Symbolic Execution for Software Testing: Three Decades Later: A Summary

Caleb Bowers

I. Summary

What follows is my best attempt at summary information for [1]. This paper provides an "overview of modern symbolic execution techniques, discus[es] their key challenges in terms of path exploration, constraint solving, and memory modeling, and discuss[es] several solutions drawn primarily from the authors' own work." The authors do not propose that they will provide a comprehensive summary of symbolic execution techniques, but rather they wish to highlight the key challenges as exhibited in their own work.

A. Background

Symbolic execution provides a powerful methodology for generating test suites that can offer higher coverage than traditional testing approaches, and is capable of finding errors hidden deep within complex software applications. These capabilities are provided by symbolic execution's ability to explore as many different paths of program execution as possible within a given amount of time. For each path explored, symbolic execution *i*) generates a set of concrete input variables that execute along that path and *ii*) checks for the presence of traditional property violations and program errors.

The historical development of symbolic execution began with the idea that researchers make use of *symbolic values*, rather than actual data values as inputs to an execution instance. Additionally, program variables are represented as *symbolic expressions* over the symbolic input values. The output values of a program executing on such a symbolic logic are expressed as a function of the input values. All symbolic execution paths of a program are represented in a tree structure and can all be evaluated by running the program over a set of inputs such that all the symbolic paths are explored exactly once. Symbolic execution may fail to generate an input if the constraint on a symbolic path cannot be efficiently solved by a constraint solver. More plainly, even though a symbolic input may exist and the path itself may be valid, symbolic execution will not be able to determine the veracity of this path, because of the constraint is too complex to evaluate. Similar solution restrictions exist for looping (where infinite paths may exist and only a timeout can resolve this).

B. Modern Symbolic Execution Techniques

Modern symbolic execution techniques offer the ability to mix concrete and symbolic execution, so a researcher can test their program on sample inputs for trivial input values while symbolically abstracting the more crucial for more rigorous testing. Two examples of these hybrid approaches are:

- *Concolic Testing*: Or, Directed Automated Random Testing
  - *Advantages*:
    - Maintains a concrete state and a symbolic state.
    - Concrete state maps all variables to their concrete values.
    - Symbolic state maps only variables that have non-concrete values.
  - *Disadvantages*:
    - Requires initial concrete values for inputs–could be a challenge to generate valid ones.
    - Could be constrained by limitations of constraint solver.
- *Execution-Generated Testing*
  - *Advantages*:

        * Intermixes concrete and symbolic execution by dynamically checking if all values are concrete.
    - *Disadvantages*:
        * Valid concrete input values search space may be large.

A big advantage to both of these methods is that by mixing the concrete and symbolic state spaces, is that the impacts of imprecision in symbolic execution and the potential timeouts of constraint solving can be reduced by the use of concrete values. This enables the intermixing method to generate test inputs for paths for which symbolic execution may get stuck. If symbolic execution is stuck, it can turn to concrete values in an attempt to resolve and avoid aborting the execution.

C. Challenges and Solutions

1) Path Explosion: There exists an intensely large number of program paths even for relatively small programs, so the most relevant paths must be explored first. To reduce the number of paths there are several heuristic techniques as well as implementing sound program analysis techniques. Heuristically speaking, one may use static control-flow graphs to help improve the statistical chance of finding an uncovered execution path. Randomly generating paths to explore also proves helpful in reducing the amount of paths to explore. Evolutionary search makes use of a fitness function to drive exploration of the input space towards a variety of optimal criteria (e.g., path mutation).

A variety of program analysis techniques can also be employed to reduce state space. Namely, using *select* expressions on possible execution paths, reusing lower-level function analysis form previous computations, and automatically pruning redundant paths during execution path exploration (e.g., repeated code execution).

2) Constraint Solving: Symbolic execution has long been held hostage by the performance and efficiency of constraint solvers. Though recent improvements in constraint solvers are what enabled researchers to leverage symbolic execution, it still creates the largest bottleneck in execution. To reduce the impact of the constraint solvers, several methods exist. Eliminating irrelevant constraints for a given branch/path of execution can greatly improve the speed of the constraint solver, which directly depends on the number of constraints. Additionally, a solver can incrementally solve constraints for sets of branches. After one branch is solved, it can move on to the next dependent branch[es].

3) Memory Modeling: Often when translating program statements into symbolic constraints, a trade-off between precision in scalability is balanced. Determining how to balance this trade-off should rely on the actual code being analyzed, rather than static code principles for given variables or functions.

4) Handling Concurrency: The paper does not provide *how* symbolic execution can be adapted to handle concurrent applications; it only states that symbolic execution *can* handle concurrency and has been shown to be successful.

D. Tools

The authors split this section into three parts; the first two of which focus on tools specifically for Concolic testing and EGT, respectively. The tools DART, CUTE, and CREST all provide Concolic testing capability. DART ultimately seeks to evaluate all execution paths and CUTE extended DART to account for multi-threaded programs manipulating pointer based dynamic data structures. CREST is an open source executor meant for C programs and allows for building and experimenting with different heuristic options for selecting relevant paths.

EXE and KLEE were built for systems level code exhibiting high levels of complexity. KLEE is an improved redesign of EXE and can store a much larger number of possible states than EXE due to its state object sharing approach.

II. Conclusion

The authors conclude by stating the efficacy and value of symbolic execution in generating erroneous inputs and program areas. Overall, the authors have provided compelling reason to believe that symbolic logic is valuable, namely by showing its capability in finding incorrect inputs and errors, and by providing evidence of a strong user and research community. Several examples from industry (primarily, Microsoft) have also seen the value of symbolic execution, especially as the world becomes more software dependent. I think this is a valuable area of research and will prove incredibly important over the next 25 years as software becomes more autonomous and interwoven in humanity's daily existence.

References

[1] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795