A Survey of Symbolic Executions Techniques

Hallet Adrien Sens Loan

October 11, 2018

Abstract

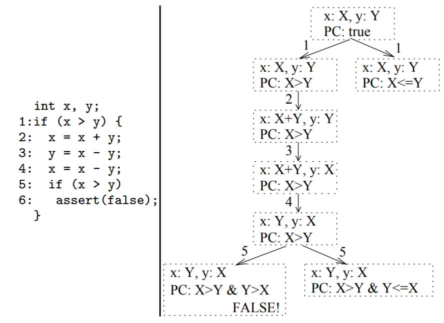
1 Introduction

1.1 A definition

The first occurences of symbolic execution described the then-new method as a middle ground [5] between the two most-used method of its time. On one hand, program testing (*e.g.: unit testing*) can not always detect a fault in a program and producing a correct test sample and proving that it indeed is correct is not that easy. On the other hand, program proving can indeed ensure that a program is correct from its entry point to the result but it heavily relys on the proof definitions by the programmer and the formal definition of the problem.
Nowadays, symbolic execution is both described as (part of) the core of many modern techniques to software testing [7] and an effective way to create tests suites with extensive coverage. [2]

1.2 The concept

The idea behind symbolic execution is to test an algorithm with *symbolic values* rather than concrete values. So instead of using unit testing where a variable is set to a (usually random) value, the symbolic execution maintains a formula that contains all the possible values for the code to reach a particular point in the program. This formula is updated every time the program reaches a branching point. In figure 1, we show an example from [8] of a symbolic execution. Notice how it produces constraints over the variables to explore the algorithm's branching tree.



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

Figure 1: Swapping two integers and its symbolic execution tree

2 History

3 Method

3.1 Useful concepts

Algorithms can be modeled as graphs where nodes are basic blocks (*i.e.: a part, one or multiple instructions with a single entry and exit point*) and edges are the branches (issued from conditional statements). Def-use pairs use the same concept, although they base their graph on the *definition* and *usage* of a variable. With the *branches* and *def-use pairs*, we can model an algorithm's behavior to follow the values of its variables and determine the *execution path*.

3.2 Basis

Symbolic execution works on those concepts by updating an internal list of symbols. The execution generates a new symbol for each introduced variable in an algorithm [5]. The symbolic execution runs over the algorithm's statements and builds the symbolic values when it encounters a branching point. The symbolically executed algorithm creates a *state* [8] containing the symbolics values, a counter identifying the next line to be executed and a *path condition*. This path condition is a simple boolean formula over the symbols, it creates a constraint for the algorithm to reach the current state of the program (this also allows to check for unreachable paths in programs [1]). The path condition allows to recreate the execution up to its state. The states are stored in a *symbolic execution tree* with the states as nodes and the transitions as edges (see figure 1).

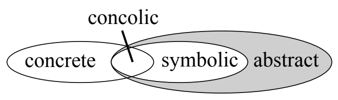


Figure 2: Concrete and abstract execution machine models

3.3 Problems

Symbolic Execution cannot be that perfect and hosts its bundle of problems that reduce either the confidence in or the performances of the concept. We have seen symbolic execution tree in 3.2. Small algorithms can use such methods but actual programs need to be tested in *integration*. In large environments, the tree's branching factor will produce too many nodes (*state-space explosion*) for the performances to stay relevant, sometimes creating infinite loops in the graph [6]. Reducing the state space or pruning them is not enough. To improve the performances, we can depth-first-search the graph but it does not prevent infinite loops until we add a max depth (as KLEE or EXE do). Pruning heuristics (*e.g.:def-use pairs distance*) can be used to reduce the tree's branching factor, we may randomly select a path (emphi.e: the path condition), weighting the shallowest nodes to avoid dead-loops. The random technique is exploited by a lot of *fuzzing techniques* [3] which uses the injection of random values to detect program's faults. Another solution lies in the *concolic execution* (see 4.1).

4 Variants

4.1 Concolic execution

The name ”concolic” is a portmanteau of the words ”concrete” and ”symbolic”, the idea of this testing method is to mix symbolic execution alongside concrete ones.

Concolic execution approaches

This technique concept was first introduced on 2005 [4] (more details on section 5.1). Since then the idea was further extended and combined with other testing techniques.

However, the general principle has been explored with different angles.

4.1.1 Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) also kwown as *dynamic test generation* [4] is a popular approach of concolic execution. Its main feature is to have the concrete execution drive the symbolic execution.

We need to add a new store in order to save the concrete execution information σ_c.
We first choose an arbitrary value as input for our parameters. Then it executes the program concretely and symbolically at the same time updating both stores and the path constraints. Whenever the concrete execution takes a branch, the symbolic execution is directed toward the same branch and the constraints extracted from the branch condition are added to the current set of path constraints.
In order to explore different paths, the path conditions given by one or more branches can be negated and the solver invoked to find a satisfying assignment for the new constraints.
We can repeat this process as many time as we want to achieve the desired coverage.

Notice that it exists different strategies on the choice of the branch to negate, this heuristic choice which is crucial depends on the tool.

Downside

False Negative Missed path. For example, when another function from the one tested is not symbolically tracked but its result is needed to explore a particular path.

Path Divergence In some situations the engine can't guess that no input can provoke an error. For example, assert on a negative value of an absolute value due to the untracked side effect of the `abs()` function.

5 Tools and languages

5.1 *DART* : Directed Automated Random Testing

DART is presented as a tool for automatically testing software using concolic testing method. It was introduced in 2005 making it the first the first tool to be created using concolic techniques and more specifically dynamic symbolic execution techniques (see section 4.1.1).

5.1.1 Methodology

DART combines three main techniques [4] in order to automate the process testing for a particular software :

1. An automated extraction of the interface of a program with its external environment using static source-code parsing
2. An automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in
3. A dynamic analysis of how program behaves under random testing and automatic generation of new test inputs to direct systemically the execution along alternatives program paths

DART chooses the *depth-first* strategy whenever it has to negate a branch.

5.1.2 Example

Let consider the following program :

```
Function foo(int x, int y):
    if x != y then
        if 2 * x == x + 10 then
            ERROR;
        end
    end
    return SUCCESS;
```

This function is defective as it may lead to an error statement for some value of x and y.
DART start by guessing values for both x and y for instance 269167349 and 889801541. With this values the function return successfully, during the execution two predicates were formed created by the `if` statements, in our case the path constraint at the end is : $\langle x_0 \neq y_0, 2 \times x_0 \neq x_0 + 10 \rangle$ with x_0 and y_0 both beings *symbolic variables.*

While we maintain this predicates, all path will lead to the same end. So in order to force the program through a potential different outcome we change one of the predicate and look at the result. If we negate the last predicate we have the following path constraint : $\langle x_0 \neq y_0, 2 \times x_0 = x_0 + 10 \rangle$ in which $x_0 = 10$ and $y_0 = 889801541$ is a solution. Using this values as inputs the program end up into the `ERROR` as wanted.

5.1.3 Key strength/originality

The main strength of DART is that testing can be performed completely automatically on any program that compiles – there is no need to write any test driver or harness code.
During testing, DART detects standard errors such as program crashes, assertion violations, and non-termination.
DART provides an attractive alternative approach to static analyzers, because it is based on high-precision dynamic analysis instead, while being fully automated as static analysis. The main advantage of DART over static analysis is that every execution leading to an error that is found by DART is guaranteed to be sound. Two areas where we expect DART to compete especially well against static analyzers are the detection of interprocedural bugs and of bugs that arise through the use of library functions.

DART is overall complementary to static analysis since it has its own limitations, namely the computational expense of running tests and the sometimes limited effectiveness of dynamic test generation to improve over random testing.

6 Conclusions

References

[1] Romain Aissat, Frédéric Voisin, and Burkhart Wolff. Infeasible paths elimination by symbolic execution techniques. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 36–51, Cham, 2016. Springer International Publishing.

[2] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. 56:82–90, 02 2013.

[3] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers and Security*, 75:118 – 137, 2018.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[5] J.C. King. A new approach to program testing. 10:228–233, 06 1975.

[6] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, May 2010.

[7] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *International Conference on Software Engineering (ICSE 2018)*, 5 2018.

[8] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.