ISSN: 2321-7782 (Online)
Volume 3, Issue 1, January 2015
International Journal of Advance Research in Computer Science and Management Studies
Research Article / Survey Paper / Case Study
Available online at: www.ijarcsms.com

Symbolic Execution in Software Testing-A Review

Saleema D[1]
PG Scholar
Department of Computer Science
College of Engineering Perumon
Kerala, India

Bejoy Abraham[2]
Associate Professor
Department of Computer Science
College of Engineering Perumon
Kerala, India

Abstract: Symbolic Execution is a program analysis technique used in automated software testing. It uses symbolic variables instead of actual data. That means execute the program symbolically rather than concretely which maintains a path condition that is updated whenever a branch instruction is encountered. Automatically generates high coverage test inputs on termination of a path or when it hits a bug by solving the current path constraint. There have been a variety of symbolic execution (SE) strategies like Generalized versus Classical, and Directed versus Undirected Symbolic Execution. In this article we present a review and a comparative study of various Symbolic Execution strategies used in software testing. We will refer Symbolic Execution as Parameterized Execution.

Keywords: General Parameterized Execution; Directed Symbolic Execution.

I. INTRODUCTION

Software testing has an important role in quality software development. But the poor performance of random and manual approaches has led to much recent works in using symbolic execution to generate high coverage test inputs. Instead of running code on manually or randomly constructed input, they run it on symbolic input initially allowed to be "anything." Use symbolic values instead of actual data and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system follows both branches at once, maintaining on each path a set of constraints called the path condition which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition to find concrete values. Symbolic execution is a well-known program analysis technique which traditionally arose in the context of checking sequential programs with a fixed number of integer variables. It requires dedicated tools to perform the analyses and do not handle concurrent systems with complex inputs.

II. GENERAL PARAMETERIZED EXECUTION

General Parameterized Execution is the normal General Symbolic Execution. Two common techniques for checking correctness of software are *Testing* and *Model Checking*. Modern software systems are concurrent and manipulate complex dynamically allocated data structures (e.g., linked lists or binary trees).Testing is widely used but doesn't give us an assurance that whether it is good at finding errors related to concurrent behaviors. Model Checking is an automatic process and is good at analyzing concurrent systems which suffers from state-space explosion problem and typically requires a closed system i.e., bound on input sizes.

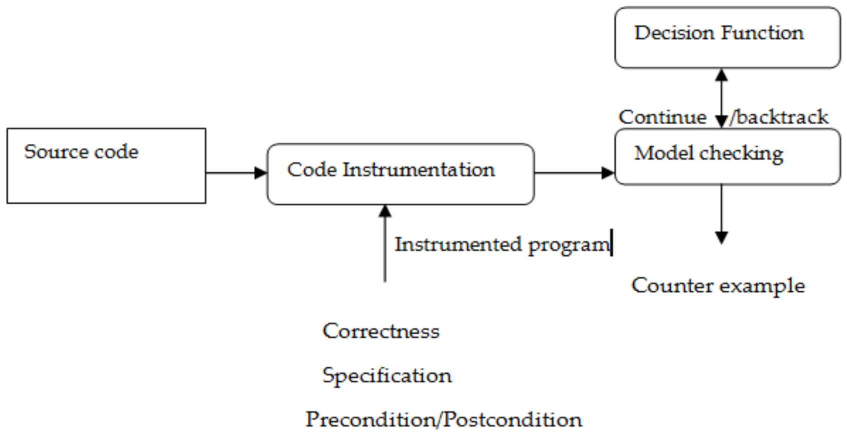
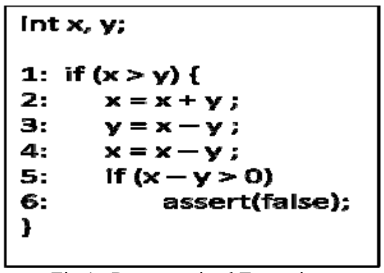


Fig 1.Model Checking Process



```
Int x, y;

1:  if (x > y) {
2:       x = x + y ;
3:       y = x — y ;
4:       x = x — y ;
5:       if (x — y > 0)
6:              assert(false);
}
```

Fig 1. Parameterized Execution

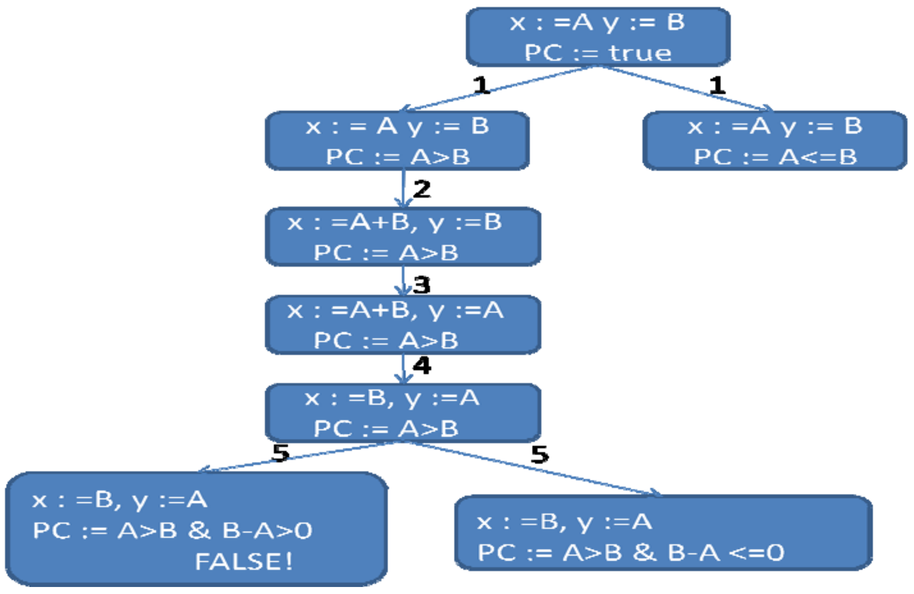


Fig 3. Symbolic Execution Tree

A .Two-way Generalization of symbolic execution

a. Lazy initialization

Lazy initialization is an algorithm for generalizing traditional symbolic execution to support advanced constructs of modern programming languages, such as Java and C++. A key feature of the *lazy initialization* algorithm is that it starts execution of the method on inputs with u*ninitialized* fields and use *lazy initialization* to assign values to these field, i.e., it initialize fields when they are first accessed during the method's symbolic execution.

If(F is uninitialized)

{

If(F is a reference field of user-defined type T)

```
{non deterministically initialize F to
        1. null
        2. a new object of class T(with uninitialized field values)
        3. an object created during a prior initialization of a field of type T
If(method preconditions is violated)
backtrack();
 }
If(F is a primitive field)
initialize to a new symbolic value of appropriate type
```

Fig 4.Lazy initialization algorithm

Decision procedure checks whether the path condition is satisfied. If not, it backtracks.

b. Instrumentation

There are mainly two steps in instrumentation

1. The integer fields and operations are instrumented

Here the declared type of integer fields of input objects is changed to *Expression*, which is a library class for manipulation of symbolic integer expression

2. The field accesses are instrumented

Here the field reads are replaced by *get* method, *get* methods implement the *lazy initialization* and filed updates are replaced by *set* method.

Symbolic execution during model checking is powerful but we don't know how well it scales to real applications. The limitations of symbolic execution is the generation of complex constraints or parts of which cannot be symbolically executed and their generated constraints can be too complex to solve.

III. Directed Symbolic Execution

Directed Symbolic Execution is used to solve Line Reachability problem. Given a target line in the program, we have to check whether we can find an input that drives the program to that line. It is equivalent to find an input that causes the program to enter a particular state expressed as a conditional

E.g. **if(i>MAX) assert(0);**

Applications reproduce the error in a bug report, target line from stack trace, triage errors reported by static analysis, target line from error report, detect false positives, improve code understanding and guide the search towards a target line. The different Directed symbolic execution strategies are

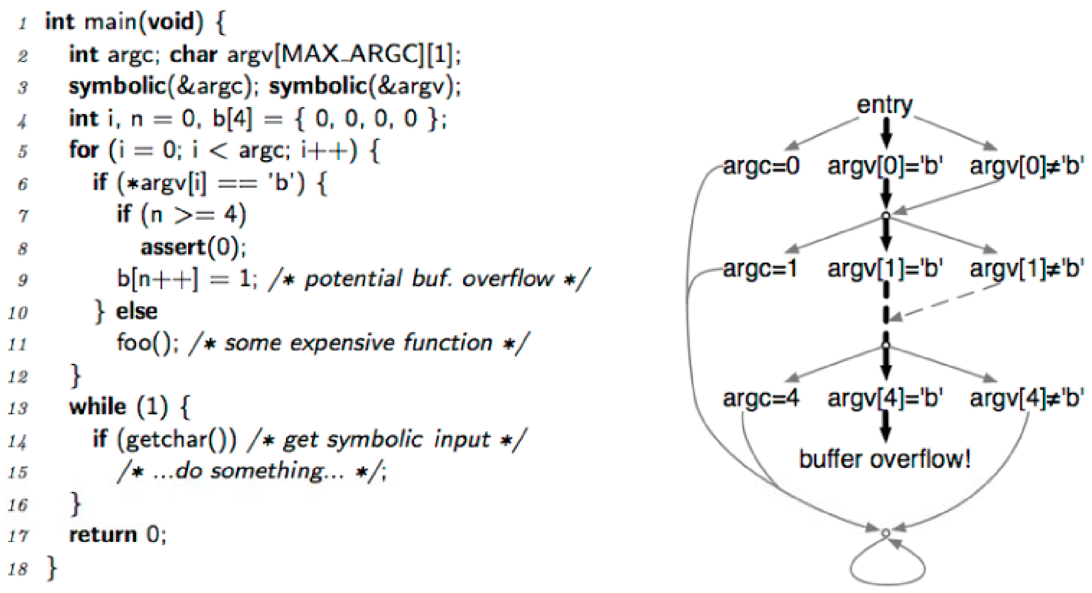
A. Shortest-Distance Symbolic Execution (SDSE)

B. Call-Chain-Backward Symbolic Execution (CCBSE)

C. Mixing CCBSE with a forward strategy (Mix-CCBSE)

1. *Shortest-Distance Symbolic Execution (SDSE)*

Bias the search using *inter-procedural CFG-distances* to target lines. Start symbolic execution *in the function* containing the target line. Set the function as *target*, and repeat this *backward up the call chain*, until we find a feasible path from the start of the program.



```
1  int main(void) {
2    int argc; char argv[MAX_ARGC][1];
3    symbolic(&argc); symbolic(&argv);
4    int i, n = 0, b[4] = { 0, 0, 0, 0 };
5    for (i = 0; i < argc; i++) {
6      if (*argv[i] == 'b') {
7        if (n >= 4)
8          assert(0);
9        b[n++] = 1; /* potential buf. overflow */
10     } else
11       foo(); /* some expensive function */
12   }
13   while (1) {
14     if (getchar()) /* get symbolic input */
15       /* ...do something... */;
16   }
17   return 0;
18 }
```

Fig 5.Shortest Distance Symbolic Execution

In this program statement 3, input is made symbolic .b is a buffer of size 4, n is the next position in b to write into buffer. A for-loop iterating over argv if the argument is a letter b, put a 1 into b if the argument is NOT a letter b, run foo() which is expensive. Line 13 to 16 is an an infinite loop doing work. Buffer overflow occurs when n ≥ 4. A conditional to detect buffer overflow is set. We are setting the assert(0) is the target. We get strait path as the shortest path, for this

1. Build a CFG for each function
2. Build an inter-procedural CFG by splitting function calls into call and return nodes, and connecting them to function entries and exits
3. Compute shortest distances using CFL reachability, specifically PN-paths. Invalid PN-path, because path enters bar from one call site and returns to another call site.

2. *Call-Chain-Backward Symbolic Execution(CCBSE)*

Start at function (foo) containing the target. Run forward symbolic execution using some strategy X to find the target. Once found, go back to its caller (bar). Run forward symbolic execution to find the call to foo. Before searching foo, check if we can continue along the known path to-target in foo. If realizable, we immediately have a path-to-target from bar by stitching the two paths. We don’t need to explore foo again. Repeat this process until we go back to main.

```
1  void main() {
2    int m, n, i;
3    symbolic(&m); symbolic(&n);
4    for (i=0;i<1000;i++)
5      if (m == i) f(m, n);
6  }
7  void f(int m, int n) {
8    int i, a, sum=0;
9    for (i=0;i<6;i++) {
10     a = n%2;
11     if (a) sum += a+1;
12     n/=2;
13   }
14   while(1) {
15     if (sum==0 && m==7)
16       assert(0);
17   }
18 }
```

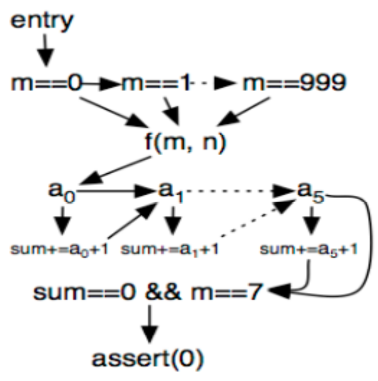


Fig 6. Call Chain Backward Symbolic Execution

Here f(m, n) is called for all m in [0,1000) calculate the sum of 6 least sig. bits of n infinite loop; fail with the condition holds Failing condition 1: m==7; directly determined by m in f(m,n). The only path among 26 paths that has sum==0. The true branch at line 11 is never taken. Failing condition 2: sum==0; indirectly determined by n in f(m,n) . Every time f(m, n) is called, CCBSE first checks if the green path is realizable. It's always realizable since n is unconstrained. When m==7, the assertion is reachable. CCBSE avoids re-exploring the 26 paths in f. simultaneously run CCBSE, forward search from main, try merging when the forward search reaches a function that has a path to target found by CCBSE. The main advantage is that it is better than forward/ CCBSE alone due to smaller search spaces. But the problem of doubling the runtime of forward search (plus overhead).

3. Mixing CCBSE with Forward Strategies

Simultaneously run CCBSE and Forward search from main function. Try merging when the forward search reaches a function that has a path to target found by CCBSE. In best case scenario, it is better than Forward, CCBSE alone due to smaller search space. In worst case, doubling the run time of Forward search (plus overhead).

IV. Evaluation

Otter, a symbolic executor for C which compares different strategies, Inter SDSE with Intra SDSE, CCBSE with Random-Path. It is found that Distance heuristic works very well in practice. Using inter procedural distance is crucial for SDSE. SDSE timed out, but CCBSE (RP) did not time out on mknod, and it has better overall performance than InterSDSE. Mixing CCBSE with Otter-KLEE is good, even better than CCBSE and Otter-KLEE alone. SDSE is fast in many cases, but can perform very poorly sometimes. Mixing CCBSE with Otter-KLEE gives the best overall performance. Limitations include threats to validity unavoidably differs from the originals, Different front-end/parser and standard libraries from original.

V. Conclusion

Directed symbolic execution guides the search towards a target line. In this paper we tried to compare performance of Shortest-Distance Symbolic Execution (SDSE), Call-Chain-Backward Symbolic Execution (CCBSE) and then Mixing CCBSE with a forward strategy (Mix-CCBSE) across generalized Symbolic Execution. Results show that SDSE is fast in many cases, but can perform poorly sometimes. Mixing CCBSE with KLEE has the best overall performance.

Acknowledgement

We would like to express our sincere gratitudes to C. Cadar, D. Dunbar, and D. Engler, V.Ganesh, P. M. Pawlowski, D. L. Dill, P. Godefroid, S. Khurshid, C. S. P asareanu, K. Sen, N. Tillmann, and W. Visser,] J. C. King, for their preliminary works in Symbolic Execution and Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks for their works in directed symbolic execution as a reference in making this paper.

References

1. C. Cadar, P. Godefroid, S. Khurshid, C. S. P_as_areanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice Preliminary assessment," in Proc. 33rd Int. Conf. Softw. Eng 2011, pp.1066–1071.
2. J. C. King, "Symbolic execution and program testing," Commun.ACM,vol.19, pp. 385–394, Jul. 1976.
3. C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in Proc.8th USENIX Conf. Operat. Syst. Des.Implementation, 2008, pp. 209–224.
4. P. D. Marinescu and C. Cadar, "Make test-zesti: A symbolic execution solution for improving regression testing," in Proc. 34th Int. Conf. Softw. Eng. Jun. 2012, pp. 716–726.
5. "Directed Symbolic Execution" by Kin –Keung Ma, Khoo Yit Phang Jeffrey S.Foster and Michael Hicks,University of Maryland
6. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems" programs In OSDI.
7. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: automatically generating inputs of death In CCS".
8. Saswat Anand, Corina S. Pasareanu, and Willem Visser. Jpf-se: "A symbolic execution extension to java path finder" in International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007).
9. Busybox. http://busybox.net/.
10. Gnu c library. http://www.gnu.org/s/libc/.
11. The java path_finder wiki. http://babelfish.arc.nasa.gov/trac/jpf./s/libc [3] The java pathfinder wiki. http://babelfish.arc.nasa.gov/trac/j pf. Gnu c libraribc/.

AUTHOR(S) PROFILE

Saleema D, pursuing M.Tech Degree in Computer and Information Science in department of Computer Science, College of Engineering Perumon, kerala, India, received B.Tech degree in Computer Science and Engineering from College of Engineering Perumon in 2010.

Bejoy Abraham, the Associate Professor and Head of the department of Computer Science, College of Engineering Perumon Kerala, India currently doing his Phd in University of Kerala, India, received his M.Tech degree in Software Engineering from Cochin University of Science and Technology in 2000, Kerala, India and B.tech in Computer Science and Engineering from C.S.I Institute of Technology, Thovalai in 2000.