








# Correct and Complete Symbolic Execution for Free<sup>\*</sup>

Erik Voogd <sup>1</sup> , Einar Broch Johnsen<sup>1</sup> ,  
Åsmund Aqissiaq Arild Kløvstad<sup>1</sup> , Jurriaan Rot<sup>2</sup> , and Alexandra Silva<sup>3</sup> 

<sup>1</sup> University of Oslo, Oslo, Norway

<sup>2</sup> Radboud University, Nijmegen, the Netherlands

<sup>3</sup> Cornell University, Ithaca NY, USA

Corresponding author:  erikvoogd@live.nl

**Abstract.** Symbolic execution is a powerful technique for program analysis. However, the formal semantics underlying symbolic execution is often developed on an ad-hoc basis and decoupled from the concrete semantics of the programming language. To overcome this issue, we introduce *symbolic SOS*: a rule format that allows us to simultaneously specify concrete *and* symbolic operational semantics. We prove that symbolic semantics, when generated from symbolic SOS, is both correct and complete with respect to the corresponding concrete semantics. The approach relies only on an algebraic signature of the source language, and is thus language-independent.

## 1 Introduction

Symbolic execution is an established program analysis technique with a long history [10, 15, 16], that is widely used for bug finding, verification and even program synthesis [1, 3, 11, 19, 21]. More recently, the systematic study of the formal foundations of symbolic execution in its own right has gained increasing interest [2, 9, 18, 24]. In particular, De Boer and Bonsangue [8, 9] introduce the notions of *correctness* and *completeness*, formalizing a correspondence between symbolic execution and concrete program execution.

Proving that a given symbolic semantics is correct and complete, however, is a tedious task that would benefit from automation. This paper provides the basis for such automation by answering the following question: *what are sufficient conditions for a language specification to define a symbolic semantics that is correct and complete by design?* We take language specifications to be defined in Structural Operational Semantics (SOS) [20, 25], a standard formalism to specify programming language semantics as collections of inference rules. Our goal is to extract, for a given language with an operational semantics expressed as an SOS specification, a correct and complete symbolic semantics.

---

<sup>\*</sup> This research is partially supported by the NWO grant No. OCENW.M20.053, ERC grant Autoprobe (no. 101002697) and a Royal Society Wolfson fellowship.

Our starting point is the *GSOS* rule format [5]. The syntactic restrictions on rules in GSOS specifications ensure that the resulting semantics is well-defined and compositional with respect to bisimilarity. The GSOS format was later generalized by Turi and Plotkin into *abstract GSOS* [25], which formalizes specifications through interaction between algebra (representing syntax) and coalgebra (representing transition systems). Goncharov et al. have recently introduced *stateful structural operational semantics* (SSOS) [13], which adapts abstract GSOS to a stateful setting.

In order to automatically obtain symbolic semantics from SOS specifications, we refine SSOS to *symbolic SOS*. A symbolic SOS specification defines both a concrete and a symbolic operational semantics, in terms of two (small-step) transition systems. Our main result is that this symbolic semantics is always correct and complete with respect to the concrete semantics. Any programming language defined in the symbolic SOS format thereby comes equipped with a corresponding correct and complete symbolic semantics. To prove this general correspondence result, we introduce the notion of *syncrete bisimulation*, which gives a sufficient condition for correctness and completeness. We then show that for every symbolic SOS specification, the induced concrete and symbolic semantics are bisimilar, and therefore the symbolic semantics is correct and complete.

*Contributions* In summary, this paper introduces: (i) the *symbolic SOS* rule format (Section 4) from which both a concrete and a symbolic semantics can be derived; (ii) the notion of *syncrete bisimulation*, which provides a coinductive proof technique for correctness and completeness (Section 6); and (iii) a justification for the rule format (Theorem 1) stating that the two derived semantics from a symbolic SOS specification are always syncretely bisimilar.

## 2 Overview

This section summarizes the technical development and presents our key results. We also introduce our running example: the imperative toy language **While**. **While** serves as a minimal concrete example to illustrate our results, but the approach is language-independent. In Section 7, we consider additional common programming constructs, to showcase the power of our approach.

*Example 1 (The Syntax of While)*. The syntax of **While** is given by three grammars for expressions, Boolean expressions and program statements. Let us consider the following grammar for expressions and Boolean expressions:

$$e ::= x \mid n \mid e + e \mid e - e \mid e * e \quad b ::= \top \mid \neg b \mid b \wedge b \mid e < e \mid e = e$$

where  $x \in \mathcal{X}$  ranges over program variables and  $n \in \mathbb{Z}$  over integers. The syntax of the **While** programming language is given by the grammar

$$p ::= \text{Skip} \mid x := e \mid p \ ; \ p \mid \text{if } b \ p \ p \mid \text{while } b \ p$$

These program statements represent inaction, assignments, sequencing, branching, and unbounded iteration.

The semantics of `While` can be specified using an SOS format to define a transition system that formalizes the evolution of program configurations. In particular, the SSOS format [13] considers pairs  $(p, s)$ , where  $p$  is the program to be executed and  $s$  is a state. Usually, states associate values to program variables. Consider, for example, the following rules for sequencing and assignment:

$$\frac{(p, s) \downarrow_c s'}{(p \ ; \ q, s) \longrightarrow_c (q, s')} \qquad \frac{}{(x := e, s) \downarrow_c s[x \mapsto s(e)]}$$

The transition relation  $\longrightarrow_c$  denotes one step of *concrete execution*, evolving to a new program and state, and  $\downarrow_c$  denotes termination. The subscript  $c$  here emphasizes that system states are *concrete*. With  $[x \mapsto s(e)]$ , an assignment updates the value of variable  $x$  to  $s(e)$ , which is the expression  $e$  evaluated in the state  $s$  according to a standard interpretation of arithmetic operations. Each rule actually represents a (potentially infinite) family of rules, one for each state. Together, these two (families of) rules describe a concrete execution model consisting of sequences of assignments to program variables.

For symbolic execution we may also define a *symbolic* semantics, again using the SSOS format. This is defined as a transition relation between states  $\sigma$  which are substitutions, i.e., they associate expressions to variables. The symbolic state moreover contains a set of path constraints which we add later. Symbolic rules are often identical to their concrete counterpart, up to the interpretation of states. Indeed, consider the analogous symbolic rules for sequencing and assignments:

$$\frac{(p, \sigma) \downarrow_s \sigma'}{(p \ ; \ q, \sigma) \longrightarrow_s (q, \sigma')} \qquad \frac{}{(x := e, \sigma) \downarrow_s \sigma[x \mapsto \sigma(e)]}$$

Here, the *symbolic* transition relation  $\longrightarrow_s$  denotes one step of *symbolic execution*, evolving to a new program and symbolic state, and  $\downarrow_s$  denotes termination.

The rules look identical to their concrete counterparts, but the state updates differ subtly. While the concrete rule updates the state to map the variable  $x$  to a new value, namely the expression  $e$  evaluated in state  $s$ , the symbolic rule updates the state to map the variable  $x$  to a new expression:  $\sigma(e)$  is the expression  $e$  with all variables substituted according to  $\sigma$ .

For symbolic execution to be useful, it must indeed be an abstraction of the concrete execution. That is, informally, for each concrete step there must be a symbolic step whose final state describes the state update of the concrete step. For some rules in concrete execution, however, it is unclear what the matching symbolic step should look like. Consider the concrete rules for branching:

$$\frac{s \models b}{(\text{if } b \ p \ q, s) \longrightarrow_c (p, s)} \qquad \frac{s \models \neg b}{(\text{if } b \ p \ q, s) \longrightarrow_c (q, s)}$$

These rules express that the program `if b p q` evolves to  $p$  if  $s$  satisfies  $b$ , and to  $q$  otherwise; the state  $s$  remains unaltered in either case. These rules are deterministic: branching is resolved by checking whether or not states satisfy the Boolean expression  $b$ , guarding the control-flow of the program.

From a *symbolic* state, however, the question of whether the symbolic state satisfies  $b$  cannot be resolved. We may enable *both* transitions

$$\frac{}{(\text{if } b \ p \ q, \sigma) \longrightarrow_s (p, \sigma)} \quad \frac{}{(\text{if } b \ p \ q, \sigma) \longrightarrow_s (q, \sigma)}$$

rendering the symbolic transition relation non-deterministic. Starting from a single state, therefore, a program generates many different sequences of transition steps, called *paths*. In contrast, concrete execution generates a *single* path. So which symbolic execution path simulates the concrete execution path?

To answer this question, states in symbolic execution are enhanced with a *path condition* and the transition relation now relates triples of programs, states, and a path condition. Path conditions aggregate the Boolean conditions that guide a program’s control flow under the current substitution. In the case of conditional branching, this is realized through the following two rules:

$$\frac{}{(\text{if } b \ p \ q, \sigma, \varphi) \longrightarrow_s (p, \sigma, \varphi \wedge \sigma(b))} \quad \frac{}{(\text{if } b \ p \ q, \sigma, \varphi) \longrightarrow_s (q, \sigma, \varphi \wedge \sigma(\neg b))}$$

Both steps augment the path condition by substituting for the variables  $x$  their associated expressions  $\sigma(x)$  in the expressions  $b$  and  $\neg b$ .<sup>4</sup> The resulting system is still technically non-deterministic, but the path condition “determinizes” the symbolic execution by specifying exactly which concrete executions it simulates.

To argue that the path condition of a symbolic execution path is indeed a precondition for concrete executions, the two types of executions are connected by proving *correctness* and *completeness*. We define these notions following De Boer and Bonsangue [9]. Below, the initial configuration  $(\sigma_0, \varphi_0) = (\text{id}, \top)$  consists of the identity substitution  $\sigma_0$  on variables and the Boolean *truth* formula  $\top$ .

**Correctness.** Symbolic execution is *correct* with respect to concrete execution if all *symbolic* execution paths

$$(p_0, \sigma_0, \varphi_0) \longrightarrow_s (p_1, \sigma_1, \varphi_1) \longrightarrow_s \dots \longrightarrow_s (p_k, \sigma_k, \varphi_k) \quad (1)$$

simulate the *concrete* execution paths from  $(p_0, s_0)$  for which  $s_0 \models \varphi_k$ . Formally,

$$(p_0, s_0 \bullet \sigma_0) \longrightarrow_c (p_1, s_0 \bullet \sigma_1) \longrightarrow_c \dots \longrightarrow_c (p_k, s_0 \bullet \sigma_k) \quad (2)$$

is the *unique* concrete execution path starting from  $p_0$  and  $s_0 \models \varphi_k$ . Here,  $s \bullet \sigma$  denotes *evaluation of  $s$  after substitution  $\sigma$* —this is made formal in Definition 5.

**Completeness.** Symbolic execution is *complete* with respect to concrete execution if every concrete execution path

$$(p_0, s_0) \longrightarrow_c (p_1, s_1) \longrightarrow_c \dots \longrightarrow_c (p_k, s_k) \quad (3)$$

<sup>4</sup> We assume that this always results in a Boolean expression; abstracting from program-level type correctness.

is simulated by a symbolic execution path as in Equation (1), whose resulting path conditions are satisfied by  $s_0$ , i.e.,  $s_0 \models \varphi_j$  for all  $j \in [0..k]$ .

Correct- and completeness are properties of symbolic execution that are defined with respect to the concrete semantics: *correctness* is when every symbolic execution path corresponds to a realizable concrete computation, and *completeness* is when all concrete paths are represented by some symbolic path. In the context of program analysis, on the other hand, correctness is about *coverage* and completeness is about *precision* [2, 9].

The process of defining concrete and symbolic semantics separately and then proving correctness and completeness is cumbersome. As observed, the symbolic and concrete rules are (almost) identical, leading to our key question:

**Question:** *Can we obtain correct and complete symbolic semantics directly from a language specification?*

We answer this question by defining a *symbolic* SOS rule format. A key insight for our work is the observation that both concrete and symbolic transition systems can arise from the same underlying specification, if this specification is sufficiently structured to obtain both semantics. In particular, the resulting state needs to be carefully constructed to ensure symbolic simulation, and care must be taken to allow building a path condition.

In symbolic SOS specifications (Section 4), states are meta-variables, i.e., placeholders for both concrete *and* symbolic states. In Section 5 we make explicit how the concrete semantics is derived from a symbolic SOS specification. Then, in Section 6, we derive the symbolic semantics and we define the notion of *syncrete bisimulations*. Crucially, we show that the derived semantics are related by a syncrete bisimulation relation, ensuring both correctness and completeness.

### 3 Preliminaries

For our programming language, we take a signature  $(\Sigma, E, P, B, \sharp, \beta)$  and a fixed set of program variables  $\mathcal{X}$ . The signature consists of ways to generate (i) *programs* using operators in  $\Sigma$ ; (ii) *expressions* using  $E$ -operators; (iii) *predicates* using  $P$ ; and (iv) *Boolean expressions* using  $B$ . Every operator has some arity given by  $\sharp: \Sigma + E + P + B \rightarrow \mathbb{N}$ . The meaning of  $\beta$  is explained below.

For any set  $X$ , write  $\mathcal{E}X$  for the set of terms over  $X$  using operations in  $E$ ; we call these *expressions* over  $X$ . Then  $\mathcal{E}\mathcal{X}$  is the set of *program* expressions, used in, e.g., assignments. With  $\mathcal{V}$  a set of values (integers, rationals, lists, etc.), a family  $\varepsilon$  of maps  $(\varepsilon_{\mathbf{f}}: \mathcal{V}^{\sharp(\mathbf{f})} \rightarrow \mathcal{V})_{\mathbf{f} \in E}$  interprets  $E$ -operators. The inductive extension  $\bar{\varepsilon}$  of  $\varepsilon$  over itself, i.e.,  $\bar{\varepsilon}: \mathcal{E}\mathcal{V} \rightarrow \mathcal{V}$  with  $\bar{\varepsilon}(v) = \varepsilon(v)$ ,  $v \in \mathcal{V}$  and  $\bar{\varepsilon}(\mathbf{f}(e_1, \dots, e_{\sharp(\mathbf{f})})) = \varepsilon_{\mathbf{f}}(\bar{\varepsilon}(e_1), \dots, \bar{\varepsilon}(e_{\sharp(\mathbf{f})}))$  evaluates expressions over values. We will write  $\varepsilon$  for  $\bar{\varepsilon}$ .

*States* during concrete execution (or *concrete states*) are mappings  $s: \mathcal{X} \rightarrow \mathcal{V}$  that assign values to program variables. Given  $\varepsilon: \mathcal{E}\mathcal{V} \rightarrow \mathcal{V}$ , program expressions  $e \in \mathcal{E}\mathcal{X}$  can be evaluated in any state by inductive extension of the map  $s: \mathcal{X} \rightarrow \mathcal{V}$  to  $\bar{s}: \mathcal{E}\mathcal{X} \rightarrow \mathcal{V}$  defined by  $\bar{s}(x) = s(x)$  and  $\bar{s}(\mathbf{f}(e_1, \dots, e_{\sharp(\mathbf{f})})) = \varepsilon_{\mathbf{f}}(\bar{s}(e_1), \dots, \bar{s}(e_{\sharp(\mathbf{f})}))$ . Then  $\bar{s} = \bar{\varepsilon} \circ \mathcal{E}s$ , where  $\mathcal{E}s: \mathcal{E}\mathcal{X} \rightarrow \mathcal{E}\mathcal{V}$  performs uniform

substitution of variables by values in the expression according to  $s$ . Sometimes we write  $s$  for  $\bar{s}$ .

The set  $\mathcal{PX}$  of *predicates* over  $X$  consists of expressions  $\pi(e_1, \dots, e_n)$ , where  $\pi \in P$  is an  $n$ -ary predicate operator, i.e.,  $\sharp(\pi) = n$ , and each  $e_i \in \mathcal{EX}$  is an expression over  $X$ . Common examples include membership, equality, and inequalities. Assume we can *interpret* predicates over values using  $I: \mathcal{PV} \rightarrow \{\top, \text{F}\}$ . We say a concrete state  $s \in \mathcal{V}^{\mathcal{X}}$  *satisfies* a program predicate  $\pi(e_1, \dots, e_n) \in \mathcal{PX}$ , written  $s \models_{\varepsilon, I} \pi(e_1, \dots, e_n)$  iff  $I(\pi(\bar{s}(e_1), \dots, \bar{s}(e_n))) = \top$ . We will leave dependence on  $\varepsilon$  and  $I$  implicit by just writing  $s \models \pi(e_1, \dots, e_n)$ .

With  $B$  as a set of logical operators such that every  $n$ -ary  $\mathcal{L} \in B$  has a truth table  $\{\top, \text{F}\}^n \rightarrow \{\top, \text{F}\}$ ,  $(E, P, B)$  is a first-order logic signature without quantification. Let  $\mathcal{BX}$  be the set of *Boolean expressions* over  $X$  inductively generated by operators in  $B$  over predicates in  $\mathcal{PX}$ . We assume  $B$  contains binary conjunction  $\wedge$  and disjunction  $\vee$ , and unary negation  $\neg$ , each with its conventional truth table. The constant  $\top \in B$  (true) is satisfied by all states, and  $\perp \in B$  (false) by no states. Write  $s \models b$  if a state  $s \in \mathcal{V}^{\mathcal{X}}$  satisfies  $b \in \mathcal{BX}$ .

The set  $\Sigma^*(X)$  of *programs* with free variables  $X$  is inductively generated by operators in  $\Sigma$ . Let  $\mathcal{T} = \Sigma^*(\emptyset)$  be the set of (*closed*) programs. These are the only programs that will be equipped with a semantics. Operators in  $\Sigma$  may use *expressions* in  $\mathcal{EX}$  and *Boolean expressions* in  $\mathcal{BX}$ . Crucial to our work,  $\beta: \Sigma \rightarrow \mathcal{BX}$  assigns to every operator  $\mathbf{f} \in \Sigma$  an associated *guard*  $\beta(\mathbf{f}) \in \mathcal{BX}$  governing control flow of the semantics. For the rest of the paper, we consider the signature  $(\Sigma, E, P, B, \sharp, \beta)$ , the set of variables  $\mathcal{X}$ , and the interpretations  $\varepsilon$  and  $I$  fixed.

*Example 2 (The While Signature).* The signature  $(\Sigma, E, P, B, \sharp, \beta)$  for the **While** language from Example 1 is as follows:  $\Sigma$  contains the constant  $\text{Skip} \in \Sigma$ , a binary operator for sequencing, and one constant for every pair  $(x, e) \in \mathcal{X} \times \mathcal{EX}$  of variable and expression representing an assignment  $x := e$ . Each of these operators has  $\top$  as guard.  $\Sigma$  furthermore contains, for each Boolean expression  $b \in \mathcal{BX}$ , a binary operator **if**  $b \cdot \cdot$  for branching with guard  $\beta(\text{if } b \cdot \cdot) = b$  and a unary operator **while**  $b \cdot$  for unbounded iteration with guard  $\beta(\text{while } b \cdot) = b$ . The set  $E$  contains the binary operators  $+$ ,  $-$ ,  $*$ , and all integers. The set  $P$  contains binary  $<$  and  $=$ , and  $B$  contains constant  $\top$ , unary  $\neg$ , and binary  $\wedge$ . We let  $\mathcal{V} = \mathbb{Z}$  and  $\varepsilon$  and  $I$  are standard; e.g.,  $\varepsilon(6 * 7) = 42$  and  $I(6 * 7 > 0 \vee \perp) = \top$ .

## 4 Symbolic Rule Format

In this section, we introduce the *symbolic SOS* rule format. The format is purely syntactic, to the point that meta-variables are used as placeholders for both programs and states. This allows us to derive both concrete semantics (Section 5) and symbolic semantics (Section 6) from a single specification in the format.

Let  $\mathcal{XVar} = \{\mathbf{x}, \mathbf{y}, \dots\}$  and  $\mathcal{SVar} = \{\mathbf{a}, \mathbf{b}, \dots\}$  be sets of meta-variables that are placeholders for programs and states, respectively. An (uninterpreted) *transition* is either progressive or terminating. A *progressive* transition is an expression

of the form  $(\mathbf{x}, \mathbf{a}) \longrightarrow (\mathbf{y}, \mathbf{b})$  with  $\mathbf{x}, \mathbf{y} \in \text{XVar}$  and  $\mathbf{a}, \mathbf{b} \in \text{SVar}$ . Here,  $\mathbf{x}$  is called the *source* and is said to transition to the *target*  $\mathbf{y}$  with *input*  $\mathbf{a}$  and *output*  $\mathbf{b}$ . A *terminating* transition lacks a target and only produces output, written  $(\mathbf{x}, \mathbf{a}) \downarrow \mathbf{b}$ . Terms  $\mathbf{t} \in \Sigma^*(\text{XVar})$  may be used as sources and targets. Using a special termination symbol  $\checkmark \in \Sigma$ , we let  $(\mathbf{x}, \mathbf{a}) \downarrow \mathbf{b}$  be synonymous to  $(\mathbf{x}, \mathbf{a}) \longrightarrow (\checkmark, \mathbf{b})$ . We use  $\ell$  to denote uninterpreted transitions, progressive or terminating. An SOS rule consists of a set  $\{\ell_i\}_{i=1..n}$ , called the *premises*, together with a *conclusion*  $\ell$ .

**Definition 1 (Symbolic SOS Rule).** A symbolic SOS rule for an operator  $\mathbf{f} \in \Sigma$  of arity  $n = \sharp(\mathbf{f})$  is a rule  $\frac{\ell_1 \quad \dots \quad \ell_n \quad \phi}{\ell}$  where  $\phi \in \mathcal{B}\mathcal{X}$  is a Boolean expression called the trigger of the rule and

- the source of the conclusion  $\ell$  is  $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)$ ;
- the source of the premise  $\ell_i$  (each  $i$ ) is  $\mathbf{x}_i$  and its target (if progressive) is  $\mathbf{y}_i$ ;
- the input of each premise  $\ell_i$  and the conclusion input is  $\mathbf{a}$ ;
- the output of premise  $\ell_i$  is  $\mathbf{b}_i$ ;
- if  $\ell$  is progressive, its target is in  $\Sigma^*(\{\mathbf{x}_1, \dots, \mathbf{x}_n\} \cup \{\mathbf{y}_i \mid \ell_i \text{ is progressive}\})$ ;
- the conclusion output is a map  $\mathcal{X} \rightarrow \mathcal{E}(\{\mathbf{a}, \mathbf{b}_1, \dots, \mathbf{b}_n\} \times \mathcal{X})$ .

This definition is an adaptation of the format of *stateful SOS* [13]; we have replaced states by meta-variables, added extra structure in the conclusion output, and added a Boolean trigger for control-flow. Our specification, defined just below, requires the trigger to be either the guard  $\beta(\mathbf{f})$  of  $\mathbf{f}$ , or its negation. The last item says that the conclusion output can store an expression over  $\text{SVar} \times \mathcal{X}$  for every variable, but it restricts to using meta-variables in  $\text{SVar}$  occurring in the rule. An expression in  $\mathcal{E}(\text{SVar} \times \mathcal{X})$  can be interpreted as a value in  $\mathcal{V}$  once the meta-variables have been replaced by concrete states—this is technically outlined in Section 5—turning the conclusion output into a new concrete state. Replacing the meta-variables by *symbolic* states (defined later), the map  $\mathcal{X} \rightarrow \mathcal{E}(\text{SVar} \times \mathcal{X})$  can be interpreted as a new symbolic state, as outlined in Section 6.

A *symbolic SOS specification* requires each operator to have exactly *two* rules whose triggers are complementary. Letting these triggers coincide with an operator’s guard and its complement, one rule has trigger  $\beta(\mathbf{f})$ , the other  $\neg\beta(\mathbf{f})$ . The behavior of a program may also depend on whether or not any of its subterms terminates (e.g. sequencing). There are therefore two rules for every operator  $\mathbf{f}$  and every set  $W \subseteq \{1, \dots, \sharp(\mathbf{f})\}$  indicating which premises are progressive.

**Definition 2 (Symbolic SOS Specification).** A symbolic SOS specification for a signature  $(\Sigma, E, P, B, \sharp, \beta)$  is a set  $\Xi$  of symbolic SOS rules with the following condition: for every operator  $\mathbf{f} \in \Sigma \setminus \{\checkmark\}$  with  $n = \sharp(\mathbf{f})$  and for every subset  $W \subseteq \{1, \dots, n\}$ , there are exactly two rules  $\mathcal{R}_1, \mathcal{R}_2 \in \Xi$  such that

- the premises  $\{\ell_i\}_{i \in W}$  of both  $\mathcal{R}_1$  and  $\mathcal{R}_2$  are progressive, and
- the premises  $\{\ell_i\}_{i \in [1..n] \setminus W}$  of both  $\mathcal{R}_1$  and  $\mathcal{R}_2$  are terminating.

Moreover, for these rules, the triggers of  $\mathcal{R}_1$  and  $\mathcal{R}_2$  are  $\beta(\mathbf{f})$  and  $\neg\beta(\mathbf{f})$ .

$$\begin{array}{l}
\text{skip} \frac{}{(\text{Skip}, \mathbf{a}) \downarrow \mathbf{a}} \qquad \text{assign} \frac{}{(x := e, \mathbf{a}) \downarrow \mathbf{a}[x \mapsto \mathbf{a}(e)]} \\
\text{seq-0} \frac{(\mathbf{x}, \mathbf{a}) \downarrow \mathbf{a}'}{(\mathbf{x} \ ; \ \mathbf{y}, \mathbf{a}) \longrightarrow (\mathbf{y}, \mathbf{a}')} \qquad \text{seq-n} \frac{(\mathbf{x}, \mathbf{a}) \longrightarrow (\mathbf{x}', \mathbf{a}')}{(\mathbf{x} \ ; \ \mathbf{y}, \mathbf{a}) \longrightarrow (\mathbf{x}' \ ; \ \mathbf{y}, \mathbf{a}')} \\
\text{if-T} \frac{b}{(\text{if } b \ \mathbf{x} \ \mathbf{y}, \mathbf{a}) \longrightarrow (\mathbf{x}, \mathbf{a})} \qquad \text{while-T} \frac{b}{(\text{while } b \ \mathbf{x}, \mathbf{a}) \longrightarrow (\mathbf{x} \ ; \ \text{while } b \ \mathbf{x}, \mathbf{a})} \\
\text{if-F} \frac{\neg b}{(\text{if } b \ \mathbf{x} \ \mathbf{y}, \mathbf{a}) \longrightarrow (\mathbf{y}, \mathbf{a})} \qquad \text{while-F} \frac{\neg b}{(\text{while } b \ \mathbf{x}, \mathbf{a}) \downarrow \mathbf{a}}
\end{array}$$

**Fig. 1.** A symbolic SOS specification for the **While** language, from which both symbolic and concrete semantics can be derived.

*Example 3 (Specification for While).* A symbolic SOS specification for the language system for **While** (from Example 2) is shown in Figure 1. We omit the trigger of a rule if it is  $\top$ . The rule for  $\neg\top$  does not matter for the semantics (see Sections 5 and 6), because no state ever satisfies  $\neg\top$ . The rules for branching, iteration, and sequencing are syntactic sugar for sets of rules. Rule **while-F**, for instance, represents two rules: one with premise  $(\mathbf{x}, \mathbf{a}) \longrightarrow (\mathbf{x}', \mathbf{a}')$  and one with premise  $(\mathbf{x}, \mathbf{a}) \downarrow \mathbf{a}'$ . Premise targets and outputs are not used in these instances.

When writing  $\mathbf{a}$  (or  $\mathbf{a}'$ ) in the conclusion output of a rule, we mean the map  $\mathcal{X} \rightarrow \mathcal{E}(\text{SVar} \times \mathcal{X})$  that sends  $x$  to  $(\mathbf{a}, x)$ . Rule **assign** uses common notation for function *updates*  $\mathbf{a}[x \mapsto \mathbf{a}(e)]$ , i.e., a function that maps every  $y$  to  $(\mathbf{a}, y)$  except  $x$ , which is mapped to  $\mathbf{a}(e)$ . Here,  $\mathbf{a}(e)$  is shorthand for the expression  $e$  with every variable  $y$  substituted by  $(\mathbf{a}, y)$ .

In the following two sections, the meta-variables for states will be substituted by *concrete* states (Section 5) and by *symbolic* states (Section 6). There, the reasons for our choice of shorthand notations for  $\mathbf{a}$  and  $\mathbf{a}(e)$  will be made clear.

## 5 Concrete Semantics

We now show how concrete semantics is derived from a symbolic SOS specification. We make this derivation explicit to juxtapose it with the derivation of the symbolic semantics, and to show how the meta-variables can be formally interpreted as actual states, both concrete and symbolic.

Recall that  $\mathcal{T} = \Sigma^*(\emptyset)$  is the set of all programs. Usually, the meta-variables of language specification rules range over all programs. Formally, the symbolic SOS rules from the previous section are equipped with a *meta-substitution*,<sup>5</sup> i.e., a map  $\psi_X: \text{XVar} \rightarrow \mathcal{T}$ . This mapping canonically extends to  $\Sigma^*(\text{XVar}) \rightarrow \mathcal{T}$ , performing uniform substitution on programs over meta-variables. Usually, a meta-substitution  $\psi_X$  is partially defined, namely on the sources  $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$  and the targets  $\{\mathbf{y}_i \mid \ell_i \text{ is progressive}\}$  of a rule's premises  $\ell_1, \dots, \ell_n$ . The rule format ensures that  $\psi_X$  can also be applied to the conclusion source  $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)$

<sup>5</sup> Not to be confused with substitutions of values or expressions.



and the conclusion target  $\mathbf{t}$ . The meta-variables in the rules are all distinct, but since meta-substitutions can be injective, programs in the rule may coincide.

The key insight here is that states, much like programs, can also be interpreted symbolically using meta-variables:

**Definition 3 (Meta-Substitution of Concrete States).** A meta-substitution of concrete states is a map  $\psi_S: SVar \rightarrow \mathcal{V}^{\mathcal{X}}$ .

We will often combine meta-substitutions of programs  $\psi_X: XVar \rightarrow \mathcal{T}$  and of states  $\psi_S: SVar \rightarrow \mathcal{V}^{\mathcal{X}}$  into one meta-substitution  $\psi: XVar + SVar \rightarrow \mathcal{T} + \mathcal{V}^{\mathcal{X}}$ .

Meta-substitutions of states are usually only partially defined, namely on the input  $\mathbf{a}$ —which is the same for all premises and for the conclusion—and on the premise outputs  $\{\mathbf{b}_1, \dots, \mathbf{b}_n\}$  occurring in a rule. The rule format guarantees that a meta-substitution of states  $\psi_S$  can be applied to the rule's conclusion output  $\mathbf{u}: \mathcal{X} \rightarrow \mathcal{E}(SVar \times \mathcal{X})$ , which would be of type  $\psi_S(\mathbf{u}): \mathcal{X} \rightarrow \mathcal{E}(\mathcal{V}^{\mathcal{X}} \times \mathcal{X})$ . But now that we have access to concrete states, we can use function evaluation  $\text{eval}: \mathcal{V}^{\mathcal{X}} \times \mathcal{X} \rightarrow \mathcal{V}, (s, x) \mapsto s(x)$ , to interpret these pairs occurring in an expression as values from the concrete state. Evaluating the expression of values for each variable using  $\varepsilon: \mathcal{E}\mathcal{V} \rightarrow \mathcal{V}$  provides us with a new state given by  $\psi_S(\mathbf{u}): \mathcal{X} \rightarrow \mathcal{E}(\mathcal{V}^{\mathcal{X}} \times \mathcal{X}) \xrightarrow{\mathcal{E}(\text{eval})} \mathcal{E}(\mathcal{V}) \xrightarrow{\varepsilon} \mathcal{V}$  for the conclusion output.

A *concrete execution model* is a deterministic unlabeled transition system  $(\mathcal{T} \times \mathcal{V}^{\mathcal{X}}, \rightarrow_c)$ . The concrete execution model  $\rightarrow_c$  intended by a symbolic SOS specification  $\Xi$  for the signature  $(\Sigma, E, P, B, \#, \beta)$  is recursively defined on the structure of programs in  $\mathcal{T}$  as follows:

**Definition 4 (Concrete Semantics).** Let  $\mathbf{f} \in \Sigma$  be an operator with  $n = \#\mathbf{f}$ , let  $p_1, \dots, p_n \in \mathcal{T}$  be programs for which transitions have been defined, and let  $s \in \mathcal{V}^{\mathcal{X}}$  be an arbitrary state. Let  $W \subseteq \{1, \dots, n\}$  indicate which transitions  $(p_i, s) \rightarrow_c (p^{(i)}, s^{(i)})$  are progressive, and let  $\mathcal{R}_1, \mathcal{R}_2 \in \Xi$  be the two rules for  $\mathbf{f}$  and  $W$  with triggers  $\beta(\mathbf{f})$  and  $\neg\beta(\mathbf{f})$ , respectively. Then let  $(\mathbf{f}(p_1, \dots, p_n), s) \rightarrow_c (\psi(\mathbf{t}), \psi(\mathbf{u}))$  by definition, with the meta-substitution

$$\psi: XVar + SVar \rightarrow \mathcal{T} + \mathcal{V}^{\mathcal{X}}, \quad \mathbf{x} \mapsto \begin{cases} p_i & \mathbf{x} = \mathbf{x}_i \\ p^{(i)} & \mathbf{x} = \mathbf{x}'_i \end{cases} \quad \mathbf{b} \mapsto \begin{cases} s & \mathbf{b} = \mathbf{a} \\ s^{(i)} & \mathbf{b} = \mathbf{b}_i \end{cases}$$

using conclusion target and output  $(\mathbf{t}, \mathbf{u})$  of  $\mathcal{R}_1$  if  $s \models \beta(\mathbf{f})$  and of  $\mathcal{R}_2$  otherwise.

Constants in  $\Sigma$  use axioms in the specification and constitute the base cases in this recursive definition. The resulting relation is clearly deterministic: every pair  $(p, s)$  defines exactly one outgoing transition, except when  $p = \checkmark$ .

*Example 4.* Consider a program in `While` that computes absolute values:

$$p_{\text{abs}} \triangleq \text{if } (x < 0) \{x := 0 - x\} \{\text{Skip}\}$$

We have a specification from Example 3 which induces a concrete execution model  $(\rightarrow_c, \mathcal{T} \times \mathcal{V}^{\mathcal{X}})$ . Let  $s$  be a concrete state that maps  $x$  to  $-42$ . Then  $(p_{\text{abs}}, s) \rightarrow_c (x := 0 - x, s) \downarrow_c s'$ , where  $s': x \mapsto 42$ . The number 42 was obtained by evaluating the expression  $0 - (\mathbf{a}, x)$  after the meta-substitution  $\psi_S: \mathbf{a} \mapsto s$ .

## 6 Symbolic Semantics

We develop the semantics of symbolic execution, based on the same specification as we used to derive the concrete semantics. The meta-substitution will now substitute meta-variables by *symbolic* states. After describing symbolic states and revisiting meta-substitutions, we introduce *syncrete bisimulation* to inductively formalize correctness and completeness. Symbolic execution semantics is derived from the same specification as the concrete semantics. This semantics is both correct and complete with respect to the concrete semantics (Theorem 1).

The domain of a symbolic state  $\sigma: \mathcal{X} \rightarrow \mathcal{E}\mathcal{X}$ , like that of a concrete state, can be inductively extended from  $\mathcal{X}$  to  $\mathcal{E}\mathcal{X}$ . This extension  $\bar{\sigma}: \mathcal{E}\mathcal{X} \rightarrow \mathcal{E}\mathcal{X}$  is defined by  $\bar{\sigma}(x) = \sigma(x)$  and  $\bar{\sigma}(\mathbf{f}(e_1, \dots, e_n)) = \mathbf{f}(\bar{\sigma}(e_1), \dots, \bar{\sigma}(e_n))$ . Now  $\bar{\sigma} = \mu_{\mathcal{X}} \circ \mathcal{E}\sigma$ , where  $\mathcal{E}\sigma: \mathcal{E}\mathcal{X} \rightarrow \mathcal{E}^2(\mathcal{X})$  performs uniform substitution of variables by expressions, and  $\mu_{\mathcal{X}}: \mathcal{E}^2(\mathcal{X}) \rightarrow \mathcal{E}\mathcal{X}$ —seemingly the identity function—glues expressions together. Contrast this with the inductive extension  $\bar{s}$  of a concrete state  $s: \mathcal{X} \rightarrow \mathcal{V}$  for which  $\bar{s} = \bar{\varepsilon} \circ \mathcal{E}s$ : one evaluates expressions of expressions as a new expression, the other evaluates expressions of values as a value. We sometimes write  $\sigma$  for  $\bar{\sigma}$ .

Symbolic states as substitutions can be applied to predicates in  $\mathcal{P}\mathcal{X}$  by letting  $\sigma(\pi(e_1, \dots, e_n)) := \pi(\sigma(e_1), \dots, \sigma(e_n))$ . Similarly, they can be applied recursively on Boolean expressions with predicates as base cases.

**Definition 5 (Symbolic States as Concrete State Transformers).** *Let  $s \in \mathcal{V}^{\mathcal{X}}$  be a concrete state and  $\sigma \in (\mathcal{E}\mathcal{X})^{\mathcal{X}}$  a symbolic state. Then  $s$  after  $\sigma$  is the new concrete state  $s \bullet \sigma := \bar{s} \circ \sigma: \mathcal{X} \xrightarrow{\sigma} \mathcal{E}\mathcal{X} \xrightarrow{\mathcal{E}s} \mathcal{E}\mathcal{V} \xrightarrow{\varepsilon} \mathcal{V}$ .*

The new state  $s \bullet \sigma$  evaluates an expression  $e$  by inductive extension, but we can also first apply  $\sigma$  and then evaluate expression  $\sigma(e)$  in the initial state  $s$ . These two always coincide: evaluating  $e$  in  $s \bullet \sigma$  is equivalent to evaluating  $\sigma(e)$  in the initial state  $s$ :

**Lemma 1 (Substitution Lemma).** *Let  $s \in \mathcal{V}^{\mathcal{X}}$  and  $\sigma \in (\mathcal{E}\mathcal{X})^{\mathcal{X}}$ . Then (i) for all expressions  $e \in \mathcal{E}\mathcal{X}$ ,  $(s \bullet \sigma)(e) = s(\sigma(e))$ ; and (ii) for all Boolean expressions  $b \in \mathcal{B}\mathcal{X}$ ,  $s \bullet \sigma \models b$  iff  $s \models \sigma(b)$ .*

*Example 5 (Symbolic states as concrete state transformers).* Consider two variables  $x, y$ , a symbolic state  $\sigma = [x \mapsto 2 * x, y \mapsto x]$  and  $s_0 = [x \mapsto 21, y \mapsto 0]$  a concrete state. Then  $(s_0 \bullet \sigma)(x) = \varepsilon(\mathcal{E}s(2 * x)) = \varepsilon(2 * 21) = 42$  and similarly  $(s_0 \bullet \sigma)(y) = 21$ . In general, for this  $\sigma$ , the map  $s \mapsto s \bullet \sigma$  is a concrete state transformer that doubles the value of  $x$  and sets  $y$  equal to the old value of  $x$ .

Symbolic semantics is derived by interpreting meta-variables as symbolic states:

**Definition 6 (Meta-Substitution of Symbolic States).** *A meta-substitution of symbolic states is a map  $\widehat{\psi}_S: SVar \rightarrow (\mathcal{E}\mathcal{X})^{\mathcal{X}}$ .*

The way  $\widehat{\psi}_S$  acts on the conclusion output  $\mathbf{u}: \mathcal{X} \rightarrow \mathcal{E}(SVar \times \mathcal{X})$  of a symbolic SOS rule is analogous to meta-substitution of concrete states:

$$\widehat{\psi}_S(\mathbf{u}): \mathcal{X} \rightarrow \mathcal{E}((\mathcal{E}\mathcal{X})^{\mathcal{X}} \times \mathcal{X}) \xrightarrow{\mathcal{E}(\text{eval})} \mathcal{E}^2(\mathcal{X}) \xrightarrow{\mu_{\mathcal{X}}} \mathcal{E}\mathcal{X}$$

The meta-substitution  $\widehat{\psi}_S$  is first universally applied to  $\mathbf{u}$ , then every pair  $(\sigma, x)$  is evaluated within the expressions, and finally, the resulting expression is glued.

*Example 6 (Meta-substitution for assignment).* Let  $s_0 = [x \mapsto 21, y \mapsto 0]$  and  $\sigma = [x \mapsto 2 * x, y \mapsto x]$ , and let  $s = [x \mapsto 42, y \mapsto 21]$ . In Example 5, we saw that  $\sigma$  transforms  $s_0$  to  $s$ , i.e.,  $s_0 \bullet \sigma = s$ . Suppose  $\psi_S$  and  $\widehat{\psi}_S$  are concrete and symbolic meta-substitutions such that  $\psi_S(\mathbf{a}) = s$  and  $\widehat{\psi}_S(\mathbf{a}) = \sigma$ . Consider an assignment  $x := 0 - x$  as in Example 4 and its transition axiom  $(x := 0 - x, \mathbf{a}) \downarrow \mathbf{u}$  in the specification from Example 3. With  $\mathcal{X} = \{x, y\}$ , we have  $\mathbf{u}: \mathcal{X} \rightarrow \mathcal{E}(\{\mathbf{a}\} \times \mathcal{X})$  such that  $\mathbf{u}: x \mapsto 0 - \mathbf{a}(x)$  and  $\mathbf{u}: y \mapsto \mathbf{a}(y)$ . Putting  $s' := \psi_S(\mathbf{u})$  and  $\sigma' := \widehat{\psi}_S(\mathbf{u})$ , we have  $s'(x) = 0 - s(x) = -42$  and  $\sigma'(x) = 0 - \sigma(x) = 0 - 2 * x$ . For  $y$ ,  $s'(y) = s(y) = 21$  and  $\sigma'(y) = \sigma(y) = x$ . Therefore,  $s_0 \bullet \sigma' = s'$ .

In this example, the concrete and symbolic states are transformed concertedly by the assignment update. Specifically,  $s_0 \bullet \widehat{\psi}_S(\mathbf{u}) = \psi_S(\mathbf{u})$  follows from  $s_0 \bullet \widehat{\psi}_S(\mathbf{a}) = \psi_S(\mathbf{a})$  because  $\mathbf{a}$  is the only meta-variable occurring in  $\mathbf{u}$ . Thus, the example illustrates a general inductive property of our rule format: at every step, the change in symbolic state matches the change in concrete state.

**Lemma 2 (Meta-Substitution Lemma).** *Let  $\psi_S: SVar \rightarrow \mathcal{V}^{\mathcal{X}}$  be a concrete and  $\widehat{\psi}_S: SVar \rightarrow (\mathcal{E}\mathcal{X})^{\mathcal{X}}$  a symbolic meta-substitution and let  $s_0 \in \mathcal{V}^{\mathcal{X}}$  be a concrete state. For all  $\mathbf{u}: \mathcal{X} \rightarrow \mathcal{E}(SVar \times \mathcal{X})$ , if  $s_0 \bullet \widehat{\psi}_S(\mathbf{b}) = \psi_S(\mathbf{b})$  for all  $\mathbf{b} \in SVar$  that occur in  $\mathbf{u}$ , then  $s_0 \bullet \widehat{\psi}_S(\mathbf{u}) = \psi_S(\mathbf{u})$ .*

A *symbolic execution model* is a nondeterministic unlabeled transition system  $(\mathcal{T} \times (\mathcal{E}\mathcal{X})^{\mathcal{X}} \times \mathcal{B}\mathcal{X}, \longrightarrow_s)$ . We now define the symbolic execution model  $\longrightarrow_s$  intended by a symbolic SOS specification  $\Xi$  for the signature  $(\Sigma, E, P, B, \sharp, \beta)$ . For this, we let  $n = \sharp(\mathbf{f})$  and, given arbitrary state  $\sigma$  and path condition  $\varphi$ , we recursively define the set of outgoing transitions for  $(\mathbf{f}(p_1, \dots, p_n), \sigma, \varphi)$ , where we have already defined the sets  $\mathbb{P}_i = \{(p', \sigma', \varphi') \mid (p_i, \sigma, \varphi) \longrightarrow_s (p', \sigma', \varphi')\}$  of outgoing transitions for the subterms  $p_i$ . An  $n$ -tuple  $\xi \in \prod_{i \in [1..n]} \mathbb{P}_i$  contains one possible combination of targets for  $p_1, \dots, p_n$ . Let  $W_\xi \subseteq \{1, \dots, n\}$  be the set of  $i$  with  $p_\xi^{(i)} \neq \checkmark$ , indicating progressive premises; write  $\xi = (p_\xi^{(i)}, \sigma_\xi^{(i)}, \varphi_\xi^{(i)})_{i \in [1..n]}$ .

**Definition 7 (Symbolic Semantics).** *Let  $\mathbf{f} \in \Sigma$  be an operator,  $\sigma \in (\mathcal{E}\mathcal{X})^{\mathcal{X}}$  a symbolic state,  $\varphi \in \mathcal{B}\mathcal{X}$  a path condition, and  $p_1, \dots, p_n \in \mathcal{T}$  a set of programs. For  $\xi \in \prod_{i \in [1..n]} \mathbb{P}_i$ , let  $\mathcal{R}_{\xi,1} \in \Xi$  and  $\mathcal{R}_{\xi,2} \in \Xi$  be the rules for  $\mathbf{f}$  and  $W_\xi$  with triggers  $b_\mathbf{f}$  and  $\neg b_\mathbf{f}$ , respectively. Let  $(\mathbf{f}(p_1, \dots, p_n), \sigma, \varphi) \longrightarrow_s (p', \sigma', \varphi')$ , by definition, for all  $(p', \sigma', \varphi')$  in the set*

$$\left\{ (\widehat{\psi}_\xi(\mathbf{t}_{\xi,1}), \widehat{\psi}_\xi(\mathbf{u}_{\xi,1}), \varphi \wedge \sigma(b) \wedge \Phi_\xi), (\widehat{\psi}_\xi(\mathbf{t}_{\xi,2}), \widehat{\psi}_\xi(\mathbf{u}_{\xi,2}), \varphi \wedge \neg \sigma(b) \wedge \Phi_\xi) \right\}_{\xi \in \prod_i \mathbb{P}_i}$$

where  $\mathbf{t}_{\xi,j}, \mathbf{u}_{\xi,j}$  ( $j = 1, 2$ ) are the conclusion targets and outputs of  $\mathcal{R}_{\xi,j}$ ,

$$\widehat{\psi}_\xi: XVar + SVar \rightarrow \mathcal{T} + (\mathcal{E}\mathcal{X})^{\mathcal{X}}, \quad \mathbf{x} \mapsto \begin{cases} p_i & \text{if } \mathbf{x} = \mathbf{x}_i \\ p_\xi^{(i)} & \text{if } \mathbf{x} = \mathbf{x}'_i \end{cases} \quad \mathbf{b} \mapsto \begin{cases} \sigma & \text{if } \mathbf{b} = \mathbf{a} \\ \sigma_\xi^{(i)} & \text{if } \mathbf{b} = \mathbf{b}_i \end{cases}$$

and  $\Phi_\xi := \bigwedge_{i \in [1..n]} \varphi_\xi^{(i)}$  is the conjunction of all path conditions in the premises.

Here,  $\Phi_\xi$  includes *all* premise path conditions, including conditions potentially not used in the conclusion. This condition may appear too strong for some rule instances in symbolic execution techniques. However, since  $\prod_{i \in [1..n]} \mathbb{P}_i$  comprises all combinations of transitions, and since every  $\xi$  induces a step for both  $\sigma(b)$  and  $\neg\sigma(b)$ , the resulting set of path conditions covers all of the input path condition  $\varphi$ . Many of the resulting steps may have coinciding continuations.

**Proposition 1 (Path Condition One-Step Coverage).** *Let  $\longrightarrow_s$  be the intended symbolic execution system of a symbolic SOS specification, and let  $(p, \sigma, \varphi)$  be a symbolic configuration. For  $A = \{\varphi' \mid (p, \sigma, \varphi) \longrightarrow_s (p', \sigma', \varphi')\}$ :*

- $\varphi_1 \wedge \varphi_2 \equiv \perp$  for all  $\varphi_1, \varphi_2 \in A$  such that  $\varphi_1 \neq \varphi_2$ ; and
- $\bigvee A \equiv \varphi$ , where  $\bigvee A$  denotes finite disjunction of all elements in  $A$ .

*Example 7.* We return to program  $p_{\text{abs}}$  from Example 4 and the symbolic SOS specification from Example 3. Using the two axioms for an *if* statement, the derived symbolic execution semantics gives the two transitions  $(p_{\text{abs}}, \text{id}, \top) \longrightarrow_s (x := -x, \text{id}, \top \wedge (x < 0))$  and  $(p_{\text{abs}}, \text{id}, \top) \longrightarrow_s (\text{Skip}, \text{id}, \top \wedge \neg(x < 0))$ . Continuing for one more step we obtain a set of four reachable configurations; the two on the left stem from  $x := -x$ ; the other two from **Skip**:

$$\left\{ \begin{array}{ll} (\checkmark, x \mapsto -x, \top \wedge (x < 0) \wedge \top), & (\checkmark, \text{id}, \top \wedge \neg(x < 0) \wedge \top), \\ (\checkmark, x \mapsto -x, \top \wedge (x < 0) \wedge \neg\top), & (\checkmark, \text{id}, \top \wedge \neg(x < 0) \wedge \neg\top) \end{array} \right\}$$

The bottom configurations can be discarded: no state satisfies  $\neg\top$ . We further simplify path conditions by removing  $\top$ -conjuncts. Then  $\longrightarrow_s$  reduces the *if* statement to two possible transformations:  $x \mapsto -x$  if  $x < 0$  and **id** otherwise.

*Syncrete bisimulation* is a coinductive formalization of correctness and completeness for symbolic execution. We prove that our rule format induces a syncrete bisimulation relation between concrete and symbolic execution semantics, namely the identity relation: every program is syncretely bisimilar to itself.

**Definition 8 (Syncrete Bisimulation).** *Let  $(\mathcal{T} \times \mathcal{V}^\mathcal{X}, \longrightarrow_c)$  be a concrete execution model and  $(\mathcal{T} \times (\mathcal{E}\mathcal{X})^\mathcal{X} \times \mathcal{B}\mathcal{X}, \longrightarrow_s)$  a symbolic execution model. A relation  $R \subseteq \mathcal{T} \times \mathcal{T}$  is a syncrete bisimulation between  $\longrightarrow_c$  and  $\longrightarrow_s$  if, for all  $\sigma \in (\mathcal{E}\mathcal{X})^\mathcal{X}$ ,  $\varphi \in \mathcal{B}\mathcal{X}$ , and initial states  $s_0 \in \mathcal{V}^\mathcal{X}$  s.t.  $s_0 \models \varphi$ , whenever  $pRq$ :*

- if  $(p, s_0 \bullet \sigma) \longrightarrow_c (p', s')$  then there is  $(q', \sigma', \varphi')$  such that
  - (i)  $(q, \sigma, \varphi) \longrightarrow_s (q', \sigma', \varphi')$  (ii)  $s' = s_0 \bullet \sigma'$  (iii)  $p'Rq'$  and (iv)  $s_0 \models \varphi'$ .
- if  $(q, \sigma, \varphi) \longrightarrow_s (q', \sigma', \varphi')$  and  $s_0 \models \varphi'$  then  $(p, s_0 \bullet \sigma) \longrightarrow_c (p', s_0 \bullet \sigma')$  for some  $p' \in \mathcal{T}$  such that  $p'Rq'$ .

The first item makes every step in the symbolic system coinductively complete: every concrete step is matched by a symbolic step that refines the path condition in a way that the initial state  $s_0$  remains satisfied. The second item makes every step in the symbolic system coinductively correct. Here, it may seem like any symbolic state  $\sigma$  can be chosen, but the updated path condition  $\varphi'$  always

contains the Boolean formula  $b$  that guards control-flow under substitution by  $\sigma$ . Hence, the condition  $s_0 \models \varphi'$  entails  $s_0 \bullet \sigma \models b$ .

In the following results, let  $\Xi$  be a symbolic SOS specification and let  $\longrightarrow_c$  be the intended concrete model and  $\longrightarrow_s$  the intended symbolic model.

**Theorem 1.** *The identity relation on the set  $\mathcal{T}$  of programs is a syncretic bisimulation between  $\longrightarrow_c$  and  $\longrightarrow_s$ .*

By induction on the length of the transition chain, with the definitions of correctness and completeness from Section 2:

**Corollary 1 (Correctness and Completeness).** *The intended model of symbolic execution  $\longrightarrow_s$  is correct and complete with respect to  $\longrightarrow_c$ .*

The induced small-step model provides a correct and complete core for symbolic execution. Full symbolic execution amounts to providing a search strategy for the execution tree built by  $\longrightarrow_s$ , and the result is guaranteed to correspond to concrete program behavior by Corollary 1.

## 7 Extensions

We consider two extensions to **While**: *arrays* (see De Boer and Bonsangue [9]) and a *probabilistic* programming constructs (see Voogd et al. [26]). We immediately obtain a concrete execution model  $(\mathcal{T} \times \mathcal{V}^{\mathcal{X}}, \longrightarrow_c)$  and a symbolic execution model  $\longrightarrow_s$  that is both correct and complete with respect to  $\longrightarrow_c$ .

*Arrays.* Arrays can be incorporated in **While** by imposing some structure on  $\mathcal{X}$ ,  $E$  and  $B$ . Let the *variables* be a disjoint union of regular and array variables  $\mathcal{X} + \mathcal{A}$  with values in  $\mathbb{Z} + (\mathbb{N} \rightarrow \mathbb{Z})$ . Regular variables  $x \in \mathcal{X}$  are assigned integers and  $a \in \mathcal{A}$  partial integer-valued functions with index domain  $\mathbb{N}$ .

Let *expressions* include  $a[e]$ ,  $a[e := e']$  and  $|a|$  for  $a \in \mathcal{A}$  and  $e, e' \in \mathcal{E}\mathcal{X}$ . The semantics is modeled by  $\varepsilon$ ; we let  $\varepsilon$  map  $a[e]$  to  $\varepsilon(a)(\varepsilon(e))$ , and  $a[e := e'](e'')$  to  $\varepsilon(e')$  if  $\varepsilon(e) = \varepsilon(e'')$  or  $\varepsilon(a[e''])$  otherwise. We let  $\varepsilon(|a|)$  be the size of the set on which  $\varepsilon(a)$  is defined. Now extend  $\Sigma$  from Example 1 by allowing the left-hand side of an assignment to be an array expression and introduce a new constant **Error** to denote out-of-bounds access.

Let  $\delta \in P$  be a unary predicate indicating absence of indexing errors. The semantics of the closed predicate  $\delta(e)$  is inductively defined by (i)  $\delta(n) := \top$  for all constants  $n \in E$ ; (ii)  $\delta(a[e])$  iff  $(0 \leq e < |a|) \wedge \delta(e)$ ; (iii)  $\delta(a[e := e'])$  iff  $(0 \leq e < |a|) \wedge \delta(e) \wedge \delta(e')$ ; and (iv)  $\delta(\mathbf{f}(e_1, \dots, e_n)) \equiv \delta(e_1) \wedge \dots \wedge \delta(e_n)$  for  $n$ -ary operation symbols  $\mathbf{f} \in E$ .

Now let us define a symbolic SOS specification for signature  $(\Sigma, E, P, B, \sharp, \beta)$ . Array assignments require a new pair of rules:

$$\frac{\delta(a[e])}{(a[e] := e', s) \downarrow s[a \mapsto a[s(e) := s(e')]]} \qquad \frac{\neg\delta(a[e])}{(a[e] := e', s) \longrightarrow (\mathbf{Error}, s)}$$

The conclusion of the left rule denotes the map  $\mathcal{X} + \mathcal{A} \rightarrow \mathcal{E}(\{\mathbf{s}\} \times (\mathcal{X} + \mathcal{A}))$  that maps each variable  $x$  to  $s(x)$  (including arrays) except that  $a$  is mapped to the expression  $a[s(e) := s(e')]$  with  $e, e'$  updated to replace each variable  $y \in \mathcal{X} + \mathcal{A}$  with  $s(y)$ . The rule on the right signals an error that may be handled by other mechanisms.

The rules in Figure 1 with trivial triggers can be safely replaced with two rules: one with the additional trigger  $\delta(e)$  for expressions in the program term and one with  $-\delta(e)$  progressing to an error. For if- and while- rules some additional machinery is needed to maintain the requirements of Definition 2. For each Boolean expression  $b$  we introduce a new binary operator **sif**  $b \cdot \cdot$  (for “safe if”). We then have exactly two rules for each of **if**  $b$  and **sif**  $b$ .

$$\begin{array}{cc} \text{if-safe} \frac{\delta(b)}{(\text{if } b \text{ x y, a}) \rightarrow (\text{sif } b \text{ x y, a})} & \text{if-err} \frac{-\delta(b)}{(\text{if } b \text{ x y, a}) \rightarrow (\text{Error, a})} \\ \text{if-T} \frac{b}{(\text{sif } b \text{ x y, a}) \rightarrow (\text{x, a})} & \text{if-F} \frac{-b}{(\text{sif } b \text{ x y, a}) \rightarrow (\text{y, a})} \end{array}$$

Thus an if-statement first checks if its condition contains a nil error, and only then proceeds safely to one of its branches. A “safe while” is implemented analogously by first checking its condition, and then proceeding as before.

*Randomization.* For probabilistic sampling during program execution, we consider a set of logical variables  $\mathcal{Y} = \{y_k\}_{k \in \mathbb{N}}$  that represent samples; states are now maps  $\mathcal{X} + \mathcal{Y} \rightarrow \mathcal{V}$ . We consider a signature  $(\Sigma, E, P, B, \#, \beta)$  similar to the While language (see Example 2). To ensure probabilistic independence of samples, assignments  $x := e$  cannot involve variables from  $\mathcal{Y}$ ; they are still represented by a pair  $(x, e) \in \mathcal{X} \times \mathcal{E}\mathcal{X}$ , but  $\Sigma$  is extended with a sampling statement  $x \sim \text{rnd}$ . Consider the rule for sampling (with guard  $\top$ ):

$$\overline{x \sim \text{rnd, a} \downarrow \{x \mapsto (\mathbf{a}, y_0), y_0 \mapsto (\mathbf{a}, y_1), y_1 \mapsto (\mathbf{a}, y_2), \dots\}}$$

which stores the first available sample  $y_0$  in  $x$  and shifts all other samples one position. Writing  $(s, \rho)$  for the state  $\mathcal{X} + \mathcal{Y} \rightarrow \mathcal{V}$ , the concrete rule is

$$\overline{x \sim \text{rnd, s, } \rho \downarrow (s[x \leftarrow \rho_0], \text{tl}(\rho))}$$

Taking the head and tail does not work in the symbolic counterpart of this rule. A solution is to introduce a sampling index  $k$  and using the rule

$$\overline{x \sim \text{rnd, } \sigma, k \downarrow \sigma[x \mapsto y_k], k + 1}$$

An indexing scheme like this must be proven correct in the presence of all the rules in the programming system. The symbolic model of this language system produces symbolic states  $\sigma : \mathcal{X} + \mathcal{Y} \rightarrow \mathcal{E}(\mathcal{X} + \mathcal{Y})$ . Keeping  $k$  constant in other rules ensures that the part  $\mathcal{Y} \rightarrow \mathcal{E}(\mathcal{X} + \mathcal{Y})$  left-shifts the stream by  $k$ , always giving a new variable in  $\mathcal{Y}$ .

For this to be a true randomization of programs, one assumes that the values for  $\mathcal{Y}$ , given by the map  $\rho : \mathcal{Y} \rightarrow \mathcal{V}$ , adhere to some probability law. This is a modeling issue; we argue that the symbolic system produces symbolic states that accurately represent program behavior in that they produce the same result once this initial state (randomized or not) is evaluated by the symbolic state. For a detailed account of symbolic execution of probabilistic programs, see [26].

## 8 Related Work

De Boer and Bonsangue formalize symbolic execution for the `While` language and define the notions of correctness and completeness of symbolic execution [9]. They employ a small-step transition system and inductive proofs of correctness and completeness over its transitive closure. In contrast, our work offers a coinductive alternative (allowing for non-finite computations) that captures both correctness and completeness in terms of syncrete bisimulation. We do this by quantifying over conceptual initial states  $s_0$ , and making concrete small-steps on  $s_0 \bullet \sigma$  rather than big-steps on  $s_0$  itself. A symbolic reconfiguration from  $\sigma$  to  $\sigma'$  then corresponds to a concrete reconfiguration of  $s_0 \bullet \sigma$  to  $s_0 \bullet \sigma'$ .

Goncharov et al. developed *stateful SOS* (SSOS) for stateful programs [13], extending GSOS [5] with *state*, focusing crucially on the compositionality of the derived semantics. Via a reduction to GSOS, SSOS specifications are shown to correspond precisely to natural transformations which induce a denotational behavior that ensures compositionality in *resumption* semantics, a very fine-grained semantics in which very few programs are considered equivalent [13]. In particular, programs that induce the same state transformation — like  $x := 1 \ ; \ x := x + 1$  and  $x := 1 \ ; \ x := x * 2$  — may not be equivalent under resumption semantics. Goncharov et al. consider two coarser semantics — trace and termination — which fail to be compositional in general. They therefore propose further restrictions on the SSOS format to ensure compositionality also in these settings. The unrestricted SSOS format forms the basis for symbolic SOS in our work, but we refine state transformations to ensure that they can be symbolically simulated.

The  $\mathbb{K}$  framework shares our goal of defining language semantics with correct and complete analysis tools [22]. In particular, Lucanu et al. develop a language-independent coinductive description of symbolic execution [2, 18], based on *Reachability Logic* [23]. They use matching logic and reachability logic to define semantics as rewrite rules, whereas we provide syntactic restrictions on the common stateful SOS format and introduce syncrete bisimulation as a useful formalization of the correspondence between symbolic and concrete (small-step) semantics. As a consequence, our proofs mostly use structural induction over programs whereas their proofs use correspondences between proof trees.

Bodin et al. propose another language-independent framework that provides analysis tools “for free” with their pretty-big-step semantics [7] and Skeletal Semantics [6]. They provide a framework of simple building blocks (bones) that assemble into programs (skeletons). Skeletons are given interpretations, and generic consistency results between interpretations are established. Finally, they define

concrete and abstract interpretations and instantiate the consistency results with language-dependent lemmas. Their approach differs from ours by focusing on *structural* building blocks of semantics rather than a rule format. Additionally they focus on abstract interpretation rather than symbolic execution.

## 9 Discussion

We briefly consider two interesting aspects of the presented work: (1) the conditions on the rule formats to simultaneously construct concrete and symbolic semantics, and (2) extensions to more low-level state representations.

Our symbolic SOS format provides a *sufficient* condition for both concrete and symbolic semantics to be constructed simultaneously. However, identifying *necessary* conditions for rule formats that ensure correctness and completeness would be very challenging, because a lot of design choices have to be made to bridge the gap between the desired properties and the semantics specification. Our work on symbolic SOS is based on the following important design choices:

- Symbolic SOS builds on GSOS, which ensures that bisimilarity is always a congruence and that canonical operational models exist. GSOS provides a sufficient condition for this property. GSOS seems fairly close to the limits of well-behaved SOS formats. (For more liberal formats such as `ntyft/ntyxt` which allows both look-ahead and negative premises [14], the interpretation is more subtle [12] and it can even be difficult to decide whether a (unique) model exists [17].)
- Symbolic SOS builds on stateful SOS, which ensures that the properties above hold in a stateful setting. Symbolic SOS imposes structure such that states can be interpreted both concretely and symbolically. Lemma 2 (meta-substitution) proves that every step in one system is simulated by a step in the other. Rules in a specification must come in pairs — generalizable to complementary tuples of arbitrary size — with mutually disjoint conditions.

For our techniques to apply to a language, its operational semantics must be expressible with rules that syntactically enforce these properties: GSOS-like restrictions for compositionality and our added requirement on state structure.

The sets of variables and values, and the signatures of expressions and Boolean expressions have intentionally been kept abstract. Section 7 shows how symbolic execution correctness and completeness is maintained with additional structuring of the signatures. We believe that other extensions, e.g., for pointers or aliasing, would work similarly. Heaps could then be implemented by imposing structure on the states (both concrete and symbolic) similar to the arrays of Section 7. By distinguishing non-heap and heap variables and evaluating heap variables in partial maps (like the array variables), a pointer map is emulated. A predicate similar to the absence of indexing errors can be used to detect null pointer exceptions. See [9] for a discussion on aliasing. In this context, it would be interesting to further extend the path conditions with separation logic for pointers [4]. This would not affect the results presented in this paper, provided the



meta-substitution lemma (Lemma 2) is maintained, ensuring that the symbolic SOS rules define matching transitions for the concrete and symbolic semantics.

## 10 Conclusion

We present a language-independent rule format for program semantics that induces both concrete and symbolic models. The induced models enjoy a bisimilarity relationship that ensures correctness and completeness of symbolic execution. Our approach thus allows to define operational semantics for a language and immediately obtain a symbolic execution engine that is correct by construction, providing a formal basis for analysis and verification tools.

Technically, we exploit that symbolic states represent transformations of concrete states to augment stateful SOS with a more structured notion of state, thereby obtaining *symbolic* stateful SOS. From symbolic SOS specifications, we show how to derive execution models in terms of symbolic and concrete transition systems. We formulate the novel notion of *syncrete bisimulation* and use it to prove that the derived symbolic execution model is correct and complete. The proof makes crucial use of the notion of bisimulation and a very general substitution lemma that relates symbolic states and concrete state transformations.

Our results work for concrete semantics that can be understood as *deterministic* state transformers. An interesting direction of development would be to generalize this to nondeterministic settings, such as concurrent programs. This would require a deeper investigation of the natural transformations induced by the symbolic SOS rule format and their categorical semantics. Goncharov et al. [13] also highlight this direction of research for the non-symbolic case.

*Acknowledgements* The authors would like to thank the anonymous reviewers for their insightful questions and feedback.

## 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. Arusoaie, A., Lucanu, D., Rusu, V.: A generic framework for symbolic execution. In: Erwig, M., Paige, R.F., Wyk, E.V. (eds.) Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8225, pp. 281–301. Springer (2013). [https://doi.org/10.1007/978-3-319-02654-1\\_16](https://doi.org/10.1007/978-3-319-02654-1_16), [https://doi.org/10.1007/978-3-319-02654-1\\_16](https://doi.org/10.1007/978-3-319-02654-1_16)
3. Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)
4. Berdine, J., Calcagno, C., O’Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) Proc. Third Asian Symposium on Programming Languages and Systems (APLAS 2005). Lecture Notes in Computer Science, vol. 3780, pp. 52–68. Springer (2005), [https://doi.org/10.1007/11575467\\_5](https://doi.org/10.1007/11575467_5)

5. Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. *J. ACM* **42**(1), 232–268 (Jan 1995). <https://doi.org/10.1145/200836.200876>
6. Bodin, M., Gardner, P., Jensen, T., Schmitt, A.: Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.* **3**(POPL) (jan 2019). <https://doi.org/10.1145/3290357>
7. Bodin, M., Jensen, T., Schmitt, A.: Certified abstract interpretation with pretty-big-step semantics. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*. p. 29–40. *CPP '15*, Association for Computing Machinery (2015). <https://doi.org/10.1145/2676724.2693174>
8. de Boer, F.S., Bonsangue, M.: On the nature of symbolic execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) *Formal Methods – The Next 30 Years*. pp. 64–80. Springer International Publishing, Cham (2019)
9. de Boer, F.S., Bonsangue, M.: Symbolic execution formally explained. *Formal Aspects of Computing* **33**(4), 617–636 (2021)
10. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT - a formal system for testing and debugging programs by symbolic execution. In: Shooman, M.L., Yeh, R.T. (eds.) *Proc. International Conference on Reliable Software 1975*. pp. 234–245. ACM (1975). <https://doi.org/10.1145/800027.808445>
11. Fragoso Santos, J., Maksimović, P., Ayoun, S.É., Gardner, P.: Gillian, part i: A multi-language platform for symbolic execution. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 927–942 (2020)
12. van Glabbeek, R.J.: The meaning of negative premises in transition system specifications II. *J. Log. Algebraic Methods Program.* **60-61**, 229–258 (2004). <https://doi.org/10.1016/J.JLAP.2004.03.007>, <https://doi.org/10.1016/j.jlap.2004.03.007>
13. Goncharov, S., Milius, S., Schröder, L., Tsampas, S., Urbat, H.: Stateful Structural Operational Semantics. In: Felty, A.P. (ed.) *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 228, pp. 30:1–30:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). <https://doi.org/10.4230/LIPIcs.FSCD.2022.30>
14. Groote, J.F.: Transition system specifications with negative premises. *Theor. Comput. Sci.* **118**(2), 263–299 (1993). [https://doi.org/10.1016/0304-3975\(93\)90111-6](https://doi.org/10.1016/0304-3975(93)90111-6), [https://doi.org/10.1016/0304-3975\(93\)90111-6](https://doi.org/10.1016/0304-3975(93)90111-6)
15. Katz, S., Manna, Z.: Towards automatic debugging of programs. *ACM SIGPLAN Notices* **10**(6), 143–155 (1975)
16. King, J.C.: Symbolic execution and program testing. *Communications of the ACM* **19**(7), 385–394 (1976)
17. Klin, B., Nachyla, B.: Some undecidable properties of SOS specifications. *J. Log. Algebraic Methods Program.* **87**, 94–109 (2017). <https://doi.org/10.1016/J.JLAMP.2016.08.005>, <https://doi.org/10.1016/j.jlamp.2016.08.005>
18. Lucanu, D., Rusu, V., Arusoaie, A.: A generic framework for symbolic execution: A coinductive approach. *Journal of Symbolic Computation* **80**, 125–163 (2017)
19. Maksimović, P., Ayoun, S.É., Santos, J.F., Gardner, P.: Gillian, part II: Real-world verification for JavaScript and C. In: Silva, A., Leino, K.R.M. (eds.) *Computer Aided Verification*. pp. 827–850. Springer (2021)
20. Plotkin, G.D.: A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* **60-61**, 17–139 (2004), originally a tech. report from Aarhus University, 1981.

21. 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>
22. Rosu, G.: K - a semantic framework for programming languages and formal analysis tools. In: Peled, D., Pretschner, A. (eds.) *Dependable Software Systems Engineering*. NATO Science for Peace and Security, IOS Press (2017)
23. Stefanescu, A., Ciobăcă, Ș., Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G.: All-path reachability logic. In: Dowek, G. (ed.) *Proc. Joint International Conference on Rewriting and Typed Lambda Calculi (RTA-TLCA 2014)*. *Lecture Notes in Computer Science*, vol. 8560, pp. 425–440. Springer (2014), [https://doi.org/10.1007/978-3-319-08918-8\\_29](https://doi.org/10.1007/978-3-319-08918-8_29)
24. Steinhöfel, D.: Abstract execution: automatically proving infinitely many programs. Ph.D. thesis, Technische Universität Darmstadt (2020)
25. Turi, D., Plotkin, G.: Towards a mathematical operational semantics. In: *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. pp. 280–291 (1997). <https://doi.org/10.1109/LICS.1997.614955>
26. Voogd, E., Johnsen, E.B., Silva, A., Susag, Z.J., Wasowski, A.: Symbolic semantics for probabilistic programs. In: *Proc. 20th Intl. Conf. on Quantitative Evaluation of SysTems (QEST 2023)*. *Lecture Notes in Computer Science*, vol. 14287, pp. 329–345. Springer (2023)