Journal of Information Processing Systems, Vol.7, No.2, June 2011 DOI : 10.3745/JIPS.2011.7.2.363

An Adequacy Based Test Data Generation Technique Using Genetic Algorithms

Ruchika Malhotra* and Mohit Garg*

Abstract— As the complexity of software is increasing, generating an effective test data has become a necessity. This necessity has increased the demand for techniques that can generate test data effectively. This paper proposes a test data generation technique based on adequacy based testing criteria. Adequacy based testing criteria uses the concept of mutation analysis to check the adequacy of test data. In general, mutation analysis is applied after the test data is generated. But, in this work, we propose a technique that applies mutation analysis at the time of test data generation only, rather than applying it after the test data has been generated. This saves significant amount of time (required to generate adequate test cases) as compared to the latter case as the total time in the latter case is the sum of the time to generate test data and the time to apply mutation analysis to the generated test data. We also use genetic algorithms that explore the complete domain of the program to provide near-global optimum solution. In this paper, we first define and explain the proposed technique. Then we validate the proposed technique using ten real time programs. The proposed technique is compared with path testing technique (that use reliability based testing criteria) for these ten programs. The results show that the adequacy based proposed technique is better than the reliability based path testing technique and there is a significant reduce in number of generated test cases and time taken to generate test cases.

Keywords—Software Testing, Adequacy Based Testing Criteria, Reliability Based Testing Criteria, Genetic Algorithms, Mutation Analysis

1. INTRODUCTION

Software Testing is the process of executing a program with the intent of finding errors [1]. Software testing is essentially a combination of software validation and verification. Testing is a very complex, yet very important activity. It is one of the most significant means to ensure the quality of the software. Software testing accounts for nearly 50% of the total development cost of the software. Exhaustive testing is not possible because there are no limits on how much we can test. Thus, to limit the process of testing, the concept of testing criteria is used. In this sense, testing is defined as the process of generating test data that satisfies some defined testing criteria. Satisfying the testing criteria marks an end to testing process.

Testing can be broadly classified as functional and structural testing. Functional testing focuses on testing the functionality of the system using some functional test criteria such as equivalence classes [2], random testing [2], etc. Structural testing focuses on testing the structure of

Manuscript received December 6, 2010; revised March 6, 2011; accepted March 18, 2011.
Corresponding Author: Ruchika Malhotra
* Dept. of Software Engineering, Department of Computer Engineering, Delhi Technological University, Bawana Road, Delhi 110042, India (ruchikamalhotra2004@yahoo.com, mohit_270488@yahoo.co.in)

 Copyright ⓒ 2011 KIPS (ISSN 1976-913X)

the system using some structural test criteria such as paths, functions, conditions, branches etc. The structural testing criteria are broadly classified into two categories viz. 'Reliability based testing criteria' and 'Adequacy based testing criteria' [3]. Reliability based testing criteria are used to show the correctness of the program in terms of achieving some coverage that may be either 'control flow coverage' or 'Data flow coverage'. Control flow coverage criteria include path testing [4], condition testing [5], branch testing [6, 7], etc. Data flow coverage criteria include all-uses [8], all-du (definition use) paths [9, 23, 26, 27] etc. Reliability based criteria aims to generate a set of reliable test cases. A test case set is reliable if its execution ensures that the program is correct on all its inputs [3]. Adequacy based testing criteria is used to show the adequacy of the test cases in terms of whether or not they can identify faults in the program. DeMillo and Offutt [3] has rightly stated that "A test case set is adequate if it causes all the incorrect versions of the program to fail to execute successfully. Adequacy requires that the test case set detects faults rather than show correctness".

One of the major differences between reliability based and adequacy based testing criteria [3] is that reliability based criteria verify the correctness of the program with regard to a pre-defined testing criteria but does not take into account the identification of faults in the program. On the contrary, adequacy based testing criteria verify the adequacy of the test data. This informs the tester about whether or not there are any faults in the program, so that the quality of the program can be enhanced by focusing upon these faults.

In this paper, we propose a test data generation technique that uses adequacy based test criteria and genetic algorithms. To check the adequacy of the test cases, we use the concept of mutation analysis in the proposed technique. In general mutation analysis is applied after the test data has been generated. However, in this work, we apply mutation analysis at the time of generating test data only. This has two significant benefits:

- The test data set that is generated is assured to be adequate. Hence, we need not apply mutation analysis after the test data is generated.
- The total time required to generate test data includes only the time taken to generate test data and not the time to check its adequacy. This is so, because the generated test data is assured to be adequate. This has an advantage over other techniques that are based on reliability based testing criteria as in these techniques, the total time required for generating test cases is the sum of the time taken to generate test data and the time taken to check its adequacy.

We have used genetic algorithms (a heuristic based search procedure) because of their ability to generate near global optimum solution and their widespread use in the literature of heuristics. The use of genetic algorithms is also facilitated by the fact that many of the testing problems can be formulated as search problems. For instance, in our case, the problem of generating adequate test data can be formulated as the problem of searching the input domain of the program, for those input values that satisfy the adequacy test criteria or that can identify faults in the program.

The main objective of this work is to show the effectiveness of adequacy based testing criteria over the reliability based testing criteria. In order to achieve this objective, we have validated our technique on ten real time programs developed in C language and have compared our technique (adequacy based) with path testing technique (reliability based) for these programs. We

have chosen path testing technique because path testing strategy can alone detect almost 65% of the errors in the program [28]. The results suggests that the adequacy based proposed technique is better than the reliability based path testing technique and there is a significant reduce in number of generated test cases and time taken to generate test cases.

The rest of the paper is organized as follows:. Section 2 presents the related work. Section 3 represents the key research concepts of genetic algorithms and mutation analysis. Section 4 presents the proposed technique. Section 5 presents the experimental analysis and results. Section 6 concludes the paper.

2. RELATED WORK

In various research work carried out in the field of test data generation, different researchers have used different testing criteria while generating test data.

L.A. Clarke generated the software test data by using ‘path coverage testing criteria’. The idea was to select some target path, execute the path symbolically, identify constraints, and then to generate the test case values such that the identified constraints are satisfied [10].

J.W. Duran and S.C. Ntafos generated the software test data using ‘random testing technique’. In their work, they (1) compared random testing with partition testing, and (2) evaluated the random testing process using various coverage criteria such as segment coverage, branch coverage, path coverage, etc [2].

B. Korel focused on generating software test data using ‘path coverage testing criteria’. It was a dynamic path testing technique that generates test cases by executing the program with different possible test case values [11].

R.A. DeMillo and A. J. Offutt focused on generating test case values using ‘adequacy based testing criteria’ [3].

S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios discussed about application of genetic algorithm in software testing [29].

C.C. Michael, G.E. McGraw, M.A. Schatz, C.C. Walton identified the application of heuristic search techniques in solving various software engineering problems. This is facilitated by the fact that many of the software engineering problems can be stated as the search problems. For instance, the testing problem can be stated as the problem of searching the input domain of the program, for those test case values that satisfy the pre defined testing criteria [12].

C.C. Michael, G.E. McGraw, M.A. Schatz proposed a technique for automated test data generation using ‘branch coverage testing criteria’. They used genetic algorithm to generate the test cases. Michael and McGraw generated a tool named GADGET to generate test data. The tool makes use of a branch table that keeps a track of all the branch conditions for both of their true and false parts [5].

J. Wegener, A. Baresel, and H. Sthamer focused on generating test data by using several, ‘structural test coverage criteria’ using evolutionary approaches like genetic algorithms. They identified that in general, all the test data generation technique focus on only single test criteria at a time. But, in their work, they have considered all the test coverage criteria and have provided an effective classification of these criteria [13].

J. C. Lin and P. L. Yeh discussed about automatic test data generation using ‘path testing criteria’ and genetic algorithms [30].

N. Mansour and M. Salame focused on generating software test data using 'path coverage testing criteria' and hamming distance as a fitness function [14].

P. McMinn [24], M. Harman and J. Wegener [25], discussed about search based software test data generation.

K. Dahal and A. Hossain focused on generating software test data using UML based software specifications and genetic algorithm. This technique used 'transition coverage test criteria' that identifies the number of transitions fired on receiving events. The sequence of events represents the candidate test case values. The number of transitions that are fired on receiving a certain sequence of events is used as a fitness function [15].

A. Bouchachia proposed a modification to genetic algorithm by incorporating immune operators to it. This approach used 'Condition coverage testing criteria'. In general, the genetic algorithm includes three main operators viz. selection, cross over, and mutation. In immune genetic algorithm, one more operator known as're-selection' is added to the genetic algorithm [16].

X. Shen, Q. Wang, P. Wang, Bo Zhou proposed the hybrid scheme of genetic algorithm and tabu search that came to known as GATS algorithm. 'Function coverage' is used as testing criteria. Tabu search is a local search technique and is used as a mutation operator in this work [17].

M. A. Ahmed and I. Hermadi attempted to generate test data for multiple paths using genetic algorithm [31].

M. Harman focused on automated test data generation using search based software engineering. Automated test data generation using genetic algorithm is a part of search based software engineering [22].

A.S. Ghiduk, M.J. Harrold, M.R. Girgis proposed an approach to generate test data using 'du (definition use) paths coverage testing criteria'. In their work, they focused upon generating the dominance tree from the control flow graph of the program [8].

P.R. Srivastava and T.H. Kim proposed a technique for generating test cases using 'path coverage testing criteria' and genetic algorithm. This technique was based on considering the criticality of the path. In this technique, the edges of the control flow graph are assigned weights. The fitness function is calculated by taking the sum of the weights of all the edges of a particular path. The path with maximum value of fitness function is the most critical [18].

A. Rauf and S. Anwar proposed a technique to generate software test data using 'GUI based test criteria'. GUI applications are event driven. It receives certain events as input from the user and follows certain path as a result of receiving that event. This technique uses genetic algorithm to generate software test data. The sequence of events represents the candidate test case values. Number of paths followed, out of the total number of paths is used as a fitness function [19].

On the basis of above mentioned related works, we identify a framework for the classification of test criteria. This is shown in figure 1.

Referring to the above- mentioned related works and figure 1, we can deduce that much of the work has been done on reliability based test criteria while much less work has been done on adequacy based test criteria. As shown above, all the related works except for that presented in [3] focus upon generating test cases using reliability based test criteria only. In this paper, we focus upon adequacy based test criteria for generating test cases. We also use genetic algorithms in order to include its benefits in our technique. The work proposed in this paper shows the effectiveness of adequacy based test criteria over reliability based test criteria, by comparing both on ten real time programs.

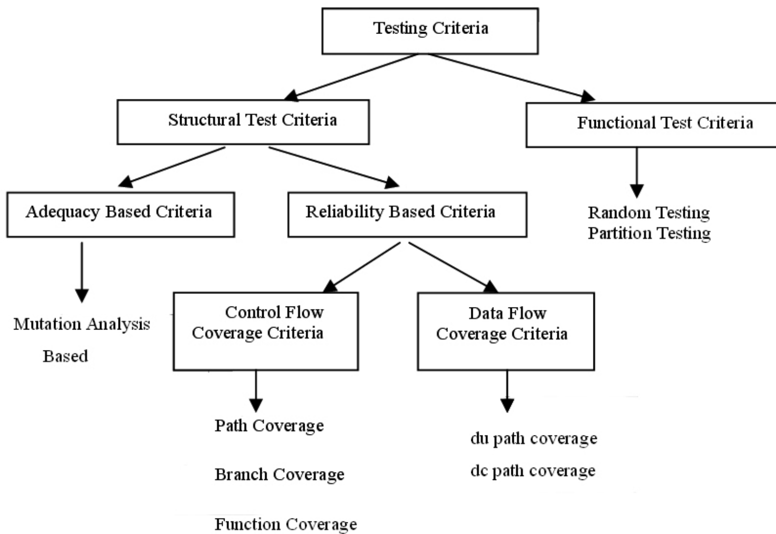

Fig. 1. General Classification framework for testing criteria

3. Key Research Concepts

Our technique is based on the use of 'Genetic Algorithm' and 'Mutation Analysis'. These two concepts are the main focus of this section.

3.1. Genetic Algorithm

Genetic algorithms are the heuristic search algorithms that are used to solve a variety of optimization problems. Genetic algorithms mimic the process of natural biological evolution and the Darwin's principal of the survival of the fittest. The genetic algorithms cause a population of individuals to evolve from one generation to another, each time allowing the best characteristics of one generation to pass to the next generation. *Genetic Algorithms are heuristic in nature.* They are generally good, but sometimes, they may not be better. But on average, they improve the quality of the search that we perform.

The block diagram for basic procedure of genetic algorithm is shown in figure 2.

The **Initializing** phase requires generating the initial population of individuals or chromosomes. This is the starting point of genetic algorithms. Chromosome refers to a set of the values of the input variables that are obtained from the input domain. The structure and length of chromosome depends upon the number and the range of the input values [23]. For e.g. if there is an input variable x ranging from 0 to 31 and if chromosomes are to be represented in a bit manner, then chromosomes will represented by a 5 bit pattern. The initial population is generated randomly or through seeding. Seeding means that if a tester has in-depth knowledge of the search space, he can include some beneficial points/chromosomes (that are near to global optimum) in the initial population. This adds to the efficiency of genetic algorithm.

The **Evaluation** involves calculating the fitness value of each chromosome. The fitness is a measure of goodness of each chromosome relative to the global optimum solution. It measures

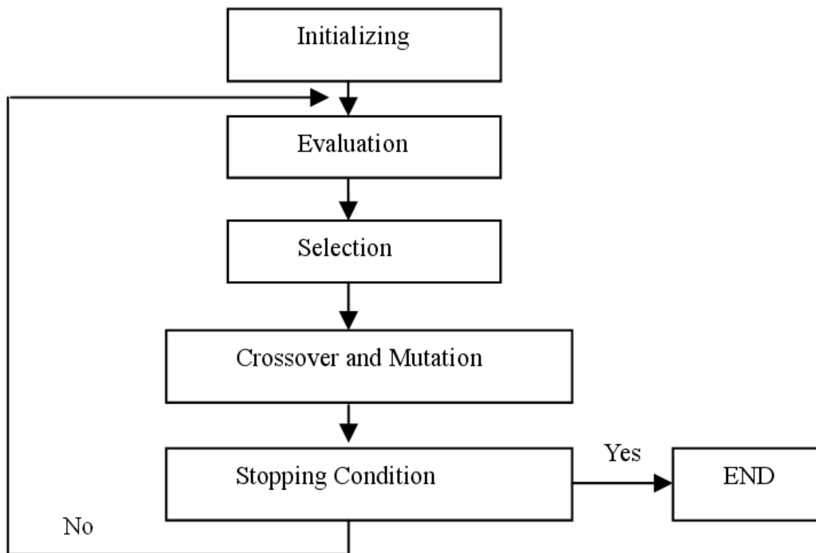

Fig. 2. Block Diagram of Genetic Algorithm

how close a chromosome is towards a global optimum solution. The chromosome with higher fitness value is more near towards the global optimum as compared to the chromosome with less fitness value.

Selection is simply a replication of some chromosomes from the current population based upon their fitness value. The basic philosophy of selection is that "highly fit chromosomes survive while the worst fit dies out". This ensures that only the best characteristics are transmitted from the current generation to the next generation. The output of selection is a mating pool that contains the chromosomes that mate with each other to generate offspring.

Once a mating pool is obtained, **Crossover** comes into play. It is essentially a random exchange of genetic material between two chromosomes. This exchange is governed by a crossover site. Crossover is a 2 step process:

1. Mating: 2 chromosomes from the mating pool are selected at random. These are the pairs that will be mating to produce off spring.
2. Exchange of genes: Once the chromosome mates are selected, these mates exchange a part of their string as determined by a cross-over site.

Mutation is an occasional but an important concept of genetic algorithm. Mutation is a random change of a bit in a chromosome i.e. flipping of a bit from 0→1 or 1→0. For more details on genetic algorithms, refer to [20].

3.2 Mutation Analysis

Mutation Analysis is basically a structural based approach to measure the adequacy of the test data. It is also known as 'Fault Seeding' technique. The technique is basically based on seeding or inserting certain faults in the program and then checking if the generated test data can find

these faults or not. If it can, the test data is considered to be adequate, else it is inadequate. Faults are introduced by making some changes to the original program. A changed copy of the program is called a mutant.

In this paper, we use the same approach of mutation analysis as described in [21]. The only difference is that, we perform mutation analysis at the time of test data generation rather than performing it after the test data has been generated.

3.2.1 Mutant Generation

In mutation analysis, faults are seeded in the program. This means that some syntactic changes are made to the program statements. The changed program is called a mutant. "One most important property of mutation analysis is that the mutant should follow different execution path than the original program after the execution of the mutated statement".

The mutants are generated using mutation operators. A variety of mutation operators have been explored by researchers. Some of them include:

- Statement deletion
- Replace each arithmetic operation with another one, e.g. + with * and – with /.
- Replace each Boolean relation with another one, e.g. > with >=, ==, and <=) etc.

Rules for identifying mutants are as follows:

1. Only first order mutants are generated. First order mutants are mutants that contain a single change. In general, only first order mutants are sufficient and are used in testing. Second and higher order mutants (that contain multiple changes) make it difficult to manage the mutants, thus adding to complexity. Thus, only first order mutants are generated.

2. In general, there are no limits on the number of mutants that can be generated. To circumvent this problem, we restrict the domain of mutation operators. We generate mutants by applying mutation operators from this domain only. The domain of mutation operators that we use in our work are:

Operand Replacement Operator:

Replace a single operand with another operand or a constant.

E.g. if(x>y){} → original statement
if(5>y){}→mutated statement generated by replacing x by a constant 5

Expression Modification Operator:

Replace an operator with some other operator or insert new operator.

E.g. if(x==y){} → original statement
if(x>=y){}→mutated statement generated by replacing == by >=

Statement Modification Operator:

Delete the entire if-else statement.
Replace a line by a return statement, etc.

4. Proposed Technique

In the subsequent sections, we describe the proposed technique in detail.

4.1 Overview of the Proposed Technique

In this section, we present an overview of the proposed technique. In the proposed technique, we generate software test data using adequacy based testing criteria and genetic algorithm. One of the most common ways to generate a set of adequate test cases is mutation analysis. In general, mutation analysis is applied after the test data is generated, to check whether the generated test data is adequate or not. However, in this paper, we apply mutation analysis at the time of generating test cases only rather than applying it after the test data is generated. This will lead to the generation of adequate test cases only, and hence we need not check for the adequacy of test cases again.

The test cases with respect to mutation analysis can be of two types viz. adequate and inadequate. A test case is adequate if it can identify faults in the program, else it is inadequate. The idea is to separate the adequate test cases from the inadequate ones at the time of test data generation only and to output only those test cases that are adequate. This is done by means of generating certain constraints. Once the constraints are generated, we apply genetic algorithm to solve these constraints. These solutions represent the test case values. In order to solve the constraints using genetic algorithms, we derive a fitness expression from these constraints (using an appropriate fitness function).

Broadly, the proposed technique follows the below mentioned steps:

- Identify a set of mutants for the program under test. (Refer to section 3.2.1)
- Identify the constraints (between original program and mutant) such that solving these constraints ensures that the mutants are killed.
- Solve the constraints using genetic algorithms. These solutions represent the test cases.

In the proposed technique, we build a mutation table to keep track of all the mutants that are generated. The table contains several fields such as: Mutant_Id, Parent statement, Mutated Statement and Status. This is shown in table 1.

- Mutant_Id is a symbol Mi followed by the statement number that is mutated. The symbol and statement number are separated by a colon (:).
- Parent statement is a statement in the original program that is targeted to be changed in the mutant.
- Mutated statement is the changed statement.
- Status can be processed (if mutant is killed) or unprocessed (if mutant is alive)

The procedure for generating constraints and fitness function is discussed in subsequent sec-

tions.

Table 1. Mutation Table

Mutant_Id	Parent Statement	Mutated Statement	Status

4.2 Constraint Generation

The constraints are generated between the correct statement of the original program and the changed statement of the mutated program. The idea is to generate the constraints such that the solutions of these constraints are guaranteed to kill the mutants. These solutions represent effective test case values.

An important property that must be kept in mind before generating constraints is that they should be generated such that they cause the mutants to follow a different execution path than the original program, after the execution of the mutated statement**.** This means that though the execution path followed by an original program and a mutant is same till the execution of the mutated statement, it should be different after the execution of the mutated statement. It is only then the mutant will generate different output as compared to the original program and hence is considered to be killed. The constraints are represented in Disjunctive Normal Form (DNF).

Example: Let us consider the following program (to find the GCD of two numbers) and a mutant for it as shown in figure 3.

Referring to table1, we can build a mutation table for the mutant generated in figure 3. This is shown in table 2.

Original Program

```
1.void GCD(int a, int b)
2.{
3. while(a!=b)
4.    {
5.      if(a>=b-1)
6.           a=a-b;
7.      else if(a<b-1)
8.           b=b-a;
9.    }
10. printf("\nGCD=%d",a);
11.}
```

Mutated Program

```
1.void GCD(int a, int b)
2.{
3. while(a!=b)
4.    {
5.      if(a<b-1).
6.           a=a-b;
7.      else if(a<b-1)
8.           b=b-a;
9.    }
10. printf("\nGCD=%d",a);
11.}
```

Fig. 3. An example program and its mutant

Table 2. Mutation Table for mutant generated in Fig. 3.

Mutant_Id	Parent Statement	Mutated Statement	Status
M1:5	a>=b-1	a<b-1	unprocessed (means mutant is alive)

The table contains only single entry as only one mutant is generated. In table 2, M1:5 represents mutant M1 corresponding to statement 5 of program in figure 3, (a>=b-1) represents original statement, (a<b-1) represents mutated statement. The value unprocessed in status column specifies that mutant is still alive.

The constraints are generated between the correct statement of the original program and the changed statement of the mutated program such that both the statements generate different outputs. Thus, we generate the following constraint between the original and the mutated statement, mentioned in table 2:

(a >=b-1)! = (a<b-1) i.e. statements a>=b-1 and a<b-1 should produce different outputs

We have made a general assumption that, E1 and E2 represents the expression on LHS and RHS of the ‘not equal to’ operator and ~E1 and ~E2 represents negation of E1 and E2. Thus, for the above identified constraint:

E1: (a >=b-1),   ~E1: (a<b-1),   E2: (a < b-1),   ~E2: (a>=b-1)

Now, to cause E1 and E2 to produce different outputs, we want that either (E1 is true and E2 is false) or (E1 is false and E2 is true). Thus the above constraint can be more formally represented (in DNF) as:

(E1 and ~E2) or (~E1 and E2)

Each of the expression E in the constraints generated above may contain single or multiple conditions.

4.3 Fitness Function Construction

As discussed in section 4.2, solutions to the constraints that are generated represent the effective test case values. We solve constraints using genetic algorithm. Genetic algorithms require fitness function. The fitness function is a measure of goodness of each input value relative to the global optimum solution. It measures how close an input value is towards a global optimum solution. The generation of fitness function is a very crucial step. An effective and efficient fitness function cause genetic algorithm to perform better. Generation of fitness function is a problem dependent activity. Once a fitness function is generated and provided as an input to the genetic algorithm, the algorithm then works independent of the underlying problem.

In this technique, we construct fitness function from the generated constraints, using an appropriate fitness function concept. The fitness function concept that we use here is same as proposed in [4]. This concept is actually based on calculating the difference between values of program variables, such that the difference is minimized. Our aim is to generate test case values that can minimize the value of fitness function.

Table 3 represents the fitness function concept (as proposed in [4]) and is shown below:

Adhering to the effectiveness of the fitness function concept proposed in [4], we have used the same here. We can use both min as well as max fitness function. But since we are using GA Optimization Tool of MATLAB as our genetic algorithm software, we are using min. fitness function. This is so because GA tool is implicitly configured for minimizing the fitness function value. However, if still max fitness function is to be used, we can easily do that by generating the fitness function and taking its negative.

Steps to construct fitness function from the constraints:

1. Compute the fitness values for each of E1, ~E1, E2, ~E2 (identified in section 4.2) using

Table 3. Fitness function [4]

Field No.	C (Condition)	f (C) : fitness value
1	a == b	abs(a - b)
2	a ! = b	- abs(a - b)
3	a > b	(b - a)
4	a < b	(a - b)
5	C1 and C2 / C1 && C2	f(C1) + f(C2)
6	C1 or C2 / C1 || C2	min{f(C1), f(C2)}

table 3.

2. Generate a fitness function for the entire constraint by combining fitness values computed in the first step using table 3.

For the example program in figure 3, the following constraint was identified in section 4.2:
(E1 and ~E2) or (~E1 and E2) where, E1: a >= b-1, E2: a < b-1, ~E1: a<b-1, ~E2: a>=b-1

Referring to the steps mentioned in section 4.3, the fitness function can be constructed from the above constraint as follows:

1. Compute fitness values for E1, ~E1, E2, ~E2 as:
f(E1)=b-1-a, f(E2)=a-(b-1), f(~E1)=a-(b-1), f(~E2)=b-1-a.
2. Generate fitness function by combining the fitness values as:
f: min{[(b-1-a) + (b-1-a)], [(a-(b-1)) + (a-(b-1))]} (refer to row 5 and 6 of table 3)

4.4 GA Optimization Tool Settings

- Genetic Algorithm Software: GA Optimization Tool of MATLAB
- Representation Scheme: Double Vector
- Selection Technique: Roulette Wheel
- Cross over Technique: Single Point Cross Over
- Cross over Rate: 80% (or 0.8)
- Mutation Rate: 0.01
- Initial Population Size: 100
- Maximum no. of Generations: 100

4.5 Proposed Technique Steps

1. Identify Program Mutants. (Refer to section 3.2.1 for rules for identifying mutants.)
2. Build a mutation table to maintain a record of all generated mutants. Initialize the status field of the table to 'unprocessed' (means mutant is alive) for all the generated mutants. (Refer to table 1 for building mutation table.)
3. Select one unprocessed mutant from mutation table. Generate a constraint for this mutant. Build a fitness function from this constraint. (Refer to section 4.2 and 4.3 for constraint and fitness function generation.)
4. Provide this fitness function to GA tool of MATLAB to generate test data (Refer to section 4.4 for GA Tool settings). Check if this test data can kill other mutants along with the cur-

rent mutant for which fitness function was generated. If yes, all these mutants are killed and there is no need to generate test data for these mutants separately. Mark all these mutants in the mutation table as processed. (The check is performed by inputting the generated test data to mutants other than the mutant for which test data is generated).

5. Repeat steps 3 and 4, until all the mutants in mutation table are marked as processed (means mutant is killed).

This technique fits well to any number of inputs. There are no bounds on the number of input variables. The only requirement is to properly generate the constraints and to use them to construct the fitness function in terms of input variables. For e.g. we can easily generate test cases for a program 'to calculate number of days between two dates', that requires six input variables. (The results are shown in table 8 and table 9 in section 5).

4.6 Example using Triangle Classification Problem

We use triangle classification program as an example to explain the working of the proposed technique.

4.6.1 Triangle Classification Program:

The triangle classification program is used to classify a triangle as equilateral, isosceles, or scalene. The input of the program is a triple of three positive numeric variables, say (a,b,c). Each of these variables has some range of allowed values. The output of the program is one of these: 'equilateral triangle', 'isosceles triangle', 'scalene triangle', 'invalid input values entered', or 'values out of range'. The program is shown below:

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main()
4. {
5.      int a, b, c;
6.      printf("Enter the sides of the triangle");
7.      scanf("%d %d %d", &a, &b, &c);
8.      if ((a>0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100))
9.      {
10.             If (((a + b)>c) && ((b + c)>a) && ((c + a)>b))
11.             {
12.                     if ((a == b) && (b == c))
13.                     {
14.                             printf("Equilateral Triangle");
15.                     }
16.                     else if ((a == b) || (b == c) || ( c == a))
17.                     {
18.                             printf("Isosceles Triangle");
19.                     }
20.                     else
21.                     {
22.                             printf("Scalene Triangle");
```

```
23.                                 }
24.                     }
25.                     else
26.                     {
27.                                 printf("Invalid sides of a triangle");
28.                     }
29.         }
30.         else
31.         {
32.                     printf("Input values out of range");
33.         }
34. getch();
35. }
```

4.6.2 Control Flow Graph:

The control flow graph for the program in section 4.6.1 is shown in figure 4.

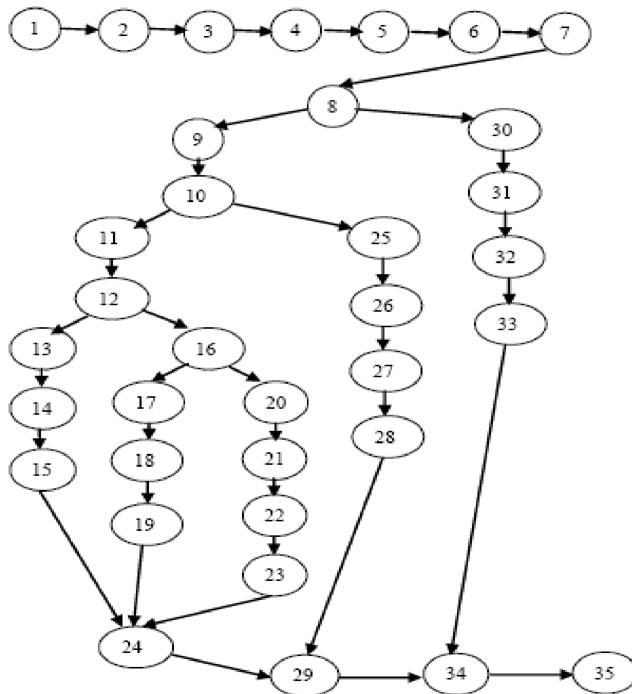

Fig. 4. Control flow graph for the triangle classification program

4.6.3 Mutant Generation

Let us suppose we generate mutants corresponding to statements 8, 10, and 12 of the program described in section 4.6.1. The following mutation table shown in table 4 represents these mutants.

Though only one of the conditions of the entire statement is mutated, we consider the complete statement while generating constraints. In column 2 of table 4, we have mutated variable ‘a’ for mutant id 8. But this is not a restriction. We can mutate any variable among a, b or c. In fact, multiple mutants can be generated corresponding to a single mutant id. The only thing to remember is that the mutants should be generated following the rules for mutant generation (refer to section 3.2.1). The same thing applies well to mutant id 10 as well as 12 of table 4.

4.6.4 Constraint Generation

For Mutant_Id 8:
Let E1: [**(a>0)** && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)] and
E2: [**(a==0)** && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)]

Constraint: E1 ! = E2 or [(E1 and ~E2) or (~E1 and E2)]
Similarly, the constraints can be generated for the other 2 mutants.

4.6.5 Fitness Function Generation

1. Compute fitness values for E1, ~E1, E2, ~E2.
E1 has 6 conditions. For E1 to be true, all the 6 conditions needs to be true as these are connected via && (Boolean ‘and’) operator. Let these 6 conditions are represented as:
E11: (a>0), E12: (a<=100), E13: (b>0), E14: (b<=100), E15: (c>0), E16: (c<=100)
Since all these conditions need to be true, we compute the following fitness values using table3:
f(E11)= -a, f(E12)= a - 100, f(E13)= -b, f(E14)= b - 100, f(E15)= -c, f(E16)= c – 100
→f(E1) = f(E11) + f(E12) + f(E13) + f(E14) + f(E15) + f(E16) (refer to row 5 of table 3)
→f(E1)= (-a) + (a - 100) + (-b) + (b - 100) + (-c) + (c - 100)

Similarly, we can compute value of f(~E1), f(E2), f(~E2). The values are:
→f(~E1) = min{a, (100 - a), b, (100 - b), c, (100 - c)}
→f(E2) = abs(a – 0) + (a - 100) + (-b) + (b - 100) + (-c) + (c - 100)
→f(~E2) = min{-abs(a - 0), (100 - a), b, (100 - b), c, (100 - c)}

2. Generate fitness function by combining the fitness values (computed in above step).
f = min {(f(E1) + f(~E2)), (f(~E1) + f(E2))}
f = min { [{(-a) + (a - 100) + (-b) + (b - 100) + (-c) + (c - 100)} + { min{-abs(a - 0), (100 - a),

Table 4. Mutation Table for the program described in section 4.6.1

Mutant_Id	Parent Statement	Mutated Statement	Status
M1:8	[(a>0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)]	[(a==0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)]	unprocessed
M2:10	[((a + b)>c) && ((b + c)>a) && ((c + a)>b)]	[((a + b)<=c) && ((b + c)>a) && ((c + a)>b)]	unprocessed
M3:12	[(a == b) && (b == c)]	[(a > b) && (b == c)]	unprocessed

Table 5. Fitness function for the mutants identified in Table 4.

Mutant_Id	Fitness Function
M1:8	min { [{(-a) + (a - 100) + (-b) + (b - 100) + (-c) + (c - 100)} + { min{-abs(a - 0), (100 - a), b, (100 - b), c, (100 - c)} }], [{min{a, (100 - a), b, (100 - b), c, (100 - c)}} + { abs(a – 0) + (a - 100) + (-b) + (b - 100) + (-c) + (c - 100)}] }
M2:10	min { [{(c - (a + b)) + (b - (c + a)) + (a – (b + c))} + {min {(c – (a + b)), (c + a - b), (b + c - a)} }], [{min {(a + b -c), (c + a - b), (b + c - a)} } + {(a + b - c) + (b – (c + a)) + (a – (b + c))}] }
M3:12	min {[{abs(a - b) + abs(b - c)} + {min{(a - b), -abs(b - c)}}], [{min{-abs(a - b), -abs(b - c)}} + {(b - a) + abs(b - c)}]}

b, (100 - b), c, (100 - c)} }], [{min{a, (100 - a), b, (100 - b), c, (100 - c)}} + { abs(a – 0) + (a - 100) + (-b) + (b - 100) + (-c) + (c - 100)}] }

Similarly, we can compute the fitness function for Mutant_Id 10 and 12. We have computed the fitness functions for these two mutants. The final results for all the 3 mutants are shown in table 5.

5. Experimental Analysis And Results

The performance of this technique is measured for ten real time programs. The proposed technique is compared with path testing technique (discussed in [4]), for these programs. We have chosen path testing technique because path testing strategy can alone detect almost 65% of errors in the program [28]. The ten programs that are chosen are developed using C language and range from 35 to 200 lines of source code. In this work, we use the same fitness function as proposed in [4]. This is so because our aim is to compare the adequacy based test criteria with the reliability based criteria, where both the criteria are GA based only. We use GA tool of MATLAB as our genetic algorithm software. The experimental settings for the GA tool are shown in section 4.4.

The experimental analysis is performed as follows for all the ten programs:

For a program, 5 mutants are generated and 5 possible execution paths are chosen. For each of these, test cases are generated. A total of 10 runs are carried out for each mutant and path. (There is no specific rule to select only 5 mutants or paths. We can select any number of mutants or paths for analysis. But, since there are large number of possible mutants and paths in a program, we have selected only few for the sake of simplicity.) The analysis is done in terms of **(1)** number of test cases that are generated, **(2)** percentage savings obtained in terms of time required in generating test cases for the proposed technique as compared to path testing technique.

To keep the things simple, the complete details of the experiments performed are shown for the triangle classification program only. The results for other programs are presented in subsequent subsections. For the triangle classification program (described in section 4.6.1):

Test Case Generation using the proposed technique: 5 mutants are generated for the triangle classification program. The mutation table for these mutants is shown in table6.

Table 6 describes the 5 mutants identified from the triangle classification program. First 3 rows describe the mutants generated corresponding to statement 8 of the program and last 2 rows describe the mutants generated corresponding to statement 10 of the program. (refer to

Table 6. Mutation table for the mutants identified from the program in section 4.6.1

Mutant_Id	Parent Statement	Mutated Statement	Status
M1:8	[(a>0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)]	[(a==0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)]	unprocessed
M2:8	[(a>0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)]	[(a==0) && (a<=100) &&(b>0) && (b>100) && (c>0) && (c<=100)]	unprocessed
M3:8	[(a>0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c<=100)]	[(a==0) && (a<=100) &&(b>0) && (b<=100) && (c>0) && (c>100)]	unprocessed
M4:10	[((a + b)>c) && ((b + c)>a) && ((c + a)>b)]	[((a + b)<=c) && ((b + c)>a) && ((c + a)>b)]	unprocessed
M5:10	[((a + b)>c) && ((b + c)>a) && ((c + a)>b)]	[((a + b)>c) && ((b + c)>a) && ((c + a)==b)]	unprocessed

section 4.6.1 for triangle classification program). The value unprocessed in the status field (in table 6) signifies that the corresponding mutant is still alive.

Test Case Generation using path testing technique: 5 Execution paths are selected from the control flow graph for the triangle classification program. (Refer to figure 4 for flow graph)

P1: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 24 29 34 35
P2: 1 2 3 4 5 6 7 8 9 10 11 12 16 17 18 19 24 29 34 35
P3: 1 2 3 4 5 6 7 8 9 10 11 12 16 20 21 22 23 24 29 34 35
P4: 1 2 3 4 5 6 7 8 9 10 25 26 27 28 29 34 35
P5: 1 2 3 4 5 6 7 8 30 31 32 33 34 35

The fitness function for the mutants identified in table 6 is constructed using the technique described in section 4.3 and the fitness values for the paths identified above are constructed using the technique discussed in [4]. Once the fitness functions for the mutants and paths are constructed, we input the functions one by one for all mutants and paths to the GA optimization tool of MATLAB and record the test case results. The test case results are shown in table 7.

Cells in table 7 represent the number of test cases that were generated. For e.g. 13 in the first cell indicates that among 100 test values that were generated, there were 13 unique test case values.(100 test values were generated because initial population is set to size 100). The experimental observations showed that test cases generated for mutant M1:8 also killed mutants M2:8 and M3:8, and test cases generated for mutant M4:10 also killed mutant M5:10. Thus we did not generate the test cases for M2:8, M3:8, and M5:10.
The observations results from Table 7 indicate that for the triangle classification program:

- Overall time required to generate test cases for the proposed technique is less than the time required for path testing technique. This is because we constructed the fitness function for only 2 mutants out of 5 mutants, whereas we had to construct fitness function for all the 5 paths. Hence, using adequacy based criteria; **we achieved 60% savings in terms of time as compared to path based testing technique for triangle classification program.**

Table 7. Experimental Results for the mutants (in table 6) and paths

Runs	Mutants			Paths				
	M1:8	M4:10		P1	P2	P3	P4	P5
1	13	3		11	12	12	9	11
2	5	4		10	10	15	14	12
3	2	3		9	14	14	16	13
4	2	4		12	12	12	12	11
5	4	3		14	15	15	12	14
6	3	3		8	9	8	14	10
7	5	3		16	16	17	18	16
8	1	2		12	12	11	11	12
9	2	1		14	14	14	14	15
10	5	2		6	4	4	13	8
Total	42	28		112	118	122	133	122
Average	4.2	2.8		11.2	11.8	12.2	13.3	12.2
Approximate no. of test cases	4	3		11	12	12	13	12
	7			60				

- Number of test cases that are generated in proposed technique is less as compared to path testing technique. This is so because, in our technique, we are assured that all the generated test cases will be adequate. But, in path testing technique, there is no guarantee of the adequacy of all the generated test cases. Some of these test cases may be adequate while others may not. (Though the test data that cover the selected paths may also kill mutants, but in our technique the generated test data set is guaranteed to kill all the identified mutants. On the contrary, the test data set generated in path testing technique is not guaranteed to kill all the identified mutants. It might kill some but may not kill all the identified mutants).

The experiments for other nine programs are performed in the same manner as that performed for triangle classification program. The results for these programs are shown below in subsections 5.1 and 5.2. As mentioned in section 5, the analysis is done in terms of (1) number of test cases that are generated, (2) percentage savings obtained in terms of time required in generating test cases for the proposed technique as compared to path testing technique.

5.1 Experimental Results for Number of Test Cases

The results for the number of test cases that are generated in proposed and path testing technique are shown in table 8.

We have also included the results for triangle classification program in table 8 in order to keep the results for all the programs at one place. The values in table 8 indicates the number of test cases that are obtained by taking the average of values of 10 runs, approximating the average value, and taking the sum of approximate values for all mutants(for the proposed technique) and all paths(for the path testing technique).

The bar chart depicting the percentage reduction achieved in number of test cases in proposed technique as compared to path testing technique is shown in figure 5. The values in figure 5 (showing % reduction in no. of test cases) are calculated as:

Table 8. Number of test cases generated in proposed technique and path testing technique

Program No.	Program	Proposed Technique (Adequacy Based)	Path Testing Technique (Reliability Based)
1	Previous Date Problem	9	26
2	Quadratic Equation Problem	9	13
3	GCD of two numbers	3	9
4	Linear Search	9	25
5	Binary Search	13	20
6	Payroll System	5	11
7	Commission Problem	8	12
8	Computing the median	11	57
9	Number of days between two dates	17	29
10	Triangle Classification program	7	60

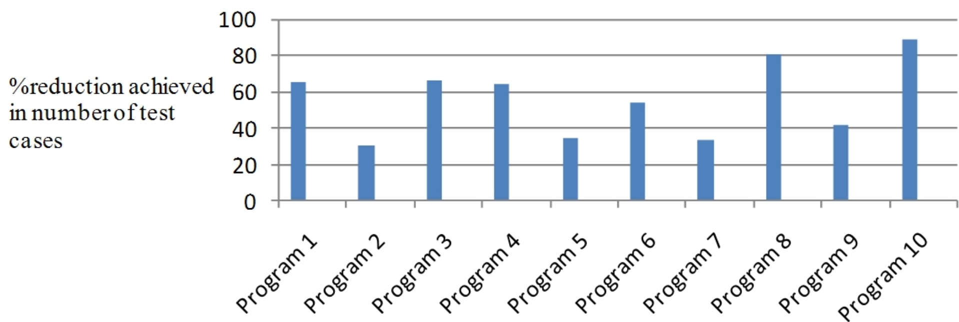

Fig. 5. Percentage reduction achieved in number of test cases

$$\frac{\text{no. of test cases in path testing technique - no. of test cases in proposed technique}}{\text{number of test cases in path testing technique}}$$

5.2 Experimental Results for Percentage Savings in Time

The results for percentage savings obtained in terms of time required in generating test cases for the proposed technique as compared to path testing technique are shown in table 9.

The value 40% in the second row indicates that for the quadratic equation problem, 40% of the total time is saved in generating test cases in proposed technique as compared to path testing technique. This means that out of 5 mutants, we generated test cases for only 3 mutants. The remaining 2 mutants were killed by the test cases generated for the other three mutants. On the contrary, in path testing technique, we had to generate test cases for all 5 paths.

The bar chart depicting the percentage savings obtained in terms of time (taken to generate test cases) in proposed technique as compared to path testing technique is shown in figure 6.

From Table 8 and Figure 5, we observe that number of generated test cases is less in proposed technique as compared to path testing technique and there is 30.76-88.33% reduction in number

Table 9. Percentage savings obtained in terms of time required in generating test cases

Program No.	Program	Proposed Technique (as compared to path testing technique)
1	Previous Date Problem	60%
2	Quadratic Equation Problem	40%
3	GCD of two numbers	60%
4	Linear Search	20%
5	Binary Search	40%
6	Payroll System	40%
7	Commission Problem	60%
8	Computing the median	40%
9	Number of days between two dates	20%
10	Triangle Classification program	60%

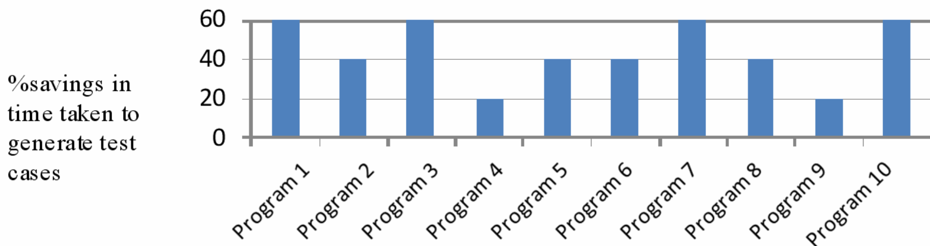

Fig. 6. Percentage savings obtained in terms of time required in generating test cases

of generated test cases in proposed technique as compared to path testing technique. From Table 9 and Figure 6, we observe that we achieved 20%-60% savings in time consumed to generate test cases. Hence the results suggests that the adequacy based proposed technique is better than the reliability based path testing technique and there is a significant reduction in number of generated test cases and time taken to generate test cases.

6. Conclusions

In this paper, we have focused upon two basic aspects i.e. generation of adequate test cases and application of genetic algorithms in generating test cases. We have basically emphasized the generation of test cases so that these are adequate and as a result need not be subjected to mutation analysis after they are generated. This is done by integrating the mutation analysis with the test data generation. We have also applied genetic algorithms in generating the test cases in order to incorporate the benefits of genetic algorithms in test data generation process. Hence, by integrating mutation analysis with test data generation, we reduce the additional effort that is required to guarantee the adequacy of the test cases. Moreover, application of genetic algorithm is also promising as it provides the results that are globally optimum as compared to other local techniques. We have validated our technique using ten real time programs developed in C language. We have also compared our technique that is adequacy based with a path testing tech-

nique that is reliability based for these ten programs.

The experimental results show that there was 30.76%-88.33% reduction in number of test cases and 20% - 60% savings in time (taken to generate test cases) in our technique as compared to path testing technique. Hence, the initial results suggest that there is a significant amount of savings in time of test data generation and significant reduce in number of test cases.

Future directions involve (1) performing and validating the technique on larger projects and comparing it with other techniques so as to assure its efficiency, (2) performing this technique with data types such as string, alphanumeric etc (other than numeric data types). Moreover, we also aim at generating a tool for implementing the proposed technique in order to make the entire process of test data generation completely automated.

REFERENCES

[1] K.K. Aggarwal, Y. Singh, "Software Engineering", New Age International Publishers, 2006.

[2] J.W. Duran, S.C. Ntafos, "An evaluation of random testing", IEEE Trans. on Software Engineering, 10(4): 438–443,1984.

[3] R. DeMillo, A.J. Offutt, "Constraint-based automatic test data generation", IEEE Trans. on Software Engineering 17(9):900–910,1991.

[4] Yong Chen, Yong Zhong, "Automatic Path-oriented Test Data Generation Using a Multi-Population Genetic Algorithm", Fourth International Conference on Natural Computation, 78-0-7695-3304-9/08©2008 IEEE, pp:566-570.

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

[6] B.F. Jones, H.-H. Sthamer, D.E. Eyres, Automatic structural testing using genetic algorithms", Software Engineering Journal September 1996, pp.299-306.

[7] S. Khor, P. Grogono, "Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically", Proceedings of the 19th International Conference on Automated Software Engineering (ASE"04), 1068-3062/04 © IEEE.

[8] A.S. Ghiduk, M.J. Harrold, M.R. Girgis, "Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage", 14th Asia-Pacific Software Engineering Conference, 1530-1362/07 © 2007 IEEE, pp.41-48.

[9] A.S. Andreou, K.A. Economides, A.A. Sofokleous, "An automatic software test-data generation scheme based on data flow criteria and genetic algorithms", Seventh International Conference on Computer and Information Technology, 0-7695-2983-6/07 © 2007 IEEE, pp.867-872.

[10] L. Clarke, "A system to generate test data and symbolically execute programs IEEE Trans. on Software Eng., Vol.SE-2, No.3, pp.215-. 222, Sept. 1976.

[11] B. Korel 1990, "Automated software test generation", IEEE Trans. on Software Engineering 16(8): 870–879.

[12] C.C. Michael, G.E. McGraw, M.A. Schatz, C.C. Walton, "Genetic Algorithms for Dynamic Test Data Generation", 0-8186- 7961-1/97©1997 IEEE, pp.307-308.

[13] J. Wegener, A. Baresel, and H. Sthamer, Evolutionary Test Environment for Automatic Structural Testing, Information and Software Technology, 43:841–854, 2001.

[14] N. Mansour, M. Salame, "Data Generation for Path Testing", Software Quality Journal, 12, 121–136, 2004, Kluwer Academic Publishers.

[15] K. Dahal, A. Hossain, "Test Data Generation from UML State Machine Diagrams using GAs", International Conference on Software Engineering Advances, 0-7695-2937-2/07 © 2007 IEEE. pp:834-840.

[16] A. Bouchachia, "An Immune Genetic Algorithm for Software Test Data Generation", Seventh International Conference on Hybrid Intelligent Systems, 0-7695-2946-1/7 © 2007 IEEE. pp.84-89.

[17] X. Shen, Q. Wang, P. Wang, Bo Zhou, "Automatic Generation of Test Case based on GATS Algo-

rithm", 2007AA04Z148, supported by Nation 863 Project.

[18] P.R. Srivastava, T. Kim, "Application of Genetic Algorithm in Software Testing", International Journal of Software Engineering and Its Applications, Vol. 3, No. 4, October 2009, pp.87-96.

[19] A. Rauf, S. Anwar, "Automated GUI Test Coverage Analysis using GA", 2010 Seventh International Conference on Information Technology, 978-0-7695-3984-3/10 © 2010 IEEE, pp.1057-1062

[20] D.E. Goldberg, Genetic Algorithms: in Search, Optimization & Machine Learning, Addison Wesley, MA. 1989.

[21] Y. Jia, M. Harman, "An Analysis and Survey of the Development of Mutation Testing", IEEE Transactions on Software Engineering, 0098-5589/10/$26.00 © 2010 IEEE.

[22] M. Harman, "Automated Test Data Generation using Search Based Software Engineering", Second International Workshop on Automation of Software Test (AST'07) 0-7695-2971-2/07 $20.00 © 2007IEEE.

[23] M. R Girgis, "Automatic Test Data Generation For Data Flow Testing Using A Genetic Algorithm", Journal of Universal Computer Science, Vol.11, No.6, pp.898-915, June 2005.

[24] P. McMinn, "Search-based software test data generation: A survey", Software Testing, Verification and Reliability, 14(2):105–156, June 2004.

[25] M. Harman, J. Wegener, "Search based testing (keynote)", In 6th Meta heuristics International Conference (MIC 2005),Vienna, Austria, Aug. 2005.

[26] J. W. Laski, B. Korel, "Data flow oriented program testing strategy", IEEE Transactions on Software Engineering, Vol.9, No.3, pp.347-354, 1983.

[27] S. Rapps, E. J. Weyuker, "Selecting software test data using data flow information", IEEE Transactions on Software Engineering, Vol.11, No.4, pp.367-375, 1985.

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

[29] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, K. Karapoulios, "Application of genetic algorithm to software testing", In Proceedings of 5th International Conference on Software Engineering and its Applications Toulouse, France, 1992, pp.625-636.

[30] J. C. Lin, P. L. Yeh, "Automatic test data generation for path testing using Gas", Information Sciences, vol. 131, 2001, pp.47-64.

[31] M. A. Ahmed, I. Hermadi, "GAbased multiple paths test data generator", Computers and Operations Research (2007).

Ruchika Malhotra

She is an assistant professor at the Department of Software Engineering, Delhi Technological University (formerly known as Delhi College of Engineering), Delhi, India. She was an assistant professor at the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Prior to joining the school, she worked as full-time research scholar and received a doctoral research fellowship from the University School of Information Technology, Guru Gobind Singh Indraprastha Delhi, India. She received her master's and doctorate degree in software engineering from the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Her research interests are in software testing, improving software quality, statistical and adaptive prediction models, software metrics, neural nets modeling, and the definition and validation of software metrics. She has published more for than 50 research papers in international journals and conferences. Malhotra can be contacted by e-mail at ruchikamalhotra2004@yahoo.com.

Mohit Garg

He received the B.Tech degree in Computer Science from Guru Gobind Singh Indraprastha University, Delhi, India in 2009. He is now undertaking an M.Tech degree course as a research scholar at Delhi Technological University (formerly known as Delhi College of Engineering), Delhi, India. His areas of interests include Software Engineering, Software Testing, Databases, Data Mining. His research interests are in Search Based Software Engineering, Automated Test Data Generation using Genetic Algorithms, Model Prediction using Genetic Algorithms, Software Quality Analysis using Quality Metrics'. Mohit can be contacted by e-mail at mohit_270488@yahoo.co.in.