ISSN:0975-9646

# A Novel Approach for an Early Test Case Generation using Genetic Algorithm and Dominance Concept based on Use cases

Alekhya Varikuti ,Deepika Puvvula

*Department of Computer Science and Engineering, GITAM University, Visakhapatnam, A.P., India .*

*Abstract*— **This paper presents an integrated approach for generating test cases using Genetic Algorithm and dominance relation with fitness function. We investigated early generation of test cases using use cases and scenarios. In the process of generating test cases first we constructed control flow graph based on that dominance tree is generated. Test cases are generated by applying Genetic algorithm on dominance tree with concepts of crossover and mutation.**

*Keywords*—**Genetic Algorithms, Dominance, Usecase, Scenarios, Testing;**

## I. Introduction

Software testing is used to improve the quality and increase the reliability of software. Software testing is a complex, labor-intensive, and time consuming task that accounts for approximately 50% of the cost of a software system development [1]. Test case generation is the process of identifying input that satisfies test data adequacy criterion.

We used a GA-based technique and dominance concept with a new fitness function that reduces the test requirements and overcomes the existing problems.

Here we are considering use cases and their corresponding scenarios for generating test cases. The goal of testing is to obtain several test cases to check that all the information included in the use cases have been successfully implemented under the test. Test cases are optimized by using genetic algorithm with dominance relation concepts. A control-flow graph in which the nodes represent actions (activity, method execution) and the arcs indicate the flow of control from one action to another. Control flow graph consists of start node and stop node where each node represents statement and anode with two or more outgoing arc is called branch node.

In dominance tree, a node d dominates a node n if every path from the start node to n must go through d. This is written as d dom n. A node x in a control flow graph dominates another node y (x dom y) if every path from r to y contains x. A dominance relation is said to be strict if x y.

## II. ANALYSIS

There has been much previous work in automatically generating test data. Most commonly encountered are random test-data generation, symbolic test-data generation, dynamic test-data generation and test data generation based on GA.

Random test data generation consists of generating inputs at random until useful inputs are found [2,3]. The problem with this approach is clear with complex programs or complex adequacy criteria, an adequate test input may have to satisfy very specific requirements.

Symbolic test data generation consists of assigning symbolic values to variables to create an abstract, mathematical characterization of the program’s functionality. With this approach, test-data generation can be reduced to a problem of solving an algebraic expression. Many test-data generation methods that use symbolic execution to find inputs that satisfy a test requirement proposed [4,5,6,7]. A number of problems arise in symbolic execution like indefinite loops, where the number of iterations depends on non constant expression and the index of array where data is referenced indirectly.

Dynamic test-data generation is based on the idea that if some desired test requirement is not satisfied, data collected during execution can be used to determine which tests come closest to satisfying the requirement. Two limitations are commonly found in dynamic test-data generation systems. First many systems make it difficult to generate tests for large programs because they work only on simplified programming languages. Second, many systems use gradient descent techniques to perform function minimization.

Several search based test data generation techniques have been developed [8,9,10].These techniques are focused on finding test data to satisfy a number of control flow testing and data flow testing. Genetic Algorithm has been the most widely employed in search based optimization techniques. Test data generation methods based on genetic algorithms have many problems due to use of fitness function.

To solve these problems we proposed a new GA-based technique and apply the concepts of dominance with a new fitness function that reduces the test requirements and overcomes the problems of the previous GA-based test-data generation methods.

## III. PROPOSED METHODOLOGY

In the proposed algorithm, the following concepts were introduced for an early test case generation and test case optimization

***A. Use Cases:***

A use case is a specification of actions, including variants, which a system (or other entity) can perform, interacting with an actor of the system. A use case is a specific way of using the system by performing some part of the functionality [11].

***B. Scenarios:***

Scenarios represent different run types and operations of a use case. Each use case there will be one or more scenarios.

***C. Control Flow Graph:***

A control-flow graph (in short flow graph) is a directed graph. The nodes in the graph represent actions (activity, method execution) and the arcs indicate the flow of control from one action to another. A flow graph has two special nodes: the start node and the stop node. The stop node has no outgoing arcs and every node in a flow graph lies on some path from the start node to the stop node (the one-entry one-exit property). A node with one outgoing arc is called an action node. A node with two or more outgoing arcs is called a branch node.

A directed graph or digraph G = (V, E) consists of a set V of nodes or vertices, where each node represents a statement, and a set E of directed edges or arcs, where a directed edge e = (n, m) is an ordered pair of adjacent nodes, called Tail and Head of e, respectively. For a node n in V, in degree (n) is the number of arcs entering and out degree (n) the number of arcs leaving it. Fig 1.a represents control flow graph G for ATM application based on use case and scenarios.

***D. Dominance:***

Let G = (V, E) be a digraph with two distinguished nodes n0 and nk. A node n dominates a node m if every path P from the entry node n0 to m contains n. Several algorithms are given in the literature to find the dominator nodes in a digraph[12]. By applying the dominance relations between the nodes of a digraph G, we can obtain a tree (whose nodes represent the digraph nodes) rooted at n0. This tree is called the dominator tree Fig 1. (b) show dominance tree DT(G) for ATM application based on Control Flow Graph. A (rooted) tree DT (G) = (V, E) is a digraph in which one distinguished node n0, called the root, is the Head of no arcs; every node n except the root n0 is a Head of just one arc and there exists a (unique) path (dominance path) from the root n0 to each node n; we denote this path by dom(n). Tree nodes of out degree zero are called leaves. Here input variables are the functionality of an application.

***E. Principles of Genetic Algorithms:***

The basic concepts of GAs are commonly applied to a variety of problems involving search and optimization. GAs search methods are rooted in the mechanisms of evolution and natural genetics. GAs generates a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms.

The principle behind GAs is that they create and maintain a population of individuals represented by chromosomes (essentially a character string analogous to the chromosomes appearing in DNA). These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to rules of selection, mutation and reproduction. Each individual in the environment (represented by a chromosome) receives a measure of its fitness in the environment. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information) and is analogous to the process of sexual reproduction. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step.

The structure of a simple GA is given below.

```
Simple Genetic Algorithm ()
{
Initialize population;
Evaluate population;
While termination criterion not reached {
Select solutions for next population;
Perform crossover and mutation;
Evaluate population;
 }
}
```

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have occurred.

***F. GA-based Test-Case Generation:***

The proposed GA for test case generation, developed a new GA-based technique and apply the concepts of the dominance relations between nodes of the control flow graph with new fitness function. The algorithm searches for test cases that satisfy the all-statements criterion.

*1) Representation:* The proposed GA uses a binary vector as a chromosome to represent values of the program input variables. The length of the vector depends on the required precision and the domain length for each input variable.

Suppose we wish to generate test cases for a flow graph of k input variables x1,…, xk where each variable xi can take values from a domain Di = [ai, bi].

*2) Initial population:* Each chromosome (as a test case) is represented by a binary string of length m. We randomly generate pop_size m-bit strings to represent the initial population, where pop_size is the population size. The appropriate value of pop_size is experimentally determined. Each chromosome is converted to k decimal numbers representing values of k input variables x1… xk.

*3) Evaluation function:* The algorithm uses a new evaluation (fitness) function to evaluate the generated test data. This new fitness function depends on the concepts of the dominance relations between nodes of the control flow graph. The algorithm uses this new fitness function to evaluate each test case by executing with it as input, and recording the traversed nodes in the program that are covered by this test case. We denote to the set of traversed nodes by exePath. Also, it finds the dominance path dom(n) of the target node n. The fitness function is the ratio of the number of covered nodes of the dominance path of the target node to the total number of nodes of the dominance path of the target node. The fitness value ft(vi) for each chromosome vi (i = 1, …, pop_size) is calculated as follows:

1. Find exePath: the set of the traversed nodes in the program that are covered by a test case.
2. Find dom(n): dominance path of the target node n (the set of dominator nodes from the entry of the dominator tree to n).
3. Determine $(dom(n) - exePath)$ : uncovered nodes of the dominance path (the difference between the dominance path and the traversed nodes).
4. Determine $(dom(n) - exePath)'$ : covered nodes of the dominance path (the complement set of the difference set between the dominance path and the traversed nodes).
5. Calculate $|dom(n) - exePath)'|$ : number of covered nodes of the dominance path (cardinality of the complement set).
6. Calculate $|dom(n)|$ : number of nodes of the dominance path of the target node n (cardinality of the dominance set). Then,

$$ft(v_i) = \frac{|(dom(n) - exePath)'|}{|dom(n)|}$$

The fitness value is the only feedback from the problem for the GA. A test case that is represented by the chromosome vi is optimal if its fitness value $ft(v_i) = 1$

*4) Selection:* After computing the fitness of each test case in the current population, the algorithm selects test cases from all the members of the current population that will be parents of the new population. In the selection process, the GA uses the roulette wheel method.

For the selection of a new population with respect to the probability distribution based on fitness values, a roulette wheel with slots sized according to fitness is used. Such roulette wheel is constructed as follows:

1. Calculate the fitness value $ft(v_i)$ for each chromosome (i=1,...pop_size).
2. Find the total fitness of the population $F = \sum_{i=1}^{pop\_size} ft(v_i)$ .
3. Calculate the relative fitness value $rft$ for each chromosome $rft(v_i) = \frac{ft(v_i)}{F}$ .
4. Calculate the cumulative fitness value $cft$ for each chromosome $cft(v_i) = \begin{cases} rft(v_i) \\ cft(v_{i-1}) + rft(v_i) \end{cases}$

The selection process is based on spinning the roulette wheel pop_size times; each time we select a single chromosome for a new population in the following way:

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

• If $r < cft(v_i)$ then select the first chromosome $v_i$ otherwise select the i-th chromosome $v_i (2 \le i \le pop\_size)$ such that $cft(v_i) \le r < cft(v_{i+1})$.

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

Crossover: It operates at the individual level. During crossover, two parents (chromosomes) exchange substring information (genetic material) at a random position in the chromosome to produce two new strings (offspring). The objective here is to create better population over time by combining material from pairs of (fitter) members from the parent population. Crossover occurs according to a crossover probability. The probability of crossover PXOVER gives us the expected number $PXOVER \times pop\_size$ of chromosomes, which undergo the crossover operation. We proceed in the following way:

For each chromosome in the parent population:

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

• If r < PXOVER then select given chromosome for crossover.

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

Mutation: It is performed on a bit-by-bit basis. Mutation always operates after the crossover operator, and flips each bit with the pre-determined probability. The probability of mutation PMUTATION, gives us the expected number of mutated bits $PMUTATION \times m \times pop\_size$ . Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation (i.e., change from 0 to 1 or vice versa). So we proceed in the following way:

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

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

• If r < PMUTATION then mutate the bit.

In the traditional GA approach the population would evolve until one individual from the whole set which represents the solution is found. In our case, this condition would correspond to finding groups of data items achieving the test requirements (i.e., covering the set of leaves of the dominator tree) of the tested program. We let the population evolves until a combined subset of the population achieves the desired test requirement. The evolution stops when a set of individuals has traversed the dominance path of the test requirement and its fitness value ft(vi) = 1. The solution is this set.

*6) Elitist:* The elitist function enhances the current population by storing the best member of the previous population. If the best member of the current population is worse than the best member of the previous population it exchanges them, and the best member of the current population would replace the worst member of the current population. After that, it stores the best member of the current population.

*7) Algorithm:*

```
/* A GA algorithm to generate test cases */
Input:
The use cases to be tested.
Number of  input variable;
Domain and precision;
Population size;
Maximum no. Of generation (Max_Gen);
Probability of crossover;
Probability of mutation;
Output:
Set of test cases and set of nodes covered by each test case;
List of uncovered nodes,(if any);
Begin
Step0:Setup
1.    Classify the use cases and scenarios
2.    Build the control flow graph based on scenarios of each use case
3.    Build the dominator tree DT
4.    Find the set of leaves L of the dominator tree.
Step1: Intailization
   Initialize the score board to zero;
   nRun←0;
   Set of test cases ← φ
   nCases←0;
Step2: Generate test cases
 For each uncovered node and selected before in the set of nodes to be tested
 Begin
nRun←nRun + 1;
Create Intial_population
Current _population ←Intial_population
No_of_Generation←0;
     For each member of current population do
     Begin
Convert the current chromosome to the corresponding set of decimal values
Execute with the data set as input;
If(the current node is covered) then
Mark the current node as covered;
End If
     End For
     Keep the best member of the current population;
     While (current node is not covered and no_of_Generations ≤  Max_Gen) do
     Begin
Select set of parents of new population from members of current population using roulette wheel method;
Create New_population using crossover and mutation operators;
Current_population ←New_population;
      For each member of Current_population do
      Begin
Convert current chromosome to the corresponding set of decimal values;
Execute with the data set as input;
Evaluate the current test case;
If(current node are covered) then
Mark the current node as covered;
      End if;
      End for;
   Elitist function: If the best member of the current population is worse than the best   member of the previous population then exchange them, and the best member of the current population would be replace the worst member of the population.
      Increment No_of_Generation;
      End While;
If (The current node is covered) Then
nCase←nCases+1;
Add this test case to set of test cases for p;
Update the score board;
Check all uncovered nodes by this test case.
End If
End For;
Step3: Produce Output
Return no of test cases for use cases and set of nodes   covered by test case;
Report on uncovered node, If say;
```

## IV. CASE STUDY

We consider an ATM application and applied the algorithm. Initially, we consider use case and scenarios of ATM application as shown in Table 1. Based on use case and scenarios we construct the control flow graph. Based on control flow graph we construct dominance tree and apply genetic algorithm to generate test cases. The uses of GA parameters were as follows: Maximum Generation=100, Pxover=0.8, Pmutation=0.15.

Table 1:Usecases and Scenarios of ATM Application.

| Usecases | Scenarios |
|---|---|
| Insert card validation | • Wrongly inserted card<br>• card is not valid<br>• valid card |
| Language selection process | • language selected<br>• cancel |
| ATM PIN validation | • Valid PIN<br>• Invalid PIN and check thrice<br>• Cancel |
| Selection Menu Validation | • With draw<br>• Transfer<br>• Balance Enquiry |
| With draw | • collect cash<br>• not sufficient amount in ATM<br>• not Sufficient amount in account<br>• limit exceed per day |
| Transfer | • valid destination<br>• not sufficient amount in source account |
| Balance Details | • Balance enquiry<br>• Mini statement<br>• cancel |
| Transaction Slip | • possible<br>• Paper not available |

Fig. 1. (a) Control Flow Graph G

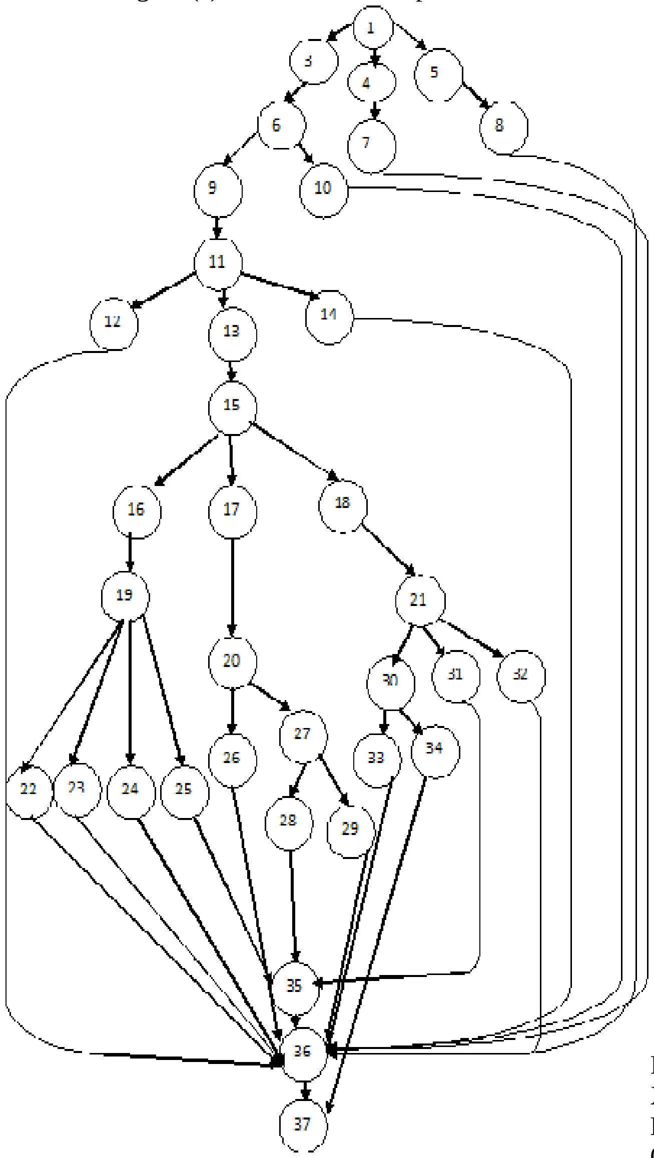
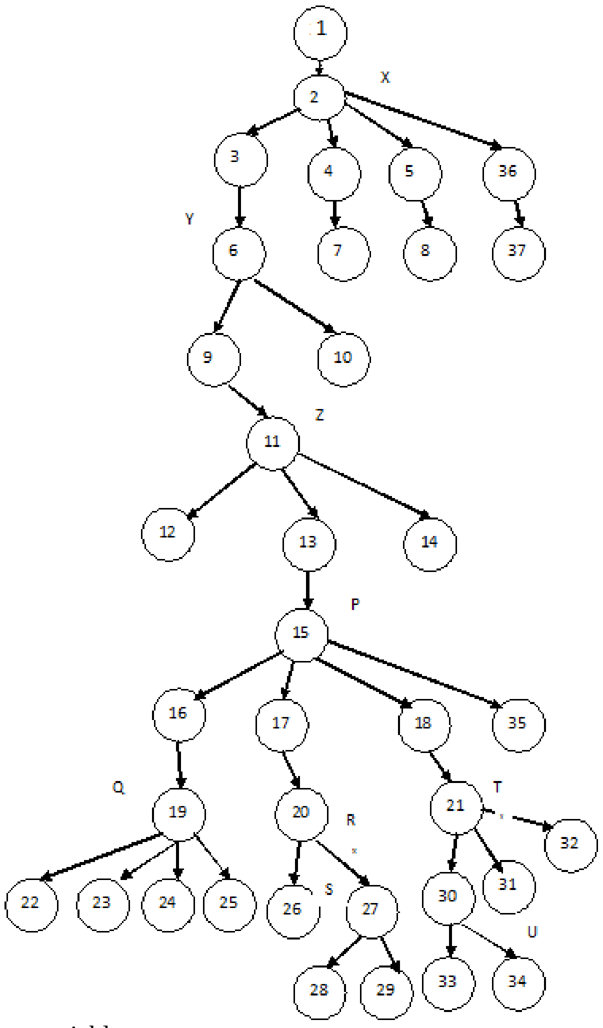


Fig 1 (b) Dominance tree DT(G)



Here X represent Insert Card Validation.

Y represent language selection

Z represent ATM Pin Validation

P represents Selection Menu Validation

Q represent Withdraw option

R represent Transfer

S represent Destination Validation

T Balance Details

U Transaction slip

Input variables:
X, Y, Z, P, Q, R, S, T, U.
Domain:
0..3,0..1,0..2,0..3,0..3,0..1,0..1,0..2,0..1.
Domain tree leaf nodes:
Dom (7) =1, 2, 4, 7.
Dom (8) =1, 2, 5, 8.
Dom (37) =1, 2, 36, 37.
Dom (10) =1, 2, 3,6,10.
Dom (12) =1, 2, 3,6,9,11,12.
Dom (14) =1, 2, 3,6,9,11,14.
Dom (22) =1, 2,3,6,9,11,13,15,16,19,22.
Dom(23)=1,2,3,6,9,11,13,15,16,19,23.
Dom(24)=1,2,3,6,9,11,13,15,16,19,24.
Dom(25)=1,2,3,6,9,11,13,15,16,19,25.
Dom(26)=1,2,3,6,9,11,13,15,17,20,26.
Dom (28) =1, 2, 3, 6,9,11,13,15,17,20,27,28.
Dom (29) = 1, 2, 3, 6,9,11,13,15,17,20,27,29.
Dom (33) = 1, 2, 3, 6,9,11,13,15,18,21,30,33.
Dom (34) = 1, 2, 3, 6,9,11,13,15,18,21,30,34.
Dom(31)= 1,2,3, 6,9,11,13,15,18,21,31.
Dom(32)= 1,2,3, 6,9,11,13,15,18,21,32.
Dom (35) = 1, 2, 3, 6,9,11,13,15,35.

### *A. Results:*

The final report of the result is given in Table-2. The test requirement to be covered is shown and the number of generations in which the test requirement is covered is given. And also the status in which test requirement is covered or not is specified and finally the test cases are generated. Here input variables represent functionality of an application.

#### *1) Final Report:*

Total number of requirement: 18
Number of covered requirement: 18
The Covered Requirement are:7,8,37,10,12,14,22,23,24,25,26,28,29,33,34,31,32,35.
Number of uncovered requirement is: 0
Number of Runs: 18.
Total Number of generation: 25.
Maximum generation: 100
Input Variables : X,Y,Z,P,Q,R,S,T,U.

For example, in Table-2, the test requirement to be covered i.e., dom(7) represents the leaf node of dominance tree. The number of generations is performed until it covers all nodes of dom(7). Based on the domain, we provide values to the input variables (X, Y) that satisfies dom(7) and finally that value becomes the test cases.

## V. CONCLUSION

This paper explains a new GA-based technique and applies the concepts of dominance with a new fitness function that reduces the test requirements. In order to perform early stage test case generation we consider use cases and scenarios. Early stage test case generation help in finding fault detection and reliability of the software.

Table-2 Final Results

| ***Test Requirement to be covered*** | ***Number of generations*** | ***Covered (Yes-Y/No-N)*** | ***Test Case*** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | ***X*** | ***Y*** | ***Z*** | ***P*** | ***Q*** | ***R*** | ***S*** | ***T*** | ***U*** |
| 7 | 1 | Y | 1 | 1 | | | | | | | |
| 8 | 1 | Y | 2 | 0 | | | | | | | |
| 37 | 1 | Y | 3 | 1 | | | | | | | |
| 10 | 2 | Y | 0 | 1 | | | | | | | |
| 12 | 2 | Y | 0 | 0 | 0 | | | | | | |
| 14 | 1 | Y | 0 | 0 | 2 | | | | | | |
| 22 | 2 | Y | 0 | 0 | 1 | 0 | 0 | | | | |
| 23 | 1 | Y | 0 | 0 | 1 | 0 | 1 | | | | |
| 24 | 3 | Y | 0 | 0 | 1 | 0 | 2 | 0 | 1 | 2 | 0 |
| 25 | 1 | Y | 0 | 0 | 1 | 0 | 3 | 0 | 1 | 0 | 1 |
| 26 | 2 | Y | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 2 | 0 |
| 28 | 2 | Y | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 0 |
| 29 | 1 | Y | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 2 | 0 |
| 33 | 1 | Y | 0 | 0 | 1 | 2 | 1 | 1 | 0 | 0 | 0 |
| 34 | 2 | Y | 0 | 0 | 1 | 2 | 3 | 0 | 1 | 0 | 1 |
| 31 | 1 | Y | 0 | 0 | 1 | 2 | 3 | 1 | 0 | 1 | 0 |
| 32 | 1 | Y | 0 | 0 | 1 | 2 | 3 | 1 | 0 | 3 | 0 |
| 35 | 1 | Y | 0 | 0 | 1 | 3 | 0 | 0 | 1 | 3 | 0 |

## VI. REFERENCES

[1] B. Beizer (1990). Software Testing Techniques. Second Edition,Van Nostrand Reinhold, New york

[2] H.D.Mills,M.D.Dyer, and R.C.Linger (1987).Clean room Software Engineering.IEEE Software 4(5), pp. 19-25.

[3] J. M. Voas, L. Morell, and K. W. Miller (1991). Predicting where Faults Can Hide From Testing. IEEE, 8(2), pp. 41-48.384 Informatica 34 (2010) 377–385 A.S. Ghiduk et al.

[4] J. C. King (1976). Symbolic Execution and Program Testing. Communications of the ACM, 19 (7), pp. 385-394.

[5] W. E. Howden (1977). Symbolic Testing and the DISSECT Symbolic Evaluation System. IEEE Transactions on Software Engineering, 3(4), pp. 266-278.

[6] Min Pei, E. D. Goodman, Zongyi Gao, and Kaixiang Zhong (1994). Automated Software Test Data Generation Using a Genetic Algorithm" Technical Report GARAGe of Michigan State University.

[7] M. Roper, I. Maclean, A. Brooks, J. Miller, and M. Wood (1995). Genetic Algorithms and the Automatic Generation of Test Data. Technical report RR/95/195[EFoCS-19-95].

[8] R. P. Pargas, M. J. Harrold, R. R. Peck (1999 ). Test Data Generation Using Genetic Algorithms" Journal of Software Testing, Verifications, and Reliability, vol. 9, pp. 263- 282.

[9] Jin-Cherng Lin and Pu-Lin Yeh (2000). Using Genetic Algorithms for Test Case Generation in Path Testing. Proceedings of the 9th Asian Test Symposium (ATS'00).

[10] M. R. Girgis (2005). Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. Journal of Universal computer Science, vol. 11, no. 5, pp. 898-915..

[11] Mehmet Aksit, Klaas van den Berg, & Pim van den Broek(1995) Use Cases in object oriented software development AMIDST/WP2/N003/V02

[12]T. Lengauer and R. E. Trajan (1979). A Fast Algorithm for Finding Dominators in a Flowgraph. ACM Transactions on programming Languages and Systems, vol. 1, pp. 121-141.

## APPENDIX

A part of the result applying the system to test requirement of dom(10)
Test requirement of Dom (10) = 1 2 3 6 10
Population Size=4
Maximum generation=100
Crossover Probability=0.80
Mutation Probability=0.15
Input Variables: 3
Domain: 0..3,0..1,0..2
Generation-1

```
Input Variable:        X   Y   Z
Individual1 =0, 0, 0 =00   0 00
Individual2 =2, 1, 1 =10   1 01
Individual3 =3, 0, 1=11    0 01
Individual4 =1, 1, 0 =01   1 00
Evaluation of population:
Traversed Path= 1 2 3 6 9 11 12
Uncovered dominator path=10
Fitness Value= 0.4
Traversed Path= 1 2 5 8
Uncovered dominator path= 3 6 10
Fitness Value= 0.4
Traversed Path= 1 2 36 37
Uncovered dominator path= 3 6 10
Fitness Value= 0.4
Traversed Path= 1 2 4 7
Uncovered dominator path= 3 6 10
Fitness Value= 0.4
Generation-2
Selection
Selection is performed by Roulette Wheel depended on fitness
Selected cases of Parents of New Population

Individual1 =0, 0, 0 =00   0 00
Individual2 =2, 1, 1 =10   1 01
Individual3 =1, 1, 1=01    1 01
Individual4 =1, 1, 0 =01   1 00
```

Crossover

| Crossover position | Selection of individual | Before Crossover | Before Crossover |
|---|---|---|---|
| 3 | 1 | 00 0 00 | 00 1 00 |
|  | 4 | 01 1 00 | 01 0 00 |

Mutation

| Individual | before Mutation | Mutation bit | After Mutation |
|---|---|---|---|
| 4 | 01 0     00 | 1 | 11 0 00 |

```
New Population
  Individual 1= 0, 1,0=  00 1 00
  Individual 2= 2, 1, 1= 10 1 01
  Individual 3= 1, 1, 1= 01 1 01
  Individual 4= 3, 0, 0= 11 0 00
Evaluation Of Population
  Individual 1
  Traversed Path= 1 2 3 6 10
  Uncovered Path= -
  Fitness Value= 1.0
  Individual 2
  Traversed Path= 1 2 5 8
  Uncovered Path= 3 6 10
  Fitness Value= 0.4
  Individual 3
  Traversed Path= 1 2 4 7
  Uncovered Path= 3 6 10
  Fitness Value= 0.4
  Individual 4
  Traversed Path= 1 2 36 37
  Best Fitness is: 1.0
Average of fitness: 0.55
No of Generations=2
The Test Requirement is Satisfied and Generated test case is 0, 1, 0 at
        Individual 1
```

## AUTHORS BIOGRAPHY

**Alekhya Varikuti** is pursuing M.Tech in Software Engineering from GITAM UNIVERSITY, Visakhapatnam, A.P., INDIA. Her research areas include Software Testing, Software Quality Assurance, and Software Reliability.

**Deepika Puvvula** is pursuing M.Tech in Software Engineering from GITAM UNIVERSITY, Visakhapatnam, A.P., INDIA. Her research areas include Software Quality Assurance, Software Reliability and Software Estimation Techniques