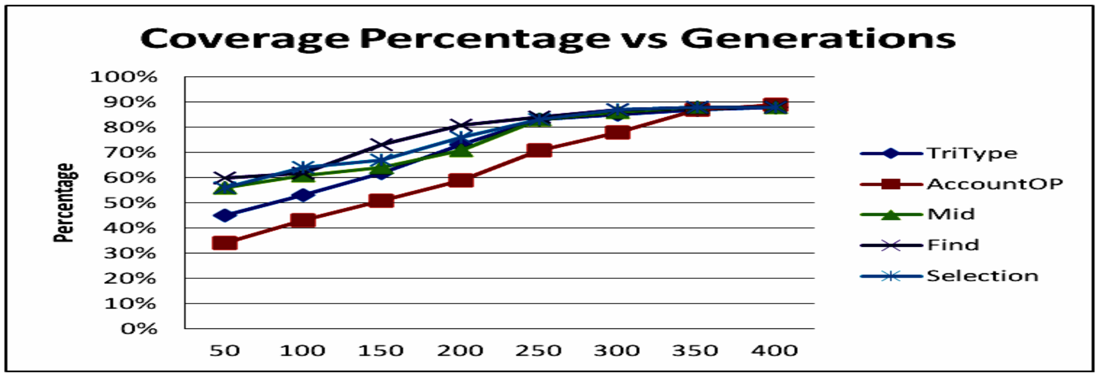

Figure 5. Coverage vs Generations

Table IV. Results of E-ETODF for Infeasible Detection

No of Paths	Accuracy Percentage					Avg. Accuracy
	TriType	AccountOP	Mid	Find	Selection	
5	100%	100%	100%	100%	100%	100%
8	100%	100%	100%	100%	100%	100%
10	100%	100%	100%	100%	100%	100%
12	100%	95%	96%	97%	100%	99%
15	98%	95%	94%	98%	98%	97%
18	95%	93%	91%	95%	95%	94%

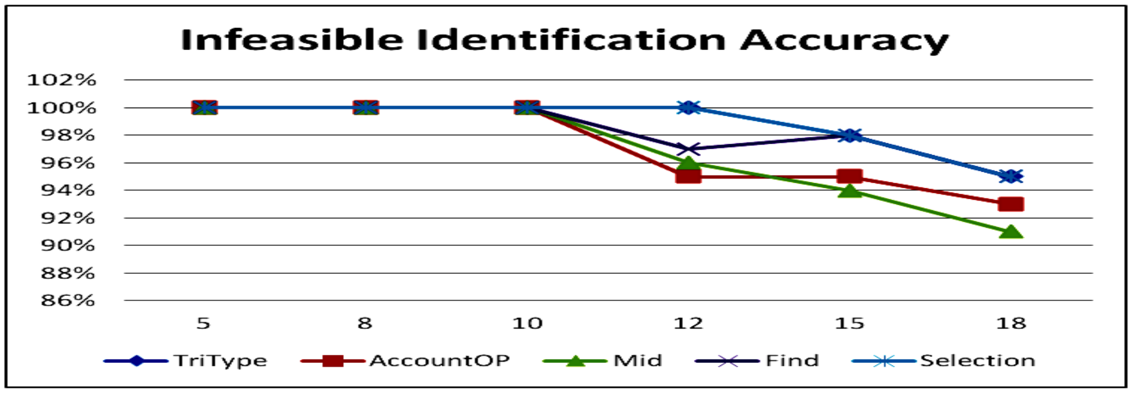

Figure 6. Graphical Representation of Infeasible Detection

From the above experiments, it has been concluded that our approach is very effective in terms of test path generation and for detection of infeasible paths. In both cases, our approaches have performed well. For test path generation, our approach has achieved higher coverage and we have observed that the coverage increases with the increase of number of generations. We have run ETODF between 50 and 400 runs; the coverage achieved is 88% on average for all tests. With increasing number of generations further, we have observed that there is no improvement in coverage. Table III and Figure 5 depicts our experimental results on subject programs with ETODF. For infeasible path detection, we have input different number of paths to ETODF and evaluate for infeasible detection. We have run ETODF with fixed number of iterations for each path i.e. 250 iterations. ETODF successfully identified more than 94%, infeasible path in all cases. ETODF will not identify some paths and it will be identified if we increase the number of iterations or decrease the input domain. Table IV and Figure 6 depict our experimental results on subject programs with ETODF for in feasible detection of test paths.

VI. RELATED WORK

Tonella [7] performed the unit testing of classes using genetic algorithm. In this approach test cases are generated for unit testing of classes using genetic algorithm. McMinn and Holcombe [8] proposed a solution for the state problem in evolutionary testing using the ant colony model.. McMinn and Holcombe [8] also proposed an extended chaining approach for the solution of state problem in evolutionary testing. The basic idea of the chaining approach is to find a sequence of statements, involving internal variables, which needs to be executed prior to the test goal. Watkins [9] performed various experiments to compare different fitness functions proposed by different researchers. Wegener *et al.* [10] Wegener *et. al.* [11] proposed an automatic structural testing environment in their work. Baresel *et al.* [12] suggests some modifications in fitness function design to improve evolutionary structural testing.

McMinn [13] provides a comprehensive survey on evolutionary testing approaches and discusses the application of evolutionary testing. The author discusses the different approaches in which evolutionary testing has been applied and also provides future directions in each individual area. Evolutionary approaches are mostly used in the area of automated test data generation [14-19]. Cheon et al. [20] proposed a specification based fitness function for evolutionary testing of object oriented program. They also proposed automation of Java program testing at unit level using evolutionary approaches. Dharsana et al. [21] generate test cases for Java based programs and also performed optimization of test cases using genetic algorithm. Jones et al. [22] performed automatic structural testing using genetic algorithm in their approach. Bilal and Nadeem [23] proposed a state based fitness function for object oriented programs using genetic algorithm.

VII. CONCLUSION AND FUTURE WORK

This paper presents a novel approach applying evolutionary algorithms for the automatic generation of test paths using data flow relations in a program and then in feasible paths are detected by using genetic algorithm. Our approach starts with a random initial population of test paths and then based on the selected testing criteria new paths are generated by applying a genetic algorithm. A fitness function evaluates each chromosome (path) based on the selected data flow testing criteria and computes its fitness. The approach has been implemented in Java by a prototype tool called ETODF for validation. In experiments with this prototype, our approach has much better results as compared to random testing. We will extend this concept to other levels of testing ,i.e. integration testing and system testing. Currently we have compared our experimental with random testing only. In future, we will also carry out a complete empirical case study for the verification of our approach using all data flow coverage criterion and compare the experimental results with other approaches like hill climbing, tabu search etc.

REFERENCES

[1] Boris Beizer, “Software Testing Techniques”, International Thomson Computer Press, 1990.

[2] Lee Copeland, “A Practitioner’s Guide to Software Test Design”, STQE Publishing, 2004.

[3] Khan, M. "Different Approaches to White Box Testing Technique for Finding Errors”, International Journal of Software Engineering and Its Applications, Vol.5, No.3, July, (2011).

[4] Rao, V. and Madiraju, S. "Genetic Algorithms and Programming -An Evolutionary Methodology”, International Journal of Hybrid Information Technology, Vol.3, No.4, October, (2010).

[5] Srivastava1, P. and Kim, T. "Application of Genetic Algorithm in Software Testing", International Journal of Software Engineering and Its Applications, Vol.3, No.4, October, (2009).

[6] Khan, S.A., and Nadeem, A., “Applying Evolutionary Approaches to Data Flow Testing at Unit Level”. Software Engineering, Business Continuity, and Education Communications in Computer and Information Science, 2011.

[7] Tonella, P., (2004 July) "Evolutionary Testing of Classes", In Proceedings of the ACM SIGSOFT International Symposium of Software Testing and Analysis, Boston, MA, pp. 119-128.

[8] McMinn, P., Holcombe, M., (2003 July) “The state problem for evolutionary testing.” In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), Lecture Notes in Computer Science vol. 2724, pages 2488-2497, Chicago, USA. Springer-Verlag.

[9] Watkins, A., (1995 July) "The automatic generation of test data using genetic algorithms." In Proceedings of the Fourth Software Quality Conference, pages 300--309. ACM, 1995.

[10] Wegener, J., Baresel, A., Sthamer, H., (2001) “Evolutionary test environment for automatic structural testing.” Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms, 43 pp.841–854.

[11] Wegener, J., Buhr, K., Pohlheim, H., (2002 July) “Automatic test data generation for structural testing of embedded software systems by evolutionary testing”, In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pages 1233-1240, New York, USA. Morgan Kaufmann.

[12] Baresel, A., Sthamer, H., Schmidt, M., (2002 July) "Fitness Function Design to improve Evolutionary Structural Testing", Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 02), New York (NY), USA.

[13] McMinn, P., (2004) "Search-based Software Test Data Generation: a Survey", Journal of Software Testing, Verifications, and Reliability, vol. 14, no. 2, pp. 105-156, June.

[14] McGraw, G., Michael, C., Schatz, M., (2001) "Generating software test data by evolution." IEEE Transactions on Software Engineering, 27(12):1085—1110.

[15] Pargas, R., Harrold, M., Peck, R., (1999) “Test-data generation using genetic algorithms. Software Testing”, Verification and Reliability, 9(4):263-282.

[16] Roper, M., (1997 May) "Computer aided software testing using genetic algorithms." In 10th International Software Quality Week, San Francisco, USA.

[17] Tracey, N., Clark, J., Mander, K., McDermid, J., (2000) "Automated test-data generation for exception conditions", SOFTWARE—PRACTICE AND EXPERIENCE, vol., Pages 61-79, January.

[18] Sthamer, H., (1996) "The automatic generation of software test data using genetic algorithms", PhD Thesis, University of Ghamorgan, Pontyprid, Wales, Great Britain.

[19] Seesing, A., Gross, H., (2006) "A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software", International Transactions on Systems Science and Applications, vol. 1, no. 2, pp. 127-134.

[20] Cheon, Y., Kim, M., (2006 July) "A specification-based fitness function for evolutionary testing of object-oriented programs", Proceedings of the 8th annual conference on Genetic and evolutionary computation, Washington, USA.

[21] Dharsana, C.S.S., Askarunisha, A., (2007 December) "Java based Test case Generation and Optimization Using Evolutionary Testing". International Conference on Computational Intelligence and Multimedia Applications, Sivakasi, India.

[22] Jones, B., Sthamer, H., Eyres, D., (1996) "Automatic structural testing using genetic algorithms", Software Engineering Journal, vol. 11, no. 5, pp. 299 – 306.

[23] Bilal, M., Nadeem, A., (2009 April) “A State based Fitness Function for Evolutionary Testing of Object-Oriented Programs”. Studies in Computational Intelligence, 2009, Volume 253/2009, 83-94, DOI: 10.1007/978-3-642-05441-9. Software Engineering Research, Management and Applications 2009.

[24] S.A Khan, A. Nadeem, "A tool for data flow testing using evolutionary approaches (ETODF)",Proceedings of the ninth International Conference on Emerging Technologies(ICET),ICET 2013, December 09-10, 2013 Islamabad, Pakistan.