Symbolic Execution is (Not Quite) All You Need

Version 0.0.2

Sophie Smithburg

***Abstract*—TODO**

***Index Terms*—keyword1, keyword2, keyword3**

I. Notation (Out of order, TODO: find best place for section)

- Σ is the concrete runtime space, and in our case, it is parametric over the definition of an ISA. It's basically the main memory, registers, CPU flags, any state that is documented for a particular CPU or host machine. In the case of the more abstract view, like a labeled transition system, which we focus on in this paper.
- X is computed indirectly, but can be defined directly. X is the set of all the state of the subsystem (like the programming language under analysis) that can be controlled by a program in that programming language. It is possible to extend this in the future to include input by way of any predefined input channel, but modeling codata introduces non-determinism, which is out of scope for the current paper.
- π is the mapping from Σ to X, which we get precisely by the indirect computation mechanism hinted at above.
- H is the host machine, it can be seen as an LTS or a bog standard ISA
- G is the true transition system, or notionally, what would be the specification of the programming language if the programming language were defined by the implementation? In some very real sense, a lot of programming languages are. Take, for example, C, Python, and the fact that it's been stated about the specifications in the Python enhancement proposal process and other documents in the standard that if one were to follow them, it's likely, without other guidance, they would end up with an entirely different language.
- $\tau \in \mathcal{T}$ is a trace, or a sequence of states and labels for the transitions between them, generated when we run the host machine H over the programming language implementation $\mathcal{I}$ and a program from $\mathcal{C}$, which implies all the values relevant to updates (it is of course more convenient when the values for updates are provided for us; we presume this, we think without loss of generality, but perhaps that's wrong!)
- $\mathcal{T}$ is the set of all traces τ
- G' is the transition system we extract from traces $\tau \in \mathcal{T}$ of running $H_{\mathcal{I}}$, i.e. the host machine H or host labeled transition system without loss of generality. Additionally, G' is composed from R.
- $\mathcal{I}$ the programming language implementation under analysis
- Γ is the formal grammar for $\mathcal{I}$
- $\mathrm{holes}(\gamma)$ is the set of holes in production γ
- κ_h is the sentinel value for hole h in production γ
- $\mathcal{C}$ is a covering set of programs that enumerate all productions in Γ, and alternatives in each production, in a minimized way to be elaborated upon later
- L is the set of all labels in our transition system; these are derived from the corresponding production under analysis $\gamma \in \Gamma$
- $\ell \in L$ is a label for a step in our transition system
- $R_\ell(x, x') := \mathrm{Guard}_\ell(x) \wedge \mathrm{Update}_\ell(x, x')$ summarizes the preconditions and state transformations for a step labeled ℓ
- R^* is the transitive closure of $\bigcup_\ell R_\ell$

II. Introduction

Acknowledgments

References

[1] L. Bettscheider and A. Zeller, "Look ma, no input samples! Mining input grammars from code with symbolic parsing," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*. ACM, 2024, pp. 333–337.

[2] T. A. Henzinger, R. Majumdar, and J.-F. Raskin, "A classification of symbolic transition systems," in *Proceedings of STACS 2000*, ser. Lecture Notes in Computer Science, vol. 1770. Springer, 2000, pp. 13–34.

[3] R. J. van Glabbeek, "The linear time – branching time spectrum," in *CONCUR '90: Theories of Concurrency: Unification and Extension*, ser. Lecture Notes in Computer Science, vol. 458. Springer, 1990, pp. 278–297.

[4] ——, "The linear time – branching time spectrum II: The semantics of sequential systems with silent moves," in *CONCUR '93: 4th International Conference on Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 715. Springer, 1993, pp. 66–81.

[5] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[6] J. Craaijo, F. Verbeek, and B. Ravindran, "libLISA: Instruction discovery and analysis on x86-64," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1–29, 2024.

[7] D. Lucanu, V. Rusu, and A. Arusoaie, "A generic framework for symbolic execution: A coinductive approach," *Journal of Symbolic Computation*, vol. 80, pp. 125–163, 2017.

[8] E. Voogd, Å. A. A. Kløvstad, E. B. Johnsen, and A. Wasowski, "Compositional symbolic execution semantics," *Theoretical Computer Science*, vol. 1044, p. 115263, 2025.

[9] A. Lööw, S. H. Park, D. Nantes-Sobrinho, S.-É. Ayoun, O. Sjöstedt, and P. Gardner, "Compositional symbolic execution for the next 700 memory models," *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2025.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.