Symbolic Execution - An efficient approach for test case generation

S.Vengadeswaran, K.Geetha

Dept. of Computer Science and Engineering

College of EngineeringGuindy, Anna University

Chennai, India

meetvengadesh@gmail.com,kgeetha@cs.annauniv.edu

Abstract—**In the process of software development, testing is the critical activity which consumes more than 50% of resources and time. A challenging step in software testing process is to generate test cases that will sufficiently test the functionality of the software being developed. A technique commonly used to generate the test cases is through branch coverage. Even though the above technique successfully generates test cases for the input program, there are few limitations during which this technique does not prove to be useful. Especially when an infeasible path is encountered in the program under test, the test cases generated does not reflect the reality. To overcome such situation test cases generation through symbolic execution is proposed in this paper.**

Index Terms—**White box testing, Test case generation, Symbolic execution, Infeasible paths, Coverage efficiency.**

I. INTRODUCTION

Software testing is the process of executing a program or system with an intent of finding an error [1]. Testing does not guarantee that there are no faults in the software. Acording to Dijikstra,"Testing shows the presence of bugs, not the absence" [1]. Hence to make the software foolproof exhaustive testing is needed. Since exhaustive test generation is laborious and time consuming it is not possible to perform it manually. The alternative approach is Automatic generation of test cases [2][3][4].

Software testing is completely different from ordinary testing normally encountered, suppose a Cylinder is to be tested for the ultimate pressure it can withstand, the cylinder can be tested with the required ultimate pressure by which it can be inferred that the cylinder is safe for all pressure up to the tested ultimate pressure. But this kind of reasoning does not work for software systems because the software systems are neither linear nor continuous. Also the number of required test cases varies exponentially in direct proportion to the number of inputs, and testing of all possible combinations is very costly.

A major ambiguity in testing is to generate test cases that will sufficiently test the functionality of the software. Most of the test case generation technique falls under two categories, static and dynamic [1].

Static testing[4] is done before compilation and is a verification process which is done without executing the program. Static testing helps in prevention of defects.

Dynamic testing[5] is performed after compilation and is a validation process done by executing the program. Dynamic testing is about finding and fixing the defects. Structural (White box) and behavioural (Black box) testing are the major categories of Dynamic testing.

Structural Testing focuses on the structural part (i.e.) coding / programming part, it is a white-box testing where the internal structure of the software is taken into account to derive the test cases.

Behavioural testing[18] focuses on the functional part of the application. It is a Black box testing where test cases are derived to test the application against the functionality. It does not concentrate on the internal structure of the program.

The most promising technique in unit level test case generation[6] is Branch/ Path coverage. Numerous tools are available for automatic generation of test cases using the above techniques. Most tools consist of Parser which generates the Control Flow Graph for the program under test [7]. From the Control flow graph, the test cases are generated using Test case Generator.

A major limitation in this technique is that the infeasible paths are not properly identified and test cases are generated for such paths also. During testing, these test cases reports error and desired behaviour cannot be achieved. Identifying and rectifying such error will consume unnecessary time and resources. To overcome this problem, an efficient test case generation using Symbolic execution is proposed in this paper.

The paper is organised as follows **Section2**: Definition of Symbolic execution and a motivating example, which clearly describes the necessity and benefits of symbolic execution. **Section 3** includes proposed tool for generating test cases using symbolic execution and describes the process includes in symbolic execution such as construction of execution tree, deriving and solving the path constraints and generation the test cases. **Section 4** includes Tools performance and results. Conclusion and References are given in **Section 5 and 6** respectively.

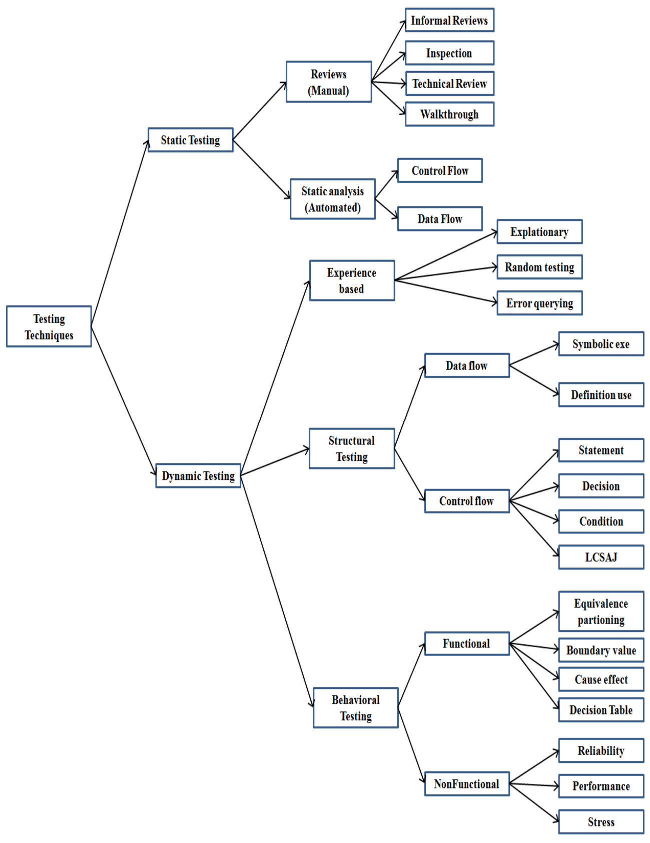


Fig. 1.Taxonomy for test case generation techniques

II. SYMBOLIC EXECUTION

A. *Definitions*

Symbolic execution technique was not a recent development but the technique has received renewed interest in recent years onlydue to high growth in the computing powers. Test case generation through symbolic execution prove to be more efficient with high branch coverage and due to elimination of infeasible paths.

Symbolic execution[8][9]executes a program using symbolic values instead of concrete inputs such as integers, characters etc. The execution proceeds as normal execution except that the output will be symbolic formulas over input symbol.

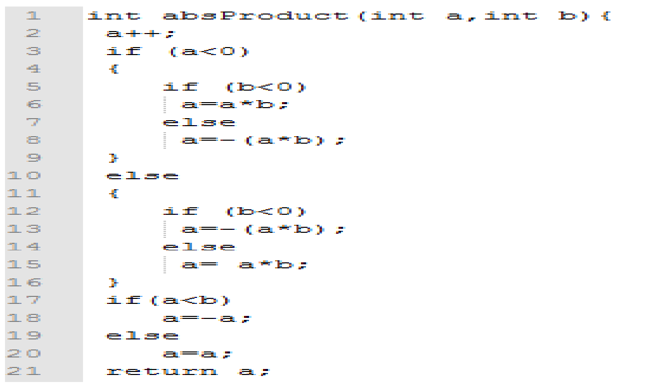
In the process of execution of the program using symbolic execution technique the input variable (solution) to the path will be executed so as to evaluate the constraints of the path. During such execution the input variables are updated at every execution of the path based on the semantics of each of the instructions.

All the constraints of each path are tracked. At the end of the execution of a path, the corresponding Path constraint is fed to a constraint solver.If the solver is able to find a solution for the PC, that solution represents the concrete inputs that lead to the execution of the path being considered[9][10].

This paper concentrates on the symbolic execution approach, as well as path selection method in order to generate test cases which satisfy each path.

B. *Motivating examples*

Let us consider an example which clearly describes the necessity and benefits of symbolic execution.



```
int absProduct(int a,int b){
 a++;
 if (a<0)
 {
    if (b<0)
     a=a*b;
    else
     a=-(a*b);
 }
 else
 {
    if (b<0)
     a=-(a*b);
    else
     a= a*b;
 }
 if(a<b)
    a=-a;
 else
    a=a;
 return a;
```

Fig.2.Sample code fragement

A java program is given as input. An equivalent Control flow graph for the given program is constructed. The Control Flow Graph (CFG)[17] of a program is a model of the program's flow represented by a graph. The CFG uses nodes to represent the program's basic blocks and edges to indicate the flows between the basic blocks. A path is a finite sequence of nodes connected with edges of the CFG.There may be an infinite number of paths for each program but some of them are invalid as there may not be any test cases to execute the program through that path. These paths are infeasible paths. Hence in any of the path based testing technique, good performance is dependent on identification of feasible paths.

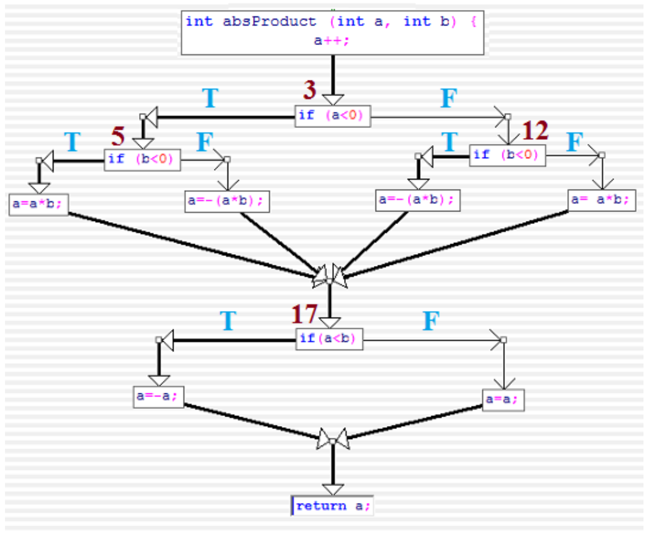


Fig.3. Control flow graph for sample code fragment

The possible control paths and their corresponding path conditions are listed in Table1. One set of possible test cases which are generated by satisfying that corresponding path condition is also listed, but one important ambiguity is It is not possible to follow the Path: **3T 5T 17T, 3T 5F 17T, 3F 12T 17T, 3F 12F 17T**.The test cases generated (a= -4 b= -2), (a= -4 b= 2), (a= 2, b= 4)does not traverse that path. Because the program consist of infeasible path.

This is the program for finding Absolute of productof two numbers, the program consist of infeasible paths byincluding another decision at line 17 by checking the valueof a is less than b, since the value of a is product of a and b in previous steps where a and b being integers, True part of decision (a<b)does not exist.

TABLE I

Control paths and corresponding path conditions when using path coverage

Path coverage			
S.No	***Control Path***	***Path Condition***	***Test Cases***
1	3T 5T 17T	(a<0) (b<0) (a<b)	a= -4 , b= -2
2	3T 5T 17F	(a<0) (b<0) (a>=b)	a= -2 , b= -4
3	3T 5F 17T	(a<0) (b>=0) (a<b)	a= -4, b= 2
4	3T 5F 17F	(a<0) (b>=0) (a>=b)	Infeasible
5	3F 12T 17T	(a>=0) (b<0) (a<b)	Infeasible
6	3F 12T 17F	(a>=0) (b<0) (a>=b)	a= 4 , b= -2
7	3F 12F 17T	(a>=0) (b>=0) (a<b)	a= 2, b= 4
8	3F 12F 17F	(a>=0) (b>=0) (a>=b)	a= 4, b= 2

This is the major limitation in test case generation through branch coverage and other techniques wherein test cases are generated even for infeasible paths without identifying them. This will result in reporting errors, finding and rectifying the errors are confusing, time and resources consuming. To overcome such problems we propose generation of test cases through Symbolic execution, as stated earlier Symbolic execution is execution of program with symbols representingarbitrary values instead of normal inputs. The process executes in the similar way, except the output, in the form of symbolic formulas over the input values.

III. TOOL DESCRIPTION

The tool employs an efficient path selection from symbolic execution tree, integrated with random testing[12] for producing test cases. The tool handles the path selection mechanism and constraint solving problems efficiently. The tool targets on the identified feasible paths and a strategy for linear programming to determine the feasibility of path.

The output result of the tool generated so far gives very interesting results and proves to be an efficient tool by showing that high coverage efficiency is achieved with limited resources of time and cost. First all the methods involved in the program are identified and the parameters involved in each of the method are listed. Then symbols are assigned for each of the input parameters. For the sample program listed in Figure1, the characters 'a' and 'b' are represented as **α**1 and **α**2 in a symbolic execution tree.

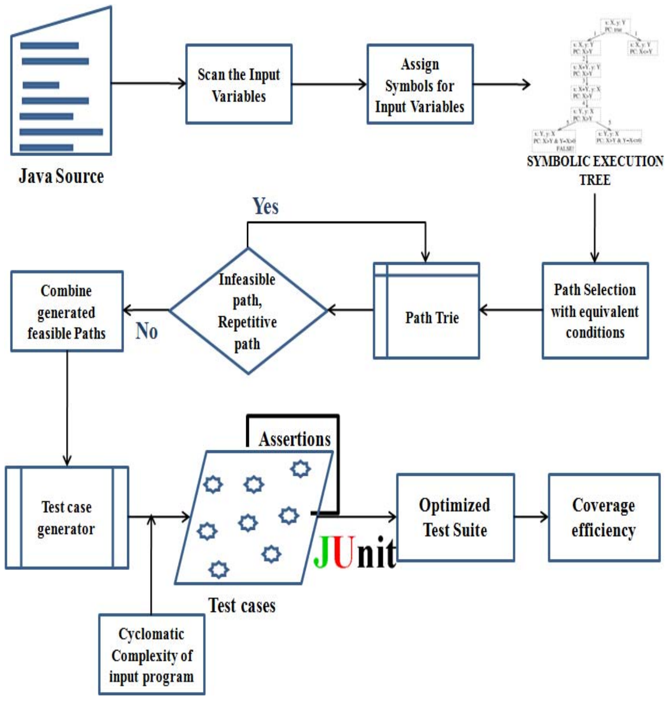

Java Source
Scan the Input Variables
Assign Symbols for Input Variables
SYMBOLIC EXECUTION TREE
Path Selection with equivalent conditions
Path Trie
Yes
Infeasible path, Repetitive path
No
Combine generated feasible Paths
Test case generator
Assertions
Test cases
JUnit
Cyclomatic Complexity of input program
Optimized Test Suite
Coverage efficiency

Fig. 4. Flow diagram for the proposed system

A. Symbolic execution tree

Symbolic execution tree for the program shown in Figure1 is generated in Figure5. On executing the program in Figure1, in line1, the value of 'a' become a+1 i.e. α+1 since 'a' is symbolically represented by α. Then executing line 2 which is a decision node stating (α<0) i.e. α +1<0, True path will be taken when if condition is satisfied. On symbolic representation the path condition becomes α +1<0. Next execution in this path is also a decision node with condition (b<0) i.e. β<0 'b' will take same value β in the True path since it is not varied. The value of variables assumed in each step will be value as updated in the previous step. The next execution in the same path is a statement node where the value of 'a' is updated as a= a*b i.e. a= (α +1)* β, where α +1 and β are both less than zero as per conditions satisfied in this path.

This updated value of 'a' is used for next execution when 'a' is encountered. Regarding the execution through the else part in the decision node in line5 (b<0), the path condition become b>=0 i.e. β>=0. Next execution in this path will be a=-(a*b) i.e. a= -((α +1)* β) where α1+1 and β greater than zero as per conditions satisfied in this path. Next execution for the both the paths will be the decision at line no17 i.e. (a<b) for each of the paths the path condition is execute with varied variables of previous execution of each of the path.

Similar process is repeated for the other conditions and the path is traced with the path conditions to generate symbolic execution tree as shown in the Figure 5. This Symbolic execution tree had a great impact on elimination of infeasible path.

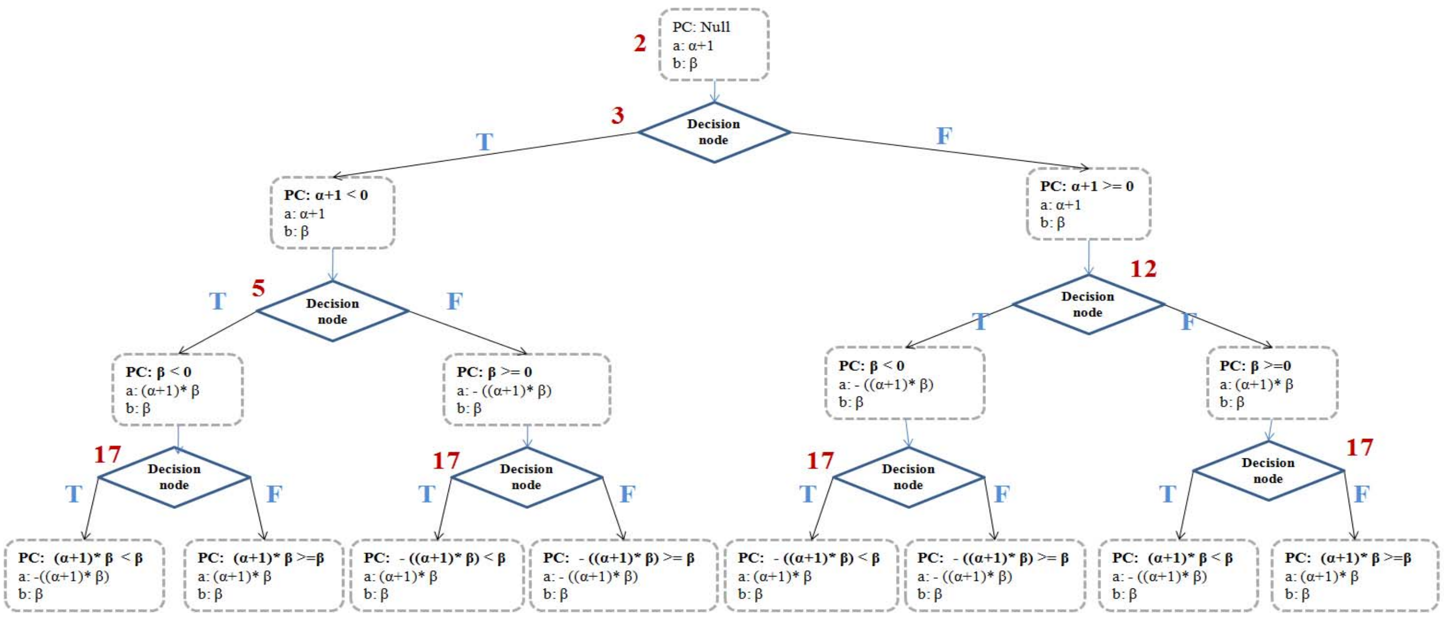


Fig.5. Symbolic execution tree for sample code fragment

B. Path generation

The next important step in the test case generation process is generation of all possible paths traversing through the symbolic execution tree shown in Figure5. Symbolic execution tree is a directed-unweighted graph and several techniques available for path generation of such graph.Somepromising techniques are BFS, DFS [13].Tackling the path explosion problem and selecting high quality test cases are considered as major challenges [21].

In this paper, path generation is executed through McCabe's algorithm to find basis set of path.

- The input will be the Symbolic Execution Tree.
- Baseline path which is the normal execution of the program is identified at the first instance.
- Then the Algorithm proceeds by retracing the paths visited and flipping the conditions one at a time.
- The process repeats up to all flips done.

TABLE II

Control paths and corresponding path conditions when using Symbolic execution

Symbolic execution		
Control Path	***Path Condition***	***Test Cases***
3T 5T 17T	(α+1 < 0) (β< 0) ((α+1)*β<β)	Infeasible
3T 5T 17F	(α+1 <0) (β<0) ((α+1)*β>=β)	a= -5 , b= -9
3T 5F 17T	(α+1>= 0) (β>=0) (-((α+1)*β)<β)	Infeasible
3T 5F 17F	(α+1>= 0) (β>=0) (-((α+1)*β)>=β)	a= -1 , b=0
3F 12T 17T	(α+1>= 0) (β<0) (-((α+1)*β) <β)	Infeasible
3F 12T 17F	(α+1>= 0) (β<0) (-((α+1)*β) >=β)	a= 1 , b= -1
3F 12F 17T	(α+1>= 0) (β>=0) ((α+1)*β<β)	Infeasible
3F 12F 17F	(α+1>= 0) (β>=0) ((α+1)*β>=β)	a=8 , b=9

C. Solving path constraints

The paths generated with their corresponding path condition are stored in a database named Path trie. Then each path, have to be evaluated for the branch conditions in order to check their feasibility. The path condition for each possible path has to be evaluatedi.e. the instructions being executed and keeping track of the constraints that the inputs must satisfy for tracing that path[10][20].

Consider the path **3T 5T 17T** generated in table2 holds the conditions 1. **(α+1<0), 2.(β< 0), 3.((α+1)*β<β)**This shows all the above conditions has to be satisfied for traversing through the path**3T 5T 17T** but all cannot be simultaneously true for any value of **α**1 and **α2**. Hence no test cases are generated to satisfy all the condition and hence the path is shown as infeasible. Then infeasible paths are removed from the database.Then test case generator generates random values for the feasible path constraints.

D. Cyclomatic complexity

Cyclomatic complexity is a software metric used to indicate the complexity of a program [14]. Cyclomatic complexity is calculated by any of the following formula.

$$\text{Cyclomatic complexity CCN} = \text{E} - \text{N} + 2\text{P}$$

Where E and N are the number of edges and nodes in the control flow graph, respectively.

The cyclomatic complexity also gives the maximum number of test cases required to achieve complete branch coverage; and minimum number of test cases required for complete path coverage [14].

It can be assumed that the total number of test cases for complete path coverage will be equal to number of paths in a graph that can be traversed. This idealsituationdoes not normally result due to the fact that some paths are often infeasible. Hence the actual possible paths are sometimes less than CCN, due to this infeasible path.

$$\text{" Branch coverage} \leq \text{CCN} \leq \text{Path coverage "}$$

IV. PERFORMANCE EVALUATION

A. Evaluation parameters

The proposed technique has been tested with real cases of four benchmarking sample programs; the program includes the triangle classifier program which is widely used in various research papers in the test case generation. The results of triangle classifier problem are given Table1.This approach solves the ambiguity in triangle classifier problem. The other programs also have chosen where the interesting results can be achieved.

The elimination of infeasible paths and reductionin false positive rate is taken into account for evaluation. Consider the Path **3T 5F 17F** in the Table1, holds the condition **(a<0) (b>=0) (a>=b),** Test cases should be generated by satisfying all the three conditions, but it is not possible to satisfy all the three conditions simultaneously, hence this path is considered as **"infeasible"**, when using branch coverage technique. but actually it does not, in symbolic execution technique for the same path holds the condition**(α+1>= 0) (β>=0) (-((α+1)*β)>=β)** from Table2, it's now possible to generates the test cases for that path also and thereby shows more true positives compared with other techniques.

Coverage efficiency is one important evaluation parameter which gives clear benefits [19]. Several tools [15][16] are available to calculate the coverage efficiency. It is measured as

$$\text{Path coverage \%} = \frac{\text{Number of paths executed}}{\text{Total number of paths}} \text{X } 100$$

$$\text{Statement coverage \%} = \frac{\text{Number of lines executed}}{\text{Total number of lines}} \text{X}100$$

$$\text{Cyclomatic complexity CCN} = \begin{matrix} \text{E} - \text{N} + 2\text{P} \\ \text{(or)} \\ \text{Number of decision points} + 1 \end{matrix}$$

B. Results

The results obtained are encouraging and Symbolic execution technique performs better in elimination of infeasible paths, also the coverage efficiency is far higher. Cyclomatic complexityis considered to ensure the minimum test cases required to ensure the program. Our approach does not generate more test cases than Cyclomatic complexity. It also helps in infinite loop. Table 3 and Table 4 in Results (B) showsobtained valuefor sample set of programs. Here the coverage value is calculated using EMMA Code coverage tool [15].Test coverage will be further improved if concolic testing is combined with the symbolicexecution[22].

TABLE III

Comparison results between symbolic execution and branch coverage for set of sample programs

Input Program			**Branch coverage**		**Symbolic execution**	
Sample program	***No. of Branches***	***No of Path***	***Infeasible paths***	***False positive***	***Infeasible paths***	***False positive***
Sample1	4	8	1	1	2	0
Sample2	6	7	1	0	2	0
Sample3	5	16	6	0	8	0
Sample4	4	9	2	1	3	0

TABLE IV
Number of feasible paths, test cases and coverage efficiency for each test case during Symbolic execution

Symbolic execution based test case generation					
Sample program	***CCN***	***No of Path***	***No of feasible paths***	***No .of Test cases***	***Coverage efficiency***
Sample1	8	8	6	6	94.8%
Sample2	7	7	5	5	100%
Sample3	16	16	8	8	95.5%
Sample4	9	9	6	6	98%

TABLE V
Coverage efficiency for samples using different approaches

Path coverage			
Sample program	***Symbolic execution***	***Branch coverage***	***Random testing***
Sample1	94.8%	88%	86%
Sample2	100%	83.5%	85%
Sample3	95.5%	90%	86%
Sample4	98%	82%	80.5%

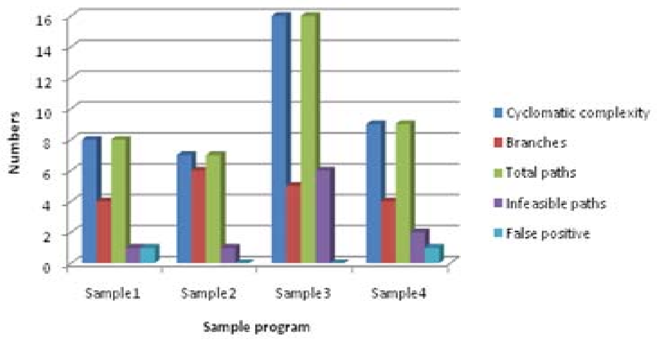


Fig. 6. Number of infeasible paths and false positive for set of sample programs during Branch coverage.

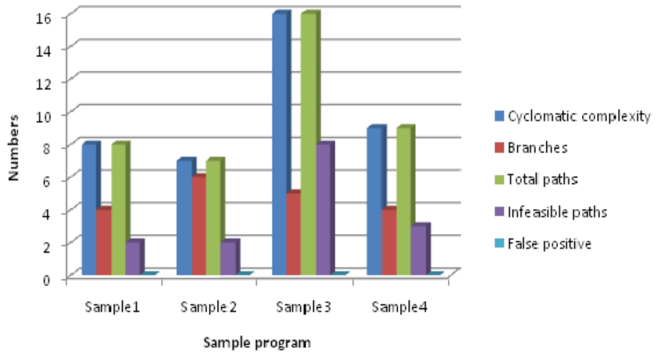


Fig. 7. Number of infeasible paths and false positive for set of sample programs during symbolic execution.

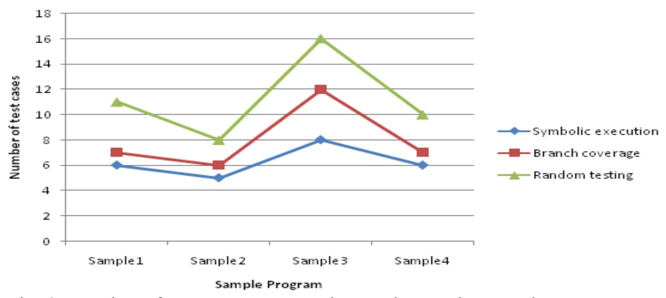


Fig. 8. Number of test cases generated to explore entire sample program.

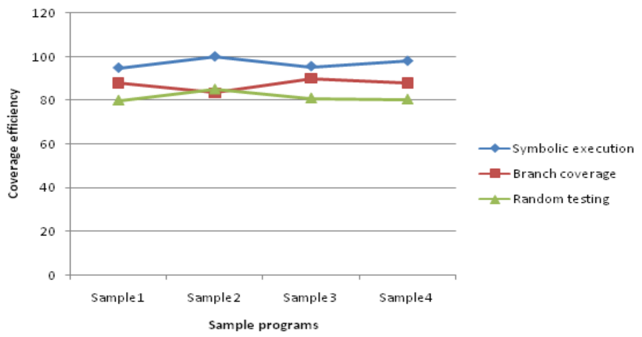


Fig. 9. Coverage efficiency calculated for sample program using different approaches.

V. CONCLUSION

In this paper, an automatic test case generation based on symbolic execution isproposed. This approach effectively eliminates infeasible paths present in the input program which in turn reduces ambiguity and system resources. It also utilizes an efficient path selection strategy from the symbolic execution tree to produce test cases for each traced path.The test cases generated from this approach showshigh coverage efficiency compared with other test generation techniques. Also with a minimum set of test cases,the entire input program can be explored which in turn greatly reduces the testing effort. With a set of sample programs, this approach has a high possibility of finding infeasible paths and shows more true positives compared with other techniques.Primary experimental results show that this approach is efficient and effective. Larger experimental programs are to be carried out for further evaluation.

REFERENCES

[1] B. Beizer, “SoftwareTesting Techniques,” 2nd. Edition, Van Nostrand Reinhold, 1990.

[2] M.J. Gallagher, V.L. Narasimhan, “Adtest: A test data generation suite for ada software systems,” IEEE Transactions on Software Engineering, vol.23, issue.8, pp.473-484, Aug1997.

[3] L.Baresi, M. Miraz, “Testful: Automatic unit-test generation for java classes,” In Proceedings of the 32nd ACMInternational Conference on Software Engineering (ICSE'10), Vol.2, pp.281-284 2010.

[4] Vorobyov, Krishnan,“Combining static analysis and constraint solving for automatic test case generation,” In Proceedings of theFifth IEEE International Conference onSoftware Testing, Verification and Validation (ICST),pp.915-920,Apr2012.

[5] B. Korel, “Automated software test data generation,” IEEETransaction on Software engg,vol16,iss.8, pp.870-879,1990.

[6] P. Tonella, “Evolutionary Testing of Classes,” SIGSOFT Software Eng. Notes, vol. 29, no. 4, 119-128, 2004.

[7] E.Díaz, J.Tuya, R.Blanco, “A modular tool for automated coverage in software testing,” In Eleventh Annual International Workshop onSoftware Technology and Engineering Practice, pp. 241 – 246, Sep 2003.

[8] J.C.King,“Symbolicexecutionandprogramtesting,” comm.ofACM,vol.19,issue.7,pp.385–394,July1976.

[9] M.X.Lin, Y.L Chen, K.Yu, G.S.Wu“Lazy symbolic execution for test data generation,”IET software vol.5, iss.2, pp.132-141, 2011.

[10] Mike Papadakis, Nicos Malevris, “A symbolic execution tool based on the elimination of infeasible paths,” In Proceedings of theIEEE5th international conference on software engineering advances (ICSEA), pp.435-440,Aug 2010.

[11] Thomas J. McCabe, “A complexity measure”, IEEETrans. on Software Engineering, vol.2, issue.4, Dec 1976.

[12] C. Pacheco, M. D. Ernst, “Randoop:feedback-directed random testing for Java,” Inproceedings of the 22nd ACM conference on OOPSLA '07, pp.815-816, 2007.

[13] http://chuck.ferzle.com/notes/notes/algorithms/bfsanddfs.pdf

[14] http://en.wikipedia.org/wiki/cyclomatic_complexity

[15] http://emma.sourceforge.net/

[16] http://www.atlassian.com/software/clover/overview

[17] http://en.wikipedia.org/wiki/Control_flow_graph

[18] L. Mariani,M.Pezzè,O.Riganelli, M.Santoro,"AutoBlackTest:A tool for automatic black-box testing," In Proceedings of the 33rd International Conference on Software Engineering (ICSE), pp.1013-1015,May2011.
[19] G.C.Morrison, C.P.Inggs, W.C.Visser, "Automated coverage calculation and test case generation," In Proceedings of the South African Institute for Computer Scientists and Information Technologists (SAICSIT'12), pp.84-93,2012.
[20] K. Munakata,S.Fujiwara, S.Tokumoto, T.Uehara,"Test case selection based on path conditions of symbolic execution," In 19th Asia-Pacific Software Engineering Conference (APSEC), Vol.1, pp.318-321, 2012.
[21] José Miguel Rojas, Miguel Gómez-Zamalloa, "A framework for guided test case generation in constraint logic programming," InLogic-based program synthesis and transformation, Vol.7844, pp.176-193, 2013.
[22] Yan Hu, He Jiang, "Effective test case generation via concolic execution," In Proceedings of the International Conference on Information Technology and Software Engineering , Vol.212, pp.157-164,2013.