

Concurrent Symbolic Execution with Trace Semantics in Coq

Åsmund Aqissiaq Arild Kløvstad

University of Oslo, Oslo, Norway

aaklovst@ifi.uio.no

Symbolic Execution (SE) is a technique for program analysis using symbolic expressions to abstract over program state, thereby covering many program states simultaneously. SE has been used since the mid 70's [5, 2] in both testing and analysis, but its formal aspects have only recently begun to be explored [1] and unified [8]. We present a Coq formalization of sections 1-3 of de Boer et al.'s work, extend the language with parallelism, and extend the model to symbolic and concrete trace semantics. Finally we consider a context-based presentation of the trace semantics in the style of Felleisen [4] and lay out a direction for future work.

Theory The most basic setting for SE, presented in [1], is a small imperative language with assignment, sequential composition, branching (*if*) and iteration (*while*). *Statements* in this language are denoted by $s, s', s_0 \dots$. Additionally we consider a set of *variables* Var and *expressions* $Expr$ consisting of variables and standard operations. Variables are denoted by $x, y, z \dots$ and expressions by $e, e', e_0 \dots$. Both expressions and statements are assumed to be well-typed.

With these definitions we can start to define symbolic semantics. The (symbolic) state is realized by a *substitution* – a function $Var \rightarrow Expr$. A substitution is denoted by $\sigma, \sigma' \dots$ and an update by $\sigma[x := e]$. Notably, since a lone variable is also an expressions, we can construct the identity substitution $id : x \mapsto x$, and a substitution can be (inductively) applied to an expression resulting in a new expression. Denote such an application by $e\sigma$. Finally, the symbolic semantics of the language are a transition system of triples (s, σ, ϕ) where ϕ is a boolean expression describing the path condition. For the concrete semantics we require a set of *values* Val . We then consider functions $V : Var \rightarrow Val$ and assume such a valuation is well-typed and can be used to evaluate an expression. Denote such a valuation by $V(e)$. Similar to symbolic semantics, concrete semantics are given by a transition system on pairs (s, V) ,

To relate symbolic and concrete states, we observe that a valuation can be “composed” with a substitution to obtain a new valuation. Denote such a composition by $V \circ \sigma$ and note that $(V \circ \sigma)(e) = V(e\sigma)$ for all expressions e . This allows us to define and prove notions of soundness and completeness, where $\rightarrow_{S/C}^*$ denotes the reflexive and transitive closure of the transition relations.

Theorem 1 (Soundness [1]) *If $(s, id, true) \rightarrow_S^* (s', \sigma, \phi)$ and $V(\phi) = true$, then $(s, V) \rightarrow_C^* (s', V \circ \sigma)$*

Intuitively, every symbolic execution whose path condition is true in the initial valuation V corresponds to a concrete execution with the same initial valuation and the final valuation is the result of composing with the symbolic substitution.

Theorem 2 (Completeness [1]) *If $(s, V) \rightarrow_C^* (s', V')$ then there exist σ, ϕ such that $(s, id, true) \rightarrow_S^* (s', \sigma, \phi)$, $V' = V \circ \sigma$ and $V(\phi) = true$.*

Intuitively, for every concrete execution there exists a symbolic execution whose path condition is satisfied by the initial valuation and whose final substitution recovers the final concrete valuation. Boer et al. then extend the language with (recursive) procedure calls. This is done by distinguishing local and global state and operating with a stack of *closures* consisting of a program fragment and the local state.

Coq Implementation We implement this basic setup in Coq. Expressions are divided into boolean and arithmetic expressions, and both expressions and statements are inductive types. Boolean expressions may contain arithmetic as the arguments to a less than or equal-operator. Valuations and substitutions are functions from strings to natural numbers and expressions respectively and expressions are evaluated by standard functions. We implement the transition relations as inductive predicates on pairs of configurations and take their reflexive and transitive closure by steps to the right. Soundness and completeness are proven by induction on this relation and a case analysis of the final step. The extension to procedure calls distinguishes local and global variables by construction, but is otherwise straightforward.

Trace semantics Our contribution extends the work of de Boer et al. to trace semantics in a language with a parallel operator $s \parallel s'$. Let symbolic and concrete traces be inductively defined by

$$\tau_S ::= [] \mid \tau_S :: (x := e) \mid \tau_S :: b \quad \text{and} \quad \tau_C ::= [] \mid \tau_C :: (x := v)$$

where b is a boolean expression and v a concrete value. The final state of a trace can be recovered from an initial state by folding over the trace and updating the state. Denote the result by $\tau \Downarrow_{V_0}$. Note that this is *substitution* in the case of symbolic traces, but a *valuation* in the case of concrete traces. The path condition of symbolic traces can also be recovered as the conjunction of all its boolean expressions. Denote this path condition by $pc(\tau)$. The semantics are given by a transition system on pairs (s, τ) – with τ symbolic or concrete – parametrized by an initial state. Let \rightarrow_σ denote a symbolic transition from initial state σ , and \rightarrow_V a concrete transition from initial state V . We also recover soundness and completeness in the style of de Boer et al.

Theorem 3 (Trace Soundness) *If $(s, []) \rightarrow_{id}^* (s', \tau_S)$ and $V_0(pc(\tau_S)) = \text{true}$, then there exists τ_C s.t. $(s, []) \rightarrow_{V_0}^* (s', \tau_C)$ and $V_0 \circ (\tau_S \Downarrow_{id}) = \tau_C \Downarrow_{V_0}$*

Theorem 4 (Trace Completeness) *If $(s, []) \rightarrow_{V_0}^* (s', \tau_C)$ there exist τ_S s.t. $(s, []) \rightarrow_{id}^* (s', \tau_S)$, $V_0 \circ (\tau_S \Downarrow_{id}) = \tau_C \Downarrow_{V_0}$, and $V_0(pc(\tau_S)) = \text{true}$*

Both of these theorems are proven for the language extended with procedure calls or parallel composition by induction on the transition relation¹.

Discussion As an alternative to de Boer et al.’s transition system semantics we implement reduction-in-context semantics in the style of Felleisen [4]. With this approach we consider configurations of the form $(C[s], \tau)$ where $C[s]$ represent the program fragment s in the context C . This allows us to define a generic machinery for reductions in context and a smaller “head reduction” relation on (s, τ) specific to the symbolic or concrete semantics at hand. The Coq implementation is inspired by the approach of Xavier Leroy’s course on mechanized semantics [6]. Trace soundness and completeness can be stated as before, and are proven by induction on the reduction relation.

Conclusion and Future Work We have provided an overview of de Boer and Bonsangue’s formalization of SE in Coq. Additionally we extended the formalization to include parallel composition and trace semantics, and consider an alternative implementation of the semantics which results in more generic machinery that can be re-used in the symbolic and concrete case. In the future we aim to use this mechanized framework to investigate reduction techniques for symbolic execution in concurrent settings and define useful notions of sound- and completeness for such reductions.

¹In fact they both hold for appropriate initial traces (not just $[]$) but this formulation provides more clutter than insight

References

- [1] Frank S de Boer & Marcello Bonsangue (2021): *Symbolic execution formally explained*. *Formal Aspects of Computing* 33(4), pp. 617–636.
- [2] Robert S Boyer, Bernard Elspas & Karl N Levitt (1975): *SELECT—a formal system for testing and debugging programs by symbolic execution*. *ACM SigPlan Notices* 10(6), pp. 234–245.
- [3] Crystal Chang Din, Reiner Hähnle, Ludovic Henrio, Einar Broch Johnsen, Violet Ka I Pun & Silvia Lizeth Tapia Tarifa (2022): *LAGC Semantics of Concurrent Programming Languages*. *arXiv preprint arXiv:2202.12195*.
- [4] Matthias Felleisen & Robert Hieb (1992): *The revised report on the syntactic theories of sequential control and state*. *Theoretical Computer Science* 103(2), pp. 235–271, doi:[https://doi.org/10.1016/0304-3975\(92\)90014-7](https://doi.org/10.1016/0304-3975(92)90014-7). Available at <https://www.sciencedirect.com/science/article/pii/0304397592900147>.
- [5] James C King (1976): *Symbolic execution and program testing*. *Communications of the ACM* 19(7), pp. 385–394.
- [6] Xavier Leroy (2020): *Mechanized Semantics*. Course materials. Available at <https://github.com/xavierleroy/cdf-mech-sem>.
- [7] Antoni Mazurkiewicz (1977): *Concurrent Program Schemes and their Interpretations*. *DAIMI Report Series* 6(78), doi:10.7146/dpb.v6i78.7691. Available at <https://tidsskrift.dk/daimipb/article/view/7691>.
- [8] Dominic Steinhöfel (2022): *Symbolic Execution: Foundations, Techniques, Applications, and Future Perspectives*, pp. 446–480. Springer International Publishing, Cham, doi:10.1007/978-3-031-08166-8_22. Available at https://doi.org/10.1007/978-3-031-08166-8_22.

A Transition Rules

A.1 Transition relation for symbolic execution

$$\begin{aligned}
 (x := e; s, \sigma, \phi) &\rightarrow_S (s, \sigma[x := e\sigma], \phi) \\
 (\text{if } e \{s_1\}\{s_2\}; s, \sigma, \phi) &\rightarrow_S (s_1; s, \sigma, \phi \wedge e\sigma) \\
 (\text{if } e \{s_1\}\{s_2\}; s, \sigma, \phi) &\rightarrow_S (s_2; s, \sigma, \phi \wedge \neg e\sigma) \\
 (\text{while } e \{s\}; s', \sigma, \phi) &\rightarrow_S (s; \text{while } e \{s\}; s', \sigma, \phi \wedge e\sigma) \\
 (\text{while } e \{s\}; s', \sigma, \phi) &\rightarrow_S (s', \sigma, \phi \wedge \neg e\sigma)
 \end{aligned}$$

A.2 Transition relation for concrete execution

$$\begin{aligned}
 (x := e; s, V) &\rightarrow_C (s, V[x := V(e)]) \\
 (\text{if } e \{s_1\}\{s_2\}; s, V) &\rightarrow_C (s_1; s, V) \text{ if } V(e) = \text{true} \\
 (\text{if } e \{s_1\}\{s_2\}; s, V) &\rightarrow_C (s_2; s, V) \text{ if } V(e) = \text{false} \\
 (\text{while } e \{s\}; s', V) &\rightarrow_C (s; \text{while } e \{s\}; s', V) \text{ if } V(e) = \text{true} \\
 (\text{while } e \{s\}; s', V) &\rightarrow_C (s', V) \text{ if } V(e) = \text{false}
 \end{aligned}$$

B Reduction rules

B.1 Generic Context Rules

$$\begin{array}{c}
\frac{}{is_context(s)} \text{identity context} \\
\frac{is_context(C)}{is_context(C[s]||s')} \text{left-par} \\
\frac{is_context(C)}{is_context(C[s];s')} \text{sequence} \\
\frac{is_context(C)}{is_context(s'||C[s])} \text{right-par} \\
\frac{(s,t) \rightarrow_{head} (s',t') \quad is_context(C)}{(C[s],t) \rightarrow (C[s'],t')}
\end{array}$$

Where \rightarrow_{head} is either the symbolic or concrete head reduction relation given below:

B.2 Head Reduction Rules

Symbolic head reduction

$$\begin{aligned}
&(x := e, \tau) \rightarrow (skip, \tau :: (x := e)) \\
&(if\ b\ \{s_1\}\{s_2\}, \tau) \rightarrow (s_1, \tau :: b) \\
&(if\ b\ \{s_1\}\{s_2\}, \tau) \rightarrow (s_2, \tau :: \neg b) \\
&(skip; s, \tau) \rightarrow (s, \tau) \\
&(skip||s, \tau) \rightarrow (s, \tau) \\
&(s||skip, \tau) \rightarrow (s, \tau)
\end{aligned}$$

Concrete head reduction

$$\begin{aligned}
&(x := e, \tau) \rightarrow (skip, \tau :: (x := V(e))) \\
&(if\ b\ \{s_1\}\{s_2\}, \tau) \rightarrow (if\ V(b)\ then\ s_1\ else\ s_2, \tau) \\
&(skip; s, \tau) \rightarrow (s, \tau) \\
&(skip||s, \tau) \rightarrow (s, \tau) \\
&(s||skip, \tau) \rightarrow (s, \tau)
\end{aligned}$$