(Challenges of?) Formalizing Concurrent Symbolic Execution with Trace Semantics in Coq

Åsmund Aqissiaq Arild Kløvstad
UiO
Oslo, Norway
Institute of Informatics
PSY, maybe?
aaklovst@ifi.uio.no

Eduard & and/or others?
UiO
Oslo, Norway
Institute of Informatics
PSY, maybe?

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 [4, 2], but its formal aspects have only recently begun to be explored by – among others– de Boer [1] and Steinhöfel [5].

Something about trace semantics

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 trace semantics. Finally we consider a context-based presentation of the semantics in the style of Felleisen [3] and compare the two approaches.

Introduction

- Quick def. of symbolic execution
- Recent formalization ([1])
- Concurrency and reduction
- plan for the rest of this

Frank and Marcello's transition systems (in Coq)

This ends up rehashing a lot and taking a lot of space. Can I say "just go read Frank and Marcello's paper" and/or just list definitions?

- $(s_0; s, \sigma) \to_S (s, \sigma')$
- correctness/soundness and completeness ($V \circ \sigma$)
- proofs by induction on steps
- variations (IMP, procedure calls)

This section gives an overview of de Boer et al.'s formalization of symbolic execution and our Coq implementation of it. It consists of both symbolic and concrete semantics realized as transition systems as well as correctness and completeness results relating the two.

The most basic setting for symbolic execution, presented in [1, sec. 2], is a small imperative language with assignment, sequential composition, branching (`if`) and iteration (`while`). *Statements* in this language are denoted by $s, s', s_0 \ldots$ Additionally we consider a set of *variables* Var and *expressions* $Expr$

Submitted to:
PLACES 2023

consisting of variables and standard operations. Variables are denoted by $x, y, z...$ and expressions by $e, e', e_0...$ Both expressions and statements are assumed to be well-typed.

In the Coq implementation expressions are divided into boolean and arithmetic expressions, and both expressions and statements are inductive types. *Var* is the type of strings.

With these definitions we can start to define symbolic semantics. The (symbolic) state is realized by a *substitution* – a function $Var \to Expr$. A substitution is denoted by $\sigma, \sigma'...$ and an update by $\sigma[x := e]$. Notably, since a lone variable is also an expressions, we can construct the identity substitution $x \mapsto x$, 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 example, symbolic assignment and branching are realized by

$$(x := e; s, \sigma, \phi) \to_S (s, \sigma[x := e\sigma], \phi)$$

$$(if\ e\ \{s_1\}\{s_2\}; s, \sigma, \phi) \to_S (s_1; s, \sigma, \phi \wedge e\sigma)$$

$$(if\ e\ \{s_1\}\{s_2\}; s, \sigma, \phi) \to_S (s_2; s, \sigma, \phi \wedge \neg e\sigma)$$

For the concrete semantics we require a set of *values Val*. We then consider functions $V : Var \to Val$ and assume such a valuation is well-typed and can be used to evaluate an expression. Denote such a valuation by $V(e)$. In Coq a valuation is simply a function from variables to natural numbers and boolean/arithmetic expressions are evaluated to booleans and naturals respectively.

Similar to symbolic semantics, concrete semantics are given by a transition system on pairs (s, V). For example:

$$(x := e; s, V) \to_C (s, V[x := V(e)])$$

define valuation update?

$$(if\ e\ \{s_1\}\{s_2\}; s, V) \to_C (s_1; s, V), \text{ if V(e) = true}$$

$$(if\ e\ \{s_1\}\{s_2\}; s, V) \to_C (s_2; s, V), \text{ if V(e) = false}$$

these examples take a lot of space, but are maybe more useful than in-text definitions

For both transition systems we can take the reflexive and transitive closures $\to^*_{S/C}$, in Coq implemented stepwise to the right.

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 (proof by induction on expressions).

This allows us to define and prove notions of soundness and completeness.

Theorem 1 (Soundness) *If* $(s, id, true) \to^*_S (s', \sigma, \phi)$ *and* $V(\phi) = true$, *then* $(s, V) \to^*_C (s', V \circ \sigma)$

Theorem 2 (Completeness) *If* $(s, V) \to^*_C (s', V')$ *then there exist* σ, ϕ *such that* $(s, id, true) \to^*_S (s', \sigma, \phi)$, $V' = V \circ \sigma$ *and* $V(\phi) = true$.

Both theorems are proven in Coq by induction on the transition relations and a case analysis of the final step.

Section 3 of de Boer et al. extends the language with (potentially 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.

In the Coq mechanization the types of local and global variables are distinct by construction, and otherwise the approach is analoguous to the basic language described above.

Trace semantics

- $(s_0; s, \tau) \rightarrow_S (s, \tau') + (V \circ \tau \downarrow)$
- $(s, \tau) \rightarrow_S (s', \tau')$ with skip
- Head-reductions in context (cite Xavier Leroy and Felleisen)
- pros and cons?
  - fewer rules
  - skip-special case

We extend the work of de Boer et al. to trace semantics in a language with a parallel operator. Let symbolic traces be inductively defined by

$$\tau := [\,] \mid \tau :: (x := e) \mid \tau :: b, b \text{ a boolean expression}$$

and concrete traces by

$$\tau := [\,] \mid \tau :: (x := v), v \in Val$$

The final (symbolic) state of a (symbolic) trace can be recovered from an initial state by folding over the trace and updating the state. Denote the result by $\tau \Downarrow$. 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 $pc(\tau)$.

Thus the semantics are given by a transition system on pairs (s, τ) – with τ symbolic or concrete – parametrized by an initial state. Let $\rightarrow_\sigma$ denote a symbolic transition from initial state σ, and $\rightarrow_V$ a concrete transition from initial state V.

We also recover soundness in completeness in the style of de Boer et al.

Theorem 3 (Trace Soundness) *If* $(s, [\,]) \rightarrow^*_{id} (s', \tau)$ *and* $V_0(pc(\tau)) = true$, *then there exists* τ' *s.t* $(s, [\,]) \rightarrow^*_{V_0} (s', \tau')$ *and* $V_0 \circ (\tau \Downarrow) = \tau' \Downarrow$

Theorem 4 (Trace Completeness) *If* $(s, [\,]) \rightarrow^*_{V_0} (s', \tau)$ *there exist* τ' *s.t* $(s, [\,]) \rightarrow^*_{id} (s', \tau')$, $V_0 \circ (\tau' \Downarrow) = \tau \Downarrow$, *and* $V_0(pc(\tau')) = true$

Both of these theorems are proven for the language extended with procedure calls or parallel composition by induction on the transition relation [1].

Reduction Semantics

As an alternative to de Boer et al.'s transition system semantics we implement reduction-in-context semantics in the style of Felleisen [3].

With this approach we consider configurations of the form $(C[s], \tau)$ which 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. examples?

The Coq implementation is inspired by the approach of Xavier Leroy in his course on mechanized semantics [2]. Trace soundness and completeness can be stated as before, and proven by induction on the reduction relation.

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

[2] `https://github.com/xavierleroy/cdf-mech-sem/blob/f8dc6f7e2cb42f0861406b2fa113e2a7e825c5f3/FUN.v`

Trace reduction? / future work

- by equivalence relation
- sound: can continue from equivalent trace (easy)
- complete: don't reduce away necessary traces (hard)

Using the mechanized framework described above, we aim to investigate reduction techniques for symbolic execution and define useful notions of sound- and completeness for such reductions.

Conclusion

We have provided an of de Boer and Bonsangue's formalization of symbolic execution in Coq. Additionally we extend 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.

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] 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. Available at `https://www.sciencedirect.com/science/article/pii/0304397592900147`.

[4] James C King (1976): *Symbolic execution and program testing*. *Communications of the ACM* 19(7), pp. 385–394.

[5] Dominic Steinhöfel (2020): *Abstract execution: automatically proving infinitely many programs*. Ph.D. thesis, Technische Universität.