On the Nature of Symbolic Execution

– invited talk –

Frank S. de Boer
Centrum Wiskunde & Informatica (CWI)
and Leiden Institute of Advanced Computer Science
Leiden University
The Netherlands
e-mail: F.S.de.Boer@cwi.nl

Marcello M. Bonsangue
Leiden Institute of Advanced Computer Science
Leiden University
The Netherlands
e-mail: m.m.bonsangue@liacs.leidenuniv.nl

***Abstract*—In the symbolic execution of a program real values are replaced by so-called symbolic values. Consequently, programming expressions cannot be evaluated, and thus the state, i.e., the assignment of values to program variables, in a symbolic execution is replaced by a substitution which assigns to each program variable an expression. The goal of symbolic execution is to generate a path condition that specifies concrete input values for which the actual execution of the program follows the same path (as generated by the symbolic execution).**

In the full version of this abstract we provide a formal definition of symbolic execution in terms of a symbolic transition system and prove its correctness with respect to an operational semantics which models the execution on concrete values. Our approach is modular, starting with a formal model for a basic programming language with a statically fixed number of programming variables. This model is extended to a programming language with recursive procedures which are called by a call-by-value parameter mechanism. Finally, we show how to extend this latter model of symbolic execution to arrays and object-oriented languages which feature dynamically allocated variables.

***Index Terms*—Automatic test generation, pointer reasoning , program verification, symbolic execution, software testing.**

I. Introduction

Symbolic execution [1] plays a crucial role in modern testing techniques, debugging, and automated program analysis. In particular, it is used for generating test cases [2], [3]. Instead of executing a program with concrete values, symbolic execution uses abstract symbols. As a consequence program expressions cannot be evaluated, and thus the program state (i.e., the assignment of values to program variables), in a symbolic execution is replaced by a substitution which assigns to each program variable an expression denoting its current value.

Symbolic execution is performed by maintaining the current symbolic state together with appropriate constraints (the so called path condition) on the abstract symbols. The path condition is refined whenever the symbolic execution reaches a branch in the control flow that depends on the symbolic state. In this case appropriate constraints are added to the path conditions, and if the new path conditions are satisfiable, then different symbolic states are considered for each branch of the execution. This way, a path condition consists of accumulating branch conditions of the control flow and thus logically characterize the particular choices underlying a specific path. Test values are generated by finding concrete values for the symbolic ones that satisfy the path conditions. For this task as well as for checking satisfiability of refined path conditions powerful satisfiability modulo theories constraint solvers are used [4].

Although symbolic execution techniques have improved enormously in the last few years with application in several software analysis tools [3], not much effort has been spent on its formal justification. Yet, from a theoretical point of view, symbolic execution provides a correct and complete testing methodology. Completeness ensures that every concrete finite execution path starting from a certain concrete input can be executed symbolically by generating a path condition satisfied by that input. Correctness is of utmost importance as it guarantees that the execution of a concrete value satisfying a path conditions should follows an identical path of the symbolic execution generating that path condition.

While correctness is relatively easy to obtain for programs with a statically fixed number of variables it becomes challenging when considering arrays and dynamically allocated variables because of aliasing. In fact, the symbolic execution community has concentrated most of the effort on effectiveness (improvement in speed-up) and significance (improvement in code coverage) and payed little attention to correctness so far [3].

Much of the code of today's programs involve various forms of arrays and dynamically allocated variables providing a challenge on how symbolic execution treat memory. While it is clear that memory should be fully symbolic [1], in the presence of aliasing, the uncertainty on the possible values of a symbolic pointer is treated either by forking the symbolic state and refining the path condition into several ones [5], or, alternatively, by using logical models for fully symbolic pointers that are supported by modern constraint solvers. The latter approach using, for example, the theory of arrays [6] has resulted in powerful symbolic execution tools handling arrays by code pre-processing [7]–[9]. Formal correctness of the theory behind these tools however is acknowledged as a potential problems that might limit the validity of the internal engine, and is validated only experimentally by testing [10].

There exist only few works on reasoning about properties of heap data structures in the context of symbolic execution, and all of them concentrate on an efficient description of

978-1-7281-5724-5/19/$31.00 ©2019 IEEE
DOI 10.1109/SYNASC49474.2019.00009

structural properties of heap structures, for example via a specification language [11]. In [12] the symbolic state is extended with a heap configuration used to maintain objects which are initialized only when they are first accessed during an execution. Similarly, to reason about aliasing, [13] models each heap as a graph, with nodes drawn from a set of objects and updated lazily. In all of the above work no explicit formal account of the underlying model of the symbolic execution, and its correctness, is presented.

The main contribution of our work fully presented in [14] is a formal definition of symbolic execution in terms of a symbolic transition system and a general definition of its correctness with respect to an operational semantics which models the actual execution on concrete values. Our general starting point is that the basic idea of symbolic execution is to represent the program state, i.e., the assignment of values to program variables, by a corresponding substitution which assigns to each program variable an expression denoting its current value. Further, symbolic execution by its very nature is *syntax-directed* which implies that the *abstraction level* of the symbolic transition system should coincide with that of the programming language. This general requirement implies that symbolic execution operates on substitutions which (only) assign *programming* expressions to the variables (and no other expressions which express properties of the run-time).

The only other approach to a formal modeling of symbolic execution, we are aware of, is the work presented in [15]. A major difference with our approach is that in [15] symbolic execution is defined in terms of a general logic (called "Reachability Logic") for the description of transition systems which abstracts from the specific characteristics of the programming language. A symbolic execution then consists basically of a sequence of logical specifications of the consecutive transitions. On the other hand, a model of the logic defines a concrete transition system. Thus correctness basically follows from the semantics of the logic. In our approach we both model symbolic execution and the concrete semantics (of any language) independently as transition systems. However, in both cases the transitions are directly defined in terms of the program to be executed. This allows to address the specific characteristics of the programming language (like dynamically allocated variables) still in a general manner. Further, the approach of [15], is not syntax oriented, and does not use concrete assignment of actual values to program variables and substitutions in symbolic states.

REFERENCES

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] E. Albert, P. Arenas, M. Gómez-Zamalloa, and J. M. Rojas, "Test case generation by symbolic execution: Basic concepts, a clp-based instance, and actor-based concurrency," in *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, 2014, pp. 263–309.

[3] R. Baldoni, E. Coppa, D. Cono D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, May 2018.

[4] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[5] M. Trtík and J. Strejček, "Symbolic memory with pointers," in *Automated Technology for Verification and Analysis*, F. Cassez and J.-F. Raskin, Eds. Springer International Publishing, 2014, pp. 380–395.

[6] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 519–531.

[7] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 209–224.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, 2008.

[9] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise pointer reasoning for dynamic test generation," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. ACM, 2009, pp. 129–140.

[10] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. ACM, 2017, pp. 68–78.

[11] P. Braione, G. Denaro, and M. Pezzè, "Symbolic execution of programs with heap inputs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 602–613.

[12] X. Deng, J. Lee, and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. IEEE Computer Society, 2006, pp. 157–166.

[13] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Halbwachs and L. D. Zuck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 365–381.

[14] F. S. de Boer and M. Bonsangue, "On the nature of symbolic execution," in *Formal Methods – The Next 30 Years*, M. H. ter Beek, A. McIver, and J. N. Oliveira, Eds. Springer International Publishing, 2019, pp. 64–80.

[15] D. Lucanu, V. Rusu, and A. Arusoaie, "A generic framework for symbolic execution," *J. Symb. Comput.*, vol. 80, no. P1, pp. 125–163, 2017.