ISSN 2007-9737

Test Case Generation Using Symbolic Execution

Saumendra Pattnaik[1], Bidush Kumar Sahoo[2], Chhabi Rani Panigrahi[3],
Binod Kumar Pattanayak[1], Bibudhendu Pati[3]

[1] Siksha 'O' Anusandhan,
Dept. of Computer Science & Engineering,
India

[2] Gandhi Institute for Education & Technology Bhubaneswar,
Dept. of Computer Science & Engineering,
India

[3] Rama Devi Women's University,
Dept. of Computer Science,
India

{saumendrapattnaik, binodpattanayak, bidush.sahoo,
panigrahichhabi, patibibudhendu}@gmail.com

Abstract. Testing is a well-known technique for identifying errors in software programs. Testing can be done in two ways: Static analysis and Dynamic analysis. Symbolic execution plays a vital role in static analysis for test case generation and to find the unreachable path with minimum test cases. Unreachable path is a part of a program which can never be executed i.e., the symbolic execution doesn't continue for that path and the current execution stops there. It generates a test suite for loop-free programs that is achieved by path coverage. In the best case program loops implies increase in the number of paths exponentially and in the worst case the program will not terminate. The functions of symbolic execution are test input generation, unreachable path detection, finding bugs in software programs, debugging. In this paper, we focus on dead code detection and test input generation using symbolic execution. Our execution for Java programs uses Java Path Finder (JPF) model tester. Our analysis shows that the symbolic execution method can be used to reduce symbolic execution time and to find out the unreachable path with less number of test cases.

Keywords. Symbolic execution, path coverage, unreachable path, test input generation.

1 Introduction

Testing is very important in the software evolution and maintenance process as it is required to find out the defects that were made in the development phase [21].

It is used for identifying the correctness and improving the quality of the software application. Testing can be done using static analysis or dynamic analysis. Symbolic execution plays an important role in static analysis for the test case generation and to explore the unreachable path with minimum number of test cases. Symbolic execution evaluates a program by considering inputs that results in execution of a program. This execution depends upon choosing of paths that are operated by a set of input values. A program is executed with symbols instead of real inputs in symbolic execution. Here, a source code is given as input for generating symbolic execution tree.

There are several reduction algorithms to reduce the symbolic execution tree. The reduction of the symbolic execution tree is done by eliminating the unreachable path and hence we generate the reduced test case generation. It basically focuses on generation of the test cases, unreachable path detection and model checking of concurrent programs which take inputs as the complex structures. The basic applications of symbolic execution are test input generation, unreachable path detection, finding bugs in software programs, debugging.

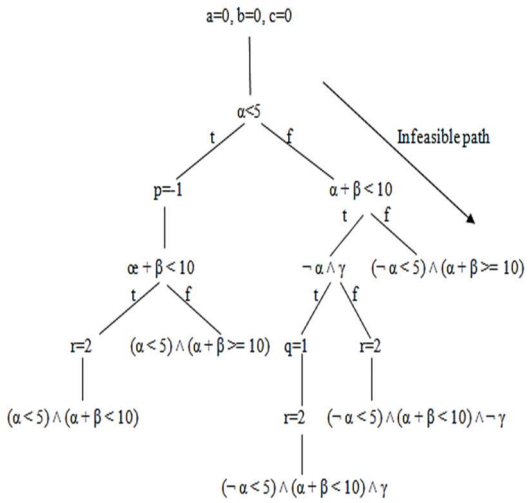

Fig. 1. A sample program on symbolic execution and the corresponding symbolic execution tree with symbolic values

Khurshid et al. [2] presented a two-fold observation of the traditional symbolic execution. Firstly, it enables the model checking of the concurrent programs to perform symbolic execution of the program.

Secondly, it provides a framework based on the symbolic execution algorithm that handles the complex data structures such as linked lists and trees. The symbolic execution is an efficient method that mainly focuses on achieving high coverage test suites, generates per-path guarantees and depicts the errors in the software programs.

Symbolic execution has been considered in the context of Symbolic Path Finder (SPF) [12].

SPF is an emblematic execution system based on the Java Path Finder (JPF) [13] show checking instrument set for Java Byte code investigation. Various applications of symbolic execution are test input generation, unreachable path detection, finding bugs in software programs, debugging.

In this paper, authors focus on dead code detection and test case generation techniques.

The paper is organized as follows. In Section 2, a sample program on symbolic execution is presented. Various dead code detection techniques proposed in the literature are presented in Section 3. Section 4 provides the symbolic execution techniques that are used in testing. In Section 5, the overall discussion on various techniques proposed in the literature for test case generation and dead code detection along with their advantages and disadvantages are presented.

2 Symbolic Execution Tree

A sample program to illustrate symbolic execution and the corresponding symbolic execution tree is shown as in Fig. 1.

```
int x = œ, y = ß, z = ŗ;
int p = 0, q = 0, r = 0;
if (x < 5) {
p = -1;
}
if (x + y < 10) {
if (!x && z)
{
q=1;
}
r=2;
```

Initially x, y and z have the symbolic values ŗ, ß and œ respectively. At each branch point, Path Condition (PC) is updated so that alternative paths can be chosen.

If the path condition evaluates to false, i.e., no data inputs pass through that path then it implies that the symbolic state is infeasible or is an unreachable path.

The unreachable path indicates that the symbolic execution for that path can not be continued and the current execution stops there.

3 Dead Code Detection Techniques

Dead code is the part of a program that can never be executed and whose result is never used in any other estimation. Various researchers [9, 22] and other sources as in [23] have found that the classical software code bases contains 5-10% “dead code”, i.e., code that can be eliminated without lowering the functionalities.

There are different techniques used for the detection of the dead code analysis. These techniques are described in the following subsections.

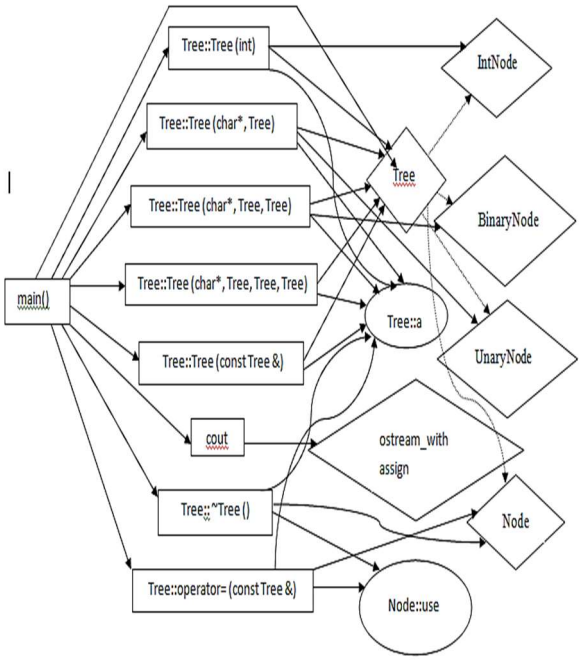

Fig. 2. Shows that all the members of the test class are exercised for the sample Tree class

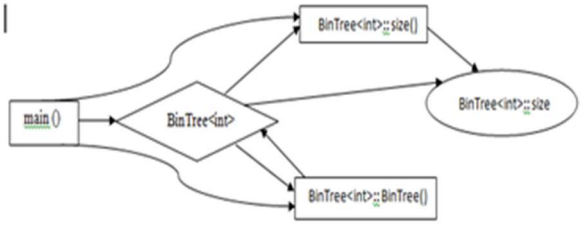

Fig. 3. Dependencies of Program entities on BinTree: size i.e. directly or transitively

3.1 Forward Reachability Analysis

Yih-Farn Chen et al. [9] basically found the representation logics that are used for reachability of code and detection of unreachable code techniques. These two tasks help to eliminate excess software baggage and holds software reuse metrics.

Forward Reachability Analysis is used for detecting dead code by determining a Reachable Entity Set and computing software reuse metrics. The reuse metrics is used to compute the code reuse and improve the quality and productivity. Reuse metrics consists of cost benefit analysis, percentage of reuse, reusability assessment, maturity assessment and failure modes analysis. There are two choices based on reachability of code and are described as follows:

- Software Reuse: It is also known as code reuse. It means using the existing software or to build a new software by using software knowledge. The goal of this is to reduce the cost of software production.

- Dead Code Detection: The part of the source code of a program which can never be executed, i.e., the symbolic execution does not

continue for that path and the current execution stops there:

```
Class Tree {
    Public:
    Tree (int);
    Tree (Char*, Tree);
    Tree (Char*, Tree, Tree);
    Tree (Char*, Tree, Tree, Tree);
    Tree (Const Tree & a);
    ~Tree () {
    if (-–b->use == 0)
    delete b;}};
    void operator = (const Tree & a);
    main () {
    Tree t = Tree (“-”, Tree (“*”, 7), Tree (“+”, 5, 3), Tree (“-”, 4,
                    5, 6));
    cout<< t << “\n”;
    t = Tree (“-”, t, t);
    cout<< t << “\n”;
    t = Tree (“+”, t, t, t);
    cout<<t<<”\n”;}
```

3.2 Reverse Reachability Analysis

Various authors have examined to use this technique of reachability code analysis. Researchers have found to support the complete reachability analysis that has been defined as an objective patterns at the selected abstraction level so that the programmers has the ability to analyze the different safeness implementations.

Reverse Reachability Analysis helps programmers to determine all the program entities that are dependent on an entity either directly or indirectly. If BinTree::size is altered, then other entities in the graph may be affected and is shown as in Fig. 3.

3.3 Visibility Analysis

Gansner et al. [9] accomplishes the reachability investigation on the regulation relationship in the class legacy chain of importance and discovers all part capacities and factors obvious to class BinaryNode in Koenig's illustration [10]. Regulation relationship happens between each parent class and a part.

Perceivability Analysis settles which part capacities and factors in a class legacy chain of command are seeable to a determined class. All part capacities in BinaryNode are seeable to itself. All open and shielded individuals from Node are likewise noticeable to BinaryNode on the grounds that BinaryNode has an open legacy association with Node.

Likewise, Node is a companion of Tree and along these lines all individuals from Tree are noticeable to Node. The visibility analysis of BinaryNode and Node is shown in Fig. 4.

4 Symbolic Execution Techniques in Testing

In this section, different techniques used for symbolic execution is presented.

4.1 Dynamic Symbolic Execution (DSE)

Data Flow Testing (DFT) [6] focuses on introducing a hybrid DFT structure. The heart of the structure is based on DSE and checks the reachability in software model checking to improve the testing performance.

DSE is a dynamic approach. It is a novel and efficient approach for automatic generation of the test cases. It combines the classical symbolic execution with real execution and generates many possible program pathways in the given amount of time. It aims at covering feasible pairs.

It takes the desired def-use pair du (l_d, l_u, x) as input and the Control Flow Graph (CFG) is constructed. It determines the test case for feasible test objectives and removes infeasible test objectives. It starts with arbitrary test input values. These test input values cause the execution path to cover the def-use pair. There are two approaches to deal with this problem.

Firstly we remove the invalid branching nodes by redefinition pruning technique and secondly, it applies Cut-Point Guided Search (CPGS) to choose which branching point to initially take. This approach can develop the DFT by 60-80% as compared to the testing span.The search results shows that the Dynamic Symbolic Execution approach reduces the DFT by 40% as compared to testing span to improve the testing performance than the Counter Example-Guided Abstraction Refinement (CEGAR) based approach.

Path explosion is a challenging problem in this approach because in a cheap amount of time to trigger the desired pair. Henceforth, more work will be done based on this approach to enhance the

data flow testing technique on large programs. Fig. 5 shows the workflow of DSE.

It starts with an arbitrary test inputs values followed by an execution path. Then, it removes the invalid branching nodes by redefinition pruning technique and finds the known paths by generating the constraint system.

4.2 Counter Example-Guided Abstraction Refinement (CEGAR)

CEGAR [6] is a static approach. It is utilized for creating the experiments and checking the fleeting wellbeing properties of the product. Here, a source program and an impermanent wellbeing determination is taken, which either demonstrates that the program fulfills the particular or produces a counter example to show the infringement.

This approach works in two stages: show checking and experiment created from the counterexamples. It first begins with a base or coarse program deliberation and over and again channels it so as to test for infeasible combine. We have to set a checkpoint to decide if the variable banner is valid.

The defutilize match is infeasible when the check point is inaccessible and a counterexample is returned when the def-utilize combine is attainable.

This approach is more intense than DSE approach as it enhances the scope by 20%. Consequently, more work will be done to make a profound correlation between both the methodologies. The workflow of CEGAR is shown in Fig. 6.

A C program is taken as input to the abstract model.

It then goes to the model checker where it automatically checks whether the model satisfies the given requirements and generates no error or bug found.

Then by the help of model checker, it goes to the simulator by taking counterexample as input where a program allows a computer to execute programs written for a different operating system by generating the simulation successfully and detecting the bug.

After that it goes to the refinement process where it removes the invalid the execution paths.

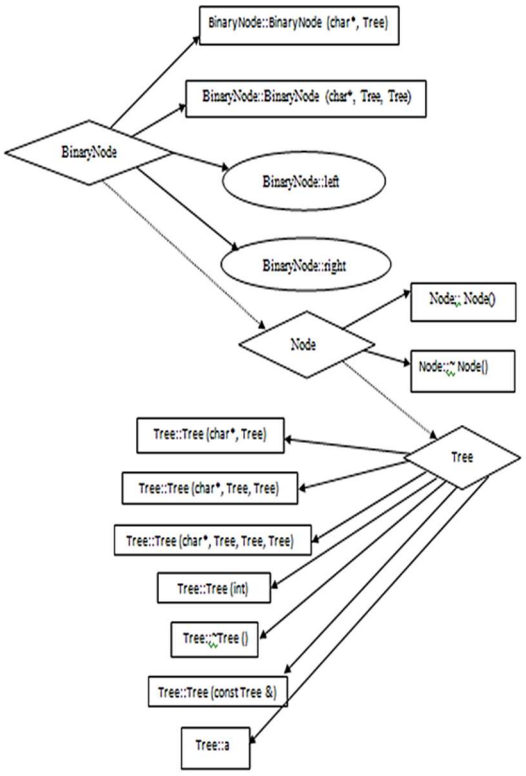

Fig. 4. Visibility analysis of BinaryNode and Node

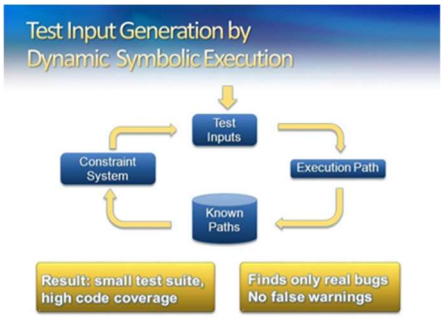

Fig. 5. Workflow of Dynamic Symbolic Execution

4.3 Document-Assisted Symbolic Execution (DASE)

DASE [3] is another novel approach for programmed age of the experiments and mistake recognizable proof to expand the adequacy and effectiveness of the representative execution. DASE separates the information impulse from

archives consequently and afterward utilizes the info impulse to center around testing mistake dealing with codes, which are vital that assistance the inquiry systems to enhance the representative execution and process the execution ways.

A program report is given as info that naturally separates the information impulse from that. There are two classes of info impulse: the setup of an information record and the substantial information estimations of order line decision.

The outcome shows that when contrasted with KLEE, DASE recognized 12 obscure imperfections that KLEE neglected to distinguish and out of 88 just 6 have been affirmed by the engineers.

DASE improves line investigation, branch examination and call investigation by 14.2–120.3%, 2.3–167.7%, and 16.9–135.2% in contrast with KLEE. Testing with invalid information is a testing issue. Subsequently, keeping in mind the end goal to test a program with invalid info esteems DASE approach centers by counterbalancing the information requirements.

4.4 Directed Automated Random Testing (DART)

DART [11] approach is also known as Concolic testing [14]. It dynamically operates the symbolic execution when the program is accomplished on real values. It supports two events: a real event and a symbolic event. A real event outlines all mutable to the real values and a symbolic event an outline all mutable to the non-real values.

A program is taken as input and the corresponding symbolic execution tree is drawn. It first starts with the generation of random inputs and execute the program concretely and symbolically. At each branch point, either it will follow the true path or it will end with the false path and generates the Path Condition (PC) for each path.

It finally surveys that all the paths of the program are examined and generates the test inputs. Path explosion and constraint solving is a challenging problem. There are two approaches to solve the path explosion problem: heuristics approach and sound program analysis approach.

There are two approaches to solve the constraint solving problem: irrelevant constraint elimination approach and incremental solving. Hence, more research can be performed for

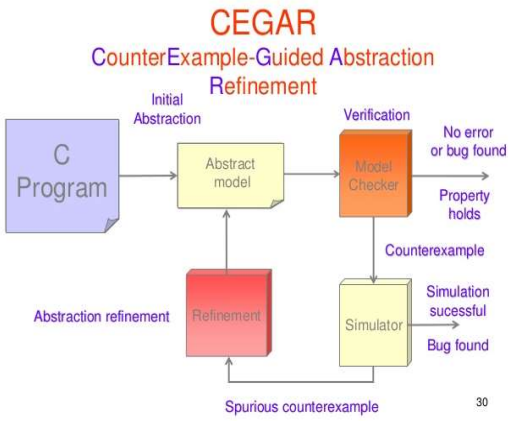

Fig. 6. Workflow of Counter Example-Guided Abstraction Refinement

automatic generatation of test inputs that covers the software bugs; generates the test suite that achieves high coverage; gives per-path guarantees and finding defects in software from low-level to high-level application programs.

The program analyzer which analyzes the behavior of the computer programs is divided into two parts. The first part is path selector in which control flow graph is taken as input and the second part is test data generator in which both control flow graph and data dependence is taken as input.

Data dependence is a condition where the program instruction refers to the data of the previous instructions. Then the path selector generates the selected paths and goes to the test data generator where the selected path information is taken as input to the path selector and produces the test data. Fig. 7 shows the workflow of DART.

4.5 Executed-Generated Testing (EGT)

EGT approach [15] is the instance of the modern symbolic execution techniques. It functions entirely between the real event and the symbolic event of the program. A source program is given as input and operated in the same program when the values are real.

It has the ability to mix both real and symbolic execution dynamically before finding all actions when the input values are all real. If so, then the action is executed in the same program or if more than one value is symbolic, the operation is

executed symbolically by maintaining a path condition for each path.

The disadvantages of this approach are elimination of irrelevant constraints, path explosion and memory modeling.

Hence, more research can be done in contributing a fashion to develop the test inputs that finds the errors which ranges from low to high-level syntactic features. Fig. 8 shows the workflow of EGT.

4.6 conc-iSE: Incremental Symbolic Execution Approach of Concurrent Programs

Incremental Symbolic Execution approach [16] for simultaneous programming is an approach to produce new test contributions by investigating the new execution ways between the two program variants.

These two program renditions i.e., old program variant P and new program form P' with an arrangement of execution abstract of program P is taken as information and over and over break down the present execution ways utilizing P'. Its yield is to examine the new recognition in P'.

This could be conceivable by evacuating the excess execution ways by rundown based calculation.

Consequently, amid the representative execution of P′, it breaks the regressive change-affect investigation. In this investigation, it fundamentally measures the arrangement of directions that may influence the changed guidelines set up of estimating the arrangement of guidelines that may be influenced by the changed directions [17].

The outcome demonstrates that this approach would overall be able to lessen the emblematic execution time and to expel the excess execution ways and string interleaving in the incremental representative execution.

The upper piece of the figure begins with the two program variants i.e., old program form P and new program adaptation P' with an arrangement of execution summation of program P is taken as info and more than once examine the present execution ways utilizing P' and produces the new recognition in P' by pruning the repetitive states or execution ways.

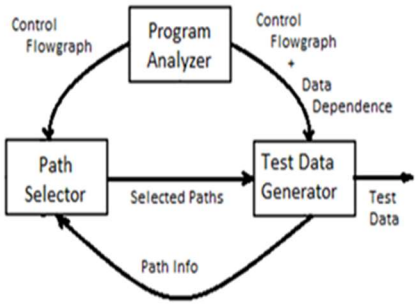

Fig. 7. Workflow of directed automated random testing

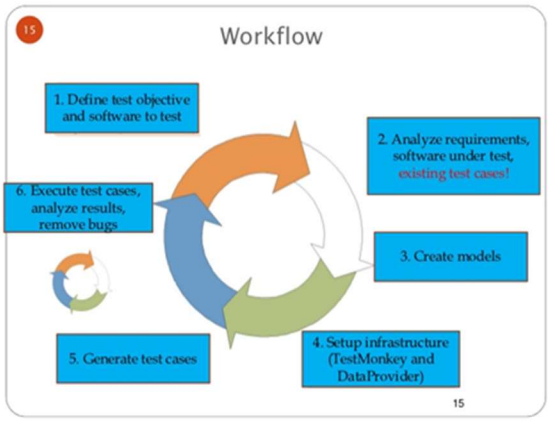

Fig. 8. Workflow of executed-generated testing

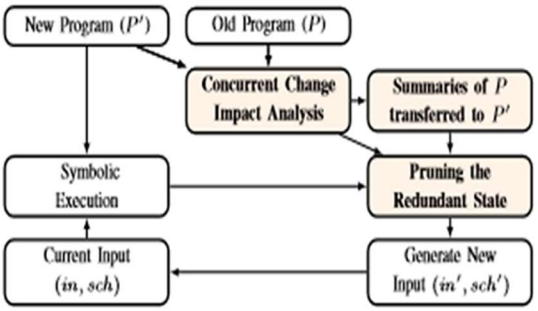

Fig. 9. Workflow of incremental symbolic execution

The lower some portion of the figure begins with a subjective current test inputs. Amid the representative execution, the new states are created and each new state produces another match incorporates the information info and string

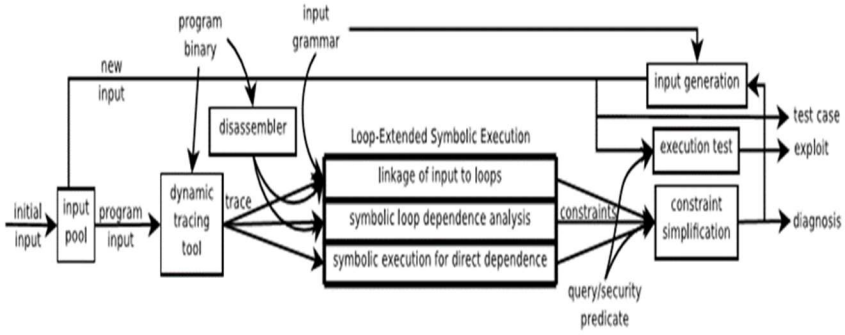

Fig. 10. Workflow of Loop-Extended Symbolic Execution

interleaving to touch base at the new state. The workflow of conc-iSE is shown in Fig. 9.

4.7 Loop-Extended Symbolic Execution (LESE)

Another representative execution strategy called LESE is proposed in [18] that sums up from the genuine execution to an arrangement of program executions that includes the distinctive number of redundancies for each circle as in the underlying execution.

Circle Extended Symbolic Execution is utilized to acquire appropriate outcomes when contrasted with emblematic execution when it is utilized as a part of projects with circles.

It makes mistake recognizing apparatuses more proficient and permits age of the experiments to achieve high test scope as fast as could reasonably be expected. A source program with circles is adopted as contribution to this strategy.

It begins with the predicate which is otherwise called inquiry predicate.

The predicate might be the branch condition related with the program point, and an execution that achieves the point, yet does not fulfill the predicate and after that yields to determine that the condition on a contribution to the program which creates the execution brings about a similar way and furthermore points the predicate to be valid.

It has the issue of recognizing and deciding support abundance impulse that produces on unmodified Windows and Linux sets.

The outcome demonstrates that the Loop-Extended Symbolic Execution can make an assortment of program examination, including the security applications, faster and more productive 19]. The workflow of LESE is shown in Fig. 10.

5 Discussion

In this section, we present the test case generation and dead code detection proposed in the literature along with their advantages and disadvantages, which are summarized in Table 1.

6 Conclusion

Various techniques used in symbolic execution provide a way to improve the test case generation and bug detection that achieves high test coverage suites, gives per-path correctness guarantees.

It also reduces the overall symbolic execution time and has the capacity to mix both real and symbolic execution [20]. In this work, we present different techniques that reduce the DFT in terms of testing time to improve the testing performance.

From the study, it is found that reduction of path explosion, constraint solving, determining buffer excess compulsion and the reduction of execution time in testing are some of the explored area in this field and can be taken as future work.

Tabla 1. Comparison of the proposed method with state of the art

Sl No.	Techniques Used	Test Case Generation	Dead Code Detection	Advantages	Disadvantages
1.	Dynamic Symbolic Execution (DSE)[11]	Analyzes program paths by determining path constraints.	Analysis of the reachability of code	-Designed a combined symbolic execution for automatic DFT.	-Path explosion is a challenging problem as it is required to find execution path to cover the desired pair. -DSE reduces the DFT in terms of testing time to improve the testing performance.
2	Counter Example-Guided Abstraction Refinement (CEGAR) [12]	Coverage criteria i.e., statement or branch coverage from counterexample paths.	Checking the practicability of the execution paths.	-Introduced a smooth encoding of DFT via CEGAR.	- Applying DFT on large multi-threaded software programs produces a broad analysis on it.
3.	Document-Assisted Symbolic Execution (DASE)[13]	Extracts input compulsion from documents automatically and focuses on execution paths that resemble valid inputs for improvement in the effectiveness of symbolic execution.		-Proposed and implemented an approach to improve symbolic execution for generation of test cases and bug finding.	-In order to test a program with invalid inputs, DASE approach focuses by cancelling out the input constraints.
4.	Directed Automated Random Testing (DART) [14]	Generates test suites that achieve high- coverage.	Examines the infeasible paths of the program using the depth-first search strategy.	-Capacity to combine both real and symbolic execution. -Proposed an effective symbolic execution technique to generate the inputs and performs symbolic execution dynamically. -Provides per-path correctness guarantees.	- Path explosion -Constraint solving
5.	Executed-Generated Testing (EGT) [15]	Approves the formation of high-coverage test suites.	Combines both real and symbolic execution and for the current path a path condition is maintained.	-Capacity to combine both real and symbolic execution. -Proposed a strategy to blend genuine and emblematic execution powerfully before checking each activity when esteems are for the most part genuine. -Achieves high test coverage suites.	-Elimination of irrelevant constraints. -Path explosion.
6.	conc-iSE: Incremental Symbolic Execution approach of concurrent programs[16]	Analyzes only the executions that affect the code changes between two versions of a program.	Reduces the overall symbolic execution time.	-Adapted an approach for concurrent programs to generate the new test inputs between the two program versions. -Reducing the symbolic execution time and removing the redundant execution path.	
7.	Loop-Extended Symbolic Execution (LESE) [18]	Allows to achieve high test coverage more quickly and to acquire better results when it is used in programs with loops.	-Analyzing buffer-overflow accountabilities in software programs after developing refuted candidates.	-Proposed a new approach by allowing test case generation to achieve high test coverage and automatic bug detection tools more effective.	-Identifying and determining buffer excess compulsion is a challenging problem that produces on unmodified Windows and Linux pairs.

References

1. **Cadar, C., Sen, K. (2013).** Symbolic execution for software testing: three decades later. Proceedings Communications of the ACM, Vol. 56, No. 2, pp. 82–90. DOI: 10.1145/2408776.2408795.

2. **Khurshid, S., Păsăreanu, C.S., Visser, W. (2003).** Generalized symbolic execution for model checking and testing. Proceedings International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, Heidelberg. pp. 553–568. DOI: 10.1007/3-540-36577-X_40.

3. **Wong, E., Zhang, L., Wang, S., Liu, T., Tan, L. (2015).** Dase: Document-assisted symbolic execution for improving automated software testing. Proceedings 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 620–631. DOI: 10.1109/ICSE.2015.78.
4. **Csallner, C., Tillmann, N., Smaragdakis, Y. (2008).** DySy: Dynamic Symbolic Execution for Invariant Inference. Proceedings of the 30th international conference on Software engineering, pp. 281–290.
5. **Guo, S., Kusano, M., Wang, C. (2016).** Conc-iSE: Incremental symbolic execution of concurrent software. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 531–542. DOI: 10.1145/2970276.2970332.
6. **Su, T., Fu, Z., Pu, G., He, J., Su, Z. (2015).** Combining symbolic execution and model checking for data flow testing. Proceedings 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, pp. 654–665. DOI: 10.1109/ICSE.2015.81.
7. **Kersten, R., Person, S., Rungta, N., Tkachuk, O. (2015).** Improving coverage of test cases generated by symbolic pathfinder for programs with loops. ACM SIGSOFT Software Engineering Notes, Vol. 40, No. 1, pp. 1–5. DOI: 10.1145/2693208.2693243.
8. **Saxena, P., Poosankam, P., McCamant, S. Song, D. (2009).** Loop-extended symbolic execution on binary programs. Proceedings of the eighteenth international symposium on Software testing and analysis, pp. 225–236. DOI: 10.1145/1572272.1572299.
9. **Chen, Y.F., Gansner, E.R., Koutsofios, E. (1998).** A C++ data model supporting reachability analysis and dead code detection. IEEE Transactions on Software Engineering, Vol. 24, No. 9, pp. 682–694. DOI: 10.1109/32.713323.
10. **Koenig, A. (1988).** An example of dynamic binding in C++. Journal of Object-Oriented Programming, Vol. 1, No. 3, pp. 60–62.
11. **Godefroid, P., Klarlund, N., Sen, K. (2005).** DART: directed automated random testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 213–223. DOI: 10.1145/1065010.1065036.
12. **Păsăreanu, C.S., Rungta, N. (2010).** Symbolic PathFinder: symbolic execution of Java bytecode. Proceedings of the IEEE/ACM international conference on automated software engineering, pp. 179–180. DOI: 10.1145/1858996.1859035.
13. **Anand, S., Păsăreanu, C.S., Visser, W. (2007).** JPF–SE: A symbolic execution extension to java pathfinder. International conference on tools and algorithms for the construction and analysis of systems, pp. 134–138. Springer, Berlin, Heidelberg.
14. **Sen, K., Marinov, D., Agha, G. (2005).** CUTE: a concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes, Vol. 30, No. 5, pp. 263–272. DOI: 10.1145/1095430.1081750.
15. **Betts, A., Chong, N., Deligiannis, P., Donaldson, A.F., Ketema, J., (2017).** Implementing and evaluating candidate-based invariant generation. IEEE Transactions on Software Engineering, Vol. 44, No. 7, pp. 631–650. DOI: 10.1109/TSE.2017.2718516.
16. **Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D. (2001).** Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, Vol. 27, No. 2, pp. 99–123. DOI: 10.1109/32.908957.
17. **Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K. (2014).** Optimal dynamic partial order reduction. ACM SIGPLAN Notices, Vol. 49, No. 1, pp. 373–384. DOI: 10.1145/2578855.2535845.
18. **Chattopadhyay, A. (2014).** Dynamic invariant generation for concurrent programs. Doctoral dissertation, Virginia Tech.
19. **Nimmer, J.W., Ernst, M.D. (2002).** Invariant inference for static checking: An empirical evaluation. ACM SIGSOFT Software Engineering Notes, Vol. 27, No. 6, pp. 11–20. DOI: 10.1145/605466.605469.
20. **Allen-Weiss, M. (2007).** Data structures and algorithm analysis in C++. Pearson Education India.
21. **Panigrahi, C.R., Mall, R. (2010).** Model-based regression test case prioritization. ACM SIGSOFT Software Engineering Notes, Vol. 35, No. 6, pp. 1–7. DOI: 10.1145/1874391.1874405.
22. **Streitel, F, Steidl, D., Jürgens, E. (2014).** Dead code detection on class level. Softwaretechnik-Trends, Vol. 34, No. 2.
23. **Pizzutillo, P. (2013).** Static analysis: Leveraging source code analysis to reign in application maintenance cost.

Article received on 27/12/2020; accepted on 14/11/2021.
Corresponding author is Chhabi Rani Panigrahi.