Path Coverage Test Case Generation Using Genetic Algorithms

Nuntanee Chuaychoo[1], Supaporn Kansomkeat[2]
[1]*Department of Management of Information Technology, Faculty of Engineering,*
Prince of Songkla University, Hat Yai, Songkhla, Thailand.
[2]*Department of Computer Science, Faculty of Science,*
Prince of Songkla University, Hat Yai, Songkhla, Thailand.
supaporn.k@psu.ac.th

***Abstract*—Software testing is an important step in the software process. It makes the developed software more accurate and reliable. Generating test cases is a key element in the process of software testing. The quality of testing depends on the test effectiveness. This paper presents a method for generating test cases using genetic algorithms. A test case generation tool is developed with Visual Basic .NET for supporting our method. The proposed method was applied on case studies and mutation testing was used in our experiment for performance evaluation. The results show that the generated test cases are appropriate for software testing.**

***Index Terms*—Software Testing; Genetic Algorithms; Test Case Generation.**

I. Introduction

Software testing is a key step in the software development process. The test is an activity designed to improve software quality. One important step in software testing is the test case selection process. A good test selection contributes to an effective testing. To enable effective testing, the test cases in the set must cover all features that should be tested and the number of test cases should not be too high

Genetic Algorithm (GA) is an optimization algorithm based on evolutionary processes in nature. GA is used to find the right answer with the principle of natural selection to simulate the evolution of life. The GA process finds the answer by exchanging parameters between chromosomes, which will give an improved answer until finally the right answer. There are many methods proposed for generating test cases by applying GA. For example, Jones et al. [1] presented automatic structural software testing using a fitness function of GA based on the Hamming distance. Their structural testing was applied for testing a variety of programs of varying complexity. Pargas et al. [2] presented a technique for automatic test-data generation using GA. They used control dependencies in the program to guide the search for test data to satisfy test requirements. NIE [3] proposed the PSO test case generation algorithm with enhanced exploration ability (EEA-PSO) to restrain the prematurity and reduce the test case generation costs. Mohi-Aldeen et al. [4] used NSA (Negative Selection Algorithm) to automatically generate test cases to satisfy the path coverage of software.

Along with the principles of GA that are consistent with the test cases refinement, the abstract test cases are modified until they meet the targets. In this study, we present a method for generating test cases by using genetic algorithm. In the proposed method, the abstract test cases are generated randomly. Then, they are refined by genetic algorithm for improving the test parameters. In addition, a test case generation tool is developed according to our method for automatically generating test cases. Mutation testing is used to evaluate generated test cases.

II. Software Testing

Software testing [5] is an important activity in software quality assurance that shows software reliability. The purpose of software testing is to find errors that may occurred during the software development.

There are two techniques widely used in software testing.

1. Black-Box Testing: This technique is used to test for verify the performance of the duties under the requirements specification, and ignores the internal structure of the program.

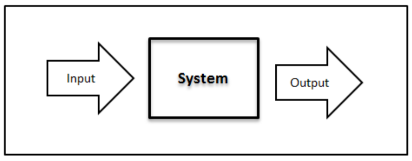


Figure 1: Black-Box Testing

2. White-Box Testing: The testing focuses on the internal program structure. The basic idea of this test is that a crash of the software will be detected. This technique will help to analyze the program to find bugs that may arise from the work of the various commands in the program.

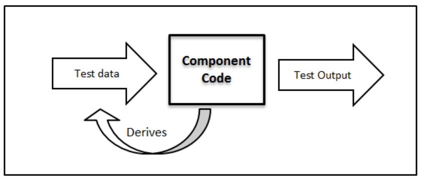


Figure 2: White-Box Testing

III. Genetic Algorithm

Genetic algorithm [6] is a process that relies on the theory of evolution of nature offered by John Holland. Genetic algorithm can be used to find better and more affordable solutions through learning by itself.

Figure 3 shows the process of the genetic algorithm [6].

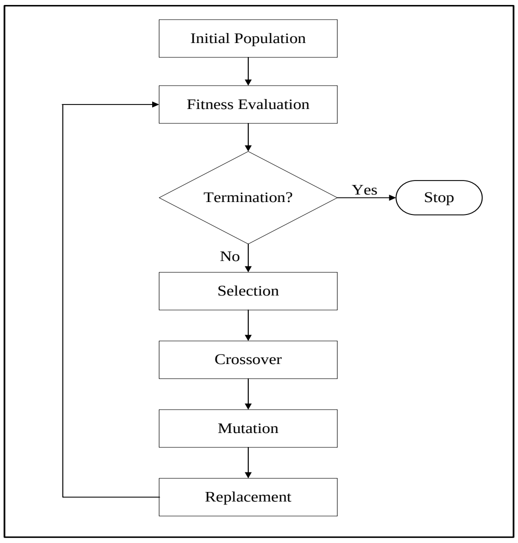


Figure 3: Genetic Algorithm [6]

The main steps of the genetic algorithm can be listed as follows:

1. Initial Population: In this step, initial population of solutions (chromosomes) are randomly produced or manually created from their data type domain.
2. Fitness Evaluation: All chromosomes are evaluated and assigned fitness values. The fitness evaluation is a process used to determine the execute condition in the genetic algorithm process. This process is repeated until a termination condition has been reached.
3. Selection: New population will be selected from the parent population. There are many methods how to select the best chromosome for use in next generation of genetic process such as roulette wheel selection, Boltzman selection, rank selection and some others.
4. Crossover: An importance process of the genetic algorithm is the crossover. The parent population has to switch genes on each chromosome with a switching rate called the probability of crossover. This probability is a real number from 0 to 1 that is randomly generated. In this process, we get a new population that is different from the parent population.
5. Mutation: Mutation is a process of genetic algorithm. Genes in a chromosome will be changed after crossover. This principle helps to create a new answer. The percentage of change is determined by the mutation rate called the probability of mutation.
6. Replacement: The new population after crossover and mutation will be taken to replace the parent population and will become the new parent population.
7. Parameters: The key parameters of genetic algorithm are as following:
   a. Population size
   b. Probability of crossover
   c. Probability of mutation

IV. Control Flow Graph

Control Flow Graph, CFG [7] is a directed graph used to analyze the function of a computer program. This graph demonstrates the functionality of the software under various conditions. Therefore, the control flow graph analysis can help to test the functionality of the software. A CFG can be built from the program source code. It consists of nodes and edges. A node represents a statement or a basic block of statements. An edge represents the flow of control between nodes. Notations for representing control flow are shown in Figure 4.

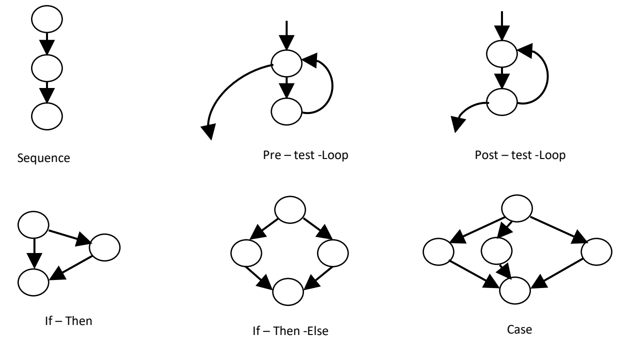


Figure 4: Notions for Representing Control Flow [7]

V. Test Case Generation

In this section, the proposed test cases generating process using genetic algorithm will be described. Our approach consists of two main stages as shown in Figure 5.

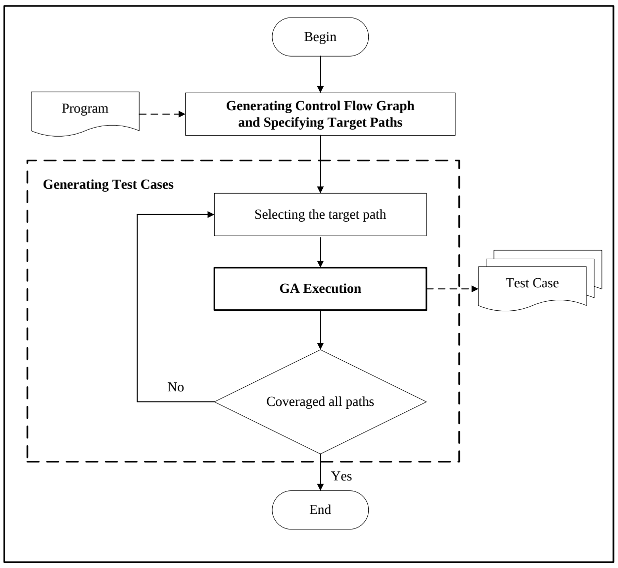
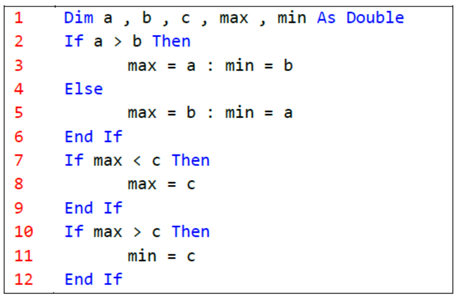

Begin
Program
Generating Control Flow Graph and Specifying Target Paths
Generating Test Cases
Selecting the target path
GA Execution
Test Case
No
Coveraged all paths
Yes
End

Figure 5: Framework of the proposed approach

A. *Generating Control Flow Graph and Specifying Target Paths*

This procedure consists of two sub-stages:

1. Generating a control flow graph: A control flow graph is generated from a program code. The generated graph consists of nodes connected by edges. Figure 6 shows the control flow graph of Max-Min program that consists of 10 nodes and 12 edges.
2. Specifying Target Paths: The target paths are specified by using the control flow graph generated from the previous step. In this research, the path coverage (PC) is used to specify target paths. The PC requires every path in the tested program has to be executed at least once. Figure 7 shows all target paths of the Max-Min control flow graph in Figure 6.



```
1   Dim a , b , c , max , min As Double
2   If a > b Then
3           max = a : min = b
4   Else
5           max = b : min = a
6   End If
7   If max < c Then
8           max = c
9   End If
10  If max > c Then
11          min = c
12  End If
```

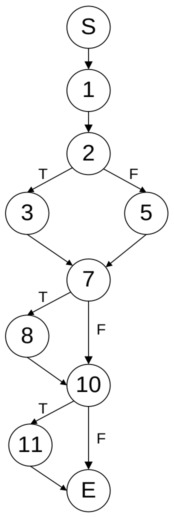


Figure 6: Control Flow Graph of Max-Min Program

Path 1	S	1	2T	3	7T	8	10T	11	E
Path 2	S	1	2T	3	7T	8	10F	E	
Path 3	S	1	2T	3	7F	10T	11	E	
Path 4	S	1	2T	3	7F	10F	E		
Path 5	S	1	2F	5	7T	8	10T	11	E
Path 6	S	1	2F	5	7T	8	10F	E	
Path 7	S	1	2F	5	7F	10T	11	E	
Path 8	S	1	2F	5	7F	10F	E		

Figure 7: All Paths of Max-Min CFG

B. Generating Control Flow Graph and Specifying Target Paths

In this process, test cases are created by using the genetic algorithm. The generation process starts by selecting a target path. This target path is set to be the considering path for finding a test case by using the Genetic Algorithm Execution (GA Execution). The generation process is repeated until all target paths are considered.

Figure 8 shows the steps of GA Execution as following:

Step 1: Initial Test Cases

The input space partitioning approach proposed by Ammann and Offutt [8] is used to generate initial test cases (chromosomes). We consider the data type of each parameter in the program. The number of partition depends on the data type of the parameter. For example, considering integer or double parameters, the range can be partitioned into three blocks: less than zero, zero, and greater than zero. The initial test cases contain all possible test cases of the considered target path. After all parameters are partitioned into blocks, one block from each parameter is combined to be a test case. We apply the Each Choice (EC) criterion [8] for the combination blocks. The EC requires that each block of parameters must be used in at least one test case.

For example, the Max–Min program has three input parameters (a, b, and c) that are double data types. Each parameter can be partitioned into three blocks ([a<0.0, a=0.0, a>0.0], [b<0.0, b=0.0, b>0.0], and [c<0.0, c=0.0, c>0.0]), so we can combine them into three sets: {a < 0.0 , b < 0.0 , c < 0.0}, {a = 0.0 , b = 0.0 , c = 0.0}, and {a > 0.0 , b > 0.0 , c > 0.0}. A set represents a test case. Then, the test data for each partition of each set are generated randomly. Table 1 shows the initial test data for the Max–Min program.

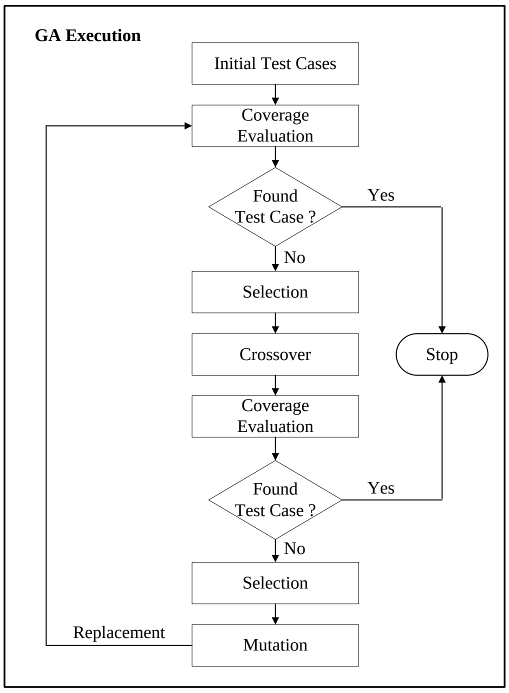


Figure 8: Genetic Algorithm Execution

Table 1
Initial Test Data for Max-Min Program

Set	a	B	c
1	-2.2	-1.4	-4.5
2	0.0	0.0	0.0
3	3.7	2.4	5.3

Step 2: Coverage Evaluation

In this research, we use the branch distance calculation proposed by Korel [9] for fitness evaluation. Branch distance is calculated based on the type of branch predicate. Branch predicate and branch distance of the Korel's distance function are shown in Table 2. Table 3 shows all branch predicates of the Max-Min Program.

Table 2
Korel's Distance Function [9]

Branch Predicate	Branch Distance
a < b	a - b
a <= b	a - b
a > b	b - a
a >= b	b - a
a == b	abs(a-b)
a <> b	abs(a-b)
a && b	a + b
a || b	min(a , b)

Table 3
Decision for Max-Min Program

Branch Predicate	Branch Distance
a > b	b - a
max < c	max - c
max > c	c - max

Korel's distance function is used to compute the distance between the target path and the considering path. In this study, the selected target path and an execution paths that traversed by test data of a test case are compared; if they are different then the all branch distances of this execution path are calculated to be the fitness value of the test case. In case they are same then the GA execution process for the target path is finished and the test case is set for the selected target path.

For example, the selected target path is the first path in Figure 7. We found that every execution path of test data sets in Table I is different; therefore all branch distances of every execution paths are calculated as shown in Table 4.

Table 4
Branch Distance for Test Sets

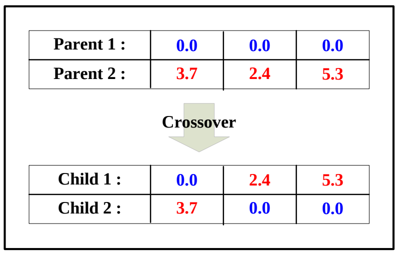
Set	A	b	c	Branch Distances
1	-2.2	-1.4	-4.5	0.8
2	0.0	0.0	0.0	0.0
3	3.7	2.4	5.3	-1.6

Step 3: Selection

The selection step is done to select test cases for the evolution process, crossover, and mutation. The sets of test are sorted by descending order of branch distance values. The data sets with small distance values will be selected for the evolution process. For example, the test data sets in Table IV are sorted to be the new sequence sets 3, 2, and 1.

Step 4: Crossover

After sorting in the selection process, the first two data sets are chosen for crossover operation. In this research, we use single-point crossover. Parameters of the chosen two data sets (parent) are exchanged at a random position to produce two new test data sets (child). Figure 9 shows an example for a single point crossover at the first position.



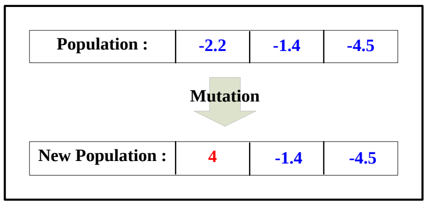
Parent 1 :	0.0	0.0	0.0
Parent 2 :	3.7	2.4	5.3

Child 1 :	0.0	2.4	5.3
Child 2 :	3.7	0.0	0.0

Figure 9: Single Point Crossover

Step 5: Mutation

After sorting in the selection process, mutation is applied to the first data set using single point mutation. In our method, the position for mutation is selected randomly. The value of this position is replaced by another value that is generated randomly based on the range of its data type. Figure 10 shows an example for the single point mutation at first position that changes the value from -2.2 to 4.

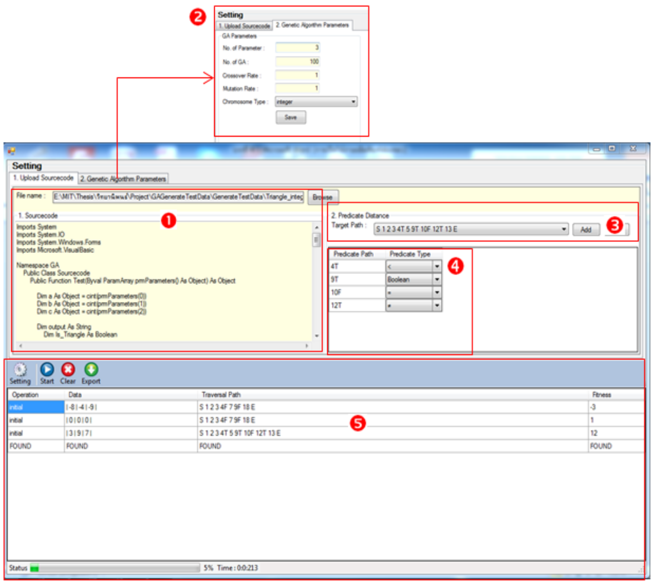


Population :	-2.2	-1.4	-4.5

New Population :	4	-1.4	-4.5

Figute 10: Single Point Mutation

To support our idea, we implemented the application that provides GUI for generating test data. The application is developed by Visual Basic .NET works on Windows base. User can set parameters of the genetic algorithm to find the best solution in generating test sets. The user interface is shown in Figure 11.



No. of Parameter : 3
No. of GA : 100
Crossover Rate : 1
Mutation Rate : 1
Chromosome Type : integer

File name : E:\MIT\Thesis\วิทยานิพนธ์\Project\GAGenerateTestData\GenerateTestData\Triangle_integ

```
Imports System
Imports System.IO
Imports System.Windows.Forms
Imports Microsoft.VisualBasic

Namespace GA
  Public Class Sourcecode
    Public Function Test(Byval ParamArray pmParameters() As Object) As Object

      Dim a As Object = cint(pmParameters(0))
      Dim b As Object = cint(pmParameters(1))
      Dim c As Object = cint(pmParameters(2))

      Dim output As String
        Dim Is_Triangle As Boolean
```

Target Path : S 1 2 3 4T 5 9T 10F 12T 13 E

Predicate Path	Predicate Type
4T	<
9T	Boolean
10F	=
12T	=

Operation	Data	Traversal Path	Fitness
initial	|-8|-4|-9|	S 1 2 3 4F 7 9F 18 E	-3
initial	|0|0|0|	S 1 2 3 4F 7 9F 18 E	1
initial	|3|9|7|	S 1 2 3 4T 5 9T 10F 12T 13 E	12
FOUND	FOUND	FOUND	FOUND

Status 5% Time : 0:0:213

Figure 11: Test Cases Generation GUI

The application contains five parts. Part ❶ is used to upload a program file written in Visual Basic .NET. Part ❷ is used to set parameters for genetic algorithm which are the numbers of the parameters, population size, probability of crossover, probability of mutation and chromosome type. Part ❸ is used to define target paths. Part ❹ is used to specify branch predicates. After execution, the results are shows in part ❺.

VI. CASE STUDY

In this section, three small programs are used to evaluate our approach. The case study programs are as follows:

1. Calculate Maximum–Minimum
   This program receives three numbers that are double types and finds the maximum and minimum numbers. There are three branch predicates in the source code.
2. Triangle Problem
   This program receives three sides of triangle that are integer numbers and classifies type of triangle (Not a Triangle, Equilateral, Scalene, and Isosceles). There are four branch predicates in this source code.
3. Calculate Middle Data
   This program gets three characters and identifies the character that has the middle value. For example the characters are ‘a’, ‘c’ and ‘b’ then the result is character ‘b’. There are three branch predicates in this source code.

Test cases of these three programs are generated by our tool. As stated before, there are three key parameters in the genetic algorithm: population size, probability of crossover, and probability of mutation are set to be 100, 1, and 1, respectively. Probabilities of crossover and mutation are set to be 1 for the execution crossover and mutation operations in every generation which increase the probability to find the appropriate test cases. The results are shown in Table 5.

Table 5
Test Cases for Case Studies Programs

Program	Total Branch Predicates'	Total Target Paths	Found Test cases	Not found Test cases
Calculate Maximum – Minimum	3	8	6	2
Triangle Problem	4	8	4	4
Find Middle Character	3	6	6	0

As shown in Table 5, some target paths of the Calculate Maximum-Minimum and Triangle Problem programs cannot generate test cases. Because these target paths are infeasible paths that cannot be traversed by any test cases. Possible causes of infeasible paths are source codes that contain logically inconsistent predicates or some data that produce an infinity loop in program, and others.

Mutation testing [10] is used to evaluate the effectiveness of the test cases created by our method. Mutation testing starts by making small syntactic changes into the original program. The mutated programs are called mutants. Mutants are obtained by applying mutation operators. For example, an arithmetic operator may be changed from addition to subtraction. Mutation analysis is based on the number of killed mutants. The behavior of the original program is different from the mutant called killed mutant. There are cases where it is not possible to find a test case that could kill this mutant called equivalent mutant. The ratio of killed mutants to the total mutants minus the number of equivalent mutants is called mutation score.

The test cases evaluation process is carried out as follows:

1. Test cases of case study programs are generated by using our algorithm.
2. Mutants for each case study are created by applying the mutation operators designed by Offutt et al. [11]. As shown in Table VI, there are 21 mutants for calculating the maximum and minimum program, 38 mutants for the triangle problem program, and 21 mutants for the calculation of the middle data program.
3. Test cases are executed with the original program and the mutant programs to calculate mutation scores. If a mutation score is one or near to one, it indicates the effectiveness of the test cases. The evaluation results are shown in Table 6.

Table 6
Test Cases Evaluation for Case Studies Programs

Program	Total Test cases	Total Mutants	Killed mutants	Mutation Score
Calculate Maximum – Minimum	6	21	19	0.9
Triangle Problem	4	38	25	0.86
Find Middle Character	6	21	21	1

VII. CONCLUSIONS

This paper presents a method for generating test cases using genetic algorithm. First, a flow graph is created from the program source code and the paths of the graph are specified by following path coverage criteria. Then, the genetic algorithm is used for generating test cases. Finally, we use mutation testing to evaluate the effectiveness of the test cases. Experimental data show that tests generated by our method have a strong ability to detect faults. In addition, this research has also developed a test case generation tool based on the proposed method. In the future, we intend to improve the tool to automate the whole test process and find solutions for infeasible paths. Moreover, more complex programs are needed for evaluating our method and we need more studies on the optimization algorithms for test case generation, such as hybrid GA and PSO [12,13], which have a higher efficiency than GA algorithms.

ACKNOWLEDGEMENT

The authors would like to thank for the support given to this research by the Faculty of Engineering and Faculty of Science, Prince of Songkla University, Thailand.

REFERENCES

[1] Jones B.F, Sthamer H.-H and Eyres D.E. 1996. Automatic structural testing using genetic algorithms. *Software Engineering Journal*: 299-306.

[2] Pargas R. P., Harrold M. J. and Peck R. R.. 1999. Test-Data Generation Using Genetic Algorithms. *Journal of Software Testing, Verification and Reliability I*: 1-19.

[3] P. NIE. 2012. A PSO Test Case Generation Algorithm with Enhances Exploration Ability. *Journal of Computational Information System*: 5785-5793.

[4] Mohi-Aldeen S. M., Mohamad R. and Deris S.. 2014. Automatic Test Case Generation for Structural Testing Using Negative Selection Algorithm. *International Conference of Recent Trends in Information and Communication Technologies*: 270-280.

[5] Ghezzi C., Jazayeri M., and Mandrioli D. 2002. Fundamentals of Software Engineering 2nd. USA: Prentice Hall PTR Upper Saddle River.

[6] Winter G., Periaux J., Galan M.and Cuesta P. 1996. Genetic Algorithms in Engineering and Computer Science. USA: Wiley Publisher.

[7] Jorgensen P. C. 2007. Software Testing A Craftsman's Approach 3rd. USA: AuerbachPiblictions.

[8] Ammann P. and Offutt J. 2008. Introduction to Software Testing. UK: Cambridge University Press.

[9] Korel B. 1996. Automated test data generation for programs with procedures. *Proceedings of International Symposium on Software Testing and Analysis*: 209-215.

[10] Guti´errez-Madro˜nal L., Jos´e J. Dom´ınguez-Jim´enez and I. Medina-Bulo. 2014. Mutation Testing: Guideline and Mutation Operator Classification. *The Ninth International Multi-Conference on Computing in the Global Information Technology*: 171-179.

[11] Offutt, A.J. Lee, A. Rothermel, G. Untch R., and Zapf C. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transaction on Software Engineering Methodology*. 99-118.

[12] Singh A., Garg N. and Saini T. 2014. A hybrid Approach of Genetic Algorithm and Particle Swarm Technique to Software Test Case Generation. *International Journal of Innovations in Engineering and Technology*: 208-214.

[13] Singla S., Kumar D., Rai H M and Singla P. 2011. A hybrid PSO Approach to Automate Test Data Generation for Data Flow Coverage with Dominance Concepts. *International Journal of Advanced Science and Technology*: 15-25.