Denotational Semantics for Symbolic Execution

Erik Voogd, Åsmund Aqissiaq Arild Kløvstad, and Einar Broch Johnsen

Department of Informatics, University of Oslo, Oslo, Norway
`{erikvoogd,aaklovst,einarj}@ifi.uio.no`

Abstract. Symbolic execution is a technique to systematically explore all possible paths through a program. This technique can be formally explained by means of small-step transition systems that update symbolic states and compute a precondition corresponding to the taken execution path (called the path condition). To enable swift and robust compositional reasoning, this paper defines a denotational semantics for symbolic execution. We prove the correspondence between the denotational semantics and both the small-step execution semantics and a concrete semantics. The denotational semantics is a function defined piecewise using a partitioning of the input space. Each part of the input space is a path condition obtained from symbolic execution, and the semantics of this part is the corresponding symbolic substitution interpreted as a function on the initial state space. Correctness and completeness of symbolic execution is encapsulated in a graceful identity of functions. We provide mechanizations in Coq for our main results.

Keywords: Formal methods, Programming semantics, Denotational semantics, Symbolic execution

1 Introduction

Major successes in program analysis, particularly for debugging, test case generation, and verification, have been achieved by *symbolic execution* [2–6, 8–10]: a powerful simulation technique in which symbolic states represent a wide range of concrete program states. It has only recently been formalized and proven correct [2] with respect to a concrete operational semantics.

With symbolic execution, program states associate program variables to symbolic expressions rather than concrete values. Assignments in the program can then be understood as updating the symbolic state through substitutions σ. When encountering control-flow statements guarded by Boolean expressions, no concrete choice can be made. Instead, the transition system modeling symbolic execution branches in both possible directions (theoretically using nondeterminism, in practice exploring both branches), and updates its own state by storing the Boolean guard under substitution. It thus generates the *path condition* ϕ: an aggregation of all Boolean control-flow guards under substitution. If a program p has a finite trace in the symbolic execution system that ends in a symbolic state (σ, ϕ), then the final state of a concrete execution of p on an initial state satisfying ϕ can also be obtained by performing the substitution σ on that intial state. Thus, symbolic execution is really a way to partition program behavior into different branches, where behavior of the branch is tracked by the symbolic subsitution σ, and the path conditions ϕ are preconditions specifying the branch that is taken.

Symbolic execution is a compelling technique for verification purposes: given a postcondition ψ and a terminated symbolic execution (σ, ϕ) of a program p, the formula $\phi \wedge \sigma\,\psi$ is a precondition for ψ. The path condition ϕ ensures that program behavior corresponds to σ, and applying σ to the postcondition ψ – this is done variable-wise – is a way of inverting program behavior corresponding to σ on the set specified by ψ—this inversion is made formal later in (3). Ranging over the (possibly infinite) set of terminated symbolic executions, taking the disjunction of all the formulae $\phi \wedge \sigma\,\psi$ yields the *weakest* precondition in disjunctive normal form.

All this and more can be reasoned about more effectively when symbolic execution is equipped with a denotational semantics, which is the goal of this work. To this end, two seemingly obvious, yet crucial observations are in order:

- The symbolic substitutions σ are syntactic objects representing functions that denotationally transform concrete initial states (cf. (1) in Section 4).
- The path conditions ϕ are syntactic objects representing subsets of the initial state space, and form a *subpartition* for it. For programs that terminate on all inputs, the path conditions form a *partition* of the input space.

It is well-understood that syntactically performing a substitution within a substitution means to do function composition. This means that symbolic execution traces can be denotationally composed by performing nested substitutions. A natural question to ask now is: what happens to the path condition when we compose symbolic execution traces?

Example. As a simple example, consider a program[1] that stipulates the behavior of the *absolute value* function for real numbers:

$$p_{\text{abs}} \equiv \texttt{if (x<0) \{ x := -x; \} else \{ Skip; \}}$$

Symbolically executing a program is usually done starting from the *initial* configuration $(\sigma_0, \top)$: the identity substitution σ_0 along with the path condition $\top$ (true) specifying the entire input space. The program p_{abs} above has two symbolic executions, both terminating. One of these executions is

$$(p_{\text{abs}}, \sigma_0, \top) \rightsquigarrow (\texttt{x:=-x}, \sigma_0, \top \wedge \texttt{x} < 0) \rightsquigarrow (\texttt{Skip}, (\texttt{x} \mapsto -\texttt{x}), \top \wedge \texttt{x} < 0)$$

where `Skip` is the terminated program. The first step analyzes the *if* statement, in this case picks the *true* branch, and updates the path condition accordingly with the conjunct $\texttt{x} < 0$. The second step analyzes the assignment $\texttt{x:=} -\texttt{x}$ and

[1] Symbolic execution may seem to be a trivial exercise for this simple program, but note that, as programs grow, it is highly effective in several areas of program analysis.

updates the substitution accordingly. Hence, we have $(p_{\mathrm{abs}}, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\mathtt{Skip}, \sigma, \phi)$ where σ and ϕ are as above, and $\overset{*}{\rightsquigarrow}$ is a reflexive-transitive closure.

Suppose now that we have analyzed two programs p and q and obtained

$$(p, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\mathtt{Skip}, \sigma_p, \phi_p) \qquad \text{and} \qquad (q, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\mathtt{Skip}, \sigma_q, \phi_q)$$

It is not straightforward from the usual small-step symbolic operational semantics how these two traces compose to the sequenced program $p \fatsemi q$. This is because the second execution does not continue from the configuration where the first one left off; it used the usual initial configuration. One would *expect* to obtain a symbolic trace

$$(p \fatsemi q, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\mathtt{Skip}, \sigma, \phi)$$

where σ is σ_p *within* σ_q (syntactically), meaning σ_q *after* σ_p when interpreted as functions. Regarding the path condition, one will expect to obtain $\phi = \phi_p \wedge \sigma_p\, \phi_q$ (i.e., σ_p applied to all the variables occuring in ϕ_q), since executing p yields ϕ_p, and executing q yields ϕ_q, but this time we started from σ_p instead of σ_0.

Contribution. These and similar facts regarding compositionality of symbolic execution traces are not easily proven using small-step transition systems. In this paper, we introduce *denotational semantics for symbolic execution* to support such compositional reasoning. Historically, denotational semantics have been very effective for compositional reasoning, enabling swift and potent reasoning about programs. Our experience in reasoning about symbolic execution for, e.g., concurrent or probabilistic programs [19], has shown us that this novel view of symbolic execution is fruitful and, for some proofs, even necessary. Example 6 in Section 4 illustrates how compositionality of sequencing in symbolic execution can be applied using our denotational semantics.

This new denotational semantics for symbolic execution formalizes the ideas described above: symbolic substitutions σ are interpreted as functions $|\sigma|$ on the initial state space, and the collection of path conditions ϕ are interpreted as a partition of the initial state space where execution terminates. The denotational semantics (presented in Definition 1) then *selects* the right partition ϕ_i of the initial state space, and picks the corresponding function $|\sigma_i|$. This selection process is informally denoted by $\bigoplus$ in the pictorial representation of the denotational semantics in Figure 1 (right).

To introduce this new semantics, we use the toy language While, presented in Section 2, which supports unbounded loops. We describe its concrete denotational semantics as a recursively defined function. After that, in Section 3, we introduce our main contribution: a denotational semantics for symbolic execution. This semantics corresponds to the concrete semantics given in Section 2, as stated in Theorem 1. This result, which is simply a very graceful identity of functions, trickles down to correctness and completeness of the transition systems implementing concrete and symbolic execution.

In Section 4, we present symbolic execution in two equivalent ways: first, as done by De Boer and Bonsangue [2], we extract *traces*—finite lists of assignments and Boolean assertions—from programs, and define the subsitutions

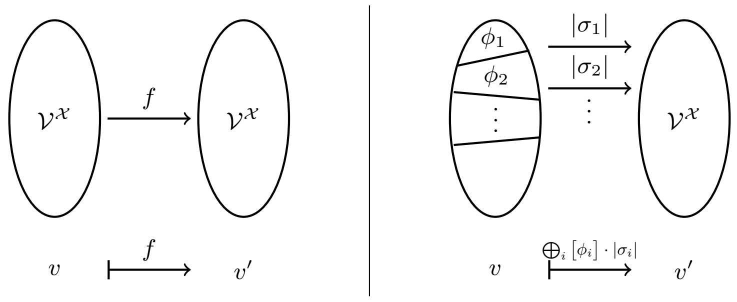
$\mathcal{V}^{\mathcal{X}}$
f
$\mathcal{V}^{\mathcal{X}}$
$v \xmapsto{f} v'$
ϕ_1
ϕ_2
$|\sigma_1|$
$|\sigma_2|$
$\mathcal{V}^{\mathcal{X}}$
$v \xmapsto{\bigoplus_i [\phi_i] \cdot |\sigma_i|} v'$

Fig. 1: Pictorial representations of denotational semantics (left concrete, right symbolic execution). $\mathcal{V}$ is the value space; $\mathcal{X}$ the set of variables

and path conditions on these traces. Every trace constitutes a part of the denotational semantics presented in Section 3, as stated in Theorem 2. Second, the substitutions and path conditions can be directly generated by a transition system—this is more in line with implementation practice. We provide a proof of correspondence between these two approaches to symbolic execution in Proposition 1. Most results up until Section 4 have been mechanized[2] in the Coq theorem prover [7]—they are labeled with the symbol 🐓. In Section 5, we discuss a straightforward extension of our work to procedure calls, with support for mutual recursion.

2 The Language While

In the language While, programs p are generated by means of assignments `x:=`e of expressions e to variables `x` (free of side-effects), sequencing, conditional branching and unbounded loops. Expressions $e \in \mathbb{E}$ are generated by operators `op` over the variables `x` $\in \mathcal{X}$. Zero-arity operators can be considered constants (in $\mathbb{Q}$, for example). There is a distinct subset $\mathbb{BE} \subseteq \mathbb{E}$ of *Boolean expressions* $b \in \mathbb{BE}$ that are used for branching and loops. There are at least the following three distinguished operators: the constant *truth* $\top \in \mathbb{BE}$, the unary operator $\neg$ for negation, and the binary operator $\wedge$ for conjunction.

$$\begin{array}{lcl} e \in \mathbb{E} & ::= & \mathtt{x} \\ & \mid & \mathtt{op}(e_1, \ldots, e_n) \\[1em] p \in \mathbb{P} & ::= & \mathtt{Skip} \\ & \mid & \mathtt{x} := e \\ & \mid & p \,⨾\, p \\ & \mid & \mathtt{if}\ b\ p\ p \\ & \mid & \mathtt{while}\ b\ p \end{array}$$

Concrete semantics. Variables $\mathtt{x} \in \mathcal{X}$ take values in a value space $\mathcal{V}$. To evaluate Boolean expressions $b \in \mathbb{BE}$, we assume there is a distinguished *truth value* $\mathbf{1} \in \mathcal{V}$. A (concrete) program state is a *valuation* $v : \mathcal{X} \to \mathcal{V}$, or $v \in \mathcal{V}^{\mathcal{X}}$ that assigns a value to each program variable. The *updated* valuation $v[\mathtt{y} \mapsto a]$ (some $a \in \mathcal{V}$) denotes the valuation v' for which $v'(\mathtt{x}) = v(\mathtt{x})$ if $\mathtt{x} \neq \mathtt{y}$ and $v'(\mathtt{y}) = a$.

[2] The mechanized theory is available at https://doi.org/10.5281/zenodo.8096802.

Expressions e are functions $|e| : \mathcal{V}^{\mathcal{X}} \to \mathcal{V}$ such that $|\mathtt{x}|(v) = v(\mathtt{x})$, and evaluated recursively. That is, $|\mathtt{op}(e_1, \ldots, e_n)|(v) = \overline{\mathtt{op}}(|e_1|(v), \ldots, |e_n|(v))$, where $\overline{\mathtt{op}}$ denotes the interpretation of the operator. The Boolean expressions $b \in \mathbb{BE}$ of branching and iteration conditions are interpreted as an indicator function. That is, v satisfies b (by definition), written $v \vDash b$, if and only if $|b|(v) = \mathbf{1}$. Boolean expressions b may thus be interpreted as *subsets* $|b| \subseteq \mathcal{V}^{\mathcal{X}}$ of the state space, where $v \in |b|$ iff $|b|(v) = \mathbf{1}$. The Boolean expression $\top$ is defined as $|\top| = \mathcal{V}^{\mathcal{X}}$. Negation is interpreted as set complement in $\mathcal{V}^{\mathcal{X}}$ and conjunction is set intersection. We sometimes write e or b in lieu of $|e|$ or $|b|$.

The semantics of programs $p \in \mathbb{P}$ are partial functions $f_p : \mathcal{V}^{\mathcal{X}} \rightharpoonup \mathcal{V}^{\mathcal{X}}$ defined inductively as:

$$f_p : v \mapsto \begin{cases} v & \text{if } p = \mathtt{Skip} \\ v[\mathtt{x} \mapsto e(v)] & \text{if } p = \mathtt{x}\mathtt{:=}e \\ (f_{p_2} \circ f_{p_1})(v) & \text{if } p = p_1 \fatsemi p_2 \\ f_{p_1}(v) & \text{if } p = \mathtt{if}\ b\ p_1\ p_2 \text{ and } v \vDash b \\ f_{p_2}(v) & \text{if } p = \mathtt{if}\ b\ p_1\ p_2 \text{ and } v \nvDash b \\ f_q^m(v) & \text{if } p = \mathtt{while}\ b\ q, \text{ where } m := \min\{j \in \mathbb{N} : f_q^j(v) \nvDash b\} \end{cases}$$

Here, f^m denotes m-fold iterated applications of f (and identity for $m = 0$). Partiality of a function arises when *while* loops diverge: there may not exist $j \in \mathbb{N}$ such that $f_q^j(v) \notin b$. If p is undefined for input v, we write $f_p(v)\!\uparrow$. On the other hand, if p *is* defined for v, we write $f_p(v)\!\downarrow$.

The definition of the partial function for the while case is equivalent to a least fixed point construction using total functions, extending the codomain with *undefinedness* $(\bot)$. The partial order of functions is pointwise, and the relation on $\mathcal{V}^{\mathcal{X}} \cup \{\bot\}$ is the identity unioned with $\{\bot \leq v \mid v \in \mathcal{V}^{\mathcal{X}}\}$.

Example 1. Consider the program p_{abs} from Section 1, and let $\mathcal{V} = \mathbb{Z}$ and $\mathcal{X} = \{\mathtt{x}\}$, so $\mathcal{V}^{\mathcal{X}} = \mathbb{Z}$. We have, e.g., $f_{p_{\text{abs}}} : -2 \mapsto 2$ and $f_{p_{\text{abs}}} : 42 \mapsto 42$.

3 Symbolic Execution Semantics

We now turn to the central definition in this work. The denotational semantics for symbolic execution is defined using the subset $\mathbb{F}_p \subseteq (\mathcal{V}^{\mathcal{X}} \to \mathcal{V}^{\mathcal{X}}) \times \mathcal{P}(\mathcal{V}^{\mathcal{X}})$, defined inductively below over the structure of p. The semantics of a program p will then be a piecewise definition of pairs $(F, B) \in \mathbb{F}_p$. We therefore refer to an element $(F, B) \in \mathbb{F}_p$ as a *piece* of p; F is the piece *behavior* and B is the piece *precondition.* Every piece corresponds to a symbolic execution, as we will show later.

– For *inaction*, the state remains unaltered and there is no restriction on the precondition:

$$\mathbb{F}_{\mathtt{Skip}} := \{(v \mapsto v, \mathcal{V}^{\mathcal{X}})\}$$

- An *assignment* has no restriction on the precondition, but the state is updated according to the assignment:

$$\mathbb{F}_{\texttt{x:=}\,e} := \{(v \mapsto v[\texttt{x} \mapsto e(v)], \mathcal{V}^{\mathcal{X}})\}$$

- When *sequencing* two programs p and q, range over all pairs of executions and compose them. The first precondition should be satisfied and, after executing the first component, the second precondition should be satisfied:

$$\mathbb{F}_{p\,\text{;}\,q} := \{(F_2 \circ F_1, B_1 \cap F_1^{-1}[B_2]) : (F_1, B_1) \in \mathbb{F}_p, (F_2, B_2) \in \mathbb{F}_q\}$$

For later use, we will also denote this structure by $\mathbb{F}_q \odot \mathbb{F}_p$.

- The two branches of an *if* statement are put together in a union of sets—the precondition is updated accordingly ($-^{\complement}$ denotes complement):

$$\mathbb{F}_{\texttt{if}\ b\ p\ q} := \{(F, B \cap b\ ) : (F, B) \in \mathbb{F}_p\} \cup \{(F, B \cap b^{\complement}) : (F, B) \in \mathbb{F}_q\}$$

- In a *while* statement, the disjoint union is for every possible number of iterations m. For $m = 0$, the behavior is that of `Skip`, $v \mapsto v$, and the precondition is the negation of the Boolean formula. Every next number $m + 1$ of loop iterations takes all possible executions of m iterations, pre-composes all possible additional iterations, and updates the preconditions accordingly:

$$\mathbb{F}_{\texttt{while}\ b\ p} := \bigcup_{m=0}^{\infty} (\Omega_{b,p})^m \{(v \mapsto v, b^{\complement})\},$$

where $(\Omega_{b,p})^m$ denotes m applications of the mapping $\Omega_{b,p}$ from $(\mathcal{V}^{\mathcal{X}} \to \mathcal{V}^{\mathcal{X}}) \times \mathcal{P}(\mathcal{V}^{\mathcal{X}})$ to itself that pre-composes an additional iteration of the loop:

$$\Omega_{b,p} : \mathbb{F} \mapsto \{(F \circ F_p, b \cap B_p \cap F_p^{-1}[B]) : (F, B) \in \mathbb{F}, (F_p, B_p) \in \mathbb{F}_p\}$$

Example 2. For the program p_{abs} from Section 1, with $\mathcal{V}^{\mathcal{X}} = \mathbb{Z}$, we have $\mathbb{F}_{p_{\text{abs}}} = \{(F_1, B_1), (F_2, B_2)\}$, where $F_1 : x \mapsto -x$ and $B_1 = \mathbb{Z}_{<0}$; $F_2 : x \mapsto x$ and $B_2 = \mathbb{N}$.

The preconditions form a *subpartition* of the input space; they may not cover the whole input space due to non-termination:

Lemma 1 (Pairwise Disjoint Preconditions). *Let* $(F, B), (F', B') \in \mathbb{F}_p$. *If* $B \cap B' \neq \emptyset$ *then* $(F, B) = (F', B')$.

Proof (Sketch). By induction on the structure of p. The base cases vacuously hold because the $\mathbb{F}_p$ are singletons. The inductive steps are mechanically verified.

The lemma justifies the following definition, where a unique (F, B), if it exists, is picked:

Definition 1 (Denotational Semantics of Symbolic Execution). *Let p be a program. The* symbolic semantics *of p is the partial function $\mathcal{F}_p : \mathcal{V}^{\mathcal{X}} \rightharpoonup \mathcal{V}^{\mathcal{X}}$ defined by*

$$\mathcal{F}_p : v \mapsto \begin{cases} F(v) & \text{if } (F, B) \in \mathbb{F}_p \text{ s.t. } v \in B \\ \text{undefined} & \text{otherwise} \end{cases}$$

With this semantics, correctness and completeness of symbolic execution with respect to concrete execution are encapsulated in one elegant identity:

Theorem 1 (Concrete Correspondence). *For all p, $f_p = \mathcal{F}_p$.*

4 Symbolic Execution

The semantics described in Section 3 is a denotational semantics for symbolic execution systems, such as the one described by De Boer and Bonsangue [2]. We will provide a detailed proof of this in the sequel, by defining *traces through a program* and showing that each $(F, B) \in \mathbb{F}_p$ corresponds to a trace. Every such trace corresponds to a final substitution and path condition obtained from symbolic execution.

Traces form a subclass of programs that are free of branching and loops. The syntax of traces is generated by the following grammar:

$$\mathbb{T} \ni t \ ::= \ (\quad \texttt{x:=}e \quad | \quad b \quad)^*$$

They are finite lists of assignments and Boolean assertions.

Traces are extracted from a program p through a nondeterministic transition relation $\longrightarrow \subseteq (\mathbb{P} \times \mathbb{T}) \times (\mathbb{P} \times \mathbb{T})$. The following symbolic transition rules implement the extraction of traces; we write $\cdot$ to attach an element at the end of the list (and later also overload it to denote concatenation of traces, and furthermore for deconstruction):

$$\boxed{\begin{array}{ll}
(\texttt{if}\ b\ p_1\ p_2, t) \longrightarrow (p_1, t \cdot b) & (\texttt{x:=}e, t) \longrightarrow (\texttt{Skip}, t \cdot \texttt{x:=}e) \\
(\texttt{if}\ b\ p_1\ p_2, t) \longrightarrow (p_2, t \cdot \neg b) & (\texttt{Skip} \fatsemi p, t) \longrightarrow (p, t) \\
(\texttt{while}\ b\ p, t) \longrightarrow (p \fatsemi \texttt{while}\ b\ p, t \cdot b) & \dfrac{(p, t) \longrightarrow (p', t')}{(p \fatsemi q, t) \longrightarrow (p' \fatsemi q, t')} \\
(\texttt{while}\ b\ p, t) \longrightarrow (\texttt{Skip}, t \cdot \neg b) &
\end{array}}$$

Fig. 2: Inductive transition rules for trace extraction

The reflexive-transitive closure $\xrightarrow{*}$, starting from the empty trace ε, produces all finite traces through a program p:

$$\mathcal{T}_p := \{t \in \mathbb{T} : (p, \varepsilon) \xrightarrow{*} (\texttt{Skip}, t)\}.$$

The unfolding of while loops produces infinite traces (not considered in $\mathcal{T}_p$). The system is progressive; the program `Skip` is the only one that cannot make a transition, and is considered the terminated program. Nondeterminism arises only from the outgoing transitions from *if* and *while* statements.

Example 3. The program p_{abs} from Section 1 has two traces: $(\texttt{x} < 0) \cdot (\texttt{x:=} -\texttt{x})$ and $\neg(\texttt{x} < 0)$.

If $(p, s) \xrightarrow{*} (q, u)$ then s is a prefix of u, i.e., $u = s \cdot t$ for some trace t. Moreover, $(p, s) \xrightarrow{*} (q, s \cdot t)$ if and only if $(p, \varepsilon) \xrightarrow{*} (q, t)$. Sequencing of programs is concatenation of traces: $u \in \mathcal{T}_{p \,;\, q}$ if and only if there are $s \in \mathcal{T}_p$ and $t \in \mathcal{T}_q$ such that $u = s \cdot t$.

4.1 Final Substitutions

Below we will show how to extract *substitutions* from traces. A (symbolic) substitution is a map $\sigma : \mathcal{X} \to \mathbb{E}$ from variables to expressions over variables. Expressions $e \in \mathbb{E}$ should be interpreted *symbolically*; the denotation $|e|$ will always be made explicit from now on.

The *updated* substitution $\sigma[\mathtt{x} \mapsto e]$ for some $e \in \mathbb{E}$ maps $\mathtt{x} \mapsto e$ and leaves every other variable $\mathtt{y}$ unchanged: $\mathtt{y} \mapsto \sigma\,\mathtt{y}$ for $\mathtt{y} \neq \mathtt{x}$. A substitution σ can be naturally extended to expressions $e \in \mathbb{E}$ by $\sigma\,\mathtt{op}(e_1, \ldots, e_n) := \mathtt{op}(\sigma\,e_1, \ldots, \sigma\,e_m)$. The identity, or *initial* substitution $\{\mathtt{x} \mapsto \mathtt{x}\}_{\mathtt{x} \in \mathcal{X}}$ is denoted σ_0.

Semantically, expression evaluation, as in $|e| : \mathcal{V}^{\mathcal{X}} \to \mathcal{V}$, extends naturally to symbolic substitutions σ. In fact, $|\sigma|$, given a concrete state $v \in \mathcal{V}^{\mathcal{X}}$, provides the evaluations of the expressions associated to the variables by the substitution. That is,

$$|\sigma| : \mathcal{V}^{\mathcal{X}} \to \mathcal{V}^{\mathcal{X}}, \quad v \mapsto (\mathtt{x} \mapsto |\sigma\,\mathtt{x}|(v)) \tag{1}$$

In other words, $|\sigma|$ is a concrete state transformer. Note that we overload the notation $|\cdot|$ here: on the left, it interprets a substitution; on the right, it interprets an expression.

We have $|\sigma_0|(v) = v$ for all $v \in \mathcal{V}^{\mathcal{X}}$, which corresponds to the behavior of `Skip`. Induction over expressions (not unexpectedly) shows that evaluating expressions after the semantic effect of a substitution is denotationally the same as performing the substitution within the expression:

$$(|e| \circ |\sigma|)(v) = |\sigma\,e|(v) \tag{2}$$

for every expression $e \in \mathbb{E}$ and every $v \in \mathcal{V}^{\mathcal{X}}$. This holds in particular for Boolean expressions $b \in \mathbb{BE}$, so that, for $v \in \mathcal{V}^{\mathcal{X}}$, it holds that $|\sigma|(v) \vDash b$ if and only if $v \vDash \sigma\,b$, and so

$$|\sigma\,b| = |\sigma|^{-1}\big[|b|\big] \tag{3}$$

Behaviors of traces are extracted as a symbolic substitution as follows:

Definition 2 (Trace Substitution). *The function* $\mathsf{Sub} : \mathbb{T} \to \mathbb{E}^{\mathcal{X}} \to \mathbb{E}^{\mathcal{X}}$ *is defined inductively over the structure of traces* $t \in \mathbb{T}$ *as follows:*

$$\begin{aligned}
\mathsf{Sub}(\varepsilon, \sigma) &= \sigma \\
\mathsf{Sub}(\mathtt{x}\,\mathtt{:=}\,e \cdot t, \sigma) &= \mathsf{Sub}(t, \sigma[\mathtt{x} \mapsto \sigma\,e]) \\
\mathsf{Sub}(b \cdot t, \sigma) &= \mathsf{Sub}(t, \sigma)
\end{aligned}$$

The substitution of a trace *t, denoted* $\mathsf{Sub}(t)$*, is defined to be* $\mathsf{Sub}(t, \sigma_0)$.

In this definition, the notation Sub is overloaded: a trace and a substitution define a new substitution, but if only a trace is specified, the substitution is taken to be the initial one, σ_0. If $t \in \mathcal{T}_p$ (meaning $(p, \varepsilon) \xrightarrow{*} (\mathtt{Skip}, t)$), the substitution $\mathsf{Sub}(t)$ is called a *final substitution* of p. Final substitutions $\mathsf{Sub}(t)$ of a program p are thus interpreted as functions $|\mathsf{Sub}(t)|$ that transform inputs according to the trace t through p.

Example 4. The final substitution of the trace $t = (\mathtt{x} < 0) \cdot (\mathtt{x} \mathtt{:=} - \mathtt{x}) \in \mathcal{T}_{p_{\text{abs}}}$ is

$$\mathsf{Sub}(t) = \mathsf{Sub}((\mathtt{x} < 0) \cdot (\mathtt{x} \mathtt{:=} - \mathtt{x}), \sigma_0) = \mathsf{Sub}(\mathtt{x} \mathtt{:=} - \mathtt{x}, \sigma_0) = \mathsf{Sub}(\varepsilon, \sigma_0[\mathtt{x} \mapsto -\mathtt{x}])$$

and this is just $\mathtt{x} \mapsto -\mathtt{x}$. Note the distinction in font typesetting between this substitution and the function F_1 in Example 2. The substitution here is really the syntactic object $\mathtt{x} \mapsto -\mathtt{x}$ whose denotation is F_1. This distinction is crucial for understanding symbolic execution from a denotational perspective.

Concatenation of traces is composition of the substitutions:

Lemma 2 (Composition of Substitutions). *For all traces $s, t \in \mathbb{T}$: $|\mathsf{Sub}(s \cdot t)| = |\mathsf{Sub}(t)| \circ |\mathsf{Sub}(s)|$ as functions.*

Proof. We have $\mathsf{Sub}(s \cdot t, \sigma) = \mathsf{Sub}(t, \mathsf{Sub}(s, \sigma))$ by induction on s. Also $|\mathsf{Sub}(s, \sigma)| = |\mathsf{Sub}(s, \sigma_0)| \circ |\sigma|$, where the interesting inductive step is

$$\begin{aligned}
|\mathsf{Sub}(\mathtt{x} \mathtt{:=} e \cdot s, \sigma)| &= |\mathsf{Sub}(s, \sigma[\mathtt{x} \mapsto \sigma\, e])| \\
&\overset{\text{IH}}{=} |\mathsf{Sub}(s)| \circ |\sigma[\mathtt{x} \mapsto \sigma\, e]| \\
&\overset{*}{=} |\mathsf{Sub}(s)| \circ |\mathsf{Sub}(\mathtt{x} \mathtt{:=} e)| \circ |\sigma| \\
&\overset{\text{IH}}{=} |\mathsf{Sub}(s, \mathsf{Sub}(\mathtt{x} \mathtt{:=} e))| \circ |\sigma| \\
&= |\mathsf{Sub}(\mathtt{x} \mathtt{:=} e \cdot s)| \circ |\sigma|
\end{aligned}$$

where, at (*), one uses $|\mathsf{Sub}(\mathtt{x} \mathtt{:=} e)| \circ |\sigma| = |\sigma[\mathtt{x} \mapsto \sigma\, e]|$. Indeed, for $\mathtt{y} \neq \mathtt{x}$ and arbitrary input v, both sides reduce to $|\sigma\, \mathtt{y}|(v)$, and for $\mathtt{y} = \mathtt{x}$, where $\mathtt{x}$ is the variable used in the assignment, the left-hand side reduces to $|e|(|\sigma|(v))$; the right-hand side to $|\sigma\, e|(v)$—these are equal as mentioned (2). Now

$$|\mathsf{Sub}(s \cdot t, \sigma_0)| = |\mathsf{Sub}(t, \mathsf{Sub}(s, \sigma_0))| = |\mathsf{Sub}(t, \sigma_0)| \circ |\mathsf{Sub}(s, \sigma_0)|,$$

which was to be shown.

4.2 Path Conditions

Given a program p and a final substitution $\mathsf{Sub}(t)$ of some trace $t \in \mathcal{T}_p$, how do we know for which inputs p behaves like $|\mathsf{Sub}(t)|$? To answer this question, we extract a precondition from the Boolean assertions in the trace. This precondition is called the *path condition* in symbolic execution, and represents the unique part of the input space that triggers p to behave like $\mathsf{Sub}(t)$. The Boolean conditions have to be taken under appropriate substitutions; this makes their definition somewhat intricate.

Definition 3 (Trace Path Condition). *The function* $\mathsf{PC} : \mathbb{T} \to \mathbb{E}^{\mathcal{X}} \to \mathbb{E}$ *is defined inductively over the structure of traces* $t \in \mathbb{T}$ *as follows:*

$$\begin{aligned} \mathsf{PC}(\varepsilon, \sigma) &= \top \\ \mathsf{PC}(\mathtt{x} \mathbin{:=} e \cdot t, \sigma) &= \mathsf{PC}(t, \sigma[\mathtt{x} \mapsto \sigma\, e]) \\ \mathsf{PC}(b \cdot t, \sigma) &= \sigma\, b \wedge \mathsf{PC}(t, \sigma) \end{aligned}$$

The path condition of a trace t*, denoted* $\mathsf{PC}(t)$*, is defined to be* $\mathsf{PC}(t, \sigma_0)$*.*

The notation PC is again overloaded: if only a trace is provided, the substitution is taken to be the initial one. Interestingly, PC treats assignments in the same way as Sub. Whereas Sub ignores Boolean assertions, PC uses them to generate the Boolean precondition. Being a Boolean expression, the path condition has an interpretation (denoted $|\cdot|$) as a subset of the initial state space $\mathcal{V}^{\mathcal{X}}$.

Example 5. Suppose now that p_{abs} is preceded by an assignment $\mathtt{x} \mathbin{:=} \mathtt{x} + 2$, so let $q_{\text{abs}} = \mathtt{x} \mathbin{:=} \mathtt{x} + 2 \mathbin{⨟} p_{\text{abs}}$. This q_{abs} has a trace $t = (\mathtt{x} \mathbin{:=} \mathtt{x} + 2) \cdot (\mathtt{x} < 0) \cdot (\mathtt{x} \mathbin{:=} -\mathtt{x})$ in $\mathcal{T}_{q_{\text{abs}}}$. Its path condition is

$$\begin{aligned} \mathsf{PC}(t, \sigma_0) &= \mathsf{PC}((\mathtt{x} < 0) \cdot (\mathtt{x} \mathbin{:=} -\mathtt{x}), \sigma_0[\mathtt{x} \mapsto \mathtt{x} + 2]) \\ &= (\mathtt{x} + 2 < 0) \wedge \mathsf{PC}(\mathtt{x} \mathbin{:=} -\mathtt{x}, (\mathtt{x} \mapsto \mathtt{x} + 2)) \end{aligned}$$

and this is $(\mathtt{x} + 2 < 0) \wedge \top$.

Similar to substitutions (Lemma 2), path conditions can be composed (backwards) when traces are sequenced:

Lemma 3 (Backward-Composition of Path Conditions). *For all traces* $s, t \in \mathbb{T}$*:* $|\mathsf{PC}(s \cdot t)| = |\mathsf{PC}(s)| \cap F^{-1}\big[|\mathsf{PC}(t)|\big]$ *where* $F = |\mathsf{Sub}(s)|$*.*

Proof. By induction on s, for every substitution σ, $\mathsf{PC}(s \cdot t, \sigma) \equiv \mathsf{PC}(s, \sigma) \wedge \mathsf{PC}(t, \mathsf{Sub}(s, \sigma))$, where $\equiv$ denotes the equivalence $b \equiv b'$ defined by $|b| = |b'|$. Syntactic equality fails due to extra truth conjuncts in the base case. By induction on t, one also shows that $|\mathsf{PC}(t, \sigma)| = |\sigma|^{-1}\big[|\mathsf{PC}(t)|\big]$, where one crucially uses the fact (3) that $|\sigma\, b| = |\sigma|^{-1}\big[|b|\big]$ for all b. Now

$$|\mathsf{PC}(s \cdot t, \sigma_0)| = |\mathsf{PC}(s, \sigma_0) \wedge \mathsf{PC}(t, \mathsf{Sub}(s, \sigma_0))| = |\mathsf{PC}(s)| \cap |\mathsf{Sub}(s)|^{-1}\big[|\mathsf{PC}(t)|\big]$$

A trace $t \in \mathcal{T}_p$ is *feasible* if $|\mathsf{PC}(t)| \neq \emptyset$.

Theorem 2 (Trace Correspondence). *Let* p *be a program. There is a one-to-one correspondence between feasible traces* $t \in \mathcal{T}_p$ *and pieces* $(F, B) \in \mathbb{F}_p$ *with* $B \neq \emptyset$*.*

Proof (Sketch). The bijection is $\Phi_p : \mathcal{T}_p \to \mathbb{F}_p, t \mapsto (|\mathsf{Sub}(t)|, |\mathsf{PC}(t)|)$. For all p, there are three things to show: well-definedness, surjectivity, and injectivity. Well-definedness here means $(|\mathsf{Sub}(t)|, |\mathsf{PC}(t)|) \in \mathbb{F}_p$ for $t \in \mathcal{T}_p$. These three things are proven by induction on the structure of p.

Unfeasible traces are not considered in the correspondence, because they have no semantic contribution to the program. They are moreover semantically hard to identify, because one is forced to reason about the nature of F. On the other hand, two pieces $(F, B), (F', B')$ for *feasible* traces can easily be distinguished by their path conditions, since they have to be disjoint (Lemma 1).

$$\begin{array}{ll}
(\texttt{if}\ b\ p_1\ p_2, \sigma, \phi) \rightsquigarrow (p_1, \sigma, \phi \wedge \sigma\, b) & (\texttt{x:=}e, \sigma, \phi) \rightsquigarrow (\texttt{Skip}, \sigma[\texttt{x} \mapsto \sigma\, e], \phi) \\
(\texttt{if}\ b\ p_1\ p_2, \sigma, \phi) \rightsquigarrow (p_2, \sigma, \phi \wedge \neg\sigma\, b) & (\texttt{Skip}\ ⨟\ p, \sigma, \phi) \rightsquigarrow (p, \sigma, \phi) \\
(\texttt{while}\ b\ p, \sigma, \phi) \rightsquigarrow (p\ ⨟\ \texttt{while}\ b\ p, \sigma, \phi \wedge \sigma\, b) & \dfrac{(p, \sigma, \phi) \rightsquigarrow (p', \sigma', \phi')}{(p\ ⨟\ q, \sigma, \phi) \rightsquigarrow (p'\ ⨟\ q, \sigma', \phi')} \\
(\texttt{while}\ b\ p, \sigma, \phi) \rightsquigarrow (\texttt{Skip}, \sigma, \phi \wedge \neg\sigma\, b) &
\end{array}$$

Fig. 3: Inductive transition rules for direct symbolic execution

4.3 Direct Symbolic Execution

We have extracted traces from a program and defined the final substitutions and path conditions for them. Instead, we could have extracted these directly, as is done in practice. For the rules of a transition system that does exactly this, see Figure 3. Again, $\overset{*}{\rightsquigarrow}$ denotes the transitive closure of $\rightsquigarrow$. Note that every rule here has a corresponding rule in Figure 2, and a simple analysis will show that both systems produce the same results:

Proposition 1 (Symbolic Execution via Traces). *Let p be a program.*

- *If $(p, \varepsilon) \xrightarrow{*} (p', t)$ then $(p, \sigma_0, \top) \overset{*}{\rightsquigarrow} (p', \mathsf{Sub}(t), \phi)$ where $|\phi| = |\mathsf{PC}(t)|$.*
- *If $(p, \sigma_0, \top) \overset{*}{\rightsquigarrow} (p', \sigma, \phi)$ then there is a trace t such that $(p, \varepsilon) \xrightarrow{*} (p', t)$ with $\mathsf{Sub}(t) = \sigma$ and $|\mathsf{PC}(t)| = |\phi|$.*

This proposition holds in particular for $p' = \texttt{Skip}$, yielding a correspondence between $\mathcal{T}_p$ and pairs of final substitutions and path conditions obtained from direct symbolic execution.

Proof (Sketch). By induction on the length of the transition chains. The inductive step consists of a case analysis of all single-step transitions, which is a straightforward unfolding of definitions.

The following are immediate corollaries of Theorems 1 and 2 and the above proposition.

Corollary 1 (Correctness). *If $(p, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\texttt{Skip}, \sigma, \phi)$ then $f_p(v) = |\sigma|(v)$ for all v such that $v \vDash \phi$.*

Corollary 2 (Completeness). *If $f_p(v)\downarrow$ then there is a symbolic execution $(p, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\texttt{Skip}, \sigma, \phi)$ such that $v \vDash \phi$ which is unique in this property.*

Example 6. Consider the program q_{abs} from Example 5. The program p_{abs} has the two terminating symbolic executions $(\texttt{x} \mapsto -\texttt{x}, \texttt{x} < 0)$ and $(\sigma_0, \texttt{x} \geq 0)$, whose denotations are respectively $(\alpha : x \mapsto -x, \mathbb{Z}_{<0})$ and $(\mathrm{id}_{\mathbb{Z}} : x \mapsto x, \mathbb{Z}_{\geq 0})$. The assignment $\texttt{x:=x} + 2$ has one symbolic execution $(\texttt{x} \mapsto \texttt{x} + 2, \top)$ with denotation $(\beta : x \mapsto x + 2, \mathbb{Z})$. The denotational semantics immediately says that the sequence $q_{\text{abs}} = \texttt{x:=x} + 2\ ⨟\ p_{\text{abs}}$ has two symbolic executions with denotations

$$\begin{aligned}
(\alpha \circ \beta,\ \mathbb{Z} \cap \beta^{-1}[\mathbb{Z}_{<0}]) &= (x \mapsto -(x+2), \mathbb{Z}_{<-2}), \\
(\mathrm{id} \circ \beta,\ \mathbb{Z} \cap \beta^{-1}[\mathbb{Z}_{\geq 0}]) &= (x \mapsto x + 2, \qquad \mathbb{Z}_{\geq -2}),
\end{aligned}$$

This example illustrates a more general potential of denotational semantics for symbolic execution. Indeed, consider again the two symbolic executions

$$(p, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\texttt{Skip}, \sigma_p, \phi_p) \qquad \text{and} \qquad (q, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\texttt{Skip}, \sigma_q, \phi_q)$$

from Section 1. Then, using the denotational semantics of symbolic executions, it follows naturally that $(p \mathbin{⨾} q, \sigma_0, \top) \overset{*}{\rightsquigarrow} (\texttt{Skip}, \sigma, \phi)$ for some (σ, ϕ) with

$$|\sigma| = |\sigma_q| \circ |\sigma_p| \qquad \text{and} \qquad |\phi| = |\phi_p| \cap |\sigma_p|^{-1}[|\phi_q|]$$

That is, σ is denotationally equivalent to σ_p *within* σ_q – formally $\{\texttt{x} \mapsto \sigma_p\,(\sigma_q\,\texttt{x})\}$ – and ϕ is denotationally equivalent to $\phi_p \wedge \sigma_p\,\phi_q$.

5 Extension to Procedure Calls

In this section we extend While with procedure calls. Let $\texttt{P}, \texttt{Q}, \ldots$ range over procedure names and extend the syntax of program statements with

$$p ::= \quad \ldots \quad | \quad \texttt{P(}\vec{e}\texttt{)}$$

Here, $\vec{e}$ denotes a finite list $e_1, \ldots, e_n$ of expressions that are passed as arguments. They are evaluated to a list of values written $|\vec{e}|(v)$, accordingly. For a finite ordered set of variables $\mathcal{U} = \{\texttt{u}_1, \ldots, \texttt{u}_n\}$, write $\mathcal{U} \texttt{:=} \vec{e}$ for the sequence of assignments $\texttt{u}_1 \texttt{:=} e_1 \mathbin{⨾} \ldots \mathbin{⨾} \texttt{u}_n \texttt{:=} e_n$. Its semantics $v \mapsto v[\mathcal{U} \mapsto |\vec{e}|(v)]$ is clear.

It is assumed that procedures are always declared; a *procedure declaration* $\texttt{P} :: p$ binds the procedure name $\texttt{P}$ to the program p. A *structured program* $[\texttt{P} :: p]^*\,p$ is then a list of procedure declarations followed by a single main program statement. For notational simplicity, we assume that the names of declared procedures in a structured program are distinct and let *every* local variable in a procedure be a parameter. Moreover, *one* finite set $\mathcal{U} = \{\texttt{u}_1, \ldots, \texttt{u}_n\}$ of local variables is used for *all* procedures. Then, for a procedure declaration $\texttt{P} :: p$, p contains variables from $\mathcal{X}$ and $\mathcal{U}$. The main function only uses variables in $\mathcal{X}$—the set of *global* variables. $\mathcal{X}$ is disjoint from $\mathcal{U}$. Every procedure call always passes n—the size of $\mathcal{U}$—arguments $\vec{e}$ for the parameters. We use *void* procedures without return values; these can be encoded using a global return variable.

5.1 Concrete Semantics

A parameter k is used to track the recursion depth. Following the terminology of Owens et al. [15], we refer to this k as the *clock*. The clock is only instantiated by the main function and carried accross different procedures, allowing for arbitrarily long chains of nested procedure calls and even mutual recursion.

Let $\mathcal{Y}$ be an infinite set of variables used to substitute the local variables $\mathcal{U}$, and let $\mathcal{Y}$ be disjoint from $\mathcal{X}$. For a procedure call $\texttt{P(}\vec{e}\texttt{)}$ with clock value k, there is always a finite set $\mathcal{Y}_k \subseteq \mathcal{Y}$ of fresh variables available. We substitute

$\mathcal{Y}_k$ for the local variables $\mathcal{U}$ in the body p of a procedure `P` to avoid overwriting local variables in calls from lower depths. This is written $p[\mathcal{U}/\mathcal{Y}_k]$. A state during procedure calls is an evaluation $w \in \mathcal{V}^{\mathcal{X}\cup\mathcal{Y}}$. Write $w = (v, u)$, where $v : \mathcal{X} \to \mathcal{V}$ is the *global* component and $u : \mathcal{Y} \to \mathcal{V}$ contains the *local* states.

The semantics $g_{p,k} : \mathcal{V}^{\mathcal{X}\cup\mathcal{Y}} \rightharpoonup \mathcal{V}^{\mathcal{X}\cup\mathcal{Y}}$ of *procedure statements p with clock value* $k \in \mathbb{N}$ is defined inductively over k and p as:

$$g_{p,k} : w \mapsto \begin{cases} \vdots & \\ g_{q[\mathcal{U}/\mathcal{Y}_{k-1}],k-1}(w[\mathcal{Y}_{k-1} \mapsto |\vec{e}|(w)]) & \text{if } p = \texttt{P(}\vec{e}\texttt{)},\ k > 0, \text{ and } \texttt{P} :: q \\ \text{undefined} & \text{if } p = \texttt{P(}\vec{e}\texttt{)} \text{ and } k = 0 \end{cases}$$

The semantics of all other cases is the same as in the concrete semantics of Section 2, with the exception that the clock value k is passed around; it is never altered except at procedure calls. At a procedure call, the local state is prepared, i.e., w is updated with $\mathcal{Y}_{k-1} \mapsto |\vec{e}|(w)$, and $\mathcal{Y}_{k-1}$ substitutes the set $\mathcal{U}$ of local variables occuring in q, which is the body of the procedure labeled `P`. For this reason, we have either $\texttt{x} \in \mathcal{X}$ or $\texttt{x} \in \mathcal{Y}_k$ for all variables `x` occuring in assignments and Boolean expressions in the definitions at clock value k, since q only contains variables from $\mathcal{X}$ and $\mathcal{U}$. Hence, expressions in a local environment at clock value k are over $\mathcal{X} \cup \mathcal{Y}_k$ and can be evaluated accordingly.

We let the semantics of the main program follow that of Section 2, extended to procedure calls as follows: if $p = \texttt{P(}\vec{e}\texttt{)}$ and $\texttt{P} :: q$ is a declaration then

$$f_p(v) := v', \text{ where } (v', u') = g_{q[\mathcal{U}/\mathcal{Y}_k],k}(v, u_0[\mathcal{Y}_k \mapsto |\vec{e}|(v)]),$$

Here, k is the minimum clock value such that the right-hand side is defined, and $u_0 \in \mathcal{V}^{\mathcal{Y}}$ is some initialized local state; e.g., $u_0(\texttt{y}) = \mathbf{1}$ for all $\texttt{y} \in \mathcal{Y}$. To take the minimum clock value k such that the computation is defined is an approach similar to while loops, where we chose the minimum integer such that the state violated the loop guard. Like before, this integer may not exist.

If a clock value *does* exist, one can choose *any* sufficiently large one:

Lemma 4. *Let p be a statement and $w \in \mathcal{V}^{\mathcal{X}\cup\mathcal{Y}}$.*

- *If $g_{p,k}(w)\uparrow$ then for all $j < k$: $g_{p,j}(w[\mathcal{Y}_j \mapsto \mathcal{Y}_k])\uparrow$.*
- *If $g_{p,k}(w)\downarrow$ then for all $\ell > k$ and $\texttt{x} \in \mathcal{X}$: $g_{p,\ell}(w[\mathcal{Y}_\ell \mapsto \mathcal{Y}_k])(\texttt{x}) = g_{p,k}(w)(\texttt{x})$.*

The proof is by induction over the clock value ℓ, with base case $k + 1$, and induction over p.

5.2 Symbolic Semantics

The denotational semantics for symbolic execution of procedure call statements is presented in Figure 4. The definition follows Section 3 for general statements, with the exception that a parameter k for recursion depth is passed around. For sequencing, recall the notation $\odot$ introduced in Section 3. The while case is an

p	$\mathbb{G}_{p,k}$
`Skip`	$\{(\mathrm{id}_{\mathcal{V}^{\mathcal{X}\cup\mathcal{Y}}}, \mathcal{V}^{\mathcal{X}\cup\mathcal{Y}})\}$
$\texttt{x:=}\,e$	$\{(w \mapsto w[\texttt{x} \mapsto \lvert e\rvert(w)], \mathcal{V}^{\mathcal{X}\cup\mathcal{Y}})\}$
$q \mathbin{;} r$	$\mathbb{G}_{r,k} \odot \mathbb{G}_{q,k}$
$\texttt{if}\ b\ q\ r$	$\{(G, B \cap \lvert b\rvert) : (G,B) \in \mathbb{G}_{q,k}\} \cup \{(G, B \cap \lvert b\rvert^{\complement}) : (G,B) \in \mathbb{G}_{r,k}\}$
$\texttt{while}\ b\ q$	$\bigcup_{m=0}^{\infty}(\Omega_{b,q,k})^m\{(\mathrm{id}_{\mathcal{V}^{\mathcal{X}\cup\mathcal{Y}}}, \lvert b\rvert^{\complement})\}$
$\texttt{P(}\vec{e}\texttt{)}$	$\begin{cases}\mathbb{G}_{q[\mathcal{U}/\mathcal{Y}_{k-1}],k-1} \odot \mathbb{G}_{\mathcal{Y}_{k-1}\texttt{:=}\vec{e},0} & \text{if } k>0 \text{ where } \texttt{P} :: q\\ \{(\mathrm{id}_{\mathcal{V}^{\mathcal{X}\cup\mathcal{Y}}}, \emptyset)\} & \text{if } k=0\end{cases}$

Fig. 4: Denotational semantics for symbolic execution with procedure calls

infinite union of m-fold applications of the operator $\Omega_{b,q,k}$, which is the same as $\Omega_{b,q}$ as introduced in Section 3, but uses $\mathbb{G}_{q,k}$ instead of $\mathbb{F}_q$.

In case $k = 0$, the function becomes undefined at a procedure call. This is reflected in the fact that the piece precondition is set to the emptyset.

The semantics of the main program statement is extended to procedure calls in a way similar to while loops. For a procedure declaration $\texttt{P} :: p$, we define

$$\mathbb{F}_{\texttt{P(}\vec{e}\texttt{)}} := \bigcup_{k=0}^{\infty} \mathbb{G}_{p,k} \odot \mathbb{G}_{\mathcal{Y}_k:=\vec{e},0}$$

Since the resulting global state is independent of the choice of k, Lemma 1 still holds and Definition 1 is still justified.

5.3 Symbolic Execution Traces

To extract symbolic traces we simply add the rule

$$(\texttt{P(}\vec{e}\texttt{)}, t) \longrightarrow (p[\mathcal{U}/\mathcal{Y}_k], t)$$

for declarations $\texttt{P} :: p$, where $\mathcal{Y}_k$ is the fresh set of variables that we may assume to correspond to the k-th recursive call in the denotational semantics of symbolic execution and the concrete semantics. Theorems 1 and 2 still hold for While extended with procedure calls.

6 Related Work

We have drawn inspiration from earlier formal descriptions of symbolic execution [2], where de Boer and Bonsangue proved correctness and completeness of symbolic executions with respect to an operational-style semantics modeling concrete execution. Whereas their proofs work directly by induction on the execution chains, it is interesting to note that correctness and completeness in our setting arise as straightforward corollaries of the correspondence between the denotational and concrete semantics. Moreover, de Boer and Bonsangue used substitutions to define *evaluation after substitution*; we have semanticized this by interpreting substitutions as mathematical functions on the state space $\mathcal{V}^{\mathcal{X}}$. Although crucial to our work, this semantics of substitutions is far from unexpected, as substitutions are syntactic objects describing mathematical functions. However, we feel this fact is easily overlooked when reasoning about symbolic execution. Defining a denotational semantics for symbolic execution amends this.

De Boer and Bonsangue [2] described two ways of obtaining the final symbolic substitutions and path conditions. These are exactly the two methods we described in Section 4. Proving that they are equivalent (Proposition 1) could not be done syntactically: the conjuncts appearing in the corresponding path conditions are different, but equivalent. Having a denotational semantics here was essential for the proof.

Kneuper [12] gives a denotational semantics of symbolic execution based on sets of sequences of symbolic states and a function extending these sequences. Steinhöfel [17, Ch. 3] describes a more general approach based on *concretization* of symbolic states. A similar approach is taken by Porncharoenwase et al. [16] who describe symbolic execution of a Scheme dialect through big-step semantics. Whereas the present work defines symbolic semantics for a language and relates them to concrete semantics, these works describe semantics for the *exploration* of symbolic states.

Owens et al. [15] mechanize what they call a *functional* big-step semantics for a toy language called FOR, which is similar to ours, but has *for* loops instead of *while* loops, and models assignments as side-effects of expressions. Their functional big-step semantics is essentially identical to our concrete semantics, and we have drawn inspiration from their work for our proof mechanizations in Coq, but our work is the first to approach *symbolic execution* from a denotational perspective.

Nakata and Uustalu [14] explored four different trace-based coinductive operational semantics for the While language: big-step and small-step, functional and relational—all of them for *concrete* execution, whereas we include *symbolic*. In the terminology of [14, 15], the present work could have been titled: *Functional Big-Step Semantics for Symbolic Execution*. We deemed "denotational" more appropriate, as the purpose of our work is to elucidate the *denotation* of the syntactic objects generated in symbolic execution, and to enable compositional reasoning; this has historically been the use of denotational semantics in formal methods.

7 Conclusions and Future Work

We have defined a denotational semantics for symbolic execution as a function defined piecewise on a partition of the input space. Each part is the interpretation of the path condition in symbolic execution, and the piecewise definition on this part is the corresponding symbolic substitution, interpreted as a function on the input space. The correspondence between this denotational semantics and a concrete semantics (Theorem 1), which is a simple identity of functions, has correctness (or *coverage*) and completeness (or *precision*) of the symbolic semantics as immediate corollaries, as formulated in Corollaries 1 and 2. Having this denotational semantics allows for compositional reasoning about symbolic executions, which can be particularly unintuitive for the path condition.

These results have been mechanized in the theorem prover Coq. The proofs are all constructive; we have used the *constructive definite description axiom* (consistent but not constructively provable) to assume we can find the minimum integer regarding while loops and recursive procedure calls. This assumption is not surprising or unrealistic, as symbolic execution in practice deals with finite traces only. A reason to consider infinite symbolic executions (which we aim for in future work by using a *stream* semantics) is to allow *arbitrarily long but finite* symbolic executions.

The denotational semantics extends easily to more language constructs, such as procedures (Section 5). Other work [19] illustrates the use of a denotational semantics for proof techniques involving *probabilistic* language constructs such as sampling and observe statements. A denotational semantics for such language constructs are straightforward extensions of the work presented here.

A highly interesting extension of the work in this paper is to incorporate parallelization; compositional correctness and completeness of a small-step symbolic semantics for parallel programs has recently been mechanized in [11]. In a denotational setting, parallelization can be addressed by means of a trace semantics and corresponding coinductive techniques (see, e.g., [18]); furthermore, concurrency is very context-sensitive, which makes assigning a denotational (or functional big-step) semantics challenging. In future work we plan to study a trace-based denotational semantics of symbolic execution, allowing parallelization as well as non-termination. We further consider describing a language-independent approach to symbolic execution using coalgebras. This has previously been studied coinductively [13] (but without coalgebras). Finally, having both an *operational* and a *denotational* semantics, a natural follow-up question is: can we show a correspondence between our denotational semantics and an *axiomatic* semantics for symbolic execution (e.g., in the style of the rules of the KeY verification system [1, Chap. 3])? We believe such a correspondance could be used to enrich verification techniques based on symbolic execution.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
2. de Boer, F.S., Bonsangue, M.: Symbolic execution formally explained. Formal Aspects of Computing **33**(4), 617–636 (2021)

3. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08). pp. 209–224. USENIX Association (2008)
4. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) Proc. 13th ACM Conference on Computer and Communications Security (CCS'06). pp. 322–335. ACM (2006)
5. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proc. 33rd International Conference on Software Engineering (ICSE 2011). pp. 1066–1071. ACM (2011)
6. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
7. Coq Development Team: The Coq proof assistant (Sep 2022). https://doi.org/10.5281/zenodo.7313584
8. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05). pp. 213–223. ACM (2005)
9. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Pasareanu, C.S. (eds.) Proc. 27th International Conference on Computer Aided Verification (CAV 2015). Lecture Notes in Computer Science, vol. 9206, pp. 273–289. Springer (2015)
10. Hentschel, M., Bubel, R., Hähnle, R.: The symbolic execution debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. Int. J. Softw. Tools Technol. Transf. **21**(5), 485–513 (2019)
11. Kløvstad, Å.A.A., Kamburjan, E., Johnsen, E.B.: Compositional correctness and completeness for symbolic partial order reduction. In: Pérez, G.A., Raskin, J. (eds.) Proc. 34th Intl. Conf. on Concurrency Theory (CONCUR 2023). LIPIcs, vol. 279, pp. 9:1–9:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.CONCUR.2023.9
12. Kneuper, R.: Symbolic execution: a semantic approach. Science of computer programming **16**(3), 207–249 (1991)
13. Lucanu, D., Rusu, V., Arusoaie, A.: A generic framework for symbolic execution: A coinductive approach. Journal of Symbolic Computation **80**, 125–163 (2017)
14. Nakata, K., Uustalu, T.: Trace-based coinductive operational semantics for While: big-step and small-step, relational and functional styles. In: Proc. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009). Lecture Notes in Computer Science, vol. 5674, pp. 375–390. Springer (2009)
15. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: Proc. 25th European Symposium on Programming (ESOP 2016). Lecture Notes in Computer Science, vol. 9632, pp. 589–615. Springer (2016)
16. Porncharoenwase, S., Nelson, L., Wang, X., Torlak, E.: A formal foundation for symbolic evaluation with merging. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498709, https://doi.org/10.1145/3498709
17. Steinhöfel, D.: Abstract execution: automatically proving infinitely many programs. Ph.D. thesis, Technische Universität Darmstadt (2020)
18. Uustalu, T.: Coinductive big-step semantics for concurrency. In: Yoshida, N., Vanderbauwhede, W. (eds.) Proc. 6th Workshop on Programming Language Ap-

proaches to Concurrency and Communication-cEntric Software (PLACES 2013). EPTCS, vol. 137, pp. 63–78 (2013)

19. Voogd, E., Johnsen, E.B., Silva, A., Susag, Z.J., Wąsowski, A.: Symbolic semantics for probabilistic programs. In: Jansen, N., Tribastone, M. (eds.) Proc. 20th Intl. Conf. on Quantitative Evaluation of Systems, (QEST 2023). Lecture Notes in Computer Science, vol. 14287, pp. 329–345. Springer (2023). https://doi.org/10.1007/978-3-031-43835-6_23