Automated Test Data Generation Based On Individual Constraints and Boundary Value

Hitesh Tahbildar[1], and Bichitra Kalita[2]

**[1] Computer Engineering and Application Department, Assam Engineering institute
Guwahati, Assam 781003, India**

**[2] Computer Application Department, Assam Engineering College
Guwahati, Assam 781013, India**

Abstract

Testing is an important activity in software development. Unfortunately till today testing is done manually by most of the industry due to high cost and complexity of automation. Automated testing can reduce the cost of software significantly. Automated Software Test Data Generation is an activity that in the course of software testing automatically generates test data for the software under test. Most of the automated test data generation uses constraint solver to generate test data. But it cannot generate test data when the constraints are not solvable. Although method can be found to generate test data even if the constraints are unsolvable, but it is poor in terms of code coverage.

In this paper, we propose a test data generation method to improve test coverage and to avoid the unsolvable constraints problem. Our method is based on the individual constraints and same or dependent variable to create the path table which holds the information about the path traversed by various input test data. For generating unique test data for all the linearly independent feasible path we created equivalence class from the path table on the basis of path traversed by the various input test data. The input data is taken based on individual constraints or boundary value. Our results are compared with cyclomatic complexity and number of possible infeasible paths. The comparison shows the effectiveness of our method.

Keywords: *Independent feasible path, scalability, equivalence class.*

1. Introduction

Automated testing is a good way to cut down time and cost of software development. It is seen that for large software projects 40% to 70% of development time is spent on testing. Therefore automation is very much necessary. Test automation is a process of writing computer programs that can generate test cases, or else test cases need to be generated manually. Automated testing save time, money

and increase test coverage. Software testing tools and technique usually lack in portability, reliability, applicability, and scalability. Till today, there are four approaches of automatic test data generation. They are Random [18, 19, 14], Goal oriented, Intelligent approach, and Path oriented [8, 16, 9]. Random testing is quick and simple but not reliable. Goal oriented approach do not require path selection step but there is difficulty in selecting goal and adequate data. Intelligent approach might face the problems of high computation. The reason behind popularity of path oriented testing is its strongest path coverage criteria. Main problem of path oriented test data generation is infeasible path and complexity of data types [1]. Path oriented testing can be implemented by symbolic execution [20], actual value [16], and combined method [12, 21, 11]. Conventional method of test data generation using symbolic execution collect the path predicate and then solve it with a constraint solver[17]. There are many issues of symbolic execution. Some of them are unsolvable constraint, aliasing, solving for infinite loops, and size of symbolic expression etc. [9]. We cannot generate test data when constraints are not solvable. Another major problem of symbolic execution is detection of path feasibility. There are programs where more than 50% of paths are infeasible.[9]. Avoidance of infeasible path can expedite test data generation process significantly. Xio [5] proposes a constraint prioritization method to generate test data using data sampling score. The method solves prioritized constraints to generate test data for longest feasible path. The method considers longest feasible path i.e. it does not consider any unsolvable constraints as the constraints are collected based on some inputs. But it cannot give the guarantee of better or full coverage. Xio method had chosen the orthogonal data selection method for the input test data. Orthogonal data selection is to select data at the

opposite ends of the range of values [5]. The method overcomes the situation of infeasible path which is one major problem of automated software testing. It gives us guarantee for only one test data as any data satisfies either of the complementary constraints. The major problem of the conventional method of symbolic execution is constraint solver. It removes the difficulties of conventional method of symbolic execution using prioritized constraint instead of using constraint solver. It provides a solution for infeasible path problem but the method is weak in terms of code coverage. The orthogonal selection have some drawbacks. The orthogonal test data selection will not be applicable for those sample programs having single input variable constraint. The path covered by the test data generated by this method covers only one path or a few paths. We propose a method to improve the test coverage and also to avoid constraint solving. The detection of path feasibility [17] is also taken care. Our method is based on the variables involved in the constraints. We are grouping the constraints in such a way that the constraints belongs to the same groups or equivalent class of variables are solvable for all possibilities. That is unsolvable constraints are not put to the same equivalence class of variables. For each equivalent class we generate path based on individual constraints or boundary value.

The paper is organized as follows: Section 2 presents a survey of related works of test data generation methods. Section 3 explains our approach with sample programs. Section 4 explains our experimental results and advantages of our approach of test data generation. Finally in Section 5 we conclude with some observation and future work to be done in test data generation.

2. Survey of related works

The literature of test data generation methods says that the main problem of symbolic execution [10, 9, 20] is constraint solving [22]. Either constraint solving may take more time or some constraints are not solvable. Test case prioritization is used to improve the rate of fault detection specially for regression testing [7, 13, 2]. In [5], constraint prioritization technique with sampling scores method is used to deal with problem of constraint solving. The steps of the Xio methods are construction of control flow graph, finding edge priority, finding complementary pairs, and sample table and sample scores. But this method [5] works only for specific programming constructs. In [19, 18] random testing is cleverly implemented to generate test data. In conventional testing we believe that number of test case should be equal to cyclomatic complexity. McCABE in [23] showed that when actual number of paths tested is compared with cyclomatic complexity, several additional paths are discovered that would normally be overlooked. Ngo and Tan proposed an approach to detect infeasible paths using actual value execution [15]. The method combines empirical properties of infeasible paths with dynamic information collected during test data generation process. Antonia emphasizes on research for improvement of scalability of test data generation algorithms [4].

3. Our Approach

3.1 Steps of Our Approach

To improve the coverage of the sample program which is the major drawback in the method as suggested by [5], we propose a method based on the variables involved in the constraint. The flow graph of our method is shown in figure 1.

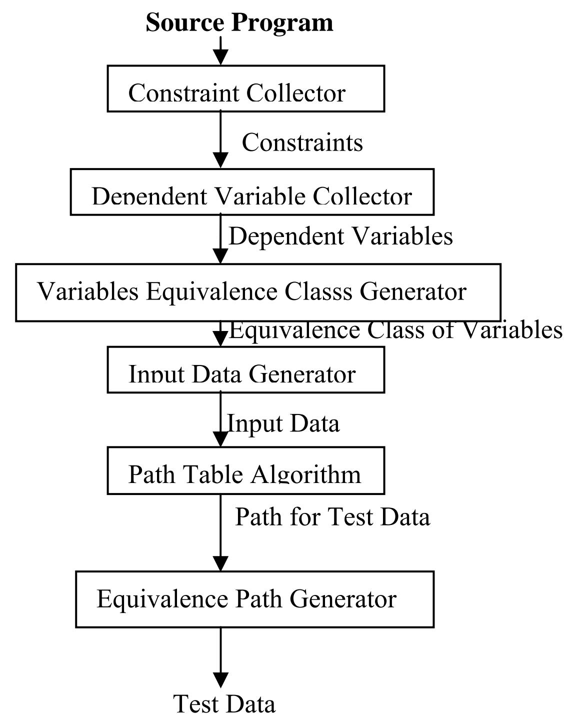


Fig. 1 Steps of our approach.

3.1.1 Constraint Finder

The constraints in the program are found out with the help of this tool.

3.1.2 Dependent Variable Finder

The variables which are either same or dependent on each other are found out with the help of this tool.

3.1.3 Equivalence Class of Variables

Initially the equivalence class is blank. The first constraint is added to the equivalence class. Next the second constraint is checked with the first constraint and if the variable in the second constraint is either same or dependent on the variable of the first constraint then the second constraint is added to the same class or else the second constraint is added to a new class. Similarly for all the constraints the variables are checked with the variables of the constraint already in the equivalence class, to find out if they are either equal or dependent. Then they are added to the same class on which the dependent/same variable constraint is located or else they are added to a new class.

3.1.4 Input Test Data Generator

The input test data is generated on the basis of the equivalence class of variables. Here 3 cases may arise. They are discussed below:-

CASE 1: NOT ALL BUT SOME VARIABLES INVOLVED IN THE CONSTRAINT ARE SAME OR DEPENDENT Some but not all variables involved in the constraint are same or dependent on the variables involved in other constraint. In this case we will at least get more than one equivalence class of variables. As for example the sample program1, the constraints in the sample program are $w1 == 5$, $w1 > 5$, $w1 + w2 >= 8$, $w1 + w2 + w3 >= 12$. Here the two constraint $w1 == 5$ and $w1 > 5$ involves the same variable $w1$ and hence they are included in the same equivalence class and the other two constraints are added in a new class respectively. The control flow graph of sample program 1 is shown in figure 2.

The equivalence class for this sample program 1 is shown in table 1.

Table 1: Equivalence class for sample program 1

Class	*Constraint*
1	w1 == 5 , w1 > 5
2	w1 + w2 >= 8
3	w1 + w2 + w3 >= 12

The test data generated for various possibilities of sample program 1 constraints are shown in table 2 and table 3

CASE 2: ALL VARIABLES INVOLVED IN THE CONSTRAINT ARE SAME OR DEPENDENT The control flow graph of sample program 2 and program 3 is shown in figure 5 and 3 respectively.

The variables involved in each of the constraint may same/

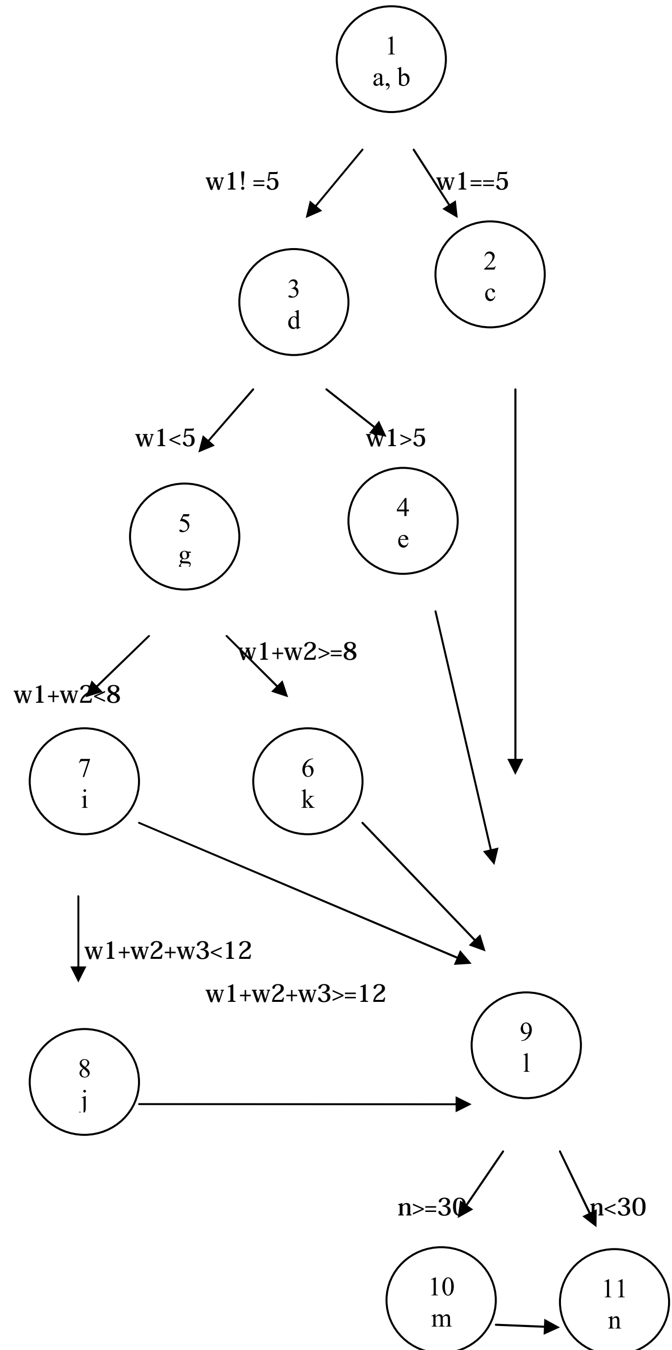


Fig 2: Control flow graph of program 1

Table 2: possible combination of constraints

w1==5	*w1+w2>=8*	*w1+w2+w3>=12*
TRUE	TRUE	TRUE
TRUE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE
FALSE	FALSE	FALSE

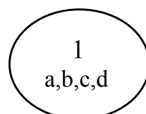
dependent on each other. In this case only one class will be generated in the table of the equivalence class of variables. For this condition the input test data is taken by the boundary value analysis of each of the constraint individually. For example the sample program 3. The constraints in the sample program are *marks* > 100, *marks* >= 50 and *grade*! = "". Here the first two constraints consist of the same variable marks and the variable grade involved in the third constraint depends on the variable marks. The equivalence class of variables for this sample program is shown in table 4.

Thus the input test data will be taken by the boundary analysis of the constraint *marks* > 100, *marks* >= 50 and *grade*! = "" but the variable grade is the output hence the input test is taken considering the boundary analysis of the first two constraint.

CASE 3: INDIVIDUALLY THE VARIABLES INVOLVED IN THE CONSTRAINT ARE NOT SAME OR DEPENDENT BUT COMBINATION OF TWO OR MORE CONSTRAINT MAY MAKE A VARIABLE INVOLVED IN A CONSTRAINT DEPENDENT ON THE COMBINATION

Table 3: possible combination of constraints

w1 > 5	*w1 + w2 >= 8*	*w1 + w2 w3 >=12*
TRUE	TRUE	TRUE
TRUE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE
FALSE	FALSE	FALSE



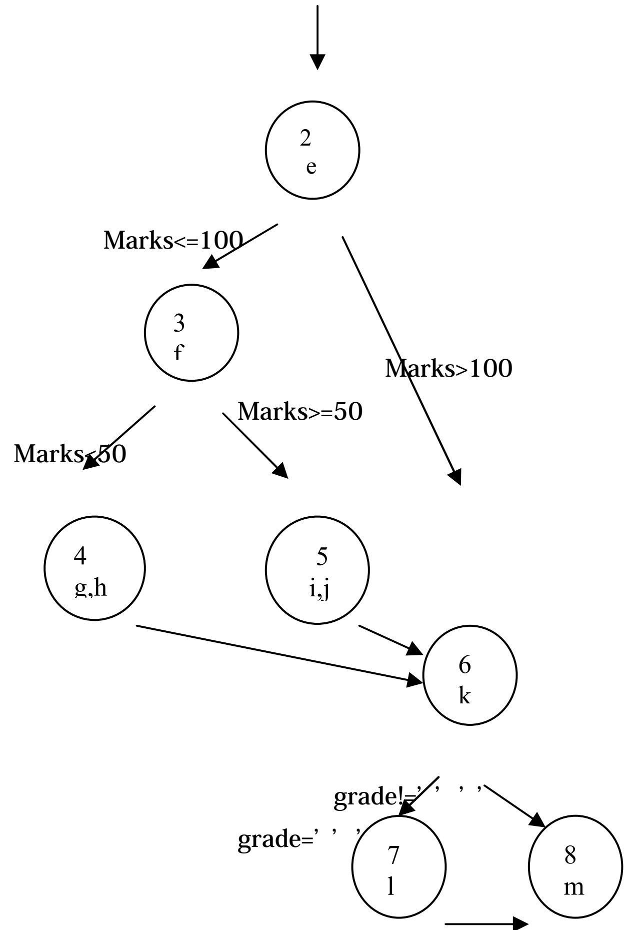


Fig 3: Control flow graph of program 3

This case can be explained with the help of the sample program 4.The control flow graph of sample program 4 is shown in figure 4.

The constraints in the sample program are:- *a > b; a > c; b >c; a == b; b == c* The constraint *a > b* and *a == b* involves the same variable a & b and the constraint *b > c* and *b == c* also involves the same variable *b* and *c*. But the combination of the constraint *a > b* & *a > c* makes the constraint *b > c* & *b == c* dependent on the combination. Similarly But the combination of the constraint *a > b* & *a > c* makes the constraint *a > c* dependent on the combination and also the combination of the constraint *b > c* & *a > c* makes the constraint *a > b* & *a == b* dependent on the combination. Hence the input test data is taken by

solving the constraint combination as follows: - $a > b$ & $a > c$

$a > b\&b > c$

$a > c\&b > c$

$a > c\&b == c$

$a > c\&a == b$

$a > b\&b == c$

$b > c\&a == b$

Table 4: Equivalence class of sample program 3

Class	***Constraint***
1	*Marks>100, marks >=50 and grade! = " "*

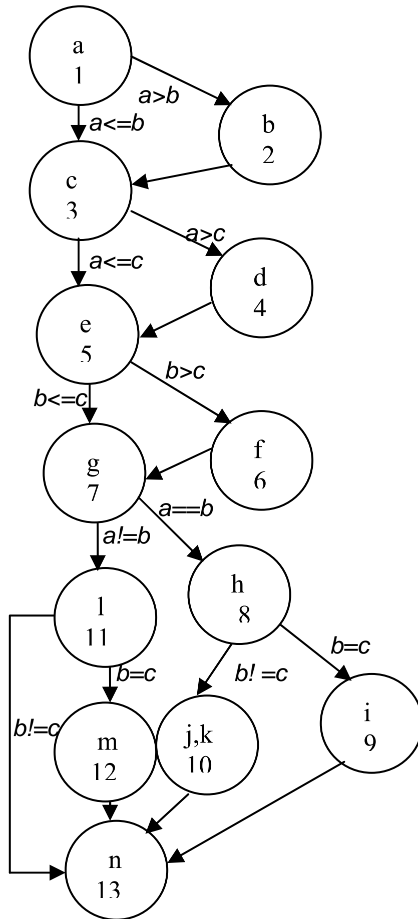


Fig 4: Control flow graph of program 3

3.2 Path Table Algorithm

1. Path = NULL
2. Current node = start
3. If Current node!= End
4. go to step 6
5. else go to step 13
6. Path = Path + Current node
7. if Current node has only one child node then
8. Current node = Child node
9. Otherwise, if constraint at the node is true then
10. Current node = Left Child node
11. Else Current node = Right child node
12. go to step 3
13. Path = Path + End

3.3 Equivalence Path Generation Algorithm

1. For all path generated do the steps from 2 to 7
2. If Equivalence path table is empty then
3. Add the path to Equivalence path table
4. Otherwise, do the steps from 5 to 7
5. Compare the path with all paths in equivalence path table.
6. As soon as a match is found simply discard the path.
7. If no match is found add the path to Equivalence path table.

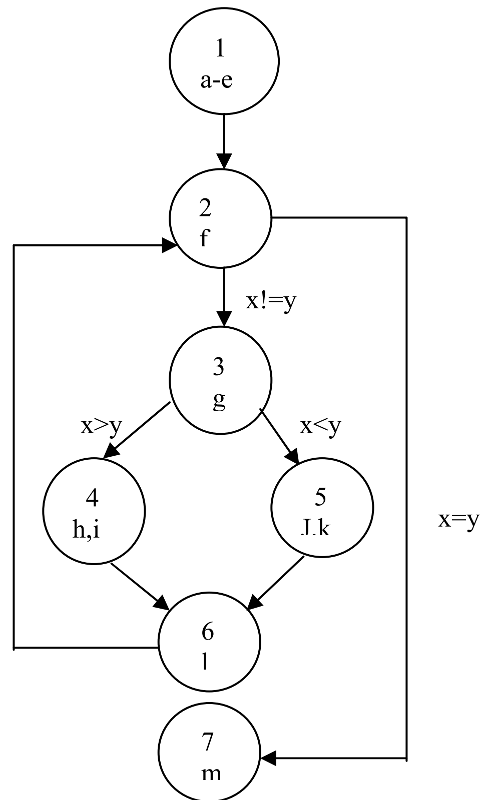


Fig 5: Control flow graph of program 2

3.4 Input Test Data

The input test data is taken based on feasible individual constraints or boundary value of the constraints variables.

4. Experimental Results

4.1 Result of Sample Program 1

The path table for sample program 1 is shown in table 5

4.1. 1 Path Table and equivalence class

The equivalence class and path table for sample program 1 of control flow graph figure 1 is shown in table 6.

The unique feasible paths and test data are shown in table 6

4.2 Experimental Result of Sample Program 2

4.2.1 Path Table and equivalence class

Path table for sample program 2 of control flow graph figure 5 is shown in table 7. Three different paths are generated by our input algorithm. The unique feasible paths and test data are shown in table 8

Table 5: Path Table for sample program 1

w1	*w2*	*w3*	*Path covered*
5	3	4	a,b,e,l,n
5	3	3	a,b,e,l,n
5	2	6	a,b,e,l,n
5	2	3	a,b,e,l,n
4	4	4	a,b,d,g,k,l,m,n
4	4	3	a,b,d,g,k,l,m,n
4	3	6	a,b,d,g,i,j,l,m,n
4	3	4	a,b,d,g,i,l,m,n
6	2	4	a,b,d,e,l,m,n
6	2	3	a,b,d,e,l,m,n
6	1	5	a,b,d,e,l,m,n
6	1	4	a,b,d,e,l,m,n
4	4	4	a,b,d,g,k,l,m,n
4	4	3	a,b,d,g,k,l,m,n
4	3	6	a,b,d,g,i,l,m,n
4	3	4	a,b,d,g,l,m,n

Table 6: Equivalence class and test data

Class	*Path covered*	*value*
1	a,b,c,d,e,l,n	w1=5, w2=3, w3=4
2	a,b,c,d,g,k,l,m,n	w1=4, w2=4, w3=4
3	a,b,d,g,i,j,l,m,n	w1=4, w2=3, w3=6
4	a,b,d,g,i,l,n	w1=4, w2=3, w3=4
5	a,b,d,e,l,m,n	w1=6, w2=2, w3=4

4.3 Experimental Result of Sample Pro-gram 3

4.3.1 Path Table and equivalence class

Path table for sample program of control flow graph figure 3 is shown in table 9. The unique feasible paths and test data are shown in table 10

Thus the final test data's are the data taken from each equivalence class

4.4 Experimental Result of Sample Program 4

The path table of sample program 4 has 32 paths and by forming equivalence class we get 11 unique feasible paths.

4.5 Discussions and Comparison

According to Ngo and Tan [15] the main cause of infeasible program paths is the correlation between some conditional statements along the path. Two conditional statements are correlated if along some paths, the outcome of the latter can be implied from the outcome of the earlier. We compare our results for minimum number of paths covered by our method with cyclomatic complexity and number of infeasible paths generated for programs with the infeasible path detection method proposed by [15].

Table 7: PATH Table for sample program 2

x	*y*	*Path covered*
4	4	a,b,c,d,e,f,m
4	8	a,b,c,d,e,f,g,h,i,l,f
8	4	a,b,c,d,e,f,g,j,k,l,f
10	5	a,b,c,d,e,f,g,h,i,l,f
5	10	a,b,c,d,e,f,g,j,k,l,f
10	10	a,b,c,d,e,f,m

Table 8: Equivalence class and test data

Class	*Path covered*	*Value*
1	a,b,c,d,e,f,m	x=4, y=4
2	a,b,c,d,e,f,g,h,i,l,f	x=8, y=4
3	a,b,c,d,e,f,g,j,k,l,f	x=4, y=8

For the sample program 1
Node1 and Node9 are empirically correlated because
(a) There exist a path from node1 to node9 for the conditional statement w1 == 5 and also for the conditional statement w1! = 5.
(b) node1 which has the statement n = 10 and w1 = = 5 and the node9 has the statement n >= 30. The two nodes are not transitively dependent.
(c) $\delta(node1) = \delta(node9)$ because the conditional statement w1 == 5 depends on the variable w1 and the conditional statement n>= 30 in node9 depends on n and n depends on w1.

Thus either true (node1) and true (node9) is feasible and false (node1) and false (node9) is feasible, or true (node1) and false (node9) is feasible and false (node1) and true (node9) is feasible

Similarly,

node3 and node9 are empirically correlated.
node5 and node9 are empirically correlated
node7 and node9 are empirically correlated

The table 11 shows all the linearly independent paths along with the constraint involved in the path

But according to the infeasibility condition the feasible paths are:-

1) Class A or Class B
2) Class C or Class D
3) Class E or Class F
4) Class G or Class H
5) Class I or Class J

Using [15] we find that in sample program 1 there are 5 in-

Table 9: PATH Table for sample program 3

Marks	***Path Covered***
99	a,b,c,d,e,f,i,j,l,m 100
100	a,b,c,d,e,f,i,j,k,l,m
101	a,b,c,d,e,k,m
49	a,b,c,d,e,f,g,h,k,l,m
50	a,b,c,d,e,f,i,j,l,m
51	a,b,c,d,e,f,i,j,k,l,m

Table 10: Equivalence class and test data

Class	Path covered	Value
1	a,b,c,d,e,f,i,j,k,l,m	99
2	a,b,c,d,e,k,m	101
3	a,b,c,d,e,f,g,h,k,l, m	49

feasible paths. Again the number of equivalence class of the sample program 1 is 5. Hence, our method covers all the linearly independent feasible path of the sample program 1.The cyclomatic complexity of the sample program 1 is 6, and our approach has generated 5 test data.

For the sample program 2 there are no empirically correlated as no pair of nodes satisfy the empirically correlated condition. The table 12 shows all the linearly independent paths along with the constraint involved in the path.

From the table 12 it is seen that there are 3 linearly independent feasible paths and our approach also generates 3 equivalence classes. Hence, our method covers all the linearly independent feasible paths of the sample program 2. The cyclomatic complexity of the sample program 2 is 3, and our approach has generated 3 test data. In sample program 2 there are no infeasible path.

For the sample program 3 node3 and node6 are empirically correlated. Thus either true(node3) and true(node6) is feasible and false(node3) and false(node6) is feasible, or true(node3) and false(node6) is feasible and false(node3) and true(node6) is feasible.

Similarly, there will be three linearly independent feasible path in the program and the number of equivalence classes are also 3. Hence, our method covers all the linearly independent feasible paths of the sample program 3. In sample program 3 there are 50% infeasible paths.

The cyclomatic complexity of the sample program 3 is 4, and our approach has generated 3 test data.

For the sample program 4
There are 32 linearly independent paths. There are 21 infeasible paths. Hence the number of feasible paths are 11 and the number of equivalence classes of our implemented method is also 11. Hence, our method covers all the linearly independent feasible paths of the sample program 11. In

Table 11: Linearly independent path table for sample program 1

Class	***Constraints***	***Path covered***
A	w1==5 n>=30	a,b,d,e,l,m,n
B	w1==5 n<30	a,b,d,e,l,n
C	w1>5 n>=30	a,b,d,e,l,m,n
D	w1>5 n<30	a,b,d,e,l,,n
E	w1! =5 w1<5 w1+w2>=8, n>=30	a,b,d,g,k,l,m,n
F	w1! =5 w1<5 w1+w2>=8, n<30	a,b,d,g,k,l,,n
G	w1! =5 w1<5 w1+w2<8, w1+w2+w3>=12, n>=30	a,b,d,g,i,j,l,m,n
H	w1+w2<8, w1+w2+w3>=12, n<30	a,b,d,g,i,j,l,,n
I	w1! =5 w1<5 w1+w2<8, w1+w2+w3<12, n>=30	a,b,d,g,i,l,m,n
J	w1! =5 w1<5 w1+w2<8, w1+w2+w3<12, n<30	a,b,d,g,i,l,n

Table 12: Linearly independent path table for sample program 2

Class	Condition	Path covered
A	x=y	a,b,c,d,e,f,m

B	x! =y x>y	a,b,c,d,e,f,g,h,i,l,f
C	x! =y x<y	a,b,c,d,e,f,g,j,k,l,f

sample program 4 there are 66% infeasible paths. The cyclomatic complexity of the sample program 4 is 7, and our approach has generated 11 test data.

4.5.1 Advantages of Our Approach

Our approach of dividing the input test data into the equivalence classes gives data for all the linearly independent feasible path. The time and memory complexity is much lower than the [5] method as we have generated the test data without considering the priority table. Thus we have generated reliable test data at a lower cost. The path table does not include the infeasible path and so we have left the infeasible path at the beginning and have not tried to find the test data from the beginning.

5. Conclusions and Future Work

The proposed method has generated test data almost for all the feasible independent paths. As for example for the sample program1, the proposed method has covered the path involving the constraint w1==5 & n < 730, w1>5 & n>=30 , w1+w2>=8 & n>=30, w1+w2+w3>=12 & n>=30 but could not provide test data covering the constraint w1==5 & n>=30, w1>5 & n<30, w1+w2>=8 & n<30, w1+w2+w3>=12 & n<30. But those path which have not been covered by the proposed method are infeasible paths if we go through the body of the program. Similarly, the proposed method covers all the feasible paths of the other sample programs also. For the sample program 3 we have seen that according to Ngo and Tan [15] there are three infeasible paths and the total number of linearly independent path is 6, hence there are three feasible paths and our method also generated three test data for three independent paths. For those types of sample program which do not have infeasible path the proposed method covers all the linearly independent paths as the sample program 2. Coverage is inversely proportional to the number of infeasible paths. Thus the proposed method generates all the feasible linearly independent paths of any program and the presence of infeasible path does not create any problem in generating the test data for testing a program. Though the test data generated by the implemented method is less than the cyclomatic complexity for the sample program 1 and the sample program 3 but the cyclomatic complexity includes both the feasible and infeasible path. Again the cyclomatic complexity gives only the value of minimum number of linearly independent path. But our method generates test data for all the feasible linearly independent path. Hence our method gives more accurate result in compare to the cyclomatic complexity. We describe an approach for test data generation with lower computation cost. The main issue of test data generation using this method is how to take input for finding the paths. We tested our input test data algorithm with 5 different types of programs. Our experimental results shows that this method can generate reliable test data at a lower cost but it is not optimum. The disadvantage of our method is reliability of input while extracting the paths. Since a program may have many numbers of infeasible paths therefore we cannot determine whether number of equivalence class cover minimum number of test to be covered like cyclomatic complexity. Our algorithm for solving path constraints is encouraging. Because the number of equivalence class is less than equal to number of paths. In future we will further research for finding methods of input selection so that it covers every feasible paths of our program. The method should be tested with more examples for accuracy. The method should improve the scalability by including more data types like dynamic data structure and string.

References

[1] Shahid Mahmood,"A Systematic Review of Automated Test Data Generation Techniques", School of Engineering, Blekinge Institute of Technology Box 520 SE-372 25 Ronneby, Sweden, October 2007.

[2] David Godwin Jason Racicot Mechelle Gittens, Keri Romanufa.. "All code coverage is not created equal" A case study in prioritized code coverage." Technical report, IBM Toronto Lab, 2006.

[3] J. Edvardsson,, "Survey on Automatic Test Data Generation," In Proceedings of the Second Conference On Computer Science and Systems Engineering (CCSSE'99),Linkoping, pp. 21-28 10/1999.

[4] Antonia Bertolino, "Software testing research: Achive- ments, challenges, dreams," In Future of software Engineering, 2007.

[5] J. Jenny Li Xio Ma and David m. Weiss., "Pri- oritized constraints with data sampling scores for automatic test data generation." In Eight ACIS In- ternational Confe ence on Software Engineering, articial Intelligence, Network, 2007.

[6] Jon Edvardsson, "A survey on automatic test data gen- eration," In In Proceedings of the Second Con- ference on Computer Science and Engineering in Linkoping, pages 21-28, October 1999.

[7] Chengyum Chu Mary Jean Harrold Gregg Rother- mel, Roland H. Untech., “Test case prioritization: An empiri cal study," In Proceedings of the International Conference on Software Manitenance, Ox- ford, U. K., September 1999.

[8] Tsong Yueh Chen, Fei-Ching Kuo and Zhi Quan Zhou “Teaching Automated test Case Generation," In In Pro-ceedings of the Fifth International Con ference on Quality Software(QSIC'05, IEEE, 2005.

[9] Chen Xu Jian Zhang and Xiaoliang Wang., “Path- ori-ented test data generation using symbolic execution and constraint solving techniques," In Proceedings of the Interna- tional Conference on Software Engineering and Formal Meth- ods, 2004.

[10] Jian Zhang.”Symbolic Exection of Program Paths Involving Pointer and Stucture Variables" Proceedings of the Fourth International Conference on Quality Software(QSIC'04) IEEE ,2004.

[11] H.Tahbildar and B. Kalita. “Automated test data gen-eration for programs having array of variable length and loops with variable number of iteration." In Proceedings of International MultiConference of Engineers and Computer Scientists 2010 VOL I, IMECS 2010, March 17 - 19, 2010, Hong Kong.

[12] Bruno Marre Nicky Williams, Patricia Mouy. “On-the fly generation of k-path tests for c functions. " In Proceedings of the 19th International Conference on Automated Software Engineering, 2004.

[13] J. Jenny Li. "Prioritize code for testing to improve code coverage of complex software." ,2005.

[14] Arnaud Gotlieb and Matthieu Petit “Path-oriented ran-dom testing" Proceeding of the first international Work shop on Random Testing(RT06),July 20,Portland, ME, USA 2006.

[15] Minh Ngoc Ngo *, Hee Beng Kuan Tan, “Heuristics-based infeasible path detection for dynamic test data generation," International Journal Information and Software Technology , ELSEVIER, Page 641655, 2008.

[16] BOGDAN KOREL., ‘ ‘ Automated software test data generation," IEEE Trans. on Software Engineering, (9036267), March 1990.

[17] J. Zhang, Xiaoxu Wang., “A Constraint Solver and its Application to Path Feasibility Analysis," International Journal of Software Engineering and Knowledge Engineering, 11(2): pp. 139-156, 2001.

[18] Koushik Sen.,Darko Marinov, Gul Agha, “CUTE: A Concolic Unit Testing Engine for C" ACM , pp. 5-9, 09 2005.

[19] Patrice Godefroid, Nils Klarlund and Koushik Sen., “DART: Directed Automated Random Testing," PLDI05, June 12-15 2005.

[20] Lori A. Clarke. , “A system to generate test data and symbolically execute programs.," IEEE Trans. On Software Engineering, SE-2(3), Septem-ber 1976.

[21] Bruno Marre Nicky Williams, Patricia Mouy, and Murie Roger. “Pathcrawler: Automatic generation of Path tests by combining static and dynamic analysis." 2005.

[22] Richard A. DeMillo “Constraint-Based Automatic Test Data Generation" IEEE Trans. on Software Engineering, 17(9):900{910, September, 1991.

[23] THOMAS J. McCABE “A Complexity measure" IEEE Trans. on Software Engineering, VOL. SE-2, No-4, DECEMBER 1976.

Annexure I

Sample Program - 1

```
public void monitor (int w1, int w2, int w2)
{
a. int n=10;
b. if(w1==5)
c. n=n*2;
d. elseif(w1>5)
e. n=n*3;
f. else{
g. if(w1+w2>=8)
h. n=n*4;
i. elseif(w1+w2+w3>=12)
j. n=n*5;
k.}
l. if(n>=30)
m. raiseAlarm;
n.}
}
```

Sample Program – 2

```
{
void main()
a. int m, n, x, gcd;
b. scanf("%d", &m);
c.scanf("%d",&n);
d.x=m;
e. y=n;
```

```
f. while(x! =y){
g. if(x>y)
h. {
i. x=x-y;
j. else
k. y=y-x;
l.}
m. gcd=x;
}
```

Sample Program-3

```
{
void main()
a.int marks;
b.char grade[4];
c.scanf("%d", &marks);
d.grad=" ";
e.if(marks<=100)
f.if(marks<50)
g.{
h.grade="Fail";
i.else
j. grade="Pass";
k. if (grade!="")
l.UPDATE THE STUDENT WITH STUDENT ID
RECORD;
m.printf("%s", grade);
}
```

Sample Program-4

```
int tritype (int a, int b, int c)
{
a. if (a > b)
b. swap (a , b);
c. if (a > c)
d. swap (a , c);
e. if (b > c)
f. swap (b , c);
g. if (a = = b)
h. if (b = = c)
i. type = EQUILATERAL;
j. else
k. type = ISOSCELES;
l. else if (b = = c)
m. type = ISOSCELES;
n. return type;
}
```

H. Tahbildar Received his B. E. degree in Computer science and Engineering from Jorhat Engineering College, Dibrugarh university in 1993 and M. Tech degree in Computer and Information Technology from Indian Institute of Technology, Kharagpur in 2000. Presently he is doing Phd and his current research interest is Automated Software Test data generation, Program Analysis. He is working as HOD, Computer Engineering Department, Assam Engineering Institute , Guwahati, INDIA

B. Kalita: Ph.d degree awarded in 2003 in Graph Theory. At present holding the post of Associate Professor, Deptt of Computer Application, Twenty research papers have got published in national and international level related with graph theory, Application graph theory in VLSI design, software testing and theoretical computer science. Field of interest: Graph theory, VLSI Design, Automata theory,, network theory test data generation etc. Associated with the professional bodies, such as Life member of Indian Science Congress association, Life member of Assam Science Society, Life member of Assam Academy of Mathematics, Life member of Shrimanta Sankar deva sangha (a cultural and religious society). Delivered lecture and invited lectures fourteen times in national and international level.

.